Window Programming in Mozart

Christian Schulte

Version 1.3.2
June 15, 2006

Abstract

This document is an introduction to window programming in Mozart. Mozart uses a high-level object-oriented interface to Tk for window programming. The interface inherits from Oz concurrency, objects and first-class procedures. From Tk the interface inherits a set of powerful graphical abstractions. This document exemplifies both aspects: the basic usage of the graphical abstractions and how to profit from Oz's language features.

Credits

Mozart logo by Christian Lindig

License Agreement

Contents

1

Introduction

This document is an introduction to window programming in Mozart. Window programming means to build graphical and interactive interfaces for applications.

In Mozart, the basic building blocks for window programming are widgets: objects that represent graphical entities like labels, buttons, and menus. Windows are described compositionally by means of object hierarchies and are subject to dynamic and interactive modification. Other entities we will deal with are for example fonts and images. The appearance of widgets is managed by geometry managers. Interaction events, such as pressing a mouse button, trigger execution of procedures or methods.

Mozart uses an object-oriented interface to Tk for window programming. The interface inherits from Oz concurrency, objects and first-class procedures. From Tk the interface inherits a set of powerful graphical abstractions. This document introduces both aspects: the basic usage of the graphical abstractions and how to profit from Oz's language features. The interface employs a simple generic mapping to Tk. People familiar with Tk will get acquainted very soon.

The Examples

The documents features a large number of examples which are designed to be tried by the reader. All examples are contained in a demo file[1] to be used with the *"The Oz Programming Interface"*.

Further Information

One particular advantage of using Tk as graphics engine for Mozart is the wealth of excellent documentation for Tcl and Tk.

A must read (or at least see or browse) in this particular respect is the original book[2] of John Ousterhout. A very fascinating account on how to employ the graphical primitives in Tk for developing high-level graphical applications is [1].

The definitive entry point into the full collection of Tcl/Tk related resources and informations is the web page at Scriptics[2], a company co-founded by John Ousterhout.

[1] `WindowProgramming.oz`
[2] `http://www.scriptics.com/`

The details of all commands and widgets for Tcl and Tk can be found in the man-pages that ship with the Tcl and Tk distributions. For convenience, the Mozart release includes HTML versions of them[3], which carry the following copyright[4].

Acknowledgements

I am grateful to Michael Mehl, who co-authored an earlier version of this document. Peter Van Roy contributed the paragraph in Section 5.3 that explains why `tkReturn` is indeed asynchronous.

[3] `../tcltk/contents.htm`
[4] `../tcltk/copyright.htm`

2

Getting Started

This chapter shows a small graphical application from which we will identify the most important concepts found in the window interface. The presentation is really designed for identification of issues. An explanation of these issues follows in the remaining chapters of this document.

2.1 Our First Graphical Application

Figure 2.1 shows a screen dump of our first graphical application. It allows to enter text into the entry field. Pressing the button toggles the capitalization of the text in that entry field.

Figure 2.1 Our first graphical application.

The program for the small graphical application is concerned with the following three issues:

widgets Create the graphical entities called widgets. The application consists of three widgets: a toplevel widget (this is the outermost window), an entry for entering text, and a button.

geometry Arrange the entry and button such that they appear inside the toplevel widget.

actions Attach an action to the button widget such that pressing the button changes the capitalization of the entry text.

In the sections below, we exhibit the code handling these issues. The complete application then has the following structure:

⟨**Widget creation** 4a⟩
⟨**Geometry management** 4b⟩

2.1.1 Widgets

The following program fragment

4a ⟨**Widget creation** 4a⟩≡
```
W={New Tk.toplevel tkInit(title:'Capitalization')}
E={New Tk.entry    tkInit(parent:W)}
B={New Tk.button   tkInit(parent: W
                          text:   'Change Capitalization'
                          action: ⟨Action definition 4c⟩)}
```

creates and initializes the widget objects of our application. Creating and initializing a widget object creates a graphical image for the widget object. We refer to the graphical image just as the widget. Most often we do not distinguish between the object and its widget. All but the toplevel widget are linked by the `parent` features: this defines a widget hierarchy for closing widgets and geometry management.

2.1.2 Geometry

Here we define the geometry for the entry and button widgets:

4b ⟨**Geometry management** 4b⟩≡
```
{Tk.send pack(E B fill:x padx:4 pady:4)}
```

2.1.3 Actions

The remaining program fragment:

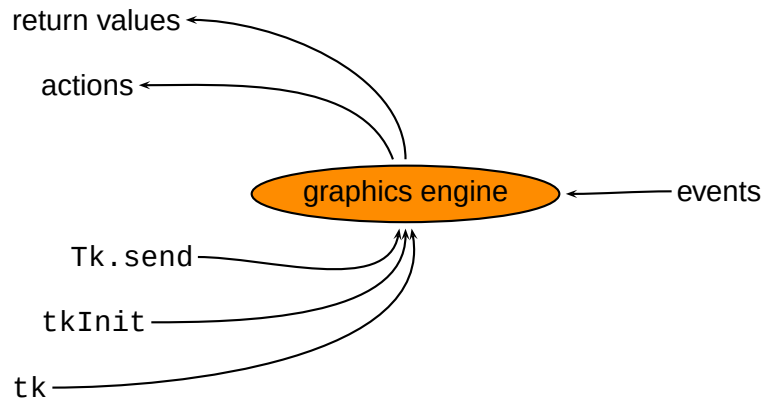
4c ⟨**Action definition** 4c⟩≡
```
proc {$}
   S={E tkReturn(get $)}
in
   {E tk(delete 0 'end')}
   {E tk(insert 0 {Map S ⟨Change capitalization 69a⟩})}
end
```

defines the action as a procedure to be executed when the button is pressed. It retrieves the current content `S` from entry `E`, clears `E`, and reinserts `S` in `E`, but with toggled capitalization. `tkReturn` illustrates another important issue: returning values from widgets.

2.2 The Architecture

Figure 2.2 shows a sketch of the architecture of the window interface in Mozart. Its core part is the graphics engine. The graphics engine computes graphical output and displays it according to the input received.

Figure 2.2 Architecture sketch.



Initializing widgets with the method `tkInit` and applying widgets to the `tk` method send messages to the graphics engine. Additionally, the procedure `Tk.send` we used for geometry management sends messages to the graphics engine. The graphics engine understands tickles, `Tk.send` in fact just takes a tickle and sends it to the engine. Also `tkInit` and `tk` methods map straightforwardly to tickle messages.

The graphics engine is sequential, each tickle is executed in order. User events are considered only when the graphics engine is idle, then the attached actions are executed.

2.3 Implementation

The implementation idea of the window interface is quite simple. The graphics engine is executed by a separate operating system process (it runs in fact a modified `wish`). Any communication with the graphics engine is done via strings. The interface maps tickles and tickle messages to strings (this is done in a fairly efficient way). In the reverse direction, strings can be mapped back to Oz data types like integers, atoms, lists of integers or lists of atoms.

To get an impression of the efficiency of the implementation, try some examples, as well as the demos[1] and the tools[2] that come with Mozart. They are all built on top of the Tk window programming interface.

[1] *"Mozart Demo Applications"*
[2] *"Oz Shell Utilities"*

3

Widgets

This chapter introduces some of the widget objects provided by the Tk interface. Examples illustrate the most common options and the values they can take.

3.1 Toplevel Widgets and Widget Objects

A toplevel widget serves as the outermost container for other widgets. It can be created from the class `Tk.toplevel` and can be initialized by applying it to a message with label `tkInit`. An example is shown in Figure 3.1. The features `width` and `height` of the message together with their values specify that the toplevel widget is 150 pixels wide and 50 pixels high.

Figure 3.1 Creating a toplevel widget.


```
W={New Tk.toplevel tkInit(width:150 height:50)}
```

Creating and initializing a widget object creates a graphical image for the widget object. We will refer to the graphical image just as the widget. Most often we will not distinguish between the object and its widget. A toplevel widget is special in that its graphical image appears immediately on the screen after the widget object has been initialized. Other widgets require to be managed by a so-called geometry manager before they appear on the screen. See Chapter 4 for a discussion of geometry managers.

A widget object can be sent a message with label `tk` to change the appearance or behavior of its widget. For example, the background color of the toplevel widget `W` can be changed to purple by

```
{W tk(configure background:purple)}
```

Additionally, a widget object understands messages that query its state. These will be discussed later.

A widget object can be closed by applying the object to the message `tkClose`. Closing the widget object also destroys the widget displayed on the screen. Section 3.4 contains more details concerning how widget objects are closed.

The structure of messages with labels `tkInit` and `tk` depend on the particular widget under consideration. However, all of these messages share a common structure. The following section explains this structure and shows how to build messages such that they are understood by widgets.

Reference information on toplevel widgets can be found in `toplevel`[1].

3.2 The Graphics Engine, Tickles, and Widget Messages

Widget objects as well as instances of other classes defined in the `Tk` module are built as object oriented frontends to a single graphical agent, the graphics engine.

3.2.1 The Graphics Engine

The graphics engine receives messages and executes them. By executing a message the engine creates widgets, configures the appearance and behavior of widgets, or computes a geometry for the layout of a widget on the screen.

tickles The messages the engine understands are tickles. The procedures `Tk.send` and `Tk.batch` take a tickle or a list of tickles and send it to the graphics engine. We use these two procedures especially to send tickles for geometry management, as is discussed in Chapter 4.

translating object messages to tickles Messages sent to widgets and other objects of the Tk interface are translated in a straightforward fashion to tickles. These tickles are then forwarded to the graphics engine.

3.2.2 Tickles and Widget Messages

Tickles are used to describe messages for the graphics engine. A tickle is either a boolean value, the name `unit`, a virtual string, a record that has neither a name as label nor as feature, or a tickle object. A tickle object is any instance of a class that the Tk module provides, unless otherwise mentioned (the only exception is the class `Tk.listener`, see Section 5.6).

An initialization message with label `tkInit` must be a record without integer features. The field of a feature must be a tickle. Only the special features `parent`, `action`, `url`, and `args` may take different values. These features we will discuss later.

options To the features we refer to as configuration options, or for short as options. Their values we refer to as option values.

[1] `../tcltk/TkCmd/toplevel.htm`

commands and arguments A message with label `tk` must be a record with at least a single integer feature and maybe some other integer features and some options. The value of the first integer feature we call the command, whereas we refer to the remaining values for the integer features as arguments. For example, in the message

```
tk(set active background:purple)
```

`set` is the command, `active` is the single argument, and `background` is an option with value `purple`.

3.2.3 Translating Tickles

The graphics engine does not understand tickles but strings that follow a particular structure. This means that each tickle sent to the graphics engine is first translated to a string. The translation is generic, for our purposes here it suffices to give a short example. The full translation details can be found in Chapter *The Module* `Tk`, *(System Modules)*.

For example,

```
example("test" 1 2.0 side:left(right:true) fill:x)
```

is translated to

```
"example test 1 2.0 -side left -right 1 -fill x"
```

That is, a record is translated to a string consisting of the label and the features and the translation of the fields. Atomic features are prepended by a `"-"` and integer features are ignored.

3.2.4 Special Tickles

Additionally, special tickles are supported (see Figure 3.2). Their usage becomes clear in the examples that are presented in this document.

Figure 3.2 Examples of special tickles.

Example	Translation	Mnemonic	Used
`o(10 12 fill:red)`	`10 12 -fill red`	option	see Figure 6.1
`l(red green blue)`	`[red green blue]`	list	list of tickles
`q(red green blue)`	`{red green blue}`	quote	see Figure 6.1
`s(red green blue)`	`"red green blue"`	string	string of tickles
`p(4 7 linestart)`	`{4.7 linestart}`	position	see (page 58)
`b([a(b:1) c(d:2)])`	`a -b 1 c -d 2`	batch	see (page 20)
`v(1#"\nno quote")`	`1\nno quote`	virtual string	verbatim virtual strings
`c(255 128 0)`	`#FF8000`	color	see Figure 5.10
`d(pack(grid row:4))`	`grid -row 4`	delete	skip record label

3.3 Frames

Frame widgets are similar to toplevel widgets in that they serve as containers for other widgets. The difference is that a frame is used as container within other widgets, whereas a toplevel widget is the outermost container. The main purpose of frames is to determine geometrical layouts for the widgets they contain. More on geometry management we see in Chapter 4.

3.3.1 Relief Options

relief and border Frames support the `relief` and `borderwidth` options. These options determine a three dimensional border to be drawn around a frame. The values for the `relief` option must be one of `groove`, `ridge`, `flat`, `sunken`, and `raised`. The different styles of borders which correspond to theses values are shown in Figure 3.3.

Figure 3.3 Frame widgets with different values for relief.


```
Fs={Map [groove ridge flat sunken raised]
    fun {$ R}
       {New Tk.frame tkInit(parent:W width:2#c height:1#c
                            relief:R borderwidth:4)}
    end}
{{Nth Fs 3} tk(configure background:black)}
{Tk.send pack(b(Fs) side:left padx:4 pady:4)}
```

parent widgets The `tkInit` message contains the option `parent` which links the frames into its parent, the toplevel widget `W`. All widgets but toplevels need a parent widget, this is discussed in Section 3.4.

The program shown in Figure 3.3 maps the list of relief option values to frame objects. To make the frame with the option `flat` visible, its background is configured to be black.

