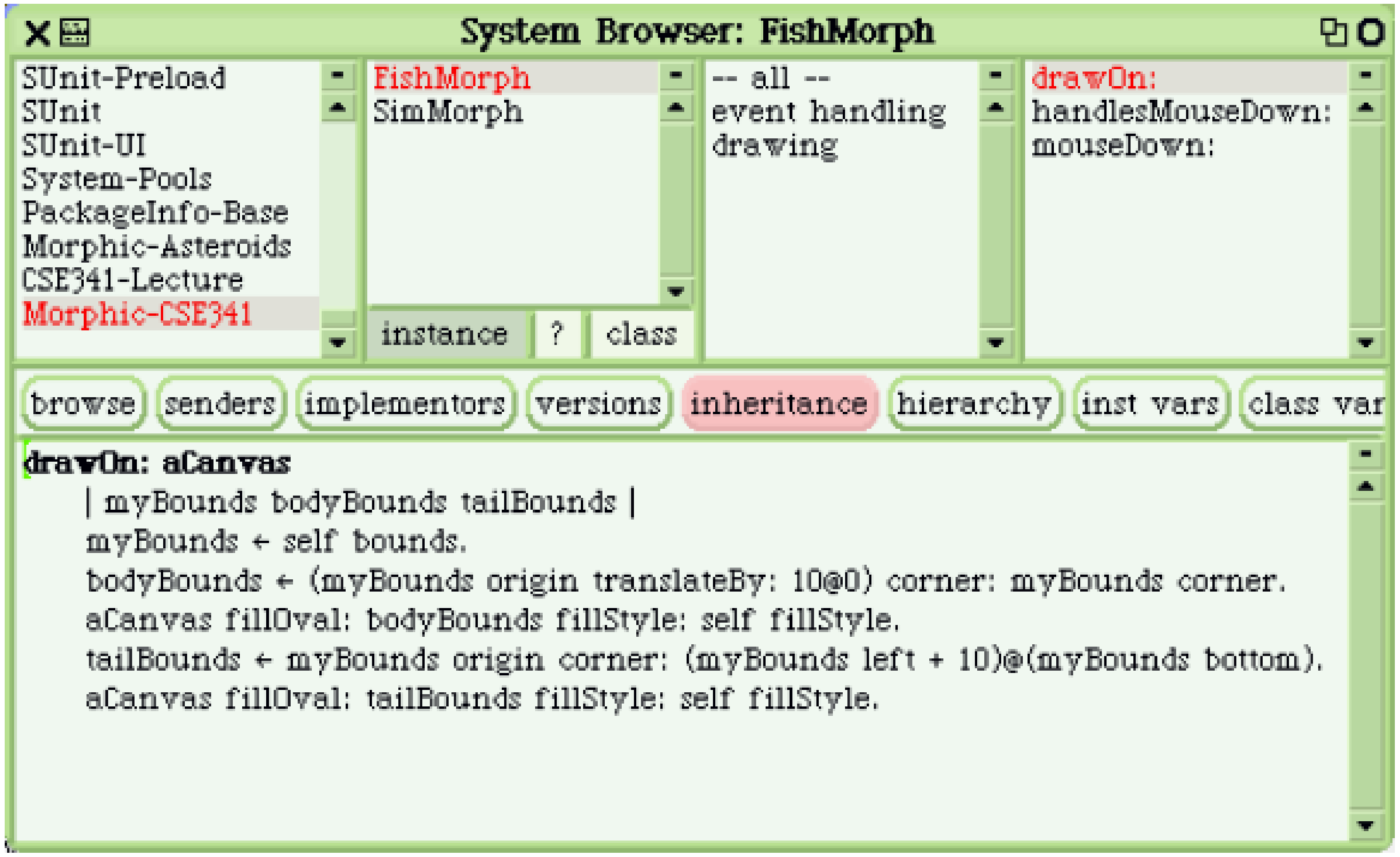
Exploratory programming using Squeak and Morphic

Keunwoo Lee
CSE 341 -- Programming Languages
University of Washington
Dept. of Computer Science and Engineering

