SEMESTER-V

17CAP504W RUBY PROGRAMMING 4H - 4C

Instruction Hours / week: L: 4 T: 0 P: 0 Marks: Internal: 40 External: 60 Total: 100
End Semester Exam: 3Hours

Scope:

This course covers the fundamental components of the Ruby Programming Language. Emphasis is placed on the object oriented aspects of Ruby. Topics include arrays, hashes, regular expressions, I/O, exceptions, modules, and applications areas. Ruby is a programming language with a focus on simplicity and productivity.

Objective: To help students to

- Develop server-side Ruby scripts for publishing on the Web
- Employ control structures, methods, procs, arrays and hashes to create Ruby programs
- Distinguish and use various Ruby datatypes
- Master the use of arrays and hashes
- Use the extensive pre bundled classes
- Use the I/O facilities of Ruby to read and write binary and text files
- Master the use of Iterators to loop through various data structures
- Use Exceptions in handling various run time errors
- Create Ruby modules
- Use the wide variety of Ruby Modules that come with the Ruby distribution
- Use object-oriented programming conventions to develop dynamic interactive Ruby applications

UNIT I

Introduction to Ruby: Installing Ruby - THE STRUCTURE AND EXECUTION OF RUBY PROGRAMS: Lexical Structure- Syntactic Structure - Block Structure in Ruby- File Structure - Program Execution. DATA TYPES: Numbers - Text - String Literals - Character Literals - String Operators - Accessing Characters and Substrings - Iterating Strings – Arrays – Hashes – Ranges – Symbols – True & False – Ruby Documentation: RDoc and ri.

UNIT II

STATEMENTS AND CONTROL STRUCTURES: Conditionals – Loops - Iterators and Enumerable objects: custom iterators – enumerators – External iterators – Blocks: Variable scope – passing argument to blacks. Flow-altering statements like return and break- The special-case BEGIN and END statements. CLASSES : Creating and initializing class – Accessor and attributes – class methods – class variables – Defining operators. SUBCLASSING AND INHERITANCE: visibility – Overriding methods. OBJECTS: Object creation and initialization.

UNIT III

METHODS: Defining a Method, Calling a Method; Undefining methods – Methods with Exception – Operator methods and names – Method Arguments – Method objects - Defining Attribute Accessor Methods - Dynamically Creating Methods. EXCEPTIONS AND EXCEPTION HANDLING: Hierarchy – Exception classes and objects – Raising Exception with raise – Handling Exception with rescue – Exception propagation – Else clause and ensure class.

16CAP504W RUBY PROGRAMMING

UNIT IV

MODULES: Namespaces - Modules as Mixins - Includable Namespace Modules - Loading and Requiring Modules - Executing Loaded Code. Reflection and Meta programming: Evaluating Strings and Blocks - Querying, Setting, and Testing Variables – Regular Expressions. FILES AND DIRECTORIES: Listing and manipulating Directories and testing files. BASIC INPUT AND OUTPUT: Opening Stream – Reading from a Stream – Writing to a stream – Random Access Methods – Closing, Flusing and testing streams.

UNIT V

THREADS AND PROCESSES: Thread Life Cycle – Thread scheduling – Thread Exclusion – Deadlock. Ruby Tk: Introduction- Widgets and classes. Networks: A Very Simple Client - A Very Simple Server – Datagram - A Multiplexing Server - Fetching Web Pages. Ruby on Rails: Building a development Environment: Installation – Installing Databases – Code editors – web server Configuration – Creating an web application.

SUGGESTED READINGS:

1. Dave Thomas, Andrew Hunt (2013), Programming Ruby 1.9 & 2.0: The Pragmatic Programmers Guide, 2nd Edition, The Pragmatic Bookshelf.
2. David Flanagan, (2008), "The Ruby Programming Language", 1st Edition, O'Reilly Media.
3. Eldon Alameda (2011), "Practical Rails Projects" Apress, Berkeley, CA, USA.
4. David Black, (2006), "Ruby for Rails", Manning Publications.

WEB SITES :

1. http://www.tutorialspoint.com/ruby/ruby_tk_guide.htm
2. www.fincher.org/tips/Languages/Ruby
3. www.troubleshooters.com/codecorn/ruby/basictutorial.htm
4. www.ruby-lang.org/en/documentation/quickstart

Karpagam Academy of Higher Education
(Established Under Section 3 of UGC Act 1956)
Eachanari (po), Coimbatore-21

COURSE: RUBY PROGRAMMING [17CAP504W]

LECTURE PLAN - UNIT I

S.No.	Lecture Duration (Hr)	Topics to be Covered	Support Materials
1	1	Introduction to Ruby	R1: 1-17
2	1	Installing Ruby,	W1, J1
3	1	The structure and execution of ruby programs, Lexical Structure,-	R1:P(25-35)
4	1	Syntactic Structure	W1
		Block Structure in Ruby	R1:P(26-33)
		File Structure	R1:P(35-36)
		Program Execution.	R1:P(39-42)
5	1	Data types: Numbers, Text,	
6	1	String Literals ,	
		Character Literals	
		String Operators , Accessing Characters and Substrings, ,	W1
		, Iterating Strings ,Arrays	R1:P(46-64)
7	1	Hashes ,Ranges , Symbols : True & False ,	R1:P(67-71)
8	1	Ruby Documentation: RDoc and ri.	T1:9-11
9	1	Recapitulation and Discussion of important questions	
Total no. of periods planned for Unit I : 9			

Textbooks (T1) : Dave Thomas, Andrew Hunt, 2013, Programming Ruby 1.9 & 2.0: The Pragmatic Programmers Guide 2nd Edition, The Pragmatic Bookshelf.
Ref erence Book (R1): David Flanagan, 2008, "The Ruby Programming Language", 1st Edition, O'Reilly Media.
Website (W1) : WWW.ruby_doc.org/docs/Tutorials
Journal (J1) : Problem discovery comes before problem solving issue 8.1:: Published by Greerry Brown on March 2015

Karpagam Academy of Higher Education
(Established Under Section 3 of UGC Act 1956)
Eachanari (po), Coimbatore-21

COURSE: RUBY PROGRAMMING [17CAP504W]
LECTURE PLAN - UNIT II

S.No.	Lecture Duration (Hr)	Topics to be Covered	Support Materials
1	1	statements and control structures: Conditionals , Loops	R1:P(117-127)
2	1	Iterators and Enumerable objects: custom iterators – enumerators – External iterators	R1:P(130-140)
3	1	Blocks: Variable scope	R1:P(140-146)
		passing argument to blocks	W1
4	1	The special-case BEGIN and END statements	R1:P(165-166)
5	1	Flow-altering statements like return and break	R1:P(146-154)
6	1	CLASSES : Creating and initializing class	W3, W4
		Accessor and attributes	W4
7	1	class methods – class variables – Defining operators.	W5
		SUBCLASSING AND INHERITANCE: visibility – Overriding methods.	W3
8	1	OBJECTS: Object creation and initialization	W3
9	1	Recapitulation and Discussion of important question	W4
Total no. of periods planned for Unit II : 9			

Textbooks (T1) : Dave Thomas, Andrew Hunt, 2013, Programming Ruby 1.9 & 2.0: The Pragmatic Programmers Guide 2nd Edition, The Pragmatic Bookshelf.
Reference Book (R1): David Flanagan, 2008, "The Ruby Programming Language", 1st Edition, O'Reilly Media.
Website (W3) : www.ruby_doc.org/docs/Tutorials
Website (W4) : www.codeacademy.com/Tracks/ruby/resource
Website (W5) : www.ruby.doc.org/core/objects.html

Karpagam Academy of Higher Education
(Established Under Section 3 of UGC Act 1956)
Eachanari (po), Coimbatore-21

COURSE: RUBY PROGRAMMING [17CAP504W]

LECTURE PLAN - UNIT III

S.No.	Lecture Duration (Hr)	Topics to be Covered	Support Materials
1	1	METHODS: Defining a Method, Calling a Method; Undefining methods	T1:P(74-76)
2	1	Methods with Exception	R1:P(240-241)
3	1	Operator methods and names	R1:P(243-253)
4	1	Defining Attribute Accessor Methods	W6,W7
5	1	Method Arguments Method objects	W6,W7
6	1	Dynamically Creating Methods	W7
		Hierarchy – Exception classes and objects	T1: P(23-27)
7	1	EXCEPTIONS AND EXCEPTION HANDLING	W6
		Raising Exception with raise	W6
8	1	Handling Exception with rescue	R1:P(154-165),T1: P(109-113)
		Exception propagation – Else clause and ensure class	W7
9	1	Recapitulation and important questions discussion	
Total no. of periods planned for Unit III : 9			

Textbooks (T1) : Dave Thomas, Andrew Hunt, 2013, Programming Ruby 1.9 & 2.0: The Pragmatic Programmers Guide 2nd Edition, The Pragmatic Bookshelf.
Reference Book (R1): David Flanagan, 2008, "The Ruby Programming Language", 1st Edition, O'Reilly Media.
Website (W6) : www.tutorials.com/ruby/ruby_regular
Website (W7) : rubyamqp.info/articles/error_handling/

1.
2.

Karpagam Academy of Higher Education
(Established Under Section 3 of UGC Act 1956)
Eachanari (po), Coimbatore-21

COURSE: RUBY PROGRAMMING [17CAP504W]

LECTURE PLAN - UNIT IV

S.No.	Lecture Duration (Hr)	Topics to be Covered	Support Materials
1	1	MODULES: Namespaces	T1:P(113-116)
2	1	Modules as Mixins - Includable Namespace Modules	R1:P(337-344)
		Loading and Requiring Modules	W9
3	1	Executing Loaded Code	R1:P(357-364)
4	1	Reflection and Meta programming: Evaluating Strings and Blocks	W8
5	1	Querying, Setting, and Testing Variables	W8
		Regular Expressions	W9
6	1	FILES AND DIRECTORIES: Listing and manipulating Directories and testing files.	R1:P(417-426)
7	1	BASIC INPUT AND OUTPUT: Opening Stream, Reading from a Stream	R1:P(461-466)
8	1	Writing to a stream ,Random Access Methods	W8,W9
		Closing, Flusing and testing streams	W8,W9
9	1	Recapitulation and important questions discussion	
Total no. of periods planned for Unit IV : 9			

Textbooks (T1) : Dave Thomas, Andrew Hunt, 2013, Programming Ruby 1.9 & 2.0: The Pragmatic Programmers Guide 2nd Edition, The Pragmatic Bookshelf.
Reference Book (R1): David Flanagan, 2008, "The Ruby Programming Language", 1st Edition, O'Reilly Media.
Website (W8) : www.newcircle.com/bookshelf/ruby_tutorial
Website (W9) : https://www.sitepoint.com/ruby-mixins-2/
Journal(J2) : A self guided course on streams, files, file formats and sockets issue 79:: Published by Gregary brown.2014

Karpagam Academy of Higher Education
(Established Under Section 3 of UGC Act 1956)
Eachanari (po), Coimbatore-21

COURSE: RUBY PROGRAMMING [17CAP504W]
LECTURE PLAN - UNIT V

S.No.	Lecture Duration (Hr)	Topics to be Covered	Support Materials
1	1	THREADS AND PROCESSES: Thread Life Cycle	W6,W10
2		– Thread scheduling – Thread Exclusion –Dead lock	W6,W11
3		Ruby Tk: Introduction, Widgets and classes	W1,W6
4		Networks: A Very Simple Client, Avery simple server	W1,W6
5		Datagram - A Multiplexing Server - Fetching Web Pages.	W6
6		Ruby on Rails: Building a development	W6
7	1	Environment: Installation – Installing Databases	W6
8	1	Code editors – web server Configuration, Creating an web application	W10,W6
9	1	Recapitulation and Discussion on important questions	
10	1	Discussion of previous ESE question papers	
11	1	Discussion of previous ESE question papers	
12	1	Discussion of previous ESE question papers	
Total no. of periods planned for Unit V : 12			

Textbooks (T1) : Dave Thomas, Andrew Hunt, 2013, Programming Ruby 1.9 & 2.0: The Pragmatic Programmers Guide 2nd Edition, The Pragmatic Bookshelf.
Reference Book (R1): David Flanagan, 2008, "The Ruby Programming Language", 1st Edition, O'Reilly Media.

Website (W10) : www.sitepoint.com/threads-ruby
Website (W11) : www.ruby-doc.org/docs/test.html

Unit I

Introduction to Ruby: Installing Ruby - THE STRUCTURE AND EXECUTION OF RUBY PROGRAMS: Lexical Structure- Syntactic Structure - Block Structure in Ruby- File Structure - Program Execution. DATA TYPES: Numbers - Text - String Literals - Character Literals - String Operators - Accessing Characters and Substrings - Iterating Strings – Arrays – Hashes – Ranges – Symbols – True & False – Ruby Documentation: RDoc and ri.

Introduction to Ruby

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

Installing Ruby

You can download Ruby from https://www.ruby-lang.org/en/downloads/

The Structure and Execution of Ruby Programs: Lexical Structure

The Ruby interpreter parses a program as a sequence of *tokens*. Tokens include com3ments, 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

Comments in Ruby begin with a # character and continue to the end of the line. The Ruby interpreter ignores the # character and any text that follows it (but does not ignore the newline character, which is meaningful whitespace and may serve as a statement terminator). If a # character appears within a string or regular expression literal (see Chapter 3), then it is simply part of the string or regular expression and does not introduce a comment:

```
# This entire    line is a comment
x = "#This is  a string"                    #    And this is a comment
```

Embedded documents

Ruby supports another style of multiline comment known as an *embedded document*. These start on a line that begins =begin and continue until (and include) a line that begins =end. Any text that appears after =begin or =end is part of the comment and is also ignored, but that extra text must be separated from the =begin and =end by at least one space.

Embedded documents are a convenient way to comment out long blocks of code with-out prefixing each line with a # character:

```
=begin Someone needs to fix the broken code below!
Any code here is commented out

=end
```

Note that embedded documents only work if the = signs are the first characters of each line:

```
=begin This used to begin a comment. Now it is itself commented out! The code that goes here is no longer commented out
=end
```

Documentation comments

Ruby programs can include embedded API documentation as specially formatted com-ments that precede method, class, and module definitions. The rdoc tool extracts doc-umentation comments from Ruby source and formats them as HTML or prepares them for display by ri. Documentation of the rdoc tool is beyond the scope of this book; see the file lib/rdoc/README in the Ruby source code for details.

Documentation comments must come immediately before the module, class, or method whose API they document. They are usually written as multiline comments where each line begins with #, but they can also be written as embedded documents that start =begin rdoc. (The rdoc tool will not process these comments if you leave out the "rdoc".)

Literals

Literals are values that appear directly in Ruby source code. They include numbers, strings of text, and regular expressions. (Other literals, such as array and hash values, are not individual tokens but are more complex expressions.)

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. See §4.6 for a complete list of Ruby operators. Punc-tuation characters also serve to delimit string, regular expression, array, and hash literals, and to group and separate expressions, method arguments, and array indexes. We'll see miscellaneous other uses of punctuation scattered throughout Ruby syntax.

Identifiers

An identifier is simply a name. Ruby uses identifiers to name variables, methods, classes, and so forth. Ruby identifiers consist of letters, numbers, and underscore characters, but they may not begin with a number. Identifiers may not include whitespace or nonprinting characters, and they may not include punctuation characters except as described here.
Identifiers that begin with a capital letter A–Z are constants, and the Ruby interpreter will issue a warning (but not an error) if you alter the value of such an identifier. Class and module names must begin with initial capital letters. The following are identifiers:

```
i
x2

old_value_internal# Identifiers may begin with underscores

PI          # Constant
```

By convention, multiword identifiers that are not constants are written with under-scores like_this, whereas multiword constants are written LikeThis or LIKE_THIS.

Case sensitivity

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.

Unicode characters in identifiers

Ruby's rules for forming identifiers are defined in terms of ASCII characters that are not allowed. In general, all characters outside of the ASCII character set are valid in identifiers, including characters that appear to be punctuation. In a UTF-8 encoded file, for example, the following Ruby code is valid:

```
def ×(x,y)  # The name of this method is the Unicode multiplication
sign

  x*y # The body of this method multiplies itsarguments
  end
```

The special rules about forming identifiers are based on ASCII characters and are not enforced for characters outside of that set. An identifier may not begin with an ASCII digit, for example, but it may begin with a digit from a non-Latin alphabet. Similarly, an identifier must begin with an ASCII capital letter in order to be considered a constant. The identifier Å, for example, is not a constant.

Two identifiers are the same only if they are represented by the same sequence of bytes. Some character sets, such as Unicode, have more than one codepoint that represents the same character. No Unicode normalization is performed in Ruby, and two distinct codepoints are treated as distinct characters, even if they have the same meaning or are represented by the same font glyph.

2.1.4.3 Punctuation in identifiers

Punctuation characters may appear at the start and end of Ruby identifiers. They have the following meanings:

Global variables are prefixed with a dollar sign. Following Perl's example, Rubydefines a number of global variables that include other punctuation characters, such as $_ and $-K. See Chapter 10 for a list of these special globals.

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. This naming convention is often to distinguish mutator methods that alter the object on which they are invoked from variants that return a modified copy of the original object.

Here are some example identifiers that contain leading or trailing punctuation characters:

```
$files          # A global variable
@data           # An instance variable
@@counter       # A class variable
empty?          # A Boolean-valued method or predicate

sort!           # An in-place alternative to the regular sort method

timeout=        # A method invoked by assignment
```

A number of Ruby's operators are implemented as methods, so that classes can redefine them for their own purposes. It is therefore possible to use certain operators as method names as well. In this context, the punctuation character or characters of the operator are treated as identifiers rather than operators.

Syntactic Structure

The basic unit of syntax in Ruby is the *expression*. The Ruby interpreter *evaluates* ex-pressions, producing values. The simplest expressions are *primary expressions*, which represent values directly. Number and string literals, described earlier in this chapter, are primary expressions. Other primary expressions include certain keywords such as true, false, nil, and self. Variable references are also primary expressions; they evaluate to the value of the variable.

more complex values can be written as compound expressions:

```
[1,2,3]             # An Array literal

{1=>"one",
2=>"two"}           # A Hash literal

1..3                # A Range literal
```

Operators are used to perform computations on values, and compound expressions are built by combining simpler subexpressions with operators:

```
= 1              # An assignment expression

= x + 1      # An expression with two operators
```

Expressions can be combined with Ruby's keywords to create *statements*, such as the if statement for conditionally executing code and the while statement for repeatedly executing code:

```
if x < 10 then  # If this expression is true

x = x + 1       # Then execute this tatement

end             # Marks the end of the conditional

 while x     10
       < do     # While this expression is true...

 print x        # Execute this statement

                # Then execute this
x = x + 1       statement

end             # Marks the end of the loop
```

Block Structure in Ruby

Ruby programs have a block structure. Module, class, and method definitions, and most of Ruby's statements, include blocks of nested code. These blocks are delimited by keywords or punctuation and, by convention, are indented two spaces relative to the delimiters. 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* (in practice, however, the term "block" is often used for both). 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 are never delimited with curly braces in Ruby—keywords usually serve as the delimiters instead. The specific syntax for statement bodies, method bodies, and class and module bodies are documented in Chapters 5, 6, and 7.

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. These rules are related to the deployment of Ruby programs and are not directly relevant to the language itself.

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.

Second, if a Ruby program contains a "coding" comment that comment must appear on the first line or on the second line if the first line is a shebang.

Third, if a file contains a line that consists of the single token __END__ with no whitespace before or after, then the Ruby interpreter stops processing the file at that point. The remainder of the file may contain arbitrary data that the program can read using the IO stream object DATA. (See Chapter 10 and §9.7 for more about this global constant.)

Ruby programs are not required to fit in a single file. Many programs load additional Ruby code from external libraries, for example. Programs use require to load code from another file. require searches for specified modules of code against a search path, and prevents any given module from being loaded more than once. See §7.6 for details.

The following code illustrates each of these points of Ruby file structure:

```
#!/usr/bin/ruby -w          shebang comment
# -*- coding: utf-8 -*-     coding comment
require 'socket'     load networking library

... program code goes here

__END__ mark end of code
... program data goes here
```

Program Execution

Ruby is a scripting language. This means that Ruby programs are simply lists, or scripts, of statements to be executed. By default, these statements are executed sequentially, in the order they appear. Ruby's control structures (described in Chapter 5) alter this default execution order and allow statements to be executed conditionally or repeat-edly, for example.

Programmers who are used to traditional static compiled languages like C or Java may find this slightly confusing. There is no special main method in Ruby from which exe-cution begins. The Ruby interpreter is given a script of statements to execute, and it begins executing at the first line and continues to the last line.

(Actually, that last statement is not quite true. The Ruby interpreter first scans the file for BEGIN statements, and executes the code in their bodies. Then it goes back to line 1 and starts executing sequentially. See §5.7 for more on BEGIN.)

Another difference between Ruby and compiled languages has to do with module, class, and method definitions. In compiled languages, these are syntactic structures that are processed by the compiler. In Ruby, they are statements like any other. When the Ruby interpreter encounters a class definition, it executes it, causing a new class to come into existence. Similarly, when the Ruby interpreter encounters a method definition, it executes it, causing a new method to be defined. Later in the program, the interpreter will probably encounter and execute a method invocation expression for the method, and this invocation will cause the statements in the method body to be executed.

The Ruby interpreter is invoked from the command line and given a script to execute. Very simple one-line scripts are sometimes written directly on the command line. More commonly, however, the name of the file containing the script is specified. The Ruby interpreter reads the file and

executes the script. It first executes any BEGIN blocks. Then it starts at the first line of the file and continues until one of the following happens:

It executes a statement that causes the Ruby program to terminate.

It reaches the end of the file.

It reads a line that marks the logical end of the file with the token `__END__`.

Before it quits, the Ruby interpreter typically (unless the exit! method was called) executes the bodies of any END statements it has encountered and any other "shutdown hook" code registered with the at_exit function.

DATA TYPES: Numbers

Numbers

Ruby includes five built-in classes for representing numbers, and the standard library includes three more numeric classes that are sometimes useful.

All number objects in Ruby are instances of Numeric. All integers are instances of Integer. If an integer value fits within 31 bits (on most implementations), it is an instance of Fixnum. Otherwise, it is a Bignum. Bignum objects represent integers of arbi-trary size, and if the result of an operation on Fixnum operands is too big to fit in a Fixnum, that result is transparently converted to a Bignum. Similarly, if the result of an operation on Bignum objects falls within the range of Fixnum, then the result is a Fixnum. Real numbers are approximated in Ruby with the Float class, which uses the native floating-point representation of the platform.