Note that we left out the code to create the toplevel widget. Here and in the following we assume that the variable `W` is bound to a toplevel widget. The example file[2], however, contains the code needed to create the toplevel widget.

The exact meaning of the `pack` command used in this example is explained in Section 4.2.

[2] `WindowProgramming.oz`

3.3.2 Screen Distance Options

As value for the `borderwidth` option we used an integer in the example shown in Figure 3.3. Just giving a number specifies a distance in screen pixels. If the number is followed by one of the letters `c`, `m`, `i`, and `p` the screen distance is given in centimeters, millimeters, inches (2.54 centimeters), or printers' points (1/72 inch).

A convenient way to specify screen distances that employ units is to use a virtual string that appends the unit letter to the number, as used in Figure 3.3.

3.3.3 Color Options

To make the frame with the `relief` option `flat` visible, we configured the background color to be black. Color options can be given either symbolically or numerically.

symbolic color values A symbolic color value can be given as virtual string like `black`, `"red"`, or `"dArK"#blUe`, where the capitalization does not have any significance.

numerical color values A numerical color value is determined by three integers between `0` and `255`. The three integers describe the red, green, and blue part of the color. A numerical color value in Oz can be specified by a ternary tuple with label `c`, where the three fields of the tuple are the three integers. For example, the base colors red, green, and blue are described by the tuples `c(255 0 0)`, `c(0 255 0)`, and `c(0 0 255)` respectively.

Two examples that make frequent use of color options can be found in Section 5.8 and Section 5.9.

3.3.4 Abbreviations and Synonyms

Some of the most common options have the following synonyms:

```
background   bg
foreground   fg
borderwidth  bd
```

option abbreviations In addition to synonyms, it is also possible to abbreviate options provided that the abbreviation is unambiguous. For example, it is correct to abbreviate the `background` option by `ba` but not by `b` since `b` is also an abbreviation for `bitmap` and `borderwidth`.

3.3.5 Additional Tk Information

The full Tk-reference information for each widget is shipped with the Mozart distribution. For an example, see the reference information for `frame`[3].

Reference information on options that are supported by all widgets are explained in `options`[4].

[3] `../tcltk/TkCmd/frame.htm`
[4] `../tcltk/TkCmd/options.htm`

3.4 The Widget Hierarchy

masters and slaves Widgets are arranged in a hierarchy. Each widget has a single parent. The only exceptions can be toplevel widget objects, which do not have to have a parent. The parent of a widget is given by the option `parent` in the `tkInit` message. Usually parent widgets are containers. To the parent of a widget we also refer to as its master. To the widget itself we refer to as slave of its master.

For example, in the previous example shown in Figure 3.3 the five frame widgets are slaves of the single toplevel widget.

The purpose of the hierarchy is threefold:

1. The geometry for widgets is computed according to the hierarchy. This is discussed in Chapter 4.
2. Creation and initialization has to follow the hierarchy. To initialize a widget object it is necessary that its parent widget object is already created and initialized. Otherwise initialization of a slave blocks until its master is initialized.
3. Closing a parent widget object also closes all its slaves. The slaves are closed by applying them to the `tkClose` message. A widget object gets closed and its widget gets destroyed only after all of its slaves have been closed.

After a widget object has been closed, using it in tickles sent directly to the graphics engine, e.g. by `Tk.send` or `Tk.batch`, issues a runtime error.

3.5 Label Widgets

A label widget displays a text string or a bitmap. Options for frames are also valid options for labels, additional options determine what the label displays. The reference documentation for labels is `label`[5].

Figure 3.4 Example for labels displaying bitmaps and text.


```
L1={New Tk.label tkInit(parent:W bitmap:info)}
L2={New Tk.label tkInit(parent:W text:'Labels: bitmaps and text')}
{Tk.send pack(L1 L2 side:left padx:2#m pady:2#m)}
```

Figure 3.4 shows an example where the label `L1` displays a bitmap and the label `L2` displays text. As with other widgets, the options of a label widget can be reconfigured by sending the widget object a `tk` message with the command `configure`. Execution of the following expression changes the bitmap to an exclamation mark:

[5] `../tcltk/TkCmd/label.htm`

```
{L1 tk(configure bitmap:warning)}
```

3.5.1 Bitmap Options

Label widgets and several other widgets allow to display bitmaps. There are two different kinds of bitmaps: predefined bitmaps and bitmaps stored in files.

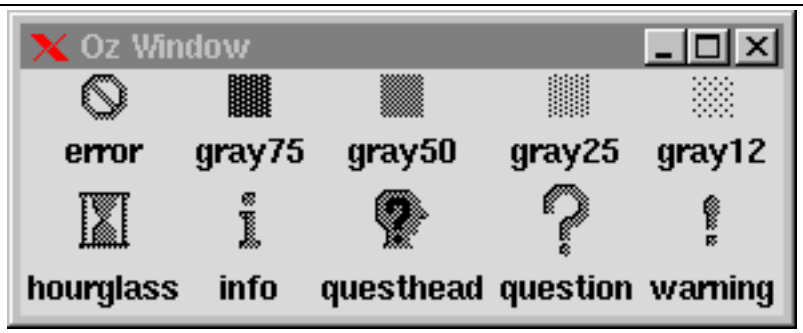
If the first character of the `bitmap` option value is an `@`, the value is interpreted as filename. For instance, feeding

```
{L2 tk(configure bitmap:     '@'#{Property.get 'oz.home'}#
                             '/doc/wp/queen.xbm'
                   foreground: orange)}
```

displays a bitmap stored in a file. Here the file name is given relative to where the Mozart system has been installed, that is relative to `{Property.get 'oz.home'}` (for the system module `Property` see Chapter *Emulator Properties:* `Property`, *(System Modules)*).

predefined bitmaps If the first character is different from `@`, it is interpreted as the name of a predefined bitmap. A program that displays all predefined bitmaps and their names you can see in Figure 3.5. The program uses the grid geometry manager which is discussed in Section 4.3.

Figure 3.5 Predefined bitmaps.




```
{List.forAllInd [error     gray75 gray50    gray25   gray12
                 hourglass info   questhead question warning]
 proc {$ I D}
    R=(I-1) div 5
    C=(I-1) mod 5
 in
    {Tk.batch [grid(row:R*2   column:C
                    {New Tk.label tkInit(parent:W bitmap:D)})
               grid(row:R*2+1 column:C
                    {New Tk.label tkInit(parent:W text:D)})]}
 end}
```

bitmap colors Bitmaps have two colors. These colors can be configured with the `foreground` and `background` options. The color of the bitmaps' pixels is given by the foreground color.

3.5.2 Font Options

A font to be used for displaying text can be specified by the `font` option. Valid values for the `font` option are either platform specific font names or instances of the class `Tk.font`. An instance of the class `Tk.font` is also a tickle object but is not a widget.

Platform dependent font names are for example X font names. If you are running a Unix based system, you can for example display the available names by using the `xlsfonts` program.

However the preferred way to specify fonts is to be platform independent of course. The program in Figure 3.6 uses this technique.

Figure 3.6 Example for different fonts.


```
{ForAll [times helvetica courier]
 proc {$ Family}
    {ForAll [normal bold]
     proc {$ Weight}
        F={New Tk.font  tkInit(family: Family
                               weight: Weight
                               size:   12)}
        L={New Tk.label tkInit(parent: W
                               text:   'A '#Weight#' '#Family#' font.'
                               font:   F)}
     in
        {Tk.send pack(L)}
     end}
 end}
```

The init message for creating a font determines with the options `family` (the style of the font), `weight` (whether it is bold or normal), and `size` (how large is the font in

point, if the number is positive, in pixels if it is less than zero) how the font looks. `Tk.font` supports more options, for a complete overview consult `font`[6].

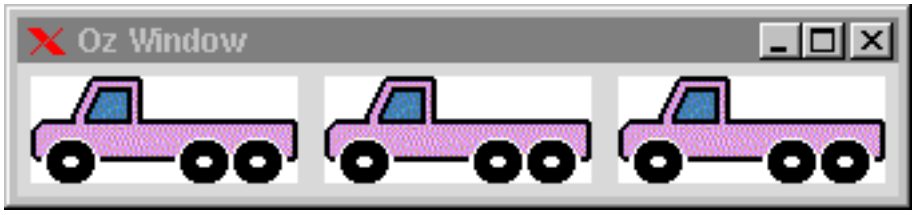
Regardless of the platform, the families `courier`, `times`, and `helvetica` are supported.

3.6 Images

Besides of text and bitmaps labels can display images. Images differ from bitmaps in that they allow for more than two colors to be displayed.

Images are provided as objects in Oz. These objects are also tickle objects (see Section 3.2), but are different from widget objects.

Figure 3.7 Three labels displaying the same image.




```
D ={Property.get 'oz.home'}#'/doc/wp/'
I ={New Tk.image tkInit(type:photo format:ppm file:D#'truck-left.ppm')}
L1={New Tk.label tkInit(parent:W image:I)}
L2={New Tk.label tkInit(parent:W image:I)}
L3={New Tk.label tkInit(parent:W image:I)}
{Tk.send pack(L1 L2 L3 padx:1#m pady:1#m side:left)}
```

The program in Figure 3.7 creates an image object and displays the image in three labels. Changing the configuration of the image, changes the displayed image in all label widgets. For example, feeding the following expression

```
{I tk(configure file:D#'truck-right.ppm')}
```

replaces all three displayed trucks by trucks heading in the inverse direction.

type and format Images can be of two different types. The value of the `type` configuration option can be `photo` (as in our example), or `bitmap`. If the type is `photo`, the image can display files in two different formats. The format is specified by the `format` option. Valid values for the `format` option are `gif` and `ppm`.

bitmap images In case the value for the `type` option is `bitmap`, the value given for the `file` option must be a valid bitmap file.

[6] `../tcltk/TkCmd/font.htm`

referring to images by URLs In addition to files, images can also be referred to by URLs. For example,

```
{New Tk.image tkInit(type:photo format:gif
                     url:'http://foo.com/bar.gif'}
```

would have loaded a gif file from the given URL. Note that the graphics engine itself is not capable of handling URLs. Instead, the image object handles URLs by localizing a URL to a local file (see also Chapter *Resolving URLs:* `Resolve`, *(System Modules)*). The local file then is used by the graphics engine.

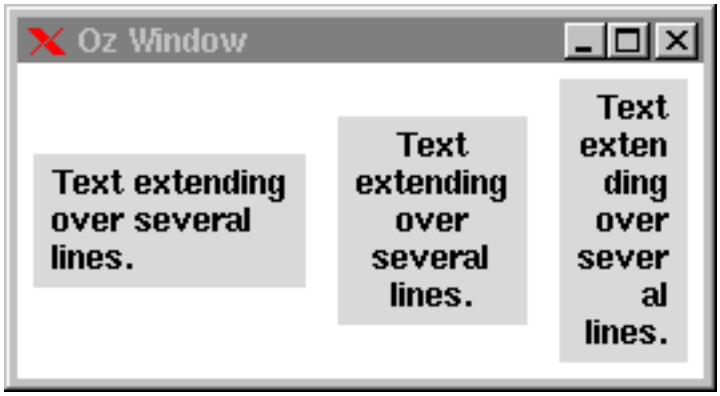
In Section 8.4 an abstraction is presented that eases the handling of images considerably.

Reference information on images can be found in `image`[7].

3.7 Messages

aspect and justify Message widgets display text extending over several lines. How the text is distributed over several lines is determined by one of the options `width` and `aspect`. Each of the line is justified according to the value of the option `justify`. Possible values are `left` (the default value), `center`, and `right`. Figure 3.8 shows the result of different combinations of aspect and justification.

Figure 3.8 Messages with `justify` and `aspect` options.




```
S ='Text extending over several lines.'
Ms={Map [left#200 center#100 right#50]
    fun {$ J#A}
       {New Tk.message tkInit(parent:W text:S justify:J aspect:A)}
    end}
{Tk.send pack(b(Ms) side:left padx:2#m pady:2#m)}
```

[7] `../tcltk/TkCmd/image.htm`

aspect and width If the option `width` is present, the value (a screen distance, see Section 3.3.2) gives the length of each line. If no `width` option is present, the aspect ratio between height and width of the text is given by the option `aspect`. The value specifies the aspect as

```
100 * width / height
```

For example a value of `100` means that the text is as high as wide, a value of `200` means that the text is twice as wide as high.

Reference information on message widgets can be found in `message`[8].

[8] `../tcltk/TkCmd/message.htm`

4

Geometry Managers

This chapter explains geometry managers. Geometry managers compute how much space widgets occupy and at which position they appear on the screen. Last but not least they make widgets appear on the screen with the geometry computed previously.

4.1 Widgets and Parcels

A geometry manager computes the size and location of widgets, that is the geometry, and displays the widgets on the screen. The geometry manager computes the geometry according to the widget hierarchy. During computation of the geometry, the manager takes the following three things into account:

1. The geometry requested by slave widgets. Widgets like labels and messages request just enough space to displays their text or bitmap.
2. The geometry requested by master widgets. Usually master widgets do not request an explicit geometry. But for example, if a frame widget is initialized with explicit values for width and height, the geometry manager takes these values into account.
3. The options given to the geometry manager.

parcels The geometry manager computes for each slave of a master widget a so-called parcel. The parcel is a rectangle and describes the space and the position computed for the slave. From the slaves' parcels the manager computes the parcel for the master. If the master does not request a specific geometry on its own, the manager will assign the master a parcel that encloses all slave parcels. Otherwise the geometry manager distributes the space in the parcel for the master to the slave parcels. This may shrink or grow the parcels for the slaves.

