Ruby Language Basics II

Sang Shin
JPassion.com
"Code with Passion!"

Topics

- Methods Basics
- Methods Advanced
  - Arguments, Visibility, Method with a ! (bang)
- Modules
- Control structures
- Exception handling
- Ruby operators
- Regular expression

Method Basics

Method Definitions

- Methods are defined using the keyword *def* followed by the method name and a set of arguments

```
def my_method1 (argument1, argument2)
  puts (argument1, argument2)
end

 # Parentheses (..) are optional both in method definition
 # as well as in method invocation
def my_method2 argument1, argument2    # method definition
  puts argument1, argument2            # method invocation
end
```

Class Method vs. Instance Method

- A class can contain both class and instance methods
- Class method is defined with *self.*<*method_name*>

```
class MyClass
  def self.find_everybody   # class method
      User.find(:all)
  end
  def my_instance_method  # instance method
  end
end
```

- Class method is invoked with a class

```
MyClass.find_everybody
```

How to Invoke Methods

- Methods are called using the following syntax:

  method_name (argument1, argument2,…)

- The parentheses can be omitted

  method_name argument1, argument2 # with arguments
  method_name # with no arguments

- If you use method result immediately for calling another method, however, then you have to use parentheses:

  # A method returns an array and we want to reverse
  # element order of the returned array using "reverse"
  # method of Array class.
  results = method_name (argument1, argument2).reverse

Return Value of a Method

- A method returns the value of the last expression evaluated – return statement is optional

```
# return value of x+y expression
def add_method (x,y)
  puts "something"
  x + y     # return statement is optional
end
```

Explicit “return” statement

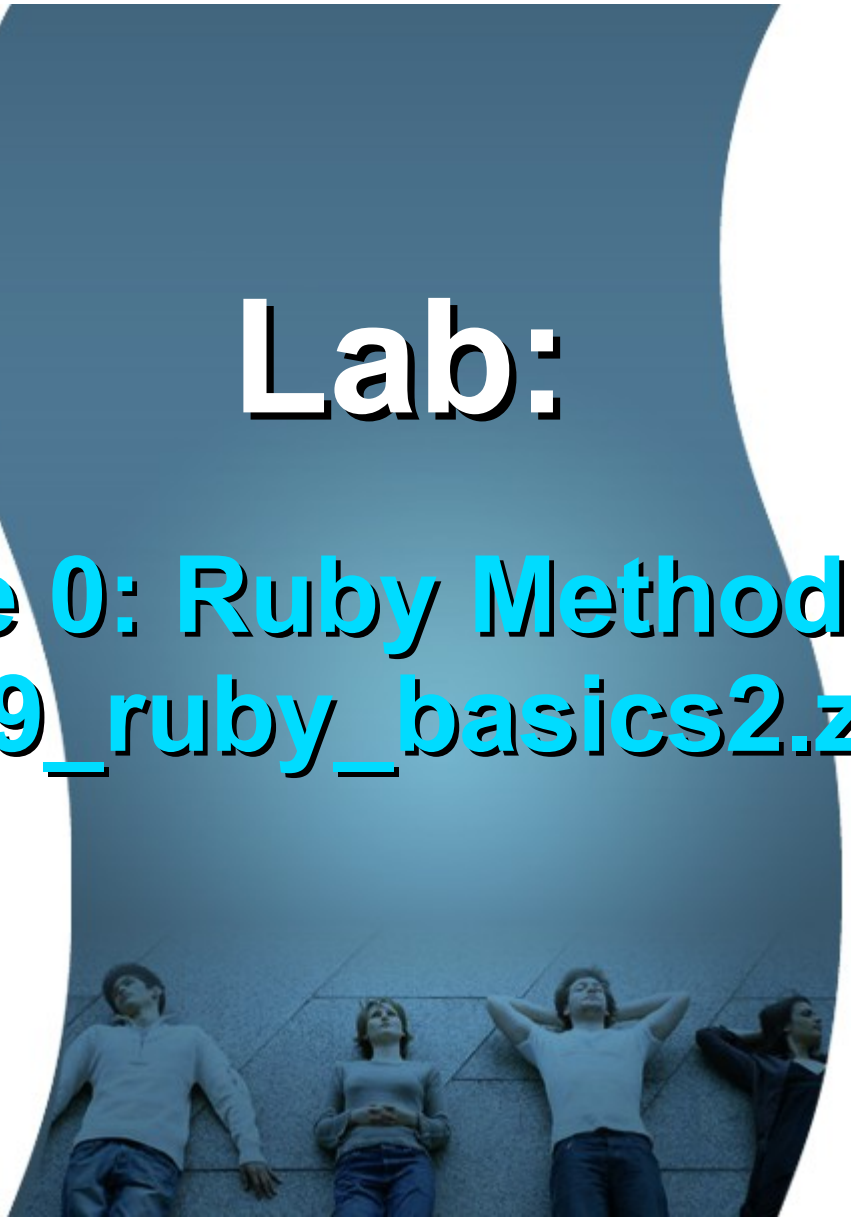
- An explicit “return” statement can also be used to return from function with a value, prior to the end of the function declaration

```
def add_method(x,y)
  return x + y
  puts “this is not evaluated”
end
```

