Ruby Language Basics

Sang Shin
Michèle Garoche
http://www.javapassion.com
"Learn with Passion!"



Acknowledgment & hands-on lab

- Some slides of this presentation is created from the following online resources
 - http://en.wikibooks.org/wiki/Ruby_Programming
- The corresponding hands-on lab is available from
 - http://www.javapassion.com/rubyonrails/#Ruby_Lang uage Basics

Topics (Page 1)

- What is Ruby?
- Ruby naming convention
- Interactive Ruby (IRB)
- Ruby object
- Ruby types
 - String, Hash, Symbol
- Ruby class
- Inheritance

Topics (Page 2)

- Methods
 - > Arguments, Visibility, Method with a! (bang)
- Variables
- Modules
- Reserved keywords
- Regular expression
- Control structures
- Exception handling

What is Ruby?

Ruby is...

A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write.

Ruby as a Language

- Interpreted
- Dynamically typed
- Optimized for people
 - > Easy to read and write
 - > Powerful
 - > Fun
- Everything is an object
 - > There are no primitives Java language has primitives which are not objects

Ruby Language History

- Created 1993 by Yukihiro "Matz" Matsumoto
 - "More powerful than Perl and more OO than Python"
- Ruby 1.9.x is current (as of July 2010)

Ruby Naming Conventions

Ruby Naming Conventions

- Ruby file .rb suffix
 - > myprog.rb
- Class & Module names MixedCase
 - > MyClass
- methods lower case with underscores
 - > my_own_method
- local variables lower case with underscores (same as methods)
 - > my_own_variable
- Instance variables @ prefix to variable name
 - > @my_instance_variable

Ruby Naming Conventions

- Class variables @@ prefix to variable name
 - > @@my_class_variable
- Global variables \$ prefix to variable name
 - > \$my global variable
- Constants
 - > UPPER CASE

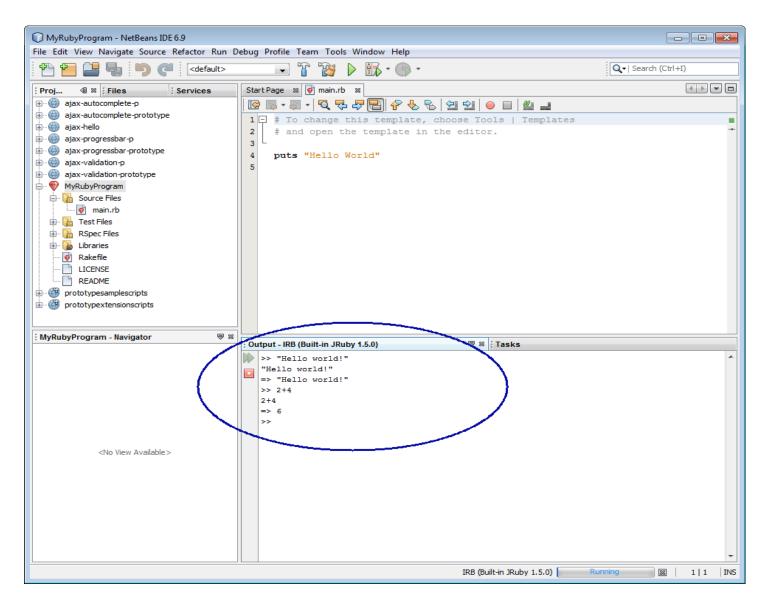
IRB (Interactive Ruby)

IRB (Interactive Ruby)

- When learning Ruby, you will often want to experiment with new features by writing short snippets of code. Instead of writing a lot of small text files, you can use *irb*, which is Ruby's interactive mode.
- You can use irb at the command line

```
$ irb --simple-prompt
>> 2+2
=> 4
>> 5*5*5
=> 125
>> exit
```

IRB in NetBeans IDE



JRuby IRB at the Command line

 It is *jirb* under <NetBeans-Install-Dir>/ruby/jruby-1.5.0/bin directory

```
C:\Program Files (x86)\NetBeans 6.9\ruby\jruby-1.5.0\bin>jirb
irb(main):001:0> puts "life is good"
life is good
=> nil
irb(main):002:0> puts 3+4
7
=> nil
irb(main):003:0> exit
```

JRuby IRB Console (Swing app)

 It is jirb-swing under <NetBeans-Install-Dir>/ruby/jruby-1.5.0/bin directory

C:\Program Files (x86)\NetBeans 6.9\ruby\jruby-1.5.0\bin>jirb_swing

```
JRuby IRB Console (tab will autocomplete)
                                                                                      _ 🗆 ×
Welcome to the JRuby IRB Console [1.1]
irb(main):001:0> puts "Life is good"
Life is good
=> ni1
irb(main):002:0> puts 1000*3
3000
=> nil
irb(main):003:0>
```

Ruby Object

In Ruby, Everything is an Object

- Like Smalltalk, Ruby is a pure object-oriented language — everything is an object.
- In contrast, languages such as C++ and Java are hybrid languages that divide the world between objects and primitive types.
 - The hybrid approach results in better performance for some applications, but the pure object-oriented approach is more consistent and simpler to use.

