Ruby: Object-Oriented Concepts

Lecture 4

Classes

- Classes have methods and variables

```
class LightBulb     # name with CamelCase
  def initialize    # special method name
    @state = false # @ means "instance variable"
  end
  def on?
    @state          # implicit return
  end
  def flip_switch!  # name with snake_case
    @state = !@state
  end
end
```

- Instantiation calls *initialize* method

```
f = LightBulb.new #=> <LightBulb:0x0000e71c2322
                           @state=false>
f.on? #=> false
```

Visibility

- Instance variables are always private
  - Private to *object*, not class
- Methods can be private, protected, or public (default)

```
class LightBulb
  private def inside
    …
  end

  def access_internals(other_bulb)
    inside # ok
    other_bulb.inside # no! inside is private
    self.inside  # no explicit recv'r allowed
  end
end
```

Getters/Setters


```
class LightBulb
  def initialize(color, state: false)
    @color = color # not visible outside object
    @state = state # not visible outside object
  end
  def color
    @color
  end
  def state
    @state
  end
  def state=(value)
    @state = value
  end
end
```

Attributes


```
class LightBulb
  def initialize(color, state: false)
    @color = color
    @state = state
  end
  def color
    @color
  end

  attr_accessor :state # name is a symbol



end
```

Attributes


```
class LightBulb
  def initialize(color, state: false)
    @color = color
    @state = state
  end

  attr_reader :color


  attr_accessor :state




end
```

Attributes


```
class LightBulb
  attr_reader :color
  attr_accessor :state
  attr_writer :size

  def initialize(color, state: false)
    @color = color
    @state = state
    @size = 0
  end
end
```

Classes Are Always Open

- A class can always be extended

```
class Street
  def construction … end
end
…
class Street
  def repave … end # Street now has 2 methods
end
```

- Applies to core classes too

```
class Integer
  def log2_of_cube # lg(self^3)
    (self**3).to_s(2).length - 1
  end
end
500.log2_of_cube #=> 26
```

Classes are Always Open (!)

- Existing methods can be redefined!

- When done with system code (libraries, core …) called "monkey patching"
- Tempting, but… Just Don't Do It

No Overloading

- Method identified by (symbol) name
  - No distinction based on number of arguments
- Approximation: default arguments

```
def initialize(width, height = 10)
  @width = width
  @height = height
end
```

- Old alternative: trailing options hash

```
def initialize(width, options)
```

- Modern style: default keyword arguments

```
def initialize(height: 10, width:)
```

A Class is an Object Instance too

- Even classes are objects, created by :new

```
LightBulb = Class.new do #class LightBulb
  def initialize
    @state = false
  end
  def on?
    @state
  end
  def flip_switch!
    @state = !@state
  end
end
```

Instance, Class, Class Instance


```
class LightBulb
  @state1           # class instance var
  def initialize
    @state2 = …     # instance variable
    @@state3 = …    # class variable
  end
  def bar           # instance method
    …               # sees @state2, @@state3
  end
  def self.foo      # class method
    …               # sees @state1, @@state3
  end
end
```

Inheritance

- Single inheritance between classes

```
class LightBulb < Device
  …
end
```

  - Default superclass is Object (which inherits from BasicObject)
- Keyword **super** to call parent's method
  - No args means forward all args

```
class LightBulb < Device
  def electrify(current, voltage)
    do_work
    super # with current and voltage
  end
end
```

Modules

- Another container for definitions

```
module Stockable
  MAX = 1000
  class Item … end
  def self.inventory … end # utility fn
  def order … end
end
```

- Cannot, themselves, be instantiated

```
s = Stockable.new          # NoMethodError
i = Stockable::Item.new    # ok
Stockable.inventory        # ok
Stockable.order            # NoMethodError
```

Modules as Namespaces

- Modules create independent namespaces
  - cf. packages in Java
- Access contents via scoping (::)

```
Math::PI      #=> 3.141592653589793
Math::cos 0 #=> 1.0
widget = Stockable::Item.new
x = Stockable::inventory
Post < ActiveRecord::Base
BookController < ActionController::Base
```

- Style: use dot to invoke utility functions (ie module methods)

```
Math.cos 0   #=> 1.0
Stockable.inventory
```

Modules are Always Open

- Module contains several related classes
- Style: Each class should be in its own file
- So split module definition

```
# game.rb
module Game
end

# game/card.rb
module Game
  class Card … end
end

# game/player.rb
module Game
  class Player … end
end
```

Modules as "Mixins"

- Another container for method definitions

```
module Stockable
  def order … end
end
```

- A module can be *included* in a class

```
class LightBulb < Device
  include Stockable, Comparable …
end
```

- Module's (instance) methods become (instance) methods of the class

```
bulb = LightBulb.new
bulb.order              # from Stockable
if bulb <= old_bulb # from Comparable
```

Requirements for Mixins

- Mixins often rely on certain aspects of classes into which they are included
- Example: Comparable methods use **#<=>**

```
module Comparable
  def <(other) … end
  def <=(other) … end
end
```

- Enumerable methods use **#each**
- Recall *layering* in SW I/II? Roughly:
  - Class implements kernel methods
  - Module implements secondary methods

Software Engineering

- All the good principles of SW I/II apply
- Single point of control over change
  - Avoid magic numbers
- Client view: abstract state, contracts, invariants
- Implementer view: concrete rep, correspondence, invariants
- Checkstyle tool: rubocop
- Documentation: YARD
  - Notation for types: yardoc.org/types.html

```
@param words Array<String> the lexicon
```

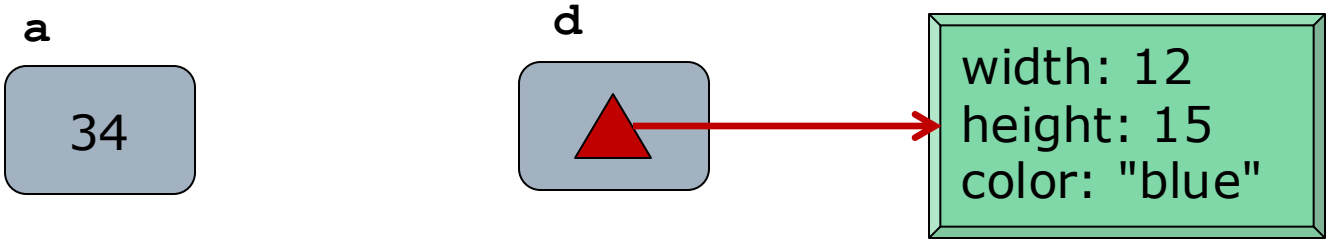
Summary

- Classes as blueprints for objects
  - Contain methods and variables
  - Public vs private visibility of methods
  - Attributes for automatic getters/setters
- Metaprogramming
  - Classes are objects too
  - “Class instance” variables
- Single inheritance
- Modules are namespaces and mixins

