Ruby Tutorial

Tutorialspoint.com

***Ruby* is a scripting language designed by Yukihiro Matsumoto, also known as Matz.**

Ruby runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX. This tutorial gives an initial push to start you with Ruby. For more detail kindly check tutorialspoint.com/ruby

What is Ruby ?

Ruby is a pure object oriented programming language. It was created in 1993 by Yukihiro Matsumoto of Japan. Ruby is a general-purpose, interpreted programming language like PERL and Python.

What is IRb ?

Interactive Ruby (IRb) provides a shell for experimentation. Within the IRb shell, you can immediately view expression results, line by line.

This tool comes along with Ruby installation so you have nothing to do extra to have IRb working. Just type irb at your command prompt and an Interactive Ruby Session will start.

Ruby Syntax:

- Whitespace characters such as spaces and tabs are generally ignored in Ruby code, except when they appear in strings.
- Ruby interprets semicolons and newline characters as the ending of a statement. However, if Ruby encounters operators, such as +, -, or backslash at the end of a line, they indicate the continuation of a statement.
- Identifiers are names of variables, constants, and methods. Ruby identifiers are case sensitive. It mean Ram and RAM are two different itendifiers in Ruby.
- Ruby comments start with a pound/sharp (#) character and go to EOL.

Reserved words:

The following list shows the reserved words in Ruby. These reserved words should not be used as constant or variable names in your program, however, be used as method names.

BEGIN	do	next	then
END	else	nill	true
alias	elsif	not	undef
and	end	or	unless
begin	ensure	redo	until
break	false	rescue	when
case	for	retry	while
class	if	return	while
def	in	self	__FILE__

defined?	module	super	__LINE__

Here Docs in Ruby:

Here are different examples:

```
#!/usr/bin/ruby -w
print <<EOF
    This is the first way of creating
    her document ie. multiple line string.
EOF
print <<"EOF";                # same as above
    This is the second way of creating
    her document ie. multiple line string.
EOF
print <<`EOC`                 # execute commands
        echo hi there
        echo lo there
EOC
print <<"foo", <<"bar"  # you can stack them
        I said foo.
foo
        I said bar.
bar
```

Ruby Data Types:

Basic types are numbers, strings, ranges, arrays, and hashes.

Integer Numbers in Ruby:

```
123                  # Fixnum decimal
1_6889               # Fixnum decimal with underline
-5000                # Negative Fixnum
0377                 # octal
0xee                 # hexadecimal
0b1011011            # binary
?b                   # character code for 'b'
?\n                  # code for a newline (0x0a)
12345678901234567890 # Bignum
```

Float Numbers in Ruby:

```
1023.4               # floating point value
1.0e6                # scientific notation
4E20                 # dot not required
4e+20                # sign before exponential
```

String Literals:

Ruby strings are simply sequences of 8-bit bytes and they are objects of class String.

- 'VariableName': No interpolation will be done
- "#{VariableName} and Backslashes \n:" Interpolation will be done
- %q(VariableName): No interpolation will be done
- %Q(VariableName and Backslashes \n): Interpolation will be done
- %(VariableName and Backslashes \n): Interpolation will be done

- `echo command interpretation with interpolation and backslashes`
- %x(echo command interpretation with interpolation and backslashes)

Backslash Notations:

Following is the list of Backslash notations supported by Ruby:

Notation	Character represented
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\a	Bell (0x07)
\e	Escape (0x1b)
\s	Space (0x20)
\nnn	Octal notation (n being 0-7)
\xnn	Hexadecimal notation (n being 0-9, a-f, or A-F)
\cx, \C-x	Control-x
\M-x	Meta-x (c | 0x80)
\M-\C-x	Meta-Control-x
\x	Character x

Ruby Arrays:

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

Example:

```
#!/usr/bin/ruby
ary = [  "Ali", 10, 3.14, "This is a string", "last element", ]
ary.each do |i|
   puts i
end
```

This will produce following result:

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

Ruby Hashes:

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

Example:

```
#!/usr/bin/ruby
hsh = colors = { "red" => 0xf00, "green" => 0x0f0 }
hsh.each do |key, value|
   print key, " is ", value, "\n"
end
```

This will produce following result:

```
green is 240
red is 3840
```

Ruby Ranges:

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

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

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

Example:

```
#!/usr/bin/ruby
(10..15).each do |n|
   print n, ' '
end
```

This will produce following result:

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

Variable Types:

- $global_variable
- @@class_variable
- @instance_variable
- [OtherClass::]CONSTANT
- local_variable

Ruby Pseudo-Variables:

They are special variables that have the appearance of local variables but behave like constants. You can not assign any value to these variables.

- **self:** The receiver object of the current method.
- **true:** Value representing true.
- **false:** Value representing false.
- **nil:** Value representing undefined.
- **__FILE__:** The name of the current source file.
- **__LINE__:** The current line number in the source file.

Ruby Predefined Variables:

Following table lists all the Ruby's predefined variables.

Variable Name	Description
$!	The last exception object raised. The exception object can also be accessed using => in *rescue* clause.
$@	The *stack backtrace* for the last exception raised. The *stack backtrace* information can retrieved by Exception#backtrace method of the last exception.
$/	The input record separator (newline by default). *gets, readline,* etc., take their input record separator as optional argument.
$\	The output record separator (nil by default).
$,	The output separator between the arguments to print and Array#join (nil by default). You can specify separator explicitly to Array#join.
$;	The default separator for split (nil by default). You can specify separator explicitly for String#split.
$.	The number of the last line read from the current input file. Equivalent to ARGF.lineno.
$<	Synonym for ARGF.
$>	Synonym for $defout.
$0	The name of the current Ruby program being executed.
$$	The process pid of the current Ruby program being executed.
$?	The exit status of the last process terminated.
$:	Synonym for $LOAD_PATH.
$DEBUG	True if the -d or --debug command-line option is specified.
$defout	The destination output for *print* and *printf* (*$stdout* by default).
$F	The variable that receives the output from *split* when -a is specified. This variable is set if the -a command-line option is specified along with the -p or -n option.
$FILENAME	The name of the file currently being read from ARGF. Equivalent to ARGF.filename.
$LOAD_PATH	An array holding the directories to be searched when loading files with the load and require methods.
$SAFE	The security level • 0 --> No checks are performed on externally supplied (tainted) data. (default) • 1 --> Potentially dangerous operations using tainted data are forbidden. • 2 --> Potentially dangerous operations on processes and files are forbidden. • 3 --> All newly created objects are considered tainted. • 4 --> Modification of global data is forbidden.

