

UNIT-1

Ruby Introduction:

- **Ruby** is an open-source object-oriented scripting language invented in the mid-90s by Yukihiro Matsumoto.
- Unlike languages such as C and C++, a scripting language doesn't talk directly to hardware. It's written to a text file and then parsed by an interpreter and turned into code. These programs are generally procedural in nature, meaning they are read from top to bottom.
- Object-oriented languages, on the other hand, break out pieces of code into objects that can be created and used as needed. You can reuse these objects in other parts of the program, or even other applications.
- Yukihiro wanted to create a scripting language that leveraged object-oriented programming and increase code reuse to help speed up development. And so the Ruby programming language was born, using simple language and syntax to handle data and logic to solve problems.
- The Ruby programming language is a highly portable general-purpose language that serves many purposes.
 1. Ruby is great for building desktop applications, static websites, data processing services and also automation tools.
 2. It's used for web servers, DevOps, and web scraping and crawling.
- when you add in the functionality of the Rails application framework, you can do even more, especially database-driven web applications.

How Ruby on Rails?

- Ruby stands alone as a high-level programming language.
- Ruby on Rails is the application framework that boosted its popularity and made it a great language for the cloud.

SCRIPTING LANGUAGES (CS3208PE)

- Ruby on Rails is “an open-source web framework that is optimized for programmer happiness and sustainable productivity.”
- The Ruby on Rails framework consists of pre-written Ruby code for things like communication, file handling, database connections, and more.
- There are over a million websites written in Ruby on Rails — a wide range of heavy-hitting business and entertainment sites, including GitHub, Twitch, Bloomberg, SoundCloud, Hulu, Square, Basecamp, Airbnb, Hulu, The Weather Channel, Instacart, and Twitter.

	Ruby	Ruby on Rails
• What it is	A language	A framework
• What inspired it	Perl, Smalltalk	Django
• What it's written in	C	Ruby
• What it's used for	Desktop apps, static sites	Data-driven web apps, marketplaces

Ruby vs. Python

- One of the languages Ruby gets compared to most often is Python.
- Ruby and Python have a lot in common and can be used for many of the same purposes, which can make it hard for developers who are deciding which language to learn or which to use for a specific project.
- Both Ruby and Python are high-level server-side scripting languages with clear and easy-to-read syntax, but there are some important technical differences.

Differences between Ruby vs Python

SCRIPTING LANGUAGES (CS3208PE)

Some of the differences between Ruby vs Python include:

- 1. Python supports multiple IDEs, whereas Ruby supports only EclipseIDE.
- 2. With Python you're limited to the Django framework; with Ruby, you're limited to Rails.
- 3. Ruby uses a powerful blocks feature, but Python offers more libraries.
- 4. Ruby is a true object-oriented language, but Python has more traction among data scientists.

• **Why should I learn Ruby?**

- 1. The Ruby programming language is designed for programmer productivity & Easy for Developers
- 2. it's high level and has a simple syntax.
- 3. Developer can write code with in less time and can focus on finding a solution to your problem.

Note: While many low-level languages require lines and lines of code for the smallest thing, with Ruby, you can write your first cloud application in just a few hours.

Survey Reports on Ruby

- The 2020 Stack Overflow Developer Survey names Ruby the 14th most popular programming language in the world with 7.1% of respondents being Ruby on Rails developers.
- It's also an excellent choice for building applications quickly and definitely has the edge over Python when it comes to web development. Hundreds of thousands of Ruby websites are running across the world.

The structure and Execution of Ruby Programs:

Ruby Arrays

Literals of Ruby Array are created by placing a comma-separated series of object references between the square brackets. A trailing comma is ignored.

Example

```
#!/usr/bin/ruby

ary = [ "fred", 10, 3.14, "This is a string", "last
element", ]
ary.each do |i|
  puts i
end
```

This will produce the following result –

```
fred
10
3.14
This is a string
last element
```

Ruby Hashes

A literal Ruby Hash is created by placing a list of key/value pairs between braces, with either a comma or the sequence => between the key and the value. A trailing comma is ignored.

Example

```
#!/usr/bin/ruby

hsh = colors = { "red" => 0xf00, "green" => 0x0f0,
"blue" => 0x00f }
hsh.each do |key, value|
  print key, " is ", value, "\n"
end
```

This will produce the following result –

```
red is 3840
green is 240
blue is 15
```

Ruby Ranges:

A Range represents an interval which is a set of values with a start and an end. Ranges may be constructed using the `s..e` and `s...e` literals, or with `Range.new`.

Ranges constructed using `..` run from the start to the end inclusively. Those created using `...` exclude the end value. When used as an iterator, ranges return each value in the sequence.

A range `(1..5)` means it includes 1, 2, 3, 4, 5 values and a range `(1...5)` means it includes 1, 2, 3, 4 values.

Example

```
#!/usr/bin/ruby

(10..15).each do |n|
  print n, ' '
end
```

This will produce the following result –

```
10 11 12 13 14 15
```

Package Management with RUBYGEMS:

RUBYGEMS is a standardized packaging and installation framework for libraries and applications, making it easy to locate, install, upgrade, and uninstall Ruby packages.

It provides users and developers with four main facilities.

1. A standardized package format
2. A central repository for hosting packages in this format
3. Installation and management of multiple, simultaneously installed versions of the same library,

4. End-user tools for querying, installing, uninstalling, and otherwise manipulating these packages.

In the RubyGems world, developers bundle their applications and libraries into single files called gems.

RubyGems provides a command-line tool, appropriately named `gem`, for manipulating the gem files.

We discuss three important things In Package Management with RUBYGEMS which are

1. Install RubyGems on your computer.
2. Use RubyGems to install other applications and libraries.
3. Write your own gems.

1. Install RubyGems on your computer

To use RubyGems, we need to download and install the RubyGems system from the project's home page at <http://rubygems.rubyforge.org>.

After downloading and unpacking the distribution, we can install it using the included installation script.

```
% cd rubygems0.7.0
```

```
% ruby install.rb
```

Depending on your operating system, you may need suitable privileges to write files into Ruby's `site_ruby/` and `bin/` directories.

The best way to test that RubyGems was installed successfully or not by using following

```
% gem help
```

RubyGems is a sophisticated package manager for Ruby. This is a basic help message containing pointers to more information

```
% gem help
RubyGems is a sophisticated package manager for Ruby. This is
a basic help message containing pointers to more information.

Usage:
  gem -h/--help
  gem -v/--version
  gem command [arguments...] [options...]

Examples:
  gem install rake
  gem list --local
  gem build package.gemspec
  gem help install
```

Installing Application Gems

Locating and installing Rake with RubyGems is simple by using command

```
% gem install -r rake
```

Attempting remote installation of 'Rake'

Successfully installed rake, version 0.4.3

```
% rake --version
```

rake, version 0.4.3

RubyGems downloads the Rake package and installs it. Because Rake is an application, RubyGems downloads both the Rake libraries and the command-line program rake. you could use RubyGems' version requirement operators to specify criteria by which a version would be selected.

```
% gem install -r rake v"< 0.4.3"
```

Attempting remote installation of 'rake'

Successfully installed rake, version 0.4.2

```
% rake--version
```

rake, version 0.4.2

Both the `require_gem` method and the `add_dependency` attribute in a `Gem::Specification` accept an argument that specifies a version dependency.

RubyGems version dependencies are of the form operator major.minor.patch_level.

Listed below is a table of all the possible version operators.

Table 17.1. Version operators

Both the `require_gem` method and the `add_dependency` attribute in a `Gem::Specification` accept an argument that specifies a version dependency. RubyGems version dependencies are of the form `operator major.minor.patch_level`. Listed below is a table of all the possible version operators.

Operator	Description
=	Exact version match. Major, minor, and patch level must be identical.
!=	Any version that is not the one specified.
>	Any version that is greater (even at the patch level) than the one specified.
<	Any version that is less than the one specified.
>=	Any version greater than or equal to the specified version.
<=	Any version less than or equal to the specified version.
~>	“Boxed” version operator. Version must be greater than or equal to the specified version <i>and</i> less than the specified version after having its minor version number increased by one. This is to avoid API incompatibilities between minor version releases.

Installing and Using Gem Libraries

we use RubyGems to install Ruby libraries to develop own programs. Since RubyGems enables you to install and manage multiple versions of the same library.

For complete installation first of all we need to find and install the BlueCloth gem.

```
% gem query -rn Blue
```

```
*** REMOTE GEMS ***
```

```
BlueCloth (0.0.4, 0.0.3, 0.0.2)
```

```
BlueCloth is a Ruby implementation of Markdown, a text-to-HTML  
conversion tool for web writers. Markdown allows you to write using  
an easy-to-read, easy-to-write plain text format, then convert it  
to structurally valid XHTML (or HTML).
```

The latest is downloaded by default.


```
% gem install -r BlueCloth
Attempting remote installation of 'BlueCloth'
Successfully installed BlueCloth, version 0.0.4
```

Generating API Documentation

Being that this is your first time using BlueCloth, you're not exactly sure how to use it. You need some API documentation to get started. RubyGems will generate RDoc documentation for the gem it is installing.

```
% gem install -r BlueCloth --rdoc
Attempting remote installation of 'BlueCloth'
Successfully installed BlueCloth, version 0.0.4
Installing RDoc documentation for BlueCloth-0.0.4...
WARNING: Generating RDoc on .gem that may not have RDoc.
       bluecloth.rb: cc.....
Generating HTML...
```

We have two ways to view this. The hard way (though it really isn't that hard) is to open RubyGems' documentation directory and browse the documentation directly. As The most reliable way to find the documents is to ask the gem command where your RubyGems main directory is located For example:

```
% gem environment gemdir
/usr/local/lib/ruby/gems/1.8
```

RubyGems stores generated documentation in the doc/ subdirectory of this directory, in this case /usr/local/lib/ruby/gems/1.8/doc. You can open the file index.html and view the documentation. If you find yourself using this path often, you can create a shortcut.

Here's one way to do that on Mac OS X

```
% gemdoc=`gem environment gemdir`/doc
% ls $gemdoc
BlueCloth-0.0.4
% open $gemdoc/BlueCloth-0.0.4/rdoc/index.html
```

boxe

Let's Code

Now you've got BlueCloth installed and you know how to use it, you're ready to write some code

With RubyGems, though, we can take advantage of its packaging and versioning support.

To do this, we use `require_gem` in place of `require`.

```
require 'rubygems'
require_gem 'BlueCloth', ">= 0.0.4"
doc = BlueCloth::new <<MARKUP
  This is some sample [text][1]. Just learning to use [BlueCloth][1].
  Just a simple test.
  [1]: http://ruby-lang.org
MARKUP
puts doc.to_html
```

produces:

```
<p>This is some sample <a href="http://ruby-lang.org">text</a>. Just
learning to use <a href="http://ruby-lang.org">BlueCloth</a>.
Just a simple test.</p>
```

The first two lines are the RubyGems-specific code. The first line loads the RubyGems core libraries that we'll need in order to work with installed gems.

```
require 'rubygems'
```

Creating Your Own Gems

RubyGems makes things for the users of an application or library and are probably ready to make a gem of our own. If we're creating code to be shared with the open-source community, RubyGems are an ideal way for end-users to discover, install, and uninstall our code.

MomLog : - is an open-source license software which provide a powerful way to manage internal, company projects, or even personal projects, since they make upgrades and rollbacks so simple.

Package Layout

1. The first task in creating a gem is organizing your code into a directory structure that makes sense

SCRIPTING LANGUAGES (CS3208PE)

2. Put all of your Ruby source files under a subdirectory called lib/.
3. Always include a README file including a project summary, author contact information, and pointers for getting started.
4. Use RDoc format for this file so you can add it to the documentation that will be generated during gem installation.
5. Tests should go in a directory called test/.
6. Any executable scripts should go in a subdirectory called bin/.
7. Source code for Ruby extensions should go in ext/.

The Gem Specification

The gem specification, or `gemspec` is a collection of metadata in Ruby or YAML that provides key information about your gem. The `gemspec` is used as input to the gem-building process. You can use several different mechanisms to create a gem, Here's your first, basic MomLog gem.

```
require 'rubygems'
SPEC = Gem::Specification.new do |s|
  s.name      = "MomLog"
  s.version   = "1.0.0"
  s.author    = "Jo Programmer"
  s.email     = "jo@joshost.com"
  s.homepage  = "http://www.joshost.com/MomLog"
  s.platform  = Gem::Platform::RUBY
  s.summary   = "An online Diary for families"
  s.candidates = Dir.glob("{bin,docs,lib,test}/**/*")
  s.files     = candidates.delete_if do |item|
    item.include?("CVS") || item.include?("rdoc")
  end
  s.require_path = "lib"
  s.autorequire  = "momlog"
  s.test_file    = "test/ts_momlog.rb"
  s.has_rdoc     = true
  s.extra_rdoc_files = ["README"]
  s.add_dependency("BlueCloth", ">= 0.0.4")
end
```

Runtime Magic

The next two attributes, `require_path` and `autorequire`, let you specify the directories that will be added to the `$LOAD_PATH` when `require_gem`

SCRIPTING LANGUAGES (CS3208PE)

loads the gem, as well as any files that will automatically be loaded using require. In this example, lib refers to a relative path under the MomLog gem directory, and the auto require will cause lib/momlog.rb to be required when require_gem "MomLog" is called.

Building the Gem File

The MomLog gemspec we just created is runnable as a Ruby program. Invoking it will create a gem file, MomLog0.5.0.gem.

```
% ruby momlog.gemspec
Attempting to build gem spec 'momlog.gemspec'
Successfully built RubyGem
Name: MomLog
Version: 0.5.0
File: MomLog-0.5.0.gem
```

Now that you've got a gem file, you can distribute it like any other package. You can put it on an FTP server or a Web site for download or e-mail it to your friends. Once your friends have got this file on their local computers (downloading from your FTP server if necessary), they can install the gem (assuming they have RubyGems installed too) by calling

```
% gem install MomLog-0.5.0.gem
Attempting local installation of 'MomLog-0.5.0.gem'
Successfully installed MomLog, version 0.5.0
```

Ruby and web: Writing CGI scripts:

Q) What is Common Gateway Interface (CGI)?

CGI is not a language. It's a simple protocol that can be used to communicate between Web forms and your program. A CGI script can be

SCRIPTING LANGUAGES (CS3208PE)

written in any language that can read STDIN, write to STDOUT, and read environment variables, i.e. virtually any programming language, including C, Perl, or even shell scripting.

Q) How to Write CGI Scripts

The most basic Ruby CGI script can be written as

```
#!/usr/bin/ruby

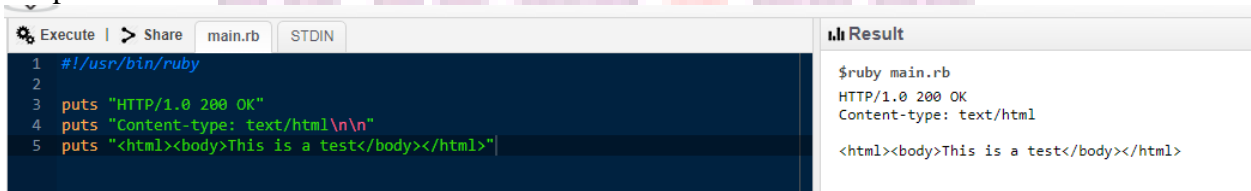
puts "HTTP/1.0 200 OK"
puts "Content-type: text/html\n\n"
puts "<html><body>This is a test</body></html>"
```

If you call this script test.cgi and uploaded it to a Unix-based Web hosting provider with the right permissions, you could use it as a CGI script.

For example, if you have the Web site <https://www.example.com/> hosted with a Linux Web hosting provider and you upload test.cgi to the main directory and give it execute permissions, then visiting <https://www.example.com/test.cgi> should return an HTML page saying This is a test.

Here when test.cgi is requested from a Web browser, the Web server looks for test.cgi on the Web site, and then executes it using the Ruby interpreter. The Ruby script returns a basic HTTP header and then returns a basic HTML document.

Output:



Execute > Share	main.rb	STDIN	Result
<pre>1 #!/usr/bin/ruby 2 3 puts "HTTP/1.0 200 OK" 4 puts "Content-type: text/html\n\n" 5 puts "<html><body>This is a test</body></html>"</pre>			<pre>\$ruby main.rb HTTP/1.0 200 OK Content-type: text/html <html><body>This is a test</body></html></pre>

SCRIPTING LANGUAGES (CS3208PE)

Q) What is the use of cgi.rb?

Ruby comes with a special library called cgi that enables more sophisticated interactions than those with the preceding CGI script.

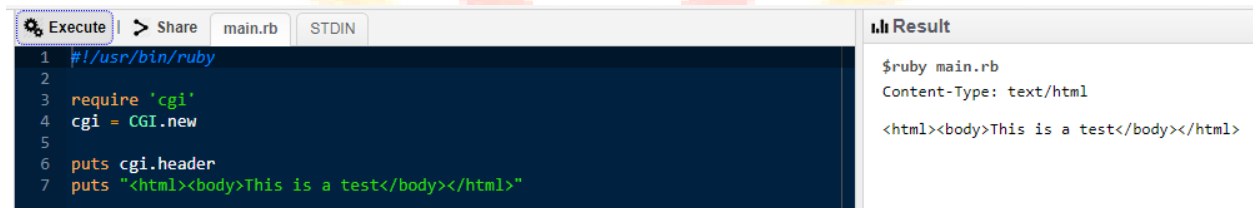
A basic CGI script that uses cgi –

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new

puts cgi.header
puts "<html><body>This is a test</body></html>"
```

output:



The screenshot shows a web-based interface for running a Ruby script. On the left, a code editor displays the script: `#!/usr/bin/ruby`, `require 'cgi'`, `cgi = CGI.new`, `puts cgi.header`, and `puts "<html><body>This is a test</body></html>"`. On the right, the 'Result' panel shows the output: `$ruby main.rb`, `Content-Type: text/html`, and `<html><body>This is a test</body></html>`.

Here, you created a CGI object and used it to print the header line for you.

Form Processing

Using class CGI gives you access to HTML query parameters in two ways.

Suppose we are given a URL of `/cgi-bin/test.cgi?FirstName = Zara&LastName = Ali`.

You can access the parameters `FirstName` and `LastName` using `CGI#[]` directly as follows –

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new
cgi['FirstName'] # => ["Zara"]
cgi['LastName']  # => ["Ali"]
```

SCRIPTING LANGUAGES (CS3208PE)

There is another way to access these form variables. This code will give you a hash of all the key and values –

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new
h = cgi.params # =>
{"FirstName"=>["Zara"],"LastName"=>["Ali"]}
h['FirstName'] # => ["Zara"]
h['LastName']  # => ["Ali"]
```

Following is the code to retrieve all the keys –

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new
cgi.keys # => ["FirstName", "LastName"]
```

If a form contains multiple fields with the same name, the corresponding values will be returned to the script as an array. The [] accessor returns just the first of these. index the result of the params method to get them all.

Note – Ruby will take care of GET and POST methods automatically. There is no separate treatment for these two different methods.

An associated, but basic, form that could send the correct data would have the HTML code like so –

```
<html>
  <body>
    <form method = "POST" action =
"http://www.example.com/test.cgi">
      First Name :<input type = "text" name =
"FirstName" value = "" />
      <br />
      Last Name :<input type = "text" name =
"LastName" value = "" />
      <input type = "submit" value = "Submit Data"
/>
    </form>
  </body>
</html>
```

Ruby - CGI Cookies

Q) What are HTTP cookies?

HTTP cookies are small blocks of data created by a web server while a user is browsing a website and placed on the user's computer or other device by the user's web browser. Cookies are placed on the device used to access a website.

HTTP protocol is a stateless protocol. But for a commercial website, it is required to maintain session information among different pages. For example, one user registration ends after completing many pages. But how to maintain user's session information across all the web pages.

In many situations, using cookies is the most efficient method of remembering and tracking preferences, purchases, commissions, and other information required for better visitor experience or site statistics.

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

Q) How to Handle Cookies in Ruby

we 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.

