**What is pure OOP language?**

Smalltalk is a "pure" object-oriented programming language, meaning that, unlike Java and C++, there is no difference between values which are objects and values which are primitive types. In Smalltalk, primitive values such as integers, Booleans and characters are also objects, in the sense that they are instances of corresponding classes, and operations on them are invoked by sending messages. A programmer can change the classes that implement primitive values, so that new behavior can be defined for their instances-for example, to implement new control structures-or even so that their existing behavior will be changed. This fact is summarized in the commonly heard phrase "In Smalltalk everything is an object", which may be more accurately expressed as "all values are objects", as variables are not.   
  
Since all values are objects, classes themselves are also objects. Each class **is an instance of the metaclass of that class.** Metaclasses in turn are also objects, and are all instances of a class called Metaclass. Code blocks are also objects.

### Seeing Everything as an Object

**In Ruby**, everything is an object. Every bit of information and code can be given their own properties and actions. Object-oriented programming calls properties by the name *instance variables* and actions are known as *methods*. Ruby’s pure object-oriented approach is most commonly demonstrated by a bit of code which applies an action to a number.

5**.**times { print "We \*love\* Ruby -- it's outrageous!" }

Here no 5 is used as object.

In many languages, numbers and other primitive types are not objects.Ruby follows the influence of the Smalltalk language by giving methods and instance variables to all of its types. This eases one’s use of Ruby, since rules applying to objects apply to all of Ruby.

Ruby’s Flexibility

Ruby is seen as a flexible language, since it allows its users to freely alter its parts. Essential parts of Ruby can be removed or redefined, at will. Existing parts can be added upon. Ruby tries not to restrict the coder.

For example, addition is performed with the plus (+) operator. But, if you’d rather use the readable word plus, you could add such a method to Ruby’s built-in Numeric class.

**class** **Numeric**

**def** **plus**(x)

self**.**+(x)

**end**

**end**

y **=** 5**.**plus 6

# y is now equal to 11

Ruby features OS independent threading. Thus, for all platforms on which Ruby runs, you also have multithreading, regardless of if the OS supports it or not, even on MS-DOS!

Ruby is highly portable: it is developed mostly on GNU/Linux, but works on many types of UNIX, Mac OS X, Windows 95/98/Me/NT/2000/XP, DOS, BeOS, OS/2, etc.

Ruby is simple in appearance, but is very complex inside, just like our human body-Matz

**Ruby is server side scripting language.**

**Something about scripting languages**

A high-level programming language that is interpreted by another program at runtime rather than compiled by the computer's processor as other programming languages (such as C and C++) are. Scripting languages, which can be embedded within HTML, commonly are used to add functionality to a Web page, such as different menu styles or graphic displays or to serve dynamic advertisements. These types of languages are client-side scripting languages, affecting the data that the end user sees in a browser window. Other scripting languages are server-side scripting languages that manipulate the data, usually in a database, on the server.

Scripting languages came about largely because of the development of the Internet as a communications tool. JavaScript, ASP, JSP, PHP, Perl, Tcl and Python are examples of scripting languages.

## 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 rich set of built-in functions, which can be used directly into Ruby scripts.

## Ruby *BEGIN and END* Statement

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

END: 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

|  |  |
| --- | --- |
| **Reserved word** | **Description** |
| **BEGIN** | Code, enclosed in { }, to run before the program runs. |
| **END** | Code, enclosed in { }, to run when the program ends. |
| **alias** | Creates an alias for an existing method, operator, or global variable. |
| **and** | Logical operator; same as && except and has lower precedence. |
| **begin** | Begins a code block or group of statements; closes with end. |
| **break** | Terminates a while or until loop, or a method inside a block. |
| **case** | Compares an expression with a matching when clause; closes with end. See when. |
| **class** | Defines a class; closes with end. |
| **def** | Defines a method; closes with end. |
| **Defined?** | A special operator that determines whether a variable, method, super method, or block exists. |
| **do** | Begins a block, and executes code in that block; closes with end. |
| **else** | Executes following code if previous conditional is not true,set with if, elsif, unless, or case. See if, elsif. |
| **elsif** | Executes following code if previous conditional is not true,set with if or elsif. |
| **end** | Ends a code block (group of statements) started with begin, class, def, do, if, etc. |