- This is useful when you want to terminate a loop or return from a function as the result of a conditional expression

Lab:

Exercise 0: Ruby Method Basics 5509_ruby_basics2.zip



Methods:
Arguments

Default Value Argument

- A default argument value can be specified in method definition

```
def some_method(value='default', arr=[ ])
  puts value
  puts arr.length
end
some_method('something')       # arr is not passed
```

- The method call above will output:

```
something
0                    (default argument value [].length)
```

Variable Length Argument List

- The last argument of a method may be preceded by an asterisk(*), which is sometimes called the 'splat' operator - this indicates that more arguments may be passed to the function. Those arguments are collected up and an array is created.

```
def calculate_value(x,y,*otherValues)
  puts otherValues # otherValues is an array
end

calculate_value(1,2,'a','b','c')  # ['a', 'b', 'c']
calculate_value(1,2,'a','b','c', 'd')  # ['a', 'b', 'c', 'd']
```

Array Argument as “*array”

- The asterisk (*) operator may also precede an Array argument in a method call. In this case the Array will be expanded and the values passed in as if they were separated by commas.

```
arr = ['a','b','c']
calculate_value(*arr)
```

- has the same result as:

```
calculate_value('a','b','c')
```

Passing a Hash as an Argument

- Another technique that Ruby allows is to pass a Hash argument when invoking a function, and that gives you best of all worlds - named arguments, and variable argument length
- Very common in Ruby/Rails programming

```
def accepts_hash( var )
    print "got: ", var.inspect     # will print out what it received
end

# Pass a hash as an argument
accepts_hash( {:arg1 => 'giving arg1', :argN => 'giving argN'} )
# => got: {:argN=>"giving argN", :arg1=>"giving arg1"}
```

Parentheses ( ) for the Arguments, Braces { } for a Hash Argument

- Parentheses can be omitted for the arguments
- If the last argument is a Hash, braces { } of the Hash can be omitted as well. The following three work the same.

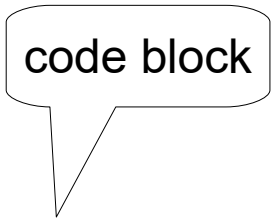
```
# Arguments are enclosed with ( ), hash is enclosed with braces { }
accepts_hash( { :arg1 => 'giving arg1', :argN => 'giving argN' } )

# Argument are enclosed with (), but no { } for a hash argument
accepts_hash( :arg1 => 'giving arg1', :argN => 'giving argN' )

# No () for arguments, no { } for a hash - very common
accepts_hash :arg1 => 'giving arg1', :argN => 'giving argN'
accepts_hash arg1: 'giving arg1', argN: 'giving argN'   (from Ruby 1.9)
```

Calling a Method with a Code Block

- Note: We have not learned Code block yet.. so if you don't understand things on this page, that is fine...
- If you are going to pass a code block to function, however, you need parentheses for arguments – we will learn about code block later on




```
# You need parentheses for arguments since there is a block
accepts_hash( :arg1 => 'giving arg1', :argN => 'giving argN' )  { |s| puts s }
accepts_hash( { :arg1 => 'giving arg1', :argN => 'giving argN' } )  { |s| puts s }

# Compile error since there is no ( ) with code block
accepts_hash :arg1 => 'giving arg1', :argN => 'giving argN'   { |s| puts s }
```

Methods:
Method with a ! (Bang)

Method with ! (Bang)

- In Ruby, methods that end with an exclamation mark (also called a "bang") modify the object
- Methods that do not end in an exclamation point return data, but do not modify the object.

```
>> x="jpassion"
=> "jpassion"

>> x.upcase
=> "JPASSION"
>> x
=> "jpassion"

>> x.upcase!
=> "JPASSION"
>> x
=> "JPASSION"
```