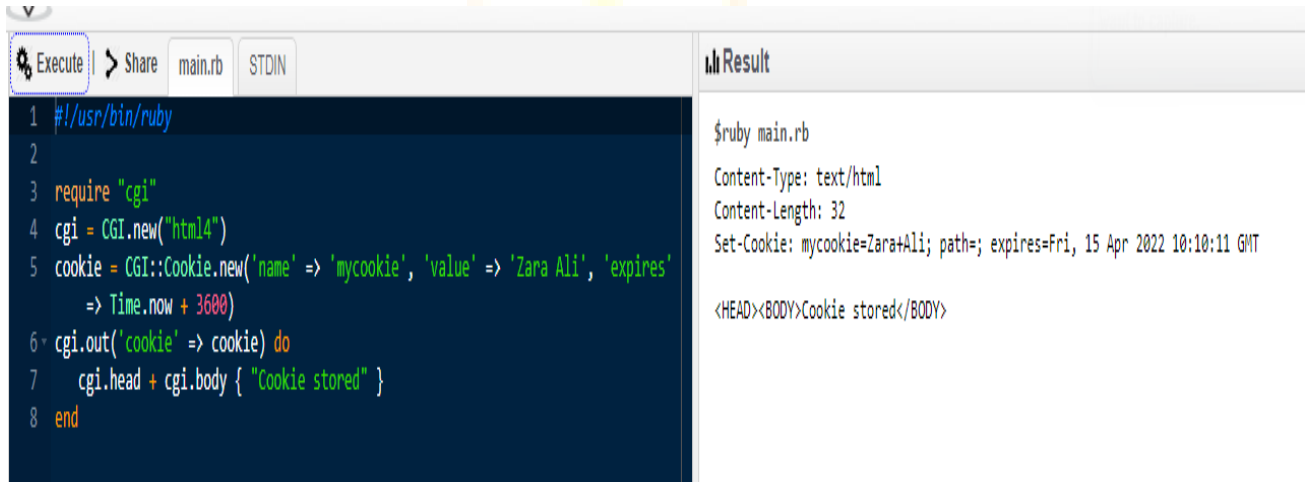
```
#!/usr/bin/ruby

require "cgi"
cgi = CGI.new("html4")
```


SCRIPTING LANGUAGES (CS3208PE)

```
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
```

output:



The screenshot shows an online Ruby interpreter interface. On the left, a code editor displays a Ruby script that sets a cookie named 'mycookie' with the value 'Zara Ali' and an expiration time of 3600 seconds. The script uses `CGI::Cookie.new` and `CGI.out`. On the right, the 'Result' pane shows the output of the script, which includes the HTTP headers for setting a cookie and the body content 'Cookie stored'.

```
1 #!/usr/bin/ruby  
2  
3 require "cgi"  
4 cgi = CGI.new("html4")  
5 cookie = CGI::Cookie.new('name' => 'mycookie', 'value' => 'Zara Ali', 'expires'  
  => Time.now + 3600)  
6 cgi.out('cookie' => cookie) do  
7   cgi.head + cgi.body { "Cookie stored" }  
8 end
```

Result

```
$ruby main.rb  
Content-Type: text/html  
Content-Length: 32  
Set-Cookie: mycookie=Zara+Ali; path=; expires=Fri, 15 Apr 2022 10:10:11 GMT  
  
<HEAD><BODY>Cookie stored</BODY>
```

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
```

output:



The screenshot shows the same online Ruby interpreter interface. The code editor now displays a script that retrieves the value of the 'mycookie' cookie using `cgi.cookies['mycookie']` and outputs it. The 'Result' pane shows the output, which is the value of the cookie, 'Zara Ali'.

```
1 #!/usr/bin/ruby  
2  
3 require "cgi"  
4 cgi = CGI.new("html4")  
5 cookie = cgi.cookies['mycookie']  
6 cgi.out('cookie' => cookie) do  
7   cgi.head + cgi.body { cookie[0] }  
8 end
```

Result

```
$ruby main.rb  
Content-Type: text/html  
Content-Length: 19  
  
<HEAD><BODY></BODY>
```

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

Ruby - CGI Sessions

Q) What is A CGI::Session?

A session is a period of time wherein a user interacts with an app. Session maintains a persistent state for Web users in a CGI environment. Sessions should be closed after use, as this ensures that their data is written out to the store. When you've permanently finished with a session, you should delete it.

```
#!/usr/bin/ruby

require 'cgi'
require 'cgi/session'
cgi = CGI.new("html4")

sess = CGI::Session.new( cgi, "session_key" =>
"a_test", "prefix" => "rubysess.")
lastaccess = sess["lastaccess"].to_s
sess["lastaccess"] = Time.now
if cgi['bgcolor'][0] =~ /[a-z]/
  sess["bgcolor"] = cgi['bgcolor']
end
```

```
cgi.out {  
  cgi.html {  
    cgi.body ("bgcolor" => sess["bgcolor"]) {  
      "The background of this page" +  
      "changes based on the 'bgcolor'" +  
      "each user has in session." +  
      "Last access time: #{lastaccess}"  
    }  
  }  
}
```

Accessing `"/cgi-bin/test.cgi?bgcolor = red"` would turn the page red for a single user for each successive hit until a new "bgcolor" was specified via the URL.

Session data is stored in a temporary file for each session, and the prefix parameter assigns a string to be prepended to the filename, making your sessions easy to identify on the filesystem of the server.

CGI::Session still lacks many features, such as the capability to store objects other than Strings, session storage across multiple servers.

Class CGI::Session

A CGI::Session maintains a persistent state for web users in a CGI environment. Sessions may be memory-resident or may be stored on disk.

Class Methods

Ruby class Class CGI::Session provides a single class method to create a session -

```
CGI::Session::new( cgi[, option])
```

Starts a new CGI session and returns the corresponding CGI::Session object. option may be an option hash specifying one or more of the following -

session_key - Key name holding the session ID. Default is `_session_id`.

session_id - Unique session ID. Generated automatically

SCRIPTING LANGUAGES (CS3208PE)

- **new_session** – If true, create a new session id for this session. If false, use an existing session identified by session_id. If omitted, use an existing session if available, otherwise create a new one.
- **database_manager** – Class to use to save sessions; may be CGI::Session::FileStore or CGI::Session::MemoryStore. Default is FileStore.
- **tmpdir** – For FileStore, directory for session files.
- **prefix** – For FileStore, prefix of session filenames.

Instance Methods

Sr.No.	Methods & Description
1	[] Returns the value for the given key. See example above.
2	[]= Sets the value for the given key. See example above.
3	delete Calls the delete method of the underlying database manager. For FileStore, deletes the physical file containing the session. For MemoryStore, removes the session from memory.
4	update Calls the update method of the underlying database manager. For FileStore, writes the session data out to disk. Has no effect with MemoryStore.

Q) Creating Forms and HTML using CGI method

CGI contains a huge number of methods used to create HTML. You will find one method per tag. In order to enable these methods, you must create a CGI object by calling CGI.new.

To make tag nesting easier, these methods take their content as code blocks. The code blocks should return a String, which will be used as the content for the tag.

For example –

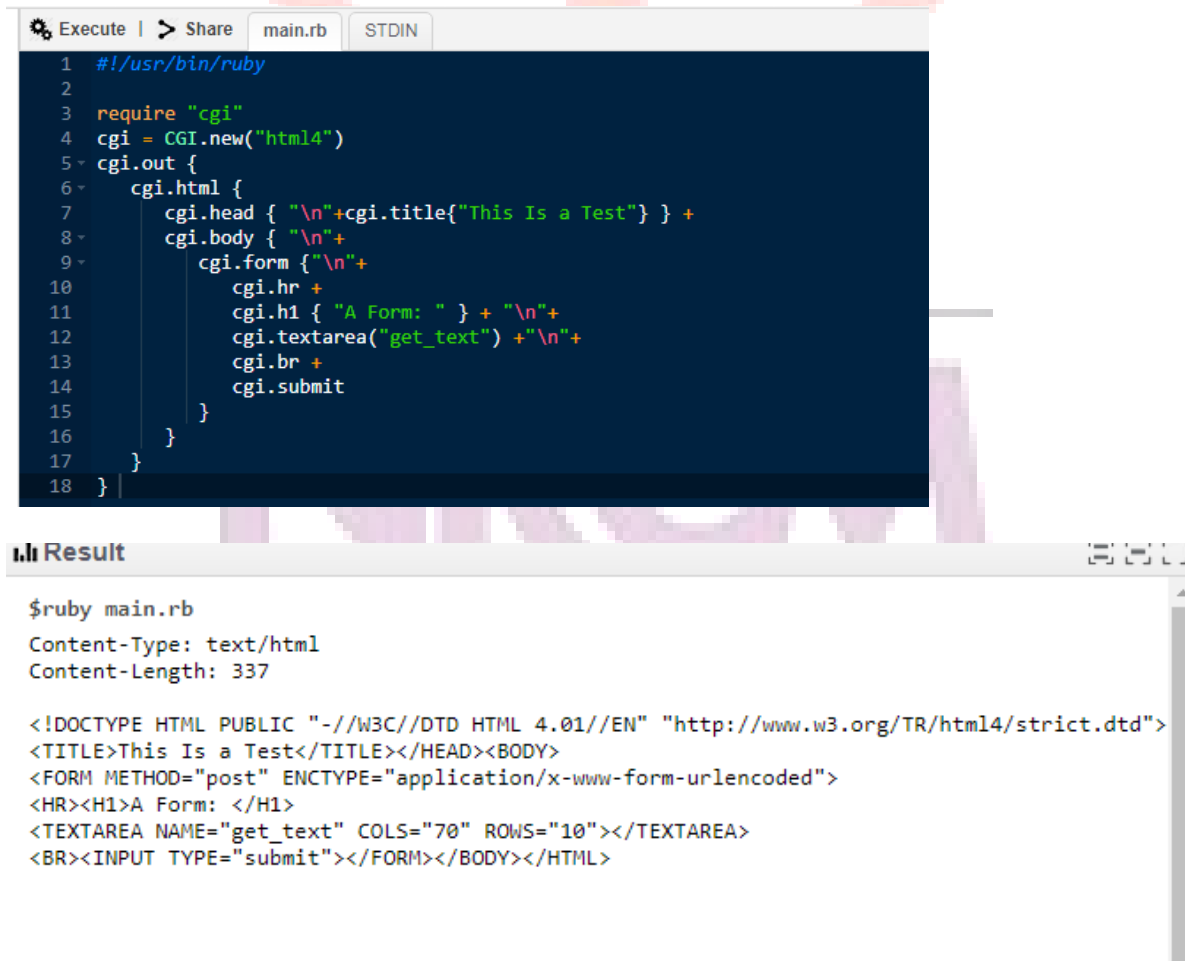
```
#!/usr/bin/ruby
require "cgi"
cgi = CGI.new("html4")
cgi.out {
  cgi.html {
    cgi.head { "\n"+cgi.title{"This Is a Test"} } +
```

SCRIPTING LANGUAGES (CS3208PE)

```
    cgi.body { "\n"+
      cgi.form { "\n"+
        cgi.hr +
        cgi.h1 { "A Form: " } + "\n"+
        cgi.textarea("get_text") + "\n"+
        cgi.br +
        cgi.submit
      }
    }
  }
}
```

NOTE – The form method of the CGI class can accept a method parameter, which will set the HTTP method (GET, POST, and so on...) to be used on form submittal. The default, used in this example, is POST.

Output:



```
Execute | > Share main.rb STDIN
1  #!/usr/bin/ruby
2
3  require "cgi"
4  cgi = CGI.new("html4")
5  cgi.out {
6    cgi.html {
7      cgi.head { "\n"+cgi.title{"This Is a Test"} } +
8      cgi.body { "\n"+
9        cgi.form { "\n"+
10          cgi.hr +
11          cgi.h1 { "A Form: " } + "\n"+
12          cgi.textarea("get_text") + "\n"+
13          cgi.br +
14          cgi.submit
15        }
16      }
17    }
18  }
```

Result

```
$ruby main.rb
Content-Type: text/html
Content-Length: 337

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<TITLE>This Is a Test</TITLE></HEAD><BODY>
<FORM METHOD="post" ENCTYPE="application/x-www-form-urlencoded">
<HR><H1>A Form: </H1>
<TEXTAREA NAME="get_text" COLS="70" ROWS="10"></TEXTAREA>
<BR><INPUT TYPE="submit"></FORM></BODY></HTML>
```

Choice of Web Servers

So far, we've been running Ruby scripts under the control of the Apache Web server. However, Ruby 1.8 and later comes bundled with WEBrick, a flexible, pure-Ruby HTTP server toolkit. Basically, it's an extensible plug-in-based framework that lets you write servers to handle HTTP requests and responses. Here's a basic HTTP server that serves documents and directory indexes.

```
#!/usr/bin/ruby
require 'webrick'
include WEBrick

s = HTTPServer.new( :Port => 2000, :DocumentRoot => File. join(Dir.pwd, "/html") )
trap("INT") { s.shutdown }
s.start ••
```

The HTTPServer constructor creates a new Web server on port 2000. The code sets the document root to be the html/ subdirectory of the current directory. It then uses Kernel.trap to arrange to shut down tidily on interrupts before starting the server running. If you point your browser at `http://localhost:2000`, you should see a listing of your html subdirectory. WEBrick can do far more than serve static content. You can use it just like a Java servlet container

SOAP and Web Services

SOAP:

- The Simple Object Access Protocol (SOAP), is a cross-platform and language-independent RPC protocol based on XML and, usually (but not necessarily) HTTP.
- It uses XML to encode the information that makes the remote procedure call, and HTTP to transport that information across a network from clients to servers and vice versa.

- SOAP has several advantages over other technologies like COM, CORBA etc: for example, its relatively cheap deployment and debugging costs, its extensibility and ease-of-use, and the existence of several implementations for different languages and platforms. •

Installing SOAP4R:

- SOAP4R is the SOAP implementation for Ruby developed by Hiroshi Nakamura and can be downloaded from – download SOAP

If you are aware of gem utility then you can use the following command to install SOAP4R and related packages.

```
$ gem install soap4r --include-dependencies
```

Writing SOAP4R Servers

- SOAP4R supports two different types of servers
 - CGI/FastCGI based (SOAP::RPC::CGIStub)
 - Standalone (SOAP::RPC::StandaloneServer)

Step 1 - Inherit SOAP::RPC::Standalone Server Class

To implement your own stand-alone server you need to write a new class, which will be child of SOAP:: StandaloneServer as follows

```
– class MyServer < SOAP::RPC::StandaloneServer
```

```
.....
```

```
end
```

- Step 2 - Define Handler Methods Second step is to write your Web Services methods, which you would like to expose to the outside world. They can be written as simple Ruby methods. For example, let's write two methods to add two numbers and divide two numbers is

```
– class MyServer < SOAP::RPC::StandaloneServer
```

```
.....
```

```
# Handler methods
```

```
def add(a, b)
```

```
return a + b
```

```
End
def div(a, b)
return a / b
End
end
```

- Step 3 - Expose Handler Methods

Next step is to add our defined methods to our server. The initialize method is used to expose service methods with one of the two following methods –

```
class MyServer < SOAP::RPC::StandaloneServer
def initialize(*args)
add_method(receiver, methodName, *paramArg)
end
end
```

- To understand the usage of inout or out parameters, consider the following service method that takes two parameters (inParam and inoutParam), returns one normal return value (retVal) and two further parameters: inoutParam and outParam –

```
def aMeth(inParam, inoutParam)
retVal = inParam + inoutParam
outParam = inParam . inoutParam
inoutParam = inParam * inoutParam
return retVal, inoutParam, outParam
end
```

- Step 4 - Start the Server:

The final step is to start your server by instantiating one instance of the derived class and calling start method.

```
myServer = MyServer.new('ServerName' , 'urn:ruby: ServiceName' ,
hostname, port)
```


myServer.start

Example

Now, using the above steps, let us write one standalone server –

```
require "soap/rpc/standalone_server"
```

```
begin
```

```
class MyServer < SOAP::RPC::StandaloneServer
```

```
# Expose our services
```

```
def initialize(*args)
```

```
  add_method(self, 'add', 'a', 'b')
```

```
  add_method(self, 'div', 'a', 'b')
```

```
end
```

```
# Handler methods
```

```
def add(a, b)
```

```
  return a + b
```

```
end
```

```
def div(a, b)
```

```
  return a / b
```

```
End
```

```
  end
```

```
server = MyServer.new("MyServer", 'urn:ruby:calculation', 'localhost', 8080)
```

```
trap('INT'){ server.shutdown }
```

```
server.start rescue => err
```

```
puts err.message
```

```
end
```

Ruby

Simple Tk Application and Ruby Widgets

The standard graphical user interface (GUI) for Ruby is Tk. Tk started out as the GUI for the Tcl scripting language developed by John Ousterhout.

Installation

The Ruby Tk bindings are distributed with Ruby but Tk is a separate installation. Windows users can download a single click Tk installation from ActiveState's ActiveTcl.

Mac and Linux users may not need to install it because there is a great chance that its already installed along with OS but if not, you can download prebuilt packages or get the source from the Tcl Developer Xchange.

Tk is an extension of Tcl. Tk provides an X Window system-based toolkit you can use in Tcl scripts to build GUIs. As you might expect, Tk provides a set of Tcl commands beyond the core built-in set. You can use these Tk commands to create windows, menus, buttons, and other user-interface components and to provide a GUI for your Tcl scripts.

Tk uses the X Window system for its graphic components, known as widgets. A widget represents a user-interface component, such as a button, scroll bar, menu, list, or even an entire text window. Tk widgets provide a Motif-like, three-dimensional appearance.

Q) How to run “Hello, World!” in Tk

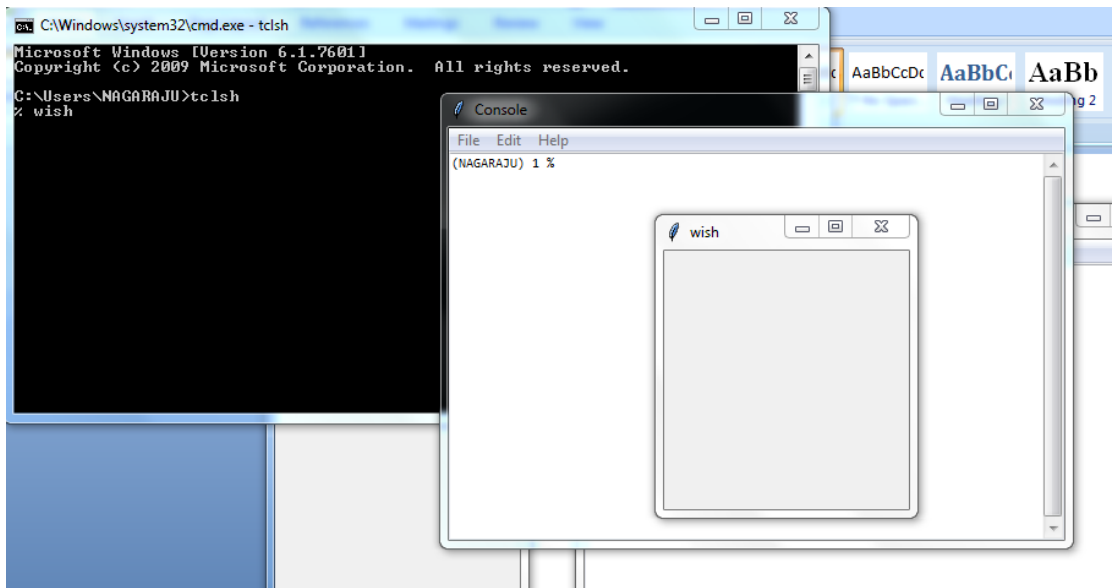
Tk is a major-enough extension to Tcl to warrant its own shell, called wish (the windowing shell). The wish shell interprets all built-in Tcl commands, as well as the Tk commands. You must start X before you can run wish; after all, wish enables you to use X to create graphical interfaces.

The wish program should be in the /usr/bin directory, which should be in your PATH environment variable by default. To start wish, all you have to do is type the following at the shell prompt in a terminal window:

```
wish
%
```

The wish program displays its prompt (the percent sign) and a small window, as shown in the upper-right corner of Figure 25-1.

SCRIPTING LANGUAGES (CS3208PE)

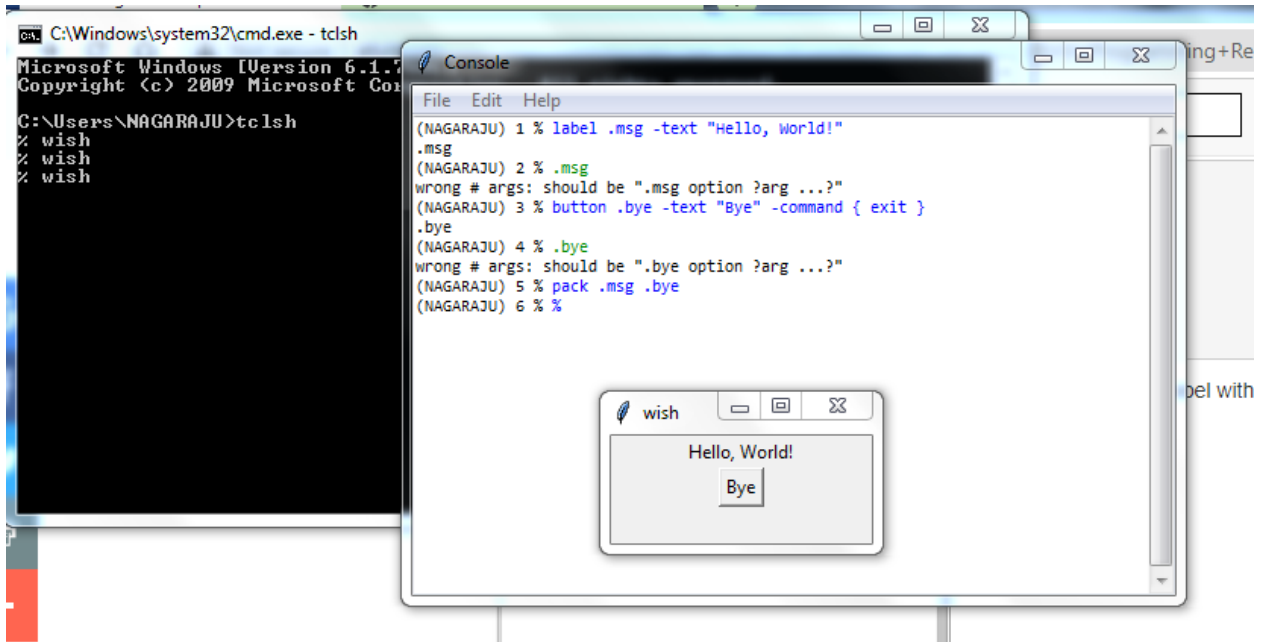


Therefore, wish provides an interactive prompt from which you can enter Tk commands to create a graphical interface. As wish interprets the commands, it displays the resulting graphical interface in the window.

To see how this interactive creation of graphical interface works, try the following commands at the wish prompt:

```
% label .msg -text "Hello, World!"  
.msg  
% button .bye -text "Bye" -command { exit }  
.bye  
% pack .msg .bye  
%
```

SCRIPTING LANGUAGES (CS3208PE)



your path to success...

A typical structure for Ruby/Tk programs is to create the main or root window (an instance of TkRoot), add widgets to it to build up the

SCRIPTING LANGUAGES (CS3208PE)

user interface, and then start the main event loop by calling Tk.mainloop.

The traditional Hello, World! example for Ruby/Tk looks something like this –

```
require 'tk'

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

Here, 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.

