Introduction to Ruby

Ruby is "A Programmer's Best Friend".

Ruby is a pure object-oriented programming language. It was created in 1993 by Yukihiro Matsumoto of Japan. Matsumoto is also known as ―Matz‖ in the Ruby community.

Features of ruby:

- Ruby is an open-source and is freely available on the Web, but it is subject to a license.
- Ruby is a general-purpose, interpreted programming language.
- Ruby is a true object-oriented programming language.
- Ruby is a server-side scripting language similar to Python and PERL.
- Ruby can be used to write Common Gateway Interface (CGI) scripts.
- Ruby can be embedded into Hypertext Markup Language (HTML).
- Ruby has a clean and easy syntax that allows a new developer to learn very quickly and easily.
- Ruby has similar syntax to that of many programming languages such as C++ and Perl.
- Ruby is very much scalable and big programs written in Ruby are easily maintainable.
- Ruby can be used for developing Internet and intranet applications.
- Ruby can be installed in Windows and POSIX environments.
- Ruby support many GUI tools such as Tcl/Tk, GTK, and OpenGL.
- Ruby can easily be connected to DB2, MySQL, Oracle, and Sybase.
- Ruby has a rich set of built-in functions, which can be used directly into Ruby scripts.

Scalar Types and Their Operations

Ruby has three categories of data types:

1. Scalars
2. Arrays, and
3. Hashes.

The most commonly used types: scalars. There are two categories of scalar types: numeric‘s and character strings.

Numeric and String Literals

All numeric data types in Ruby are descendants of the Numeric class. The immediate child classes of Numeric are Float and Integer. The Integer class has two child classes: Fixnum and Bignum. An integer literal that fits into the range of a machine word, which is often 32 bits, is a Fixnum object. An integer literal that is outside the Fixnum range is a Bignum object. There is no length limitation on integer literals. If a Fixnum integer grows beyond the size limitation of Fixnum objects, it is coerced to a Bignum object. Likewise, if an operation on a Bignum object results in a value that fits into a Fixnum object, it is coerced to a Fixnum type.

Underscore characters can appear embedded in integer literals. Ruby ignores such underscores, allowing large numbers to be slightly more readable. For example, instead of 124761325, 124_761_325 can be used.

A numeric literal that has either an embedded decimal point or a following exponent is a Float object, which is stored as the underlying machine‘s double-precision floating-point type. The decimal point must be embedded; that is, it must be both preceded and followed by at least one digit. So, .435 is not a legal literal in Ruby.

All string literals are String objects, which are sequences of bytes that represent characters. There are two categories of string literals:

- single quoted
- double quoted.

Single-quoted string literals cannot include characters specified with escape sequences, such as newline characters specified with \n. If an actual single-quote character is needed in a string literal that is delimited by single quotes, the embedded single quote is preceded by a backslash, as in the following example:

```
_I\‘ll meet you at O\‘Malleys‘
```

If an escape sequence is embedded in a single-quoted string literal, each character in the sequence is taken literally as itself. For example, the sequence \n in the following string literal will be treated as two characters—a backslash and an n:

```
„Some apples are red, \n some are green"
```

Double-quoted string literals differ from single-quoted string literals in two ways: First, they can include special characters specified with escape sequences; second, the values of variable names can be interpolated into the string, which means that their values are substituted for their names.

In many situations, special characters that are specified with escape sequences must be included in string literals. For example, if the words on a line must be spaced by tabs, a double-quoted literal with embedded escape sequences for the tab character can be used as in the following string:

"Runs \t Hits \t Errors"

A double quote can be embedded in a double-quoted string literal by preceding it with a backslash. The null string (the string with no characters) can be denoted with either — or —‖.

Variables and Assignment Statements

- The form of variable names is a lowercase letter or an underscore, followed by any number of uppercase or lowercase letters, digits, or underscores.
- The letters in a variable name are case sensitive, meaning that fRIZZY, frizzy, frIzZy, and friZZy are all distinct names.
- Programmer-defined variable names do not include uppercase letters.
- double-quoted string literals can include the values of variables specified by placing the code in braces and preceding the left brace with a pound sign (#).

