Topic 3:

Object–Oriented Languages

"Object–oriented programming is an exceptionally bad idea which could only have originated in California."

— Edsger W. Dijkstra (supposedly!)

An Incomplete History of Object–Oriented Languages

- Simula $\rightarrow$ Simula67
  - A short–lived family of simulation languages
  - Needed coroutines, introduced classes to support them
- Smalltalk
  - Simula67 $\rightarrow$ Flex $\rightarrow$ Smalltalk–72,–80 (Alan Kay, Dynabook)
  - Strictly object–oriented (only objects, messages, & blocks)
  - Popularized graphical user interfaces (GUIs)
- C $\rightarrow \{$ C++, Java, C# $\}$, FORTRAN $\rightarrow$ Fortran 2003, ...
  - Popular imperative languages extended with OO features
- Ruby
  - A fully–objected–oriented scripting language
  - Created by Yukihiro 'Matz' Matsumoto in the 1990s

Why Do OO Languages Exist? (1 / 3)

Five often–cited reasons:

(1) Code Reuse

Classes written for one project can be easily used in others, via:

(a)

(b)

(c)

(d)

Why Do OO Languages Exist? (2 / 3)

(2)

(3)

Why Do OO Languages Exist? (3 / 3)

(4)

(5)

What Makes a Language “Object–Oriented”?

Commonly accepted characteristics:

- Objects and Classes! OK, sure, but beyond that . . .
-
-
-
-

Additional OO Language Considerations (1 / 2)

(1)

Additional OO Language Considerations (2 / 2)

(2)

Ways to add classes into a language's typing system:

Ruby: A Dynamic General–Purpose OO Language

A little of Ruby's history:

- Created by Yukihiro "Matz" Matsumoto in the 1990s
  - Why? Matz didn't like Perl or Python, but still wanted an object–oriented scripting language
  - Why 'Ruby'? A gemstone name, like 'Perl'
- Current version: 4.0.6 (released 2026–07–14)
  - Open–source (Ruby License and 2–clause BSDL)
  - Still actively maintained by Matz as 'BDFL' ('Benevolent Dictator for Life')
  - For our needs, 1.9.3 or later is adequate
  - Lectura's (Debian's) version is 3.2.3

```
$ ruby --version
```


Running Ruby (1 / 2)

Two basic options:

(1) Use a 'REPL'

- irb is Ruby's REPL (Read–Evaluate–Print Loop)
  - Handy for 'what happens in this case?' language questions
  - To run it on lectura: irb
  - To exit it: Ctrl–D
- Web sites offering online interactive Ruby include:
  - `try.ruby-lang.org`
  - `repl.it/languages/ruby` (must sign–up)

Running Ruby (2 / 2)

(2) Use a ruby interpreter

- Better than irb for running complete programs
- This interpreter converts Ruby source code to bytecode, then interprets (executes) the bytecode
- On lectura: ruby filename.rb

Test your Program #1 Ruby code on lectura before you use turnin to submit your code to the TAs for grading!

Getting Online Help with Ruby

1. Official online documentation
   - `ruby-doc.org/3.2.3` (Lectura's ruby version)
2. Reference links on the class web page
   - Free online books!
   - Find another good resource? Please let us know!
3. Google for an authoritative (non–AI) answer
4. (Last resort!) Consult a GenAI, LLM, etc.
   - Remember: We are allowing you to use AI tools on programming assignments only. Learn the languages yourself — you'll need to write code on exams!

Hello, World!

In Ruby, this can be literally a one–liner.

puts and printf are both Kernel module methods

Examples:

```
puts("Hello, World!")
puts "Hello, World!"

printf "Hello, World!\n"
```

Notes:

Basics of Variables

Ruby variables are not declared, and do not have types!

Examples:

Learning the ~~Types~~ Classes of Values

How? By asking the value for its class!

