CMSC 330: Organization of Programming Languages

Introduction to Ruby

Clickers improve student engagement

Biochem Mol Biol Educ. 2009 Mar;37(2):84-91. doi: 10.1002/bmb.20264.

Using clickers to improve student engagement and performance class.

Addison S[1], Wright A, Milner R.

Author information

Abstract

Students say

ren
@rennnn__

Follow

Clickers are the invention of satan I'm convinced.

5:45 PM - 26 Nov 2012 · San Diego, CA, United States

Rachel Paddock
@RachelPaddock

Whoever invented clickers.... I despise you.

11:33 AM - 29 Nov 2012

1

Cait Corf
@caitcorf

Follow

BUT WHY MUST I BE SO STUPID?! The only reason I stayed is because it this class has I clickers,guess what I forgot to bring to class today?

12:18 PM - 15 Mar 2013

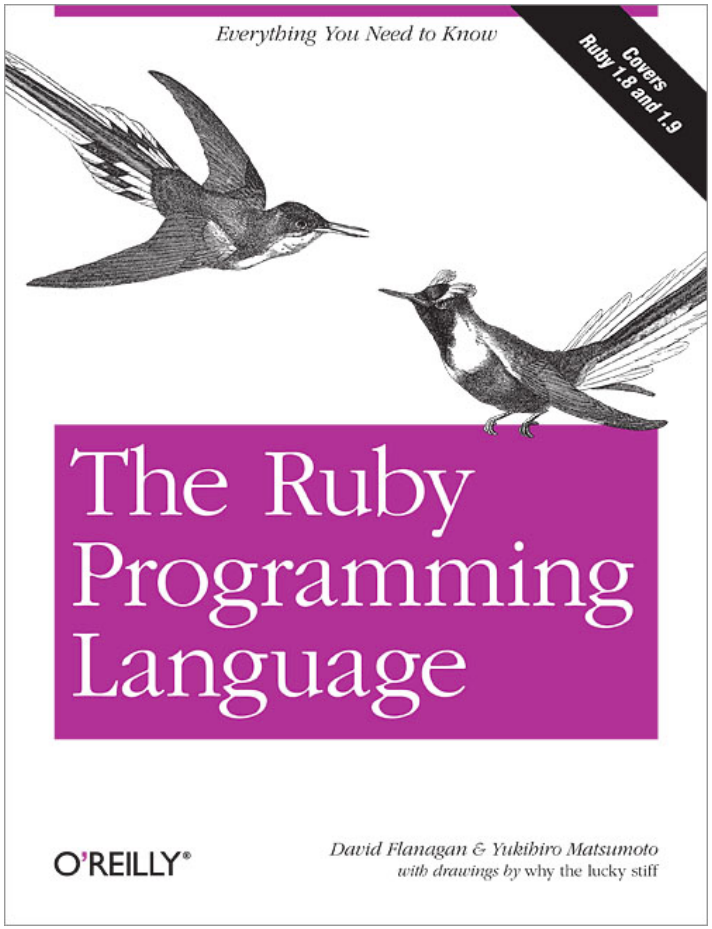
I have my clicker

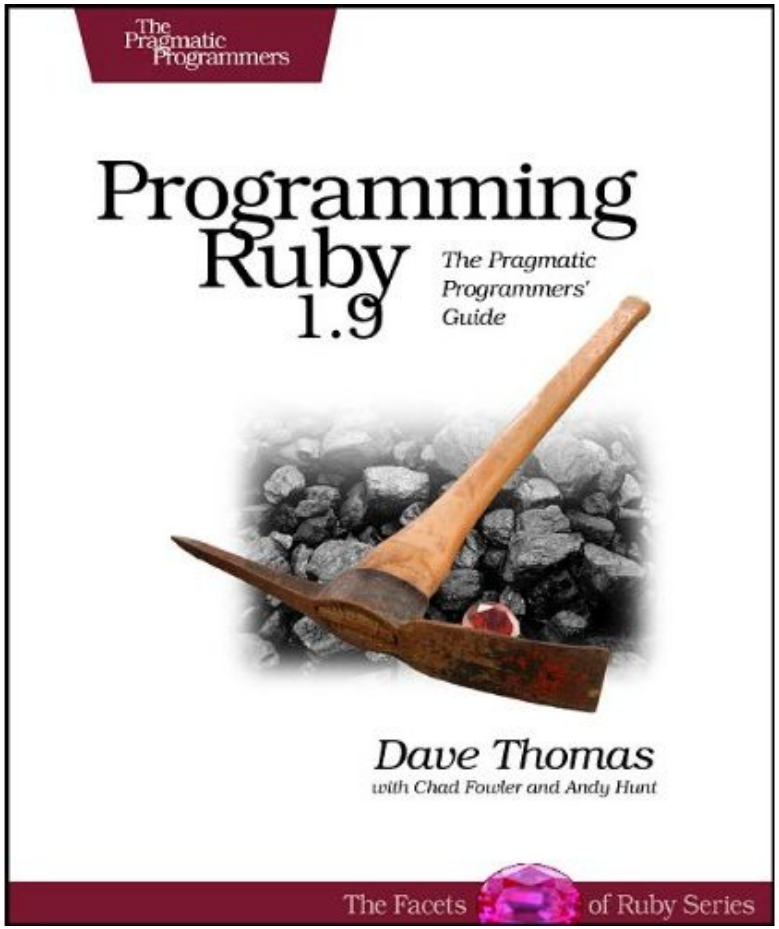
A. True
B. False

Ruby

- An *object-oriented, imperative, dynamically typed (scripting) language*
  - Created in 1993 by Yukihiro Matsumoto (Matz)
  - “Ruby is designed to make programmers happy”
  - Core of Ruby on Rails web programming framework (a key to its popularity)
  - Similar in flavor to many other scripting languages
    - Much cleaner than perl
  - Full object-orientation (even primitives are objects!)

Books on Ruby





- Earlier version of Thomas book available on web
  - See course web page

Applications of Scripting Languages

- Scripting languages have many uses
  - Automating system administration
  - Automating user tasks