For example, if the value of *tue_high* is *83*, then the string *"Tuesday"s high temperature was #{tue_high}"*
has the following value:

"Tuesday"s high temperature was 83"

A scalar variable that has not been assigned a value by the program has the value nil.

Ruby Constants

Ruby has constants, which are distinguished from variables by their names, which always begin with uppercase letters. A constant is created when it is assigned a value, which can be any constant expression. In Ruby, a constant can be assigned a new value, although it causes a warning message to the user. Ruby includes some predefined, or implicit, variables. The name of an implicit scalar variable begins with a dollar sign. The rest of the name is often just one more special character, such as an underscore (_), a circumflex (^), or a backslash (\).
Example:

```
VAR1=1
00
VAR2=2
00
```

Ruby Pseudo-Variables

They are special variables that have the appearance of local variables but behave like constants. You cannot 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.

Numeric Operators

Most of Ruby‘s numeric operators are similar to those in other common programming languages, so they should be familiar to most readers. There are the binary operators: + for addition, - for subtraction, * for multiplication, / for division, ** for exponentiation, and % for modulus. The modulus operator is defined as follows: x % y produces the remainder of

the value of x after division by y. If an integer is divided by an integer, integer division is done. Therefore, 3 / 2 produces 1.

Operator	Associativity
**	Right
unary +, –	Right
*, /, %	Left
binary +, –	Left

The operators listed first have the highest precedence.

- Ruby does not include the increment (++) and decrement (--) operators.

Ruby includes the Math module, which has methods for basic trigonometric and transcendental functions. Among these methods are cos (cosine), sin (sine), log (logarithm), sqrt (square root), and tan (tangent). The methods of the Math module are referenced by prefixing their names with Math., as in Math.sin(x). All of these take any numeric type as a parameter and return a Float value.

Included with the Ruby implementation is an interactive interpreter, which is very useful to the student of Ruby. It allows one to type any Ruby expression and get an immediate response from the interpreter. The interactive interpreter‘s name is Interactive Ruby, whose acronym, IRB, is the name of the program that supports it. For example, if the command prompt is a percent sign (%), one can type % irb

after which irb will respond with its own prompt, which is

irb(main):001:0>

At this prompt, any Ruby expression or statement can be typed, whereupon irb interprets the expression or statement and returns the value after an implication symbol (=>), as in the following example:

*irb(main):001:0> 17 * 3*

=> 51

irb(main):002:0>

The lengthy default prompt can be easily changed. We prefer the simple —>>— prompt. The default prompt can be changed to this with the following command:

irb(main):002:0> conf.prompt_i = “>>“

From here on, we will use this simple prompt.

String Methods

The Ruby String class has more than 75 methods. Many of these methods can be used as if they were operators. In fact, we sometimes call them operators, even though underneath they are all methods.

The String method for catenation is specified by plus (+), which can be used as a binary

```
>>"Happy"+" "+"Holidays!"
=> "Happy Holidays!"
```

operator. This method creates a new string from its operands:

The << method appends a string to the right end of another string, which, of course, makes sense only if the left operand is a variable. Like +, the << method can be used as a binary operator. For example, in the interactions

```
>>mystr="G'day, "
=> "G'day, "
>>mystr << "Friend"
=> "G'day, Friend"
```

The other most commonly used methods of Ruby are similar to those of other programming languages. Among these are the ones shown in Table below; all of them create new strings. As

Method	Action
capitalize	Converts the first letter to uppercase and the rest of the letters to lowercase
chop	Removes the last character
chomp	Removes a newline from the right end if there is one
upcase	Converts all of the lowercase letters in the object to uppercase
downcase	Converts all of the uppercase letters in the object to lowercase
strip	Removes the spaces on both ends
lstrip	Removes the spaces on the left end
rstrip	Removes the spaces on the right end
reverse	Reverses the characters of the string
swapcase	Converts all uppercase letters to lowercase and all lowercase letters to uppercase

As stated previously, all of these methods produce new strings, rather than modify the given string in place. However, all of the methods also have versions that modify their objects in place. These methods are called bang or mutator methods and are specified by following their names with an exclamation point (!). To illustrate the difference between a string method and its bang counterpart, consider the following interactions:

```
>>str="Welcome"
=> "Welcome"
>>str.upcase
=> "WELCOME"
>>str
=> "Welcome"
>>str.upcase!
=> "WELCOME"
>>str
=> "WELCOME"
```

Ruby strings can be indexed, somewhat as if they were arrays. To get the character

```
>>str="Welcome"
=> "Welcome"
>>str[0]
=> "W"
>>str[-1]
=> "e"
```

:

Note: If a negative subscript is used as an index, the position is counted from the right.

