Brief Summary of Ruby Language

Methods

Every procedure in Ruby is a method of some object. Some method calls appear to be function calls as in other languages, but in fact they are actually invocations of methods belonging to self. Parentheses can be omitted if unambiguous.

```
"string".size()              # call method of "string"
"string".size                # parentheses can be omitted.
print "hello world\n"        # call `print' of self
```

As in Smalltalk, in Ruby operators are special forms of method calls. Consider this, for example:

```
1 + 1                        # invoke + method with argument 1
ary[0]           # i.e. `ary.[](0)'
ary[0] = 55                  # i.e. `ary.[]=(0,55)'
```

Unlike in Smalltalk, operators follow usual operator precedence of programming languages, thus:

```
1 + 2 * 3 == 1 + (2 * 3)
```

To define methods, use def statements.

```
def foo(arg1, arg2)
  print arg1, arg2, "\n"
End
```

A def statement at the top level defines a global method that can be used just like a function. Usual methods can be defined by def statements within class definition statements.

```
class Foo
  def foo
    print "foo method in Foo class\n"
  end
end
```

Blocks

Blocks in Ruby are something in between CLU's iterators and Smalltalk's blocks. Like CLU's iterator, a block is a special syntax to pass extra procedural information to the method. For example:

```
10.times do |i|
  print i, "\n"
end
```

Here, 10.times is a method invocation, with a block starting from "do" to "end." The times method of Integer class evaluates a block 10 times with incremental values from zero. Values given to the block are assigned to the variable surrounded by || for each iteration.

In this code, do .. end can be replaced by ....

As in Smalltalk, blocks are used not only for loop abstraction, but also for conditionals, callbacks, and so on:

```
[1,2,3].collect{|x| x*2}     # => [2,4,6]
 [1,2,3].delete_if{|x| x == 1}      # => [2,3]
 [1,2,3].sort{|a,b| b<=>a}    # => [3,2,1]
 "abc".sub(/[ac]/){|x|x.upcase}# => "AbC"
```

Blocks in Ruby are not first-class objects; rather, you can objectify them explicitly. Objectified blocks are sometimes are called closures because they wrap code blocks along with local variable bindings:

```
c = lambda{|x| x + 2}              # objectify block, returns closure
 c.call(5)                 # => 7, evaluate closure
```

The word *lambda* is traditionally used in the Lisp world for the function to make closure. Ruby provides the plain alias proc, too.

No Declarations

A variable in Ruby programs can be distinguished by the prefix of its name. Unlike in Perl, the prefix denotes the scope of the variable. It removes declarations from Ruby to let you code less. In addition, by knowing variable scope at a glance enhances readability.

Scope	**Prefix**	**Example**
global variable	$	$global
instance variable	@	@instance
local variable	lowercase or _ local	
constant	uppercase	Constant
class variable	@@	@@class_var (since 1.5.3)

Local variable scope can be introduced by the following statement: class, module, def, and blocks. Scopes introduced by class, module, and def do not nest. But from blocks, local variables in the outer scope can be accessed. Local variables in Ruby follow lexical scoping.

Differences from other languages

Some features that differ notably from languages such as C, Perl:

- The language syntax is sensitive to the capitalization of identifiers, in all cases treating capitalized variables as constants. Class and module names are constants and refer to objects derived from Class and Module.
- The sigils $ and @ do not indicate variable role as in Perl, but rather function as scope resolution operators.
- Floating point literals must have digits on both sides of the decimal point: neither .5 nor 2. are valid floating point literals, but 0.5 and 2.0 are.
- (In Ruby, integer literals are objects that can have methods apply to them, so requiring a digit after a decimal point helps to clarify whether 1.e5 should be parsed analogously to 1.to_f or as the exponential-format floating literal 1.0e5. The reason for requiring a digit before the decimal point is less clear; it might relate either to method invocation again, or perhaps to the .. and ... operators, for example in the fragment 0.1...3.)
- Boolean non-boolean datatypes are permitted in boolean contexts (unlike in e.g. Smalltalk and Java), but their mapping to boolean values differs markedly from some other languages: 0 and "empty" (e.g. empty list, string or associative array) all evaluate to *true*, thus changing the meaning of some common idioms in related or similar languages such as Lisp, Perl and Python.
- A consequence of this rule is that Ruby methods by convention — for example, regular-expression searches — return numbers, strings, lists, or other non-*false* values on success, but nil on failure.