The Complex, BigDecimal, and Rational classes are not built-in to Ruby but are distrib-uted with Ruby as part of the standard library. The Complex class represents complex numbers, of course. BigDecimal represents real numbers with arbitrary precision, using a decimal representation rather than a binary representation. And Rational represents rational numbers: one integer divided by another.

All numeric objects are *immutable*; there are no methods that allow you to change the value held by the object. If you pass a reference to a numeric object to a method, you need not worry that the method will modify the object. Fixnum objects are commonly used, and Ruby implementations typically treat them as immediate values rather than as references. Because numbers are immutable, however, there is really no way to tell the difference.

Integer Literals

An integer literal is simply a sequence of digits:

```
0

123

12345678901234567890
```

If the integer values fit within the range of the Fixnum class, the value is a Fixnum. Otherwise, it is a Bignum, which supports integers of any size. Underscores may be inserted into integer literals (though not at the beginning or end), and this feature is sometimes used as a thousands separator:

```
1_000_000_000    # One billion (or 1,000 million in the UK)
```

If an integer literal begins with zero and has more than one digit, then it is interpreted in some base other than base 10. Numbers beginning with 0x or 0X are hexadecimal (base 16) and use the letters a through f (or A through F) as digits for 10 through 15. Numbers beginning 0b or 0B are binary (base 2) and may only include digits 0 and 1. Numbers beginning with 0 and no subsequent letter are octal (base 8) and should consist of digits between 0 and 7. Examples:

```
0377          # Octal representation of 255
```

```
0b1111_11     Binary representation of
11            # 255

              Hexadecimal representation of
0xFF          # 255
```

To represent a negative number, simply begin an integer literal with a minus sign. Literals may also begin with a plus sign, although this never changes the meaning of the literal.

Floating-Point Literals

A floating-point literal is an optional sign followed by one or more decimal digits, a decimal point (the . character), one or more additional digits, and an optional expo-nent. An exponent begins with the letter e or E, and is followed by an optional sign and one or more decimal digits. As with integer literals, underscores may be used within

floating-point literals. Unlike integer literals, it is not possible to express floating-point values in any radix other than base 10. Here are some examples of floating-point literals:

```
0.0
```

-3.14

6.02e23 # This means 6.02×10^{23}

1_000_000.01 # One million and a little bit more

Ruby requires that digits appear before and after the decimal point. You cannot simply write .1, for example; you must explicitly write 0.1. This is necessary to avoid ambiguity in Ruby's complex grammar. Ruby differs from many other languages in this way.

Text

Text is represented in Ruby by objects of the String class. Strings are mutable objects, and the String class defines a powerful set of operators and methods for extracting substrings, inserting and deleting text, searching, replacing, and so on. Ruby provides a number of ways to express string literals in your programs, and some of them support a powerful string interpolation syntax by which the values of arbitrary Ruby expressions can be substituted into string literals. The sections that follow explain string and character literals and string operators.

Textual patterns are represented in Ruby as Regexp objects, and Ruby defines a syntax for including regular expressions literally in your programs. The code /[a-z]\d+/, for example, represents a single lowercase

letter followed by one or more digits. Regular expressions are a commonly used feature of Ruby, but regexps are not a fundamental datatype in the way that numbers, strings, and arrays are.

String Literals

Ruby provides quite a few ways to embed strings literally into your programs.

Single-quoted string literals

The simplest string literals are enclosed in single quotes (the apostrophe character).

The text within the quote marks is the value of the string:

```
'This is a simple Ruby string literal'
```

If you need to place an apostrophe within a single-quoted string literal, precede it with a backslash so that the Ruby interpreter does not think that it terminates the string:

```
'Won\'t you read O\'Reilly\'s book?'
```

The backslash also works to escape another backslash, so that the second backslash is not itself interpreted as an escape character. Here are some situations in which you need to use a double backslash:

```
'This string literal ends with a single backslash: \\'
```

```
'This is a backslash-quote: \\\"
```

```
'Two backslashes: \\\\'
```

In single-quoted strings, a backslash is not special if the character that follows it is anything other than a quote or a backslash. Most of the time, therefore, backslashes need not be doubled (although they can be) in string literals. For example, the following two string literals are equal:

```
'a\b' == 'a\\b'
```

Single-quoted strings may extend over multiple lines, and the resulting string literal includes the newline characters. It is not possible to escape the newlines with a backslash:

```
'This is a long string literal \

that includes a backslash and a newline'
```

If you want to break a long single-quoted string literal across multiple lines without embedding newlines in it, simply break it into multiple adjacent string literals; the Ruby interpreter will concatenate them during the parsing process. Remember, though, that you must escape the newlines (see Chapter 2) between the literals so that Ruby does not interpret the newline as a statement terminator:

```
message =

'These three literals are '\

'concatenated into one by the interpreter. '\

'The resulting string contains no newlines.'
```

Double-quoted string literals

String literals delimited by double quotation marks are much more flexible than single-quoted literals. Double-quoted literals support quite a few backslash escape sequences, such as `\n` for newline, `\t` for tab, and `\"` for a quotation mark that does not terminate the string:

```
"\t\"This quote begins with a tab and ends with
a newline\"\n"


"\\"    # A single backslash
```

Character Literals

Single characters can be included literally in a Ruby program by preceding the character with a question mark. No quotation marks of any kind are used:

```
?A # Character literal for the ASCII character
A


?" # Character literal for the double-quote
character
```

```
# Character literal for the question mark
character
```

Although Ruby has a character literal syntax, it does not have a special class to represent single characters.

String Operators

The `String` class defines several useful operators for manipulating strings of text. The

operator concatenates two strings and returns the result as a new `String` object:

```
planet = "Earth"

"Hello" + " " + planet # Produces "Hello Earth"
```

Java programmers should note that the + operator does not convert its righthand operand to a string; you must do that yourself:

```
"Hello planet #" + planet_number.to_s    # to_s converts to a string
```

Of course, in Ruby, string interpolation is usually simpler than string concatenation with +. With string interpolation, the call to to_s is done automatically:

```
"Hello planet ##{planet_number}"
```

The << operator appends its second operand to its first, and should be familiar to C++ programmers. This operator is very different from +; it alters the lefthand operand rather than creating and returning a new object:

```
greeting = "Hello"

greeting << " " << "World"

puts greeting# Outputs "Hello World"
```

Like +, the << operator does no type conversion on the righthand operand. If the right-hand operand is an integer, however, it is taken to be a character code, and the corresponding character is appended. In Ruby 1.8, only integers between 0 and 255 are allowed. In Ruby 1.9, any integer that represents a valid codepoint in the string's encoding can be used:

```
alphabet =
"A"

alphabet          #
<<          ?B  Alphabet  is now "AB"

alphabe   <         And now it
t         < 67  # is              "ABC"

alphabe   < 25              Rub 1.8: codes must be >=0 and <
t         < 6   # Error in  y     256
```

The * operator expects an integer as its righthand operand. It returns a String that repeats the text specified on the lefthand side the number of times specified by the righthand side:

```
ellipsis = '.'*3    # Evaluates to '...'
```

If the lefthand side is a string literal, any interpolation is performed just once before the repetition is done. This means that the following too-clever code does not do what you might want it to:

```
a = 0;
```

```
"#{a=a+1} " * 3   # Returns "1 1 1 ", not "1 2
3 "
```

String defines all the standard comparison operators. == and != compare strings for equality and inequality. Two strings are equal if—and only if—they have the same length and all characters are equal. <, <=, >, and >= compare the relative order of strings by comparing the character codes of the characters that make up a string. If one string

is a prefix of another, the shorter string is less than the longer string. Comparison is based strictly on character codes. No normalization is done, and natural language col-lation order (if it differs from the numeric sequence of character codes) is ignored.

String comparison is case-sensitive.* Remember that in ASCII, the uppercase letters all have lower codes than the lowercase letters. This means, for example, that "Z" < "a". For case-insensitive comparison of ASCII characters, use the casecmp method (see §9.1) or convert your strings to the same case with downcase or upcase methods before comparing them. (Keep in mind that Ruby's knowledge of upper- and lowercase letters is limited to the ASCII character set.)

Accessing Characters and Substrings

Perhaps the most important operator supported by `String` is the square-bracket array-index operator `[]`, which is used for extracting or altering portions of a string. This operator is quite flexible and can be used with a number of different operand types. It can also be used on the lefthand side of an assignment, as a way of altering string content.

In Ruby 1.8, a string is like an array of bytes or 8-bit character codes. The length of this array is given by the `length` or `size` method, and you get or set elements of the array simply by specifying the character number within square brackets:

```
s = 'hello'; # Ruby 1.8

s[0]     # 104: the ASCII character code for the
first character 'h'

s[s.length-1]     # 111: the character code of
the last character 'o'
```

```
s[-1]    # 111: another way of accessing the
last character


s[-2]    # 108: the second-to-last character


s[-s.length] # 104: another way of accessing
the first character


s[s.length]  # nil: there is no character at
that index
```

Notice that negative array indexes specify a 1-based position from the end of the string. Also notice that Ruby does not throw an exception if you try to access a character beyond the end of the string; it simply returns `nil` instead.

Ruby 1.9 returns single-character strings rather than character codes when you index a single character. Keep in mind that when working with multibyte strings, with char-acters encoded using variable numbers of bytes, random access to characters is less efficient than access to the underlying bytes:

```
s =       #
'hello'; Ruby 1.9

          #    the first character of the
s[0]      'h': string, as a string

s[s.leng  #
th-1]     'o': the last character 'o'

           'o another way of accessing the
s[-1]     #': last character

           'l the second-to-last
s[-2]     #': character
```

In Ruby 1.8, setting the deprecated global variable `$=` to `true` makes the ==, <, and related comparison operators perform case-insensitive comparisons. You should not do this, however; setting this variable produces a warning message, even if the Ruby interpreter is invoked without the `-w` flag. And in Ruby 1.9, `$=` is no longer supported.

```
s[-s.length] # 'h': another way of accessing
the first character


s[s.length]  # nil: there is no character at
that index
```

To alter individual characters of a string, simply use brackets on the lefthand side of an assignment expression. In Ruby 1.8, the righthand side may be an ASCII character code or a string. In Ruby 1.9, the righthand side must be a string. You can use character literals in either version of the language:

```
s[0] =     Replace first
?H         # character with a    capital H

s[-1] =    Replace  character with a
?O         # last     capital         O

 s[s.length] ERROR!   assign      end  the
      = ?! # Can't    beyond the  of   string
```

The righthand side of an assignment statement like this need not be a character code: it may be any string, including a multicharacter string or the empty string. Again, this works in both Ruby 1.8 and Ruby 1.9:

```
s =        # Begin
"hello"    with    a greeting

   =""     #
s[-         Dele th last character; s is now
```

```
1]          te   e  "hell"

s[-  "p!    Chan ne last character and add one; s
1]  ="     #ge   w  is now "help!"
```

More often than not, you want to retrieve substrings from a string rather than individual character codes. To do this, use two comma-separated operands between the square brackets. The first operand specifies an index (which may be negative), and the second specifies a length (which must be nonnegative). The result is the substring that begins at the specified index and continues for the specified number of characters:

```
s =
"hello"

s[0,2]    # "he"

          # "o": returns a string, not the
s[-1,1]   character code ?o

          # "": a zero-length substring
s[0,0]    is always empty

          # "hello": returns all the characters
s[0,10]   that are available

s[s.leng  # "": there is an empty string
```

```
th,1]     immediately beyond the end

s[s.length # nil: it is an error to read
+1,1]      past that

           # nil: negative lengths don't
s[0,-1]    make any sense
```

If you assign a string to a string indexed like this, you replace the specified substring with the new string. If the righthand side is the empty string, this is a deletion, and if the lefthand side has zero-length, this is an insertion:

```
s =
"hello"

s[0,             # Replace first     capital
1]   = "H"       letter with a       letter

s[s.length,0] = " world"  # Append by   end of the
assigning beyond the                    string

s[5,              Insert a comma, without
0]   =","        #deleting anything

s[5,              Delete with no     =
6]   =""         #insertion; s      = "Hellod"
```

Another way to extract, insert, delete, or replace a substring is by indexing a string with a `Range` object. We'll explain ranges in detail in §3.5 later. For our purposes here,

`Range` is two integers separated by dots. When a `Range` is used to index a string, the return value is the substring whose characters fall within the `Range`:

```
s =
"hello"

           # "ll":
           characters 2 and
s[2..3]    3

s[-3..-     "llo":    indexes work,
1]         #negative  too

                            one
            "h": this  inclu character
s[0..0]    #Range     des   index
```

```
s[0...0    # "": this
]          Range is     Empty

            "": this
s[2..1]    #Range is    also empty

s[7..10     nil: this Range is outside
]          #the string bounds

s[-2..-
1] =        Replacement: s becomes
"p!"       #"help!"
```

It is also possible to index a string with a string. When you do this, the return value is the first substring of the target string that matches the index string, or `nil`, if no match is found. This form of string indexing is really only useful on the lefthand side of an assignment statement when you want to replace the matched string with some other string:

```
           #
s =        Star with the word
"hello"    t    "hello"

           #
while(s[" Whil the string contains the
l"])       e    substring "l"
```

```
    s["l"] =   Replace first occurrence
    "L";      #of "l" with "L"

  end         #Now we have "heLLo"
```

Iterating Strings

In Ruby 1.8, the `String` class defines an `each` method that iterates a string line-by-line. The `String` class includes the methods of the `Enumerable` module, and they can be used to process the lines of a string. You can use the `each_byte` iterator in Ruby 1.8 to iterate through the bytes of a string, but there is little advantage to using `each_byte` over the `[]` operator because random access to bytes is as quick as sequential access in 1.8.

The situation is quite different in Ruby 1.9, which removes the `each` method, and in which the `String` class is no longer `Enumerable`. In place of `each`, Ruby 1.9 defines three clearly named string iterators: `each_byte` iterates sequentially through the individual bytes that comprise a string; `each_char` iterates the characters; and `each_line` iterates the lines. If you want to process a string character-by-character, it may be more efficient to use `each_char` than to use the `[]` operator and character indexes:

```
s = "¥1000"

s.each_char {|x| print "#{x} " } # Prints "¥ 1 0 0 0". Ruby
1.9
```

```
0.upto(s.size-1) {|i| print "#{s[i]} "}    # Inefficient with
multibyte chars
```

Arrays

An array is a sequence of values that allows values to be accessed by their position, or *index*, in the sequence. In Ruby, the first value in an array has index `0`. The `size` and `length` methods return the number of elements in an array. The last element of the array is at index `size-1`. Negative index values count from the end of the array, so the last element of an array can also be accessed with an index of `-1`. The second-to-last has an index of `-2`, and so on. If you attempt to read an element beyond the end of an array (with an index >= `size`) or before the beginning of an array (with an index < `-size`), Ruby simply returns `nil` and does not throw an exception.

Ruby's arrays are untyped and mutable. The elements of an array need not all be of the same class, and they can be changed at any time. Furthermore, arrays are dynamically resizeable; you can append elements to them and they grow as needed. If you assign a value to an element beyond the end of the array, the array is automatically extended with `nil` elements. (It is an error, however, to assign a value to an element before the beginning of an array.)

An array literal is a comma-separated list of values, enclosed in square brackets:

```
            #  array that     Fixn
[1, 2, 3] An holds three    um  objects

[-10...0,    A array two ranges;   commas are
0..10,]     #n of    trailing      allowed

[[1,2],[3,4  A array nested
],[5]]      #n of    arrays
```

Ruby includes a special-case syntax for expressing array literals whose elements are short strings without spaces:

```
            a
words =     test  # Same ['this', 'is',
%w[this is  ]     as:    'a', 'test']

open = %w|          Same  ['(', '[',
( [ {      <|     #as:   '{', '<']

white =    \r       Same  ["\s", "\t",
%W(\s \t   \n)     #as:   "\r", "\n"]
```

You can also create arrays with the `Array.new` constructor, and this provides options for programmatically initializing the array elements:

```
empty =         []: returns emp arr
Array.new     # a new       ty  ay

   =
ni Array.new(  [nil,  nil]new array with 3
ls 3)          # nil,      :nil elements
```

```
 zeros =          [0,  0, 0]:  arr with 0
Array.new(4, 0) # 0,   new     ay  4    elements

   =                          an
co Array.new(n   Ma   ne copy existin
py ils)         # ke  a w    of g       array

count =                     #
Array.new(3)      i+   [1,2,3]  elements computed
{|i|              1}         :3 from index
```

To obtain the value of an array element, use a single integer within square brackets:

```
a = [0,   9,    # Array holds the squares
1, 4,     16]   of the indexes

a[0]    # First element is 0

          Last
a[-1]   # element is 16

a[-2]   # Second to last element is 9

a[a.siz   Another way to query the
e-1]    # last element

a[-
a.size   Another way to query the
]       # first element

          Query beyo the end returns
a[8]    # ing   nd   nil

          Query befo the start returns
a[-8]   # ing   re   nil, too
```

Like strings, arrays can also be indexed with two integers that represent a starting index and a number of elements, or a `Range` object. In either case, the expression returns the specified subarray:

```
a = ('a'..'e').to_a    # Range converted to ['a', 'b', 'c',
'd', 'e']


            # []: this subarray has
a[0,0]      zero elements

            # ['b']: a one-element
a[1,1]      array

            # ['d','e']: the last two
a[-2,2]     elements of the array

            # ['a', 'b', 'c']: the first
a[0..2]     three elements

a[-2..-     # ['d','e']: the last two
1]          elements of the array

a[0...-     # ['a', 'b', 'c', 'd']: all but
1]          the last element
```

When used on the lefthand side of an assignment, a subarray can be replaced by the elements of the array on the righthand side. This basic operation works for insertions and deletions as well:

```
a[0,2] = 'B'    a becomes  'B''c', 'd',
['A',    ]     #['A',       ,'e']
```

```
a[2...5]=    'D', a becomes  'B''C', 'D',
['C',      'E'] # ['A',       ,'E']

a[0,0] =        Insert    at the
[1,2,3]        # elements   beginning of a

a[0. =          Delete   eleme
.2]  []        # those    nts

     =
a[-  ['Z'       Replace  eleme
1,1] ]         # last       ntwith another

a[-  =          For single    the array is
1,1] 'Z'       # elements,     optional

a[-  =   # Delete last 2  i 1.8replace with
2,2] nil elements        n   ;nil in 1.9
```

In addition to the square bracket operator for indexing an array, the `Array` class defines a number of other useful operators. Use `+` to concatenate two arrays:

```
a =  2, 3] +    #   2 3  4,
[1,  [4, 5]     [1, , ,  5]

 =   [[6, 7,     [1 2 3  4, [6, 7,
aa +8]]        # , , ,  5,    8]]

 =               Erro righthand    be an
aa +9          #   r: side must    array
```

The – operator subtracts one array from another. It begins by making a copy of its lefthand array, and then removes any elements from that copy if they appear anywhere in the righthand array:

```
['a', 'b', 'c', 'b', 'a'] - ['b', 'c', 'd']     # ['a', 'a']
```

The + operator creates a new array that contains the elements of both its operands. Use

to append elements to the end of an existing array:

```
a =       # Start with an
[]        empty array

a         # a
<< 1      is   [1]

  <   << #  i
a < 2   3 a  s [1, 2, 3]

  < [4,5  #  i [1, 2, 3, [4,
a < ,6]   a  s 5, 6]]
```

Like the `String` class, `Array` also uses the multiplication operator for repetition:

```
a = [0] * 8  # [0, 0, 0, 0, 0, 0, 0, 0]
```

The `Array` class borrows the Boolean operators `|` and `&` and uses them for union and intersection. `|` concatenates its arguments and then removes all duplicate elements from the result. `&` returns an array that holds elements that appear in both of the operand arrays. The returned array does not contain any duplicate elements:

```
a =
[1, 1, 2, 2, 3, 3, 4]

b =
[5, 5, 4, 4, 3, 3, 2]

a |   [1, 2, 3, 4, 5]: duplicates are
b    # removed

b |   [5, 4, 3, 2, 1]: elements are the same,
a    # but order is different

a &
b    # [2, 3, 4]

b &
a    # [4, 3, 2]
```

Note that these operators are not transitive: `a|b` is not the same as `b|a`, for example. If you ignore the ordering of the elements, however, and consider the arrays to be unordered sets, then these operators make more sense. Note also that the algorithm by which union and intersection are performed is not specified, and there are no guarantees about the order of the elements in the returned arrays.

The `Array` class defines quite a few useful methods. The only one we'll discuss here is the `each` iterator, used for looping through the elements of an array:

```
a = ('A'..'Z').to_a    # Begin with an array of letters

a.each {|x| print x }  # Print the alphabet, one letter at a
time
```

Hashes

A *hash* is a data structure that maintains a set of objects known as *keys*, and associates a value with each key. Hashes are also known as *maps* because they map keys to values. They are sometimes called *associative arrays* because they associate values with each of the keys, and can be thought of as arrays in which the array index can be any object instead of an integer. An example makes this clearer:

```
# This hash will map the names of digits to the digits
themselves

numbers = Hash.new     # Create a new, empty, hash object

numbers["one"] = 1     # Map the String "one" to the Fixnum 1

numbers["two"] = 2 # Note that we are using array
notation here numbers["three"] = 3
```

```
sum = numbers["one"] + numbers["two"] # Retrieve values like
this
```

Hash Literals

A hash literal is written as a comma-separated list of key/value pairs, enclosed within curly braces. Keys and values are separated with a two-character "arrow": `=>`. The `Hash` object created earlier could also be created with the following literal:

```
numbers = { "one" => 1, "two" => 2, "three" => 3 }
```

In general, `Symbol` objects work more efficiently as hash keys than strings do:

```
numbers = { :one => 1, :two => 2, :three => 3 }
```

Symbols are immutable interned strings, written as colon-prefixed identifiers; they are explained in more detail in §3.6 later in this chapter.