Ruby:
Objects and Dynamic Types

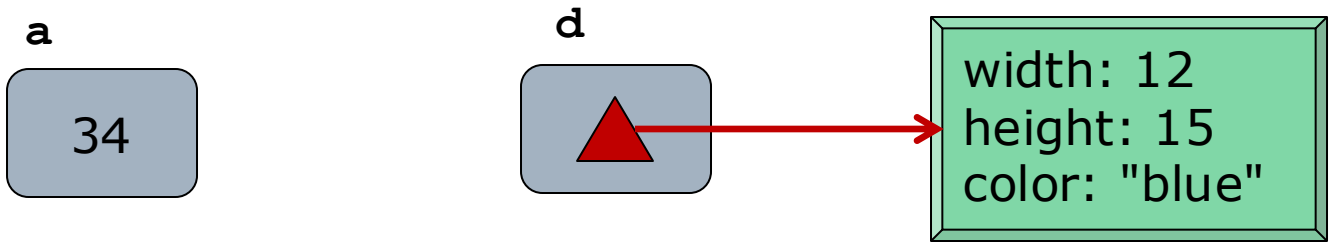
Primitive vs Reference Types

- Recall Java type dichotomy:
  - Primitive: int, float, double, boolean,…
  - Reference: String, Set, NaturalNumber,…
- A variable is a "slot" in memory
  - Primitive: the slot holds the *value* itself
  - Reference: the slot holds a *pointer* to the value (an object)



Object Value vs Reference Value

- Variable of reference type has *both*:
  - Reference value: value of the slot itself
  - Object value: value of object it points to (corresponding to its mathematical value)
- Variable of primitive type has *just one*
  - Value of the slot itself, corresponding to its mathematical value



Two Kinds of Equality

- Question: "Is x equal to y?"
  - A question about the *mathematical* value of the variables x and y
- In Java, depending on the type of x and y we either need to:
  - Compare the values of the *slots*

```
x == y // for primitive types
```

  - Compare the values of the *objects*

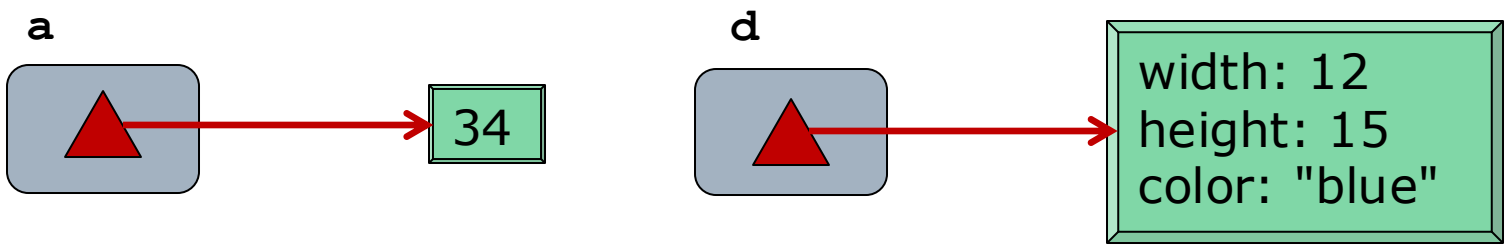
```
x.equals(y) // for non-primitive types
```

Ruby: "Everything is an Object"

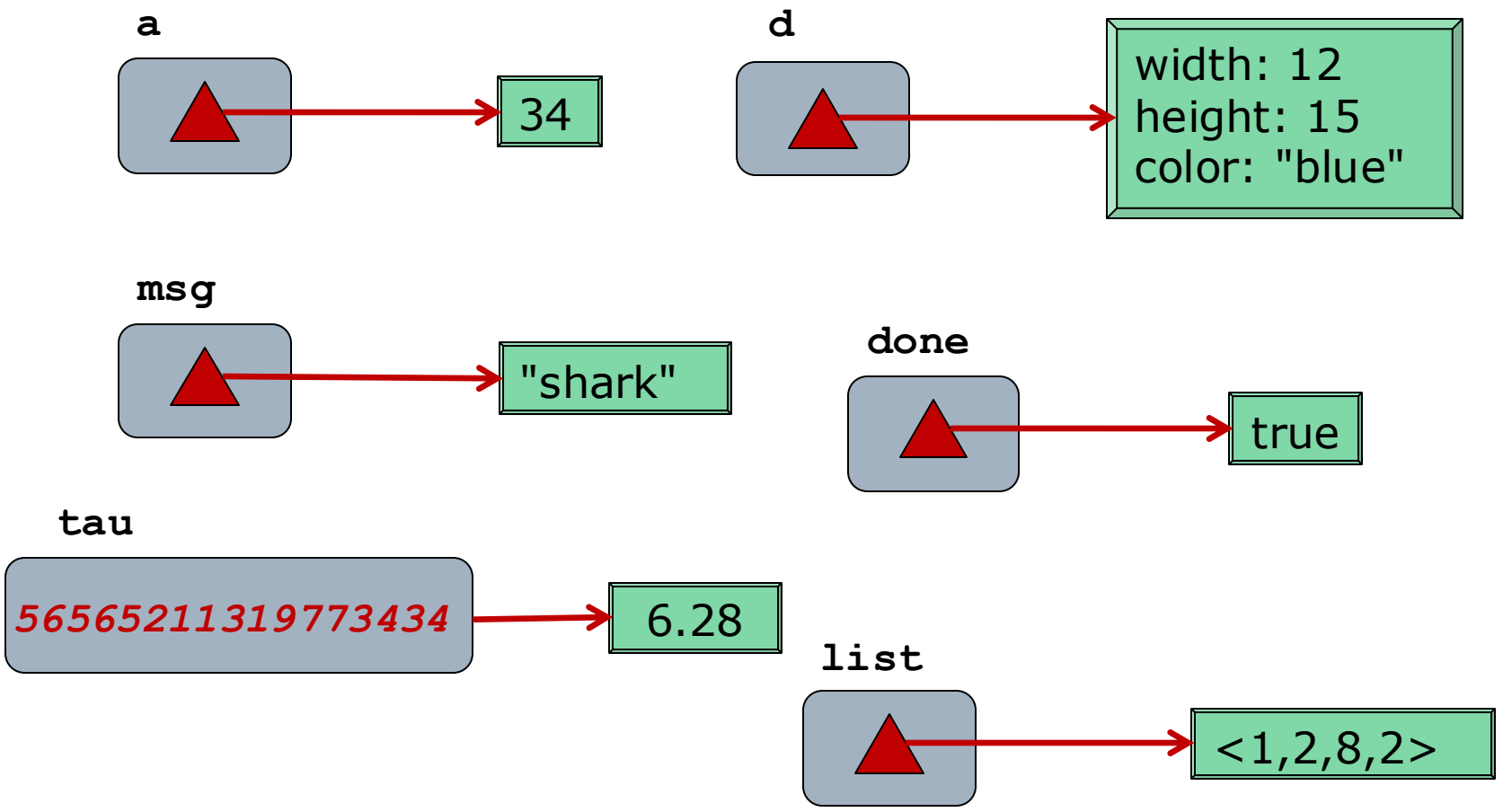
- In Ruby, *every* variable maps to an object
  - Integers, floats, strings, sets, arrays, …
- Benefit: A more consistent mental model
  - References are *everywhere*
  - Every variable has *both* a reference value and an object value
  - Comparison of mathematical values is *always* comparison of object value
- Ruby terminology: Reference value is called the *object id*
  - The 8-byte number stored in the slot
  - Unique identifier for corresponding object

```
tau = 6.28
tau.object_id #=> 56565211319773434
```