Examples:

Basic Operators

A partial precedence table: (which you do not need to learn!)

```
          !
         **
          -
      *  /  %
        +  -
  <  >  <=  >=
    <=>  !=
        &&
         ||
        ? :
 =   +=   -=
         not
      and   or
```

Operators as Methods

In the precedence table, all of the operators above $\&\&$ are methods. Thus, we can write them as method calls if we wish to do so:

$2 + 3 \quad \Longrightarrow$

$-5 \quad \Longrightarrow$

f2c2k1.rb ← Related example program (see class web page)

Strings (1 / 4)

Either single or double quotes delimit Ruby strings:

But use doubles for interpolation and escaped characters:

Strings (2 / 4)

Jave's strings are immutable; Ruby's strings are mutable.

```
x = "fu"
y = x
puts x.object_id, y.object_id
```


Strings (3 / 4)

More examples of Ruby string methods:

```
"␣abc␣\n".strip      ⇒
"␣abc␣\n".chomp      ⇒
"123abc".upcase      ⇒
"123abc".to_i        ⇒
"abc"[0]             ⇒
"banana".split("a") ⇒
```

To see the names of all of the String methods:

```
("a".methods - Object.methods).sort
```

Strings (4 / 4)

Two String method naming suffixes:

(1) Boolean method names end with a question mark

(2) Some methods also have a “bang” form, which (usually) modifies the receiver.

Miscellaneous Class and Value Details

(a) ‘Constants’ . . . aren’t, but names start w/ a capital letter

(b) `nil`

(c) Booleans

(d) What is `true`?

Compound Conditions

Recall that Ruby has two sets of logical operators:

High–precedence:	`&&`	`||`	`!`
Low–precedence:	`and`	`or`	`not`

But . . . why?

Selection Statements (1 / 4)

Ruby has the basics:

If:

```
if <condition> then
    <expression>
end
```

If – Else:

```
if <condition> then
    <expression>
else
    <expression>
end
```

leapyear1.rb

Selection Statements (2 / 4)

Remember the q if p form of if p then q from Discrete Math?

Ruby's got it!

Then there's the unless statement; it flips the sense of if:

Selection Statements (3 / 4)

Ruby has a case statement, but with some twists:

```
grade = case score
        when 90..100
          "A"
        when 80..89  then "B"
        when 70..79  then "C"
        when 60..69  then "D"
        else              "E"
        end
```

leapyear2.rb

Selection Statements (4 / 4)

Fun with === (the case equality operator)

Iteration (1 / 4)

Like many languages, Ruby has two forms of the while loop:

The While Statement:

```
while <condition> do
    <expression>
end
```

The While Modifier:

```
begin
    <expression>
end while <condition>
```

Iteration (2 / 4)

If you like unless for selection, for iteration you'll love . . .

The Until Statement:

```
until <condition> do
    <expression>
end
```

The Until Modifier:

```
begin
    <expression>
end until <condition>
```


Iteration (3 / 4)

Ruby does have a for loop . . .

```
for <variable> in <expression> do
    <expression>
end
```

. . . that is shunned by many Ruby programmers, in favor of . . .

Iteration (4 / 4)

. . . reliance on collection classes providing the .each method!

Forms:

```
<expr>.each do |<var>| <expr> end
<expr>.each { |<var>| <expr> }
```


Arrays (1 / 6)

In Ruby, arrays may hold references to any object.

Some options to create arrays:

How many elements are in an array?

Arrays (2 / 6)

Changing an array element is straight–forward:

There are multiple ways to append to an array:

Arrays (3 / 6)

Sub–arrays:

```
a = [1, 3.14, "yo", true]
```

2D Arrays:

```
t = [[1,2],[3,4]]
```

Arrays (4 / 6)

Ruby has array methods that support set operations.

```
s = [1,2,3]    t = [2,3,4]
```

Intersection:

Difference:

Union:

Membership:

Arrays (5 / 6)

Options for comparing arrays:

```
x = [1,2]    y = [1,2]    z = x
```

`x == y` ⇒

`x === y` ⇒

`x <=> y` ⇒

`x.object_id == y.object_id` ⇒

`x.object_id == z.object_id` ⇒

Arrays (6 / 6)

Useful Array content methods:

```
[2,1,3].sort ⇒

[2,1,3].sum  ⇒
```

Iterating through arrays:

```
for i in [2,1,3] do puts i end

[2,1,3].each { |i| puts i }
```

Iterating through strings via arrays:

```
for c in "hello".split("") do puts c end
```


String I/O (1 / 3)

For I/O with keyboard and screens, we have `gets` and `puts`.

To read a complete line as a string from the keyboard:

```
name = gets

Fred          ⇒

name.chomp!   ⇒
```

String I/O (2 / 3)

To read from a text file, we first open it for reading:

Once open, some options are:

The IO class has some non–File–object methods:

String I/O (3 / 3)

To write to a text file, we open it for writing:

```
myFile = File.new("filename", "r")
  :
myFile.close
```

Our writing methods include:

Block–based output automatically closes the file:

Learning About Existing Files

You can ask Ruby about a file's details:

```
File.file?("filename")

File.readable?("filename")

File.size?("filename")
```


Hashes (1 / 4)

'Hash' is Ruby's term for an associative list,

also known as map in Java and dictionary in Python.