Differences from C++

- There's no explicit references. That is, in Ruby, every variable is just an automatically dereferenced name for some object.
- Objects are strongly but *dynamically* typed. The runtime discovers *at runtime* if that method call actually works.
- The "constructor" is called initialize instead of the class name.
- All methods are always virtual.
- "Class" (static) variable names always begin with @@ (as in @@total_widgets).
- You don't directly access member variables—all access to public member variables (known in Ruby as attributes) is via methods.
- It's self instead of this.
- Some methods end in a '?' or a '!'. It's actually part of the method name.
- There's no multiple inheritance per se. Though Ruby has "mixins" (i.e. you can "inherit" all instance methods of a module).

- There are some enforced case-conventions (ex. class names start with a capital letter, variables start with a lowercase letter).
- Parentheses for method calls are usually optional.
- You can re-open a class anytime and add more methods.
- There's no need of C++ templates (since you can assign any kind of object to a given variable, and types get figured out at runtime anyway). No casting either.
- Iteration is done a bit differently. In Ruby, you don't use a separate iterator object (like vector<T>::const_iterator iter). Instead you use an iterator method of the container object (like each) that takes a block of code to which it passes successive elements.
- There's only two container types: Array and Hash.
- There's no type conversions. With Ruby though, you'll probably find that they aren't necessary.
- Multithreading is built-in, but as of Ruby 1.8 they are "green threads" (implemented only within the interpreter) as opposed to native threads.
- A unit testing lib comes standard with Ruby.

Syntax

The sentence structure of Ruby is comprehensively like that of Perl and Python. Class and technique definitions are motioned by watchwords, though code pieces can be both characterized by catchphrases or supports. Rather than Perl, factors are not compulsorily prefixed with a sigil. Whenever utilized, the sigil changes the semantics of extent of the variable. For viable purposes there is no qualification amongst expressions and statements. Line breaks are huge and taken as the finish of an announcement; a semicolon might be comparably utilized. Not at all like Python, space is not noteworthy.

The sentence structure of Ruby is comprehensively like that of Perl and Python. Class and technique definitions are motioned by watchwords, though code pieces can be both characterized by catchphrases or supports. Rather than Perl, factors are not compulsorily prefixed with a sigil. Whenever utilized, the sigil changes the semantics of extent of the variable. For viable purposes there is no qualification amongst expressions and statements. Line breaks are huge and taken as the finish of an announcement; a semicolon might be comparably utilized. Not at all like Python, space is not noteworthy.

Semantics

The sentence structure of Ruby is comprehensively like that of Perl and Python. Class and technique definitions are motioned by watchwords, though code pieces can be both characterized by catchphrases or supports. Rather than Perl, factors are not compulsorily prefixed with a sigil. Whenever utilized, the sigil changes the semantics of extent of the variable. For viable purposes there is no qualification amongst expressions and statements. Line breaks are huge and taken as the finish of an announcement; a semicolon might be comparably utilized. Not at all like Python, space is not noteworthy.

Ruby has been depicted as a multi-worldview programming dialect: it permits procedural programming (characterizing capacities/factors outside classes makes them a player in the

root, "self" Question), with protest introduction (everything is a question) or useful programming (it has mysterious capacities, terminations, and continuations; explanations all have values, and capacities give back the last assessment). It has bolster for thoughtfulness, reflection and metaprogramming, and in addition bolster for translator based strings. Ruby components dynamic writing, and backings parametric polymorphism.

Design Policy of Ruby

For me, the purpose of life is, at least partly, to have joy. Programmers often feel joy when they can concentrate on the creative side of programming, so Ruby is designed to make programmers happy. I consider a programming language as a user interface, so it should follow the principles of user interface.