Ruby 1.8 allows commas in place of arrows, but this deprecated syntax is no longer supported in Ruby 1.9:

```
numbers = { :one, 1, :two, 2, :three, 3 } # Same, but harder
to read
```

Ranges

`Range` object represents the values between a start value and an end value. Range literals are written by placing two or three dots between the start and end value. If two The result is a syntax much like that used by JavaScript objects.

dots are used, then the range is *inclusive* and the end value is part of the range. If three dots are used, then the range is *exclusive* and the end value is not part of the range:

```
1..10  # The integers 1 through 10, including 10

1.0...10.0 # The numbers between 1.0 and 10.0, excluding 10.0 itself
```

Test whether a value is included in a range with the `include?` method (but see below for a discussion of alternatives):

```
cold_war = 1945..1989

cold_war.include? birthdate.year
```

Implicit in the definition of a range is the notion of ordering. If a range is the values between two endpoints, there obviously must be some way to compare values to those endpoints. In Ruby, this is done with the comparison operator `<=>`, which compares its two operands and evaluates to `-1`, `0`, or `1`, depending on their relative order (or equality). Classes such as numbers and strings that have an ordering define the `<=>` operator. A value can only be used as a range endpoint if it responds to this operator. The endpoints of a range and the values “in” the range are typically all of the same class. Technically, however,

any value that is compatible with the `<=>` operators of the range endpoints can be considered a member of the range.

The primary purpose for ranges is comparison: to be able to determine whether a value is in or out of the range. An important secondary purpose is iteration: if the class of the endpoints of a range defines a `succ` method (for successor), then there is a discrete set of range members, and they can be iterated with `each`, `step`, and `Enumerable` methods. Consider the range `'a'..'c'`, for example:

```
r = 'a'..'c'

r.each {|l| print
"[#{l}]"}          # Prints "[a][b][c]"

r.step(2) { |l| print
"[#{l}]"}          #Prints "[a][c]"

                    => ['a','b','c']:
r.to_a             #Enumerable defines to_a
```

The reason this works is that the `String` class defines a `succ` method and `'a'.succ` is `'b'` and `'b'.succ` is `'c'`. Ranges that can be iterated like this are *discrete* ranges. Ranges whose endpoints do not define a `succ` method cannot be iterated, and so they can be called *continuous*. Note that ranges with integer endpoints are discrete, but floating-point numbers as endpoints are continuous.

Ranges with integer endpoints are the most commonly used in typical Ruby programs. Because they are discrete, integer ranges can be used to index

strings and arrays. They are also a convenient way to represent an enumerable collection of ascending values.

Notice that the code assigns a range literal to a variable, and then invokes methods on the range through the variable. If you want to invoke a method directly on a range literal, you must parenthesize the literal, or the method invocation is actually on the endpoint of the range rather than on the `Range` object itself:

```
1..3.to_a  # Tries to call to_a on the number 3

(1..3).to_a     # => [1,2,3]
```

Symbols

A typical implementation of a Ruby interpreter maintains a symbol table in which it stores the names of all the classes, methods, and variables it knows about. This allows such an interpreter to avoid most string comparisons: it refers to method names (for example) by their position in this symbol table. This turns a relatively expensive string operation into a relatively cheap integer operation.

These symbols are not purely internal to the interpreter; they can also be used by Ruby programs. A `Symbol` object refers to a symbol. A symbol literal is written by prefixing an identifier or string with a colon:

```
                # A
:symbol         Symbol literal
```

```
                #   sa
:"symbol"       The me literal

:'another long    Quotes useful for symbols
symbol'         #are    with spaces

s = "string"

                 Th Symb
sym = :"#{s}"   #e  ol   :string
```

Symbols also have a `%s` literal syntax that allows arbitrary delimiters in the same way that `%q` and `%Q` can be used for string literals:

```
%s["] # Same as :'"'
```

Symbols are often used to refer to method names in reflective code. For example, suppose we want to know if some object has an `each` method:

```
o.respond_to? :each
```

Here's another example. It tests whether a given object responds to a specified method, and, if so, invokes that method:

```
name = :size

if o.respond_to? name
```

```
    o.send(name)


  end
```

You can convert a `String` to a `Symbol` using the `intern` or `to_sym` methods. And you can convert a `Symbol` back into a `String` with the `to_s` method or its alias `id2name`:

```
str =       # Begin   Strin
"string"    with    a  g

            #
sym =       Convert   Symbo
str.intern  to      a  l

            #
sym =       Another   t   the same
str.to_sym  way       o do thing

str sym.to    Convert to
=   _s      #    back a  String

str sym.id2   Another t
=   name    #way     o do It
```

Two strings may hold the same content and yet be completely distinct objects. This is never the case with symbols. Two strings with the same content will both convert to exactly the same `Symbol` object. Two distinct `Symbol` objects will always have different content.

Whenever you write code that uses strings not for their textual content but as a kind of unique identifier, consider using symbols instead. Rather than writing a method that expects an argument to be either the string “AM” or “PM”, for example, you could

True, False, and Nil

We saw in §2.1.5 that `true`, `false`, and `nil` are keywords in Ruby. `true` and `false` are the two Boolean values, and they represent truth and falsehood, yes and no, on and off. `nil` is a special value reserved to indicate the absence of value.

Each of these keywords evaluates to a special object. `true` evaluates to an object that is a singleton instance of `TrueClass`. Likewise, `false` and `nil` are singleton instances of `FalseClass` and `NilClass`. Note that there is no `Boolean` class in Ruby. `TrueClass` and `FalseClass` both have `Object` as their superclass.

If you want to check whether a value is `nil`, you can simply compare it to `nil`, or use the method `nil?`:

```
o == nil     # Is o nil?

o.nil?  # Another way to test
```

Note that `true`, `false`, and `nil` refer to objects, not numbers. `false` and `nil` are not the same thing as `0`, and `true` is not the same thing as `1`. When Ruby requires a Boolean value, `nil` behaves like `false`, and any value other than `nil` or `false` behaves like `true`.

RDoc and ri

RDoc is a documentation system. If you put comments in your program files (Ruby or C) in the prescribed **RDoc**format, **rdoc** scans your files, extracts the

comments, organizes them intelligently (indexed according to what they comment on), and creates nicely formatted documentation from them. You can see **RDoc** markup in many of the C files in the Ruby source tree and many of the Ruby files in the Ruby installation.

The Ruby **ri** tool is used to view the Ruby documentation off-line. Open a command window and invoke **ri** followed by the name of a Ruby class, module or method. **ri** will display documentation for you. You may specify a method name without a qualifying class or module name, but this will just show you a list of all methods by that name (unless the method is unique). Normally, you can separate a class or module name from a method name with a period. If a class defines a class method and an instance method by the same name, you must instead use :: to refer to a class method or # to refer to the instance method. Here are some example invocations of ri

1. :ri Array
2. ri Array.sort
3. ri Hash#each
4. ri Math::sqrt

ri dovetails with **RDoc**: It gives you a way to view the information that **RDoc** has extracted and organized. Specifically (although not exclusively, if you customize it), **ri** is configured to display the **RDoc** information from the Ruby source files. Thus on any system that has Ruby fully installed, you can get detailed information about Ruby with a simple command-line invocation of **ri**.

POSSIBLE QUESTIONS

UNIT 1

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. List out the major characteristics of the database approach.

2. Discuss on the different categories of data model.

3. Write short notes on schemas, instances and database state.

4. Elaborate the three schema architecture of DBMS in detail.

5. Write a detailed note on the two types of data independence.

6. Discuss on DBMS languages.

7. What is meant by DBMS interfaces? Explain them in detail.

8. Illustrate the main phases of database design.

9. Describe the different types of attributes in the ER model.

10. Write short notes on entity types and entity sets.

PART – C (10 MARKS)

1. Design an application form using tk classes and validate all fields on Rails framework
2. Explain about different Data Types in Ruby.
3. Explain in detail about Object creation and initialization in Ruby
4. Write a ruby program to create a main thread and execute multiple process through the main thread.
5. Write a note on Defining, Calling and Undefining methods in Ruby

Subject:RUBY PROGRAMMING SUBJECT CODE: 17CAP504W

CLASS: III MCA SEMESTER: V

QESTIONS	OPTION1	OPTION2	OPTION3	OPTIONS4	ANS
_____is the developer of ruby programming language.	Yukihiro "Matz" Matsumoto	Charles babbage	William stallings	David Flanagan	Yukihiro "Matz"
Ruby is a_____programming language	static	dynamic	realistic	static and dynamic	dynamic
Ruby is very strict about_____of its objects.	encapsulation	abstraction	dynamic	binding	
43.times { print "Ruby! " } output:_____	# Prints "Ruby! Ruby! Ruby! "	# Prints "3! 3! 3! "	# Prints "30! 30! 30! "	# Prints "Compiler Exception "	# Prints "Ruby! Ruby! Ruby! "
1.upto(9) {|x| print x } output:_____	# Prints "123456789"	# Prints "123499999"	# Prints "xxxxxxxxx"	# Prints "9999999999"	# Prints "123456789"
The_____causes the interpreter to execute a single specified line of Ruby code.	-f command-line option	-e command-line option	-eee command-line option	–ae dos command-line option	-e command-line option
irb stand for_____It is a Ruby shell.	interactive Ruby	innovative ruby	irregular ruby shell	immediate ruby	interactive Ruby
ri on the command line followed by the name of a_____will display documentation.\	Ruby implementation	Ruby installation	Ruby class, module, or method, and ri	Ruby Interaction	Ruby class, module, or method, and ri
Ruby is a---------- programming paradigm	Procedural Programming	Functional Programming	Object Oriented Programming	Conventional Programming	
OOP is a programming paradigm that uses ---------- to design applications and computer programs	Objects	Classes	Inheritance	Polymorphism	Objects
Everything in Ruby Programming Language can be treated as ----------	Classes	Constructors	Objects	Classes	Objects
Which of the following is NOT a programming concept in OOP?	Methods	Abstraction	Polymorphism	Inheritance	Methods
The ---------- is simplifying complex reality by modeling classes appropriate to the problem	Abstraction	Polymorphism	Encapsulation	Inheritance	Abstraction
The ---------- is the process of using an operator or function in different ways for different data input	Abstraction	Polymorphism	Encapsulation	Inheritance	Polymorphism
The ---------- hides the implementation details of a class from other objects	Abstraction	Polymorphism	Encapsulation	Inheritance	Encapsulation
The ---------- is a way to form new classes using a classes that have been already defined	Abstraction	Polymorphism	Encapsulation	Inheritance	Inheritance
Objects are ----------of a Ruby OOP Program	Methods	Classes	Basic Building Blocks	Constructors	Basic Building Blocks
An object is a combination of ---------- and ----------	Data, Methods	Classes, Methods	Polymorphism, Inheritance	Data, Encapsulation	Data, Methods
Objects communicate together through ----------	Classes	Data	Methods	Templates	Methods
A --------- is a template for an object	Methods	Data	Variables	Class	Class
----------is a special kind of a method	Object	Class	Constructor	Data	Constructor
---------- is automatically called when an object is created	Constructor	Class	Data	Object	
The Constructor in Ruby is called ----------	Initialize	Constructor	Init	Object	Initialize
---------- do not return values	Objects	Methods	Abstract Classes	Constructors	Constructors
Constructors cannot be ----------	Inherited	Called	Created	Initiated	Inherited
The constructor of a parent object is called with a----------method	Initialize	Super	Inherit	Special	Super
An instance variable is a variable defined in a ----------	Object	Class	Method	Abstract Class	Class
Ruby has no ----------	Inheritance	Constructor Overloading	Abstraction	Encapsulation	Constructor Overloading
---------- is the ability to have multiple types of constructors in a class	Constructor Overloading	Initializing	Method Overloading	Inheritance	Constructor Overloading
---------- are functions defined inside the body of a class	Objects	Classes	Methods	Variables	Methods
Methods are used to perform operations within the---------- of our objects	Attributes	Features	Arguments	Parameters	Attributes
Methods are essential in ----------concept	Abstraction	Encapsulation	Inheritance	Polymorphism	Encapsulation
In Ruby, data is accessible only through ----------	Methods	Classes	Objects	Constructors	Methods
Class Variables start with---------- sigils in Ruby	@@	@	!	#	@@
---------- set the visibility of methods and member fields	Keywords	Access modifiers	Objects	Classes	Access modifiers
Ruby has---------- access modifiers	one	Two	Three	Four	Three
Which of the following is NOT an access modifier?	Public	Private	Default	Protected	Default
Access Modifiers can be used only on ----------	Classes	Objects	Constructors	Methods	Methods
By Default Ruby methods are ----------	Public	Private	Protected	Default	Public
The ---------- methods can be accessed from inside the definition of the class as well as from the outside of the class	Private	Public	Protected	Default	Public
---------- protects data against accidental modifications	Default	Class	Objects	Access Modifiers	Access Modifiers
---------- are the only methods that can be called outside the definition of a class	Public Methods	Private Methods	Protected Methods	Default Methods	Public Methods
---------- are the only methods that can be called inside the definition of a class	Public Methods	Private Methods	Protected Methods	Default Methods	private methods
---------- methods can be called with the self keyword specified	Public Methods	Private Methods	Protected Methods	Default Methods	Protected Methods
In Inheritance, the classes we derive from are called ----------	Base Class	Derive Class	Abstract Class	Default Class	Base Class
In Inheritance, the newly formed classes are called ----------	Base Class	Derive Class	Abstract Class	Default Class	Derive Class
---------- is used to reduce code reuse and reduction of complexity of a program	Abstraction	encapsulation	Inheritance	Polymorphism	Inheritance
In Ruby --------- operator is used to create inherit relations	<	>	@@	@	<

Unit I

Introduction to Ruby: Installing Ruby - THE STRUCTURE AND EXECUTION OF RUBY PROGRAMS: Lexical Structure- Syntactic Structure - Block Structure in Ruby- File Structure - Program Execution. DATA TYPES: Numbers - Text - String Literals - Character Literals - String Operators - Accessing Characters and Substrings - Iterating Strings – Arrays – Hashes – Ranges – Symbols – True & False – Ruby Documentation: RDoc and ri.

Introduction to Ruby

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

Installing Ruby

You can download Ruby from https://www.ruby-lang.org/en/downloads/

The Structure and Execution of Ruby Programs: Lexical Structure

The Ruby interpreter parses a program as a sequence of *tokens*. Tokens include com3ments, 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

Comments in Ruby begin with a # character and continue to the end of the line. The Ruby interpreter ignores the # character and any text that follows it (but does not ignore the newline character, which is meaningful whitespace and may serve as a statement terminator). If a # character appears within a string or regular expression literal (see Chapter 3), then it is simply part of the string or regular expression and does not introduce a comment:

```
# This entire    line is a comment
x = "#This is  a string"                    #    And this is a comment
```

Embedded documents

Ruby supports another style of multiline comment known as an *embedded document*. These start on a line that begins =begin and continue until (and include) a line that begins =end. Any text that appears after =begin or =end is part of the comment and is also ignored, but that extra text must be separated from the =begin and =end by at least one space.

Embedded documents are a convenient way to comment out long blocks of code with-out prefixing each line with a # character:

```
=begin Someone needs to fix the broken code below!
Any code here is commented out

=end
```

Note that embedded documents only work if the = signs are the first characters of each line:

```
=begin This used to begin a comment. Now it is itself commented out! The code that goes here is no longer commented out
=end
```

Documentation comments

Ruby programs can include embedded API documentation as specially formatted com-ments that precede method, class, and module definitions. The rdoc tool extracts doc-umentation comments from Ruby source and formats them as HTML or prepares them for display by ri. Documentation of the rdoc tool is beyond the scope of this book; see the file lib/rdoc/README in the Ruby source code for details.

Documentation comments must come immediately before the module, class, or method whose API they document. They are usually written as multiline comments where each line begins with #, but they can also be written as embedded documents that start =begin rdoc. (The rdoc tool will not process these comments if you leave out the "rdoc".)

Literals

Literals are values that appear directly in Ruby source code. They include numbers, strings of text, and regular expressions. (Other literals, such as array and hash values, are not individual tokens but are more complex expressions.)

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. See §4.6 for a complete list of Ruby operators. Punc-tuation characters also serve to delimit string, regular expression, array, and hash literals, and to group and separate expressions, method arguments, and array indexes. We'll see miscellaneous other uses of punctuation scattered throughout Ruby syntax.

Identifiers

An identifier is simply a name. Ruby uses identifiers to name variables, methods, classes, and so forth. Ruby identifiers consist of letters, numbers, and underscore characters, but they may not begin with a number. Identifiers may not include whitespace or nonprinting characters, and they may not include punctuation characters except as described here.
Identifiers that begin with a capital letter A–Z are constants, and the Ruby interpreter will issue a warning (but not an error) if you alter the value of such an identifier. Class and module names must begin with initial capital letters. The following are identifiers:

```
i
x2

old_value_internal# Identifiers may begin with underscores

PI          # Constant
```

By convention, multiword identifiers that are not constants are written with under-scores like_this, whereas multiword constants are written LikeThis or LIKE_THIS.

Case sensitivity

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.

Unicode characters in identifiers

Ruby's rules for forming identifiers are defined in terms of ASCII characters that are not allowed. In general, all characters outside of the ASCII character set are valid in identifiers, including characters that appear to be punctuation. In a UTF-8 encoded file, for example, the following Ruby code is valid:

```
def ×(x,y)  # The name of this method is the Unicode multiplication
sign

  x*y # The body of this method multiplies itsarguments
  end
```

The special rules about forming identifiers are based on ASCII characters and are not enforced for characters outside of that set. An identifier may not begin with an ASCII digit, for example, but it may begin with a digit from a non-Latin alphabet. Similarly, an identifier must begin with an ASCII capital letter in order to be considered a constant. The identifier Å, for example, is not a constant.

Two identifiers are the same only if they are represented by the same sequence of bytes. Some character sets, such as Unicode, have more than one codepoint that represents the same character. No Unicode normalization is performed in Ruby, and two distinct codepoints are treated as distinct characters, even if they have the same meaning or are represented by the same font glyph.

2.1.4.3 Punctuation in identifiers

Punctuation characters may appear at the start and end of Ruby identifiers. They have the following meanings:

Global variables are prefixed with a dollar sign. Following Perl's example, Rubydefines a number of global variables that include other punctuation characters, such as $_ and $-K. See Chapter 10 for a list of these special globals.

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. This naming convention is often to distinguish mutator methods that alter the object on which they are invoked from variants that return a modified copy of the original object.

Here are some example identifiers that contain leading or trailing punctuation characters:

```
$files          # A global variable
@data           # An instance variable
@@counter       # A class variable
empty?          # A Boolean-valued method or predicate

sort!           # An in-place alternative to the regular sort method

timeout=        # A method invoked by assignment
```

A number of Ruby's operators are implemented as methods, so that classes can redefine them for their own purposes. It is therefore possible to use certain operators as method names as well. In this context, the punctuation character or characters of the operator are treated as identifiers rather than operators.

Syntactic Structure

The basic unit of syntax in Ruby is the *expression*. The Ruby interpreter *evaluates* ex-pressions, producing values. The simplest expressions are *primary expressions*, which represent values directly. Number and string literals, described earlier in this chapter, are primary expressions. Other primary expressions include certain keywords such as true, false, nil, and self. Variable references are also primary expressions; they evaluate to the value of the variable.

more complex values can be written as compound expressions:

```
[1,2,3]             # An Array literal

{1=>"one",
2=>"two"}           # A Hash literal

1..3                # A Range literal
```

Operators are used to perform computations on values, and compound expressions are built by combining simpler subexpressions with operators:

```
= 1              # An assignment expression

= x + 1      # An expression with two operators
```

Expressions can be combined with Ruby's keywords to create *statements*, such as the if statement for conditionally executing code and the while statement for repeatedly executing code:

```
if x < 10 then  # If this expression is true

x = x + 1       # Then execute this tatement

end             # Marks the end of the conditional

 while x     10
       < do     # While this expression is true...

 print x        # Execute this statement

                # Then execute this
x = x + 1       statement

end             # Marks the end of the loop
```

Block Structure in Ruby

Ruby programs have a block structure. Module, class, and method definitions, and most of Ruby's statements, include blocks of nested code. These blocks are delimited by keywords or punctuation and, by convention, are indented two spaces relative to the delimiters. 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* (in practice, however, the term "block" is often used for both). 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 are never delimited with curly braces in Ruby—keywords usually serve as the delimiters instead. The specific syntax for statement bodies, method bodies, and class and module bodies are documented in Chapters 5, 6, and 7.

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. These rules are related to the deployment of Ruby programs and are not directly relevant to the language itself.

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.

Second, if a Ruby program contains a "coding" comment that comment must appear on the first line or on the second line if the first line is a shebang.

Third, if a file contains a line that consists of the single token __END__ with no whitespace before or after, then the Ruby interpreter stops processing the file at that point. The remainder of the file may contain arbitrary data that the program can read using the IO stream object DATA. (See Chapter 10 and §9.7 for more about this global constant.)

Ruby programs are not required to fit in a single file. Many programs load additional Ruby code from external libraries, for example. Programs use require to load code from another file. require searches for specified modules of code against a search path, and prevents any given module from being loaded more than once. See §7.6 for details.

The following code illustrates each of these points of Ruby file structure:

```
#!/usr/bin/ruby -w          shebang comment
# -*- coding: utf-8 -*-     coding comment
require 'socket'     load networking library

... program code goes here

__END__ mark end of code
... program data goes here
```

Program Execution

Ruby is a scripting language. This means that Ruby programs are simply lists, or scripts, of statements to be executed. By default, these statements are executed sequentially, in the order they appear. Ruby's control structures (described in Chapter 5) alter this default execution order and allow statements to be executed conditionally or repeat-edly, for example.

Programmers who are used to traditional static compiled languages like C or Java may find this slightly confusing. There is no special main method in Ruby from which exe-cution begins. The Ruby interpreter is given a script of statements to execute, and it begins executing at the first line and continues to the last line.

(Actually, that last statement is not quite true. The Ruby interpreter first scans the file for BEGIN statements, and executes the code in their bodies. Then it goes back to line 1 and starts executing sequentially. See §5.7 for more on BEGIN.)

Another difference between Ruby and compiled languages has to do with module, class, and method definitions. In compiled languages, these are syntactic structures that are processed by the compiler. In Ruby, they are statements like any other. When the Ruby interpreter encounters a class definition, it executes it, causing a new class to come into existence. Similarly, when the Ruby interpreter encounters a method definition, it executes it, causing a new method to be defined. Later in the program, the interpreter will probably encounter and execute a method invocation expression for the method, and this invocation will cause the statements in the method body to be executed.

The Ruby interpreter is invoked from the command line and given a script to execute. Very simple one-line scripts are sometimes written directly on the command line. More commonly, however, the name of the file containing the script is specified. The Ruby interpreter reads the file and

executes the script. It first executes any BEGIN blocks. Then it starts at the first line of the file and continues until one of the following happens:

It executes a statement that causes the Ruby program to terminate.