A multi-character substring of a string can be accessed by including two numbers in the brackets, in which case the first is the position of the first character of the substring and the second is the number of characters in the substring:

```
>>str="Welcome"
=> "Welcome"
>>str[3,4]
=> "come"
```

The usual way to compare strings for equality is to use the == method as an operator:

```
>>"abc" == "abc"
=> true
>>"abc" == "abd"
=> false
```

To facilitate ordering, Ruby includes the ―spaceship‖ operator, <=>, which returns -1 if the second operand is greater than the first, 0 if the two operands are equal, and 1 if the first operand is greater than the second.

Greater‖ in this case means that the text in question belongs later alphabetically. The following interactions illustrate all three cases:

```
>>"apple" <=> "banana"
=> -1
>>"banana" <=> "apple"
=> 1
>>"grape" <=> "grape"
=> 0
```

The repetition operator is specified with an asterisk (*). It takes a string as its left operand and an expression that evaluates to a number as its right operand. The left operand is replicated the number of times equal to the value of the right operand:

```
>>"cse! "*3
=> "cse! cse! cse! "
```

Simple Input and Output Screen Output (*puts*)

Output is directed to the screen with the *puts* method (or operator). The operand for puts is a string. A newline character is implicitly appended to the string. If the value of a variable is to

```
>>name="Ruby"
=> "Ruby"
>>puts "My name is #{name}"
My name is Ruby
=> nil
```

be part of a line of output, the #{...} notation can be used to insert it into a double-quoted string, as in the following interactions:

The value returned by puts is nil, and that is the value returned after the string has been displayed. Use *print*
method if you do not want to append a newline at the end of your string.

Keyboard Input (*gets*)

The *gets* method gets a line of input from the keyboard. The retrieved line includes the newline character. If the newline is not needed, it can be discarded with *chomp*:

```
>>name= gets
apples
=> "apples\n"
```

This code could be done by applying chomp directly to the value returned by gets:

```
>>name= gets.chomp
apples
=> "apples"
```

If a number is to be input from the keyboard, the string from gets must be converted to an integer with the *to_i*
method, as in the following interactions:

```
>>age=gets.to_i
27
=> 27
```

If the number is a floating-point value, the conversion method is *to_f*:

```
>>age=gets.to_f
27
=> 27.0
```

In this same way, *to_s* which converts the value of the object to a string.

```
>>age=gets.to_s
27
=> "27\n"
```

The following program created with a text editor and stored in a file with *.rb* extension:

```
## quadeqn.rb - A simple Ruby
program ## Get input
print "Enter the value
```

```
of a:" a = gets.to_i
print "Enter the value
of b:" b = gets.to_i
print "Enter the value
of c:" c = gets.to_i
print "Enter the value
of x:" x = gets.to_i
res = a * x ** 2 + b * x + c
puts "The value of the expression is: #{res}
```

A program stored in a file can be run by the command

>ruby –w filename

So, our example program can be run (interpreted) with

```
>ruby -w quadeqn.rb
Enter the value of a:1
Enter the value of b:2
Enter the value of c:3
Enter the value of x:1
The value of the expression is: 6
```

If the program is found to be syntactically correct, the response to the following command is:

```
>ruby -cw quadeqn.rb
Syntax OK
```

Arrays

Ruby includes two structured classes or types: arrays and hashes.

Arrays in Ruby are more flexible than those of most of the other common languages. This flexibility is a result of two fundamental differences between Ruby arrays and those of other common languages such as C, C++, and Java. First, the length of a Ruby array is dynamic: It can grow or shrink anytime during program execution. Second, a Ruby array can store different types of data. For example, an array may have some numeric elements, some string elements, and even some array elements.

