### Review: Protocol for Booleans

<table>
<thead>
<tr>
<th>Syntax</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>
Review: Inheritance for Booleans

Boolean is an abstract class
- Instances of True and False only

Method “ifTrue:ifFalse:” defined in True and False

All others defined in 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

Exp = LITERAL of value
    | 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
Syntax comparison: Smalltalk

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

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>= aMagnitude</td>
<td>equality (like Magnitudes)</td>
</tr>
<tr>
<td>&lt; aMagnitude</td>
<td>comparison (ditto)</td>
</tr>
<tr>
<td>&gt; aMagnitude</td>
<td>comparison (ditto)</td>
</tr>
<tr>
<td>&lt;= aMagnitude</td>
<td>comparison (ditto)</td>
</tr>
<tr>
<td>&gt;= aMagnitude</td>
<td>comparison (ditto)</td>
</tr>
<tr>
<td>min: aMagnitude</td>
<td>minimum (ditto)</td>
</tr>
<tr>
<td>max: aMagnitude</td>
<td>maximum (ditto)</td>
</tr>
</tbody>
</table>

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

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

- negated
- reciprocal
- abs
- +
- -
- coerce:
  - asInteger
  - asFraction
  - asFloat
  - strictlyPositive
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)

```plaintext
(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))