It reaches the end of the file.

It reads a line that marks the logical end of the file with the token `__END__`.

Before it quits, the Ruby interpreter typically (unless the exit! method was called) executes the bodies of any END statements it has encountered and any other "shutdown hook" code registered with the at_exit function.

DATA TYPES: Numbers

Numbers

Ruby includes five built-in classes for representing numbers, and the standard library includes three more numeric classes that are sometimes useful.

All number objects in Ruby are instances of Numeric. All integers are instances of Integer. If an integer value fits within 31 bits (on most implementations), it is an instance of Fixnum. Otherwise, it is a Bignum. Bignum objects represent integers of arbi-trary size, and if the result of an operation on Fixnum operands is too big to fit in a Fixnum, that result is transparently converted to a Bignum. Similarly, if the result of an operation on Bignum objects falls within the range of Fixnum, then the result is a Fixnum. Real numbers are approximated in Ruby with the Float class, which uses the native floating-point representation of the platform.

The Complex, BigDecimal, and Rational classes are not built-in to Ruby but are distrib-uted with Ruby as part of the standard library. The Complex class represents complex numbers, of course. BigDecimal represents real numbers with arbitrary precision, using a decimal representation rather than a binary representation. And Rational represents rational numbers: one integer divided by another.

All numeric objects are *immutable*; there are no methods that allow you to change the value held by the object. If you pass a reference to a numeric object to a method, you need not worry that the method will modify the object. Fixnum objects are commonly used, and Ruby implementations typically treat them as immediate values rather than as references. Because numbers are immutable, however, there is really no way to tell the difference.

Integer Literals

An integer literal is simply a sequence of digits:

```
0

123

12345678901234567890
```

If the integer values fit within the range of the Fixnum class, the value is a Fixnum. Otherwise, it is a Bignum, which supports integers of any size. Underscores may be inserted into integer literals (though not at the beginning or end), and this feature is sometimes used as a thousands separator:

```
1_000_000_000    # One billion (or 1,000 million in the UK)
```

If an integer literal begins with zero and has more than one digit, then it is interpreted in some base other than base 10. Numbers beginning with 0x or 0X are hexadecimal (base 16) and use the letters a through f (or A through F) as digits for 10 through 15. Numbers beginning 0b or 0B are binary (base 2) and may only include digits 0 and 1. Numbers beginning with 0 and no subsequent letter are octal (base 8) and should consist of digits between 0 and 7. Examples:

```
0377          # Octal representation of 255
```

```
0b1111_11     Binary representation of
11            # 255

              Hexadecimal representation of
0xFF          # 255
```

To represent a negative number, simply begin an integer literal with a minus sign. Literals may also begin with a plus sign, although this never changes the meaning of the literal.

Floating-Point Literals

A floating-point literal is an optional sign followed by one or more decimal digits, a decimal point (the . character), one or more additional digits, and an optional expo-nent. An exponent begins with the letter e or E, and is followed by an optional sign and one or more decimal digits. As with integer literals, underscores may be used within

floating-point literals. Unlike integer literals, it is not possible to express floating-point values in any radix other than base 10. Here are some examples of floating-point literals:

```
0.0
```

-3.14

6.02e23 # This means 6.02×10^{23}

1_000_000.01 # One million and a little bit more

Ruby requires that digits appear before and after the decimal point. You cannot simply write .1, for example; you must explicitly write 0.1. This is necessary to avoid ambiguity in Ruby's complex grammar. Ruby differs from many other languages in this way.

Text

Text is represented in Ruby by objects of the String class. Strings are mutable objects, and the String class defines a powerful set of operators and methods for extracting substrings, inserting and deleting text, searching, replacing, and so on. Ruby provides a number of ways to express string literals in your programs, and some of them support a powerful string interpolation syntax by which the values of arbitrary Ruby expressions can be substituted into string literals. The sections that follow explain string and character literals and string operators.

Textual patterns are represented in Ruby as Regexp objects, and Ruby defines a syntax for including regular expressions literally in your programs. The code /[a-z]\d+/, for example, represents a single lowercase

letter followed by one or more digits. Regular expressions are a commonly used feature of Ruby, but regexps are not a fundamental datatype in the way that numbers, strings, and arrays are.

String Literals

Ruby provides quite a few ways to embed strings literally into your programs.

Single-quoted string literals

The simplest string literals are enclosed in single quotes (the apostrophe character).

The text within the quote marks is the value of the string:

```
'This is a simple Ruby string literal'
```

If you need to place an apostrophe within a single-quoted string literal, precede it with a backslash so that the Ruby interpreter does not think that it terminates the string:

```
'Won\'t you read O\'Reilly\'s book?'
```

The backslash also works to escape another backslash, so that the second backslash is not itself interpreted as an escape character. Here are some situations in which you need to use a double backslash:

```
'This string literal ends with a single backslash: \\'
```

```
'This is a backslash-quote: \\\"
```

```
'Two backslashes: \\\\'
```

In single-quoted strings, a backslash is not special if the character that follows it is anything other than a quote or a backslash. Most of the time, therefore, backslashes need not be doubled (although they can be) in string literals. For example, the following two string literals are equal:

```
'a\b' == 'a\\b'
```

Single-quoted strings may extend over multiple lines, and the resulting string literal includes the newline characters. It is not possible to escape the newlines with a backslash:

```
'This is a long string literal \

that includes a backslash and a newline'
```

If you want to break a long single-quoted string literal across multiple lines without embedding newlines in it, simply break it into multiple adjacent string literals; the Ruby interpreter will concatenate them during the parsing process. Remember, though, that you must escape the newlines (see Chapter 2) between the literals so that Ruby does not interpret the newline as a statement terminator:

```
message =

'These three literals are '\

'concatenated into one by the interpreter. '\

'The resulting string contains no newlines.'
```

Double-quoted string literals

String literals delimited by double quotation marks are much more flexible than single-quoted literals. Double-quoted literals support quite a few backslash escape sequences, such as `\n` for newline, `\t` for tab, and `\"` for a quotation mark that does not terminate the string:

```
"\t\"This quote begins with a tab and ends with
a newline\"\n"


"\\"    # A single backslash
```

Character Literals

Single characters can be included literally in a Ruby program by preceding the character with a question mark. No quotation marks of any kind are used:

```
?A # Character literal for the ASCII character
A


?" # Character literal for the double-quote
character
```

```
# Character literal for the question mark
character
```

Although Ruby has a character literal syntax, it does not have a special class to represent single characters.

String Operators

The `String` class defines several useful operators for manipulating strings of text. The

operator concatenates two strings and returns the result as a new `String` object:

```
planet = "Earth"

"Hello" + " " + planet # Produces "Hello Earth"
```

Java programmers should note that the + operator does not convert its righthand operand to a string; you must do that yourself:

```
"Hello planet #" + planet_number.to_s    # to_s converts to a string
```

Of course, in Ruby, string interpolation is usually simpler than string concatenation with +. With string interpolation, the call to to_s is done automatically:

```
"Hello planet ##{planet_number}"
```

The << operator appends its second operand to its first, and should be familiar to C++ programmers. This operator is very different from +; it alters the lefthand operand rather than creating and returning a new object:

```
greeting = "Hello"

greeting << " " << "World"

puts greeting# Outputs "Hello World"
```

Like +, the << operator does no type conversion on the righthand operand. If the right-hand operand is an integer, however, it is taken to be a character code, and the corresponding character is appended. In Ruby 1.8, only integers between 0 and 255 are allowed. In Ruby 1.9, any integer that represents a valid codepoint in the string's encoding can be used:

```
alphabet =
"A"

alphabet          #
<<          ?B  Alphabet  is now "AB"

alphabe   <         And now it
t         < 67  # is              "ABC"

alphabe   < 25              Rub 1.8: codes must be >=0 and <
t         < 6   # Error in  y     256
```

The * operator expects an integer as its righthand operand. It returns a String that repeats the text specified on the lefthand side the number of times specified by the righthand side:

```
ellipsis = '.'*3    # Evaluates to '...'
```

If the lefthand side is a string literal, any interpolation is performed just once before the repetition is done. This means that the following too-clever code does not do what you might want it to:

```
a = 0;
```

```
"#{a=a+1} " * 3   # Returns "1 1 1 ", not "1 2
3 "
```

String defines all the standard comparison operators. == and != compare strings for equality and inequality. Two strings are equal if—and only if—they have the same length and all characters are equal. <, <=, >, and >= compare the relative order of strings by comparing the character codes of the characters that make up a string. If one string

is a prefix of another, the shorter string is less than the longer string. Comparison is based strictly on character codes. No normalization is done, and natural language col-lation order (if it differs from the numeric sequence of character codes) is ignored.

String comparison is case-sensitive.* Remember that in ASCII, the uppercase letters all have lower codes than the lowercase letters. This means, for example, that "Z" < "a". For case-insensitive comparison of ASCII characters, use the casecmp method (see §9.1) or convert your strings to the same case with downcase or upcase methods before comparing them. (Keep in mind that Ruby's knowledge of upper- and lowercase letters is limited to the ASCII character set.)

Accessing Characters and Substrings

Perhaps the most important operator supported by `String` is the square-bracket array-index operator `[]`, which is used for extracting or altering portions of a string. This operator is quite flexible and can be used with a number of different operand types. It can also be used on the lefthand side of an assignment, as a way of altering string content.

In Ruby 1.8, a string is like an array of bytes or 8-bit character codes. The length of this array is given by the `length` or `size` method, and you get or set elements of the array simply by specifying the character number within square brackets:

```
s = 'hello'; # Ruby 1.8

s[0]     # 104: the ASCII character code for the
first character 'h'

s[s.length-1]     # 111: the character code of
the last character 'o'
```

```
s[-1]    # 111: another way of accessing the
last character


s[-2]    # 108: the second-to-last character


s[-s.length] # 104: another way of accessing
the first character


s[s.length]  # nil: there is no character at
that index
```

Notice that negative array indexes specify a 1-based position from the end of the string. Also notice that Ruby does not throw an exception if you try to access a character beyond the end of the string; it simply returns `nil` instead.

Ruby 1.9 returns single-character strings rather than character codes when you index a single character. Keep in mind that when working with multibyte strings, with char-acters encoded using variable numbers of bytes, random access to characters is less efficient than access to the underlying bytes:

```
s =       #
'hello'; Ruby 1.9

          #    the first character of the
s[0]      'h': string, as a string

s[s.leng  #
th-1]     'o': the last character 'o'

           'o another way of accessing the
s[-1]     #': last character

           'l the second-to-last
s[-2]     #': character
```

In Ruby 1.8, setting the deprecated global variable `$=` to `true` makes the ==, <, and related comparison operators perform case-insensitive comparisons. You should not do this, however; setting this variable produces a warning message, even if the Ruby interpreter is invoked without the `-w` flag. And in Ruby 1.9, `$=` is no longer supported.

```
s[-s.length] # 'h': another way of accessing
the first character


s[s.length]  # nil: there is no character at
that index
```

To alter individual characters of a string, simply use brackets on the lefthand side of an assignment expression. In Ruby 1.8, the righthand side may be an ASCII character code or a string. In Ruby 1.9, the righthand side must be a string. You can use character literals in either version of the language:

```
s[0] =     Replace first
?H         # character with a    capital H

s[-1] =    Replace  character with a
?O         # last     capital         O

 s[s.length] ERROR!   assign      end  the
      = ?! # Can't    beyond the  of   string
```

The righthand side of an assignment statement like this need not be a character code: it may be any string, including a multicharacter string or the empty string. Again, this works in both Ruby 1.8 and Ruby 1.9:

```
s =        # Begin
"hello"    with    a greeting

   =""     #
s[-         Dele th last character; s is now
```

```
1]          te   e  "hell"

s[-  "p!    Chan ne last character and add one; s
1]  ="     #ge   w  is now "help!"
```

More often than not, you want to retrieve substrings from a string rather than individual character codes. To do this, use two comma-separated operands between the square brackets. The first operand specifies an index (which may be negative), and the second specifies a length (which must be nonnegative). The result is the substring that begins at the specified index and continues for the specified number of characters:

```
s =
"hello"

s[0,2]    # "he"

          # "o": returns a string, not the
s[-1,1]   character code ?o

          # "": a zero-length substring
s[0,0]    is always empty

          # "hello": returns all the characters
s[0,10]   that are available

s[s.leng  # "": there is an empty string
```

```
th,1]     immediately beyond the end

s[s.length # nil: it is an error to read
+1,1]      past that

           # nil: negative lengths don't
s[0,-1]    make any sense
```

If you assign a string to a string indexed like this, you replace the specified substring with the new string. If the righthand side is the empty string, this is a deletion, and if the lefthand side has zero-length, this is an insertion:

```
s =
"hello"

s[0,             # Replace first     capital
1]   = "H"       letter with a       letter

s[s.length,0] = " world"  # Append by   end of the
assigning beyond the                    string

s[5,              Insert a comma, without
0]   =","        #deleting anything

s[5,              Delete with no     =
6]   =""         #insertion; s      = "Hellod"
```

Another way to extract, insert, delete, or replace a substring is by indexing a string with a `Range` object. We'll explain ranges in detail in §3.5 later. For our purposes here,

`Range` is two integers separated by dots. When a `Range` is used to index a string, the return value is the substring whose characters fall within the `Range`:

```
s =
"hello"

           # "ll":
           characters 2 and
s[2..3]    3

s[-3..-     "llo":    indexes work,
1]         #negative  too

                            one
            "h": this  inclu character
s[0..0]    #Range     des   index
```

```
s[0...0    # "": this
]          Range is     Empty

            "": this
s[2..1]    #Range is    also empty

s[7..10     nil: this Range is outside
]          #the string bounds

s[-2..-
1] =        Replacement: s becomes
"p!"       #"help!"
```

It is also possible to index a string with a string. When you do this, the return value is the first substring of the target string that matches the index string, or `nil`, if no match is found. This form of string indexing is really only useful on the lefthand side of an assignment statement when you want to replace the matched string with some other string:

```
           #
s =        Star with the word
"hello"    t    "hello"

           #
while(s[" Whil the string contains the
l"])       e    substring "l"
```

```
    s["l"] =   Replace first occurrence
    "L";      #of "l" with "L"

  end         #Now we have "heLLo"
```

Iterating Strings

In Ruby 1.8, the `String` class defines an `each` method that iterates a string line-by-line. The `String` class includes the methods of the `Enumerable` module, and they can be used to process the lines of a string. You can use the `each_byte` iterator in Ruby 1.8 to iterate through the bytes of a string, but there is little advantage to using `each_byte` over the `[]` operator because random access to bytes is as quick as sequential access in 1.8.

The situation is quite different in Ruby 1.9, which removes the `each` method, and in which the `String` class is no longer `Enumerable`. In place of `each`, Ruby 1.9 defines three clearly named string iterators: `each_byte` iterates sequentially through the individual bytes that comprise a string; `each_char` iterates the characters; and `each_line` iterates the lines. If you want to process a string character-by-character, it may be more efficient to use `each_char` than to use the `[]` operator and character indexes:

```
s = "¥1000"

s.each_char {|x| print "#{x} " } # Prints "¥ 1 0 0 0". Ruby
1.9
```

```
0.upto(s.size-1) {|i| print "#{s[i]} "}    # Inefficient with
multibyte chars
```

Arrays

An array is a sequence of values that allows values to be accessed by their position, or *index*, in the sequence. In Ruby, the first value in an array has index `0`. The `size` and `length` methods return the number of elements in an array. The last element of the array is at index `size-1`. Negative index values count from the end of the array, so the last element of an array can also be accessed with an index of `-1`. The second-to-last has an index of `-2`, and so on. If you attempt to read an element beyond the end of an array (with an index >= `size`) or before the beginning of an array (with an index < `-size`), Ruby simply returns `nil` and does not throw an exception.

Ruby's arrays are untyped and mutable. The elements of an array need not all be of the same class, and they can be changed at any time. Furthermore, arrays are dynamically resizeable; you can append elements to them and they grow as needed. If you assign a value to an element beyond the end of the array, the array is automatically extended with `nil` elements. (It is an error, however, to assign a value to an element before the beginning of an array.)

An array literal is a comma-separated list of values, enclosed in square brackets:

```
            #  array that     Fixn
[1, 2, 3] An holds three    um  objects

[-10...0,    A array two ranges;   commas are
0..10,]     #n of    trailing      allowed

[[1,2],[3,4  A array nested
],[5]]      #n of    arrays
```

Ruby includes a special-case syntax for expressing array literals whose elements are short strings without spaces:

```
            a
words =     test  # Same ['this', 'is',
%w[this is  ]     as:    'a', 'test']

open = %w|          Same  ['(', '[',
( [ {      <|     #as:   '{', '<']

white =    \r       Same  ["\s", "\t",
%W(\s \t   \n)     #as:   "\r", "\n"]
```

You can also create arrays with the `Array.new` constructor, and this provides options for programmatically initializing the array elements:

```
empty =         []: returns emp arr
Array.new     # a new       ty  ay

   =
ni Array.new(  [nil,  nil]new array with 3
ls 3)          # nil,      :nil elements
```

```
 zeros =          [0,  0, 0]:  arr with 0
Array.new(4, 0) # 0,   new     ay  4    elements

   =                          an
co Array.new(n   Ma   ne copy existin
py ils)         # ke  a w    of g       array

count =                     #
Array.new(3)      i+   [1,2,3]  elements computed
{|i|              1}         :3 from index
```

To obtain the value of an array element, use a single integer within square brackets:

```
a = [0,   9,    # Array holds the squares
1, 4,     16]   of the indexes

a[0]    # First element is 0

          Last
a[-1]   # element is 16

a[-2]   # Second to last element is 9

a[a.siz   Another way to query the
e-1]    # last element

a[-
a.size   Another way to query the
]       # first element

          Query beyo the end returns
a[8]    # ing   nd   nil

          Query befo the start returns
a[-8]   # ing   re   nil, too
```

Like strings, arrays can also be indexed with two integers that represent a starting index and a number of elements, or a `Range` object. In either case, the expression returns the specified subarray:

```
a = ('a'..'e').to_a    # Range converted to ['a', 'b', 'c',
'd', 'e']


            # []: this subarray has
a[0,0]      zero elements

            # ['b']: a one-element
a[1,1]      array

            # ['d','e']: the last two
a[-2,2]     elements of the array

            # ['a', 'b', 'c']: the first
a[0..2]     three elements

a[-2..-     # ['d','e']: the last two
1]          elements of the array

a[0...-     # ['a', 'b', 'c', 'd']: all but
1]          the last element
```

When used on the lefthand side of an assignment, a subarray can be replaced by the elements of the array on the righthand side. This basic operation works for insertions and deletions as well:

```
a[0,2] = 'B'    a becomes  'B''c', 'd',
['A',    ]     #['A',       ,'e']
```

```
a[2...5]=    'D', a becomes  'B''C', 'D',
['C',      'E'] # ['A',       ,'E']

a[0,0] =        Insert    at the
[1,2,3]        # elements   beginning of a

a[0. =          Delete   eleme
.2]  []        # those    nts

     =
a[-  ['Z'       Replace  eleme
1,1] ]         # last       ntwith another

a[-  =          For single    the array is
1,1] 'Z'       # elements,     optional

a[-  =   # Delete last 2  i 1.8replace with
2,2] nil elements        n   ;nil in 1.9
```

In addition to the square bracket operator for indexing an array, the `Array` class defines a number of other useful operators. Use `+` to concatenate two arrays:

```
a =  2, 3] +    #   2 3  4,
[1,  [4, 5]     [1, , ,  5]

 =   [[6, 7,     [1 2 3  4, [6, 7,
aa +8]]        # , , ,  5,    8]]

 =               Erro righthand    be an
aa +9          #   r: side must    array
```

The – operator subtracts one array from another. It begins by making a copy of its lefthand array, and then removes any elements from that copy if they appear anywhere in the righthand array:

```
['a', 'b', 'c', 'b', 'a'] - ['b', 'c', 'd']     # ['a', 'a']
```

The + operator creates a new array that contains the elements of both its operands. Use

to append elements to the end of an existing array:

```
a =       # Start with an
[]        empty array

a         # a
<< 1      is   [1]

  <   << #  i
a < 2   3 a  s [1, 2, 3]

  < [4,5  #  i [1, 2, 3, [4,
a < ,6]   a  s 5, 6]]
```

Like the `String` class, `Array` also uses the multiplication operator for repetition:

```
a = [0] * 8  # [0, 0, 0, 0, 0, 0, 0, 0]
```

The `Array` class borrows the Boolean operators `|` and `&` and uses them for union and intersection. `|` concatenates its arguments and then removes all duplicate elements from the result. `&` returns an array that holds elements that appear in both of the operand arrays. The returned array does not contain any duplicate elements:

```
a =
[1, 1, 2, 2, 3, 3, 4]

b =
[5, 5, 4, 4, 3, 3, 2]

a |   [1, 2, 3, 4, 5]: duplicates are
b    # removed

b |   [5, 4, 3, 2, 1]: elements are the same,
a    # but order is different

a &
b    # [2, 3, 4]

b &
a    # [4, 3, 2]
```

Note that these operators are not transitive: `a|b` is not the same as `b|a`, for example. If you ignore the ordering of the elements, however, and consider the arrays to be unordered sets, then these operators make more sense. Note also that the algorithm by which union and intersection are performed is not specified, and there are no guarantees about the order of the elements in the returned arrays.

The `Array` class defines quite a few useful methods. The only one we'll discuss here is the `each` iterator, used for looping through the elements of an array:

```
a = ('A'..'Z').to_a    # Begin with an array of letters

a.each {|x| print x }  # Print the alphabet, one letter at a
time
```

Hashes

A *hash* is a data structure that maintains a set of objects known as *keys*, and associates a value with each key. Hashes are also known as *maps* because they map keys to values. They are sometimes called *associative arrays* because they associate values with each of the keys, and can be thought of as arrays in which the array index can be any object instead of an integer. An example makes this clearer:

```
# This hash will map the names of digits to the digits
themselves

numbers = Hash.new     # Create a new, empty, hash object

numbers["one"] = 1     # Map the String "one" to the Fixnum 1

numbers["two"] = 2 # Note that we are using array
notation here numbers["three"] = 3
```

```
sum = numbers["one"] + numbers["two"] # Retrieve values like
this
```

Hash Literals

A hash literal is written as a comma-separated list of key/value pairs, enclosed within curly braces. Keys and values are separated with a two-character "arrow": `=>`. The `Hash` object created earlier could also be created with the following literal:

```
numbers = { "one" => 1, "two" => 2, "three" => 3 }
```

In general, `Symbol` objects work more efficiently as hash keys than strings do:

```
numbers = { :one => 1, :two => 2, :three => 3 }
```

Symbols are immutable interned strings, written as colon-prefixed identifiers; they are explained in more detail in §3.6 later in this chapter.