  - Quick-and-dirty development

- Motivating application

Text processing

Output from Command-Line Tool

```
% wc *
     271     674    5323 AST.c
     100     392    3219 AST.h
     117    1459  238788 AST.o
    1874    5428   47461 AST_defs.c
    1375    6307   53667 AST_defs.h
     371     884    9483 AST_parent.c
     810    2328   24589 AST_print.c
     640    3070   33530 AST_types.h
     285     846    7081 AST_utils.c
      59     274    2154 AST_utils.h
      50     400   28756 AST_utils.o
     866    2757   25873 Makefile
     270     725    5578 Makefile.am
     866    2743   27320 Makefile.in
      38     175    1154 alloca.c
    2035    4516   47721 aloctypes.c
      86     350    3286 aloctypes.h
     104    1051   66848 aloctypes.o

...
```

Climate Data for IAD in August, 2005

1	2	3	4	5	6A	6B	7	8	9	10	11	12	13	14	15	16	17	18
										AVG	MX	2MIN						
DY	MAX	MIN	AVG	DEP	HDD	CDD	WTR	SNW	DPTH	SPD	SPD	DIR	MIN	PSBL	S-S	WX	SPD	DR
1	87	66	77	1	0	12	0.00	0.0	0	2.5	9	200	M	M	7	18	12	210
2	92	67	80	4	0	15	0.00	0.0	0	3.5	10	10	M	M	3	18	17	320
3	93	69	81	5	0	16	0.00	0.0	0	4.1	13	360	M	M	2	18	17	360
4	95	69	82	6	0	17	0.00	0.0	0	3.6	9	310	M	M	3	18	12	290
5	94	73	84	8	0	19	0.00	0.0	0	5.9	18	10	M	M	3	18	25	360
6	89	70	80	4	0	15	0.02	0.0	0	5.3	20	200	M	M	6	138	23	210
7	89	69	79	3	0	14	0.00	0.0	0	3.6	14	200	M	M	7	1	16	210
8	86	70	78	3	0	13	0.74	0.0	0	4.4	17	150	M	M	10	18	23	150
9	76	70	73	-2	0	8	0.19	0.0	0	4.1	9	90	M	M	9	18	13	90
10	87	71	79	4	0	14	0.00	0.0	0	2.3	8	260	M	M	8	1	10	210

...

