Protocol for Booleans

`ifTrue:ifFalse: trueBlock falseBlock`	Full conditional
`ifTrue: trueBlock`	Part conditional (for side effect)
`ifFalse: falseBlock`	Part conditional (for side effect)
`& aBoolean`	Conjunction
`| aBoolean`	Disjunction
`not`	Negation
`eqv: aBoolean`	Equality
`xor: aBoolean`	Difference
`and: altBlock`	Short-circuit conjunction
`or: altBlock`	Short-circuit disjunction

Classes True and False

```
(class True Boolean
   []
   (method ifTrue:ifFalse: (trueBlock falseBlock)
        (value trueBlock))
)
(class False Boolean
   []
   (method ifTrue:ifFalse: (trueBlock falseBlock)
        (value falseBlock))
)
```

What happens if `ifTrue:` is sent to `true`?

`ifTrue:` message dispatched to class `Boolean`

```
(class Boolean Object
   []
   (method ifTrue:ifFalse: (trueBlock falseBlock)
       (subclassResponsibility self))
   (method ifTrue: (trueBlock)
      (ifTrue:ifFalse: self trueBlock {}))
   ...
)
```

Message sent to `self` starts over
(with class of receiver)

Dispatching to `True`

```
(class True Boolean
   []
   (method ifTrue:ifFalse: (trueBlock falseBlock)
         (value trueBlock))
   ; all other methods are inherited
)
```

Your turn: `not`

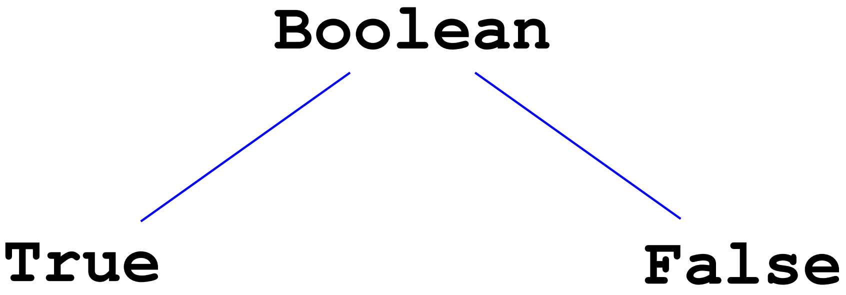
What should `not` look like?

- Implemented on what class?
- With what method definition?

Implementing `not`

```
(class Boolean Object
  []
  (method ifTrue:ifFalse: (trueBlock falseBlock)
     (subclassResponsibility self))
  (method ifTrue: (trueBlock)
    (ifTrue:ifFalse: self trueBlock {}))
  (method not ()
    (ifTrue:ifFalse: self {false} {true}))
  ...
)
```

Inheritance for Booleans



`Boolean`

`True` `False`

`Boolean` is abstract class

- Instances of `True` and `False` only

Method `ifTrue:ifFalse:` defined on `True` and `False`

All others defined on `Boolean`

Each class has one of two roles

Abstract class

- Meant to be inherited from
- Some (> 0) subclassResponsibility methods
- Examples: Boolean, Shape, Collection

Regular (“concrete”) class

- Meant to be instantiated
- No subclassResponsibility methods
- Examples: True, Triangle, List