Methods:
Visibility

Declaring Visibility

- By default, all methods in Ruby classes are public - accessible by anyone
- If desired, this access can be restricted by *private, protected* object methods
  - It is interesting that these are not actually keywords, but actual methods that operate on the class, dynamically altering the visibility of the methods

private

- The *private* methods methods can be called only from within the calling object
  - > You cannot access another instance's private methods directly.
  - > If *private* is invoked without arguments, it sets access to private for all subsequent methods.
- The *protected* methods can be called by any instance of the defining class or its subclasses.

```
class Example
  def methodA
  end
  private # all methods that follow will be made private:
          # not accessible by outside object
  def methodP
  end
end
```

Declaring Visibility: private

- *private* method can be invoked with named arguments - altering the visibility of *methodP* to private in the example below

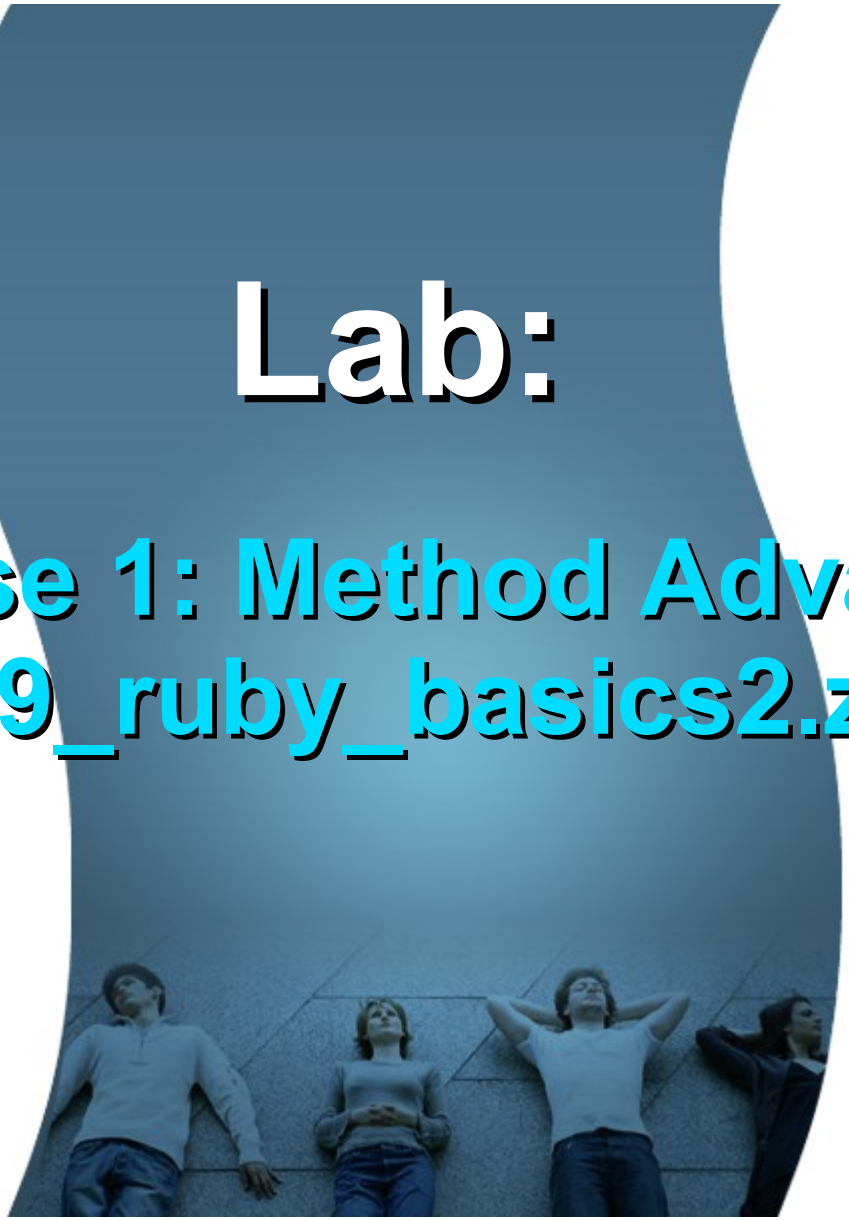
```
class Example
  def methodA
  end

  def methodP
  end

  private :methodP   # change the visibility of methodP to private
end
```

Lab:

Exercise 1: Method Advanced
5509_ruby_basics2.zip



Modules

What is a Module?