Exploratory programming

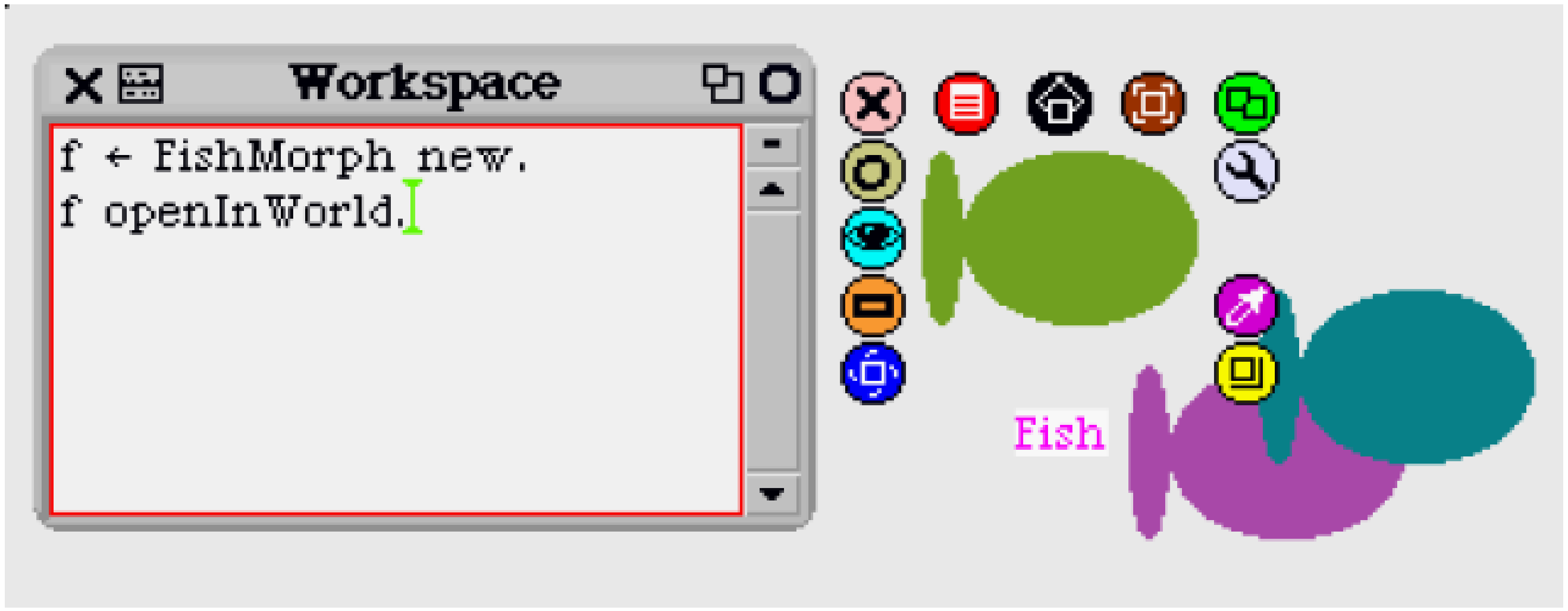
- Programming by "trying stuff out" and seeing what happens
- Slow, cumbersome in edit/compile/run loop
- Easier in read/eval/print loop (fast feedback)
- Squeak & Morphic have even more advanced support for editing code, manipulating objects interactively...
  - **Inspector/explorer**
  - **Selector browser**
  - **Stack trace debugger (for exceptions)**
  - **Tile-based scripting**

Review: FishMorph

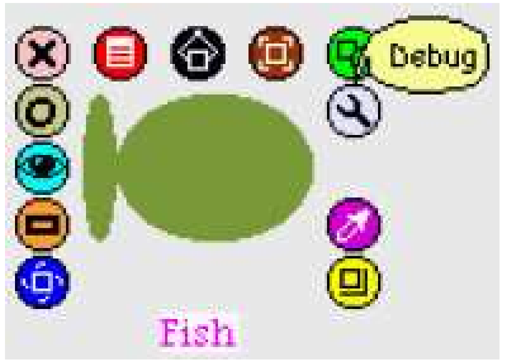


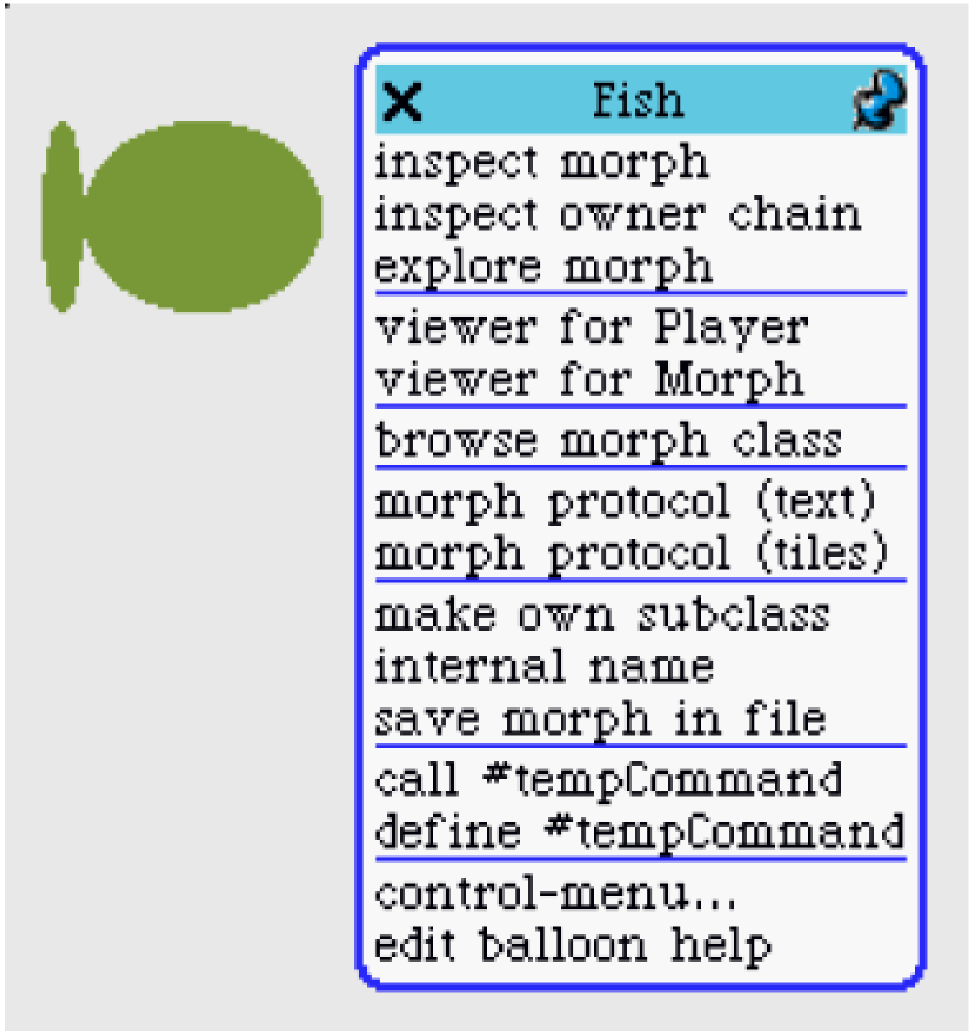
```
drawOn: aCanvas
    | myBounds bodyBounds tailBounds |
    myBounds ← self bounds.
    bodyBounds ← (myBounds origin translateBy: 10@0) corner: myBounds corner.
    aCanvas fillOval: bodyBounds fillStyle: self fillStyle.
    tailBounds ← myBounds origin corner: (myBounds left + 10)@(myBounds bottom).
    aCanvas fillOval: tailBounds fillStyle: self fillStyle.
```

