Assignment

Name: A. Srivallika

Roll No: 1926-A0501

Sub: SL

Branch: CSE-01

1) Explain about structure of RUBY programs with examples Control structures, arrays, Hashes.

- Ruby is a high-level programming Language.
- Ruby is an Interpreted like Perl, Python, Tcl/Tk.
- It is an Object-oriented like Smalltalk, Eiffel, Ada, Java.

Characteristics:

- Easy to learn
- Open source
- Rich libraries
- Very easy to extend
- Truly Object-oriented
- Less coding with fewer bugs
- Helpful community.

Draw backs:

- Performance Issues
- Threading model.

```
#Simple program (sample.rb)
puts "Scripting Language"
puts "Sample code"
>ruby sample.rb.
```

Output:
Scripting Language
sample code.

Control structure:

→ Conditional / selection statements
  → if
  → if else
  → if elsif
  → switch (case)
  → else-if ladder
  → ternary stmt

① if statements

Ruby if stmt tests the condition. The if block stmt is executed if condition is true.

Syntax:

```
if (condition)
# Code to be executed
end
```

Example:

```
puts "enter your age:"
age = gets.chomp.to_i
if age >= 18
puts "you are eligible to vote."
end.
```

Output:

enter your age: 20

you are eligible to vote.

② if-else statement:

The Ruby if-else stmt tests the condition. The if block stmt is executed if condition is true. Otherwise else block stmt is executed.

Syntax:

```
if (condition)
  // code if condition is True
else
  // code if condition is false
end.
```

Ex:

```
puts "enter your age:"
age = gets.chomp.to_i
if age >= 18
    puts " you are eligible to vote."
else
    puts " you are not eligible to vote."
end.
```

Output:

enter your age: 4

you are not eligible to vote.

(3) If-elsif statement:

Ruby if elsif stmt tests the condition the ifblock stmt is executed if condition is True. Otherwise else block stmt is executed

Syntax:

```
if (condition 1)
  // code to be executed if condition 1 is true
elsif (condition 2)
  // code to be executed if condition 2 is true
else
  // code to be executed if 2 conditions false
end.
```

Ex:

```
print "Enter a number: "
num = gets.to_i
if num<0
   puts "#{num} is negative"
elsif num==0
   puts "#{num} is zero."
else
   puts "#{num} is positive"
end.
```

Output:

```
Enter a number: 0
0 is zero.
```

Enter a number: 10.

10 is " positive

④ if-elsif ladder:

Ruby if-elsif ladder stmt tests the condition. The if block stmt is executed if condition1 is True. Otherwise elsif block stmt get executed if condition2 - - condition'n' is True otherwise the else block gets executed.

Syntax:

```
if (condition1)
// code for condition1
elsif (condition 2)
// code for condition 2
  :
  :
elsif (condition n)
// code for condition n
else
  // code for False condition)
end.
```

Example:

```
puts " enter your percentage:"
a= gets.chomp.to_i
if  a<50
puts " student is fail."
```

```
elsif  a>=51 && a<=60
  puts "student gets 'D' grade".
elsif  a>=61 && a<=70
  puts "Student get C grade".
elsif  a>=71 && a<=80
  puts "student gets B grade."
elsif  a>=81 && a<=90
  puts "student gets A grade."
else
  puts "student gets O grade"
end.
```

Output:

```
enter your percentage: 92
student gets O grade
```

5) Ternary stmt:

In Ruby ternary stmt, the if stmt is shortened first it evaluates an expression for true or false value then execute one of the stmts.

Syntax:

```
test_expression ? if_true_expression : if_false_expression
```

Example:

```
puts ("Ternary Operator")
puts "enter a number:"
var = gets.chomp.to_i
a= (var > 3 ? True : False)
puts a.
```

Output:

```
Ternary Operator
enter a number: 4
True
```

6) Case Stmt:

In Ruby, the case stmt is a selection control flow stmt. It allows the value of a variable or expression to control the flow of program execution via a multi way branch.

