



Ruby Programming

Introduction



Introducing Ruby

- Ruby is a ‘3rd generation’ scripting language
 - Shell scripting makes scripting possible
 - Perl makes scripting powerful and expressive
 - Ruby makes scripting more powerful and fun
- Ruby was invented by Yukihiro Matsumoto (aka Matz)
 - He designed Ruby “to make programmers happy”
- Key characteristics of Ruby are:
 - Complete object orientation
 - Dynamically typed (aka ‘Duck Typing’)
 - Support for functional programming
 - Extensible through meta-programming



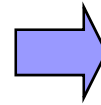
Object Orientation in Ruby

- Everything is an object in Ruby
 - `'var = 7'` declares a reference to a number object
 - `'var = /[a-z]{3}/'` declares a reference to a 'Regexp' object
- Classes are simple to declare and use:
 - Objects are created via `'MyClass.new'`
 - Methods called `'initialize'` are constructors
 - Fields are prefixed with the `'@'` sigil
- Fields do not have to be pre-declared
 - They can be added to the object as it is used
 - All fields are completely hidden in the object
 - This differs from `'private'` in Java and C#



```
ref1 = 6
ref2 = 7.8
ref3 = "abc"
ref4 = /[a-z]{3}/
ref5 = Regexp.new("[a-z]{3}")
ref6 = 2..5
ref7 = []
ref8 = {}

puts ref1.class
puts ref2.class
puts ref3.class
puts ref4.class
puts ref5.class
puts ref6.class
puts ref7.class
puts ref8.class
```



```
Fixnum
Float
String
Regexp
Regexp
Range
Array
Hash
```

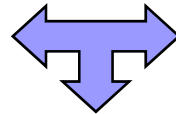
```
class Person
  def initialize(name,age)
    @name = name
    @age = age
  end
  def to_s
    return "#{@name} aged #{@age}"
  end
  def speak
    puts "Hi, I'm #{@name}"
  end
end
```

```
p1 = Person.new("Dave",27)
p2 = Person.new("Jane",28)

puts p1
puts p1.inspect
p1.speak

puts "-----"

puts p2
puts p2.inspect
p2.speak
```



```
Dave aged 27
#<Person:0x294d1c4 @age=27, @name="Dave">
Hi, I'm Dave
-----
Jane aged 28
#<Person:0x294d19c @age=28, @name="Jane">
Hi, I'm Jane
```

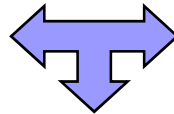


Dynamic Typing

- Ruby is a dynamically typed language
 - As in Perl variables are not prefixed with a type
 - All variables are actually references to objects
 - But as in Java all objects have a well defined type
 - No automatic type conversions are performed
 - 'var1 + var2' will fail if 'var1' is a number and 'var2' as string
- This is sometimes referred to as 'Duck Typing'
 - The call 'obj.func()' will succeed if the object pointed to by 'obj' contains a method called 'func' - regardless of the class type
 - If it walks like a duck and quacks like a duck then it is a duck...
 - This makes it very easy to write loosely-coupled code



```
class A
  def to_i
    return 123
  end
  def to_f
    return 45.6
  end
  def to_s
    return "def"
  end
end
```



```
var1 = 6
var2 = 7.8
var3 = "abc"

obj = A.new
var1 += obj.to_i
var2 += obj.to_f
var3 += obj.to_s

puts var1
puts var2
puts var3

var4 = "101"
puts var1 + var4.to_i
```

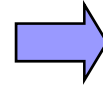
```
129
53.4
abcdef
230
```



```
class Chess
  def play
    puts "Lets play a game..."
  end
end

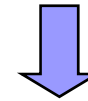
class Guitar
  def play
    puts "Lets play some music..."
  end
end

class Player
  def initialize(item)
    @item = item
  end
  def start
    @item.play
  end
end
```



```
p1 = Player.new(Chess.new)
p2 = Player.new(Guitar.new)

p1.start
p2.start
```



```
Lets play a game...
Lets play some music...
```




Blocks, Closures and Proc Objects

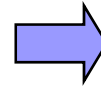
- Ruby supports closures
 - A closure is a code block that can be passed as a parameter
- There are three ways to create closures:
 - By placing a block of code after a method call
 - By creating an instance of the 'Proc' class
 - Via the 'Kernel.lambda' method
- Closures are mostly used as 'internal iterators'
 - Normally you declare a loop that iterates over items
 - With closures you can declare methods that iterate for you
 - Inside a method 'yield' transfers control to the closure
 - A closure can have any number of parameters



```
var1 = 7
var1.times {|num| puts num }

puts "-----"

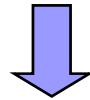
var2 = 5
var2.times do |num|
  puts num
end
```




- 0
- 1
- 2
- 3
- 4
- 5
- 6
-
- 0
- 1
- 2
- 3
- 4



```
data = "ab-CD-efg--HIJ--kl--MN--opq"  
  
result1 = data.gsub(/[a-z]{3}/) {|match| match.upcase }  
result2 = data.gsub(/[A-Z]{3}/) {|match| match.downcase }  
  
puts result1  
puts result2
```



