Review: Information revealed to self

Object knows its own representation ("instance variables"), invariants, private methods:

```
(class Fraction Number
 (num den) ;; representation (concrete!)
           ;; invariants: lowest terms, den > 0

 (method asFraction ()
   self)
 (method print ()
   (print num) (print #/) (print den))
 (method reciprocal ()
   (signReduce (setNum:den: (new Fraction) den num)))
```

Information revealed to self: your turn

How would you implement `coerce:`?
(Value of argument, representation of receiver)

```
...
(method asFraction ()
  self)
(method print ()
  (print num) (print #/) (print den))
(method reciprocal ()
  (signReduce (setNum:den: (new Fraction) den num)))
(method coerce: (aNumber)
  ...)
```

Information revealed to self: your turn

How would you implement `coerce:`?
(Value of argument, representation of receiver)

```
...
(method asFraction ()
  self)
(method print ()
  (print num) (print #/) (print den))
(method reciprocal ()
  (signReduce (setNum:den: (new Fraction) den num)))
(method coerce: (aNumber)
  asFraction aNumber)
```

Exposing information, part II

Alas! Cannot see representation of argument

How will you know “equal, less or greater”?

Exposing information, part II

Alas! Cannot see representation of argument

Protocol says "like with like"? Use private methods

```
(method num () num) ; private
(method den () den) ; private

(method = (f)  ;; relies on invariant!
    (and: (= num (num f)) {(= den (den f))}))
(method < (f)
    (< (* num (den f)) (* (num f) den)))
```

Remember behavioral subtyping

Private methods: Your turn

How will you multiply two fractions?

Private methods: Your turn

How will you multiply two fractions?

```
(method * (f)
    (divReduce
        (setNum:den: (new Fraction)
                     (* num (num f))
                     (* den (den f)))))
```

An open system

`Number` protocol: like multiplies with like

What about large and small integers?

- How to multiply two small integers?
- How to multiply two large integers?

How is algorithm known?

Each object knows its own algorithm:

- Small: Use machine-primitive multiplication
- Large: Multiply magnitudes; choose sign

Review: Two kinds of knowledge

I can send message to you:

- I know your protocol

I can inherit from you:

- I know my subclass responsibilities

Knowledge of protocol

Three levels of knowledge:

1. **I know only your public methods**
   Example: `select:`
2. **You are like me: share private methods**
   Example: `*` **and** `+` **on** `Fraction`
3. **I must get to know you: double dispatch**
   Example: `*` **and** `+` **on mix of integers**

Double dispatch: extending open systems

I claim:

- Large integers and small integers both `Integer`
- Messages `=`, `<`, `+`, `*` ought to mix freely
- Large and small integers have different private protocol

Private for large integers: `magnitude`

Private for small integers: `mul:withOverflow`

Double dispatch code operation & protocol

Example messages:

- **I answer the small-integer protocol, add me to yourself**
- **I answer the large-positive integer protocol, multiply me by yourself**

Message encodes

- **Operation to be performed**
- **Protocol accepted by argument**

Your turn: responding to double dispatch

How do you act?

1. **As small integer, you receive "add small integer n to `self`"**
2. **As small integer, you receive "multiply large positive integer N by `self`"**
3. **As large positive integer, you receive "add small integer n to `self`"**
4. **As large positive integer, you receive "multiply large positive integer N by `self`"**

Your turn: using double dispatch

On what class does each method go?

