Clojure
Protocols

Stuart Halloway
stu@clojure.com
@stuarthalloway
reading code
# atomic data types

<table>
<thead>
<tr>
<th>type</th>
<th>example</th>
<th>java equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>&quot;foo&quot;</td>
<td>String</td>
</tr>
<tr>
<td>character</td>
<td>\f</td>
<td>Character</td>
</tr>
<tr>
<td>regex</td>
<td>#&quot;fo*&quot;</td>
<td>Pattern</td>
</tr>
<tr>
<td>a. p. integer</td>
<td>42</td>
<td>Int/Long/BigInteger</td>
</tr>
<tr>
<td>double</td>
<td>3.14159</td>
<td>Double</td>
</tr>
<tr>
<td>a.p. double</td>
<td>3.14159M</td>
<td>BigDecimal</td>
</tr>
<tr>
<td>boolean</td>
<td>true</td>
<td>Boolean</td>
</tr>
<tr>
<td>nil</td>
<td>nil</td>
<td>null</td>
</tr>
<tr>
<td>ratio</td>
<td>22/7</td>
<td>N/A</td>
</tr>
<tr>
<td>symbol</td>
<td>foo, +</td>
<td>N/A</td>
</tr>
<tr>
<td>keyword</td>
<td>:foo, ::foo</td>
<td>N/A</td>
</tr>
</tbody>
</table>
## data literals

<table>
<thead>
<tr>
<th>type</th>
<th>properties</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>singly-linked, insert at front</td>
<td>(1 2 3)</td>
</tr>
<tr>
<td>vector</td>
<td>indexed, insert at rear</td>
<td>[1 2 3]</td>
</tr>
<tr>
<td>map</td>
<td>key/value</td>
<td>{a: 100, b: 90}</td>
</tr>
<tr>
<td>set</td>
<td>key</td>
<td>#{a, b}</td>
</tr>
</tbody>
</table>
function call

semantics:

structure:

```
println "Hello World"
```
function definition

define a fn: `greet`

fn name: `greet`

docstring: "Returns a friendly greeting" [your-name]

arguments: [your-name]

fn body: (str "Hello, " your-name))
(defn greet
  "Returns a friendly greeting"
  [your-name]
  (str "Hello, " your-name))
metadata

prefix with ^

(class name or arbitrary map)

(defn ^String greet
  "Returns a friendly greeting"
  [your-name]
  (str "Hello, " your-name))
what is OO?
objects provide...

Object
methods

Object
polymorphism
namespaces
uncontrolled mutation

- Interface
- Interface
- Base Class
- struct
  - xxxx
  - yyyy
  - zzzz

Uncontrolled Mutation
clojure features are a la carte
superstructure

polymorphism

structure

namespaces

identity

perception

values

functions

generic data access
in this talk

- protocols
- records

- polymorphism
  - struct
    - xxxx
    - yyyy
    - zzzz

- types
- namespaces

- identity
- perception

- values
- functions
- generic data access
records
(defrecord Foo [a b c])
-> user.Foo

named type with slots
(defrecord Foo [a b c])
  -> user.Foo

(def f (Foo 1 2 3))
  -> #'user/f

named type with slots

positional constructor
defrecord

(defrecord Foo [a b c])
-> user.Foo

(def f (Foo 1 2 3))
-> #'user/f

(:b f)
-> 2

named type with slots
positional constructor
keyword access
defrecord

(defrecord Foo [a b c])
-> user.Foo

(def f (Foo 1 2 3))
-> #'user/f

(:b f)
-> 2

(class f)
-> user.Foo

- named type with slots
- positional constructor
- keyword access
- plain ol' class
defrecord

(defrecord Foo [a b c])
-> user.Foo

(def f (Foo 1 2 3))
-> #'user/f

(:b f)
-> 2

(class f)
-> user.Foo

(supers (class f))
-> #{clojure.lang.IObj clojure.lang.IKeywordLookup java.util.Map
clojure.lang.IPersistentMap clojure.lang.IMeta
clojure.lang.Counted
clojure.lang.ISeqable
clojure.lang.IAssociative}
defrecord