Ruby 1.8 allows commas in place of arrows, but this deprecated syntax is no longer supported in Ruby 1.9:

```
numbers = { :one, 1, :two, 2, :three, 3 } # Same, but harder
to read
```

Ranges

`Range` object represents the values between a start value and an end value. Range literals are written by placing two or three dots between the start and end value. If two The result is a syntax much like that used by JavaScript objects.

dots are used, then the range is *inclusive* and the end value is part of the range. If three dots are used, then the range is *exclusive* and the end value is not part of the range:

```
1..10  # The integers 1 through 10, including 10

1.0...10.0 # The numbers between 1.0 and 10.0, excluding 10.0 itself
```

Test whether a value is included in a range with the `include?` method (but see below for a discussion of alternatives):

```
cold_war = 1945..1989

cold_war.include? birthdate.year
```

Implicit in the definition of a range is the notion of ordering. If a range is the values between two endpoints, there obviously must be some way to compare values to those endpoints. In Ruby, this is done with the comparison operator `<=>`, which compares its two operands and evaluates to `-1`, `0`, or `1`, depending on their relative order (or equality). Classes such as numbers and strings that have an ordering define the `<=>` operator. A value can only be used as a range endpoint if it responds to this operator. The endpoints of a range and the values “in” the range are typically all of the same class. Technically, however,

any value that is compatible with the `<=>` operators of the range endpoints can be considered a member of the range.

The primary purpose for ranges is comparison: to be able to determine whether a value is in or out of the range. An important secondary purpose is iteration: if the class of the endpoints of a range defines a `succ` method (for successor), then there is a discrete set of range members, and they can be iterated with `each`, `step`, and `Enumerable` methods. Consider the range `'a'..'c'`, for example:

```
r = 'a'..'c'

r.each {|l| print
"[#{l}]"}          # Prints "[a][b][c]"

r.step(2) { |l| print
"[#{l}]"}          #Prints "[a][c]"

                    => ['a','b','c']:
r.to_a             #Enumerable defines to_a
```

The reason this works is that the `String` class defines a `succ` method and `'a'.succ` is `'b'` and `'b'.succ` is `'c'`. Ranges that can be iterated like this are *discrete* ranges. Ranges whose endpoints do not define a `succ` method cannot be iterated, and so they can be called *continuous*. Note that ranges with integer endpoints are discrete, but floating-point numbers as endpoints are continuous.

Ranges with integer endpoints are the most commonly used in typical Ruby programs. Because they are discrete, integer ranges can be used to index

strings and arrays. They are also a convenient way to represent an enumerable collection of ascending values.

Notice that the code assigns a range literal to a variable, and then invokes methods on the range through the variable. If you want to invoke a method directly on a range literal, you must parenthesize the literal, or the method invocation is actually on the endpoint of the range rather than on the `Range` object itself:

```
1..3.to_a  # Tries to call to_a on the number 3

(1..3).to_a     # => [1,2,3]
```

Symbols

A typical implementation of a Ruby interpreter maintains a symbol table in which it stores the names of all the classes, methods, and variables it knows about. This allows such an interpreter to avoid most string comparisons: it refers to method names (for example) by their position in this symbol table. This turns a relatively expensive string operation into a relatively cheap integer operation.

These symbols are not purely internal to the interpreter; they can also be used by Ruby programs. A `Symbol` object refers to a symbol. A symbol literal is written by prefixing an identifier or string with a colon:

```
                # A
:symbol         Symbol literal
```

```
                #   sa
:"symbol"       The me literal

:'another long    Quotes useful for symbols
symbol'         #are    with spaces

s = "string"

                 Th Symb
sym = :"#{s}"   #e  ol   :string
```

Symbols also have a `%s` literal syntax that allows arbitrary delimiters in the same way that `%q` and `%Q` can be used for string literals:

```
%s["] # Same as :'"'
```

Symbols are often used to refer to method names in reflective code. For example, suppose we want to know if some object has an `each` method:

```
o.respond_to? :each
```

Here's another example. It tests whether a given object responds to a specified method, and, if so, invokes that method:

```
name = :size

if o.respond_to? name
```

```
    o.send(name)


  end
```

You can convert a `String` to a `Symbol` using the `intern` or `to_sym` methods. And you can convert a `Symbol` back into a `String` with the `to_s` method or its alias `id2name`:

```
str =       # Begin   Strin
"string"    with    a  g

            #
sym =       Convert   Symbo
str.intern  to      a  l

            #
sym =       Another   t   the same
str.to_sym  way       o do thing

str sym.to    Convert to
=   _s      #    back a  String

str sym.id2   Another t
=   name    #way     o do It
```

Two strings may hold the same content and yet be completely distinct objects. This is never the case with symbols. Two strings with the same content will both convert to exactly the same `Symbol` object. Two distinct `Symbol` objects will always have different content.

Whenever you write code that uses strings not for their textual content but as a kind of unique identifier, consider using symbols instead. Rather than writing a method that expects an argument to be either the string “AM” or “PM”, for example, you could

True, False, and Nil

We saw in §2.1.5 that `true`, `false`, and `nil` are keywords in Ruby. `true` and `false` are the two Boolean values, and they represent truth and falsehood, yes and no, on and off. `nil` is a special value reserved to indicate the absence of value.

Each of these keywords evaluates to a special object. `true` evaluates to an object that is a singleton instance of `TrueClass`. Likewise, `false` and `nil` are singleton instances of `FalseClass` and `NilClass`. Note that there is no `Boolean` class in Ruby. `TrueClass` and `FalseClass` both have `Object` as their superclass.

If you want to check whether a value is `nil`, you can simply compare it to `nil`, or use the method `nil?`:

```
o == nil     # Is o nil?

o.nil?  # Another way to test
```

Note that `true`, `false`, and `nil` refer to objects, not numbers. `false` and `nil` are not the same thing as `0`, and `true` is not the same thing as `1`. When Ruby requires a Boolean value, `nil` behaves like `false`, and any value other than `nil` or `false` behaves like `true`.

RDoc and ri

RDoc is a documentation system. If you put comments in your program files (Ruby or C) in the prescribed **RDoc**format, **rdoc** scans your files, extracts the

comments, organizes them intelligently (indexed according to what they comment on), and creates nicely formatted documentation from them. You can see **RDoc** markup in many of the C files in the Ruby source tree and many of the Ruby files in the Ruby installation.

The Ruby **ri** tool is used to view the Ruby documentation off-line. Open a command window and invoke **ri** followed by the name of a Ruby class, module or method. **ri** will display documentation for you. You may specify a method name without a qualifying class or module name, but this will just show you a list of all methods by that name (unless the method is unique). Normally, you can separate a class or module name from a method name with a period. If a class defines a class method and an instance method by the same name, you must instead use :: to refer to a class method or # to refer to the instance method. Here are some example invocations of ri

1. :ri Array
2. ri Array.sort
3. ri Hash#each
4. ri Math::sqrt

ri dovetails with **RDoc**: It gives you a way to view the information that **RDoc** has extracted and organized. Specifically (although not exclusively, if you customize it), **ri** is configured to display the **RDoc** information from the Ruby source files. Thus on any system that has Ruby fully installed, you can get detailed information about Ruby with a simple command-line invocation of **ri**.

POSSIBLE QUESTIONS

UNIT 1

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. List out the major characteristics of the database approach.

2. Discuss on the different categories of data model.

3. Write short notes on schemas, instances and database state.

4. Elaborate the three schema architecture of DBMS in detail.

5. Write a detailed note on the two types of data independence.

6. Discuss on DBMS languages.

7. What is meant by DBMS interfaces? Explain them in detail.

8. Illustrate the main phases of database design.

9. Describe the different types of attributes in the ER model.

10. Write short notes on entity types and entity sets.

PART – C (10 MARKS)

1. Design an application form using tk classes and validate all fields on Rails framework
2. Explain about different Data Types in Ruby.
3. Explain in detail about Object creation and initialization in Ruby
4. Write a ruby program to create a main thread and execute multiple process through the main thread.
5. Write a note on Defining, Calling and Undefining methods in Ruby

UNIT 2

Sno	QESTIONS	OPTION1	OPTION2	OPTION3	OPTIONS4	ANS
1	If a class defines a class method _____ must be used to refer the class method.	::	;;	c||	?/	::
2	If a class defines an instance method _____must be used to refer the instance method.	%	&&	#	<>	#
3	Ruby's package management system is known as _____	ri	RDoc	RubyGems	Ruby Interactives	RubyGems
4	The _____command installs the most recent version of the gem we request.	gem install	package install	gems install	gem cmd	gem install
5	_____ is the command which is used to Display RubyGems configuration information.	gem install	package install	gem environment	gem cmd	gem environment
6	The Ruby interpreter parses a program as a sequence of_____.	Statements	commands	keyword	tokens	tokens
7	The Ruby interpreter ignores the _____ character and any text that follows it.	@	*	//	#	#
8	Any text that appears after _____is part of the comment and is also ignored.	=begin or =end	begin or end	=start or =end	=begin or end=	=begin or =end
9	_____are values that appear directly in Ruby source code.	String	Command	Literals	Modules	Literals
10	Identifiers that begin with a _____which are constants.	capital letter A–Z	small letter a–z	Numerical	Alpha Numeric	capital letter A–Z
11	_____names must begin with initial capital letters.	Strings	Class and module	Literals	Expressions	Class and module
12	_____that are not constants and has been written with underscores.	identifiers	Keyword	multiword identifiers	tokens	multiword identifiers
13	Two identifiers are the same only if they are represented by the same _____.	sequence of bytes	sequence of bits	sequence of literals	sequence of tokens.	sequence of bytes
14	_____is not performed in Ruby.	byte code normalization	Unicode normalization	normalization	boyce code normalization	Unicode normalization
15	_____may appears at the start and end of Ruby identifiers.	String characters	special characters	Punctuation characters	Numeric	Punctuation characters
16	_____ are the keyword-like tokens that are treated specially by the Ruby parser when they appear at the beginning of a line.	=begin =end __END__	begin exit do	not do match	iterate end not	=begin =end __END__
17	Features of the Ruby language are actually implemented as methods of the _____classes.	Kernel, Module	Kernel, Module, Class	Kernel,Object	Kernel, Module, Class, and Object	Kernel, Module, Class, and Object
18	Without_____, the Ruby interpreter must figure out on its own where statements end.	explicit semicolons	colon	implicit semicolons	Expression	explicit semicolons
19	If the Ruby code on a line is a syntactically complete statement, Ruby uses the _____as the statement terminator.	semicolon	Exit	newline	tab	newline
20	28. _____statements are optionally be followed by an expression that provides a return value.	return and break	return and stop	block and return	block and stop	return and break
21	If the first non-space character on a line is a_____, then the line is considered a continuation line.	dash	white space	period	dollar sign	period
22	Ruby's grammar allows the _____around method invocations to be omitted in certain circumstances.	()	[]	{}	<>	()
23	_____ are packaged bits of Ruby code that you can install to extend or add functionality.	modules	Gems	classes	methods	Gems
24	_____are used to perform computations on values, and compound expressions are built by combining simpler sub-expressions with operators.	Keywords	Expressions	Regular expressions	Operators	Operators
25	33. _____are delimited by keywords or punctuation.	class	method	module	blocks	blocks
26	Explain this ruby idiom: a ||= b	a = 1 b = 2 a ||= b #=> a = 1	a = 0 b = 0 a ||= b #=> a = 1	a = 1 b = 1 a ||= b #=> a = 1	a = 2 b = 2 a ||= b #=> a = 1	a = 1 b = 2 a ||= b #=> a = 1
27	____ always refers to the current object in Ruby.	this	self	object self	this to	self
28	If the integer values fit within the range of the _____class, the value is a Fixnum.	Fixnum	bignum	ranges	numeric	Fixnum
29	Numbers beginning with _____are hexadecimal.	0x or 0X	gff	oct	base 10	0x or 0X
30	Numbers beginning _____are binary.	0b or 0B	hexa decimal	octet	base 19	0b or 0B

31	6.02e23 # This means _____	6.02 + 1023	6.02 x 1023	6.02 / 1023	6.02 % 1023	6.02 x 1023
32	Float objects cannot represent numbers larger than_____.	Float::MAX	Float::MIN	10,000	Float#MAX	Float::MAX
33	Text is represented in Ruby by objects of the _____ class.	Character	Boolean	String	ASCII	String
34	Single-quoted strings may extend over _____, and the resulting string literal includes the _____ characters.	multiple lines, period	single line, newline	multiple lines, tab	multiple lines, newline	multiple lines, newline
35	$salutation = 'hello' # Define a _____	global variable	instance variable	local variable	class variable	global variable
36	Here documents begin with _____.	<< or <<	<< or >>>	<< or <<-.	<< or >>-.	<< or <<-.
37	s = 'hello';s[0] s[s.length-1]s[-1] output:_____	hll	hlo	hoo	how	hoo
38	_____objects define the normal === operator for testing equality.	Regexp and Range	Reg and Range	Regexp and Operator	exp and Range	Regexp and Range
39	Ruby's case statement matches its expression against each of the possible cases using ===, so this operator is often called the _____.	equality operator	comparison operator	case equality operator	case inequality operator	case equality operator
40	A _____is a collection of related methods that operate on the state of an object.	class	blocks	module	package	class
41	The _____allows you to alter the characters of a string or to insert, delete, and replace substrings.	=== operator	[][] operator	[]= operator	%= operators	[]= operator
42	The _____allows you to append to a string.	<< operator	>> operator	++ operator	&& operator	<< operator
43	Double-quoted strings can include arbitrary Ruby expressions delimited by _____.	#(and)	+{and }	#{ and }	#[and]	#{ and }
44	The = operator in Ruby assigns a value to a variable and it is called_____ operator.	overridable	equal	assignment	nonoverridable	nonoverridable
45	Ruby supports_____, allowing more than one value and more than one variable in assignment expressions.	parallel assignment	distributed assignment	unique assignment	bypassed assignment	parallel assignment
46	YARV stands for _____.	"Yet Another Regular Virtual machine"	"Yet Another Ruby Virtual machine"	"Yet Another Ruby Virtual mechanism"	"Yet Another Request Virtual machine"	"Yet Another Ruby Virtual machine"
47	MRI stands for_____.	"Matz's Ruby Implementation."	"Memory Ruby Implementation."	"Matz's Ruby Interface."	"Matz's Request Implementation."	"Matz's Ruby Implementation."
48	Method names may end with an _____to indicate that they should be used cautiously.	$$	##	%%	!	
49	All number objects in Ruby are instances of _____.	Numeric	Real	Integer	String	Numeric
50	A _____is an optional sign followed by one or more decimal digits, a decimal point, one or more additional digits, and an optional exponent.	Integer-point literal	floating-point literal	decimal-point literal	character literal	floating-point literal
51	When text is enclosed in_____, that text is treated as a double-quoted string literal.	backquotes	parenthesis	braces	brackets	backquotes
52	The String class defines an _____ that iterates a string line-by-line.	for method	such method	each method	while method	each method
53	The String class includes the methods of the_____, and they can be used to process the lines of a string.	Extending module	Enlarge module	Enrich module	Enumerable module	Enumerable module
54	_____encoding of Unicode characters use variable numbers of bytes for each character.	UTF-8	USF-8	UTFF-8	UUTF-8	UTF-8
55	The _____methods return the number of characters in a string.	count and size	long and size	length and esum	length and size	length and size
56	You can explicitly set the encoding of a string with _____.	force_encoding	force_dncoding	free_encoding	force_encoding	force_encoding
57	a = [1, 1, 2, 2, 3, 3, 4] b = [5, 5, 4, 4, 3, 3, 2] a | b output: _____	[1, 2, 3, 4, 5]	[1, 2, 2, 3, 4, 5]	[1, 2, 3, 4, 5, 5]	[1, 1, 2, 2, 3, 3, 4, 4, 5, 5]	[1, 2, 3, 4, 5]
58	a = [1, 1, 2, 2, 3, 3, 4] b = [5, 5, 4, 4, 3, 3, 2] a & b _____? b & a _____?	[2, 3, 3][4, 2, 2]	[2, 5, 4][4, 3, 1]	[2, 3, 4][4, 3, 2]	[2, 3, 6][4, 3, 6]	[2, 3, 4][4, 3, 2]
59	A _____is a data structure that maintains a set of objects known as keys, and associates a value with each key.	hash	array	dictionary	mixin	hash

METHODS: Defining a Method, Calling a Method; Undefining methods – Methods with Exception – Operator methods and names – Method Arguments – Method objects - Defining Attribute Accessor Methods - Dynamically Creating Methods. EXCEPTIONS AND EXCEPTION HANDLING: Hierarchy – Exception classes and objects – Raising Exception with raise – Handling Exception with rescue – Exception propagation – Else clause and ensure class.

Methods

Methods are defined with the def keyword. This is followed by the method name and an optional list of parameter names in parentheses. The Ruby code that constitutes the method body follows the parameter list, and the end of the method is marked with the end keyword. Parameter names can be used as variables within the method body, and the values of these named parameters come from the arguments to a method invocation. Here is an example method:

```
Define a method named 'factorial' with a single parameter 'n' def
factorial(n)
    if n < 1                # Test the argument value for validity
      raise "argument must
                          be > 0"
    elsif n == 1            # If the argument is 1
      1                     # then the value of the method invocation is 1
    else                    # Otherwise, the factorial of n is n times
      n * factorial(n-1)    # the factorial of n-1
    end
  end
```

This code defines a method named factorial. The method has a single parameter named n. The identifier n is used as a variable within the body of the method. This is a recursive method, so the body of the method includes an invocation of the method. The invocation is simply the name of the method followed by the argument value in parentheses.

Method Return Value

Methods may terminate normally or abnormally. Abnormal termination occurs when the method raises an exception. The factorial method shown earlier terminates abnormally if we pass it an argument less than 1. If a method terminates normally, then the value of the method invocation expression is the value of the last expression evaluated within the method

body. In the factorial method, that last expression will either be 1 or n*factorial(n-1).

The return keyword is used to force a return prior to the end of the method. If an expression follows the return keyword, then the value of that expression is returned. If no expression follows, then the return value is nil. In the following variant of the factorial method, the return keyword is required:

```
def factorial(n)
  raise "bad argument" if n < 1
  return 1 if n == 1
  n * factorial(n-1)
end
```

We could also use return on the last line of this method body to emphasize that this expression is the method's return value. In common practice, however, return is omitted where it is not required.

Ruby methods may return more than one value. To do this, use an explicit return statement, and separate the values to be returned with commas:

```
Convert the Cartesian point (x,y) to polar (magnitude, angle)
coordinates def polar(x,y)
  return Math.hypot(y,x),
Math.atan2(y,x) end
```

When there is more than one return value, the values are collected into an array, and the array becomes the single return value of the method. Instead of using the return statement with multiple values, we can simply create an array of values ourselves:

```
  Convert polar coordinates to Cartesian
coordinates def cartesian(magnitude,
angle)
  [magnitude*Math.cos(angle),
magnitude*Math.sin(angle)] end
```

Methods and Exception Handling

def statement that defines a method may include exception-handling code in the form of rescue, else, and ensure clauses, just as a begin statement can. These exception-handling clauses go after the end of the method body but before the end of the def statement. In short methods, it can be particularly tidy to associate your rescue clauses with the def statement. This also means you don't have to use a begin statement and the extra level of indentation that comes with it.

Invoking a Method on an Object

Methods are always invoked on an object. (This object is sometimes called the receiver in a reference to an object-oriented paradigm in which methods are called "messages" and are "sent to" receiver objects.) Within the body of a method, the key-word self refers to the object on which the method was invoked. If we don't specify an object when invoking a method, then the method is implicitly invoked on self.

Notice, however, that you've already seen examples of invoking methods on objects, in code like this:

```
first = text.index(pattern)
```

Like most object-oriented languages, Ruby uses to separate the object from the method to be invoked on it. This code passes the value of the variable pattern to the method named index of the object stored in the variable text, and stores the return value in the variable first.

Method Names

By convention, method names begin with a lowercase letter. (Method names can begin with a capital letter, but that makes them look like constants.) When a method name is longer than one word, the usual convention is to separate the words with underscores like this rather than using mixed case likeThis.

Method names may (but are not required to) end with an equals sign, a question mark, or an exclamation point. An equals sign suffix signifies that the method is a *setter* that can be invoked using assignment syntax.'

The first convention is that any method whose name ends with a question mark returns a value that answers the question posed by the method invocation. The empty method of an array, for example, returns true if the array has no elements. Methods like these are called *predicates* and. Predicates typically return one of the Boolean values true or false, but this is not required, as any value other than false or nil works like true when a Boolean value is required. (The Numeric method nonzero?, for example, returns nil if the number it is invoked on is zero, and just returns the number otherwise.)

The second convention is that any method whose name ends with an exclamation mark should be used with caution. The Array object, for example, has a sort method that makes a copy of the array, and then sorts that copy. It also has a sort! method that sorts the array in place. The

exclamation mark indicates that you need to be more careful when using that version of the method.

Often, methods that end with an exclamation mark are *mutators*, which alter the in-ternal state of an object. But this is not always the case; there are many mutators that do not end with an exclamation mark, and a number of nonmutators that do. Mutating methods (such as Array.fill) that do not have a nonmutating variant do not typically have an exclamation point.

Consider the global function exit: it makes the Ruby program stop running in a con-trolled way. There is also a variant named exit! that aborts the program immediately without running any END blocks or shutdown hooks registered with at_exit. exit! isn't a mutator; it's the "dangerous" variant of the exit method and is flagged with to remind a programmer using it to be careful.

Operator Methods

Many of Ruby's operators, such as +, *, and even the array index operator [], are implemented with methods that you can define in your own classes. You define an operator by defining a method with the same "name" as the operator. (The only exceptions are the unary plus and minus operators, which use method names +@ and -@.) Ruby allows you to do this even though the method name is all punctuation. You might end up with a method definition like this:

```
def +(other)              # Define binary plus operator: x+y is x.+(y)
  self.concatenate(other)
end
```

Methods that define a unary operator are passed no arguments. Methods that define binary operators are passed one argument and should operate on self and the argument. The array access operators [] and []= are special because they can be invoked with any number of arguments. For []=, the last argument is always the value being assigned.