Raw Census 2000 Data for DC

```
u108_S,DC,000,01,0000001,572059,72264,572059,12.6,572059,572059,572059,0,0,
    0,0,572059,175306,343213,2006,14762,383,21728,14661,572059,527044,15861
    7,340061,1560,14605,291,1638,10272,45015,16689,3152,446,157,92,20090,43
    89,572059,268827,3362,3048,3170,3241,3504,3286,3270,3475,3939,3647,3525
    ,3044,2928,2913,2769,2752,2933,2703,4056,5501,5217,4969,13555,24995,242
    16,23726,20721,18802,16523,12318,4345,5810,3423,4690,7105,5739,3260,234
    7,303232,3329,3057,2935,3429,3326,3456,3257,3754,3192,3523,3336,3276,29
    89,2838,2824,2624,2807,2871,4941,6588,5625,5563,17177,27475,24377,22818
    ,21319,20851,19117,15260,5066,6708,4257,6117,10741,9427,6807,6175,57205
    9,536373,370675,115963,55603,60360,57949,129440,122518,3754,3168,22448,
    9967,4638,14110,16160,165698,61049,47694,13355,71578,60875,10703,33071,
    35686,7573,28113,248590,108569,47694,60875,140021,115963,58050,21654,36
    396,57913,10355,4065,6290,47558,25229,22329,24058,13355,10703,70088,657
    37,37112,21742,12267,9475,9723,2573,2314,760,28625,8207,7469,738,19185,
    18172,1013,1233,4351,3610,741,248590,199456,94221,46274,21443,24831,479
    47,8705,3979,4726,39242,25175,14067,105235,82928,22307,49134,21742,1177
    6,211,11565,9966,1650,86,1564,8316,54,8262,27392,25641,1751,248590,1159
    63,4999,22466,26165,24062,16529,12409,7594,1739,132627,11670,32445,2322
    5,21661,16234,12795,10563,4034,248590,115963,48738,28914,19259,10312,47
    48,3992,132627,108569,19284,2713,1209,509,218,125
...
```

A Simple Example

- Let's start with a simple Ruby program

ruby1.rb:

```
# This is a ruby program
x = 37
y = x + 5
print(y)
print("\n")
```

```
% ruby -w ruby1.rb
42
%
```

Language Basics

comments begin with #, go to end of line

variables need not be declared

```
# This is a ruby program
x = 37
y = x + 5
print(y)
print("\n")
```

line break separates expressions (can also use ";" to be safe)

no special main() function or method

Run Ruby, Run

There are two basic ways to run a Ruby program

- ruby -w *filename* – execute script in *filename*
  - tip: the -w will cause Ruby to print a bit more if something bad happens
  - Ruby filenames should end with '.rb' extension
- irb – launch interactive Ruby shell
  - Can type in Ruby programs one line at a time, and watch as each line is executed

    irb(main):001:0> 3+4

    ⇒7
  - Can load Ruby programs via load command
    - Form: load *string*
    - String must be name of file containing Ruby program
    - E.g.: load 'foo.rb'

▸Ruby 1.9.3 is installed on Grace cluster (upgrading to 2.4)

Run Ruby, Run (cont.)

- Suppose you want to run a Ruby script as if it were an executable (e.g. “double-click”, or as a command)
  - Windows
    - Must associate .rb file extension with ruby command
    - If you installed Ruby using the Windows installer, this was done automatically
    - The Ruby web site has information on how to make this association

Run Ruby, Run (cont.)

- Suppose you want to run a Ruby script as if it were an executable (cont.)
  - *nix (Linux / Unix / etc.)

```
#!/usr/local/bin/ruby -w
print("Hello, world!\n")
```

    - The first line ("shebang") tells the system where to find the program to interpret this text file
    - Must chmod u+x *filename* first, or chmod a+x *filename* so everyone has exec permission
    - Warning: Not very portable: Depends on location of Ruby interpreter
      - /usr/local/bin/ruby vs. /usr/bin/ruby vs. /opt/local/bin/ruby etc.

Some Ruby Language Features

- Implicit declarations
  - Java, C have explicit declarations
- Dynamic typing
  - Java, C have (mostly) static typing
- Everything is an object
  - No distinction between objects and primitive data
  - Even “null” is an object (called *nil* in Ruby), as are classes
- No outside access to private object state
  - *Must* use getters, setters
- No method overloading
- Class-based and Mixin inheritance

Implicit vs. Explicit Declarations

- In Ruby, variables are implicitly declared
  - First use of a variable declares it and determines type

```
x = 37;  // no declaration needed – created when assigned to
y = x + 5
```

    - x, y now exist, are integers

- Java and C/C++ use explicit variable declarations
  - Variables are named and typed before they are used

```
int x, y;   // declaration
x = 37;   // use
y = x + 5;  // use
```

Tradeoffs?

Explicit Declarations	Implicit Declarations
More text to type	Less text to type
Helps prevent typos	Easy to mistype variable name

```
var = 37
If (rare-condition)
  y = vsr + 5
```

Typo!

