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>Expression</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

True

False

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 ("concrete") class
  • Meant to be instantiated
  • No subclass Responsibility methods
  • Examples: True, Triangle, List
Syntax comparison: Impcore

\[ \text{Exp} = \text{LITERAL of value} \]

| 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

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
  | 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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equality</td>
</tr>
<tr>
<td>&lt;</td>
<td>comparison</td>
</tr>
<tr>
<td>&gt;</td>
<td>comparison</td>
</tr>
<tr>
<td>&lt;=</td>
<td>comparison</td>
</tr>
<tr>
<td>&gt;=</td>
<td>comparison</td>
</tr>
<tr>
<td>min:</td>
<td>minimum</td>
</tr>
<tr>
<td>max:</td>
<td>maximum</td>
</tr>
</tbody>
</table>

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>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>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
- +
- –

coerce:

- asInteger
- asFraction
- asFloat

**strictlyPositive**

**nonnegative**

**negative**
Example class \texttt{Fraction}: initialization

\begin{verbatim}
(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 ...
)
\end{verbatim}
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))`
Review: Information revealed to self

Object knows its own representation ("instance variables"), invariants, private methods:

(class Fraction Number
  [num den] ;; representation (concrete!)
  ;; invariants: den > 0, lowest terms
  ;; by signReduce, divReduce
  (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)
Exposing information, part II

Alas! Cannot see representation of argument

How will you know “equal, less or greater”? 
Exposing information, part II

Alas! Cannot see representation of argument

Protocol says “like with like”? Use private methods

(method num () num) ; private
(method den () den) ; private

(method = (f) ;; relies on invariant!
    (and: (= num (num f)) { (= den (den f))})
(method < (f)
    (< (* num (den f)) (* (num f) den)))

Remember behavioral subtyping
Private methods: Your turn

How will you multiply two fractions?
Private methods: Your turn

How will you multiply two fractions?

(method * (f)
  (divReduce
   (setNum:den: (new Fraction)
    (* num (num f))
    (* den (den f)))))
An open system

**Number protocol:** like multiplies with like

What about large and small integers?
- How to multiply two small integers?
- How to multiply two large integers?

How is algorithm known?

Each object knows its own algorithm:
- Small: Use machine-primitive multiplication
- Large: Multiply magnitudes; choose sign
Review: Two kinds of knowledge

I can send message to you:
  • I know your protocol

I can inherit from you:
  • I know my subclass responsibilities
Knowledge of protocol

Three levels of knowledge:
1. I know only your public methods
   Example: select:
2. You are like me: share private methods
   Example: * and + on Fraction
3. I must get to know you: double dispatch
   Example: * and + on mix of integers
Double dispatch: extending open systems

I claim:
- Large integers and small integers both \texttt{Integer}
- Messages =, <, +, * ought to mix freely
- Large and small integers have different private protocol

Private for large integers: \texttt{magnitude}

Private for small integers: \texttt{mul:withOverflow}
Double dispatch code operation & protocol

Example messages:
- I answer the small-integer protocol, add me to yourself
- I answer the large-positive integer protocol, multiply me by yourself

Message encodes
- Operation to be performed
- Protocol accepted by argument
Your turn: responding to double dispatch

How do you act?

1. As small integer, you receive “add small integer \( n \) to self”
2. As small integer, you receive “multiply large positive integer \( N \) by self”
3. As large positive integer, you receive “add small integer \( n \) to self”
4. As large positive integer, you receive “multiply large positive integer \( N \) by self”
Your turn: using double dispatch

On what class does each method go?