We use 'case' instead of 'switch' & 'when' instead of 'case'. The case stmt matches one stmt with multiple conditions just like a switch stmt in other languages.

Syntax:

```
case expression
[when expression [, expression, ...] [then]
  code] ...
[else
  code]
end.
```

Ex:

```
print "enter subject name:"
sub = gets.chomp
case sub
    when "OS"
        puts "Operating System".
    when "SL"
        puts "Scripting Language"
    when "CD"
        puts "Compiler Design"
    when "ML"
        puts "Machine Learning"
    when "DAA"
        puts "Design and Analysis Algorithms"
    when "FIoT"
        puts "Fundamentals of Internet of Things"
    when "CS"
        puts "Cyber Security"
    when "ES"
        puts "Environmental Science"
end. else
        puts "Invalid"
end
```

Outputs: enter subject name: DAA
Design and Analysis Algorithms.

→ Loops in Ruby: A loop is the repetitive execution of a piece of code for a given amount of repetitions or until a certain condition is met.

- For loop
- while loop
- do while loop

① For loop:

→ Entry controlled loop.

→ because the condition to be tested is present at the beginning of the loop body.

Syntax:

```
for var_name [, var ...] in exp [do]
  # code to be executed
end.
```

Ex:

```
for i in 0..9 do
    puts "#{i}"
end
```

Output:

```
0
1
2
3
4
5
6
7
8
9
```

2) while loop:

The while stmt is a control flow stmt that allows code to be executed repetadly based on a given Boolean condition. It executes the code while the condition is True.

Syntax:

```
while Condition
  Code
end
```

ex:

```
i = 0
sum = 0
while i <= 10
  i = i + 1
  sum = sum + i
end
puts "The sum of first 10 numbers is #{sum}"
```

Output:

The sum of first 10 numbers is 55

3) do-while loop:

do-while loop similar to while loop with the only difference that it checks the condition after executing the stmts. Hence it will execute the loop body one time for sure.

- Exit Control loop.

Syntax:

```
loop do,
#code to be executed
break if booleanExpression
end
```

until stmt:

Ruby until loop will executes the stmts till the given condition evaluates to true. Basically it just opposite to the while loop which executes until the given condition evaluates to false

EX:

```
i=1
until  i==10
    print i*10, "\n"
      i+=1
  end.
```

Output:

```
10
20
30
40
50
60
70
80
90
```

Break stmt:

The Ruby break stmt is used to terminate a loop. mostly used in while loop where value is printed till the condition is true. then break stmt terminates the loop.

Syntax:

```
break
```

EX:

```
i=1
while true
  if  i*5 >= 25
      break
  end
      puts  i*5
      i+=1
end
```

Output:

```
5
10
15
20
```

redo stmt:

Ruby redo stmt is used to repeat the current iteration of the loop. the redo stmt is executed without evaluating the loop's condition.

→ It is used inside a loop

Syntax:

```
redo
```

Ex:

```
q=0
while (q<5)
  puts q
  q+=1
  redo if q==5
end
```

Output:

```
0
1
2
3
4
5
```

→ next stmt:

The Ruby next stmt is used to skip loop's next iteration. Once the next stmt is executed no further iteration will be performed

— next is similar to continue stmt

Ex:

```
for q in 5...11
  if q==7 then
    next
  end
  puts q
end
```

Output:

```
5
6
```

Arrays:

- Ruby arrays are ordered collections of objects. they can hold objects like Integer, number, hash, string, symbol or any other array.
- Its indexing starts with 0. the negative index starts with -1 from the end of the array.

A Ruby array is created in many ways

- Using literal Constructor []
- Using new class method

Creating Arrays using literal Constructor []

A single array can contain different type of objects.

Ex:

```
student = [1, 8.5, "Sri"]
puts student
```

Output:

```
1
8.5
"Sri"
```

Ex:

```
alpha = Array("a"..."z")
puts alpha
```

Output:

```
["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k",
"l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w",
"x", "y", "z"]
```

Array Creation Using new class method:

A Ruby array is constructed by calling :: new method with zero, one or more than one arguments.