What is an Object?

- Using Smalltalk terminology, an object can do exactly three things.
 - > Hold state, including references to other objects.
 - > Receive a message, from both itself and other objects.
 - > In the course of processing a message, send messages, both to itself and to other objects.
- If you don't come from Smalltalk background, it might make more sense to rephrase these rules as follows:
 - > An object can contain data, including references to other objects.
 - An object can contain methods, which are functions that have special access to the object's data.
 - > An object's methods can call/run other methods/functions.

In Ruby, Everything Is An Object

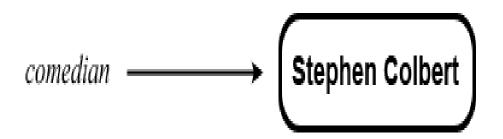
- 'Primitives' are objects
 - > -1.abs
- nil is an object
 - > nil.methods
- Classes are objects
 - Song.new invoking the new class method on 'Song' class
 - Create instances of themselves
- Code blocks are objects
 - They can be pass around, even as parameters or return value
 - > Also known as closure

Variables and Objects

Create a String object containing the text
"Stephen Colbert". We also told Ruby to use the
variable comedian to refer to this object.
(Works the same as in Java)

```
>> comedian = "Stephen Colbert"
```

=> "Stephen Colbert"



Ruby Types

Ruby Types

- String
- Number
- Symbol
 - New concept if you are coming from Java background
- Array
- Hash

Ruby Types: Strings

String Literals

 One way to create a String is to use single or double quotes inside a Ruby program to create what is called a string literal

```
puts 'Hello world'
puts "Hello world"
```

 Double quotes allow you to embed variables or Ruby code inside of a string literal – this is commonly referred to as interpolation.

```
def my_method(name)
    puts "Your name is # {name}"
end
```

String Literals with Interpolation

- Notation
 - > #{expression}
- Expression can be an arbitrary Ruby expression
- If variable that is referenced by #{expression} is not available (has not been assigned), a NameError exception will be raised:

"trying to print # {undefined} variable"
NameError: undefined local variable or method
`undefined' for main:Object

Escape Sequences

- \" double quote
- \\ single backslash
- \a bell/alert
- \b backspace
- \r carriage return
- \n newline
- \s space
- \t tab

Escape Sequences

```
puts "Hello\t\tworld"

puts "Hello\b\b\b\b\b\bGoodbye world"

puts "Hello\rStart over world"

puts "1. Hello\n2. World"
```

puts and print

 puts automatically prints out a newline after the text

```
>> puts "Say", "hello"
Say
hello
```

print function only prints out a newline if you specify one

```
>> print "Say", "hello", "\n" Sayhello
```

% Notation

- %w causes breaks in white space to result in an string array
 - > %w(a b c)
 - > => ["a", "b", "c"]

Ruby Types: Symbols

What is Symbol?

- A Ruby symbol is the internal representation of a name
- It is a class in Ruby language

```
:my_value.class #=> Symbol
```

 You construct the symbol for a name by preceding the name with a colon.

```
:my_symbol
```

- Atomic, immutable and unique
 - > Can't be parsed or modified
 - > All references to a symbol refer to the same object

```
:my_value.equal?(:my_value) #=> true
"my_value".equal?("my_value") #=> false
```

Symbols vs. Strings

- Symbols are always interchangeable with strings
 - In any place you use a string in your Ruby code, you can use a symbol
- Important reasons to use a symbol over a string
 - If you are repeating same string many times in your Ruby code, let's say 10000 times, it will take 10000 times of memory space of the string while if you are using a symbol, it will take a space for a single symbol
- Minor reasons to use a symbol over a string
 - Symbol is easier to type than string (no quotes to type)
 - > Symbol stands out in the editor
 - > The different syntax can distinguish keys from values in hash
 - >:name => 'Sang Shin'

Ruby Types: Hash

Hash

- Hashes are basically the same as arrays, except that a hash not only contains values, but also keys pointing to those values (like in Java)
- Each key can occur only once in a hash.
- A hash object is created by writing Hash.new or by writing an optional list of comma-separated key => value pairs inside curly braces (most common form of creating a hash)

Hash and Symbol

 Usually Symbols (instead of Strings) are used for Hash keys (because it allows for quicker access), so you will see hashes declared like following:

```
hash\_sym = \{ :a => 1, :b => 2, :c => 3 \}
```

Instead of

$$hash_sym = \{ "a" => 1, "b" => 2, "c" => 3 \}$$

Where Do Symbols Typically Used?