Creating morphs

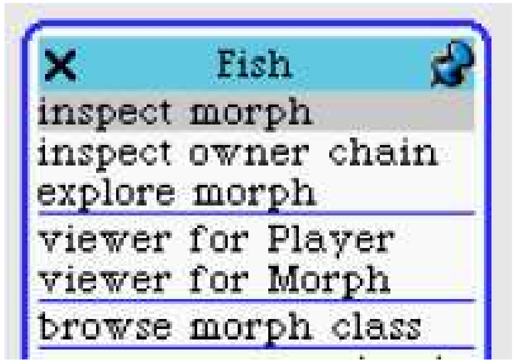

```
f ← FishMorph new.
f openInWorld.
```

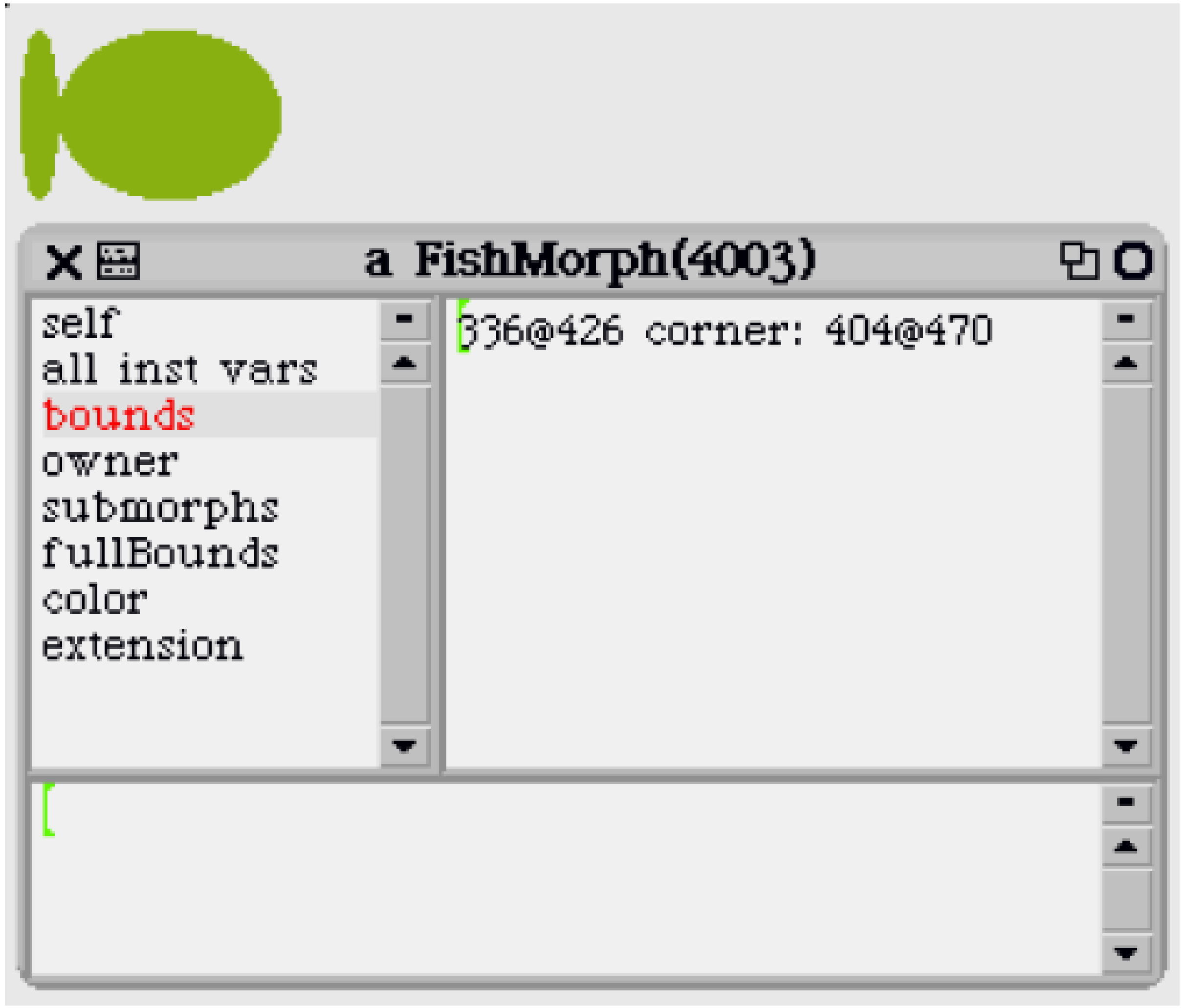
The debug halo menu





The inspector



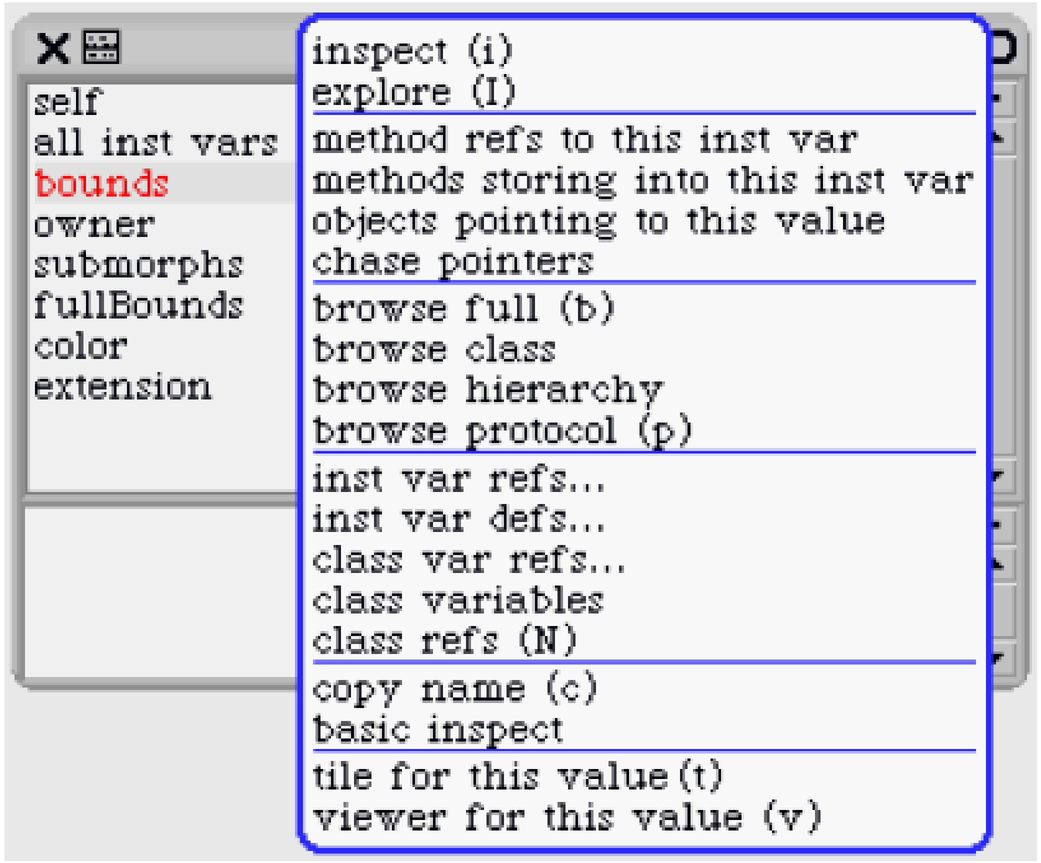


```
336@426 corner: 404@470
```

(Can also open an inspector on any object by sending **inspect** message.)

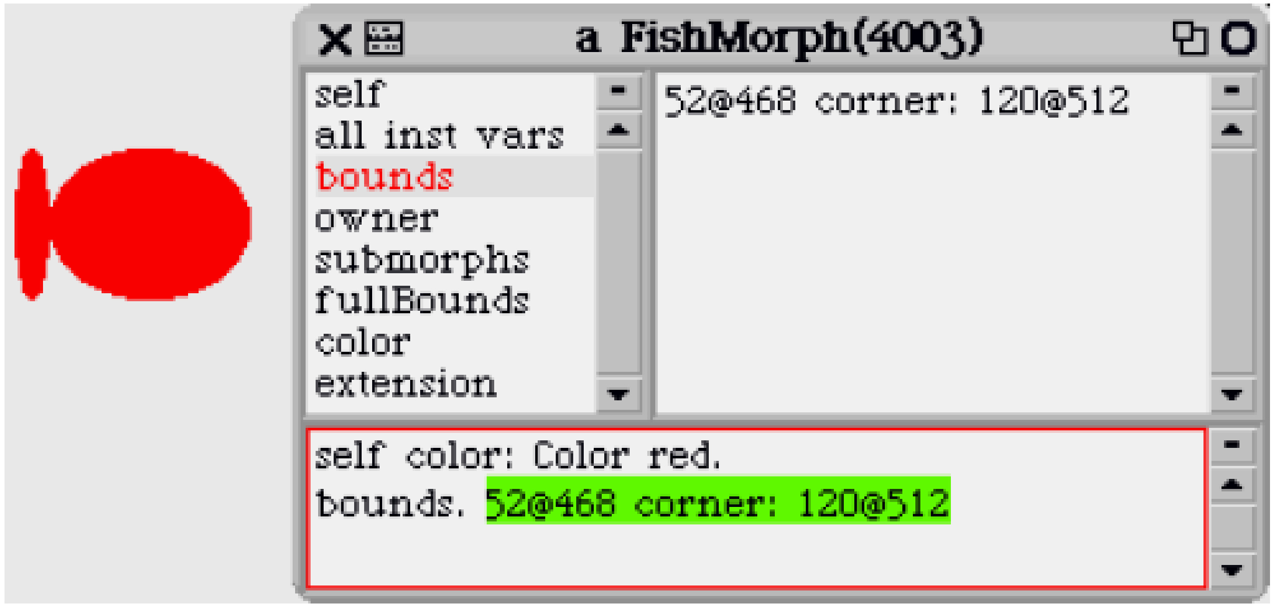
Inspecting instance variables

- Can view, edit values in-place (write expression in value display pane and accept (Alt-s))
- Yellow-click to bring up context menu for instance var



Inspector mini-workspace

- Bottom pane behaves as object-specific workspace
- Workspace's environment is like no-argument method:
  - instance variables accessible
  - self bound to object
  - super begins lookup in superclass of self's class



```
52@468 corner: 120@512
```

```
self color: Color red.
bounds. 52@468 corner: 120@512
```