Ruby arrays can be created in two different ways. First, an array can be created by sending the new message to the predefined Array class, including a parameter for the size of the array. The second way is simply to assign a list literal to a variable, where a list, literal is a list of literals delimited by brackets. For example, in the following interactions, the first array is created with new and the second is created by assignment:\

```
>>list1 = Array.new(5)
=> [nil, nil, nil, nil, nil]
>>list2 = [2,2.145,"Vijay",[]]
=> [2, 2.145, "Vijay", []]
```

An array created with the new method can also be initialized by including a second parameter, but every element is given the same value. Thus, we may have the following interactions:

```
>>list = Array.new(5,"Hi")
=> ["Hi", "Hi", "Hi", "Hi", "Hi"]
```

All Ruby array elements use integers as subscripts, and the lower bound subscript of every array is zero. Array elements are referenced through subscripts delimited by brackets ([]).

The length of an array can be retrieved with the *length* method, as illustrated in the following interactions:

```
=> [2, 2.145, "Vijay", []]
>>list2.length
=> 4
```

Built-In Methods for Arrays and Lists

Ruby has four methods for adding elements at the beginning or at end of the array:

1. ***unshift( )*:** The unshift method takes a scalar or an array literal as a parameter and appends it to the beginning
2. ***shift( )*:** Removes and returns the first element of the array
3. ***push( ):*** The push method takes a scalar or an array literal and adds it to the end of the array:
4. ***pop( ):*** Removes and returns the last element of the array

```
>>list = [2,3,4]
=> [2, 3, 4]
>>list.push(5,6)
=> [2, 3, 4, 5, 6]
>>list.pop()
=> 6
>>list.unshift(10,20)
=> [10, 20, 2, 3, 4, 5]
>>list.shift()
=> 10
>>list
=> [20, 2, 3, 4, 5]
```

```
>>list
=> [20, 2, 3, 4, 5]
>>list.max
=> 20
>>list.min
=> 2
```

5. **max and min:** return the smallest or largest element in an array respectively
6. **uniq:** returns an array with no duplicate elements

```
>>list = [1,2,3,4,3,2,1]
=> [1, 2, 3, 4, 3, 2, 1]
>>list.uniq
=> [1, 2, 3, 4]
```

7. **compact** – return an array with no nil elements

```
>>list = Array.new(5)
=> [nil, nil, nil, nil, nil]
>>list[1]=10
=> 10
>>list[4]=20
=> 20
>>list[2]=30
=> 30
>>list
=> [nil, 10, 30, nil, 20]
>>list.compact
=> [10, 30, 20]
```

8. **Set operations:** There are three methods that perform set operations on two arrays: **&** for set intersection;

- for set difference, and | for set union.

```
>>list1 = [1,2,3,2,1,4]
=> [1, 2, 3, 2, 1, 4]
>>list2 = [1,2,3,5,4,2]
=> [1, 2, 3, 5, 4, 2]
>>list1 & list2
=> [1, 2, 3, 4]
>>list1 | list2
=> [1, 2, 3, 4, 5]
>>list1 - list2
=> []
```

Hashes

Associative arrays are arrays in which each data element is paired with a key, which is used to identify the data element; associative arrays often are called ***hashes***. There are two fundamental differences between arrays and hashes: First, arrays use numeric subscripts to address specific elements, whereas hashes use string values (the keys) to address elements;

second, the elements in arrays are ordered by subscript, but the elements in hashes are not.