Options to the geometry manager affect usually the way how parcels are computed and how widgets are put into their parcels, if the parcels are larger (or smaller) than the parcel initially requested by the widget.

In the following we will show the two most important geometry managers which are provided in Tk. One is the packer, which can be used for simple arrangements, like placing several widgets in a row or in a line. The other geometry manager we will discuss is implemented by the `grid` command. As the name suggests, the grid command allows for arranging widgets in a grid-like fashion.

It is perfectly possible to mix geometry managers in a single toplevel widget provided that all slaves of a master are managed by the same manager. For example, suppose a toplevel widget that contains two frames which contain widgets themselves. Both frames must be managed by the same manager. The widgets in the frames can be managed by two different managers.

We discuss only the most important options these two managers provide, a complete description can be found in `pack`[1] and `grid`[2]. More information on the packer can also be found in John Ousterhout's book[2] in Chapter 17.

4.2 The Packer

The packer supports simple arrangements of widgets in rows and columns. Arranging widgets nicely usually also means that some vertical and horizontal space has to be inserted, either designed to provide for additional space or to fill up space not occupied by the widget's original size.

The different ways how to affect the geometry we will study by means of examples. For this, let us assume we are dealing with three label widgets. The following function creates a toplevel widget with background color white for better visibility, and returns a list of three labels.

```
fun {NewLabels}
   W={New Tk.toplevel tkInit(background:white)}
in
   {Map ['label' 'Second label widget' '3rd label']
    fun {$ A}
       {New Tk.label tkInit(parent:W text:A)}
    end}
end
```

To display the labels in the toplevel widget, the packer can be invoked as follows:

```
[L1 L2 L3] = {NewLabels}
{Tk.send pack(L1 L2 L3)}
```

This computes and displays a geometry for the toplevel widget as shown in Figure 4.1. Rather than giving a tickle which contains each of the labels as field we can give a batch tickle. A batch tickle is a tuple with label `b` where its single argument must be a list of tickles. By using a batch tickle, we can rewrite our example from above to

```
{Tk.send pack(b({NewLabels}))}
```

where the list of tickles is the list of labels as returned by the function `NewLabels`.

[1] ../tcltk/TkCmd/pack.htm
[2] ../tcltk/TkCmd/grid.htm

Figure 4.1 Plain geometry computed by the packer.

4.2.1 Side Options

The label widgets in the previous examples were placed from the top to the bottom of the toplevel widget. The side where the widgets are packed against can be determined with the `side` option. The default value for this option is `top`. The examples in Figure 4.2 show the geometry which is computed when `left` and `bottom` are given as values for the side option. Valid values for the side option are `top`, `bottom`, `left`, and `right`.

Figure 4.2 Geometries computed by the packer according to `side` option.


```
{Tk.send pack(b({NewLabels}) side:left)}
```


```
{Tk.send pack(b({NewLabels}) side:bottom)}
```

4.2.2 Padding

The geometry computed for widgets by the packer can be given additional space in two different ways: either externally or internally. Additional external space can be specified with the options `padx` and `pady`. The values for these options must be valid screen distances (see Section 3.3.2), specifying how much additional space should be provided by the master widget around the packed widgets. The internal space can be specified by the `ipadx` and `ipady` options, where the values must be screen distances as well. These values determine by how much space the packed widgets are expanded in each of their four borders. The examples in Figure 4.3 show the effects on the geometries computed by the packer for both internal and external padding.

Figure 4.3 Additional space provided by the packer.


```
{Tk.send pack(b({NewLabels}) padx:1#m pady:1#m)}
```


```
{Tk.send pack(b({NewLabels}) ipadx:2#m ipady:2#m)}
```

4.2.3 Anchors

With the `anchor` option it can be specified where in a widget's parcel the packer places the widget. If no `anchor` option is given, the packer places the widget in the center of its parcel. Otherwise, the widget is placed according to the option's value, which can be one of `center`, `n`, `s`, `w`, `e`, `nw`, `ne`, `sw`, and `se`. The Figure 4.4 shows the geometry computed when `w` is used as anchor.

4.2.4 Filling and Expansion

For pleasant overall geometry it is imported that widgets have similar geometries. The packer employs two different schemes how widgets can be arranged to have similar geometries. One is filling: the widget extends over its entire parcel. The other one is expansion: the widget's parcel is extended such that the parcels of all slaves in a master occupy the master's parcel entirely.

Figure 4.5 shows the geometry computed when the option `fill` with value `x` is used. Possible values for the `fill` option are `x`, `y`, `both`, and `none` (which is the default).

Expansion is only significant when the parcels of the slave do not fill the master's parcel completely. In all our previous examples, the parcel of the master was computed by the packer to be just large enough to contain the slave's parcels. So there was no additional space in the master's parcel to be filled by expansion of slave parcels.

Figure 4.4 Using the `anchor` option for packing.


```
{Tk.send pack(b({NewLabels}) anchor:w padx:1#m pady:1#m)}
```

Figure 4.5 Using the `fill` option for packing.


```
{Tk.send pack(b({NewLabels}) fill:x)}
```

Figure 4.6 shows three toplevel widgets which have been resized manually by dragging with the mouse. The top most example shows that when the parcel of the toplevel widget grows, the remaining space is filled by the label widgets. In the example in the middle, only the parcels of the label widget's are expanded. At the bottom, the parcels are expanded and then filled up in both horizontal and vertical direction by the label widgets.

4.3 The Grid Geometry Manager

The grid geometry arranges widgets in a grid-like fashion. For each widget to be managed by the `grid` command, a row and a column number is given. The manager computes parcels for the widgets such that all parcels in the same column have the same width and all parcels in the same row have the same height.

Figure 4.7 shows how eight labels are placed by the `grid` command. Note that it is not necessary that all positions in the grid are occupied by a widget. In our example in Figure 4.7, the position at row and column `2` does not contain a widget.

4.3.1 Padding

The `grid` command supports padding in the same way as the packer does. In the above example we used external padding by giving `padx` and `pady` options. It is also possible to use internal padding with the options `ipadx` and `ipady`.

Figure 4.6 Resizing effects for filling and expansion.


```
{Tk.send pack(b({NewLabels}) fill:x)}
```


```
{Tk.send pack(b({NewLabels}) expand:true)}
```


```
{Tk.send pack(b({NewLabels}) fill:both expand:true)}
```

Figure 4.7 Using the `grid` command.


```
proc {GL W R C S}
   L={New Tk.label tkInit(parent:W text:S)}
in
   {Tk.send grid(L row:R column:C padx:4 pady:4)}
end
{GL W 1 1 nw}   {GL W 1 2 north} {GL W 1 3 ne}
{GL W 2 1 west}                  {GL W 2 3 east}
{GL W 3 1 sw}   {GL W 3 2 south} {GL W 3 3 sw}
```

4.3.2 Span Options

The grid command can also compute geometries where widgets occupy more than a single row or column. In the example shown in Figure 4.8 the label widget `L` is managed by the `grid` command to occupy both two rows and two columns. How much rows and columns a widget's parcel spans is specified with the `columnspan` and `rowspan` options.

4.3.3 Sticky Options

The `grid` command combines the `anchor` and `fill` options from the packer in a single `sticky` option. The value given for a sticky option determines both the side the widget is placed in its parcel, and how the widget is to be stretched to fill its parcel.

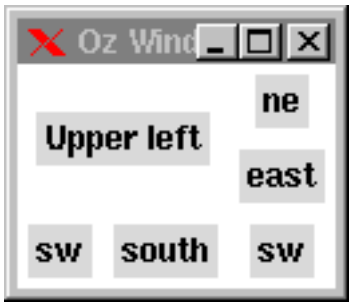
Valid values for the `sticky` option are all combinations of the letters `n`, `s`, `w`, and `e` in any order. Giving one of `n` and `s` (or of `w` and `e`) specifies the anchor position of a widget. Giving both `n` and `s` (or both `w` and `e`) requests that the widget should fill its parcel horizontally (or vertically). For an example see Figure 4.9.

4.3.4 Weight Options

The grid geometry manager employs a different scheme for expansion of parcels than the packer. Rows and columns in the grid can be assigned an integer weight. Additional space available in the master of the grid is distributed between the rows and columns according to their relative weight.

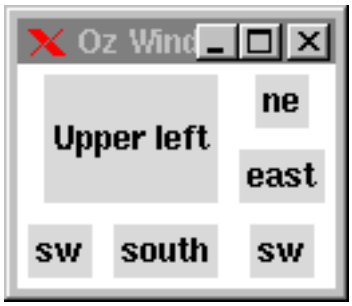
For example, if we take the last example and want that all additional space is given to the third row and third column, we can do this by

Figure 4.8 Using the `columnspan` and `rowspan` options.




```
{Tk.send grid({New Tk.label tkInit(parent:W text:'Upper left')}
               row:1    rowspan:2
               column:1 columnspan:2
               padx:4 pady:4)}
{GL W 1 3 ne} {GL W 2 3 east}
{GL W 3 1 sw} {GL W 3 2 south} {GL W 3 3 sw}
```

Figure 4.9 Using the `sticky` option with the `grid` command.

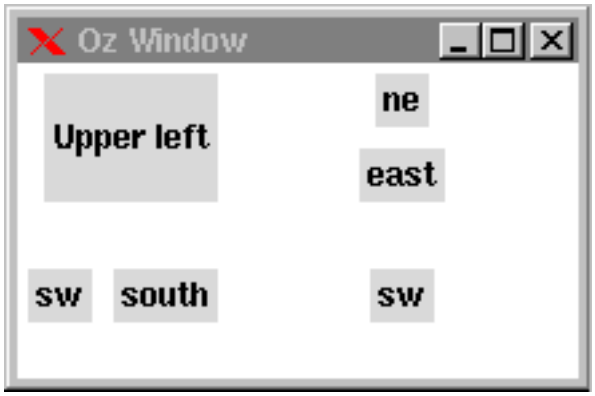



```
{Tk.send grid({New Tk.label tkInit(parent:W text:'Upper left')}
               row:1    rowspan:2
               column:1 columnspan:2
               sticky: nse
               padx:4 pady:4)}
```

```
{Tk.batch [grid(rowconfigure    W 3 weight:1)
           grid(columnconfigure W 3 weight:1)]}
```

Figure 4.10 shows the result of resizing the window.

Figure 4.10 Result of resizing a window.



4.4 Using Anchors for Widgets

The `anchor` option for the packer and the `sticky` option for the grid geometry manager determine where the widget is placed in its parcel. In the same way several kind of widgets, e.g., message and label widgets, take an anchor `option`, which determines where the displayed item, e.g., the text or bitmap, is placed within the widget.

Figure 4.11 shows an example for the three label widgets used throughout Section 4.2. The possible values for the anchor options are the same as described in Section 4.2.3.

Figure 4.11 Widgets with `anchor` options.


```
[L1 L2 L3]={NewLabels}
{Tk.send pack(L1 L2 L3 fill:x)}
{L1 tk(configure anchor:w)}
{L3 tk(configure anchor:e)}
```

5

More Widgets

This chapters presents widgets which are intended to be interactive: buttons to invoke actions, entries to enter text, scales to enter numbers, and listboxes to choose elements of lists.

5.1 Buttons and Actions

actions Button widgets are similar to label widgets: they display text, bitmaps, or images. Additionally, they provide for actions: pressing the mouse button over the button widget, invokes an action. An action is either a procedure or a pair of object and message. If the action is a procedure, pressing the widget button creates a thread in which the procedure is applied. Otherwise, a thread is created that applies the object to the message provided. Actions are discussed in more detail in Section 5.6.

Figure 5.1 shows a program which creates two buttons and attaches actions to them. Pressing button `B1` browses the atom `pressed`, whereas pressing button `B2` closes the toplevel widget object `T`.

action values The `action` option is different from other options in that it has not a generic translation as explained in Section 3.2. Valid values for this option are not tickles, but as already mentioned, procedures or object message pairs.

Internally, an object providing for the `action` option, creates a Tcl script which when executed invokes the Oz procedure or object. This script is used then as value for the configuration option `command`. All widgets which provide for the `command` option in Tk, provide for the `action` option in Oz.

changing actions Actions can be deleted or changed by the method `tkAction`. For example, deleting the action for button `B1` and changing the action for `B2` can be done by executing:

```
{B1 tkAction}
{B2 tkAction(action: B1 # tkClose)}
```

More information on buttons can be found in `button`[1].

[1] `../tcltk/TkCmd/button.htm`

Figure 5.1 Buttons with attached actions.


```
B1={New Tk.button tkInit(parent: W
                         text:   'Press me!'
                         action: proc {$}
                                    {Browse pressed}
                                 end)}
B2={New Tk.button tkInit(parent: W
                         bitmap: error
                         action: W#tkClose)}
{Tk.send pack(B1 B2 fill:x padx:1#m pady:1#m)}
```

5.2 Checkbuttons, Radiobuttons, and Variables

checkbuttons Checkbutton widgets are used for binary choices. An indicator to the left shows whether the button is 'on' or 'off'. The state of the indicator is given by a tickle variable. A tickle variable is a tickle object that provides messages to set and to query the value of the variable.

radiobuttons Radiobuttons are used for non-binary choices. Several radiobuttons are linked together to a group. Selecting a radio button de-selects all buttons belonging to the same group and displays a mark in an indicator to the left. Radiobuttons are linked together by tickle variables: all buttons belonging to the same group share the same variable.

