Prototypes with Multiple Dispatch - Outline

1. Object-Oriented Programming

2. The Design Space

3. A Scenario

4. The Problem(s)

5. Multiple Dispatch

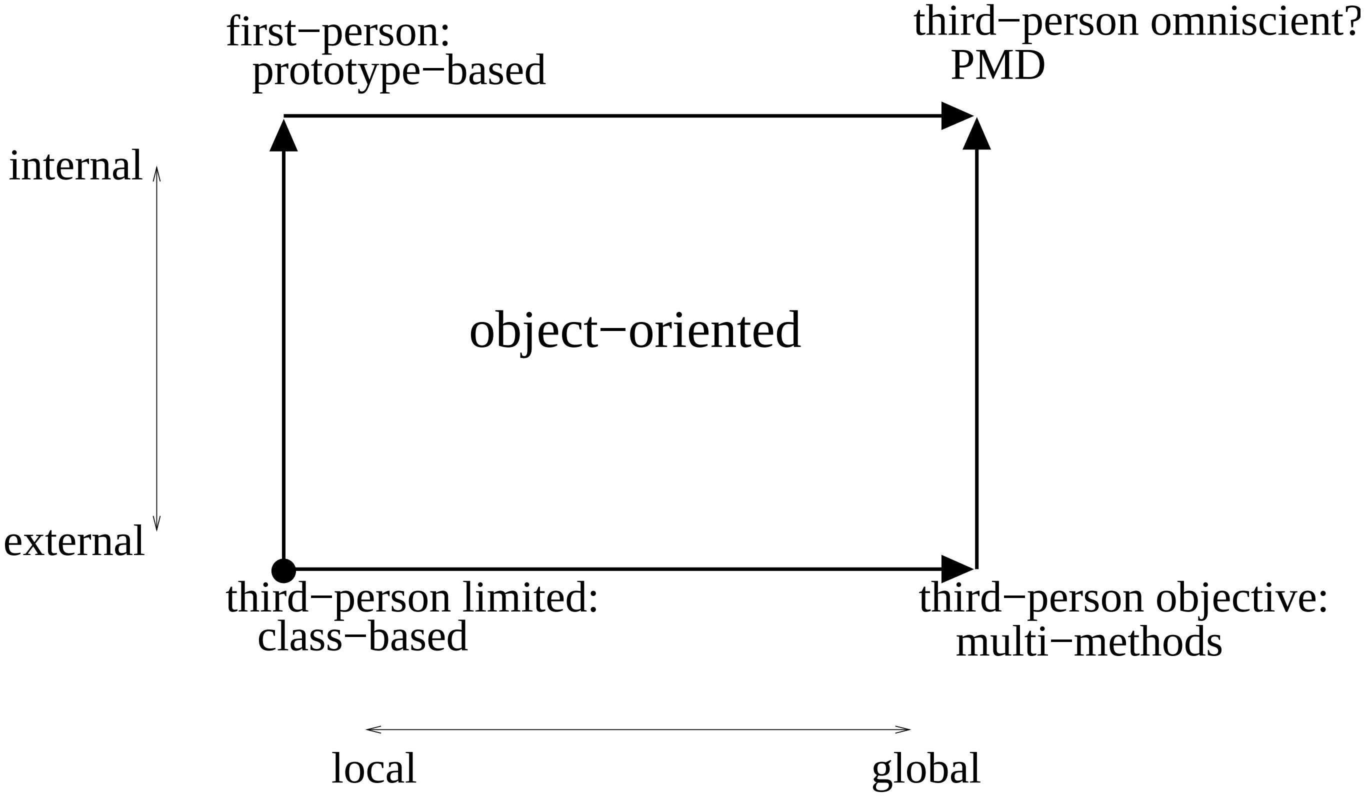
6. Prototypes

7. PMD

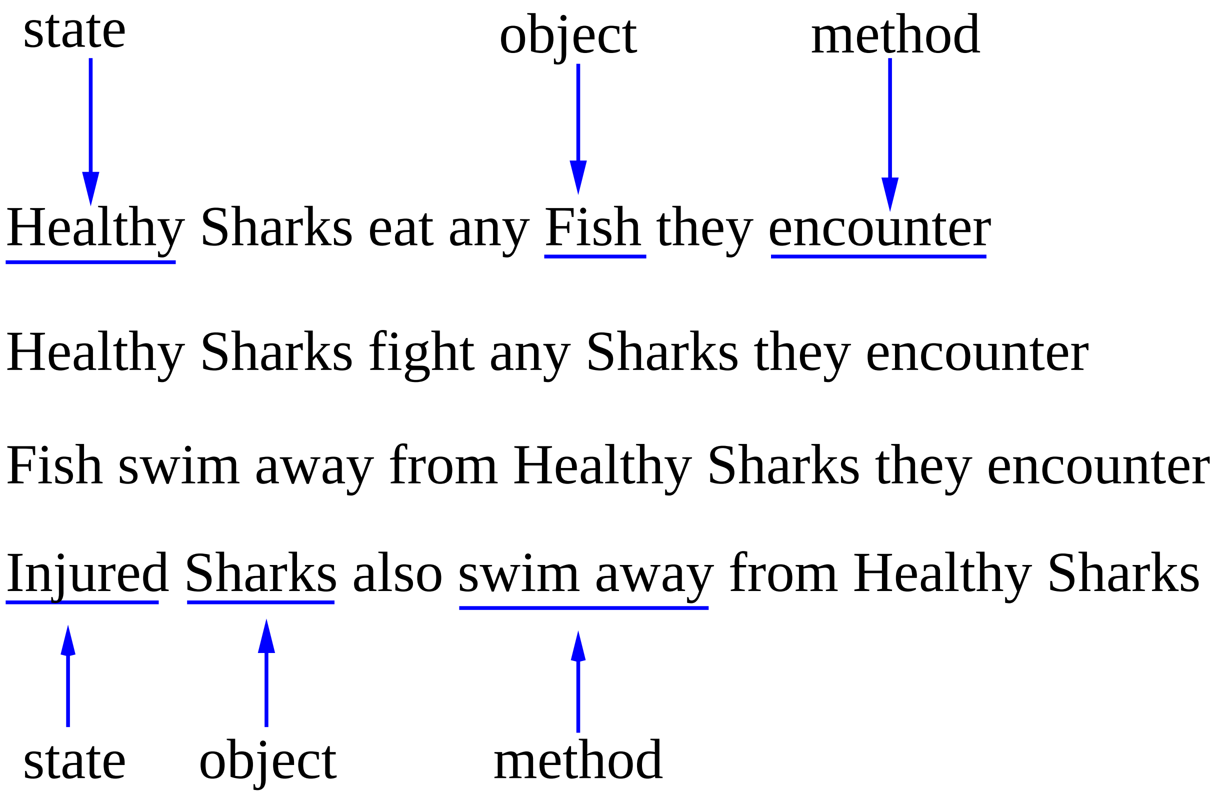
Object-Oriented Programming

- Data as “objects”: state with an identity
- Objects perform abstract “methods” to manipulate their state
- Objects compose with or “inherit” other objects
- Programs are stories about objects instead of recipes about bits

