UNIT - 1

Program:

Definition: A complete set of instructions that can be executed independently by a computer to perform a specific task.

Script:

A set of instructions written in a scripting language designed to automate tasks within another program is called a script.

Example:

Imagine you want to calculate the area of a rectangle.

Program: You could write a complete C++ program that takes the length and width as input, performs the calculation, and displays the result. This program would be compiled and run independently.

Script: You could write a Python script that uses a Python library to perform the calculation. This script would be executed within a Python interpreter or embedded in another program to calculate the area dynamically.

Programming Languages:

Definition: Programming languages are used to create standalone software applications or systems. They involve writing code that is compiled or interpreted to produce executable programs.

Key Points:

- ❖ Used for creating standalone software applications.
- ❖ Code is compiled or interpreted to produce executable programs.

Examples include C, C++, Java, and Swift.

Scripting Languages:

Definition: Scripting languages are used to automate tasks within other software environments. They involve writing scripts that are interpreted and executed line by line at runtime.

Key Points:

- ❖ Used for automating tasks within other software environments.
- ❖ Scripts are interpreted and executed line by line at runtime.

- Often used for tasks such as automation, web development, and system administration.
- Examples include Perl, Ruby, TCL, JavaScript (for client-side scripting), and Bash (for shell scripting).

Ruby Introduction

What is Ruby

Ruby is a dynamic, open source, object oriented and reflective programming language. Ruby is considered similar to Perl and Smalltalk programming languages. It runs on all types of platforms like Windows, Mac OS and all versions of UNIX.

Dynamic: In programming languages, "dynamic" typically refers to a language's ability to execute code and make decisions at runtime, as opposed to compile time. Dynamic languages allow for flexibility and adaptability during program execution. This can include dynamic typing (where variables are not assigned a fixed type and can change during execution), dynamic method invocation (where methods can be called based on runtime conditions), and dynamic code generation (where new code can be created and executed during runtime).

Reflective: Reflective programming, on the other hand, refers to a language's ability to examine and modify its own structure and behavior at runtime. This means that the program can introspect itself, analyze its own code, and modify its behavior or structure dynamically while it is running. Reflective languages provide features such as introspection (examining the properties of objects, classes, or methods at runtime), intercession (modifying the behavior of objects, classes, or methods at runtime), and meta-programming (writing code that generates or modifies other code dynamically).

History of ruby:

Ruby was designed and developed by Yukihiro "Martz" Matsumoto in mid 1990s in Japan.

Why the name Ruby:

The name "Ruby" originated during a chat session between Matsumoto and Keiju Ishitsuka. Two names were selected, "Coral" and "Ruby". Matsumoto chose the later one as it was the birthstone of one of his colleagues.

Features of ruby:

- Object-oriented
- Flexibility

- Expressive feature
- Mixins
- Visual appearance
- Dynamic typing and Duck typing
- Exception handling
- Garbage collector
- Portable
- Keywords
- Statement delimiters
- Variable constants
- Naming conventions
- Keyword arguments
- Method names
- Singleton methods
- Missing method
- Case Sensitive

Hello World Program Execution in different languages:

Java	**Ruby**
class HelloWorld	**puts "Hello World"**
{	
public static void main(String argos[])	
{	
System.out.println("Hello World");	
}	
}	

Java Execution:

javac file_name.java

java file_name

Ruby Execution:

ruby file_name.rb

The structure and Execution of Ruby Programs:

1. Lexical Structure
2. Syntactic Structure
3. File Structure
4. Program Encoding
5. Program Execution

Lexical Structure:

The Ruby interpreter parses a program as a sequence of tokens. Tokens include comments, literals, punctuation, identifiers, and keywords. This section introduces these types of tokens and also includes important information about the characters that comprise the tokens and the whitespace that separates the tokens.

Comments:
Statements that are not executed by the interpreter are called Comments.They are written by a programmer to explain their code so that others who look at the code will understand it in a better way.

Types of Ruby comments:

1. Single line comment
2. Multi line comment

Single line comment:

The Ruby single line comment is used to comment only one line at a time. They are defined with **#** character.

Example:

```
#This is a single line comment.
i = 10  #Here i is a variable.
puts i
```

Multiline comment:

The Ruby multi line comment is used to comment multiple lines at a time. They are defined with =begin at the starting and =end at the end of the line.

```
=begin
   This
   is
   multi line
   comment
=end
```

*Multiline comment is also known as an embedded document.

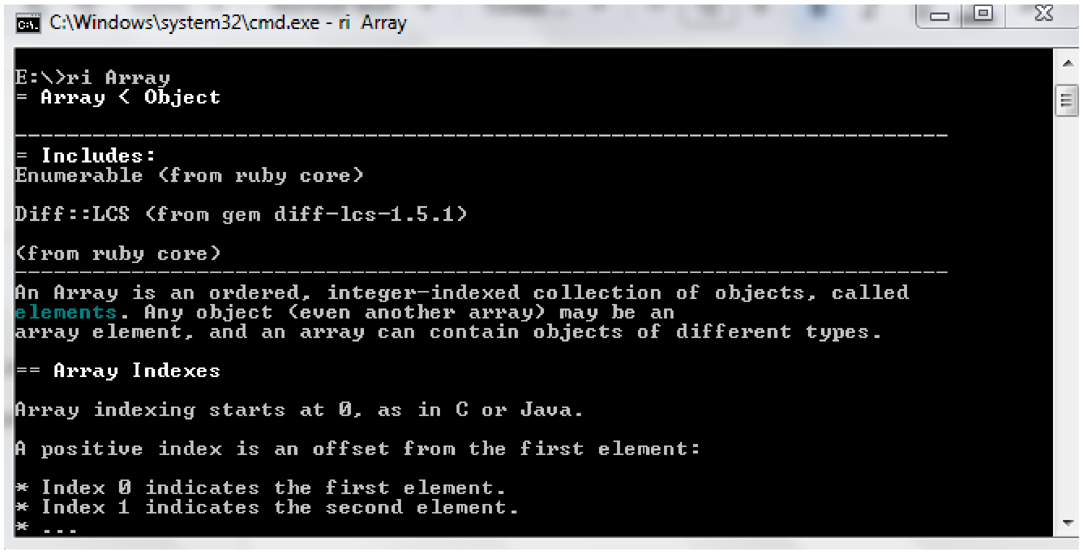
Documentation comments (rDoc and ri):

rDoc stands for Ruby Documentation

rDoc is a tool used in Ruby for generating documentation for Ruby code. It allows developers to document their code in a structured manner using comments, and then automatically generates HTML or other formats of documentation from those comments.

ri stands for Ruby Interactive. It's a command-line tool that comes bundled with Ruby. ri allows you to access documentation for Ruby classes, modules, and methods directly from the command line.

Example for ri:



```
E:\>ri Array
= Array < Object

------------------------------------------------------------------------
= Includes:
Enumerable (from ruby core)

Diff::LCS (from gem diff-lcs-1.5.1)

(from ruby core)
------------------------------------------------------------------------
An Array is an ordered, integer-indexed collection of objects, called
elements. Any object (even another array) may be an
array element, and an array can contain objects of different types.

== Array Indexes

Array indexing starts at 0, as in C or Java.

A positive index is an offset from the first element:

* Index 0 indicates the first element.
* Index 1 indicates the second element.
* ...
```

**To get this , ruby must be installed

Literals:

A literal is any value we specify as a constant in the code.They include numbers, strings of text, and regular expressions etc.

Example:

1	# An integer literal
1.0	# A floating-point literal
'One'	# A string literal
"Two"	# Another string literal
/three/	# A regular expression literal

Punctuation:

Ruby uses punctuation characters for a number of purposes. Most Ruby operators are written using punctuation characters, such as + for addition, * for multiplication, and || for the Boolean OR operation.Punctuation characters also serve to delimit string, regular expression, array, and hash literals, and to group and separate expressions, method arguments, and array indexes.

Identifiers:

An identifier is simply a name. Ruby uses identifiers to name variables, methods, classes etc

- Ruby identifiers consist of letters, numbers, and underscore characters (_).
- Identifiers may not begin with a number.
- Identifiers may begin with _ symbol..
- Identifiers may not include whitespace or nonprinting characters.
- Class and module names must begin with an initial capital letter according to Ruby convention.

Example:

```
i
x2
old_value
_internal                         # Identifiers may begin with underscores
PI                                 # Constant
```

- ❖ Ruby is a case-sensitive language. Lowercase letters and uppercase letters are distinct. The keyword end, for example, is completely different from the keyword END.
- ❖ Punctuation characters may appear at the start and end of Ruby identifiers.

Example:

$ Global variables are prefixed with a dollar sign.

@ Instance variables are prefixed with a single at sign, and class variables are prefixed with two at signs.

? As a helpful convention, methods that return Boolean values often have names that end with a question mark.

! Method names may end with an exclamation point to indicate that they should be used cautiously.

= Methods whose names end with an equals sign can be invoked by placing the method name, without the equals sign, on the left side of an assignment operator.

Keywords / Reserved words:

There are certain words that are reserved for doing specific tasks. These words are known as keywords and they have standard, predefined meaning in Ruby.

Some of the keywords available in Ruby are given below:

BEGIN	END	alias	and	begin	break	case
class	def	module	next	nil	not	or
redo	rescue	retry	return	elsif	end	false
Ensure	for	if	true	undef	unless	do
else	super	then	until	when	while etc….	

These keywords cannot be used for naming variables or constants in Ruby.

Whitespaces:

In Ruby, whitespace refers to any sequence of spaces, tabs, and newline characters in the code that serve to separate tokens.

Ruby statements are typically terminated by a newline character, indicating the end of a line of code

Never put a space between a method name and the opening parenthesis.

Syntactic structure

The expression is the most basic unit of syntax in Ruby. The Ruby interpreter is a programme that evaluates expressions and returns values. Primary expressions, which represent values directly, are the simplest. Primary expressions include number and string literals, which were discussed earlier in this chapter. Some keywords, such as true, false, nil, and self, are also main expressions. Variable references are primary expressions as well; they evaluate the variable's value.

Compound expressions can be used to express more complex values:

```
[1,2,3]  #An Array literal
{1=>"one", 2=>"two"} #A Hash literal
 1..3  #A Range literal
```

To execute computations on values, operators are utilised, and compound expressions are created by combining simpler subexpressions with operators:

```
1 (#A primary expression)
x (#Another primary expression)
x = 1 (#An assignment expression)
x = x + 1 (#An expression with two operators)
```

Statements like the if statement for conditionally executing code and the while statement for continually executing code can be created by combining expressions with Ruby's keywords:

```
if x < 10 then (#If this expression is true)
   x = x + 1 (#Then execute this statement)
 end   (#Marks the end of the conditional)
 while x < 10 do (#While this expression is true)
   print x (#Execute this statement)
   x = x + 1 (#Then execute this statement)
 end
```

Block structure:

Ruby programs have a block structure.
There are two kinds of blocks in Ruby programs. One kind is formally called a “block.” These blocks are the chunks of code associated with or passed to iterator methods:

```
3.times { print "Ruby! " }
```

In this code, the curly braces and the code inside them are the block associated with the iterator method invocation 3.times. Formal blocks of this kind may be delimited with curly braces, or they may be delimited with the keywords do and end:

```
1.upto(10) do |x|
print x
end
```

do and end delimiters are usually used when the block is written on more than one line.
To avoid ambiguity with these true blocks, we can call the other kind of block a body
A body is just the list of statements that comprise the body of a class definition, a method definition, a while loop, or whatever.
Bodies and blocks can be nested within each other, and Ruby programs typically have several levels of nested code, made readable by their relative indentation. Here is a schematic example:

```
module Stats # A module
class Dataset # A class in the module
def initialize(filename) # A method in the class
IO.foreach(filename) do |line| # A block in the method
if line[0,1] == "#" # An if statement in the block
next # A simple statement in the if
end # End the if body
end # End the block
end # End the method body
end # End the class body
end # End the module body
```

File Structure:

There are only a few rules about how a file of Ruby code must be structured.
First, if a Ruby program contains a “shebang” comment, to tell the (Unix-like) operating system how to execute it, that comment must appear on the first line.

```
#!/usr/bin/ruby
```

Using __END__:

❖ If a file contains the line __END__ without any whitespace before or after it, the Ruby interpreter stops processing the file at that point. The rest of the file can contain arbitrary data that the program can read using the IO stream object DATA.

Example:

```
#!/usr/bin/env ruby
# This script calculates the sum of numbers from 1 to 10.
sum = 0
(1..4).each do |n|
  sum += n
end
puts "The sum of numbers from 1 to 4 is: #{sum}"
__END__
This is the end of the Ruby script.
```

To read DATA lines

```
DATA.each_line do |line|
  puts line
```

end

Program encoding:

- ❖ Character encoding is a way to represent characters as numbers in binary format.
- ❖ Different encoding schemes exist, such as ASCII, UTF-8, UTF-16, ISO-8859-1 (Latin-1), etc.
- ❖ Each encoding scheme assigns a unique binary code to each character in its character set.
- ➢ By default, Ruby assumes that source code is encoded in ASCII, but it can process files with other encodings as long as they can represent the full set of ASCII characters.ASCII value for some of the characters are.
- ➢ #(hash)-35,newline-10,space-10
- ➢ In Ruby 1.8, you can specify a different encoding with the -K command-line option. To run a Ruby program that includes Unicode characters encoded in UTF-8, invoke the interpreter with the -Ku option.
- ➢ From ruby 1.9 version ,the encoding can be specified using special comments like # coding: utf-8.
- ➢ Encoding names are not case-sensitive,ASCII-8BIT,US-ASCII (7-bit ASCII),ISO-8859-1,the Japanese encoding SHIFT_JIS (also known as SJIS) and EUC-JP are encoding types.
- ➢ The encoding comment must be entirely in ASCII and must include the string "coding" followed by a colon or equals sign and the name of the desired encoding.

Setting Program Encoding:

- ❖ In Ruby 1.9 and later versions, the preferred way to specify the encoding of a program file is by placing a special "coding comment" at the start of the file.
- ❖ There are different options for setting the encoding via command-line options (-K, -E, --encoding).
- ❖ Default External Encoding:
  - ➢ Ruby also has a default external encoding, which affects how Ruby reads from files and streams.
  - ➢ The default external encoding is typically set based on the system's locale settings.
  - ➢ You can query and set the default external encoding using methods like Encoding.default_external.

Program Execution:
In Ruby, programs are scripts consisting of statements executed sequentially by default. There is no special main method like in statically compiled languages. The Ruby interpreter executes scripts from the first line to the last line, with some exceptions:
BEGIN and END Statements: Before executing the main body of the script, the interpreter scans for BEGIN statements and executes their code. Similarly, after executing the main body, it executes any END statements or registered "shutdown hook" code using the at_exit function.
Module, Class, and Method Definitions: Unlike in compiled languages where these are syntactic structures processed by the compiler, in Ruby, they are executable statements. When encountered, the interpreter executes them, causing the respective modules, classes, and methods to be defined.
Program Invocation: The Ruby interpreter is invoked from the command line, where it reads and executes the specified script file. It continues executing until encountering:

- A statement that causes the program to terminate.
- The end of the file.
- A line with the token __END__, marking the logical end of the file.

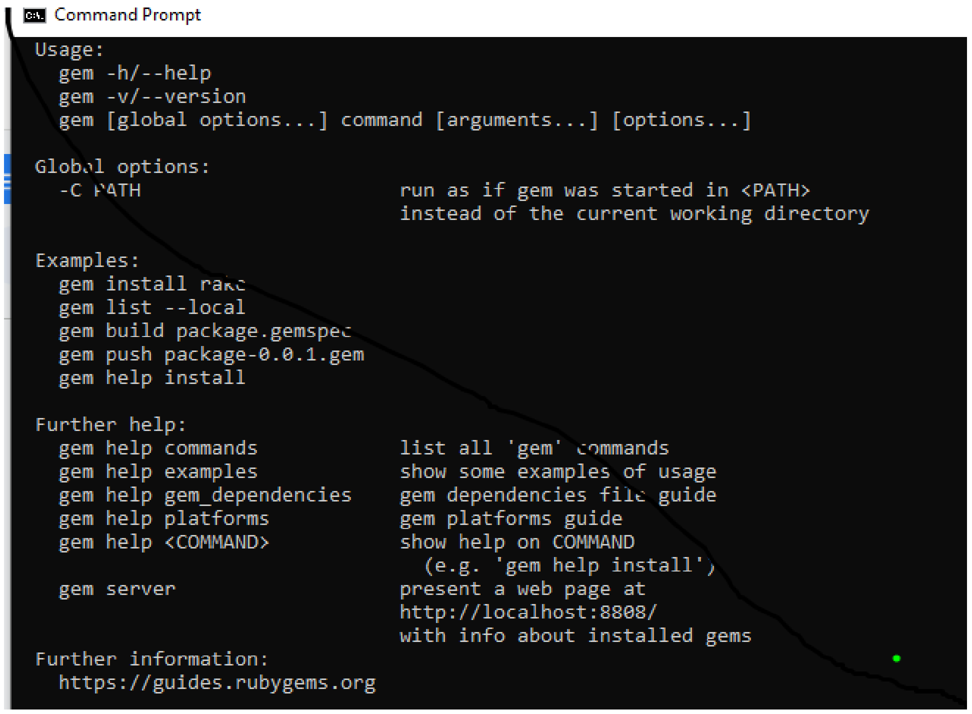
Package Management with RUBYGEMS:

Gems:
In the RubyGems world, developers bundle their applications and libraries into single files called gems.
RubyGems:
RubyGems is a standardized packaging and installation framework for libraries and applications, making it easy to locate, install, upgrade, and uninstall Ruby packages.
or
RubyGems is a big library where you can find and get these packages (gems). It helps to find, install, and manage the gems in Ruby projects.
RubyGems system provides a command-line tool, appropriately named gem, for manipulating these gem files.

Installing RubyGems

when you install Ruby, RubyGems will be automatically installed along with it. This is because RubyGems is the standard package manager for Ruby, and it's essential for managing libraries and dependencies in Ruby projects.
To check whether RubyGem was installed or not ,use the following command in command prompt

e:\ gem help



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

Global options:
  -C PATH                     run as if gem was started in <PATH>
                              instead of the current working directory

Examples:
  gem install rake
  gem list --local
  gem build package.gemspec
  gem push package-0.0.1.gem
  gem help install

Further help:
  gem help commands           list all 'gem' commands
  gem help examples           show some examples of usage
  gem help gem_dependencies   gem dependencies file guide
  gem help platforms          gem platforms guide
  gem help <COMMAND>          show help on COMMAND
                                (e.g. 'gem help install')
  gem server                  present a web page at
                              http://localhost:8808/
                              with info about installed gems
Further information:
  https://guides.rubygems.org
```

Installing Application Gems:

Rake: Rake is used to automate tasks in the development process, such as compiling code, running tests, deploying applications, and more. Rake uses Ruby's language syntax to define tasks and dependencies

Rake typically comes pre-installed with Ruby, so you usually don't need to install it separately. However, if you need to install a specific version or want to ensure you have the latest version, you can install it using RubyGems, the package manager for Ruby:

```
% gem install -r rake
Attempting remote installation of 'Rake'
Successfully installed rake, version 0.4.3
% rake --version
rake, version 0.4.3
```

- ❖ -r means remotely

If for some reason—perhaps because of a potential compatibility issue—you wanted an older version of 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
```

Operator	**Description**
=	Exact version
!=	Any version that is not one mentioned
>	Any version that is greater than the one specified
<	Any version that is less than the one specified
>=	Any version that is greater than or equal to specified
<=	Any version that is less than or equal to specified
~>	Boxed version operator. Version must be greater than or equal to the specified version and less than specified version after having its minor version number increased by one.

Creation of Rakefile and execution:
create a simple Rakefile that defines tasks to greet users in different languages:

To save the Rakefile program, you can follow these steps:

- ❖ Open a text editor or an Integrated Development Environment (IDE) of your choice.
- ❖ Copy the provided Rakefile code into a new file.
- ❖ Save the file with the name Rakefile (note the capital "R" and No file extension).

Write the below code in any editor (Notepad)

```
# Define a task to greet users in English
task :english do
  puts "Hello, World!"
end
# Define a task to greet users in French
task :french do
  puts "Bonjour tout le monde !"
end
# Define a task to greet users in Spanish
task :spanish do
  puts "¡Hola, mundo!"
end
```

After writing the above code save the file with Rakefile(R should be capital and no extension for program)

In this example:

- We've defined three tasks: :english, :french, and :spanish, each using the task method.
- Each task simply prints a greeting message in the respective language.

To run these tasks, you use the rake command followed by the task name:

```
rake english   # To greet users in English
rake french    # To greet users in French
rake spanish
```

Installing and Using Gem Libraries:

Installing predefined gems means adding already-made tools and features to your Ruby projects using RubyGems. These gems include various helpful libraries and utilities that make your Ruby coding easier and more powerful. With RubyGems, you can quickly access a diverse range of components designed to speed up your development process, simplify tasks, and expand what your applications can do. Whether you need to work with JSON data, handle CSV files, connect to databases, or interact with external services, predefined gems provide an easy way to add these capabilities to your projects. All it takes is a few simple commands to install, manage, and use these gems, boosting your Ruby development experience.

- ❖ To see available predefined gems the command used is (Execute commands in cmd)
  gem list
- ❖ To install gem the command used is
  gem install gem_name
- ❖ To see the documentation of the libraries, the command used are
  ri and rdoc

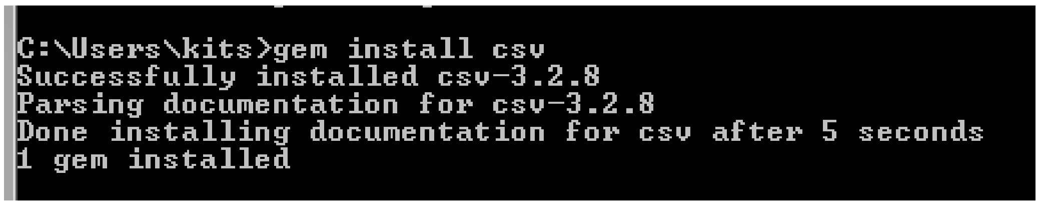
Example:

One commonly used predefined gem in Ruby is the csv gem, which provides functionality for working with comma-separated values (CSV) files.

This simple example demonstrates how to use the csv gem to read data from a CSV file in Ruby. The csv gem provides convenient methods for working with CSV files, making it easy to parse and manipulate CSV data in your Ruby applications.

Step 1: install csv gem using following command

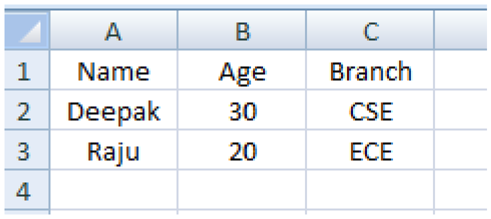
```
c:\Users\kits> gem install csv
```



```
C:\Users\kits>gem install csv
Successfully installed csv-3.2.8
Parsing documentation for csv-3.2.8
Done installing documentation for csv after 5 seconds
1 gem installed
```

Step 2: After successfully installing the csv gem , now we have to use this in our program.

Create a excel file in any directory and type the data like



	A	B	C	
1	Name	Age	Branch	
2	Deepak	30	CSE	
3	Raju	20	ECE	
4				

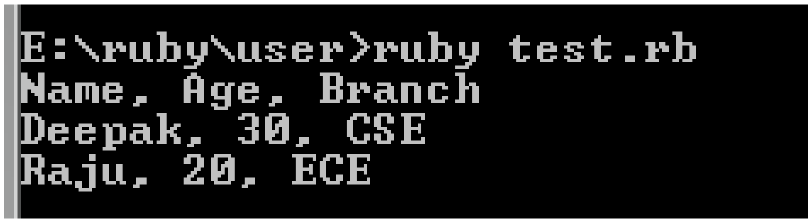
Save the above with .csv extension.

Next, Create a notepad file and type the following code (**csv and this notepad should be in the same directory**)

```
require 'csv'
# Define the path to the CSV file
csv_file = 'example.csv'
# Read data from the CSV file and print it
CSV.foreach(csv_file) do |row|
puts row.join(', ')
end
```

Save this file with test.rb

Next execute the ruby file in command prompt



```
E:\ruby\user>ruby test.rb
Name, Age, Branch
Deepak, 30, CSE
Raju, 20, ECE
```

Creating Your Own Gems:

Setup Environment: Ensure you have Ruby installed on your system. You can check the installation by running **ruby -v** in your terminal. Also, ensure you have RubyGems installed (**gem -v**). If not, install it. If ruby is installed then automatically Rubygems is also installed.

1. Package Layout:

The package layout refers to how your gem's files and directories are organized. In the case of factorial gem, we have a directory structure with lib for the main code, test for testing files, and additional files like README.md and factorial.gemspec for metadata and documentation.

You can use the **bundle gem** command to create a new gem. Open your terminal and navigate to the directory where you want to create your gem.