Figure 5.2 shows an example of a checkbutton and a group of three radiobuttons. The value with which the variable `V1` is initialized determines whether the checkbutton initially is selected. The value of the variable `V2` determines which of the radiobuttons is selected initially.

state To query the state of the radiobutton and of the checkbuttons we query the state of the corresponding variables. Feeding the following expression

```
{Browse state(bold:   {V1 tkReturnInt($)}==1
              family: {V2 tkReturnAtom($)})}
```

displays the values of the variables in the browser. See the next section (page 31) for a detailed explanation of the `tkReturnInt` and `tkReturnAtom` messages.

Figure 5.2 A checkbutton and three radiobuttons.


```
V1={New Tk.variable tkInit(false)}
C ={New Tk.checkbutton tkInit(parent:W variable:V1
                              text:'Bold' anchor:w)}
V2={New Tk.variable tkInit('Helvetica')}
Rs={Map ['Times' 'Helvetica' 'Courier']
    fun {$ F}
       {New Tk.radiobutton tkInit(parent:W
                                  variable:V2 value:F
                                  text:F anchor:w)}
    end}
{Tk.batch [grid(C     padx:2#m columnspan:3)
           grid(b(Rs) padx:2#m)]}
```

actions Very often selecting a checkbutton or a radiobutton must show immediate effect. For this purpose it is possible to attach actions to checkbuttons and radiobuttons in the same way as for buttons. Figure 5.3 shows an example where the checkbutton and the radiobuttons configure the font of a label widget. Note that the options used for initialization of the checkbutton and radiobuttons are the same as in the example shown in Figure 5.2.

Reference information on radiobuttons and checkbuttons can be found in `radiobutton`[2] and `checkbutton`[3].

5.3 Querying Tickle Objects

In the previous section we queried a Tk-variable's state with the methods `tkReturnInt` and `tkReturnAtom`. In fact it is possible to query the state of all tickle objects, in particular to query the state of widgets.

All tickle objects provide a method `tkReturn`. This method is similar to the `tk` method in that it sends a message to the graphics engine. After the message has been executed by the graphics engine, however, the `tkReturn` method returns a string, whereas the `tk` method ignores this string.

synchronization The field of the `tkReturn` message with the largest integer feature is constrained to the string returned. The method `tkReturn` is asynchronous: it

[2] `../tcltk/TkCmd/radiobutton.htm`
[3] `../tcltk/TkCmd/checkbutton.htm`

Figure 5.3 Actions for radiobuttons and checkbuttons.


```
fun {GetWeight}
   if {V1 tkReturnInt($)}==1 then bold else normal end
end
F={New Tk.font tkInit(size:24
                      family: {V2 tkReturn($)}
                      weight: {GetWeight})}
L={New Tk.label tkInit(parent:W text:'A test text.' font:F)}
{C  tkAction(action: proc {$}
                        {F tk(configure weight:{GetWeight})}
                        {L tk(configure font:F)}
                     end)}
{List.forAllInd ['Times' 'Helvetica' 'Courier']
 proc {$ I Family}
    {{Nth Rs I} tkAction(action: proc {$}
                                    {F tk(configure family:Family)}
                                    {L tk(configure font:F)}
                                 end)}
 end}
```

sends the message to the graphics engine but does not block the thread until the return string is available. Eventually the graphics engine writes the return string to the store.

The `tkReturn` message is sent asynchronously for efficiency reasons. One can start another calculation without having to wait for `tkReturn`'s result. One can send several `tkReturn` messages consecutively, and they will be sent immediately. The messages will be handled by the graphics agent in the same order as they are sent.

If you want to be sure that you have received the return value, say `X`, of `tkReturn` before continuing, then you must use a `{Wait X}` statement.

illegal return values Similar to the method `tkReturn` tickle objects provide methods that return atoms, integers, floats and lists of strings, atoms, integers, and floats. Rather than writing a string to the store they write a value to the store which is obtained by transforming the string to the particular type. If it is not possible to transform the string into a value of that type, the boolean value `false` is written to the store.

The methods that return a list of strings, atoms, integers, or floats split the string into

substrings separated by space characters. For instance, the return string `"a b c"` is transformed into the list `[a b c]` by the method `tkReturnListAtom`. Figure 5.4 lists the return methods and how the methods transform strings. Note that a string `"1.0"` is transformed by `tkReturnInt` to the integer `1`.

Figure 5.4 Returns methods and examples of return values.

Method	Example string	Return value
tkReturn	"red 1 1.0"	"red 1 1.0"
tkReturnAtom	"red 1 1.0"	'red 1 1.0'
tkReturnInt	"1"	1
tkReturnFloat	"1.0"	1.0
tkReturnList	"red 1 1.0"	["red" "1" "1.0"]
tkReturnListAtom	"red 1 1.0"	[red '1' '1.0']
tkReturnListInt	"red 1 1.0"	[false 1 1]
tkReturnListFloat	"red 1 1.0"	[false 1.0 1.0]

string handling procedures The Tk Module provides also for a set of procedures that can transform strings into atoms, integers, floats and lists of these three types. With these procedures it is possible to transform return strings in a user defined fashion. For more information see Section *Strings, (System Modules)*.

5.3.1 Querying Configuration Options

The values of configuration options of a widget can be queried with the `configure` command. Instead of giving a value to which the option is to be set, we give the tickle `unit` as value. The tickle `unit` expands to just nothing, meaning that the value is not to be set but to be queried. For example, to query the value of the `bg` option of a widget `T`, we can feed

```
{T tkReturnListAtom(configure bg:unit $)}
```

This displays a list of atoms, usually it suffices to know that the current value of the option is the fifth element of the list, whereas its default value is the fourth element of the list.

5.3.2 Querying Widget Parameters

The command `winfo` is helpful to query parameters of widgets. For instance, to query the position and geometry of a widget `T`, we can use the following:

```
{Browse {Map [rootx width rooty height]
         fun {$ A}
            {Tk.returnInt winfo(A T)}
         end}}
```

The `winfo` command provides more options than those used above, for the details please consult `winfo`[4].

[4] `../tcltk/TkCmd/winfo.htm`

5.4 Menus, Menuitems, and Menubuttons

Menu widgets serve as containers for menu entries. A menu entry can be one of the following:

`separator`	displays a horizontal line
`command`	similar to button widgets
`radiobutton`	similar to radiobutton widgets
`checkbutton`	similar to checkbutton widgets
`cascade`	displays sub menus

Menu entries are not widgets. In particular, menu entries are not managed by a geometry manager. Instead as soon as a menu entry is created it is displayed in its parent menu. To configure a menu entry after it has been created, one needs to use the `entryconfigure` command rather than the `configure` command.

tear off entry The program shown in Figure 5.5 creates two menu widgets `M1` and `M2`. The first cascade entry of the menu widget `M1` is configured such that it displays the menu `M2` when the menu is traversed. The option `tearoff` determines that the first default so-called 'tear off' entry is not created. Selecting a tear off entry displays the menu in a window on its own.

Figure 5.5 A menu with entries, including a cascaded sub menu.


```
Cs ='Wheat' 'Firebrick' 'Navy' 'Darkorange']
M1 ={New Tk.menu tkInit(parent:W  tearoff:false)}
M2 ={New Tk.menu tkInit(parent:M1 tearoff:false)}
E1 ={New Tk.menuentry.cascade
     tkInit(parent:M1 label:'Background Color' menu:M2)}
E2 ={New Tk.menuentry.separator tkInit(parent:M1)}
E3 ={New Tk.menuentry.command
     tkInit(parent:M1 label:'Quit' action: W#tkClose)}
V  ={New Tk.variable tkInit(Cs.1)}
CEs={Map Cs fun {$ C}
               {New Tk.menuentry.radiobutton
                tkInit(parent:M2 label:C var:V val:C
                       action: W#tk(configure bg:C))}
            end}
```

posting menus Usually menus are not visible. Only when needed a menu appears on the screen, we say that it is posted. After the user has traversed the menu and has selected an entry, the menu is made invisible again: it is unposted. Posting the menu `M1` at the upper left edge of the screen can be done by

```
{M1 tk(post 0 0)}
```

menubuttons From menus one can compose menu bars and popup menus. A menu bar consists of several menubutton widgets. A menubutton widget can display text, bitmaps, or images. To a menubutton a menu can be attached such that pressing the button makes the menu widget appear on the screen. We do not discuss menu bars here in detail, since the `TkTools` module provides an abstraction that supports the creation of menu bars (see Section 8.3).

popup menus The command `tk_popup` can be used to display popup menus. It takes as arguments the menu widget and the coordinates where the widget should appear on the screen. Ideally, we want the menu widget to appear after pressing the mouse button when the mouse pointer is over some widget. The next section (page 35) introduces events which allows to mattach actions to abitrary widgets.

Reference information can be found in `menu`[5] and `menubutton`[6].

5.5 Events

binding to events Button widgets allow to specify an action which is invoked when the button is pressed. This is only one particular example of attaching an action to an event. The Tk toolkit allows to attach actions to any widget with the method `tkBind`. To attach an action to an event we refer to as binding the action to the event. The action is invoked when some event occurs. Examples for events are to move the mouse pointer within a widget, or to press a mouse button when the mouse pointer is over a widget. Actions can be given arguments. The arguments depend on the type of the event, e.g., arguments can be the coordinates of the mouse pointer.

Let us look to the example from the previous section. There we created a menu widget `M1` and a toplevel widget `T`. Now we want that the menu widget is posted if the mouse button is pressed over the toplevel widget `T`. Additionally, we want the menu to get posted at the position of the mouse pointer when the mouse button was pressed.

event patterns The program in Figure 5.6 does what we want: `'<Button-1>'` for the `event` option is the so-called event pattern. The value for the `args` option describes that the action should invoked with two arguments. The first (second) one should be an integer giving the x (y) coordinate of the mouse pointer within the widget when the mouse button has been pressed. The `action` procedure pops up the menu widget at exactly the screen coordinates. These are computed from the coordinates of the upper left edge of the toplevel widget and the event arguments.

[5] `../tcltk/TkCmd/menu.htm`
[6] `../tcltk/TkCmd/menubutton.htm`

Figure 5.6 Action to popup menu.

```
{W tkBind(event:  '<Button-1>'
          args:   [int(x) int(y)]
          action: proc {$ X Y}
                     TX={Tk.returnInt winfo(rootx W)}
                     TY={Tk.returnInt winfo(rooty W)}
                  in
                     {Tk.send tk_popup(M1 X+TX Y+TY)}
                  end)}
```

5.5.1 Event Patterns

An event pattern is either a string consisting of a single character, where the character must be different from the space character and `<`. This event pattern matches a `KeyPress` event for that character.

Otherwise, an event pattern must be a string

```
'<'#Modifier#'-'#Modifier#'-'#Type#'-'#Detail#'>'
```

where only one of either `Type` or `Detail` is mandatory. This means, for example that also

```
'<'#Type#'>'
```

or

```
'<'#Detail#'>'
```

are valid event patterns.

event modifiers and types Figure 5.7 shows common event modifiers and event types. Multiple entries separated by commas can be used as synonyms. The full set of modifiers and types is described in `bind`[7].

For example, the event pattern

```
'<Shift-Double-Button>'
```

matches the event that a mouse button is double-clicked while the shift key is pressed.

If the event is `ButtonPress` or `ButtonRelease`, `Detail` may be the number of a mouse button. The number must be between `1` and `5`, where `1` is the leftmost button. The number as detail means that only events from pressing or releasing this particular button match. If no detail is given, pressing or releasing any button matches the event. If a number is given as detail, `ButtonPress` can be omitted, e.g., `<ButtonPress-1>` and `<1>` match the same events.

If the event is `KeyPress` or `KeyRelease`, `Detail` may be the specification of a keysymbol, e.g., `comma` for the `,` key. For more information please consult `bind`[8].

[7] `../tcltk/TkCmd/bind.htm`
[8] `../tcltk/TkCmd/bind.htm`

Figure 5.7 Some event modifiers and types.

Event Modifier				
`Control`	`Shift`	`Lock`	`Meta`	`Alt`
`Button1`, `B1`	`Button2`, `B2`	`Button3`, `B3`	`Button4`, `B4`	`Button5`, `B5`
`Double`	`Triple`			

Event Type	Description
`Key`, `KeyPress`	key has been pressed
`KeyRelease`	key has been released
`Button`, `ButtonPress`	mouse button has been pressed
`ButtonRelease`	mouse button has been released
`Enter`	mouse pointer has entered a widget
`Leave`	mouse pointer has left a widget
`Motion`	mouse pointer has been moved within widget

5.5.2 Event Arguments

The `args` option of the `tkBind` method takes a list of argument specifications. An argument specification describes the type of the argument and the argument itself. Figure 5.8 shows the valid types and some arguments. The types mean the same as the types for the different return methods as discussed in Section 5.3.