A.
```
(method + (aNumber)
  (addSmallIntegerTo: aNumber self))
```

B.
```
(method * (anInteger)
  (multiplyByLargePositiveInteger: anInteger self))
```

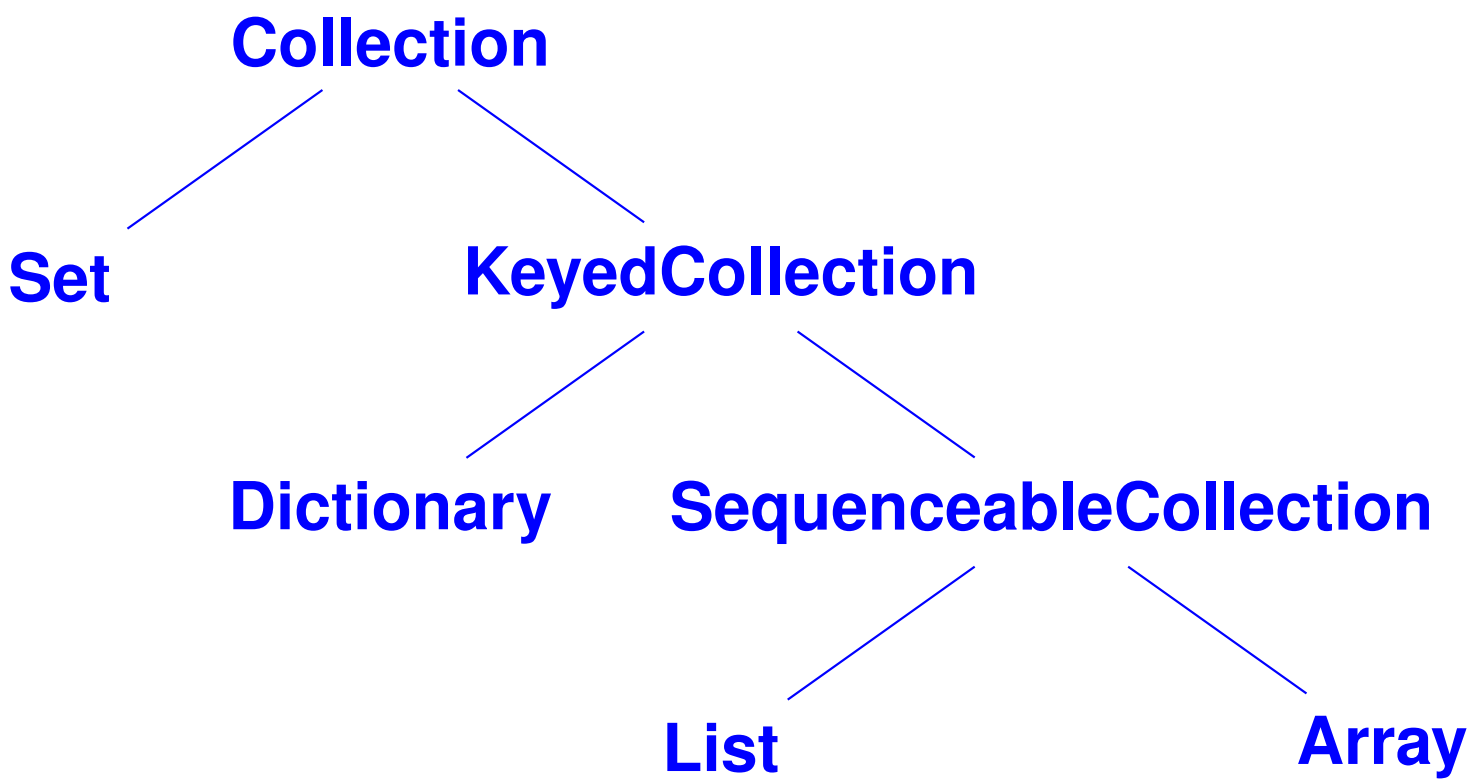
(See the "double dispatch": + then `addSmallIntegerTo:`)

Information-hiding summary

Three levels

1. I use your public protocol
2. We are alike; I add our private protocol
3. Your protocol is revealed by double dispatch

“Collection hierarchy”



Collection mutators

`add: newObject` Add argument

`addAll: aCollection` Add every element of arg

`remove: oldObject` Remove arg, error if absent

`remove:ifAbsent: oldObject exnBlock`

Remove the argument, evaluate `exnBlock` if absent

`removeAll: aCollection` Remove every element of arg

Collection observers

`isEmpty` Is it empty?