The Object-Oriented Design Space: More Complex Than It Seems



A “Simple” Scenario: Deep Sea Encounters



state object method

Healthy Sharks eat any Fish they encounter

Healthy Sharks fight any Sharks they encounter

Fish swim away from Healthy Sharks they encounter

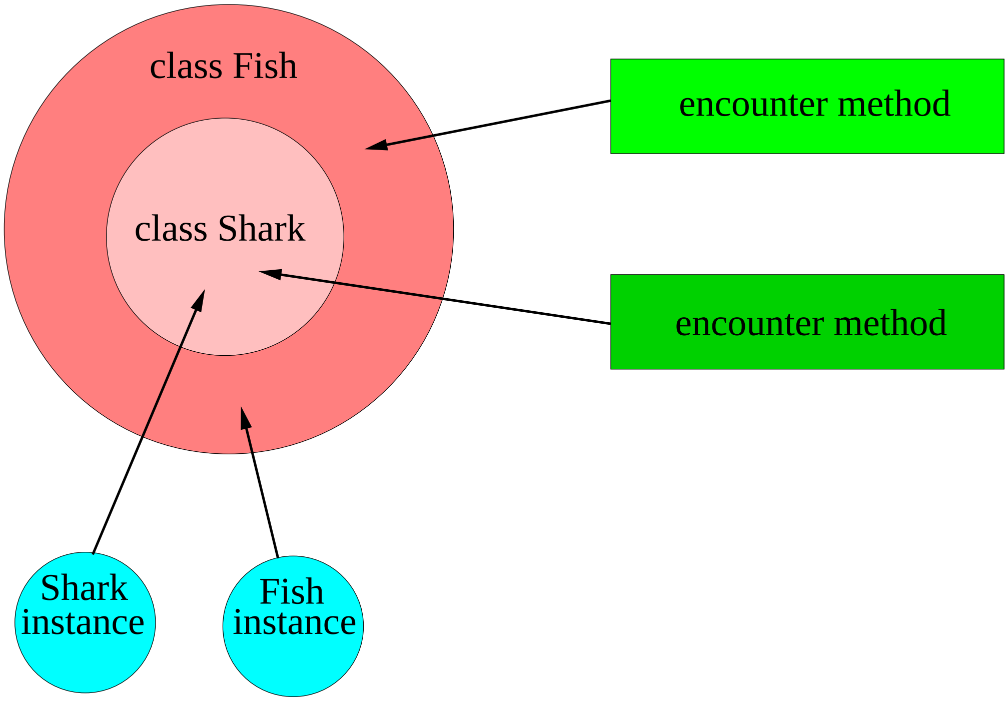
Injured Sharks also swim away from Healthy Sharks

state object method

Mainstream OO Is Not Expressive Enough

```
class: Fish
  method: encounter object
    if object is in class Shark
        and object has state Healthy
     then swim away
class: Shark
  inherit: Fish
  state: Healthy or Injured
  method: fight object
    set state to Injured
  method: encounter object
    if self has state Healthy
     then
       if object is in class Shark
         then fight object
         otherwise
           if object is in class Fish
             then eat object
     otherwise
       if self has state Injured
           and object is in class Shark
           and object has state Healthy
         then swim away
```

A Brittle Program Structure



What Went Wrong?