Inspector: uses

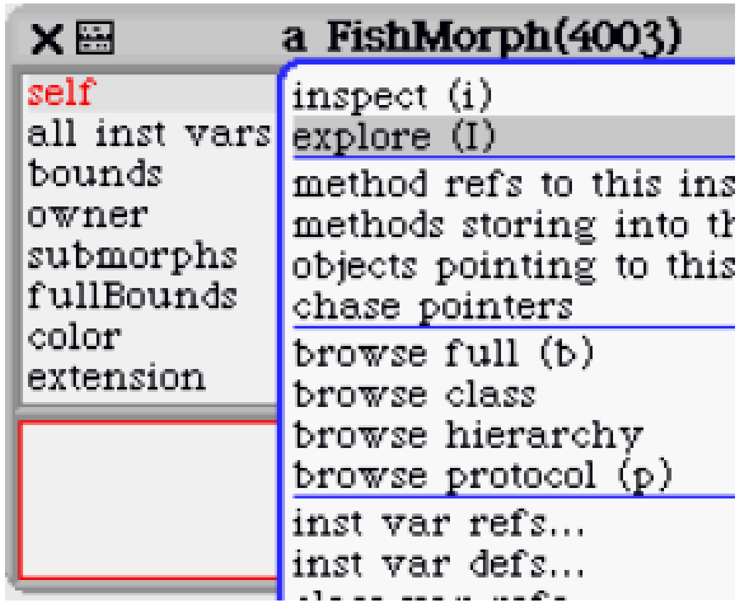
- **Try out code in object workspace; copy into a method when you've got what you want.**
- **Use context menu to explore instance vars**
  - "This value shouldn't be here! How did this get set?"

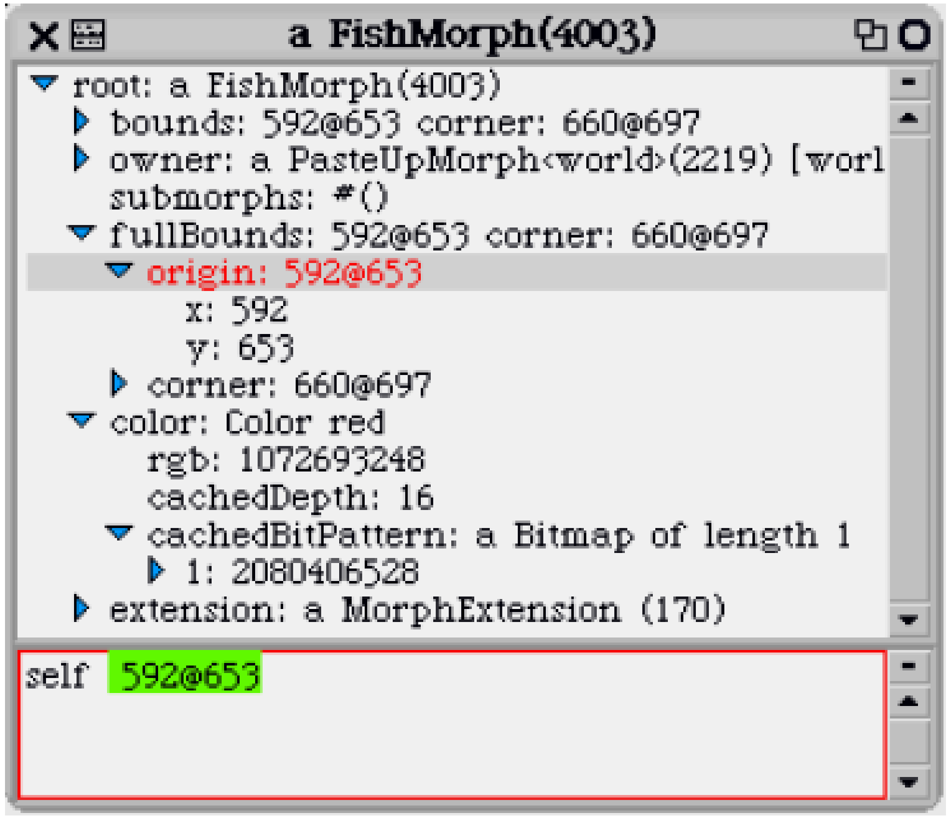
    --> use **methods storing into this inst var**
  - "What is this field's class? What methods does its class handle?"

    --> use **browse class** or **browse hierarchy**

Explorer

- Displays object graph as tree (fields as children)
- Bottom pane is also workspace (for selected item)