Mapping Arguments to Parameters

When a method definition includes parameters with default values or a parameter pre-fixed with an *, the assignment of argument values to parameters during method invocation gets a little bit tricky.

In Ruby 1.8, the position of the special parameters is restricted so that argument values are assigned to parameters from left to right. The first arguments are assigned to the ordinary parameters. If there are any

remaining arguments, they are assigned to the parameters that have defaults. And if there are still more arguments, they are assigned to the array argument.

Ruby 1.9 has to be more clever about the way it maps arguments to parameters because the order of the parameters is no longer constrained. Suppose we have a method that is declared with o ordinary parameters, d parameters with default values, and one array parameter prefixed with *. Now assume that we invoke this method with a arguments.

If a is less than o, an ArgumentError is raised; we have not supplied the minimum required number of arguments.

If a is greater than or equal to o and less than or equal to o+d, then the leftmost a–o parameters with defaults will have arguments assigned to them. The remaining (to the right) o+d–a parameters with defaults will not have arguments assigned to them, and will just use their default values.

If a is greater than o+d, then the array parameter whose name is prefixed with an * will have a–o–d arguments stored in it; otherwise, it will be empty.

Once these calculations are performed, the arguments are mapped to parameters from left to right, assigning the appropriate number of arguments to each parameter.

Method Objects

Ruby's methods and blocks are executable language constructs, but they are not objects. Procs and lambdas are object versions of blocks; they can be executed and also manipulated as data. Ruby has powerful meta programming (or *reflection*) capabilities, and methods can actually be represented as instances of the Method class. You should note that invoking a method through a Method object is less efficient than invoking it directly. Method objects are not typically used as often as lambdas and procs.

The Object class defines a method named method. Pass it a method name, as a string or a symbol, and it returns a Method object representing the named method of the receiver (or throws a NameError if there is no such method). For example:

```
m = 0.method(:succ)       # A Method representing the succ method of
Fixnum 0
```

The Method class is not a subclass of Proc, but it behaves much like it. Method objects are invoked with the call method (or the [] operator), just as Proc objects are. And Method defines an arity method just like the arity method of Proc. To invoke the Method m:

```
puts m.call  # Same as puts 0.succ. Or use puts m[].
```

Invoking a method through a Method object does not change the invocation semantics, nor does it alter the meaning of control-flow statements such as return and break. The

call method of a Method object uses method-invocation semantics, not yield semantics.
Method objects, therefore, behave more like lambdas than like procs.

Method objects work very much like Proc objects and can usually be used in place of them. When a true Proc is required, you can use Method.to_proc to convert a Method to a Proc. This is why Method objects can be prefixed with an ampersand and passed to a method in place of a block. For example:

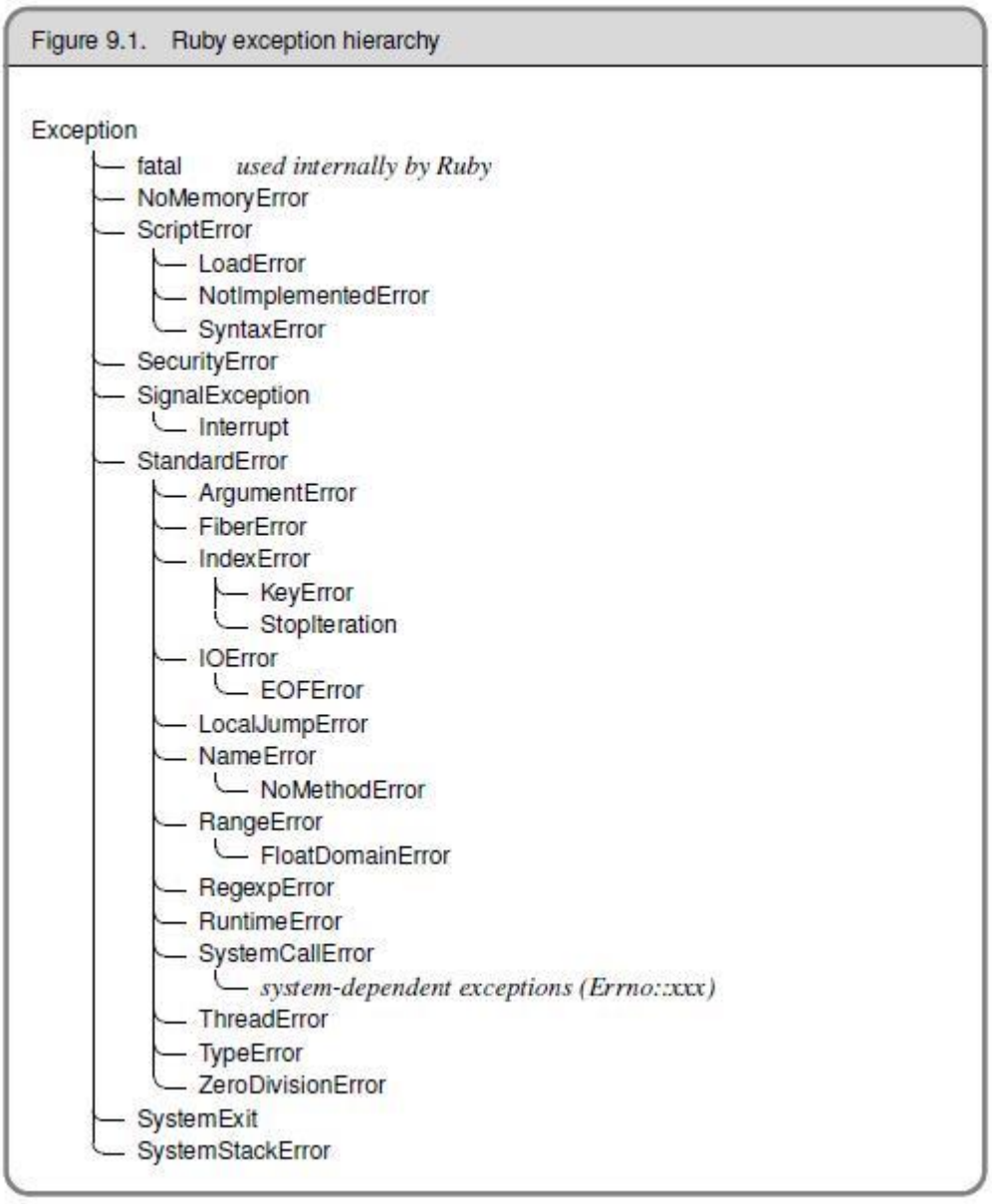
```
def square(x); x*x; end
puts (1..10).map(&method(:square))
```

One important difference between Method objects and Proc objects is that Method objects are not closures. Ruby's methods are intended to be completely self-contained, and they never have access to local variables outside of their own scope. The only binding retained by a Method object, therefore, is the value of self—the object on which the method is to be invoked.

Raising An Exception

An *exception* is a special kind of object, an instance of the class **Exception** or a descendant of that class that represents some kind of exceptional condition; it indicates that something has gone wrong. When this occurs, an exception is raised (or thrown). By default, Ruby programs terminate when an exception occurs. But it is possible to declare exception handlers. An exception handler is a block of code that is executed if an exception occurs during the execution of some other block of code. *Raising* an exception means stopping normal execution of the program and transferring the flow-of-control to the exception handling code where you either deal with the problem that's been encountered or exit the program completely. Which of these happens - dealing with it or aborting the program - depends on whether you have provided a **rescue** clause (**rescue** is a fundamental part of the Ruby language). If you haven't provided such a clause, the program terminates; if you have, control flows to the **rescue** clause.
Ruby has some predefined classes - **Exception** and its children - that help you to handle errors that can occur in your program. The following figure shows the Ruby exception hierarchy.

The chart above shows that most of the subclasses extend a class known as **StandardError**. These are the "normal" exceptions that typical Ruby programs try to handle. The other exceptions represent lower-level, more serious, or less recoverable conditions, and normal Ruby programs do not typically attempt to handle them.
The **raise** method is from the **Kernel** module. By default, **raise** creates an exception of the **RuntimeError** class. To raise an exception of a specific class, you can pass in the class name as an argument to **raise**.



Class Exception

Ruby's standard classes and modules raise exceptions. All the exception classes form a hierarchy, with the class Exception at the top. The next level contains seven different types −

- Interrupt
- NoMemoryError
- SignalException
- ScriptError
- StandardError

- SystemExit

There is one other exception at this level, **Fatal**, but the Ruby interpreter only uses this internally.

Both ScriptError and StandardError have a number of subclasses, but we do not need to go into the details here. The important thing is that if we create our own exception classes, they need to be subclasses of either class Exception or one of its descendants.

Let's look at an example −

```
class FileSaveError < StandardError
  attr_reader :reason
  def initialize(reason)
    @reason = reason
  end
end
```

Exception handling in Ruby with begin, rescue, and ensure

Begin, rescue, and ensure provide flexible exception handling. Supposed we have the following method:

```
1   def divide(a, b)
2     begin
3       a / b
4     rescue TypeError => e
5       puts "I am rescuing from a TypeError"
6       puts e
7       puts e.class
8       puts e.backtrace
9     rescue ZeroDivisionError => e
10      puts "I am rescuing from a ZeroDivisionError"
11      puts e
12      puts e.class
13      puts e.backtrace
14    else
15      puts "No exception was raised"
16    ensure
17      puts "BLAH BLAH BLAH"
18    end
19  end
```

Let’s examine what this method outputs with various inputs.

```
1  >> divide(4, 2)
2  No exception was raised
3  BLAH BLAH BLAH
```

If the function is passed valid output, the code in else is executed and so is the code in ensure. The code in ensure is executed regardless, but else is only executed if there are no exceptions.

```
1  >> divide(1, "cat")
2  I am rescuing from a TypeError
3  String can't be coerced into Fixnum
4  TypeError
5  lib/begin_rescue.rb:3:in `/'
6  lib/begin_rescue.rb:3:in `divide'
7  lib/begin_rescue.rb:22:in `<main>'
8  BLAH BLAH BLAH
```

Where there is a TypeError, the code in the TypeError rescue block and the ensure block are executed. By passing => e to the rescue TypeError => e line, we have access to the error message, error class, and error backtrace – useful stuff for debugging purposes.

```
1  >> divide(1, 0)
2  I am rescuing from a ZeroDivisionError
3  divided by 0
4  ZeroDivisionError
5  lib/begin_rescue.rb:3:in `/'
6  lib/begin_rescue.rb:3:in `divide'
7  lib/begin_rescue.rb:23:in `<main>'
8  BLAH BLAH BLAH
```

Curiously, the following code does not execute the code in the ensure block (didn't we previously establish that the code in the ensure block is always executed under all conditions).

```
1  lib/begin_rescue.rb:1:in `divide': wrong number of arguments (0 for 2)
2  (ArgumentError)
       from lib/begin_rescue.rb:21:in `<main>'
```

In this case the ArgumentError is raised before the function is executed, so the ensure never gets the chance to run. We can rescue from an ArgumentError (rescuing from ArgumentErrors is not a good idea, BTW), by putting the method call in another begin/rescue block:

```
1  Begin
2    divide
3  rescue ArgumentError
4    puts "Ahhhhh, that's better"
5  end
```

POSSIBLE QUESTIONS

UNIT 1

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. Discuss in detail about Exceptions and Exception handling in Ruby
2. RR Group of Companies is off on Fridays at any cost. If an employee wants to login on Friday, the system has to report an error and display "You can't login on Friday." Create a Ruby Class to handle the exception.
3. Describe about exception handling with rescue in detail.
4. Write a ruby script for following problems.

   i) Choose random numbers and display the behavior the number. (7)
   ii) Design a grade sheet using case statement.(7)
5. Write a method called age that calls a private method to calculate the age of the vehicle. Make sure the private method is not available from outside of the class. Use Ruby's build in time class.
6. Describe about raising an exception with raise in detail.
7. Explain about the Dynamically Creating Methods in Ruby
8. Explain about defining, calling and undefining a method with suitable examples.

PART – C (10 MARKS)

1. Design an application form using tk classes and validate all fields on Rails framework
2. Explain about different Data Types in Ruby.
3. Explain in detail about Object creation and initialization in Ruby
4. Write a ruby program to create a main thread and execute multiple process through the main thread.
5. Write a note on Defining, Calling and Undefining methods in Ruby

Sno.	QUESTIONS	OPTION1	OPTION2	OPTION3	OPTION4	ANS
1	_____mode used to Open file for writing, but append to the end of the file if it already exists.	"a"	”f”	”w”	”R”	"a"
2	_____mode used for Open for writing. Create a new file or truncate an existing one.	"a"		”w”	”f”	”w”
3	_____mode is used to Open for reading and writing. Start at beginning of file. Fail if file does not exist.	"a"	”r+”	”w”	”f”	”r+”
4	_____stream has special behavior intended to make it simple to write scripts that read the files specified on the command line or from standard input.	ARGF, or $<	ARGFILE, or $<>	ARGF, or $<>>	ARGF and $$<	ARGF, or $<
5	Specify the encoding of any IO object with the_____method.	set_decoding	set_encoding	get_encoding	not_encoding	set_encoding
6	File.open("data", "r:binary") # Open a file for_____.	deleting binary data	writing binary data	reading binary data	renaming binary data	reading binary data
7	_____disables the automatic newline conversion performed by Windows, and is only necessary on that platform.	binary	decimal	readmode	binmode	binmode
8	The_____and_____methods read a single byte and return it as a Fixnum.	getc readchar	getchar readchar	get readch	string readchar	getc readchar
9	_____Reads exactly n bytes and return them as a string.	readbytes ;	read (bytes(n)) ;	bytes ;	readbytes (n) ;	readbytes (n) ;
10	_____Read the bytes (up to a maximum of n) that are currently available for reading, and return them as a string, using the buffer string if it is specified.	read_nonblock(n, buffer=nil)	read_block(n, buffer=nil)	read_nonblock(n, buffer)	read_nonblock(nil)	read_nonblock(n, buffer=nil)
11	The_____method converts its arguments to strings, and outputs them to the stream.	printf	filef	print	pts	print
12	_____output method converts each of its arguments to a string, and writes each one to the stream.	puts	printf	pts	file	puts

13	The_____method expects a format string as its first argument, and interpolates the values of any additional arguments into that format string using the_____operator.	Printf String %	Print String ^	Puts String %	Printf Str##	Printf String %
14	_____is a low-level, unbuffered, nontranscoding version of write.	systemwrite	systemgc	syswrite	sysprint	syswrite
15	f = File.open("test.txt") f.seek(10, IO::SEEK_SET) output # Skip to_____.	absolute position 10	absolute position 100	binary 10	rollback position 10	absolute position 10
16	f = File.open("test.txt") f.seek(-10, IO::SEEK_EN output #_____.	Skip to 10 bytes from end	Skip to 10 bytes from beginning	removes 10 bytes from end	updates to 10 bytes from end	Skip to 10 bytes from end
17	f = File.open("test.txt") pos = f.sysseek(0, IO::SEEK_CUR) output #_____	Get previous position	Get current position	Set current position	Get next position	Get current position
18	f = File.open("test.txt") f.sysseek(0, IO::SEEK_SET) output:#_____	Rewind stream	forward stream	fast forward stream	stop stream	Rewind stream
19	f = File.open("test.txt") f.sysseek(pos, IO::SEEK_SET) output:#_____	Return to original position	move to next position	Return to last position	Return to previous position	Return to original position
20	When you are done reading from or writing to a stream, you must close it with the _____method.	out	finish	close	done	close
21	To write Internet client applications, use_____the class.	TCPSocket	client	TCPclient	TCPserver	TCPSocket
22	Obtain a TCPSocket instance with the_____class method.	TCPSocket .read	TCPSocket .open	TCP.open	TCPSocket .start	TCPSocket .open
23	A TCPServer object is a factory for_____objects.	TCPSocket	client	TCPclient	TCPserver	TCPSocket

24	Call_____to specify a port for your service and create a TCPServer object.	TCPSocket.read	TCPSocket.open	TCP.open	TCPServer.open	TCPServer.open
25	The_____module defines a handful of low-level methods that can be occasionally useful for debugging or metaprogramming.	ObjectSpace	class	root	object	ObjectSpace
26	_____is an iterator that can yield every object.	each_class	each_operator	each_object	for_object	each_object
27	_____is the inverse of Object.object_id.	Object_id2ref	ObjectSpace._id2ref	ObjectSpace._id	ObjectSpaceref	ObjectSpace._id2ref
28	Object.object_id, it takes an object ID as its argument and raises a_____if there is no object with that ID.	RangeError	stderror	typeerror	logical error	RangeError
29	_____allows a block of code to be invoked when a specified object is garbage collected.	define_finalizer	ObjectSpace.define_final	ObjectSpace.finalizer	ObjectSpace.define_finalizer	ObjectSpace.define_finalizer
30	_____to delete all finalizer blocks registered for an object.	define_finalizer	ObjectSpace.define_final	ObjectSpace.undefine_finalizer	ObjectSpace.define_finalizer	ObjectSpace.undefine_finalizer
31	Garbage collection functionality is also available through the _____module.	GC	AC	SC	SYSGC	GC
32	_____method, which forces Ruby's garbage collector to run.	GC.start	GC_finalizer	ObjectSpace.GC_finalizer	ObjectSpace.define_finalizer	GC.start

33	The combination of the_____methods allows the definition of "weak reference" objects.	_id2ref and define_finalizer	_GCref and define_final	_id2ref and finalizer	_id8ref and GC_final	_id2ref and define_finalizer
34	The_____provides a powerful way to catch and handle arbitrary invocations on an object.	method_accuring	method_missing	method_unwanted	GC	method_missing
35	The const_missing method calls_____to define a real constant to refer to each value it computes.	Module.const_missing	class.const_set	Module.const_set	const_set	Module.const_set
36	The _____ method returns an instance of TracedObject.	trace	find	search	catch	trace
37	Global method _____ , which accepts an object and executes a block under the protection of the Mutex associated with that object.	synchronized	attached	merged	joining	synchronized
38	Synchronized method consisted of the implementation of the _____method.	mutex	Object.mutex	attached	thread	Object.mutex
39	_____is a delegating wrapper class based on method_missing.	Synchronized_mutex	thread	SynchronizedObject	object	SynchronizedObject
40	_____like this incurs the slight overhead of parsing the string of code.	class_check	class_eval	class_modifies	level_class	class_eval
41	The attr_reader and attr_accessor method They accept attribute names as their arguments, and ndynamically create methods with those names.	reader and accessor	attr_reader and attr_writer	attr_access and attr_checker	attr_reader and attr_accessor	attr_reader and attr_accessor
42	The_____method defines class attributes rather than instance attributes.	class_attrs	class_reader	class_writer	attr_reader	class_attrs

43	The_____option loads the specified library before it starts running the program.	-q	-w	–r	-a	–r
44	Kernel.require and Kernel.load methods defines an_____hook to track definitions of new classes.	Object.inherited	Object.specified	class.inherited	kernel specifier	Object.inherited
45	_____# Matches the text "Rub" followed by an optional "y".	/Ruby/	/Ruby?/	/ruby?/	/Ruby?/m	/Ruby?/
46	_____# Matches Unicode characters in Multiline mode.	/./mu	/..../mu	/./m	/./\n	/./mu
47	_____# Matches a single slash character, no escape required.	%r|/.,|	%r|/|	%W|/|	%r|/RR/|	%r|/|
48	_____# Matches open and close parentheses.	/\(\\\)/	/\(\)/mul	/\(\)/	/\(\)/P	/\(\)/
49	prefix = ","_____# Matches a single backslash.	/\\/	/ \ /	/ ss\\ /	/\\dd?>/	/\\/
50	_____# Matches a comma followed by an ASCII TAB character	/#{prefix}\t/	/#{,}\t/	/#{\t}\t/	/ascii\t/	/#{prefix}\t/
51	[1,2].map{|x| /#{x}/} output:_____.	# => [/1/, /1/]	# => [/2/, /2/]	# => [/1/, /2/]	# => [/1/, /0/]	# => [/1/, /2/]
52	[1,2].map{|x| /#{x}/o} output: _____	# => [/1/, /1/]	# => [/01/, /01/]	# => [/o/, /1/]	# => [/o/, /o/]	# => [/1/, /1/]
53	Regexp.new("ruby?", Regexp::IGNORECASE) equivalent to_____.	# /ruby?/ingo	# /ruby?/I	# /ruby?/i	# /ruby/i	# /ruby?/i
54	_____# Match a digit /[0-9]/	/\d/	/\ddd/	/\0-9/	/\D/	/\d/
55	_____# Match a nondigit: /[^0-9]/	/\D/	/\ddd/	/\0-9/	/read/	/\D/
56	_____ # Match a whitespace character: /[\t\r\n\f]/	/\space/	/\white/	/\s/	/\ws/	/\s/
57	_____# Match "rub" plus 0 or more ys.	/ruby?/	/ruby+/	/ruby>/	/ruby*/	/ruby*/
58	_____ # Match "rub" plus 1 or more ys.	\/ruby^/	/ruby*/	/ruby+/	/ruby@/	/ruby+/

59	_____# Match "Ruby", "Ruby, ruby, ruby", etc.	/([Rr]uby(,)?)+/	/([R][r]uby(,)?)+/	/([Rr]uby)+/	/([Rr]uby(,)?)/	/([Rr]uby(,)?)+/
60	_____# Match ruby&rails or Ruby&Rails.	/([Rr])uby&\1ails/	/([Ruby&\1ails/	/([Ruby&Rails/	/([R])uby&ails/	/([Rr])uby&\1ails/

MODULES: Namespaces - Modules as Mixins - Includable Namespace Modules - Loading and Requiring Modules - Executing Loaded Code. Reflection and Meta programming: Evaluating Strings and Blocks - Querying, Setting, and Testing Variables – Regular Expressions. FILES AND DIRECTORIES: Listing and manipulating Directories and testing files. BASIC INPUT AND OUTPUT: Opening Stream – Reading from a Stream – Writing to a stream – Random Access Methods – Closing, Flusing and testing streams.

Modules

Like a class, a *module* is a named group of methods, constants, and class variables. Modules are defined much like classes are, but the module keyword is used in place of the class keyword. Unlike a class, however, a module cannot be instantiated, and it cannot be subclassed. Modules stand alone; there is no "module hierarchy" of inheritance.

Modules are used as namespaces and as mixins. The subsections that follow explain these two uses.