For Example: In E: (in any directory) drive create a folder, Example ruby(name can be anything)

After creating folder move to that folder

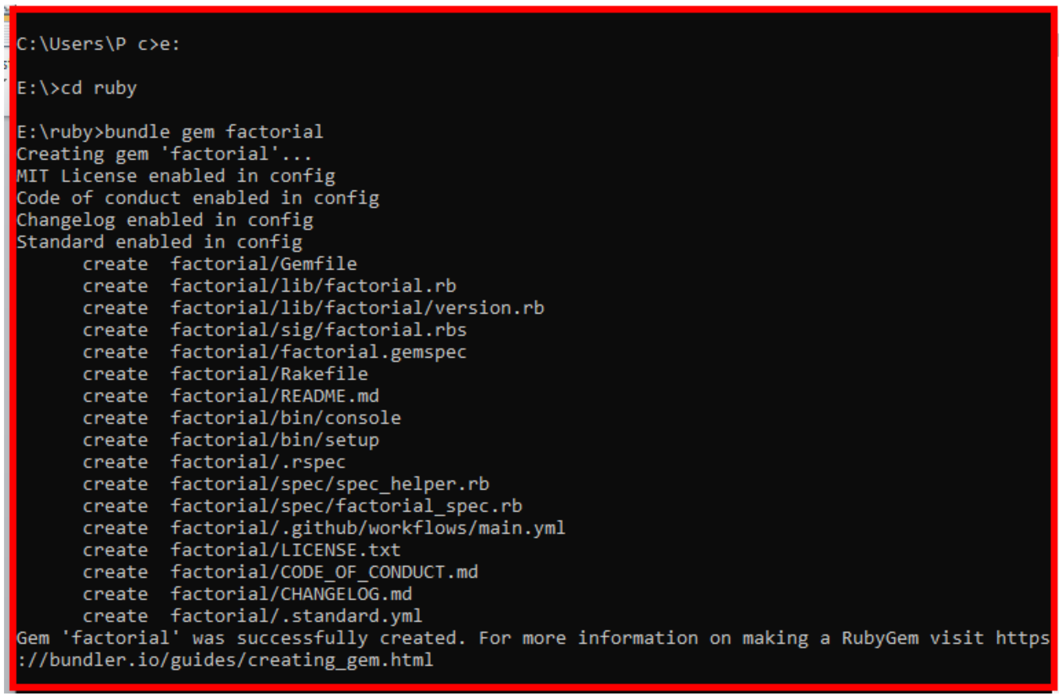
E:/>cd ruby

E:/>ruby

Now execute the command ***bundle gem factorial***

E:\ruby>bundle gem factorial

This will create a directory named factorial with the basic structure for a gem.

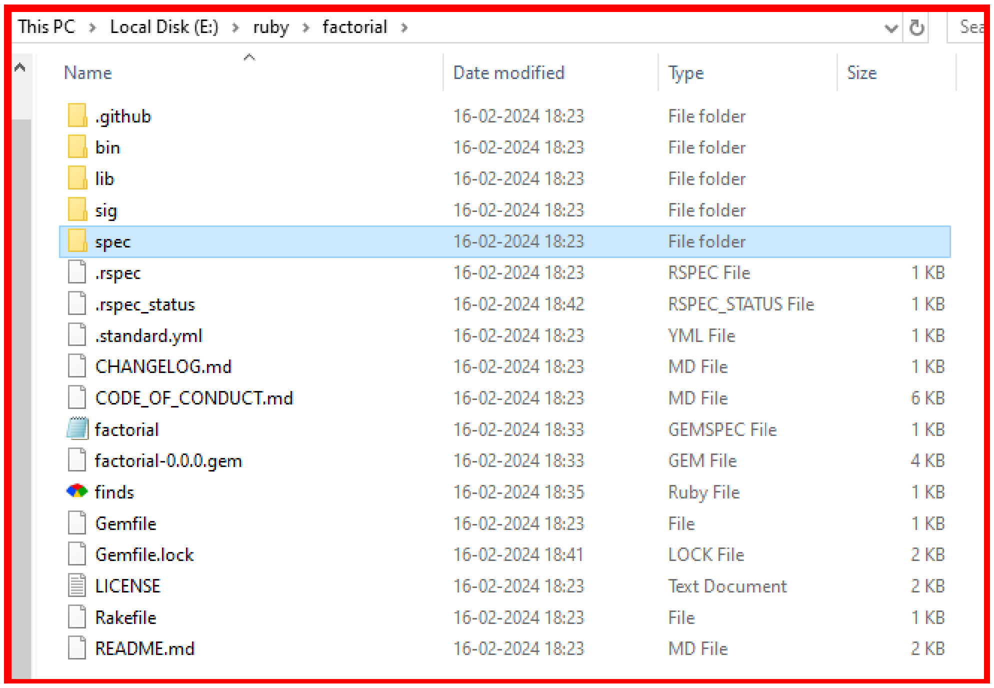


```
C:\Users\P c>e:

E:\>cd ruby

E:\ruby>bundle gem factorial
Creating gem 'factorial'...
MIT License enabled in config
Code of conduct enabled in config
Changelog enabled in config
Standard enabled in config
      create  factorial/Gemfile
      create  factorial/lib/factorial.rb
      create  factorial/lib/factorial/version.rb
      create  factorial/sig/factorial.rbs
      create  factorial/factorial.gemspec
      create  factorial/Rakefile
      create  factorial/README.md
      create  factorial/bin/console
      create  factorial/bin/setup
      create  factorial/.rspec
      create  factorial/spec/spec_helper.rb
      create  factorial/spec/factorial_spec.rb
      create  factorial/.github/workflows/main.yml
      create  factorial/LICENSE.txt
      create  factorial/CODE_OF_CONDUCT.md
      create  factorial/CHANGELOG.md
      create  factorial/.standard.yml
Gem 'factorial' was successfully created. For more information on making a RubyGem visit https
://bundler.io/guides/creating_gem.html
```

Now in E:\ruby a folder is created


This PC › Local Disk (E:) › ruby › factorial ›

Name	Date modified	Type	Size
.github	16-02-2024 18:23	File folder	
bin	16-02-2024 18:23	File folder	
lib	16-02-2024 18:23	File folder	
sig	16-02-2024 18:23	File folder	
spec	16-02-2024 18:23	File folder	
.rspec	16-02-2024 18:23	RSPEC File	1 KB
.rspec_status	16-02-2024 18:42	RSPEC_STATUS File	1 KB
.standard.yml	16-02-2024 18:23	YML File	1 KB
CHANGELOG.md	16-02-2024 18:23	MD File	1 KB
CODE_OF_CONDUCT.md	16-02-2024 18:23	MD File	6 KB
factorial	16-02-2024 18:33	GEMSPEC File	1 KB
factorial-0.0.0.gem	16-02-2024 18:33	GEM File	4 KB
finds	16-02-2024 18:35	Ruby File	1 KB
Gemfile	16-02-2024 18:23	File	1 KB
Gemfile.lock	16-02-2024 18:41	LOCK File	2 KB
LICENSE	16-02-2024 18:23	Text Document	2 KB
Rakefile	16-02-2024 18:23	File	1 KB
README.md	16-02-2024 18:23	MD File	2 KB

2. The Gem Specification:

The gem specification (factorial.gemspec) is a crucial file that defines the metadata for your gem. This includes details such as the gem's name, version, description, author information, and files to include in the gem. It's essentially the blueprint for your gem's construction.

❖ factorial.gemspec file present in e:/ruby/factorial folder, Write the following code in factorial.gemspec file.

```
# frozen_string_literal: true
 require_relative "lib/factorial/version"
Gem::Specification.new do |spec|
 spec.name = "factorial"
 spec.version = "0.0.0"
 spec.authors = "Deepak"
 spec.email = "abc@gmail.com"
 spec.summary = "Factorial program"
 spec.description = "this program is for factorial"
 spec.files      = ["lib/factorial.rb"]
 spec.homepage    ="https://rubygems.org/gems/factorial"
 spec.license = "MIT"
 spec.required_ruby_version = ">= 2.6.0"
   end
```

❖ After writing the above code, save the file and close.

3. Runtime Magic:

In the lib/factorial.rb file, we define the actual functionality of our gem. Here, we implement the calculate method of the Factorial module, which computes the factorial of a given number. This is where the core logic of your gem resides, and it's what users will interact with when they use your gem**.**

❖ Inside the **lib/factorial.rb** file, we have to define the functionality of the gem.
open **factorial.rb** file and remove all content and type the following code

```
# frozen_string_literal: true
require "factorial"
module Factorial
def self.calculate(n)
if n == 0 || n == 1
      1
  else
      n * calculate(n - 1)
  end
  end
  end
```

Then save the file and close the file

4. Adding Tests and Documentation:

Tests and documentation are essential components of any well-maintained gem. In **test/factorial_test.rb**, we write tests to ensure that our **calculate** method behaves as expected under different scenarios. Documentation comments in **lib/factorial.rb** help users understand how to use the gem and what each method does.

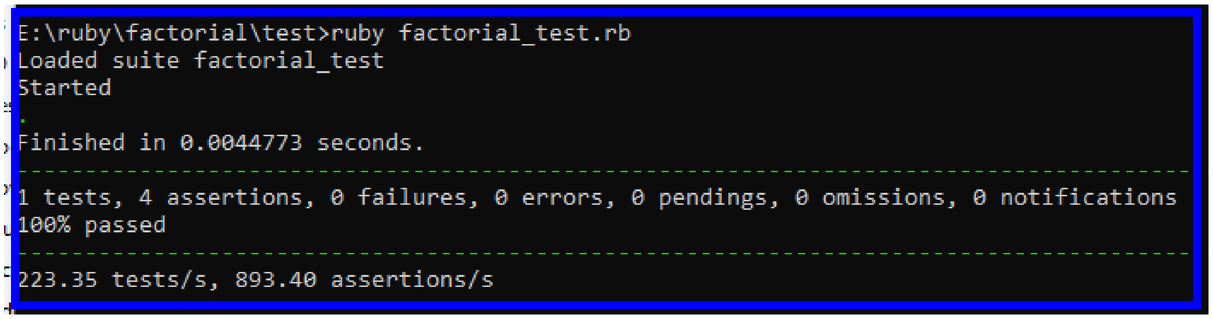
❖ Create a folder ***test*** in e:/ruby/factorial.

❖ Create a notepad file(factorial_test.rb) and type the following code

```
 require 'test/unit'
 require '../lib/factorial'
 class TestFactorial < Test::Unit::TestCase
def test_factorial
assert_equal(1, Factorial.calculate(0))
 assert_equal(1, Factorial.calculate(1))
 assert_equal(120, Factorial.calculate(5))
 assert_equal(3628800, Factorial.calculate(10))
```

end

end



```
E:\ruby\factorial\test>ruby factorial_test.rb
Loaded suite factorial_test
Started
.
Finished in 0.0044773 seconds.
-------------------------------------------------------------------------------
1 tests, 4 assertions, 0 failures, 0 errors, 0 pendings, 0 omissions, 0 notifications
100% passed
-------------------------------------------------------------------------------
223.35 tests/s, 893.40 assertions/s
```

5. Adding Dependencies:

If your gem relies on other gems or libraries, you specify those dependencies in the gemspec. For example, if our factorial gem required the bigdecimal gem for handling large numbers, we would add it as a dependency in the gemspec. This ensures that users can easily install and use your gem along with its dependencies.

To add a dependency on the **bigdecimal** gem in your gemspec file, you would include it like this:

❖ ***spec.add_dependency 'bigdecimal'***

This line tells RubyGems that your gem depends on the bigdecimal gem. When users install your gem, RubyGems will automatically ensure that the bigdecimal gem is also installed if it's not already.

6. Ruby Extension Gems:

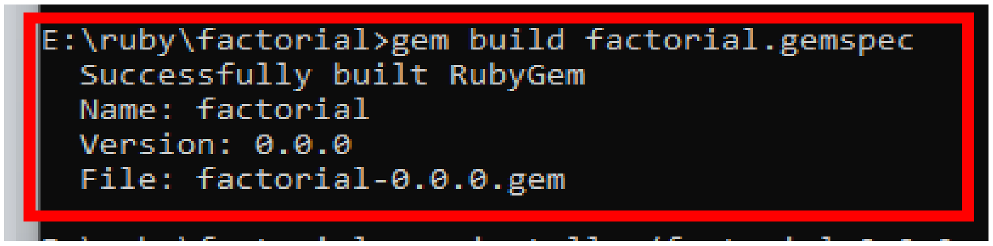
In some cases, your gem might include C extensions or native code for performance reasons or to interface with external libraries. You'd need to handle these extensions properly and specify them in the gemspec to ensure that they're compiled and included correctly when users install your gem.

7. Building the Gem File:

Once you've defined your gemspec and organized your files, you can use the **gem build** command to generate a gem file (**.gem**). This file contains all the necessary files and metadata required to distribute your gem.

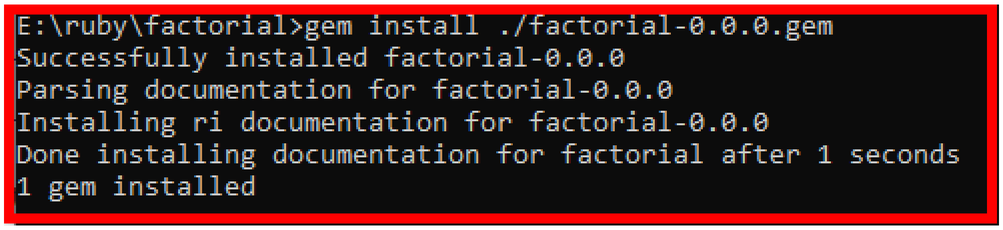
Open cmd(E:\ruby/factorial) and type the following command

gem build factorial.gemspec



```
E:\ruby\factorial>gem build factorial.gemspec
  Successfully built RubyGem
  Name: factorial
  Version: 0.0.0
  File: factorial-0.0.0.gem
```

Install Gem Locally: You can install the gem locally to test it in your system.



```
E:\ruby\factorial>gem install ./factorial-0.0.0.gem
Successfully installed factorial-0.0.0
Parsing documentation for factorial-0.0.0
Installing ri documentation for factorial-0.0.0
Done installing documentation for factorial after 1 seconds
1 gem installed
```

Now we have to use factorial gem

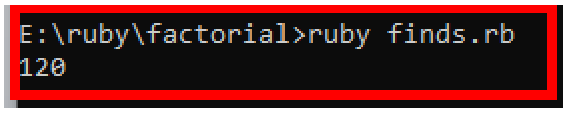
Create a notepad in E:/ruby/factorial and the following code

require "factorial"

puts Factorial.calculate(5)

Save the notepad with .rb extension

Next open cmd and execute the file



```
E:\ruby\factorial>ruby finds.rb
120
```

8. Maintaining Your Gem:

Maintaining your gem involves ongoing tasks such as fixing bugs, adding features, updating documentation, and ensuring compatibility with new versions of Ruby and its dependencies. It also involves responding to user feedback, addressing issues, and keeping your gem's ecosystem healthy and up-to-date.

RUBY AND THE WEB

Writing CGI Scripts:

What are CGI scripts:

CGI, or Common Gateway Interface, is a protocol for web servers to interact with external programs. Ruby provides a built-in library called CGI that makes writing CGI scripts relatively easy. These scripts:

1. Run on the web server and are triggered by user requests (like clicking a button or submitting a form).
2. Access information from the request (like form data, cookies, headers).
3. Generate dynamic content (like HTML pages) to be sent back to the user.

Uses:

1. CGI scripts were used to build small-scale web applications.
2. Scripts could handle form submissions, validate data, and generate responses.
3. They could add dynamic elements to static websites without a full framework.

Different types of CGI scripts:

Form Processing Scripts:
These scripts handle data submitted through HTML forms on web pages. They extract form data from the HTTP request, process it (e.g., validate input, perform calculations), and generate a response (e.g., confirmation message, updated page).

Database Interaction Scripts:
CGI scripts can interact with databases to perform tasks such as retrieving data, updating records, or executing queries. They receive input parameters from users or other sources, process them, interact with the database using SQL queries and return results as HTML or other formats.

File Upload and Management Scripts:
These scripts manage file uploads from users to the web server. They receive files submitted via HTML forms, validate them (e.g., check file types, size limits), and store them on the server filesystem. Additionally, they may handle file retrieval, deletion, or other management tasks.

User Authentication and Authorization Scripts:
CGI scripts can authenticate users and enforce access control policies on web applications. They verify user credentials (e.g., username/password) against a database or other authentication source, set session tokens or cookies for authenticated users, and enforce authorization rules to restrict access to certain resources or actions.

Dynamic Content Generation Scripts:
These scripts dynamically generate web content based on various factors such as user

input, database queries, system status, or external data sources. They construct HTML pages, generate charts or graphs, or customize content based on user preferences or behavior.

Email Handling Scripts:

CGI scripts can send and receive emails as part of web applications. They process email messages received from users or external systems, send email notifications or confirmations, and handle email-related tasks such as formatting messages, adding attachments, or parsing incoming emails for specific content.

Execution of CGI scripts:

Install XAMPP:

- ➢ Download and install XAMPP from the official website: XAMPP.
- ➢ Follow the installation wizard and install it in the desired directory (e.g., **C:\xampp**).

4. **Start Apache Server:**
    - ➢ Launch the XAMPP Control Panel.
    - ➢ Start the Apache server by clicking on the "Start" button next to Apache.
5. **Install Ruby:**
    - ➢ Download RubyInstaller from RubyInstaller for Windows.
    - ➢ Run the installer and follow the installation instructions. Make sure to check the option to add Ruby executables to your PATH during installation.
6. **Configure Apache for CGI:**
    - ➢ Open the XAMPP Control Panel.
    - ➢ Click on the "Config" button for Apache, then select "httpd.conf" to edit the Apache configuration file.
    - ➢ Find the following lines and uncomment (remove the # at the beginning of the lines) or add them if they don't exist:

```
LoadModule cgi_module modules/mod_cgi.so
AddHandler cgi-script .cgi .rb
ScriptInterpreterSource registry
```

Save the changes and close the editor.

Install cgi by using following command ====> **gem install cgi**

7. **Create a CGI Script:**
    - ➢ Open a text editor like Notepad or any code editor of your choice.
    - ➢ Write your CGI script. For example, let's create a simple script named **hello.rb** that outputs a basic HTML page:

```
#!/usr/bin/env ruby
require 'cgi'
```

```
cgi = CGI.new
name = "Deepak"
puts cgi.header
puts "<html>"
puts "<head>"
puts "<title>CGI Ruby Example</title>"
puts "</head>"
puts "<body>"
puts "<h1>Hello, #{name.empty? ? 'Anonymous' : name}!</h1>"
puts "</body>"
puts "</html>"
```

Save the file in the **cgi-bin** directory within your XAMPP installation. For example, you can save it in **C:\xampp\cgi-bin**.

8. **Set Executable Permission:**
   - Right-click on the **hello.rb** file, select "Properties," and then go to the "Security" tab.
   - Click on "Edit" to change permissions, and make sure that "Read & execute" is checked for the appropriate user group (e.g., Everyone).
9. **Access the CGI Script via Web Browser:**
   - Open a web browser and navigate to **http://localhost/cgi-bin/hello.rb**.
   - You should see the output of your CGI script displayed in the browser window.

Example 2:

Create html file in cgi-bin folder and name of the file is index.html

```
<!DOCTYPE html>
<head>
    <title>CGI Ruby Example</title>
</head>
<body>
  <h1>Enter Your Name</h1>
  <form action="name.rb" method="post">
    <label for="name">Name:</label><br>
    <input type="text" id="name" name="name"><br>
```

```
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

Create ruby file same folder i.e in cgi-bin and name it with name.rb

name.rb

```
#!/usr/bin/env ruby
require 'cgi'
cgi = CGI.new
name = cgi['id']
puts cgi.header
puts "<html>"
puts "<head>"
puts "<title>CGI Ruby Example</title>"
puts "</head>"
puts "<body>"
puts "<h1>Hello, #{name.empty? ? 'Anonymous' : name}!</h1>"
puts "</body>"
puts "</html>"
```

Open any browser window and execute the html file **http://localhost/cgi-bin/index.html**

erb:

ERB stands for Embedded RuBy. It's a feature in Ruby that allows you to embed Ruby code within a text document, typically used for generating dynamic content in web applications. ERB is commonly used in web frameworks like Ruby on Rails for generating HTML pages dynamically.

With ERB, you can write plain text documents (often HTML or XML) with snippets of Ruby code embedded within <% %> or <%= %> tags. Here's a brief explanation of the two main types of ERB tags:
<% %> (ERB scriptlet tags): These tags are used to embed Ruby code that doesn't output anything directly to the document. They are typically used for control structures like loops and conditionals.
<%= %> (ERB output tags): These tags are used to embed Ruby code that outputs a value to the document. The result of the Ruby expression inside these tags will be included in the output document.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Welcome</title>
</head>
<body>
  <% if @logged_in %>
    <p>Welcome back, <%= @username %>!</p>
  <% else %>
    <p>Welcome, Guest!</p>
  <% end %>
</body>
</html>
```

Cookies:
cookies are small pieces of data that websites store on your device. They act like memory for websites, enabling them to remember things about you and your browsing activity.
Working of cookies:

1. **Visiting a website:** When you visit a website for the first time, the web server may send your browser a cookie.
2. **Storing the cookie:** Your web browser stores the cookie information on your device, typically in a file.
3. **Subsequent visits:** When you revisit the same website, your browser sends the stored cookie back to the web server.

This exchange of cookies allows websites to:

- ❖ **Maintain user sessions:** Cookies can remember if you're logged in or not, eliminating the need to re-enter credentials every time you visit a page.
- ❖ **Personalize your experience:** Websites can use cookies to tailor content and functionality based on your preferences or past behavior. For instance, an e-commerce site might use cookies to recommend products you might be interested in.
- ❖ **Track user behavior:** Cookies can be used to track what pages you visit and how you interact with a website. This data is often used for analytics purposes or targeted advertising.

Example:

```
require 'cgi'
cgi = CGI.new
# Read the value of the 'username' cookie
username_cookie = cgi.cookies['Deepak']
puts cgi.header
if username_cookie.empty?
  # Create cookie if it doesn't exist
  new_cookie = CGI::Cookie.new('name' => 'Deepak', 'value' => 'Guest', 'expires' => Time.now
+ 3600)  # Expires in 1 hour
  puts new_cookie
  puts "Content-Type: text/html\n\n"
  puts "<html>"
  puts "<body>"
  puts "<h1>Welcome, Guest!</h1>"
  puts "<p>A 'Deepak' cookie has been set.</p>"
  puts "</body>"
  puts "</html>"
else
  # Greet user with stored username if cookie exists
  username = username_cookie.first.value
  puts "Content-Type: text/html\n\n"
  puts "<html>"
  puts "<body>"
  puts "<h1>Welcome back, #{username}!</h1>"
  puts "<p>You have returned to our site.</p>"
  puts "</body>"
  puts "</html>"
end
```

Choice of Web Servers:

WEBrick is a small, simple HTTP server toolkit that comes bundled with Ruby. It's often used for development and testing purposes due to its ease of use and lightweight nature. WEBrick is written in Ruby and provides a basic but functional web server that can serve both static and dynamic content.

Here's a simple example of how you can use WEBrick to create a basic web server in Ruby:

```
require 'webrick'
# Define a class for handling requests
class MyHandler < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
        # Set the response content type
    response.content_type = 'text/plain'
        # Set the response body
    response.body = "Hello, World! You requested: #{request.path}"
        # Send the response
    response.status = 200
  end
end
 # Create a server instance
server = WEBrick::HTTPServer.new(Port: 8000)
 # Mount the handler to respond to requests
server.mount '/', MyHandler
 # Trap signals to gracefully shutdown the server
trap('INT') { server.shutdown }
 # Start the server
server.start
```

We require the 'webrick' library to use the WEBrick server.

We define a class MyHandler that inherits from WEBrick::HTTPServlet::AbstractServlet. This class overrides the do_GET method to handle GET requests. In this method, we set the content type, response body, and status code.
We create a WEBrick::HTTPServer instance on port 8000.
We mount our handler class to the root path '/' so that it handles all requests to the server.
We set up a signal trap to gracefully shutdown the server when a SIGINT signal (e.g., Ctrl+C) is received.
Finally, we start the server using server.start.

You can save this code in a file (e.g., simple_server.rb) and then run it using Ruby. After running the script, open any browser and type the at http://localhost:8000 and see the response "Hello, World! You requested: /".

SOAP Web Services:

Imagine you have two different software programs, each written in a different language. Now, suppose you want these programs to talk to each other and share data. This is where web services come in.

A web service acts like a middleman between these programs. It provides a way for them to communicate over the internet.Types of Web Services: There are different types of web services, but two common ones are:

- ❖ SOAP: This is more structured and formal, often used in enterprise environments.
- ❖ REST: This is simpler and more flexible, often used for simpler interactions over the web.

Example: Let's say you have a weather forecasting program written in Python and a mobile app written in JavaScript. By using a web service, the app can ask the Python program for the current weather data (like temperature and humidity). The Python program processes this request and sends back the weather information in a format that the app understands.

SOAP:

SOAP web services are a way for applications to communicate with each other over the internet. Here's a breakdown of what SOAP stands for and how it works:

- ❖ **SOAP stands for Simple Object Access Protocol.** It's a set of rules that define how messages are formatted and exchanged between applications.
- ❖ **SOAP uses XML (Extensible Markup Language) for data exchange.** XML provides a structured way to represent data, making it understandable by different applications regardless of their programming language.
- ● **Key components in a SOAP web service:**
  - ○ **Service Provider:** This is the application that offers a specific functionality or data. It exposes its services through a SOAP interface.
  - ○ **Service Requester:** This is the application that wants to use the service offered by the provider. It sends SOAP messages to the provider to invoke the service and receive responses.

    - **WSDL (Web Services Description Language):** This is an optional component that acts like a contract between the provider and requester. It describes the service offered by the provider, including the functions available, the expected data format, and the return values.

Here's a typical flow of how a SOAP web service works:

1. The service requester creates a SOAP message containing details about the service it wants to invoke and any relevant data.
2. The SOAP message is sent to the service provider using HTTP (Hypertext Transfer Protocol).
3. The service provider receives the SOAP message, parses the XML data, and executes the requested service.
4. The service provider generates a response SOAP message containing the results of the service execution.
5. The response message is sent back to the service requester using HTTP.
6. The service requester parses the response message and uses the extracted data.

Advantages of SOAP web services:

- **Platform and Language Independence:** SOAP allows applications written in different programming languages and running on various platforms to communicate seamlessly.
- **Security:** SOAP provides mechanisms for implementing security features like authentication and authorization.
- **Standardization:** SOAP is a well-established standard with wide industry support.

Disadvantages of SOAP web services:

- **Complexity:** SOAP messages can be complex and verbose due to the XML structure, leading to increased processing overhead.
- **Performance:** Compared to simpler protocols like REST (Representational State Transfer), SOAP can be slower due to the parsing of XML data.
- **Steeper Learning Curve:** Developing SOAP web services requires understanding the SOAP protocol, XML, and potentially WSDL, which can have a steeper learning curve.

RubyTk – Simple Tk Application

Tk:

In Ruby, Tk is a library that provides bindings to the Tk toolkit, a graphical user interface (GUI) toolkit that originated as part of the Tcl scripting language. Tk allows you to create and manipulate graphical user interfaces in your Ruby applications.

