RUBY Programming

Ruby is a Pure Object-Oriented Programming language. It was created in 1993 by Yukihiro Matsumoto of Japan.

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 Ruby 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.

All ruby files will have extension **.rb**. So, put the following source code in a test.rb file.

```
#!/usr/bin/ruby
puts "Hello World!";
```

Here, I assumed that you have Ruby interpreter available in /usr/bin directory. Now, try to run this program as follows:

```
$ ruby test.rb
```

This will produce the following result:

```
Hello, Ruby!
```

Whitespace in Ruby Program:

Whitespace characters such as spaces and tabs are generally ignored in Ruby code, except when they appear in strings. Sometimes, however, they are used to interpret ambiguous statements

Line Endings in Ruby Program:

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.

Ruby Identifiers:

Identifiers are names of variables, constants, and methods. Ruby identifiers are case sensitive. It mean Ram and RAM are two different identifiers in Ruby. Ruby identifier names may consist of alphanumeric characters and the underscore character (_).

Reserved Words:

These reserved words may not be used as constant or variable names. The following list shows the reserved words in Ruby.

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

Ruby *BEGIN* Statement:

Syntax:

```
BEGIN
{
   code
}
```

Declares *code* to be called before the program is run.

Example:

```
#!/usr/bin/ruby

puts "This is main Ruby Program"
BEGIN {
   puts "Initializing Ruby Program"
}
```

This will produce the following result:

```
Initializing Ruby Program
This is main Ruby Program
```

Ruby *END* Statement:

Syntax:

```
END
   {
      code
   }
```

Declares *code* to be called at the end of the program.

Example:

```
#!/usr/bin/ruby

puts "This is main Ruby Program"

END {
   puts "Terminating Ruby Program"
}
BEGIN {
   puts "Initializing Ruby Program"
}
```

This will produce the following result:

```
Initializing Ruby Program
This is main Ruby Program
Terminating Ruby Program
```

Ruby Comments: A comment hides a line, part of a line, or several lines from the Ruby interpreter. There are two types of comments:

1. Single Line Comment (**#**)
2. Multi Line Comment (**=begin/=end**)

Example for Single line Comment:

```
# I am a comment. Just ignore me.
```

Example for Multi line Comment:

```
=begin
This is a comment.
This is a comment, too.
I said that already.
=end
```

Operators:

Ruby supports a rich set of operators, as you'd expect from a modern language. Most operators are actually method calls. For example, a + b is interpreted as a.+(b), where the + method in the object referred to by variable a is called with b as its argument.

For each operator (+ - * / % ** & | ^ << >> && ||), there is a corresponding form of abbreviated assignment operator (+= -= etc.)

Arithmetic Operators:

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 0
**	Exponent - Performs exponential (power) calculation on operators	a**b will give 10 to the power 20

Comparison Operators:

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a == b) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.

>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
<=>	Combined comparison operator. Returns 0 if first operand equals second, 1 if first operand is greater than the second and -1 if first operand is less than the second.	(a <=> b) returns -1.
===	Used to test equality within a when clause of a *case* statement.	(1...10) === 5 returns true.

Assignment Operators:

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	c = a + b will assignee value of a + b into c
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**=	Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a

Parallel Assignment:

Ruby also supports the parallel assignment of variables. This enables multiple variables to be initialized with a single line of Ruby code. For example:

```
a = 10
b = 20
c = 30
```

may be more quickly declared using parallel assignment:

```
a, b, c = 10, 20, 30
```

Parallel assignment is also useful for swapping the values held in two variables:

```
a, b = b, c
```

Bitwise Operators:

A bitwise operator works on bits and performs bit by bit operation.

Assume if a = 60; and b = 13; now in binary format they will be as follows:

a = 0011 1100

b = 0000 1101

----

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

There are following Bitwise operators supported by Ruby language

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(a & b) will give 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(a | b) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(a ^ b) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~a) will give -61, which is 1100 0011 in 2's complement form due to a signed binary number.

<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	a << 2 will give 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 will give 15, which is 0000 1111

Logical Operators:

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
and	Called Logical AND operator. If both the operands are true, then the condition becomes true.	(a and b) is true.
or	Called Logical OR Operator. If any of the two operands are non zero, then the condition becomes true.	(a or b) is true.
&&	Called Logical AND operator. If both the operands are non zero, then the condition becomes true.	(a && b) is true.
||	Called Logical OR Operator. If any of the two operands are non zero, then the condition becomes true.	(a || b) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(a && b) is false.
not	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	not(a && b) is false.