To create a Hash object:

The purpose of the default value:

Hashes (2 / 4)

Storing key–value pairs into an existing hash is easy:

```
h={}
h[2]="two"              ⇐ 2 is the key, “two” is the value
h["key"]=true           ⇐ keys/values are heterogeneous
puts h
2=>"two", "key"=>true
```

As is replacing values of a key:

```
h[2]="dos"
puts h
2=>"dos", "key"=>true
```


Hashes (3 / 4)

Accessing the content of a hash: irb responses shown

```
h["key"] ⇒ true

h.keys   ⇒ [2, "key"]

h.values ⇒ ["dos", true]
```

Iteration over a hash — two options:

```
h.each do |k,v| puts k; puts v end

h.keys.each do |k| puts k; puts h[k] end
```

Hashes (4 / 4)

To associate one key with n values, make an array of values:

```
h={}
h[1] = ["one"]        ⇐ 1's value is the array
h[1] << "uno"
h[1] += ["eins"]
puts h                {1=>["one", "uno", "eins"]}
```

averages.rb

Stand–Alone ("Top–Level") Methods (1 / 5)

Comparable to static methods in Java.

Format:

Notes:

formati.rb

Aside: How to Load and Test Methods in irb

Loading is easy: `load "file.rb"`

⇒ But irb will load and execute the code!

To load the code without immediate execution, add protection:

```
<method definition code here>
begin
  <method invocation(s) here>
end if $0 != "irb"
```

formati2.rb

Stand–Alone ("Top–Level") Methods (2 / 5)

To which class do top–level methods belong? Ask Ruby!

```
self.class

self.class.private_methods
```

methodid.rb

Stand–Alone (“Top–Level”) Methods (3 / 5)

Review: Overloading allows multiple methods with the same name but different formal argument lists.

Can we overload methods in Ruby? Let’s try!

```
def one a; puts a end
def one b,c; puts b+c end
```

```
one "Hello"          ⇒
one "Hello", "World" ⇒
one 3, 4             ⇒
```

Answer:

Stand–Alone (“Top–Level”) Methods (4 / 5)

Review: Overriding is the replacement or hiding of an existing method with a new method of the same name.

Can we override methods in Ruby?

Stand–Alone ("Top–Level") Methods (5 / 5)

Ruby's scoping of variables is as expected:

```
def func a=2 end
a=1
func
puts a
```

The behavior changes when a global variable (danger!) is used:

```
def tion $a=2 end
$a=1
tion
puts $a
```


Creating Classes — Just the Basics (1 / 2)

- Form: `class <Name> ... end`
- Instance Variables are prefixed with @
- Constructor's name:
- Getters/Accessors: Share names of the instance variables

  Setters/Mutators: Ditto, with an equal–sign (=) suffix

Getter:
```
def x
  @x
end
```

Setter:
```
def x= (arg)
  @x = arg
end
```

Creating Classes — Just the Basics (2 / 2)

- Want your objects to have a string representation?
- All instance variables are private
- The class `MyClass` does not need to be stored in a file named `MyClass.rb`!

fraction.rb

Inheritance in Ruby

Ruby supports single inheritance only:

- Syntax: `class Subclass < Superclass`
- Subclasses inherit the superclasses' methods
- Subclass methods can call superclass methods
- Subclass methods may override superclass methods
  - Method calls resolve to the closest override

Ruby does not support:

- Abstract classes
- Interfaces
- Generics

account.rb & savings.rb

So: Is Ruby an Object–Oriented Language?

Checklist!

- Abstraction?
- Encapsulation?
- Inheritance?
- Polymorphism:
  - ad hoc?
  - parametric?
  - subtype?

Conclusion?

Is There More To Know About Ruby? Lots!

- Modules and the ‘mixin’ idea
  (adds capabilities to classes without inheriting them)
- “Duck Typing” (a duck if it walks, swims, and quacks like one!)
  (all varieties of objects that serve a specific purpose can be asked to perform that service in the same way)
- “Monkey Patching”
  (run–time modifications of existing classes)
- “Gems”
  (library add–ons, providing new functionality)
- Rails
  (a popular web development framework)

⋮