- Modules are way of grouping together some functions and variables and classes, thus providing namespaces
  - Similar to Java packages, which provides namespaces for Java classes
- A class “C” in a Module “M” is referenced as *M::C*
- Methods can be present in a Module
- A Module cannot be instantiated – object cannot be created from a module

Module Provides Namespace

```
puts "----Define People module with Stalk class"
module People
  class Stalk
    def about
      "I am a person."
    end
  end
end

puts "----Define Plants module with Stalk class"
module Plants
  class Stalk
    def about
      "I am a plant."
    end
  end
end

puts "----Create an instance of Stalk class of People Module"
a = People::Stalk.new

puts "----Create an instance of Stalk class of Plants Module"
b = Plants::Stalk.new
```

Mix-in with a module

- A module can contain just methods (instead of classes)
- You can “include” a module into a class - it is called Mix-in

```
module Aeronautics
  def launch()          # A module can have a method
    "3, 2, 1 Blastoff!"
  end
end

class RocketShip
  include Aeronautics  # Include a module
end

r = RocketShip.new
puts r.launch           # You can invoke a method
                        # of an included module
```

Mix-in with multiple modules

- You can mix in as many modules as you like

```
module Aeronautics
  def launch()
    "3, 2, 1 Blastoff!"
  end
end

module Calculator
  def add(x, y)
    x + y
  end
end

class RocketShip
  include Aeronautics
  include Calculator
end

r = RocketShip.new
puts r.launch      # 3, 2, 1 Blastoff!
puts r.add(3, 4)  # 7
```

Requiring a Module

- If your module is in another file, you must first *require* that module before you can use it in *include* statement

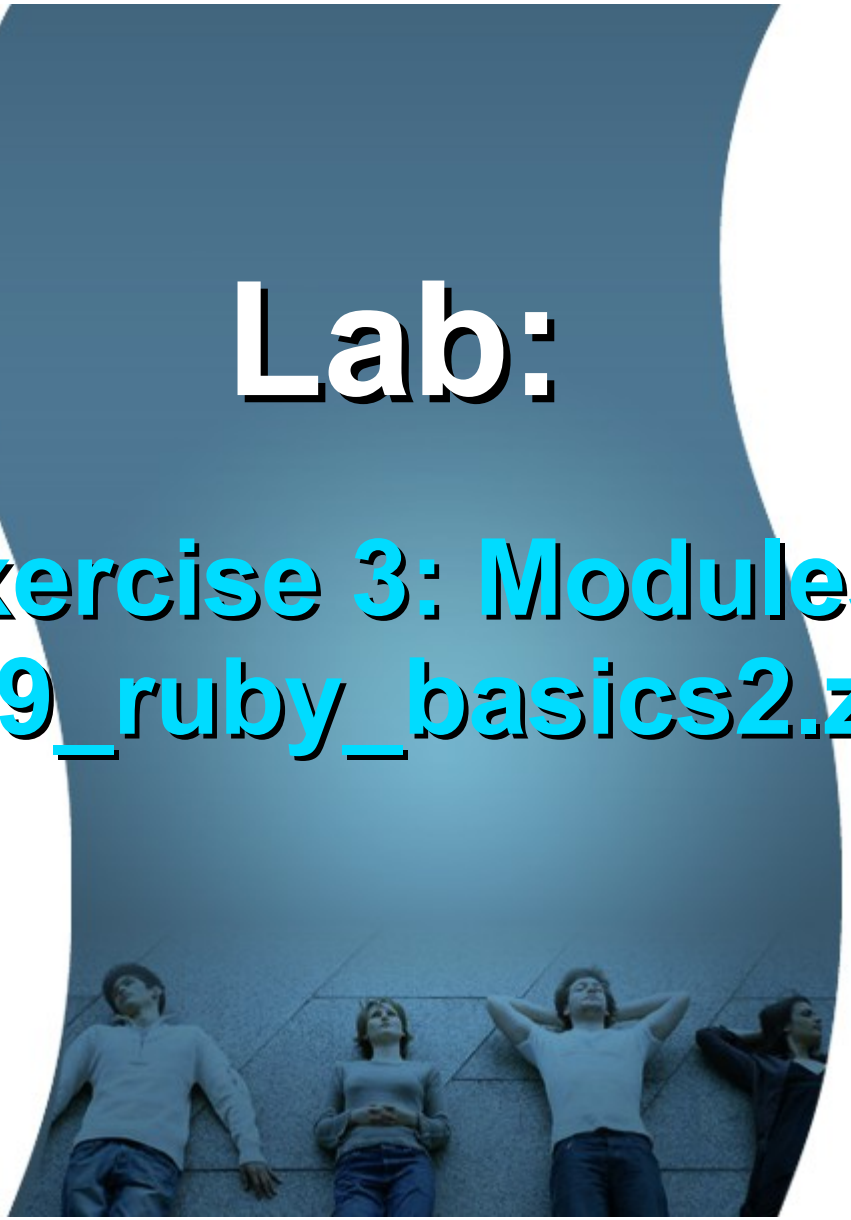
```
require './RubyModule_define'

puts "----Create MyNumber class which includes Stringify module"
class MyNumber
  include Stringify # "Stringify" module is defined in "RubyModule_define.rb"
  def initialize(value)
    @value = value
  end
end

puts "----Create MyNumber object and call stringify method from the Stringify module"
my_number = MyNumber.new(2)
puts my_number.stringify  # Should print Two
```