Ternary operator:

This first evaluates an expression for a true or false value and then execute one of the two given statements depending upon the result of the evaluation. The conditional operator has this syntax:

Operator	Description	Example
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

Range operators:

Sequence ranges in Ruby are used to create a range of successive values - consisting of a start value, an end value and a range of values in between.

In Ruby, these sequences are created using the ".." and "..." range operators. The two-dot form creates an inclusive range, while the three-dot form creates a range that excludes the specified high value.

Operator	Description	Example
..	Creates a range from start point to end point inclusive	1..10 Creates a range from 1 to 10 inclusive
...	Creates a range from start point to end point exclusive	1...10 Creates a range from 1 to 9

defined? operator:

defined? is a special operator that takes the form of a method call to determine whether or not the passed expression is defined. It returns a description string of the expression, or *nil* if the expression isn't defined.

There are various usage of defined? operator:

Usage 1

```
defined? variable # True if variable is initialized
```

For Example:

```
foo = 42
defined? foo    # => "local-variable"
defined? $_     # => "global-variable"
defined? bar    # => nil (undefined)
```

Usage 2

```
defined? method_call # True if a method is defined
```

For Example:

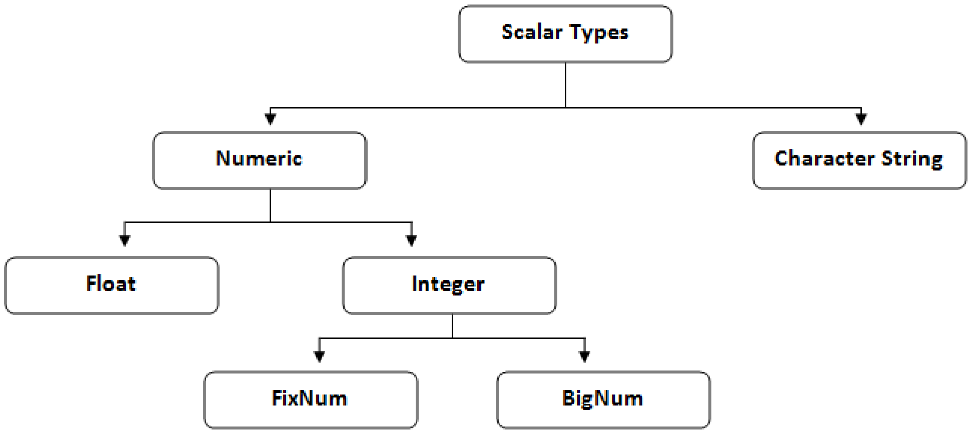
```
defined? puts        # => "method"
defined? puts(bar)   # => nil (bar is not defined here)
defined? unpack      # => nil (not defined here)
```

Data Types in Ruby:

There are Three types of data types used in Ruby and those are..

1. Scalar
2. Arrays
3. Hashes

Scalars: The scalar types are further classified as numeric and character strings and show in below figure.



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.

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
```

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
```

Variables in a Ruby Class:

Ruby provides five types of variables:

- **Local Variables:** Local variables are the variables that are defined in a method. Local variables are not available outside the method. Local variables begin with a lowercase letter or _.
- **Instance Variables:** Instance variables are available across methods for any particular instance or object. That means that instance variables change from object to object. Instance variables are preceded by the at sign (@) followed by the variable name.
- **Class Variables:** Class variables are available across different objects. A class variable belongs to the class and is a characteristic of a class. They are preceded by the sign @@ and are followed by the variable name.
- **Global Variables:** Global variables begin with $. Uninitialized global variables have the value *nil*.
- **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.

Local Variables: Local variables are the variables that are defined in a method. Local variables are not available outside the method. Local variables begin **with a lowercase letter or _.**

Global Variables: Global variables begin **with $.** Uninitialized global variables have the value *nil* .

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
```

Instance Variables: Instance variables begin **with** @. Uninitialized instance variables have the value *nil*.

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
```

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

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
      @@no_of_customers += 1
   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()
       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: 2
Total number of customers: 2
```

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.

Example:

```
#!/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
```

Input / Output: (Simple I/O):

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
```

***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:

```
#!/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
```

Control Statements:

Conditional Statements:

Ruby *if...else* Statement:

Syntax:

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

if expressions are used for conditional execution. The values *false* and *nil* are false, and everything else are true. Notice Ruby uses elsif, not else if nor elif.

Executes *code* if the *conditional* is true. If the *conditional* is not true, *code* specified in the else clause is executed.