Figure 5.8 Event arguments.

Argument Type				
`string(A)`, `A`	`atom(A)`	`int(A)`	`float(A)`	
`list(string(A))`, `list(A)`	`list(atom(A)`	`list(int(A)`	`list(float(A)`	

Event Argument	Description
`x`	x coordinate
`y`	y coordinate
`K`	string describing the symbol of the key pressed
`A`	character describing the key pressed

5.5.3 Invoking Actions

When an event matches an event pattern to which an action has been bound by `tkBind`, a new thread is created in which the action is executed. If the action is a procedure the arity of the procedure has to be equal to the length of the argument list specified in `tkBind`.

If the action is a pair of object and message, the object is applied to message with the arguments appended. For example, after creating an event binding by

```
{T tkBind(event:  '<1>'
          args:   [int(x)]
          action: O # invoke(button))}
```

pressing the leftmost button at x-coordinate 42 creates a thread that executes the statement

```
{O invoke(button 42)}
```

5.5.4 Appending and Deleting Event Bindings

If `tkBind` is used as before, any other existing binding for the event pattern specified is overwritten. If no action is specified any existing binding for the event pattern is deleted.

For a single event pattern there may be more than one binding. To create an event binding that does not overwrite already existing bindings, we can give the option `append` with value `true`. For instance, if we do not only want to popup the menu but also to display `pop` in the Browser, we can create an additional binding by

```
{W tkBind(event:  '<1>'
          append: true
          action: proc {$} {Browse pop} end)}
```

5.6 More on Actions: Listeners

In previous sections we used procedures or pairs of object and message as actions. Each time an action is invoked, a new thread is created. While this is fine as it comes to efficiency (threads in Oz are light weight), it may cause trouble in that the order in which actions are invoked might be lost: the threads are created in the right order but there is no guarantee that they will run in that order.

The class `Tk.listener` fixes this. An instance of a subclass of `Tk.listener` has a thread of its own in which it serves action messages in order of invocation. For example, in

```
L ={New class $ from Tk.listener
            meth b1 {Browse b1} end
            meth b2 {Browse b2} end
         end tkInit}
B1={New Tk.button tkInit(parent:W text:'One' action: L#b1)}
B2={New Tk.button tkInit(parent:W text:'Two' action: L#b2)}
{Tk.send pack(B1 B2 side:left)}
```

the methods `b1` and `b2` are always executed in the same order in which the corresponding buttons are pressed.

When the `tkInit` method of the class `Tk.listener` is executed, a new thread together with a message stream is created. Whenever an action is invoked, where the object `O` of an object message pair `O#M` is an instance of `Tk.listener`, no new thread is created but `M` is appended at the end of the message stream. The thread then serves the message `M` as soon as all previous messages on the stream have been served completely. It serves `M` by executing the object application `{O M}`.

An additional message `M` to be served can be given to a listener by the method `tkServe`. For example, by

```
{L tkServe(b1)}
```

the message `b1` is served by `L`.

5.7 Entries and Focus

An entry widget lets the user enter a single line of text into the widget. An example is shown in Figure 5.9. The initialization message for the entry widget specifies the width of the entry in characters. The same holds true for the width of label widgets displaying text: a value for the width without an unit appended is taken as width in characters and not in pixels.

Figure 5.9 An entry widget to enter text.


```
L={New Tk.label tkInit(parent:W text:'File name:')}
E={New Tk.entry tkInit(parent:W width:20)}
{Tk.batch [pack(L E side:left pady:1#m padx:1#m)
           focus(E)]}
```

focus To be able to enter text into an entry, the entry needs to have the focus. If a widget has the focus, all input from the keyboard is directed to that widget. A widget that has the focus is drawn with a frame around it. To give the focus to widget, we can use the `focus` command as in the above example.

An entry widget can be given the focus also by clicking it with the mouse button. It is also possible to give the focus to button widgets. That allows to invoke actions with keys, and to move the focus between several widgets by pressing keys. For more on this, see `focus`[9].

returning entered text To query the string entered in a widget the command `get` is provided. To display the entered string in the Browser we can execute

```
{Browse {E tkReturnAtom(get $)}}
```

The string entered in an entry can be deleted, additional characters can be inserted, and so on. More on entry widgets you can find in `entry`[10].

[9] `../tcltk/TkCmd/focus.htm`
[10] `../tcltk/TkCmd/entry.htm`

5.8 Scales

A scale widget allows to select a number from a certain range by moving a slider. Each time the slider is moved, an action attached to the slider is invoked with a single argument giving the current number value of the slider.

In Figure 5.10 an example is shown which allows to display a color determined by three sliders for the intensity of the base colors red, green, and blue. The object `F` stores the intensity for each base color in an attribute. Whenever the method `bg` is executed it changes the intensity for one of the base colors and changes the background color to the combination of all three base colors.

The sliders are configured with the `label` option to display the name of the base color as their labels. The other options besides of `action` and `args` are self explanatory, more information on them can be found in `scale`[11].

The value for the `args` option must be a type specification similar to that used for the specification of argument types in event bindings (see Section 5.5.2. The only difference is that no event argument specification is required. Invoking the action is also similar. For instance, if the scale for the color `red` changes its value to `10`, the message `bg(red 10)` will eventually be served by the listener `L`.

5.9 Listboxes and Scrollbars

scanning A listbox displays a list of strings and allows the user to select one or more of them. If the listbox contains more lines than it can display at once, the user can select the strings to be displayed by scanning the listbox. The listbox can be scanned by pressing the second mouse button and moving the mouse pointer up or down while the button is still being pressed.

A more convenient way than scanning is to use scrollbar widgets. A scrollbar widgets allows the user to determine the portion of strings displayed by moving a slider. Scrollbars are independent of a particular widget type: they can be also attached to other widgets including entries.

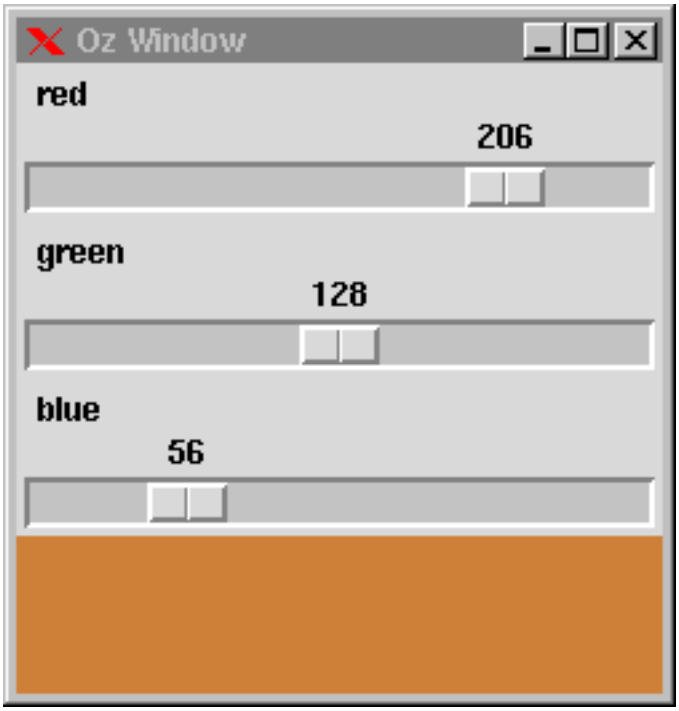
Figure 5.11 shows a program that allows to select a color from a list of colors. The list of colors is provided by some external file inserted at the beginning of the program. The listbox object is initialized and creates an event binding for pressing the left mouse button as follows. First the currently selected index `I` is retrieved (the strings in the list box are indexed). Then the string `C` at this index is retrieved and used as background color of the listbox widget.

attaching scrollbars To attach a scrollbar to a widget we use the predefined procedure `Tk.addYScrollbar`. It creates event bindings for the scrollbar such that moving the scrollbar's slider affects the visible strings of the listbox. Also it creates event bindings for the listbox such that scanning the listbox is reflected by the scrollbar.

More information on listboxes can be found in `listbox`[12] and more information on scrollbars in `scrollbar`[13].

[11] `../tcltk/TkCmd/scale.htm`
[12] `../tcltk/TkCmd/listbox.htm`
[13] `../tcltk/TkCmd/scrollbar.htm`

Figure 5.10 Scales to configure a frame's background color.




```
L ={New class $ from Tk.listener
           attr red:0 green:0 blue:0
           meth bg(C I)
              C := I {F tk(configure bg:c(@red @green @blue))}
           end
        end tkInit}
F ={New Tk.frame tkInit(parent:W height:2#c)}
Ss={Map [red green blue]
    fun {$ C}
       {New Tk.scale tkInit(parent:W orient:horizontal length:8#c
                            label:C 'from':0 to:255
                            action: L # bg(C)
                            args:   [int])}
    end}
{Tk.send pack(b(Ss) F fill:x)}
```

Figure 5.11 A listbox together with a vertical scrollbar.


```
L={New Tk.listbox tkInit(parent:W height:6)}
{L tkBind(event:  '<1>'
          action: proc {$}
                     I={L tkReturn(curselection $)}
                     C={L tkReturn(get(I) $)}
                  in
                     {L tk(configure bg:C)}
                  end)}
S={New Tk.scrollbar tkInit(parent:W)}
{ForAll ⟨Color names 69b⟩
 proc {$ C}
    {L tk(insert 'end' C)}
 end}
{Tk.addYScrollbar L S}
{Tk.send pack(L S fill:y side:left)}
```

5.10 Toplevel Widgets and Window Manager Commands

To manipulate toplevel widgets which are managed by the window manager similar to how other widgets are managed by a geometry manager, Tcl/Tk provides for the `wm` command.

For example, by

```
{Tk.send wm(iconify T)}
```

the toplevel widget `T` is iconified whereas by

```
{Tk.send wm(deiconify T)}
```

it is deiconified. For more information see `wm`[14].

Two important forms of the `wm` command are supported such that they can be given as options to the `tkInit` method of the `Tk.toplevel` class.

[14] `../tcltk/TkCmd/wm.htm`

titled toplevel For example

```
W={New Tk.toplevel tkInit(title:'Something different')}
```

creates a toplevel widget with the title `Something different`.

withdrawn toplevels Sometimes it is important to create a toplevel widget in a withdrawn state: the toplevel widget does not appear on the screen. This can be used to first create all widgets to be contained in the toplevel widget, invoke a geometry manager for them, and only then make the toplevel widget appear on the screen. A toplevel widget can be created in withdrawn state by

```
W={New Tk.toplevel tkInit(withdraw:true)}
```

To make the toplevel widget appear, the window manager command

```
{Tk.send wm(deiconify W)}
```

can be used.

Reference information on the window manager command can be found in `wm`[15].

5.11 Selecting Files

Tk provides for predefined dialogs to select files for being opened or for being saved.

Selecting a file to be opened can be done with the command `tk_getOpenFile`. For example, an arbitrary file can be selected as shown in Figure 5.12. If the command returns the empty string (that is `nil`), the selection dialog has been canceled. Otherwise, the string `S` gives the filename of the file to be opened.

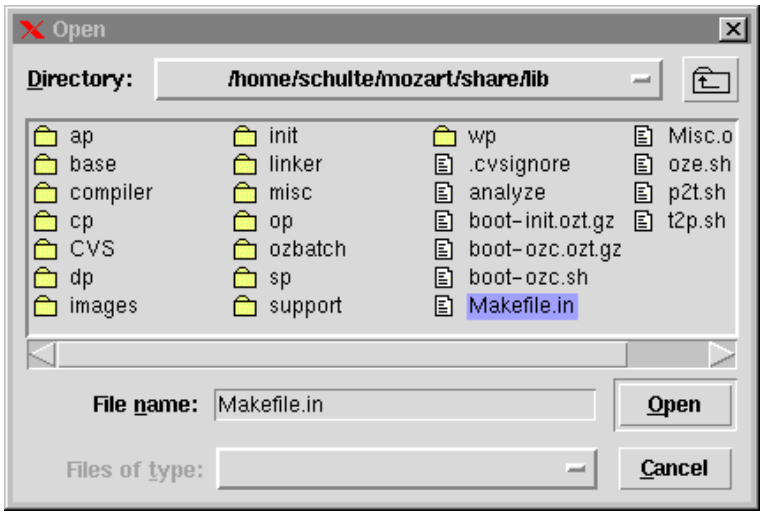
The visual appearance of the file selector depends on the operating system Oz runs on. For example, the file selector for Unix based operating systems is shown in Figure 5.12. Running Oz under Windows uses the Windows specific file selector dialog.

To select filenames for saving the command `tk_getSaveFile` can be used in the same way as above. The difference is that this command does not require that a file with the selected filename already exists.

Reference information on both commands can be found in `tk_getOpenFile`[16].

[15] `../tcltk/TkCmd/wm.htm`
[16] `../tcltk/TkCmd/tk_getOpenFile.htm`

Figure 5.12 Selecting files.




```
case {Tk.return tk_getOpenFile}
of nil then skip
elseof S then {Browse file({String.toAtom S})}
end
```

5.12 Example: Help Popups

In the following we want to look at a small example which provides for a generic interactive help popup window. The idea is that if the mouse pointer stays over a widget for some time without pressing a mouse button, a small help text is displayed. The help text should disappear if the mouse pointer leaves the screen area covered by the widget.