Lab:

Exercise 3: Modules
5509_ruby_basics2.zip



Control Structure

Control Structure: Assignment

```
puts "----Every assignment returns the assigned value"
puts a = 4       #=> 4

puts "----Assignments can be chained"
puts a = b = 4   #=> 4
puts a+b         #=> 8

puts "----Shortcuts"
puts a += 2      #=> 6
puts a = a + 2   #=> 8

puts "----Parallel assignment"
a, b = b, a
puts a           #=> 4
puts b           #=> 8

puts "----Array splitting"
array = [1,2]
a, b = *array
puts a           #=> 1
puts b           #=> 2
```

Control Structure: Conditionals

```
puts "----if/else condition"
if (1 + 1 == 2)
  puts "One plus one is two"
else
  puts "Not a chance!"
end

puts "----if and unless conditions"
puts "Life is good!" if (1 + 1 == 2)
puts "Surprising" unless (1 + 1 == 2)

puts "----? condition"
puts (1 + 1 == 2)?'True':'Not True'
```

Control Structure: Conditionals

```
puts "----case/when/then condition"
spam_probability = rand(100)
puts "spam_probability = " + spam_probability.to_s

case spam_probability
when 0...10 then puts "Lowest probability"
when 10...50 then puts "Low probability"
when 50...90 then puts "High Probability"
when 90...100 then puts "Highest probability"
end
```

Control Structure: Loop

```
puts "---- while loop"
while (i < 10)
  i *= 2
end
puts i          #=> 16

puts "---- while loop 2"
i *= 2 while (i < 100)
puts i          #=> 128

puts "---- while loop with begin/end"
begin
  i *= 2
end while (i < 100)
puts i          #=> 256
```

Control Structure: Loop

```
puts "---- until"
i *= 2 until ( i >= 1000)
puts i        #=> 1024

puts "---- loop"
loop do
  break i if (i >= 4000)
  i *= 2
end
puts i        #=> 4096

puts "---- times"
4.times do
   i *= 2
end
puts i        #=> 65536
```

