UNIT - 1

Ruby:

Ruby is a pure object-oriented programming language. It was created in 1993 by Yukihiro Matsumoto of Japan.

You can find the name Yukihiro Matsumoto on the Ruby mailing list at www.ruby-lang.org. Matsumoto is also known as Matz in the Ruby community.

Ruby is "A Programmer's Best Friend".

Ruby has features that are similar to those of Smalltalk, Perl, and Python. Perl, Python, and Smalltalk are scripting languages. Smalltalk is a true object-oriented language. Ruby, like Smalltalk, is a perfect object-oriented language. Using Ruby syntax is much easier than using Smalltalk syntax.

. Ruby can be used to write Common Gateway Interface (CGI) scripts. Ruby can be embedded into Hypertext Markup Language (HTML).

Features of Ruby

- Ruby is an open-source and is freely available on the Web, but it is subject to a license.
- Ruby is a general-purpose, interpreted programming language.
- Ruby is a true object-oriented programming language.
- Ruby is a server-side scripting language similar to Python and PERL.
- Ruby can be used to write Common Gateway Interface (CGI) scripts.
- Ruby can be embedded into Hypertext Markup Language (HTML).
- Ruby has a clean and easy syntax that allows a new developer to learn very quickly and easily.
- Ruby has similar syntax to that of many programming languages such as C++ and Perl.
- Ruby is very much scalable and big programs written in Ruby are easily maintainable.
- Ruby can be used for developing Internet and intranet applications.
- Ruby can be installed in Windows and POSIX environments.

- Ruby support many GUI tools such as Tcl/Tk, GTK, and OpenGL.
- Ruby can easily be connected to DB2, MySQL, Oracle, and Sybase.
- Ruby has a rich set of built-in functions, which can be used directly into Ruby scripts.

Tools You Will Need:

For performing the examples discussed in this tutorial, you will need a latest computer like Intel Core i3 or i5 with a minimum of 2GB of RAM (4GB of RAM recommended). You also will need the following software −

- Linux or Windows 95/98/2000/NT or Windows 7 operating system.
- Apache 1.3.19-5 Web server.
- Internet Explorer 5.0 or above Web browser.
- Ruby 1.8.5

Rails:

Rails, is a server-side web application framework written in Ruby under the MIT License. Rails is a model–view–controller (MVC) framework, providing default structures for a database, a web service, and web pages.

Before we ride on Rails, let us recapitulate a few points of Ruby, which is the base of Rails.

Ruby is the successful combination of

- Smalltalk's conceptual elegance,
- Python's ease of use and learning, and
- Perl's pragmatism.

Ruby is

- A high-level programming language.
- Interpreted like Perl, Python, Tcl/TK.
- Object-oriented like Smalltalk, Eiffel, Ada, Java.

Why Ruby?

Ruby originated in Japan and now it is gaining popularity in US and Europe as well. The following factors contribute towards its popularity

- Easy to learn
- Open source (very liberal license)
- Rich libraries
- Very easy to extend
- Truly object-oriented
- Less coding with fewer bugs
- Helpful community

What is Rails?

- An extremely productive web-application framework.
- Written in Ruby by David Heinemeier Hansson.
- You could develop a web application at least ten times faster with Rails than you could with a typical Java framework.
- An open source Ruby framework for developing database-backed web applications.
- Configure your code with Database Schema.
- No compilation phase required.

Full Stack Framework

- Includes everything needed to create a database-driven web application, using the Model-View-Controller pattern.
- Being a full-stack framework means all the layers are built to work seamlessly together with less code.
- Requires fewer lines of code than other frameworks.

Convention over Configuration

- Rails shuns configuration files in favor of conventions, reflection, and dynamic runtime extensions.
- Your application code and your running database already contain everything that Rails needs to know!

Rails Strengths

Rails is packed with features that make you more productive, with many of the following features building on one other.

Meta programming

Where other frameworks use extensive code generation from scratch, Rail framework uses Meta programming techniques to write programs. Ruby is one of the best languages for Meta programming, and Rails uses this capability well. Rails also uses code generation but relies much more on Meta programming for the heavy lifting.

Active Record

Rails introduces the Active Record framework, which saves objects into the database. The Rails version of the Active Record discovers the columns in a database schema and automatically attaches them to your domain objects using metaprogramming.

Convention over configuration

Most web development frameworks for .NET or Java force you to write pages of configuration code. If you follow the suggested naming conventions, Rails doesn't need much configuration.

Scaffolding

You often create temporary code in the early stages of development to help get an application up quickly and see how major components work together. Rails automatically create much of the scaffolding you'll need.

Built-in testing

Rails create simple automated tests you can then extend. Rail also provides supporting code called harnesses and fixtures that make test cases easier to write and run. Ruby can then execute all your automated tests with the rake utility.

Three environments

Rails gives you three default environments: development, testing, and production. Each behaves slightly differently, making your entire software development cycle easier. For example, Rails creates a fresh copy of the Test database for each test run.

The Structure and Execution of Ruby:

Beginning with Ruby programming:

1. Finding a Compiler:
Before starting programming in Ruby, a compiler is needed to compile and run our programs. There are many online compilers that can be used to start Ruby without installing a compiler:

https://www.jdoodle.com/execute-ruby-online
https://repl.it/

There are many compilers available freely for compilation of Ruby programs.

2. Programming in Ruby:
To program in Ruby is easy to learn because of its similar syntax to already widely used languages.

Writing program in Ruby:
Programs can be written in Ruby in any of the widely used **text editors** like **Notepad++, gedit** etc. After writing the programs save the file with the extension **.rb**

Let's see some basic points of programming:

Comments: To add single line comments in Ruby Program, # (hash) is used.

Syntax:
Comment

To add multi-line comments in Ruby, a block of =begin and =end (reserved key words of Ruby) are used.