Everything is an Object



Everything is an Object



Operational Detail: Immediates

- For small integers, the mathematical value is *encoded in the reference value!*
  - LSB of reference value is 1
  - Remaining bits encode value, 2's complement

```
x = 0
x.object_id  #=> 1 (0b00000001)
y = 6
y.object_id  #=> 13 (0b00001101)
```

- Known as an "immediate" value
  - Others: true, false, nil, symbols, string literals
- Benefit: Performance
  - No change to model, *everything is an object*

Objects Have Methods

- Familiar "." operator to invoke (instance) methods

```
list = [6, 15, 3, -2]
list.size #=> 4
```

- Since numbers are objects, they have methods too!

```
3.to_s   #=> "3"
3.odd?   #=> true
3.lcm 5  #=> 15
1533.digits #=> [3, 3, 5, 1]
3.+ 5 #=> 8
3.class #=> Integer
3.methods #=> [:to_s, :inspect, :+, …]
```

Pitfall: Equality Operator

- Reference value is still useful sometimes
  - "Do these variables refer to the same object?"
- So we still need 2 methods:

```
x == y
x.equal? y
```

- Ruby semantics are the *opposite* of Java!
  - == is *object value* equality
  - .equal? is *reference value* equality
- Example

```
a1, a2 = [1, 2], [1, 2] # "same" array
a1 == a2 #=> true (obj values equal)
a1.equal? a2 #=> false (ref vals differ)
```

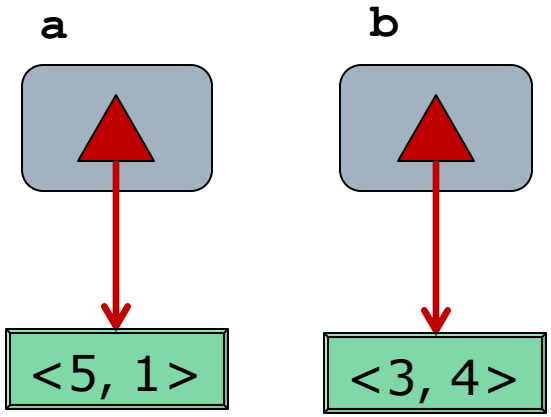
To Ponder

Evaluate (each is true or false):

```
3 == 3

3.equal? 3

[3] == [3]

[3].equal? [3]
```

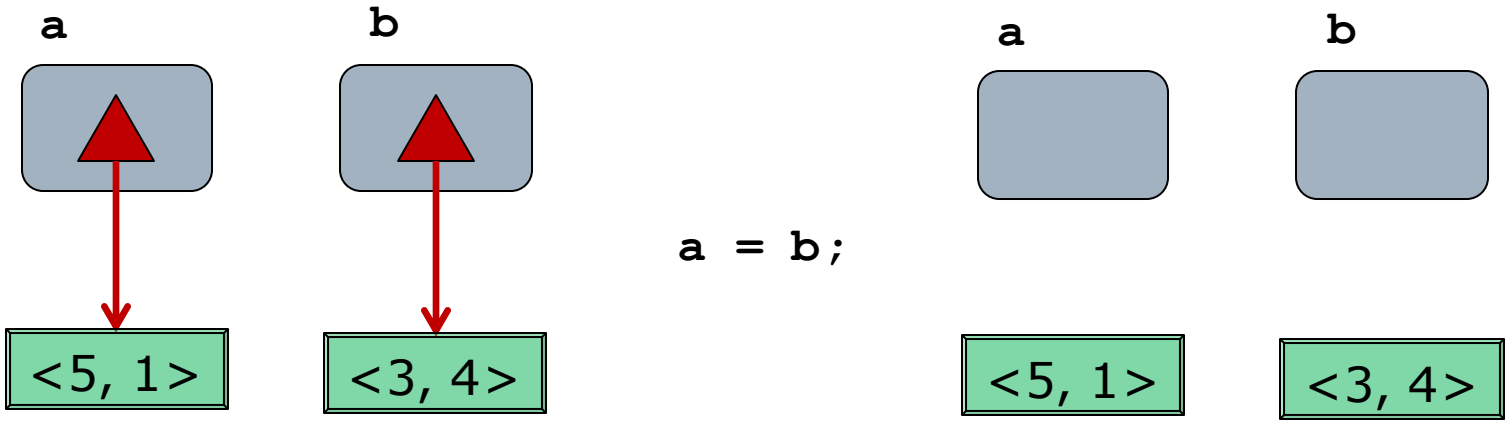
Assignment (Just Like Java)