- Symbols are often used as
 - > Hash keys (:name => 'Brian', :hobby => 'golf')
 - > Arguments of a method (:name, :title)
 - > Method names (:post comment)
- Symbols are used in Rails pervasively

Ruby Class

Ruby Classes

 Every object in Ruby has a class. To find the class of an object, simply call that object's class method.

```
"This is a string".class #=> String
9.class #=> Fixnum
["this","is","an","array"].class #=> Array
{:this => "is", :a => "hash"}.class #=> Hash
:symbol.class #=> Symbol
```

Defining a Class

Use class keyword

```
# Define Chocolate class
class Chocolate
    def eat
        puts "That tasted great!"
    end
end
```

Instantiation of an Object

- An object instance is created from a class through the a process called instantiation (like in Java)
- In Ruby, this takes place through a Class method new (similar to Java)

```
an_object = MyClass.new(parameters)
```

 This function sets up the object in memory and then delegates control to the *initialize* function of the class if it is present. Parameters passed to the new function are passed into the *initialize* function.

```
class MyClass
def initialize(parameters)
end
end
```

 Simple RocketShip Class – We will study this code in the following slides

```
class RocketShip < Object
 attr accessor :destination
 def initialize(destination)
  @destination = destination
 end
 def launch()
  "3, 2, 1 Blast off!"
 end
end
```

Single Inheritance

```
class RocketShip < Object # < Object is optional
                             # like in Java
 attr accessor:destination
 def initialize(destination)
  @destination = destination
 end
 def launch()
  "3, 2, 1 Blast off!"
 end
end
```

Constructors in Ruby are named initialize

```
class RocketShip < Object
 attr accessor :destination
 def initialize(destination)
  @destination = destination
 end
 def launch()
  "3, 2, 1 Blast off!"
 end
end
# new() allocates a RocketShip instance and initialize()
# is called for initializing that instance
r = RocketShip.new('Netptune')
```

Attributes are easily defined with attr_accessor

```
class RocketShip < Object
 # No need to define getter and setter for an attribute
 attr accessor:destination
 def initialize(destination)
  @destination = destination
 end
 def launch()
  "3, 2, 1 Blast off!"
 end
end
r = RocketShip.new
r.destination = 'Saturn' # Set the attribute with value
```

Ruby Class: Inheritance

Inheritance

- A class can inherit functionality and variables from a super class, sometimes referred to as a parent class or base class. (Like in Java)
- Ruby does not support multiple inheritance and so a class in Ruby can have only one super class. (Like in Java)
- All non-private variables and functions are inherited by the child class from the super class. (Like in Java)

Overriding a method

instance = ChildClass.new

instance.a method

 If your class overrides a method from parent class (super class), you still can access the parent's method by using 'super' keyword class ParentClass # This is a parent classs def a method puts 'b' end end class ChildClass < ParentClass # This is a child class def a method # Call a method of a parent class super puts 'a' end end

Variables

Types of Variables

- Local variables
- Instance variables
- Class variables
- Global variables
- Pre-defined variables

Local Variables

 A local variable is only accessible in the context where you define it (usually in a method, be it class or instance method). For example:

```
i0 = 1
loop {
  i1 = 2
  print defined?(i0), "\n"  # "i0" was initialized in the ascendant block
  print defined?(i1), "\n"  # "i1" was initialized in this block
  break
}
print defined?(i0), "\n"  # "i0 was initialized in this block
print defined?(i1), "\n"  # "i1" was initialized in the "loop" block
  # but not in this block
51
```

Instance Variables

- A variable whose name begins with '@'
 - > @foobar
- An instance variable belongs to the object itself
- It is visible only inside the instance methods (of all objects)
- Uninitialized instance variables have a value of nil.