Syntax:
=begin

Statement 1

Statement 2

...

Statement n

=end

Example:
A simple program to print **"Hello CSE!! Welcome to Scripting Languages"**

```
# Sample Ruby Program
Puts 'Hello CSE!! Welcome to Scripting Languages'
```

Save and name the above programs as ***testruby.rb***

Compile & run : ruby testruby.rb

Output: Hello CSE!! Welcome to Scripting Languages

Explanation: First line consists of single line comment with **"#"** as prefix. Second line consists of the message to printed and **puts** is used to print the message on the screen.
As everything has some advantages and disadvantages, Ruby also has some advantages along with some disadvantages.

Advantages of Ruby:

- The code written in Ruby is small, elegant and powerful as it has fewer numbers of lines of code.
- Ruby allows simple and fast creation of Web application which results in less hard work.
- As Ruby is free of charge that is Ruby is free to copy, use, modify, it allow programmers to make necessary changes as and when required.
- Ruby is a dynamic programming language due to which there is no tough rules on how to built in features and it is very close to spoken languages.

Disadvantages of Ruby:

- Ruby is fairly new and has its own unique coding language which makes it difficult for the programmers to code in it right away but after some practice its easy to use. Many programmers prefer to stick to what they already know and can develop.
- The code written in Ruby is harder to debug, since most of the time it generates at runtime, so it becomes difficult to read while debugging.
- Ruby does not have a plenty of informational resources as compared to other programming languages.

- Ruby is an interpreted scripting language, the scripting languages are usually slower than compiled languages therefore, Ruby is slower than many other languages.

Applications:

- Ruby is used to create web applications of different sorts. It is one of the hot technology at present to create web applications.

Ruby offers a great feature called Ruby on Rails (RoR). It is a web framework that is used by programmers to speed up the development process and save time.

Package Management with Rubygems:

RubyGems is a package utility for Ruby, which installs Ruby software packages and keeps them up-to-date.

Usage Syntax

$ gem command [arguments...] [options...]

Example

Check to see whether RubyGems is installed

$gem –version

0.9.0

RubyGems Commands

Here is a list of all important commands for Ruby Gems

Sr.No.	Command & Description
1	**Build** Builds a gem from a gemspec.
2	**Cert** Adjusts RubyGems certificate settings.

3	**Check** Checks installed gems.
4	**Cleanup** Cleans up old versions of installed gems in the local repository.
5	**Contents** Displays the contents of the installed gems.
6	**Dependency** Shows the dependencies of an installed gem.
7	**Environment** Displays RubyGems environmental information.
8	**Help** Provides help on the 'gem' command.
9	**Install** Installs a gem into the local repository.
10	**List** Displays all gems whose name starts with STRING.
11	**Query** Queries gem information in local or remote repositories.
12	**Rdoc**

	Generates RDoc for pre-installed gems.
13	**Search** Displays all gems whose name contains STRING.
14	**Specification** Displays gem specification (in yaml).
15	**Uninstall** Uninstalls a gem from the local repository.
16	**Unpack** Unpacks an installed gem to the current directory.
17	**Update** Updates the named gem (or all installed gems) in the local repository.

RubyGems Common Command Options

Following is the list of common options

Sr.No.	Command & Description
1	**--source URL** Uses URL as the remote source for gems.
2	**-p, --[no-]http-proxy [URL]** Uses HTTP proxy for remote operations.

3	**-h, --help** Gets help on this command.
4	**--config-file FILE** Uses this config file instead of default.
5	**--backtrace** Shows stack backtrace on errors.
6	**--debug** Turns on Ruby debugging.

Ruby Gems Install Command Options

This is a list of the options, which use most of the time when you use RubyGems while installing any Ruby package

Sr.No.	Command & Description
1	**-v, --version VERSION** Specifies version of gem to install.
2	**-l, --local** Restricts operations to the LOCAL domain (default).
3	**-r, --remote** Restricts operations to the REMOTE domain.
4	**-b, --both**

	Allows LOCAL and REMOTE operations.
5	**-i, --install-dir DIR** Where to install.
6	**-d, --[no-]rdoc** Generates RDoc documentation for the gem on install.
7	**-f, --[no-]force** Forces gem to install, bypassing dependency checks.
8	**-t, --[no-]test** Runs unit tests prior to installation.
9	**-w, --[no-]wrappers** Uses bin wrappers for executables.
10	**-P, --trust-policy POLICY** Specifies gem trust policy.
11	**--ignore-dependencies** Do not install any required dependent gems.
12	**-y, --include-dependencies** Unconditionally installs the required dependent gems.

Examples

This will install 'SOAP4R', either from local directory or remote server including all the dependencies −

gem install soap4r --include-dependencies

This will install 'rake', only from remote server −

gem install rake --remote

This will install 'rake' from remote server, and run unit tests, and generate RDocs −

gem install --remote rake --test --rdoc --ri

eruby packages:

eRuby stands for *embedded Ruby*. It's a tool that embeds fragments of Ruby code in other files such as HTML files similar to ASP, JSP and PHP.

eRuby allows Ruby code to be embedded within (delimited by) a pair of <% and %> delimiters. These embedded code blocks are then evaluated in-place, i.e., they are replaced by the result of their evaluation.

Syntax

Here is a syntax to write single line of *eRuby* code −

<% ruby code %>

They function like blocks in Ruby and are terminated by <% end %>.

```
<ul>
<% 3.times do %>

  <li>list item</li>

<% end %>
</ul>
```

All Ruby code after the # is ignored and treated as comments.

```
<%# ruby code %>
```

Example

Here's a sample eRuby file −

```
This is sample eRuby file<br>
The current time here is <%=Time.now%>.
<%[1,2,3].each{|x|print x,"<br>\n"}%>
```

Here's the output from this sample file −

This is sample eRuby file
The current time here is Wed Aug 29 18:54:45 JST 2001.
1
2
3

CGI Programming:

CGI or Common Gateway Interface was created to server content over HTTP web servers using external scripting languages like Perl, Python, Ruby, or compiled binaries of C,C++, etc. Apache, the most popular web server and also others can be easily configured to run CGI scripts.