$stdin	Standard input (STDIN by default).
$stdout	Standard output (STDOUT by default).
$stderr	Standard error (STDERR by default).
$VERBOSE	True if the -v, -w, or --verbose command-line option is specified.
$- x	The value of interpreter option -x (x=0, a, d, F, i, K, l, p, v). These options are listed below
$-0	The value of interpreter option -x and alias of $/.
$-a	The value of interpreter option -x and true if option -a is set. Read-only.
$-d	The value of interpreter option -x and alias of $DEBUG
$-F	The value of interpreter option -x and alias of $;.
$-i	The value of interpreter option -x and in in-place-edit mode, holds the extension, otherwise nil. Can enable or disable in-place-edit mode.
$-I	The value of interpreter option -x and alias of $:.
$-l	The value of interpreter option -x and true if option -lis set. Read-only.
$-p	The value of interpreter option -x and true if option -pis set. Read-only.
$_	The local variable, last string read by gets or readline in the current scope.
$~	The local variable, *MatchData* relating to the last match. Regex#match method returns the last match information.
$ n ($1, $2, $3...)	The string matched in the nth group of the last pattern match. Equivalent to m[n], where m is a *MatchData* object.
$&	The string matched in the last pattern match. Equivalent to m[0], where m is a *MatchData* object.
$`	The string preceding the match in the last pattern match. Equivalent to m.pre_match, where m is a *MatchData* object.
$'	The string following the match in the last pattern match. Equivalent to m.post_match, where m is a MatchData object.
$+	The string corresponding to the last successfully matched group in the last pattern match.
$+	The string corresponding to the last successfully matched group in the last pattern match.

Ruby Predefined Constants:

The following table lists all the Ruby's Predefined Constants.

NOTE: TRUE, FALSE, and NIL are backward-compatible. It's preferable to use true, false, and nil.

Constant Name	Description
TRUE	Synonym for true.
FALSE	Synonym for false.
NIL	Synonym for nil.

ARGF	An object providing access to virtual concatenation of files passed as command-line arguments or standard input if there are no command-line arguments. A synonym for $<.
ARGV	An array containing the command-line arguments passed to the program. A synonym for $*.
DATA	An input stream for reading the lines of code following the __END__ directive. Not defined if __END__ isn't present in code.
ENV	A hash-like object containing the program's environment variables. ENV can be handled as a hash.
RUBY_PLATFORM	A string indicating the platform of the Ruby interpreter.
RUBY_RELEASE_DATE	A string indicating the release date of the Ruby interpreter
RUBY_VERSION	A string indicating the version of the Ruby interpreter.
STDERR	Standard error output stream. Default value of *$stderr*.
STDIN	Standard input stream. Default value of $stdin.
STDOUT	Standard output stream. Default value of $stdout.
TOPLEVEL_BINDING	A Binding object at Ruby's top level.

Regular Expressions:

Syntax:

```
/pattern/
/pattern/im    # option can be specified
%r!/usr/local! # general delimited regular expression
```

Modifiers:

Modifier	Description
i	Ignore case when matching text.
o	Perform #{} interpolations only once, the first time the regexp literal is evaluated.
x	Ignores whitespace and allows comments in regular expressions
m	Matches multiple lines, recognizing newlines as normal characters
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.

Various patterns:

Pattern	Description
^	Matches beginning of line.
$	Matches end of line.
.	Matches any single character except newline. Using m option allows it to match newline as well.

[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets
re*	Matches 0 or more occurrences of preceding expression.
re+	Matches 0 or 1 occurrence of preceding expression.
re{ n}	Matches exactly n number of occurrences of preceding expression.
re{ n,}	Matches n or more occurrences of preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of preceding expression.
a| b	Matches either a or b.
(re)	Groups regular expressions and remembers matched text.
(?imx)	Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?-imx)	Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?: re)	Groups regular expressions without remembering matched text.
(?imx: re)	Temporarily toggles on i, m, or x options within parentheses.
(?-imx: re)	Temporarily toggles off i, m, or x options within parentheses.
(?#...)	Comment.
(?= re)	Specifies position using a pattern. Doesn't have a range.
(?! re)	Specifies position using pattern negation. Doesn't have a range.
(?> re)	Matches independent pattern without backtracking.
\w	Matches word characters.
\W	Matches nonword characters.
\s	Matches whitespace. Equivalent to [\t\n\r\f].
\S	Matches nonwhitespace.
\d	Matches digits. Equivalent to [0-9].
\D	Matches nondigits.
\A	Matches beginning of string.
\Z	Matches end of string. If a newline exists, it matches just before newline.
\z	Matches end of string.
\G	Matches point where last match finished.
\b	Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
\B	Matches nonword boundaries.
\n, \t, etc.	Matches newlines, carriage returns, tabs, etc.
\1...\9	Matches nth grouped subexpression.
\10	Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

File I/O:

Common methods include:

- File.join(p1, p2, ... pN) => "p1/p2/.../pN" platform independent paths
- File.new(path, modestring="r") => file
- File.new(path, modenum [, permnum]) => file
- File.open(fileName, aModeString="r") {|file| block} -> nil
- File.open(fileName [, aModeNum [, aPermNum]]) {|file| block} -> nil
- IO.foreach(path, sepstring=$/) {|line| block}
- IO.readlines(path) => array

Here is a list of the different modes of opening a file:

Modes	Description
r	Read-only mode. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Read-write mode. The file pointer will be at the beginning of the file.
w	Write-only mode. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Read-write mode. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Write-only mode. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Read and write mode. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

Operators and Precedence:

Top to bottom:

```
:: .
[]
**
-(unary) +(unary) ! ~
*  /  %
+  -
<<  >>
&
|  ^
>  >=  <  <=
<=> == === != =~ !~
&&
||
.. ...
=(+=, -=...)
not
and or
```

All of the above are just methods except these:

```
=, ::, ., .., ..., !, not, &&, and, ||, or, !=, !~
```

In addition, assignment operators(+= etc.) are not user-definable.

Control Expressions:

S.N.	Control Expression

1	if bool-expr [then]
	  body
	elsif bool-expr [then]
	  body
	else
	  body
	end

2	unless bool-expr [then]
	  body
	else
	  body
	end

3	expr if bool-expr
4	expr unless bool-expr

5	case target-expr
	  when comparison [, comparison]... [then]
	    body
	  when comparison [, comparison]... [then]
	    body
	  ...
	[else
	  body]
	end

6	loop do
	  body
	end

7	while bool-expr [do]
	 body
	end

8	until bool-expr [do]
	 body
	end

9	begin
	 body
	end while bool-expr

10	begin
	 body
	end until bool-expr

11	for name[, name]... in expr [do]
	  body
	end

12	expr.each do | name[, name]... |
	  body
	end

13	`expr while bool-expr`
14	`expr until bool-expr`

- **break** terminates loop immediately.
- **redo** immediately repeats w/o rerunning the condition.
- **next** starts the next iteration through the loop.
- **retry** restarts the loop, rerunning the condition.

