**Ruby** is a dynamic, reflective, object-oriented, general-purpose programming language. It was designed and developed in the mid-1990s by Yukihiro "Matz" Matsumoto in Japan.

Puts “Welcome to Ruby Language” # => Welcome to Ruby Language

Name = “John”

Puts “My name is #Name” # => My name is John

Puts “The sum of 50 and 50 is #{50+50}” # => The sum of 50 and 50 is 100

puts "10" + "10" # => 1010

**Comments in Ruby :**

Comments are lines of annotation within Ruby code that are ignored at runtime.

# This is a single line comment.

=begin

This is a multiline comment and con spwan as many lines as you

like. But =begin and =end should come in the first line only.

=end

@counter # keeps track times page has been hit

@siteCounter # keeps track of times all pages have been hit

**Operators :**

The special symbols that triggers some special meaning to perform some operation. The operators are same as present in C++/Java.

**Ruby Arrays:**

Array is a structure that holds different type of data in contiguous allocation.

**Ex:**

ary = [ "fred", 10, 3.14, "This is a string", "last element", ]

ary.each do |i|

puts i

end

Output :

fred

10

3.14

This is a string

last element

**Ruby Hashes:**

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

Ex:

hsh = colors = { "red" => 0xf00, "green" => 0x0f0, "blue" => 0x00f }

hsh.each do |key, value|

print key, " is ", value, "\n"

end

Output :

green is 240

red is 3840

blue is 15

**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.

(10..15).each do |n|

print n, ' '

end

Output :

10 11 12 13 14 15

**Branching structures in Ruby:**

**Ruby if...else Statement:**

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

$debug=1

print "debug\n" if $debug

**Ruby unless Statement:**

x=1

unless x>2

puts "x is less than 2"

else

puts "x is greater than 2"

end

Output :

x is less than 2

**Ruby case Statement:**

$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

Output :

little child

**Ruby while Statement:**

$i = 0

while $i < 5 do

puts("Inside the loop i = #$i" )

$i +=1

end

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

$i = 0

$num = 5

begin

puts("Inside the loop i = #$i" )

$i +=1

end while $i < $num

**Ruby until Statement:**

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.

$i = 0

$num = 5

until $i > $num do

print "#$i" + “ ”

$i +=1;

End

Output :

0 1 2 3 4 5

**Ruby for Statement:**

for i in 0..5

print "#{i} "

end

Output :

0 1 2 3 4 5

**Each block :**

(0..5).each do |i|

print "#{i} "

end

Output :

0 1 2 3 4 5

**Ruby next Statement:**

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).

for i in 0..5

if i < 2 then

next

end

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

end

Output :

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 redo Statement:**

Restarts this iteration of the most internal loop, without checking loop condition. Restarts yield or call if called within a block.

for i in 0..5

if i < 2 then

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

redo

end

end

This will produce the following result and will go in an infinite loop:

Value of local variable is 0

Value of local variable is 0 ………………………….

**Exception handling in Ruby :**

The actual code is placed in the *begin* block and the exception handling script is placed in the *rescue* block and the script that has to be executed whether or not the exception is raised should be placed in the *ensure* block.

begin

a = 10, b = 0

a = a/b

rescue Exception => ex

puts “Exception caught : #{ex}”

retry # restart from beginning

ensure

puts “Program execution stopped”

end

**Methods in Ruby :**

Methods are the named block of code that can be invoked as and when required. Ruby method definition starts with *def* keyword followed by method name and arguments (if any) and body of the method and finally closed by *end* keyword.

Ex:

def sum(a, b)

puts “The sum of #a and #b is #{a+b}”

a+b # returns the value of a+b

end

*return* keyword is not mandatory in ruby methods.

**Method returning multiple values :**

Methods in ruby can return any type data and any number of data.

Ex: method returning an array of values

...

def return\_array

a = []

a << 10

a << “data”

a # or return a

end

...

b = []

b = return\_array

puts b # => [10, "data"]

Ex: method returning multiple values

...

def return\_multiple\_values

a = 10.50

b = “some text”

return a, b

end

a1, b1 = return\_multiple\_values

puts a1 # => 10.5

puts b1 # => "some text"

**Method overloading:**

Multiple methods having same name but with different arguments or body are termed as Overloaded methods.

def add(num1=0, num2=0)

num1+num2

end

def add(str1='', str2='')

str1+str2

end

add(50, 50) # => 100

add(“john”, “maclane”) # => johnmaclane

**Object Oriented Programming concepts in Ruby:**

Ruby is a perfect Object Oriented Programming Language. The features of the object-oriented programming language include:

* Data Encapsulation
* Data Abstraction
* Polymorphism
* Inheritance

An object-oriented program involves classes and objects. A class is the blueprint from which individual objects are created.