- Assignment copies the *reference value*
- Result: Both variables point to the *same* object (ie an "alias")
- Parameter passing works this way too



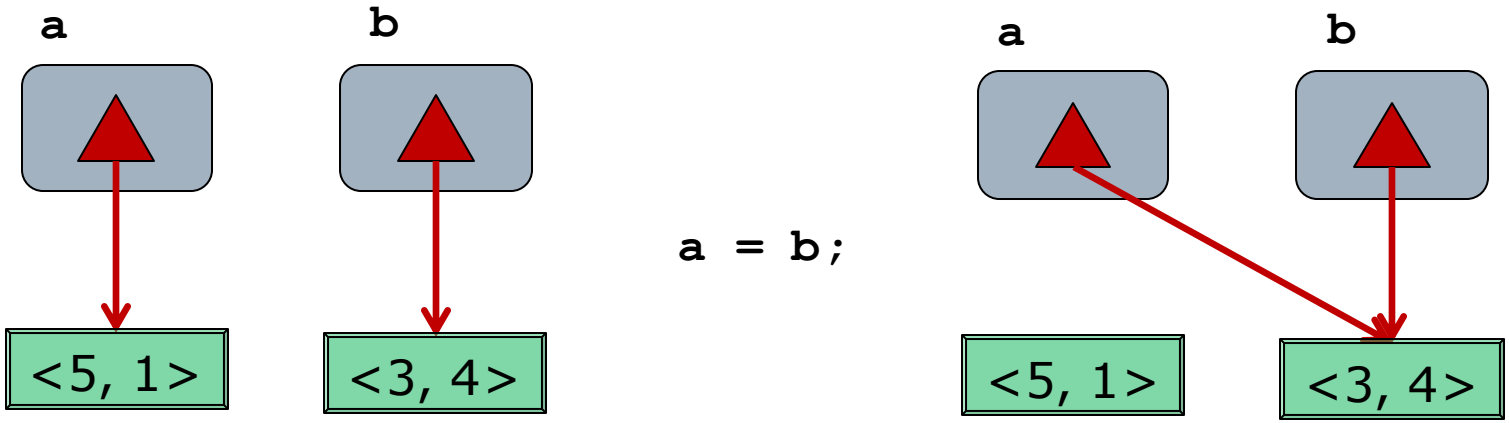
Assignment (Just Like Java)

- Assignment copies the *reference value*
- Result: Both variables point to the *same* object (ie an "alias")
- Parameter passing works this way too



Assignment (Just Like Java)

- Assignment copies the *reference value*
- Result: Both variables point to the *same* object (ie an "alias")
- Parameter passing works this way too



Aliasing Mutable Objects

- When aliases exist, a statement can change a variable's object value without mentioning that variable

```
x = [3, 4]
y = x       # x and y are aliases
y[0] = 13   # changes x as well!
```

- Question: What about numbers?

```
i = 34
j = i       # i and j are aliases
j = j + 1   # does this increment i too?
```

Immutability

- Recall in Java strings are *immutable*
  - No method changes the value of a string
  - A method like concat returns a new instance
- Benefit: Aliasing immutable objects is safe
- Immutability is used in Ruby too
  - Numbers, true, false, nil, symbols

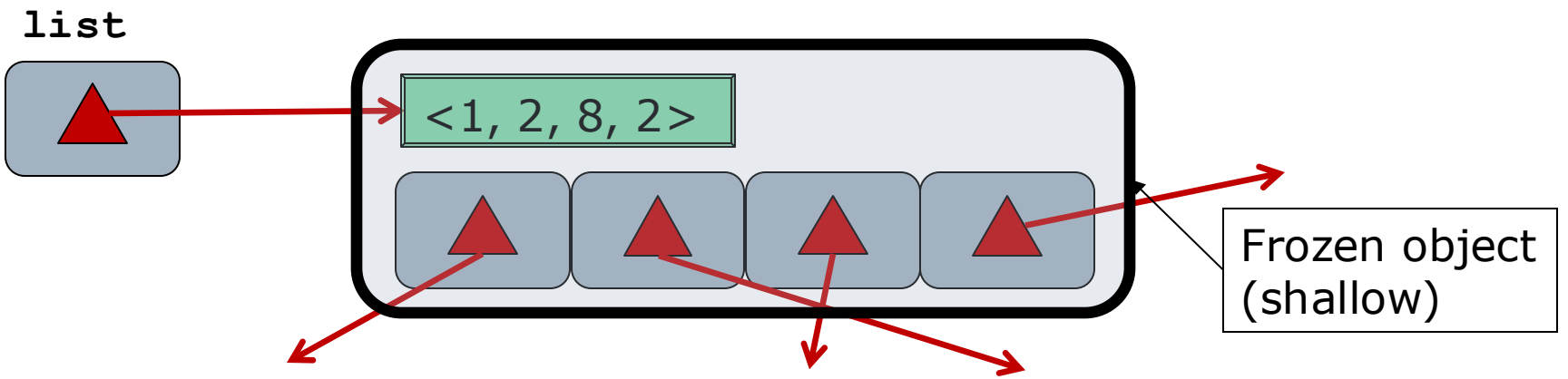
```
list = [3, 4]
list[0] = 13 # changes list's object value
             # list points to same object
n = 34
n = n + 1    # changes n's reference value
             # n points to different object
```

  - Pitfall: Unlike Java, strings in Ruby are *mutable*
  - But objects (including strings) can be "frozen"

Freezing

- Makes a (single) object immutable
  - The *object value* can not change

```
list = [1, 2, 8, 2].freeze
list.length #=> 4
list[0] = 3 # error: can't modify a
            #        frozen object
list = [7, -1] # ok: ref value changed
```