If you would run this script, it would produce the following result –



Ruby Widget

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	<p>background => String</p> <p>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.</p>
2	<p>colormap => String</p> <p>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.</p>

3	<p>container => Boolean</p> <p>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.</p>
4	<p>height => Integer</p> <p>Specifies the desired height for the window in pixels or points.</p>
5	<p>width => Integer</p> <p>Specifies the desired width for the window in pixels or points.</p>

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 -



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.
- TkCheckBox 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

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.

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
```

Canvas

- Tk provides a Canvas widget with which you can draw and produce PostScript output.
- 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.



```
require 'tk'
class Draw
  def do_press(x, y)
    @start_x = x
    @start_y = y
    @current_line = TkLine.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
  end
end
```

```
@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

- 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.
- 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.

UNIT-2

Extending Ruby

Just to introduce extension writing, let's write one. This extension is purely a test of the process—it does nothing that you couldn't do in pure Ruby. We'll also present some stuff without too much explanation—all the messy details will be given later. The extension we write will have the same functionality as the following Ruby class.

```
Class MyTest
  def initialize
    @arr = Array.new
  end
  def add(obj)
    @arr.push(obj)
  end
end
```

Ruby Objects in C:

Everything in Ruby is an object, and all variables are references to objects. Most Ruby objects are represented as C pointers to an area in memory that contains the object's data and other implementation details. In C code, all these references are via variables of type VALUE, so when you pass Ruby objects around, you'll do it by passing VALUES.

Working with Immediate Objects: immediate values are not pointers: Fixnum, Symbol, true, false, and nil are stored directly in VALUE.

FIXNUM_P(value) → non zero if value is a Fixnum

SYMBOL_P(value) → non zero if value is a Symbol

NIL_P(value) → non zero if value is nil

RTEST(value) → non zero if value is neither nil nor false

Fixnum values are stored as 31-bit numbers that are formed by shifting the original number left 1 bit and then setting the LSB, or least significant bit (bit0), to 1.

Working with Strings:

C Data Types to Ruby Objects:

INT2NUM(int) → Fixnum or Bignum

INT2FIX(int)→ Fixnum(faster)

LONG2NUM(long→ Fixnum or Bignum

LONG2FIX(int)→ Fixnum(faster)

LL2NUM(long long)→ Fixnum or Bignum (if native system supports long long type)

ULL2NUM(long long)→ Fixnum or Bignum (if native system supports long long type)

CHR2FIX(char)→ Fixnum

rb_str_new2(char*)→ String

rb_float_new(double)→ Float

Ruby Objects to C Data Types:

int NUM2INT(Numeric) (Includes type check)

int FIX2INT(Fixnum) (Faster) unsigned int NUM2UINT(Numeric) (Includes type check)

unsigned int FIX2UINT(Fixnum) (Includes type check)

long NUM2LONG(Numeric)(Includes type check)

long FIX2LONG(Fixnum) (Faster)

unsigned long NUM2ULONG(Numeric) (Includes type check)

char NUM2CHR(Numeric or String)(Includes type check) double NUM2DBL(Numeric)

Ruby String objects are actually references to an RString structure

RString structure contains both a length and a pointer field. You can access the structure via the RSTRING macro.

VALUE str;

RSTRING(str)->len→ length of the Ruby string

RSTRING(str)->ptr→ pointer to strings to rage

Global Variables:

The easiest way to do this is to have the variable be a VALUE (that is, a Ruby object). You then bind the address of this C variable to the name of a Ruby variable. In this case, the \$prefix is optional, but it helps clarify that this is a global variable. And remember: making a stack-based variable a Ruby global is not going to work(for long).

The Juke box Extension:

Interfacing C code with Ruby and sharing data and behavior between the two worlds.

C Data Type Wrapping:

- VALUE Data_Wrap_Struct(VALUE class, void(*mark)(),

Void (*free)(),void*ptr)

Wraps the given C data type ptr, registers the two garbage collection routines (see below), and returns a VALUE pointer to a genuine Ruby object. The C type of the resulting object is T_DATA, and its Ruby class is class.

- **VALUE Data_Make_Struct** (VALUE class, ctype, void(*mark)(), void(*free)(), ctype*)
Allocates and sets to zero a structure of the indicated type first and then proceeds as Data_Wrap_Struct. c-type is the name of the C data type that you're wrapping, not a variable of that type.
- **Data_Get_Struct**(VALUE obj, ctype, ctype*) Returns the original pointer. This macro is a type-safe wrapper around the macro DATA_PTR(obj), which evaluates the pointer.

Object Creation:

Creating an object of class CD Player in your Ruby program.

Cd =CD Player.new

The implementation of new in class Class is fairly simple: it allocates memory for the new object and then calls the object's initialize method to initialize that memory.

Code for CD player class:

```
#include "ruby.h"
#include "cdjukebox.h"
static VALUE cd_player;
//Helper function to free a vendor CD Jukebox
static void cd_free(void *p){
    free_jukebox(p);
}
//Allocate a new CDPlayer object, wrapping
//the vendor's CD Jukebox structure
static VALUE cd_alloc(VALUE klass){
    CDJukebox *jukebox;
    VALUE obj;
    //Vendor library creates the Jukebox
```

```
jukebox=new_jukebox();
//thenwewrapitinsideaRubyCDPlayerobject
obj=Data_Wrap_Struct(klass,0,cd_free,jukebox);
returnobj;
}

//Assign the newly created CDPLayer to a
//particularunit
staticVALUEcd_initialize(VALUEself,VALUEunit){
    intunit_id;
    CDJukebox*jb;
    Data_Get_Struct(self,CDJukebox,jb);
    unit_id=NUM2INT(unit);
    assign_jukebox(jb,unit_id);
    returnself;
}

//Copy a cross state(used by clone and dup).For jukeboxes, we
//actuallycreateanewvendorobjectandsetitsunitnumberfrom
//theold
staticVALUEcd_init_copy(VALUEcopy,VALUEorig){
    CDJukebox*orig_jb;
    CDJukebox*copy_jb;
    if(copy==orig)
        returncopy;
    //we can initialize the copy from other CDPlayers or their
    //subclassesonly
    if(TYPE(orig)!=T_DATA||
    RDATA(orig)>
    dfree!=(RUBY_DATA_FUNC)cd_free){
        rb_raise(rb_eTypeError,"wrongargumenttype");
    }
    //copy all the fields from the original object's CDJukebox
```



```
//structuretothenewobject
Data_Get_Struct(orig,CDJukebox,orig_jb);
Data_Get_Struct(copy,CDJukebox,copy_jb);
MEMCPY(copy_jb,orig_jb,CDJukebox,1);
returncopy;
}

//Theprogresscallbackyieldstothe caller the percent complete
staticvoidprogress(CDJukebox*rec,intpercent){
if(rb_block_given_p()){
if(percent>100)percent=100;
if(percent<0)percent=0;
rb_yield(INT2FIX(percent));
}
}

//Seektoagivenpartofthetrack,invokingtheprocesscallback
//aswego
staticVALUE
cd_seek(VALUEself,VALUEdisc,VALUEtrack){
CDJukebox*jb;
Data_Get_Struct(self,CDJukebox,jb);
jukebox_seek(jb,
NUM2INT(disc),
NUM2INT(track),
progress);
returnQnil;
}

//Returntheaverageseektimeforthisunit
staticVALUE
cd_seek_time(VALUEself)
cd_seek_time(VALUEself)
{
```

```
double tm;
CDJukebox*jb;
Data_Get_Struct(self,CDJukebox,jb);
tm=get_avg_seek_time(jb);
returnrb_float_new(tm);
}
//Returnthisplayer'sunitnumber
staticVALUE
cd_unit(VALUEself){
CDJukebox*jb;
Data_Get_Struct(self,CDJukebox,jb);
returnINT2NUM(jb>
unit_id);
}
voidInit_CDPlayer(){
cCDPlayer=rb_define_class("CDPlayer",rb_cObject);
rb_define_alloc_func(cCDPlayer,cd_alloc);
rb_define_method(cCDPlayer,"initialize",cd_initialize,1);
rb_define_method(cCDPlayer,"initialize_copy",cd_init_copy,1);
rb_define_method(cCDPlayer,"seek",cd_seek,2);
rb_define_method(cCDPlayer,"seek_time",cd_seek_time,0);
rb_define_method(cCDPlayer,"unit",cd_unit,0);
}
NowwecancontrolourjukeboxfromRubyinainice,object-orientedway.
require'CDPlayer'
p=CDPlayer.new(13)
puts"Unitis#{p.unit}"
p.seek(3,16){|x|puts"#{x}%done"}
puts"Avg.timewas#{p.seek_time}seconds"
p1=p.dup
puts"Clonedunit=#{p1.unit}"
```

Produces:

Unitis13

26%done

79%done

100%done

Avg.timewas1.2seconds

Cloned unit=1

Memory Allocation:

You may sometimes need to allocate memory in an extension that won't be used for object storage—perhaps you've got a giant bit map for a Bloom filter, an image, or a Whole bunch of little structures that Ruby doesn't use directly.

To work correctly with the garbage collector, you should use the following memory allocation routines. These routines do a little bit more work than the standard malloc.

For instance, if `ALLOC_N` determines that it cannot allocate the desired amount of memory, it will invoke the garbage collector to try to reclaim some space. It will raise a `NoMemError` if it can't or if the requested amount of memory is invalid.

API: Memory Allocation:

➤ `type*ALLOC_N ctype,n`

Allocates `n` `c-type` objects, where `c-type` is the literal name of the `Ctype`, Not a variable of that type.

➤ `type*ALLOC ctype`

Allocates a `c-type` and casts the result to a pointer of that type.

➤ `REALLOC_N var,ctype,n`

Reallocates `n` `c-types` and assigns the result to `var`, a pointer to a variable Of type `c-type`.

➤ `type*ALLOCA_N ctype,n`

Allocates memory for `n` objects of `c-type` on the stack—this memory will be automatically freed when the function that invokes `ALLOCA_N` returns.

Ruby Type System:

In Ruby, we rely less on the type(or class) of an object and more on its capabilities. This is called duck typing. You'll find many examples of this if you examine the source code for the interpreter itself. For example, the following code implements the Kernel.exec method.

```
VALUE
rb_f_exec(argc,argv)
int argc;
VALUE*argv;
{
  VALUE prog=0;
  VALUE tmp;
  if(argc==0){
    rb_raise(rb_eArgError,"wrong number of arguments");
  }
  tmp=rb_check_array_type(argv[0]);
  if(!NIL_P(tmp)){
    if(RARRAY(tmp)>
len!=2){
      rb_raise(rb_eArgError,"wrong first argument");
    }
    prog=RARRAY(tmp)>
ptr[0];
    Safe String Value(prog);
    argv[0]=RARRAY(tmp)>
ptr[1];
  }
  if(argc==1&&prog==0){
    VALUE cmd=argv[0];
    Safe String Value(cmd);
    rb_proc_exec(RSTRING(cmd)>
ptr);
  }
  else{
    proc_exec_n(argc,argv,prog);
  }
}
```

```
rb_sys_fail(RSTRING(argv[0])>
ptr);
returnQnil; /*dummy*/
}
```

The first parameter to this method may be a string or an array containing two strings. However, the code doesn't explicitly check the type of the argument. Instead, it First call `srb_check_array_type`, passing in the argument. What does this method do?

Let's see.

```
VALUE
rb_check_array_type(ary)
VALUE ary;
{
returnrb_check_convert_type(ary,T_ARRAY,"Array","to_ary");
}
```

The plot thickens. Let's track down `nrb_check_convert_type`.

```
VALUE
rb_check_convert_type(val,type,tname,method)
VALUE val;
Int type;
Const char*tname,*method;
{
VALUE v;
/*always convert T_DATA*/
if(TYPE(val)==type&&type!=T_DATA)return val;
v=convert_type(val,tname,method,Qfalse);
if(NIL_P(v))return Qnil;
if(TYPE(v)!=type){
rb_raise(rb_eType Error,"%s#%s should return %s",
rb_obj_classname(val),method,tname);
}
Return v;
}
```

Embedding a Ruby Interpreter:

To extending Ruby by adding Ccode, you can also turn the problem around and Embed Ruby itself within your application. You have two ways to do this. The first is to Let the interpreter take control by calling

ruby_run. This is the easiest approach, but it has one significant drawback:

—the interpreter never returns from a ruby_run call.

```
#include "ruby.h"
Int main(void){
/*...our own application stuff...*/
ruby_init();
ruby_init_loadpath();
ruby_script("embedded");
rb_load_file("start.rb");
ruby_run();
exit(0);
}
```

To initialize the Ruby interpreter, you need to call `ruby_init()`. But on some platforms, you may need to take special steps before that.

```
#ifndef NT
Nt Initialize(&argc, &argv);
#endif
#ifdef __MACOS__ && defined(__MWERKS__)
argc = ccommand(&argv);
#endif
```

See `main.c` in the Ruby distribution for any other special defines or setup needed for your platform.

You need the Ruby include and library files accessible to compile this embedded code. On my box (MacOSX) interpreter installed in a private directory, my Make file looks like this.

```
WHERE=/Users/dave/ruby1.8/lib/ruby/1.8/powerpcdarwin/
CFLAGS=-I$(WHERE)-g
LDFLAGS=-L$(WHERE)-lruby-ldl-lobjc
embed:embed.o
$(CC)-oembedembed.o$(LDFLAGS)
```

The second way of embedding Ruby allows Ruby code and your C code to engage in more of a dialogue: the C code calls some Ruby code, and the Ruby code responds. You do this by initializing the interpreter as normal. Then, rather than entering the interpreter's main loop, you instead invoke specific methods in your Ruby code. When these methods return, your C code gets control back. There's a wrinkle, though. If the Ruby code raises an exception and it isn't caught, your C program will terminate. To overcome this, you need to do what the interpreter does and protect all

calls that could raise an exception. This can get messy. The `rb_protect` method call wraps the call to another C function. That second function should invoke our Ruby method. However, the method wrapped by `rb_protect` is defined to take just a single parameter. To pass more involves some ugly C casting. Here's a simple Ruby class that implements a method to return the sum of the numbers From one to max.

```
class Summer
  def sum(max)
    raise "Invalid maximum#{max}" if max < 0
    (max*max+max)/2
  end
end
```

Let's write a C program that calls an instance of this class multiple times. To Create the instance, we'll get the class object (by looking for a top-level constant whose Name is the name of our class). We'll then ask Ruby to create an instance of that class—

`rb_class_new_instance` is actually `acalltoClass.new`. (The two initial 0 parameters are the argument count and a dummy pointer to the arguments themselves.) Once

we have that object, we can invoke its `sum` method using `rb_funcall`.

```
#include "ruby.h"
Static int d_sum;
int Values[]={5,10,15,1,20,0};
static VALUE wrap_sum(VALUE args){
  VALUE* values=(VALUE*)args;
  VALUE summer=values[0];
  VALUE max=values[1];
  Return rb_funcall(summer,id_sum,1,max);
}
static VALUE protected_sum(VALUE summer, VALUE max){
  int error;
  VALUE args[2];
  VALUE result;
  args[0]=summer;
  args[1]=max;
  result=rb_protect(wrap_sum,(VALUE)args,&error);
  return error?Qnil:result;
}
Int main(void){
  Int value;
```

```
int*next=Values;
ruby_init();
ruby_init_loadpath();
ruby_script("embedded");
rb_require("sum.rb");
//get an instance of Summer
VALUE summer=rb_class_new_instance(0,0,
rb_const_get(rb_cObject,rb_intern("Summer")));
id_sum=rb_intern("sum");
while(value=*next++){
VALUE result=protected_sum(summer,INT2NUM(value));
if(NIL_P(result))
printf("Sumto%d doesn't compute!\n",value);
else
printf("Sumto%d is %d\n",value,NUM2INT(result));
}
ruby_finalize();
exit(0);
}
```

The Ruby interpreter was not originally written with embedding in mind. Probably the biggest problem is that it maintains state in global variables, so it is n't thread-safe.

You can embed Ruby—just one interpreter per process.

EmbeddedRubyAPI:

➤ void **ruby_init()**

Sets up and initializes the interpreter. This function should be called before any other Ruby-related functions.

➤ Void **ruby_init_loadpath()**

Initializes the \$: (loadpath) variable; necessary if your code loads any library modules.

➤ Void **ruby_options**(intargc,char**argv)

Gives the Ruby interpreter the command-line options.

➤ Void **ruby_script**(char*name)

Sets the name of the Ruby script (and\$0)to name.

- Void **rb_load_file**(char*file)

Loads the given file in to the interpreter.

- Void **ruby_run**()

Runs the interpreter.

- Void **ruby_finalize**()

Shuts down the interpreter.

Bridging Ruby to Other Languages.

You can write extensions in just about any language, as long as you can bridge the two languages with C. Almost anything is possible, including awkward marriages of Ruby and C++, Ruby and Java, and so on.

But you may be able to accomplish the same thing without resorting to C code.

For example, you could bridge to other languages using middle ware such as SOAP or COM.

Ruby C Language API:

Last, but by no means least, here are some C-level functions that you may find useful when writing an extension. Some functions require an ID: you can obtain an ID for a string by using `rb_intern` and reconstruct the name from an ID by using `rb_id2name`.

The following listing is not complete. Many more functions are available—too many to document the mail, as it turns out. If you need a method that you can't find here, check `ruby.h` or `intern.h` for likely candidates.

- **Ruby Language Core**

`class.c`, `error.c`, `eval.c`, `gc.c`, `object.c`, `parse.y`, `variable.c`

- **Utility Functions**

`dln.c`, `regex.c`, `st.c`, `util.c`

- **Ruby Interpreter**

`dmyext.c`, `inits.c`, `keywordsmain.c`, `ruby.c`, `version.c`

- **Base Library**

`array.c`, `bignum.c`, `compar.c`, `dir.c`, `enum.c`, `file.c`, `hash.c`, `io.c`, `marshal.c`, `math.c`, `numeric.c`, `pack.c`, `prec.c`, `process.c`, `random.c`, `range.c`, `re.c`, `signal.c`, `sprintf.c`, `string.c`, `struct.c`, `time.c`

API: Defining Classes:

- VALUE **rb_define_class**(char*name, VALUE superclass)

Defines a new class at the top level with the given name and super class (for class Object, userb_cObject).

- VALUE **rb_define_module**(char*name)

Defines a new module at the top level with the given name.

- VALUE **rb_define_class_under**(VALUE under, char*name, VALUE superclass)

Defines a nested class under the class or module under.

- VALUE **rb_define_module_under**(VALUE under, char*name)

Defines a nested module under the class or module under.

- Void **rb_include_module**(VALUE parent, VALUE module)

Includes the given module in to the class or module parent.

- Void **rb_extend_object**(VALUE obj, VALUE Emodule)

Extends obj with module.

- VALUE **rb_require**(constchar*name)

Equivalent to require name. Returns Qtrue or Qfalse.

API:Defining Structures:

- VALUE **rb_struct_define**(char*name, char*attribute..., NULL)

Defines a new structure with the given attributes.

- VALUE **rb_struct_new**(VALUEsClass, VALUEargs..., NULL)

Creates an instance of sClass with the given attribute values.

- VALUE **rb_struct_aref**(VALUEstruct, VALUEidx)

Returns the element named or indexed by idx.

- VALUE **rb_struct_aset**(VALUE struct, VALUE idx, VALUE val)

Sets the attribute named or indexed by idx to val.