Syntax comparison: Impcore to Smalltalk

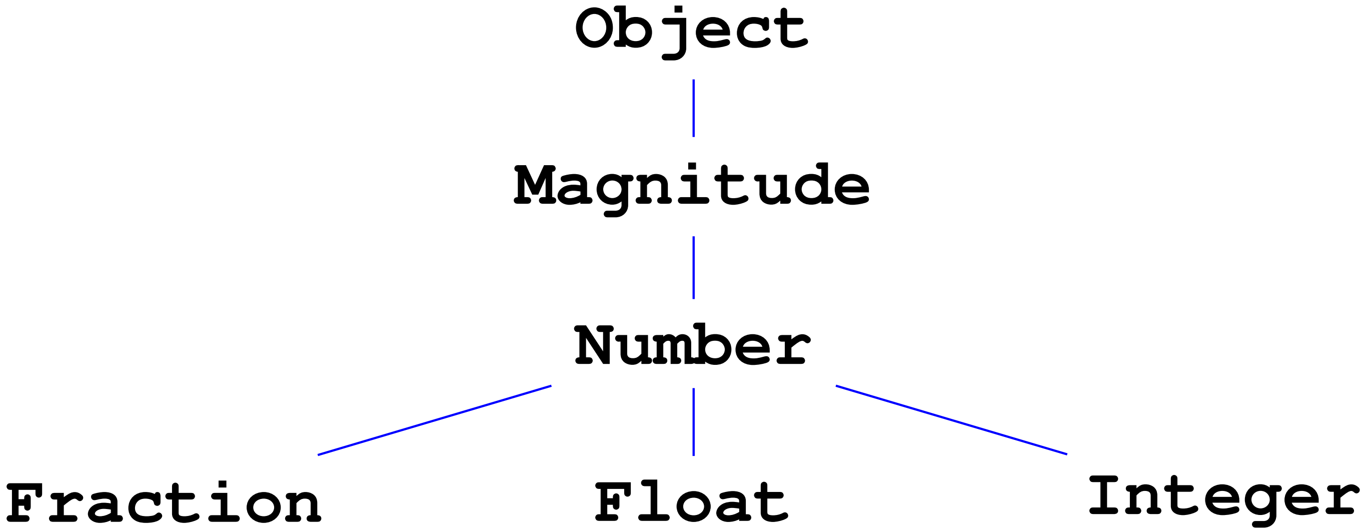
```
Exp = LITERAL of rep
    | VAR     of name
    | SET     of name * exp
    | IF      of exp * exp * exp
    | WHILE   of exp * exp
    | BEGIN   of exp list
    | APPLY   of name * exp list
    | SEND    of name * exp * exp list
    | BLOCK   of name list * exp list
```

~~| IF of exp * exp * exp~~

~~| WHILE of exp * exp~~

~~| APPLY of name * exp list~~

“Number hierarchy”



