To Do TODAY

1. Push something to first-commits on GitHub
2. Find your group! Submit on Discord:
   1. Group name

To Ponder

Evaluate

```
 1/3   /   1/2

-1/3   /   1/2

 1/3r  /   1/2r

(1/3r) / (1/2r)

0.1 + 0.2 - 0.3
```

Ruby: Introduction, Basics

Lecture 3

Sample Code Snippet

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: %i[edit, update]

  def update
    if @user.update(user_params)
      redirect_to user_url(@user), notice: "Success."
    else
      render :edit, status: :unprocessable_entity
    end
  end

  def user_params
      params.require(:user).permit(:name, :email,
                                   :password)
  end
end
```

Ruby vs Java: Similarities

- Imperative and object-oriented
  - Classes and instances (ie objects)
  - Inheritance
- Strongly typed
  - Classes determine valid operations
- Some familiar operators
  - Arithmetic, bitwise, comparison, logical
- Some familiar keywords
  - **`if`**, **`then`**, **`else`**, **`while`**, **`for`**, **`class`**, **`new`**…

But Ruby Looks Different

- Punctuation
  - Omits ;'s and often ()'s on function calls
  - Function names can end in ? or !
- New keywords and operators
  - **`def`**, **`do..end`**, **`yield`**, **`unless`**
  - **`**`** (exp), **`=~`** (match), **`<=>`** (spaceship)
- Rich core libraries
  - Collections: Hashes, Arrays
  - Strings and regular expressions
  - Enumerators for iteration

Deeper Differences As Well

- Interpreted (typically)
  - Run a program directly, without compiling
- Dynamically typed
  - Objects have types, variables don't
- Everything is an object
  - C.f. primitives in Java
- *Code* can be passed into a function as a parameter
  - Java has added this too ("lambdas")

Compiling Programs

- Program = Text file
  - Contains easy-to-understand statements like "print", "if", "while", *etc.*
- But a computer can only execute *machine instructions*
  - Instruction set architecture of the CPU
- A *compiler* translates the program (source code) into an executable (machine code)
  - Recall "Bugs World" from CSE 2231
- Examples: C, C++, Objective-C, Ada…

Interpreting Programs

- An interpreter reads a program and executes it *directly*
- Advantages
  - Platform independence
  - Read-eval-print loop (aka REPL)
  - Reflection
- Disadvantages
  - Speed
  - Later error detection (*i.e.*, at run time)
- Examples: JavaScript, Python, Ruby

Combination of Both

- A language is not *inherently* compiled or interpreted
  - A property of its implementation
- Sometimes a combination is used:
  - Compile source code into an intermediate representation (byte code)
  - Interpret the byte code
- Examples of combination: Java, C#

Ruby is (Usually) Interpretted

- REPL with Ruby interpreter, irb

```
$ irb
>> 3 + 4
=> 7
>> puts "hello world"
hello world
=> nil
>> def square(x) x**2 end
=> :square
>> square -4
=> 16
```

Literals

- Numbers (Integer, Float, Rational, Complex)
  `83, 0123, 0x53, 0b1010011, 0b101_0011`
  `123.45, 1.2345e2, 12345E-2`
  `2/3r, 4+3i`
- Strings
  - Delimeters " " and ' '
  - Interpolation of `#{…}` occurs (only) inside " "
    `"Sum 6+3 is #{6+3}"` *is* *"Sum 6+3 is 9"*
  - Custom delimeters with `%Q�…�` and `%q�…�`
- Ranges
  - 0..4 is end *inclusive* (0, 1, 2, 3, 4)
  - 0...4 is end *exclusive* (0, 1, 2, 3)
- Arrays and hashes (later)

Comments and Statements

- Single-line comments start with **#**
  - Don't confuse it with string interpolation!
- Multi-line comments bracketed by

```
=begin
=end
```

  - Must appear at beginning of line
- Every statement has a value result
- Convention: **=>** to indicate result

```
"Hi #{name}" + "!" #=> "Hi Liam!"
puts "Bye #{name}" #=> nil
```

Operators

- Arithmetic: `+   -   *   /   %   **`
  - / is either ÷ or div, depending on operands
  - Integer / (div) rounds towards -∞, not 0
  - % is modulus, not remainder

```
1 / 3.0 #=> 0.3333333333333333
1 / 3   #=> 0 (same as Java)
-1 / 3  #=> -1 (not 0, differs from Java)
-1 % 3  #=> 2 (not -1, differs from Java)
```

- Bitwise: `~   |   &   ^   <<   >>`

```
5 | 2  #=> 7 (ie 0b101 | 0b10)
13 ^ 6 #=> 11 (ie 0b1101 ^ 0b0110)
5 << 2 #=> 20 (ie 0b101 << 2)
```

To Ponder

Evaluate

```
 1/3   /   1/2

-1/3   /   1/2

 1/3r  /   1/2r

(1/3r) / (1/2r)