Here are some key points about Tk in Ruby:

GUI Development: Tk enables you to create graphical user interfaces for your Ruby applications. You can create windows, buttons, labels, entry fields, menus, and other GUI elements using Tk.

Cross-Platform: Tk is cross-platform, meaning that Tk-based Ruby applications can run on different operating systems (Windows, macOS, Linux) without modification.

Event-Driven: Tk is event-driven, meaning that it responds to user actions such as button clicks, mouse movements, and keyboard input. You can define callbacks to handle these events and perform actions accordingly.

Widgets: Tk provides a wide range of widgets (GUI components) that you can use to build your application's interface. These include buttons, labels, entry fields, text areas, check buttons, radio buttons, listboxes, and more.

Layout Management: Tk provides facilities for laying out GUI components in your application's windows. You can use geometry managers like pack, grid, and place to arrange widgets in specific positions and sizes.

Applications of rubytk:

RubyTk shines in creating desktop applications with graphical user interfaces (GUIs) for various purposes. Here are some common applications of RubyTk:

1. Develop basic productivity tools like calculators, note-taking apps, to-do list managers, file launchers.
2. Create custom configuration or settings windows for other programs.
3. Design data visualization tools to display information in a user-friendly format (charts, graphs).
4. Design data management interfaces for CRUD (Create, Read, Update, Delete) operations on databases.

** Install tk library by using following command ***gem install tk***

Example: Open notepad and the following code and save the file with **tkexample.rb**

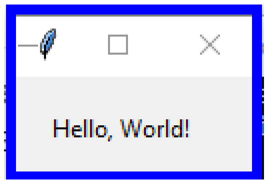
Open command prompt and execute the ruby file

ruby tkexample.rb

```
require 'tk'
root = TkRoot.new { title "First Application" }
```

```
TkLabel.new(root) do
text 'Hello, World!'
pack('padx' => 15, 'pady' => 15, 'side' => 'left')
end
Tk.mainloop
```

Output:



```
Hello, World!
```

Widgets

Widgets in Tk Ruby refer to the various graphical elements(Components) you can create to build a graphical user interface (GUI) for your Ruby applications. These widgets serve as the building blocks for creating interactive and visually appealing interfaces. Some common widgets include buttons, labels, entry fields, checkboxes, radio buttons, listboxes, and frames.
Each widget has its own set of properties and methods that allow you to customize its appearance and behavior. For example, you can set the text displayed on a button, define actions to be performed when a button is clicked, or specify the layout and alignment of widgets within a window.
To create a Tk widget instance, add "Tk" to the widget's name, like TkLabel, TkButton, or TkEntry, and then use the new method to initialize it.

Setting Widget Options

There are two ways to set options for widgets

1. **Using a Hash**: In Perl/Tk, options for widgets are usually passed as key-value pairs within a Hash. For example, if you want to create a label widget with text "Hello, World!" and specify padding and alignment, you would pass these options as a Hash to the widget constructor.

   Example:

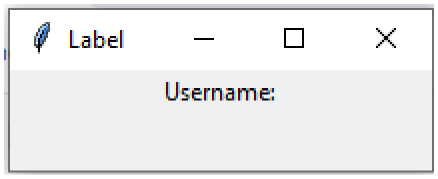
   ```
   TkLabel.new(parent_widget, 'text' => 'Hello, World!').pack('padx' => 5, 'pady' => 5, 'side' =>
   'left')
   ```

2. **Using a Code Block:** In Ruby, you can alternatively pass options using a code block. Within the block, the option names are used as method names, and their values are passed as arguments to these methods. This way, you can set the options in a more Ruby-like manner.

   TkLabel:

```
require 'tk'
root=TkRoot.new {title "Label"}
TkLabel.new(root)do
text "Username:"
pack
end
Tk.mainloop
```

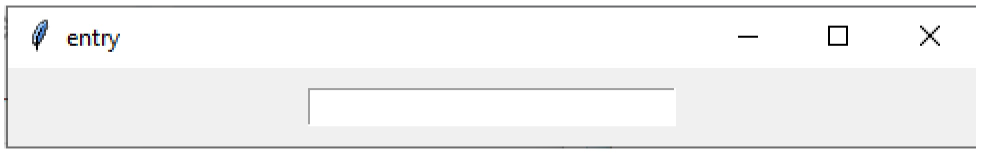
Output:



TkEntry:

```
require 'tk'
root=TkRoot.new{title "entry"}
TkEntry.new(root){
width 30
pack('padx'=>300,'pady'=>10)
}
Tk.mainloop
```

Output:

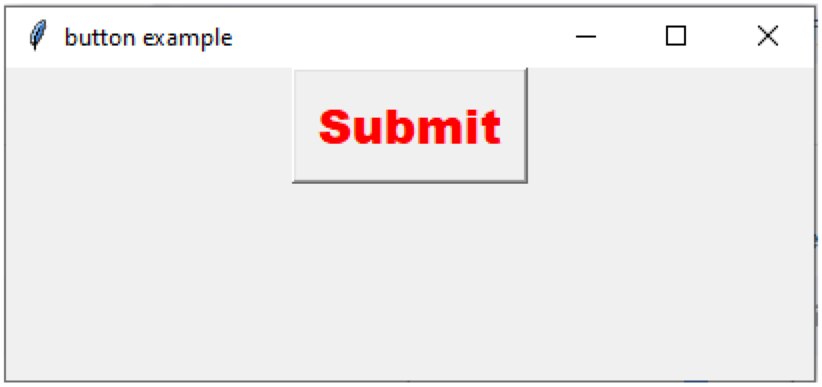


TkButton:

```
require 'tk'
root=TkRoot.new{title "button example"}
TkButton.new(root){
text "Submit"
foreground 'red'
font TkFont.new(size:18,family:"Arial Black")
relief 'flat'
pack
}
```

Tk.mainloop

Output:



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

Getting Widget Data

We can get information back from widgets by using callbacks and by binding variables. Callbacks are very easy to set up.

Callback typically refers to a function or block of code that gets executed in response to a specific user action or event, such as clicking a button, typing in a text field, or selecting an item from a list.

For example, when you set up a callback for a button click event in Tk Ruby, you're essentially specifying what code should be executed when the button is clicked. This code is encapsulated within a Proc object and passed to the command option of the TkButton widget. When the button is clicked, the callback (i.e., the code specified in the Proc object) gets executed.

When an event, such as clicking a button, occurs in Tk Ruby, the Proc object associated with the event (often specified using the command option) is indeed responsible for executing the lines of code defined within it. So, in the case of a button click event, the Proc object's responsibility is to execute the code specified in the command option, which typically includes the actions to be taken in response to the button click.

Example:

require 'tk'

TkButton.new do

text "EXIT"

```
command { exit }
pack('side'=>'left', 'padx'=>10, 'pady'=>10)
end
Tk.mainloop
```

Output: When the program is executed, it creates a window containing a button labeled "EXIT". If you click the button, the window will be closed, thus terminating the application.

Setting/Getting Options Dynamically

configure:

The configure method plays a crucial role in customizing the appearance and behavior of widgets.It is used to change the properties of an existing Tk widget.

Common Options:

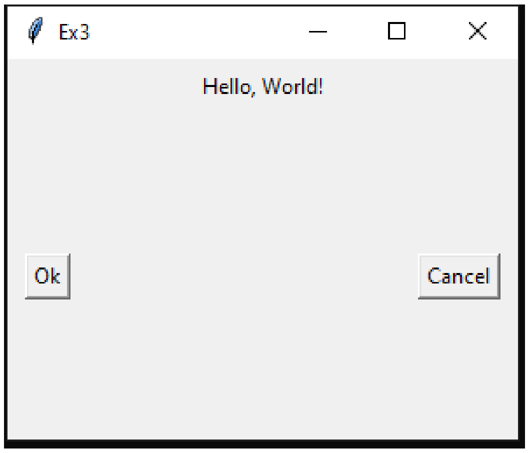
- ❖ **text:** Change the displayed text in labels, buttons, etc.
- ❖ **font:** Modify the font style and size.
- ❖ **background:** Set the background color.
- ❖ **foreground:** Set the text color.
- ❖ **width:** Define the width of the widget in pixels or characters.
- ❖ **height:** Define the height of the widget in pixels.
- ❖ **state:** Control the widget's state (e.g., :disabled, :normal).
- ❖ **relief:** Set the border style (e.g., :flat, :raised, :groove).

Example:

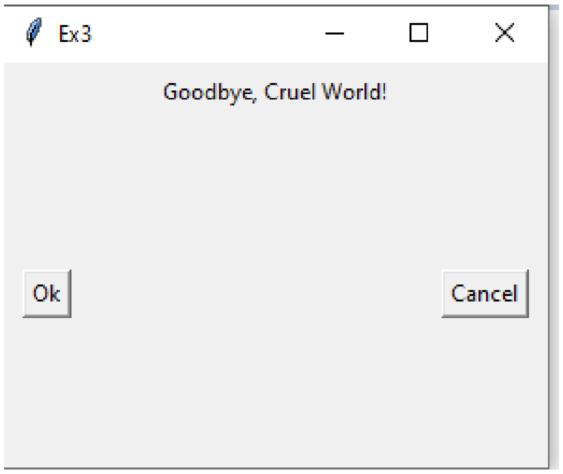
```
require 'tk'
root = TkRoot.new { title "Ex3" }
lbl = TkLabel.new(root) do
justify 'center'
text 'Hello, World!'
pack('padx'=>5, 'pady'=>5, 'side' => 'top')
end
TkButton.new(root) do
text "Ok"
command { exit }
pack('side'=>'left', 'padx'=>10, 'pady'=>10)
end
```

```
TkButton.new(root) do
text "Cancel"
command { lbl.configure('text'=>"Goodbye, Cruel World!") }
pack('side'=>'right', 'padx'=>10, 'pady'=>10)
end
Tk.mainloop
```

Output:



When we click on cancel the label content will be changed.



Geometry Management:

Geometry management refers to the process of arranging and positioning widgets within a window. It plays a crucial role in defining the layout and visual structure of your application. Tk provides three main geometry managers:

1. Pack
2. Grid
3. Place

Pack:

The pack geometry manager organizes widgets in rows or columns inside the parent window or the widget. To manage widgets easily, the pack geometry manager provides various options, such as fill, expand, and side.

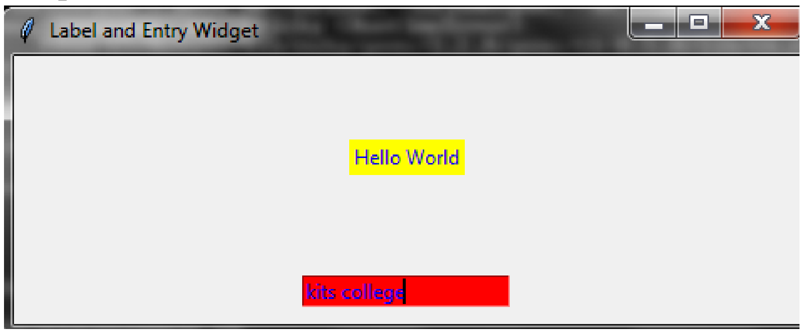
Here is a simple syntax to create a pack Widget −

```
pack('padx'=>10, 'pady'=>10, 'side'=>'left')
```

Example:

```
require 'tk'
top = TkRoot.new {title "Label and Entry Widget"}
#code to add a label widget
lb1 = TkLabel.new(top) {
  text 'Hello World'
  background "yellow"
  foreground "blue"
  pack('padx'=>200, 'pady'=>50, 'side'=>'top')
}
#code to add a entry widget
e1 = TkEntry.new(top) {
  background "red"
  foreground "blue"
  pack('padx'=>100, 'pady'=>10, 'side'=>'top')
}
Tk.mainloop
```

Output:



Grid:

The grid geometry manager is the most flexible and easy-to-use geometry manager. It logically divides the parent window or the widget into rows and columns in a two-dimensional table.

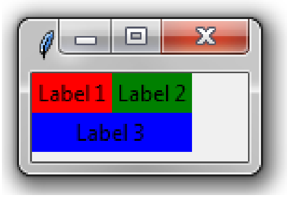
Syntax:

grid('row'=>x, 'column'=>y)

Example:

```
require 'tk'
# Create a Tk root window
root = TkRoot.new { title "Grid Layout Example" }
# Create labels
label1 = TkLabel.new(root) { text 'Label 1'; background 'red' }
label2 = TkLabel.new(root) { text 'Label 2'; background 'green' }
label3 = TkLabel.new(root) { text 'Label 3'; background 'blue' }
# Arrange labels using grid geometry manager
label1.grid('row' => 0, 'column' => 0)
label2.grid('row' => 0, 'column' => 1)
label3.grid('row' => 1, 'column' => 0)
# Start the Tk event loop
Tk.mainloop
```

Output:



Place:

The place geometry manager allows you to place a widget at the specified position in the window. You can specify the position either in absolute terms or relative to the parent window or the widget.

To specify an absolute position, use the x and y options. To specify a position relative to the parent window or the widget, use the relx and rely options.

In addition, you can specify the relative size of the widget by using the relwidth and relheight options provided by this geometry manager.

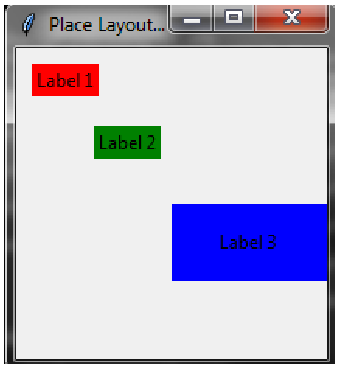
Syntax

place(relx'=>x, 'rely'=>y)

Example:

```
require 'tk'
# Create a Tk root window
root = TkRoot.new { title "Place Layout Example" }
# Create labels
label1 = TkLabel.new(root) { text 'Label 1'; background 'red' }
label2 = TkLabel.new(root) { text 'Label 2'; background 'green' }
label3 = TkLabel.new(root) { text 'Label 3'; background 'blue' }
# Place labels using place geometry manager
label1.place('x' => 10, 'y' => 10)
label2.place('x' => 50, 'y' => 50)
label3.place('x' => 100, 'y' => 100, 'width' => 100, 'height' => 50)
# Start the Tk event loop
Tk.mainloop
```

Output:



Binding Events

"bind" refers to the process of associating an event with a specific action or callback function. When an event occurs, such as a mouse click, keypress, or window resize, the associated action or function is executed.

Syntax:

widget.bind(event, callback_function)

- widget: The widget to which the event is bound.
- event: The event to bind, specified as a string (e.g., "<Button-1>" for left mouse click).
- callback_function: The function to be called when the event occurs**.**

Example:

```
require 'tk'
root=TkRoot.new
b=TkButton.new(root)do
text "Click here"
pack
end
l=TkLabel.new(root)do
text "This text will be changed"
pack
end
l.focus
b.command{l.configure(text:"changed")}
l.bind("Key-a"){l.configure(background:"red")}
Tk.mainloop
```

Canvas:

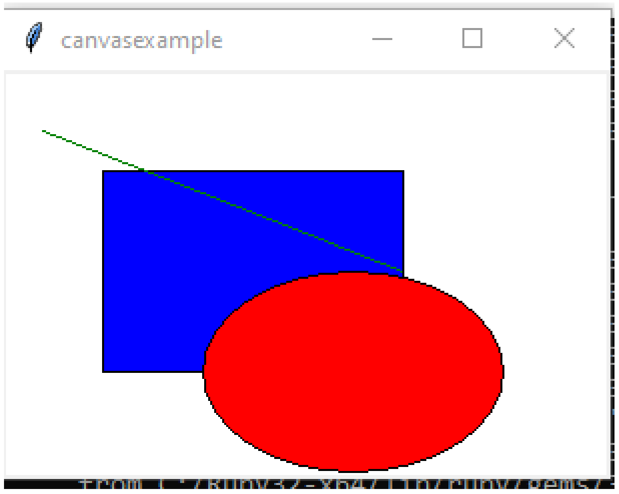
Canvas widget is used to draw graphical elements such as lines, rectangles, circles, and text in a window.

Example:

```
require 'tk'
root = TkRoot.new
canvas = TkCanvas.new(root) do
  width  300
  height 200
  background 'white'
  pack
end
# Draw a rectangle (x,y,width,height)
rectangle = canvas.create('rectangle', 50, 50, 200, 150, fill: 'blue')
```

```
# Draw an oval()
oval = canvas.create('oval', 100, 100, 250, 200, fill: 'red')
# Draw a line
line = canvas.create('line', 20, 30, 200, 100, fill: 'green')
Tk.mainloop
```

Output:



Scrolling:

Scrolling in Ruby TK allows you to view content that's larger than the visible area of a widget like a text box or a frame. Here's a breakdown of how it works:

The Widgets Involved:

- **Scrollbar:** This is a separate widget that displays a slider and arrows. By dragging the slider or clicking the arrows, users can navigate through the hidden content.
- **Viewable Area:** This is the portion of the main widget (text box, frame, etc.) that's currently visible on the screen

Making it Work:

1. **Creating the Scrollbar:** You use the Tk::Scrollbar class to create a scrollbar widget. You can customize its appearance using options like orient (vertical or horizontal scrolling) and command (to link it with the viewable area).
2. **Linking the Scrollbar:** The command option in the scrollbar widget references methods like xview or yview of the viewable area widget. These methods control the visible portion of the content.
3. **User Interaction:** When the user interacts with the scrollbar (drags the slider or clicks the arrows), the command option triggers the xview or yview methods in the viewable area widget. These methods adjust the displayed content accordingly.

```
require 'tk'
begin
  root = TkRoot.new
  text = TkText.new(root) { width 40; height 10 }
  text.insert('end', "This is a much shorter text.\n" *100)
  scroll_bar = TkScrollbar.new(root)
  text.yscrollcommand(scroll_bar.method(:set))
  scroll_bar.command(proc { |*args| text.yview(*args) })
  scroll_bar.pack(side: 'right', fill: 'both', expand: true)
  text.pack(side: 'left', expand: true, fill: 'both')
  Tk.mainloop
end
```

UNIT - II

Extending Ruby:

Extending Ruby means adding extra powers to the Ruby programming language by bringing in tools or capabilities from other languages like C or C++. This lets programmers use existing software written in those languages directly within their Ruby programs. For example, if there's a really fast and powerful tool written in C that you want to use in your Ruby program, you can "extend" Ruby to include that tool's abilities.This enables developers to leverage existing libraries, access low-level system functionality, or improve performance for certain tasks.

Ruby Objects in C

Everything you interact with, from numbers and strings to actions (methods), is treated as an object. When you work with these objects in C extensions for Ruby, you need to understand how they are represented and accessed.

Most Ruby objects reside in memory, managed by pointers of type VALUE in C. These pointers act like addresses, pointing to the object's location in memory where its actual data is stored (like the value of a number or the characters in a string).

There's a special case for simple values like integers (Fixnums), symbols, true/false, and nil. These are stored directly within the VALUE variable itself, forgoing the need for separate memory allocation. This makes accessing and manipulating them faster.

VALUE acts as a container type in the Ruby C API, able to hold either pointers (for most objects) or immediate values (simple values). The Ruby interpreter utilizes the object's memory address to distinguish between the two:

- ❖ If the lower bits of the VALUE variable are zero (due to memory alignment), it indicates a pointer.
- ❖ If the lower bits are not zero, it signifies an immediate value stored within the VALUE variable itself.

Each object, whether a regular object accessed through a pointer or an immediate value stored directly, has its own structure. This structure holds essential information:

- ➔ **Regular objects:**
    - ◆ They have a table of instance variables to store their specific data, like the value of an integer or the characters in a string.
    - ◆ They also have information about their class (type) attached to them. This tells the interpreter what kind of object it is (e.g., String, Array, Hash) and allows it to access the appropriate methods defined for that class.
- ➔ **Immediate values:**
    - ◆ Their value is directly stored within the VALUE variable itself.

- ◆ Class information might be attached in a different way, depending on the specific type of immediate value.

Working With Immediate Objects:

When you're working with Ruby's internal data structures, some values are stored directly in a special way. These include small numbers (Fixnums), symbols, true, false, and nil.

For Fixnums, instead of storing them as they are, Ruby shifts them left by one bit and sets the rightmost bit (bit 0) to 1. This special bit pattern helps distinguish them from other types of data. So, if a value is used as a pointer to a Ruby structure, it always has its rightmost bit set to 0.

To check if a value is a Fixnum, you just need to check if its rightmost bit is set to 1. Ruby provides a macro called FIXNUM_P to do this check easily. Similarly, there are similar checks for other immediate values like symbols, nil, and checking if a value is neither nil nor false.

In C, these other immediate values (true, false, and nil) are represented by the constants Qtrue, Qfalse, and Qnil respectively. You can directly test Ruby values against these constants, or use conversion macros which handle the necessary typecasting for you.

Similar tests let you check for other immediate values.

FIXNUM_P(value) → nonzero if value is a Fixnum
SYMBOL_P(value) → nonzero if value is a Symbol
NIL_P(value) → nonzero if value is nil
RTEST(value) → nonzero if value is neither nil nor false.

Working with Strings:

In C programming, we often handle strings as sequences of characters terminated by a null character ('\0'). However, Ruby's strings are more flexible and can contain null characters within them. To safely work with Ruby strings in C code, Ruby provides a structure called RString, which contains both the length of the string and a pointer to where the string is stored in memory.

To access these properties of a Ruby string in C code, we use the RSTRING macro. For example, RSTRING(str)->len gives us the length of the string, and RSTRING(str)->ptr gives us a pointer to where the string is stored.

For example, if we want to iterate over all the characters in a string, we would call StringValue on the original object to ensure it's a string, then access its length and pointer through the RSTRING macro.

If we just need the pointer to the string's contents and don't care about the length, we can use the convenience method StringValuePtr, which resolves the string reference and returns the C pointer to the contents directly.

SafeStringValue method works similarly to StringValue, which converts an object into a string. However, SafeStringValue does an additional security check. If the string it receives is tainted (meaning it's from an untrusted source), and the current security level is higher than zero (indicating a restricted environment), it will throw an exception rather than converting the string.

Working with Other Objects:

In Ruby, values are the fundamental building blocks that hold different kinds of information. Think of them as multi-purpose containers. Some VALUEs directly store data, like simple numbers or short text strings. Others are more like signposts – they point to a specific location where a larger, more complex Ruby object is stored.

Each type of Ruby object, like strings, arrays, or numbers, has a predefined structure behind the scenes. These structures are written in the C programming language and are detailed within a file called ruby.h. It's like having architectural plans for different kinds of objects. Ruby provides special helper functions (called macros) that start with "RClassname" which allow you to access and work with these internal structures.

You can check the underlying structure of a VALUE by using the TYPE(obj) macro. This is like checking the type of container a value is stored in. The Check_Type macro helps ensure you're working with the right kind of object. It's a safety measure that prevents you from accidentally trying to manipulate the wrong kind of data.

Let's use an array as an example. If you have a VALUE that represents an array, you can use the RARRAY(arr) macro to access the array's internal structure. This lets you see things like the length of the array (how many items it holds), its capacity (how much space is allocated), and even a pointer to where the array's data is actually stored.

Global Variables

- In most cases, your C extensions use classes and objects to share data with Ruby code. This is the recommended approach for organized and safe data exchange.
- However, there might be situations where you need a single, global variable accessible to both sides.
- **Create a VALUE (Ruby object):** This is a container that can hold various data types in Ruby.
    - In this example, hardware_list is a VALUE that initially stores an empty array.

- **Bind the VALUE's address to a Ruby variable name:**
    - The rb_define_variable function associates the address of hardware_list with the Ruby variable $hardware.
    - The $ prefix is optional but commonly used to indicate a global variable.
- **Fill the array with data:**
    - rb_ary_push adds elements like "DVD", "CDPlayer1", and "CDPlayer2" to the array.

Accessing the Global Variable:

- Now, the Ruby code can access the hardware_list using $hardware as if it were a regular Ruby array.
- The example shows how to retrieve the list:

$hardware # => ["DVD", "CDPlayer1", "CDPlayer2"]

Hooked and virtual variables are more advanced techniques in Ruby extensions written in C that go beyond simple global variables. Here's a breakdown to help you understand them better:

Regular Global Variables:

Imagine a bulletin board outside a classroom with a section for announcements. Anyone can add or remove notices from there. This is similar to a regular global variable:

Value: Stored directly, like a written notice on the board.

Access: Anyone (Ruby or C code) can read or modify the value.

Hooked Variables:

Think of a bulletin board where someone has written "Check the teacher's desk for today's assignment." This message tells you where to look for the actual information, not the information itself. A hooked variable works similarly:

Value: Not directly stored. Instead, it has a named function (called a "hook") that calculates the value when accessed.

Access: When Ruby code tries to access the hooked variable, the hook function is called to generate the value on the fly. C code can also call the hook function directly.

Virtual Variables:

Imagine a virtual bulletin board that only exists in your mind. When you look for an announcement, you automatically recall a specific piece of information. A virtual variable functions like this:

Value: Never stored anywhere. It has a hook function that calculates the value every time it's accessed.

Access: Similar to hooked variables, the hook function is called when the virtual variable is accessed in Ruby code or C code.

Memory Allocation:

In Ruby, sometimes you need to allocate memory for special purposes, like big images or lots of small data. Ruby's garbage collector helps manage memory, but when you're doing it yourself, use special tools provided by Ruby. These tools make sure that if you can't allocate memory, the garbage collector tries to free up space. They're more advanced than basic memory allocation functions like 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

1. **type * ALLOC_N(c-type, n):** This function is used in Ruby extensions for allocating memory.

   c-type: This is the first argument, and it represents the data type of the objects you want to allocate memory for. It should be the actual name of the C data type, like int, float, or a custom structure you defined.

   n: This is the second argument, and it represents the number of objects you want to allocate memory for. It should be an integer expression that evaluates to a positive number.

   Example: For example, if you call ALLOC_N(int, 10), it will allocate memory for 10 integer objects. Since each int takes 4 bytes, the total memory allocated will be 40

2.type * ALLOC(c-type): type * ALLOC(c-type) is another specific function used in Ruby extensions for memory allocation, but unlike ALLOC_N, it only allocates memory for one object.