**Defining a Class in Ruby:**

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

class Customer

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. You 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. 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.

**Custom Method to create Ruby Objects :**

Methods are the named block of code that can invoked as and when required. When the methods are defined under a class then it needs class qualifier (an object or the class name itself) to invoke it.

You can pass parameters to method new and those parameters can be used to initialize class variables.

The initialize method is a special type of method, which will be executed when a new instance is created for the class.

Ex:

class Customer

def initialize(id, name, addr)

@cust\_id=id

@cust\_name=name

@cust\_addr=addr

end

end

In this example, you declare the initialize method with id, name, and addr as local variables. In the initialize method, you pass on the values of these local variables to the instance variables @cust\_id, @cust\_name, and @cust\_addr. Here local variables hold the values that are passed along with the new method.

Now, you can create objects as follows:

cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")

cust2=Customer.new("2", "Poul", "New Empire road, Khandala")

**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 and closed by *end* keyword.

The method names are always preferred in **lowercase letters**.

Ex:

class Sample

def function

statement 1

statement 2

end

end

Now in the following example, create one object of Sample class and call hello method and see the result:

class Sample

def hello

puts "Hello Ruby!"

end

end

object = Sample. new

object.hello

This will produce the following result:

*Hello Ruby!*

**Variables in a Ruby Class:**

Ruby provides four types of variables:

1. **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 \_.

**Ex**:

class Example

VAR1 = 100

VAR2 = 200

def show

puts "Value of first Constant is #{VAR1}"

puts "Value of second Constant is #{VAR2}"

end

end

object=Example.new()

object.show

Output :

Global variable in Class1 is 10

Global variable in Class2 is 10

1. **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.

**Ex**:

class Customer

def initialize(id, name)

@cust\_id=id

@cust\_name=name

end

def display\_details()

puts "Customer id #@cust\_id"

puts "Customer name #@cust\_name"

end

end

cust1=Customer.new("1", "John")

cust2=Customer.new("2", "Poul")

cust1.display\_details()

cust2.display\_details()

Output :

Customer id 1

Customer name John

Customer id 2

Customer name Poul

1. **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 *double at sign* @@ and are followed by the variable name.

**Ex**:

class Customer

@@no\_of\_customers=0

def initialize(id=”123”, name=”abc”) #method with default argument

@cust\_id=id

@cust\_name=name

end

def display\_details()

puts "Customer id #@cust\_id"

puts "Customer name #@cust\_name"

end

def total\_no\_of\_customers()

@@no\_of\_customers += 1

puts "Total number of customers: #@@no\_of\_customers"

end

end

cust1=Customer.new("1", "John")

cust2=Customer.new("2", "Poul”)

cust1.total\_no\_of\_customers()

cust2.total\_no\_of\_customers()

Output :

Total number of customers: 1

Total number of customers: 2

1. **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* ($).

**E**x:

$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

Output :

Global variable in Class1 is 10

Global variable in Class2 is 10

**Class methods :**

Class methods can be accessed only with the class name as qualifier and instance methods can be accessed only by the classes object.

class Accounts

def reading\_charge

puts "instance method"

end

def Accounts.return\_date

puts Time.now

end

end

acc = Accounts.new

acc.reading\_charge # => instance method

acc.return\_date # => NoMethodError: undefined method `return\_date' Accounts.return\_date # => 2014-07-21 17:15:14 +0530

**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.

class Example

VAR1 = 100

def show

# VAR1 += 1 -> SyntaxError: dynamic constant assignment

puts "Value of first Constant is #{VAR1}"

end

end

object=Example.new()

object.show

Output :

Value of first Constant is 100

**Inheritance in Ruby:**

Inheritance is the mechanism by which one class can inherit the properties and methods from another existing class. Ruby supports Single and multi-level inheritance only.