Defining a Class:

Class names begin w/ capital character.

```
class Identifier [< superclass ]
  expr..
end
```

Singleton classes, add methods to a single instance

```
class << obj
  expr..
end
```

Defining a Module:

Following is the general syntax to define a module in ruby

```
module Identifier
  expr..
end
```

Defining a Method:

Following is the general syntax to define a method in ruby

```
def method_name(arg_list, *list_expr, &block_expr)
  expr..
end
# singleton method
def expr.identifier(arg_list, *list_expr, &block_expr)
  expr..
end
```

- All items of the arg list, including parens, are optional.
- Arguments may have default values (name=expr).
- Method_name may be operators (see above).
- The method definitions can not be nested.
- Methods may override following operators:
  - .., |, ^, &, <=>, ==, ===, =~,
  - >, >=, <, <=,
  - +, -, *, /, %, **, <<, >>,
  - ~, +@, -@, [], []= (2 args)

Access Restriction:

- **public** - totally accessible.
- **protected** - accessible only by instances of class and direct descendants. Even through hasA relationships. (see below)
- **private** - accessible only by instances of class (must be called nekkid no "self." or anything else).

Example:

```
class A
  protected
  def protected_method
    # nothing
  end
end
class B < A
  public
  def test_protected
    myA = A.new
    myA.protected_method
  end
end
b = B.new.test_protected
```

Raising and Rescuing Exceptions:

Following is the syntax:

```
raise ExceptionClass[, "message"]
begin
  expr..
[rescue [error_type [=> var],..]
  expr..]..
[else
  expr..]
[ensure
  expr..]
end
```

Catch and Throw Exceptions:

- catch (:label) do ... end
- throw :label jumps back to matching catch and terminates the block.
- + can be external to catch, but has to be reached via calling scope.
- + Hardly ever needed.

Exceptions Classes:

Following is the class hierarchy of *Exception* class:

- Exception
  - NoMemoryError
  - ScriptError
    - LoadError
    - NotImplementedError
    - SyntaxError

- SignalException
    - Interrupt
- StandardError (default for rescue)
    - ArgumentError
    - IOError
        - EOFError
    - IndexError
    - LocalJumpError
    - NameError
        - NoMethodError
    - RangeError
        - FloatDomainError
    - RegexpError
    - RuntimeError (default for raise)
    - SecurityError
    - SystemCallError
        - Errno::*
    - SystemStackError
    - ThreadError
    - TypeError
    - ZeroDivisionError
- SystemExit
- fatal

Ruby Command Line Options:

```
$ ruby [ options ] [.] [ programfile ] [ arguments ... ]
```

The interpreter can be invoked with any of the following options to control the environment and behavior of the interpreter.

Option	Description
-a	Used with -n or -p to split each line. Check -n and -p options.
-c	Checks syntax only, without executing program.
-C dir	Changes directory before executing (equivalent to -X).
-d	Enables debug mode (equivalent to -debug).
-F pat	Specifies pat as the default separator pattern ($;) used by split.
-e prog	Specifies prog as the program from the command line. Specify multiple -e options for multiline programs.
-h	Displays an overview of command-line options.
-i [ext]	Overwrites the file contents with program output. The original file is saved with the extension ext. If ext isn't specified, the original file is deleted.
-I dir	Adds dir as the directory for loading libraries.
-K [kcode]	Specifies the multibyte character set code (e or E for EUC (extended Unix code); s or S for SJIS (Shift-JIS); u or U for UTF-8; and a, A, n, or N for ASCII).
-l	Enables automatic line-end processing. Chops a newline from input lines and appends a newline to output lines.
-n	Places code within an input loop (as in while gets; ... end).
-0[octal]	Sets default record separator ($/) as an octal. Defaults to \0 if octal not specified.
-p	Places code within an input loop. Writes $_ for each iteration.

-r lib	Uses *require* to load *lib* as a library before executing.
-s	Interprets any arguments between the program name and filename arguments fitting the pattern -xxx as a switch and defines the corresponding variable.
-T [level]	Sets the level for tainting checks (1 if level not specified).
-v	Displays version and enables verbose mode
-w	Enables verbose mode. If programfile not specified, reads from STDIN.
-x [dir]	Strips text before #!ruby line. Changes directory to *dir* before executing if *dir* is specified.
-X dir	Changes directory before executing (equivalent to -C).
-y	Enables parser debug mode.
--copyright	Displays copyright notice.
--debug	Enables debug mode (equivalent to -d).
--help	Displays an overview of command-line options (equivalent to -h).
--version	Displays version.
--verbose	Enables verbose mode (equivalent to -v). Sets $VERBOSE to true
--yydebug	Enables parser debug mode (equivalent to -y).

Ruby Environment Variables:

Ruby interpreter uses the following environment variables to control its behavior. The ENV object contains a list of all the current environment variables set.

Variable	Description
DLN_LIBRARY_PATH	Search path for dynamically loaded modules.
HOME	Directory moved to when no argument is passed to Dir::chdir. Also used by File::expand_path to expand "~".
LOGDIR	Directory moved to when no arguments are passed to Dir::chdir and environment variable HOME isn't set.
PATH	Search path for executing subprocesses and searching for Ruby programs with the -S option. Separate each path with a colon (semicolon in DOS and Windows).
RUBYLIB	Search path for libraries. Separate each path with a colon (semicolon in DOS and Windows).
RUBYLIB_PREFIX	Used to modify the RUBYLIB search path by replacing prefix of library path1 with path2 using the format path1;path2 or path1path2.
RUBYOPT	Command-line options passed to Ruby interpreter. Ignored in taint mode (Where $SAFE is greater than 0).
RUBYPATH	With -S option, search path for Ruby programs. Takes precedence over PATH. Ignored in taint mode (where $SAFE is greater than 0).
RUBYSHELL	Specifies shell for spawned processes. If not set, SHELL or COMSPEC are checked.

Ruby File I/O and Directories

Ruby provides a whole set of I/O-related methods implemented in the Kernel module. All the I/O methods are derived from the class IO.

The class *IO* provides all the basic methods, such as *read, write, gets, puts, readline, getc,* and *printf*.

This chapter will cover all ithe basic I/O functions available in Ruby. For more functions please refere to Ruby Class *IO*.

The *puts* Statement:

In previous chapters, you assigned values to variables and then printed the output using *puts* statement.

The *puts* statement instructs the program to display the value stored in the variable. This will add a new line at the end of each line it writes.

Example:

```
#!/usr/bin/ruby

val1 = "This is variable one"
val2 = "This is variable two"
puts val1
puts val2
```

This will produce following result:

```
This is variable one
This is variable two
```

The *gets* Statement:

The *gets* statement can be used to take any input from the user from standard screen called STDIN.

Example:

The following code shows you how to use the gets statement. This code will prompt the user to enter a value, which will be stored in a variable val and finally will be printed on STDOUT.

```
#!/usr/bin/ruby

puts "Enter a value :"
val = gets
puts val
```

This will produce following result:

```
Enter a value :
This is entered value
This is entered value
```

The *putc* Statement:

Unlike the *puts* statement, which outputs the entire string onto the screen, the *putc* statement can be used to output one character at a time.