- ❖ Similar to ALLOC_N, it interacts with the Ruby garbage collector to ensure efficient memory management.
- ❖ If there's not enough memory available even after the garbage collector cleans up, ALLOC will raise an error called NoMemError.
- ❖ Unlike standard malloc in C, the ALLOC function automatically **casts** the allocated memory to a pointer of the same type (c-type *). This means it returns the **starting address** of the allocated memory location, allowing you to directly access and modify the object's data.

3. **REALLOC_N(var, c-type, n):** This function is used in Ruby extensions for resizing previously allocated memory.

- ❖ var: This is the first argument, and it's a pointer variable of type c-type*. It points to the memory location that you want to resize.
- ❖ c-type: This is the second argument, and it represents the data type of the objects stored in the memory pointed to by var.
- ❖ n: This is the third argument, and it represents the new number of objects you want to fit in the resized memory.
- ❖ This function attempts to resize the memory block pointed to by var to accommodate n objects of type c-type.
- ❖ It considers two possibilities:

- ➔ **Increasing size:** If n is larger than the current number of objects, REALLOC_N tries to expand the memory to hold n objects.
- ➔ **Decreasing size:** If n is smaller than the current number of objects, REALLOC_N tries to shrink the memory to fit n objects and potentially free up unused space.

4.type * ALLOCA_N(c-type, n):

ALLOCA_N allocates memory on the stack for n objects of type c-type.

ALLOCA_N is a macro that allocates memory on the stack for an array of a specified type and size. This memory is automatically freed when the function called ALLOCA_N returns. It's useful for quickly allocating memory for small to moderate-sized arrays within a function without needing to manually free it later. However, it's best suited for small data structures due to stack space limitations. For larger memory needs or data persistence beyond a function's scope, heap allocation is more appropriate.

Ruby Type System

Duck typing means that when writing code, you're more concerned with what an object can do (its behavior) rather than what specific type or class it belongs to.

Example:

Duck typing in Ruby can be illustrated with a classic example involving animals. Imagine we have a function called make_sound that expects an object to respond to a method called sound. Instead of checking if the object is specifically a "Duck", we simply check if it responds to the sound method. If it does, we treat it as if it were a duck, regardless of its actual type. For instance:

```
class Duck
def sound
puts "Quack!"
end
end
class Dog
def sound
puts "Woof!"
```

```
end
end
def make_sound(animal)
animal.sound
end
duck = Duck.new
dog = Dog.new
make_sound(duck) # Output: Quack!
make_sound(dog)  # Output: Woof!
```

Creating an Extension

1. Create the C source code file(s) in a given directory.
2. Optionally create any supporting Ruby files in a lib subdirectory.
3. Create extconf.rb.
4. Run extconf.rb to create a Makefile for the C files in this directory.
5. Run make.
6. Run make install.

Suppose we want to create a C extension for Ruby that provides a function to add two numbers together. Here's how we would do it:

Create the C source code file(s):

- ➔ Create a dedicated directory for your extension to organize the related files.
- ➔ Inside this directory, create a C source file (e.g., my_extension.c) that will contain the C code implementing your extension's functionality.

Let's create a file named addition.c in our directory with the following content:

```
#include "ruby.h"
VALUE method_add(VALUE self, VALUE num1, VALUE num2) {
 int result = NUM2INT(num1) + NUM2INT(num2);
   return INT2NUM(result);
}
void Init_addition() {
 VALUE MyClass = rb_define_class("MyClass", rb_cObject);
 rb_define_method(MyClass, "add", method_add, 2);
}
```

Create supporting Ruby files:

If your extension requires additional Ruby code to interact with the C functionalities or define helper methods, you can create a subdirectory named lib within your main directory. Place any relevant Ruby files (e.g., modules, classes) in this lib subdirectory for better organization.

We won't create any supporting Ruby files for this example.

Create extconf.rb:

Create a file named extconf.rb in the main directory.

This file acts as a configuration script, providing instructions for building the extension and specifying necessary information like the extension name.

Create a file named extconf.rb in the same directory with the following content:

```
require 'mkmf'
create_makefile('addition')
```

Run extconf.rb:

Open your terminal and navigate to the directory containing your extension files.

Run the command ruby extconf.rb in the terminal. This command processes your extconf.rb file and creates a Makefile that defines the build process for your extension.

Open your terminal, navigate to the directory containing your files, and run ruby extconf.rb. This will generate a Makefile for your C files.

Run make:

Once the Makefile is generated, execute the command make in the terminal. This command uses the instructions in the Makefile to compile the C code and create the extension library.

After extconf.rb has generated the Makefile, run make in your terminal. This will compile your C code into a shared library.

Run make install:

Finally, run the command make install in the terminal. This command installs the compiled extension library into your Ruby environment, making it available for use in your Ruby programs.

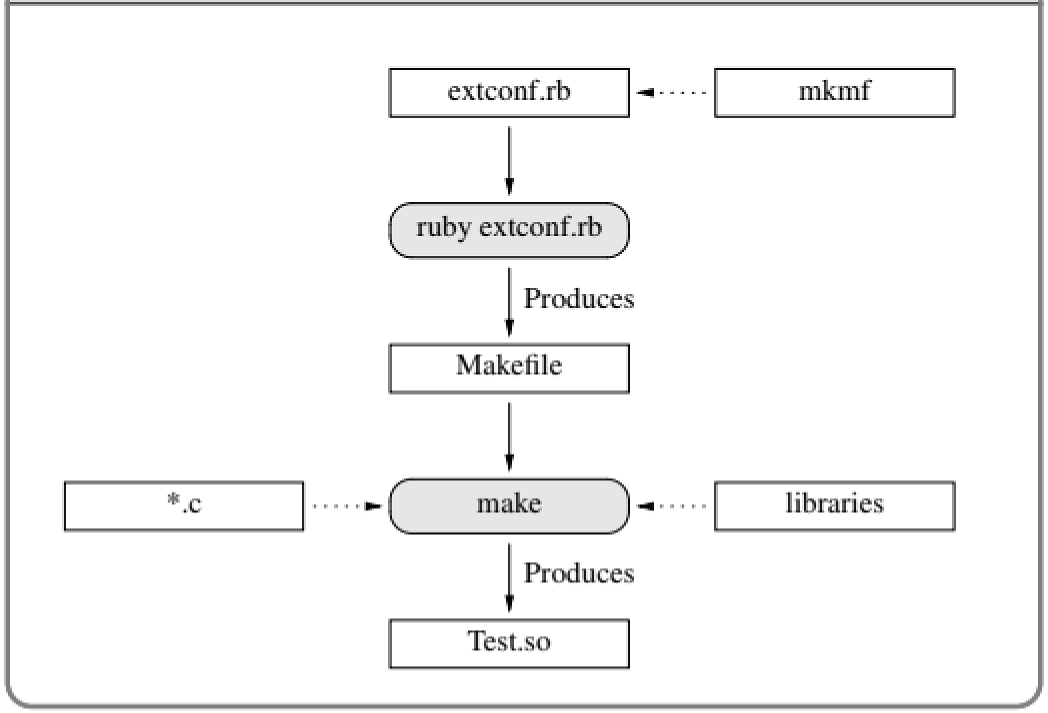
Finally, run make install. This will install your Ruby C extension, making it available for use in Ruby scripts.

After following these steps, you can use your C extension in Ruby scripts like this:

```
require 'addition'
obj = MyClass.new
puts obj.add(3, 5) #=> 8
```

This will create an instance of MyClass and call the add method, which is defined in the C code, to add the two numbers.

Figure 21.2. Building an extension



API: Defining Classes:

1.rb_define_class():

Syntax: VALUE rb_define_class(char *name, VALUE superclass)

Defines a new class at the top level with the given name and superclass (for class Object, use rb_cObject).

Example:

rb_define_class("MyClass", rb_cObject);

MyClass is defined with rb_define_class. It will have Object as its superclass since rb_cObject represents the Object class.

2.rb_define_module():

Syntax: VALUE rb_define_module(char *name)

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

Example: VALUE MyModule = rb_define_module("MyModule");

3.rb_define_class_under():

Syntax: VALUE rb_define_class_under(VALUE under, char *name, VALUE superclass)

Defines a nested class under the class or module under.

Example: VALUE my_class = rb_define_class_under(my_module, "MyClass", rb_cObject);

4.rb_define_module_under():

Syntax:

VALUE rb_define_module_under(VALUE under, char *name)

Defines a nested module under the class or module under.

Example: VALUE my_submodule = rb_define_module_under(my_module, "MySubmodule");

5.rb_include_module():

Syntax:void rb_include_module(VALUE parent, VALUE module)

Includes the given module into the class or module parent.

- ➔ parent: A VALUE representing a class or module to include module into.
- ➔ module: A VALUE representing the module to be included.

Example:

```
// Include MyModule into MyClass
   rb_include_module(MyClass, MyModule);
```

API: Defining Structures

1.rb_struct_define():

Syntax: VALUE rb_struct_define(char *name, char *attribute..., NULL)

Defines a new structure with the given attributes.

Example:

```
VALUE MyStruct;
void Init_MyStruct() {
   MyStruct = rb_struct_define("MyStruct", "name", "age", "city", NULL);
}
```

2.rb_struct_new():

Syntax: VALUE rb_struct_new(VALUE sClass, VALUE args..., NULL)

This function, part of the Ruby C API, creates a new instance (object) of a previously defined structure (struct) in Ruby.

Example:

```
void Init_MyStruct() {
   MyStruct = rb_struct_define("MyStruct", "name", "age", "city", NULL);
}
VALUE name_value = rb_str_new2("Alice");
VALUE age_value = rb_fixnum_new(30);
VALUE city_value = rb_str_new2("New York");
VALUE instance = rb_struct_new(MyStruct, name_value, age_value, city_value, NULL);
```

3.rb_struct_aref():

Syntax:VALUE rb_struct_aref(VALUE struct, VALUE idx)

It is used to access a member variable within a previously created structure instance (struct).

Example:

```
VALUE instance = rb_struct_new(MyStruct, name_value, age_value, city_value, NULL);
   // Accessing member variables using rb_struct_aref
   VALUE name = rb_struct_aref(instance, 0);
   VALUE age = rb_struct_aref(instance, 1);
```

4.rb_struct_aset():

Syntax: VALUE rb_struct_aset(VALUE struct, VALUE idx, VALUE val)

This function allows you to modify the value of a member variable within a previously created structure instance (struct) in Ruby C extensions.It essentially sets the value at a specific index (position) within the structure instance.

Example:

```
void Modify_Member_Variable() {
   VALUE name_value = rb_str_new2("Alice");
   VALUE age_value = rb_fixnum_new(30);
   VALUE city_value = rb_str_new2("New York");
```

```
    VALUE instance = rb_struct_new(MyStruct, name_value, age_value, city_value, NULL);
    // Modify the 'age' member variable (index 1) to 35
    rb_struct_aset(instance, 1, rb_fixnum_new(35));
}
```

API: Defining Methods

The parameter argc in these function definitions stands for "argument count". It tells you how many arguments a specific Ruby method can take.
Think of a Ruby method like a function that can be called to perform a task. Arguments are like inputs you provide to the function to tell it what to do or what data to work with.

1. **Fixed number of arguments:**
   0..17 VALUE func(VALUE self, VALUE arg...)
   This function can take **0 to 17 arguments** (inclusive).self refers to the object on which the method is called.arg... represents a variable number of arguments, similar to variadic functions in C. These arguments are typically accessed using indexing or iteration techniques within the C function.
2. **Variable number of arguments (C-style):**
   -1= VALUE func(int argc, VALUE *argv, VALUE self)
   This function can accept any number of arguments (similar to ... in the previous example).argc holds the actual number of arguments passed.argv is a C-style array of VALUE pointers, pointing to the individual arguments passed.self still refers to the object.This variation uses C-style memory access for the arguments, requiring more careful handling in C code.
3. **Variable number of arguments (Ruby-style):**
   -2 VALUE func(VALUE self, VALUE args)
   This function can also accept any number of arguments.self functions as usual.
   args is a Ruby array containing all the arguments passed.This variation provides a more Ruby-like way to access arguments using array methods without direct memory manipulation.

Differences:

0..17 VALUE func(VALUE self, VALUE arg...): Arguments are accessed using indexing or iteration techniques within the C function.

-1 VALUE func(int argc, VALUE *argv, VALUE self): Arguments are accessed using C-style array indexing on the argv pointer array. This requires careful memory management.

-2 VALUE func(VALUE self, VALUE args): Arguments are accessed using Ruby array methods on the args object. This is more convenient and safer than C-style access.

1.rb_define_method():

Defines a new instance method for a class or module in Ruby. It allows you to extend the functionality of existing classes or create custom methods for new classes.

Syntax:

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

- ❖ **VALUE classmod:** Represents the class or module where the method will be defined.
- ❖ **char *name:** A null-terminated string containing the name of the method to be defined.
- ❖ **VALUE(*func)():** A pointer to a C function that will implement the logic of the method. This function takes the following arguments:
- ❖ **int argc:** An integer specifying the expected number of arguments for the Ruby method. This helps ensure type safety and prevents unexpected behavior if the number of arguments passed doesn't match the method's definition.

2.rb_define_alloc_func():

Defines an allocation function for a Ruby class or module.

This function is responsible for creating new instances of that class/module when a call like MyClass.new is made in Ruby code.

Syntax: void rb_define_alloc_func(VALUE classmod, VALUE(*func)())

- ❖ VALUE classmod**:** Represents the Ruby class or module for which the allocation function is being defined.
- ❖ VALUE(*func)()**:** A pointer to a C function that will be responsible for allocating memory and initializing new instances of the class/module. This function:
  - ➢ Takes no arguments.
  - ➢ Returns a VALUE which represents the newly allocated Ruby object.

3.rb_define_module_function():

Defines a function that can be called directly on the module itself using the module name and dot notation (e.g., ModuleName.function_name). These functions are private and cannot be called on instances of the module.

Syntax:

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

4.rb_define_global_function():

Defines a global function in Ruby, accessible from anywhere in your Ruby code without the need for an object or module prefix.These functions are similar to built-in Ruby functions like puts or print, but they are defined in your C extension.

Syntax: void rb_define_global_function(char *name, VALUE(*func)(),int argc)

char *name: A null-terminated string containing the desired name for the global function.

VALUE(*func)(): A pointer to a C function that implements the logic of the global function. This function:

- Takes an arbitrary number of VALUE arguments, which represent the arguments passed to the global function when called in Ruby.

int argc: An integer specifying the expected number of arguments for the global function.

5.rb_define_singleton_method():

Singleton methods defined with rb_define_singleton_method are specific to the individual class or module instance on which they are defined. They are not inherited by subclasses or instances created later.

Syntax:

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

API: Defining Variables and Constants

1.rb_define_const()

Defines a constant within a specified class or module.

Constants are immutable values accessible throughout your Ruby code using the class or module name followed by a double colon (::) and the constant name.

Syntax:

void rb_define_const(VALUE classmod, char *name, VALUE value)

2.rb_define_global_const()

Constants defined with rb_define_global_const are accessible from anywhere in your Ruby code, potentially across different files and modules.

Syntax:

void rb_define_global_const(char *name, VALUE value)

3.rb_define_variable()

The function rb_define_variable serves the purpose of defining variables within a specific scope.

his variable becomes accessible within the current scope, which can be:

Global scope: If called directly in the extension's initialization code.

Module scope: If called within a module or class definition.

Method scope: If called within a C function defined in the extension.

4.rb_define_class_variable()

This variable is accessible to all instances of the class and the class itself.

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

Syntax:

void rb_define_class_variable(VALUE class, const char *name,VALUE val)

API: Calling Methods

1.rb_class_new_instance():

This function belongs to Ruby's C extension API and is used to create a new instance (object) of a Ruby class.

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

This function takes three arguments:

Number of arguments: This specifies how many arguments were passed to the function when it was called.

Arguments: This is an array containing the actual arguments passed to the function.

Class: This identifies the specific Ruby class for which a new instance needs to be created.

API: Exceptions:

1.rb_raise():

rb_raise is a function in Ruby's C API used to explicitly raise an exception during program execution.

syntax:

void rb_raise(VALUE exception, const char *fmt, ...)

exception: This is a VALUE object representing the class of the exception you want to raise. Ruby provides built-in error classes like rb_eRuntimeError or rb_eArgError. You can also use custom exception classes defined in your C extension.

fmt: This is a null-terminated C string that specifies the error message. It can use C-style formatting similar to printf.

...: This represents optional variable arguments that are used for formatting the error message with fmt.

Example:

```
int result = do_something_complex();
if (result == -1) {
  rb_raise(rb_eRuntimeError, "Failed to perform complex operation!");
}
```

2.rb_fatal():

rb_fatal is a function in Ruby's C API used for reporting critical errors that halt the Ruby interpreter immediately.Unlike rb_raise which throws an exception, rb_fatal terminates the Ruby interpreter immediately. No further code execution happens after calling rb_fatal.There's no guarantee that any cleanup code will be run after rb_fatal.

Syntax:

```
void rb_fatal( const char *fmt, ... )
```

Example:

```
void *ptr = malloc(1024 * 1024);
if (ptr == NULL) {
  rb_fatal("Failed to allocate memory!");
}
```

In this example, if malloc fails to allocate memory, rb_fatal is called with an error message, and the Ruby interpreter terminates immediately.

3.rb_sys_fail()

In Ruby's C API, rb_sys_fail is a function used to report critical errors that have severe consequences for the system. It's designed for situations where continued execution is not possible or would lead to a highly unstable state.

Use Cases:

★ Out-of-memory conditions when memory allocation fails critically.
★ File system errors where essential files cannot be accessed or modified.
★ System call failures that indicate a fundamental issue with the platform.

Syntax:

4.void rb_sys_fail(const char *msg)

Example:

```
FILE *fp = fopen("important_data.txt", "w");
if (fp == NULL) {
  rb_sys_fail("Failed to open critical data file!");
}
```

In this example, if fopen fails to open a crucial file, rb_sys_fail is called, indicating a system-level issue. The program might still technically continue execution, but reliable operation is compromised.

5.rb_rescue():

rb_rescue helps you write code that can gracefully handle exceptions and provide alternative behavior.

The rescue function provides a way to recover from errors or provide default values.

Consider using rb_ensure if you need code to always be executed, regardless of exceptions.

Syntax:

VALUE rb_rescue(VALUE (*body)(), VALUE args, VALUE(*rescue)(), VALUE rargs)

API: Iterators:

Iterator: Ruby iterators provide a way to process elements in collections like arrays or hashes. You typically use iterator blocks with methods like each, map, reduce, etc. These methods call the block you provide once for each element in the collection.

1.rb_iter_break( )

Purpose of rb_iter_break

Within an iterator block, if a certain condition is met for an element, you might want to stop iterating altogether, even though there are more elements remaining. rb_iter_break helps achieve this by signaling an immediate exit from the loop.

Syntax:

void rb_iter_break( )

2.rb_each()

In Ruby's C API, rb_each is a function designed to iterate over elements in an enumerable object (like an array, hash, string, etc.). However, it's important to note that rb_each isn't directly available in Ruby code. It's an internal function primarily used for implementing core Ruby methods like each and other iterators.

Syntax:

VALUE rb_each(VALUE obj)

3.rb_yield():

rb_yield(VALUE arg) is an internal function within Ruby's C API. It's not something you'd use directly in everyday Ruby programming.

Executes a block of code associated with a method call in Ruby, but from C code.

Primarily for extending Ruby or creating C extensions that interact with blocks in a very specific way.

VALUE arg is a Ruby value that can potentially be passed to the block, but not always.

Focus on using yield within Ruby methods to work with blocks. rb_yield is for specialized C-level interactions.

4.rb_iterate():

The rb_iterate function is a C function provided by the Ruby C API. It's used for iterating over a block of Ruby code, calling a given method with specified arguments and potentially a block.

Syntax:

VALUE rb_iterate(VALUE (*method)(),VALUE args,VALUE (*block)(),VALUE arg2)

Example:

```
#include "ruby.h"
VALUE my_method(VALUE arg) {
  // Do something with the argument
  return Qnil; // In this example, we return nil
}
VALUE my_block(VALUE arg) {
  // Do something with the block argument
  return Qnil; // In this example, we return nil
}
int main() {
  // Initialize Ruby VM
  ruby_init();
  // Define arguments
  VALUE args = ...; // Define your arguments
  VALUE arg2 = ...; // Define your additional argument
  // Call rb_iterate
  rb_iterate(my_method, args, my_block, arg2);
  // Clean up Ruby VM
  ruby_cleanup(0);
    return 0;
}
```

5.rb_catch():

Implements exception handling using a non-standard mechanism.

Temporarily intercepts exceptions (denoted by throw) within a block of code.

Executes the designated code and returns a value based on whether an exception is thrown.

Syntax:

VALUE rb_catch(const char *tag, VALUE (*proc)(), VALUE value)

Example:

```
def my_proc(value)
 if value < 0
```

```
    throw "neg_val", "Encountered negative value"
  else
    value * 2
  end
end
result = rb_catch("neg_val") do
  my_proc(-5)
end
puts result
```

API: Accessing Variables

1.rb_iv_get():

In Ruby, rb_iv_get(VALUE obj, char *name) is a function used to retrieve the value of an instance variable associated with a specific object.

- ❖ obj (VALUE): The Ruby object from which you want to retrieve the instance variable.
- ❖ name (char *name): A C-style string representing the name of the instance variable (without the leading @ symbol).

Example:

```
class Person
  def initialize(name, age)
    @name = name  # Instance variable assignment (using `@`)
    @age = age
  end
  def get_name
    rb_iv_get(self, "@name")  # Accessing instance variable using rb_iv_get
  end
end
person = Person.new("Alice", 30)
name = person.get_name
puts name  # Output: "Alice"
```

2.rb_ivar_get():

The difference lies in how the instance variable name is specified:

- ❖ rb_iv_get() requires you to specify the name of the instance variable as a separate argument.
- ❖ rb_ivar_get() doesn't require you to specify the instance variable name explicitly; it assumes that you want to access the instance variable with the same name as the variable you're assigning the result to in your C code.

Syntax:

VALUE value = rb_ivar_get(obj);

3.rb_iv_set():

In Ruby, rb_iv_set(VALUE obj, char *name, VALUE value) is a C function used to assign a value to an instance variable associated with a specific object.

- ❖ Set or modify the value of an instance variable within a C extension.

Example:

```
#include <ruby.h>

VALUE set_name(VALUE self, VALUE new_name) {
  rb_iv_set(self, "@name", new_name);  // Set instance variable using rb_iv_set
  return self;  // Return the object itself (optional)
}
```

4.rb_gv_set():

In Ruby, rb_gv_set(const char *name, VALUE value) is a C function used to set or modify the value of a global variable.

- ❖ Assign a value to a global variable within a C extension.
- ❖ Global variables are accessible from anywhere within the Ruby program.

Example:

```
#include <ruby.h>
void set_global_count(VALUE count) {
  rb_gv_set("global_count", count);  // Set global variable using rb_gv_set
}
```

5.rb_gv_get()

In Ruby, rb_gv_get(const char *name) is a C function used to retrieve the value of a global variable within a C extension.

- ❖ Access the value of a global variable from a C extension.

UNIT - III

In the context of computing, both "script" and "program" refer to sequences of instructions that are executed by a computer. However, there are some distinctions between the two:

- **Script:**
  - A script is typically a sequence of commands or instructions written in a scripting language.
  - It is often interpreted rather than compiled, meaning that the instructions are executed line by line by an interpreter at runtime.
  - Scripts are commonly used for tasks such as automating repetitive tasks, system administration, web development (server-side scripting), and quick prototyping.
  - **Examples of scripting languages** include Python, Perl, Ruby, JavaScript (when used outside of web browsers, e.g., Node.js), and shell scripting languages like Bash.
  - **Example:**

```
# Script to greet the user by name
name = input("What's your name? ")
print(f"Hello, {name}!")
```

- **Program:**
  - A program is a sequence of instructions written in a programming language.
  - It can be compiled into machine code or bytecode before execution.
  - Programs can be large and complex, consisting of multiple files/modules and utilizing libraries and frameworks.
  - They are used for developing a wide range of applications, including desktop software, mobile apps, web applications, games, operating systems, and more.
  - **Examples of programming languages** used for writing programs include C, C++, Java, C#, Swift, Kotlin, and many others.

Example:

```
// Program to calculate the area of a circle (simplified)
public class CircleArea {
  public static void main(String[] args) {
    double radius = 5.0;
    double area = Math.PI * radius * radius;
    System.out.println("Area of the circle: " + area);
  }
}
```

Feature	Script	Program

Complexity	Simpler, focused on one specific task	More complex, can handle multiple functionalities
Development Time	Faster to write and test	Can take longer to develop and debug
Purpose	Automate repetitive tasks	Build full-fledged applications, games, or systems
Language	Scripting language (e.g., Python, JavaScript)	Programming language (e.g., Java, C++, C#)
Accessibility	Easier to learn for beginners	Requires more programming knowledge

Origins of scripting:

The term "script" in computing began in the early 1970s with UNIX. It started with "shell scripts," sequences of commands stored in files and run like typing them directly. This use of "script" spread to other languages like AWK and Perl, where scripts are text files executed directly without compilation.

Early uses of "script" mirrored other fields. DOS dial-up connections used scripts in a special language to automate connection steps, similar to film scripts guiding actors and cameras. Apple's HyperCard, a hypertext system, used "scripts" in its HyperTalk language to trigger actions based on user interactions (like mouse clicks) to be user-friendly (avoiding the scary word "program"). In both these cases, scripts were about control: controlling a modem's actions or changing how things appeared on the screen. (Script and control are often linked, but scripts have other uses too.)

An important distinction is made: scripts become truly powerful when they go beyond simple sequences and incorporate programming concepts like loops and branching. This is when they become more than just following commands in order.

Scripting today:

Scripting as a way to give instructions to your computer in a simpler language. Instead of complex code, you use something easier to understand, kind of like how recipes are easier to follow than engineering manuals.

There are three main ways scripting is used:

1. **Building things fast (Rapid Application Development):** Think of it like using Legos. Scripting languages let you connect pre-made building blocks (software components) to create new programs quickly. This is useful when you need a simple program and don't have time to write everything from scratch.
2. **Controlling existing programs:** Scripting can be like a remote control for your software. If a program allows you to control it with scripts, you can write scripts to automate tasks you do often. For example, you could write a script to automatically format all your reports in a specific way.
3. **Sometimes a simpler way to program:** Scripting languages can sometimes be used for more general programming tasks, especially for things like managing computers behind the scenes (system administration). Imagine you have to do a lot of repetitive tasks on your computer, like copying files or setting up new user accounts. Scripting languages can automate these tasks, making your work much faster.