We will build a procedure `AttachHelp` such that help texts are enabled by application of the procedure to a widget and a help text. We proceed in three steps, the first is to create a function to create a toplevel widget that displays the help text, the second is a listener class (that is, a subclass of `Tk.listener`), and the last step is the definition of `AttachHelp` itself.

5.12.1 Displaying Help

The procedure `MakePopup` shown in Figure 5.13 takes a widget and the help text as its argument and returns a function to create a toplevel widget containing the text at a position relative to the widget on the screen.

The returned function creates a toplevel widget in withdrawn state and configures the toplevel widget such that it:

1. is not equipped with a frame from the window manager. This is done by using the window manager command `overrideredirect`: the window manager is advised to not ‘redirect’ the toplevel widget into a frame.

Figure 5.13 Creating a help window.

45a ⟨**Definition of MakePopup** 45a⟩≡

```
fun {MakePopup Parent Text}
   fun {$}
      [X Y H]={Map [rootx rooty height]
               fun {$ WI}
                  {Tk.returnInt winfo(WI Parent)}
               end}
      W={New Tk.toplevel tkInit(withdraw:true bg:black)}
      M={New Tk.message
         tkInit(parent:W text:Text bg:khaki aspect:400)}
   in
      {Tk.batch [wm(overrideredirect W true)
                 wm(geometry W '+'#X+10#'+'#Y+H)
                 pack(M padx:2 pady:2)
                 wm(deiconify W)]}
      W
   end
end
```

2. appears at a position relative to `X` and `Y` coordinates of the widget parent, which done by the `geometry` window manager command.

3. appears on the screen by deiconifying it.

5.12.2 The Listener Class

The listener class `HelpListener` is shown in Figure 5.14. The method `init` initializes an instance of this class by creating a procedure for creation of the popup widget.

When the mouse pointer enters the parent widget, the method `enter` gets executed. This method stores a new variable `C` in the attribute `cancel` which serves as flag whether the help popup must be closed. The newly created thread waits until either one second has elapsed (`A` gets bound after 1000 milliseconds) or the widget has been left (`C` gets bound). Then possibly the widget for the help text is created, which gets closed when the parent widget is left. Note that if both `A` and `C` happen to be bound at the same time, the popup will be created and then closed immediately.

The `leave` method signals that the help popup must be closed by binding the variable stored in `cancel`.

5.12.3 `AttachHelp`

The definition of `AttachHelp` is shown in Figure 5.15. It creates a listener and creates event bindings that are served by that listener.

5.12.4 Using Help Popups

A small example that shows how to use help popups is shown in Figure 5.16.

Figure 5.14 The listener class `HelpListener`.

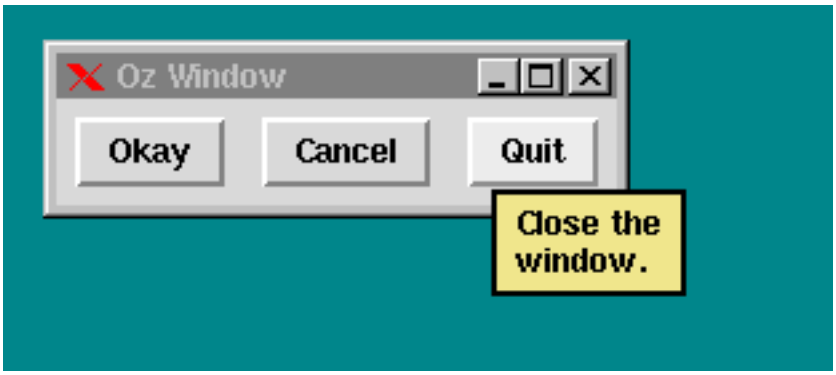
46a ⟨Definition of HelpListener 46a⟩≡

```
class HelpListener from Tk.listener
   attr
      cancel: unit
      popup:  unit
   meth init(parent:P text:T)
      popup := {MakePopup P T}
      Tk.listener,tkInit
   end
   meth enter
      C
   in
      cancel := C
      thread A={Alarm 1000} in
         {WaitOr C A}
         if {IsDet A} then W={@popup} in
            {Wait C} {W tkClose}
         end
      end
   end
   meth leave
      @cancel=unit
   end
end
```

Figure 5.15

```
local
   ⟨Definition of MakePopup 45a⟩
   ⟨Definition of HelpListener 46a⟩
in
   proc {AttachHelp Widget Text}
      L={New HelpListener init(parent:Widget text:Text)}
   in
      {Widget tkBind(event:'<Enter>'  action:L#enter append:true)}
      {Widget tkBind(event:'<Leave>'  action:L#leave append:true)}
      {Widget tkBind(event:'<Button>' action:L#leave append:true)}
   end
end
```

Figure 5.16 Demo of the `HelpPopup` class.




```
Bs={Map ['Okay'   # 'Do nothing meaningful.'
         'Cancel' # 'Do nothing at all.'
         'Quit'   # 'Close the window.']
    fun {$ Text # Help}
       B={New Tk.button tkInit(parent:W text:Text)}
    in
       {AttachHelp B Help} B
    end}
{Tk.send pack(b(Bs) side:left padx:2#m pady:2#m)}
```

6

Canvas Widgets

Canvas widgets allow to create and manipulate graphical items. In particular, arbitrary widgets can be embedded within canvas widgets. Reference information on canvas widgets can be found in `canvas`[1].

6.1 Getting Started

items A canvas widget displays items. An item is created with the `create` command, followed by coordinates and options. The number of coordinates and the options depend on the particular type of item to be created. An item is of one the following types:

`arc`

An arc item displays a piece of a circle.

`bitmap`

A bitmap item displays a bitmap with a given name.

`image`

Displays an image.

`line`

A line item consists of several connected segments. It is possible to configure line items such that Bézier splines are used.

`oval`

An oval can either be a circle or an ellipsis.

`polygon`

A polygon is described by three or more line segments. As with line items, it is possible to use Bézier splines.

`rectangle`

Displays a rectangle.

[1] `../tcltk/TkCmd/canvas.htm`

`text`

Displays text consisting of a single or several lines.

`window`

Displays a widget in the canvas where the canvas widget serves as geometry manager for the widget.

For example,

```
C={New Tk.canvas tkInit(parent:W)}
{Tk.send pack(C)}
{C tk(create rectangle 10 10 1#c 1#c fill:red outline:blue)}
```

creates a red rectangle with a blue outline near to the upper left corner of the canvas widget `C`. More information on the different items can be found in `canvas`[2].

6.2 Example: Drawing Bar Charts

As a more interesting example let us consider a program to draw bar charts. The definition of a class to display barcharts is shown in Figure 6.1. Before any item is created in the canvas by the method `bars`, the canvas widget is configured such that the scrollable region is just large enough for the barchart to be drawn.

The method `DrawBars` creates for each element of the list `Ys` a rectangle item as well as a text item, which both correspond to the value of the particular item. The value of `O` is used as option for the rectangle items. This value depends on `Tk.isColor` which is `true` if the screen is a color screen, and `false` otherwise. For a color screen the rectangle items are filled with the color `wheat`. For a black and white screen, the rectangle items are drawn in a stippled fashion: only those pixels are drawn with the fill color (that is `black`) where the stipple bitmap contains a pixel.

Figure 6.2 shows how the bar chart canvas is used in order to display data.

6.3 Canvas Tags

Each item in a canvas is identified by a unique integer. This integer can be returned by using the `tkReturnInt` method for creating items rather than the `tk` method. The returned integer can then be used to manipulate the corresponding item. However, returning values from the graphics engine involves latency. But there are some good news here, since it is not necessary to refer to items by numbers.

tags Canvas widgets offer a more powerful and easier method to manipulate single items or even groups of items. Items can be referred to by tags. A single item can be referred to by as many tags as you like to. Tags are provided as objects in Oz. Before an item can be added to a tag, a tag object must be created from the class `Tk.canvasTag` and initialized with respect to a particular canvas.

To add an item to a tag, the option `tags` is used when creating the item. For instance,

[2] `../tcltk/TkCmd/canvas.htm`

Figure 6.1 A canvas for displaying bar charts.

```
local
   O=if Tk.isColor then o(fill:wheat)
     else              o(stipple:gray50 fill:black)
     end
   D=10  D2=2*D B=10
in
   class BarCanvas from Tk.canvas
      meth DrawBars(Ys H X)
         case Ys of nil then skip
         [] Y|Yr then
            {self tk(create rectangle X H X+D H-Y*D2 O)}
            {self tk(create text      X H+D text:Y anchor:w)}
            {self DrawBars(Yr H X+D2)}
         end
      end
      meth configure(SX SY)
         {self tk(configure scrollregion:q(B ~B SX+B SY+B))}
      end
      meth bars(Ys)
         WY=D2*({Length Ys}+1) HY=D2*({FoldL Ys Max 0}+1)
      in
         {self configure(WY HY)}
         {self DrawBars(Ys HY D)}
      end
   end
end
```

```
R={New Tk.canvasTag tkInit(parent:C)}
{C tk(create rectangle 10 10 40 40 fill:red tags:R)}
```

creates a new rectangle item and adds it to the tag `R`.

A second oval item can be added to the tag `R` by

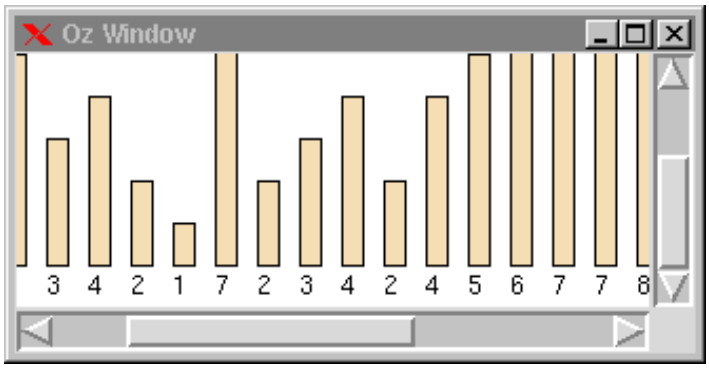
```
{C tk(create oval 20 20 40 40 tags:R)}
```

All items referred to by a tag can be manipulated simultaneously. The following moves all items 40 pixels to the right:

```
{R tk(move 40 0)}
```

Figure 6.3 shows a small program that creates items interactively. Pressing the mouse button over the canvas widget creates either a rectangle item or an oval item at the position of the mouse pointer. All rectangle items created are added to the tag `R`, and all oval items are added to the tag `O`.

Figure 6.2 Using a canvas for drawing barcharts.




```
C={New BarCanvas    tkInit(parent:W bg:white width:300 height:120)}
H={New Tk.scrollbar tkInit(parent:W orient:horizontal)}
V={New Tk.scrollbar tkInit(parent:W orient:vertical)}
{Tk.addXScrollbar C H} {Tk.addYScrollbar C V}
{Tk.batch [grid(C row:0 column:0)
           grid(H row:1 column:0 sticky:we)
           grid(V row:0 column:1 sticky:ns)]}
{C bars([1 3 4 5 3 4 2 1 7 2 3 4 2 4
         5 6 7 7 8 4 3 5 6 7 7 8 4 3])}
```

configuring items Items can be configured with the command `itemconfigure`, which is similar to the command `configure` for widgets. The color of all rectangle and oval items in our previous example can be changed by:

```
{R tk(itemconfigure fill:wheat)}
{O tk(itemconfigure fill:blue)}
```

Besides of the `move` command there are other commands for manipulating items. For instance, executing the following statement

```
{O tk(delete)}
```

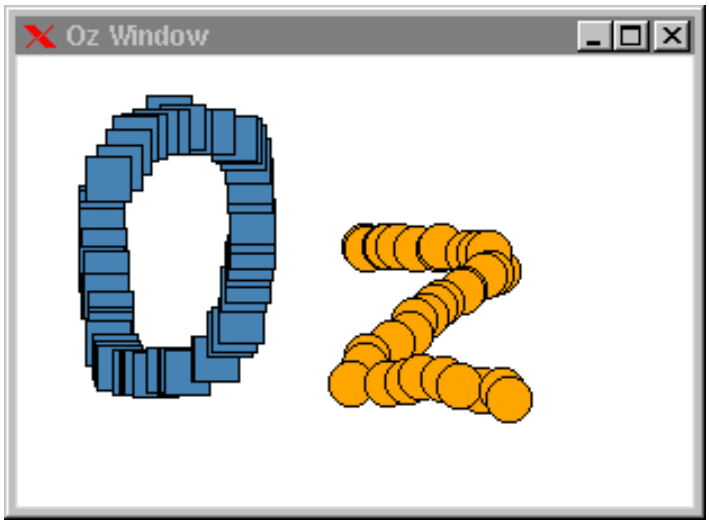
deletes all oval items attached to the tag `O`. Other commands allow to scale items, to change the coordinates of items and so on. More information on possible commands are available from `canvas`[3].

6.3.1 Event Bindings

Similar to widgets, event bindings can be created for tags. Creating an event binding for a tag means to create the binding for all items referred to by the tag. The following example creates an event binding for all oval items.