API:DefiningMethods:

void **rb_define_method**(VALUE classmod, char*name, VALUE(*func)(), int argc)

Defines an instance method in the class or module class mod with the given name, implemented by the C function func and taking argc arguments.

Void **rb_define_alloc_func**(VALUE classmod, VALUE(*func)())

Identifies the allocator for classmod.

Void **rb_define_module_function**(VALUE module, char*name, VALUE(*func)(), int argc)

Defines a method in class module with the given name, implemented by the C function func and taking argc arguments.

Void **rb_define_global_function**(char*name, VALUE(*func)(), int argc)

Defines a global function (a private method of Kernel) with the given name, implemented by the C function func and taking argc arguments.

API: Defining Variables and Constants:

- Void **rb_define_const**(VALUE classmod, char*name, VALUE value)

Defines a constant in the class or module classmod, with the given name and value.

- Void **rb_define_global_const**(char*name, VALUE value)

Defines a global constant with the given name and value.

- Void **rb_define_variable**(const char*name, VALUE*object)

Exports the address of the given object that was created in C to the Ruby name space as name. From Ruby, this will be a global variable, so name should start with a leading dollar sign. Be sure to honor Ruby's rules for allowed variable names; illegally named Variables will not be accessible from Ruby.

Void **rb_define_class_variable**(VALUE class, const char*name, VALUE eval)

Defines a class variable name (which must be specified with a @@ prefix) in the given class, initialized to value.

void **rb_define_virtual_variable**(const char *name, VALUE(*getter)(), void(*setter)())

Exports a virtual variable to a Ruby namespace as the global \$name. No actual storage

Exists for the variable; attempts to get and set the value will call the given functions with the prototypes.

API: Calling Methods:

VALUE **rb_class_new_instance**((int argc, VALUE*argv, VALUE klass))

Return a new instance of class klass. argv is a pointer to an array of argc parameters.

VALUE **rb_funcall**(VALUE recv, ID id, int argc, ...)

Invokes the method given by id in the object recv with the given number of arguments argc and the arguments themselves (possibly none).

VALUE **rb_funcall2**(VALUE recv, ID id, int argc, VALUE*args)

Invokes the method given by id in the object recv with the given number of arguments argc and the arguments themselves given in the C array args.

- `VALUErb_funcall3(VALUErecv, IDid, intargc, VALUE*args)`

Same as `rb_funcall2` but will not call private methods.

API:Exceptions:

Void **`rb_raise`**(VALUEexception, constchar*fmt,...)

Raises an exception. The given stringfmt and remainin garguments are interpreted as with printf.

Void **`rb_fatal`**(constchar*fmt,...)

Raises a Fatal exception, terminating the process. No rescue blocks are called, but Ensure blocks will be called. The given string fmt and remaining arguments are Interpreted as with printf.

Void **`rb_bug`**(constchar*fmt,...)

Terminates the process immediately—no handlers of any sort will be called. The given String fmt and remaining arguments are interpreted as with printf. You should call this function only if a fatal bug has been exposed. You don't write fatalbugs, do you?

Void **`rb_sys_fail`**(constchar*msg)

Raises a platform-specific exception corresponding to the last known system error, with The given msg.

VALUE **`rb_protect`**(VALUE(*body)(), VALUEargs, int*result)

Executes body with the given args and returns nonzero in result if any exception was raised.

Void **`rb_notimplement`**()

Raises a Not Imp Error exception to indicate that the enclosed function is not implemented yet or not available on this platform.

Void **`rb_exit`**(intstatus)

Exits Ruby with the given status. Raises a System Exit exception and calls registered exit functions and finalizers.

UNIT-3

INTRODUCTION TO PERL AND SCRIPTING

Scripts and programs:

Scripting is the action of writing scripts using a scripting language, distinguishing neatly between programs, which are written in conventional programming language such as C, C++, java, and scripts, which are written using a different kind of language.

We could reasonably argue that the use of scripting languages is just another kind of programming. Scripting languages are used for is qualitatively different from conventional programming languages like C++ and Ada address the problem of developing large applications from the ground up, employing a team of professional programmers, starting from well-defined specifications, and meeting specified performance constraints.

Scripting languages, on other hand, address different problems:

- Building applications from 'off the shelf' components
- Controlling applications that have a programmable interface
- Writing programs where speed of development is more important than run-time efficiency.

The most important difference is that scripting languages in corporate features that enhance the productivity of the user in one way or another, making them accessible to people who would not normally describe themselves as programmers, their primary employment being in some other capacity. Scripting languages make programmers of us all, to some extent.

Origin of scripting:

The use of the word 'script' in a computing context dates back to the early 1970s, when the originators of the UNIX operating system create the term 'shell script' for sequence of Commands that were to be read from a file and follow in sequence as if they had been typed in at the

keyword.e.g.an‘AWKscript’,a‘perl script’etc..the name‘script’being used for a text File that was intended to be executed directly rather than being compiled to a different form of file prior to execution.

Other early occurrences of the term ‘script’ can be found. For example, in a DOS-based system, use of a dial-up connection to a remote system required a communication package that used proprietary language to write scripts to automate the sequence of operations required to establish a connection to a remote system.

Note that if we regard a scripts as a sequence of commands to control an application or a device, a configuration file such as a UNIX‘make file’ could be regard as a script.

However, scripts only become interesting when they have the added value that comes from using programming concepts such as loops and branches.

Scripting today:

SCRIPTING IS USED WITH 3 DIFFERENT MEANINGS:

1. A new style of programming which allows applications to be developed much faster than traditional methods allow, and makes it possible for applications to evolve rapidly to meet changing user requirements. This style of programming frequently uses a scripting language to interconnect ‘off the shelf ‘components that are themselves written in conventional language. Applications built in this way are called ‘glue applications’, and the language is called a ‘glue language’.

A glue language is a programming language(usually an interpreted scripting language)that is designed or suited for writing glue code– code to connect software components. They are especially useful for writing and maintaining: Custom commands for a command shell Smaller programs than those that are better implemented in a compiled language "Wrapper “programs for executable, like a batch file that moves or manipulates files and does other things with the operating system before or after running an application like a word processor, spreadsheet, data

SCRIPTING LANGUAGES (CS3208PE)

base, assembler, compiler, etc. Scripts that may change Rapid prototypes of a solution eventually implemented in another, usually compiled, language.

Glue language examples:

AppleScript

ColdFusion

DCL

Embeddable Common Lisp

ecl

Erlang

JCL

JScript and JavaScript

Lua

m4

Perl

PHP

Pure

Python

Rebol

Rexx

Ruby

Scheme

Tcl

Unix Shell

Scripts (ksh, csh, bash, sh and others)

VBScript

Work Flow Language

Windows PowerShell

XSLT

2. Using a scripting language to 'manipulate, customize and automate the facilities of an existing system', as the ECMAScript definition puts

it. Here the script is used to control an application that provides a programmable interface: this may be an API, though more commonly the application is constructed from a collection of objects whose properties and methods are exposed to the scripting language. Example: use of Visual Basic for applications to control the applications in the Microsoft Office Suite.

3. Using a scripting language with its rich functionality and ease of use as an alternate to a conventional language for general programming tasks, particularly system programming and administration. Examples: are UNIX system administrators have for a long time used scripting languages for system maintenance tasks, and administrators of WINDOWS NT systems are adopting a scripting language, PERL for their work.

Characteristics of scripting languages:

These are some properties of scripting languages which differentiate SL from programming languages.

Integrated compile and run: SL's are usually characterized as interpreted languages, but this is just an over simplification. They operate on an immediate execution, without need to issue separate command to compile the program and then to run the resulting object file, and without the need to link extensive libraries into the object code. This is done automatically. A few SL'S are indeed implemented as strict interpreters.

Low overheads and ease of use:

1. Variables can be declared by use
2. The number of different data types is usually limited
3. Everything is string by context it will be converted as number (vice versa)
4. Number of data structures is limited (arrays)

Enhanced functionality enhanced functionality in some areas. For: SL's usually have example, most languages provide string manipulation based on the use of regular expressions, while other languages provide easy

access to low-level operating system facilities, or to the API, or object exported by an application.

Efficiency is not an issue: ease of use is achieved at the expense of efficiency, because efficiency is not an issue in the applications for which SL'S are designed.

A scripting language is usually interpreted from source code or bytecode. By contrast, the software environment the scripts are written for is typically written in a compiled language and distributed in machine code form.

Scripting languages may be designed for use by end users of a program – end-user development–or may be only for internal use by developers, so they can write portions of the program in the scripting language.

Scripting languages typically use abstraction, a form of information hiding, to spare users the details of internal variable types, data storage, and memory management.

Scripts are often created or modified by the person executing them, but they are also often distributed, such as when large portions of games are written in a scripting language.

The characteristics of ease of use, particularly the lack of an explicit compile-link-load sequence, are sometimes taken as the sole definition of a scripting language.

Users For Scripting Languages:

Users are classified into two types

1. Modern applications
2. Traditional users

Modern applications of scripting languages are:

1. Visual scripting: A collection of visual objects is used to construct a graphical interface. This process of constructing a graphical interface is known as visual scripting. The properties of visual objects include text on button, background and foreground colors. These properties of objects can be changed by writing program in a suitable language.

The outstanding visual scripting system is visual basic. It is used to develop new applications. Visual scripting is also used to create enhanced web pages.

2. Scripting components: In scripting languages we use the idea to control the scriptable objects belonging to scripting architecture. Microsoft's visual basic and excel are the first applications that used the concept of scriptable objects. To support all the applications of microsoft the concept of scriptable objects was developed.

3. Web scripting: web scripting is classified into three forms. they are processing forms, dynamic web pages, dynamically generating HTML.

Applications of traditional scripting languages are:

1. System administration,
2. Experimental programming,
3. Controlling applications.

Application areas:

Four main usage areas for scripting languages:

1. Command scripting languages
2. Application scripting languages
3. Markup language
4. Universal scripting languages

1. **Command scripting languages** are the oldest class of scripting languages. They appeared in 1960, when a need for programs and tasks control arised. The most known language from the first generation of such languages is JCL (Job Control Language), created for IBM OS/360 operating system. Modern examples of such languages include shell language, described above, and also text-processing languages, such as sed and awk. These languages were one of the first to directly include support for regular expression matching - a feature that later was included into more general-purpose languages, such as Perl.

2. **Application scripting languages:** Application scripting languages were developed in 1980s, in the era of personal computers, when such

important applications as spreadsheets and database clients were introduced, and interactive session in front of the PC became the norm. One of the prime examples of these languages is Microsoft-created Visual Basic language, and especially its subset named Visual Basic for Applications, designed explicitly for office applications programming.

3.Markup languages are a special case in the sense that they are not a real programming languages, but rather a set of special command words called 'tags' used to mark up parts of text documents, that are later used by special programs called processors, to do all kinds of transformations to the text, such as displaying it in a browser, or converting it to some other data format. The basic idea of markup languages is the separation of contents and structure, and also including formatting commands and interactive objects into the documents. The first markup language named GML (Generic Markup Language) was created in 1969 by IBM. In 1986, ISO created a standard called SGML, based on GML ideas.

4.Universal scripting languages :The languages that belong to this class are perhaps the most well-known. The very term "scripting languages" is associated with them. Most of these languages were originally created for the Unix environment. The goals however were different.The Perl programming language was made for report generation, which is even reflected in its name (Practical Extraction and Report Language). It is commonly said that the primary reason for its enormous popularity is the ability to write simple and efficient CGI scripts for forming dynamic web pages with this language. Perl was there in the right place at the right time. The Python language was originally made as a tool for accessing system services of the experimental operating system Amoeba. Later it became a universal object-oriented scripting language. Implementations exist for the Java Virtual Machine and also for Microsoft Intermediate Language used on Microsoft .NET platform.

Unlike Perl and Python, which make it easy to write completely standalone programs, TCL relies heavily on C and C++ extension modules.

web scripting:

Web is the most fertile areas for the application of scripting languages.

Web scripting divides into three areas

- a. processing forms
- b. creating pages with enhanced visual effects and user interaction and
- c. generating pages 'on the fly' from material held in database.

Processing Web forms:

In the original implementation of the web , when the form is submitted for processing, the information entered by the user is encoded and sent to the server for processing by a CGI script that generates an HTML page to be sent back to the Web browser.

This processing requires string manipulation to construct the HTML page that constitutes the replay, and may also require system access, to run other processes and to establish network connections. Perl is also a language that uses CGI scripting.

Alternatively for processing the form with script running on the server it possible to do some client –side processing within the browser to validate form data before sending it to the server by using JavaScript, VBScript etc.

Dynamic Web pages:

'Dynamic HTML' makes every component of a Webpage (headings, anchors, tablesetc.) a scriptable object. This makes it possible to provide simple interaction with the user using scripts written in JavaScript/Jscript or VBScript, which are interpreted by the browser. Microsoft'sActiveXtechnologyallowsthecreationofpageswithmoreelaborate user interaction by using embedded visual objects called ActiveX controls. These controls are scriptable objects, and can in fact be

scripted in a variety languages. This can be scripted by using Perl scripting engine.

Dynamically generated HTML:

Another form of dynamic Webpage is one in which some or all of the HTML is generated by scripts executed on the server. A common application of the technique is to construct pages whose content is retrieved from a database. For example, Microsoft's IIS web server Implements Active Server Pages(ASP),which in corporate scripts in Jscript or VBScript.

The universe of scripting languages:

Scripting can be traditional or modern scripting, and Web scripting forms an important part of modern scripting. Scripting universe contains multiple overlapping worlds:

- the original UNIX world of traditional scripting using Perl and Tcl
- the Microsoft world of Visual Basic and Active controls
- the world of VBA for scripting compound documents
- the world of client-side and server-side Web scripting.

The overlap is complex, for example web scripting can be done in VBScript, Java Script/Jscript, Perl or Tcl. This universe has been enlarged as Perl and Tcl are used to implement complex applications for large organizations e.g Tcl has been used to Develop a major banking system, and Perl has been used to implement an enterprise-wide document management system for a leading aerospace company.

Names and Values in Perl:

Names:

Like any other programming language,Perl manipulates variables which have name (or identifier) and a value: a value is assigned to a variable by an assignment statement of the form
`name=value;`

Variable names resemble nouns in English, and like English, Perl distinguishes between singular and plural nouns. A singular name is associated with a variable that holds a single item of data (a scalar value), a plural name is associated with a variable that holds a collection of data items (an array or hash). A notable characteristic of Perl is that variable names start with a special character that denotes the kind of thing that the names stand for: scalar data (\$), array (@), hash (%), subroutine (&) etc. The syntax also allows a name that consists of a single non-alphanumeric character after the initial special character, eg. \$\$, \$?; such names are usually reserved for the Perl system. If we write an assignment, eg. $j = j + 1$, the occurrence of j on the left denotes a storage location, while the right-hand occurrence denotes the contents of the storage location. We sometimes refer to these as the lvalue and rvalue of the variable: more precisely we are determining the meaning of the identifier in a left-context or a right-context. In the assignment $a[j] = a[j] + 1$, both occurrences of j are determined in a right-context, even though one of them appears on the left of the assignment.

In conventional programming languages, new variables are introduced by a declaration, which specifies the name of the new variable and also its type, which determines the kind of value that can be stored in the variable and, by implication, the operation that can be carried out on that variable.

Scalar data:

Strings and numbers:

In common with many scripting languages, Perl recognizes just two kinds of scalar

data: strings and numbers. There is no distinction between integer and real numbers as

different types. Perl is a dynamically typed language: the system keeps track of whether a

variable contains a numeric value or a string value, and the user doesn't have to worry about

the difference between strings and numbers since conversions between the two kinds of data

are done automatically as required by the context in which they are used.

Boolean values:

All programming languages need some way of representing truth values and Perl is no

exception. Since scalar values are either numbers or strings, some convention is needed for

representing Boolean values, and Perl adopts the simple rule that numeric zero, "0" and the empty string (" ") mean false, and anything else means true.

Numeric constants:

Numeric constants can be written in a variety of ways, including specific notation, octal

and hexadecimal. Although Perl tries to emulate natural human communication, the common

practice of using commas or spaces to break up a large integer constant into meaningful digit

groups cannot be used, since the comma has a syntactic significance in Perl. Instead,

underscores can be included in a number literal to improve legibility.

String constants:

String constants can be enclosed in single or double quotes. The string is terminated by

the first next occurrence of the quote which started it, so a single-quoted string can include

double quotes and vice versa. The `q`(quote) and `qq`(double quote) operators allow you to use

any character as a quoting character. Thus

`q / any string /` or `q (any string)`

are the same as

`'any string'` and `qq / any string /` or `qq (any string)`

are the same as

`"any string"`

Variables and assignment:

Assignment:

Borrowing from C, Perl uses `'='` as the assignment operator. It is important to note that an assignment statement returns a value, the value assigned. This permits statements like

```
$b = 4 + ( $a = 3 ) ;
```

which assigns the value 3 to `$a` and the value 7 to `$b`. If it is required to interpolate a variable

value without an intervening space the following syntax, borrowed from UNIX shell scripts, is

used:

```
$a = "Java ;
```

```
$b = "$ { a }Script"; which gives $b the value "JavaScript".
```

`<STDIN>` - a special value:

When the 'variable' `<STDIN>` appears in a context where a scalar value is required, it evaluates to a string containing the next

line from standard input, including the

terminating newline. If there is no input queued, Perl will wait until a line is typed and the return key pressed. The empty string is treated as false in a Boolean context. If <STDIN>

appears on the right-hand side of an assignment to a scalar variable, the string containing the input line is assigned to the variable named on the right. If it appears in any other scalar context the string is assigned to the anonymous variable: this can be accessed by the name `$_`; many operations use it as a default.

Scalar Expressions:

Scalar data items are combined into expressions using operators. Perl has a lot of operators, which are ranked in 22 precedence levels. These are carefully chosen so that the 'obvious' meaning is what you get, but the old advice still applies: if in doubt, use brackets to force the order of evaluation. In the following sections we describe the available operators in their natural groupings—arithmetic, strings, logical etc.

Arithmetic operators:

Following the principles of 'no surprises' Perl provides the usual arithmetic operators, including auto-increment and auto-decrement operators after the

manner of C: note that in

```
$c= 17 ; $d= ++$c;
```

The sequence is increment and the assign, whereas in

```
$c= 17 ; $d = $c++;
```

The sequence is assign then increment. As C, binary arithmetic operations can be combined

with assignment, e.g.

```
$a += 3;
```

This adds 3 to \$a, being equivalent to

```
$a = $a + 3;
```

As in most other languages, unary minus is used to negate a numeric value; an almost never-used unary plus operator is provided for completeness.

String Operators

Perl provides very basic operators on strings: most string processing is one using built-in

functions expressions, as described later.

Unlike many languages use + as a concatenation operator for strings,

Perl uses a period for

this purpose: this lack of overloading means that an operator uniquely determines the context

for its operands. The other string operator is x, which is used to replicate strings, e.g.

```
$a = "Hello" x 3;
```

Sets \$a to "HelloHelloHello".

The capability of combining an operator with assignment is extended to string operations. E.g.

```
$foo .= " ";
```

Appends a space to \$foo.

So far, things have been boringly conventional for the most part. However, we begin to get a

taste of the real flavor of Perl when we see how it adds a little magic when some operators,

normally used in arithmetic context, are used in a string context.

Two examples illustrate this.

1. Auto increment :

If a variable has only ever been used in a string context, the auto increment operator can be applied

to it. If the value consists of a sequence of letters, or a sequence of letters followed by

a sequence of digits, the auto increment takes place in string mode starting with the right

most character, with 'carry' along the string. For example, the sequence

```
$a = 'a0'; $b = 'Az9';
```

```
Print ++$a, ' ', ++$b; "/n";
```

Prints a1Ba0.

2. Unary minus :

This has an unusual effect on nonnumeric values. Unary minus applied to a string which

starts with a plus or minus character returns the same string, but

starting with the opposite

sign. Unary minus applied to an identifier returns a string consisting of

minus prefixed to the

characters of the identifier. Thus if we have a variable named

\$config with the value "foo",

then `-config` evaluates the string "-

foo". This is useful, for example, in constructing command

arguments are introduced by -

Comparison operators:

The value of comparisons

is returned as 1 if true, and an empty string ("") if false, in

accordance with the convention described earlier.

Two families of comparison operators provide,

one for numbers and one for strings. The

operator used determines the context, and perl converts the operands

as required to match

the operator.

This duality is necessary because a comparison between strings made up entirely numerical

digits should apply the usual rules for sorting strings ASCII as a collating sequence, and this may

not give the same result as the numerical comparison ('5' < '10') returns the value true as a

numerical comparison having been converted into (5 < 10) whereas the string comparison ('5' lt

'10') returns false, since 10 comes before 5 in the canonical sort order for ASCII strings.

The comparison operator (`<` for numbers, `cmp` for strings), performs a three way

test, returning -1 for less-than, 0 for equal and +1 for greater-than.