In this article we'll be looking at configuring Apache to run CGI scripts and writing simple CGI scripts in Ruby.

Configuring Apache

In the demo example I'll be using a directory rb-bin you may change this to whatever you like, cgi-bin is very popular and generally pre-configured. Do not forget to make all your scripts in the directory executable.

Add the following to Apache's config file, and then restart Apache.

Code:

```
ScriptAlias /rb-bin/

<Directory "/var/www/rb-bin">
   AllowOverride None
   Options ExecCGI
   Order allow,deny
   Allow from all
</Directory>
```

Basics of CGI Scripts

Unlike PHP, with Ruby you'll need to send your own headers. Headers and content of a HTTP response is separated by 2 newlines. Let's see how to send headers, let's write our first Ruby CGI script.

Code:

```
#!/usr/bin/ruby

print "Content-Type: text/html\n\n"
print "<h2>Hello World!</h2>"
```

Handling Form Data

For handling forms we'll need a Ruby library **cgi**, fortunately it comes bundled with Ruby. Here's how to read form values.

Code:

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new
name = cgi['Name']
puts cgi.header("type" => "text/html", "cache_control" => "no-cache, no-store")
puts "<h2>Hello #{name}</h2>"
```

In the above example we get the name & print it out.

Using Cookies

Cookies are accessed and created using the cgi library, so it's pretty simple. Let's try it with some code.

Code:

```
#!/usr/bin/ruby

require 'cgi'

cgi = CGI.new
```

```
## setting cookies
cookie1 = CGI::Cookie.new('name' => 'name','value' => 'Pradeep','expires' =>
Time.now + 3600)
cookie2 = CGI::Cookie.new('name' => 'city','value' => 'Kolkata','expires' =>
Time.now + 3600)

print cgi.header("cookie" => [cookie1,cookie2])
```

Reading cookies:

Code:

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new
print cgi.header()
name = cgi.cookies['name']
print "Name is #{name}"
```

Cookies:

How It Works?

Your server sends some data to the visitor's browser in the form of a cookie. The browser may accept the cookie. If it does, it is stored as a plain text record on the visitor's hard drive. Now, when the visitor arrives at another page on your site, the cookie is available for retrieval. Once retrieved, your server knows/remembers what was stored.

Cookies are a plain text data record of five variable-length fields −

- **Expires** − The date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.
- **Domain** − The domain name of your site.
- **Path** − The path to the directory or web page that sets the cookie. This may be blank if you want to retrieve the cookie from any directory or page.

- **Secure** − If this field contains the word "secure", then the cookie may only be retrieved with a secure server. If this field is blank, no such restriction exists.
- **Name = Value** − Cookies are set and retrieved in the form of key and value pairs.

Handling Cookies in Ruby

You can create a named cookie object and store any textual information in it. To send it down to the browser, set a **cookie** header in the call to *CGI.out*.

```
#!/usr/bin/ruby

require "cgi"
cgi = CGI.new("html4")
cookie = CGI::Cookie.new('name' => 'mycookie', 'value' => 'Zara Ali', 'expires' =>
Time.now + 3600)
cgi.out('cookie' => cookie) do
  cgi.head + cgi.body { "Cookie stored" }
end
```

The next time the user comes back to this page, you can retrieve the cookie values set as shown below −

```
#!/usr/bin/ruby

require "cgi"
cgi = CGI.new("html4")
cookie = cgi.cookies['mycookie']
cgi.out('cookie' => cookie) do
  cgi.head + cgi.body { cookie[0] }
end
```

Cookies are represented using a separate object of class CGI::Cookie, containing the following accessors

Attribute	Returned Value

name	Cookie name
value	An array of cookie values
path	The cookie's path
domain	The domain
expires	The expiration time (as a Time object)
secure	True if secure cookie

Choice of Webservers:

When you deploy a Ruby application without a Procfile a default webserver will be used. For Rack this means `$ bundle exec rackup` is run for Rails `$ rails server`. Depending on the version of these libraries you are using and what gems you have in your Gemfile, the WEBrick server may be used to run your production application.
Even if your application does not use WEBrick, it is HIGHLY recommended you do not rely on this implicit behavior and instead explicitly declare how you want your webserver started via a `Procfile`. If you don't have a preference we recommend configuring your application to run on Puma. When selecting a webserver make sure that it can handle concurrent requests.
The rest of this article contains information about WEBrick and why it is not a good idea to run it in Production.

What is WEBrick

The Ruby standard library comes with a default web server named WEBrick. As this library is installed on every machine that has Ruby, most frameworks such as Rails and Rack use WEBrick as a default development web server.

While WEBrick should be fine for development, it was not designed to handle a high concurrent workload that a Ruby app must serve in production. A production web server should be used instead.

Why not WEBrick

WEBrick will spawn a new thread for each request, this behavior can be fine for some applications but due to the GVL this means that WEBrick cannot make use of multiple cores. If you need maximum performance Heroku recommends a multi-process multi-threaded server, we currently recommend Puma.
If you do not specify a web server in your Procfile, it is likely that you're running WEBrick in production. If you continue to run WEBrick it is likely that requests will take a long time, possibly timeout, and you will need to use many more dynos than your application requires.
Rather than doing this, ensure you use a production web server.

Production web server

A production Ruby web server is capable of handling multiple request concurrently. We currently recommend using the Puma web server.
Please add it as a dependency to your Gemfile, add a Procfile that specifies your production web server, commit to git and deploy.