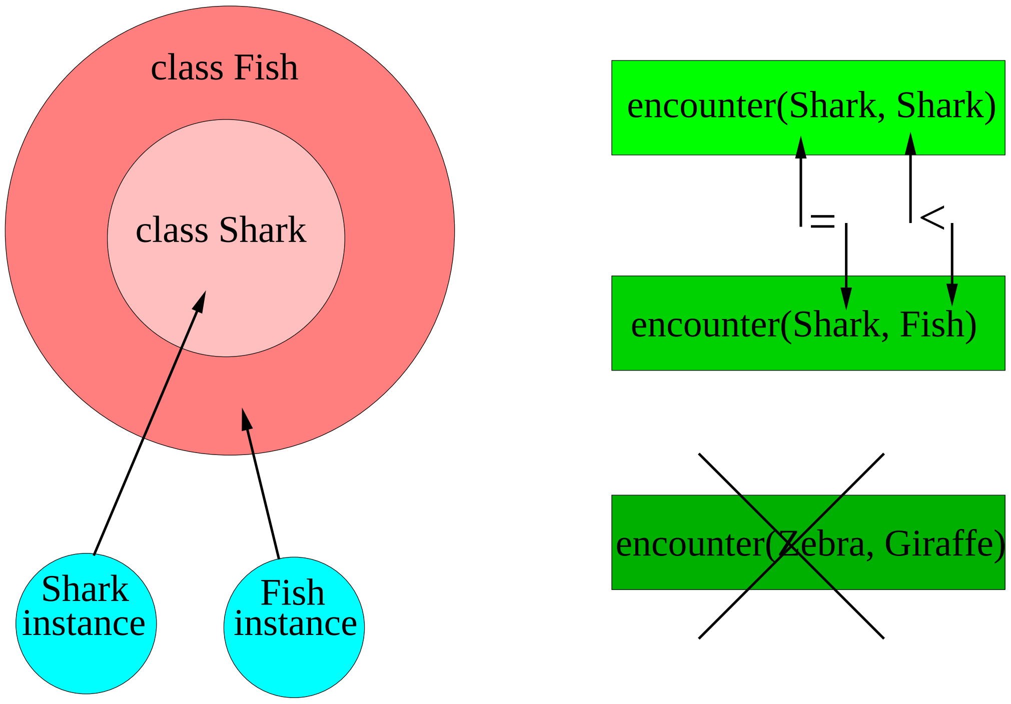
1. The programmer's view is too local... make it global!

2. The programmer's view is too external... internalize it!

Multiple Dispatch: A Global View

```
class: Fish
class: Shark
  inherit: Fish
  state: Injured or Healthy
method: fish:Fish encounter shark:Shark
  if shark has state Healthy
    then fish swim away
method: shark:Shark encounter fish:Fish
  if shark has state Healthy
    then shark eat fish
method: shark:Shark fight other shark:Shark
  set shark state to Injured
method: shark:Shark encounter other shark:Shark
  if shark has state Healthy
    then shark fight other shark
    otherwise
      if shark has state Injured
        then shark swim away
```

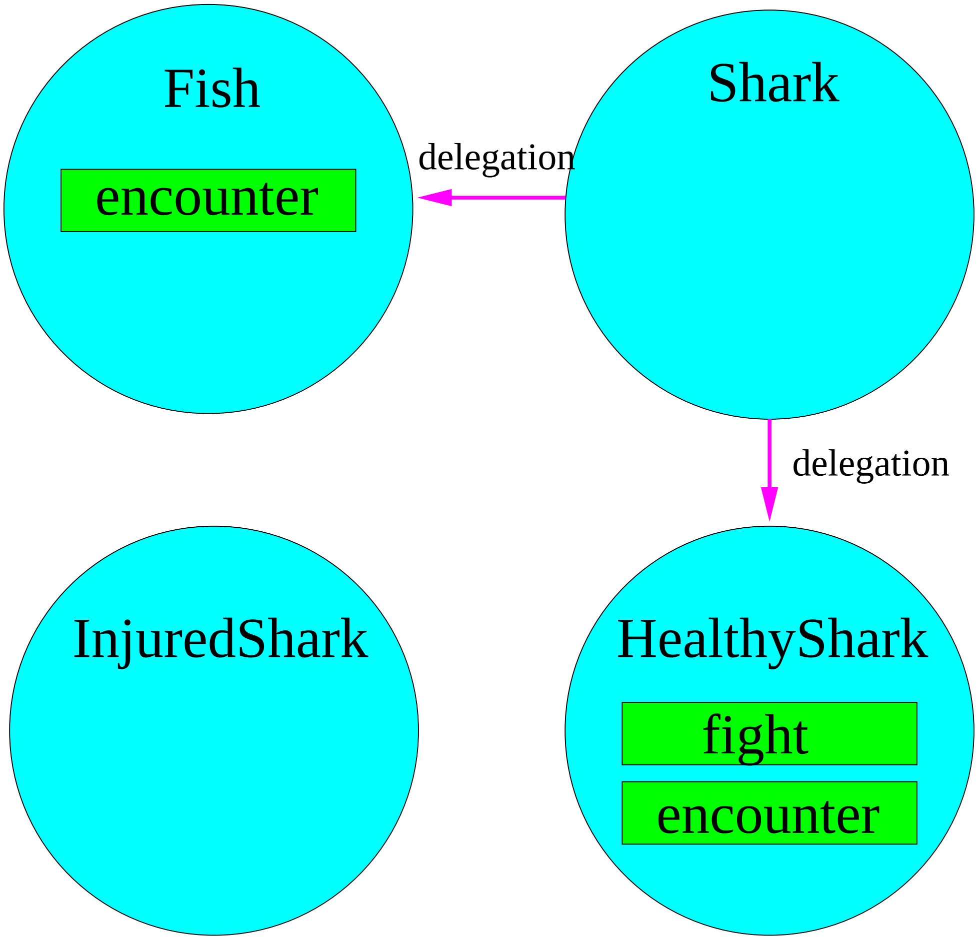
Multiple Dispatch: What Happened?