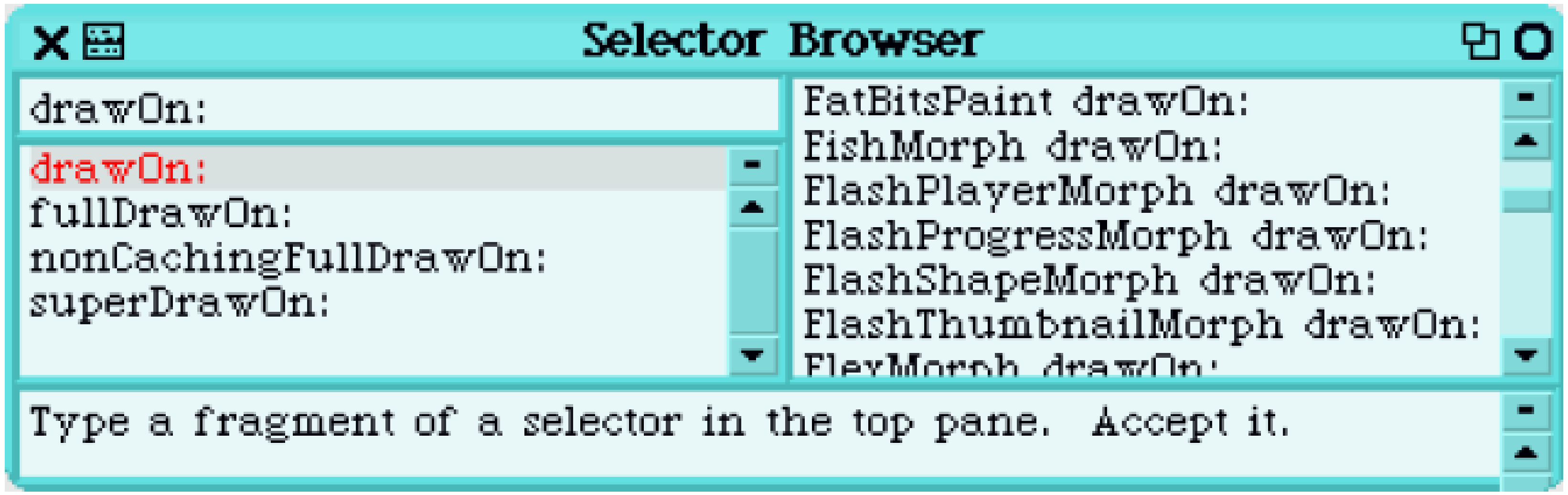
```
a FishMorph(4003)
root: a FishMorph(4003)
  bounds: 592@653 corner: 660@697
  owner: a PasteUpMorph<world>(2219) [worl
  submorphs: #()
  fullBounds: 592@653 corner: 660@697
    origin: 592@653
      x: 592
      y: 653
    corner: 660@697
  color: Color red
    rgb: 1072693248
    cachedDepth: 16
    cachedBitPattern: a Bitmap of length 1
      1: 2080406528
  extension: a MorphExtension (170)

self 592@653
```

Selector browser (method finder)

- **Often want to know who handles a message**

  e.g., when you see a message send and want to know who the receiver might be
- **With selector browser, can search for all implementors of a method**

  (among other things; read docs in bottom pane)



```
drawOn:
```

```
drawOn:
fullDrawOn:
nonCachingFullDrawOn:
superDrawOn:
```

```
FatBitsPaint drawOn:
FishMorph drawOn:
FlashPlayerMorph drawOn:
FlashProgressMorph drawOn:
FlashShapeMorph drawOn:
FlashThumbnailMorph drawOn:
FlexMorph drawOn:
```

Type a fragment of a selector in the top pane. Accept it.

Smalltalk exceptions

- raising: Exception methods **signal, signal:**

  ```
  Exception subclass: #NotFound ...
  ```

- handling: BlockContext method **on:do:**

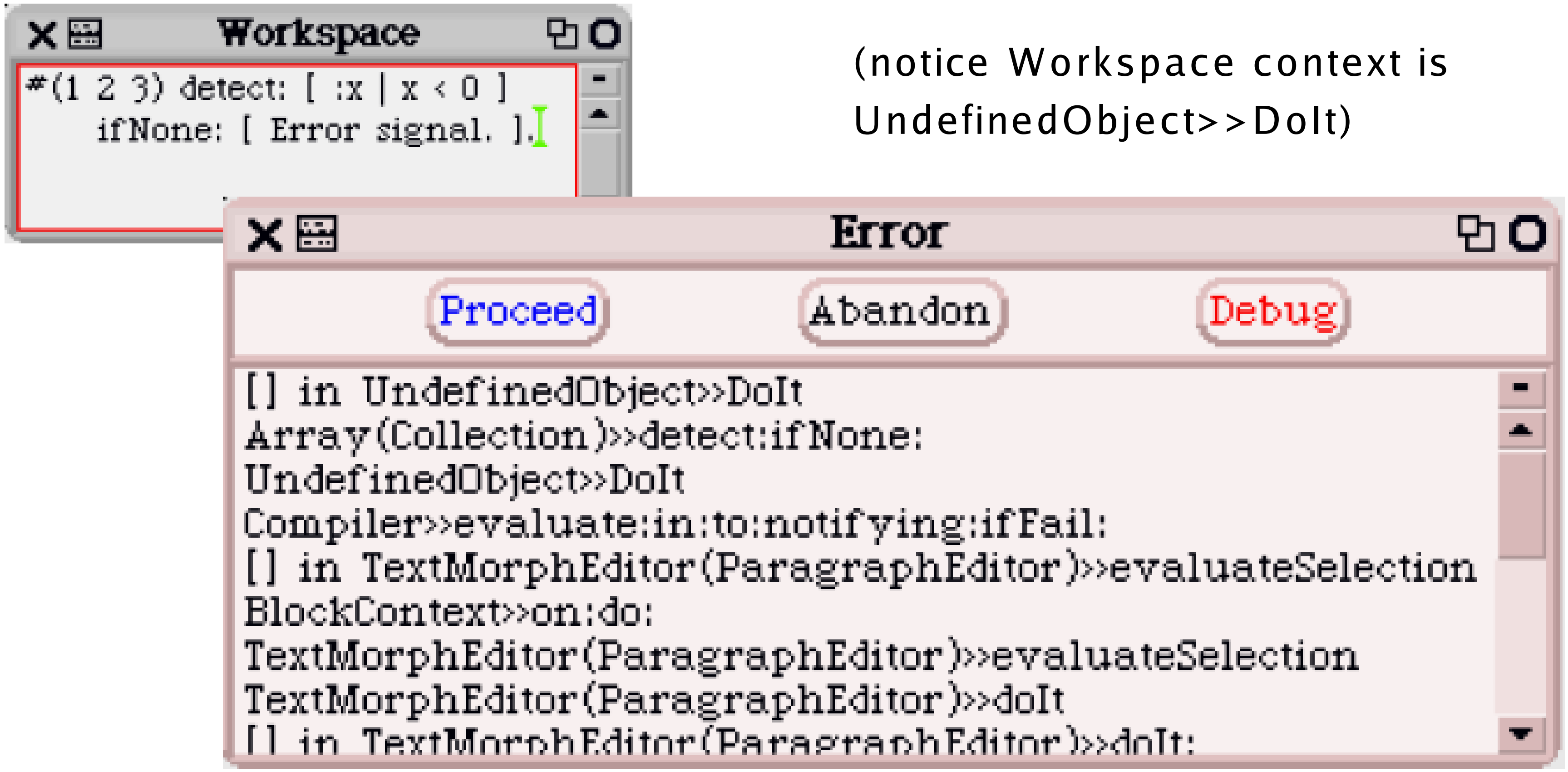
  ```
  [ aTree find: [ :x | x > 0 ]
     ifAbsent: [ NotFound new signal: 'no positives!' ] ]
  on: NotFound
  do: [ :exn |
     Transcript show: 'got NotFound exception'; cr. ].
  ```

