Chapter 19

Ruby Tk

The Ruby Application Archive contains several extensions that provide Ruby with a graphical user interface (GUI), including extensions for Fox, GTK, and others.

The Tk extension is bundled in the main distribution and works on both Unix and Windows systems. To use it, you need to have Tk installed on your system. Tk is a large system, and entire books have been written about it, so we won't waste time or resources by delving too deeply into Tk itself but instead concentrate on how to access Tk features from Ruby. You'll need one of these reference books in order to use Tk with Ruby effectively. The binding we use is closest to the Perl binding, so you probably want to get a copy of *Learning Perl/Tk* [Wal99] or *Perl/Tk Pocket Reference* [Lid98].

Tk works along a composition model—that is, you start by creating a container (such as a `TkFrame` or `TkRoot`) and then create the widgets (another name for GUI components) that populate it, such as buttons or labels. When you are ready to start the GUI, you invoke `Tk.mainloop`. The Tk engine then takes control of the program, displaying widgets and calling your code in response to GUI events.

Simple Tk Application

A simple Tk application in Ruby may look something like this.

```
require 'tk'
root = TkRoot.new { title "Ex1" }
TkLabel.new(root) do
  text  'Hello, World!'
  pack('padx' => 15, 'pady' => 15, 'side' => 'left')
end
Tk.mainloop
```

Let's look at the code a little more closely. After loading the `tk` extension module, we create a root-level frame using `TkRoot.new`. We then make a `TkLabel` widget as a

child of the root frame, setting several options for the label. Finally, we pack the root frame and enter the main GUI event loop.

It's a good habit to specify the root explicitly, but you could leave it out—along with the extra options—and boil this down to a three-liner.

```
require 'tk'
TkLabel.new { text 'Hello, World!'; pack }
Tk.mainloop
```

That's all there is to it! Armed with one of the Perl/Tk books we reference at the start of this chapter, you can now produce all the sophisticated GUIs you need. But then again, if you'd like to stick around for some more details, here they come.

Widgets

Creating widgets is easy. Take the name of the widget as given in the Tk documentation and add a `Tk` to the front of it. For instance, the widgets `Label`, `Button`, and `Entry` become the classes `TkLabel`, `TkButton`, and `TkEntry`. You create an instance of a widget using `new`, just as you would any other object. If you don't specify a parent for a given widget, it will default to the root-level frame. We usually want to specify the parent of a given widget, along with many other options—color, size, and so on. We also need to be able to get information back from our widgets while our program is running by setting up *callbacks* (routines invoked when certain events happen) and sharing data.

Setting Widget Options

If you look at a Tk reference manual (the one written for Perl/Tk, for example), you'll notice that options for widgets are usually listed with a hyphen—as a command-line option would be. In Perl/Tk, options are passed to a widget in a `Hash`. You can do that in Ruby as well, but you can also pass options using a code block; the name of the option is used as a method name within the block and arguments to the option appear as arguments to the method call. Widgets take a parent as the first argument, followed by an optional hash of options or the code block of options. Thus, the following two forms are equivalent.

```
TkLabel.new(parent_widget) do
  text    'Hello, World!'
  pack('padx'  => 5,
       'pady'  => 5,
       'side'  => 'left')
end
# or
TkLabel.new(parent_widget, 'text' => 'Hello, World!').pack(...)
```

One small caution when using the code block form: the scope of variables is not what you think it is. The block is actually evaluated in the context of the widget's object, not the caller's. This means that the caller's instance variables will not be available in the block, but local variables from the enclosing scope and globals will be (not that you use global variables, of course.) We'll show option passing using both methods in the examples that follow.

Distances (as in the `padx` and `pady` options in these examples) are assumed to be in pixels but may be specified in different units using one of the suffixes `c` (centimeter), `i` (inch), `m` (millimeter), or `p` (point). `"12p"`, for example, is twelve points.

Getting Widget Data

We can get information back from widgets by using callbacks and by binding variables.

Callbacks are very easy to set up. The `command` option (shown in the `TkButton` call in the example that follows) takes a `Proc` object, which will be called when the callback fires. Here we pass the proc in as a block associated with the method call, but we could also have used `Kernel.lambda` to generate an explicit `Proc` object.

```
require 'tk'
TkButton.new do
  text "EXIT"
  command { exit }
  pack('side'=>'left', 'padx'=>10, 'pady'=>10)
end
Tk.mainloop
```

We can also bind a Ruby variable to a Tk widget's value using a `TkVariable` proxy. This arranges things so that whenever the widget's value changes, the Ruby variable will automatically be updated, and whenever the variable is changed, the widget will reflect the new value.