Prototypes: An Internal View

```
object: Fish
  method: encounter object
    if object same as Shark
       and object delegates to HealthyShark
      then swim away
object: Shark
  delegate to: Fish
object: HealthyShark
  method: fight object
    replace HealthyShark on self with InjuredShark
  method: encounter object
    if object is same as Shark
      then fight object
    otherwise
      if object is same as Fish
        then eat object
object: InjuredShark
```

Prototypes: What Happened?



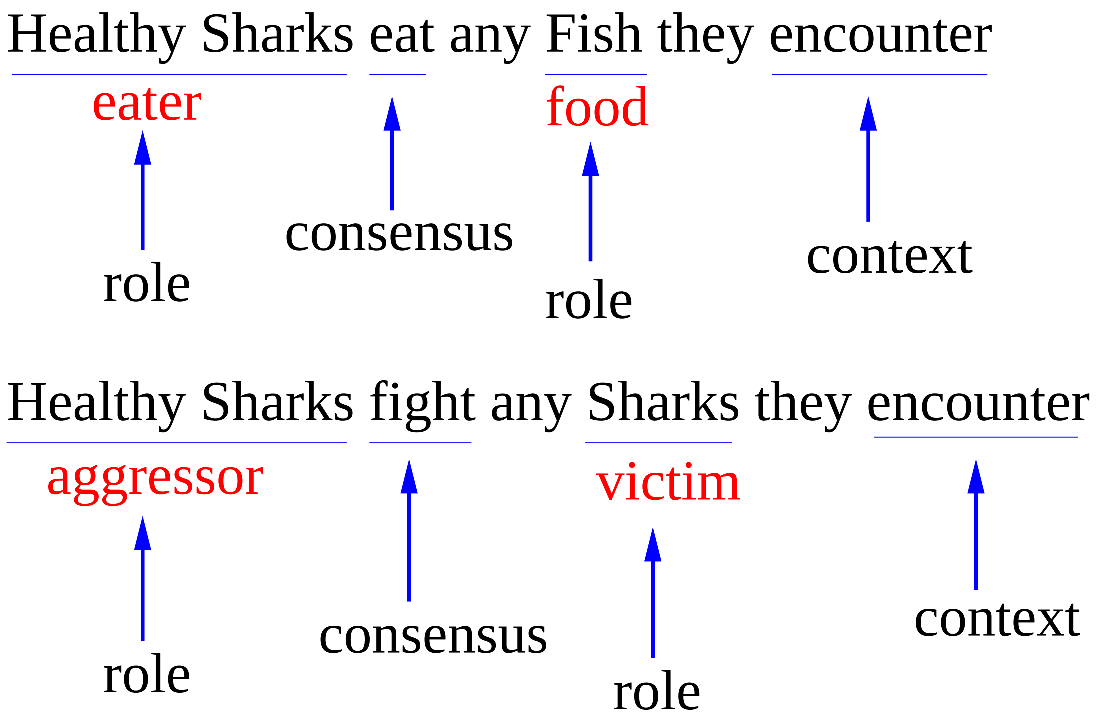
Why Not Combine The Two?