Here are some real-world examples:

- ❖ Creating a website: Scripting languages like JavaScript can be used to make websites more interactive, like adding animations or allowing users to search for information.
- ❖ Automating social media posts: Scripting can be used to schedule social media posts in advance or automatically respond to comments.
- ❖ Analyzing data: Scripting languages like Python can be used to analyze large amounts of data, which can be helpful for businesses or researchers.

Characteristics of scripting languages:

The languages used for these different kinds of scripting have many features in common, which serve to define an overall concept of a scripting language,and to differentiate it from the concept of programming language. We list here some of the features that characterize scripting language.

1.Integrated compile and run

2.Low overheads and ease of use.

3.Enhanced functionality.

4.Efficiency is not an issue.

1.Integrated Compile and Run:

- ❖ Traditional programming languages often involve a two-step process:

    - ➢ **Compile:** You write the code, and a separate program (compiler) translates it into a machine-readable format (like translating instructions for a robot into its language).
    - ➢ **Run:** Once compiled, you execute the code, telling the computer to follow the translated instructions.
- ❖ Scripting languages often combine these steps. You write and execute the code at the same time, like giving step-by-step commands directly to a robot. This makes scripting faster for small tasks or when you're learning and experimenting.

2. Low Overheads and Ease of Use:

- ❖ Scripting languages are designed to be simpler to learn and use compared to traditional programming languages. They typically have:
    - ➢ **Simpler syntax:** The way you write instructions is more natural and easier to understand, with less complex rules and structures.
    - ➢ **Fewer requirements:** They might not need as much setup or configuration compared to traditional languages.

Think of it like building with Legos. Scripting languages are like pre-made building blocks that you can snap together easily, while traditional languages might require you to build each block from scratch with more complex tools.

3. Enhanced Functionality:

- ❖ Scripting languages often come with built-in features for common tasks, saving you time and effort. These features can include:
    - ➢ **Interacting with web pages:** Scripting languages like JavaScript can manipulate web pages, making them interactive (like adding animations or allowing user input).
    - ➢ **Working with other programs:** Scripts can control other programs that have built-in scripting capabilities, automating tasks (like scheduling social media posts).
    - ➢ **Common data processing tools:** Scripting languages might have built-in functions for handling data, making it easier to analyze information.

4. Efficiency is not an Issue (for most scripting tasks):

- ❖ Scripting languages might not be the most efficient choice for extremely complex tasks that require maximum speed. This is because of the "compile and run" integration, which can add some overhead compared to the pre-compiled nature of traditional languages.
- ❖ However, for most scripting applications, speed isn't a major concern. The ease of use and faster development time often outweigh the slight efficiency trade-off. Scripting shines in situations where getting something done quickly and easily is more important than squeezing out the absolute fastest performance.

Uses for scripting languages:

1.Traditional scripting

2.Modern scripting

1.Traditional scripting

1.System administration automating everyday tasks, building data reduction tools

2.Controlling applications remotely

3.System and application extensions

4.'Experimental' programming

5.Building command-line interfaces to applications based on C libraries

6.Server-side form on the Web using CGI.

Automating Everyday Tasks:

Imagine a system administrator who spends hours every week resetting passwords for forgetful users. A script can be written to automate this process. The script would connect to the user database, verify the request, and reset the password according to predefined rules. This frees up the administrator's time for more complex tasks.

Building Data Reduction Tools:

Think of a company that receives daily sales data from hundreds of stores. A script can be written to analyze this data, calculating total sales, identifying top-selling items, and highlighting any unusual trends. This summarized information is much easier to understand than looking at raw data from hundreds of sources.

Controlling Remotely:

Imagine having multiple servers in different locations. A script can be written to check the status of these servers remotely, ensuring they are running smoothly. If a server goes down, the script can even send an alert or attempt to restart it automatically.

System and Extension Examples:

- ❖ **Adding a custom menu to an existing program:** A script could be used to create a menu within a complex software program, providing users with quick access to frequently used functions.
- ❖ **Extending a web browser's functionality:** A script could be written to add features to a web browser, like automatically blocking certain types of ads or translating web pages on the fly.

Experimental Programming:

Scripting languages are often used as a "scratchpad" for programmers. They can quickly write and test new code ideas or algorithms in a scripting language before investing time in a more complex language for a larger project. It's like trying out a recipe with readily available ingredients before committing to a complex meal.

Command-Line Interfaces (CLIs):

While most users interact with computers through a mouse and graphical interfaces, scripting languages can be used to create text-based interfaces. These can be simpler to develop and can be useful for automating tasks or for experienced users who prefer keyboard shortcuts.

Server-Side Scripting (CGI):

When you fill out a form on a website, for example, a contact form or a search bar, the information you submit needs to be processed somehow. In the early days of the web, scripting languages like CGI were commonly used on web servers to handle this processing. The script would receive the submitted data, perform any necessary actions (like storing it in a database or sending an email), and potentially generate a response to be displayed on the webpage.

While CGI is less common today, it provides a good historical example of how scripting languages were used in web development. Modern web development often uses different scripting languages and frameworks for handling form submissions and other web interactions.

Traditional Scripting Languages (Open Source):

- ❖ **Open Source languages** are a cornerstone of traditional scripting. Their code is freely available for anyone to see, use, and modify. This allows for a collaborative development process where programmers can contribute improvements and share best practices.
- ❖ **Perl and TCL** are two prominent examples of traditional scripting languages. They were among the first languages to be widely adopted for scripting tasks, especially in system administration. While other languages like Python have become more popular in recent years, Perl and TCL are still used in some environments.

2.Modern scripting

1. Visual scripting.
2. Using scriptable components - macros for desktop applications and compound documents
3. Client-side and server-side Web scripting.

Visual scripting:

Visual scripting is the process of constructing a graphical interface from a collection of visual objects ('controls' in Microsoft 'widgets' in Tcl/Tk and Perl-Tk), which have properties (attributes), e.g. foreground and background colours , text on a button etc. that can be set by a program written in an appropriate language Some objects(e.g button) respond to external events such as mouse clicks and the response to each action is defined by a script.

Scriptable components:

- They are software components designed to be controlled by scripting languages.

- This means they expose an "interface," which is like a set of instructions on how to interact with them using a scripting language.
- Visual components: Buttons, menus, and other interactive elements in applications like Visual Basic can be scripted to change behavior (e.g., changing button text based on user actions).
- Embedded objects: Imagine a chart in a word processing document. Scripting could be used to update the chart data dynamically.
- Component objects: These are larger functionalities within an application. Examples include spell checkers, database connections, or internet access functions. Scripting can be used to customize their behavior.
- Model elements: Think of a website. Scripting languages can manipulate elements on the page to create dynamic interactions.
- Scriptable objects make software development more efficient and flexible. Developers can reuse components and customize their behavior using scripts instead of writing everything from scratch.
- This approach is becoming increasingly important in modern software due to its modularity and ease of customization

The Component Object Model (COM) is a software development standard created by Microsoft in 1993. It's a way to create software components that can interact with each other, regardless of the programming language they're written in. Here are some key points about COM:

- **Platform-independent:** COM components can work on different operating systems as long as they are compatible with the COM standard.
- **Distributed:** COM allows components to be spread across different computers on a network and still communicate with each other.
- **Object-oriented:** COM components are built using the principles of object-oriented programming, which helps with code organization and reusability.

How COM works:

- **Interfaces:** The core of COM is the concept of interfaces. These are like contracts that define what a component can do, without specifying how it does it. This allows different programming languages to interact with the component as long as they understand the interface.
- **Implementation:** The actual code that implements the functionality of the component is separate from the interface. This separation allows for flexibility and easier development.
- **Clients and Servers:** A COM component can act as a client (requesting services) or a server (providing services) to other components.

Benefits of COM:

- **Code reusability:** COM components can be reused in different applications, saving development time.
- **Interoperability:** COM allows components written in different languages to work together.
- **Modular design:** COM promotes a modular approach to software development, making it easier to maintain and update.

Web scripting

The Web is one of the most fertile areas for the application of languages at the present time, and readers of the popular technical press probably believe that is solely concerned with generating interactive Web pages. Web divides into three areas: (i) forms, (ii) creating pages with enhanced visual effects and user interaction and (iii) pages 'on the from material held in a database.

Processing Web forms

Early Web and Forms:

- In the beginning, HTML forms were the primary way for user interaction. Users could enter data in text boxes, radio buttons, etc.
- However, these forms were limited. Submitting the form sent the data to the server, and a whole new page had to be generated based on the user input.

Server-Side Processing (CGI):

- **This traditional approach involved:**
  - User submits the form.
  - Data is sent to the server.
  - A server-side script (often written in Perl, a popular choice for system administrators) processes the data.
  - The script generates a new HTML page based on the processed data.
  - The new page is sent back to the user's browser.
- **This process required:**
  - Decoding form data.
  - Text manipulation to build the response page.
  - Potentially, system access for other tasks (like database connections).

Client-Side Scripting (JavaScript/JScript or VBScript):

- As an alternative, client-side scripting languages like JavaScript or VBScript emerged.
- These languages allowed some processing to happen directly within the user's web browser.
- For example, a script could validate form data (like checking if an email address is entered correctly) before sending it to the server. This reduces unnecessary server load and potentially improves user experience by providing immediate feedback.
- Compatibility note: JavaScript/JScript worked in both Netscape Navigator and Internet Explorer, while VBScript was specific to Internet Explorer.

Dynamic web pages are the opposite of static web pages. Here's a breakdown of the key differences:

- Static Web Pages: Imagine a brochure - it displays the same information every time you look at it. Static web pages are similar. They are created with HTML and display the same content for every visitor.
- Dynamic Web Pages: These are like interactive brochures that can change based on the user or situation. Here's how they achieve dynamism:
    - Server-side Scripting: When a user requests a dynamic web page, the server doesn't just send a pre-made HTML file. Instead, it runs a script (often in languages like PHP, Python, or ASP.NET) that generates the HTML content on the fly. This content can be customized based on factors like user input (e.g., search queries in a search engine), user data (e.g., personalized content based on login information), or even real-time data (e.g., stock prices).
    - Client-side Scripting: Sometimes, dynamic behavior can also be achieved using Javascript running directly in the user's web browser. This can involve things like form validation (checking if an email is entered correctly) or updating parts of the page without reloading the entire thing (e.g., refreshing a chat window).

Benefits of Dynamic Web Pages:

- More engaging user experience: Dynamic pages can provide a more interactive and personalized experience for users.
- Up-to-date content: Information can be displayed in real-time or based on the latest data.
- Scalability: Dynamic pages can handle a large number of users more efficiently than static pages.

Examples of Dynamic Web Pages:

- Social media platforms (like Facebook or Twitter)
- E-commerce websites (product details and shopping carts)
- Online banking applications
- News websites with constantly updated content

Perl(Practical Extraction and Report Language):

Perl is a high-level, interpreted programming language known for its powerful text processing capabilities, extensive libraries, and flexibility. Originally developed by Larry Wall in the late 1980s, Perl has evolved into a mature and versatile language used in various domains such as system administration, web development, network programming, and bioinformatics.

1. **Syntax and Features:**
   - ❖ Perl syntax is characterized by its flexibility and expressiveness, allowing developers to write concise and readable code. It offers support for procedural, object-oriented, and functional programming paradigms.
   - ❖ Perl features a rich set of built-in functions and operators for tasks such as string manipulation, file handling, regular expressions, and data structure manipulation.
   - ❖ Perl supports dynamic typing, automatic memory management, and automatic conversion between different data types.

2. Text Processing:

- ❖ One of Perl's strongest features is its powerful text processing capabilities. It provides native support for regular expressions, making it well-suited for tasks involving pattern matching, searching, and substitution in text data.
- ❖ Perl's regular expression engine is highly optimized and offers a wide range of features for complex pattern matching and manipulation.

3. System Administration:

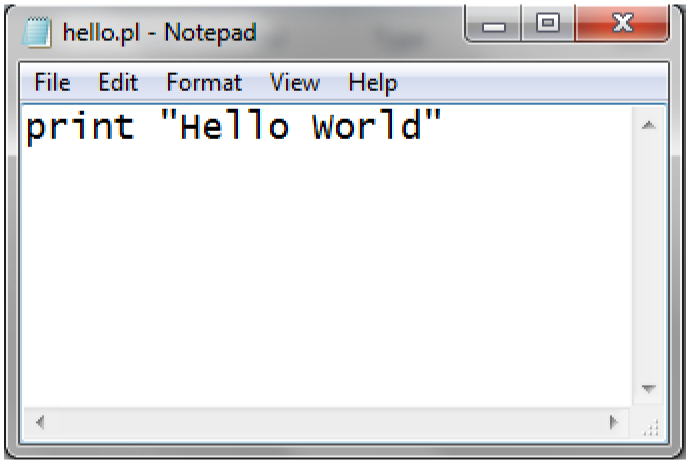
- ❖ Perl is widely used in system administration and automation tasks due to its ability to interact with system resources, files, processes, and network services.
- ❖ Many Unix-like operating systems include Perl as part of their standard installation, making it a popular choice for writing system scripts and utilities.

4. Web Development:

- ❖ Perl was one of the first programming languages used for web development. It gained popularity in the early days of the web for its ability to generate dynamic content and handle web server interactions.
- ❖ While its usage in web development has declined in favor of newer languages and frameworks, Perl still powers many legacy web applications and continues to be used for specific tasks such as CGI scripting

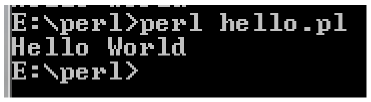
Hello World Program Execution:

1. **Write Perl Code:**Use a text editor to write Perl code in a plain text file. Save the file with a .pl extension, which indicates that it contains Perl code.



```
print "Hello World"
```

2. **Execute the Perl Script:**
   - ❖ Open a command prompt or terminal window on your computer.
   - ❖ Navigate to the directory where the Perl script file is located using the cd command (change directory).
   - ❖ Once you are in the correct directory, you can execute the Perl script by typing perl followed by the name of the Perl script file.
   - ❖ For example, if your Perl script file is named hello.pl, you can execute it by typing perl hello.pl and pressing Enter.
   - ❖ The Perl interpreter will read the Perl code from the script file, interpret it, and execute the instructions as specified in the code.



```
E:\perl>perl hello.pl
Hello World
E:\perl>
```

Names and Values:

1. **Variable Names (identifiers):**

❖ Variable names in Perl must begin with a letter or an underscore (_), followed by any combination of letters, digits, and underscores.

- ❖ Variable names are case-sensitive, meaning that $var, $Var, and $VAR are considered distinct variables.
- ❖ It's a good practice to choose descriptive and meaningful names for variables to make your code more readable and maintainable.

2. Assigning Values:

- ❖ Values can be assigned to variables using the assignment operator (=).
- ❖ Perl is a dynamically typed language, so you don't need to declare the type of a variable before assigning a value to it.
- ❖ Values can be literals (such as numbers, strings, or boolean values) or the result of expressions or function calls.

Perl distinguishes between singular and plural nouns (names). 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).
In Perl, variable names begin with special characters like '$' for scalar values, '@' for lists, '%' for associative arrays, and '&' for subroutines, making it easy to understand the type of data each variable holds.

my ,our and local keywords:

1. **my:** Declares a lexically scoped variable. Variables declared with "my" are only visible within the block in which they are declared. They have a limited scope and are typically used for temporary or local variables.
   Example:
```
sub example {
   my $local_var = 10;
   print $local_var;  # This will print 10
}
```
2. **our:** Declares a package (global) variable. Variables declared with "our" are accessible from any part of the package in which they are declared. They have a global scope within the package and can be accessed by other functions or modules within the same package.
   Example:
```
 package MyPackage;
 our $global_var = 20;
 sub example {
    print $global_var;  # This will print 20
 }
```
3. **local:** Temporarily assigns a new value to a global variable within the current scope. Unlike "my" and "our", "local" does not create a new variable; it only modifies the value

of an existing global variable within the current scope. Once the scope exits, the original value of the global variable is restored.

Example:

```
our $global_var = 30;

sub example {
    local $global_var = 40;
    print $global_var;  # This will print 40
}
print $global_var;  # This will print 30
```

Scalar data:

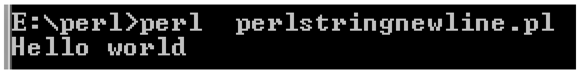
Strings:

- ❖ Strings in Perl are sequences of characters enclosed within single quotes ('') or double quotes ("").
- ❖ They can contain letters, numbers, special characters, and whitespace.
- ❖ Perl provides various operators and functions for string manipulation, such as concatenation (.), interpolation, substring extraction, and pattern matching with regular expressions.

Example:

```
my $str1 = 'Hello';
my $str2 = "world";
my $concat = $str1 . ' ' . $str2;
print "$concat\n";
```

Output:



```
E:\perl>perl  perlstringnewline.pl
Hello world
```

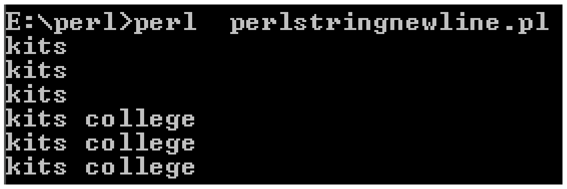
Using the q (quote) and qq (double quote) operators, which allow you to use any character as a quoting character. For example:

- ❖ q/any string/ or q{any string} are equivalent to 'any string'
- ❖ qq/any string/ or qq(any string) are equivalent to "any string"

Example:

```
my $a=q/kits/;
my $b=q{kits};
my $c='kits';
my $x=qq/kits college/;
my $y=qq{kits college};
my $z="kits college";
print "$a\n";
print "$b\n";
print "$c\n";
print "$x\n";
print "$y\n";
print "$z\n";
```

Output:



```
E:\perl>perl  perlstringnewline.pl
kits
kits
kits
kits college
kits college
kits college
```

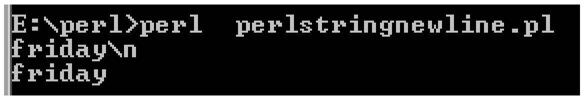
In Perl, the strings 'friday\n' and "friday\n" are different due to how they are interpreted:

1. **'friday\n':** This is a single-quoted string literal. In single-quoted strings, escape sequences like \n are not interpreted. Therefore, 'friday\n' represents the characters 'f', 'r', 'i', 'd', 'a', 'y', '', and 'n'. The backslash (\) is treated as a literal character, not as an escape sequence. So, the string will print as "friday\n" literally, with the backslash and 'n' characters visible.
2. **"friday\n":** This is a double-quoted string literal. In double-quoted strings, escape sequences like \n are interpreted as special characters. Therefore, "friday\n" represents the characters 'f', 'r', 'i', 'd', 'a', 'y', and a newline character. The \n sequence is interpreted as a newline character, causing the string to end after "friday" and start a new line. So, when printed, "friday\n" will display "friday" followed by a newline.

Example:

```
my $a='friday\n';
my $b="friday\n";
print "$a\n";
print "$b";
```

Output:



```
E:\perl>perl  perlstringnewline.pl
friday\n
friday
```

<u>Numbers:</u>

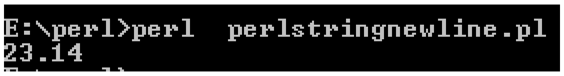
- ❖ Numbers in Perl can be integers or floating-point numbers.
- ❖ They can be represented in decimal, hexadecimal (with a leading '0x'), octal (with a leading '0'), or binary (with a leading '0b') formats.

- ❖ Perl supports basic arithmetic operations such as addition (+), subtraction (-), multiplication (*), division (/), and modulus (%).

Example:

```
my $a=10;
my $b=13.14;
print $a+$b;
```

Output:



```
E:\perl>perl  perlstringnewline.pl
23.14
```

Perl is a loosely typed language, meaning variables do not have strict data types. They can hold either strings or numbers, and Perl automatically converts between them as needed during operations

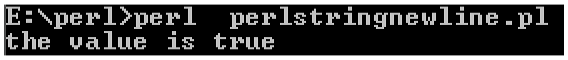
Boolean values:

Perl adopts the simple rule that numeric zero, "0" , the special value undef (indicating an uninitialized variable) and the empty string ("") mean false, and anything else means true

Example:

```
$a=10;
$b="";
if($a){
print "the value is true";
}
if($b) #this is empty and it won't be executed
{
print "the value is false";
}
```

Example:



```
E:\perl>perl  perlstringnewline.pl
the value is true
```

<u>Variables and assignment</u>

Assignment:

- ❖ Perl uses the same = symbol as C for assignment.
- ❖ The assignment statement itself returns a value - the value being assigned.

- This allows for chaining assignments like in the example: $b = 4 + ($a = 3).
  - $a = 3 assigns 3 to $a and returns 3.
  - $b = 4 + 3 uses the returned value (3) from the first assignment.

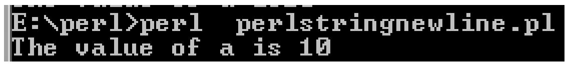
String Interpolation:

- Perl allows embedding the value of a scalar variable within a double-quoted string.
- No special syntax is needed if the variable name starts with $ (e.g., $a = "Burger").
- The variable's value is directly inserted into the string.

Example:

```
$a=10;
print "The value of a is $a"
```

Output:



```
E:\perl>perl  perlstringnewline.pl
The value of a is 10
```

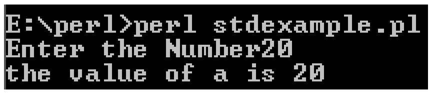
<u><STDIN></u>

It is used to read input from the standard input stream, typically provided by the user via the keyboard.

Example:

```
print "Enter the Number";
$a=<STDIN>;
print "the value of a is $a"
```

Output:



```
E:\perl>perl stdexample.pl
Enter the Number20
the value of a is 20
```

<u>Scalar expressions:</u>

Scalar data items (whether literals or values of variables) are combined into expressions using operators.

Operators:

1. Arithmetic Operators
2. String Operators
3. Comparison Operators
4. Logical Operators
5. Bitwise Operators
6. Conditional Operators

1. **Addition (+)**: Adds two operands.
   Example: $sum = $a + $b (adds values in $a and $b and stores the result in $sum).
2. **Subtraction (-)**: Subtracts the second operand from the first.
   Example: $difference = $x - $y (subtracts $y from $x and stores the result in $difference).
3. **Multiplication (*)**: Multiplies two operands.
   Example: $product = $width * $height (multiplies $width and $height and stores the result in $product).
4. **Division (/)**: Divides the first operand by the second.
   Example: $average = $total / $count (divides $total by $count and stores the result in $average).
   Note: Division by zero results in an error.
5. **Modulus (%)**: Gives the remainder after division.
   Example: $remainder = 10 % 3 (gives the remainder of 10 divided by 3, which is 1, and stores it in $remainder).
6. **Exponent (**)**: Raises the first operand to the power of the second.
   Example: $area = $radius ** 2 (squares the value in $radius and stores the result in $area).

Increment and Decrement:

1. **Increment (++)**: Increases the value of a variable by 1.
   Pre-increment (++$a): Increments the value and then uses the new value. Post-increment ($a++): Uses the current value and then increments. (The difference is subtle but can matter in specific situations.)
2. **Decrement (--)**: Decreases the value of a variable by 1. Similar to increment, it can be pre-decrement (--$b) or post-decrement ($b--).

Combined Assignment Operators:

Perl provides shorthand operators that combine assignment with the operation.

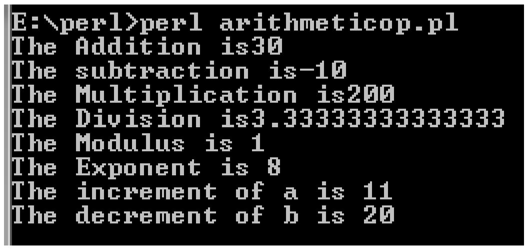
Example: += adds and assigns, -= subtracts and assigns, etc.
$x += 5 is equivalent to $x = $x + 5.

Program:

```
$a=10;
```

```
$b=20;
print "The Addition is".($a+$b)."\n";
print "The subtraction is".($a-$b)."\n";
print "The Multiplication is".($a*$b)."\n";
print "The Division is" .(10/3)."\n";
print "The Modulus is ".(10%3)."\n";
print "The Exponent is ".(2**3)."\n";
print "The increment of a is ".(++$a)."\n";
print "The decrement of b is ".($b++)."\n"
```

Output:



```
E:\perl>perl arithmeticop.pl
The Addition is30
The subtraction is-10
The Multiplication is200
The Division is3.33333333333333
The Modulus is 1
The Exponent is 8
The increment of a is 11
The decrement of b is 20
```

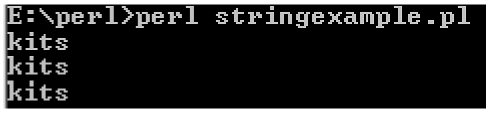
2. String Operator:

'x' operator is used to repeat a string a certain number of times.

Example:

```
$a="kits\n" x 3;
print $a;
```

Output:



```
E:\perl>perl stringexample.pl
kits
kits
kits
```

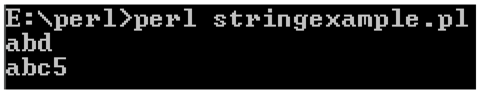
Auto Increment:

When the auto-increment operator (++) is applied to a string, Perl attempts to find the next string in lexicographic order. It does this by incrementing the last character of the string. If the last character is a letter, Perl increments it to the next letter in alphabetical order. If the last character is a digit, Perl increments it as a number. If the last character is a non-alphanumeric character, Perl increments it to the next character in ASCII order.

Example:

```
$a="abc";
print ++$a."\n";
$b="abc4";
print ++$b."\n";
```

Output:



```
E:\perl>perl stringexample.pl
abd
abc5
```

Unary minus:

when the unary minus operator is applied to a string that cannot be interpreted as a number, it simply treats the string as a string and prepends a minus sign to it.