0.1 + 0.2 - 0.3
```

To Ponder

Evaluate

```
 1/3    /   1/2      #=> 0

-1/3    /   1/2      #=> -1

 1/3r   /   1/2r     #=> 1/6r

(1/3r) / (1/2r)    #=> 2/3r

0.1 + 0.2 - 0.3    #=> 5.55111512312e-17
```

Operators (Continued)

- Comparison: `<  >  <=  >=  <=>`
  - Last one is so-called "spaceship operator"
  - Returns -1/0/1 iff LHS is smaller/equal/larger than RHS

    ```
    'cab' <=> 'da' #=> -1
    'cab' <=> 'ba' #=> 1
    ```
- Logical: `&&  ||  !  and  or  not`
  - Words have low precedence (below =)
  - "do_this or do_that" idiom needs low-binding

```
x = crazy or raise 'problem'
```

Pseudo Variables

- Objects
  - `self`, the receiver of the current method (recall "this" keyword in Java)
  - `nil`, nothingness (recall null)
- Booleans
  - `true`, `false`
  - nil evaluates to false
  - 0 is *not* false, it is true just like 1 or -4!
- Specials
  - `__FILE__`, the current source file name
  - `__LINE__`, the current line number

Significance in Names

- A variable's *name* affects semantics!
- Variable name determines its scope
  - Local: start with lowercase letter (or _)
  - Global: start with $
    - Many pre-defined global variables exist, *e.g.*:
      - `$/` is the input record separator (newline)
      - `$;` is the default field separator (space)
  - Instance: start with @
  - Class: start with @@
- Variable name determines mutability
  - Constant: start with uppercase (**`Size`**) but idiom is all upper case (**`SIZE`**)

Basic Statements: Conditionals

- Classic structure

```
if (boolean_condition) [then]
  ...
else
   ...
end
```

- But usually omit ( )'s and "then" keyword

```
if x < 10
  puts 'small'
end
```

- `if` can also be a *statement modifier*

```
x = x + 1 if x < LIMIT
```

  - Good for single-line body
  - Good when statement execution is common case
  - Good for positive conditions

Variations on Conditionals

- Unless: equivalent to "if not…"

```
unless size >= 100
  puts 'small'
end
```

  - Do not use else with unless
  - Do not use negative condition (`unless !...`)
- Can also be a statement modifier

```
x = x + 1 unless x >= LIMIT
```

  - Good for: single-line body, positive condition
  - Used for: Guard at beginning of method

```
raise 'negative argument' unless x >= 0
```

Pitfalls with Conditionals

- Keyword **`elsif`** (not "else if")

```
if x < 10
  puts 'small'
elsif x < 20
  puts 'medium'
else
  puts 'large'
end
```

- If's *do not* create nested lexical scope

```
if x < 10
  y = x
end
puts y # y is defined, but could be nil
puts z # NameError: undefined local var z
```

Case Statements are General

```
[variable = ] case expression
when nil
   statements execute if the expr was nil
when value # e.g. 0, 'start'
   statements execute if expr equals value
when type # e.g. String
   statements execute if expr resulted in Type
when /regexp/ # e.g. /[aeiou]/
   statements execute if expr matches regexp
when min..max
   statements execute if the expr is in range
else
   statements
end
```

Basic Iteration: While and Until

- Classic loop structure
  ```
  while boolean_condition [do]
    …
  end
  ```
  - Can also be used as a statement modifier
    ```
    work while awake
    ```
- `until` is equivalent to "while not…"
  ```
  until i > count
    …
  end
  ```
  - Can also be a used as a statement modifier
- Pitfall: Modified *block* executes at least once
  ```
  sleep while is_dark # may not sleep at all
  begin i = i + 1 end while i < MAX
               # always increments i at least once
  ```

Functions

- Definition: keyword **`def`**

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

- Notice: no types in signature
  - No types for parameters
  - No type for return value
- But all functions return *something*
  - Value of last statement is implicitly returned
  - Convention: Omit explicit return statement

```
def foo(x, y)
  x + y # last statement executed
end
```

Function Calls

- Dot notation for method call

```
Math::PI.rationalize() # recvr Math::PI
```

- Convention: Omit ( )'s in definition of functions with no parameters

```
def launch() … end # bad
def launch   … end # good
```

- Paren's can be omitted in calls too!

```
Math::PI.rationalize
puts 'hello world'
```

  - Convention: Omit for "keyword-like" calls

```
attr_reader :name, :age
```

  - Note: needed when chaining

```
foo(13).equal? value
```

Sample Code Snippet

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: %i[edit update]

  def update
    if @user.update(user_params)
      redirect_to @user, notice: "Success."
    else
      render :edit, status: :unprocessable_entity
    end
  end

  def user_params
      params.require(:user).permit(:name, :email,
                                       :password)
  end
end
```

Summary

- Ruby is a general-purpose, imperative, object-oriented language
- Ruby is (usually) interpreted
  - REPL
- Familiar flow-of-control and syntax
  - Some new constructs (e.g., unless, until)
  - Terse (e.g., optional parentheses, optional semicolons, statement modifiers)