Example:

```
#!/usr/bin/ruby
x=1
if x > 2
   puts "x is greater than 2"
elsif x <= 2 and x!=0
   puts "x is 1"
else
   puts "I can't guess the number"
end
```

```
OUTPUT:
x is 1
```

Ruby *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:

```
#!/usr/bin/ruby
x=1
unless x>2
   puts "x is less than 2"
 else
  puts "x is greater than 2"
end
```

This will produce the following result:

```
x is less than 2
```

Ruby *case* Statement

Syntax:

```
case expression
[when expression [, expression ...] [then]
   code ]...
[else
   code ]
end
```

- Compares the *expression* specified by case and that specified by when using the === operator and executes the *code* of the when clause that matches.
- The *expression* specified by the when clause is evaluated as the left operand. If no when clauses match, *case* executes the code of the *else* clause.
- A when statement's expression is separated from code by the reserved word then, a newline, or a semicolon.

Example:

```
#!/usr/bin/ruby
$age =  5
case $age
when 0 .. 2
    puts "baby"
when 3 .. 6
    puts "little child"
when 7 .. 12
    puts "child"
when 13 .. 18
    puts "youth"
else
    puts "adult"
end
```

This will produce the following result:

```
little child
```

Looping Statements:

Ruby *while* Statement:

Syntax:

```
while conditional [do]
   code
end
```

Executes *code* while *conditional* is true. A *while* loop's *conditional* is separated from *code* by the reserved word do, a newline, backslash \, or a semicolon ;.

Example:

```
#!/usr/bin/ruby
$i = 0
$num = 5

while $i < $num  do
   puts("Inside the loop i = #$i" )
   $i +=1
end
```

This will produce the following result:

```
Inside the loop i = 0
Inside the loop i = 1
Inside the loop i = 2
Inside the loop i = 3
Inside the loop i = 4
```

Ruby *until* Statement:

```
until conditional [do]
   code
end
```

Executes *code* while *conditional* is false. An *until* statement's conditional is separated from *code* by the reserved word *do*, a newline, or a semicolon.

Example:

```
#!/usr/bin/ruby
$i = 0
$num = 5
until $i > $num  do
   puts("Inside the loop i = #$i" )
   $i +=1;
end
```

This will produce the following result:

```
Inside the loop i = 0
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
```

Ruby *for* Statement:

Syntax:

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

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

Example:

```
#!/usr/bin/ruby
for i in 0..5
   puts "Value of local variable is #{i}"
end
```

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

```
Value of local variable is 0
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
```

Ruby *break* Statement:

Syntax:

```
break
```

Terminates the most internal loop. Terminates a method with an associated block if called within the block (with the method returning nil).

Example:

```
#!/usr/bin/ruby
for i in 0..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 0
Value of local variable is 1
Value of local variable is 2
```

Ruby *next* Statement:

Syntax:

```
next
```

Jumps to next iteration of the most internal loop. Terminates execution of a block if called within a block (with *yield* or call returning nil).

Example:

```
#!/usr/bin/ruby
for i in 0..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 2
Value of local variable is 3
Value of local variable is 4
Value of local variable is 5
```

Arrays:

- Ruby arrays are ordered, integer-indexed collections of any object. Each element in an array is associated with and referred to by an index.
- Array indexing starts at 0, as in C or Java. A negative index is assumed relative to the end of the array --- that is, an index of -1 indicates the last element of the array, -2 is the next to last element in the array, and so on.
- Ruby arrays can hold objects such as String, Integer, Fixnum, Hash, Symbol, even other Array objects. Ruby arrays are not as rigid as arrays in other languages. Ruby arrays grow automatically while adding elements to them.

Creating Arrays:

There are many ways to create or initialize an array. One way is with the *new*class method:

```
names = Array.new
```

You can set the size of an array at the time of creating array:

```
names = Array.new(20)
```

The array *names* now has a size or length of 20 elements. You can return the size of an array with either the size or length methods:

```
#!/usr/bin/ruby
names = Array.new(20)
puts names.size  # This returns 20
puts names.length # This also returns 20
```

This will produce the following result:

```
20
20
```

You can assign a value to each element in the array as follows:

```
#!/usr/bin/ruby
names = Array.new(4, "mac")
puts "#{names}"
```

This will produce the following result:

```
macmacmacmac
```

Array Built-in Methods:

S.No	Method	Description
01	**array.first**	Returns the first element
02	**array.last**	Returns the last element
03	**array.length**	Returns the number of elements in *self*. May be zero.
04	**array.size**	Returns the length of *array* (number of elements). Alias for length.
05	**array.push(obj, ...)**	Pushes (appends) the given obj onto the end of this array
06	**array.unshift(obj, ...)**	Prepends objects to the front of array, other elements up one.
07	**array.pop**	Removes the last element from *array* and returns it, or *nil* if *array* is empty.
08	**array.shift**	Returns the first element of *self* and removes it (shifting all other elements down by one). Returns *nil* if the array is empty.
09	**array.sort**	Returns a new array created by sorting self
10	**array.reverse**	Returns a new array containing array's elements in reverse order.
11	**array.slice(index) [or] array.slice(start, length)**	Returns the element at *index*, or returns a subarray starting at *start* and continuing for *length* elements.
12	**array & other_array**	Returns a new array containing elements common to the two arrays, with no duplicates
13	**array + other_array**	Returns a new array built by concatenating the two arrays together to produce a third array
14	**array - other_array**	Returns a new array that is a copy of the original array, removing any items that also appear in other_array.
15	**array <=> other_array**	Compares str with other_str, returning -1 (less than), 0 (equal), or 1 (greater than). The comparison is casesensitive.
16	**array | other_array**	Returns a new array by joining array with other_array, removing duplicates
17	**array == other_array**	Two arrays are equal if they contain the same number of elements and if each element is equal to (according to Object.==) the corresponding element in the other array.
18	**array[index] [or] array[start, length] [or] array[range] [or] array.slice(index) [or] array.slice(start, length) [or]**	Returns the element at *index*, or returns a subarray starting at *start* and continuing for *length* elements, or returns a subarray specified by *range*. Negative indices count backward from the end of the array (-1 is the last element). Returns *nil* if the index (or starting index) is out of range.

	array.slice(range)	
19	**array.at(index)**	Returns the element at index. A negative index counts from the end of self. Returns nil if the index is out of range
20	**array.clear**	Removes all elements from array.
21	**array.concat(other_array)**	Appends the elements in other_array to *self*
22	**array.delete_at(index)**	Deletes the element at the specified *index*, returning that element, or *nil* if the index is out of range.
23	**array.each { |item| block }**	Calls *block* once for each element in *self*, passing that element as a parameter.

Example:

```
# Write simple Ruby programs that uses arrays in Ruby

# Array creation
a=Array.new

puts "Enter number elements in array:"
n=gets.to_i;

puts "Enter arry elements: "
for i in (0...n) do
  a[i]=gets.to_i
end

puts "Array elemets are: #{a}"

# Modify the first element.
a[0] = 100

# Display the array.
puts "Array elements are: #{a}"

# Display First element
puts "Firsrt element is: #{a.first}";

# Display second element
puts "Second element is: #{a[1]}";
```

```
# Display Last element
puts "last element is: #{a.last}";

# Adding new elements to array
a.push(50);

# Display the array.
puts "After push array elements are: #{a}"

# Deleting last element
a.pop;

# Display the array.
puts "After deleting array elements are: #{a}"
```

Hashes:

- A Hash is a collection of key-value pairs like this: "employee" => "salary". It is similar to an Array, except that indexing is done via arbitrary keys of any object type, not an integer index.
- The order in which you traverse a hash by either key or value may seem arbitrary and will generally not be in the insertion order. If you attempt to access a hash with a key that does not exist, the method will return *nil*.

Creating Hashes:

We can create an empty hash with the *new* class method:

```
months = Hash.new
```

We can also use *new* to create a hash with a default value, which is otherwise just *nil*:

```
months = Hash.new( "month" )
or

months = Hash.new "month"
```

When we access any key in a hash that has a default value, if the key or value doesn't exist, accessing the hash will return the default value:

```
#!/usr/bin/ruby
months = Hash.new( "month" )

puts "#{months[0]}"

puts "#{months[72]}"
```

This will produce the following result:

```
month
month
```

This will return a new hash populated with the given objects. Now using created object we can call any available instance methods. For example:

```
#!/usr/bin/ruby
months = Hash.new( "month" )

months = {"1" => "January", "2" => "February"}

keys = months.keys

puts "#{keys}"
```

This will produce the following result:

```
["1", "2"]
```

Built Methods for Hashes:

S.No	Method	Description
01	**hash.[key]**	Using a key, references a value from hash. If the key is not found, returns a default value.
02	**hash.[key]=value**	Associates the value given by *value* with the key given by *key*.
03	**hash.clear**	Removes all key-value pairs from hash.
04	**hash.delete(key)**	Deletes a key-value pair from *hash* by *key*
05	**hash.each { |key,value| block }**	Iterates over *hash*, calling the block once for each key, passing the key-value as a two-element array.
06	**hash.index(value)**	Returns the *key* for the given *value* in hash, *nil* if no matching value is found.
07	**hash.indexes(keys)**	Returns a new array consisting of values for the given key(s). Will insert the default value for keys that are not found. This method is deprecated. Use select.
08	**hash.keys**	Creates a new array with keys from *hash*.
09	**hash.values**	Returns a new array containing all the values of *hash*.
10	**hash.length**	Returns the size or length of *hash* as an integer.
11	**hash.merge(other_hash)**	Returns a new hash containing the contents of *hash* and *other_hash*, overwriting pairs in hash with duplicate keys with those from *other_hash*
12	**hash.rehash**	Rebuilds *hash* based on the current values for each *key*. If values have changed since they were inserted, this method reindexes *hash*.
13	**hash.replace(other_hash)**	Replaces the contents of *hash* with the contents of *other_hash*.
14	**hash.shift**	Removes a key-value pair from *hash*, returning it as a two-element array.
15	**hash.size**	Returns the *size* or length of *hash* as an integer.
16	**hash.sort**	Converts *hash* to a two-dimensional array containing arrays of key-value pairs, then sorts it as an array.

17	**hash.store(key, value)**	Stores a key-value pair in *hash*.

Example:

```
# Write programs which uses associative arrays concept of Ruby

# Create a new hash.
ages = Hash["name1"=>25,"name2"=>27,"name3"=>21,"name4"=>19]

# Display the Hash.
puts "Hash is: #{ages}"

# Size of Hash
puts "Hash size is: #{ages.size}"

#Adding new elements
ages["name5"]=31
ages.store("name6",23)

# Display the Hash.
puts "Hash is: #{ages}"

# Keys values are...
key_var=ages.keys
puts "Hash Keys : #{key_var}"

# Hash values are...
val_var=ages.values
puts "Hash Values : #{val_var}"
```

Strings:

- A String object in Ruby holds and manipulates an arbitrary sequence of one or more bytes, typically representing characters that represent human language.
- The simplest string literals are enclosed in single quotes (the apostrophe character). The text within the quote marks is the value of the string:

For Example..

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

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

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

The backslash also works to escape another backslash, so that the second backslash is not itself interpreted as an escape character.

Following are string-related features Ruby.

Expression Substitution:

Expression substitution is a means of embedding the value of any Ruby expression into a string using #{ and }:

```
#!/usr/bin/ruby

x, y, z = 12, 36, 72

puts "The value of x is #{ x }."

puts "The sum of x and y is #{ x + y }."

puts "The average was #{ (x + y + z)/3 }."
```

This will produce the following result:

```
The value of x is 12.
The sum of x and y is 48.

The average was 40.
```

General Delimited Strings:

With general delimited strings, we can create strings inside a pair of matching though arbitrary delimiter characters, e.g., !, (, {, <, etc., preceded by a percent character (%). Q, q, and x have special meanings. General delimited strings can be

```
%{Ruby is fun.}  equivalent to "Ruby is fun."

%Q{ Ruby is fun. } equivalent to " Ruby is fun. "

%q[Ruby is fun.]  equivalent to a single-quoted string

%x!ls! equivalent to back tick command output `ls`
```

Escape Characters:

Following table is a list of escape or non-printable characters that can be represented with backslash notation.

NOTE: In a double-quoted string, an escape character is interpreted; in a single-quoted string, an escape character is preserved.

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x

\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

String Built-in Methods:

str.capitalize - Capitalizes a string.
str.capitalize! - Same as capitalize, but changes are made in place.
str.casecmp - Makes a case-insensitive comparison of strings.
str.center - Centers a string.
str.chomp - Removes the record separator ($/), usually \n, from the end of a string. If no record separator exists, does nothing.
str.chomp! - Same as chomp, but changes are made in place
str.chop - Removes the last character in str.
str.chop! - Same as chop, but changes are made in place.
str.concat(other_str) - Concatenates other_str to str.
str.count(str, ...) Counts one or more sets of characters. If there is more than one set of characters, counts