Note that the comparison operators are non associative, so an expression like

`$a > $b > $c`

is erroneous.

logical operators:

The logical operators allow to combine conditions using the usual logical operations

'not' (!, not), 'and' (&&, and) and 'or' (||, or). Perl implements the 'and' and 'or' operators in

'shortcut' mode, i.e. evaluation stops as soon as the final result is certain using the rules false

`&&b = false`, and `true || b = true`.

Before Perl 5, only the !, && and || operators were provided. The new `set`, `not`, `and`, `or`, are

provided partly to increase readability, and partly because their extra-low precedence makes it

possible to omit brackets in most circumstances-the precedence ordering is chosen so that

numerical expressions can be compared without having to enclose them in brackets, e.g.

Print "OK\n" if \$a<10 and \$b<12;

Bitwise operators:

The unary tilde(~) applied to a numerical argument performs bitwise negation on its operand,

generating the one's complement. If applied to a string operand it complements all the bits in

the string-effective way of inverting a lot of bits. The remaining bitwise operators- &(and), |

(or) and ^ (exclusive or)-

have a rather complicated definition. If either operand is a number or a

variable that has previously been used as a number, both operands are converted to integers if

needed, and the bitwise operation takes place between the integers. If the both operands are

strings, and if variables have never been used as numbers, Perl performs the bitwise operation

between corresponding bits in the two strings, padding the shorter strings with zeros as required.

Conditional expressions:

A conditional expression is one whose value is chosen from two alternatives at run-time

depending on the outcome of a test. The syntax is borrowed from C:

Test ? true_exp : false_exp

The first expression is evaluated as Boolean value : if it returns true the whole expression is

replaced by true_exp, otherwise it is replaced by false_exp, e.g.

`$a= ($a<0)? 0 : $a;`

1.8.4 Control structures:

TheControlStructuresforconditionalexecutionand repetition all the control

mechanisms is similar to C.

1. BLOCKS:

A block is a sequence of one or more statements enclosed in curly braces.

Eg: `{ $positive=1;
$negative=-1;}`

The last statement is the block terminated by the closing brace.

In, Perl they use conditions to control the evaluation of one or more blocks. Blocks can appear almost anywhere that a statement can appear such a block called bare block.

2. CONDITIONS:

A condition is a Perl expression which is evaluated in a Boolean context: if it

evaluates to zero or the empty string the condition is false, otherwise it is true.

Conditions usually make use of relational operators.

Eg: `$total>50`

`$total>50 and $total<100`

Simple Conditions can be combined into a complex condition using the logical

operators. A condition can be negated using the `!` operator.

Eg: `!($total>50 and $total<100)`

3. CONDITIONAL EXECUTION:

If-then-else statements

`if ($total>0){`

`print "$total\n"-`

`if ($total>0){`

```
print "$total\n"  
} else {  
print "bad total!\n"-
```

A single statement is a block and requires braces round it. The if statement requires that the expression forming the condition is enclosed in brackets. The construct extends to multiple selections

```
Eg: if ($total>70) {  
$grade="A";  
} elsif ($total >50) {  
$grade="B";  
} elsif ($total>40) {  
$grade="C";  
} else {  
$grade="F";  
$total=0;  
}
```

Alternatives to if-then-else

To use a conditional expression in place of an if-then-else construct.

```
if ($a<0)  
($b=0)  
else ($b=1)
```

can be written as

```
$b= ($a<0)? 0:1;
```

To use the 'or' operator between statements

Eg: open (IN, \$ARGV[0] or die

```
"Can't open $ARGV*0+\n";
```

Statement qualifiers

A single statement(not a block) can be followed by a conditional modifier.

```
Eg: print"OK\n" if $volts>=1.5;
```

```
print "Weak\n" if $volts>=1.2 and  
$volts<1.5;
```

```
print "Replace\n" if $volts<1.2;
```

Code using Conditional expressions,

```
Eg: print (($volts>=1.5)? "Ok\n";  
(($volts>=1.2)? "Weak\n";  
"Replace\n"));
```

4. REPETITION:

Repetition mechanisms include both

- Testing Loops
- Counting Loops

TESTING LOOPS

```
While ($a! = $b)
```

```
if($a>$b){
```

```
$a=$a-$b
```

```
} else {
```

```
$b=$b-$a
```

```
}
```

```
}
```

With the if statement, the expression that forms the condition must be enclosed in brackets. But now, while can be replaced by until to give the same effect. Single

statement can use while and until as statement modifiers to improve readability.

```
Eg: $a +=2 while $a<$b;
```

```
$a+=2until$a>$b;
```

Here, although the condition is written after the statement, it is evaluated before the statement is executed, if the condition is initially false the statement will be never

executed.

When the condition is attached to a do loop is same as a statement modifier, so the block is executed at least once.

```
do {
```

```
.....
```

```
} while $a! = $b;
```

Counting Loops:

In C,

```
for ($i= 1;$i<=10;$i++) {  
    $i_square=$i*$i;  
    $i_cube=$i**3;  
    print "$i\t$i_square\t$i_cube\n";  
}
```

In Perl,

```
foreach $i (1...10),  
    $i_square=$i* $i;  
    $i_cube=$i**3;  
    print "$i\t$i_square\t$i_cube\n";  
}
```

LIST, ARRAYS AND HASHES:

LISTS:

A list is a collection of scalar data items which can be treated as a whole, and has a temporary existence on the run-time stack.

It is a collection of variables, constants (numbers or strings) or expressions, which is to be treated as a whole.

It is written as a comma-separated sequence of values,

eg: "red", "green", "blue".

A list often appears in a script enclosed in round brackets. For eg:

("red" , "green", "blue")

Shorthand notation is acceptable in lists, for eg:

(1..8)

("A".. "H" , "O".. "Z")

qw(the quick brown fox) is a shorthand for ("the" , "quick" , "brown" , "fox").

Arrays and Hashes: These are the collections of scalar data items which have an assigned storage space in memory, and can therefore be accessed using a variable name.

Arrays:

An array is an ordered collection of data whose comparisons are identified by an ordinal

index: It is usually the value of an array variable.

The name of the variable always starts with an @, eg: @days_of_week.

NOTE: An array stores a collection, and List is a collection, So it is natural to assign a list

to an array.

Eg: @rainfall = (1.2 , 0.4 , 0.3 , 0.1 , 0 , 0 , 0);

A list can occur as an element of another list.

Eg: @foo = (1 , 2 , 3, "string");

@foobar = (4 , 5 , @foo , 6);

The foobar result would be (4 , 5 , 1 , 2 , 3 , "string" , 6);

Hashes:

An associative array is one in which each element has two components : a key and a

value, the element being 'indexed' by its key.

Such arrays are usually stored in a hashtable to facilitate efficient retrieval, and for this reason Perl uses the term hash for an associative array.

Names of hashes in Perl start with a % character: such a name establishes a list context.

The index is a string enclosed in braces (curly brackets).

Eg: `$somehash{aaa} = 123;`

`$somehash, "$a" = 0; //The key is a the current value of $a.`

`%anotherhash = %somehash;`

Working With Arrays and Lists:

Array Creation:

➤ Array variables are prefixed with the @sign and are populated using either paranthesis or the qwoperator.

Eg: `@array = (1 , 2 , "Heelo");`

`@array = qw/This is an array/;`

➤ In C,C++, Java; Array is a collection of homogeneous elements, whereas; In Perl, Array is a collection of heterogeneous elements.

Accessing Array Elements:

➤ When accessing an individual element, we have to use the '\$' symbol followed by

variable name along with the index in the square brackets.

Eg: `$bar = $foo[2];`

`$foo[2] = 7;`

➤ A group of contiguous elements is called a slice , and is accessed using a simple syntax:

`@foo[1..3]` is the same as the list `($foo[1], $foo[2], $foo[3])`

➤ A slice can be used as the destination of an assignment,

Eg: `@foo*1..3+ = ("hop" , "skip" , "jump");`

➤ Like a slice, a selection can appear on the left of an assignment: this leads to a useful

idiom for rearranging the elements in a list.

Eg: To swap the first two elements of an array, we write as;

`@foo[0 , 1] = @foo[1 , 0];`

Manipulating Lists:

Perl provides several built-in functions for list manipulation. Three useful ones are:



shiftLIST: ReturnsthefirstitemofLIST, andmovestheremainingitemsdown, reducing

the size of LIST by 1.

➤ unshift ARRAY, LIST : The opposite of shift. Puts the items in LIST at the beginning of

ARRAY, moving the original contents up by the required amount.

➤ pushARRAY, LIST : Similartounshift, but adds the values in LISTtotheendofARRAY.

Iterating over Lists:

foreach: The foreach loop performs a simple iteration over all the elements of a list.

Eg: foreach \$item (list) {

.....

}

Theblockisexecuted

repeatedlywiththevariables\$itemtakingeachvaluefromthelist

in turn. The variable can be omitted, in which case \$_ will be used.

ThenaturalPerlidiomformanipulatingallitemsinanarrayis;

foreach (@array){

.....#process \$_

}

Working With Hashes:

➤ A hash is a set of key/value pairs.

➤ Hash variables are preceded by a “%” sign.



Torefertoasingleelementofahash, youwillusethehashvariablenamepreceded by

a ‘\$’ sign and followed by the “key” associated with the value in the curly brackets.

➤ It is also called as associative array.

Creating Hashes:

➤ We can assign a list of key-value pairs to a hash, as, for example,
`%foo = (key1, value1, key2, value2,);`
➤ An alternative syntax is provided using the `=>` operator to associate key-value pairs,
thus:

`%foo= (banana=>'yellow',apple=>'red',grapes=>'green',.....);`

Manipulating Hashes:

Perl provides a number of built-in functions to facilitate manipulation of hashes. If we have a hash called `HASH`,

➤ `keys % HASH` returns a list of the keys of the elements in the hash,
and
➤ `values % HASH` returns a list of the values of the elements in the hash.

Eg: `%foo= (banana=>'yellow',apple=>'red',grapes=>'green',.....)`;

`keys % HASH` returns banana, apple ,grapes

`values % HASH` returns yellow, red, green.

These functions provide a convenient way to iterate over the elements of a hash using `foreach`:

```
foreach (keys % HASH) {  
  process $magic($_)  
}
```

Other useful operators for manipulating hashes are `delete` and `exists`.

➤ `delete $HASH{$key}` removes the element

➤ `exists $HASH{$key}` returns true.

1.10 Strings, Pattern Matching & Regular Expressions in Perl:

The most powerful features of Perl are in its vast collection of string manipulation operators

and functions. Perl would not be as popular as it is today in bioinformatics applications if it did

not contain its flexible and powerful string manipulation capabilities.

String concatenation:

To concatenate two strings together, just use the . dot:

```
$a . $b;  
$c=$a.$b;  
$a=$a.$b;  
$a .=$b;
```

The first expression concatenates \$a and \$b together, but the result was immediately lost

unless it is saved to the third string \$c as in case two. If \$b is meant to be appended to the end

of \$a, use the .= operator will be more convenient. As is any other assignments in Perl, if you

see an assignment written this way \$a=\$a op expr, where op stands for any operator and expr

stands for the rest of the statement, you can make a shorter version by moving the op to the

front of the assignment, e.g., \$a op= expr.

Substring extraction

The counterpart of string concatenation is substring extraction. To extract the substring at

certain location inside a string, use the substr function:

```
$second_char = substr($a, 1, 1);  
$last_char = substr($a, -1, 1);  
$last_three_char = substr($a, -3);
```

The first argument to the substr function is the source string, the second argument is the start

position of the substring in the source string, and the third argument is the length of the

substring to extract. The second argument can be negative, and if that being the case, the start

position will be counted from the back of the source string. Also, the third argument can be

omitted. In that case, it will run to the end of the source string.

A particularly interesting feature in Perl is that the `substr` function can be assigned into as well,

meaning that in addition to string extraction, it can be used as string replacement:

```
substr($a, 1, 1) = 'b'; # change the second character to b
substr($a, -1) = 'abc'; # replace the last character as abc (i.e., also add
two new letters
bc)
```

```
substr($a, 1, 0) = 'abc'; # insert abc in front of the second character
```

Substring search

In order to provide these second argument to `substr`, usually you need to locate the substring to

be extracted or replaced first. The `index` function does the job:

```
$loc1 = index($string, "abc");
$loc2 = index($string, "abc", $loc1 + 1);
print "not found" if $loc2 < 0;
```

The `index` function takes two arguments, the source string to search, and the substring to be

located inside the source string. It can optionally take a third argument to mean the start

position of the search. If the `index` function finds no substring in the source string anymore,

then it returns -1.

Regular expression

Regular expression is a way to write a pattern which describes certain substrings. In general,

the number of possible strings that can match a pattern is large, thus you need to make use of

theregularexpressiontodescribetheminsteadoflistingallpossibilities.

Ifthepossible

substring matches are just one, then maybe the index function is more efficient.

The following are some basic syntax rules of regular expression:

Any character except the following special ones stands for itself. Thus abc matches 'abc', and xyz matches 'xyz'.

The character . matches any single character. To match it only with the . character itself,

put an escape \ in front of it, so \. will match only '.', but . will match anything. To match

the escape character itself, type two of them \\ to escape itself.

Ifinsteadofmatchinganycharacter,youjustwanttomatchasubsetofcharacters,put

all of them into brackets [], thus [abc] will match 'a', 'b', or 'c'. It is also possible to

shorten the listing if characters in a set are consecutive, so [a-z] will match all lowercase

alphabets,[0-

9]willmatchallsingledigits,etc.Acharactersetcanbenegatedbythespecial^character,thus[^0-9]willmatchanythingbutnumbers,and[^a-f]willmatch

anything but 'a' through 'f'. Again, if you just want to match the special symbols

themselves, put an escape in front of them, e.g., \[, \^ and \].

All the above so far just match single characters. The power of regular expression lies in

itsabilitytomatchmultiplecharacterswithsometmetasymbols.The*willmatch 0or

SCRIPTING LANGUAGES (CS3208PE)

more of the previous symbol, the + will match 1 or more of the previous symbol, and d?

will match 0 or 1 of the previous symbol. For example, a* will match

'aaaa...' for any

number of a's including none", a+ will match

1 or more a's, and a? will match zero or

one a's. A more complicated example is to match numbers, which can be written this

way [0-9]+. To matching real numbers, you need to write [0-9]+\.[0-

9]*. Note that the

decimal point and fraction numbers can be omitted, thus we use ?, and * instead of +.

If you want to combine two regular expressions together, just write them consecutively.

If you want to use either one of the two regular expressions, use the | meta symbol.

Thus, a|b will match a or b, which is equivalent to [ab], and a+|b+ will match any string

of a's or b's. The second case cannot be expressed using character subset because [ab]+

does not mean the same thing as a+|b+.

Finally, regular expressions can be grouped together with parentheses to change the

order of their interpretation. For example, a(b|c)d will match 'abd' or 'acd'. Without the

parentheses, it would match 'ab' or 'cd'.

The rules above are simple, but it takes some experience to apply them successfully on the

actual substrings you wish to match. There are no better ways to learn this than simply to write

some regular expressions and see if they match the substrings you have in mind.

The following are some examples:

`[A-Z][a-z]*` will match all words whose first character are capitalized

`[A-Za-z_][A-Za-z0-9_]*` will match all legal perl variable names

`[+-]?[0-9]+\.[0-9]*([E][+-]?[0-9]+)?` will match scientific numbers

`[acgtACGT]+` will match all DNA strings

`^>` will match the `>` symbol only at the beginning of a string

`a$` will match the `a` letter only at the end of a string

In the last two examples above, we introduced another two special symbols. The `^` which when not used inside a character set negates the character set, stands for the beginning of the string. Thus, `^>` will match `'>'` only when it is the first character of the string. Similarly, `$` inside a

regular expression means the end of the string, so `a$` will match `'a'` only when it is the last

character of the string. These are so called anchor symbols.

Another commonly used anchor is `\b` which stands for the boundary of a word. In addition, Perl

introduces predefined character sets for some commonly used patterns, thus `\d` stands for

digits and is equivalent to `[0-`

`9]`, `\w` stands for word letters or numbers, and `\s` stands for space

characters `' '`, `\t`, `\n`, `\r`, etc. The capital letter version of these negates

their meaning, thus `\D`

matches non-digit characters, `\W` matches non-word characters, and `\S`

matches non-whitespaces. The scientific number pattern above can

therefore be rewritten as:

`[+-]?\d+\.[0-9]*([E][+-]?\d+)?`

Pattern matching:

SCRIPTING LANGUAGES (CS3208PE)

Regular expressions are used in a few Perl statements, and their most common use is in pattern

matching. To match a regular expression pattern inside a \$string, use the string operator =~

combines with the pattern matching operator / /:

```
$string =~ /\w+/; # match alphanumeric words in $string
```

```
$string =~ /\d+/; # match numbers in $string
```

The pattern matching operator // does not alter the source \$string. Instead, it just returns a

true or false value to determine if the pattern is found in \$string:

```
if ($string =~ /\d+)/{  
    print "there are numbers in $string\n";  
}
```

Sometimes not only you want to know if the pattern exists in a string, but also what it actually

matched. In that case, use the parentheses to indicate the matched substring you want to

know, and they will be assigned to the special \$1, \$2, ..., variables if the match is successful:

```
if ($string =~ /(\d+)\s+(\d+)\s+(\d+)/) {  
    print "first three matched numbers are $1, $2, $3 in $string\n";  
}
```

Note that all three numbers above must be found for the whole pattern to match successfully,

thus \$1, \$2 and \$3 should be defined when the if statement is true. The same memory of

matched substrings within the regular expression are \1, \2, \3, etc. So, to check if the same

number happened twice in the \$string, you can do this:

```
if ($string =~ /(\d).+\1/){  
    print "$1 happened at least twice in $string\n";  
}
```

}

You cannot use \$1 in the pattern to indicate the previously matched number because \$ means

the end of the line inside the pattern. Use \1 instead.

Pattern substitution:

In addition to matching a pattern,

you can replace the matched substring with a new string

using the substitution operator. In this case, just write the substitution string after the pattern

to match and replace:

```
$string =~ s/\d+/0/; # replace a number with zero
```

```
$string =~ s:/\;/; # replace the forward slash with backward slash
```

Unlike the pattern matching operator, the substitution operator does change the string if a

match is found. The second example above indicates that you do not always need to use / to

break the pattern and substitution parts apart; you can basically use any symbol right after the

separator as the separator. In the second case above,

since what we want to replace is the

forward slash symbol, using it to indicate the pattern boundary would be very cumbersome and

need a lot of escape characters:

```
$string =~ s/\/\/\/\//; # this is the same but much harder to read
```

For pattern matching,

you can also use any separator by writing them with m operator, i.e.,

m:/: will match the forward slash symbol. Naturally, the substitution

string may (and often

does) contain the \1, \2 special memory substring to mean the just matched substrings. For

example, the following will add parentheses around the matched number in the source \$string:

```
$string =~ s/(\d+)/(\1)/;
```

The parentheses in the replacement string have no special meanings, thus they were just added to surround the matched number.

Modifiers to pattern matching and substitution:

You can add some suffix modifier to Perl pattern matching or substitution operator to tell

them more precisely what you intend to do:

/g tells Perl to match all existing patterns, thus the following prints all numbers in

```
$string
```

```
while($string =~ /(\d+)/g){
```

```
print "$1\n";
```

```
}
```

```
$string =~ s/\d+/0/g; # replace all numbers in $string with zero
```

/i tells Perl to ignore cases, thus

```
$string =~ /abc/i; # matches AbC, abC, Abc, etc.
```

/m tells perl to ignore newlines, thus

"a\na\na" =~ /a\$/m will match the last a in the \$string, not the a before the first newline

if /m is not given.

Perl- Subroutines:

A Perl subroutine or function is a group of statements that together perform a task.

You can

divide up your code into separate subroutines. How you divide up your code among different

subroutines is up to you, but logically the division usually is so each function performs a specific task.

Define and Call a Subroutine

The general form of a subroutine definition in Perl programming language is as follows –

```
sub subroutine_name {  
    body of the subroutine  
}
```

The typical way of calling that Perl subroutine is as follows –

```
subroutine_name ( list of arguments );
```

Let's have a look into the following example, which defines a simple function and then calls it.

Because Perl compiles your program before executing it, it doesn't matter where you declare

your

subroutine.

```
#!/usr/bin/perl  
# Function definition  
sub Hello{  
    print "Hello, World!\n";  
}
```

```
# Function call
```

```
Hello();
```

When above program is executed, it produces the following result –

Hello, World!

Passing Arguments to a Subroutine

You can pass various arguments to a subroutine like you do in any other programming language

and they can be accessed inside the function using the special array `@_`.

Thus the first argument

to the function is in `$_[0]`, the second in `$_[1]`, and so on.

You can pass arrays and hashes as arguments like any scalar but

passing more than one array or

hash normally causes them to lose their separate identities.