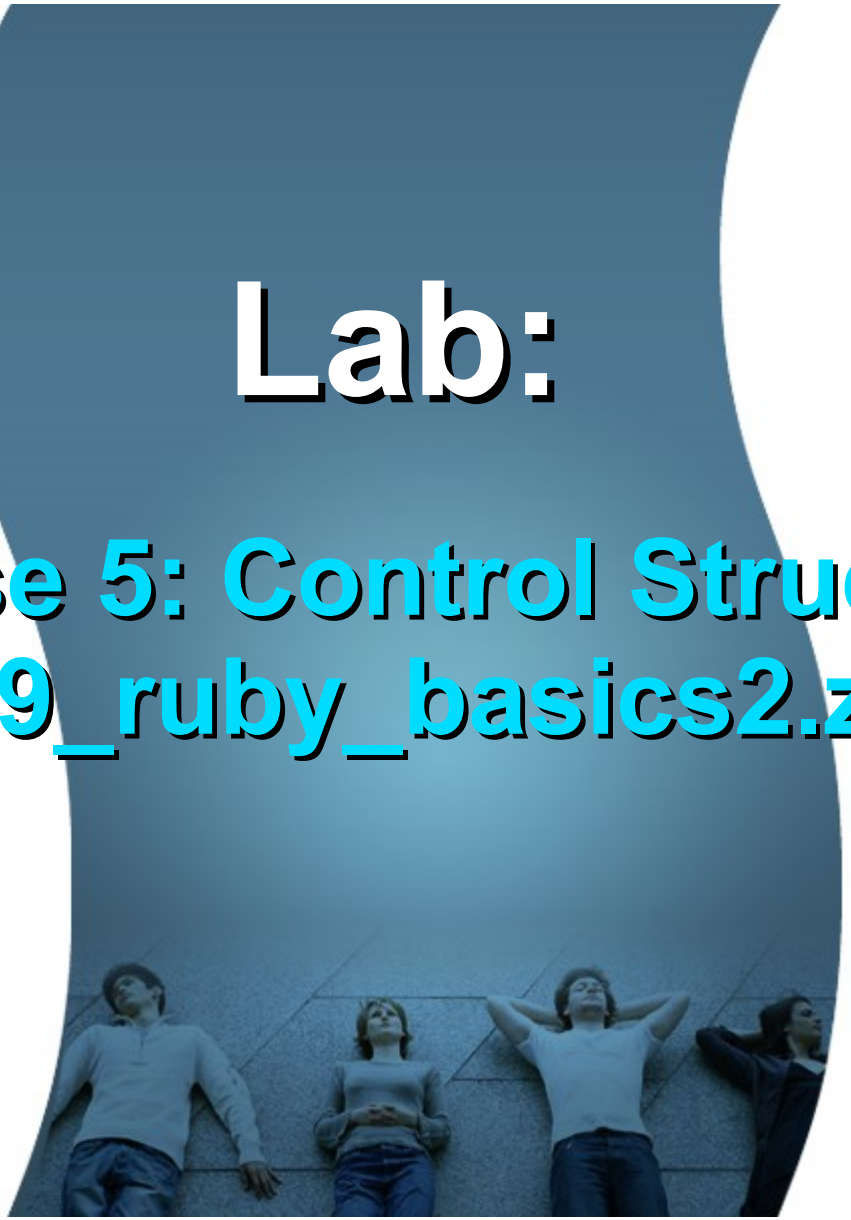
Control Structure: Loop

```
puts "---- array"
r =[]
for i in 0..7
  next if i % 2 == 0
  r << i
end
puts r

puts "----Many things are easier with blocks"
puts (0..7).select { |i| i % 2 != 0}
```

Lab:

Exercise 5: Control Structures
5509_ruby_basics2.zip



Exception Handling

Exception Class

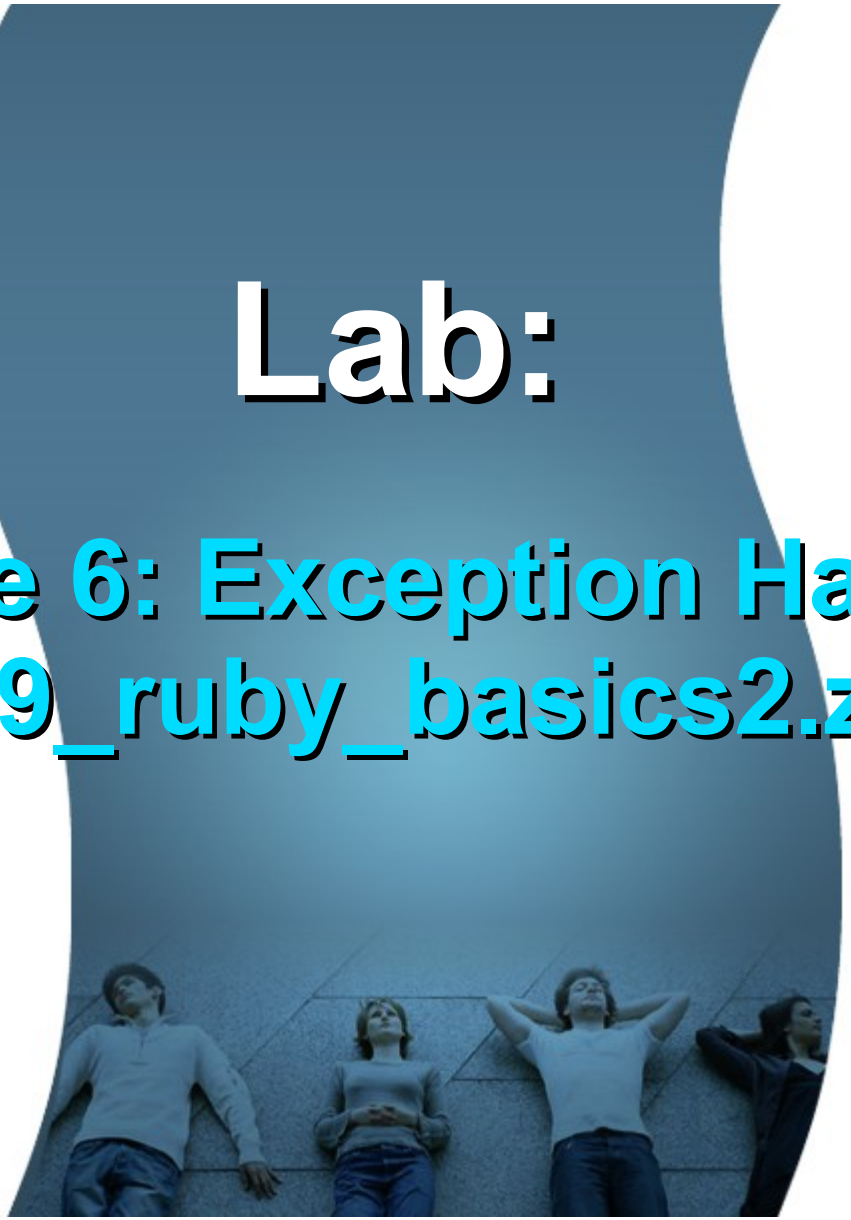
- Exceptions are implemented as classes (objects), all of whom are descendents of the *Exception* class
- List of Exceptions
  - *ArgumentError, IndexError, Interrupt*
  - *LoadError, NameError, NoMemoryError*
  - *NoMethodError, NotImplementedError*
  - *RangeError, RuntimeError*
  - *ScriptError, SecurityError, SignalException*
  - *StandardError, SyntaxError*
  - *SystemCallError, SystemExit, TypeError*