Note: Exception defines class methods **signal** and **signal:**; can usually just send signal messages to Exception subclasses directly:

```
NotFound signal: 'no positives!'
```

Stack trace debugger

- Unhandled exceptions propagate to "top level", where the inspector is invoked.



```
#(1 2 3) detect: [ :x | x < 0 ]
    ifNone: [ Error signal. ].
```

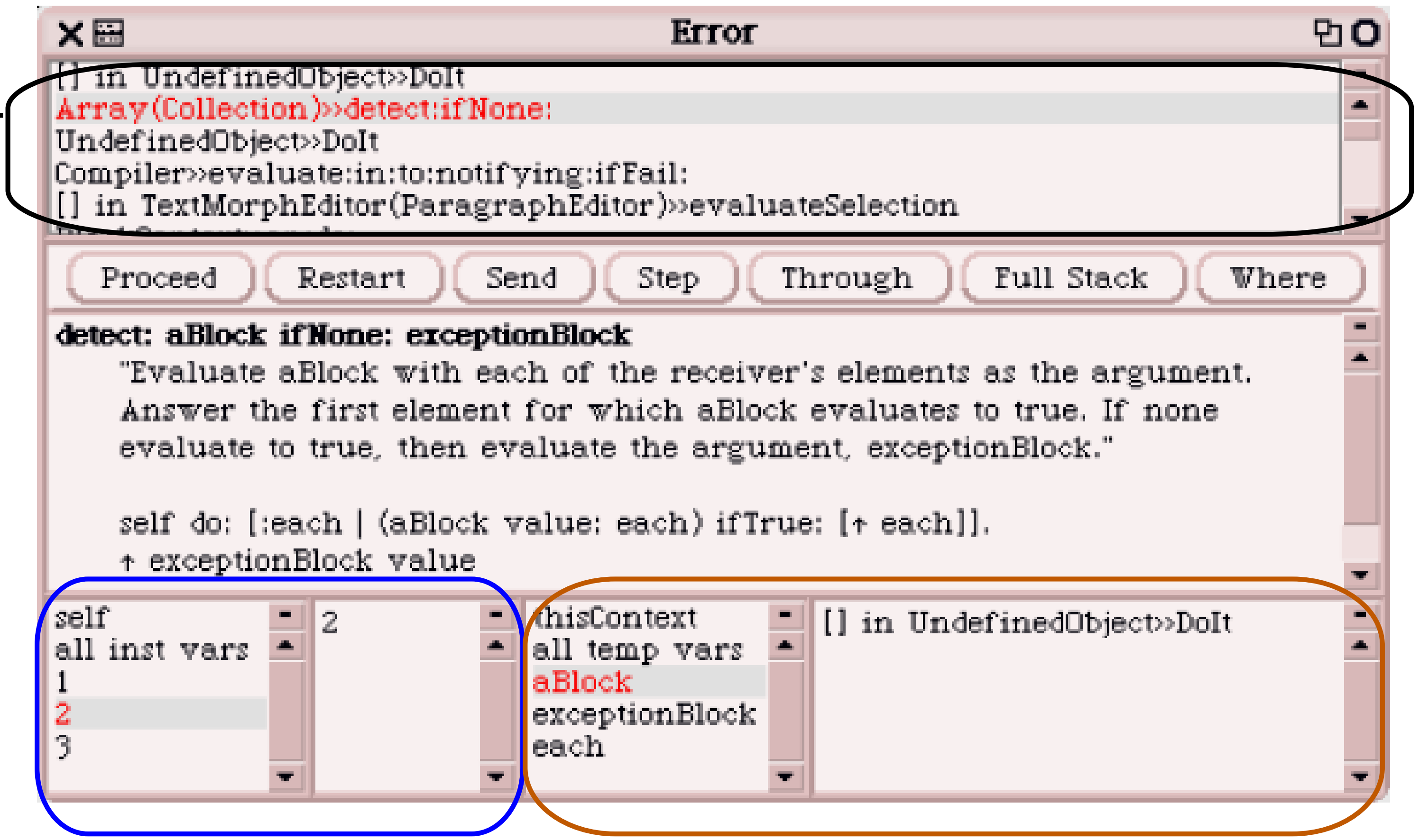
```
[] in UndefinedObject>>DoIt
Array(Collection)>>detect:ifNone:
UndefinedObject>>DoIt
Compiler>>evaluate:in:to:notifying:ifFail:
[] in TextMorphEditor(ParagraphEditor)>>evaluateSelection
BlockContext>>on:do:
TextMorphEditor(ParagraphEditor)>>evaluateSelection
TextMorphEditor(ParagraphEditor)>>doIt
[] in TextMorphEditor(ParagraphEditor)>>doIt:
```