Example:

The output of the following code is just the character H:

```
#!/usr/bin/ruby

str="Hello Ruby!"
putc str
```

This will produce following result:

```
H
```

The *print* Statement:

The *print* statement is similar to the *puts* statement. The only difference is that the *puts* statement goes to the next line after printing the contents, whereas with the *print* statement the cursor is positioned on the same line.

Example:

```
#!/usr/bin/ruby

print "Hello World"
print "Good Morning"
```

This will produce following result:

```
Hello WorldGood Morning
```

Opening and Closing Files:

Until now, you have been reading and writing to the standard input and output. Now we will see how to play with actual data files.

The *File.new* Method:

You can create a *File* object using *File.new* method for reading, writing, or both, according to the mode string. Finally you can use *File.close* method to close that file.

Syntax:

```
aFile = File.new("filename", "mode")
   # ... process the file
aFile.close
```

The *File.open* Method:

You can use *File.open* method to create a new file object and assign that file object to a file. However, there is one difference in between *File.open* and *File.new* methods. The difference is that the *File.open* method can be associated with a block, whereas you cannot do the same using the *File.new* method.

```
File.open("filename", "mode") do |aFile|
   # ... process the file
end
```

Here is a list of The Different Modes of Opening a File:

Modes	Description
r	Read-only mode. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Read-write mode. The file pointer will be at the beginning of the file.
w	Write-only mode. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Read-write mode. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Write-only mode. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Read and write mode. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

Reading and Writing Files:

The same methods that we've been using for 'simple' I/O are available for all file objects. So, gets reads a line from standard input, and *aFile.gets* reads a line from the file object aFile.

However, I/O objects provides additional set of access methods to make our lives easier.

The *sysread* Method:

You can use the method *sysread* to read the contents of a file. You can open the file in any of the modes when using the method sysread. For example :

```
#!/usr/bin/ruby

aFile = File.new("/var/www/tutorialspoint/ruby/test", "r")
if aFile
   content = aFile.sysread(20)
   puts content
else
```

```
   puts "Unable to open file!"
end
```

This statement will output the first 20 characters of the file. The file pointer will now be placed at the 21st character in the file.

The *syswrite* Method:

You can use the method syswrite to write the contents into a file. You need to open the file in write mode when using the method syswrite. For example :

```
#!/usr/bin/ruby

aFile = File.new("/var/www/tutorialspoint/ruby/test", "r+")
if aFile
   aFile.syswrite("ABCDEF")
else
   puts "Unable to open file!"
end
```

This statement will write "ABCDEF" into the file.

The *each_byte* Method:

This method belongs to the class *File*. The method *each_byte* is always associated with a block. Consider the following code sample: :

```
#!/usr/bin/ruby

aFile = File.new("/var/www/tutorialspoint/ruby/test", "r")
if aFile
   aFile.syswrite("ABCDEF")
   aFile.each_byte {|ch| putc ch; putc ?. }
else
   puts "Unable to open file!"
end
```

Characters are passed one by one to the variable ch and then displayed on the screen as follows:

```
T.h.i.s. .i.s. .l.i.n.e. .o.n.e.
.T.h.i.s. .i.s. .l.i.n.e. .t.w.o.
.T.h.i.s. .i.s. .l.i.n.e. .t.h.r.e.e.
.A.n.d. .s.o. .o.n.......
```

The *IO.readlines* Method:

The class *File* is a subclass of the class IO. The class IO also has some methods which can be used to manipulate files.

One of the IO class methods is *IO.readlines*. This method returns the contents of the file line by line. The following code displays the use of the method *IO.readlines*:

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

```
arr = IO.readlines("/var/www/tutorialspoint/ruby/test")
puts arr[0]
puts arr[1]
```

In this code, the variable arr is an array. Each line of the file *test* will be an element in the array arr. Therefore, arr[0] will contain the first line, whereas arr[1] will contain the second line of the file.

The *IO.foreach* Method:

This method also returns output line by line. The difference between the method *foreach* and the method *readlines* is that the method *foreach* is associated with a block. However, unlike the method *readlines*, the method *foreach* does not return an array. For example:

```
#!/usr/bin/ruby

IO.foreach("test"){|block| puts block}
```

This code will pass the contents of the file *test* line by line to the variable block, and then the output will be displayed on the screen.

Renaming and Deleting Files:

You can rename and delete files programmatically with Ruby with the *rename* and *delete* methods.

Following is the example to rename an existing file *test1.txt*:

```
#!/usr/bin/ruby

# Rename a file from test1.txt to test2.txt
File.rename( "test1.txt", "test2.txt" )
```

Following is the example to delete an existing file *test2.txt*:

```
#!/usr/bin/ruby

# Delete file test2.txt
File.delete("text2.txt")
```

File Modes and Ownership:

Use the *chmod* method with a mask to change the mode or permissions/access list of a file:

Following is the example to change mode of an existing file *test.txt* to a mask value:

```
#!/usr/bin/ruby

file = File.new( "test.txt", "w" )
file.chmod( 0755 )
```

Following is the table which can help you to choose different mask for *chmod* method:

Mask	Description
0700	rwx mask for owner
0400	r for owner
0200	w for owner
0100	x for owner
0070	rwx mask for group
0040	r for group
0020	w for group
0010	x for group
0007	rwx mask for other
0004	r for other
0002	w for other
0001	x for other
4000	Set user ID on execution
2000	Set group ID on execution
1000	Save swapped text, even after use

File Inquiries:

The following command tests whether a file exists before opening it:

```
#!/usr/bin/ruby

File.open("file.rb") if File::exists?( "file.rb" )
```

The following command inquire whether the file is really a file:

```
#!/usr/bin/ruby

# This returns either true or false
File.file?( "text.txt" )
```

The following command finds out if it given file name is a directory:

```
#!/usr/bin/ruby

# a directory
File::directory?( "/usr/local/bin" ) # => true

# a file
File::directory?( "file.rb" ) # => false
```

The following command finds whether the file is readable, writable or executable:

```
#!/usr/bin/ruby

File.readable?( "test.txt" )   # => true
File.writable?( "test.txt" )   # => true
File.executable?( "test.txt" ) # => false
```

The following command finds whether the file has zero size or not:

```
#!/usr/bin/ruby

File.zero?( "test.txt" )      # => true
```

The following command returns size of the file :

```
#!/usr/bin/ruby

File.size?( "text.txt" )     # => 1002
```

The following command can be used to find out a type of file :

```
#!/usr/bin/ruby

File::ftype( "test.txt" )     # => file
```

The ftype method identifies the type of the file by returning one of the following: *file, directory, characterSpecial, blockSpecial, fifo, link, socket, or unknown.*

The following command can be used to find when a file was created, modified, or last accessed :

```
#!/usr/bin/ruby

File::ctime( "test.txt" ) # => Fri May 09 10:06:37 -0700 2008
```

```
File::mtime( "text.txt" ) # => Fri May 09 10:44:44 -0700 2008
File::atime( "text.txt" ) # => Fri May 09 10:45:01 -0700 2008
```

Directories in Ruby:

All files are contained within various directories, and Ruby has no problem handling these too. Whereas the *File* class handles files, directories are handled with the *Dir* class.

