By Aman King

The author is Developer, ThoughtWorks He can be reached at king@thoughtworks.com.

Ruby:

A Gem of a Language!

About 3 years back, a search on the Internet for 'Ruby' would have shown you links about the gemstone, but today you won't find a link to a website dealing with precious stones in the first few pages of the search results. So what is this new 'Ruby' all about?

Introduction

Ruby is a programming language created by Yukihiro 'Matz' Matsumoto, released in 1995. Its birth had its origins in certain values and principles that were held dear by its creator. The most important of these are "Programming should be fun" and "Focus on humans, not machines."

Matz describes his language as, "Ruby is an interpreted scripting language for quick and easy object-oriented programming. It may seem a little strange at first, but it is designed to be easily read and written."

More formally, Ruby is an interpreted, reflective, object-oriented, open-source, and general-purpose programming language. It is developed from various languages such as Smalltalk, Perl, Python, Lisp, and Ada.

So, why is Ruby catching attention? Some reasons are:

- Word of mouth: Ruby is the most talked about programming language in the tech circles.
- It is easy to start learning Ruby.
- It is fun and powerful.
- Ruby has easy portability as interpreters are available for most platforms.
- And most important of all, Ruby on Rails is a quick and viable Ruby-based alternative to existing Web frameworks

Let's delve further into what makes Ruby special, bringing it out of the hobbyists' realms into real world projects.

Ruby Principles

Personally I feel Ruby changes the way you think about programming. It brings to the forefront ideas that were realized sometime back but have become popular only now. As Kent Beck says, "I always thought Smalltalk would beat Java, I just didn't know it would be called 'Ruby' when it did." Ward Cunningham adds, "For me, Ruby is the successful combination of Smalltalk's conceptual elegance, Python's ease of use and learning, and Perl's pragmatism."

Programming Should be Fun

Matz believes choosing a programming language should be about how you feel while programming in it. He says, "Programmers often feel joy when they can concentrate on the creative side of programming, so Ruby is designed to make programmers happy." This is aided by features like concise syntax, powerful regular expressions, closures, and so on.

Focus on Humans, not Machines

"Often people focus on the machines, as they think, by doing this, the machine will run faster. By doing this, the machine will run more effectively," Matz says. He continues, "But in fact we need to focus on humans. We are the masters. They are the slaves." Ruby allows for this by being portable, doing behind-the-scenes optimization, and by remaining free from machine limitations.

The simplest example is of how a language deals with numbers. In most languages numbers are explicitly tied to a datatype which has a minimum and a maximum value depending on occupied bytes. In Ruby, programmers can treat a number like a number, without worrying about datatypes and their bounds: Ruby transparently manages datatypes for numbers. For example, 1 is Fixnum but 9999999999 is Bignum.

Code that is Easy to Read and Write

Ruby lends itself to communicative syntax. It almost reads like English, making it a good choice for implementing internal Domain Specific Languages (languages targeted at specific problem domains, usually specifying what to do rather than how to do).

```
Example:
if door.is_closed?
  4.times { door.knock } until door.is_open?
  unless door.leads_to_hell_man!
end
```

Principle of Flexibility

Matz says, "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 arbitrarily extensible class libraries." You can treat built-in classes like user defined classes and extend them as and when required.

Ruby Features

Strong OO

Everything is an object in Ruby! This includes what are usually primitive datatypes in other languages (like int, char, etc in Java or C++). Ruby pushes object orientation to its limits, respecting objects more than classes. This is evident from concepts like singleton methods (methods defined on an object instance rather than its class), private members being inaccessible to other instances of the same class and duck-typing.