[3] `../tcltk/TkCmd/canvas.htm`

Figure 6.3 A canvas for creating rectangles and ovals.




```
C={New Tk.canvas    tkInit(parent:W width:300 height:200 bg:white)}
R={New Tk.canvasTag tkInit(parent:C)}
O={New Tk.canvasTag tkInit(parent:C)}
{C tkBind(event:  '<1>'
          args:   [int(x) int(y)]
          action: proc {$ X Y}
                     {C tk(create rectangle X-10 Y-10 X+10 Y+10
                           tags:R fill:steelblue)}
                  end)}
{C tkBind(event:  '<2>'
          args:   [int(x) int(y)]
          action: proc {$ X Y}
                     {C tk(create oval X-10 Y-10 X+10 Y+10
                           tags:O fill:orange)}
                  end)}
{Tk.send pack(C)}
```

```
Colors={New class $ from BaseObject
               attr cs:(Cs=red|green|blue|yellow|orange|Cs
                        in
                           Cs)
               meth get(?C)
                  Cr in C|Cr = (cs := Cr)
               end
            end noop}
{O tkBind(event:  '<3>'
          action: proc {$}
                     {O tk(itemconfigure
                           fill:{Colors get($)})}
                  end)}
```

Clicking with the right mouse button on an oval item, configures all items referred to by `O` to employ a different color. The `Colors` object serves as color generator. Each time the method `get` is invoked, it returns a color from the circular list of colors stored in the attribute `cs`.

6.4 Example: An Animated Time Waster

In this section we want to program a procedure that signals to the user that a particular computation is still running and entertains the user by some animation.

Figure 6.4 shows a subclass of `Tk.canvasTag` that creates a bitmap item showing a magnifying glass and starts a thread to move that bitmap randomly. The random movement can be stopped by binding the variable `D` given as value for the feature `done`. If the animation has stopped indeed, the variable `S` gets bound, as you can see in method `move`.

Figure 6.4 An animated time waster class.

```
local
   fun {RandCoord} {OS.rand} mod 20 + 15 end
in
   class RandMag from Tk.canvasTag
      meth init(parent:P done:D stopped:S)
         {self tkInit(parent:P)}
         {P tk(create bitmap 0 0
               bitmap:'@'#{Property.get 'oz.home'}#
                      '/doc/wp/magnifier.xbm'
               tags:self foreground:blue)}
         thread {self move(D S)} end
      end
      meth move(D S)
         {WaitOr {Alarm 400} D}
         if {IsDet D} then S=unit else
            {self tk(coords {RandCoord} {RandCoord})}
            {self move(D S)}
         end
      end
   end
end
```

The procedure `WaitDone` shown in Figure 6.5 takes a variable `Done` which is used for signalling when the computation we are waiting for is finished. It creates a randomly moving magnifier item and as soon as the magnifier signals that it has been stopped (by `Stopped`) the toplevel windows is closed.

For example,

```
declare Done
{WaitDone Done}
```

Figure 6.5 A procedure for creating wait dialogs.


```
proc {WaitDone Done}
   W={New Tk.toplevel tkInit(withdraw:true)}
   L={New Tk.label    tkInit(parent:W text:'Computing...')}
   C={New Tk.canvas   tkInit(parent:W width:50 height:50)}
   Stopped
in
   {Tk.batch [wm(overrideredirect W true)
              pack(L C side:left pady:2#m padx:2#m)
              wm(deiconify W)]}
   _={New RandMag init(parent:C done:Done stopped:Stopped)}
   thread {Wait Stopped} {W tkClose} end
end
```

creates a waiting dialog which disappears by binding `Done`

```
Done=unit
```

7

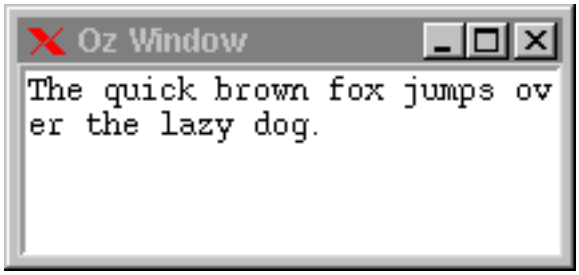
Text Widgets

Text widgets display (as suggested by the name) one or more lines of text, where the text can be edited. It offers commands to manipulate segments of text and to embed other widgets into the flow of text. This chapter attempts to give a short overview on text widgets, for the details consult `text`[1].

7.1 Manipulating Text

Let us start with a very simple example where we want to display a given text in a text widget. Figure 7.1 shows a program that does the job.

Figure 7.1 Displaying text.




```
T={New Tk.text tkInit(parent:W width:28 height:5 bg:white)}
{T tk(insert 'end' "The quick brown fox jumps over the lazy dog.")}
```

scanning Similar to listboxes (page 40), a text widget supports scanning: The text can be scanned by pressing the second mouse button and moving the mouse pointer while the button is still being pressed. And of course, in the same way as canvas widgets scrollbars can be attached to a text widget.

text wrapping The text is wrapped where word boundaries (that is, spaces) are not taken into account. Changing the wrapping such that word boundaries are preserved can be done as follows:

```
{T tk(configure wrap:word)}
```

[1] `../tcltk/TkCmd/text.htm`

positions Positions in the displayed text can be referred to by positions. A position can be denoted by a tickle `p(L C)`, where `L` gives the line (starting from one) and `C` the position in that line (also starting from zero). Positions also can take modifiers, for more details on this issue see `text`[2]. Another helpful position is `'end'` which refers to the position after the last character.

getting text Portions of the text can be retrieved. For example,

```
{T tkReturnAtom(get p(1 4) p(1 9) $)}
```

returns the atom `quick`.

inserting text Positions also specify where to insert text, for example

```
{T tk(insert p(1 4) "very very ")}
```

inserts the text directly before quick.

deleting text In the same way text can also be deleted. For example

```
{T tk(delete p(1 4) p(1 14))}
```

deletes again the text `"very very "`.

disabling input We do not discuss here how to employ a text widget as a powerful editor, please see again `text`[3]. If you try to place a cursor inside the text widget and make some character strokes, you will notice that by default a text widget accepts input. To prevent a user from altering the text in a display only situation the widget can be configured as follows:

```
{T tk(configure state:disabled)}
```

7.2 Text Tags and Marks

creating tags In the same way as canvas widgets, text widgets support tags. While canvas tags refer to sets of items (see Section 7.2), text tags refer to sets of characters and allow to configure and manipulate the set of characters. For example, the following

```
B={New Tk.textTag tkInit(parent:T foreground:brown)}
```

creates a new tag, where the tag is configured such that all characters that will be referred to by this tag (initially, no characters) are displayed in brown color.

[2] `../tcltk/TkCmd/text.htm`
[3] `../tcltk/TkCmd/text.htm`

adding text Already inserted text can be added to a tag by defining the text portion to be added with positions. The following

```
{B tk(add p(1 10) p(1 15))}
```

adds the text part `"brown"` to the tag `B`, which changes the color of that text to brown.

configuring tags Changing the configuration of a tag takes effect on all characters that are referred to by that tag. For example, if the tag `B` is configured for a larger font as follows

```
{B tk(configure font:{New Tk.font tkInit(size:18)})}
```

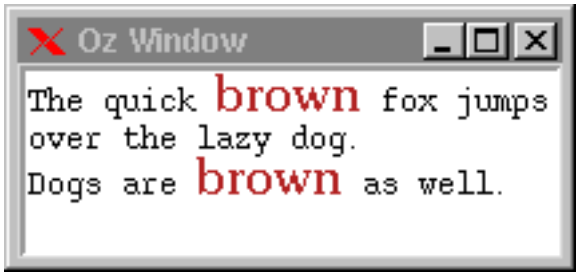
the text portion `"brown"` now appears in that larger font.

inserting and adding text The `insert` command also supports tags directly. The following

```
{T tk(insert 'end' "\nDogs are ")}
{T tk(insert 'end' "brown" B)}
{T tk(insert 'end' " as well.")}
```

adds three portions of text to the text widget, where the text `"brown"` is both inserted and added to the tag `B`, which makes it appear both in brown color and with a large font. Now the text widget looks as shown in Figure 7.2.

Figure 7.2 Using tags with text widgets.



In the same way as described in Section 6.3 for canvas tags, events can be attached to text tags. We will exemplify this in the next section (page 60).

In addition to tags, text widgets also support marks. Marks refer to positions in the text rather than to particular characters as tags do. They are supported by the class `Tk.textMark`. For their use, see again `text`[4].

[4] `../tcltk/TkCmd/text.htm`

7.3 Example: A ToyText Browser

In the following we discuss a tiny ToyText browser that supports following of links in the text and going back to previously displayed pages. A ToyText hypertext is a record, where the features are the links and the fields describe pages. For an example, see (page 70). A page consists out of head and body, where the body is a list of elements. An element is either a virtual string or a record `a(ref:R Es)`, where R is a feature of the record and `Es` is a list of elements.

Figure 7.3 shows the main routine to display a ToyText page in a text widget `T`. The procedure `Display` takes a list of references `Rs` as input, and displays the page that is referred to by its first element.

Figure 7.3 Displaying a ToyText page.

60a ⟨**Definition of Display** 60a⟩≡

```
proc {Display Rs}
   case Rs of nil then skip
   [] R|Rr then
     {T tk(delete p(0 0) 'end')}
     {Head ToyText.R.head Rr} {Body ToyText.R.body Rs}
   end
end
```

Figure 7.4 shows how the head of a ToyText page is displayed, where `E` is the virtual string to be displayed and `Rs` is the list of current references without its first element. The tag `HT` is configured such that clicking it displays the previous page.

Figure 7.4 Displaying the head of a ToyText page.

60b ⟨**Definition of Head** 60b⟩≡

```
local
   HF={New Tk.font    tkInit(family:helvetica size:18 weight:bold)}
   HT={New Tk.textTag tkInit(parent:T font:HF foreground:orange)}
in
   proc {Head E Rs}
      {T tk(insert p(0 0) E#'\n' HT)}
      {HT tkBind(event:  '<1>'
                 action: proc {$} {Display Rs} end)}
   end
end
```

Figure 7.5 shows how the body of a ToyText page is displayed, where `Es` is the list of elements, `CT` is the current tag to which inserted text is added, and `Rs` are the current references, including a reference to the page currently under display as first element. To display a reference element, a new tag `RT` is created that carries as action a procedure that displays the referred page.

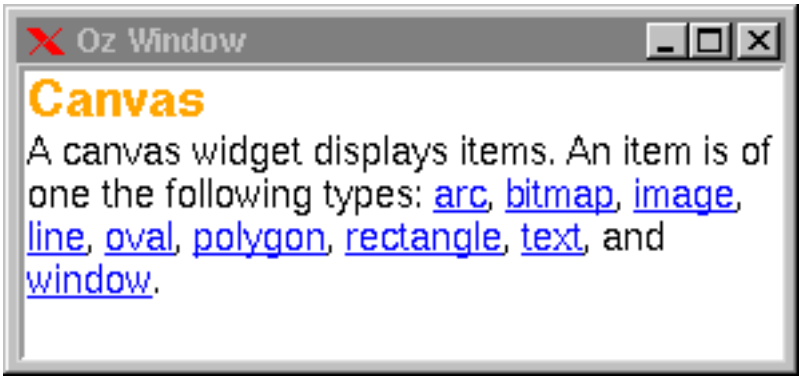
Figure 7.6 shows the complete ToyText browser and how it looks when displaying pages.

Figure 7.5 Displaying the body of a ToyText page.

61a ⟨**Definition of Body** 61a⟩≡

```
local
   BF={New Tk.font    tkInit(family:helvetica size:12 weight:normal)}
   BT={New Tk.textTag tkInit(parent:T font:BF)}
   proc {Do Es CT Rs}
      case Es of nil then skip
      [] E|Er then
         case E
         of a(ref:R Es) then
            RT={New Tk.textTag tkInit(parent:T font:BF
                                      foreground:blue underline:true)}
         in
            {RT tkBind(event: '<1>'
                       action: proc {$} {Display R|Rs} end)}
            {Do Es RT Rs}
         else
            {T tk(insert 'end' E CT)}
         end
         {Do Er CT Rs}
      end
   end
in
   proc {Body Es Rs}
      {Do Es BT Rs}
   end
end
```

Figure 7.6 A ToyText browser.




```
proc {ToyBrowse ToyText Root}
   W={New Tk.toplevel tkInit}
   T={New Tk.text tkInit(parent:W width:40 height:8 bg:white wrap:word)}
   ⟨Definition of Head 60b⟩
   ⟨Definition of Body 61a⟩
   ⟨Definition of Display 60a⟩
in
   {Tk.send pack(T)}
   {Display [Root]}
end
{ToyBrowse ⟨Sample ToyText 70a⟩ canvas}
```

8

Tools for Tk

This chapter presents some common graphical abstractions you might find useful when building graphical user interfaces in Oz. The graphical abstractions are provided by the module `TkTools` and are built on top of the functionality provided by the `Tk` module.

8.1 Dialogs

A dialog displays some graphical information and several buttons. A simple abstraction to build dialogs is provided by the class `TkTools.dialog`.

Figure 8.1 shows an example dialog for deleting a file. The class `TkTools.dialog` is a subclass of `Tk.frame`. Creating and initializing an instance of this class creates a toplevel widget together with buttons displayed at the bottom of the toplevel widget. The instance itself serves as container for user-defined widgets that are displayed at the top of the dialog (as the label and the entry widget in our example).