Assignment Operators

- Parallel assignment

```
x, y, z = y, 10, radius
```

- Arithmetic contraction
  - `+= -= *= /= %= **=`
  - Pitfall: no `++` or `--` operators (use `+= 1`)
- Logical contraction
  - `||= &&=`
  - Idiom: `||=` for initializing potentially nil variables
  - Pitfall (minor):
    - `x ||= y` not quite equivalent to `x = x || y`
    - Better to think of it as `x || x = y`
    - Usually amounts to the same thing

Declared vs Dynamic Types

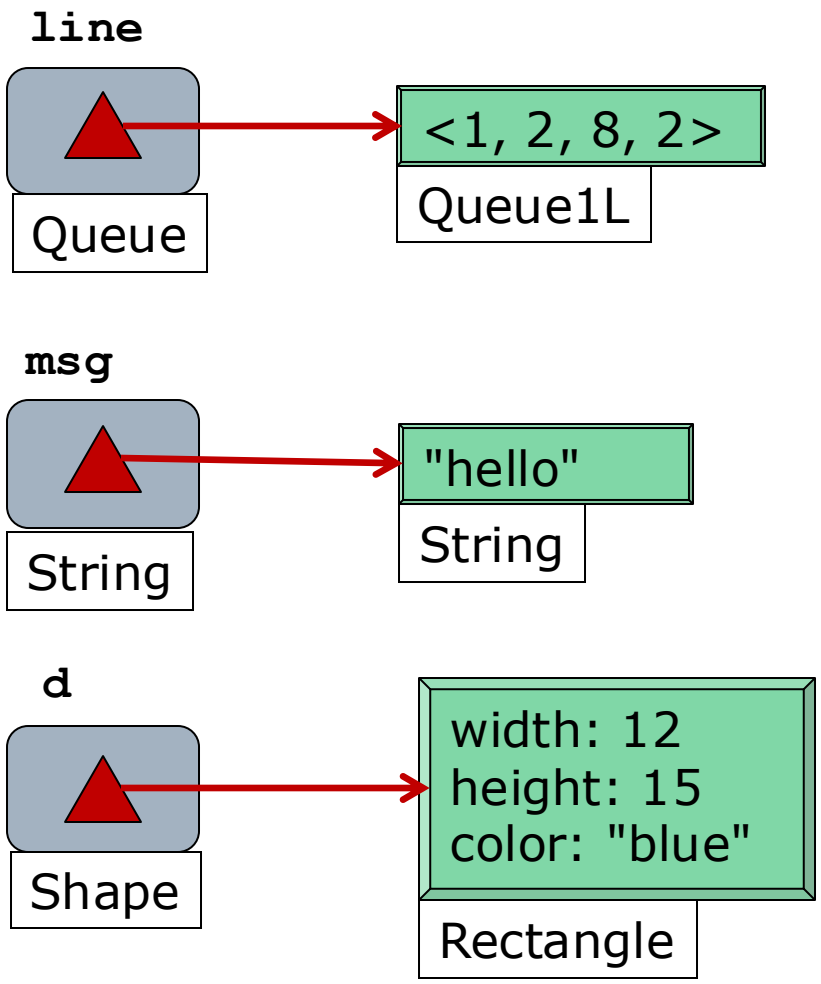
- In Java, types are associated with *both*
  - Variables (declared / static type), and
  - Objects (dynamic / run-time type)

```
Queue line = new Queue1L();
```

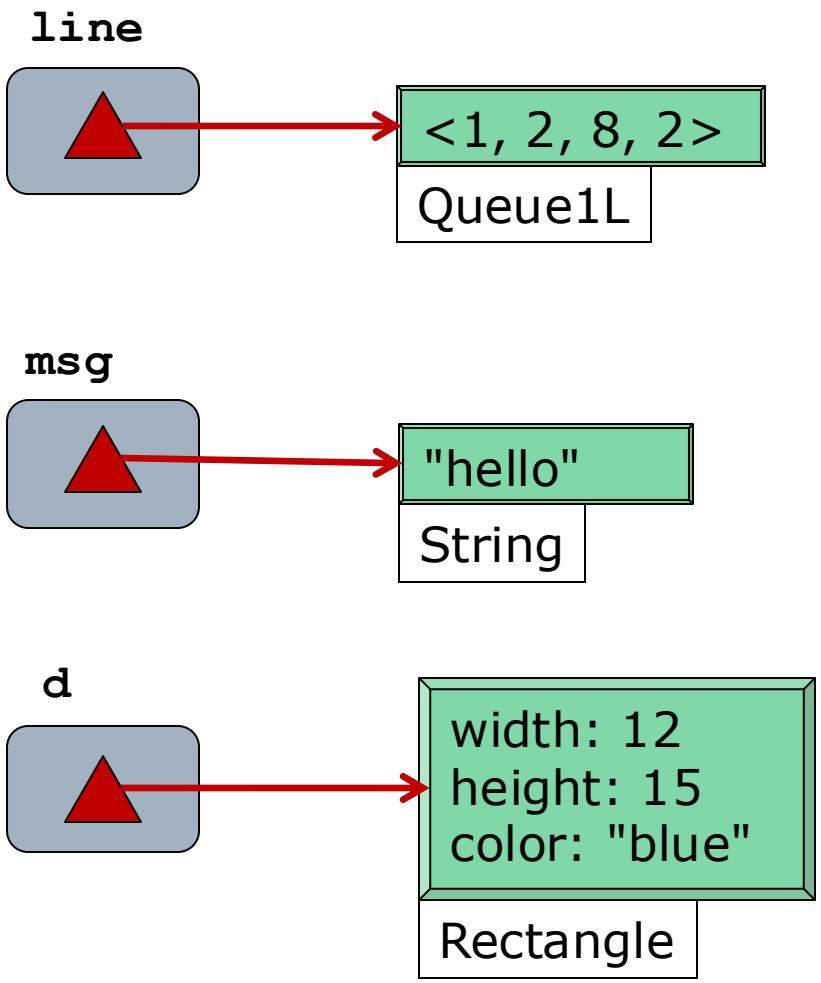
- Recall: Programming to the interface
- Compiler uses *declared* type for checks

```
line.inc(); // error no such method
line = new Set1L(); // err. wrong type

boolean isEmpty (Set s) {…}
if isEmpty(line) … // error arg type
```

Statically Typed Language



Dynamically Typed Language



Dynamically Typed Language

- Equivalent definitions:
  - No static types
  - Dynamic types only
  - Variables do not have type, objects do

Function Signatures

- Statically typed

```
String parse(char[] s, int i) {… return e;}
out = parse(t, x);
```

  - Declare parameter and return types
    - See **s**, **i**, and **parse**
  - The *compiler* checks conformance of
    - (Declared) types of arguments (**t**, **x**)
    - (Declared) type of return expression (**e**)
    - (Declared) type of expression *using* parse (**out**)
- Dynamically typed

```
def parse(s, i) … e end
out = parse t, x
```

  - You are on your own!