A. (method + (aNumber)
   (addSmallIntegerTo: aNumber self))

B. (method * (anInteger)
   (multiplyByLargePositiveInteger: anInteger self))

(See the “double dispatch”: + then
addSmallIntegerTo:)
Information-hiding summary

Three levels
1. I use your public protocol
2. We are alike; I add our private protocol
3. Your protocol is revealed by double dispatch
Collection mutators

add: newObject  Add argument
addAll: aCollection  Add every element of arg
remove: oldObject  Remove arg, error if absent
remove:ifAbsent: oldObject exnBlock  Remove the argument, evaluate exnBlock if absent
removeAll: aCollection  Remove every element of arg
Collection observers

isEmpty  Is it empty?
size    How many elements?
includes: anObject  Does receiver contain arg?
occurrencesOf: anObject  How many times?
detect: aBlock  Find and answer element
     satisfying aBlock (cf \mu Scheme exists?)
detect:ifNone: aBlock exnBlock  Detect,
     recover if none
asSet    Set of receiver’s elements
**Collection iterators**

do: aBlock  For each element x, evaluate (value
   aBlock x).
inject:into: thisValue binaryBlock
   Essentially μScheme foldl
select: aBlock  Essentially μScheme filter
reject: aBlock  Filter for not satisfying aBlock
collect: aBlock  Essentially μScheme map
Implementing collections

(class Collection Object
  () ; abstract
  (method do: (aBlock)
    (subclassResponsibility self))
  (method add: (newObject)
    (subclassResponsibility self))
  (method remove:ifAbsent (oldObj exnBlock)
    (subclassResponsibility self))
  (method species ()
    (subclassResponsibility self))
  \{other methods of class Collection\}
)
Reusable methods

(other methods of class Collection) =
(method addAll: (aCollection)
  (do: aCollection [block(x) (add: self x)])
  aCollection)
(method size () [locals temp]
  (set temp 0)
  (do: self [block(_) (set temp (+ temp 1))])
  temp)

These methods always work
Subclasses can override (redefine) with more efficient versions
species method

Create “collection like the reciever”

Example: filtering

```smalltalk
(other methods of class Collection) =
(method select: (aBlock) [locals temp]
  (set temp (new (species self)))
  (do: self [block (x)
    (ifTrue: (value aBlock x)
      {(add: temp x)})]]
  temp)
```
Subtyping mathematically

Always transitive

\[ \tau_1 <: \tau_2 \quad \tau_2 <: \tau_3 \]
\[ \tau_1 <: \tau_3 \]

Key rule is subsumption:

\[ e : \tau \quad \tau <: \tau' \]
\[ e : \tau' \]

*(implicit subsumption: no cast)*
Subtyping is not inheritance

Subtype understands more messages:

\[
\{m_1 : \tau_1, \ldots, m_n : \tau_n, \ldots, m_{n+k} : \tau_{n+k}\} \prec \{m_1 : \tau_1, \ldots, m_n : \tau_n\}
\]

If an object understands messages \(m_1, \ldots, m_n\), and possibly more besides, you can use it where \(m_1, \ldots, m_n\) are expected

- Methods must behave as expected

Behavioral subtyping (in Ruby, “duck typing”)
The four crucial Collection methods

(class Collection Object
 () ; abstract
 (method do: (aBlock)
   (subclassResponsibility self))
(method add: (newObject)
   (subclassResponsibility self))
(method remove:ifAbsent (oldObj exnBlock)
  (subclassResponsibility self))
(method species ()
  (subclassResponsibility self))
{others methods of class Collection}
(class Set Collection

(members) ; list of elements
(class-method new () (initSet (new super)))
(method initSet () ; private method
  (set members (new List))
  self)
(method do: (aBlock) (do: members aBlock))
(method remove:ifAbsent: (item exnBlock)
  (remove:ifAbsent: members item exnBlock))
(method add: (item)
  (ifFalse: (includes: members item)
    {(add: members item)})
  item)
(method species () Set)
(method asSet ( () self) ; extra efficient
(class Set Collection
  (members) ; list of elements
  (class-method new () (initSet (new super)))
  (method initSet () ; private method
    (set members (new List))
    self)
  (method do: (aBlock) (do: members aBlock))
  (method remove:ifAbsent: (item exnBlock)
    (remove:ifAbsent: members item exnBlock))
  (method add: (item)
    (ifFalse: (includes: members item)
      {
        (add: members item))
    item)
  (method species () Set)
  (method asSet () self) ; extra efficient
)