|  |  |
| --- | --- |
| **ensure** | Always executes at block termination; use after last rescue. |
| **false** | Logical or Boolean false; instance of False Class; a pseudo variable. See true. |
| **for** | Begins a for loop; used with in. |
| **if** | Executes code block if conditional statement is true. Closes with end. Compare unless, until. |
| **in** | Used with for loop. See for. |
| **module** | Defines a module; closes with end. |
| **next** | Jumps to the point immediately before the evaluation of the loop’s conditional. Compare redo. |
| **nil** | Empty, uninitialized, or invalid; always false, but not the same as zero; object of NilClass; a pseudovariable. |
| **not** | Logical operator; same as !. |
| **or** | Logical operator; same as || except or has lower precedence. |
| **redo** | Jumps after a loop’s conditional. Compare next. |
| **rescue** | Evaluates an expression after an exception is raised; used before ensure. |
| **retry** | When called outside of rescue, repeats a method call;inside rescue, jumps to top of block (begin). |
| **return** | Returns a value from a method or block. May be omitted,but method or block always return a value, whether it is explicit or not. |
| **self** | Current object (receiver invoked by a method); a pseudo variable. |
| **super** | Calls method of the same name in the superclass. The superclass is the parent of this class. |
| **then** | Separator used with if, unless, when, case, and rescue. May be omitted, unless conditional is all on one line. |
| **true** | Logical or Boolean true; instance of TrueClass; a pseudovariable. See false. |
| **undef** | Makes a method undefined in the current class. |
| **unless** | Executes code block if conditional statement is false.Compare if, until. |
| **until** | Executes code block while conditional statement is false.Compare if, unless. |
| **when** | Starts a clause (one or more) under case. |
| **while** | Executes code while the conditional statement is true. |
| **yield** | Executes the block passed to a method. |
| **\_ \_FILE\_ \_** | Name of current source file; a pseudovariable. |
| **\_ \_LINE\_ \_** | Number of current line in the current source file; a pseudovariable. |

**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

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

**Operators**

## 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. 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 ($).

# Difference between class variables and class instance variables?

A class variable (@@) is shared among the class and all of its descendants. A class instance variable (@) is not shared by the class's descendants.

**Class variable (@@)**

Let's have a class Foo with a class variable @@i, and accessors for reading and writing @@i:

class Foo

@@i = 1

def self.i

@@i

end

def self.i=(value)

@@i = value

end

end

And a derived class:

class Bar < Foo

end

We see that Foo and Bar have the same value for @@i:

p Foo.i # => 1

p Bar.i # => 1

And changing @@i in one changes it in both:

Bar.i = 2

p Foo.i # => 2

p Bar.i # => 2

**Class instance variable (@)**

Let's make a simple class with a class instance variable @i and accessors for reading and writing @i:

class Foo

@i = 1

def self.i

@i

end

def self.i=(value)

@i = value

end

end

And a derived class:

class Bar < Foo

end

We see that although Bar inherits the accessors for @i, it does not inherit @i itself:

p Foo.i # => 1

p Bar.i # => nil

We can set Bar's @i without affecting Foo's @i:

Bar.i = 2

p Foo.i # => 1

p Bar.i # => 2

**Numbers in Ruby**

Numbers are not primitives; each number is an object, an instance of one of Ruby’s numeric classes. **Numeric** is Ruby’s base class for numbers.

* **Fixnum class**: is used for integers, fixed-length numbers with bit lengths of the native machine word, minus 1.
* **Float class**: is for floating-point numbers, which use the native architecture’s double-precision floating-point representation internally.
* **Bignum class**: is used to hold integers larger than Fixnum can hold. Bignums are created automatically if any operation or assignment yields a result too large for Fixnum.

## Ruby 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

## Ruby Ternary operator:

There is one more operator called 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 |

## Ruby 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 |

## Ruby defined? operators:

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)

## Usage 3

# True if a method exists that can be called with super user

defined? super

For Example:

defined? super # => "super" (if it can be called)

defined? super # => nil (if it cannot be)