Methods

CS105 : Saelee

ILLINOIS INSTITUTE
OF TECHNOLOGY

When programs get too **large,** they are harder to **understand** and **maintain**

How to manage complexity?

e.g. Management

#@#$!

#$^%@

!@#$%@

&* #@!!

e.g., Car Interface

Gas, Brake, Clutch

Engine

Master Cylinder

Transmission

Headers

Valve Timing

Brake calipers

Synchros

Gear Ratios

Layout & Displacement

Hydraulics

Brake discs

Clutch Plate

e.g., the iPhone

Complexity is best managed through **abstraction**

e.g., Mastermind

Secret & Guesses → Hints

Prompt for and read guess

Count direct matches

Count indirect matches

Input/Output

Array iteration

Boolean expressions

Win condition

String to Integer conversion

Nested loops

Comparison logic

To **manage complexity,** it helps to break a program into **modules**

Each **module** should **implement** a clearly defined function

Code Modules

`puts` — Convert an object to a string and print it to the screen

`each` — Invoke a block for each element in an array

`last` — Fetch the last element in an array

`to_i` — Convert a String to an Integer

`gets` — Obtain a String from the keyboard

`*` — Multiply a number by another

In Ruby, **methods** are used to **modularize** code

The Ruby **core library** consists of a ton of prewritten **methods**

You can write your own methods, too…

Method Definition Syntax

- ```
  def method_name
    # the method "body"
    # is made up of this code
  end
  ```
- Method bodies can consist of an arbitrary amount of code
  - Variable definitions
  - Control structures
  - Method invocations
  ```

Definition and Invocation are separate steps!

Invoking (Calling) Methods

- We already know how to do this!
- Just “call” the method by its name
  - To invoke the method `def`’d as `method_name`:
    - `method_name`

Definition & Invocation

Method definition

Method invocation

```ruby
def foo
  puts "foo was invoked"
end

foo
foo
```

```
foo was invoked
foo was invoked
```

A defined method is **not run**
by the interpreter
until it is invoked

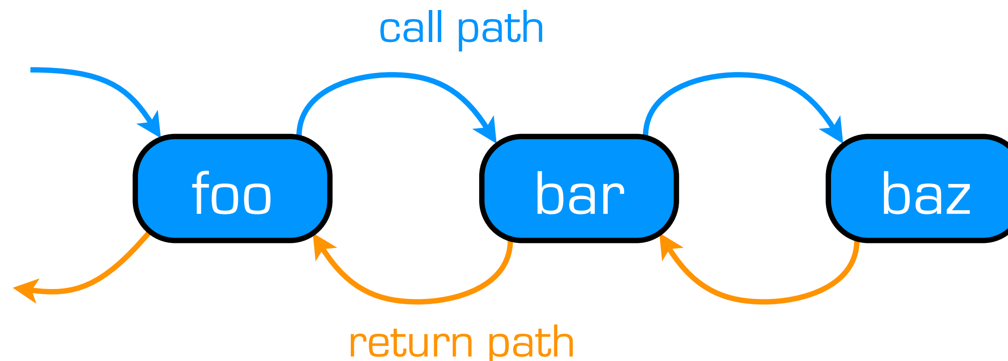
Definitions & Invocations

```ruby
def foo
  puts "foo was invoked"
end

def bar
  puts "bar was invoked"
end

def baz
  puts "baz was invoked"
  foo
  bar
end

foo
bar
baz
```

```
foo was invoked
bar was invoked
baz was invoked
foo was invoked
bar was invoked
```

Call and Return Order

- When multiple method calls are chained together, they return in a **last-in-first-out** (**LIFO**) order
- e.g., if we call **foo**, and **foo** calls **bar**, then **bar** calls **baz** ...
  - **baz** returns to **bar**, **bar** returns to **foo**, then **foo** returns



Variables in Methods

- Variables created and used within a method are “local” to that method (i.e., local **scope**)
  - They cannot be seen or modified outside that method
  - Allows us to not worry about accidental variable name conflicts between methods
- Note: applies to variables that begin with a **lowercase letter**
  - Capitalized variables are “global” constants (should only be assigned a value once)

Variable Scopes

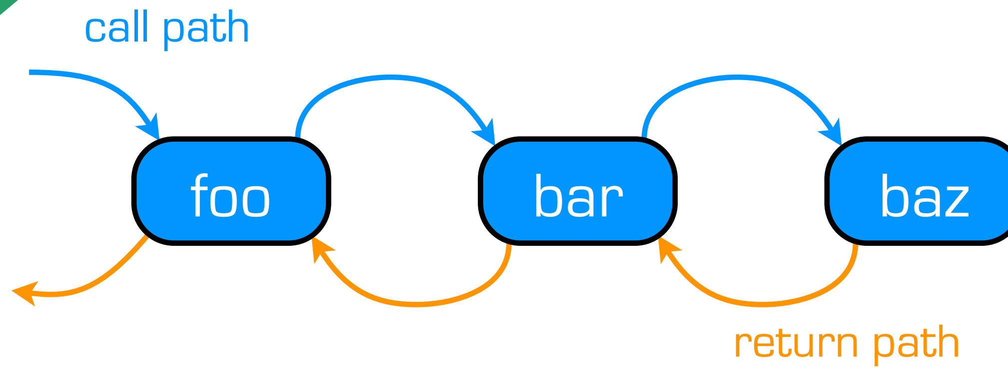
```ruby
SEEN_BY_ALL = "global setting"

