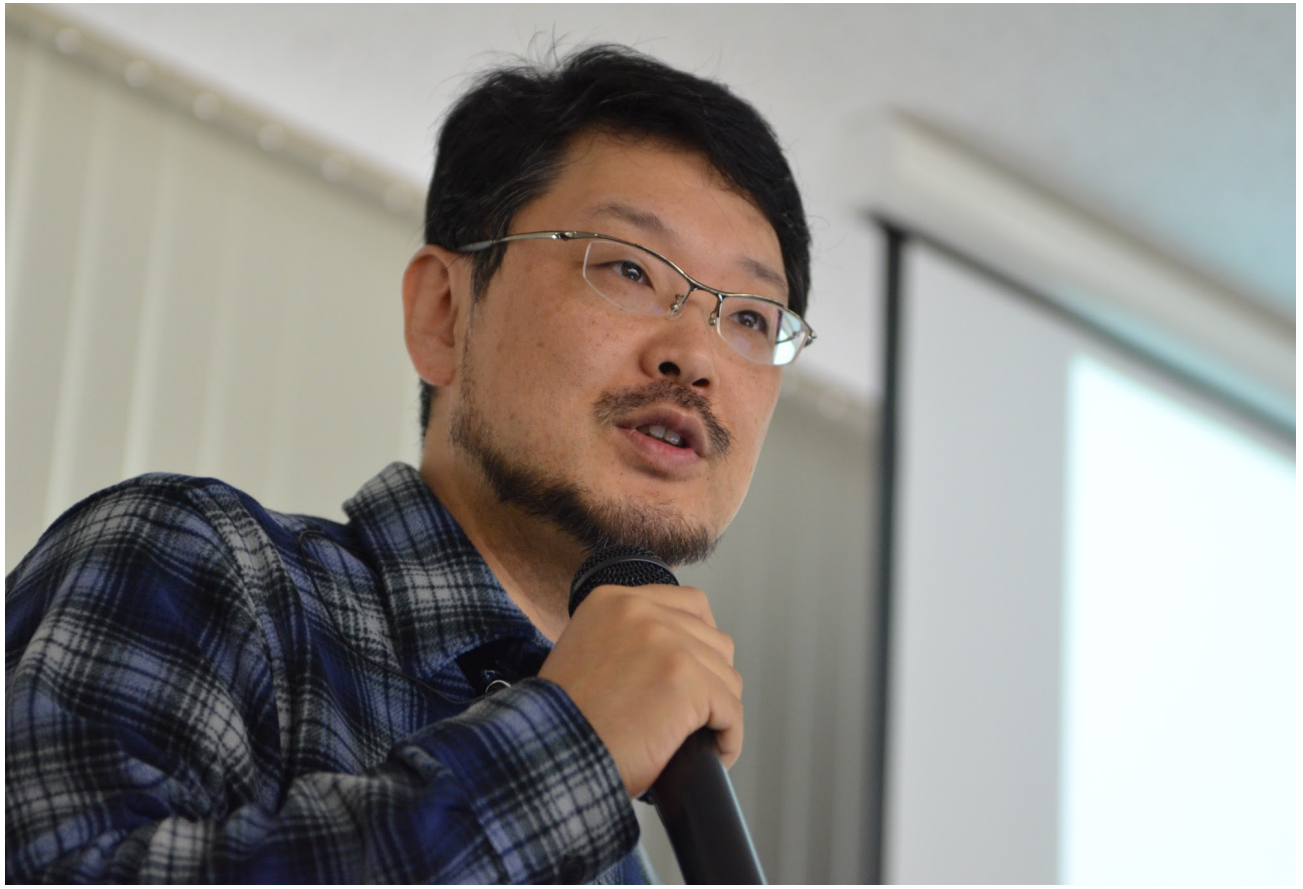
RUBY

Yukihiro Matsumoto



“Ruby is simple in appearance, but is very complex inside, just like our human body."

Matz : Programming Difficult?

3.times { print ‘awesome’ }

History

1994

2000

2005

TODAY

FUTURE

Interpreted

MRI – Matz Ruby Interpreter

OOPS

its PURE!

IRB vs CL

InteractiveRuby vs CommandLine

Variables

Any Variable = Any Data

Variables

- Always starts with lowercase letter
- Have no spaces
- No special characters in between

Variables

- BAD Practice
  - camelCase
  - 1st_lesson
  - students_array
  - pts
- GOOD Practice
  - snake_case
  - first_lesson
  - students
  - points

Variables

Types:

- Constant
- Global
- Class
- Instance
- Local

Variables

Constant

MY_CONSTANT = “I am available throughout your app.”

Variables

Global

$i_am_global = “I am also available throughout your app.”

Variables

Class
@@class_variable = 0

Variables

Instance

@instance_var = “I am available throughout current instance.”