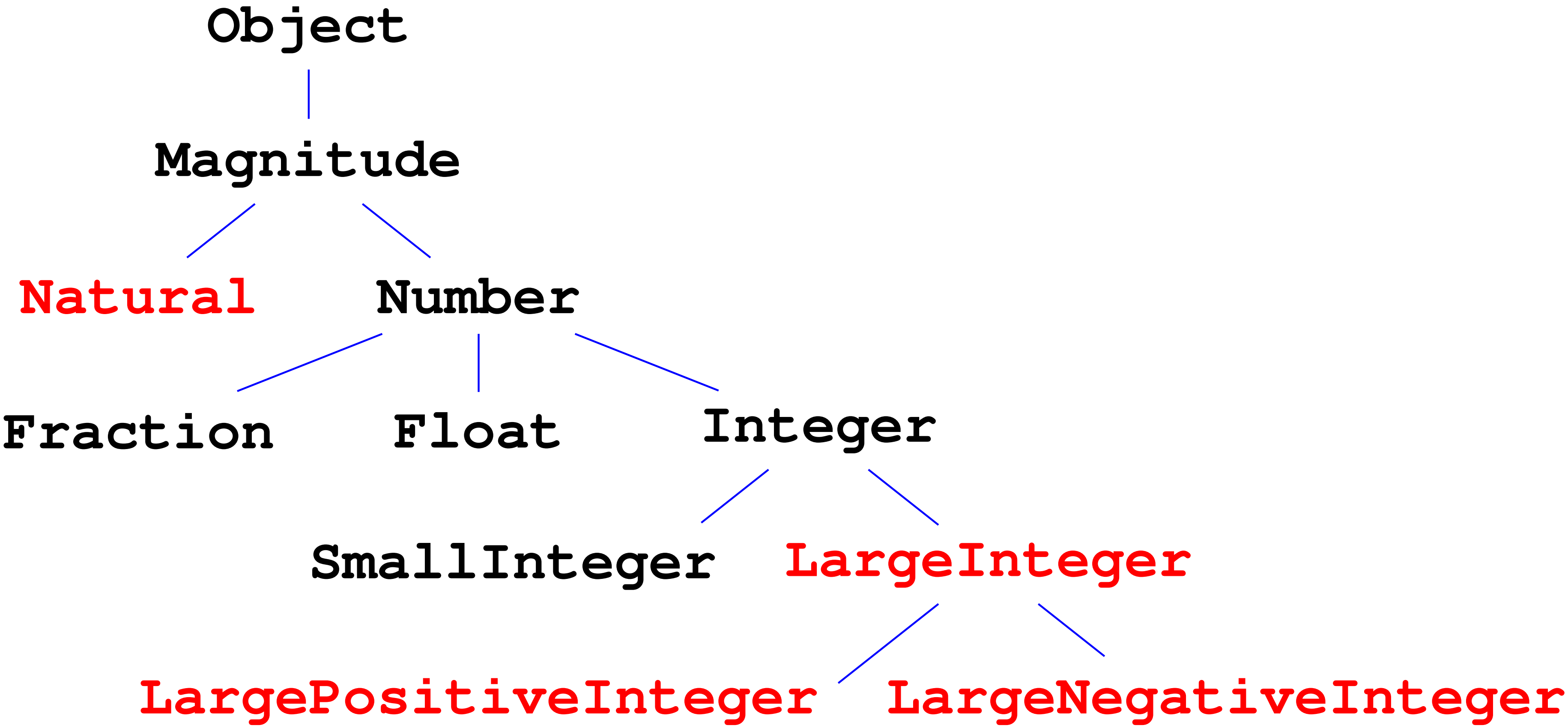
`Object`

`Magnitude`

`Number`

`Fraction` `Float` `Integer`

“Extended Number hierarchy”



Object

Magnitude

Natural

Number

Fraction

Float

Integer

SmallInteger

LargeInteger

LargePositiveInteger

LargeNegativeInteger

Instance protocol for `Magnitude`

`= aMagnitude`	equality (like Magnitudes)
`< aMagnitude`	comparison (ditto)
`> aMagnitude`	comparison (ditto)
`<= aMagnitude`	comparison (ditto)
`>= aMagnitude`	comparison (ditto)
`min: aMagnitude`	minimum (ditto)
`max: aMagnitude`	maximum (ditto)

Subclasses: `Date`, `Natural`

- Compare `Date` with `Date`, `Natural` w/`Natural`, `...`

Your turn: object-oriented design

`= aMagnitude`	equality
`< aMagnitude`	comparison
`> aMagnitude`	comparison
`<= aMagnitude`	comparison
`>= aMagnitude`	comparison
`min: aMagnitude`	minimum
`max: aMagnitude`	maximum

Questions:

- Which methods “subclass responsibility”?
- Which methods on `Magnitude`?

Implementation of `Magnitude`

```
(class Magnitude Object
  [] ; abstract class
  (method =  (x) (subclassResponsibility self))
                      ; may not inherit = from Object
  (method <  (x) (subclassResponsibility self))
  (method >  (y) (< y self))
  (method <= (x) (not (> self x)))
  (method >= (x) (not (< self x)))
  (method min: (aMagnitude)
     (if (< self aMagnitude) {self} {aMagnitude}))
  (method max: (aMagnitude)
     (if (> self aMagnitude) {self} {aMagnitude}))
)
```

Instance protocol for `Number`

`negated`	
`reciprocal`	
`abs`	absolute value
`+ aNumber`	addition
`- aNumber`	subtraction
`* aNumber`	multiplication
`/ aNumber`	division (converted!)
`negative`	sign check
`nonnegative`	sign check
`strictlyPositive`	sign check

More instance protocol for `Number`

`coerce: aNumber`	class of receiver, value of argument
`asInteger`	conversion
`asFraction`	conversion
`asFloat`	conversion

Your turn: Object-oriented design

Given `Magnitude`, minimal set of these methods:

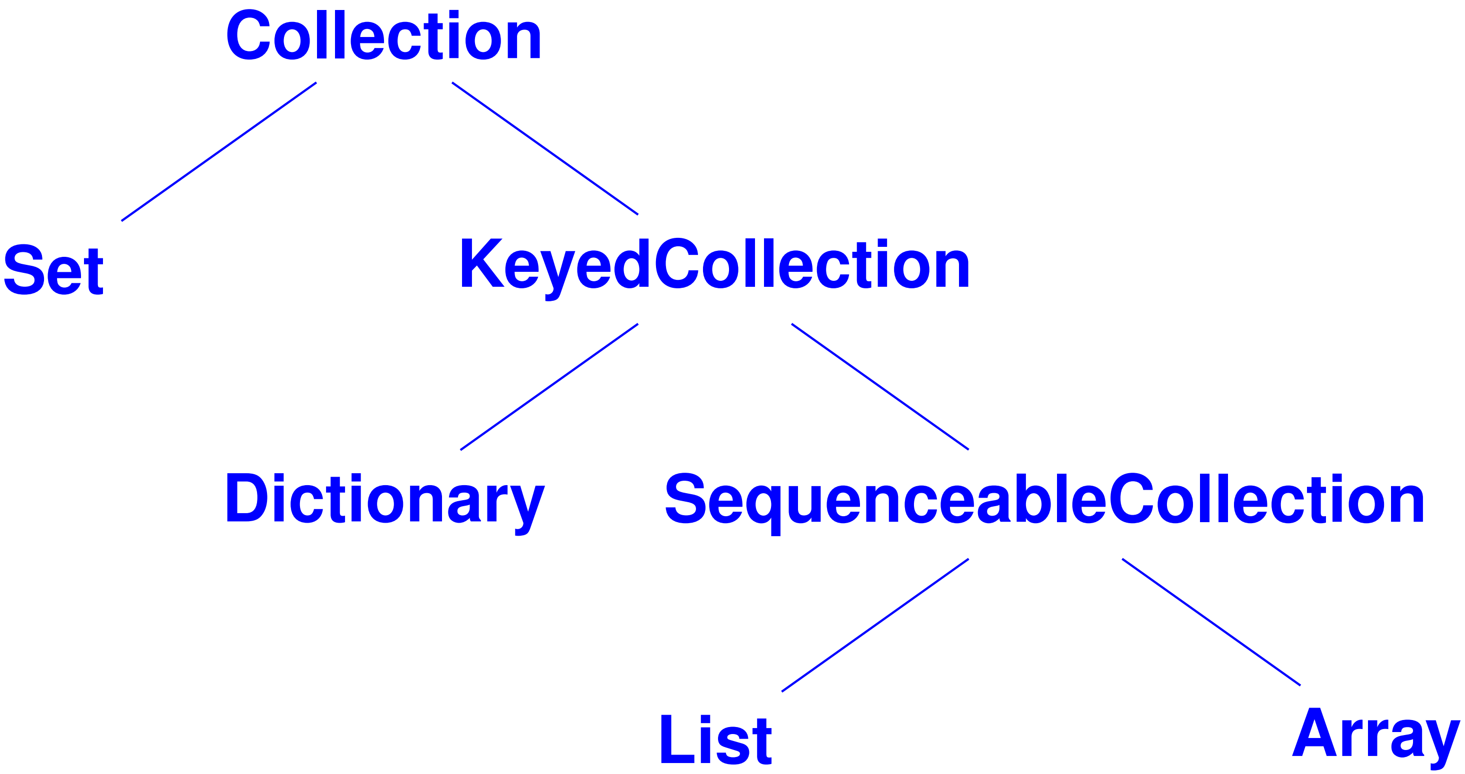
`negated`
`reciprocal`
`abs`
`+`
`-`
`*`
`/`
`negative`
`nonnegative`
`strictlyPositive`
`coerce:`
`asInteger`
`asFraction`
`asFloat`

Number methods

```
(method -    (y)  (+ self (negated  y)))
(method abs ()   (ifTrue:IfFalse (negative self)
                     {(negated  self)}
                     {self}))
(method /    (y)  (* self (reciprocal y)))

(method negative      ()  (<  self (coerce: self 0)))
(method nonnegative   ()  (>= self (coerce: self 0)))
(method strictlyPositive ()
                          (>  self (coerce: self 0)))
```

“Collection hierarchy”



Collection

Set

KeyedCollection

Dictionary

SequenceableCollection

List

Array

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)
```