SOAP and Web services:

SOAP stands for Simple Object Access Protocol. It is a XML-based protocol for accessing web services.

SOAP is a W3C recommendation for communication between two applications.

SOAP is XML based protocol. It is platform independent and language independent. By using SOAP, you will be able to interact with other programming language applications.

Advantages of Soap Web Services

WS Security: SOAP defines its own security known as WS Security.

Language and Platform independent: SOAP web services can be written in any programming language and executed in any platform.

Disadvantages of Soap Web Services

Slow: SOAP uses XML format that must be parsed to be read. It defines many standards that must be followed while developing the SOAP applications. So it is slow and consumes more bandwidth and resource.

WSDL dependent: SOAP uses WSDL and doesn't have any other mechanism to discover the service.

Simple TK Application:
A simple Tk application in Ruby might look something like this:

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

Let's look at the code a little more closely. After loading in the tk extension module, we create a root-level frame using TkRoot.new. We then make a label 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!' }
Tk.mainloop
```

Widgets:

A widget is **an element of a graphical user interface (GUI) that displays information or provides a specific way for a user to interact with the operating system or an application**.

Standard Configuration Options

All widgets have a number of different configuration options, which generally control how they are displayed or how they behave. The options that are available depend upon the widget class of course.

Here is a list of all the standard configuration options, which could be applicable to any Ruby/Tk widget.

There are other widget specific options also, which would be explained along with widgets.

Ruby/Tk Widget Classes

There is a list of various Ruby/Tk classes, which can be used to create a desired GUI using Ruby/Tk.

- TkFrame Creates and manipulates frame widgets.
- TkButton Creates and manipulates button widgets.
- TkLabel Creates and manipulates label widgets.
- TkEntry Creates and manipulates entry widgets.
- TkCheckButton Creates and manipulates checkbutton widgets.
- TkRadioButton Creates and manipulates radiobutton widgets.
- TkListbox Creates and manipulates listbox widgets.
- TkComboBox Creates and manipulates listbox widgets.
- TkMenu Creates and manipulates menu widgets.
- TkMenubutton Creates and manipulates menubutton widgets.
- Tk.messageBox Creates and manipulates a message dialog.
- TkScrollbar Creates and manipulates scrollbar widgets.
- TkCanvas Creates and manipulates canvas widgets.
- TkScale Creates and manipulates scale widgets.

- TkText Creates and manipulates text widgets.
- TkToplevel Creates and manipulates toplevel widgets.
- TkSpinbox Creates and manipulates Spinbox widgets.
- TkProgressBar Creates and manipulates Progress Bar widgets.
- Dialog Box Creates and manipulates Dialog Box widgets.
- Tk::Tile::Notebook Display several windows in limited space with notebook metaphor.
- Tk::Tile::Paned Displays a number of subwindows, stacked either vertically or horizontally.
- Tk::Tile::Separator Displays a horizontal or vertical separator bar.
- Ruby/Tk Font, Colors and Images Understanding Ruby/Tk Fonts, Colors and Images

Ruby/Tk Geometry Management

Geometry Management deals with positioning different widgets as per requirement. Geometry management in Tk relies on the concept of master and slave widgets.

A master is a widget, typically a top-level window or a frame, which will contain other widgets, which are called slaves. You can think of a geometry manager as taking control of the master widget, and deciding what will be displayed within.

The geometry manager will ask each slave widget for its natural size, or how large it would ideally like to be displayed. It then takes that information and combines it with any parameters provided by the program when it asks the geometry manager to manage that particular slave widget.

There are three geometry managers *place*, *grid* and *pack* that are responsible for controlling the size and location of each of the widgets in the interface.

- grid Geometry manager that arranges widgets in a grid.
- pack Geometry manager that packs around edges of cavity.
- place Geometry manager for fixed or rubber-sheet placement.

Ruby/Tk Event Handling

Ruby/Tk supports *event loop*, which receives events from the operating system. These are things like button presses, keystrokes, mouse movement, window resizing, and so on.

Ruby/Tk takes care of managing this event loop for you. It will figure out what widget the event applies to (did the user click on this button? if a key was pressed, which textbox had the focus?), and dispatch it accordingly. Individual widgets know how to respond to events, so for example a button might change color when the mouse moves over it, and revert back when the mouse leaves.

At a higher level, Ruby/Tk invokes callbacks in your program to indicate that something significant happened to a widget For either case, you can provide a code block or a *Ruby Proc* object that specifies how the application responds to the event or callback.

Let's take a look at how to use the bind method to associate basic window system events with the Ruby procedures that handle them. The simplest form of bind takes as its inputs a string indicating the event name and a code block that Tk uses to handle the event.

For example, to catch the *ButtonRelease* event for the first mouse button on some widget, you'd write −

```
someWidget.bind('ButtonRelease-1') {
  ....code block to handle this event...
}
```

An event name can include additional modifiers and details. A modifier is a string like *Shift*, *Control* or *Alt*, indicating that one of the modifier keys was pressed.

So, for example, to catch the event that's generated when the user holds down the *Ctrl* key and clicks the right mouse button.

```
someWidget.bind('Control-ButtonPress-3', proc { puts "Ouch!" })
```

Many Ruby/Tk widgets can trigger *callbacks* when the user activates them, and you can use the *command* callback to specify that a certain code block or procedure is invoked when that happens. As seen earlier, you can specify the command callback procedure when you create the widget −

```
helpButton = TkButton.new(buttonFrame) {
  text "Help"
  command proc { showHelp }
```

```
}
```

Or you can assign it later, using the widget's *command* method −

```
helpButton.command proc { showHelp }
```

Since the command method accepts either procedures or code blocks, you could also write the previous code example as −

```
helpButton = TkButton.new(buttonFrame) {
  text "Help"
  command { showHelp }
}
```

You can use the following basic event types in your Ruby/Tk application −

The configure Method

The *configure* method can be used to set and retrieve any widget configuration values. For example, to change the width of a button you can call configure method any time as follows −

```
require "tk"

button = TkButton.new {
  text 'Hello World!'
  pack
}
button.configure('activebackground', 'blue')
Tk.mainloop
```

To get the value for a current widget, just supply it without a value as follows −

```
color = button.configure('activebackground')
```

You can also call configure without any options at all, which will give you a listing of all options and their values.

The cget Method

For simply retrieving the value of an option, configure returns more information than you generally want. The cget method returns just the current value.