Ex:

```
student = Array.new(20)
puts student.size
puts student.length
```

Array Built-in methods:

we need to have an instance of array object to call an array method. To create an instance of Array object syntax is:

Syntax:

```
Array.[](...) (or) Array[...] or [...]
```

Ex:

```
digits = Array(0..9)
num = digit.at(6)
puts "#{num}"
```

Output:

6

Accessing array elements:

```
days = ["mon", "Tue", "wed", "Thu", "Fri", "Sat", "Sun"]
puts days[0] # mon
puts days[-2] # sat
puts days[2,3] # wed, thu, fri
```

Hashes:

Ruby hash is a collection of unique keys, and their values. They are similar to arrays but arrays use integer as an index & hash use any object type

Syntax:

```
name = { "key1" => "value1", "key2" => "value2", ... }
                    (or)
name = {'key1': "value1", "key2": "value2", ... }
```

Ex:

```
Color = { "Rose" : "red", "Lily" : "purple", "mangold":
  "yellow", "Jasmine": "white" }

puts  color['Rose']
puts  color['Lily']
```

Output:

```
red
purple
```

② Explain about writing CGI scripts, cookies, with example.

Ruby to write CGI scripts quite easily. to have a Ruby script generate HTML output, all you need is something like this.

```
#!/usr/bin/ruby
print "Content-type: text/html\r\n\r\n";
print "<html> <body> Hello world! It's #{time.now}
```

</body> </html> \r\n".

Using cgi.rb:

To work with CGI scripts we need to config the XAMPP server.

Step1: Open XAMPP Control Panel.

Step2: Open 'httpd.config' file

Step3: Search for addhandler and add '.rb'.

Step4: Search for <Directory "C:/xampp/cgi-bin"> and add require all granted privilages

Step5: Save the file

Step6: Start the server

Step7: Write the program and Save the program under Xampp → cgi-bin folder

Step8: run the file in browser - localhost/cgi-bin/filename.rb

Program:

```
#!"C:/Ruby30-x64/bin/ruby.exe"
require 'cgi'
cgi = CGI.new
puts cgi.header
puts "<html><body>This is a test</body></html>"
```

Save as: C6 → Xampp → cgi-bin folder.

run: localhost/cgi-bin/test23.rb.

Output:

This is a test

Cookies:

Cookies are a way of letting web applications store their state on the user's machine.

The Ruby CGI class handles the loading and saving of cookies for you. You can access the cookies associated with the current request using CGI#cookies method, and you can set cookies back into browser by setting the cookies parameter of CGI#out of reference either a single cookie or an array of cookies.

```
#!/usr/bin/ruby
COOKIE_NAME = "chocolate chip"
require 'cgi'
cgi = CGI.new
values = cgi.cookies[COOKIE_NAME]
if values.empty?
  msg = "Its looks as if you haven't visited recently" ;-)"
else
  msg = "you last visited #{values[0]}"
end
cookie = CGI::Cookie.new(COOKIE_NAME,
  Time.now.to_s)
cookie.expires = Time.now + 30*24*3600
cgi.out("Cookie" => cookie) {msg}
```

3) what is SOAP and Web Services, explain.

SOAP: The Simple Object Access Protocol (SOAP), is a cross-platform and language-independent RPC protocol based on XML and HTTP.

- It uses XML to encode the information that makes the remote procedure call, and HTTP to transport that information across a network from clients to server

Advantages:

- Cheap deployment
- Debugging costs
- It extensibility
- Easy-of-use
- Several implementations for different languages and platforms.

```
Class InterestCalculator
    attr_reader: call_count
    def initialize
        @call_count = 0
    end
    def compound(principal, rate, freq, years)
        @call_count += 1
        principal * (1.0 + rate/freq) ** (freq * years)
    end
end
```