Just as a class object is an instance of the Class class, a module object is an instance of the Module class. Class is a subclass of Module. This means that all classes are modules, but not all modules are classes. Classes can be used as namespaces, just as modules can. Classes cannot, however, be used as mixins.

Modules as Namespaces

Modules are a good way to group related methods when object-oriented programming is not necessary. Suppose, for example, you were writing methods to encode and decode binary data to and from text using the Base64 encoding. There is no need for special encoder and decoder objects, so there is no reason to define a class here. All we need are two methods: one to encode and one to decode. We could define just two global methods:

```
def base64_encode
end

def base64_decode
end
```

To prevent namespace collisions with other encoding and decoding methods, we've given our method names the base64 prefix. This solution works, but most programmers prefer to avoid adding methods to the global

namespace when possible. A better sol-ution, therefore, is to define the two methods within a Base64 module:

```
module Base64
  def self.encode
  end

  def self.decode
  end
end
```

Note that we define our methods with a self. prefix, which makes them "class meth-ods" of the module. We could also explicitly reuse the module name and define the methods like this:

```
module Base64
  def Base64.encode
  end

  def Base64.decode
  end
end
```

Defining the methods this way is more repetitive, but it more closely mirrors the invocation syntax of these methods:

```
This is how we invoke the methods of the
Base64 module text = Base64.encode(data)
data = Base64.decode(text)
```

Note that module names must begin with a capital letter, just as class names do. Defining a module creates a constant with the same name as the module. The value of this constant is the Module object that represents the module.

Modules may also contain constants. Our Base64 implementation would likely use a constant to hold a string of the 64 characters used as digits in Base64:

```
module Base64
  DIGITS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' \
           'abcdefghijklmnopqrstuvwxyz' \
           '0123456789+/'
end
```

Outside the Base64 module, this constant can be referred to as Base64::DIGITS. Inside the module, our encode and decode methods can refer to it by its simple name DIGITS. If the two methods had some need to share nonconstant data, they could use a class variable (with a @@ prefix), just as they could if they were defined in a class.

Modules As Mixins

The second use of modules is more powerful than the first. If a module defines instance methods instead of the class methods, those instance methods can be mixed in to other classes. Enumerable and Comparable are well-known examples of mixin modules. Enumerable defines useful iterators that are implemented in terms of an each iterator. Enumerable doesn't define the each method itself, but any class that defines it can mix in the Enumerable module to instantly add many useful iterators. Comparable is similar; it defines comparison operators in terms of the general-purpose comparator <=>. If your class defines <=>, you can mix in Comparable to get <, <=, == >, >=, and between? for free.

To mix a module into a class, use include. include is usually used as if it were a language keyword:

```
class Point
  include Comparable
end
```

In fact, it is a private instance method of Module, implicitly invoked on self—the class into which the module is being included. In method form, this code would be:

```
class Point
  include(Comparable)
end
```

Because include is a private method, it must be invoked as a function, and we cannot write self.include(Comparable). The include method accepts any number of Module objects to mix in, so a class that defines each and <=> might include the line:

include Enumerable, Comparable. The inclusion of a module affects the type-checking method is_a? and the switch-equality operator ===.

For example, String mixes in the Comparable module and, in Ruby 1.8, also mixes in the Enumerable module:

```
"text".is_a? Comparable   # => true
Enumerable === "text"     # => true in Ruby 1.8, false in 1.9
```

Note that instanceof? only checks the class of its receiver, not superclasses or modules, so the following is false:

```
"text".instance_of? Comparable   # => false
```

Although every class is a module, the include method does not allow a class to be included within another class. The arguments to include must be modules declared with module, not classes.

It is legal, however, to include one module into another. Doing this simply makes the instance methods of the included modules into instance methods of the including module. As an example, consider this code

```
module Iterable   # Classes that define next can include this module
  include
  Enumerable      #          Define iterators on top of each
  def each        #          And define each on top of next
    loop { yield self.next }
  end
end
```

The normal way to mix in a module is with the Module.include method. Another way is with Object.extend. This method makes the instance methods of the specified mod-ule or modules into singleton methods of the receiver object. (And if the receiver object is a Class instance, then the methods of the receiver become class methods of that class.) Here is an example:

```
countdown =
Object.new              #    A plain old object
                             The each iterator as a singleton
def countdown.each      #    method
  yield 3
  yield 2
  yield 1
end
countdown.extend(Enumerabl Now the object has all Enumerable
e) #                         methods
                             Prints "[1, 2, 3]"
print countdown.sort #
```

Includable Namespace Modules

It is possible to define modules that define a namespace but still allow their methods to be mixed in. The Math module works like this:

```
                #
Math.sin(0) =>  0.0: Math is a namespace
include 'Math' # The Math namespace can be included
                  = 0.0: Now we have easy access to the
sin(0)          # > functions
```

The Kernel module also works like this: we can invoke its methods through the Kernel namespace, or as private methods of Object, into which it is included.

Like the public, protected, and private methods, the module_function method can also be invoked with no arguments. When invoked in this way, any instance methods sub-sequently defined in the module will be module functions: they will become public class methods and private instance methods. Once you have invoked module_function with no arguments, it remains in effect for the rest of the module definition—so if you want to define methods that are not module functions, define those first.

Loading and Requiring Modules

Ruby programs may be broken up into multiple files, and the most natural way to partition a program is to place each nontrivial class or module into a separate file. These separate files can then be reassembled into a single program (and, if well-designed, can be reused by other programs) using require or load. These are global functions defined in Kernel, but are used like language keywords. The same require method is also used for loading files from the standard library.

load and require serve similar purposes, though require is much more commonly used than load. Both functions can load and execute a specified file of Ruby source code. If the file to load is specified with an absolute path, or is relative to ~ (the user's home directory), then that specific file is loaded. Usually, however, the file is specified as a relative path, and load and require search for it relative to the directories of Ruby's load path (details on the load path appear below).

Despite these overall similarities, there are important differences between load and require:

In addition to loading source code, require can also load binary extensions to Ruby. Binary extensions are, of course, implementation-dependent, but in C-based implementations, they typically take the form of shared library files with exten-sions like *.so* or *.dll*.

load expects a complete filename including an extension. require is usually passed a library name, with no extension, rather than a filename. In that case, it searches for a file that has the library name as its base name and an appropriate source or native library extension. If a directory contains both an *.rb* source file and a binary extension file, require will load the source file instead of the binary file.

load can load the same file multiple times. require tries to prevent multiple loads of the same file. (require can be fooled, however, if you use two different, but equivalent, paths to the same library file. In Ruby 1.9, require expands relative paths to absolute paths, which makes it somewhat harder to fool.) require keeps track of the files that have been loaded by appending them to the global array $" (also known as $LOADED_FEATURES). load does not do this.

load loads the specified file at the current $SAFE level. require loads the specified library with $SAFE set to 0, even if the code that called require has a higher value for that variable. See §10.5 for more on $SAFE and Ruby's security system. (Note that if $SAFE is set to a value higher than 0, require will refuse to load any file with a tainted filename or from a world-writable directory. In theory, therefore, it should be safe for require to load files with a reduced $SAFE level.)

Executing Loaded Code

load and require execute the code in the specified file immediately. Calling these meth-ods is not, however, equivalent to simply replacing the call to load or require with the code contained by the file.*

Files loaded with load or require are executed in a new top-level scope that is different from the one in which load or require was invoked. The loaded file can see all global variables and constants that have been defined at the time it is loaded, but it does not have access to the local scope from which the load was initiated. The implications of this include the following:

The local variables defined in the scope from which load or require is invoked are not visible to the loaded file.

Any local variables created by the loaded file are discarded once the load is complete; they are never visible outside the file in which they are defined.

At the start of the loaded file, the value of self is always the main object, just as it is when the Ruby interpreter starts running. That is, invoking load or require within a method invocation does not propagate the receiver object to the loaded file.

Reflection and Meta programming

Evaluating Strings and Blocks

One of the most powerful and straightforward reflective features of Ruby is its eval method. If your Ruby program can generate a string of valid Ruby code, the Kernel.eval method can evaluate that code:

```
x = 1
eval "x + 1"          # => 2
```

eval is a very powerful function, but unless you are actually writing a shell program (like *irb*) that executes lines of Ruby code entered by a user you are unlikely to really need it. (And in a networked context, it is almost never safe to call eval on text received from a user, as it could contain malicious code.) Inexperienced programmers some-times end up using eval as a crutch. If you find yourself using it in your code, see if there isn't a way to avoid it. Having said that, there are some more useful ways to use eval and eval-like methods.

Querying, Setting, and Testing Variables

In addition to listing defined variables and constants, Ruby Object and Module also define reflective methods for querying, setting, and removing instance variables, class variables, and constants. There are no special purpose methods for querying or setting local variables or global variables, but you can use the eval method for this purpose:

```
x = 1
varname = "x"
eval(varname)        # => 1
eval("varname = '$g'")      # Set varname to "$g"
eval("#{varname} = x")    # Set $g to 1
```

```
eval(varname)        # => 1
```

Note that eval evaluates its code in a temporary scope. eval can alter the value of instance variables that already exist. But any new instance variables it defines are local to the invocation of eval and cease to exist when it returns. (It is as if the evaluated code is run in the body of a block—variables local to a block do not exist outside the block.)

You can query, set, and test the existence of instance variables on any object and of class variables and constants on any class or module:

```
o = Object.new
o.instance_variable_set(:@x, 0)   # Note required @ prefix
o.instance_variable_get(:@x)      # => 0

o.instance_variable_defined?(:@x)   #
=>                                  True
Object.class_variable_set(:@@x,     1)
#                                   Private in Ruby 1.8
Object.class_variable_get(:@
@x)                          #      Private in Ruby 1.8
Object.class_variable_defined?(:@@
x) #                                => true; Ruby 1.9 and later
Math.const_set(:EPI,
Math::E*Math::PI)
Math.const_get(:EPI)         # => 8.53973422267357
Math.const_defined? :EPI  # => true
```

Regular Expressions

A *regular expression* (also known as a regexp or regex) describes a textual pattern. Ruby's Regexp class* implements regular expressions, and both Regexp and String de-fine pattern matching methods and operators. Like most languages that support regular expressions, Ruby's Regexp syntax follows closely (but not precisely) the syntax of Perl 5.

Regexp Literals

Regular expression literals are delimited by forward slash characters:

```
/Ruby?/     # Matches the text "Rub" followed by an optional "y"
```

The closing slash character isn't a true delimiter because a regular expression literal may be followed by one or more optional flag characters

that specify additional infor-mation about the how pattern matching is to be done. For example:

```
/ruby?/i # Case-insensitive: matches "ruby" or "RUB", etc.
/./mu    # Matches Unicode characters in Multiline mode
```

The allowed modifier characters are shown in following Table

Regular expression modifier characters

Modifier	Description
	Ignore case when matching text.
	The pattern is to be matched against multiline text, so treat newline as an ordinary character: allow . to match newlines.
	Extended syntax: allow whitespace and comments in regexp.
	Perform #{} interpolations only once, the first time the regexp literal is evaluated.
u,e,s,n	Interpret the regexp as Unicode (UTF-8), EUC, SJIS, or ASCII. If none of these modifiers is specified, the regular expression is assumed to use the source encoding.

Like string literals delimited with %Q, Ruby allows you to begin your regular expres-sions with %r followed by a delimiter of your choice. This is useful when the pattern you are describing contains a lot of forward slash characters that you don't want to escape:

```
%r|/|          # Matches a single slash character, no escape required
%r[</(.*)>]i# Flag characters are allowed with this syntax, too
```

Regular expression syntax gives special meaning to the characters (), [], {}, ., ?, +, *, |, ^, and $. If you want to describe a pattern that includes one of these characters literally, use a backslash to escape it. If you want to describe a pattern that includes a backslash, double the backslash:

```
/\(\)/      # Matches open and close parentheses
/\\/        # Matches a single backslash
```

Regular expression literals behave like double-quoted string literals and can include escape characters such as \n, \t, and (in Ruby 1.9) \u (see Table 3-1 in Chapter 3 for a complete list of escape sequences):

```
money = /[$\u20AC\u{a3}\u{a5}]/ # match dollar,euro,pound, or yen sign
```

Also like double-quoted string literals, Regexp literals allow the interpolation of arbi-trary Ruby expressions with the #{} syntax:

```
prefix = ","
/#{prefix}\t/   # Matches a comma followed by an ASCII TAB character
```

Note that interpolation is done early, before the content of the regular expression is parsed. This means that any special characters in the interpolated expression become part of the regular expression syntax. Interpolation is normally done anew each time a regular expression literal is evaluated. If you use the o modifier, however, this interpo-lation is only performed once, the first time the code is parsed. The behavior of the o modifier is best demonstrated by example:

```
[1,2].map{|x| /#{x}/} # => [/1/, /2/]
[1,2].map{|x| /#{x}/o}        # => [/1/, /1/].
```

POSSIBLE QUESTIONS

UNIT 1

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. Discuss in detail about mixins in modules.
2. Describe about creating, deleting and renaming files and directories in detail.
3. Explain about Files and Directories with examples.
4. Discuss in detail about Basic input and output in Ruby with suitable examples.
5. Write a Ruby program for the situation given below.

   A primary school in a rural village of Coimbatore has 3 sections of second standard. Unfortunately you have to engage all the three sections simultaneously. The students of the three sections should get an impression that the teacher will visit our classroom for every five minutes.

6. Discuss in detail about Basic input and output statements in Ruby.
7. Write a ruby script to read a file, writing into a file and Count the number of lines and characters in a file.
8. Explain about Modules as Mixins in Ruby
9. Discuss in detail about listing and manipulating directories and testing files in Ruby
10. Explain about composing the modules and inclusion of files with suitable example.

PART – C (10 MARKS)

1. Design an application form using tk classes and validate all fields on Rails framework
2. Explain about different Data Types in Ruby.
3. Explain in detail about Object creation and initialization in Ruby
4. Write a ruby program to create a main thread and execute multiple process through the main thread.
5. Write a note on Defining, Calling and Undefining methods in Ruby

Sno	QESTIONS	OPTION1	OPTION2	OPTION3	OPTIONS4	ANS
1	Procs are_____containing code. They can be placed inside a variable and passed around like any other object	anonymous or nameless methods	True methods	naming methods	mixins	anonymous or nameless methods
2	_____is the process of reclaiming the memory space.	Garbage collection	scheduling	cleaning	threading	Garbage collection
3	Ruby's_____are defined using characters of the ASCII character set.	Syntactic rules	lexical rules	Semantic	procedures	lexical rules
4	To run a Ruby program that includes Unicode characters encoded in___, invoke the interpreter with the _____ option.	UTF-8, -Ku	UTF-9 , -Kuf	UTF-10 , -Kuf	UTF-10 , -K	UTF-8, -Ku
5	The Ruby_____reads the file and executes the script.	compiler	module	interpreter	virtual machine	interpreter
6	A hash literal is written as a_____of key/value pairs, enclosed within_____.	comma-separated list ,curly braces	period-separated list ,curly braces	colon-separated list ,curly braces	semi colon-separated list ,brackets	comma-separated list ,curly braces
7	The_____objects work more efficiently as hash keys than strings.	array	Symbol	number	alpha-numeric	Symbol
8	Symbols are_____, written as colon-prefixed identifiers.	mutable interned strings	immutable external strings	immutable interned strings	immutable interned numbers	immutable interned strings
9	Objects used as keys in a hash that returns a_____hashcode for the key.	Fixnum	Bignum	Equal	Slashed	Fixnum
10	Changing the content of an object typically_____its hash code.	Not changes	changes	Grouping	isolates	changes
11	If mutable hash keys are used,_____method of the Hash must be called every time to mutate a key.	Hashing	rebuild	rehash	bind	rehash
12	A_____object represents the values between a start value and an end value.	Value	Range	String	class	Range
13	Range literals are written by placing_____dots between the start and end value	one or two	five or six	two or three	ten to twelve	two or three
14	If two dots are used, then the range is_____and the_____value is part of the range.	inclusive, end	exclusive, end	inclusive, start	exclusive, start	inclusive, end
15	The comparison operator <=>, which compares its two operands and evaluates to_____.	–2, 0, or 2	–3, 1, or 3	–10, 0, or 100	–1, 0, or 1	–1, 0, or 1
16	33. r = 'a'..'c' r.each{|i| print "[#{i}]"}output"_____	Prints"[a][b][c]"	Prints"[a][a][a]"	Prints"[a][1][1]"	Prints"[1][1][1]"	Prints"[a][b][c]"
17	34. r = 'a'..'c' r.step(2) { |i| print "[#{i}]"}output;_____	Prints "[a][d]"	Prints "[a][b]"	Prints "[a][b][c]"	Prints "[a][c]"	Prints "[a][c]"
18	Ruby interpreter maintains a_____in which it stores the names of all the classes, methods, and variables.	symbol table	array table	hash table	relational table	symbol table
19	All objects inherit from a class named_____and share the methods defined by that class.	root	base	Object	dynamic	Object
20	The_____and methods provide the default technique for creating new classes.	new initialize	start initialize	new mute	new start	new initialize
21	Ruby uses a technique called_____to automatically destroy objects that are no longer needed.	Allocated	garbage collection	synchronization	normalization	garbage collection
22	The Ruby_____class represents an error or unexpected condition in a program and encapsulates an error message.	Exception	Arrar	thread	error	Exception
23	The_____module defines four conversion methods that behave as global conversion functions.	Module	Class	Kernel	Mixin	Kernel
24	Classes can also override the_____and_____methods directly to produce any kind of copy they desire.	clone dup	copy dup	clone dos	clone similar	clone dup
25	Exception objects are instances of the_____class or one of its subclasses.	Root class	Exception	Error	Restricted	Exception
26	Ruby uses the Kernel method_____clause to handle exceptions.	Raise	Block	rescue	Handle	rescue
27	The most of Exception subclasses extend a class known as _____	Standard	StandardError	Error	FormalError.	StandardError
28	The_____method returns a string that may provide human-readable details about what went wrong.	message	information	mode	clue	message
29	_____method returns an array of strings that represents the call stack at the point that the exception was raised.	backtrace	information	mode	message	backtrace
30	Exception objects are typically created by the_____method.	message	rescue	raise	begin	raise
31	If you create your own exception object, you can set the stack trace with the_____method.	set_backtrace	get_backtrace	set_backerase	stop_backtrace	set_backtrace
32	If raise is called with no arguments, it creates a new_____object (with no message)	stdError	RuntimeError	static error	root clause	RuntimeError
33	if raise is used with no arguments inside a_____clause, it simply re-raises the exception that was being handled.	message	fail	mute	rescue	rescue
34	If raise is called with a_____Exception object as its argument, it raises that exception.	double	single	null	multiple	single
35	If_____is called with a single string argument, it creates a new_____exception object, with the specified string as its message, and raises that exception.	raise, RuntimeError	rescue RuntimeError	raise stdError	raise StaticError	raise, RuntimeError
36	raise accepts a string as its optional_____argument.	third	first	second	forth	second
37	The_____statement exists simply to delimit the block of code within which exceptions are to be handled.	begin	start	first	remote	begin
38	In a rescue clause, the global variable_____refers to the Exception object that is being handled.	$$	$~	$!	^!	$!
39	The_____clause is an alternative to the rescue clauses; it is used if none of the rescue clauses are needed.	not	else	irrelevant	alter	else
40	If the code executes a return statement, then the execution skips the else clause and jumps directly to the _____clause before returning.	ensure	raise	caller	kernel	ensure
41	The purpose of the_____clause is to ensure that housekeeping details get taken care of.	raise	ensure	caller	kernel	ensure
42	A_____of execution is a sequence of Ruby statements that run in parallel with the main sequence of statements that the interpreter is running.	series	block	thread	set	thread
43	The return value of the block becomes available through the_____method of the Thread object.	main	rule	break	value	value
44	c.superclass ;Returns the _____	superclass of a class c.	subclass of a class c.	relevant class of a class c.	irrelevant class of a class c.	superclass of a class c.
45	o.instance_of? c ; Determines whether the object _____	class == c.	o == c.	o.class == c.	class == 0.	o.class == c.
46	The_____object returns the bindings in effect at the location of the call.	Kernel.binding	Kernel.joining	Kernel.mergeing	Kernel	Kernel.binding
47	The_____class defines quite a few class methods for working with files as entries in a filesystem.	Object	module	File	Directory	File
48	The_____method converts a relative path to a fully qualified path.	File.extend_path	File.expand_path	File.dir_path	Dir.expand_path	File.expand_path
49	The_____method tests whether two filenames refer to the same file.	File.identical?	File.same?	File.ideal?	File.union?	File.identical?
50	_____tests whether a filename matches a specified pattern.	File.match	File.fnmatch	Dir.fnmatch	File.charmatch	File.fnmatch
51	Add_____if you want "hidden" files and directories whose names begin with "." to match.	File::FNM_MATCH	File::DOTMATCH	File::FNM_DOTMATCH	File::FNM_periodMATCH	File::FNM_DOTMATCH
52	The_____method is used to list the contents of a directory.	Dir.List	List.entries	File.entries	Dir.entries	Dir.entries
53	puts Dir.getwd# Print _____	current working directory	stop working directory	List working directory	remove working directory	current working directory
54	If you pass a block to the_____method, the directory will be restored to its original value when the block exits.	cdir	chdir	adder	create dir	chdir
55	Two threads maynot call_____with a block at the same time.	Dir.threadfile	chdir	Dir.chdir	thread.chdir	Dir.chdir
56	The efficiency of using_____is that Ruby only has to make one call to the OS to obtain all file metadata.	stat	status	rule	effect	stat
57	File.utime(atime, mtime, f) output:_____	update times	Change times	remove times	rollup	Change times
58	The_____module in the standard library allows us to create a stream wrapper around a string object.	stringio	stringclass	charIO	IOFile	stringio
59	_____objects represent the "standard input" and "standard output" streams used to read from and write to the console.	ReadWrite	File	class	IO	IO
60	The_____class defines some utility methods that read the entire contents of a file with one call.	Abstract	Nodal	File	Dir	File

THREADS AND PROCESSES: Thread Life Cycle – Thread scheduling – Thread Exclusion – Deadlock. Ruby Tk: Introduction- Widgets and classes. Networks: A Very Simple Client - A Very Simple Server – Datagram - A Multiplexing Server - Fetching Web Pages. Ruby on Rails: Building a development Environment: Installation – Installing Databases – Code editors – web server Configuration – Creating an web application.

Thread Life cycle

A new threads are created with Thread.new. You can also use the synonyms Thread.start and Thread.fork.