We show this in the following example. Notice how the `TkCheckButton` is set up; the documentation says that the `variable` option takes a *var reference* as an argument. For this, we create a Tk variable reference using `TkVariable.new`. Accessing `mycheck.value` will return the string "0" or "1" depending on whether the checkbox is checked. You can use the same mechanism for anything that supports a var reference, such as radio buttons and text fields.

```
require 'tk'
packing = { 'padx'=>5, 'pady'=>5, 'side' => 'left' }
checked = TkVariable.new
def checked.status
  value == "1" ? "Yes" : "No"
end
```

```
status = TkLabel.new do
  text checked.status
  pack(packing)
end
TkCheckButton.new  do
  variable checked
  pack(packing)
end
TkButton.new do
  text "Show status"
  command { status.text(checked.status) }
  pack(packing)
end
Tk.mainloop
```

Setting/Getting Options Dynamically

In addition to setting a widget's options when it's created, you can reconfigure a widget while it's running. Every widget supports the `configure` method, which takes a `Hash` or a code block in the same manner as `new`. We can modify the first example to change the label text in response to a button click.

```
require 'tk'
root = TkRoot.new { title "Ex3" }
top = TkFrame.new(root) { relief 'raised'; border 5 }
lbl = TkLabel.new(top) do
  justify 'center'
  text    'Hello, World!'
  pack('padx'=>5, 'pady'=>5, 'side' => 'top')
end
TkButton.new(top) do
  text "Ok"
  command { exit }
  pack('side'=>'left', 'padx'=>10, 'pady'=>10)
end
TkButton.new(top) do
  text "Cancel"
  command { lbl.configure('text'=>"Goodbye, Cruel World!") }
  pack('side'=>'right', 'padx'=>10, 'pady'=>10)
end
top.pack('fill'=>'both', 'side' =>'top')
Tk.mainloop
```

Now when the `Cancel` button is clicked, the text in the label will change immediately from "`Hello, World!`" to "`Goodbye, Cruel World!`"

You can also query widgets for particular option values using `cget`.

```
require 'tk'
b = TkButton.new do
  text     "OK"
  justify  "left"
  border   5
end
b.cget('text')      →   "OK"
b.cget('justify')   →   "left"
b.cget('border')    →   5
```

Sample Application

Here's a slightly longer example, showing a genuine application—a pig latin generator. Type in the phrase such as **Ruby rules**, and the `Pig It` button will instantly translate it into pig latin.


```
require 'tk'
class PigBox
  def pig(word)
    leading_cap = word =~ /^[A-Z]/
    word.downcase!
    res = case word
      when /^[aeiouy]/
        word+"way"
      when /^([^aeiouy]+)(.*)/
        $2+$1+"ay"
      else
        word
    end
    leading_cap ? res.capitalize : res
  end
  def show_pig
    @text.value = @text.value.split.collect{|w| pig(w)}.join(" ")
  end
  def initialize
    ph = { 'padx' => 10, 'pady' => 10 }     # common options
    root = TkRoot.new { title "Pig" }
    top = TkFrame.new(root) { background "white" }
    TkLabel.new(top) {text 'Enter Text:' ; pack(ph) }
    @text = TkVariable.new
    TkEntry.new(top, 'textvariable' =>  @text).pack(ph)
    pig_b = TkButton.new(top) { text 'Pig It'; pack ph}
    pig_b.command { show_pig }
    exit_b = TkButton.new(top) {text 'Exit'; pack ph}
    exit_b.command { exit }
    top.pack('fill'=>'both', 'side' =>'top')
  end
end
PigBox.new
Tk.mainloop
```

Geometry Management

In the example code in this chapter, you'll see references to the widget method `pack`. That's a very important call, as it turns out—leave it off and you'll never see the widget. `pack` is a command that tells the geometry manager to place the widget according to constraints that we specify. Geometry managers recognize three commands.

Command	Placement Specification
`pack`	Flexible, constraint-based placement
`place`	Absolute position
`grid`	Tabular (row/column) position

As `pack` is the most commonly used command, we'll use it in our examples.

Binding Events

Our widgets are exposed to the real world; they get clicked, the mouse moves over them, the user tabs into them; all these things, and more, generate *events* that we can capture. You can create a *binding* from an event on a particular widget to a block of code, using the widget's `bind` method.