Example:

```
$a='kits';
print -$a;.
```

Output:

```
-kits
```

Comparison Operators:

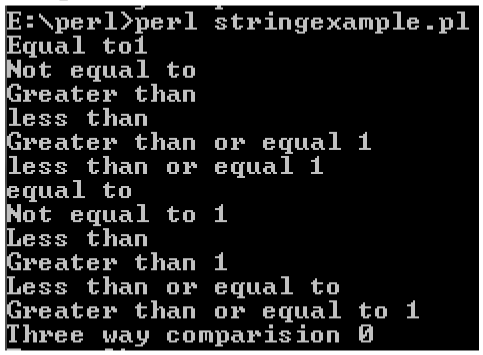
The comparison operators indeed return a value of 1 when the comparison is true, and an empty string (which is considered false in Perl context) when the comparison is false.

1. Equality Operators:
   - ❖ ==: Checks if two values are numerically equal.
   - ❖ !=: Checks if two values are not numerically equal.
   - ❖ eq: Checks if two strings are equal.
   - ❖ ne: Checks if two strings are not equal.
2. Numeric Comparison Operators:
   - ❖ <: Checks if the left operand is numerically less than the right operand.
   - ❖ >: Checks if the left operand is numerically greater than the right operand.
   - ❖ <=: Checks if the left operand is numerically less than or equal to the right operand.
   - ❖ >=: Checks if the left operand is numerically greater than or equal to the right operand.
3. String Comparison Operators:
   - ❖ lt: Checks if the left operand is stringwise less than the right operand.
   - ❖ gt: Checks if the left operand is stringwise greater than the right operand.
   - ❖ le: Checks if the left operand is stringwise less than or equal to the right operand.
   - ❖ ge: Checks if the left operand is stringwise greater than or equal to the right operand.
4. Three-way Comparison Operator:
   - ❖ <=>: Compares two numeric values and returns -1 if the left operand is less than the right, 0 if they are equal, and 1 if the left operand is greater than the right.

Program:

```
$a=10;
$b=10;
print "Equal to".($a==$b),"\n";
print "Not equal to".($a!=$b)."\n";
print "Greater than".($a>$b)."\n";
print "less than".($a<$b)."\n";
print "Greater than or equal ".($a>=$b)."\n";
print "less than or equal ".($a<=$b)."\n";
$x="kits";
$y="college";
print "equal to".($x eq $y)."\n";
print "Not equal to ".($x ne $y)."\n";
print "Less than ".($x lt $y)."\n";
print "Greater than ".($x gt $y)."\n";
print "Less than or equal to ".($x le $y)."\n";
print "Greater than or equal to ".($x ge $y)."\n";
print "Three way comparison ".(10<=>10)
```

Output:



```
E:\perl>perl stringexample.pl
Equal to1
Not equal to
Greater than
less than
Greater than or equal 1
less than or equal 1
equal to
Not equal to 1
Less than
Greater than 1
Less than or equal to
Greater than or equal to 1
Three way comparision 0
```

Logical Operators:

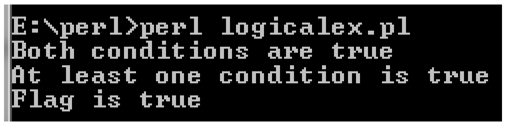
Perl provides logical operators to perform boolean operations.

1. Logical AND (&&): Returns true if both operands are true.
2. Logical OR (||): Returns true if at least one of the operands is true.
3. Logical NOT (!): Returns true if the operand is false and false if the operand is true.

Example:

```
my $num1 = 10;
my $num2 = 5;
if ($num1 > 5 && $num2 < 10)
{
   print "Both conditions are true\n";
}
else
{
   print "At least one condition is false\n";
}
my $str1 = "apple";
my $str2 = "banana";
if ($str1 eq "apple" || $str2 eq "banana")
{
   print "At least one condition is true\n";
}
else
{
   print "Both conditions are false\n";
}
my $flag = 1;
if (!$flag) {
   print "Flag is false\n";
} else {
   print "Flag is true\n";
}
```

Output:



```
E:\perl>perl logicalex.pl
Both conditions are true
At least one condition is true
Flag is true
```

Conditional Operators:

A conditional expression is one whose value is chosen from two alternatives at run-time depending on the outcome of a test.

Example:

```
$a=100;
print $a= ($a < 0) ? 0 : $a;
```

Output:

```
100
```

<u>Control structures</u>

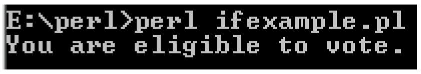
Control structures are fundamental building blocks that dictate the flow of execution within a program. They provide a way to make decisions based on conditions and repeat code blocks as needed, allowing you to create programs that respond to different situations and perform tasks efficiently.

if:

Example:

```
my $age = 25;
if ($age >= 18) {
  print "You are eligible to vote.\n";
}
```

Output:



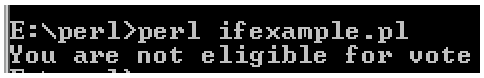
```
E:\perl>perl ifexample.pl
You are eligible to vote.
```

if else:

Example:

```
my $age = 15;
if ($age >= 18)
{
print "You are eligible to vote.\n";
}
else
{
print "You are not eligible for vote";
}
```

Output:



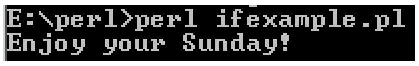
```
E:\perl>perl ifexample.pl
You are not eligible for vote
```

elsif (else if):

Example:

```
my $day = "Sunday";
if ($day eq "Saturday")
{print "It's the weekend!\n";
}
elsif ($day eq "Sunday")
{print "Enjoy your Sunday!\n";
}
else
{print "Back to work tomorrow.\n";}
```

Output:



```
E:\perl>perl ifexample.pl
Enjoy your Sunday!
```

unless:

Example:

```
my $is_registered = 0;
unless ($is_registered) {
  print "Please register to participate.\n";
}
```

Output:

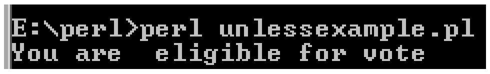
```
Please register to participate.
```

This code executes the block only if $is_registered is false (0).

unless-else:

```
my $a = 25;
unless ($a<=18) {
  print "You are  eligible for vote\n";
}
else
{
print "You are not eligible for vote";
}
```

Output:



```
E:\perl>perl unlessexample.pl
You are  eligible for vote
```

Repetition (Looping):

While:

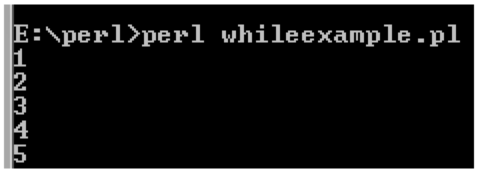
- ❖ The while loop repeatedly executes a code block as long as a specified condition remains true.
- ❖ The condition is evaluated at the beginning of each iteration.
- ❖ Once the condition becomes false, the loop terminates.

Example:

This code prints numbers from 1 to 5 using a while loop that continues as long as $a is less than or equal to 5.

```
my $a=1;
while($a<5)
{
print "kits college\n";
$a++;
}
```

Output:



```
E:\perl>perl whileexample.pl
1
2
3
4
5
```

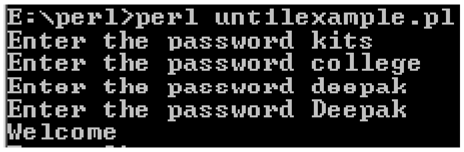
Until loop:

- ❖ The until loop is the opposite of while.
- ❖ It executes a code block repeatedly until a specified condition becomes true.
- ❖ The condition is evaluated at the beginning of each iteration.
- ❖ Once the condition becomes true, the loop terminates.

Example:

```
my $a="";
until($a eq "Deepak")
{
print "Enter the password ";
chomp($a=<STDIN>)
}
print "Welcome";
```

Output:



```
E:\perl>perl untilexample.pl
Enter the password kits
Enter the password college
Enter the password deepak
Enter the password Deepak
Welcome
```

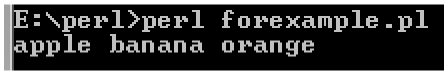
For loop:

- ❖ The for loop provides two ways to iterate:
  - ➢ Over a list of elements (@list).
  - ➢ Using a counter variable with a start, end, and increment value.
- ❖ In the list iteration, $_ refers to the current element in the list.

Example-1:

```
my @fruits = ("apple", "banana", "orange");
for (@fruits) {
  print "$_ ";  # $_ refers to the current element in the list
}
print "\n";
```

Output:



```
E:\perl>perl forexample.pl
apple banana orange
```

Example- 2:

```
# Iterating with a counter
for (my $i = 1; $i <= 3; $i++) {
  print "$i ";
}
print "\n";
```

Output:

1 2 3

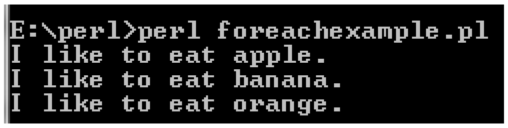
foreach loop:

The foreach loop is a convenient way to iterate through the elements of a list or hash in Perl. It provides a concise syntax for processing each item and is often preferred over traditional for loops when working with collections.

Example:

```
my @fruits = ("apple", "banana", "orange");
foreach my $fruit (@fruits) {
  print "I like to eat $fruit.\n";
}
```

Output:



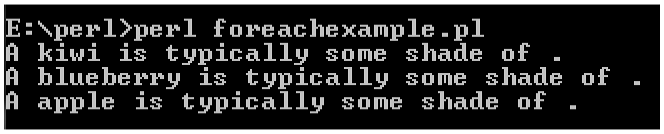
```
E:\perl>perl foreachexample.pl
I like to eat apple.
I like to eat banana.
I like to eat orange.
```

Iterating through Hashes:

Example:

```
my %colors = ("red" => "apple", "green" => "kiwi", "blue" => "blueberry");
foreach my $fruit (values %colors) {
  print "A $fruit is typically some shade of $color.\n";  # Use $color outside the loop for
efficiency
}
```

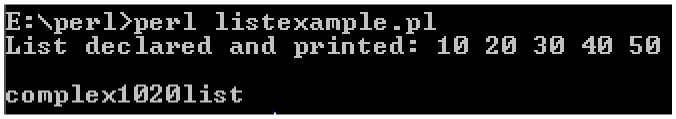
Output:



```
E:\perl>perl foreachexample.pl
A kiwi is typically some shade of .
A blueberry is typically some shade of .
A apple is typically some shade of .
```

list:

The list is a sequence of scalar values. However, the list is not a data structure in Perl. There are limited operations that can be performed on the list in Perl. Since no variable refers to this list, lists cannot be used for operations other than printing.
Example:
(10, 20, 30);
("this", "is", "a", "list", "in", "perl");
Example:
print("List declared and printed: ");
print join(' ', 10, 20, 30, 40, 50);
print "\n\n" ;
# Complex lists
print("complex", 10, 20, "list");
print("\n\n");
Output:



```
E:\perl>perl listexample.pl
List declared and printed: 10 20 30 40 50

complex1020list
```

Arrays:

Arrays are ordered collections of elements that can hold various data types (numbers, strings, even other arrays or hashes). They are a fundamental data structure in Perl, allowing you to group and manage related data efficiently.
The name of such a variable always starts with an @, e.g. @days_of_week
The association between arrays and lists is a close one: an array stores a collection, and a list is a collection, so it is natural to a list to an array,
e.g. @rainfa11 = (1.2, 0.4, 0.3, 0.1, 0, 0, 0);
A list can occur as an element of another list.
Example:
@foo = (1,2, 3, "string");
@foobar = (4, 5, @foo, 6);
gives foobar the value (4,5,1,2,3,"string",6)

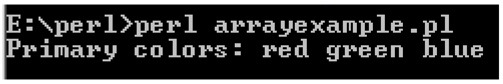
Using the qw function: This function is useful for creating arrays with multiple whitespace-separated elements:

Example:

my @colors = qw(red green blue);
print "Primary colors: ";
foreach my $color (@colors)
{
  print "$color ";
}

print "\n";

Output:



```
E:\perl>perl arrayexample.pl
Primary colors: red green blue
```

Accessing elements:

Example:

@fruits=("apple","banana","orange");

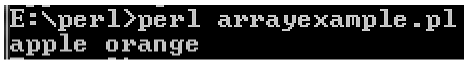
my $first_fruit = $fruits[0]; # $first_fruit will be "apple"

my $last_item = $fruits[-1];   # $last_item will be "blue" (negative index for the last element)

print $first_fruit." ";

print $last_item;

Output:



```
E:\perl>perl arrayexample.pl
apple orange
```

Perl provides a rich set of built-in functions for manipulating arrays:

1. **Adding elements:**
   - ❖ push(@array, element): Adds an element to the end of the array.
   - ❖ unshift(@array, element): Adds an element to the beginning of the array.
2. **Removing elements:**
   - ❖ pop(@array): Removes and returns the last element from the array.
   - ❖ shift(@array): Removes and returns the first element from the array.
3. **Sorting:**
   - ❖ sort @array: Sorts the elements of the array in ascending order (by default).
   - ❖ You can customize sorting order using additional arguments to sort.
4. Arrays are dynamic in size. You can add or remove elements as needed.
5. Perl provides functions to check the size of an array (scalar @array) or get the index of the last element ($#array).
6. You can create multidimensional arrays (arrays of arrays) to represent complex data structures.

Slice:

Slicing an array in Perl allows you to extract a specific portion of elements from an existing array.

1. Using Square Brackets with Start and End Indexes:

This method provides a concise way to extract a subset of elements based on their index positions.

Syntax:

```
@sliced_array = @original_array[@start_index..@end_index];
```

Example:

```
my @fruits = ("apple", "banana", "orange", "mango", "pineapple");
my @sliced_fruits = @fruits[1..3];  # Extract elements from index 1 (inclusive) to 3 (exclusive)
print "@sliced_fruits";  # Output: @sliced_fruits: banana orange
```

Hashes:

hashes (also sometimes called associative arrays) are powerful data structures used to store collections of key-value pairs. Unlike arrays where elements are accessed by numerical indexes, hashes provide a more flexible way to manage data using descriptive keys.

Creating Hashes:

You can create hashes using curly braces {}, separating keys and values with colons =>:

Example:

```
my %fruits = ("apple" => "red", "banana" => "yellow", "orange" => "orange");
```

The percent sign % at the beginning indicates it's a hash.

Accessing Elements:

```
my $apple_color = $fruits{"apple"};  # $apple_color will be "red"
```

Iterating over Hashes:

There are two common ways to iterate through the key-value pairs of a hash:

1. **Using foreach:** This loop iterates over the keys of the hash, and within the loop, you can access the corresponding values:

   Example: foreach $fruit(keys %fruits)

   ```
   {
   print "$fruit"." ";
   }
   ```

2. **Using each function:** This function returns a key-value pair at each iteration:

   Example:

   ```
   while (($key, $value) = each %fruits) {
     print "$key: $value\n";
   ```

```
}
```

Hash Operations:

- ❖ exists $hash{$key}: Checks if a specific key exists in the hash.
- ❖ delete $hash{$key}: Removes a key-value pair from the hash.
- ❖ %new_hash = %old_hash: Creates a new hash by copying all key-value pairs from the old hash.

Pattern Matching:

Regular expressions in Perl (regex) are powerful tools for pattern matching and text manipulation. They allow you to search for specific patterns within strings and perform various operations such as finding, replacing, or extracting text based on those patterns.

Some common elements and syntax used in regular expressions in Perl:

1. **Literal Characters:** Regular expressions can include literal characters that match themselves. For example, the regex /hello/ matches the string "hello" exactly.

   Example:
   ```
   my $text = "This is a sample text string.";
   if ($text =~ /z/)
   {
     print "'s' found in the string.\n";
   }
   else
   {
   print "text not found\n";
   }
   if ($text =~ /is/)
   {
     print "'is' found in the string.\n";
   }
   ```
   Output:
   ```
   text not found
   'is' found in the string.
   ```

2. **Metacharacters:** Metacharacters are special characters with a specific meaning in regular expressions. Some common metacharacters include:
   - ❖ . (dot): Matches any single character except newline.

     Example :
     ```
     $text = "This is a string with special characters!"; if ($text =~ /e./)
     {
      print "Matched 'e' followed by any character (except newline)\n";
     }
     ```

- *: Matches zero or more occurrences of the preceding character or group.
  Example:
  ```
  $text = "aaabbbcccc";
  if ($text =~ /a*bc/) {  # Matches "a" zero or more times followed by "bc"
    print "Matched 'a' zero or more times followed by 'bc'\n";
  }
  ```
- +: Matches one or more occurrences of the preceding character or group.
  Example:
  ```
  $text = "color, colour, flavor";
  if ($text =~ /col+or/)
  {
  # Matches "col" one or more times followed by "or"
    print "Matched 'col' one or more times followed by 'or'\n";
  }
  ```

- ?: Matches zero or one occurrence of the preceding character or group.
  Example:
  ```
  $text = "Mr. Smith or Ms. Jones";
  if ($text =~ /Mr\.?\s/) {  # Matches "Mr" followed by optional "." and whitespace
    print "Matched 'Mr' optionally followed by '.' and whitespace\n";
  }
  ```
- ^: Matches the start of the string.
  Example:
  ```
  $text = "Starting with capital";
  if ($text =~ /^Starting/) {  # Matches "Starting" at the beginning of the string
    print "Matched 'Starting' at the beginning of thestring\n";
  }
  ```
- $: Matches the end of the string.
  Example:
  ```
  $text = "String ending with dot.";
  if ($text =~ /dot\.$/) {  # Matches "dot" followed by "." at the end of the string
    print "Matched 'dot.' at the end of the string\n";
  }
  ```
- []: Defines a character class, matches any single character within the brackets.
  Example:
  ```
  $text = "Phone: (123) 456-7890";
  if ($text =~ /ph(o|e)ne/) {  # Matches "ph" followed by either "o" or "e"
    print "Matched 'ph' followed by either 'o' or 'e'\n";
  }
  ```
- |: Alternation, matches either the expression before or after the pipe.

Example:

```
$text = "Red or blue or green";
if ($text =~ /red|blue|green/) {  # Matches "red" or "blue" or "green"
  print "Matched either 'red', 'blue', or 'green'\n";
}
```

- () : Groups expressions together.

Example:

```
$text = "Full name: John Doe";
if ($text =~ /Full\s+name:\s+(.+)/) {  # Capture everything after "Full name:" in $1
  print "Matched full name: $1\n";
}
```

3. **Quantifiers:** Quantifiers specify how many occurrences of a character or group are expected. They follow a character or group and can be one of the following:

- *: Matches the preceding character zero or more times.

Example:

```
$text = "abcaaa"; if ($text =~ /ab*c/)
{
print "Matched 'ab*c' (can have zero or more 'b's)"
} # This will match "abc", "abbc", "abbbc", etc.
```

- +: Matches the preceding character one or more times.

Example:

```
$text = "color, colour, flavor";
if ($text =~ /col+or/)
.{ print "Matched 'col+or' (one or more 'col')" }
# This will match "color" and "colour" (but not "flavor").
```

- ?: Matches the preceding character zero or one time.

Example:

```
$text = "Mr. Smith or Ms. Jones";
 if ($text =~ /Mr\.?\s/)
{
 print "Matched 'Mr' with optional '.' and whitespace"
}
 #This will match "Mr." and "Mr Smith" (but not "Ms. Jones").
```

- {n}: Exactly n occurrences.

Example:

```
$text = "Mississippi";
if ($text =~ /s{2}i/) {  # Matches "ss" (two "s") followed by "i"
  print "Matched 'ss' followed by 'i'\n";
}
```

- ❖ {n,}: At least n occurrences.
  Example:
  ```
  $text = "abracadabbb";
  if ($text =~ /a{3,}bbb/) {  # Matches at least three "a" followed by "bra"
    print "Matched at least three 'a' followed by 'bra'\n";
  }
  ```
- ❖ {n,m}: Between n and m occurrences.
  Example:
  ```
  $text = "color, colourr, coloooor";
  if ($text =~ /col{2,4}or/) {  # Matches "col" two to four times followed by "or"
    print "Matched 'col' two to four times followed by 'or'\n";
  }
  ```

4. **Anchors:** Anchors are used to specify positions within the string. Common anchors include:
   - ❖ ^: Matches the start of the string.
     Example:
     ```
      $text = "Starting with capital";
     if ($text =~ /^Starting/)
     {
     print "Matched 'Starting' at the beginning"
     }#This will only match "Starting with capital" (not "Another string starting with something else").
     ```
   - ❖ $: Matches the end of the string.
     Example:
     ```
     $text = "String ending with dot.";
      if ($text =~ /dot\.$/)
      {
      print "Matched 'dot.' at the end"
      }
      #This will only match "String ending with dot." (not "String with something else").
     ```
   - ❖ \b: Matches a word boundary.
5. **Character Classes:** Character classes specify a set of characters that can match at a particular position in the string. They are enclosed in square brackets [ ] and match any single character within the brackets.
6. **Modifiers:** Modifiers are added after the closing delimiter of the regular expression and affect how the pattern matching is performed. Some common modifiers include:
   - ❖ i: Case-insensitive matching.

- g: Global matching (find all matches).
- m: Treat the string as multiple lines.
- s: Treat the string as a single line.

Shorthand

1. **\d:** Matches any decimal digit (0-9).
   Example:
   ```
    $text = "Phone: 123-456-7890";
   if ($text =~ /phone:\s+\d{3}-\d{3}-\d{4}/)
   {
    print "Matched phone number format"
   }
   # This matches phone numbers with three digits separated by hyphens.
   ```
2. **\w:** Matches any "word" character, which includes alphanumeric characters (a-z, A-Z, 0-9) and the underscore character (_).
   Example:
   ```
   $text = "This is a test_string";
   if ($text =~ /\b\w+\b/)
   {
   print "Matched whole word"
    } # This will match each word individually ("This", "is", "a", "test_string").
   ```
3. **\s:** Matches any whitespace character, including space, tab, newline, carriage return, and form feed.
   Example:
   ```
   $text = "Name\tAge\nCity";
   if ($text =~ /\w+\s+\w+\s+\w+/)
   {
    print "Matched name separated by whitespace"
   }
   #This matches the format "Name Age City" with any amount of whitespace between them.
   ```

Capitalized versions negate the meaning:

1. **\D:** Matches any character that is not a digit.
2. **\W:** Matches any character that is not a "word" character.
3. **\S:** Matches any character that is not whitespace.

Regular expressions in Perl are enclosed in forward slashes /regex/ and can be used with various built-in functions like =~ operator for matching, s/// operator for substitution, and m// operator for pattern matching.

split and join functions:

split and join are two commonly used functions for manipulating strings, particularly for breaking them apart into lists and reassembling them back into strings.

split

The split function in Perl is used to divide a string into a list of substrings based on a specified delimiter.

Syntax:

split /PATTERN/, EXPR, LIMIT

PATTERN: A regular expression that specifies the delimiter. It is enclosed in slashes (/PATTERN/).

EXPR: The string to be split. If omitted, $_ (the default input variable) is used.

LIMIT: (Optional) The maximum number of fields (substrings) to return.

Example:

my $string = "one,two,three,four";

my @words = split /,/, $string;# @words now contains ('one', 'two', 'three', 'four')

my $string = "one,two,three,four";

my @words = split /,/, $string, 3;# @words now contains ('one', 'two', 'three,four') because of the LIMIT

join:

The join function in Perl is used to concatenate the elements of a list into a single string, with a specified separator between each element.

Syntax:

join SEPARATOR, LIST

SEPARATOR: The string to insert between each list element.

LIST: The list of elements to be joined into a single string.

Example:

my @words = ('one', 'two', 'three', 'four');

my $string = join ',', @words;

$string now contains "one,two,three,four"

my @words = ('one', 'two', 'three', 'four');

my $string = join ' and ', @words;

$string now contains "one and two and three and four"

Greedy and non-Greedy quantifiers: Regular expressions (regex) support both greedy and non-greedy (also known as lazy) quantifiers. These quantifiers determine how much of the input string is consumed by the regex during pattern matching.

Greedy Quantifiers

Greedy quantifiers match as much of the input string as possible. The default behavior in Perl regex is greedy.

Examples of Greedy Quantifiers:

1. * : Matches 0 or more times.
2. + : Matches 1 or more times.
3. ? : Matches 0 or 1 time.
4. {n} : Matches exactly n times.
5. {n,} : Matches at least n times.
6. {n,m} : Matches between n and m times.

Example:

```
my $text = "The quick brown fox jumps over the lazy dog.";
$text =~ /(.*)/;
print $1;  # Output: The quick brown fox jumps over the lazy dog.
```

In this example, (.*) is greedy and matches the entire string.

Non-Greedy Quantifiers

Non-greedy quantifiers match as little of the input string as possible. They are indicated by appending a ? to the greedy quantifier.

Examples of Non-Greedy Quantifiers:

1. *? : Matches 0 or more times, but as few as possible.
2. +? : Matches 1 or more times, but as few as possible.
3. ?? : Matches 0 or 1 time, but as few as possible.
4. {n}? : Matches exactly n times.
5. {n,}? : Matches at least n times, but as few as possible.
6. {n,m}? : Matches between n and m times, but as few as possible.

Example:

```
my $text = "The quick brown fox jumps over the lazy dog.";
```

$text =~ /(.*?) /;

print $1; # Output: The

In this example, (.*?) is non-greedy and matches as little of the string as possible before the first space.

Use greedy quantifiers when you want to match as much text as possible, and use non-greedy quantifiers when you need to match the smallest possible amount of text to satisfy the pattern.

subroutines:

Subroutines, also known as functions or methods in other programming languages, are a fundamental concept in Perl for code reusability, modularity, and organization. Here's a breakdown of subroutines in Perl:

What are Subroutines?

- ❖ Subroutines are reusable blocks of code that perform specific tasks.
- ❖ You can define a subroutine once and then call it from different parts of your program whenever you need that functionality.

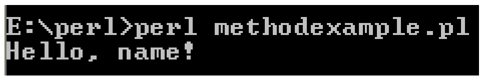
Defining Subroutines:

- ❖ The **sub** keyword is used to define a subroutine.
- ❖ The subroutine name follows the sub keyword and should adhere to Perl's identifier naming conventions (letters, digits, and underscores, starting with a letter).