Navigating Through Directories:

To change directory within a Ruby program, use *Dir.chdir* as follows. This example changes the current directory to */usr/bin*.

```
Dir.chdir("/usr/bin")
```

You can find out what the current directory is with *Dir.pwd*:

```
puts Dir.pwd # This will return something like /usr/bin
```

You can get a list of the files and directories within a specific directory using *Dir.entries*:

```
puts Dir.entries("/usr/bin").join(' ')
```

Dir.entries returns an array with all the entries within the specified directory. *Dir.foreach* provides the same feature:

```
Dir.foreach("/usr/bin") do |entry|
   puts entry
end
```

An even more concise way of getting directory listings is by using Dir's class array method:

```
Dir["/usr/bin/*"]
```

Creating a Directory:

The *Dir.mkdir* can be used to create directories:

```
Dir.mkdir("mynewdir")
```

You can also set permissions on a new directory (not one that already exists) with mkdir:

NOTE: The mask 755 sets permissions owner, group, world [anyone] to rwxr-xr-x where r = read, w = write, and x = execute.

```
Dir.mkdir( "mynewdir", 755 )
```

Deleting a Directory:

The *Dir.delete* can be used to delete a directory. The *Dir.unlink* and *Dir.rmdir* perform exactly the same function and are provided for convenience.

```
Dir.delete("testdir")
```

Creating Files & Temporary Directories:

Temporary files are those that might be created briefly during a program's execution but aren't a permanent store of information.

Dir.tmpdir provides the path to the temporary directory on the current system, although the method is not available by default. To make *Dir.tmpdir* available it's necessary to use require 'tmpdir'.

You can use *Dir.tmpdir* with *File.join* to create a platform-independent temporary file:

```
require 'tmpdir'
   tempfilename = File.join(Dir.tmpdir, "tingtong")
   tempfile = File.new(tempfilename, "w")
   tempfile.puts "This is a temporary file"
   tempfile.close
   File.delete(tempfilename)
```

This code creates a temporary file, writes data to it, and deletes it. Ruby's standard library also includes a library called *Tempfile* that can create temporary files for you:

```
require 'tempfile'
   f = Tempfile.new('tingtong')
   f.puts "Hello"
   puts f.path
   f.close
```

Built-in Functions:

Here is the complete list of ruby buil-in functions to process files and directories:

- File Class and Methods.
- Dir Class and Methods.

Ruby Exceptions

The execution and the exception always go together. If you are opening a file which does not exist then if you did not handle this situation properly then your program is considered to be of bad quality.

The program stops if an exception occurs. So exceptions are used to handle various type of errors which may occur during a program execution and take appropriate action instead of halting program completely.

Ruby provide a nice mechanism to handle exceptions. We enclose the code that could raise an exception in a *begin/end* block and use *rescue* clauses to tell Ruby the types of exceptions we want to handle.

Syntax :

```
begin
# -
rescue OneTypeOfException
# -
rescue AnotherTypeOfException
# -
else
# Other exceptions
ensure
# Always will be executed
end
```

Everything from *begin* to *rescue* is protected. If an exception occurs during the execution of this block of code, control is passed to the block between *rescue* and *end*.

For each *rescue* clause in the *begin* block, Ruby compares the raised Exception against each of the parameters in turn. The match will succeed if the exception named in the rescue clause is the same as the type of the currently thrown exception, or is a superclass of that exception.

In an event that an exception does not match any of the error types specified, we are allowed to use an *else* clause after all the *rescue* clauses.

Example:

```
#!/usr/bin/ruby

begin
   file = open("/unexistant_file")
   if file
      puts "File opened successfully"
   end
rescue
      file = STDIN
end
print file, "==", STDIN, "\n"
```

This will produce following result. You can see that *STDIN* is substituted to *file* because *open* failed.

```
#<IO:0xb7d16f84>==#<IO:0xb7d16f84>
```

Using *retry* Statement:

You can capture an exception using *rescue* block and then use *retry* statement to execute *begin* block from the beginning.

Syntax:

```
begin
    # Exceptions raised by this code will
        # be caught by the following rescue clause
rescue
    # This block will capture all types of exceptions
    retry  # This will move control to the beginning of begin
end
```

Example:

```
#!/usr/bin/ruby

begin
   file = open("/unexistant_file")
   if file
      puts "File opened successfully"
   end
rescue
   fname = "existant_file"
   retry
end
```

The following is the flow of the process:

- an exception occurred at open
- went to rescue. fname was re-assigned
- by retry went to the beginning of the begin
- this time file opens successfully
- continued the essential process.

NOTE: Notice that if the file of re-substituted name does not exist this example code retries infinitely. Be careful if you use *retry* for an exception process.

Using *raise* Statement:

You can use *raise* statement to raise an exception. The following method raises an exception whenever it's called. It's second message will never be printed. Program

Syntax:

```
raise

OR

raise "Error Message"

OR

raise ExceptionType, "Error Message"

OR

raise ExceptionType, "Error Message" condition
```

The first form simply reraises the current exception (or a RuntimeError if there is no current exception). This is used in exception handlers that need to intercept an exception before passing it on.

The second form creates a new *RuntimeError* exception, setting its message to the given string. This exception is then raised up the call stack.

The third form uses the first argument to create an exception and then sets the associated message to the second argument.

The third form is similar to third form but you can add any conditional statement like *unless* to raise an exception.

Example:

```
#!/usr/bin/ruby

begin
    puts 'I am before the raise.'
    raise 'An error has occurred.'
    puts 'I am after the raise.'
rescue
    puts 'I am rescued.'
end
puts 'I am after the begin block.'
```

This will produce following result:

```
I am before the raise.
I am rescued.
I am after the begin block.
```

One more example showing usage of *raise*:

```
#!/usr/bin/ruby

begin
  raise 'A test exception.'
rescue Exception => e
  puts e.message
  puts e.backtrace.inspect
end
```

This will produce following result:

```
A test exception.
["test.rb:4"]
```

Using *ensure* Statement:

Sometimes you need to guarantee that some processing is done at the end of a block of code, regardless of whether an exception was raised. For example, you may have a file open on entry to the block, and you need to make sure it gets closed as the block exits.

The *ensure* clause does just this. ensure goes after the last rescue clause and contains a chunk of code that will always be executed as the block terminates. It doesn't matter if the block exits normally, if it raises and rescues an exception, or if it is terminated by an uncaught exception . the *ensure* block will get run.