Class Variables

- A variable whose name begins with '@@'
 - > @@foobar
- When you set a class variable, you set it for the superclass and all of the subclasses.
- You can access class variables everywhere in the class; it is visible in the class methods and in the instance methods

Class Variables: Example

```
class Polygon
@@sides = 10
def self.sides
@@sides
end
end
class Triangle < Polygon
@@sides = 3
end
```

Class variables are shared in the class hierarchy puts Triangle.sides # => 3
puts Polygon.sides # => 3

54

Global Variables

- A variable whose name begins with '\$' has a global scope; meaning it can be accessed from anywhere within the program during runtime.
 - > \$foobar
- Global variables should be used sparingly.
 - They are dangerous because they can be written to from anywhere.
 - > Overuse of globals can make isolating bugs difficult

Pre-defined Variables

- \$0 Contains the name of the script being executed. May be assignable.
- \$* Command line arguments given for the script
- \$\$ The process number of the Ruby running this script.
- \$? The status of the last executed child process.
- \$: Load path for scripts and binary modules by load or require.
- More...

http://en.wikibooks.org/wiki/Ruby_Programming/Synt ax/Variables and Constants#Pre-defined Variables

Methods

Method Definitions

 Methods are defined using the keyword def followed by the method name

```
def output_something_nicely(value)
  puts value
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

find(:all)

end

def my_instance_method

end

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(parameter1, parameter2,...)
- The parentheses can be omitted, however
 method_name parameter1, parameter2 # with parameters
 method_name # with no parameters
- If you use method result immediately, then you have to use parentheses:
 - # You need to use parentheses if you want to work with the # result immediately.
 # eg. If a method returns an array and we want to reverse # element order:
 - results = method_name(parameter1, parameter2).reverse

Return Values

 Methods return the value of the last expression evaluated. The following code returns the value of x+y expression.

```
def calculate_value(x,y)
  x + y # return statement is optional
end
```

Return Values

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

```
def calculate_value(x,y)
  return x + y
end
```

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

Methods: Arguments

Default Value Argument

 A default parameter value can be specified during method definition to replace the value of a parameter if it is not passed into the method or the parameter's value is nil.

```
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 parameter value [].length)
```

Variable Length Argument List

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

```
def calculate_value(x,y,*otherValues)
  puts otherValues
end
```

```
calculate value(1,2,'a','b','c')
```

• In the example above the output would be ['a', 'b', 'c'].

Array Argument

 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')
```

Hash 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
 parameters, 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 (as we learned before)
- If the last argument is a Hash, braces { } of the Hash can be omitted. 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'
```

Calling a Method with a Code Block

 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: 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.

Visibility: private & protected

- If *private* is invoked without arguments, it sets access to private for all subsequent methods.
- The private methods can be called only from within the same instance
- The protected can be called both in the same instance and by other instances of the same class and its subclasses

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

Declaring Visibility: private

class Example

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

```
def methodA
end

def methodP
end

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

Methods: Method with ! (Bang)

Method with! (Bang)

- In Ruby, methods that end with an exclamation mark (also called a "bang") modify the object.
 For example, the method upcase! below changes the letters of a String to uppercase.
 - >> comedian = "Stephen Colbert"
 - => "Stephen Colbert"
 - >> comedian.upcase!
 - => "STEPHEN COLBERT"
- Methods that do not end in an exclamation point return data, but do not modify the object.

Modules

What is a Module?

- Modules are way of grouping together some functions and variables and classes, thus providing namespaces.
- A class "C" in a Module "M" is referenced as M::C
- Methods can be present in a Module
- A Module cannot be instantiated

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

 You can include the module into a class - it is called Mix-in

```
module Aeronautics
                    # A module can have a method
 def launch()
  "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
```

Requiring a Module

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

Reserved Words (Key words)

Reserved Words

- =begin =end alias and begin BEGIN
- break case class def defined? do
- else elsif END end ensure false
- for if in module next nil
- not or redo rescue retry return
- self super then true undef unless
- until when while yield

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
```

Regular Expression Usage in Ruby

Usage in Ruby - "=~" 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.

87

Character Abbreviation

```
# "foobar", "fogTS!bar, ...
/fo\w+.*bar/
%r[fo\w+.*bar]
                     # Same as above
AbbreviationAs [...]
                       Matches
                                        Opposite
            [0-9]
                       Digit character
    \d
                       W hitespace character's
    \s
            [A-Za-z0-9 W ord character
                                           \W
    \W
                       Any character
```

SequenceM atches

- zero or moorœcurrences of preceding charact
 one or moorœcurrences of preceding charact
- ? zero or onecurences of preceding character

Control Structures

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'
puts "----case/when/then condition"
spam probability = rand(100)
puts "spam probability = " + spam probability.to s
```

Control Structure: Conditionals

```
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 \text{ 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 \text{ 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 \}
```

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 # 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
                                       # Finally in Java
ensure
  # code that cleans up after a problem and its error
 handling goes here
end
```

Ruby Operators

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 $a = a \parallel b$

Thank you!

JavaPassion.com provides "Satisfaction Guaranteed" Instructor-led codecamps.

Please check upcoming codecamps from http://www.javapassion.com/codecamps