There is no need to start a thread after creating it, it begins running automatically when CPU resources become available.

The Thread class defines a number of methods to query and manipulate the thread while it is running. A thread runs the code in the block associated with the call to *Thread.new* and then it stops running.

The value of the last expression in that block is the value of the thread, and can be obtained by calling the *value* method of the Thread object. If the thread has run to completion, then the value returns the thread's value right away. Otherwise, the *value* method blocks and does not return until the thread has completed.

The class method *Thread.current* returns the Thread object that represents the current thread. This allows threads to manipulate themselves. The class method *Thread.main* returns the Thread object that represents the main thread. This is the initial thread of execution that began when the Ruby program was started.

You can wait for a particular thread to finish by calling that thread's *Thread.join* method. The calling thread will block until the given thread is finished.

Threads and Exceptions

If an exception is raised in the main thread, and is not handled anywhere, the Ruby interpreter prints a message and exits. In threads, other than the main thread, unhandled exceptions cause the thread to stop running.

If a thread **t** exits because of an unhandled exception, and another thread **s** calls *t.join or t.value*, then the exception that occurred in **t** is raised in the thread **s**.

If *Thread.abort_on_exception* is *false*, the default condition, an unhandled exception simply kills the current thread and all the rest continue to run.

If you would like any unhandled exception in any thread to cause the interpreter to exit, set the class method *Thread.abort_on_exception* to *true*.

```
t = Thread.new { ... }
t.abort_on_exception = true
```

Thread Variables

A thread can normally access any variables that are in scope when the thread is created. Variables local to the block of a thread are local to the thread, and are not shared.

Thread class features a special facility that allows thread-local variables to be created and accessed by name. You simply treat the thread object as if it were a Hash, writing to elements using []= and reading them back using [].

In this example, each thread records the current value of the variable count in a threadlocal variable with the key *mycount*.

```
#!/usr/bin/ruby


count = 0

arr = []


10.times do |i|

  arr[i] = Thread.new {

    sleep(rand(0)/10.0)

    Thread.current["mycount"] = count

    count += 1

  }

end
```

```
arr.each {|t| t.join; print t["mycount"], ", " }
puts "count = #{count}"
```

This produces the following result −

```
8, 0, 3, 7, 2, 1, 6, 5, 4, 9, count = 10
```

The main thread waits for the subthreads to finish and then prints out the value of *count* captured by each.

Thread Priorities

The first factor that affects the thread scheduling is the thread priority: high-priority threads are scheduled before low-priority threads. More precisely, a thread will only get CPU time if there are no higher-priority threads waiting to run.

You can set and query the priority of a Ruby Thread object with *priority* = and *priority*. A newly created thread starts at the same priority as the thread that created it. The main thread starts off at priority 0.

There is no way to set the priority of a thread before it starts running. A thread can, however, raise or lower its own priority as the first action it takes.

Thread Exclusion

If two threads share access to the same data, and at least one of the threads modifies that data, you must take special care to ensure that no thread can ever see the data in an inconsistent state. This is called *thread exclusion*.

Mutex is a class that implements a simple semaphore lock for mutually exclusive access to some shared resource. That is, only one thread may hold the lock at a given time. Other threads may choose to wait in line for the lock to become available, or may simply choose to get an immediate error indicating that the lock is not available.

By placing all accesses to the shared data under control of a *mutex*, we ensure consistency and atomic operation. Let's try to examples, first one without mutax and second one with mutax −

Example without Mutax

```
#!/usr/bin/ruby

require 'thread'


count1 = count2 = 0

difference = 0

counter = Thread.new do

  loop do

    count1 += 1

    count2 += 1

  end

end

spy = Thread.new do

  loop do

    difference += (count1 - count2).abs

  end

end

sleep 1

puts "count1 :  #{count1}"

puts "count2 :  #{count2}"

puts "difference : #{difference}"
```

This will produce the following result −

```
count1 :  1583766
count2 :  1583766
difference : 0
```

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

```
require 'thread'
mutex = Mutex.new

count1 = count2 = 0
difference = 0
counter = Thread.new do
  loop do
    mutex.synchronize do
      count1 += 1
      count2 += 1
    end
  end
end
spy = Thread.new do
  loop do
    mutex.synchronize do
      difference += (count1 - count2).abs
    end
  end
end
sleep 1
mutex.lock
puts "count1 :  #{count1}"
puts "count2 :  #{count2}"
puts "difference : #{difference}"
```

This will produce the following result −

```
count1 :  696591
count2 :  696591
difference : 0
```

Handling Deadlock

When we start using *Mutex* objects for thread exclusion we must be careful to avoid *deadlock*. Deadlock is the condition that occurs when all threads are waiting to acquire a resource held by another thread. Because all threads are blocked, they cannot release the locks they hold. And because they cannot release the locks, no other thread can acquire those locks.

This is where *condition variables* come into picture. A *condition variable* is simply a semaphore that is associated with a resource and is used within the protection of a particular *mutex*. When you need a resource that's unavailable, you wait on a condition variable. That action releases the lock on the corresponding *mutex*. When some other thread signals that the resource is available, the original thread comes off the wait and simultaneously regains the lock on the critical region.

Example

```
#!/usr/bin/ruby

require 'thread'

mutex = Mutex.new


cv = ConditionVariable.new

a = Thread.new {

  mutex.synchronize {

    puts "A: I have critical section, but will wait for cv"

    cv.wait(mutex)

    puts "A: I have critical section again! I rule!"

  }
```

```
}


puts "(Later, back at the ranch...)"


b = Thread.new {
  mutex.synchronize {
    puts "B: Now I am critical, but am done with cv"
    cv.signal
    puts "B: I am still critical, finishing up"
  }
}
a.join
b.join
```

This will produce the following result −

```
A: I have critical section, but will wait for cv
(Later, back at the ranch...)
B: Now I am critical, but am done with cv
B: I am still critical, finishing up
A: I have critical section again! I rule!
```

Thread States

There are five possible return values corresponding to the five possible states as shown in the following table. The *status* method returns the state of the thread.

Thread state	Return value
Runnable	run
Sleeping	Sleeping

Aborting	aborting
Terminated normally	false
Terminated with exception	nil

Thread Class Methods

Following methods are provided by *Thread* class and they are applicable to all the threads available in the program. These methods will be called as using *Thread* class name as follows −

```
Thread.abort_on_exception = true
```

Here is the complete list of all the class methods available −

Thread Instance Methods

These methods are applicable to an instance of a thread. These methods will be called as using an instance of a *Thread* as follows −

```
#!/usr/bin/ruby

thr = Thread.new do   # Calling a class method new
  puts "In second thread"
  raise "Raise exception"
end
thr.join   # Calling an instance method join
```

Ruby - Tk

Introduction

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

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 operating system.

The basic component of a Tk-based application is called a widget. A component is also sometimes called a window, since, in Tk, "window" and "widget" are often used interchangeably.

Tk applications follow a widget hierarchy where any number of widgets may be placed within another widget, and those widgets within another widget, ad infinitum. The main widget in a Tk program is referred to as the root widget and can be created by making a new instance of the TkRoot class.

- Most Tk-based applications follow the same cycle: create the widgets, place them in the interface, and finally, bind the events associated with each widget to a method.
- There are three geometry managers; *place*, *grid* and *pack* that are responsible for controlling the size and location of each of the widgets in the interface.

Installation

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

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

Simple Tk Application

A typical structure for Ruby/Tk programs is to create the main or **root** window (an instance of TkRoot), add widgets to it to build up the user interface, and then start the main event loop by calling **Tk.mainloop**.

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

```
require 'tk'


root = TkRoot.new { title "Hello, World!" }
```

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

Here, after loading the tk extension module, we create a root-level frame using *TkRoot.new*. We then make a *TkLabel* widget as a child of the root frame, setting several options for the label. Finally, we pack the root frame and enter the main GUI event loop.

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

Ruby/Tk Widget Classes

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

- TkFrame Creates and manipulates frame widgets.
- TkButton Creates and manipulates button widgets.
- TkLabel Creates and manipulates label widgets.
- TkEntry Creates and manipulates entry widgets.
- TkCheckButton Creates and manipulates checkbutton widgets.
- TkRadioButton Creates and manipulates radiobutton widgets.
- TkListbox Creates and manipulates listbox widgets.
- TkComboBox Creates and manipulates listbox widgets.
- TkMenu Creates and manipulates menu widgets.

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

Standard Configuration Options

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

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

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

Ruby/Tk Geometry Management

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

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

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

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

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

Ruby/Tk Event Handling

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The configure Method

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

```
require "tk"

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

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

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

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

The cget Method

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

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

Networks

Ruby provides two levels of access to network services. At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both connection-oriented and connectionless protocols.

Ruby also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on.

What are Sockets?

Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents.

Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The *socket* provides specific classes for handling the common transports as well as a generic interface for handling the rest.

Sockets have their own vocabulary −

Sr.No.	Term & Description
1	**domain** The family of protocols that will be used as the transport mechanism. These values are constants such as PF_INET, PF_UNIX, PF_X25, and so on.
2	**type** The type of communications between the two endpoints, typically SOCK_STREAM for connection-oriented protocols and SOCK_DGRAM for connectionless protocols.
3	**protocol** Typically zero, this may be used to identify a variant of a protocol within a domain and type.
4	**hostname** The identifier of a network interface − A string, which can be a host name, a dotted-quad address, or an IPV6 address in colon (and possibly dot) notation A string "<broadcast>", which specifies an INADDR_BROADCAST address. A zero-length string, which specifies INADDR_ANY, or

	An Integer, interpreted as a binary address in host byte order.
5	**port** Each server listens for clients calling on one or more ports. A port may be a Fixnum port number, a string containing a port number, or the name of a service.

A Simple Client

Here we will write a very simple client program, which will open a connection to a given port and given host. Ruby class **TCPSocket** provides *open* function to open such a socket.

The **TCPSocket.open(hosname, port)** opens a TCP connection to *hostname*on the *port*.

Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file.

The following code is a very simple client that connects to a given host and port, reads any available data from the socket, and then exits −

```
require 'socket'      # Sockets are in standard library

hostname = 'localhost'
port = 2000

s = TCPSocket.open(hostname, port)

while line = s.gets   # Read lines from the socket
  puts line.chop      # And print with platform line terminator
end
s.close               # Close the socket when done
```

A Simple Server

To write Internet servers, we use the **TCPServer** class. A TCPServer object is a factory for TCPSocket objects.

Now call **TCPServer.open(hostname, port** function to specify a *port* for your service and create a **TCPServer** object.

Next, call the *accept* method of the returned TCPServer object. This method waits until a client connects to the port you specified, and then returns a *TCPSocket* object that represents the connection to that client.

```
require 'socket'              # Get sockets from stdlib


server = TCPServer.open(2000)   # Socket to listen on port 2000
loop {                        # Servers run forever
  client = server.accept       # Wait for a client to connect
  client.puts(Time.now.ctime)  # Send the time to the client
  client.puts "Closing the connection. Bye!"
  client.close                # Disconnect from the client
}
```

Now, run this server in background and then run the above client to see the result.

Multi-Client TCP Servers

Most servers on the Internet are designed to deal with large numbers of clients at any one time.

Ruby's *Thread* class makes it easy to create a multithreaded server.one that accepts requests and immediately creates a new thread of execution to process the connection while allowing the main program to await more connections −

```
require 'socket'              # Get sockets from stdlib


server = TCPServer.open(2000)   # Socket to listen on port 2000
loop {                        # Servers run forever
  Thread.start(server.accept) do |client|
  client.puts(Time.now.ctime)  # Send the time to the client
  client.puts "Closing the connection. Bye!"
  client.close                # Disconnect from the client
  end
}
```

In this example, you have a permanent loop, and when server.accept responds, a new thread is created and started immediately to handle the connection that has just been accepted, using the connection object passed into the thread. However, the main program immediately loops back and awaits new connections.

Using Ruby threads in this way means the code is portable and will run in the same way on Linux, OS X, and Windows.

A Tiny Web Browser

We can use the socket library to implement any Internet protocol. Here, for example, is a code to fetch the content of a web page −

```
require 'socket'

host = 'www.tutorialspoint.com'     # The web server
port = 80                           # Default HTTP port
path = "/index.htm"                 # The file we want

# This is the HTTP request we send to fetch a file
request = "GET #{path} HTTP/1.0\r\n\r\n"

socket = TCPSocket.open(host,port)  # Connect to server
socket.print(request)               # Send request
response = socket.read              # Read complete response
# Split response at first blank line into headers and body
headers,body = response.split("\r\n\r\n", 2)
print body                          # And display it
```

To implement the similar web client, you can use a pre-built library like **Net::HTTP** for working with HTTP. Here is the code that does the equivalent of the previous code −

```
require 'net/http'                  # The library we need
host = 'www.tutorialspoint.com'     # The web server
path = '/index.htm'                 # The file we want
```

```
http = Net::HTTP.new(host)        # Create a connection
headers, body = http.get(path)    # Request the file
if headers.code == "200"          # Check the status code
  print body
else
  puts "#{headers.code} #{headers.message}"
end
```

Please check similar libraries to work with FTP, SMTP, POP, and IMAP protocols.

Ruby on Rails: Building a development Environment: Installation

Ruby on Rails recommends to create three databases - a database each for development, testing, and production environment. According to convention, their names should be −

- library_development
- library_production
- library_test

You should initialize all three of them and create a user and password for them with full read and write privileges. We are using the **root** user ID for our application.

Rails Active Record is the Object/Relational Mapping (ORM) layer supplied with Rails. It closely follows the standard ORM model, which is as follows −

- tables map to classes,
- rows map to objects and
- columns map to object attributes.

Rails Active Records provide an interface and binding between the tables in a relational database and the Ruby program code that manipulates database records. Ruby method names are automatically generated from the field names of database tables.

Each Active Record object has CRUD (**C**reate, **R**ead, **U**pdate, and **D**elete) methods for database access. This strategy allows simple designs and straight forward mappings between database tables and application objects.

Translating a Domain Model into SQL

Translating a domain model into SQL is generally straight forward, as long as you remember that you have to write Rails-friendly SQL. In practical terms, you have to follow certain rules −

- Each entity (such as book) gets a table in the database named after it, but in the plural (books).
- Each such entity-matching table has a field called *id*, which contains a unique integer for each record inserted into the table.
- Given entity x and entity y, if entity y belongs to entity x, then table y has a field called x_id.
- The bulk of the fields in any table store the values for that entity's simple properties (anything that's a number or a string).

Creating Active Record Files (Models)

To create the Active Record files for our entities for library application, introduced in the previous chapter, issue the following command from the top level of the application directory.

```
library\> rails script/generate model Book
library\> rails script/generate model Subject
```

Using HTTPs with Ruby on Rails

Obtaining an SSL certificate

There are several different types of SSL certificates . You can group them by validation level (domain validated, organization validated, extended validation), by coverage (single-name, wildcard, multi-domains, etc.) and authenticity (self-signed vs publicly-trusted certificate authorities).

In general, *single-name* or *wildcard* certificates are the most common choices. They allow you to secure a single hostname (such as www.example.com) or an entire subdomain level (such as *.example.com). Unless you need some extra level of validation, *domain validated certificates* are the cheapest and most common solution. The second most-popular alternative are the *extended validation* certificates, which are generally recognized by the green bar displayed by the browsers in the address bar.

For production environments, you are required to purchase an SSL certificate issued by a trusted certificate authority (e.g. Digicert , Comodo , Let's Encrypt) or a reseller (e.g. DNSimple). To purchase a trusted SSL certificate, follow the instructions provided by the certificate provider.

For non-production applications, you can avoid the costs associated with the SSL certificate by using a self-signed SSL certificate. If you use a self-signed certificate the connection will still be encrypted, however your browser will likely display a security warning because the certificate is not

issued by a trusted certification authority. You can follow these instructions to generate a self-signed certificate.

Regardless the type of certificate, at the end of the issuance process you should obtain the following files:

1. The public SSL certificate
2. The private key
3. Optionally, a list of intermediate SSL certificates or an intermediate SSL certificate bundle

These files are required to proceed to the next step and configure the web server to support HTTPS.

Configuring the web server to support HTTPS

In this section we'll learn how to configure the most common Ruby on Rails web servers to serve an application under HTTPS. We'll use the public SSL certificate, the private key and the intermediate SSL chain we obtained at the previous step.

For the purpose of the examples, I'll use the following file names:

- certificate.crt - the public SSL certificate
- private.key - the private key

Depending on the web server, you may need to supply the SSL intermediate chain in a single file along with the public SSL certificate, or use two separate files:

- chain.pem - the intermediate SSL certificate bundle
- certificate_and_chain.pem - the SSL certificate and intermediate SSL certificate bundle

Intermediate and SSL Certificate Bundle

The creation of the intermediate SSL certificate bundle is generally one of the most confusing step, therefore it deserves a special mention.

The bundle is just a simple text file that contains the concatenation of all the intermediate SSL certificates. The order is generally in reverse order, from the most specific intermediate SSL certificate to the most generic (and/or the root certificate).

The root certificate is generally omitted as it should be bundle in the browser or in the operating system. If the bundle has to contain the server SSL certificate, then this must appear as the first certificate in the list (as this is the most specific).

```
SERVER CERTIFICATE
INTERMEDIATE CERTIFICATE 1
INTERMEDIATE CERTIFICATE 2
INTERMEDIATE CERTIFICATE N
ROOT CERTIFICATE
```

You can use a text editor to concatenate the files together, or the cat unix utility.

```
cat certificate.crt interm1.crt intermN.crt root.csr > certificate_and_chain.pem
cat interm1.crt intermN.crt root.csr > chain.pem
```

POSSIBLE QUESTIONS

UNIT 1

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. Write a brief note on Spawing new process and private thread variables.
2. Discuss in detail about the thread life cyclewith suitable examples.
3. Create a class called MyCar. Move all the methods from MyCar class that also pertains to the MyTruck class into the vehicle class. Make sure that all the previous method calls are executing when you exit from the application.
4. Explain about Ruby's networking capabilities
5. Describe in detail about controlling thread scheduler with suitable example.
6. Describe about Widgets and classes with suitable example
7. Write a ruby program to display notebook widget
8. Explain about Thread life cycle and thread scheduling
9. Discuss about creation of thread with suitable example.
10. Explain server multiplexing with suitable example.

PART – C (10 MARKS)

1. Design an application form using tk classes and validate all fields on Rails framework
2. Explain about different Data Types in Ruby.
3. Explain in detail about Object creation and initialization in Ruby
4. Write a ruby program to create a main thread and execute multiple process through the main thread.
5. Write a note on Defining, Calling and Undefining methods in Ruby

Sno	QUESTIONS	OPTION1	OPTION2	OPTION3	OPTIONS4	ANS
1	_____ # Match "ruby" or "rube"	/ruby or rube/	/ruby|rube/	/ruby||rube/	/ru[b][y]e/	/ruby|rube/
2	_____# Case-insensitive while matching "uby".	/R(i)uby/	/R[i]uby/	/R(?i)uby/	/Ruby|ruby/	/R(?i)uby/
3	A _____ object is more powerful when the Regexp that was matched contains subexpressions in parentheses.	MatchData	CatchData	MatchObject	Match regExp	MatchData
4	s = "one, two, three" s.split output:_____	# ["one,","two,","three"]	# ["onetwothree"]	[123]	# [‘one,’,’two,’,’three’]	# ["one,","two,","three"]
5	_____ allows computers to send individual packets of data to other computers, without the overhead of establishing a persistent connection.	TCP	UDP	SDK	PGP	UDP
6	The argument to _____ specifies the maximum amount of data we are interested in receiving.	recvto	reform	recvfrom	receiver	recvfrom
7	The server code uses the _____ class without special UDPServer class for datagram-based servers.	UDPSocket	TCPSocket	UDPconnect	disconnect	UDPSocket
8	The _____ method is used to write a multiplexing server.	Kernel.deselect	Kernel.connect	Servercode	Kernel.select	Kernel.select
9	We can use the _____ to implement any Internet protocol.	socket library	server library	header library	TCP and UDP	socket library
10	_____ are used if two threads are performing regular expression matching concurrently.	$SAFE and $^	$SAFE and $~	&&SAFE and $~	$SAFE and &~	$SAFE and $~
11	The _____ class provides hash-like behavior.	Exception	Expression	Thread	Mutex	Thread
12	Set and query the priority of a Ruby Thread object with _____.	priority## and priority.	==priority and priority~.	priority and =priority=.	priority= and priority.	priority= and priority.
13	When multiple threads of the same priority need to share the CPU, it is up to the _____to decide when, and for how long, each thread runs.	thread scheduler	thread setter	thread runner	thread cycle	thread scheduler

14	schedulers are_____, which means that they allow a thread to run only for a fixed amount of time before allowing another thread of the same priority to run.	non preemptin g	preemptin g	thread scheduler	priority	preemptin g
15	Long-running compute-bound threads should periodically call _____to ask the scheduler to yield the CPU to another thread.	Thread.st op	Thread.hal t	Thread.pa ss	Thread.m ove	Thread.pa ss
16	A thread can pause itself—enter the sleeping state—by calling _____.	Thread.st op	Thread.hal t	Thread.pa ss	Thread.m ove	Thread.st op
17	Threads are created in the _____ state, and are eligible to run right away.	runnable	movable	executabl e	throwable	runnable
18	_____ class method that operates on the current thread—there is no equivalent instance method, so one thread cannot force another thread to pause.	Thread.st op	Thread.hal t	Thread.pa ss	Thread.m ove	Thread.st op
19	A thread that has paused itself with Thread.stop or Kernel.sleep can be started again with the instance methods _____.	wakeup	run.	wakeup and run.	sleep and run.	wakeup and run.
20	A thread is terminate normally by calling _____.	Thread.ter minate	Thread.exi t	Thread.m ute	destroy	Thread.exi t
21	A thread can switch itself from the runnable state to one of the terminated states simply by exiting by _____.	ensuring an exception.	stoping an exception.	raising an exception.	catching an exception	raising an exception.
22	A thread can forcibly terminate another thread by invoking the instance method _____on the thread to be terminated.	kill	Thread.exi t	Thread.m ute	destroy	kill
23	The _____ method returns an array of Thread objects representing all live threads.	Thread.co unt	Thread.rol lout	Thread.list	Thread.ret urnall	Thread.list
24	If you want to impose some order onto a subset of threads, you can create a _____object and add threads to it.	ThreadGr oup	ThreadJoi n	ThreadMe rge	Threadspli t	ThreadGr oup