Syntax:

```
begin
   #.. process
   #..raise exception
rescue
```

```
   #.. handle error
ensure
   #.. finally ensure execution
   #.. This will always execute.
end
```

Example:

```
begin
  raise 'A test exception.'
rescue Exception => e
  puts e.message
  puts e.backtrace.inspect
ensure
  puts "Ensuring execution"
end
```

This will produce following result:

```
A test exception.
["test.rb:4"]
Ensuring execution
```

Using *else* Statement:

If the *else* clause is present, it goes after the *rescue* clauses and before any *ensure*.

The body of an *else* clause is executed only if no exceptions are raised by the main body of code.

Syntax:

```
begin
   #.. process
   #..raise exception
rescue
   # .. handle error
else
   #.. executes if there is no exception
ensure
   #.. finally ensure execution
   #.. This will always execute.
end
```

Example:

```
begin
 # raise 'A test exception.'
 puts "I'm not raising exception"
rescue Exception => e
  puts e.message
  puts e.backtrace.inspect
else
   puts "Congratulations-- no errors!"
ensure
  puts "Ensuring execution"
```

```
end
```

This will produce following result:

```
I'm not raising exception
Congratulations-- no errors!
Ensuring execution
```

Raised error message can be captured using $! variable.

Catch and Throw:

While the exception mechanism of raise and rescue is great for abandoning execution when things go wrong, it's sometimes nice to be able to jump out of some deeply nested construct during normal processing. This is where catch and throw come in handy.

The *catch* defines a block that is labeled with the given name (which may be a Symbol or a String). The block is executed normally until a throw is encountered.

Syntax:

```
throw :lablename
#.. this will not be executed
catch :lablename do
#.. matching catch will be executed after a throw is encountered.
end

OR

throw :lablename condition
#.. this will not be executed
catch :lablename do
#.. matching catch will be executed after a throw is encountered.
end
```

Example:

The following example uses a throw to terminate interaction with the user if '!' is typed in response to any prompt.

```
def promptAndGet(prompt)
   print prompt
   res = readline.chomp
   throw :quitRequested if res == "!"
   return res
end

catch :quitRequested do
   name = promptAndGet("Name: ")
   age = promptAndGet("Age: ")
   sex = promptAndGet("Sex: ")
   # ..
   # process information
end
promptAndGet("Name:")
```

This will produce following result:

```
Name: Ruby on Rails
Age: 3
Sex: !
Name:Just Ruby
```

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
```

Now look at the following example which will use this exception:

```
File.open(path, "w") do |file|
begin
    # Write out the data ...
rescue
    # Something went wrong!
    raise FileSaveError.new($!)
end
end
```

The important line here is raise *FileSaveError.new($!)*. We call raise to signal that an exception has occurred, passing it a new instance of FileSaveError, with the reason being that specific exception caused the writing of the data to fail.

Ruby/DBI Tutorial

This session will teach you how to access a database using Ruby. The *Ruby DBI* module provides a database-independent interface for Ruby scripts similar to that of the Perl DBI module.

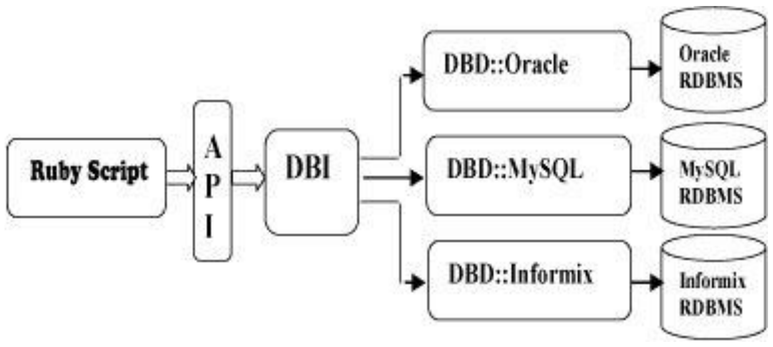
DBI stands for Database independent interface for Ruby which means DBI provides an abstraction layer between the Ruby code and the underlying database, allowing you to switch database implementations really easily. It defines a set of methods, variables, and conventions that provide a consistent database interface, independent of the actual database being used.

DBI can interface with the following:

- ADO (ActiveX Data Objects)
- DB2
- Frontbase
- mSQL
- MySQL
- ODBC
- Oracle
- OCI8 (Oracle)
- PostgreSQL
- Proxy/Server
- SQLite
- SQLRelay

Architecture of a DBI Application

DBI is independent of any database available in backend. You can use DBI whether you are working with Oracle, MySQL or Informix etc. This is clear from the following architure diagram.



The general architecture for Ruby DBI uses two layers:

- The database interface (DBI) layer. This layer is database independent and provides a set of common access methods that are used the same way regardless of the type of database server with which you're communicating.
- The database driver (DBD) layer. This layer is database dependent; different drivers provide access to different database engines. There is one driver for MySQL, another for PostgreSQL, another for InterBase, another for Oracle, and so forth. Each driver interprets requests from the DBI layer and maps them onto requests appropriate for a given type of database server.

Prerequisites:

If you want to write Ruby scripts to access MySQL databases, you'll need to have the Ruby MySQL module installed.

This module acts as a DBD as explained above and can be downloaded from http://www.tmtm.org/en/mysql/ruby/

Obtaining and Installing Ruby/DBI:

You can download and install the Ruby DBI module from the following location:

```
http://rubyforge.org/projects/ruby-dbi/
```

Before starting this installation make sure you have root privilege. Now following the following steps:

Step 1

Unpacked the downloaded file using the following command:

```
$ tar zxf dbi-0.2.0.tar.gz
```

Step 2

Go in distrubution directory *dbi-0.2.0* and configure it using the *setup.rb* script in that directory. The most general configuration command looks like this, with no arguments following the config argument. This command configures the distribution to install all drivers by default.

```
$ ruby setup.rb config
```

To be more specific, provide a --with option that lists the particular parts of the distribution you want to use. For example, to configure only the main DBI module and the MySQL DBD-level driver, issue the following command:

```
$ ruby setup.rb config --with=dbi,dbd_mysql
```

Step 3

Final step is to build the driver and install it using the following commands.

```
$ ruby setup.rb setup $ ruby setup.rb install
```

Database Connection:

Assuming we are going to work with MySQL database. Before connecting to a database make sure followings:

- You have created a database TESTDB.
- You have created EMPLOYEE in TESTDB.
- This table is having fields FIRST_NAME, LAST_NAME, AGE, SEX and INCOME.
- User ID "testuser" and password "test123" are set to access TESTDB
- Ruby Module DBI is installed properly on your machine.
- You have gone through MySQL tutorial to understand MySQL Basics.

Following is the example of connecting with MySQL database "TESTDB"

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

require "dbi"

begin
     # connect to the MySQL server
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                            "testuser", "test123")
     # get server version string and display it
     row = dbh.select_one("SELECT VERSION()")
     puts "Server version: " + row[0]
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
ensure
     # disconnect from server
     dbh.disconnect if dbh
end
```

While running this script, its producing following result at my Linux machine.

```
Server version: 5.0.45
```

If a connection is established with the datasource then a Database Handle is returned and saved into **dbh** for further use otherwise **dbh** is set to nill value and *e.err* and *e::errstr* return error code and an error string respectively.