Variables

Local
i_am_local = “I obey scopes”

Local Variable

What is the output of this code?

```
def change
   x = 5
end

x = 8
change
puts x
```

global

```
$x = 42

def change
  $x = 8
end

change
puts $x
```

Variables

puts
- print stuff to screen with newline character at end

print
- print stuff to screen without newline character at end

Variables

Taking input from user

gets

gets.chomp

Exercise

- Write a program called name.rb that asks the user to type in their name and then prints out a greeting message with their name included.

- Write a program called age.rb that asks a user how old they are and then tells them how old they will be in 10, 20, 30 and 40 years. Below is the output for someone 20 years old.

- Modify name.rb again so that it first asks the user for their first name, saves it into a variable, and then does the same for the last name. Then outputs their full name all at once.

self-assignment

```
x += y  # x=x+y
x -= y  # x=x-y
```

Parallel Assignment

x, y, z = 10, 20, 30

a, b = b, a (swapping)

Operator Precedence

PEMDMAS

P=parenthesis,E=exponential,M=multiplication,D=division,M=Modulus,A=addition,S=subtraction

Exercise

$$5 - 2 * 3 + 3$$

Strings

‘Single Letter | Word | Sentence’

‘ vs “

Singel quotes inside single quotes

Double quotation marks can also include the \n escape sequence

Concatenation

Strings can be joined using the + in a process called concatenation

Strings

- Methods
  - length
  - split
  - sub | gsub
  - upcase | downcase | capitalize
  - include?

Exercise

BANG

!

Numbers

- Integers
- Floats

Numbers

- Methods
  - zero?
  - round
  - eql?
  - round | ceil | floor
  - odd? | even?
  - positive? | negative?

Exercise

Flow Control - Conditionals

IFTTT – If This Then That

Logical Operators

Comparison : <=, <, >, >=
Equality : ==, !=
Logical : && (AND), || (OR), ! (NOT)

If

It executes code when a conditional is TRUE.

```
If <condition>
    # statement
end
```

if-else-end

```
if <condition>
    # statement
else
    # statement
end
```

if-elsif-else-end

```
if <condition>
    # statement
elsif <condition>
    # statement
else
    # statement
end
```

unless

The unless expression is the opposite of an if expression. It executes code when a conditional is false.

```
unless <condition>
    # statement
else
    #statement
end
```

single line condition

```
puts "greater" if 20 > 10

puts "not equal" unless 10 == 20
```

Flow Control - Tertiary

<condition> ? true : false

Flow Control - Case Statement

combination of case, when, else, and end

Exercise

- Write a program that takes a number from the user between 0 and 100 and reports back whether the number is between 0 and 50, 51 and 100, or above 100.

- Rewrite your program using a case statement. Wrap the statement from exercise 3 in a method and wrap this new case statement in a method. Make sure they both still work.

Loops & Iterators – loop

```
loop do
    # something
end
```

```
loop {
    # something
}
```

infinite

Loops & Iterators - Controlling

```
loop do
    # something
    break
end
```

```
loop {
    # something
    break on condition
}
```

Loops & Iterators - skip

```
loop {
    # something
    next on condition
 }
```

Loops & Iterators

```
while condition
    # something
end
```

Loops & Iterators - until

```
until condition
    # something
end
```

Opposit of while

Loops & Iterators - for

```
for i in collection do
    # something
end
```

Loops & Iterators - each

```
collection.each do | i |
    # something
end
```

Exercise

- Write a looping program that takes input from the user, performs an action, and only stops when the user types "STOP". Each loop can get info from the user.

ranges

A range represents a sequence.

".." - inclusive range
"..." - excludes the specified high value

(1..6).to_a
(78...93).to_a
(‘a’..’d’).to_a

Array

List of heterogeneous elements

[, ,]

Array

Access array element using index value
puts a[0]
puts a[0..1]

Set array element using index value
a[0] = ‘new value’

Array

Inserting

- Push()
- <<
- insert(index,value)

Array

Removing Element

- pop
- delete(element)
- delete_at(position)
- shift

Array

- Methods
  - first | last
  - count | length
  - index()
  - include?
  - sort
  - uniq
  - reverse
  - freeze

Array Manipulation - set

array + array
array – array
array & array
array | array

Exercise

- Write a program that iterates over an array and builds a new array that is the result of incrementing each value in the original array by a value of 2. You should have two arrays at the end of this program, The original array and the new array you've created. Print both arrays to the screen using the p method instead of puts.

Hash

Key - value pairs

```
fruits = { apples:  3 }
ages = { "David" => 28, "Amy"=> 19 }
User = { :name=>"Dave", :age=>28 }
```

Hash

```
my_hash = { apples:  3 , bananas: 5 }

add
  my_hash[:bananas] = 5

retrieve
  puts my_hash[:apples]

Delete
  my_hash.delete(:apples)
```

Hash

- Methods
  - keys | values
  - key(value)
  - has_key? | has_value?
  - fetch()
  - size | length | count
  - sort | sort_by
  - each { |k,v| }

Exercise

- Using some of Ruby's built-in Hash methods, write a program that loops through a hash and prints all of the keys and values inside a string.