The initialization message for a dialog must contain the `title` option, which gives the title of the dialog. The buttons to be displayed are specified by a list of pairs, where the first pair in the list describes the leftmost button. The pair consists of the label of the button and the action for the button. The action can be also the atom `tkClose`, which means that the action of the button sends a `tkClose` message to the dialog. In a similar manner, the action can be a unary tuple with label `tkClose`, which means that first the action specified by the tuple's argument is executed and then the dialog is closed. The `default` option specifies which button should be the default one. The default button is marked by a sunken frame drawn around the button.

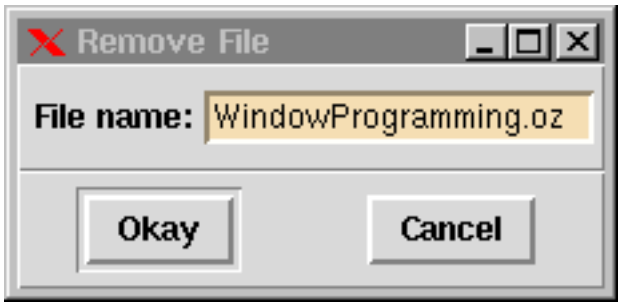
In the above example, pressing the `'Okay'` button executes an `rm` command to remove the file with the name as given by the entry widget `E`. Only if execution of this command returns `0`, the dialog is closed.

The class `TkTools.dialog` is a subclass of `Tk.frame`. In particular it allows to wait until the dialog object gets closed. For example, the execution of

```
{Wait D.tkClosed}
```

blocks until the dialog in the above example is closed.

Figure 8.1 A dialog to remove files.




```
D={New TkTools.dialog
   tkInit(title:   'Remove File'
          buttons: ['Okay' #
                    proc {$}
                       try
                          {OS.unlink {E tkReturn(get $)}}
                          {D tkClose}
                       catch _ then skip
                       end
                    end
                    'Cancel' # tkClose]
          default: 1)}
L={New Tk.label tkInit(parent:D text:'File name:')}
E={New Tk.entry tkInit(parent:D bg:wheat width:20)}
{Tk.batch [pack(L E side:left pady:2#m) focus(E)]}
```

8.2 Error Dialogs

A dialog to display error messages is provided by the class `TkTools.error`, which is a sub class of `TkTools.dialog`. Figure 8.2 shows an example of how to use `TkTools.error`.

transient widgets All dialogs provide for the option `master`. By giving a toplevel widget as value for `master`, the dialog appears as a transient widget: depending on the window manager the widget appears with less decoration, e.g., no title, on the screen. Moreover, when the master widget is closed, the dialog is closed as well.

8.3 Menubars

keyboard accelerators A menubar is a frame widget containing several menubutton widgets. To each of the menubutton widgets a menu is attached. The menu contains menuitems being either radiobutton entries, checkbutton entries, command entries (similar to button widgets), separator entries or cascade entries to which sub menus are attached. The menu entries can be equipped with keyboard accelerators describing key event bindings that can be used to invoke the action of the menu entry. A keyboard

Figure 8.2 A transient error dialog.


```
E={New TkTools.error
   tkInit(master:W
          text: 'Error in system configuration: '#
                'too much memory.')}
```

accelerator must be added to the menu entry and the right event binding needs to be created.

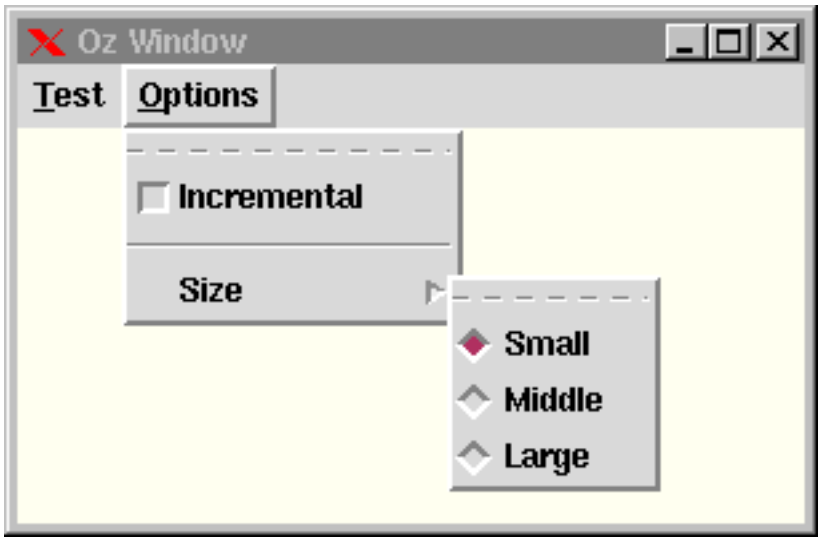
Creating a menubar by hand has to follow this structure and is inconvenient due to the large numbers of different kinds of widgets and menu entries that are to be created. To ease the creation of a menubar, the `TkTools` module provides the procedure `TkTools.menubar` that creates to a given specification a menubar and creates keyboard accelerators with the right event bindings. The specification of a menubar consists of messages used to initialize the necessary widgets and entries, where the label determines the kind of entry to be created.

Figure 8.3 shows an example for menubar creation. The procedure `TkTools.menubar` takes two widgets and two menubar specifications as input and returns a frame containing the widgets for the menubar. The widget given as first argument serves as parent for the menubar's frame, whereas the widget given as second argument receives the key bindings for the accelerators. The specification given as third (fourth) argument describe the left (right) part of the menubar.

A menubar specification consists of a list of `menubutton` messages. The valid options are those for the `tkInit` method of a menubutton widget object, where the parent field is not necessary, and the additional options `menu` and `feature`. The value for the `menu` option must be a list of specifications for the menu entries. The menu entries are specified similar to the menubuttons, the allow for the additional options `feature`, `key`, and `event`.

The value for the `key` describes the keyboard accelerator and event binding to be created. They can be used as follows:

Figure 8.3 A menubar.




```
V={New Tk.variable tkInit(0)}
B={TkTools.menubar W W
   [menubutton(text:'Test' underline:0
               menu: [command(label:   'About test'
                              action:  Browse#about
                              key:     alt(a)
                              feature: about)
                      separator
                      command(label:   'Quit'
                              action:  W#tkClose
                              key:     ctrl(c)
                              feature: quit)]
                      feature: test)
    menubutton(text:'Options' underline:0
               menu: [checkbutton(label: 'Incremental'
                                  var: {New Tk.variable tkInit(false)})
                      separator
                      cascade(label: 'Size'
                              menu: [radiobutton(label:'Small'
                                                 var:V value:0)
                                     radiobutton(label:'Middle'
                                                 var:V value:1)
                                     radiobutton(label:'Large'
                                                 var:V value: 2)])])]
   nil}
F={New Tk.frame tkInit(parent:W width:10#c height:5#c bg:ivory)}
{Tk.send pack(B F fill:x)}
```

key option value	accelerator	event binding
a	a	a
ctrl(a)	C-a	<Control-a>
alt(a)	A-a	<Alt-a>
alt(ctrl(a))	A-C-a	<Alt-Control-a>
ctrl(alt(a))	C-A-a	<Control-Alt-a>

In case one wants to use different event bindings than those generated from the `key` option value, one can specify the event pattern as value for the option `event`.

The `feature` options for menubuttons and menu entries attach features to the created objects such that the object get accessible by these features. For instance, to disable the 'About test' entry is possible with

```
{B.test.about tk(entryconfigure state:disabled)}
```

The menus attached to menubuttons or to cascade entries can be accessed under the feature `menu`. For instance the first tear off entry from the 'Test' menu can be removed with

```
{B.test.menu tk(configure tearoff:false)}
```

It is possible to extend a menubar created with `TkTools.menubar` with further entries. The following statement adds a menu entry just before the `Quit` entry:

```
A={New Tk.menuentry.command tkInit(parent:B.test.menu
                                   before:B.test.quit
                                   label: 'Exit')}
```

which can be deleted and removed from the menu again by:

```
{A tkClose}
```

8.4 Handling Images

A convenient way to create images is given by `TkTools.images`. It takes a list of URLs as input and returns a record of images, where the fields are atoms derived naturally from the URLs. The type and format of images is handled according to the extension of the URL.

For example,

```
U='http://www.mozart-oz.org/home-1.1.0/doc/wp/'
I={TkTools.images [U#'wp.gif'
                   U#'queen.xbm'
                   U#'truck-left.ppm']}
```

binds `I` to a record with features `wp`, `queen`, and `'truck-left'`, where the fields are the corresponding images.

First the basename of the URL is computed by taking the last fragment of the URL (that is, 'wp.gif' for example). The extension (the part following the period, 'gif' for example), determines the type and format of the image. The part of the basename that precedes the period yields the feature.

A

Data and Program Fragments

The following appendix features some program fragments and data specifications omitted in the chapters' text.

A.1 Getting Started

69a ⟨Change capitalization 69a⟩≡

```
fun {$ I}
   case {Char.type I}
   of lower then {Char.toUpper I}
   [] upper then {Char.toLower I}
   else I
   end
end
```

A.2 More on Widgets

69b ⟨Color names 69b⟩≡

```
[aliceblue       antiquewhite       aquamarine
 azure           beige              bisque
 black           blanchedalmond     blue
 blueviolet      brown              burlywood
 cadetblue       chartreuse         chocolate
 coral           cornflowerblue     cornsilk
 cyan            darkblue           darkcyan
 darkgoldenrod   darkgray           darkgreen
 darkgrey        darkkhaki          darkmagenta
 darkolivegreen  darkorange         darkorchid
 darkred         darksalmon         darkseagreen
 darkslateblue   darkslategray      darkslategrey
 darkturquoise   darkviolet         deeppink
 deepskyblue     dimgray            dimgrey
 dodgerblue      firebrick          floralwhite
 forestgreen     gainsboro          ghostwhite
 gold            goldenrod          gray
 green           greenyellow        grey
```

```
honeydew            hotpink              indianred
ivory               khaki                lavender
lavenderblush       lawngreen            lemonchiffon
lightblue           lightcoral           lightcyan
lightgoldenrod      lightgoldenrodyellow lightgray
lightgreen          lightgrey            lightpink
lightsalmon         lightseagreen        lightskyblue
lightslateblue      lightslategray       lightslategrey
lightsteelblue      lightyellow          limegreen
linen               magenta              maroon
mediumaquamarine    mediumblue           mediumorchid
mediumpurple        mediumseagreen       mediumslateblue
mediumspringgreen   mediumturquoise      mediumvioletred
midnightblue        mintcream            mistyrose
moccasin            navajowhite          navy
navyblue            oldlace              olivedrab
orange              orangered            orchid
palegoldenrod       palegreen            paleturquoise
palevioletred       papayawhip           peachpuff
peru                pink                 plum
powderblue          purple               red
rosybrown           royalblue            saddlebrown
salmon              sandybrown           seagreen
seashell            sienna               skyblue
slateblue           slategray            slategrey
snow                springgreen          steelblue
tan                 thistle              tomato
turquoise           violet               violetred
wheat               white                whitesmoke
yellow              yellowgreen]
```

A.3 Text Widgets

70a ⟨Sample ToyText 70a⟩≡

```
hyper(canvas:
        e(head:'Canvas'
          body:['A canvas widget displays items. '
                'An item is of one the following types: '
                a(ref:arc       ['arc']) ', '
                a(ref:bitmap    ['bitmap']) ', '
                a(ref:image     ['image']) ', '
                a(ref:line      ['line']) ', '
                a(ref:oval      ['oval']) ', '
                a(ref:polygon   ['polygon']) ', '
                a(ref:rectangle ['rectangle']) ', '
                a(ref:text      ['text']) ', and '
                a(ref:window    ['window']) '.'])
      arc:
        e(head:'Arc'
```

```
      body:['An arc item displays a piece of a '
            'circle.'])
bitmap:
   e(head:'Bitmap'
     body:['A bitmap item displays a bitmap '
           'with a given name.'])
image:
   e(head:'Image'
     body:['Displays an image.'])
line:
   e(head:'Line'
     body:['A line item consists of several '
           'connected segments.'])
oval:
   e(head:'Oval'
     body:['An oval can either be a circle or '
           'an ellipsis.'])
polygon:
   e(head:'Polygon'
     body:['A polygon is described by three or '
           'more ' a(ref:line ['line'])
           ' segments.'])
rectangle:
   e(head:'Rectangle'
     body:['Displays a rectangle.'])
text:
   e(head:'Text'
     body:['Displays text consisting of a single '
           'or several lines.'])
window:
   e(head:'Window'
     body:['Displays a widget in the canvas where '
           'the canvas widget serves as geometry '
           'manager for the widget. '
           'See also '
           a(ref:canvas ['the canvas widget']) '.']))
```

Bibliography

[1] Mark Harrison and Michael McLennan. *Effective Tcl/Tk Programming: Writing Better Programs with Tcl and Tk*. Professional Computing Series. Addison-Wesley, Reading, MA, USA, 1998.

[2] John K. Ousterhout. *Tcl and the Tk Toolkit*. Professional Computing Series. Addison-Wesley, Reading, MA, USA, 1994.

Index