RUBY VARIABLES, CONSTANTS AND LITERALS

Variables are the memory locations which hold any data to be used by any program.

There are five types of variables supported by Ruby. You already have gone through a small description of these variables in previous chapter as well. These five types of variables are explained in this chapter.

Ruby Global Variables:

Global variables begin with $. Uninitialized global variables have the value *nil* and produce warnings with the -w option.

Assignment to global variables alters global status. It is not recommended to use global variables. They make programs cryptic.

Here is an example showing usage of global variable.

```
#!/usr/bin/ruby

$global_variable = 10
class Class1
  def print_global
     puts "Global variable in Class1 is #$global_variable"
  end
end
class Class2
  def print_global
     puts "Global variable in Class2 is #$global_variable"
  end
end

class1obj = Class1.new
class1obj.print_global
class2obj = Class2.new
class2obj.print_global
```

Here $global_variable is a global variable. This will produce the following result:

NOTE: In Ruby you CAN access value of any variable or constant by putting a hash # character just before that variable or constant.

```
Global variable in Class1 is 10
Global variable in Class2 is 10
```

Ruby Instance Variables:

Instance variables begin with @. Uninitialized instance variables have the value *nil* and produce warnings with the -w option.

Here is an example showing usage of Instance Variables.

```
#!/usr/bin/ruby

class Customer
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
   end
   def display_details()
      puts "Customer id #@cust_id"
      puts "Customer name #@cust_name"
      puts "Customer address #@cust_addr"
    end
end

# Create Objects
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")

# Call Methods
cust1.display_details()
cust2.display_details()
```

Here, @cust_id, @cust_name and @cust_addr are instance variables. This will produce the following result:

```
Customer id 1
Customer name John
Customer address Wisdom Apartments, Ludhiya
Customer id 2
Customer name Poul
Customer address New Empire road, Khandala
```

Ruby Class Variables:

Class variables begin with @@ and must be initialized before they can be used in method definitions.

Referencing an uninitialized class variable produces an error. Class variables are shared among descendants of the class or module in which the class variables are defined.

Overriding class variables produce warnings with the -w option.

Here is an example showing usage of class variable:

```
#!/usr/bin/ruby

class Customer
   @@no_of_customers=0
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
   end
   def display_details()
      puts "Customer id #@cust_id"
      puts "Customer name #@cust_name"
      puts "Customer address #@cust_addr"
    end
    def total_no_of_customers()
       @@no_of_customers += 1
       puts "Total number of customers: #@@no_of_customers"
    end
end

# Create Objects
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")

# Call Methods
cust1.total_no_of_customers()
cust2.total_no_of_customers()
```

Here @@no_of_customers is a class variable. This will produce the following result:

```
Total number of customers: 1
Total number of customers: 2
```

Ruby Local Variables:

Local variables begin with a lowercase letter or _. The scope of a local variable ranges from class, module, def, or do to the corresponding end or from a block's opening brace to its close brace {}.

When an uninitialized local variable is referenced, it is interpreted as a call to a method that has no arguments.

Assignment to uninitialized local variables also serves as variable declaration. The variables start to exist until the end of the current scope is reached. The lifetime of local variables is determined when Ruby parses the program.

In the above example local variables are id, name and addr.

Ruby Constants:

Constants begin with an uppercase letter. Constants defined within a class or module can be accessed from within that class or module, and those defined outside a class or module can be accessed globally.

Constants may not be defined within methods. Referencing an uninitialized constant produces an error. Making an assignment to a constant that is already initialized produces a warning.

```
#!/usr/bin/ruby

class Example
   VAR1 = 100
   VAR2 = 200
   def show
       puts "Value of first Constant is #{VAR1}"
       puts "Value of second Constant is #{VAR2}"
   end
end

# Create Objects
object=Example.new()
object.show
```

Here VAR1 and VAR2 are constant. This will produce the following result:

```
Value of first Constant is 100
Value of second Constant is 200
```

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 Basic Literals:

The rules Ruby uses for literals are simple and intuitive. This section explains all basic Ruby Literals.

Integer Numbers:

Ruby supports integer numbers. An integer number can range from -2^{30} to 2$^{30-1}$ or -2^{62} to 2$^{62-1}$. Integers with-in this range are objects of class *Fixnum* and integers outside this range are stored in objects of class *Bignum*.

You write integers using an optional leading sign, an optional base indicator $0 for octal, 0x for hex, or 0b for binary$, followed by a string of digits in the appropriate base. Underscore characters are ignored in the digit string.

You can also get the integer value corresponding to an ASCII character or escape sequence by preceding it with a question mark.

Example:

```
123                  # Fixnum decimal
1_234                # Fixnum decimal with underline
-500                 # Negative Fixnum
0377                 # octal
0xff                 # hexadecimal
0b1011               # binary
?a                   # character code for 'a'
?\n                  # code for a newline (0x0a)
12345678901234567890 # Bignum
```

NOTE: Class and Objects are explained in a separate chapter of this tutorial.

Floating Numbers:

Ruby supports integer numbers. They are also numbers but with decimals. Floating-point numbers are objects of class *Float* and can be any of the following:

Example:

```
123.4                # 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. Double-quoted strings allow substitution and backslash notation but single-quoted strings don't allow substitution and allow backslash notation only for \\ and \'

Example:

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

puts 'escape using "\\"';
puts 'That\'s right';
```

This will produce the following result:

```
escape using "\"
That's right
```

You can substitute the value of any Ruby expression into a string using the sequence **#{ expr }**. Here, expr could be any ruby expression.

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

puts "Multiplication Value : #{24*60*60}";
```

This will produce the following result:

```
Multiplication Value : 86400
```

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 $nbeing0 - 7$
\xnn	Hexadecimal notation $nbeing0 - 9, a - f, orA - F$
\cx, \C-x	Control-x
\M-x	Meta-x $c \| 0x80$
\M-\C-x	Meta-Control-x
\x	Character x

For more detail on Ruby Strings, go through Ruby Strings.

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 = [  "fred", 10, 3.14, "This is a string", "last element", ]
ary.each do |i|
   puts i
end
```

This will produce the following result:

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

For more detail on Ruby Arrays, go through Ruby Arrays.

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, "blue" => 0x00f }
hsh.each do |key, value|
   print key, " is ", value, "\n"
end
```

This will produce the following result:

```
green is 240
red is 3840
blue is 15
```

For more detail on Ruby Hashes, go through Ruby Hashes.

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 1, 2, 3, 4 values.

Example:

```
#!/usr/bin/ruby

(10..15).each do |n|
   print n, ' '
end
```

This will produce the following result:

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

For more detail on Ruby Ranges, go through Ruby Ranges.

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