the intersection of those sets
str.delete(other_str, ...) Returns a copy of str with all characters in the intersection of its arguments deleted.
str.delete!(other_str, ...) - Same as delete, but changes are made in place.
str.downcase- Returns a copy of str with all uppercase letters replaced with lowercase.
str.downcase!-Same as downcase, but changes are made in place.
str.empty?- Returns true if str is empty (has a zero length).
str.eql?(other)- Two strings are equal if the have the same length and content.
str.length - Returns the length of str. Compare size.
str.lstrip - Returns a copy of str with leading whitespace removed.
str.lstrip! Removes leading whitespace from str, returning nil if no change was made.
str.match(pattern) Converts pattern to a Regexp (if it isn't already one), then invokes its match method on str
str.reverse- Returns a new string with the characters from str in reverse order.
str.reverse!-Reverses str in place.
str.rstrip - Returns a copy of str with trailing whitespace removed.
str.rstrip! Removes trailing whitespace from str, returning nil if no change was made.
str.strip - Returns a copy of str with leading and trailing whitespace removed.

Ruby Methods:

- Ruby methods are very similar to functions in any other programming language. Ruby methods are used to bundle one or more repeatable statements into a single unit.
- Method names should begin with a lowercase letter. If you begin a method name with an uppercase letter, Ruby might think that it is a constant and hence can parse the call incorrectly.
- Methods should be defined before calling them, otherwise Ruby will raise an exception for undefined method invoking.

Syntax:

```
def method_name [( [arg [= default]]...[, * arg [, &expr ]])]
   expr..
end
```

So you can define a simple method as follows:

```
def method_name
   expr..
end
```

You can represent a method that accepts parameters like this:

```
def method_name (var1, var2)
   expr..
end
```

You can set default values for the parameters which will be used if method is called without passing required parameters:

```
def method_name (var1=value1, var2=value2)
   expr..
end
```

Whenever you call the simple method, you write only the method name as follows:

```
method_name
```

However, when you call a method with parameters, you write the method name along with the parameters, such as:

```
method_name 25, 30
```

Example:

```
#!/usr/bin/ruby
def test(a1="Ruby", a2="Perl")
   puts "The programming language is #{a1}"
   puts "The programming language is #{a2}"
end
test "C", "C++"
test
```

This will produce the following result:

```
The programming language is C
The programming language is C++
The programming language is Ruby
The programming language is Perl
```

Ruby *return* Statement:

The *return* statement in ruby is used to return one or more values from a Ruby Method.

Syntax:

```
return [expr[`,' expr...]]
```

If more than two expressions are given, the array containing these values will be the return value. If no expression given, nil will be the return value.

Example:

```
return
OR
return 12
OR
return 1,2,3
```

Have a look at this example:

```
#!/usr/bin/ruby
def test
```

```
   i = 100
   j = 200
   k = 300
return i, j, k
end
var = test
puts var
```

This will produce the following result:

```
100
200
300
```

Variable Number of Parameters:

Suppose you declare a method that takes two parameters, whenever you call this method, you need to pass two parameters along with it. However, Ruby allows you to declare methods that work with a variable number of parameters. Let us examine a sample of this:

```
#!/usr/bin/ruby
def sample (*test)
   puts "The number of parameters is #{test.length}"
   for i in 0...test.length
      puts "The parameters are #{test[i]}"
   end
end
sample "Zara", "6", "F"
sample "Mac", "36", "M", "MCA"
```

In this code, you have declared a method sample that accepts one parameter test. However, this parameter is a variable parameter. This means that this parameter can take in any number of variables. So above code will produce following result:

```
The number of parameters is 3
The parameters are Zara
The parameters are 6
The parameters are F
The number of parameters is 4
The parameters are Mac
The parameters are 36
The parameters are M
The parameters are MCA
```

Classes and Objects:

A class in Ruby always starts with the keyword *class* followed by the name of the class. The name should always be in initial capitals. The class *Customer* can be displayed as:

```
class Customer
end
```

We terminate a class by using the keyword *end*. All the data members in the*class* are between the class definition and the *end* keyword.

Variables in a Ruby Class:

Ruby provides four types of variables:

- **Local Variables:** Local variables are the variables that are defined in a method. Local variables are not available outside the method. You will see more details about method in subsequent chapter. Local variables begin with a lowercase letter or _.
- **Instance Variables:** Instance variables are available across methods for any particular instance or object. That means that instance variables change from object to object. Instance variables are preceded by the at sign (@) followed by the variable name.
- **Class Variables:** Class variables are available across different objects. A class variable belongs to the class and is a characteristic of a class. They are preceded by the sign @@ and are followed by the variable name.
- **Global Variables:** Class variables are not available across classes. If you want to have a single variable, which is available across classes, you need to define a global variable. The global variables are always preceded by the dollar sign ($).