(defrecord Foo [a b c])
-> user.Foo

(def f (Foo. 1 2 3))
-> #'user/f

(:b f)
-> 2

(class f)
-> user.Foo

(supers (class f))
-> #{clojure.lang.IObj clojure.lang.IKeywordLookup java.util.Map
clojure.lang.IPersistentMap clojure.lang.IMeta java.lang.Object
clojure.lang.Iterable clojure.lang.ILookup clojure.lang.Seqable
clojure.lang.Counted clojure.lang.IPersistentCollection
clojure.lang.Associative}

named type with slots

positional constructor

keyword access

plain ol' class

*Rich abstracts so you don't have to
(def stu {:fname "Stu" 
  :lname "Halloway" 
  :address {:street "200 N Mangum" 
    :city "Durham" 
    :state "NC" 
    :zip 27701}})
from maps...

```clojure
(def stu {
  :fname "Stu"
  :lname "Halloway"
  :address {
    :street "200 N Mangum"
    :city "Durham"
    :state "NC"
    :zip 27701}})

(:lname stu)  =>  "Halloway"
```

---

**data-oriented**

**keyword access**
from maps...

(def stu {:fname "Stu"
    :lname "Halloway"
    :address {:street "200 N Mangum"
              :city "Durham"
              :state "NC"
              :zip 27701}})

textual representation:

```
(def stu {:fname "Stu"
    :lname "Halloway"
    :address {:street "200 N Mangum"
              :city "Durham"
              :state "NC"
              :zip 27701}})
```

(keyword access)

```
(:lname stu)
```

```
=> "Halloway"
```

(nested access)

```
(-> stu :address :city)
```

```
=> "Durham"
```
from maps...

```clojure
(def stu { :fname "Stu" 
           :lname "Halloway" 
           :address { :street "200 N Mangum" 
                      :city "Durham" 
                      :state "NC" 
                      :zip 27701} })
```

`:lname stu) => "Halloway"

`(-> stu :address :city)` => "Durham"

`(assoc stu :fname "Stuart") => {:fname "Stuart", :lname "Halloway", :address ...}`
from maps...

```clojure
(def stu {:fname "Stu"
  :lname "Halloway"
  :address {:street "200 N Mangum"
            :city "Durham"
            :state "NC"
            :zip 27701}})

(:lname stu) => "Halloway"

(-> stu :address :city) => "Durham"

(assoc stu :fname "Stuart") => {:fname "Stuart", :lname "Halloway",
                                 :address ...}

(update-in stu [:address :zip] inc)
=> {:address {:street "200 N Mangum",
              :zip 27702 ...} ...}
```

Monday, October 4, 2010
...to records!

```cljs
(defrecord Person [fname lname address])
(defrecord Address [street city state zip])
(def stu (Person "Stu" "Halloway"
     (Address "200 N Mangum"
        "Durham"
        "NC"
        27701)))

(:lname stu)
=> "Halloway"

(-> stu :address :city)
=> "Durham"

(assoc stu :fname "Stuart")
=> :user.Person{:fname "Stuart", :lname"Halloway",
    :address ...}

(update-in stu [:address :zip] inc)
=> :user.Person{:address {:street "200 N Mangum",
    :zip 27702 ...} ...}
```
(defrecord Person [fname lname address])
(defrecord Address [street city state zip])
(def stu (Person. "Stu" "Halloway"
                (Address. "200 N Mangum"
                          "Durham"
                          "NC"
                          27701)))

(:lname stu)
=> "Halloway"

(-> stu :address :city)
=> "Durham"

(assoc stu :fname "Stuart")
=> :user.Person{:fname "Stuart", :lname"Halloway",
                :address ...}

(update-in stu [:address :zip] inc)
=> :user.Person{:address {:street "200 N Mangum",
                          :zip 27702 ...} ...}
...to records!

(defrecord Person [fname lname address])
(defrecord Address [street city state zip])
(def stu (Person. "Stu" "Halloway"
              (Address. "200 N Mangum"
                         "Durham"
                         "NC"
                         27701)))

(:lname stu)
=> "Halloway"

(-> stu :address :city)
=> "Durham"

(assoc stu :fname "Stuart")
=> :user.Person{:fname "Stuart", :lname"Halloway",
                :address ...}

(update-in stu [:address :zip] inc)
=> :user.Person{:address {:street "200 N Mangum",
                          :zip 27702 ...} ...}
(defrecord Person [fname lname address])
(defrecord Address [street city state zip])
(def stu (Person. "Stu" "Halloway"
  (Address. "200 N Mangum"
    "Durham"
    "NC"
    27701)))