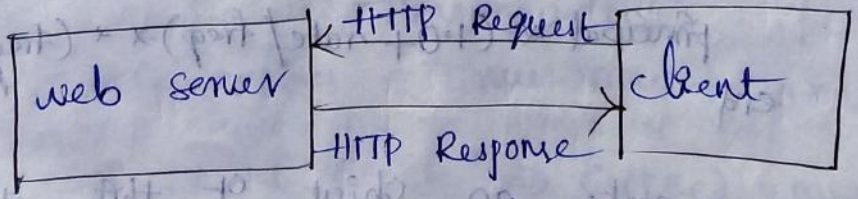
Now we'll make an object of this class available via a soap server. This will enable

client applications to call the objects methods over the n/w. We're using the standalone server here, which is convenient when testing. as we can run it from the command line. You can also run Ruby SOAP servers as CGI scripts or under mod_ruby.

```
require 'soap/rpc/standaloneServer'
require 'interestcalc'
NS = 'http://pragprog.com/interestCalc'
class Server2 < SOAP::RPC::StandaloneServer
def on_init
  calc = InterestCalculator.new
  add_method(calc, 'compound', 'principal', 'rate',
      'freq', 'years')
  add_method(calc, 'call_count')
  end
end
svr = Server2.new('calc', NS, '0.0.0.0', 12321)
trap('INT') { svr.shutdown }
svr.start
```

This code defines a class which implements a standalone SOAP server.

Web Servers



A web server uses http protocol to transfer data. In a simple situation, a user type

in a url (eg: www.google.com) in a browser and get a web page to read. So what the server does is sending a web page to the client the transformation is in http protocol which specifies the format of request and response message

Here we are using the WEBrick server

```
#!/usr/bin/ruby
require 'webrick'
Include WEBrick
S = HTTPServer.new(
  :Port          => 2000,
  :DocumentRoot => File.join(Dir.pwd, "/html")
)
trap("INT") { S.shutdown }
S.start
```

⇒ The HTTPserver Constructor creates a new webserver on port 2000

4) Explain about Ruby Tk with Examples.

The standard Graphical User Interface (GUI) for Ruby is Tk.

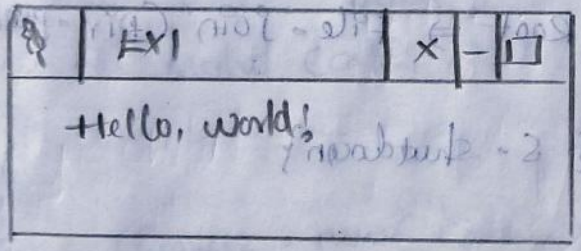
- Tk has the unique distinction of being the only cross-platform GUI. Tk runs on windows, mac and Linux and provides a native look and feel on each OS.
- To use it, we need to have Tk installed

on your system.
- The basic component of a tk-based application is called widget / container / window.

Ex6

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

Output:



After loading the tk extension module, we create a root-level frame using 'TkRoot.new'. we then make a TkLabel widget as a child of the root frame, setting several options for the label. Finally, we pack the root frame and enter the main GUI event loop.
It's a good habit to specify the root explicitly, but you could leave it out along with the extra options and boil this down to a three-linear

Steps to be followed:

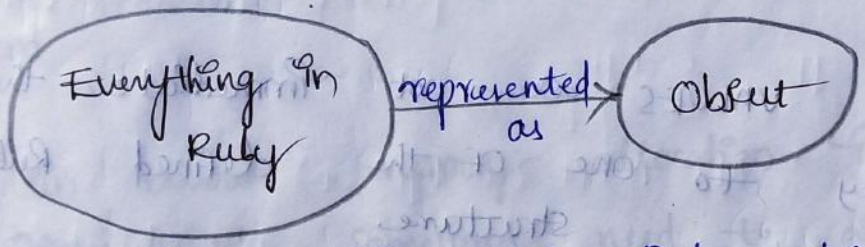
Step1: Create container widget ..... TkFrame or TkRoot

Step2: then create widgets that populates it, such as button or labels etc.