Comments

- Single line comment starts with #
- mutltiline comment starts with any of the symbol!!

```
=begin
  anything goes inside is considered as comments
=end
```

Methods

```
def say
  # method body goes here
end
```

def - Reserved word. Meaninig defintion.
say - Method name. Can be anything.
- Single line comment.
end - End of method.

Invocation/calling: say

* Methods should be defined before calling them

Methods – with params

```
def say(param)
  # method body goes here
end
```

param – Parameter.

Invocation/calling: say(‘hi’)

Methods - With defaul params

```
def say(param=’hello’)
  # method body goes here
end
```

Invocation/calling: say(‘hi’) or say

Methods - with optional params

```
def say(*p)
  # puts p
end
```

Methods

Returns evaluated result of last line by default, unless an explicit return comes before it

```
def add_three(number)
  number + 3
end
```

```
def add_three(number)
  return number + 3
end
```

```
def add_three(number)
  return nuber + 3
  number + 4
end
```

Methods - chaining

"hi there".length.to_s.to_i.times { p “hi” }

Methods - as arguments

multiply(add(20, 45), add(80, 10))

Exercise

- Write a program that prints a greeting message. This program should contain a method called greeting that takes a name as its parameter and returns a string.

- Write a program that includes a method called multiply that takes two arguments and returns the product of the two numbers.

- Write a program that prompts the user to enter a name and then outputs a greeting based on the input. If user input is empty then print “greetings” else print “welcome <username>”.

Exception Handling

```
begin
  # perform some dangerous operation
rescue
  # do this if operation fails
  # for example, log the error
end
```

OOPS

Object Oriented Programming

Class

States and Behaviors.

- State : attributes for individual objects
- Behaviors : capabilities of 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. You terminate the class definition with the keyword end

```
class MyClass
#logic
end
```

object

Instance of a class

my_object = MyClass.new

Initializing / Constructing

The purpose of the initialize method is to initialize the class/instance variables for a new object. Which gets called when an object is created.

```
class Dog
  def initialize
    puts "This object initialized!"
  end
end

sparky = Dog.new
```

```
class Dog
  def initialize(name)
    @name = name
  end
end

sparky = Dog.new(‘champ’)
```

Instance Methods

```
class Person
  def initialize(name)
    @name = name
  end

  def speak
    "BlahBlahBlah!"
  end
end

sparky = Person.new("James")
sparky.speak


#"#{@name} says arf!"
```

Exercise

Create a ruby program to write a Cat class which has instance variable name and instance method as sound. And demonstrate how instance method can be invoked on cat object.

Accessors - getter

```
def get_name
    @name
 end
```

Accessors - setters

Ruby provides a special syntax for defining setter methods: the method name is followed by an equal sign (=)

```
def set_name=(name)
    @name = name
end
```

Accessors

In Ruby it is a common practice to name the getter and setter methods using the same name as the instance variable they are accessing

```
def name
  @name
 end
```

```
 def name=(n)
   @name = n
 end
```

attr_accessor

Ruby has a built-in way to automatically create these getter and setter methods for us, using the attr_accessor method which takes a symbol of the instance variable name as an argument

attr_accessor :name

Class Methods

can call directly on the class itself, without having to instantiate any objects

```
class Person
    def self.info
    puts "A Person"
    end
End

Person.info
```

Class Variables

Class variables are accessible to every object of a class.

```
class Person
  @@count = 0
  def initialize
    @@count += 1
  end

  def self.get_count
    @@count
  end
end
```

```
p1 = Person.new
p2 = Person.new

puts Person.get_count
```

Inheritance

Inheritance is when a class receives, or inherits, attributes and behavior from another class.

The class that is inheriting behavior is called the subclass (or derived class)

The class it inherits from is called the superclass (or base class)

Inheritance

The < symbol is used to inherit a class from another class.

```
class Dog < Animal
        #some code
end
```

Inheritance

```
class Animal
  def initialize(name, color)
    @name = name
    @color = color
  end
  def speak
    puts "Hi"
  end
end
```

```
class Dog < Animal
End

d = Dog.new("Bob", "brown")
d.speak
```

Inheritance - overriding

```
class Animal
  def initialize(name, color)
    @name = name
    @color = color
  end
  def speak
    puts "Hi"
  end
end

class Dog < Animal
end
```

```
class Cat < Animal
  attr_accessor :age
  def speak
    puts "Meow"
  end
end
```

```
c = Cat.new("Lucy", "white")
c.age = 2
c.speak
```

super

Ruby has a built-in method called super, which is used to call methods from the superclass

```
class Animal
  def speak
    puts "Hi"
  end
end
```

```
class Cat <
Animal
  def speak
    super
    puts "Meow"
  end
end
```

```
c = Cat.new
c.speak
```

Module

- Like object but not object
- Added to class with include, called mixins
- Module lookup using ancestors

UFFFF!!!!

Dharshan

dharshandj@gmail.com

darshan@fsmk.org

dharshan4help@gmail.com

LinkedIn - Dharshan Bharathur

Github – darshand

Telegram - @dhar5han