Example:

```
sub sample {
  print "Hello, name!\n";
}
sample;
```

Output:



```
E:\perl>perl methodexample.pl
Hello, name!
```

UNIT - IV
Advanced perl

Finer points of looping:

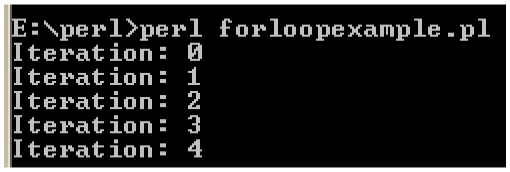
Perl offers several looping constructs for iterating over sequences of data or executing code repeatedly.

for loop:

Example:

```
for ($i = 0; $i < 5; $i++) {
  print "Iteration: $i\n";
}
```

Output:



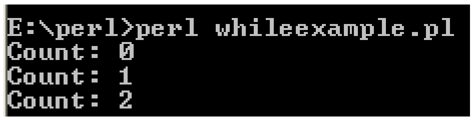
```
E:\perl>perl forloopexample.pl
Iteration: 0
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
```

while loop:

Example:

```
$count = 0;
while ($count < 3) {
  print "Count: $count\n";
  $count++;
}
```

Output:



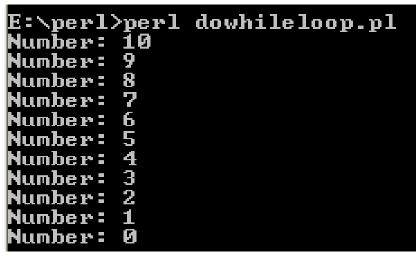
```
E:\perl>perl whileexample.pl
Count: 0
Count: 1
Count: 2
```

do …while loop:

Example:

```
$number = 10;
do {
  print "Number: $number\n";
  $number--;
} while ($number >= 0);
```

Output:



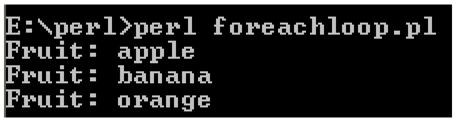
```
E:\perl>perl dowhileloop.pl
Number: 10
Number: 9
Number: 8
Number: 7
Number: 6
Number: 5
Number: 4
Number: 3
Number: 2
Number: 1
Number: 0
```

foreach loop:

```
@fruits = ("apple", "banana", "orange");
```

```
foreach $fruit (@fruits) {
  print "Fruit: $fruit\n";
}
```

Output:



```
E:\perl>perl foreachloop.pl
Fruit: apple
Fruit: banana
Fruit: orange
```

continue:

The continue block in Perl is used in conjunction with while, until, for, and foreach loops. It allows you to specify a block of code that will execute after each iteration of the loop, just before the condition is re-evaluated for the next iteration.

Example:

```
my @numbers = (1, 2, 3, 4, 5, 6);
foreach my $num (@numbers) {
  next if ($num % 2 == 0);  # Skip even numbers
  print "$num\n";
} continue {
}
```

Last, next and redo revisited:

last:

The last statement is used to exit a loop immediately. It is similar to break in other programming languages.

Example:

```
my @numbers = (1, 2, 3, 4, 5, 6);
foreach my $num (@numbers) {
    if ($num > 4) {
        last;  # Exit the loop if the number is greater than 4
    }
    print "$num\n";
}
print "Loop ended.\n";
```

next:

The next statement is used to skip the rest of the current iteration and proceed with the next iteration of the loop. It is similar to continue in other languages.

Example:

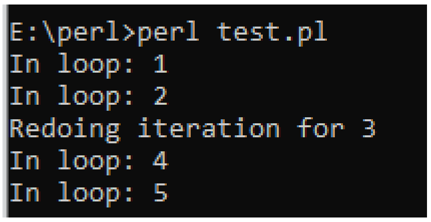
```
my @numbers = (1, 2, 3, 4, 5, 6);
foreach my $num (@numbers) {
    if ($num % 2 == 0) {
        next;  # Skip even numbers
    }
    print "$num\n";
}
print "Loop completed.\n";
```

redo:

The redo statement restarts the current iteration of the loop without evaluating the loop condition again. It is useful when you want to retry the current iteration.

Example:

```
my $i = 0;
while ($i < 5) {
$i++;
if ($i == 3) {
print "Redoing iteration for $i\n";
redo;
}
print "In loop: $i\n";
}
```



```
E:\perl>perl test.pl
In loop: 1
In loop: 2
Redoing iteration for 3
In loop: 4
In loop: 5
```

pack and unpack:

The pack and unpack functions are used to convert between Perl's data structures (like scalars, arrays) and binary data. These functions are essential for handling binary data, network protocols, file formats, and other situations where data needs to be represented in a compact or specific format.

pack Function

The pack function takes a list of values and a template string and converts these values into a binary string(some encoded format) according to the template. The template specifies how each value should be converted.

Syntax:

$binary_data = pack(TEMPLATE, LIST);

TEMPLATE: A string that specifies the format to use for each value in the list.

LIST: The list of values to be packed into a binary string.

Common Template Characters

- ❖ **A**: ASCII string (space padded).
- ❖ **a**: ASCII string (null padded).
- ❖ **H**: Hex string (high nibble first).
- ❖ **h**: Hex string (low nibble first).

- **c**: Signed char (8-bit).
- **C**: Unsigned char (8-bit).
- **s**: Signed short (16-bit).
- **S**: Unsigned short (16-bit).
- **l**: Signed long (32-bit).
- **L**: Unsigned long (32-bit).
- **f**: Single-precision float (32-bit).
- **d**: Double-precision float (64-bit).
- **x**: Null byte (padding).
- **@**: Null byte (absolute positioning).

Example:

```
$i=2;
$f=10.5;
$c='A';
$packed_data=pack("ifA",$i,$f,$c);
print $packed_data."\n";
($test,$test1,$test2)=unpack("ifA",$packed_data);
print "$test,$test1,$test2";
```

unpack Function

The unpack function is the reverse of pack. It takes a binary string and a template and converts the binary string back into a list of values.

Syntax:

```
@list = unpack(TEMPLATE, BINARY_STRING);
```

TEMPLATE: A string that specifies the format to use for each part of the binary string.

BINARY_STRING: The binary string to be unpacked.

Example:

```
($test,$test1,$test2)=unpack("ifA",$packed_data);
print "$test,$test1,$test2";
```

Practical Usage

Packing a Network Packet

```
my $packet = pack("C4n", 192, 168, 1, 1, 8080);
# C4 means 4 unsigned chars for IP address
# n means a 16-bit unsigned short for port number
```

Unpacking a Network Packet

```
my ($a, $b, $c, $d, $port) = unpack("C4n", $packet);
print "IP: $a.$b.$c.$d, Port: $port\n";  # Output: IP: 192.168.1.1, Port: 8080
```

File System:

Perl provides extensive support for interacting with the filesystem. This includes reading from and writing to files, manipulating file and directory structures, and querying file attributes.

Filehandles:

- ❖ A filehandle is a named variable that acts as a connection to a physical file.
- ❖ To read from or write to a file, you first need to open it using the open function. Always check if the file was opened successfully.

File Modes:

- ❖ There are three basic file modes:
  - ➢ <: Read mode - Opens an existing file for reading.
  - ➢ >: Write mode - Opens a file for writing. If the file exists, its contents are overwritten. If it doesn't exist, it's created.
  - ➢ >>: Append mode - Opens a file for appending data to the end. If the file doesn't exist, it's created.

Example:

Opening a File

```
# Opening a file for reading
open(my $fh, '<', 'filename.txt') or die "Cannot open file: $!";
# Opening a file for writing (creates a new file or truncates an existing one)
open(my $fh, '>', 'filename.txt') or die "Cannot open file: $!";
# Opening a file for appending (creates a new file if it doesn't exist)
open(my $fh, '>>', 'filename.txt') or die "Cannot open file: $!";
```

Closing a File

```
close($fh) or die "Cannot close file: $!";
```

Reading from a File

Reading Line by Line

```
open(my $fh, '<', 'filename.txt') or die "Cannot open file: $!";
while (my $line = <$fh>) {
   print $line;
}
close($fh);
```

Reading the Entire File into a String

```
open(my $fh, '<', 'filename.txt') or die "Cannot open file: $!";
my $content = do { local $/; <$fh> };
close($fh);
print $content;
```

Writing to a File

Writing Line by Line

```
open(my $fh, '>', 'filename.txt') or die "Cannot open file: $!";
```

```
print $fh "First line\n";
print $fh "Second line\n";
close($fh);
```

eval:

The eval function is a powerful and versatile tool that allows you to execute Perl code contained in a string at runtime or to trap runtime errors. There are two primary forms of eval:

String form: Executes Perl code contained in a string.

Block form: Evaluates a block of code and catches any runtime errors.

String Form of eval

The string form of eval is used to execute a string of Perl code. This is useful when you need to construct and execute Perl code dynamically.

Example:

```
my $code = '$x = 10; $y = 20; $z = $x + $y;';
eval $code;
print "The result is: $z\n";  # Output: The result is: 30
```

In this example, the string $code contains Perl code that is executed by eval. After execution, the variables $x, $y, and $z are available in the current scope.

Block Form of eval

The block form of eval is used to trap runtime errors within a block of code. This is particularly useful for exception handling.

Syntax:

```
eval {
   # Code that might throw an exception
};
```

Example:

```
eval {
   my $result = 10 / 0;  # Division by zero error
};
if ($@) {
   print "An error occurred: $@\n";
}
```

In this example, the division by zero generates a runtime error. The error is caught by eval, and the error message is stored in $@.

Data structures:

Arrays of arrays:

An array of arrays (also known as a multidimensional array) is essentially an array where each element is a reference to another array. This allows you to create complex data structures like matrices, tables, or any other form of nested lists.

Creating an Array of Arrays:

Using Array References

Example:

```
my @row1 = (1, 2, 3);
my @row2 = (4, 5, 6);
my @row3 = (7, 8, 9);
# Create an array of array references
my @matrix = (\@row1, \@row2, \@row3);
```

In this example, @matrix is an array where each element is a reference to one of the row arrays (@row1, @row2, and @row3).

the \@ syntax is used to create a reference to an array.An array reference is essentially a pointer to an array. Instead of working directly with the array, you work with the reference, which allows you to easily create complex data structures like arrays of arrays.

Direct Initialization

You can also initialize an array of arrays directly:

Example:

```
my @matrix = (
   [1, 2, 3],
   [4, 5, 6],
   [7, 8, 9]
);
```

Here, each sub-array (row) is created using anonymous array references ([...]), which are directly nested within @matrix.

Modifying Elements

You can modify elements in an array of arrays by directly accessing them:

```
$matrix[1][2] = 10;  # Change the third element of the second row to 10
print $matrix[1][2];  # Output: 10
```

Accessing Elements

To access elements in an array of arrays, you need to dereference the array references.

Accessing an Entire Row

```
my $first_row_ref = $matrix[0];
print "@$first_row_ref\n";  # Output: 1 2 3
```

Accessing Individual Elements

To access individual elements, you use double indexing:

```
print $matrix[0][1];  # Output: 2
print $matrix[2][0];  # Output: 7
```

Here, $matrix[0][1] accesses the second element of the first row, and $matrix[2][0] accesses the first element of the third row.

Iterating Over an Array of Arrays

You can iterate over the rows and elements using nested loops.

Iterating Over Rows

```
foreach my $row_ref (@matrix) {
   print "@$row_ref\n";
}
```

Iterating Over Rows and Columns

```
foreach my $row_ref (@matrix) {
   foreach my $element (@$row_ref) {
      print "$element ";
   }
   print "\n";
}
```

Example Program on Arrays of arrays:

```
# Creating a 2D matrix
my @matrix = (
   [1, 2, 3],
   [4, 5, 6],
   [7, 8, 9]
);

# Accessing an element
print "Element at (1, 2): $matrix[1][2]\n";  # Output: 6

# Modifying an element
$matrix[1][2] = 10;
print "Modified element at (1, 2): $matrix[1][2]\n";  # Output: 10

# Iterating over the matrix
foreach my $row_ref (@matrix) {
   foreach my $element (@$row_ref) {
      print "$element ";
   }
   print "\n";
}
```

hashes of hash:

hash of hashes is a complex data structure where each value in a hash is itself a reference to another hash. This allows for the creation of nested associative arrays, which can be used to represent more complex data structures such as records, trees, or tables.

Creating a Hash of Hashes

Using Hash References

Example:

```
# Create individual hashes
my %hash1 = (name => 'John', age => 30);
my %hash2 = (name => 'Jane', age => 25);
my %hash3 = (name => 'Doe', age => 40);
# Create a hash of hash references
my %hoh = (
   person1 => \%hash1,
   person2 => \%hash2,
   person3 => \%hash3,
);
```

In this example:

- ❖ %hash1, %hash2, and %hash3 are individual hashes.
- ❖ \%hash1, \%hash2, and \%hash3 create references to these hashes.
- ❖ %hoh is a hash where each key (person1, person2, person3) points to a reference to another hash.

Direct Initialization

You can also initialize a hash of hashes directly:

Example:

```
my %hoh = (
   person1 => {
      name => 'John',
      age  => 30,
   },
   person2 => {
      name => 'Jane',
      age  => 25,
   },
   person3 => {
      name => 'Doe',
      age  => 40,
   },
);
```

Here, each value in %hoh is an anonymous hash, created using {...}.

Accessing Elements

To access elements in a hash of hashes, you need to dereference the nested hashes.

Accessing a Hash Reference

```
my $person1_ref = $hoh{person1};
print "Person1's name: $person1_ref->{name}\n";  # Output: Person1's name: John
```

Accessing Individual Elements

To access individual elements, you use double dereferencing:

```
print "Person1's age: $hoh{person1}{age}\n";  # Output: Person1's age: 30
print "Person2's name: $hoh{person2}{name}\n";  # Output: Person2's name: Jane
```

Modifying Elements

You can modify elements in a hash of hashes by directly accessing them:

```
$hoh{person1}{age} = 31;  # Change Person1's age to 31
print "Person1's new age: $hoh{person1}{age}\n";  # Output: Person1's new age: 31
```

Iterating Over a Hash of Hashes

You can iterate over the keys and values of a hash of hashes using nested loops.

Iterating Over Top-Level Keys

```
foreach my $person (keys %hoh) {
    print "$person:\n";
    foreach my $attribute (keys %{ $hoh{$person} }) {
        print "  $attribute: $hoh{$person}{$attribute}\n";
    }
}
```

Example: Student Grades

Here's a complete example that demonstrates creating, accessing, and modifying a hash of hashes representing student grades:

```
# Create a hash of hashes for student grades
my %students = (
    alice => {
        math    => 90,
        science => 85,
    },
    bob => {
        math    => 78,
        science => 82,
    },
    carol => {
        math    => 92,
        science => 88,
    },
);
# Accessing elements
print "Alice's math grade: $students{alice}{math}\n";  # Output: 90
# Modifying an element
$students{bob}{science} = 84;
print "Bob's new science grade: $students{bob}{science}\n";  # Output: 84
# Iterating over the hash of hashes
foreach my $student (keys %students) {
    print "$student's grades:\n";
    foreach my $subject (keys %{ $students{$student} }) {
        print "  $subject: $students{$student}{$subject}\n";
    }
}
```

packages:

A package is essentially a namespace in Perl. It's a mechanism to group related code under a unique name, preventing naming conflicts.Think of it as a container for variables, functions, and other symbols.

Use the package keyword followed by the package name:

Example -1:

```
package MyPackage;
our $variable = 10;
sub hello {
    print "Hello from MyPackage\n";
}
```

# Accessing

```
print $MyPackage::variable;
MyPackage::hello();
```

Example -2 :

a.pl

```
package mypack;
our $a = 10;
1;  # Return true value at the end of the file
```

Here if we want to use the value of a from mypack to another program

b.pl

```
require './a.pl';  # Load the file containing the package
print $mypack::a, "\n";  # Access the variable using the fully qualified name
```

Advantages of packages:

1. Group related code for better readability and maintainability.
2. Prevent naming clashes with other parts of your code.
3. Create reusable code components.

Modules:
Modules are files containing Perl code, encapsulated into a package, which can be reused across different scripts and applications.
They usually end with a .pm extension.
A module starts with a package declaration and ends with a true value (usually 1;)
Modules are included in scripts using the use,use implicitly calls import method.
Modules can export functions and variables using Exporter module.
To export symbols, set up @EXPORT or @EXPORT_OK arrays and inherit from Exporter
Example:
```
package MyModule;
use Exporter 'import';
our @EXPORT = qw(hello);
our @EXPORT_OK = qw(goodbye);
sub hello {
   print "Hello\n";
}
sub goodbye {
   print "Goodbye\n";
}
1;
```
Another program:
```
use MyModule;  # Imports hello
hello();
use MyModule qw(goodbye);  # Imports goodbye explicitly
goodbye();
```
- The @ISA array in Perl is used for establishing inheritance in object-oriented Perl. It allows one package to inherit methods and properties from another package.

******package is a namespace for organizing code, while a module is a file that typically contains a package and is designed for reuse and distribution.***

<u>interfacing to the operating system:</u>

Perl was initially created on UNIX and for UNIX environments, leveraging the system calls and features native to UNIX systems. This included file handling, process control, and inter-process communication, among other functionalities.
As Perl grew in popularity, it was ported to other operating systems, including Windows. The goal during these ports was to maintain a similar syntax and functionality for system calls, making Perl scripts as portable as possible across different platforms.
ActiveState, a software company, produced a version of Perl for Windows NT (often referred to as ActivePerl). This implementation included equivalents for many UNIX system calls, allowing Perl to function similarly in Windows environments. This enabled developers to run Perl scripts on both UNIX and Windows systems with minimal modifications.
Example:

ON UNIX:

```
#!/usr/bin/perl
print "This is UNIX\n";
system("ls -l");
```

ON WINDOWS:

```
#!/usr/bin/perl
print "This is Windows\n";
system("dir");
```

In both cases, the system function is used to execute a system command. The command itself differs (ls -l for UNIX and dir for Windows), but the Perl code structure remains the same.
Here we describe a collection of facilities that are common to the UNIX and NT implementations of Perl:

Environment variables:

In Perl, environment variables are managed through a special hash called %ENV. Each key in this hash corresponds to an environment variable name, and the associated value is the value of that environment variable. By accessing %ENV, a script can read and modify environment variables.

Example:

print "PATH is $ENV{PATH}\n";

Perl provides a way to temporarily change the value of an environment variable within a specific scope using the local keyword. The local keyword temporarily changes the value of a global variable (including elements of arrays and hashes) within a block, and restores the original value when the block is exited. This can be particularly useful for making temporary changes to environment variables.

Example:

```
{
   local $ENV{PATH} = "/temporary/path/bin:$ENV{PATH}";
 }
```

File System calls:

In Perl, many functions are available to interface with the operating system, particularly for file system manipulation. These functions often return a value indicating success or failure..
using logical operators like or and || to handle errors gracefully.
chdir $x or die "cannot chdir to $x\n";
OR
chdir $x || die "cannot chdir to $x\n";
Example:
The || operator has higher precedence than or
Examples of the more common calls for manipulating the file system are:

chdir $x	Changes the current working directory to $x.
unlink $x	Deletes the file $x. This is similar to rm in UNIX or delete in NT.
rename ($x, $y)	Renames or moves file $x to $y. Similar to mv in UNIX.
link($x, $y)	Creates a hard link from $x to $y. This is not available in NT.

A hard link is essentially an additional name for an existing file on a file system. Both the original name and the hard link name point to the same data on the disk. If the original file is deleted, the hard link still provides access to the data.

symlink ($x, $y) Creates a symbolic link from $x to $y. Similar to ln -s in UNIX. Not available in NT.

A symbolic link (or symlink) is a special type of file that points to another file or directory. It is a pointer to the name of the file rather than to the actual data on the disk.

mkdir($x, 755)	Creates a directory $x with permissions 0755
rmdir $x	Removes the directory $x.
chmod(644, $x)	Changes the permissions of the file $x to 0644.

Shell commands:
Perl provides two main ways to execute shell commands from within a script: using the built-in system function and using backticks (`) or the qx// operator, often referred to as "quoted execution."
system function:
Runs a command in the system shell.it does not capture the command's output. Instead, it returns the exit status of the command.
The use of && rather than || as the connector, this is because shell commands returns zero to denote success and a non-zero value to denote failure

Example:
```
system("mkdir test_directory") and die “success”
if ($? == -1) {
   print "Failed to execute: $!\n";
} else {
   printf "Command exited with value $?;
```

}

Quoted Execution (Backticks ` or qx//):
Runs a command in the system shell and captures its output. Captures the command's output and returns it as a string.\
Example:
my $output = `ls -l`;
print "The output of 'ls -l' is:\n$output";

exec:
The exec function in Perl is used to replace the current process with a new process, running the specified command. Unlike the system function, which runs a command and then returns control to the Perl script, exec never returns. Once exec is called, the current script stops executing, and the new command takes over the process completely.
Example:
exec "perl -w scr2.pl" or die "Exec failed\n"; # the -w switch is used to enable warnings

Process Control in UNIX and WINDOWS:
Process control in Perl, especially when interfacing with operating systems like UNIX and Windows, involves managing and interacting with processes, executing commands, and controlling their execution flow.
UNIX:
Forking: Creating a new process (child process) using the fork function. This is common in UNIX-like systems.
WINDOWS:
use Win32::Process;
my $child;
Win32::Process::Create($child,"D:\\winnt\\system32\\notepad.exe","notepad temp.txt "
,0, NORMAL_PRIORITY_CLASS, ".") or die "Can't create process: $!\n";

- ❖ 0: Specifies that the child process does not inherit the open file handles of its parent process.

Once the process is created, you can manage it using the methods provided by Win32::Process:
$child->Suspend(); # Suspend the child process
$child->Resume(); # Resume the child process
$child->Wait(INFINITE); # Wait indefinitely for the child process to terminate
$child->Kill($exitcode); # Terminate the child process with a specific exit code

Creating 'Internet-aware' applications

The Internet offers a vast array of information available on various servers such as Web servers, FTP servers, POP/IMAP mail servers, and News servers. While web browsers are typically used to access information on Web and FTP servers, and clients for mail and News servers, "Internet-aware" applications can automate this process without manual intervention.

For example, a Perl application can interact with a web server to perform actions like submitting a query via a web form. This process involves sending the query to a CGI program on the server, which processes the request, retrieves the relevant information, formats it into an HTML page, and returns the page to the client. Similarly, Perl applications can connect to POP3 mail servers to check for unread messages.

Scripting languages like Perl simplify these operations by abstracting their complexities, particularly through specialized modules. The LWP (Library for WWW in Perl) collection of modules is a prime example, as it makes these interactions straightforward and requires minimal code compared to languages like C.

In summary, Perl and its LWP modules enable the creation of powerful "Internet-aware" applications that can automate interactions with various internet services, streamlining tasks that would otherwise require extensive manual effort or complex coding.

LWP::Simple

Creating "Internet-aware" applications in Perl can be done using the LWP::Simple module, which is part of the larger LWP (Library for WWW in Perl) suite. LWP::Simple provides easy-to-use functions for fetching documents from the web without needing to understand the complexities of HTTP protocol handling.

You can install it from CPAN if it's not already available:

```
cpan install LWP::Simple
```

The LWP::Simple module provides several functions for fetching web content. The most commonly used functions are:

get($url): Fetches the document at the specified URL and returns it as a string.

getprint($url): Fetches the document at the specified URL and prints it to standard output.

getstore($url, $file): Fetches the document at the specified URL and stores it in the specified file.

Example: Fetching and Printing Web Page Content

```
use strict;
use warnings;
use LWP::Simple;



# URL of the web page to fetch
```

```
my $url = 'http://www.example.com';
# Fetch the content of the URL
my $content = get($url);
# Check if the content was successfully fetched
if (defined $content) {
   print "Content of $url:\n";
   print $content;
} else {
   die "Couldn't get $url";
}
```

Example: Saving Web Page Content to a File

```
use strict;
use warnings;
use LWP::Simple;
# URL of the web page to fetch
my $url = 'http://www.example.com';
# File to save the content
my $file = 'example.html';
# Fetch the content and store it in the file
my $status = getstore($url, $file);
# Check the status of the operation
if ($status == 200) {
   print "Content saved to $file\n";
} else {
   die "Failed to get $url: HTTP status $status";
}
```

LWP::UserAgent

Purpose: LWP::UserAgent provides a more comprehensive and configurable interface for web requests. It allows detailed control over HTTP requests and responses, making it suitable for more complex web interactions.

Features:

- ❖ Full control over HTTP methods (GET, POST, PUT, DELETE, etc.).
- ❖ Ability to set custom headers, handle cookies, follow redirects, and use proxies.
- ❖ Configurable timeouts and SSL/TLS support.
- ❖ Can handle advanced use cases like authentication, multipart forms, and more.

Common Methods:

- ❖ get($url): Performs a GET request.
- ❖ post($url, \%form_data): Performs a POST request with form data.
- ❖ request($request): Sends a custom HTTP::Request object.
- ❖ cookie_jar($cookie_jar): Manages cookies.

Dirty Hands Internet Programming

Dirty hands" Internet programming in Perl typically refers to the lower-level, hands-on approach of using sockets for network programming, rather than relying on high-level libraries like LWP. Sockets provide a way to connect to other computers over a network and can be used to create both client and server applications.

Sockets in Perl

Perl provides socket programming capabilities through modules such as IO::Socket. This module simplifies the process of creating and using sockets. Here’s a basic introduction to using sockets for network communication in Perl.
Socket is a mechanism used for inter-process communication (IPC). It allows different processes to communicate with each other, either on the same machine or over a network. In Perl, the Socket module provides low-level access to the socket interface, enabling the creation of both client and server applications.