- Multiple Dispatch exploits global knowledge
- Prototypes exploit internalized concepts

But Why Can't We?

- No formal basis for combining them yet
- Multiple Dispatch depends on classes and global order of methods to work
- Prototypes depend on restricted local view for internal representation to work
- Past attempts merely relabel classes as objects and restrict usage to fake it

... Not Quite True: A Different Approach



Healthy Sharks eat any Fish they encounter

eater food

consensus context

role role

Healthy Sharks fight any Sharks they encounter

aggressor victim

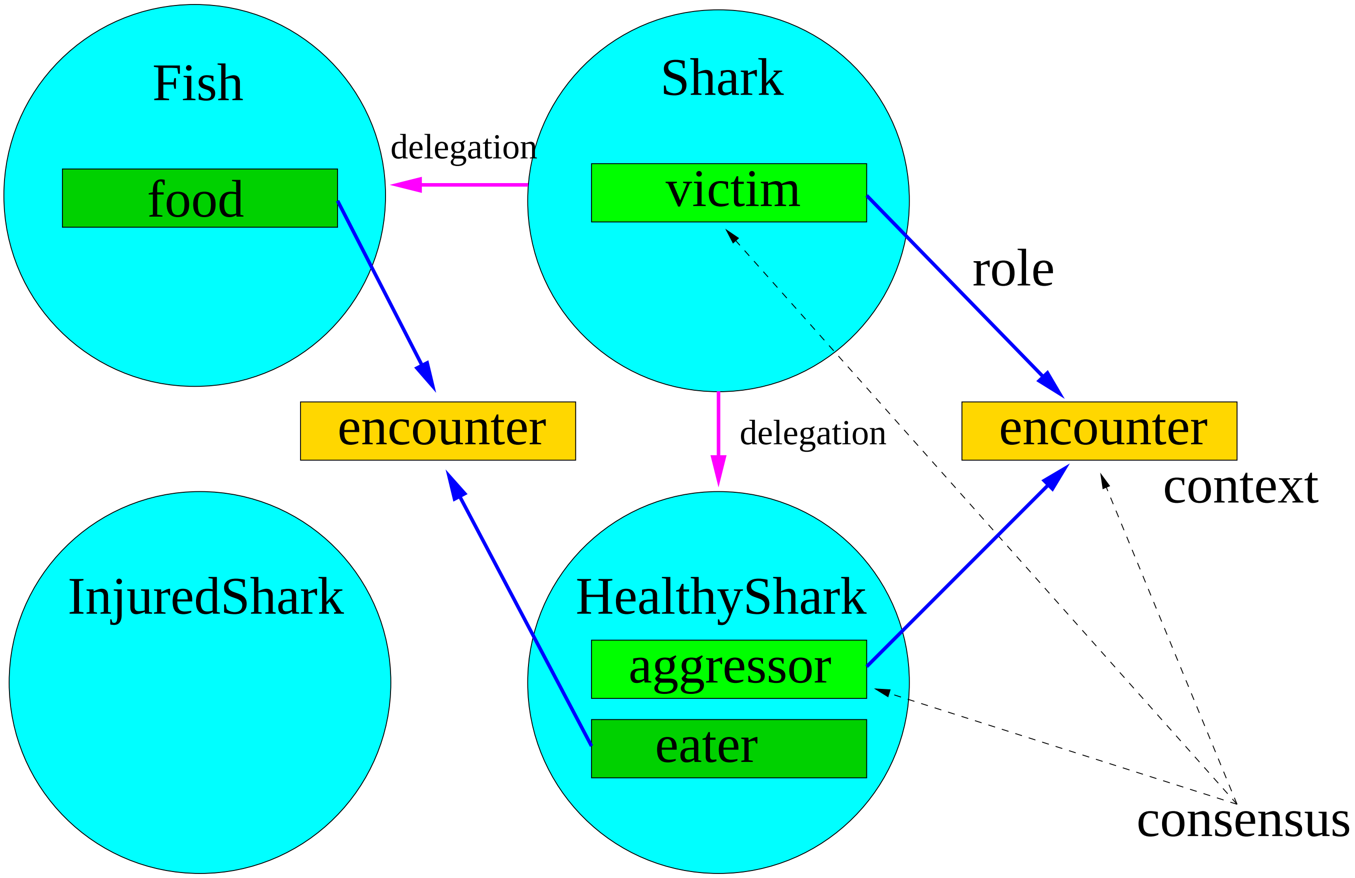
consensus context

role role

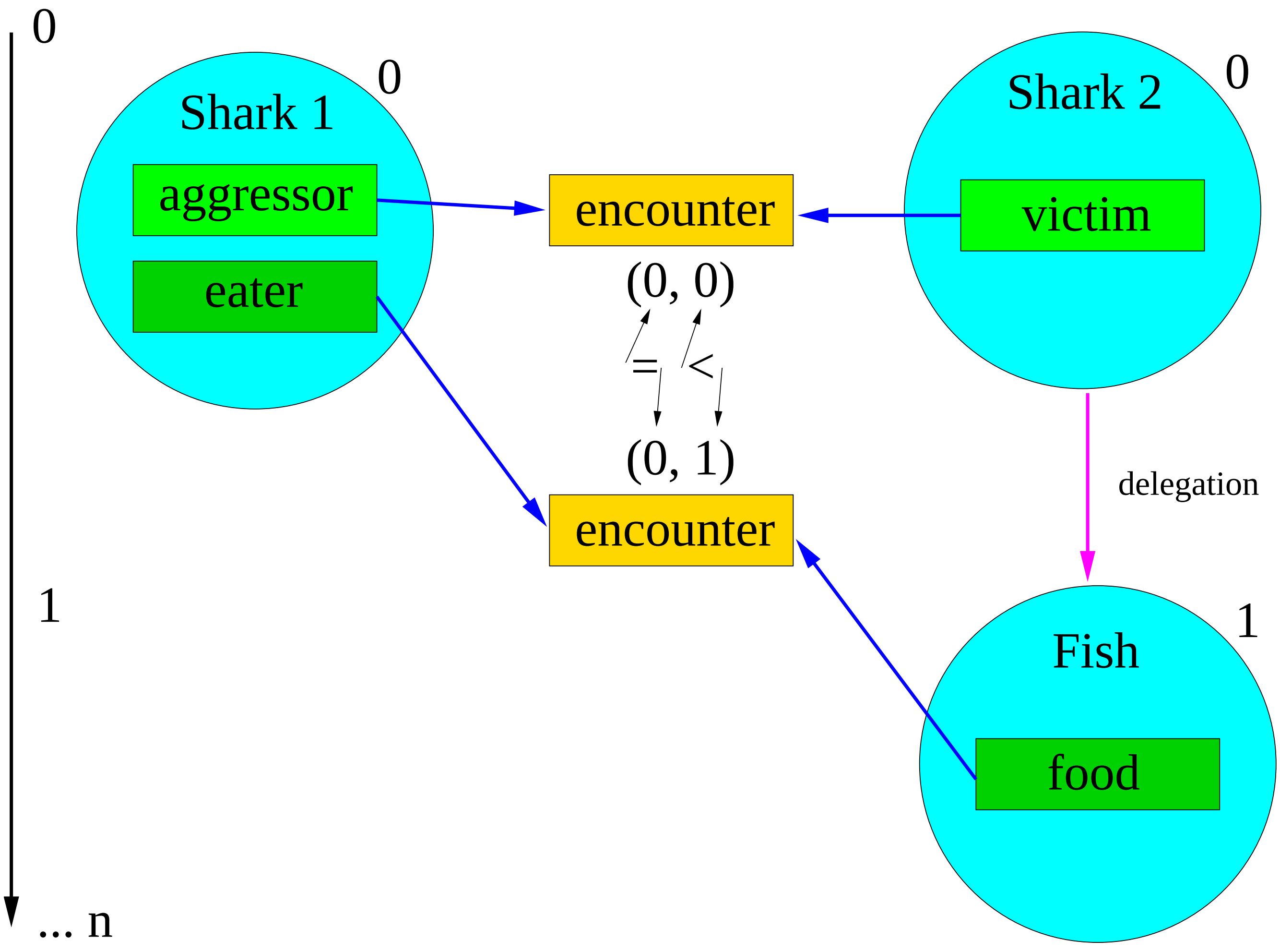
Prototypes with Multiple Dispatch: Roles in Action

```
object: Fish
object: Shark
  delegates to: Fish
object: HealthyShark
object: InjuredShark
method: innocent:Fish encounter threat:HealthyShark
  innocent swim away
method: eater:HealthyShark encounter food:Fish
  eater eat food
method: weaker:HealthyShark fight stronger:Shark
  replace HealthyShark on weaker with InjuredShark
method: aggressor:HealthyShark encounter victim:Shark
  aggressor fight victim
```