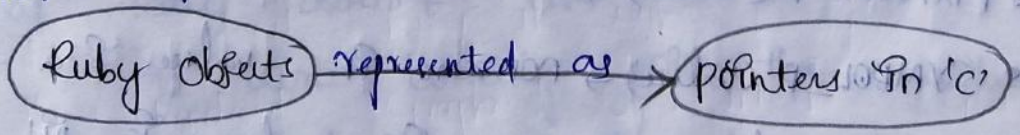
Step3: Tkmainloop ... we have to invoke it

5) Explain Ruby Objects in C.

Everything in Ruby is an object, and all variables are references to objects.

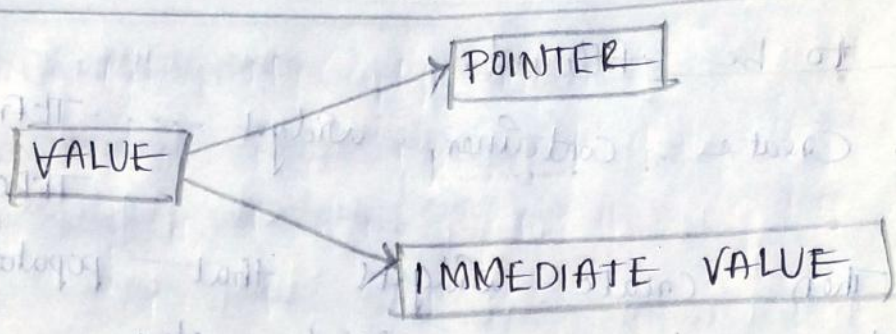


when we're looking at Ruby objects from within 'C' code most Ruby objects are represented as 'C' pointers to an area in memory that contains the objects data and other implementation details.



VALUE:

In 'C' code, all these references are via variables of type VALUE, so when you pass Ruby objects around you'll do it by passing VALUE's.



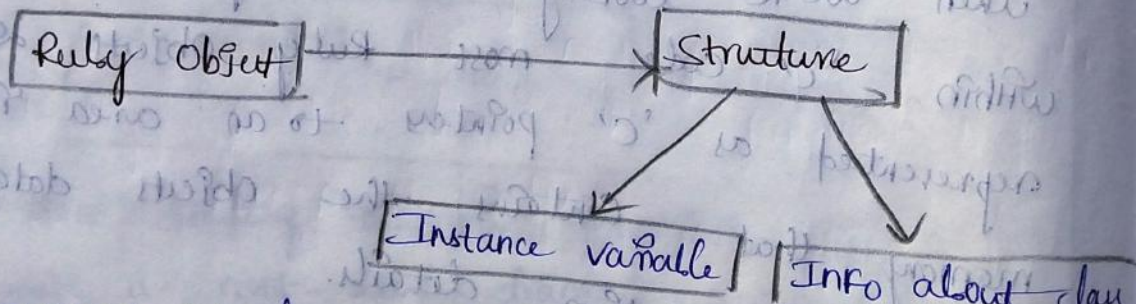
IMMEDIATE VALUES:

Ruby implements Fixnums, Symbols, true, false, and nil also so called immediate values. They are not pointers.

So sometimes VALUES are pointers, and sometimes they're immediate values.

NOTE:

when VALUES are not immediate, they are pointers to one of the defined Ruby object structure



- 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.

Class:

the class itself another object that contains a table of the methods defined or that class. Ruby is built upon this foundation.

Class → is an → Object (structure) → Contains methods.

Normal Ruby class program:

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

6) Explain about memory allocation in Ruby

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

To work correctly with the garbage collector you should be use the following memory allocation routines. These routines do a little bit more work than the standard "malloc" API's memory Allocation.

- type *ALLOC_N (c-type, n)

Allocates n c-type objects, where c-type is the literal name of the c type, not a variable of the type.

- type *ALLOC (c-type)

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

- REALLOC_N (var, c-type, n)

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

- type *ALLOCA_N (c-type, n)

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

7) Explain the Concept of Ruby Type System and Embedding a Ruby Interpreter.

