RUBY FILE I/O, 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 the basic I/O functions available in Ruby. For more functions, please refer 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 the 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 the 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 the 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 the 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 :

Following is the input text file:

```
This is a simple text file for testing purpose.
```

Now let's try to read thsi file:

```
#!/usr/bin/ruby

aFile = File.new("input.txt", "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("input.txt", "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("input.txt", "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:

```
s. .a. .s.i.m.p.l.e. .t.e.x.t. .f.i.l.e. .f.o.r. .t.e.s.t.i.n.g. .p.u.r.p.o.s.e...
.
.
```

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("input.txt")
puts arr[0]
puts arr[1]
```

In this code, the variable arr is an array. Each line of the file *input.txt* 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("input.txt"){|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 *notonethatalreadyexists* 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 built-in functions to process files and directories:

- File Class and Methods.
- Dir Class and Methods

Loading [MathJax]/jax/output/HTML-CSS/jax.js