Passing Lists to Subroutines

Because the `@_` variable is an array, it can be used to supply list to a subroutine. However,

because of the way in which Perl accepts and parses lists and arrays, it can be difficult to extract

the individual elements from `@_`. If you have to pass a list along with other scalar arguments,

then

make list as the last argument as shown below –

```
#!/usr/bin/perl
# Function definition
sub PrintList{
    my @list = @_;
    print "Given list is @list\n";
}
$a = 10;
@b = (1, 2, 3, 4);
# Function call with list parameter
PrintList($a, @b);
```

When above program is executed, it produces the following result –

Given list is 10 1 2 3 4

Passing Hashes to Subroutines

When you supply a hash to a subroutine or operator that accepts a list, then hashes

automatically

translated into a list of key/value pairs. For example –

```
#!/usr/bin/perl
# Function definition
sub PrintHash{
    my (%hash) = @_;
```

```
foreach my $ key ( keys %hash ){
    my $ value = $ hash{$ key};
    print "$ key : $ value\n";
}
}

%hash = ('name' => 'Tom ', 'age' => 19);
# Function call with hash parameter
PrintHash(%hash);
```

When above program is executed, it produces the following result –

```
name :Tom
age : 19
```

Returning Value from a Subroutine

You can return a value from subroutine like you do in any other programming language. If you are

Not returning a value from a subroutine then whatever calculation is last performed in a subroutine is automatically also the return value.

You can return arrays and hashes from the subroutine like any scalar but returning more than one

Array or hash normally causes them to lose their separate identities. So we will use references explained in the next chapter to return any array or hash from a function.

Let's try , the following example, which takes a list of numbers and then returns their average–

```
#!/usr/bin/perl

# Function definition
sub Average{
    # get total number of arguments passed.
    $ n = scalar(@ _);
    $ sum = 0;
    foreach $ item (@ _){
        $sum+=$item;
```



```
}  
$average=$sum/$n;  
return $average;  
}
```

Function call

```
$ num = Average(10, 20, 30);  
print "Average for the given num bers : $ num \n";
```

When above program is executed, it produces the following result –

Average for the given num bers : 20



your path to success...

UNIT- 4

ADVANCED PERL

Finer Points Of Looping:

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages–

Perl programming language provides the following types of loop to handle the looping requirements.

Loop Type Description

While loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

Until loop repeats a statement or group of statements until a given condition becomes true. It tests the condition before executing the loop body.

for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

For each loop The for each loop iterates over a normal list value and sets the variable VAR to be each element of the list in turn.

do...while loop Like a while statement, except that it tests the condition at the end of the loop body nested loops You can use one or more loop inside any another while, for or do..while loop.

Loop Control Statements

Loop control statements change the execution from its normal sequence.

When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C supports the following control statements. Click the following links to check their detail.

Control Statement Description

next statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

last statement Terminates the loop statement and transfers execution

to the statement immediately following the loop.

continue statement A **continue BLOCK**, it is always executed just before the conditional is about to be evaluated again.

redo statement The **redo** command restarts the loop block without evaluating the conditional again. The **continue** block, if any, is not executed.

goto statement Perl supports a **goto** command with three forms: **goto label**, **goto expr**, and **goto &name**.

```
#!/usr/local/bin/perl
```

```
for(;;)
```

```
{
```

```
printf"This loop will run forever.\n";
```

```
}
```

The Infinite Loop

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the **for** loop are required, you can make an endless loop by leaving the conditional expression empty.

You can terminate the above infinite loop by pressing the **Ctrl + C** keys.

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but as a programmer more commonly use the **for (;;)** construct to signify an infinite loop.

Multiple Loop Variables:

For loop can iterate over two or more variables simultaneously. Eg: `for($m=1,$n=1,$m<10,$m++, $n+=2)`

```
{
```

```
.....
```

```
}
```

Here **(,)** operator is a list constructor, it evaluates its left hand argument.

2.2Pack and Unpack:

Pack Function

The pack function evaluates the expressions in LIST and packs them into a binary

structure specified by EXPR. The format is specified using the characters shown in Table

below–

pack EXPR, LIST

Each character may be optionally followed by a number, which specifies a repeat count

for the type of value being packed.that is nibbles, chars, or even bits, according to the

format. A value of * repeats for as many values remain in LIST. Values can be

unpacked with the unpack function.

For example, a5 indicates that five letters are expected. b32 indicates that 32 bits are

expected. h8 indicates that 8 nybbles (or 4 bytes) are expected. P10

indicates that the

structure is 10 bytes long.

Syntax

Following is the simple syntax for this function –

Return Value

This function returns a packed version of the data in LIST using TEMPLATE to

determine how it is coded.

Here is the table which gives values to be used in TEMPLATE.

Character Description

A ASCII character string padded with null characters

A ASCII character string padded with spaces

B String of bits, lowest first

B String of bits, highest first
C A signed character (range usually -128 to 127)
C An unsigned character (usually 8 bits)
D A double-precision floating-point number
F A single-precision floating-point number
H Hexadecimal string, lowest digit first
H Hexadecimal string, highest digit first
I A signed integer
I An unsigned integer
L A signed long integer
L An unsigned long integer
N A short integer in network order
N A long integer in network order
P A pointer to a string
S A signed short integer
S An unsigned short integer
U Convert to uuencode format

```
#!/usr/bin/perl -w
$bits =pack("c",65);
# prints A, which is ASCII
65. print"bits are $bits\n";
$bits =pack("x");
# $bits is now a null
chracter. print"bits are
$bits\n";
$bits =pack("sai",255,"T",30);
# creates a seven charcter string on most
computers' print"bits are $bits\n";
@array=unpack("sai", "$bits");
#Array now contains three elements: 255, T
and 30. print"Array $array[0]\n";
```

V A short integer in VAX (little-endian) order

V A long integer in VAX order

X A null byte

X Indicates "go back one byte"

@ Fill with nulls (ASCII 0)

Example

Following is the example code showing its basic usage –

bits are

A bits

are

bits are T□Array

255

Array T

Array 30

unpack TEMPLATE, STRING

When above code is executed, it produces the following result –

Unpack Function

The unpack function unpacks the binary string STRING using the format specified in

TEMPLATE. Basically reverses the operation of pack, returning the list of packed

values according to the supplied format.

You can also prefix any format field with a %<number> to indicate that you want a 16-

bit checksum of the value of STRING, instead of the value.

Syntax

Following is the simple syntax for this function –

Return Value

This function returns the list of unpacked values.

Here is the table which gives values to be used in TEMPLATE.

SCRIPTING LANGUAGES (CS3208PE)

```
print"Array $array[1]\n";
```

```
print"Array $array[2]\n";
```

Character Description

A ASCII character string padded with null characters

A ASCII character string padded with spaces

B String of bits, lowest first

B String of bits, highest first

C A signed character (range usually -128 to 127)

C An unsigned character (usually 8 bits)

D A double-precision floating-point number

F A single-precision floating-point number

H Hexadecimal string, lowest digit first

H Hexadecimal string, highest digit first

I A signed integer

I An unsigned integer

L A signed long integer

L An unsigned long integer

```
#!/usr/bin/perl -w
```

```
$bits =pack("c",65);
```

```
# prints A, which is ASCII 65.
```

N A short integer in network order

N A long integer in network order

P A pointer to a string

S A signed short integer

S An unsigned short integer

U Convert to uuencode format

V A short integer in VAX (little-endian) order

V A long integer in VAX order

X A null byte

X Indicates "go back one byte"

@ Fill with nulls (ASCII 0)

Example:

Following is the example code showing its basic usage –

bits are

A bits

are

bits are ♦T□Array25

5 Array

T

Array30

When above code is executed, it produces the following result –

```
print"bits are $bits\n";
```

```
$bits =pack("x");
```

```
# $bits is now a null
```

```
chracter. print"bits are
```

```
$bits\n";
```

```
$bits =pack("sai",255,"T",30);
```

```
# creates a seven charcter string on most
```

```
computers' print"bits are $bits\n";
```

```
@array=unpack("sai","$bits");
```

```
#Array now contains three elements: 255, A
```

```
and 47. print"Array $array[0]\n";
```

```
print"Array $array[1]\n";
```

```
print"Array $array[2]\n";
```

```
open FILEHANDLE,
```

```
EXPR open
```

```
FILEHANDLE
```

```
sysopen FILEHANDLE, FILENAME, MODE,
```

```
PERMS sysopen FILEHANDLE,
```

```
FILENAME, MODE
```

```
open(DATA,"<file.txt");
```



```
#!/usr/bin/perl
```

```
open(DATA,"<file.txt")ordie"Couldn't open file file.txt, $!";
```

2.3Files:

The basics of handling files are simple: you associate a filehandle with an external

entity (usually a file) and then use a variety of operators and functions within Perl to

read and update the data stored within the data stream associated with the filehandle.

A filehandle is a named internal Perl structure that associates a physical file with a

name. All filehandles are capable of read/write access, so you can read from and

update any file or device associated with a filehandle. However, when you associate a

filehandle, you can specify the mode in which the filehandle is opened.

Three basic file handles are - STDIN, STDOUT, and STDERR, which represent

standard input, standard output and standard error devices respectively.

Opening and Closing Files

There are following two functions with multiple forms, which can be used to open any

new or existing file in Perl.

Here FILEHANDLE is the file handle returned by the open function and EXPR is the

expression having file name and mode of opening the file.

Open Function

Following is the syntax to open file.txt in read-only mode. Here less than < sign

indicates that file has to be opened in read-only mode.

Here DATA is the file handle which will be used to read the file. Here is the example

which will open a file and will print its content over the screen.

```
open(DATA,">file.txt")or die"Couldn't open file file.txt, $!";  
open(DATA,"+<file.txt");or die"Couldn't open file file.txt, $!";  
open DATA,"+>file.txt"or die"Couldn't open file file.txt, $!";  
open(DATA,">>file.txt")|| die"Couldn't open file file.txt, $!";  
open(DATA,"+>>file.txt")|| die"Couldn't open file file.txt, $!";
```

Following is the syntax to open file.txt in writing mode. Here less than > sign indicates

that file has to be opened in the writing mode.

This example actually truncates (empties) the file before opening it for writing, which

may not be the desired effect. If you want to open a file for reading and writing, you can

put a plus sign before the > or < characters.

For example, to open a file for updating without truncating it –

To truncate the file first –

You can open a file in the append mode. In this mode writing point will be set to the end of the file.

A double >> opens the file for appending, placing the file pointer at the end, so that you

can immediately start appending information. However, you can't read from it unless

you also place a plus sign in front of it –

Following is the table which gives the possible values of different modes.

Entities Definition

```
while(<DATA>  
) { print "$_";  
}
```

```
sysopen(DATA,"file.txt", O_RDWR);
sysopen(DATA,"file.txt", O_RDWR|O_TRUNC );
< or r Read Only Access
> or w Creates, Writes, and Truncates
>> or a Writes, Appends, and Creates
+< or r+ Reads and Writes
+> or w+ Reads, Writes, Creates, and Truncates
+>> or a+ Reads, Writes, Appends, and Creates
```

Sysopen Function

The sysopen function is similar to the main open function, except that it uses

the system open() function, using the parameters supplied to it as the parameters for the system function –

For example, to open a file for updating, emulating the +<filename format from open –

Or to truncate the file before updating –

You can use O_CREAT to create a new file and O_WRONLY- to open file in write only

mode and O_RDONLY - to open file in read only mode.

The PERMS argument specifies the file permissions for the file specified if it has to be created. By default it takes 0x666.

Following is the table, which gives the possible values of MODE.

close

FILEHANDLE

close

```
close(DATA) || die "Couldn't close file properly";
```

Entities Definition

O_RDWR Read and Write

O_RDONLY Read Only

O_WRONLY Write Only

O_CREAT Create the file

O_APPEND Append the file

O_TRUNC Truncate the file

O_EXCL Stops if file already exists

O_NONBLOC

K

Non-Blocking usability

Close Function

To close a filehandle, and therefore disassociate the filehandle from the corresponding

file, you use the close function. This flushes the filehandle's buffers and closes the system's file descriptor.

If no FILEHANDLE is specified, then it closes the currently selected filehandle. It

returns true only if it could successfully flush the buffers and close the file.

Reading and Writing Files

```
#!/usr/bin/perl
```

```
print "What is your name?\n";
```

```
$name = <STDIN>;
```

```
print "Hello
```

```
$name\n";
```

```
#!/usr/bin/perl
```

```
open(DATA, "<import.txt") or die "Can't open  
data"; @lines = <DATA>;
```

```
close(DATA);
```

```
getc FILEHANDLE
```

```
getc
```

```
read FILEHANDLE, SCALAR, LENGTH, OFFSET
```

Once you have an open filehandle, you need to be able to read and write information.

There are a number of different ways of reading and writing data into the file.

The <FILEHANDLE> Operator

The main method of reading the information from an open filehandle is the

<FILEHANDLE> operator. In a scalar context, it returns a single line from the filehandle.

For example –

When you use the <FILEHANDLE> operator in a list context, it returns a list of lines from

the specified filehandle. For example, to import all the lines from a file into an array –

getc Function

The getc function returns a single character from the specified FILEHANDLE, or STDIN if

none is specified –

If there was an error, or the filehandle is at end of file, then undef is returned instead.

read Function

The read function reads a block of information from the buffered filehandle: This

function is used to read binary data from the file.

print FILEHANDLE LIST

print

LIST

print

print "Hello World!\n";

#!/usr/bin/perl

Open file to read

```
open(DATA1,"<file1.txt");
```

```
# Open new file to write
```

```
open(DATA2,">file2.txt");
```

```
# Copy data from one file to
```

```
another. while(<DATA1>)
```

```
{
```

```
print DATA2 $_;
```

The length of the data read is defined by LENGTH, and the data is placed at the start

of SCALAR if no OFFSET is specified. Otherwise data is placed after OFFSET bytes in

SCALAR. The function returns the number of bytes read on success, zero at end of

file, or undef if there was an error.

print Function

For all the different methods used for reading information from filehandles, the main

function for writing information back is the print function.

The print function prints the evaluated value of LIST to FILEHANDLE, or to the current

output filehandle (STDOUT by default). For example –

Copying Files

Here is the example, which opens an existing file file1.txt and read it line by line and

generate another copy file file2.txt.

```
read FILEHANDLE, SCALAR, LENGTH
```

```
#!/usr/bin/perl
```

```
rename("/usr/test/file1.txt","/usr/test/file2.txt");
```

```
#!/usr/bin/perl
```

```
unlink("/usr/test/file1.txt");
```

```
tell FILEHANDLE
```

Renaming a file

Here is an example, which shows how we can rename a file file1.txt to file2.txt.

Assuming file is available in /usr/test directory.

This function renames the takes two arguments and it just rename existing file.

Deleting an Existing File

Here is an example, which shows how to delete a file file1.txt using the unlinkfunction.

Positioning inside a File

You can use to tell function to know the current position of a file and seekfunction to point a particular position inside the file.

tell Function

The first requirement is to find your position within a file, which you do using the tell function

-

```
}  
close( DATA1  
); close(  
DATA2 );  
seek FILEHANDLE, POSITION, WHENCE  
seek DATA, 256, 0;
```

```
#!/usr/bin/perl
```

```
my $file
```

```
="/usr/test/file1.txt";
```

```
my(@description, $size);
```

```
if(-e $file)
```

```
{
```

```
push@description,'binary'if(-B _);
```

This returns the position of the file pointer, in bytes, within FILEHANDLE if specified, or

the current default selected filehandle if none is specified.

seek Function

The seek function positions the file pointer to the specified number of bytes within a file –

The function uses the fseek system function, and you have the same ability to position

relative to three different points: the start, the end, and the current position. You do this

by specifying a value for WHENCE.

Zero sets the positioning relative to the start of the file. For example, the line sets the

file pointer to the 256th byte in the file.

File Information

You can test certain features very quickly within Perl using a series of test operators

known collectively as -X tests. For example, to perform a quick test of the various

permissions on a file, you might use a script like this –
tell

Here is the list of features, which you can check for a file or directory –

Operator Definition

-A Script start time minus file last access time, in days.

-B Is it a binary file?

-C Script start time minus file last inode change time, in days.

-M Script start time minus file modification time, in days.

-O Is the file owned by the real user ID?

-R Is the file readable by the real user ID or real group?

-S Is the file a socket?

-T Is it a text file?


```
push@description,'a socket'if(-S _);
push@description,'a text file'if(-T _);
push@description,'a block special file'if(-b _);
push@description,'a character special file'if(-c _);
push@description,'a directory'if(-d _);
push@description,'executable'if(-x _);
push@description,(($size ==-s _))?"$size
bytes":'empty'; print"$file is ", join(
',@description),"\n";
}
```

- W Is the file writable by the real user ID or real group?
- X Is the file executable by the real user ID or real group?
- b Is it a block special file?
- c Is it a character special file?
- d Is the file a directory?
- e Does the file exist?
- f Is it a plain file?
- g Does the file have the setgid bit set?
- k Does the file have the sticky bit set?
- l Is the file a symbolic link?
- o Is the file owned by the effective user ID?
- p Is the file a named pipe?
- r Is the file readable by the effective user or group ID?
- s Returns the size of the file, zero size = empty file.
- t Is the filehandle opened by a TTY (terminal)?
- u Does the file have the setuid bit set?
- w Is the file writable by the effective user or group ID?
- x Is the file executable by the effective user or group ID?
- z Is the file size zero?

2.4EVAL

The eval operator comes in 2 forms:

In the first form ,eval takes an arbitrary string as an operand and evaluates the

string and executed in the current context.

The value returned is the value of the last expression evaluated.

In case of syntax error or runtime error,eval returns the value “undefined” and places the error in the variable \$@

Ex:

```
$myvar = '...';
```

```
....
```

```
$value = eval “ \$$myvar “;
```

In the second form ,it takes a block as an argument and the block is compiled

only once. If there is a runtime error,the error is returned in \$@.

Instead of try we use eval and instead of catch we test \$@

Ex:

Eval

```
{
```

```
.....
```

```
}
```

```
If ($@ ne “)
```

```
{
```

```
.....
```

```
}
```

2.5Data Structures:

➤ ARRAYS OF ARRAYS

In perl a two dimensional array is constructed by creating an array of references to anonymous arrays.

For ex: @colors = ([35,39,43] , [4,5,8] , [32,31,25]) ;

The array composer converts each comma-separated list to an anonymous array in memory and returns a reference and when we write an exp. Like

```
$colors [0][1] = 64;
```

\$colors [0] is a reference to an array and 2nd subscript represents the element present in that array.

A two dimensional array can be dynamically created by using PUSH operator to add a reference to an anonymous array to the top level array.

For ex: we are interested in converting a table set of data having white spaces between the fields can be converted to two dimensional array by repeatedly using split to put the fields of a line into list and then using push to add the reference to an array.

While (<STDIN>)

```
{  
  Push @table , [split]  
}
```

➤ COMPLEX DATA STRUCTURES

Not only an array of arrays can be created but we can create hashes of hashes

,arrays of hashes and hashes of arrays.

By combining all these possibilities ,data structures of great complexity can be

created ex: doubly linked list.

We can make an element of the array hash containing three fields with keys 'L' (left

neighbour) , 'R'(right neighbour) and 'C'(content).

The values related to L and R are references to element hashes and the value

of C can be anything(scalar,variables,hash ,reference).

Ex:

We can move forwards along the list with

```
$current = $current->{'R'}
```

; And backwards with

```
$current = $current->{'L'}
```

; Create a new element

```
$new = { L =>undef , R=>undef , C=>...} ;
```

And we can insert new element after current element as

```
$new->{'R'}=$current->{'R'} ;
```

```
$current{'R'}->{'L'}= $new;
```

```
$current{'R'}=$new ;
```

```
$new->{'L'} = $current ;
```

And the current element can be deleted as

```
$current->{'L'}->{'R'} = $current->{'R'} ;
```

```
$current->{'R'}->{'L'} = $current->{'L'} ;
```

2.6 Packages:

Packages are the basis for libraries,modules and objects.

It is the unit of code with its own namespace(i.e. separate symbol table),which

determines bindings of names both at compile-time and run-time.

Initially code runs in default package main.

Variables used in a package are global to that package only.

Ex:\$A::x is the variable x in

package A. Package A ;

```
$x = 0;
```

.....

Package

```
B ;  
$x = 1 ;  
.....  
Package  
A ;  
Print $x;  
output: zero.
```

The package B declaration switches to a different symbol table then the package A points to the original symbol table having \$x = 0;.

Nested packages can be created of the form A::B provided the variables should

be of the fully qualified form Ex>\$A::B::x.

A package can have one or more BEGIN routines and also END routines. Package declaration is rarely used on its own.

2.7 Modules:

Libraries and modules are packages contained within a single file and are units of program reusability.

The power of perl is increased by the usage of modules that provide functionality in specific application areas.

To be fact module is nothing but a package contained in a separate file whose name is same as the package name with the extension .pm and makes use of built-in-support.

The use of modules make mathematical routines in the library math.pl are

converted into a module math.pm and can be written as

Ex: Use math ; at the start of the program and the subroutines are available .

The subroutine names imported are those defined in the export list of the math

module and it is possible to suppress the import of names but loses the point of the module.

Ex: `use IO : : File ;`

Indicates a requirement for the module `File.pm` which will be found in a directory called `IO`.