(:lname stu)
=> "Halloway"

(-> stu :address :city)
=> "Durham"

(assoc stu :fname "Stuart")
=> :user.Person{:fname "Stuart", :lname"Halloway",
  :address ...}

(update-in stu [:address :zip] inc)
=> :user.Person{:address {:street "200 N Mangum",
  :zip 27702 ...} ...}
protocols
(defprotocol AProtocol
  "A doc string for AProtocol abstraction"
  (bar [a b] "bar docs")
  (baz [a] "baz docs"))
(defprotocol AProtocol
  "A doc string for AProtocol abstraction"
  (bar [a b] "bar docs")
  (baz [a] "baz docs"))
	named set of generic functions
defprotocol

(defprotocol AProtocol
  "A doc string for AProtocol abstraction"
  (bar [a b] "bar docs")
  (baz [a] "baz docs"))

named set of generic functions

polymorphic on type of first argument
(defprotocol AProtocol
  "A doc string for AProtocol abstraction"
  (bar [a b] "bar docs")
  (baz [a] "baz docs"))

named set of generic functions
polymorphic on type of first argument
defines fns in same namespace as protocol
(defprotocol AProtocol
  "A doc string for AProtocol abstraction"
  (bar [a b] "bar docs")
  (baz [a] "baz docs"))

named set of generic functions

polymorphic on type of first argument

defines fns in same namespace as protocol
extending a protocol
extending a protocol

inline
extending a protocol

inline

extend protocol to multiple types
extending a protocol

inline

extend protocol to multiple types

extend type to multiple protocols
extending a protocol

inline
extend protocol to multiple types
extend type to multiple protocols
build directly from fns and maps
extending a protocol

inline
extend protocol to multiple types
extend type to multiple protocols
build directly from fns and maps

extension happens in the protocol fns, not in the types
extending a protocol

inline

extend protocol to multiple types

extend type to multiple protocols

build directly from fns and maps

extension happens in the protocol fns, not in the types
(deftype Bar [a b c]
   AProtocol
   (bar [this b] "Bar bar")
   (baz [this] (str "Bar baz " c)))))

(def b (Bar. 5 6 7))

(baz b)

=> "Bar baz 7"
extend type to protocol(s)

(baz "a")

java.lang.IllegalArgumentException:
No implementation of method: :baz of protocol:
#'user/AProtocol found for class: java.lang.String

(extend-type String
   AProtocol
   (bar [s s2] (str s s2))
   (baz [s] (str "baz " s)))

(baz "a")

=> "baz a"
extending protocol to type(s)

;; elided from clojure.java.io
(extend-protocol Coercions
    String
    (as-file [s] (File. s))
    (as-url [s] (URL. s))

    File
    (as-file [f] f)
    (as-url [f] (.toURL f))

    URI
    (as-url [u] (.toURL u))
    (as-file [u] (as-file (as-url u))))
roll-your-own

;; elided from clojure.java.io
(extend InputStream
  IOFactory
  (assoc default-streams-impl
    :make-input-stream
    (fn [x opts] (BufferedInputStream. x))
    :make-reader
    inputstream->reader))

(extend Reader
  IOFactory
  (assoc default-streams-impl
    :make-reader
    (fn [x opts] (BufferedReader. x))))
(let [x 42
      r (reify AProtocol
         (bar [this b] "reify bar")
         (baz [this ] (str "reify baz " x))))
    (baz r))

=> "reify baz 42"
Let

\( r \) (reify AProtocol

(bar [this b] "reify bar")

(baz [this ] (str "reify baz " x)))]

(baz r))

=> "reify baz 42"

instantiate an unnamed type
reify

(\[x 42
  r (\[\text{reify} \ AProtocol
      (\[\text{bar} \ [\text{this} \ b] \ "\text{reify bar}"\n      (\[\text{baz} \ [\text{this}] \ (\text{str} \ "\text{reify baz} \ " \ x)))\]
    (baz r))\n
=> "\text{reify baz} \ 42"

\text{implement 0 or more protocols or interfaces}\n
\text{instantiate an unnamed type}\n
Monday, October 4, 2010
`reify` is a function that

- Instantiates an unnamed type
  - Implements 0 or more protocols or interfaces
  - Closes over the environment like fn

```clojure
(let [x 42
      r (reify AProtocol
          (bar [this b] "reify bar")
          (baz [this ] (str "reify baz " x))))
  (baz r))
=> "reify baz 42"
```
the expression
problem
the expression problem

A

abstraction
concretion
the expression problem
the expression problem

A should be able to work with B’s abstractions, and vice versa, without modification of the original code.