Only caught when this line is actually run.

Bug could be latent for quite a while

Static Type Checking (Static Typing)

- Before program is run
  - Types of all expressions are determined
  - Disallowed operations cause compile-time error
    - Cannot run the program

- Static types are often explicit (*aka* manifest)
  - Specified in text (at variable declaration)
    - C, C++, Java, C#
  - But may also be inferred – compiler determines type based on usage
    - OCaml, C# and Go (limited)

Dynamic Type Checking

- During program execution
  - Can determine type from run-time value
  - Type is checked before use
  - Disallowed operations cause run-time exception
    - Type errors may be latent in code for a long time
- Dynamic types are *not* manifest (aka implicit)
  - Variables are just introduced/used without types
  - Examples
    - **Ruby**, Python, Javascript, Lisp

Static and Dynamic Typing

- Ruby is dynamically typed, C is statically typed

```ruby
# Ruby
x = 3
x = "foo"   # gives x a
            # new type
x.foo       # NoMethodError
            # at runtime
```

```c
/* C */
int x;
x = 3;
x = "foo"; /* not allowed */
/* program doesn't compile */
```

- Notes
  - Can always run the Ruby program; may fail when run
  - C variables declared, with types
    - Ruby variables declared *implicitly*
    - Implicit declarations most natural with dynamic typing

Tradeoffs?

- Static type checking
  - More work for programmer (at first)
    - Catches more (and subtle) errors at compile time
  - Precludes some correct programs
    - May require a contorted rewrite
  - More efficient code (fewer run-time checks)
- Dynamic type checking
  - Less work for programmer (at first)
    - Delays some errors to run time
  - Allows more programs
    - Including ones that will fail
  - Less efficient code (more run-time checks)

Java: *Mostly* Static Typing

- In Java, types are mostly checked statically

```
Object x = new Object();
x.println("hello");   // No such method error at compile time
```

- But sometimes checks occur at run-time

```
Object o = new Object();
String s = (String) o;  // No compiler warning, fails at run time
// (Some Java compilers may be smart enough to warn about
   above cast)
```

Quiz 1: Get out your clickers!

- True or false: This program has a type error

```
# Ruby
x = 3
y = "foo"
x = y
```

A. True
B. False

- True or false: This program has a type error

```
/* C */
void foo() {
  int x = 3;
  char *y = "foo";
  x = y;
}
```

A. True
B. False

Quiz 1: Get out your clickers!

- True or false: This program has a type error

```
# Ruby
x = 3
y = "foo"
x = y
```

A. True
B. False

- True or false: This program has a type error

```
/* C */
void foo() {
  int x = 3;
  char *y = "foo";
  x = y;
}
```

A. True
B. False

Control Statements in Ruby

- A control statement is one that affects which instruction is executed next
  - While loops
  - Conditionals

```
i = 0
while i < n
  i = i + 1
end
```

```
if grade >= 90 then
  puts "You got an A"
elsif grade >= 80 then
  puts "You got a B"
elsif grade >= 70 then
  puts "You got a C"
else
  puts "You're not doing so well"
end
```

Conditionals and Loops Must End!

- All Ruby conditional and looping statements must be terminated with the end keyword.
- Examples

```
if grade >= 90 then
  puts "You got an A"
end
```

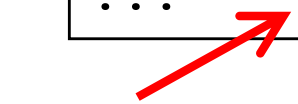
```
if grade >= 90 then
  puts "You got an A"
else
  puts “No A, sorry"
end
```

```
i = 0
while i < n
  i = i + 1
end
```

What is True?

- The guard of a conditional is the expression that determines which branch is taken

```
if grade >= 90 then
...
```



Guard

- The true branch is taken if the guard evaluates to anything except
  - false
  - nil
- Warning to C programmers: **0 is not false!**

Yet More Control Statements in Ruby

- unless cond then stmt-f else stmt-t end
  - Same as "if not cond then stmt-t else stmt-f end"

```
unless grade < 90 then
  puts "You got an A"
else unless grade < 80 then
  puts "You got a B"
end
```

- until cond body end
  - Same as "while not cond body end"