Type Can Change at Run-time

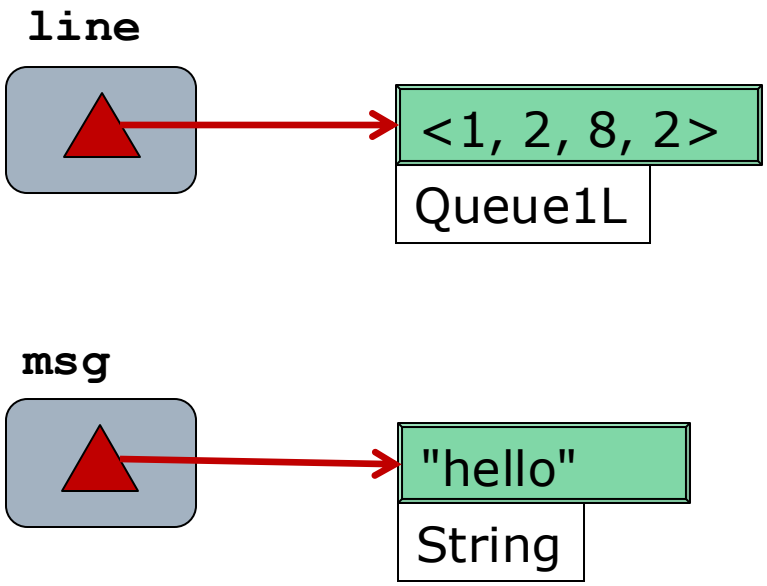
Statically Typed

```
  //a is undeclared
String a;
  //a is null string
a = "hi;
  //compile-time err
a = "hi";
a = 3;
  //compile-time err
a.push();
  //compile-time err
```

Dynamically Typed

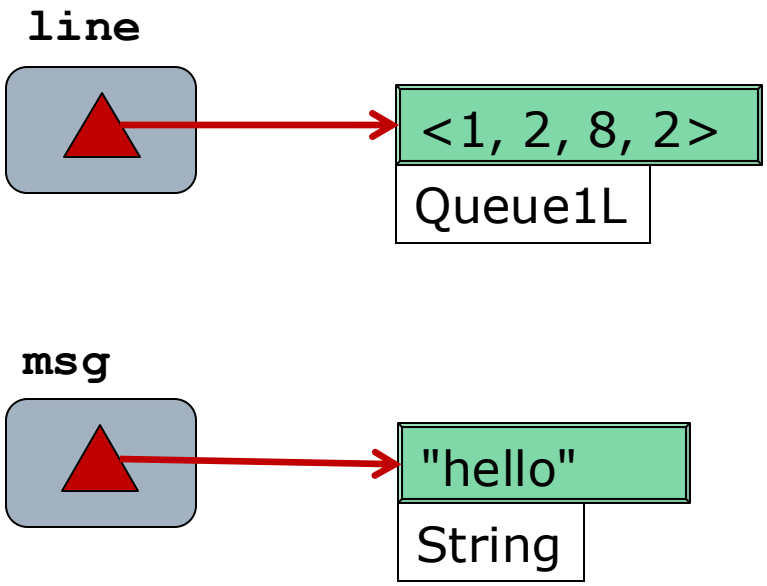
```
  # a is undefined
a = a
  # a is nil
a = "hi
  # load-time error
a = "hi"
a = 3
  # a is now a number
a.push
  # run-time error
```

Changing Dynamic Type



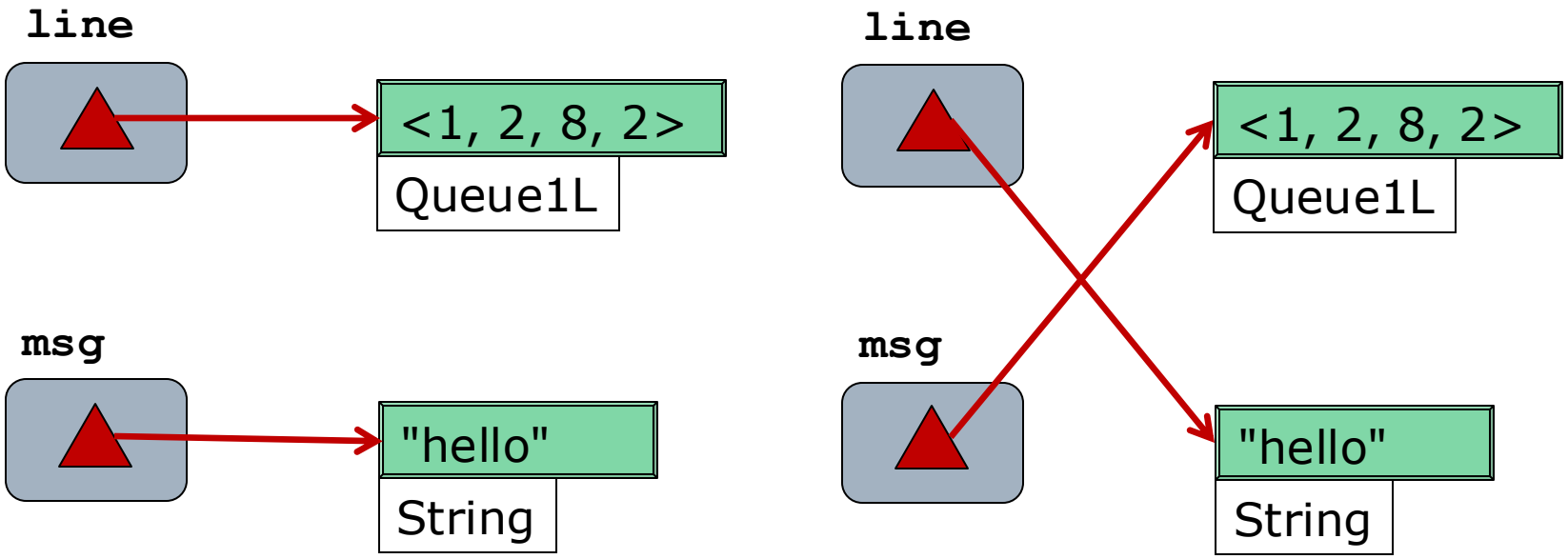
Changing Dynamic Type


```
msg, line = line, msg
```