- A: abstraction
- B: concretion
is this really a problem?

A

B

just use interfaces for abstraction (??)

abstraction

concretion
example: arraylist vs. the abstractions

- java.util.List
- clojure.lang.Counted
- clojure.lang.Seqable
- ArrayList
example: string vs. the abstractions

String

java.util.List

clojure.lang.Counted

clojure.lang.Seqable
A can't inherit from B
A can't inherit from B

B is newer than A
A can't inherit from B

B is newer than A

A is hard to change
A can't inherit from B

B is newer than A

A is hard to change

we don't control A
A can't inherit from B

B is newer than A

A is hard to change

we don't control A

happens even **within** a single lib
A can't inherit from B

B is newer than A

A is hard to change

we don't control A

happens even within a single lib
some approaches to the expression problem
1. wrappers

strings are not collections

java.util.Collection

java.util.List

String
I. wrappers

Strings are not collections, so make a NiftyString that is
wrappers = complexity
wrappers = complexity

ruin identity
wrappers = complexity

ruin identity

ruin equality
wrappers = complexity

ruin identity

ruin equality

cause nonlocal defects
wrappers = complexity

ruin identity

ruin equality

cause nonlocal defects

don't compose: \[ AB + AC \neq ABC \]
wrappers = complexity

ruin identity
ruin equality
cause nonlocal defects

don't compose: \[ AB + AC \neq ABC \]

have bad names
wrappers = complexity

ruin identity

ruin equality

cause nonlocal defects

don't compose: \( AB + AC \neq ABC \)

have bad names
2. monkey patching

- Strings are not collections.

- `java.util.List`

- `java.util.Collection`
2. monkey patching

Strings are not collections.

java.util.List

sneak in and change them!

String
2. monkey patching

Strings are not collections.

- Common in e.g. Ruby
- Not possible in Java

Java.util.List

Java.util.Collection

String

Sneak in and change them!
monkey patching = complexity
monkey patching = complexity

preserves identity (mostly)
monkey patching = complexity

preserves identity (mostly)

ruins namespacing
monkey patching = complexity

preserves identity (mostly)

ruins namespacing

causes nonlocal defects
monkey patching = complexity

preserves identity (mostly)

ruins namespacing

causes nonlocal defects

forbidden in some languages
monkey patching = complexity

preserves identity (mostly)
ruins namespacing
causes nonlocal defects

forbidden in some languages
3. generic functions (CLOS)

String

- count
- reduce
- map
3. generic functions (CLOS)

- String

- Polymorphism lives in the fns

- count
- reduce
- map
3. generic functions (CLOS)

- String

- polymorphism lives in the fns
  - count
  - reduce
  - map

- don't touch existing implementation, just use it
polymorphism a la carte

values

types

polymorphism
polymorphism a la carte

polymorphism in the fns, not the types

values

types

polymorphism
polymorphism a la carte

polymorphism in the fns, not the types

no "isa" requirement

values

types
	polymorphism
polymorphism a la carte

polymorphism in the fns, not the types

no "isa" requirement

no type intrusion necessary
polymorphism a la carte

polymorphism in the fns, not the types

no "isa" requirement

no type intrusion necessary
protocols = generic functions
  - arbitrary dispatch
  + speed
  + grouping

(and still powerful enough to solve the expression problem!)
for more information
community

main Clojure site

http://clojure.org/

google group

http://groups.google.com/group/clojure

Clojure/core team

http://clojure.com

The conj

http://clojure-conj.org/
free resources

labrepl
http://github.com/relevance/labrepl

screencasts
http://clojure.blip.tv/

full disclojure screencasts
http://vimeo.com/channels/fulldisclojure

mark volkmann’s Clojure article
http://java.ociweb.com/mark/clojure/article.html
thanks!