Like arrays, hashes can be created in two ways, with the new method or by assigning a value to a variable. In the latter case, the value is a hash value, in which each element is specified by a key– value pair, separated by the symbol =>. Hash literals are delimited by braces, as in the following manner:

```
>>ages = {"raju"=>20,"king"=>21,"anand"=>19}
=> {"raju"=>20, "king"=>21, "anand"=>19}
```

If the *new* method is sent to the Hash class without a parameter, it creates an empty hash, denoted by {}:

```
>>my_hash = Hash.new
=> {}
```

An individual element of a hash can be referenced by ―subscripting‖ the hash name with a key.

```
>>ages['raju']
=> 20
```

A new value is added to a hash by assigning the value of the new element to a reference to

```
>>ages['gopi'] = 39
=> 39
>>ages
=> {"raju"=>20, "king"=>21, "anand"=>19, "gopi"=>39}
```

the key of the new element, as in the following example:

An element is removed from a hash with the delete method, which takes an element key as a parameter:

```
>>ages.delete('raju')
=> 20
>>ages
=> {"king"=>21, "anand"=>19, "gopi"=>39}
```

The keys and values of a hash can be extracted into arrays with the methods keys and values,

```
>>ages.keys
=> ["king", "anand", "gopi"]
>>ages.values
=> [21, 19, 39]
```

respectively:

The has_key? predicate method is used to determine whether an element with a specific key is in a hash.

```
>>ages
=> {"king"=>21, "anand"=>19, "gopi"=>39}
>>ages.has_key?("gopi")
=> true
>>ages.has_key?("henry")
=> false
```

A hash can be set to empty in one of two ways: either an empty hash value can be assigned to the hash, or the clear method can be used on the hash. These two approaches are illustrated with the following statements:

```
>>ages = {"raju"=>20,"king"=>21,"anand"=>19}
=> {"raju"=>20, "king"=>21, "anand"=>19}
>>ages = {}
=> {}
>>temp = {"mon"=>45,"tue"=>42,"wed"=>39}
=> {"mon"=>45, "tue"=>42, "wed"=>39}
>>temp.clear
=> {}
```

Control Statements

Selection Statements

Ruby offers conditional structures that are pretty common to modern languages. Here, we will explain all the conditional statements and modifiers available in Ruby.

Ruby‘s *if* statement is similar to that of other languages. One syntactic difference is

that there are no parentheses around the control expression, as is the case with most of the languages based directly or even loosely on C. The following construct is illustrative:

if...else
Statement
Syntax

```
        if conditional [then]
                code...
[else                                   code...]

end
```

Executes *code* if the *conditional* is true otherwise *code* specified in the *else* clause is executed.

Example:

```
a = 4 if a > 2
else end
```

it prints : 8

```
b
=
a
*
2
b
=
a
/
2
```

if...elsif..else

Statement Syntax

```
if conditional [then]
    code...
    [elsif     conditional
    [then]
    code...]...
    [else end
```

code...]

Example:

```
snowrate=2
if snowrate <= 1
        p
u
t
s
“
L
i
g
h
t
s
n
o
w
”
e
l
```

```
s                                   a
i                                   t
f                                   e
s                                   <
n                                   =
o                                   2
w                                       puts "Moderate snow"
r
        else                        puts "Heavy snow"

        end
```

prints ―Moderate snow‖

unless Statement

Ruby has an *unless* statement, which is the same as its if statement, except that the inverse of the value of the control expression is used. The following construct illustrates an *unless* statement:

Syntax

```
unless   conditional
        [then] code
[else                               code ]

end
```

Executes *code* if *conditional* is false. If the *conditional* is true, code specified in the else clause is executed. Example:

```
a = 5
unless a > 10
        puts ―A is lesser then 10‖
else
        puts ―A is greater than 10‖
```

prints: A is lesser then 10

case statement

Ruby includes two kinds of multiple selection constructs, both named case. One Ruby case construct, which is similar to a switch, has the following form:

```
case expression
when value then
 - statement sequence
...
when value then
 - statement sequence
[else
 - statement sequence]
end
```

The value of the case expression is compared with the values of the when clauses, one at a time, from top to bottom, until a match is found, at which time the sequence of statements that follow is interpreted. The comparison is done with the === relational operator, which is defined for all built-in classes. Consider the following example:

```
print "Enter the value of
in_val:" in_val = gets.to_i
case
in_val
when    -1
then
        neg_count += 1
when 0 then
        zero_count += 1
when 1 then
        pos_count += 1
else                        print "Error - in_val is out of range"

end
```

Note that no *break* statements are needed at the ends of the sequences of selectable statements in this construct.