Ex: Single Inhertiance – a class inheriting from a single base class

class Parent

def implicit()

puts "PARENT implicit()"

end

end

class Child < Parent

end

dad = Parent.new()

son = Child.new()

dad.implicit() # => PARENT implicit

son.implicit() # => PARENT implicit

Ex: Multi-level Inhertiance – a class inheriting from a single base class which is inherited from another class

class GrandParent

def implicit()

puts "PARENT implicit()"

end

end

class Parent < GrandParent

end

class Child < Parent

end

dad = Parent.new()

son = Child.new()

dad.implicit() # => PARENT implicit

son.implicit() # => PARENT implicit

**Method overriding:**

When a same method (may be with different body) is present in both base and derived classes and when the derived class object calls that method then the derived class method will be invoked (instead of the base call method), which means the base class method is overridden and this is called as **Method overriding**.

class Parent

def override()

puts "PARENT override()"

end

end

class Child < Parent

def override()

puts "CHILD override()"

end

end

dad = Parent.new()

son = Child.new()

dad.override() # => PARENT override

son.override() # => CHILD override

**super() method in Ruby:**

class Parent

def altered()

puts "PARENT altered()"

end

end

class Child < Parent

def altered()

puts "CHILD, BEFORE PARENT altered()"

super()

puts "CHILD, AFTER PARENT altered()"

end

end

dad = Parent.new()

son = Child.new()

dad.altered()

son.altered()

Output:

PARENT altered()

CHILD, BEFORE PARENT altered()

PARENT altered()

CHILD, AFTER PARENT altered()

**Ruby Modules and Mixins:**

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.

module Trig # saved as trig.rb

PI = 3.141592654

def Trig.sin(x)

Math.sin x

end

def Trig.cos(x)

Math.cos x

end

end

**Ruby 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:

require 'trig'

y = Trig.sin(Trig::PI/4)

**Ruby include Statement:**

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

require 'trig'

class Maths

include Trig

def use

puts Week::sin(60) # => -0.3048106211022167

puts Week::cos(60) # => -0.9524129804151563

end

end

**Mixins in Ruby:**

Ruby does not support multiple inheritance directly but Ruby Modules have another wonderful use. At a stroke, they pretty much eliminate the need for multiple inheritance, providing a facility called a mixin.

module A

def a1

end

def a2

end

end

module B

def b1

end

def b2

end

end

class Sample

include A

include B

def s1

end

end

samp=Sample.new

samp.a1

samp.a2

samp.b1

samp.b2

samp.s1

**Strings in Ruby:**

String object holds and manipulates an arbitrary sequence of bytes, typically representing characters. String objects may be created using String::new or as literals.

Some of the commonly used string methods in ruby :

Message = “This ia a test message” #=> This ia a test message

Puts 10.**to\_s** # => 10

a = "hello "

a << "world" #=> "hello world"

a.**concat**(33) #=> "hello world!"

a = "hello there"

a[1] #=> "e"

a[2, 3] #=> "llo"

a[2..3] #=> "ll"

"hello".**capitalize** #=> "Hello"

"abcdef".**casecmp**("abcde") #=> 1

"aBcDeF".**casecmp**("abcdef") #=> 0

"abcdef".**casecmp**("abcdefg") #=> -1

"abcdef".**casecmp**("ABCDEF") #=> 0

"hello\n".**chomp** #=> "hello"

"hello".**chomp**("llo") #=> "he"

"hello".**delete** "l","lo" #=> "heo"

"hello".**delete** "lo" #=> "he"

"hEllO".**downcase** #=> "hello"

"hEllO".**upcase** #=> "HELLO"

"hello".**empty**? #=> false

" ".**empty**? #=> false

"".**empty**? #=> true

"hello".**gsub**("h", 'H') #=> "Hello"

"hello".**gsub**(/[aeiou]/, '\*') #=> "h\*ll\*"

"hello".**include**? "lo" #=> true

"hello".**index**('e') #=> 1

"abcd".**insert**(3, 'X') #=> "abcXd"

"abcd".**insert**(4, 'X') #=> "abcdX"

"stressed".**reverse** #=> "desserts"

string = "this is a string"

string.**slice**!(2) #=> "i"

string.**slice**!(3..6) #=> " is "

string.**slice**!(/s.\*t/) #=> "sa st"

string.**slice**!("r") #=> "r"

string #=> "thing"

" now's the time".**split**(' ') #=> ["now's", "the", "time"]

"1,2,3,4".**split**(',') #=> ["1", "2", "3", "4"]

" hello ".**strip** #=> "hello"

"\tgoodbye\r\n".**strip** #=> "goodbye"