How Does It Work?



Resolving Ambiguities: Ordering On The Fly



It Works In Theory

$$\frac{\begin{array}{c} compose(C,\overline{v}) = \left\langle \overline{v'} \right\rangle \quad l \in applicable(S,s,\overline{v'}) \\ \forall_{l' \in applicable(S,s,\overline{v})} \left(l = l' \vee rank(S,l,s,\overline{v'}) \prec rank(S,l',s,\overline{v'}) \right) \end{array}}{lookup(S,C,s,\overline{v}) = l}$$

$$\frac{\forall_{0 \le i \le n} \left(order(S,v_i) = \langle d_0, \cdots, d_m \rangle \wedge \exists_{0 \le \alpha \le m} \left(S[d_\alpha] = < \left\langle \overline{d'} \right\rangle, \{\overline{r}\}, e > \wedge < s,i,l > \in \{\overline{r}\} \right) \right)}{l \in applicable(S,s,v_0,\cdots,v_n)}$$

$$\frac{l \in applicable(S,s,v_0,\cdots,v_n)}{rank(S,l,s,v_0,\cdots,v_n) = \prod_{0 \le i \le n} min \left\{ 0 \le k \le m \middle| \begin{array}{c} order(S,v_i) = \langle d_0, \cdots, d_m \rangle \wedge \\ S\left[d_k\right] = < \left\langle \overline{d'} \right\rangle, \{\overline{r}\}, e > \wedge \\ < s,i,l > \in \{\overline{r}\} \end{array} \right\}}$$

It Works in Practice

- Dispatch algorithm fits on a slide with room to spare

```
dispatch(selector, args, n)
  for each index below n
    position := 0
    push args[index] on ordering stack
    while ordering stack is not empty
      arg := pop ordering stack
      for each role on arg with selector and index
        rank[role's method][index] := position
        if rank[role's method] is fully specified
          if no most specific method
              or rank[role's method] < rank[most specific method]
            most specific method := role's method
      for each delegation on arg
        push delegation on ordering stack
      position := position + 1
  return most specific method
```

- Implemented in the programming language Slate

```
_@True not [False].

_@False not [True].

_@True /\ _@True [True].

_@(Boolean traits) /\ _@(Boolean traits) [False].

_@False \/ _@False [False].

_@(Boolean traits) \/ _@(Boolean traits) [True].
```

Conclusion

- PMD unifies two disparate language paradigms: prototypes and multiple dispatch

- Gives object-oriented programmers new, practical tool to think about and write programs in