Ruby Type System:

In Ruby, we rely less on the type (or class) of an object and more on its capabilities. This is called duck typing. We describe in more detail in example, the following code implements the kernel.exec method.

```
VALUE
rb_f_exec(argc, argv)
int argc;
VALUE *argv;
{
    VALUE prog=0;
    VALUE tmp;
    if (argc==0) {
        rb_raise(rb_eArgError, "wrong no. of args");
    }
    tmp = rb_check_array_type(argv[0]);
    if (!NIL_P(tmp)) {
        if (RARRAY(tmp)->len != 2) {
            rb_raise(rb_eArgError, "wrong 1st argument");
        }
        prog = RARRAY(tmp)->ptr[0];
        SafeStringValue(prog);
        argv[0] = RARRAY(tmp)->ptr[1];
    }
    if (argc==1 && prog==0) {
        VALUE cmd = argv[0];
        SafeStringValue(cmd);
        rb_proc_exec(RSTRING(cmd)->ptr);
    }
    else {
        proc_exec_n(argc, argv, prog);
    }
    rb_sys_fail(RSTRING(argv[0])->ptr);
}
return Qnil; /* dummy */
```

Embedding a Ruby Interpreter:

In addition to extending Ruby by adding C code, you can also turn the problem around and embed Ruby itself within your application. You have two ways to do this. The first is to let the interpreter take control by calling 'ruby_run'. This is easiest approach, but it has one significant drawback - the interpreter never returns from a ruby_run call.

Examples

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

To initialize the Ruby interpreter, you need to call 'ruby_init()'

```
# if defined(NT)
  NtInitialize(&argc, &argv);
# endif
```

```
#if defined(__MACOS__) && defined(__MWERKS__)
    argc = ccommand(&argv);
#endif
```

The second way of embedding Ruby allows Ruby code and your C code to engage in more of a dialogue: the C code calls some Ruby code, & the Ruby code responds.

Let's look at an example. Here's a simple Ruby class that implements a method to return the sum of the numbers from one to max.

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

API: Embedded Ruby API

- void ruby_init()

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

- void ruby_init_loadpath()

Initializes the $: (load path) variable; necessary

- If your code loads any library modules
- void ruby_options (int argc, char **argv)
  Gives the Ruby Interpreter the Command-line options.
- void ruby_script (char *name)
  Sets the name of the Ruby script to name
- void rb_load_file (char *file)
  Loads the given file into the Interpreter
- void ruby_run()
  Runs the Interpreter
- void ruby_finalize()
  Shuts down the Interpreter

8) what are the characteristics of Scripting Languages?

- Integrated compile and run
- Low overheads & easy of use
- Enhanced functionality
- Efficiency is not an issue
- Both Batch and Interactive use
- Economy of Expression.
- Lack of declarations; simple scoping rules

- Flexible dynamic typing.
- Easy access to Other programs.
- Sophisticated pattern matching
- High level data types.

9) Explain the Uses for Scripting Languages.

The activities that Comprise traditional Scripting include.

- System Administration: automating everyday tasks, building data reduction tools
- Controlling applications remotely.
- System and application extensions.
- 'Experimental' programming.
- Building Command-line Interfaces to application -s based on C libraries.
- Server side form processing on the web using CGI.

Traditional Scripting is the province of what we nowadays called 'open source' languages, particularly Perl and Tcl.

System administration:

The concept of scripting first arose out of the requirements of system administration in the Unix world. The administrators use shell scripts

Driving applications remotely:

As we have seen, an early application of scripting was to control a dial-up link to a remote computer. The use of a SL, rather than a utility that just plays back a previously recorded sequence of keystrokes, is necessary because there is often an element of conditional logic involved.

System and application extensions:

In terms of the definition given in the preceding section, the UNIX system is a scriptable application, since it exposes a programmable API in the form of the system calls.

Experimental programming:

Traditional languages like 'C' were developed as tools for writing large and complex programs, many of which require teams of programmers.

Command-line Interfaces:

A major use of SL in traditional scripting is as a form of 'glue' to connect together sections of code written in some other language or through a command line Interface