The use of “`use math ('sin', 'cos', 'tan')`” is same as `BEGIN {`

`Require “ Math.pm” ;`

`Math :: import ('sin', 'cos', 'tan');`
`}`

The module names are imported by calling the `import()` method defined in the module. The package writer is free to define `import()` in any way.

2.8 Objects:

Objects in Perl provide a similar functionality as objects in real object oriented programming (OOP), but in a different way. They use the same terminology as OOP, but the words have different meanings as given below.

Object: An object with in Perl is a reference to a data type that knows what class it belongs to. The object is stored as a reference in a scalar variable. The object is said to be blessed into a class: this is done by calling the built in function `bless` in a constructor.

Constructor: A constructor is just a subroutine that returns a reference to an object.

Class: A class is a package that provides methods to deal with objects that belong to it.

Method: A method is a subroutine that expects an object reference as its first argument.

Constructors:

Objects are created by a constructor subroutine which is generally called new. Eg. Package Animal;

```
sub new {  
  my $ref = {  
  };  
  bless  
  ref;  
  return  
  ref;  
}
```

The flower brackets {} returns a reference to an anonymous hash. So the new constructor returns a reference to an object that is an empty hash, and knows that it belongs to the package Animal.

Instances:

We can create the instances for the object with this defined constructor as

```
$Dougal = new Animal;  
$Ermyntrude = new Animal;
```

This makes \$Dougal and \$Ermyntrude references to objects that are empty

hashes, and know that they belong to the Animal class.

Method Invocation:

Perl supports two syntactic forms for invoking methods one is by using arrow operator

and another one is by using Indirect objects. If a class is used to invoke the method,

that argument will be the name of the class. If an object is used to invoke the method,

that argument will be the reference to the object. Whichever it is, we'll call it the

method's invocant. For a class method, the invocant is the name of a package. For an

instance method, the invocant is a reference that specifies an object.

Method Invocation Using the Arrow Operator:

For example if `set_species`, `get_species` are the methods they can be invoked using

arrow operator as follows.

```
$Dougal -> set_species 'Dog';
```

```
$Dougal_is ->= $Dougal->get_species;
```

Method Invocation Using Indirect Objects:

The methods can be invoked by using indirect objects as given below

```
set_species $Dougal, 'Dog';
```

```
$Dougal_is = get _species $Dougal;
```

Attributes:

Subroutine declarations and definitions may optionally have attribute lists associated

with them. An attribute is a piece of data belonging to a particular object.

Unlike most

object- oriented languages, Perl provides no special syntax or support for declaring

and manipulating

attributes. Attributes are often stored in the object itself. For example, if the object is an anonymous hash, we can store the attribute values in the hash using the attribute name as the key.

E.g: sub

```
species { my
$self =shift;
my $was = $self->{'species'};
-----
-----
}
```

Class Methods And Attributes:

There are operations that are relevant to the class and not need to operate on a

specific instance are called class methods or static methods.

Similarly attributes that are common to all instances of a class are called as class

attributes. Class attributes are just package global variables and class methods are just

subroutines that do not require an object reference as the first argument e.g the new constructor.

Inheritance:

Perl only provides method inheritance. Inheritance is realized by including a special array `@ISA` in the package that defines the derived class.

For single inheritance `@ISA` is an array of one element, the name of the base class.

Multiple inheritance can be realized by making `@Isa` an array of more than one element.

Each element in the array @ISA is the name of the another package that is being

used as a class.

If a method cannot be found, the packages referenced in @ISA are recursively searched,

depth first. The current class is derived class and those referenced in @ISA are the base classes.

e.g : package Employee;

use Person;

use strict;

our @ISA = qw(Person); # inherits from Person

2.9 Interfacing to the OS:

2.10 Creating Internet Ware Applications:

The internet is a rich source of information, held on web servers, FTP servers,

POP/IMAP mail servers, news servers etc. A web browser can access information on

web servers and FTP servers, and clients access mail and news servers.

however, this

is not the way of to the information: an 'internet-aware' application can access a server

and collect the information without manual intervention. For suppose that a website

offers 'lookup' facility in which the user a query by filling in a then clicks the 'submit'

button . the data from the form in sent to a CGI program on the server(probably written

in which retrieves the information, formats it as a webpage, and returns the page to the

browser. A perl application can establish a connection to the server, send the request in

the format that the browser would use, collect the returned HTML and then extract the

fields that form the answer to the query. In the same way, a perl application can

establish a connection to a POP3 mail server and send a request which will result in the

server returning a message listing the number of currently unread messages.

Much of the power of scripting languages comes from the way in which they hide the

complexity of operations, and this is particularly the case when we make use of

specialized modules: tasks that might take pages of code in C are achieved in few lines. The

LWP (library for WWW access in perl) collection of modules is a very good case in point

it makes the kind of interaction described above almost trivial. The LWP::simple module

is an interface to web servers. It can be achieved by exploiting modules,

LWP::simple we

can retrieve the contents of a web page in a statement:

```
use LWP::simple $url=...http://www.somesite.com/index.html..;
```

```
$page=get($url);
```

2.11 Dirty Hands Internet Programming:

Modules like LWP::Simple and LWP::UserAgent meet the needs of most programmers requiring web access, and there are numerous other modules for

other types of Internet access.

EX:- Net: : FTP for access to FTP servers

Some tasks may require a lower level of access to the network, and this is provided by

Perl both in the form of modules(e.g IO: : Socket) and at an even lower level by built-in functions.

Support for network programming in perl is so complete that you can use the language

to write any conceivable internet application

Access to the internet at this level involves the use of sockets, and we explain what a

socket is before getting down to details of the programming.

Sockets are network communication channels, providing a bi-directional channel

between processes on different machines.

Sockets were originally a feature of UNIX:other UNIX systems adopted them and the

socket became the de facto mechanism of network communication in the UNIX world.

The popular Winsock provided similar functionality for Windows, allowing Windows

systems to communicate over the network with UNIX systems, and sockets are a

built-in feature of Windows 9X and WindowsNT4.

From the Perl programmer's point a network socket can be treated like an open file it is

identified by a you write to it with print, and read it from operator.

The socket interface is based on the TCP/IP protocol suite, so that all information is

handled automatically.

In TCP a reliable channel, with automatic recovery from data loss or corruption: for this

reason a TCP connection is often described as a virtual circuit.
The socket in Perl is an exact mirror of the UNIX and also permits connections using UDP (Unreliable Datagram Protocol).



your path to success...

UNIT-4

TCL

TCL Structure, syntax, Variables and Data in TCL, Control Flow, Data Structures, input/output, procedures , strings , patterns, files, Advance TCL-eval, source, exec and up level commands, Name spaces, trapping errors, event driven programs, making applications internet aware, Nuts and Bolts Internet Programming, Security Issues, C Interface. Tk-Visual Tool Kits, Fundamental Concepts of Tk, Tk by example, Events and Binding,Perl-Tk. TCL: TCL stands for “Tool Command Language” and is pronounced “tickle”; is a simple scripting language for controlling and extending applications. TCL is a radically simple open-source interpreted programming language that provides common facilities such as variables, procedures, and control structures as well as many useful features that are not found in any other major language. TCL runs on almost all modern operating systems such as Unix, Macintosh, and Windows (including Windows Mobile). While TCL is flexible enough to be used in almost any application imaginable, it does excel in a few key areas, including: automated interaction with external programs, embedding as a library into application programs, language design, and general scripting. TCL was created in 1988 by John Ousterhout and is distributed under a BSD style license (which allows you everything GPL does, plus closing your source code). The current stable version, in February 2008, is 8.5.1 (8.4.18 in the older 8.4 branch). The first major GUI extension that works with TCL is TK, a toolkit that aims to rapid GUI development. That is why TCL is now more commonly called TCL/TK. The language features far-reaching introspection, and the syntax, while simple², is very different from the Fortran/Algol/C++/Java world. Although TCL is a string based language there are quite a few object-oriented extensions for it like Snit³, incr Tcl⁴, and XOTcl⁵ to name a few. TCL is embeddable: its interpreter is implemented as a library of C

procedures that can easily be incorporated into applications, and each application can extend the core TCL features with additional commands specific to that application. Tcl was originally developed as a reusable command language for experimental computer aided design (CAD) tools. The interpreter is implemented as a C library that could be linked into any application. It is very easy to add new functions to the TCL interpreter, so it is an ideal reusable "macro language" that can be integrated into many applications. However, TCL is a programming language in its own right, which can be roughly described as a cross-breed between – LISP/Scheme (mainly for its tail-recursion capabilities) – C (control structure keywords, expr syntax) and – Unix shells (but with more powerful structuring).

117 | Page

TCL Structure The TCL language has a tiny syntax - there is only a single command structure, and a set of rules to determine how to interpret the commands. Other languages have special syntaxes for control structures (if, while, repeat...) - not so in TCL. All such structures are implemented as commands. There is a runtime library of compiled 'C' routines, and the 'level' of the GUI interface is quite high. Comments: If the first character of a command is #, it is a comment. TCL commands: TCL commands are just words separated by spaces. Commands return strings, and arguments are just further words. command argument command argument Spaces are important expr 5*3 has a single argument expr 5 * 3 has three arguments TCL commands are separated by a new line, or a semicolon, and arrays are indexed by text set a(a\ text\ index) 4 Syntax Syntax is just the rules how a language is structured. A simple syntax of English could say (Ignoring punctuation for the moment) A text consists of one or more sentences A sentence consists of one or more words' Simple as this is, it also describes Tcl's syntax very well - if you say "script" for "text", and "command" for "sentence". There's also the difference that a Tcl word can again contain a script or a command. So if { $x < 0$ } {set x 0} is a

command consisting of three words: if, a condition in braces, a command (also consisting of three words) in braces. Take this for example is a well-formed Tcl command: it calls Take (which must have been defined before) with the three arguments "this", "for", and "example". It is up to the command how it interprets its arguments, e.g. puts acos(-1) will write the string "acos(-1)" to the stdout channel, and return the empty string "", while expr acos(-1) will compute the arc cosine of -1 and return 3.14159265359 (an approximation of Pi), or string length acos(-1) will invoke the string command, which again dispatches to its length sub-command, which determines the length of the second argument and returns 8. A Tcl script is a string that is a sequence of commands, separated by newlines or semicolons. A command is a string that is a list of words, separated by blanks. The first word is the name of the command; the other words are passed to it as its arguments. In Tcl, "everything is a command" - even what in other languages would be called declaration, definition, or control structure. A command can interpret its arguments in any way it wants - in particular, it can implement a different language, like expression. A word is a string that is a simple word, or one that begins with { and ends with the matching } (braces), or one that begins with " and ends with the matching ". Braced words are not evaluated by the parser. In quoted words, substitutions can occur before the command is called: \${A-Za-z0-9_}+ substitutes the value of the given variable. Or, if the variable name contains characters outside that regular expression, another layer of bracing helps the parser to get it right puts "Guten Morgen, \${Schuler}!" If the code would say \$Schuler, this would be parsed as the value of variable \$Sch, immediately followed by the constant string üler. (Part of) a word can be an embedded script: a string in [] brackets whose contents are evaluated as a script (see above) before the current command is called. In short: Scripts and commands contain words. Words can again contain scripts and commands. (This can lead to words more than a page

long...) Arithmetic and logic expressions are not part of the Tcl language itself, but the language of the `expr` command (also used in some arguments of the `if`, `for`, `while` commands) is basically equivalent to C's expressions, with infix operators and functions.

Rules of TCL The following rules define the syntax and semantics of the Tcl language:

- (1) **Commands** A Tcl script is a string containing one or more commands. Semi-colons and newlines are command separators unless quoted as described below. Close brackets are command terminators during command substitution (see below) unless quoted.
- (2) **Evaluation** A command is evaluated in two steps. First, the Tcl interpreter breaks the command into words and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a command procedure to carry out the command, then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently.
- (3) **Words of a command** are separated by white space (except for newlines, which are command separators).
- (4) **Double quotes** If the first character of a word is double-quote (") then the word is terminated by the next double-quote character. If semi-colons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double-quotes are not retained as part of the word.
- (5) **Braces** If the first character of a word is an open brace ({) then the word is terminated by the matching close brace (}). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching

close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semi-colons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves. (6) Command substitution If a word contains an open bracket ([]) then Tcl performs command substitution. To do this it invokes the Tcl interpreter recursively to process the characters following the open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket (]). The result of the script (i.e. the result of its last command) is substituted into the word in place of the brackets and all of the characters between them. There may be any number of command substitutions in a single word. Command substitution is not performed on words enclosed in braces. (7) Variable substitution If a word contains a dollar-sign (\$) then Tcl performs variable substitution: the dollar-sign and the following characters are replaced in the word by the value of a variable. Variable substitution may take any of the following forms: \$name Tcl: the Tool Command language Name is the name of a scalar variable; the name is a sequence of one or more characters that are a letter, digit, underscore, or namespace separators (two or more colons). \$name(index) Name gives the name of an array variable and index gives the name of an element within that array. Name must contain only letters, digits, underscores, and namespace separators, and may be an empty string. 120 | Page Command substitutions, variable substitutions, and backslash substitutions are performed on the characters of index. \${name} Name is the name of a scalar variable. It may contain any characters whatsoever except for close braces. There may be any number of variable substitutions in a single word. Variable substitution is not performed on words enclosed in braces. (8) Backslash substitution If a backslash (\) appears within a word then backslash substitution occurs. In all cases but those

described below the backslash is dropped and the following character is treated as an ordinary character and included in the word. This allows characters such as double quotes, close brackets, and dollar signs to be included in words without triggering special processing. The following table lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

`\a` Audible alert (bell) (0x7). `\b` Backspace (0x8). `\f` Form feed (0xc). `\n` Newline (0xa). `\r` Carriage-return (0xd). `\t` Tab (0x9). `\v` Vertical tab (0xb). `\whitespace` A single space character replaces the backslash, newline, and all spaces and tabs after the newline. This backslash sequence is unique in that it is replaced in a separate pre-pass before the command is actually parsed. This means that it will be replaced even when it occurs between braces, and the resulting space will be treated as a word separator if it isn't in braces or quotes.

121 | Page Contents Literal backslash (`\`), no special effect.

`\ooo` The digits `ooo` (one, two, or three of them) give an eight-bit octal value for the Unicode character that will be inserted. The upper bits of the Unicode character will be 0.

`\xhh` The hexadecimal digits `hh` give an eight-bit hexadecimal value for the Unicode character that will be inserted. Any number of hexadecimal digits may be present; however, all but the last two are ignored (the result is always a one-byte quantity). The upper bits of the Unicode character will be 0.

`\uhhhh` The hexadecimal digits `hhhh` (one, two, three, or four of them) give a sixteen-bit hexadecimal value for the Unicode character that will be inserted.

Backslash substitution is not performed on words enclosed in braces, except for backslash newline as described above.

(9) Comments If a hash character (`#`) appears at a point where Tcl is expecting the first character of the first word of a command, then the hash character and the characters that follow it, up through the next newline, are treated as a comment and ignored. The comment character only has significance when it appears at the beginning of a command.

(10) Order of substitution Each character is processed exactly once by the Tcl

interpreter as part of creating the words of a command. For example, if variable substitution occurs then no further substitutions are performed on the value of the variable; the value is inserted into the word verbatim. If command substitution occurs then the nested command is processed entirely by the recursive call to the Tcl interpreter; no substitutions are performed before making the recursive call and no additional substitutions are performed on the result of the nested script. Substitutions take place from left to right, and each substitution is evaluated completely before attempting to evaluate the next. Thus, a sequence like `set y [set x 0][incr x][incr x]` will always set the variable `y` to the value, 012. (11) Substitution and word boundaries Substitutions do not affect the word boundaries of a command. For example, during variable substitution the entire value of the variable becomes part of a single word, even if the variable's value contains spaces. 122 | Page

Variables and Data in TCL As noted above, by default, variables defined inside a procedure are "local" to that procedure. And, the argument variables of the procedure contain local "copies" of the argument data used to invoke the procedure. These local variables cannot be seen elsewhere in the script, and they only exist while the procedure is being executed. In the "getAvg" procedure above, the local variables created in the procedure are "n" "r" and "avg". TCL provides two commands to change the scope of a variable inside a procedure, the "global" command and the "upvar" command. The "global" command is used to declare that one or more variables are not local to any procedure. The value of a global variable will persist until it is explicitly changed. So, a variable which is declared with the "global" command can be seen and changed from inside any procedure which also declares that variable with the "global" command. Variables which are defined outside of any procedure are automatically global by default. The TCL "global" command declares that references to a given variable should be global rather than local. However, the "global" command does not create or set the variable ... this

must be done by other means, most commonly by the TCL "set" command. For example, here is an adjusted version of our averaging procedure which saves the input list length in the global variable "currentLength" so that other parts of the script can access this information after "getAvgN" is called: `proc getAvgN { rList } \ { global currentLength set currentLength [llength $rList] if {!$currentLength} {return 0.0} set avg 0.0 foreach r $rList \ { set avg [expr $avg + $r] } set avg [expr $avg/double($currentLength)] return $avg }` Then, this adjusted version "getAvgN" could be used elsewhere as follows `global currentLength set thisList "1.0 2.0 3.0" set a [getAvgN $thisList] puts "List: $thisList Length: $currentLength Avg: $a"` We can also use global variables as an alternative to procedure arguments. For example, we can make a version of our averaging application which assumes that the input list is stored in a global variable called "currentList" `proc getCurrentAvg {} \ { global currentList currentLength set currentLength [llength $rList] if {!$currentLength} {return 0.0} set avg 0.0 foreach r $currentList \ { set avg [expr $avg + $r] } set avg [expr $avg/double($currentLength)] return $avg }` Then, this adjusted version "getCurrentAvg" could be used elsewhere as follows `global currentList currentLength set currentList "1.0 2.0 3.0" set a [getCurrentAvg] puts "List: $currentList Len: $currentLength Avg: $a"` A procedure can use global variables for persistent storage of information, including the possibility to test whether the procedure has been called previously; this is useful for procedures that might need to perform a one-time initialization. In these cases, a procedure will use a global variable which is not set anywhere else. This means, the first time the procedure is called, the global variable will not yet exist (recall that the "global" statement declares that a variable will be accessed as a global variable, but it does not define or create the variable itself). The TCL command "info exists" will evaluate to true if the given variable exists. For example, suppose we wanted to make a version of our procedure

"getAvg" which keeps an internal count of how many times it has been called. In this version, we use a global variable named "callCount_getAvg" to keep track of the number of times "getAvg" is called. Because this global variable will actually be used to store information for the specific use of the "getAvg" procedure, we need to choose a global variable name which will not be used for a similar purpose in some other procedure. The first time "getAvg" is called, the global variable does not yet exist, and must be set to zero.

```
proc getAvg { rList } \ { global callCount_getAvg if
{![info exists callCount_getAvg]} \ { set callCount_getAvg 0 } incr
callCount_getAvg puts "getAvg has been called $callCount_getAvg times"
set n [length $rList] if {!$n} {return 0.0} set avg 0.0 foreach r $rList \ { set
avg [expr $avg + $r] } set avg [expr $avg/double($n)] return $avg }
```

A more flexible way to manipulate persistent data is to use global arrays rather than scalar variables. For example, instead of the procedure-specific scalar variable "callCount_getAvg" used above, we can use a general-purpose array "callCount()" which could be used to record the call counts of any number of procedures, by using the procedure name as the array index. Many nmrWish TCL scripts use global arrays in this fashion, to simplify the sharing of many data values between procedures. Here is a version of the "getAvg" procedure with the call count tallied in a global array location ... note that an array is declared global simply by listing its name in a "global" command, exactly as for a scalar variable; no () parenthesis or index values are used.

```
proc getAvg { rList } \ { global callCount if {![info exists callCount(getAvg)]} \ { set
callCount(getAvg) 0 } incr callCount(getAvg) puts "getAvg has been used
$callCount(getAvg) times" set n [length $rList] if {!$n} {return 0.0} set avg
0.0 foreach r $rList \ { set avg [expr $avg + $r] } set avg [expr
$avg/double($n)] return $avg }
```

TCL Variable Scope and the upvar Command We have already seen that TCL procedures can generate a return value as a way to pass information back to their caller. And, we have also seen that global variables can be used to share information

between parts of a TCL script, and so these also serve as a mechanism for returning information to a caller. TCL includes the "upvar" command as a method for a given procedure to change the values of variables in the scope of its caller. This provides a way for a procedure to provide additional information to the caller, besides by using the procedure's return value. 126 | Page In the "upvar" scheme, a procedure's caller provides the names of one or more of its own variables as arguments to the procedure. The procedure then uses the "upvar" command to map these variables from the caller onto variables in the procedure. For example, here the caller passes its variable name "count" as the first argument to procedure "getNAvg":

```
set count 0
set a [getNAvg count "1.0 2.0 3.0 4.0"]
```

Then, in this version of procedure "getNAvg" the "upvar" command is used to map the first argument value "\$nPtr" onto the procedure's variable called "n" ... this means that whenever the procedure gets or changes the value of variable "n" it will actually be using the caller's variable "count".

```
proc getNAvg { nPtr rList } {
    upvar $nPtr n
    set n [llength $rList]
    if { !$n } { return 0.0 }
    set avg 0.0
    foreach r $rList {
        set avg [expr $avg + $r]
    }
    set avg [expr $avg/double($n)]
    return $avg
}
```

Control Flow: In Tcl language there are several commands that are used to alter the flow of a program. When a program is run, its commands are executed from the top of the source file to the bottom. One by one. This flow can be altered by specific commands. Commands can be executed multiple times. Some commands are conditional. They are executed only if a specific condition is met. The if command

The if command has the following general form:

```
if expr1 ?then? body1 elseif
expr2 ?then? body2 elseif ... ?else? ?bodyN?
```

The if command is used to check if an expression is true. If it is true, a body of command(s) is then executed. The body is enclosed by curly brackets.