Server:

```
#!/usr/bin/perl -w
use IO::Socket;
use strict;
use warnings;
my $socket = new IO::Socket::INET (
LocalHost => 'localhost',
LocalPort => '6666',
Proto => 'tcp',
Listen => 1,
Reuse => 1,
);
die "Could not create socket: $!n" unless $socket;
print "Waiting for data from the client end\n";
my $new_socket = $socket->accept();
while(<$new_socket>)
{
        print "Data received from user$_";
}
close($socket);
```

Client:

```
use strict;
use warnings;
use IO::Socket;
my $socket = new IO::Socket::INET (
PeerAddr => 'localhost',
PeerPort => '6666',
Proto => 'tcp',
);
die "Could not create socket: $!n" unless $socket;
print "Enter data to send:\n";
my $data = <STDIN>;
chomp $data;
print $socket " '$data'\n";
close($socket);
```

<u>Security issues:</u>

<u>Taint Checking in Perl</u>

When taint checking is enabled, Perl tracks all data coming from outside the program (e.g., user input, environment variables, command line arguments) and marks it as "tainted." Tainted data cannot be used directly in operations that affect the system (e.g., system calls, file operations) without being first sanitized.

Tainted Data

- **Sources of Tainted Data**: Any data that comes from outside the program is considered tainted. This includes:
    - User input from standard input (<STDIN>)
    - Command-line arguments (@ARGV)
    - Environment variables (%ENV)
    - Data read from files

To enable taint checking in a Perl script, you use the -T switch on the command line or include it in the shebang line.

Example Program:

```
#!/usr/bin/perl -T
use strict;
use warnings;
# Get user input
print "Enter a file name: ";
chomp(my $filename = <STDIN>);
# Example of tainted data (user input)
if ($filename =~ /^([a-zA-Z0-9_\-\.]+)$/) {
   $filename = $1;  # Untainting the data by capturing in a regex group
} else {
   die "Invalid file name: $filename\n";
}
# Open the file (safe operation after untainting)
open(my $fh, '<', $filename) or die "Could not open file '$filename': $!\n";
while (my $line = <$fh>) {
   print $line;
}
close($fh);
```

Advantages:

Security: Taint checking prevents malicious input from being executed as system commands or affecting critical files.
Robustness: By ensuring data is validated and sanitized, scripts are more reliable and less prone to unexpected behavior.

Safe module:

Taint checking provides security for code you have written which uses potentially unsafe data from outside. A different situation arises if you have to execute code that comes from an unknown or untrusted source. The Safe module allows you to set up 'Safe objects': each of these is a compartment (or 'sand box') in which untrusted code can be executed in isolation from the rest of your program. A Safe object has a namespace which is rooted at a specific level in the package hierarchy, so the code executed cannot access variables declared and used at a higher level in the hierarchy

UNIT - V

Tool Command Language (Tcl, pronounced "tickle") is a dynamic scripting language that is widely used for rapid prototyping, scripted applications, GUIs, and testing. Developed by John

Ousterhout in the late 1980s, Tcl is designed to be simple, powerful, and easily embeddable into applications.
Features:

Key Features

1. Simplicity and Ease of Learning: Tcl has a straightforward syntax that is easy to learn, making it accessible for beginners while still powerful enough for advanced users.
2. Dynamic Typing and Flexible Syntax: Variables in Tcl are dynamically typed, meaning they can hold any type of data, and the language syntax is highly flexible, allowing for rapid development and prototyping.
3. Cross-Platform Compatibility: Tcl runs on various platforms, including Windows, macOS, and various flavors of Unix/Linux, ensuring that scripts can be executed in diverse environments without modification.
4. Embedding and Extending: Tcl is designed to be embedded into applications and can be extended with new commands written in C, C++, or other languages, making it highly versatile for custom application needs.
5. Event-Driven Programming: Tcl supports event-driven programming, which is particularly useful for developing graphical user interfaces (GUIs) and handling asynchronous I/O operations.
6. Tk Toolkit for GUI Development: Tcl is often used in conjunction with Tk, a powerful toolkit for creating cross-platform GUIs. Tcl/Tk is known for its ease of use and ability to create sophisticated user interfaces with minimal effort.

Tcl is employed in a variety of domains, including:

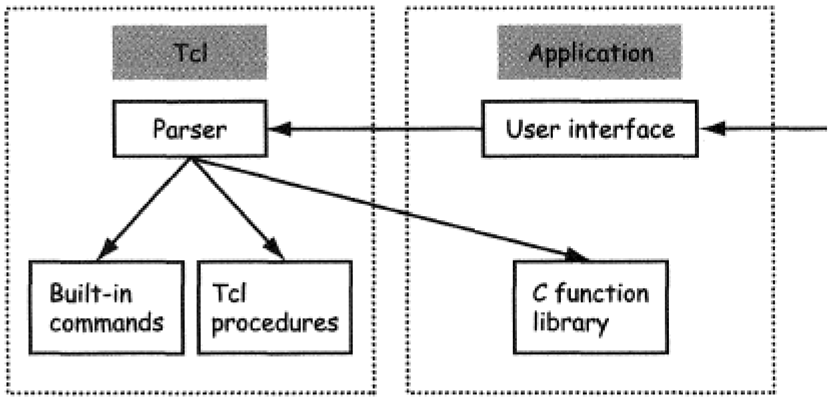
- **Automated Testing**: Its ability to rapidly create test scripts and its cross-platform nature make Tcl ideal for automated testing environments.
- **Network Applications**: Tcl provides excellent support for network programming and is used in developing network tools and services.
- **Embedded Systems**: Due to its small footprint and flexibility, Tcl is often embedded in hardware and software products to provide scripting capabilities.

Tcl structure:
Tcl implements a command-line syntax that will be familiar to users of the UNIX shell, or DOS on a Pc. Each command consists of one or more 'words' separated by whitespace: the first word is the command word (or verb), and the remaining words (if any) are arguments. The command word may identify an in-built command, a user-written Tcl procedure, or a user-supplied external function written in C: either way, the interpreter passes control to the command/procedure/function, making the argument words available to it.

Tcl Interpreter and Command Execution

- **Interpreter's Role**:

- **Parsing**: The interpreter receives user input from the host application.
- **Breaking Input**: It breaks the input into words and identifies the command word.
- **Passing Control**: The command word determines which code to execute (built-in command, user procedure, or external function).



Tcl Syntax:
In Tcl, commands are constructed from a sequence of words separated by whitespace. The first word is the command name, and the subsequent words are its arguments. A word is typically a sequence of characters that excludes spaces or tabs. Within a word, a dollar sign ($) is used to substitute the value of a variable, while a backslash (\) can introduce special characters, such as \n for a newline. Unlike in Perl, the dollar sign in Tcl is a substitution operator, not part of the variable name. Words that contain spaces can be enclosed in double quotes or curly braces. If enclosed in double quotes, the argument value includes all characters between the quotes after any substitutions. When enclosed in curly braces, no substitutions occur, and the argument value is the entire string within the braces. Additionally, a word enclosed in square brackets is treated as a script that gets evaluated immediately, with the argument being the result of that script. This structure allows for flexible and powerful command creation in Tcl.

Variables and data in Tcl:
Variables are created using the set command.
The set command is used to assign values to variables and to retrieve the value of a variable.
Example:

```
set a "kits"  ;# Assigns the string "kits" to the variable a
puts $a           ;# Retrieves and prints the value of a
```

The dollar sign ($) is used for variable substitution to access the value of a variable.
Example:

```
set name "Deepak"
puts "Hello, $name"  ;# Prints "Hello, Alice"
```

append command:

append command is another way to modify the contents of a variable by adding additional text to the end of its current value.

The append command takes a variable name as its first argument and one or more additional strings as subsequent arguments.

It concatenates these strings to the current value of the variable.

Example:

```
set a "Hello"
append a " World!"
puts $a  ;# Prints "Hello World!"
```

expr command:

The expr command evaluates an expression and returns its result.

When expr is called with multiple arguments, it concatenates them into a single string.

Variables in the expression are replaced with their values.

Commands enclosed in square brackets [] are executed, and their results are substituted into the expression.

The concatenated string, with all substitutions made, is then evaluated as an expression.

Example:

```
set a 5
set b 10
set sum_expr "2 + 3"
puts [expr $a + $b]
puts [expr $sum_expr]
puts [expr $a + $b + [expr 2 * 3]]
```

Output:

```
15
5
21
```

Control Flow:

Tcl offers a variety of control flow structures to manage the execution flow of your scripts.

if-else:

This conditional statement allows you to execute different code blocks based on a boolean expression.

Examples:

```
set a 100
if {$a < 20} {
  puts "a is lessthan 20"
} else {
  puts "a is not less than 20"
}
```

if...else if...else Statement:

```
set a 100
if { $a == 10 } {
   puts "Value of a is 10"
} elseif { $a == 20 } {
   puts "Value of a is 20"
} elseif { $a == 30 } {
   puts "Value of a is 30"
} else {
   puts "None of the values is matching"
}
puts "Exact value of a is: $a"
```

switch:

Example:

```
set grade B;
switch $grade {
   A {
         puts "Well done!"
   }
   B {
         puts "Excellent!"
   }
   C {
         puts "You passed!"
   }
   F {
         puts "Better try again"
   }
   default {
         puts "Invalid grade"
   }
}
puts "Your grade is  $grade"
```

The ? : Operator:

Example:

```
set a 10;
set b [expr $a == 1 ? 20: 30]
puts "Value of b is $b\n"
```

Loops:

while loop:

Example:

```
set a 10
while { $a < 20 } {
   puts "value of a: $a"
   incr a
}
```

for loop:

Example:

```
for { set a 10}  {$a < 15} {incr a} {
   puts "value of a: $a"
}
```

for each loop:

Example;

```
set fruits {"apple" "banana" "orange"}
foreach fruit $fruits {
  puts "Current fruit: $fruit"
}
```

Data Structures

Tcl offers two fundamental data structures for storing and manipulating collections of items: lists and arrays. Here's a breakdown of each:

Lists:

- ❖ **ordered Collection:** Lists are essentially ordered sequences of elements, similar to arrays in other programming languages.
- ❖ **Heterogeneous:** List elements can be of different data types (numbers, strings, even other lists) within the same list.

- ❖ **Dynamic Size:** Lists are dynamic, meaning their size can grow or shrink as you add or remove elements.

Creating Lists:

- ❖ You can create lists using curly braces {} with elements separated by spaces:

```
set myList {apple 10 true}  ; List containing string, number, and boolean
```

Accessing Elements:

- ❖ Use the lindex command to access elements by their index (starting from 0):

```
puts [lindex $myList 1]  ; Output: 10 (second element)
```

Negative indices start from the end:

```
puts [lindex $myList end]  ; Output: true (last element)
```

Modifying Lists:

- ❖ Use the lappend command to add elements to the end of the list:

```
lappend myList "orange"
```

lset replaces an element at a specific index:

```
lset myList 0 "banana"  ; Replace first element
```

Common List Operations:

- ❖ llength: Returns the number of elements in the list.
- ❖ lrange: Extracts a sublist based on starting and ending indices.
- ❖ lsearch: Searches for an element in the list and returns its index.

Example:

```
set list {apple 10 true}
puts $list
puts [lindex $list 0]
puts [lindex $list end]
lappend list Deepak
puts "after adding the list is $list"
lset list 0 "karthik"
puts "after replacing the list is $list"
set length [llength $list]
puts "The length of the list is $length"
```

Arrays:

- ❖ **Tcl Arrays are Associative:** Unlike lists, Tcl arrays are associative arrays. This means elements are accessed using unique keys (names) instead of numerical indices.
- ❖ **Heterogeneous Values:** Similar to lists, array elements can hold various data types.
- ❖ **Dynamic Size:** Arrays are also dynamic, allowing you to add or remove elements as needed.

Creating Arrays:

- ❖ Use set arrayName(key) value to create an array element with a specific key:

```
set fruits(apple) "red"
set fruits(banana) "yellow"
```

Accessing Elements:

- Use the key within parentheses to access the corresponding value:

```
puts $fruits(apple)  ; Output: red
```

Modifying Arrays:

- Similar to list modification commands:
  - set arrayName(key) newValue to change the value of an existing key.
  - Additional elements can be added using the same syntax.

Common Array Operations:

- array exists key: Checks if a specific key exists in the array.
- array size: Returns the number of elements in the array.
- unset arrayName(key): Removes a specific key-value pair from the array.

Key Differences:

- **Ordering:** Lists maintain order, while arrays don't guarantee any specific order for elements when iterating.
- **Access Method:** Lists use numerical indices, while arrays use keys.

Choosing Between Lists and Arrays:

- Use lists when the order of elements is important and you need frequent access by index.
- Use arrays when you need to associate data with unique names and the order is not crucial.

Example on Arrays:

```
set languages(0) "Java"
set languages(1) "C Language"
puts $languages(0)
puts $languages(1)
puts  [array size languages]
puts "Printing array elements using for each loop"
for { set index 0 }  { $index < [array size languages] }  { incr index } {
   puts "languages($index) : $languages($index)"
}
```

Associative Arrays example:

Example:

```
set student(name) "Raju"
set student(age) 14
set student(marks) 95
foreach {key value} [array get student] {
  puts "$key -> $value"
}
set student(age) 20
foreach {key value} [array get student] {
  puts "$key -> $value"
}
unset student(name)
foreach {key value} [array get student] {
  puts "$key -> $value"
}
```

input/output

puts and gets are fundamental commands for input and output operations.

puts Command: The puts command in Tcl is used to write output to a specified channel, typically the standard output. It can be used to print strings, variables, and formatted output.

Examples:

puts "Hello, World!"

This command prints "Hello, World!" followed by a newline.

puts -nonewline "Hello, World!"

This command prints "Hello, World!" without appending a newline.

gets Command:

The gets command in Tcl is used to read a line of input from a specified channel, typically the standard input.

Example:

set input [gets stdin]
puts "You entered: $input"

This command reads a line from the standard input and stores it in the input variable, then prints the entered string.

or

gets stdin input
puts "You entered: $input"

This command does the same as the previous example but directly stores the input in the input variable.

procedures:

procedures (often called "procs") are reusable blocks of code that can be called with a specific name and parameters. Procedures help organize and modularize Tcl scripts by encapsulating functionality

Defining a Procedure

To define a procedure in Tcl, you use the proc command.

Syntax

proc name {arguments} {body}

Example:

```
proc greet {name} {
	puts "Hello, $name!"
}
```

In this example, the procedure greet takes one argument (name) and prints a greeting message.
To call a procedure, use its name followed by any required arguments.

greet "Alice"

Procedures with Default Arguments:

You can provide default values for arguments in a procedure.

Example:

```
proc greet {name {greeting "Hello"}} {
	puts "$greeting, $name!"
}
```

Execute the program with following inputs:

```
greet "Bob"            ;# Prints "Hello, Bob!"
greet "Bob" "Hi"       ;# Prints "Hi, Bob!"
```

In this example, the greeting argument has a default value of "Hello". If a second argument is provided, it overrides the default value.

Returning Values

Example:

```
proc add {a b} {
  set result [expr {$a + $b}]
  return $result
}
set result [add 10 5]
puts "Addtion is : $result"
```

Output:

```
Addtion is : 15
```

strings:

strings are fundamental data types used for storing and manipulating text.

Example:

```
set myString "Hello, World!"
```

String concatenation:

Strings can be concatenated using the append command or by placing them next to each other:

Example:

```
set str1 "Hello"
set str2 "World"
set str3 "$str1 $str2"  ;
puts "$str3"
set str4 "kits"
append str4 " college"   ;
puts "$str4"
```

string length:

string length command is used to find the length of the string.

Example:

```
set str "kits college";
puts "the length of the string is [string length $str]"
```

Output:

```
12
```

string extraction:

string range command is to extract substring from the string.

Example:

```
set str "kits college"
puts [string range $str 0 4]
```

Output:

```
kits
```

String Comparison:

Strings can be compared using the string compare command:

Example:

```
puts [string compare "abc" "abc"]
puts [string compare "abc" "def"]
puts [string compare "def" "abc"]
```

Output:

```
0
-1
1
```

String Search:

Searching for substrings can be done with the string first and string last commands.

Example:

```
set str "Gujjula Deepak"
puts [string first "e" $str]
puts [string last "e" $str]
```

Output:

```
9
```

```
10
```

String Transformation:

Convert strings to uppercase or lowercase using string toupper and string tolower:

Example;

```
puts [string toupper "kits"]
puts [string tolower "KITS"]
```

Output:

```
KITS
kits
```

String Substitution:

Replace substrings using string map

Example:

```
set str "This is Deepak"
set newstring [string map {"Deepak" "Karthik"} $str]
puts "$newstring"
```

Example:

```
set str "kits"
set str1 "college"
set str2 "$str $str1"
puts $str2
set a "kamala"
set b "college"
append a " college"
puts $a
puts "Length of a is [string length $a]"
puts [string range $str2 0 3]
puts [string compare "abc" "abc"]
puts [string toupper $str]
puts [string tolower "DEEPAK"]
set x "this is raj"
set newstring [string map {"raj" "kalyan"} $x]
puts $newstring
puts [string first "s" $x]
puts [string last "s" $x]
```

Patterns:

Working with patterns in Tcl involves using various commands that support pattern matching, such as regexp for regular expressions and string match for simple wildcard matching. Here's an in-depth look at how to work with patterns in Tcl:

Example:

string match:

The string match command is used for pattern matching with wildcards:

* matches any sequence of characters (including an empty sequence).

? matches any single character.

[...] matches any one of the enclosed characters.

Example

```
set string "Deepak"
set pattern {[A-Z][a-z]*}
if {[string match $pattern $string]} {
   puts "The string matches the pattern"
} else {
   puts "The string does not match the pattern"
}
```

regexp:

Example:

```
set string "Deepak"
set pattern {[A-Z][a-z]*}
if {[regexp $pattern $string]} {
   puts "The string matches the pattern"
} else {
   puts "The string does not match the pattern"
}
```

Files

In Tcl (Tool Command Language), dealing with files involves a set of commands primarily focused on reading from and writing to files.

Opening a File:

set file [open "filename.txt" mode]

- ❖ "filename.txt" is the name of the file you want to open.
- ❖ mode specifies the mode in which the file should be opened (r for reading, w for writing, a for appending, r+ for reading and writing, etc.).

Example:

```
set file [open "data.txt" "r"]
```

Reading from the Files:

Example:

```
set data [read $file]
```

Reading Line by Line:

Example;

```
while {[gets $file line] != -1} {
        puts "Line: $line"
}
```

Writing to Files:

Example:

```
puts $file "Hello, world!"
```

Appending to a file:

puts -nonewline $file "Additional data"

Appends "Additional data" without a newline to the end of the file associated with $file.

Checking if a File Exists:

Example:

```
if {[file exists "data.txt"]} {
        puts "File exists."
} else {
        puts "File does not exist."
}
```

Returns 1 if "data.txt" exists; otherwise, returns 0.

Example:

```
# Open input file for reading
set inFile [open "input.txt" "r"]
# Open output file for writing
set outFile [open "output.txt" "w"]
# Read from input file and write to output file
while {[gets $inFile line] != -1} {
        puts $outFile $line
}
# Close files
close $inFile
close $outFile
```

Advanced TCL;

eval:
eval is a command used to evaluate a script or a command dynamically during the execution of a Tcl script. Its primary purpose is to interpret and execute Tcl code that is stored in a string or a variable.
Syntax:

```
eval script
```

where script is a string containing Tcl code.
Example;

```
set script {puts "Hello, world!"}
eval $script
```

source:
source command is used to read and execute the contents of a file as a Tcl script. This command is very useful for modularizing code, reusing scripts, or loading configurations.
Example:
Suppose you have a Tcl script file named myscript.tcl with the following content:

```
# myscript.tcl
puts "Hello from myscript.tcl"
set x 10
```

You can use the source command in another Tcl script to execute the content of myscript.tcl:

```
# main.tcl
source myscript.tcl
puts "The value of x is $x"
```

When you run main.tcl, it will output:

```
Hello from myscript.tcl
The value of x is 10
```

exec:
exec command is used to run an external program or command from within a Tcl script. This allows you to interact with the underlying operating system and execute commands that are not part of the Tcl language itself.
Example:
To execute a simple command like ls (to list directory contents on Unix-like systems):

```
set result [exec ls]
puts $result
```

This will store the output of the ls command in the result variable and then print it.

uplevel:

The uplevel command in Tcl allows you to execute a script in a different scope, or stack frame, than the current one. Essentially, it lets you run commands as if they were part of a different procedure or context.
When you call a procedure in Tcl, it creates a new stack frame, or level. Each level has its own variables and context. The uplevel command lets you run code in a different level.

Example:

```
proc first {} {
 set a 10
 puts "The value of a is $a"
 second
 puts "The value of a is $a"
}
proc second {} {
 uplevel {
 set a 20
}
}
first
```

Namespaces:

namespaces are used to organize and manage variable and procedure names, preventing name conflicts and allowing for better code modularization. Namespaces enable you to create separate contexts for variables and procedures, which can be particularly useful in larger programs or when combining code from multiple sources.
A namespace is a container for a set of commands and variables. By default, all commands and variables are defined in the global namespace. You can create additional namespaces to group related commands and variables together

Example:

```
# Define a namespace and some variables and procedures within it
namespace eval myNamespace {
    variable counter 0
    proc increment {} {
        variable counter
        incr counter
    }
    proc getCounter {} {
        variable counter
        return $counter
    }
}
```

```
# Accessing the procedures within the namespace
myNamespace::increment
puts "Counter: [myNamespace::getCounter]"  ;# Outputs: Counter: 1
myNamespace::increment
puts "Counter: [myNamespace::getCounter]"  ;# Outputs: Counter: 2
```

trapping errors:

Trapping errors in Tcl is done using the catch command. The catch command executes a script and captures any errors that occur, allowing you to handle them gracefully instead of causing the program to terminate unexpectedly.

Example:

```
proc divide {a b} {
  set result ""
  set status [catch {expr {$a / $b}} result]
if {$status==0} {
  puts "$result"
} else {
  puts "$result"
}
}
divide 10 0 #Output : divide by zero
divide 10 2 #Output :  5
```

Error Handling:

- If status is 0, it means the command executed without errors, and we print the result.
- If status is 1, it means an error occurred, and we print the error message.

event driven programs

Event-driven programming in Tcl (Tool Command Language) involves writing programs that respond to events. These events can be user actions like mouse clicks or key presses, or system-generated events like timers expiring or files becoming readable. In Tcl, event-driven programming is commonly used in graphical user interface (GUI) applications created with the Tk toolkit.

Events: Actions or occurrences that a program can respond to. Examples include mouse clicks, key presses, window resizes, etc.

Event Loop: A loop that waits for events to occur and dispatches them to the appropriate event handlers. In Tcl/Tk, the event loop is handled automatically when you enter the vwait command

Event Handlers: Procedures or scripts that are executed in response to specific events. In Tcl, you bind these handlers to events using the bind command.

Widgets: GUI elements like buttons, labels, text boxes, etc., that can generate events. Tcl/Tk provides a wide range of widgets for building GUIs.

Example:

```
# Load the Tk package
package require Tk
# Create a main window
set mainWindow [tk appname "Event Driven Example"]
# Create a button widget
button .myButton -text "Click Me" -command "buttonClickHandler"
# Pack (display) the button in the main window
pack .myButton
# Define the event handler procedure
proc buttonClickHandler {} {
   puts "Button was clicked!"
}
# Start the Tk event loop
vwait forever
```

Above example creates a window with a button, and clicking the button triggers an event that is handled by an event handler procedure.

Making applications Internet aware:

Making applications Internet-aware in Tcl (Tool Command Language) involves using the http package, which provides commands for accessing web resources via the HTTP protocol. The http package in Tcl simplifies the process of sending HTTP requests and handling the responses, allowing Tcl applications to interact with web services, download content, and communicate over the web.

Loading the http Package

Before using the http package, you need to load it. This can be done with the following command:

package require http

Sending an HTTP Request

The most common way to send an HTTP request is using the http::geturl command. This command can perform various HTTP operations (GET, POST, etc.).

Basic GET Request

To perform a basic GET request, you can use the following code:

```
set token [http::geturl "http://www.example.com"]
set response [http::data $token]
http::cleanup $token
puts $response
```

http::geturl : sends a GET request to the specified URL and returns a token.

http::data : retrieves the response data associated with the token.
http::cleanup: releases resources associated with the token.
You can check for errors in the response using http::status:

Asynchronous Requests

You can perform asynchronous HTTP requests using the -command option. This allows your application to continue processing other tasks while waiting for the response:

```
proc handleResponse {token} {
   if {[http::status $token] eq "ok"} {
      set response [http::data $token]
      puts $response
   } else {
      puts "Error: [http::error $token]"
   }
   http::cleanup $token
}
http::geturl "http://www.example.com" -command handleResponse
```

Downloading Files

To download files, you can use the -channel option to save the response directly to a file:

```
set output [open "output.html" w]
set token [http::geturl "http://www.example.com" -channel $output]
close $output
http::cleanup $token
```

Nuts and Bolts Internet Programming

Internet programming at the nuts-and-bolts level involves using sockets, which are endpoints for sending and receiving data across a network. In Tcl, socket programming provides a way to establish low-level communication between applications over TCP/IP networks. This allows you to create server and client applications that can communicate with each other directly over the network.

Sockets in Tcl

Tcl provides built-in support for socket programming through the socket command, which can be used to create both client and server sockets.

Creating a Client Socket

To create a client socket that connects to a server, you use the socket command with the server's hostname and port number.

```
# Create a client socket to connect to example.com on port 80
set sock [socket example.com 80]
# Send a request (e.g., HTTP GET request)
puts $sock "GET / HTTP/1.0\r\nHost: example.com\r\n\r\n"
# Read the response from the server
```

```
while {[gets $sock line] >= 0} {
   puts $line
}
# Close the socket
close $sock
```

Creating a Server Socket

To create a server socket that listens for incoming connections, you use the socket command with the -server option.

```
# Define the procedure to handle incoming connections
proc accept {sock addr port} {
   puts "Connection from $addr:$port"
   fileevent $sock readable [list handleClient $sock]
}
# Define the procedure to handle client communication
proc handleClient {sock} {
   if {[eof $sock]} {
      close $sock
      return
   }
   set data [gets $sock]
   puts "Received: $data"
   puts $sock "Echo: $data"
}
# Create a server socket to listen on port 12345
set server [socket -server accept 12345]
puts "Server listening on port 12345"
# Start the Tcl event loop to handle connections
vwait forever
```

Perl Tk:

Perl/Tk, often referred to as Tk, is a Perl module that provides a graphical user interface (GUI) toolkit. It is a Perl binding to the Tk toolkit, which was originally developed for the Tcl scripting language. Perl/Tk allows developers to create cross-platform GUI applications with Perl.