```
color = button.cget('activebackground')
```

Description

A *frame* is a widget that displays just as a simple rectangle. Frames are primarily used as a container for other widgets, which are under the control of a geometry manager such as grid.

The only features of a frame are its background color and an optional 3-D border to make the frame appear raised or sunken.

Syntax

Here is a simple syntax to create a Frame Widget

```
TkFrame.new {
  .....Standard Options....
  .....Widget-specific Options....
}
```

Standard Options

- borderwidth
- highlightbackground
- highlightthickness
- takefocus
- highlightcolor
- relief
- cursor

These options have been described in the previous chapter.

Widget Specific Options

Sr.No.	Options & Description
1	**background** => String This option is the same as the standard **background** option except that its value may also be specified as an undefined value. In this case, the widget will display no background or border, and no colors will be consumed from its colormap for its background and border.

2	**colormap** => String Specifies a colormap to use for the window. The value may be either *new*, in which case a new colormap is created for the window and its children, or the name of another window (which must be on the same screen), in which case the new window will use the colormap from the specified window. If the **colormap** option is not specified, the new window uses the same colormap as its parent.
3	**container** => Boolean The value must be a boolean. If true, it means that this window will be used as a container in which some other application will be embedded. The window will support the appropriate window manager protocols for things like geometry requests. The window should not have any children of its own in this application.
4	**height** => Integer Specifies the desired height for the window in pixels or points.
5	**width** => Integer Specifies the desired width for the window in pixels or points.

Event Bindings

When a new frame is created, it has no default event bindings: frames are not intended to be interactive.

Examples

```
require "tk"

f1 = TkFrame.new {
  relief 'sunken'
  borderwidth 3
  background "red"
  padx 15
  pady 20
```

```
  pack('side' => 'left')
}
f2 = TkFrame.new {
  relief 'groove'
  borderwidth 1
  background "yellow"
  padx 10
  pady 10
  pack('side' => 'right')
}

TkButton.new(f1) {
  text 'Button1'
  command {print "push button1!!\n"}
  pack('fill' => 'x')
}
TkButton.new(f1) {
  text 'Button2'
  command {print "push button2!!\n"}
  pack('fill' => 'x')
}
TkButton.new(f2) {
  text 'Quit'
  command 'exit'
  pack('fill' => 'x')
}
Tk.mainloop
```

This will produce the following result –

Binding Events:

Our widgets are exposed to the real world; they get clicked on, 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.

```
image1 = TkPhotoImage.new { file "img1.gif" }
image2 = TkPhotoImage.new { file "img2.gif" }

b = TkButton.new(@root) {
  image    image1
  command  proc { doit }
}

b.bind("Enter") { b.configure('image'=>image2) }
b.bind("Leave") { b.configure('image'=>image1) }
```

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, and the ``Leave'' event to revert back to image1.

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", proc{|x, y| do_motion (x, y)}, "%x %y")
```

Canvas:

A **Canvas** widget implements structured graphics. A canvas displays any number of items, which may be things like rectangles, circles, lines, and text.

Items may be manipulated (e.g., moved or re-colored) and callbacks may be associated with items in much the same way that the bind method allows callbacks to be bound to widgets.

Syntax

Here is a simple syntax to create this widget −

```
TkCanvas.new {
   .....Standard Options....
   .....Widget-specific Options....
}
```

Standard Options

- background
- borderwidth
- cursor
- highlightbackground
- highlightcolor
- highlightthickness
- relief
- selectbackground
- selectborderwidth
- selectforeground
- state

- takefocus
- tile
- xscrollcommand
- yscrollcommand

These options have been described in the previous chapter.

Widget Specific Options

Sr.No.	Options & Description
1	**closeenough** =>Integer Specifies a floating-point value indicating how close the mouse cursor must be to an item before it is considered to be **inside** the item. Defaults to 1.0.
2	**confine** =>Boolean Specifies a boolean value that indicates whether or not it should be allowable to set the canvas's view outside the region defined by the *scrollregion* argument. Defaults to true, which means that the view will be constrained within the scroll region.
3	**height** =>Integer Specifies a desired window height that the canvas widget should request from its geometry manager.
4	**scrollregion** =>Coordinates Specifies a list with four coordinates describing the left, top, right, and bottom coordinates of a rectangular region. This region is used for scrolling purposes and is considered to be the boundary of the information in the canvas.
5	**state** =>String

	Modifies the default state of the canvas where state may be set to one of: **normal, disabled,** or **hidden**. Individual canvas objects all have their own state option, which overrides the default state.
6	**width** =>Integer Specifies a desired window width that the canvas widget should request from its geometry manager.
7	**xscrollincrement** =>Integer Specifies an increment for horizontal scrolling, in any of the usual forms permitted for screen distances. If the value of this option is greater than zero, the horizontal view in the window will be constrained so that the canvas x coordinate at the left edge of the window is always an even multiple of xscrollincrement; furthermore, the units for scrolling will also be xscrollincrement.
8	**yscrollincrement** =>Integer Specifies an increment for vertical scrolling, in any of the usual forms permitted for screen distances. If the value of this option is greater than zero, the vertical view in the window will be constrained so that the canvas y coordinate at the top edge of the window is always an even multiple of yscrollincrement; furthermore, the units for scrolling will also be yscrollincrement.

