## Review: Protocol for Booleans

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

(class Boolean Object
    ()
    ...
    (method not ()
        (ifTrue:ifFalse: self {false} {true}))
    (method and: (aBlock)
        ...)))
Your turn: Short-circuit and:

(class Boolean Object
  ()
  ...
  (method not ()
     (ifTrue:ifFalse: self {false} {true}))
  (method and: (aBlock)
     (ifTrue:ifFalse: self aBlock {self}))
)
Syntax comparison: Impcore

\[ \text{Exp} = \text{LITERAL} \text{ of value} \]
| \text{VAR} \text{ of name} |
| \text{SET} \text{ of name} \ast \text{exp} |
| \text{IF} \text{ of exp} \ast \text{exp} \ast \text{exp} |
| \text{WHILE} \text{ of exp} \ast \text{exp} |
| \text{BEGIN} \text{ of exp list} |
| \text{APPLY} \text{ of name} \ast \text{exp list} |
Syntax comparison: Smalltalk

\[
\text{Exp} = \text{LITERAL of rep} \\
| \text{VAR of name} \\
| \text{SET of name } \ast \text{ exp} \\
| \text{IF of exp } \ast \text{ exp } \ast \text{ exp} \\
| \text{WHILE of exp } \ast \text{ exp} \\
| \text{BEGIN of exp list} \\
| \text{APPLY of name } \ast \text{ exp list} \\
| \text{SEND of name } \ast \text{ exp } \ast \text{ exp list} \\
| \text{BLOCK of name list } \ast \text{ exp list}
\]
Syntax comparison: Smalltalk

\[
\text{Exp} = \text{LITERAL of rep} \\
| \text{VAR of name} \\
| \text{SET of name} \ast \text{exp} \\
| \text{IF of exp} \ast \text{exp} \ast \text{exp} \\
| \text{WHILE of exp} \ast \text{exp} \\
| \text{BEGIN of exp list} \\
| \text{APPLY of name} \ast \text{exp list} \\
| \text{SEND of name} \ast \text{exp} \ast \text{exp list} \\
| \text{BLOCK of name list} \ast \text{exp list}
\]
“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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>negated</td>
<td></td>
</tr>
<tr>
<td>reciprocal</td>
<td></td>
</tr>
<tr>
<td>abs</td>
<td>absolute value</td>
</tr>
<tr>
<td>+ aNumber</td>
<td>addition</td>
</tr>
<tr>
<td>- aNumber</td>
<td>subtraction</td>
</tr>
<tr>
<td>* aNumber</td>
<td>multiplication</td>
</tr>
<tr>
<td>/ aNumber</td>
<td>division (converted!)</td>
</tr>
<tr>
<td>negative</td>
<td>sign check</td>
</tr>
<tr>
<td>positive</td>
<td>nonnegative</td>
</tr>
<tr>
<td>strictlyPositive</td>
<td>sign check</td>
</tr>
</tbody>
</table>


More instance protocol for Number

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coerce: aNumber</td>
<td>class of receiver, value of argument</td>
</tr>
<tr>
<td>asInteger</td>
<td>conversion</td>
</tr>
<tr>
<td>asFraction</td>
<td>conversion</td>
</tr>
<tr>
<td>asFloat</td>
<td>conversion</td>
</tr>
</tbody>
</table>
Your turn: Object-oriented design

**Given Magnitude, minimal set of these methods:**

- negated *
- reciprocal /
- abs negative
- + positive
- - strictlyPositive
- coerce: asInteger
- asFraction
- asFloat
Example class Fraction: initialization

(class Fraction Number
    (num den) ;; representation (concrete!)
    (class-method num:den: (a b)
        (initNum:den: (new self) a b))
    (method initNum:den: (a b) ; private
        (setNum:den: self a b)
        (signReduce self)
        (divReduce self))
    (method setNum:den: (a b)
        (set num a) (set den b) self) ; private
    .. other methods of class Fraction ...
)
Information revealed to self

“Instance variables” num and den
  • Directly available
  • Always and only go with self

Object knows its own representation, invariants, private methods:

(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))
```