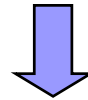
```
ab-CD-EFG--HIJ--kl--MN--OPQ  
ab-CD-efg--hij--kl--MN--opq
```



```
puts "Enter the path to a text file..."
path = gets
path.chomp!

f = File.new(path)

puts "--- File Contents ---"
f.each_line {|line| puts line }
```



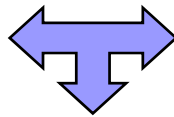
```
Enter the path to a text file...
~/input.txt
--- File Contents ---
line no1
line no2
line no3
line no4
line no5
```



```
class A
  def initialize
    @values = []
  end
  def add(value)
    @values.push(value)
  end
  def doSomething
    for val in @values
      yield val
    end
  end
end
```

```
obj = A.new
obj.add("abc")
obj.add("def")
obj.add("ghi")
obj.add("jkl")

count = 0
obj.doSomething do |item|
  puts "Item #{count} is #{item}"
  count+=1
end
```



```
Item 0 is abc
Item 1 is def
Item 2 is ghi
Item 3 is jkl
```



Support for Collections

- As with Perl collections are basic types
 - There is no need for a separate collections library
- Arrays do not have a fixed size
 - Specifying an out of range index causes new boxes to be added
- Hashes are very easy to work with
 - You can specify what value should be returned for absent keys
- Closures are used heavily in the API
 - Both arrays and hashes have an 'each' method
 - This is an internal iterator, yielding to a closure for each item
 - Hashes also have 'each_key' and 'each_value' methods

```

def printArray(array)
  puts "--- Contents Are: ---"
  array.each{|i| puts "\t#{i}" }
end

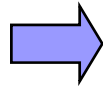
array1 = []
array1.push("abc")
array1 << "def"
array1[2] = "ghi"
printArray(array1)

array2 = ["ab", "cd", "ef"]
array2.insert(0, "zz", "yy", "xx")
array2.fill("ww", 6..8)
printArray(array2)

array1.concat(array2)
printArray(array1)

array1.map!{|item| item.upcase}
printArray(array1)

```



```

--- Contents Are: ---
  abc
  def
  ghi
--- Contents Are: ---
  zz
  yy
  xx
  ab
  cd
  ef
  ww
  ww
  ww
--- Contents Are: ---
  abc
  def
  ghi
  zz
  yy

```

```

■ xx
  ab
  cd
  ef
  ww
  ww
  ww
--- Contents Are: ---
  ABC
  DEF
  GHI
  ZZ
  YY
  XX
  AB
  CD
  EF
  WW
  WW
  WW

```



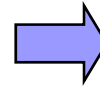
```
def printHash(hash)
  puts "--- Contents Are ---"
  hash.each do |key,value|
    puts "\t #{key} indexes #{value}"
  end
end

hash1 = {:k1=>"v1", :k2=>"v2"}
printHash(hash1)

hash1[:k3] = "v3"
hash1.store(:k4,"v4")
printHash(hash1)

hash2 = Hash.new do |h,k|
  h[k] = k.upcase + "_Value"
end

hash2["a"]
hash2["b"]
hash2["c"]
printHash(hash2)
```



```
--- Contents Are ---
    k2 indexes v2
    k1 indexes v1
--- Contents Are ---
    k2 indexes v2
    k3 indexes v3
    k4 indexes v4
    k1 indexes v1
--- Contents Are ---
    a indexes A_Value
    b indexes B_Value
    c indexes C_Value
```




Ruby's Meta-Programming Support

- Ruby is designed to let you see 'under the hood'
 - You can create abstractions and integrate them seamlessly
- In Ruby classes are executable code
 - The definition of each class is interpreted at runtime
 - A Ruby class is itself an object (an instance of 'Class')
 - Both can be extended as your program executes
- A simple example is attribute generation
 - The 'attr_accessor' method adds getter and setter methods to a class for each field name passed as a parameter
 - The 'attr_reader' method adds getter methods only

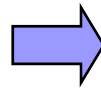
```
class Person
  attr_reader :name
  attr_accessor :age

  def initialize(name,age)
    @name = name
    @age = age
  end
end

p1 = Person.new("Dave",27)
p1.age = 30

puts p1.age
puts p1.name

puts "The methods of Person are:"
for m in p1.methods
  puts "\t #{m}"
end
```



```
30
Dave
The methods of Person are:
  age
  age=
  name
```

Other methods
omitted



Ruby's Meta-Programming Support

- The methods available from a class can be changed
 - New methods can be added with 'define_method'
 - Inherited methods can be removed with 'undef_method'
 - Overridden versions of inherited methods can be removed with 'remove_method', making the base version callable
- Extra methods can be added to individual objects
 - These are known as 'singleton methods'
- Ruby defines 'hooks' for meta-programming
 - E.g. the 'method_missing' method is triggered when a call is made to an unknown method of the current object
 - The default implementation throws an exception