Indices

Indices are used for methods such as inserting text, deleting a range of characters, and setting the insertion cursor position. An index may be specified in any of a number of ways, and different types of items may support different forms for specifying indices.

Text items support the following forms for an index −

- **number** − A decimal number giving the position of the desired character within the text item. 0 refers to the first character, 1 to the next character, and so on.

- **end** − Refers to the character or coordinate just after the last one in the item (same as the number of characters or coordinates in the item).
- **insert** − Refers to the character just before which the insertion cursor is drawn in this item. Not valid for lines and polygons.

Creating Items:

When you create a new canvas widget, it will essentially be a large rectangle with nothing on it; truly a blank canvas in other words. To do anything useful with it, you'll need to add items to it.

There are a wide variety of different types of items you can add. Following methods will be used to create different items inside a canvas −

Arc Items

Items of type arc appear on the display as arc-shaped regions. An arc is a section of an oval delimited by two angles. Arcs are created with methods of the following form −

The **TkcArc.new(canvas, x1, y1, x2, y2, ?option, value, option, value, ...?)** method will be used to create an arc.

The arguments x1, y1, x2, and y2 give the coordinates of two diagonally opposite corners of a rectangular region enclosing the oval that defines the arc. Here is the description of other options −

- **extent => degrees** − Specifies the size of the angular range occupied by the arc. If it is greater than 360 or less than -360, then degrees modulo 360 is used as the extent.
- **fill => color** − Fills the region of the arc with color.
- **outline => color** − Color specifies a color to use for drawing the arc's outline.
- **start => degrees** − Specifies the beginning of the angular range occupied by the arc.
- **style => type** − Specifies how to draw the arc. If *type* is **pieslice** (the default) then the arc's region is defined by a section of the oval's perimeter plus two line segments, one between the center of the oval and each end of the perimeter section. If *type* is **chord** then the arc's region is defined by a section of the oval's perimeter plus a single line segment connecting the two end points of the perimeter section. If *type* is **arc** then the arc's region consists of a section of the perimeter alone.

- **tags => tagList** − Specifies a set of tags to apply to the item. TagList consists of a list of tag names, which replace any existing tags for the item. TagList may be an empty list.
- **width => outlineWidth** − Specifies the width of the outline to be drawn around the arc's region.

Bitmap Items

Items of type bitmap appear on the display as images with two colors, foreground and background. Bitmaps are created with methods of the following form −

The **TkcBitmap.new(canvas, x, y, ?option, value, option, value, ...?)** method will be used to create a bitmap.

The arguments x and y specify the coordinates of a point used to position the bitmap on the display. Here is the description of other options −

- **anchor => anchorPos** − AnchorPos tells how to position the bitmap relative to the positioning point for the item. For example, if anchorPos is center then the bitmap is centered on the point; if anchorPos is n then the bitmap will be drawn so that its top center point is at the positioning point. This option defaults to center.
- **background => color** − Specifies a color to use for each of the bitmap pixels whose value is 0.
- **bitmap => bitmap** − Specifies the bitmap to display in the item.
- **foreground => color** − Specifies a color to use for each of the bitmap pixels whose value is 1.
- **tags => tagList** − Specifies a set of tags to apply to the item. TagList consists of a list of tag names, which replace any existing tags for the item. TagList may be an empty list.

Image Items

Items of type image are used to display images on a canvas. Images are created with methods of the following form: :

The **TkcImage.new(canvas,x, y, ?option, value, option, value, ...?)** method will be used to create an image.

The arguments x and y specify the coordinates of a point used to position the image on the display. Here is the description of other options −

- **anchor => anchorPos** − AnchorPos tells how to position the bitmap relative to the positioning point for the item. For example, if anchorPos is center then the bitmap is centered on the point; if anchorPos is n then the bitmap will be drawn so that its top center point is at the positioning point. This option defaults to center.
- **image => name** − Specifies the name of the image to display in the item. This image must have been created previously with the image create command.
- **tags => tagList** − Specifies a set of tags to apply to the item. TagList consists of a list of tag names, which replace any existing tags for the item. TagList may be an empty list.

Line Items

Items of type line appear on the display as one or more connected line segments or curves. Lines are created with methods of the following form −

The **TkcLine.new(canvas, x1, y1..., xn, yn, ?option, value, ...?)** method will be used to create a line.

The arguments x1 through yn give the coordinates for a series of two or more points that describe a series of connected line segments. Here is the description of other options −

- **arrow => where** − Indicates whether or not arrowheads are to be drawn at one or both ends of the line. *Where* must have one of the values **none** (for no arrowheads), **first** (for an arrowhead at the first point of the line), **last** (for an arrowhead at the last point of the line), or **both** (for arrowheads at both ends). This option defaults to **none**.
- **arrowshape => shape** − This option indicates how to draw arrowheads. If this option isn't specified then Tk picks a reasonable shape.
- **dash => pattern** − Specifies a pattern to draw the line.
- **capstyle => style** − Specifies the ways in which caps are to be drawn at the endpoints of the line. Possible values are butt, projecting, or round.
- **fill => color** − Color specifies a color to use for drawing the line.
- **joinstyle => style** − Specifies the ways in which joints are to be drawn at the vertices of the line. Possible values are bevel, miter, or round.
- **smooth => boolean** − It indicates whether or not the line should be drawn as a curve.

- **splinesteps => number** − Specifies the degree of smoothness desired for curves: each spline will be approximated with number line segments. This option is ignored unless the *smooth* option is true.
- **stipple => bitmap** − Indicates that the line should be filled in a stipple pattern; bitmap specifies the stipple pattern to use.
- **tags => tagList** − Specifies a set of tags to apply to the item. TagList consists of a list of tag names, which replace any existing tags for the item. TagList may be an empty list.
- **width => lineWidth** − Specifies the width of the line.