For instance, suppose we've created a button widget that displays an image. We'd like the image to change when the user's mouse is over the button.

```
require 'tk'
image1 = TkPhotoImage.new { file "img1.gif" }
image2 = TkPhotoImage.new { file "img2.gif" }
b = TkButton.new(@root) do
  image    image1
  command  { exit }
  pack
end
b.bind("Enter") { b.configure('image'=>image2) }
b.bind("Leave") { b.configure('image'=>image1) }
Tk.mainloop
```

First, we create two GIF image objects from files on disk, using `TkPhotoImage`. Next we create a button (very cleverly named "b"), which displays the image `image1`. We then bind the `Enter` event so that it dynamically changes the image displayed by the button to `image2` when the mouse is over the button, and the `Leave` event to revert back to `image1` when the mouse leaves the button.

This example shows the simple events `Enter` and `Leave`. But the named event given as an argument to `bind` can be composed of several substrings, separated with dashes, in the order *modifier-modifier-type-detail*. Modifiers are listed in the Tk reference and include `Button1`, `Control`, `Alt`, `Shift`, and so on. *Type* is the name of the event (taken from the X11 naming conventions) and includes events such as `ButtonPress`, `KeyPress`, and `Expose`. *Detail* is either a number from 1 to 5 for buttons or a keysym for keyboard input. For instance, a binding that will trigger on mouse release of button 1 while the control key is pressed could be specified as

```
Control-Button1-ButtonRelease
```

or

```
Control-ButtonRelease-1
```

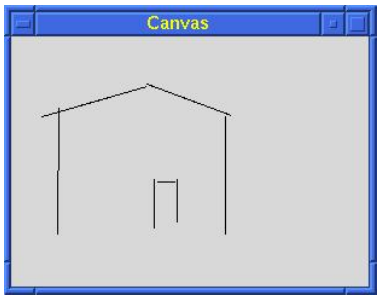
The event itself can contain certain fields such as the time of the event and the x and y positions. `bind` can pass these items to the callback, using *event field codes*. These are used like `printf` specifications. For instance, to get the x and y coordinates on a mouse move, you'd specify the call to `bind` with three parameters. The second parameter is the `Proc` for the callback, and the third parameter is the event field string.

```
canvas.bind("Motion", lambda {|x, y| do_motion (x, y)}, "%x %y")
```

Canvas

Tk provides a `Canvas` widget with which you can draw and produce PostScript output. Figure 19.1 on the next page shows a simple bit of code (adapted from the distribution) that will draw straight lines. Clicking and holding button 1 will start a line, which will be "rubber-banded" as you move the mouse around. When you release button 1, the line will be drawn in that position.

A few mouse clicks, and you've got an instant masterpiece.



As they say, "We couldn't find the artist, so we had to hang the picture…."

Figure 19.1. Drawing on a Tk Canvas

```
require 'tk'

class Draw
  def do_press(x, y)
    @start_x = x
    @start_y = y
    @current_line = TkcLine.new(@canvas, x, y, x, y)
  end

  def do_motion(x, y)
    if @current_line
      @current_line.coords @start_x, @start_y, x, y
    end
  end

  def do_release(x, y)
    if @current_line
      @current_line.coords @start_x, @start_y, x, y
      @current_line.fill 'black'
      @current_line = nil
    end
  end

  def initialize(parent)
    @canvas = TkCanvas.new(parent)
    @canvas.pack
    @start_x = @start_y = 0
    @canvas.bind("1", lambda {|e| do_press(e.x, e.y)})
    @canvas.bind("B1-Motion",
                 lambda {|x, y| do_motion(x, y)}, "%x %y")
    @canvas.bind("ButtonRelease-1",
                 lambda {|x, y| do_release(x, y)},
                 "%x %y")
  end
end

root = TkRoot.new { title 'Canvas' }
Draw.new(root)
Tk.mainloop
```

Scrolling

Unless you plan on drawing very small pictures, the previous example may not be all that useful. `TkCanvas`, `TkListbox`, and `TkText` can be set up to use scrollbars, so you can work on a smaller subset of the "big picture."

Communication between a scrollbar and a widget is bidirectional. Moving the scrollbar means that the widget's view has to change; but when the widget's view is changed by some other means, the scrollbar has to change as well to reflect the new position.

Since we haven't done much with lists yet, our scrolling example will use a scrolling list of text. In the following code fragment, we'll start by creating a plain old `TkListbox` and an associated `TkScrollbar`. The scrollbar's callback (set with `command`) will call the list widget's `yview` method, which will change the value of the visible portion of the list in the y direction.