http://clojure.org
extra
example:
rock/paper/scissors

(defprotocol Player
  (choose [p])
  (update-strategy [p me you]))
a player

(defprotocol Player
  (choose [p])
  (update-strategy [p me you]))
(defprotocol Player
  (choose [p])
  (update-strategy [p me you]))

pick :rock, :paper, or :scissors

return an updated Player based on what you and I did
(defrecord Stubborn [choice]
  Player
  (choose [__] choice)
  (update-strategy [this __ __] this))
stubborn player

(initialize with choice)

(defrecord Stubborn [choice]
  Player
  (choose [_] choice)
  (update-strategy [this _ _] this))
stubborn player

(defrecord Stubborn [choice]
  Player
  (choose [_] choice)
  (update-strategy [this _ _] this))
stubborn player

```clojure
(defrecord Stubborn [choice]
  Player
  (choose [_] choice)
  (update-strategy [this _ _] this))
```

- initialize with choice
- play the choice
- never change
(defrecord Mean [last-winner]
  Player
  (choose [_]
    (if last-winner
      last-winner
      (random-choice))
  (update-strategy [_ me you]
    (Mean. (when (iwon? me you) me))))
(defrecord Mean [last-winner]
  Player
  (choose [_]
    (if last-winner
      last-winner
      (random-choice)))
  (update-strategy [me you]
    (Mean. (when (iwon? me you) me)))

last thing that worked for me

mean player
(defrecord Mean [last-winner]
  Player
  (choose [_]
    (if last-winner
      last-winner
      (random-choice))
  (update-strategy [_ me you]
    (Mean. (when (iwon? me you) me)))))

mean player

last thing that worked for me
play last winner or random
mean player

(defrecord Mean [last-winner]
  Player
  (choose [_]
    (if last-winner
      last-winner
      (random-choice)))
  (update-strategy [_ me you]
    (Mean. (when (iwon? me you) me))))

last thing that worked for me
play last winner or random
remember how/if I won
deftype
programming constructs are not like domain data
use defrecord for domain information
use defrecord for domain information

use deftype for programming constructs
(deftype Bar [a b c])
-> user.Bar

still a named type with slots
(deftype Bar [a b c])
-> user.Bar

(deftype Bar [1 2 3])
-> #'user/o

still a named type with slots
constructor, check
(deftype Bar [a b c])
-> user.Bar

(defto (Bar 1 2 3))
-> #'user/o

(.b o)
-> 2

still a named type with slots
constructor, check
direct field access only
(deftype Bar [a b c])
-> user.Bar

(deftype still a named type with slots
    Bar [1 2 3])
-> #'user/o

(def constructor, check
    o (Bar 1 2 3))
-> #'user/o

(.b o) direct field access only
-> 2

(class o) still a plain ol' class
-> user.Bar
(deftype Bar [a b c])
-> user.Bar

(def o (Bar 1 2 3))
-> #'user/o

(.b o)
-> 2

class o)
-> user.Bar

type with slots

constructor, check

direct field access only

still a plain ol' class

yoyo*
(deftype Bar [a b c])
  -> user.Bar

(deftypenamer still a named type with slots)

def o (Bar. 1 2 3))
  -> #'user/o

constructor, check

(.b o)
  -> 2
direct field access only

(class o)
  -> user.Bar

still a plain ol' class

(supers (class o))
  -> #{java.lang.Object}
yoyo*

*you're on your own
the other constructor

```
(def f (Foo. 1 2 3 {:meta 1} {:extra 4}))
-> #'user/f
```

```
(meta f)
-> {:meta 1}
```

```
(into {} f)
-> {:a 1, :b 2, :c 3, :extra 4}
```
details
details

type fields can be primitives
type fields can be primitives

value-based equality and hash
type fields can be primitives

value-based equality and hash

in-line methods defs can inline
details

type fields can be primitives
value-based equality and hash
in-line methods defs can inline
keyword field lookups can inline
details

type fields can be primitives
value-based equality and hash
in-line methods defs can inline
keyword field lookups can inline
protocols make interfaces (interop only)
details

type fields can be primitives
value-based equality and hash
in-line methods defs can inline
keyword field lookups can inline
protocols make interfaces (interop only)
add java annotations (interop only)
details

type fields can be primitives
value-based equality and hash
in-line methods defs can inline
keyword field lookups can inline
protocols make interfaces (interop only)
add java annotations (interop only)

deftype fields can be mutable (experts only)
details

type fields can be primitives
value-based equality and hash
in-line methods defs can inline
keyword field lookups can inline
protocols make interfaces (interop only)
add java annotations (interop only)

deftype fields can be mutable (experts only)
multimethods
polymorphism

\[ \text{square.draw(canvas)} \]

\[ \text{f2(circle, canvas)} \]

\[ \text{f1(square, canvas)} \]

\[ \text{circle.draw(canvas)} \]
p is just a function

square.draw(canvas)

circle.draw(canvas)

p() {return this.class;}

f1(square, canvas)

f2(circle, canvas)
clojure multimethods

```
(defmulti blank? class)
```

dispatch by class of first arg
(defmulti blank? class)

