Example: list filter

```
-> (val ns (new List))
List( )
-> (addAll: ns '(1 2 3 4 5 6))
List( 1 2 3 4 5 6 )
-> (select: ns [block (n) (= 0 (mod: n 2))])
List( 2 4 6 )
```

`select:` dispatches to class `Collection`

Classic imperative paradigm:

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

Name `self` receives message

Example: iteration

```
-> (val ms (select: ns [block (n) (= 0 (mod: n 2))]))
List( 2 4 6 )
-> (do: ms [block (m) (print 'element) (print space)
                      (print 'is) (print space)
                      (println m)])
element is 2
element is 4
element is 6
nil
->
```

Functional code: forms of data

Iteration in Scheme: ask value about form

```
(define app (f xs)
  (if (null? xs)
      'do-nothing
      (begin
         (f (car xs))
         (app f (cdr xs)))))
```

Replace interrogation: dynamic dispatch

No interrogation about form!

Design process still works:

1. Each method defined on a class
2. Class determines
   - How object is formed (class method)
   - From what parts (instance variables)

Each form of data gets its own method!

Object-oriented code: dynamic dispatch

Instead of (app f xs), we have

```
(do: xs f-block)
```

What happens if we send “do f” to the empty list?

What happens if we send “do f” to a cons cell?

Dynamic dispatch revealed

Sending do: to the empty list:

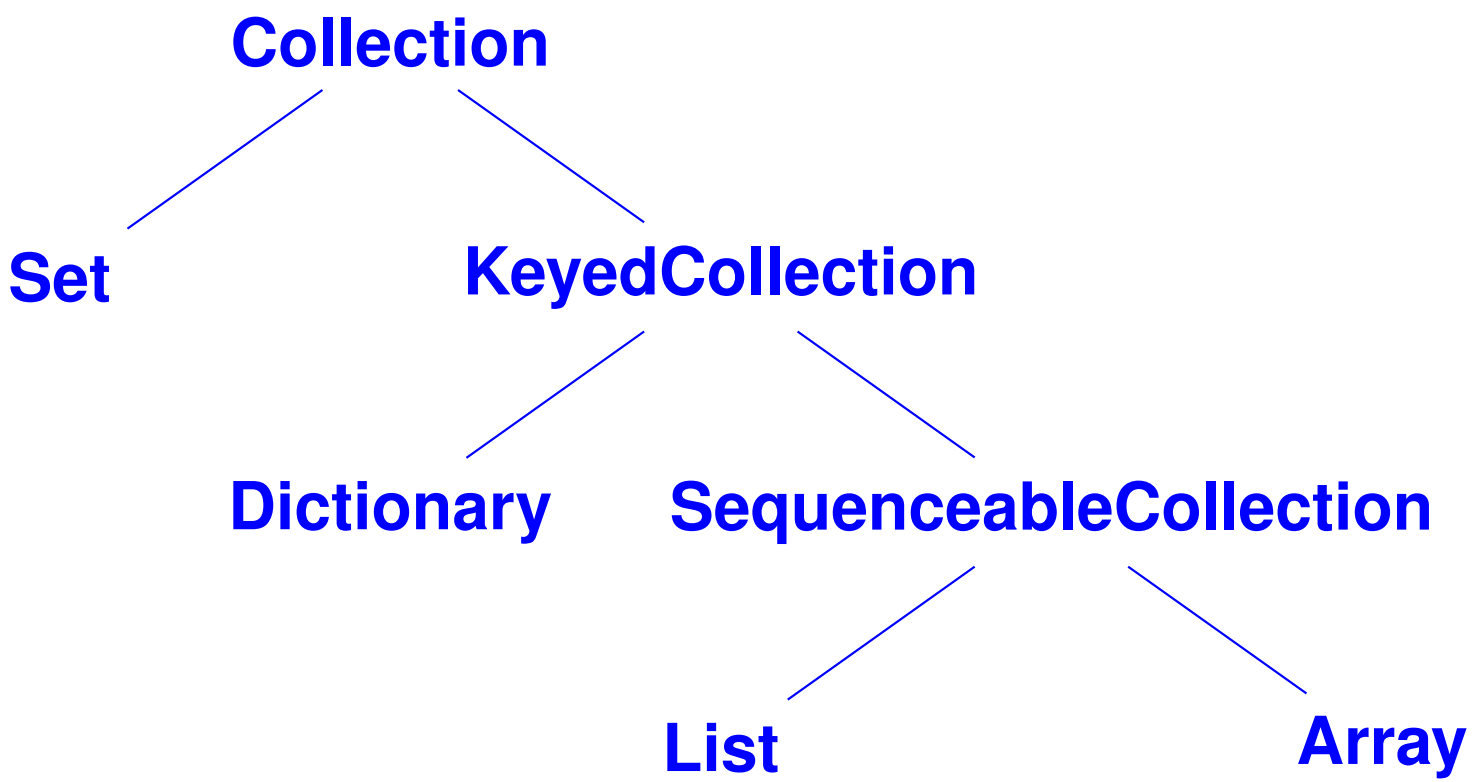
```
(method do: (aBlock) nil)
    ; nil is a global object
```

Sending do: to a cons cell:

```
(method do: (aBlock)
    ; car and cdr are "instance variables"
    (value aBlock car)
    (do: cdr aBlock))
```

What's missing? if!

“Collection hierarchy”



`select:` dispatches to class `Collection`

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

Message	*Protocol*	*Dispatched to*
`species`	`Collection`	`List`
`new`	*class*	`List`, others
`do:`	`Collection`	`List`, `Cons` (delegated)
`ifTrue:`	`Boolean`	`Boolean`, del. `True`, `False`
`value`	block	primitive
`add:`	`Collection`	`List` (then `addLast:`, `insertAfter:`)

Church encoding with blocks

Blocks are closures

- `[block (x) ...]`
- **Instead of** `[block () ...]`**, just** `{...}`

Passed as continuations to Booleans

They are objects

Block Examples

```
-> (val twice [block (n) (+ n n)])
<Block>
-> (value twice 3)
6
-> (val delayed {(println 'hello) 42})
<Block>
-> delayed
<Block>
-> (value delayed)
hello
42
```

Boolean example: minimum

```
-> (val x 10)
-> (val y 20)
-> (ifTrue:ifFalse: (<= x y) {x} {y})
10
```

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`?

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

`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`

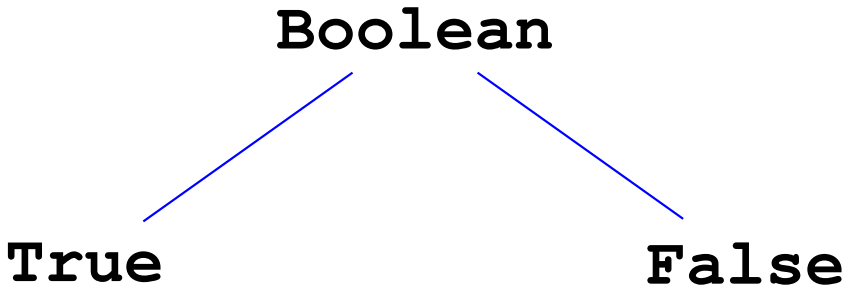
What should `not` look like?

- **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` 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