127 | Page

The if command evaluates an expression. The expression must return a boolean value. In Tcl, 1, yes, true mean true and 0, no, false mean false. In the above example, the body enclosed by { } characters is always

executed. The then command is optional. We can use it if we think, it will make the code more clear. We can use the else command to create a simple branch. If the expression inside the square brackets following the if command evaluates to false, the command following the else command is automatically executed. We have a sex variable. It has "female" string. The Boolean expression evaluates to false and we get "It is a girl" in the console.

```
#!/usr/bin/tclsh if yes { puts "This message is always shown" }
#!/usr/bin/tclsh if true then { puts "This message is always shown" }
#!/usr/bin/tclsh set sex female if {$sex == "male"} { puts "It is a boy" }
else { puts "It is a girl" } 128 | P a g e $ ./girlboy.tcl It is a girl
```

We can create multiple branches using the elseif command. The elseif command tests for another condition, if and only if the previous condition was not met. Note that we can use multiple elseif commands in our tests. In the above script we have a prompt to enter a value. We test the value if it is a negative number or positive or if it equals to zero. If the first expression evaluates to false, the second expression is evaluated. If the previous conditions were not met, then the body following the else commands would be executed.

```
$ ./nums.tcl Enter a number: 2 the number is positive
$ ./nums.tcl Enter a number: 0 #!/usr/bin/tclsh # nums.tcl
puts -nonewline "Enter a number: " flush stdout set a [gets stdin] if {$a < 0} { puts "the number is negative" } elseif { $a == 0 } { puts "the number is zero" } else { puts "the number is positive" } 129 | P a g e
the number is zero
$ ./nums.tcl Enter a number: -3 the number is negative
```

Running the example multiple times. Switch command The switch command matches its string argument against each of the pattern arguments in order. As soon as it finds a pattern that matches the string it evaluates the following body argument by passing it recursively to the Tcl interpreter and returns the result of that evaluation. If the last pattern argument is default then it matches anything. If no pattern argument matches string and no default is given, then the switch command returns an empty string. In our script, we prompt for a domain name.

There are several options. If the value equals for example to us the "United States" string is printed to the console. If the value does not match to any given value, the default body is executed and unknown is printed to the console. \$./switch_cmd.tcl Select a top level domain name:sk Slovakia #!/usr/bin/tclsh # switch_cmd.tcl puts -nonewline "Select a top level domain name:" flush stdout gets stdin domain switch \$domain { us { puts "United States" } de { puts Germany } sk { puts Slovakia } hu { puts Hungary } default { puts "unknown" } } 130 | Page

We have entered sk string to the console and the program responded with Slovakia. While command: The while command is a control flow command that allows code to be executed repeatedly based on a given Boolean condition. The while command executes the commands inside the block enclosed by curly brackets. The commands are executed each time the expression is evaluated to true. In the code example, we calculate the sum of values from a range of numbers. The while loop has three parts: initialization, testing, and updating. Each execution of the command is called a cycle. set i 0 We initiate the i variable. It is used as a counter. The expression inside the curly brackets following the while command is the second phase, the testing. The commands in the body are executed, until the expression is evaluated to false. incr i #!/usr/bin/tclsh # whileloop.tcl set i 0 set sum 0 while { \$i < 10 } { incr i incr sum \$i } puts \$sum while { \$i < 10 } { ... } 131 | Page

The last, third phase of the while loop is the updating. The counter is incremented. Note that improper handling of the while loops may lead to endless cycles. FOR command: When the number of cycles is known before the loop is initiated, we can use the for command. In this construct we declare a counter variable, which is automatically increased or decreased in value during each repetition of the loop. In this example, we print numbers 0..9 to the console. There are three phases. First, we initiate the counter i to zero. This phase is done only once. Next comes the condition. If the condition is met, the command inside the for block is executed. Then

comes the third phase; the counter is increased. Now we repeat phases 2 and 3 until the condition is not met and the for loop is left. In our case, when the counter `i` is equal to 10, the for loop stops executing. \$
./forloop.tcl 0 1 2 3 4 5 #!/usr/bin/tclsh for {set i 0} {\$i < 10} {incr i} { puts \$i } for {set i 0} {\$i < 10} {incr i} { puts \$i } 132 | Page 6789 Here we see the output of the forloop.tcl script. The foreach command: The foreach command simplifies traversing over collections of data. It has no explicit counter. It goes through a list element by element and the current value is copied to a variable defined in the construct. In this example, we use the foreach command to go through a list of planets. \$
./planets.tcl Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune #!/usr/bin/tclsh set planets { Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune } foreach planet \$planets { puts \$planet } foreach planet \$planets { puts \$planet } The usage of the foreach command is straightforward. The planets is the list that we iterate through. The planet is the temporary variable that has the current value from the list. The for each command goes through all the planets and prints them to the console. 133 | Page Running the above Tcl script gives this output. In this script, we iterate through pairs of values of a list. \$./actresses.tcl Rachel Weiss Scarlett Johansson Jessica Alba Marion Cotillard Jennifer Connelly This is the output of actresses tcl script #!/usr/bin/tclsh set actresses { Rachel Weiss Scarlett Johansson Jessica Alba \ Marion Cotillard Jennifer Connelly } foreach {first second} \$actresses { puts "\$first \$second" } foreach {first second} \$actresses { puts "\$first \$second" } We pick two values from the list at each iteration. #!/usr/bin/tclsh foreach i { one two three } item {car coins rocks} { puts "\$i \$item" } 134 | Page We can iterate over two lists in parallel. \$./parallel.tcl one car two coins three rocks This is the output of the parallel.tcl script. The break and continue commands: The break command can be used to terminate a block defined by while, for, or switch commands. We define an endless while loop. We use the break command to get out of this loop. We choose

a random value from 1 to 30 and print it. If the value equals to 22, we finish the endless while loop. `set r [expr 1 + round(rand()*30)]` Here we calculate a random number between 1..30. The `rand()` is a built-in Tcl procedure. It returns a random number from 0 to 0.99999. The `rand()*30` returns a random number between 0 to 29.99999. The `round()` procedure rounds the final number.
\$./breakcommand.tcl 28 20 8 8 12 22 .We might get something like this. The `continue` command is used to skip a part of the loop and continue with the next iteration of the loop. It can be used in combination with `for` and `while` commands. In the following example, we will print a list of numbers that cannot be divided by 2 without a remainder. `#!/usr/bin/tclsh while true { set r [expr 1 + round(rand()*30)] puts -nonewline "$r " if {$r == 22} { break } }` puts "" 135 | P a g e We iterate through numbers 1..99 with the while loop. `if {$num % 2 == 0} { continue }` If the expression `num % 2` returns 0, the number in question can be divided by 2. The `continue` command is executed and the rest of the cycle is skipped. In our case, the last command of the loop is skipped and the number is not printed to the console. The next iteration is started. Data Structures The list is the basic Tcl data structure. A list is simply an ordered collection of stuff; numbers, words, strings, or other lists. Even commands in Tcl are just lists in which the first list entry is the name of a proc, and subsequent members of the list are the arguments to the proc. Lists can be created in several way by setting a variable to be a list of values `set lst {item 1 item 2 item 3}` with the `split` command `set lst [split "item 1.item 2.item 3" "."]` with the `list` command. `set lst [list "item 1" "item 2" "item 3"]` An individual list member can be accessed with the `index` command. The brief description of these commands is `list ?arg1? ?arg2? ... ?argN?` makes a list of the arguments `split string ?splitChars?` Splits the string into a list of items wherever the `splitChars` occur in the `#!/usr/bin/tclsh set num 0 while { $num < 100 } { incr num if {$num % 2 == 0} { continue } puts "$num " }` puts "" 136 | P a g e code. `SplitChars` defaults to being whitespace. Note

that if there are two or more `splitChars` then each one will be used individually to split the string. In other words: `split "1234567" "36"` would return the following list: `{12 45 7}`. `list index` Returns the index'th item from the list. Note: lists start from 0, not 1, so the first item is at index 0, the second item is at index 1, and so on. `length list`. Returns the number of elements in a list. The items in list can be iterated through using the `foreach` command. `foreach varname list body` The `foreach` command will execute the body code one time for each list item in list. On each pass, `varname` will contain the value of the next list item. In reality, the above form of `foreach` is the simple form, but the command is quite powerful. It will allow you to take more than one variable at a time from the list: `foreach {a b} $listofpairs { ... }`. You can even take a variable at a time from multiple lists! For example: `foreach a $listOfA b $listOfB { ... }` Examples Adding and deleting members of a list The commands for adding and deleting list members are `concat ?arg1 arg2 ... argn?` Concatenates the args into a single list. It also eliminates leading and trailing spaces in the args and adds a single separator space between args. The args to `concat` may be either individual elements, or lists. If an arg is already a list, the contents of that list is concatenated set `x "a b c"` puts "Item at index 2 of the list {`$x`} is: [`lindex $x 2`]\n" set `y [split 7/4/1776 "/"]` puts "We celebrate on the [`lindex $y 1`]'th day of the [`lindex $y 0`]'th month\n" set `z [list puts "arg 2 is $y"]` puts "A command resembles: `$z\n`" set `i 0` `foreach j $x { puts "$j is item number $i in list x" incr i }` 137 | Page with the other args. `lappend list Name ?arg1 arg2 ... argn?` Appends the args to the list `listName` treating each arg as a list element. `linsert list Name index arg1 ?arg2 ... argn?` Returns a new list with the new list elements inserted just before the index th element of `listName`. Each element argument will become a separate element of the new list. If `index` is less than or equal to zero, then the new elements are inserted at the beginning of the list. If `index` has the value `end` , or if it is greater than or equal to the number of elements in the list, then the new

elements are appended to the list. `lreplace list Name first last ?arg1 ... argn?` Returns a new list with `N` elements of `listName` replaced by the `args`. If `first` is less than or equal to 0, `lreplace` starts replacing from the first element of the list. If `first` is greater than the end of the list, or the word end, then `lreplace` behaves like `lappend`. If there are fewer `args` than the number of positions between `first` and `last`, then the positions for which there are no `args` are deleted. `lset varName index newValue` The `lset` command can be used to set elements of a list directly, instead of using `lreplace`. Lists in Tcl are the right data structure to use when you have an arbitrary number of things, and you'd like to access them according to their order in the list. In C, you would use an array. In Tcl, arrays are associated arrays - hash tables, as you'll see in the coming sections. If you want to have a collection of things, and refer to the `N`th thing (give me the 10th element in this group of numbers), or go through them in order via `foreach`. Take a look at the example code, and pay special attention to the way that sets of characters are grouped into single list elements. Example `set b [list a b {c d e} {f {g h}}]` puts "Treated as a list: \$b\n" `set b [split "a b {c d e} {f {g h}}"]` puts "Transformed by split: \$b\n" `set a [concat a b {c d e} {f {g h}}]` puts "Concatated: \$a\n" `lappend a {ij K lm} ; # Note: {ij K lm} is a single element` puts "After lappending: \$a\n" 138 | P a g e More list commands - `lsearch`, `lsort`, `lrange` Lists can be searched with the `lsearch` command, sorted with the `lsort` command, and a range of list entries can be extracted with the `lrange` command. `lsearch list pattern` Searches `list` for an entry that matches `pattern`, and returns the index for the first match, or a -1 if there is no match. By default, `lsearch` uses "glob" patterns for matching. See the section on globbing. `lsort list` Sorts `list` and returns a new list in the sorted order. By default, it sorts the list into alphabetic order. Note that this command returns the sorted list as a result, instead of sorting the list in place. If you have a list in a variable, the way to sort it is like so: `set lst [lsort $lst]` `lrange list first last` Returns a list composed of the

first through last entries in the list. If first is less than or equal to 0, it is treated as the first list element. If last is end or a value greater than the number of elements in the list, it is treated as the end. If first is greater than last then an empty list is returned. Example set b [linsert \$a 3 "1 2 3"] # "1 2 3" is a single element ; puts "After linsert at position 3: \$b\n" set b [lreplace \$b 3 5 "AA" "BB"] puts "After lreplacing 3 positions with 2 values at position 3: \$b\n" set list [list {Washington 1789} {Adams 1797} {Jefferson 1801} \ {Madison 1809} {Monroe 1817} {Adams 1825}] set x [lsearch \$list Washington*] set y [lsearch \$list Madison*] incr x incr y -1 ;# Set range to be not-inclusive 139 | P a g e Input / Output Tcl uses objects called channels to read and write data. The channels can be created using the open or socket command. There are three standard channels available to Tcl scripts without explicitly creating them. They are stdin, stdout and stderr. The standard input, stdin, is used by the scripts to read data. The standard output, stdout, is used by scripts to write data. The standard error, stderr, is used by scripts to write error messages. In the first example, we will work with the puts command. It has the following synopsis: puts ?-nonewline? ?channelId? string The channelId is the channel where we want to write text. The channelId is optional. If not specified, the default stdout is assumed. The puts command writes text to the channel. #!/usr/bin/tclsh puts "Message 1" puts stdout "Message 2" puts stderr "Message 3" set subsetlist [lrange \$list \$x \$y] puts "The following presidents served between Washington and Madison" foreach item \$subsetlist { puts "Starting in [lindex \$item 1]: President [lindex \$item 0] " } set x [lsearch \$list Madison*] set srlst [lsort \$list] set y [lsearch \$srlst Madison*] puts "\n\$x Presidents came before Madison chronologically" puts "\$y Presidents came before Madison alphabetically" puts "Message 1" puts stdout "Message 2" 140 | P a g e If we do not specify the channelId, we write to stdout by default. This line does the same thing as the previous one. We only have explicitly specified

the channelId. We write to the standard error channel. The error messages go to the terminal by default. \$./printing.tcl Message 1 Message 2 Message 3 Example output. The read command: The read command is used to read data from a channel. The optional argument specifies the number of characters to read. If omitted, the command reads all of the data from the channel up to the end. The script reads a character from the standard input channel and then writes it to the standard output until it encounters the q character. set c [read stdin 1] We read one character from the standard input channel (stdin). while {\$c != "q"} { We continue reading characters until the q is pressed. The gets command The gets command reads the next line from the channel, returns everything in the line up to (but #!/usr/bin/tclsh set c [read stdin 1] while {\$c != "q"} { puts -nonewline "\$c" set c [read stdin 1] } puts stderr "Message 3" 141 | Page not including) the end-of-line character. The script asks for input from the user and then prints a message. The puts command is used to print messages to the terminal. The -no newline option suppresses the new line character. Tcl buffers output internally, so characters written with puts may not appear immediately on the output file or device. The flush command forces the output to appear immediately. The gets command reads a line from a channel. \$./hello.tcl Enter your name: Jan Hello Jan Sample output of the script. The pwd and cd commands Tcl has pwd and cd commands, similar to shell commands. The pwd command returns the current working directory and the cd command is used to change the working directory. #!/usr/bin/tclsh set dir [pwd] puts \$dir cd .. set dir [pwd] puts \$dir #!/usr/bin/tclsh puts -nonewline "Enter your name: " flush stdout set name [gets stdin] puts "Hello \$name" puts -no newline "Enter your name: " flush stdout set name [gets stdin] 142 | Page In this script, we will print the current working directory. Then we change the working directory and print the working directory again. set dir [pwd] The pwd command returns the current working directory. cd .. We change the

working directory to the parent of the current directory. We use the cd command. \$./cwd.tcl /home/janbodnar/prog/tcl/io /home/janbodnar/prog/tcl Sample output. The glob command Tcl has a glob command which returns the names of the files that match a pattern. The script prints all files with the .tcl extension to the console. The glob command returns a list of files that match the *.tcl pattern. We go through the list of files and print each item of the list to the console. \$./globcmd.tcl attributes.tcl #!/usr/bin/tclsh set files [glob *.tcl] foreach file \$files { puts \$file } set files [glob *.tcl] foreach file \$files { puts \$file } 143 | P a g e allfiles.tcl printing.tcl hello.tcl read.tcl files.tcl globcmd.tcl write2file.tcl cwd.tcl readfile.tcl isfile.tcl addnumbers.tcl This is a sample output of the globcmd.tcl script.

Procedures

A procedure is a code block containing a series of commands. Procedures are called functions in many programming languages. It is a good programming practice for procedures to do only one specific task. Procedures bring modularity to programs. The proper use of procedures brings the following advantages

- Reducing duplication of code
- Decomposing complex problems into simpler pieces
- Improving clarity of the code
- Reuse of code
- Information hiding

There are two basic types of procedures: built-in procedures and user defined ones. The built-in procedures are part of the Tcl core language. For instance, the rand(), sin() and exp() are built-in procedures. The user defined procedures are procedures created with the proc keyword. The proc keyword is used to create new Tcl commands. The term procedures and commands are often used interchangeably. We start with a simple example.


```
#!/usr/bin/tclsh
proc tclver {} {
  set v [info tclversion]
  puts "This is Tcl version $v"
}
tclve
```

In this script, we create a simple tclver procedure. The procedure prints the version of Tcl language. proc tclver {}

{
The new procedure is created with the proc command. The {} characters reveal that the procedure takes no arguments.

{
set v [info tclversion]
puts "This is Tcl version \$v"
}
tclve

This is the body of the tclver procedure. It is executed when we execute the tclver command. The body of the command lies between the curly brackets. The procedure is called by specifying its name.

\$./version.tcl

This is Tcl version 8.6

Sample output.

Procedure arguments: An argument is a value passed to the procedure. Procedures can take one or more arguments. If procedures work with data, we must pass the data to the procedures. In the following example, we have a procedure which takes one argument.

```
set v [info tclversion] puts "This is Tcl version $v" } tclver set v [info
tclversion] puts "This is Tcl version $v" } tclver #!/usr/bin/tclsh proc ftc
{f} { return [expr $f * 9 / 5 + 32] } We create a ftc procedure which
transforms Fahrenheit temperature to Celsius temperature. The
```

procedure takes one parameter. Its name `f` will be used in the body of the procedure. We compute the value of the Celsius temperature. The return command returns the value to the caller. If the procedure does not execute an explicit return, then its return value is the value of the last command executed in the procedure's body. The `ftc` procedure is executed. It takes 100 as a parameter. It is the temperature in Fahrenheit. The returned value is used by the `puts` command, which prints it to the console. Output of the example. `$./fahrenheit.tcl 212 32`

86 Next we will have a procedure which takes two arguments. `puts [ftc 100]` `puts [ftc 0]` `puts [ftc 30]` `proc ftc {f} { return [expr $f * 9 / 5 + 32]` `puts [ftc 100]` `#!/usr/bin/tclsh` `proc maximum {x y} { if {$x > $y} { return $x } else { return $y }` 146 | Page The maximum procedure returns the maximum of two values. The method takes two arguments. Here we compute which number is greater. We define two variables which are to be compared. We calculate the maximum of the two variables. This is the output of the `maximum.tcl` script. `$./maximum.tcl` The max of 23, 32 is 32

Variable number of arguments A procedure can take and process variable number of arguments. For this we use the special arguments and parameter. `#!/usr/bin/tclsh` `} set a 23 set b 32 set val [maximum $a $b]` `puts "The max of $a, $b is $val"` `proc maximum {x y} { if {$x > $y} { return $x } else { return $y }` `set a 23 set b 32 set val [maximum $a $b]` 147 | Page We define a sum procedure which adds up all its arguments. The sum procedure has a special `args` argument. It has a list of all values passed to the procedure. We call the sum procedure three times. In the first case, it takes 4 arguments, in the second case 2, in the last case one. Output of the variable `tcl` script `$./variable.tcl 10 3` `proc sum {args} { set s 0 foreach arg $args { incr s $arg } return $s }` `puts [sum 1 2 3 4]` `puts [sum 1 2]` `puts [sum 4]` `proc sum {args} { foreach arg $args { incr s $arg }` We go through the list and calculate the sum. `puts [sum 1 2 3 4]` `puts [sum 1 2]` `puts [sum 4]`

Implicit arguments The arguments in Tcl procedures may have implicit values. An implicit value is used if no

explicit value is provided. Here we create a power procedure. The procedure has one argument with an implicit value. We can call the procedure with one and two arguments. `#!/usr/bin/tclsh proc power {a {b 2}} { if {$b == 2} { return [expr $a * $a] } set value 1 for {set i 0} {$i`



your path to success...