Finally before coming out it ensures that database connection is closed and resources are released.

INSERT Operation:

INSERT operation is required when you want to create your records into a database table.

Once a database connection is established, we are ready to create tables or records into the database tables using **do** method or **prepare** and **execute** method.

Using do Statement:

Statements that do not return rows can be issued by invoking the **do** database handle method. This method takes a statement string argument and returns a count of the number of rows affected by the statement.

```
dbh.do("DROP TABLE IF EXISTS EMPLOYEE")
dbh.do("CREATE TABLE EMPLOYEE (
     FIRST_NAME  CHAR(20) NOT NULL,
     LAST_NAME  CHAR(20),
     AGE INT,
     SEX CHAR(1),
     INCOME FLOAT )" );
```

Similar way you can execute SQL *INSERT* statement to create a record into EMPLOYEE table.

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

require "dbi"

begin
     # connect to the MySQL server
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                            "testuser", "test123")
     dbh.do( "INSERT INTO EMPLOYEE(FIRST_NAME,
                   LAST_NAME,
                   AGE,
                   SEX,
                   INCOME)
          VALUES ('Mac', 'Mohan', 20, 'M', 2000)" )
     puts "Record has been created"
     dbh.commit
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
     dbh.rollback
ensure
     # disconnect from server
     dbh.disconnect if dbh
end
```

Using *prepare* and *execute*:

You can use *prepare* and *execute* methods of DBI class to execute SQL statement through Ruby code.

Record creation takes following steps

- Prearing SQL statement with INSERT statement. This will be done using **prepare** method.
- Executing SQL query to select all the results from the database. This will be done using **execute** method.
- Releasing Stattement handle. This will be done using **finish** API
- If everything goes fine then **commit** this operation otherwise you can **rollback** complete transaction.

Following is the syntax to use these two methods:

```
sth = dbh.prepare(statement)
sth.execute
   ... zero or more SQL operations ...
sth.finish
```

These two methods can be used to pass **bind** values to SQL statements. There may be a case when values to be entered is not given in advance. In such case binding values are used. A question mark (**?**) is used in place of actual value and then actual values are passed through execute() API.

Following is the example to create two records in EMPLOYEE table.

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

```
require "dbi"

begin
     # connect to the MySQL server
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                            "testuser", "test123")
     sth = dbh.prepare( "INSERT INTO EMPLOYEE(FIRST_NAME,
                   LAST_NAME,
                   AGE,
                   SEX,
                   INCOME)
                   VALUES (?, ?, ?, ?, ?)" )
     sth.execute('John', 'Poul', 25, 'M', 2300)
     sth.execute('Zara', 'Ali', 17, 'F', 1000)
     sth.finish
     dbh.commit
     puts "Record has been created"
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
     dbh.rollback
ensure
     # disconnect from server
     dbh.disconnect if dbh
end
```

If there are multiple INSERTs at a time then preparing a statement first and then executing it multiple times within a loop is more efficient than invoking do each time through the loop

READ Operation:

READ Operation on any databasse means to fetch some useful information from the database.

Once our database connection is established, we are ready to make a query into this database. We can use either **do** method or **prepare** and **execute** methods to fetech values from a database table.

Record fetching takes following steps

- Prearing SQL query based on required conditions. This will be done using **prepare** method.
- Executing SQL query to select all the results from the database. This will be done using **execute** method.
- Fetching all the results one by one and printing those results. This will be done using **fetch** method.
- Releasing Stattement handle. This will be done using **finish** method.

Following is the procedure to query all the records from EMPLOYEE table having salary more than 1000.

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

require "dbi"

begin
     # connect to the MySQL server
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                            "testuser", "test123")
     sth = dbh.prepare("SELECT * FROM EMPLOYEE
                        WHERE INCOME > ?")
     sth.execute(1000)

     sth.fetch do |row|
        printf "First Name: %s, Last Name : %s\n", row[0], row[1]
        printf "Age: %d, Sex : %s\n", row[2], row[3]
        printf "Salary :%d \n\n", row[4]
     end
     sth.finish
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
ensure
     # disconnect from server
     dbh.disconnect if dbh
end
```

This will produce following result:

```
First Name: Mac, Last Name : Mohan
Age: 20, Sex : M
Salary :2000

First Name: John, Last Name : Poul
Age: 25, Sex : M
Salary :2300
```

There are more shot cut methods to fecth records from the database. If you are interested then go through Fetching the Result otherwise proceed to next section.

Update Operation:

UPDATE Operation on any databasse means to update one or more records which are already available in the database. Following is the procedure to update all the records having SEX as 'M'. Here we will increase AGE of all the males by one year. This will take three steps

- Prearing SQL query based on required conditions. This will be done using **prepare** method.
- Executing SQL query to select all the results from the database. This will be done using **execute** method.
- Releasing Stattement handle. This will be done using **finish** method.
- If everything goes fine then **commit** this operation otherwise you can **rollback** complete transaction.

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

require "dbi"

begin
     # connect to the MySQL server
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                            "testuser", "test123")
     sth = dbh.prepare("UPDATE EMPLOYEE SET AGE = AGE + 1
                        WHERE SEX = ?")
     sth.execute('M')
     sth.finish
     dbh.commit
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
     dbh.rollback
ensure
     # disconnect from server
     dbh.disconnect if dbh
end
```

DELETE Operation:

DELETE operation is required when you want to delete some records from your database. Following is the procedure to delete all the records from EMPLOYEE where AGE is more than 20. This operation will take following steps.

- Prearing SQL query based on required conditions. This will be done using **prepare** method.
- Executing SQL query to delete required records from the database. This will be done using **execute** method.
- Releasing Stattement handle. This will be done using **finish** method.
- If everything goes fine then **commit** this operation otherwise you can **rollback** complete transaction.

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

require "dbi"

begin
     # connect to the MySQL server
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                       "testuser", "test123")
     sth = dbh.prepare("DELETE FROM EMPLOYEE
                        WHERE AGE > ?")
     sth.execute(20)
     sth.finish
     dbh.commit
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
     dbh.rollback
ensure
     # disconnect from server
     dbh.disconnect if dbh
end
```

Performing Transactions:

Transactions are a mechanism that ensures data consistency. Transactions should have the following four properties:

- **Atomicity:** Either a transaction completes or nothing happens at all.

- **Consistency:** A transaction must start in a consistent state and leave the system is a consistent state.
- **Isolation:** Intermediate results of a transaction are not visible outside the current transaction.
- **Durability:** Once a transaction was committed, the effects are persistent, even after a system failure.