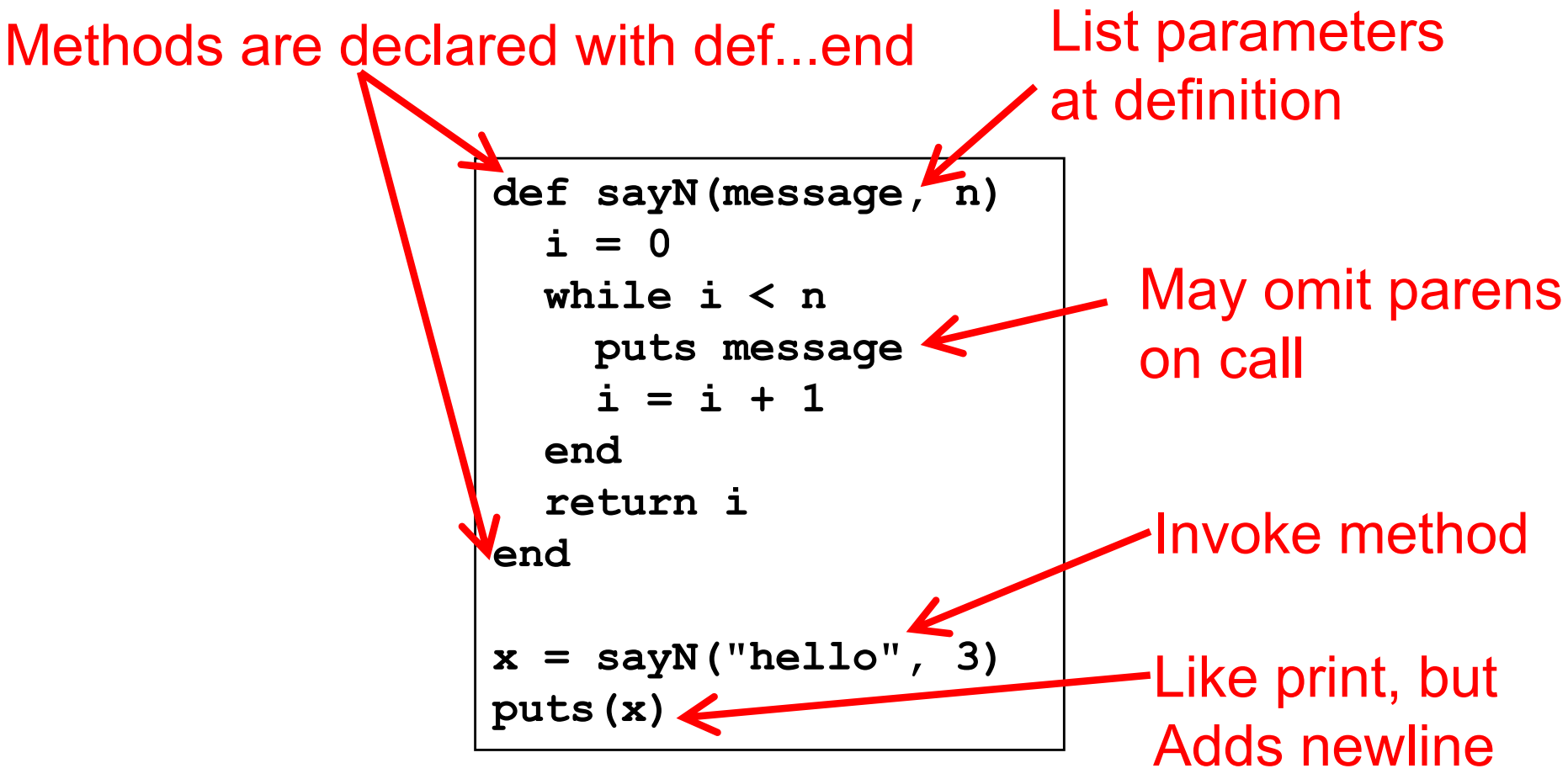
```
until i >= n
  puts message
  i = i + 1
end
```

Using If and Unless as Modifiers

- Can write if and unless after an expression
  - puts "You got an A" if grade >= 90
  - puts "You got an A" unless grade < 90

- Why so many control statements?
  - Is this a good idea? Why or why not?
    - **Good**: can make program more readable, expressing programs more directly. In natural language, many ways to say the same thing, which supports brevity and adds style.
    - **Bad**: many ways to do the same thing may lead to confusion and hurt maintainability (if future programmers don't understand all styles)

Methods in Ruby



Methods should begin with lowercase letter and be defined before they are called
Variable names that begin with uppercase letter are *constants* (only assigned once)

Terminology

- Formal parameters
  - Variable parameters used in the method
  - def sayN(message, n) in our example
- Actual arguments
  - Values passed in to the method at a call
  - x = sayN("hello", 3) in our example
- Top-level methods are “global”
  - Not part of a class. sayN is a top-level method.

