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**
Functional code: forms of data

Iteration in Scheme:

(define app (f xs)
  (if (null? xs)
    'do-nothing
    (begin
      (f (car xs))
      (app f (cdr xs))))))
Object-oriented code: dynamic dispatch

Instead of (app f xs), we have

(\texttt{do}: \texttt{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”

Collection
  Set
  KeyedCollection
    Dictionary
    SequenceableCollection
      List
      Array
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)

<table>
<thead>
<tr>
<th>Message</th>
<th>Protocol</th>
<th>Dispatched to</th>
</tr>
</thead>
<tbody>
<tr>
<td>species</td>
<td>Collection</td>
<td>List</td>
</tr>
<tr>
<td>new</td>
<td>class</td>
<td>List, others</td>
</tr>
<tr>
<td>do:</td>
<td>Collection</td>
<td>List, Cons (delegated)</td>
</tr>
<tr>
<td>ifTrue:</td>
<td>Boolean</td>
<td>Boolean, del. True, False</td>
</tr>
<tr>
<td>value</td>
<td>block</td>
<td>primitive</td>
</tr>
<tr>
<td>add:</td>
<td>Collection</td>
<td>List <em>(then addLast:, insertAfter:)</em></td>
</tr>
</tbody>
</table>
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>
<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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifTrue:ifFalse: trueBlock falseBlock</td>
<td>Full conditional</td>
</tr>
<tr>
<td>ifTrue: trueBlock</td>
<td>Part conditional (for side effect)</td>
</tr>
<tr>
<td>ifFalse: falseBlock</td>
<td>Part conditional (for side effect)</td>
</tr>
<tr>
<td>&amp; aBoolean</td>
<td>Conjunction</td>
</tr>
<tr>
<td></td>
<td>aBoolean</td>
</tr>
<tr>
<td>not</td>
<td>Negation</td>
</tr>
<tr>
<td>eqv: aBoolean</td>
<td>Equality</td>
</tr>
<tr>
<td>xor: aBoolean</td>
<td>Difference</td>
</tr>
<tr>
<td>and: altBlock</td>
<td>Short-circuit conjunction</td>
</tr>
<tr>
<td>or: altBlock</td>
<td>Short-circuit disjunction</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifTrue: ifFalse: trueBlock falseBlock</td>
<td>Full conditional</td>
</tr>
<tr>
<td>ifTrue: trueBlock</td>
<td>Part conditional (for side effect)</td>
</tr>
<tr>
<td>ifFalse: falseBlock</td>
<td>Part conditional (for side effect)</td>
</tr>
<tr>
<td>&amp; aBoolean</td>
<td>Conjunction</td>
</tr>
<tr>
<td></td>
<td>aBoolean</td>
</tr>
<tr>
<td>not</td>
<td>Negation</td>
</tr>
<tr>
<td>eqv: aBoolean</td>
<td>Equality</td>
</tr>
<tr>
<td>xor: aBoolean</td>
<td>Difference</td>
</tr>
<tr>
<td>and: altBlock</td>
<td>Short-circuit conjunction</td>
</tr>
<tr>
<td>or: altBlock</td>
<td>Short-circuit disjunction</td>
</tr>
</tbody>
</table>
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