Second form is:

```
case
when Boolean expression then expression
...
when Boolean expression then expression
else expression
end
```

```
leap = case
       when year % 400 == 0 then true
       when year % 100 == 0 then false
       else year % 4 == 0
       end
```

Looping Statements

Loops in Ruby are used to execute the same block of code a specified number of times.

The while Statement:

Executes *code* while *condition* is true. A *while* loop's *condition* is separated from *code* by the

reserved word *do*.

Syntax:

```
while   condition
        [do] code
end
```

Example

```
i = 1
while i < 5 do
        puts "Inside the loop i =
        #{$i}" i = i + 1
```

```
end
```

This will produce the following

```
result: Inside the loop i
= 1
Inside the loop i
= 2 Inside the
loop i = 3 Inside
the loop i = 4
```

while modifier

Syntax

```
begin
        code
end while condition
```

Executes *code* while *condition* is true. Irrespective of the condition, *code* is executed once before condition is evaluated.

Example

```
i = 1
begi
n
        puts "Inside the loop i =
        #{$i}" i= i + 1
end while i < 5
```

This will produce the following

```
result: Inside the loop i
= 1
Inside the loop i
= 2 Inside the
loop i = 3 Inside
the loop i = 4
```

Until Statement

Syntax:

```
until condition [do]
        code
end
```

Executes *code* while *condition* is false. An *until* statement's condition is separated from *code* by the reserved word *do*.

Example

```
i = 1
until i > 5 do
        puts "Inside the loop i =
        #{i}" i = i+1
end
```

This will produce the following

```
result: Inside the loop i
= 1
Inside the loop i
= 2 Inside the
loop i = 3 Inside
the loop i = 4
Inside the loop i
```

= 5

until

modifier

Syntax:

```
begin
        code
end until condition
```

Executes *code* while *condition* is false. If an *until* modifier follows a *begin* statement with no clauses, *code* is executed once before *conditional* is evaluated.

Example

```
i = 0
begi
n
        puts "Inside the loop i =
        #{i}" i = i+1
end until i > 5
```

This will produce the following result: Inside the loop i = 1 Inside the loop i = 2 Inside the loop i = 3 Inside the loop i = 4

```
Inside the loop i = 5
```

For Statement

Syntax

```
for variable [, variable ...] in
    expression [do] code
end
```

Executes *code* once for each element in *expression*.

Example

```
for i in 1..5
    puts "Value of local variable is #{i}"
end
```

Here, we have defined the range 1..5. The statement for *i* in 0..5 will allow *i* to take values in the range from 1 to 5 (including 5). This will produce the following result:

```
Value of local variable is 1 Value of local variable is 2 Value of local variable is 3 Value of local variable is 4 Value of local variable is 5
```

A ***for...in*** loop is almost exactly equivalent to the following: (expression).each do |variable[, variable...]|

```
    code
end
```

Example

```
(1..5).each do |i|
    puts "Value of local variable is #{i}"
end
```

This will produce the following result: Value of local variable is 1 Value of local variable is 2 Value of local variable is 3 Value of local variable is 4 Value of local variable is 5

Break Statement

Syntax

```
break
```

Terminates the loop. Terminates a method with an associated block if called within the block

Example

```
for i in 1..5
    if i > 2 then
        break
```

```
        end
        puts "Value of local variable is #{i}"
    end
```

This will produce the following result: Value of local variable is 1 Value of local variable is 2

next Statement

Syntax

```
    next
```

breaks the current iteration and place the loop in next iteration. Terminates execution of a block if called within a block

Example:

```
    for i in 1..5
        if i > 2 then
            next
        end
        puts "Value of local variable is #{i}"
    end
```

This will produce the following result: Value of local variable is 1 Value of local variable is 2

Blocks

A *block* is a sequence of code, delimited by either ***braces*** or the *do* and *end* reserved words. Blocks can be used with specially written methods to create many useful constructs, including simple *iterators* for arrays and hashes. This construct consists of a method call followed by a block. It's passed as an ―invisible‖ parameter, and executed with the *yield* statement

Usage of yield with blocks:

```
def disp
    puts "       three_times―
    yiel
    d
    yiel
    d
    yiel
    d
end
                                    disp { puts "3-CSE" }
```