Duck-typing

"If it walks like a duck, talks like a duck, it is a duck" – that's Duck-typing. In other words, any object, irrespective of its class, can be used without problem in places where certain methods are invoked on it as long as it responds to those methods. There are no class-based restrictions.

```
Example:
class MyDuck
  def quack(); puts "Quack!"; end
end

class YourDuck
  def quack(); puts "Quack quack!"; end
end
  def make_it_quack(duck)
  duck.quack
end

make_it_quack(MyDuck.new)     # => Quack!
make_it_quack(YourDuck.new) # => Quack quack!
```

Messages, not Function Calls

In Ruby, method calls on an object are treated as messages sent to the object, not function table lookups. So if a method is called on an object that doesn't have that method defined, it is not an error but simply a message sent to an object that doesn't know how to deal with it; in such a case, Ruby redirects the message to method missing(), a method defined in the Object class to raise a NoMethodError. Overriding this method proves very helpful when designing proxies, filters and mock objects.

```
Example:
class Proxy
  def initialize(); @objects = Array.new; end
  def add(obj); @objects << obj; end
  def method_missing(method, *args, &block)
    @objects.each do |obj|
 # loop over @objects array
      if obj.methods.include?(method.to_s)
        return obj.send(method, *args, &block)
 # pass on the method invocation to obj
  end
  end
    raise NoMethodError, "No object in proxy for
message '#{method.to_s}'"
```

```
   end
end

class HelloWorld
  def hello(); puts "hello world"; end
end
proxy = Proxy.new
proxy.add("AMAN")
proxy.add(HelloWorld.new)

puts proxy.downcase       # => aman
proxy.hello               # => hello world
puts proxy.bye            # => NoMethodError
```

Closures

A block of code can be passed on as an argument like any other object (this is 'closure'). These blocks are evaluated, however, in the context of the calling scope. This helps implement behavior that uses different objects' data without breaking encapsulation, and also helps distribute responsibilities properly (for example, how-to-loop logic in a collection class and processing logic in a domain class).

```
Example:
class OddEvenArray
  def initialize(contents); @contents = con-
tents; end

  def for_every(position_type, &block)
  index = (position_type == :even) ? 0 : 1
    while(index < @contents.length)
    block.call(@contents[index])
    index+=2
    end
  end
end

players = ["Ram", "John", "Aamir", "Vivek"]
even_numbered_tshirts = [ ]
OddEvenArray.new(players).for_every(:even)
{ |even_player| even_numbered_tshirts <<
even_player }

# the block above (within { }) refers local
variable 'even_numbered_tshirts' but gets exe-
cuted by
# OddEvenArray instance's for_every() method,
 which provides 'even_player' value one at a
time
puts even_numbered_tshirts  # => "Ram", "Aamir"
```

Meta-programming

Ruby enables meta-programming (programs that modify programs) in various ways.

```
Examples:
Open classes (redefining an already loaded
class' behaviour at runtime)
4.factorial # => NoMethodError, ie, 'factorial'
isn't defined yet on 4's class: Fixnum
class Fixnum
  def factorial()
    res = 1
    counter = self
    while(counter >= 1); res *= counter;
    counter -= 1; end
    res # returning res; in Ruby, return is
    implied for a method's last statement
  end
end
4.factorial # => 24
```

Eval methods (methods that every class has, to evaluate code against its context)

```
class Object
  def self.define_getters(fields)
  # defines class-level method on Object
    fields.each { |field| class_eval("def
    get_#{field}; @#{field}; end") }
 # dynamically adds instance methods
  end
end

class Person
  define_getters ["name", "age"]
  def initialize(name, age)
  ;@name, @age = name, age
  ;end
end
Person.new("Aman", 25).get_name # => "Aman"
```

Mixins

Mixins help avoid code duplication without tangly class hierarchies. Related behavior can be grouped into Modules that can be included in different classes.

```
module Flyer
  def fly(); puts "#{@name} is flying"; end
end
module Swimmer
  def swim(); puts "#{@name} is swimming"; end
end
class Duck
  def initialize(); @name = "Duck"; end
  include Flyer
  include Swimmer
end
class Crow
  def initialize(); @name = "Crow"; end
  include Flyer
end
duck = Duck.new
duck.fly # => Duck is flying
duck.swim # => Duck is swimming
Crow.new.fly # => Crow is flying
```

Conclusion

There are many more features of Ruby that you can discover as you learn, amazing stuff that make 'magical' frameworks like Rails possible. Hope, this quick introduction has succeeded in piquing your interest. Dynamic languages are an exciting new world, especially for people coming from static languages.

For more, visit http://www.ruby-lang.org/ and http://en.wikipedia.org/wiki/Ruby_(programming_language)