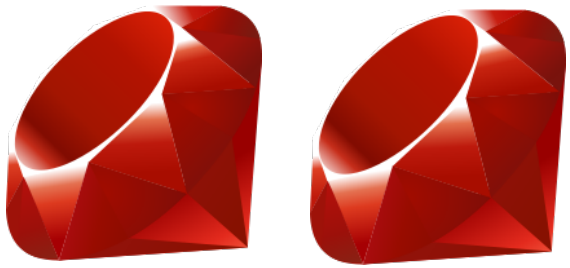

Class Inheritance

Class Inheritance

- Every class implicitly inherits from `Object`
- `Object` inherits from `BasicObject` *(1.9)*
- No multiple inheritance
  - Mixins are used instead (*…Discussed later…*)
- Child class inherits parent's behavior
- Child class can override parent's behavior

Class inheritance example

```
class Dog
  def to_s
    "Dog"
  end
  def bark
    "barks loudly"
  end
end
class SmallDog < Dog
  def bark # Override
    "barks quietly"
  end
end

dog = Dog.new # (btw, new is a class method)
small_dog = SmallDog.new
puts "#{dog}1 #{dog.bark}" # => Dog1 barks loudly
puts "#{small_dog}2 #{small_dog.bark}" # => Dog2 barks quietly
```

< Denotes inheritance

Duck typing

- If it walks like a duck and quacks like a duck – it must be a duck
- It's more important what methods the object can respond to than what type / class it is
- One of the advantages – less code to write
  - For example, no need for method overloading as you would do in Java

No method overloading needed

Method overloading usually takes on 2 forms:

1. Same number of method parameters, but different types of parameters
   - Duck typing takes care of this one
2. Varying number of parameters that can be passed to the method
   - Splat (*) takes care of this one

Method Overloading – case #1

```
class RangePrinter
  def self.one_up_to_another(one, another)
    # As long as 'one' can respond to :upto method
    # we don't really care what type it is
    # In Java or C# - these would probably need to be 2 separate methods
    # (Or a common interface would need to be defined)
      one.upto(another){|x| print x}
  end
end

a1 = 5; a2 = 7
b1 = "a8"; b2 = "b3"
RangePrinter.one_up_to_another a1, a2 # => 567
puts
RangePrinter.one_up_to_another b1, b2 # => a8a9b0b1b2b3
```

Duck typing example

Splat

- `*` prefixes parameter inside method definition
  - Whichever parameters are passed in are stuffed into an array
- Works the other way as well
  - If a method is expecting `n` elements and you have an array with `n` elements – can just pass the array in with splat syntax

Splat example

```ruby
# person.rb
class Person
  attr_accessor :name, :addresses # create attribute accessors
  def initialize (name)
    @name = name
    @addresses = []
  end
  def add_addresses(*addresses)
    # stuffs all passed in params into addresses array
    puts "Addresses passed in #{addresses}"
    addresses.each { |each_address| @addresses << each_address }
  end
  def to_s
    "#{@name} lives at #{@addresses.join(" and ")}"
  end
end
```

Splat example (continued)

```
require_relative 'person' # load in person.rb

class Address
  def initialize (address)
    @address = address
  end
  def to_s
    @address
  end
end
addr1 = Address.new("10 Lexington Market")
addr2 = Address.new("11100 Johns Hopkins Road")
person1 = Person.new("Joe")
person1.add_addresses(addr1) # => Addresses passed in [10 Lexington Market]
puts person1 # => Joe lives at 10 Lexington Market
person2 = Person.new("Jim")
person2.add_addresses(addr1, addr2)
# => Addresses passed in [10 Lexington Market, 11100 Johns Hopkins Road]
puts person2 # => Jim lives at 10 Lexington Market and 11100 Johns Hopkins Road
```

Splat reversed example

```
# Splat reversed
def three_args_no_less_and_no_more (one, two, three)
   puts "#{one}, #{two}, #{three}"
end

some_array = [1, 2, 3]

three_args_no_less_and_no_more(*some_array) # => 1, 2, 3
```