`size` How many elements?

`includes: anObject` Does receiver contain arg?

`occurrencesOf: anObject` How many times?

`detect: aBlock` Find and answer element satisfying `aBlock` (cf μScheme `exists?`)

`detect:ifNone: aBlock exnBlock` Detect, recover if none

`asSet` Set of receiver's elements

Collection iterators

`do: aBlock` For each element `x`, evaluate `(value aBlock x)`.

`inject:into: thisValue binaryBlock`

Essentially μScheme `foldl`

`select: aBlock` Essentially μScheme `filter`

`reject: aBlock` Filter for *not* satisfying `aBlock`

`collect: aBlock` Essentially μScheme `map`

Implementing collections

```
(class Collection Object
  () ; abstract
  (method do: (aBlock)
     (subclassResponsibility self))
  (method add: (newObject)
     (subclassResponsibility self))
  (method remove:ifAbsent (oldObj exnBlock)
     (subclassResponsibility self))
  (method species ()
     (subclassResponsibility self))
  ⟨other methods of class Collection⟩
)
```

Reusable methods

```
⟨other methods of class Collection⟩=
(method addAll: (aCollection)
  (do: aCollection [block(x) (add: self x)])
  aCollection)
(method size () [locals temp]
 (set temp 0)
 (do: self [block(_) (set temp (+ temp 1))])
 temp)
```

These methods always work

Subclasses can override (redefine) with more efficient versions

`species` method

Create "collection like the reciever"

Example: filtering

⟨*other methods of class* Collection⟩=

```
(method select: (aBlock) [locals temp]
  (set temp (new (species self)))
  (do: self [block (x)
     (ifTrue: (value aBlock x)
        {(add: temp x)})])
  temp)
```

Subtyping mathematically

Always transitive

$$\frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

Key rule is subsumption:

$$\frac{e : \tau \qquad \tau <: \tau'}{e : \tau'}$$

(*implicit* subsumption: no cast)

Subtyping is not inheritance

Subtype understands *more* messages:

$$\frac{}{\{m_1 : \tau_1, \ldots, m_n : \tau_n, \ldots, m_{n+k} : \tau_{n+k}\} <: \{m_1 : \tau_1, \ldots, m_n : \tau_n\}}$$

If an object understands messages $m_1, \ldots, m_n$, and possibly more besides, you can use it where $m_1, \ldots, m_n$ are expected

- **Methods must behave as expected**

Behavioral subtyping (in Ruby, "duck typing")

The four crucial `Collection` methods

```
(class Collection Object
  () ; abstract
  (method do:  (aBlock)
     (subclassResponsibility self))
  (method add: (newObject)
     (subclassResponsibility self))
  (method remove:ifAbsent (oldObj exnBlock)
     (subclassResponsibility self))
  (method species ()
     (subclassResponsibility self))
  ⟨other methods of class Collection⟩
)
```

```
(class Set Collection
  (members) ; list of elements
  (class-method new () (initSet (new super)))
  (method initSet () ; private method
     (set members (new List))
     self)
  (method do: (aBlock) (do: members aBlock))
  (method remove:ifAbsent: (item exnBlock)
    (remove:ifAbsent: members item exnBlock))
  (method add: (item)
    (ifFalse: (includes: members item)
      {(add: members item)})
    item)
  (method species () Set)
  (method asSet    () self) ; extra efficient
)
```

```
(class Set Collection
  (members) ; list of elements
  (class-method new () (initSet (new super)))
  (method initSet () ; private method
     (set members (new List))
     self)
  (method do: (aBlock) (do: members aBlock))
  (method remove:ifAbsent: (item exnBlock)
    (remove:ifAbsent: members item exnBlock))
  (method add: (item)
    (ifFalse: (includes: members item)
      {(add: members item)})
    item)
  (method species () Set)
  (method asSet    () self) ; extra efficient
)
```