Exception Handling

```
begin
    # attempt code here
rescue SyntaxError => mySyntaxError  # Similar to 'catch' in Java
    print "Unknown syntax error. ", mySyntaxError, "\n"
    # error handling specific to problem here
rescue StandardError => myStandardError
    print "Unknown general error. ", myStandardError, "\n"
    # error handling specific to problem here
else
    # code that runs ONLY if no error goes here
ensure                                          # Simiar to 'finally' in Java
    # code that cleans up after a problem and its error handling goes here
end
```

Lab:

Exercise 6: Exception Handling
5509_ruby_basics2.zip



Ruby Operators

Ruby Operators

- Most Ruby operators are actually method calls
  - For example, a + b is interpreted as a.+(b), where the + method in the object referred to by variable a is called with b as its argument.
- Ruby arithmetic operators
  - +, -, *, /< %, **
- Ruby comparison operators
  - ==, !=, >, <, >=, <=, <=>, ===
- Ruby assignment operators
  - =, +=, -=, *=, /=, %=, **=

Ruby Operators

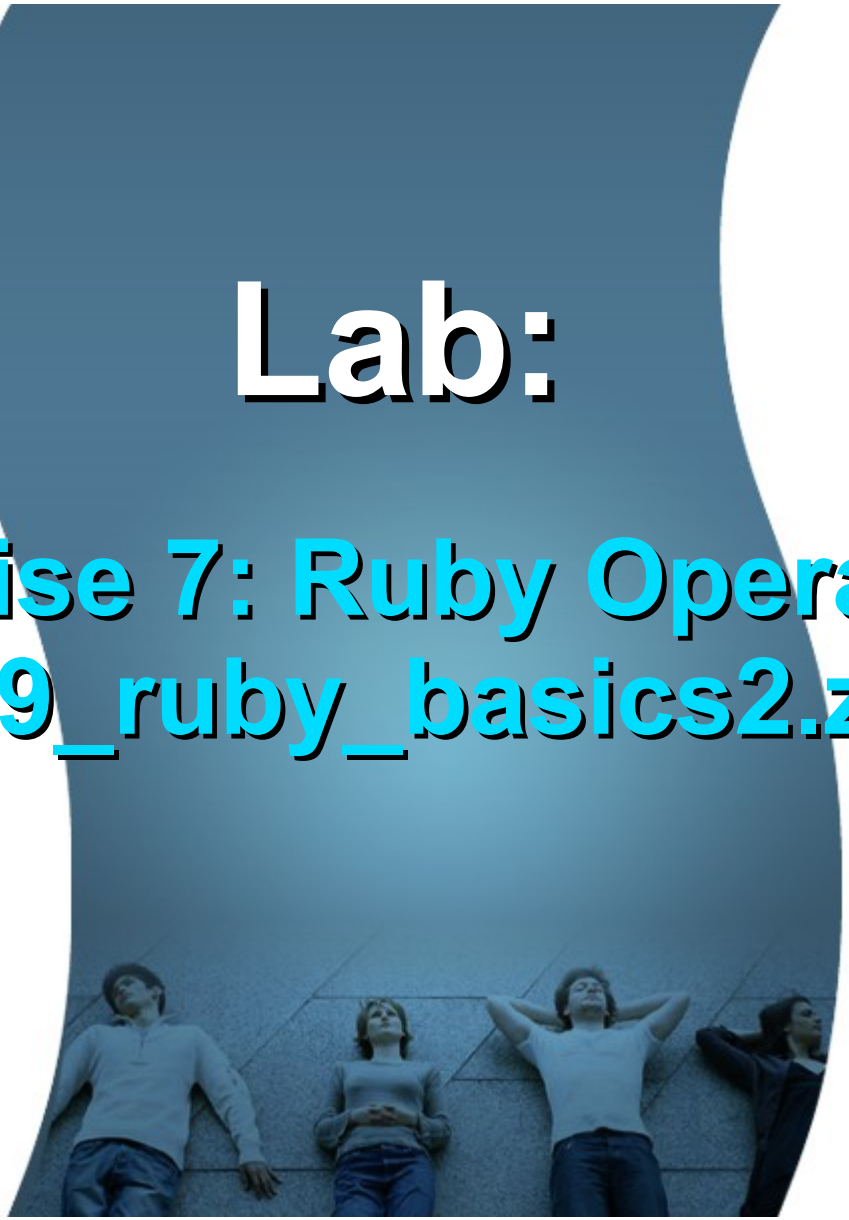
- Bitwise operators
  - &, |, ^, ~, <<, >>
- Logical operators
  - And, or, &&, ||, !, not
- Ternary operators
  - ?:
- Range operators
  - .. (inclusive), … (

Ruby Operators

- *a* || *b*
  - This expression evaluates *a* first. If it is not false or nil, then evaluation stops and the expression returns a. Otherwise, it returns b.
  - Common practice of returning a default value b if the first value has not been set
- *a* ||= *b*
  - Same as *a* = *a* || *b*

Lab:

Exercise 7: Ruby Operators
5509_ruby_basics2.zip



Regular Expression

Regular Expression

- Lets you specify a pattern for match
- Use */pattern/* or *%r{pattern}*
- Simple pattern examples

```
/ruby|rails/          # match either ruby or rails
/r(uby|ails)/         # same as above
/ab+c/                # match a string containing an a followed one
                      # or more b followed by c
/ab*c/                # same as above except zero or more b
```

“=~” matching operator

- "=~" is a matching operator with respect to regular expressions; it returns the position in a string where a match was found, or nil if the pattern did not match.

```
if subject =~ /r(uby|ails)/
    puts “subject matches the pattern”
end
```

Basic Patterns

- . (dot) - matches any single character
  - **a.c** matches "abc"
  - **.at** matches any three-character string ending with "at", including "hat", "cat", and "bat"
- [ ] - Matches a single character that is contained within the brackets
  - **[abc]** matches "a", "b", or "c"
  - **[a-z]** specifies a range which matches any lowercase letter from "a" to "z".
  - **[abcx-z]** matches "a", "b", "c", "x", "y", and "z", as does [a-cx-z].
  - **[hc]at** matches "hat" and "cat"

Basic Patterns

- [^] - Matches a single character that is not contained within the brackets
  - [^abc] matches any character other than "a", "b", or "c".
  - [^a-z] matches any single character that is not a lowercase letter from "a" to "z".
  - [^b]at matches all strings matched by *.at* except "bat".
- ^ - Matches the starting position within the string.
  - ^[hc]at matches "hat" and "cat", but only at the beginning of the string or line.
- $ - Matches the ending position of the string or the position just before a string-ending newline
  - [hc]at$ matches "hat" and "cat", but only at the end of the string or line.

Character Abbreviation

```
/fo\w+.*bar/          # “foobar”, “fogTS!bar, ...
%r[fo\w+.*bar]        # Same as above
```

Abbreviation	As [...]	Matches	Opposite
\d	[0-9]	Digit character	\D
\s		Whitespace character	\S
\w	[A-Za-z0-9_]	Word character	\W
.		Any character	

Sequence	Matches
*	**zero or more** occurrences of preceding character
+	**one or more** occurrences of preceding character
?	**zero or one** occurences of preceding character

Code with Passion!
JPassion.com