Method Return Values

- Value of the return is the value of the last executed statement in the method
  - These are the same:

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

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

- Methods can return multiple results (as an Array)

```
def dup(x)
  return x,x
end
```

Everything is an Object

- All values are (references to) objects
  - Java/C/C++ distinguish *primitives* from *objects*
- Objects communicate via method calls
- Each object has its own (private) state
- Every object is an instance of a class
  - An object's class determines its behavior:
  - The class contains method and field definitions
    - Both instance fields and per-class ("static") fields

Everything is an Object

- Examples
  - (-4).abs  No-argument instance method of Fixnum
    - integers are instances of class Fixnum
  - 3 + 4
    - infix notation for "invoke the + method of 3 on argument 4"
  - "programming".length
    - strings are instances of String
  - String.new
    - classes are objects with a new method
  - 4.13.class
    - use the class method to get the class for an object
    - floating point numbers are instances of Float

Classes

- Class names begin with an uppercase letter
- The new method creates an object
  - s = String.new creates a new String and makes s refer to it
- Every class inherits from Object

Objects and Classes

- Objects are data
- Classes are types (the kind of data which things are)
- Classes are *also* objects

Object	Class (aka *type*)
10	Fixnum
-3.30	Float
"CMSC 330"	String
String.new	String
[‘a’, ‘b’, ‘c’]	Array
Integer	Class

- Integer, Float, and String are *objects* of type Class
  - So is Class itself!

Two Cool Things to Do with Classes

- Since classes are objects, you can manipulate them however you like
  - Here, the type of y depends on p
    - Either a String or a Time object

```
if p then
  x = String
else
  x = Time
End
y = x.new
```

- You can get names of all the methods of a class
  - Object.methods
    - => ["send", "name", "class_eval", "object_id", "new", "autoload?", "singleton_methods", ...]

The nil Object

- Ruby uses a special object nil
  - All uninitialized fields set to nil (@ prefix used for fields)

    ```
    irb(main):004:0> @x
    => nil
    ```

  - Like NULL or 0 in C/C++ and null in Java
- nil is an object of class NilClass
  - It's a *singleton object* – there is only one instance of it
    - NilClass does not have a **new** method
  - nil has methods like to_s, but not other methods

    ```
    irb(main):006:0> nil + 2
    NoMethodError: undefined method `+' for nil:NilClass
    ```

Quiz 2

- What is the type of variable x at the end of the following program?

```
p = 0
if p then
  x = nil
else
  x = "hello"
end
```

A. String
B. Integer
C. NilClass
D. *Nothing* – there's a type error

Quiz 2

- What is the type of variable x at the end of the following program?

```
p = 0
if p then
  x = nil
else
  x = "hello"
end
```

A. String
B. Integer
C. **NilClass**
D. *Nothing* – there's a type error

Creating Strings in Ruby

- Substitution in double-quoted strings with #{ }
  - course = "330"; msg = "Welcome to #{course}"
  - "It is now #{Time.new}"
  - The contents of #{ } may be an arbitrary expression
  - Can also use single-quote as delimiter
    - No expression substitution, fewer escaping characters
- Here-documents

```
s = <<END
  This is a text message on multiple lines
  and typing \\n is annoying
END
```

Creating Strings in Ruby (cont.)

- Ruby has printf and sprintf
  - printf("Hello, %s\n", name);
  - sprintf("%d: %s", count, Time.now)
    - Returns a String
- to_s returns a String representation of an object
  - Can be invoked implicitly – write puts(p) instead of puts(p.to_s)
    - Like Java's toString()
- inspect converts any object to a string

```
irb(main):033:0> p.inspect
=> "#<Point:0x54574 @y=4, @x=7>"
```

Standard Library: String

- The String class has many useful methods
  - s.length # length of string
  - s1 == s2 # structural equality (string contents)
  - s = "A line\n"; s.chomp # returns "A line"
    - Return new string with s's contents except newline at end of line removed
  - s = "A line\n"; s.chomp!
    - Destructively removes newline from s
    - *Convention:* methods ending in ! modify the object
    - *Another convention:* methods ending in ? observe the object

Symbols

- Ruby *symbols* begin with a colon
  - :foo, :baz_42, :"Any string at all"
- Symbols are “interned” Strings
  - The same symbol is at the same physical address
  - Can be compared with physical equality

```
“foo” == “foo”       # true
“foo”.equal? “foo”   # false
:foo == :foo         # true
:foo.equal :foo      # true
```

