# Shape protocol

Three shapes (diamond, triangle, circle)

One protocol:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location: aSymbol</td>
<td>Where is control point?</td>
</tr>
<tr>
<td>adjustPoint:to: aSymbol coordinates</td>
<td>Move yourself</td>
</tr>
<tr>
<td>drawOn: aPicture</td>
<td>Draw on this picture</td>
</tr>
</tbody>
</table>

Your turn:

- What code is shared?
- What is unique?
- How would you organize it into classes?
## Review: Protocol for Booleans

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifTrue: ifFalse:</td>
<td>Full conditional</td>
</tr>
<tr>
<td>ifTrue:</td>
<td>Part conditional (for side effect)</td>
</tr>
<tr>
<td>ifFalse:</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 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) subclass Responsibility methods
- Examples: Boolean, Shape, Collection

Regular class
- Meant to be instantiated
- No subclass Responsibility methods
- Examples: True, Triangle, List
Syntax comparison: Impcore

\[ \text{Exp} = \text{LITERAL of value} \]
| \text{VAR} \quad \text{of name} |
| \text{SET} \quad \text{of name * exp} |
| \text{IF} \quad \text{of exp * exp * exp} |
| \text{WHILE} \quad \text{of exp * exp} |
| \text{BEGIN} \quad \text{of exp list} |
| \text{APPLY} \quad \text{of name * exp list} |
Syntax comparison: 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
Syntax comparison: 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
“Number hierarchy”

Object
  └── Magnitude
      └── Number
          ├── Fraction
          └── Float
              └── Integer
“Extended Number hierarchy”

Object
  |
  Magnitude
    |
    Natural
    |
    Fraction
    |
    Number
      |
      Integer
        |
        SmallInteger
        |
        LargeInteger
          |
          LargePositiveInteger
          |
          LargeNegativeInteger
### Instance protocol for Magnitude

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

**Subclasses:** Date, Natural

- **Compare** Date with Date, Natural w/Natural,...
Your turn: object-oriented design

- \( a \text{Magnitude} = \) equality
- \( a \text{Magnitude} < \) comparison
- \( a \text{Magnitude} > \) comparison
- \( a \text{Magnitude} \leq \) comparison
- \( a \text{Magnitude} \geq \) comparison
- \text{min: } a \text{Magnitude} \) minimum
- \text{max: } a \text{Magnitude} \) maximum

Questions:
- Which methods “subclass responsibility”?
- Which methods on \text{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})))
<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>nonnegative</td>
<td>sign check</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><code>coerce: aNumber</code></td>
<td>class of receiver, value of argument</td>
</tr>
<tr>
<td><code>asInteger</code></td>
<td>conversion</td>
</tr>
<tr>
<td><code>asFraction</code></td>
<td>conversion</td>
</tr>
<tr>
<td><code>asFloat</code></td>
<td>conversion</td>
</tr>
</tbody>
</table>
Your turn: Object-oriented design

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

- negated
- reciprocal
- abs
- +
- -
- coercible to:  
  - asInteger
  - asFraction
  - asFloat
  - strictlyPositive
Example class `Fraction`: initialization

(class Fraction Number

  [num den] ;; representation (concrete!)
  ;; invariants by signReduce, divReduce
  (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 \texttt{coerce}?: \\
(Value of argument, representation of receiver)

(method \texttt{asFraction} ()
  self)
(method \texttt{print} ()
  (print num) (print \\#/) (print den))
(method \texttt{reciprocal} ()
  (signReduce (setNum:den: (new Fraction) den num))))
(method \texttt{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))