(Blank? "blah")
-> No method in multimethod 'blank?'
   for dispatch value: class java.lang.String"

dispatch by
class of first arg

no impl yet!
(defmulti blank? class)

dispatch by class of first arg

(blank? "blah")
-> No method in multimethod 'blank?'
for dispatch value: class java.lang.String

no impl yet!

(defmethod blank? String [s]
  (every? #(Character/isWhitespace %)) s))

add impls anytime
defmulti blank? class

dispatch by class of first arg

(blank? "blah")
-> No method in multimethod 'blank?'
for dispatch value: class java.lang.String

no impl yet!

(defmethod blank? String [s]
  (every? #(Character/isWhitespace %)) s))

add impls anytime

(blank? "blah")
-> false
this isn’t special

square.draw(canvas)

circle.draw(canvas)

f1(square, canvas)

f2(circle, canvas)

p(this, that)
{return this.class;}

circle.draw(canvas)
check all args

```
(fn [this, that]
 [(class this)
  (class that)])
```

- `(fn [square, canvas])`
- `(fn [circle, canvas])`
- `(fn [square, surface])`
- `(fn [circle, surface])`
check arg twice

(defun (fn [this, that]
  [(class this)
   (opaque? this)
   (class that)]))

fn1
fn2
fn3
fn4
fn5
fn6
fn7
fn8
example: coerce

```
(defmulti coerce
  (fn [dest-class src-inst]
    [dest-class (class src-inst)]))
```

- Define a multimethod
- Based on dest (a class)
- And src (an inst)
(defmethod coerce
  [java.io.File String]
  [_ str]
  (java.io.File. str))

(defmethod coerce
  [Boolean/TYPE String] [_ str]
  (contains?
   #{"on" "yes" "true"}
   (.toLowerCase str)))

method impls
(defmethod coerce  
  :default  
  [dest-cls obj]  
  (cast dest-cls obj))
(defmulti whatami? class)

(defmethod whatami? java.util.Collection
    [_] "a collection")

(whatami? (java.util.LinkedList.))
-> "a collection"

(defmethod whatami? java.util.List
    [_] "a list")

(whatami? (java.util.LinkedList.))
-> "a list"

add methods anytime

most derived type wins
name inheritance

(defmulti interest-rate :type)
(defmethod interest-rate ::account 
    [__] 0M)
(defmethod interest-rate ::savings 
    [__] 0.02)

double colon (::) is shorthand for resolving keyword into the current namespace, e.g. ::savings == :my.current.ns/savings
deriving names

(derive ::checking ::account)
(derive ::savings ::account)

(interest-rate {::type ::checking})
-> 0M

there is no ::checking method, so select method for base name ::account
<table>
<thead>
<tr>
<th>function</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefer-method</td>
<td>resolve conflicts</td>
</tr>
<tr>
<td>methods</td>
<td>reflect on {dispatch, meth} pairs</td>
</tr>
<tr>
<td>get-method</td>
<td>reflect by dispatch</td>
</tr>
<tr>
<td>remove-method</td>
<td>remove by dispatch</td>
</tr>
<tr>
<td>prefers</td>
<td>reflect over preferences</td>
</tr>
</tbody>
</table>
multimethod elegance
multimethod elegance

solve the expression problem
multimethod elegance

solve the expression problem

no wrappers
multimethod elegance

solve the expression problem

no wrappers

non-intrusive
multimethod elegance

solve the expression problem

no wrappers

non-intrusive

open (add more at any time)
multimethod elegance

solve the expression problem

no wrappers

non-intrusive

open (add more at any time)

namespaces work fine
multimethod elegance

solve the expression problem

no wrappers

non-intrusive

open (add more at any time)

namespaces work fine