- Are symbols worth it? Probably not…

Defining Your Own Classes

```
class Point
  def initialize(x, y)
    @x = x
    @y = y
  end

  def add_x(x)
    @x += x
  end

  def to_s
    return "(" + @x.to_s + "," + @y.to_s + ")"
  end
end

p = Point.new(3, 4)
p.add_x(4)
puts(p.to_s)
```

class name is uppercase

constructor definition

instance variables prefixed with "@"

method with no arguments

instantiation

invoking no-arg method

No Outside Access To Internal State

- Instance variables (with @) can be directly accessed only by instance methods
- Outside class, they require accessors:

A typical getter

```
def x
  @x
end
```

A typical setter

```
def x= (value)
  @x = value
end
```

- Very common, so Ruby provides a shortcut

```
class ClassWithXandY
  attr_accessor :x, :y
end
```

Says to generate the x= and x and y= and y methods

No Method Overloading in Ruby

- Thus there can only be one initialize method
  - A typical Java class might have two or more constructors
- No overloading of methods in general
  - You can code up your own overloading by using a variable number of arguments, and checking at run-time the number/types of arguments
- Ruby does issue an exception or warning if a class defines more than one initialize method
  - But last initialize method defined is the valid one

Quiz 3: What is the output?

```
class Animal
   def eat(food)
      "I ate #{food}”
   end
   def eat(food,amount)
       "I ate #{amount} pounds of #{food}”
   end
end
animal = Animal.new
puts animal.eat("meat")
```

A. I ate #{food}
B. I ate meat
C. I ate pounds of meat
D. Error

Quiz 3: What is the output?

```
class Animal
   def eat(food)
      "I ate #{food}”
   end
   def eat(food,amount)
       "I ate #{amount} pounds of #{food}”
   end
end
animal = Animal.new
puts animal.eat("meat")
```

A. I ate #{food}
B. I ate meat
C. I ate pounds of meat
D. **Error**

Quiz 4: What is the output?

```
class Animal
   def eat(food)
      "I ate #{food}”
   end
def eat(food,amount)
      "I ate #{amount} pounds of #{food}”
   end
End
animal = Animal.new
puts animal.eat("meat”,23)
```

A. I ate meat
B. I ate 23 pounds of meat
C. Error
D. I ate #{amount} pounds pf #{food}

Quiz 4: What is the output?

```
class Animal
   def eat(food)
      "I ate #{food}”
   end
def eat(food,amount)
       "I ate #{amount} pounds of #{food}”
   end
End
animal = Animal.new
puts animal.eat("meat”,23)
```

A. I ate meat
B. **I ate 23 pounds of meat**
C. Error
D. I ate #{amount} pounds pf #{food}

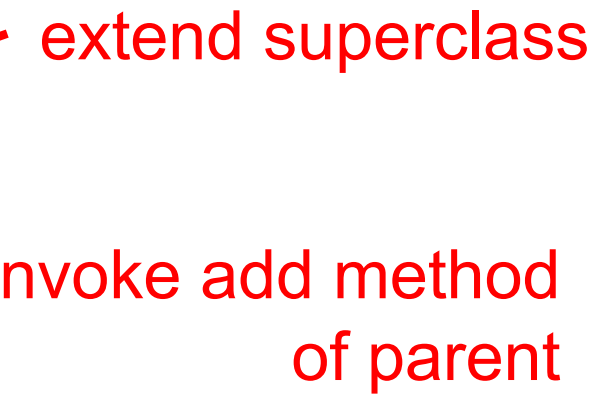
Inheritance

- Recall that every class inherits from Object

```
class A          ## < Object
  def add(x)
    return x + 1
  end
end

class B < A
  def add(y)
    return (super(y) + 1)
  end
end

b = B.new
puts(b.add(3))
```




```
b.is_a? A
true
b.instance_of? A
false
```

Mixins

- Another form of code reuse is “mix-in” inclusion
  - `include A` “inlines” A’s methods at that point
    - Referred-to variables/methods captured from context
    - In effect: it adds those methods to the current class

```
class OneDPoint
  attr_accessor :x
  include Comparable
  def <=>(other)# used by Comparable
    if @x < other.x then return -1
    elsif @x > other.x then return 1
    else return 0
    end
  end
end
```

```
p = OneDPoint.new
p.x = 1
q = OneDPoint.new
q.x = 2
x < y # true
puts [y,x].sort
# prints x, then y
```

super( ) in Ruby

- Within the body of a method
  - Call to super( ) acts just like a call to that original method
  - Except that search for method body starts in the superclass of the object that was found to contain the original method