Rectangle Items

Items of type rectangle appear as rectangular regions on the display. Each rectangle may have an outline, a fill, or both. Rectangles are created with methods of the following form

The **TkcRectangle.new(canvas, x1, y1, x2, y2, ?option, value,...?)** method will be used to create a Rectangle.

The arguments x1, y1, x2, and y2 give the coordinates of two diagonally opposite corners of the rectangle. Here is the description of other options −

- **fill => color** − Fills the area of the rectangle with color.
- **outline => color** − Draws an outline around the edge of the rectangle in color.
- **stipple => bitmap** − Indicates that the rectangle should be filled in a stipple pattern; bitmap specifies the stipple pattern to use.
- **tags => tagList** − Specifies a set of tags to apply to the item. TagList consists of a list of tag names, which replace any existing tags for the item. TagList may be an empty list.
- **width => outlineWidth** − Specifies the width of the outline to be drawn around the rectangle.

Event Bindings

Canvas has the default bindings to allow scrolling if necessary: <Up>, <Down>, <Left> and <Right> (and their <Control-*> counter parts). Further <Prior>,

<Next>, <Home> and <End>. These bindings allow you to navigate the same way as in other widgets that can scroll.

Example 1

```
require "tk"

canvas = TkCanvas.new
TkcRectangle.new(canvas, '1c', '2c', '3c', '3c', 'outline' => 'black', 'fill' => 'blue')
TkcLine.new(canvas, 0, 0, 100, 100, 'width' => '2', 'fill' => 'red')
canvas.pack

Tk.mainloop
```

This will produce the following result −

Example 2

```
require 'tk'

root = TkRoot.new
root.title = "Window"

canvas = TkCanvas.new(root) do
  place('height' => 170, 'width' => 100, 'x' => 10, 'y' => 10)
end

TkcLine.new(canvas, 0, 5, 100, 5)
TkcLine.new(canvas, 0, 15, 100, 15, 'width' => 2)
TkcLine.new(canvas, 0, 25, 100, 25, 'width' => 3)
TkcLine.new(canvas, 0, 35, 100, 35, 'width' => 4)
TkcLine.new(canvas, 0, 55, 100, 55, 'width' => 3, 'dash' => ".")
TkcLine.new(canvas, 0, 65, 100, 65, 'width' => 3, 'dash' => "-")
TkcLine.new(canvas, 0, 75, 100, 75, 'width' => 3, 'dash' => "-.")
```

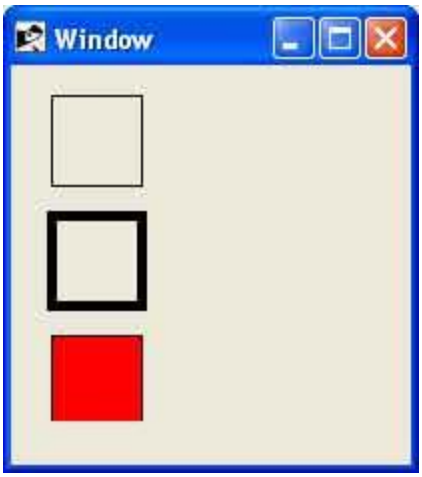
```
TkcLine.new(canvas, 0, 85, 100, 85, 'width' => 3, 'dash' => "-..")
TkcLine.new(canvas, 0, 105, 100, 105, 'width' => 2, 'arrow' => "first")
TkcLine.new(canvas, 0, 115, 100, 115, 'width' => 2, 'arrow' => "last")
TkcLine.new(canvas, 0, 125, 100, 125, 'width' => 2, 'arrow' => "both")
TkcLine.new(canvas, 10, 145, 90, 145, 'width' => 15, 'capstyle' => "round")
Tk.mainloop
```

This will produce the following result −

Example 3

```
require 'tk'

root = TkRoot.new
root.title = "Window"

canvas = TkCanvas.new(root) do
  place('height' => 170, 'width' => 100, 'x' => 10, 'y' => 10)
end

TkcRectangle.new(canvas, 10,  5,   55,  50, 'width' => 1)
TkcRectangle.new(canvas, 10,  65,  55, 110, 'width' => 5)
TkcRectangle.new(canvas, 10,  125, 55, 170, 'width' => 1, 'fill'  => "red")

Tk.mainloop
```

This will produce the following result



Example 4

```
require 'tk'

root = TkRoot.new
root.title = "Window"

canvas = TkCanvas.new(root) do
  place('height' => 170, 'width' => 100, 'x' => 10, 'y' => 10)
end

TkcLine.new(canvas, 0,  10, 100,  10, 'width' => 10, 'fill' => "blue")
TkcLine.new(canvas, 0,  30, 100,  30, 'width' => 10, 'fill' => "red")
TkcLine.new(canvas, 0,  50, 100,  50, 'width' => 10, 'fill' => "green")
TkcLine.new(canvas, 0,  70, 100,  70, 'width' => 10, 'fill' => "violet")
TkcLine.new(canvas, 0,  90, 100,  90, 'width' => 10, 'fill' => "yellow")
TkcLine.new(canvas, 0, 110, 100, 110, 'width' => 10, 'fill' => "pink")
TkcLine.new(canvas, 0, 130, 100, 130, 'width' => 10, 'fill' => "orange")
TkcLine.new(canvas, 0, 150, 100, 150, 'width' => 10, 'fill' => "grey")
Tk.mainloop
```

This will produce the following result

Scrolling:

A **Scrollbar** helps the user to see all parts of another widget, whose content is typically much larger than what can be shown in the available screen space.

A scrollbar displays two arrows, one at each end of the scrollbar, and a slider in the middle portion of the scrollbar. The position and size of the slider indicate which portion of the document is visible in the associated window.