Output:

```
-
-
-
-
-
-
-
-
-
t
h
r
e
e
_
t
i
m
e
s
3
-
C
S
E
3-CSE
3-CSE
```

Iterators

Iterators are nothing but methods supported by *collections*. Objects that store a group of data members are called collections. In Ruby, arrays and hashes can be termed collections. *Iterators* return all the elements of a collection, one after the other. The syntax is

object.iterator { | ***value*** | ***statement*** } or
object.iterator do |***value***|
statements
end

The object is typically an array, a range, or a hash

The times iterator

The *times* iterator method provides a way to build simple counting loops. Typically, *times* is

```
>>5.times {puts "Hi..!"}
Hi..!
Hi..!
Hi..!
Hi..!
Hi..!
=> 5
```

added to a number object, which repeats the attached block that number of times. Consider the following example:

The each iterator

The most common iterator is *each*, which is often used to go through arrays and apply a block to each element.

```
>>list = [2,4,6,8]
=> [2, 4, 6, 8]
>>list.each do |value| puts "#{value}" end
2
4
6
8
=> [2, 4, 6, 8]
```

The upto iterator

Iterates through successive values, starting at *start* and ending at *last* inclusive, passing each value in turn to the block.

```
>>5.upto(8){ |v| puts v }
5
6
7
8
=> 5
```

The downto iterator

This decrements a number. It reduces the number by 1 after each pass through the iterator body. If the argument is not lower, the iterator body is not executed.

```
>>5.downto(3){ |v| puts v }
5
4
3
=> 5
```

The step iterator

The step iterator method takes a terminal value and a step size as parameters and generates the values from that of the object to which it is sent and the terminal value:

```
>>10.step(20,3){ |v| puts v }
10
13
16
19
=> 10
```

The each_char iterator

get each character (as an integer) from a string

```
>>str="CSE"
=> "CSE"
>>str.each_char {|c| puts c }
C
S
E
=> "CSE"
```

The collect iterator

the *collect* iterator method takes the elements from an array, one at a time, and puts the values generated by the given block into a new array:

```
>>list=[5,10,15,20]
=> [5, 10, 15, 20]
>>list.collect {|v| v = v-5 }
=> [0, 5, 10, 15]
>>list
=> [5, 10, 15, 20]
>>list.collect! {|v| v = v+5 }
=> [10, 15, 20, 25]
>>list
=> [10, 15, 20, 25]
```

The each_key iterator

Operates on hashes and returns all the keys of hash

```
>>ages = {"raju"=>20,"ravi"=>21,"kiran"=>19}
=> {"raju"=>20, "ravi"=>21, "kiran"=>19}
>>ages.each_key do |key| puts key end
raju
ravi
kiran
```

The each_value iterator
Operates on hashes and returns all the values of hash

```
>>ages = {"raju"=>20,"ravi"=>21,"kiran"=>19}
=> {"raju"=>20, "ravi"=>21, "kiran"=>19}
>>ages.each_value do |value| puts value end
20
21
19
```

The each_pair iterator
Operates on hashes and returns both keys and values of hash

```
>>ages = {"raju"=>20,"ravi"=>21,"kiran"=>19}^Z
=> {"raju"=>20, "ravi"=>21, "kiran"=>19}
>>ages.each_pair do |key,value| puts "#{key} = #{value}" end
raju = 20
ravi = 21
kiran = 19
```

Methods

A method definition includes the method‘s header and a sequence of statements, ending with the *end* reserved word that describes its actions. A method header is the reserved word *def*, the method‘s name, and optionally a parenthesized list of formal parameters.

- Method names must begin with lowercase letters.
- If the method has no parameters, the parentheses are omitted. In fact, the parentheses are optional in all cases, but it is common practice to include them when there are parameters and omit them when there are no parameters.
- The types of the parameters are not specified in the parameter list, because Ruby variables do not have types
- The type of the return object is also not specified in a method definition.