After that callback is set up, we make the inverse association: when the list feels the need to scroll, we'll set the appropriate range in the scrollbar using `TkScrollbar#set`. We'll use this same fragment in a fully functional program in the next section.

```
list_w = TkListbox.new(frame) do
  selectmode 'single'
  pack 'side' => 'left'
end
list_w.bind("ButtonRelease-1") do
  busy do
    filename = list_w.get(*list_w.curselection)
    tmp_img = TkPhotoImage.new { file filename }
    scale   = tmp_img.height / 100
    scale   = 1 if scale < 1
    image_w.copy(tmp_img, 'subsample' => [scale, scale])
    image_w.pack
  end
end
scroll_bar = TkScrollbar.new(frame) do
  command {|*args| list_w.yview *args }
  pack    'side' => 'left', 'fill' => 'y'
end
list_w.yscrollcommand  {|first,last| scroll_bar.set(first,last) }
```

Just One More Thing

We could go on about Tk for another few hundred pages, but that's another book. The following program is our final Tk example—a simple GIF image viewer. You can select a GIF filename from the scrolling list, and a thumb nail version of the image will be displayed. We'll point out just a *few* more things.

Have you ever used an application that creates a “busy cursor” and then forgets to reset it to normal? A neat trick in Ruby will prevent this from happening. Remember how `File.new` uses a block to ensure that the file is closed after it is used? We can do a similar thing with the method `busy`, as shown in the next example.

This program also demonstrates some simple `TkListbox` manipulations—adding elements to the list, setting up a callback on a mouse button release,[1] and retrieving the current selection.

So far, we’ve used `TkPhotoImage` to display images directly, but you can also zoom, subsample, and show portions of images as well. Here we use the subsample feature to scale down the image for viewing.


```
require 'tk'
class GifViewer
  def initialize(filelist)
    setup_viewer(filelist)
  end
  def run
    Tk.mainloop
  end
  def setup_viewer(filelist)
    @root = TkRoot.new {title 'Scroll List'}
    frame = TkFrame.new(@root)
    image_w = TkPhotoImage.new
    TkLabel.new(frame) do
      image image_w
      pack 'side'=>'right'
    end
    list_w = TkListbox.new(frame) do
      selectmode 'single'
      pack 'side' => 'left'
    end
    list_w.bind("ButtonRelease-1") do
      busy do
        filename = list_w.get(*list_w.curselection)
        tmp_img = TkPhotoImage.new { file filename }
        scale   = tmp_img.height / 100
        scale   = 1 if scale < 1
        image_w.copy(tmp_img, 'subsample' => [scale, scale])
        image_w.pack
      end
    end
```

1. You probably want the button release, not the press, as the widget gets selected on the button press.

```
    filelist.each do |name|
      list_w.insert('end', name) # Insert each file name into the list
    end
    scroll_bar = TkScrollbar.new(frame) do
      command {|*args| list_w.yview *args }
      pack    'side' => 'left', 'fill' => 'y'
    end
    list_w.yscrollcommand  {|first,last| scroll_bar.set(first,last) }
    frame.pack
  end
  # Run a block with a 'wait' cursor
  def busy
    @root.cursor "watch" # Set a watch cursor
    yield
  ensure
    @root.cursor "" # Back to original
end
end
viewer = GifViewer.new(Dir["screenshots/gifs/*.gif"])
viewer.run
```

Translating from Perl/Tk Documentation

That's it, you're on your own now. For the most part, you can easily translate the documentation given for Perl/Tk to Ruby. There are a few exceptions; some methods are not implemented, and some extra functionality is undocumented. Until a Ruby/Tk book comes out, your best bet is to ask on the newsgroup or read the source code.

But in general, it's pretty easy to see what's happening. Remember that options may be given as a hash, or in code block style, and the scope of the code block is within the `TkWidget` being used, not your class instance.

Object Creation

In the Perl/Tk mapping, parents are responsible for creating their child widgets. In Ruby, the parent is passed as the first parameter to the widget's constructor.

```
Perl/Tk:  $widget = $parent->Widget( [ option => value ] )
Ruby:     widget = TkWidget.new(parent, option-hash)
          widget = TkWidget.new(parent) { code block }
```

You may not need to save the returned value of the newly created widget, but it's there if you do. Don't forget to pack a widget (or use one of the other geometry calls), or it won't be displayed.

Options

```
Perl/Tk:  -background => color
Ruby:     'background' => color
          { background color }
```

Remember that the code block scope is different.

Variable References

```
Perl/Tk:  -textvariable => \$variable
          -textvariable => varRef
Ruby:     ref = TkVariable.new
          'textvariable' => ref
          { textvariable ref }
```

Use `TkVariable` to attach a Ruby variable to a widget's value. You can then use the `value` accessors in `TkVariable` (`TkVariable#value` and `TkVariable#value=`) to affect the contents of the widget directly.