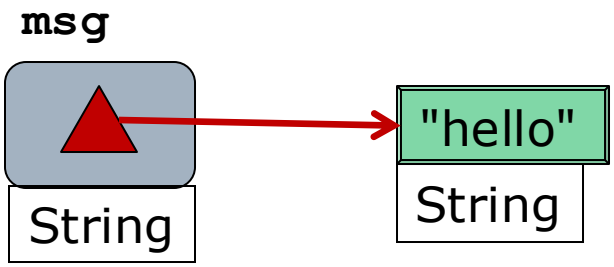


Changing Dynamic Type


```
msg, line = line, msg
```

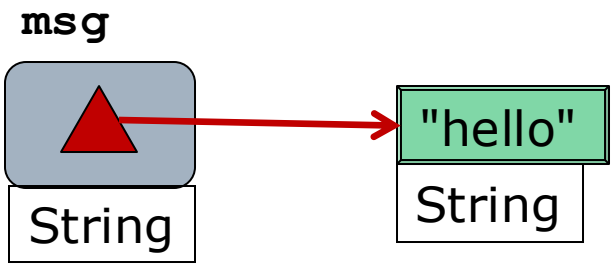


Arrays: Static Typing



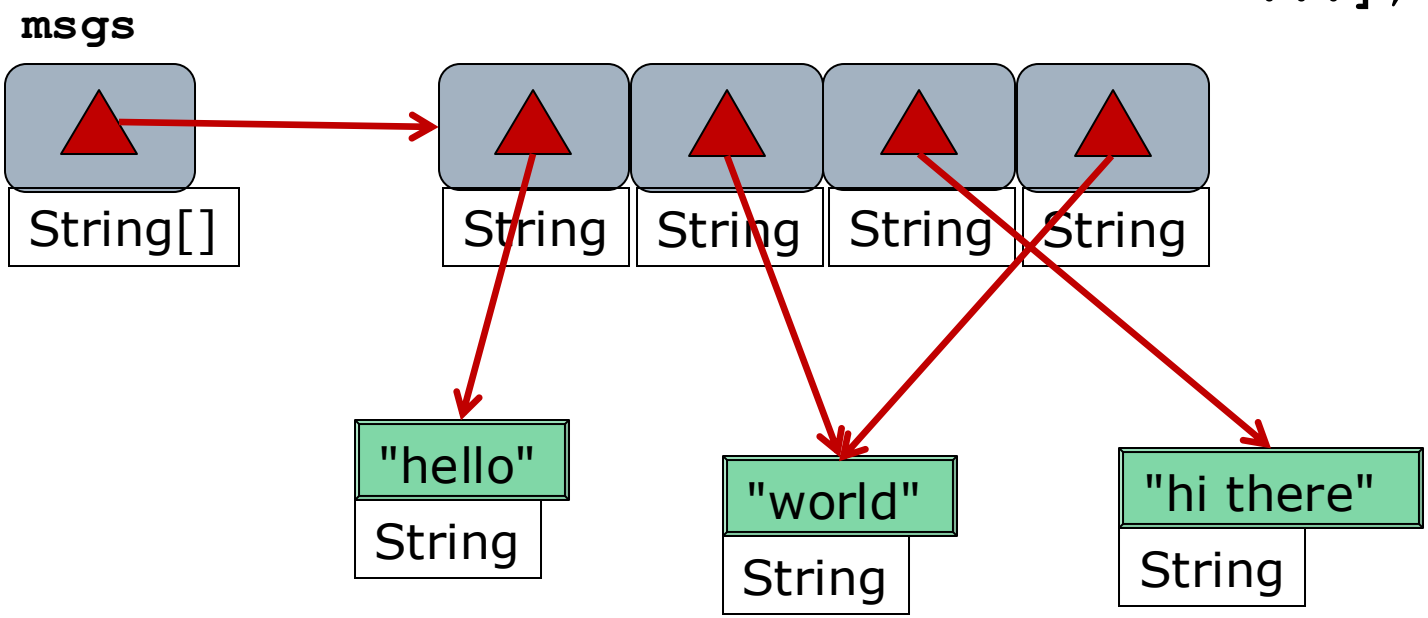

```
String msg = "hello";
```

Arrays: Static Typing

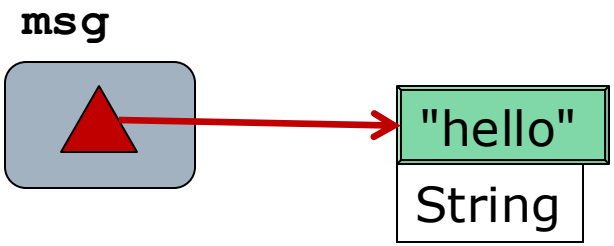



```
String msg = "hello";
```

```
String[] msgs = ["hello",
                 "world",
                 ...];
```