Syntax

Here is a simple syntax to create this widget −

```
TkScrollbar.new {
  .....Standard Options....
  .....Widget-specific Options....
}
```

Standard Options

- activebackground
- highlightbackground
- orient
- takefocus
- background
- highlightcolor
- relief
- troughcolor

- borderwidth
- highlightthickness
- repeatdelay
- cursor
- jump
- repeatinterval

These options have been described in the previous chapter.

Widget Specific Options

Sr.No.	Options & Description
1	**activerelief** => String Specifies the *relief* to use when displaying the element that is active, if any. Elements other than the active element are always displayed with a raised relief.
2	**command** => String Specifies a callback to invoke to change the view in the widget associated with the scrollbar. When a user requests a view change by manipulating the scrollbar, the callback is invoked.
3	**elementborderwidth** => Integer Specifies the width of borders drawn around the internal elements of the scrollbar.
4	**width** => Integer Specifies the desired narrow dimension of the scrollbar window, not including 3-D border, if any. For vertical scrollbars this will be the width and for horizontal scrollbars this will be the height.

Elements of Scrollbar

A scrollbar displays five elements, which are referred in the methods for the scrollbar −

- **arrow1** − The top or left arrow in the scrollbar.
- **trough1** − The region between the slider and arrow1.
- **slider** − The rectangle that indicates what is visible in the associated widget.
- **trough2** − The region between the slider and arrow2.
- **arrow2** − The bottom or right arrow in the scrollbar.

Manipulating Scrollbar

The following useful methods to manipulate the content of a scrollbar −

- **activate(?element?)** − Marks the element indicated by *element* as active, which causes it to be displayed as specified by the **activebackground** and **activerelief** options. The only element values understood by this command are **arrow1**, **slider**, or **arrow2**.
- **delta(deltaX, deltaY)** − Returns a real number indicating the fractional change in the scrollbar setting that corresponds to a given change in slider position.
- **fraction(x, y)** − Returns a real number between 0 and 1 indicating where the point given by x and y lies in the trough area of the scrollbar. The value 0 corresponds to the top or left of the trough, the value 1 corresponds to the bottom or right, 0.5 corresponds to the middle, and so on.
- **get** − Returns the scrollbar settings in the form of a list whose elements are the arguments to the most recent set method.
- **identify(x, y)** − Returns the name of the element under the point given by x and y (such as arrow1), or an empty string if the point does not lie in any element of the scrollbar. X and y must be pixel coordinates relative to the scrollbar widget.
- **set(first, last)** − This command is invoked by the scrollbar's associated widget to tell the scrollbar about the current view in the widget. The command takes two arguments, each of which is a real fraction between 0 and 1. The fractions describe the range of the document that is visible in the associated widget.

Event Bindings

Ruby/Tk automatically creates class bindings for scrollbars that gives them the following default behavior. If the behavior is different for vertical and horizontal scrollbars, the horizontal behavior is described in parentheses −

- Pressing button 1 over arrow1 causes the view in the associated widget to shift up (left) by one unit so that the document appears to move down (right) one unit. If the button is held down, the action auto-repeats.
- Pressing button 1 over trough1 causes the view in the associated widget to shift up (left) by one screenful so that the document appears to move down (right) one screenful. If the button is held down, the action auto-repeats.
- Pressing button 1 over the slider and dragging causes the view to drag with the slider. If the jump option is true, then the view doesn't drag along with the slider; it changes only when the mouse button is released.
- Pressing button 1 over trough2 causes the view in the associated widget to shift down (right) by one screenful so that the document appears to move up (left) one screenful. If the button is held down, the action auto-repeats.
- Pressing button 1 over arrow2 causes the view in the associated widget to shift down (right) by one unit so that the document appears to move up (left) one unit. If the button is held down, the action auto-repeats.
- If button 2 is pressed over the trough or the slider, it sets the view to correspond to the mouse position; dragging the mouse with button 2 down causes the view to drag with the mouse. If button 2 is pressed over one of the arrows, it causes the same behavior as pressing button 1.
- If button 1 is pressed with the Control key down, then if the mouse is over arrow1 or trough1 the view changes to the very top (left) of the document; if the mouse is over arrow2 or trough2 the view changes to the very bottom (right) of the document; if the mouse is anywhere else then the button press has no effect.
- In vertical scrollbars the Up and Down keys have the same behavior as mouse clicks over arrow1 and arrow2, respectively. In horizontal scrollbars these keys have no effect.
- In vertical scrollbars Control-Up and Control-Down have the same behavior as mouse clicks over trough1 and trough2, respectively. In horizontal scrollbars these keys have no effect.

- In horizontal scrollbars the Up and Down keys have the same behavior as mouse clicks over arrow1 and arrow2, respectively. In vertical scrollbars these keys have no effect.
- In horizontal scrollbars Control-Up and Control-Down have the same behavior as mouse clicks over trough1 and trough2, respectively. In vertical scrollbars these keys have no effect.
- The Prior and Next keys have the same behavior as mouse clicks over trough1 and trough2, respectively.
- The Home key adjusts the view to the top (left edge) of the document.
- The End key adjusts the view to the bottom (right edge) of the document.

Examples

```
require "tk"

list = scroll = nil

list = TkListbox.new {
  yscroll proc{|idx|
    scroll.set *idx
  }
  width 20
  height 16
  setgrid 1
  pack('side' => 'left', 'fill' => 'y', 'expand' => 1)
}
scroll = TkScrollbar.new {
  command proc{|idx|
    list.yview *idx
  }
 pack('side' => 'left', 'fill' => 'y', 'expand' => 1)
}

for f in Dir.glob("*")
  list.insert 'end', f
end

Tk.mainloop
```

This will produce the following result