Syntax:

```
def method_name ( )
        statements
        [ return value ]
end
```

Example:

```
def add (a , b)
        c = a + b
        puts ―Sum=#{c}‖
end
add(2,
5)
```

will print **7**

undef will remove the method

```
>>  undef
add nil
```

Classes

A class is defined as a template for objects, of which any number can be created. An object has a state, which is maintained in its collection of instance variables, and a behavior, which is defined by its methods. An object can also have constants and a constructor.

The Basics of Classes

The methods and variables of a class are defined in the syntactic container that has the following form:

class class_name
...
end

- Class names, like constant names, must begin with uppercase letters.
- Instance variables are used to store the state of an object. They are defined in the class definition, and every object of the class gets its own copy of the instance variables.
- The name of an instance variable must begin with an at sign (@), which distinguishes instance variables from other variables.
- A class can have a single constructor, which in Ruby is a method with the name *initialize*, which is used to initialize instance variables to values.
- A constructor can take any number of parameters, which are treated as local variables; therefore, their names begin with lowercase letters or underscores.
- Classes in Ruby are dynamic in the sense that members can be added at any time
- Methods can also be removed from a class, by providing another class definition in which the method to be removed is sent to the method *remove_method* as a parameter.

Following is an example of a class, named Stack2, that defines a stack like data structure implemented in an array.

```
## Stack_2.rb - A Class implements stack like structure
in an array class Stack_2
        def initialize(len = 5)  ##
                Constructor     @stack     =
                Array.new(len)  @max_len
                = len
                @top = -1
        end
        def    push(num)    ##push
          method   if   @top   ==
          @max_len
                puts "Stack is full"
else                                        @
                                            t
                                            o
                                            p
                                            +
                                            =
                                            1
                                            @
                                            s
                                            t
                                            a
                                            c
                                            k
                                            [
                                            @
                                            t
                                            o
                                            p
                                            ]
                                            =
          end end
```

```
n                                                    m
u
        def pop()       ##pop
          method if @top == -1
                puts "Stack is empty"
else                                                 @top -= 1

          end end

        def  disp()      ##       printing
           method for i in 0..@top
                print "#{@stack[i]} "
                                                                e
                                    end                         n
                                                                d
end                                                             p
                                                                u
                                                                t
                                                                s
my_stack                =
Stack_2.new(2)
my_stack.pop()
my_stack.push(10)
my_stack.push(20)
my_stack.push(30)
my_stack.disp()
my_stack.push(40)
my_stack.pop()
my_stack.pop()
my_stack.disp()
```

Output:

```
>ruby -w Stack_2.rb
Stack is empty
10  20  30
Stack is full
10
```

Access Control

The Ruby supports three levels of access control for methods are defined as follows:

- **"Public access"** means that the method can be called by any code.
- **"Protected access"** means that only objects of the defining class and its subclasses may call the method.
- **"Private access"** means that the method can only be used by object that defines itself. So, no code can ever call the private methods of another object.

All instance variables has private access by default, and you can not change its access control.

```
class My_class
  def meth1
  ...
  end
  ...
private
  def meth7
  ...
  end
  ...
protected
  def meth11
  ...
  end
  ...
end  # of class My_class
```

Inheritance

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application.

Inheritance also provides an opportunity to reuse the code functionality and fast implementation time but unfortunately Ruby does not support multiple levels of inheritances but Ruby supports **mixins**. A mixin is like a specialized implementation of multiple inheritance in which only the interface portion is inherited.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base class or superclass**, and the new class is referred to as the **derived class or sub-class**.

Ruby also supports the concept of subclassing, i.e., inheritance and following example explains the concept. The syntax for extending a class is simple. Just add a < character and the name of the superclass to your class statement. For example, following define a class *BigBox* as a subclass of *Box*:

```
class Box                       # definition of Base
        class def initialize(w,h)       ## constructor
        method
                @width, @height = w, h
        end
        def getArea             ##      instance
                method @width * @height
        end
end
class BigBox < Box              ## definition of subclass
        def printArea           ##  add  a  new  instance
```

```
        method @area = @width * @height
        puts "Big box area is : #@area"
    end
end

box = BigBox.new(10, 20)    ## create an
object box.printArea()      # print the area
```

When the above code is executed, it produces the following result: Big box area is : 200