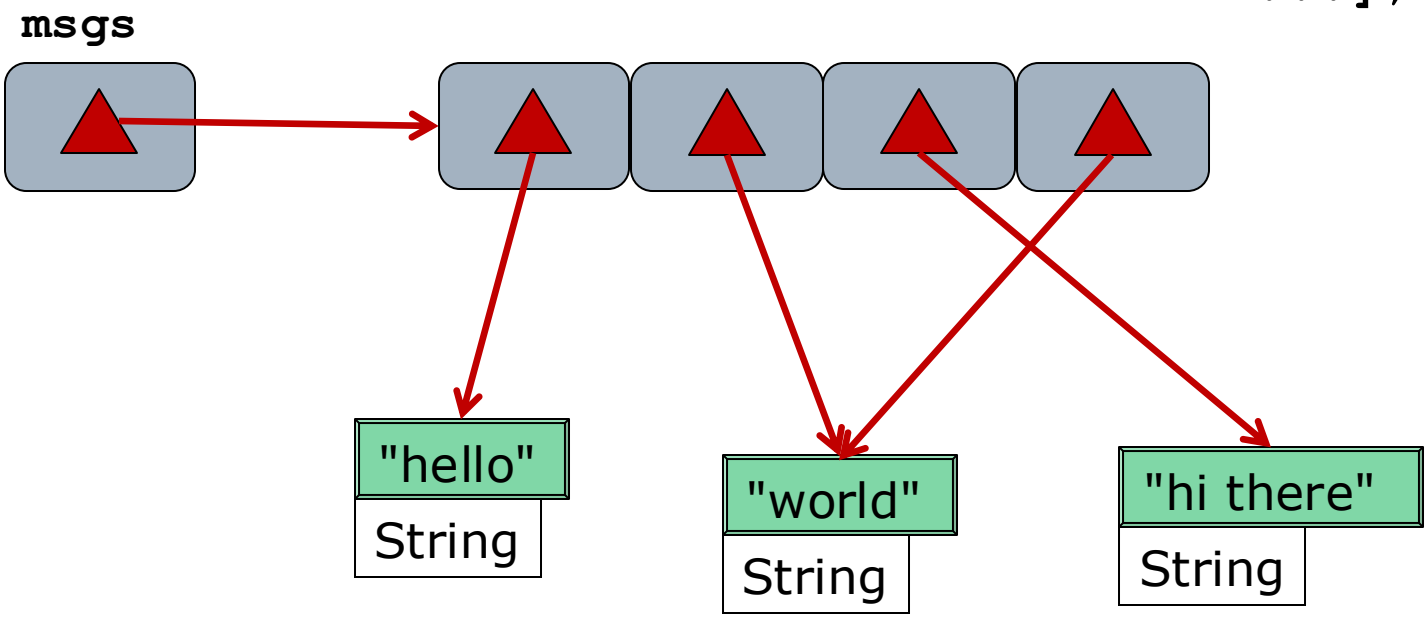


Arrays: Dynamic Typing



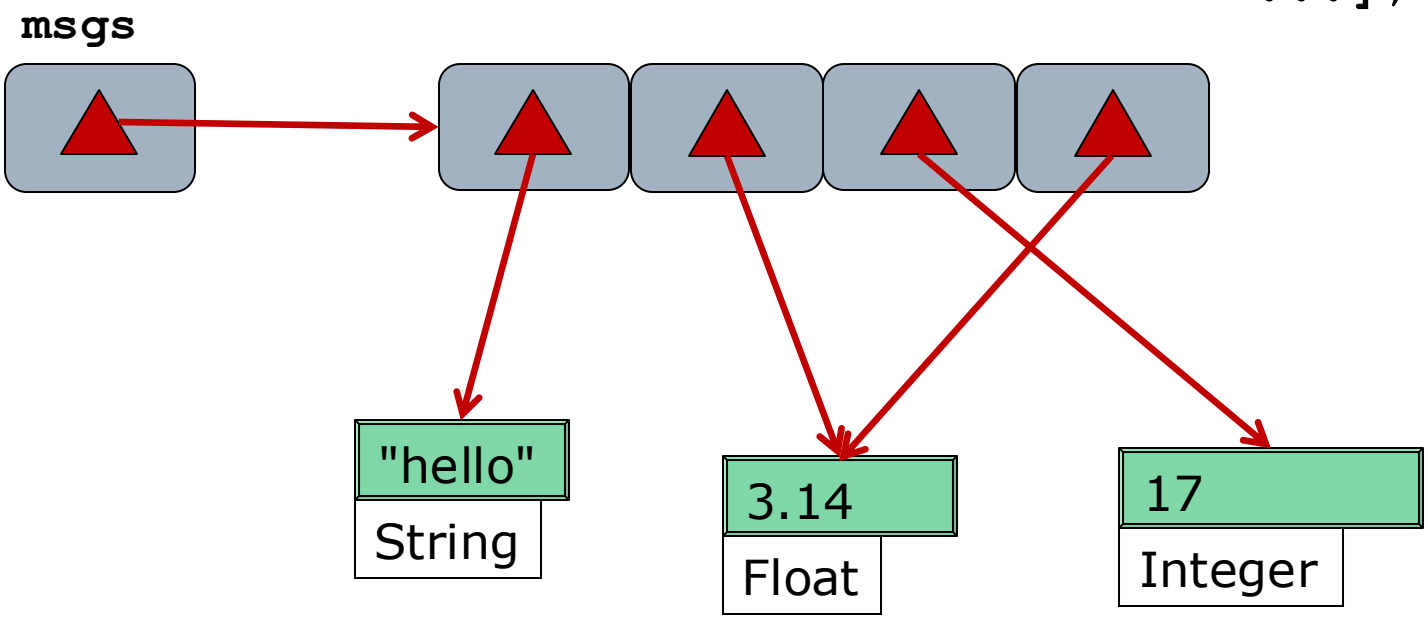

```
msg = "hello";
```

```
msgs = ["hello",
        "world",
        ...];
```



Consequence: Heterogeneity


```
msgs = ["hello",
        3.14,
        ...];
```



Tradeoffs

Statically Typed

- Earlier error detection
- Clearer APIs
- More compiler optimizations
- Richer IDE support

Dynamically Typed

- Less code to write
- Less code to change
- Quicker prototyping
- No casting needed

Strongly Typed

- Just because variables don't have types, doesn't mean you can do anything you want

```
>> 'hi'.upcase
=> "HI"
>> 'hi'.odd?
NoMethodError: undefined method `odd?' for String
>> puts 'The value of x is ' + x
TypeError: can't convert Integer to String
```

Summary

- Object-oriented
  - References are everywhere
  - Assignment copies reference value (alias)
  - Primitives (immediates) are objects too
  - == vs .equal? are flipped
- Dynamically type
  - Objects have types, variables do not
- Strongly Typed
  - Incompatible types produce (run time) error