def foo
  my_var = "foo's setting"
  bar
  puts my_var
  puts SEEN_BY_ALL
end

def bar
  my_var = "bar's setting"
  puts my_var
  puts SEEN_BY_ALL
end

my_var = "main setting"
foo
puts my_var
```

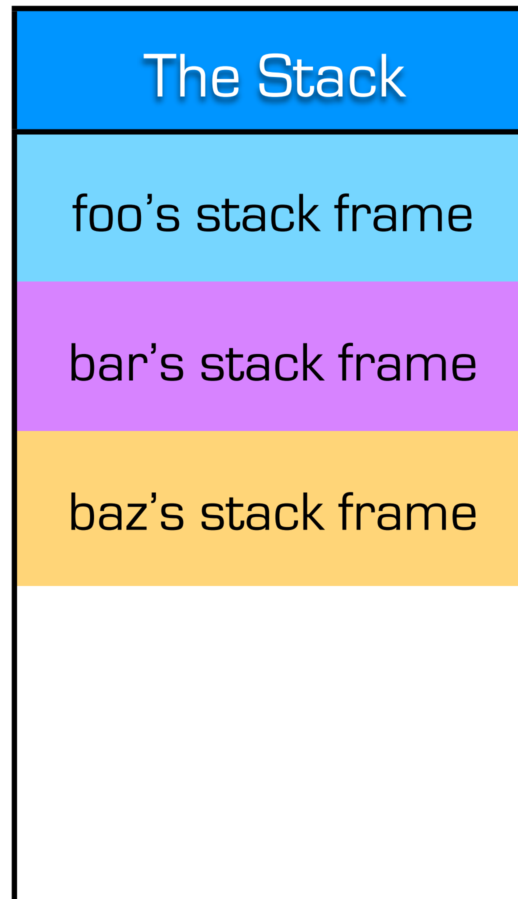
```
bar's setting
global setting
foo's setting
global setting
main setting
```

The Call **Stack**

- When a method is called, it reserves space for itself in memory in a region known as the **Stack**
  - When a method returns, it releases its stack space for reuse (by other methods)
  - Stack space is allocated/deallocated according to method call/return
    - a.k.a. "**LIFO memory management**"
- Local variables and other method specific items are stored on the stack

Stack Management





By convention, the stack grows top-down

Communication with methods: **Parameters** and **Return values**

Returning Values

- The “value” of a method is simply the **value of the last expression** in a method
- If this isn't good enough, we can also explicitly specify the return value of a function with the `return` statement

Method Parameters

- In the method **definition**, list variable names (comma separated) for parameters after the method name
  - Local variables that refer to parameter values
- In the method **invocation**, list values to be passed as parameters (comma separated) after the method name
  - **Any object** can be passed as a parameter
- Parentheses around the parameter lists are optional, but frequently a **good idea** for clarity

Parameters and Return Values

2 parameters

```ruby
def max(first, second)
  maxval = first
  if second > maxval
    maxval = second
  end
  maxval
end

puts max(82, 99)
```

last value of method (return value)

method call, passing parameters

Parameters and Return Values

```
def max(first, second)
  if first > second
    return first
  else
    return second
  end
end

puts max(82, 99)
```

explicit return values

```
99
```

Parameters and Return Values

```
def max(first, second)
  if first > second
    first
  else
    second
  end
end

puts max(82, 99)
```

explicit **return** not necessary

```
99
```

The "value" of an **if** statement is the last value of the clause that is executed

Note: Number of parameters in the definition and the call must **match**!

Argument Mismatch

```ruby
def foo(arg1, arg2, arg3)
  # do something
end

foo 1, 2
```

```
ArgumentError: wrong number of arguments (2 for 3)
   method foo in test.rb at line 5
   at top level in test.rb at line 5
```

What if I don't know how many parameters I need?

- Solution 1: pack them into an array, and pass the array as a parameter
- Solution 2: "splat" syntax for automatically designating a method parameter as a catch-all

“Splat” Parameter

Other parameters are automatically stored in this array

```ruby
def max(first, *rest)
  maxval = first
  rest.each do |val|
    if maxval < val
      maxval = val
    end
  end
  maxval
end

puts max(10, 2, 5, 12, 18, 8, 9)
```

```
18
```

Default Parameter Values

- Sometimes certain parameters may have **default values** that can be used if the caller doesn't specify them
- We can specify them in the method definition
  - They then become **optional** in the method call

Default Values

```ruby
def greet(name="Everyone")
  puts "Hello, #{name}!"
end

greet
greet "Michael"
```

```
Hello, Everyone!
Hello, Michael!
```