Example:

Using the class variable @@no_of_customers, we can determine the number of objects that are being created. This enables in deriving the number of customers.

```
class Customer
   @@no_of_customers=0
end
```

Creating Objects in Ruby using *new* Method:

Objects are instances of the class. You will now learn how to create objects of a class in Ruby. we can create objects in Ruby by using the method *new* of the class.

The method *new* is a unique type of method, which is predefined in the Ruby library. The new method belongs to the *class* methods.

Here is the example to create two objects cust1 and cust2 of the class Customer:

```
cust1 = Customer. new
cust2 = Customer. new
```

Here, cust1 and cust2 are the names of two objects. You write the object name followed by the equal to sign (=) after which the class name will follow. Then, the dot operator and the keyword *new* will follow.

Custom Method to create Ruby Objects :

- We can pass parameters to method *new* and those parameters can be used to initialize class variables.
- When we plan to declare the *new* method with parameters, you need to declare the method *initialize* at the time of the class creation.
- The *initialize* method is a special type of method, which will be executed when the *new* method of the class is called with parameters.

Here is the example to create initialize method:

```
class Customer
   @@no_of_customers=0
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
   end
end
```

Member Functions in Ruby Class:

- In Ruby, functions are called methods. Each method in a *class* starts with the keyword *def* followed by the method name.

- The method name always preferred in **lowercase letters**. We end a method in Ruby by using the keyword *end*.

An Example for Classes and Objects:

```
#!/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 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.

In ruby, two iterators are available, these are

1. *each* and
2. *collect*.

Ruby *each* Iterator:

The each iterator returns all the elements of an array or a hash.

Syntax:

```
collection.each do |variable|
   code

end
```

Executes *code* for each element in *collection*. Here, *collection* could be an array or a ruby hash.

Example:

```
#!/usr/bin/ruby

ary = [1,2,3,4,5]

ary.each do |i|

   puts i

end
```

This will produce the following result:

```
1

2

3

4

5
```

Ruby *collect* Iterator:

The *collect* iterator returns all the elements of a collection.

Syntax:

```
collection = collection.collect
```

The *collect* method need not always be associated with a block. The *collect* method returns the entire collection, regardless of whether it is an array or a hash.

Example:

```
#!/usr/bin/ruby

a = [1,2,3,4,5]

b = Array.new

b = a.collect{ |e| e }
```

```
puts b
```

This will produce the following result:

```
1
2
3
4
5
```

Ruby Regular Expressions (Pattern Matching):

- A *regular expression* is a special sequence of characters that helps you match or find other strings or sets of strings using a specialized syntax held in a pattern.
- A *regular expression literal* is a pattern between slashes or between arbitrary delimiters followed by %r as follows:

Syntax:

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

Example:

```
#!/usr/bin/ruby
line1 = "Cats are smarter than dogs";
line2 = "Dogs also like meat";
if ( line1 =~ /Cats(.*)/ )
  puts "Line1 contains Cats"
end
if ( line2 =~ /Cats(.*)/ )
  puts "Line2 contains  Dogs"
end
```

This will produce the following result:

```
Line1 contains Cats
```

Regular-expression patterns:

Except for control characters, **(+ ? . * ^ $ ( ) [ ] { } | \)**, all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Ruby

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 1 or more occurrence of preceding expression.
re?	Matches 0 or 1 occurrence of preceding expression.
a| b	Matches either a or b.
\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.

Character classes:

Example	Description
/[Rr]uby/	Match "Ruby" or "ruby"
/rub[ye]/	Match "ruby" or "rube"
/[aeiou]/	Match any one lowercase vowel
/[0-9]/	Match any digit; same as /[0123456789]/
/[a-z]/	Match any lowercase ASCII letter
/[A-Z]/	Match any uppercase ASCII letter
/[a-zA-Z0-9]/	Match any of the above
/[^aeiou]/	Match anything other than a lowercase vowel
/[^0-9]/	Match anything other than a digit

Repetition Cases:

Example	Description
/ruby?/	Match "rub" or "ruby": the y is optional
/ruby*/	Match "rub" plus 0 or more ys
/ruby+/	Match "rub" plus 1 or more ys
/\d{3}/	Match exactly 3 digits
/\d{3,}/	Match 3 or more digits
/\d{3,5}/	Match 3, 4, or 5 digits

Ruby Blocks:

Ruby has a concept of Block.

- A block consists of chunks of code.
- We assign a name to a block.
- The code in the block is always enclosed within braces ({}).
- A block is always invoked from a function with the same name as that of the block. This means that if we have a block with the name *test*, then you use the function *test* to invoke this block.
- We invoke a block by using the *yield* statement.

Syntax:

```
block_name{
   statement1

   statement2

   ..........