The DBI provides two methods to either *commit* or *rollback* a transaction. There is one more method called *transaction* which can be used to implement transactions. There are two simple approaches to implement transactions:

Approach I:

The first approach uses DBI's *commit* and *rollback* methods to explicitly commit or cancel the transaction:

```
dbh['AutoCommit'] = false # Set auto commit to false.
begin
  dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
          WHERE FIRST_NAME = 'John'")
  dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
          WHERE FIRST_NAME = 'Zara'")
  dbh.commit
rescue
  puts "transaction failed"
  dbh.rollback
end
dbh['AutoCommit'] = true
```

Approach II:

The second approach uses the *transaction* method. This is simpler, because it takes a code block containing the statements that make up the transaction. The *transaction* method executes the block, then invokes *commit* or *rollback* automatically, depending on whether the block succeeds or fails:

```
dbh['AutoCommit'] = false # Set auto commit to false.
dbh.transaction do |dbh|
  dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
          WHERE FIRST_NAME = 'John'")
  dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
          WHERE FIRST_NAME = 'Zara'")
end
dbh['AutoCommit'] = true
```

COMMIT Operation:

Commit is the operation which gives a green signal to database to finalize the changes and after this operation no change can be reverted back.

Here is a simple example to call **commit** method.

```
  dbh.commit
```

ROLLBACK Operation:

If you are not satisfied with one or more of the changes and you want to revert back those changes completely then use **rollback** method.

Here is a simple example to call **rollback** metho.

```
     dbh.rollback
```

Disconnecting Database:

To disconnect Database connection, use disconnect API.

```
    dbh.disconnect
```

If the connection to a database is closed by the user with the disconnect method, any outstanding transactions are rolled back by the DBI. However, instead of depending on any of DBI's implementation details, your application would be better off calling commit or rollback explicitly.

Handling Errors:

There are many sources of errors. A few examples are a syntax error in an executed SQL statement, a connection failure, or calling the fetch method for an already canceled or finished statement handle.

If a DBI method fails, DBI raises an exception. DBI methods may raise any of several types of exception but the two most important exception classes are *DBI::InterfaceError* and *DBI::DatabaseError*.

Exception objects of these classes have three attributes named *err*, *errstr*, and *state*, which represent the error number, a descriptive error string, and a standard error code. The attributes are explained below:

- **err:** Returns an integer representation of the occurred error or *nil* if this is not supported by the DBD.The Oracle DBD for example returns the numerical part of an *ORA-XXXX* error message.
- **errstr:** Returns a string representation of the occurred error.
- **state:** Returns the SQLSTATE code of the occurred error.The SQLSTATE is a five-character-long string. Most DBDs do not support this and return nil instead.

You have seen following code above in most of the examples:

```
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
     dbh.rollback
ensure
     # disconnect from server
     dbh.disconnect if dbh
end
```

To get debugging information about what your script is doing as it executes, you can enable tracing. To do this, you must first load the dbi/trace module and then call the *trace* method that controls the trace mode and output destination:

```
require "dbi/trace"
..............

trace(mode, destination)
```

The mode value may be 0 (off), 1, 2, or 3, and the destination should be an IO object. The default values are 2 and STDERR, respectively.

Code Blocks with Methods

There are some methods which creates handles. These methods can be invoked with a code block. The advantage of using code block along with methods is that they provide the handle to the code block as its parameter and automatically clean up the handle when the block terminates. There are few examples to understand the concept

- **DBI.connect :** This method generates a database handle and it is recommended to call *disconnect* at the end of the block to disconnect the database.
- **dbh.prepare :** This method generates a statement handle and it is recommended to *finish* at the end of the block. Within the block, you must invoke *execute* method to execute the statement.
- **dbh.execute :** This method is similar except we don't need to invoke execute within the block. The statement handle is automatically executed.

Example 1:

DBI.connect can take a code block, passes the database handle to it, and automatically disconnects the handle at the end of the block as follows.

```
dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                       "testuser", "test123") do |dbh|
```

Example 2:

dbh.prepare can take a code block, passes the statement handle to it, and automatically calls finish at the end of the block as follows.

```
dbh.prepare("SHOW DATABASES") do |sth|
       sth.execute
       puts "Databases: " + sth.fetch_all.join(", ")
end
```

Example 3:

dbh.execute can take a code block, passes the statement handle to it, and automatically calls finish at the end of the block as follows:

```
dbh.execute("SHOW DATABASES") do |sth|
   puts "Databases: " + sth.fetch_all.join(", ")
end
```

DBI *transaction* method also takes a code block which has been described in above.

Driver-specific Functions and Attributes:

The DBI lets database drivers provide additional database-specific functions, which can be called by the user through the *func* method of any Handle object.

Driver-specific attributes are supported and can be set or gotten using the **[]=** or **[]** methods.

DBD::Mysql implements the following driver-specific functions:

S.N.	Functions with Description
1	**dbh.func(:createdb, db_name)** Creates a new database
2	**dbh.func(:dropdb, db_name)** Drops a database
3	**dbh.func(:reload)** Performs a reload operation
4	**dbh.func(:shutdown)** Shut down the server
5	**dbh.func(:insert_id) => Fixnum** Returns the most recent AUTO_INCREMENT value for a connection.
6	**dbh.func(:client_info) => String** Returns MySQL client information in terms of version.
7	**dbh.func(:client_version) => Fixnum** Returns client information in terms of version. Its similar to :client_info but it return a fixnum instead of sting.
8	**dbh.func(:host_info) => String** Returns host information
9	**dbh.func(:proto_info) => Fixnum** Returns protocol being used for the communication
10	**dbh.func(:server_info) => String** Returns MySQL server information in terms of version.
11	**dbh.func(:stat) => String** Returns current stat of the database
12	**dbh.func(:thread_id) => Fixnum** Return current thread ID.

Example:

```
#!/usr/bin/ruby

require "dbi"
begin
   # connect to the MySQL server
   dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                             "testuser", "test123")
   puts dbh.func(:client_info)
   puts dbh.func(:client_version)
   puts dbh.func(:host_info)
   puts dbh.func(:proto_info)
   puts dbh.func(:server_info)
   puts dbh.func(:thread_id)
   puts dbh.func(:stat)
rescue DBI::DatabaseError => e
   puts "An error occurred"
   puts "Error code:    #{e.err}"
   puts "Error message: #{e.errstr}"
ensure
   dbh.disconnect if dbh
end
```

This will produce following result:

```
5.0.45
50045
Localhost via UNIX socket
10
5.0.45
150621
Uptime: 384981  Threads: 1  Questions: 1101078  Slow queries: 4 \
Opens: 324  Flush tables: 1  Open tables: 64  \
Queries per second avg: 2.860
```

Further Detail:

Refer to the link http://www.tutorialspoint.com/ruby