Global Variables in Ruby

- Ruby has two kinds of global variables
  - Class variables beginning with @@ (`static` in Java)
  - Global variables across classes beginning with $

```
class Global
  @@x = 0

  def Global.inc
    @@x = @@x + 1; $x = $x + 1
  end

  def Global.get
    return @@x
  end
end
```

```
$x = 0
Global.inc
$x = $x + 1
Global.inc
puts(Global.get)
puts($x)
```

define a class ("singleton") method

Special Global Variables

- Ruby has a special set of global variables that are implicitly set by methods
- The most insidious one: $_
  - Last line of input read by gets or readline
- Example program

```
gets     # implicitly reads input line into $_
print    # implicitly prints out $_
```

- Using $_ leads to shorter programs
  - And confusion
  - We suggest you avoid using it

What is a Program?

- In C/C++, a program is...
  - A collection of declarations and definitions
  - With a distinguished function definition
    - int main(int argc, char *argv[]) { ... }
  - When you run a C/C++ program, it's like the OS calls main(...)
- In Java, a program is...
  - A collection of class definitions
  - With some class (say, MyClass) containing a method
    - public static void main(String[] args)
  - When you run java MyClass, the main method of class MyClass is invoked

A Ruby Program is...

- The class Object
  - When the class is loaded, any expressions not in method bodies are executed

defines a method of Object
(i.e., top-level methods belong to Object)

invokes self.sayN

invokes self.puts
(part of Object)

```
def sayN(message, n)
  i = 0
  while i < n
    puts message
    i = i + 1
  end
  return i
end

x = sayN("hello", 3)
puts(x)
```

Quiz 5: What is the output?

```
class Thing
  @@things = 0
  def initialize(name)
    @name = name
    @@things += 1
  end
  def self.get_things
    return @@things
  end
end
Thing.new("thing20");
Thing.new("thing6")
puts Thing.get_things
```

A. 0

B. 1

C. 2

D. 3

Quiz 5: What is the output?

```
class Thing
  @@things = 0
  def initialize(name)
    @name = name
    @@things += 1
  end
  def self.get_things
    return @@things
  end
end
Thing.new("thing20");
Thing.new("thing6")
puts Thing.get_things
```

A. 0

B. 1

C. **2**

D. 3

Quiz 6: What is the output?

```
class Dog
  def initialize
    @bark = "ruff ruff"
  end
  def speak
    "I like to #{@bark}"
  end
end
fido = Dog.new
puts fido.speak()
```

A. I like to bark
B. I like to #{@bark}
C. I like to ruff ruff
D. ruff ruff

Quiz 6: What is the output?

```
class Dog
  def initialize
    @bark = "ruff ruff"
  end
  def speak
    "I like to #{@bark}"
  end
end
fido = Dog.new
puts fido.speak()
```

A. I like to bark
B. I like to #{@bark}
C. **I like to ruff ruff**
D. ruff ruff

Quiz 7: What is the output?

```
class Computer
  def initialize
    @sound = "beep beep"
  end
  def self.about
    "Sometimes I go #{@sound}"
  end
end
print Computer.about
```

A. Sometimes I go #{@sound}"
B. Sometimes I go
C. Error
D. Sometimes I go nil

Quiz 7: What is the output?

```
class Computer
  def initialize
    @sound = "beep beep"
  end
  def self.about
    "Sometimes I go #{@sound}"
  end
end
print Computer.about
```

A. Sometimes I go #{@sound}"

B. **Sometimes I go**

C. Error

D. Sometimes I go nil

Quiz 8: What is the output?

```
class Person
  def initialize(first, last)
    @first = first
    @last = last
  end
  def full_name
    "#{@first} #{@last}"
  end
end
class Doctor < Person
   def full_name
      "Dr. #{super}"
   end
end
d = Doctor.new("Phil", "McGraw")
print d.full_name
```

A. Dr. #{super}
B. Dr.
C. Dr. Phil McGraw
D. Error

Quiz 8: What is the output?

```
class Person
  def initialize(first, last)
    @first = first
    @last = last
  end
  def full_name
    "#{@first} #{@last}"
  end
end
class Doctor < Person
   def full_name
      "Dr. #{super}"
   end
end
d = Doctor.new("Phil", "McGraw")
print d.full_name
```

A. Dr. #{super}
B. Dr.
C. Dr. Phil McGraw
D. Error