}
```

Here, we will learn to invoke a block by using a simple *yield* statement. We will also learn to use a *yield* statement with parameters for invoking a block. We will check the sample code with both types of *yield* statements.

yield Statement:

Let's look at an example of the yield statement:

```
#!/usr/bin/ruby

def test

   puts "You are in the method"

   yield

   puts "You are again back to the method"

   yield

end

test {puts "You are in the block"}
```

This will produce the following result:

```
You are in the method
You are in the block

You are again back to the method

You are in the block
```

We can also pass parameters with the yield statement. Here is an example:

```
#!/usr/bin/ruby
def test
```

```
   yield 5

   puts "You are in the method test"

   yield 100

end

test {|i| puts "You are in the block #{i}"}
```

This will produce the following result:

```
You are in the block 5
You are in the method test

You are in the block 100
```

Blocks and Methods:

We have seen how a block and a method can be associated with each other. We normally invoke a block by using the yield statement from a method that has the same name as that of the block. Therefore, you write:

```
#!/usr/bin/ruby
def test

  yield

end

test{ puts "Hello world"}
```

This example is the simplest way to implement a block. You call the test block by using the *yield* statement. But if the last argument of a method is preceded by &, then you can pass a block to this method and this block will be assigned to the last parameter.

```
#!/usr/bin/ruby

def test(&block)
   block.call
end
test { puts "Hello World!"}
```

This will produce the following result:

```
Hello World!
```

BEGIN and END Blocks

Every Ruby source file can declare blocks of code to be run as the file is being loaded (the BEGIN blocks) and after the program has finished executing (the END blocks).

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

```
BEGIN {
  # BEGIN block code
  puts "BEGIN code block"
}
END {
  # END block code
  puts "END code block"
}
  # MAIN block code
puts "MAIN code block"
```

A program may include multiple BEGIN and END blocks. BEGIN blocks are executed in the order they are encountered. END blocks are executed in reverse order. When executed, above program produces the following result:

```
BEGIN code block
MAIN code block

END code block
```

Modules:

Modules are a way of grouping together methods, classes, and constants. Modules give you two major benefits.

- Modules provide a *namespace* and prevent name clashes.
- Modules implement the *mixin* facility.

Modules define a namespace, a sandbox in which your methods and constants can play without having to worry about being stepped on by other methods and constants.

Syntax:

```
module Identifier
   statement1

   statement2

   ...........
end
```

Module constants are named just like class constants, with an initial uppercase letter. The method definitions look similar, too: Module methods are defined just like class methods.

```
#!/usr/bin/ruby

# Module defined in trig.rb file

module Trig

   PI = 3.141592654
   def Trig.sin(x)
   # ..
   end
   def Trig.cos(x)
   # ..
   end
end
```

***include* Statement:**

We can embed a module in a class. To embed a module in a class, you use the *include* statement in the class:

Syntax:

```
include modulename
```

If a module is defined in a separate file, then it is required to include that file using *require* statement before embedding module in a class.

Example:

Consider following module written in *support.rb* file.

```
module Week
   FIRST_DAY = "Sunday"
   def Week.weeks_in_month
      puts "You have four weeks in a month"
   end
   def Week.weeks_in_year
      puts "You have 52 weeks in a year"
   end
end
```

Now, you can include this module in a class as follows:

```
#!/usr/bin/ruby
$LOAD_PATH << '.'
require "support"
class Decade
include Week
   no_of_yrs=10
   def no_of_months
```

```
      puts Week::FIRST_DAY
      number=10*12
      puts number
   end
end
d1=Decade.new
puts Week::FIRST_DAY
Week.weeks_in_month
Week.weeks_in_year
d1.no_of_months
```

This will produce the following result:

```
Sunday
You have four weeks in a month
You have 52 weeks in a year
Sunday
120
```

require Statement:

The require statement is similar to the include statement of C and C++ and the import statement of Java. If a third program wants to use any defined module, it can simply load the module files using the Ruby *require* statement:

Syntax:

```
require filename
```

Here, it is not required to give **.rb** extension along with a file name.

Example:

```
$LOAD_PATH << '.'
require 'trig.rb'
require 'moral'
y = Trig.sin(Trig::PI/4)
wrongdoing = Moral.sin(Moral::VERY_BAD)
```