Principle of Conciseness

I want computers to be my servants, not my masters. Thus, I'd like to give them orders quickly. A good servant should do a lot of work with a short order.

Principle of Consistency

As with uniform object treatment, as stated before, a small set of rules covers the whole Ruby language. Ruby is a relatively simple language, but it's not *too* simple. I've tried to follow the principle of "least surprise." Ruby is not too unique, so a programmer with basic knowledge of programming languages can learn it very quickly.

Principle of Flexibility

Because languages are meant to express thought, a language should not restrict human thought, but should help it. Ruby consists of an unchangeable small core (that is, syntax) and arbitrary extensible class libraries. Because most things are done in libraries, you can treat user-defined classes and objects just as you treat built-in ones.

Individual Methods and Prototype-Based Programming

Although Ruby is classified as a class-based object-oriented language, it allows you to define methods for individual objects.

```
foo = Object.new              # create an Object
 def foo.method1              # add a method to it
  print "method1\n"
 end
 foo.method1                  # prints "method1\n"
```

In addition, it allows prototype-based object-oriented programming:

```
bar = foo.clone          # make new object after prototype
 bar.method1             # `clone' copies individual method too
 def bar.method2         # new method added
  print "method2\n"
 end
 bar.method2             # prints "method2\n"
```

Exceptions

As a modern language, Ruby supports exceptions. All methods in the class library raise exceptions for unusual status. Ruby allows the programmer to omit most error detecting code, and it frees the programmer from worrying about execution under unsatisfied preconditions.

Exceptions are raised by the raise method:

```
raise                          # raise unnamed RuntimeError exception
raise "error!"                 # raise RuntimeError with message "error!"
raise IOError, "closed"        # raise IOError with message "closed"
```

Exceptions are caught by the rescue clause of the begin statement.

```
begin
  codeA
rescue SomeError, AnotherError
  codeB
end
```

If codeA raises an exception, which is a subclass of either SomeError or AnotherError, the exception is caught and codeB is executed. All exceptions are subclasses of the Exception class.

Threads

Ruby supports user-level threading. It's implemented by using setjump()/longjmp() and stack copying. It is not efficient and cannot utilize multiple CPUs. It is also several percents slower than sequential execution. However, it works on all platforms, including MS-DOS machines. Because Ruby behaves uniformly on all platforms, programmers need not worry about threading compatibility. Threading often improves programs' response time by making heavy procedures run in the background.

To start a new thread, you simply call Thread::new with a block:

```
Thread::new do
  # code to executed in a thread
End
```

Garbage Collection

The reference counting technique used by Python does not recycle circular references, so programmers in this language must cut circular references explicitly later. Ruby uses a conservative mark-and-sweep method of garbage collection, which frees programmers from worrying about circular references. The garbage collector scans C stack and registers as well,

so that C/C++ written extensions do not have to maintain refcounts or protect local variables. This makes extension development easier than it is for other languages. There is no INCREF/DECREF and no mortal.

Conclusion: Solving Everyday Tasks

Ruby includes many preferable features that help programmers enjoy programming. This language is good for a wide range of problem domains, from text-processing one-liners to the full-featured GUI mail user agent. Hopefully Ruby will fit into your tool box and help you. The Ruby community is growing very rapidly; I invite you to join in.

References

- http://www.ruby-lang.org/—You should be able to access everything about Ruby from this site—it is the Ruby home page.
- http://www.ruby-lang.org/en/raa.html —This is the Ruby application archive, the CPAN equivalent to be.
- http://www-4.ibm.com/software/developer/library/ruby.html—This IBM article discusses the significance of Ruby.
- http://blade.nagaokaut.ac.jp/ruby/ruby-talk/index.shtml—You can access the Ruby talk mailing list archive from here.
- http://www.ruby-lang.org/en/man-1.4/index.html—An online reference manual is available here.
- http://www.pragprog.com:8080/rubyfaq/—Read through the Ruby FAQ at this site.