(notice Workspace context is UndefinedObject>>DoIt)

Inspecting the stack

execution stack prior to signal

(selected stack frame in red)



Error

```
[] in UndefinedObject>>DoIt
Array(Collection)>>detect:ifNone:
UndefinedObject>>DoIt
Compiler>>evaluate:in:to:notifying:ifFail:
[] in TextMorphEditor(ParagraphEditor)>>evaluateSelection
```

```
detect: aBlock ifNone: exceptionBlock
	"Evaluate aBlock with each of the receiver's elements as the argument.
	Answer the first element for which aBlock evaluates to true. If none
	evaluate to true, then evaluate the argument, exceptionBlock."

	self do: [:each | (aBlock value: each) ifTrue: [↑ each]].
	↑ exceptionBlock value
```

```
self
all inst vars
1
2
3
```

```
2
```

```
thisContext
all temp vars
aBlock
exceptionBlock
each
```

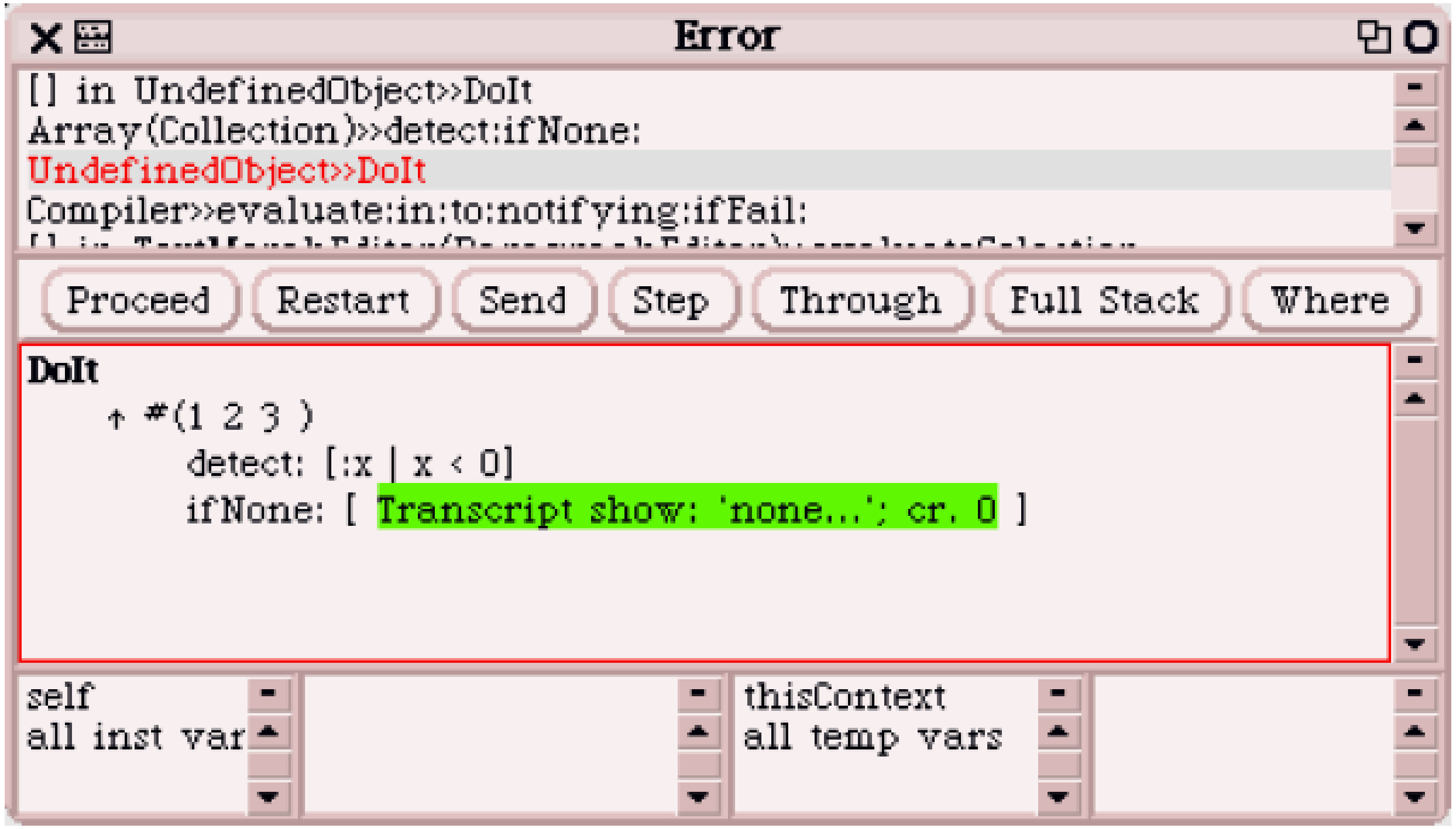
```
[] in UndefinedObject>>DoIt
```

receiver and instance variables

local variables in current context

Interactive stack debugging

- Can edit code of methods directly in debugger
- Use "accept" (Alt-s) to save changes
- Can restart message send, step through evaluation

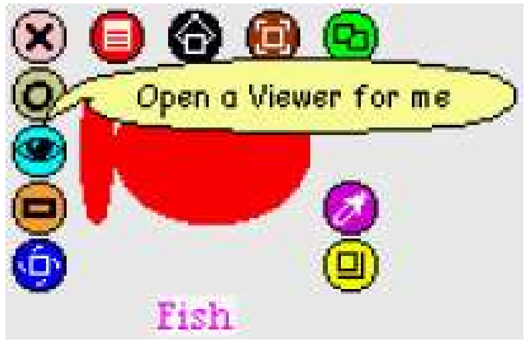


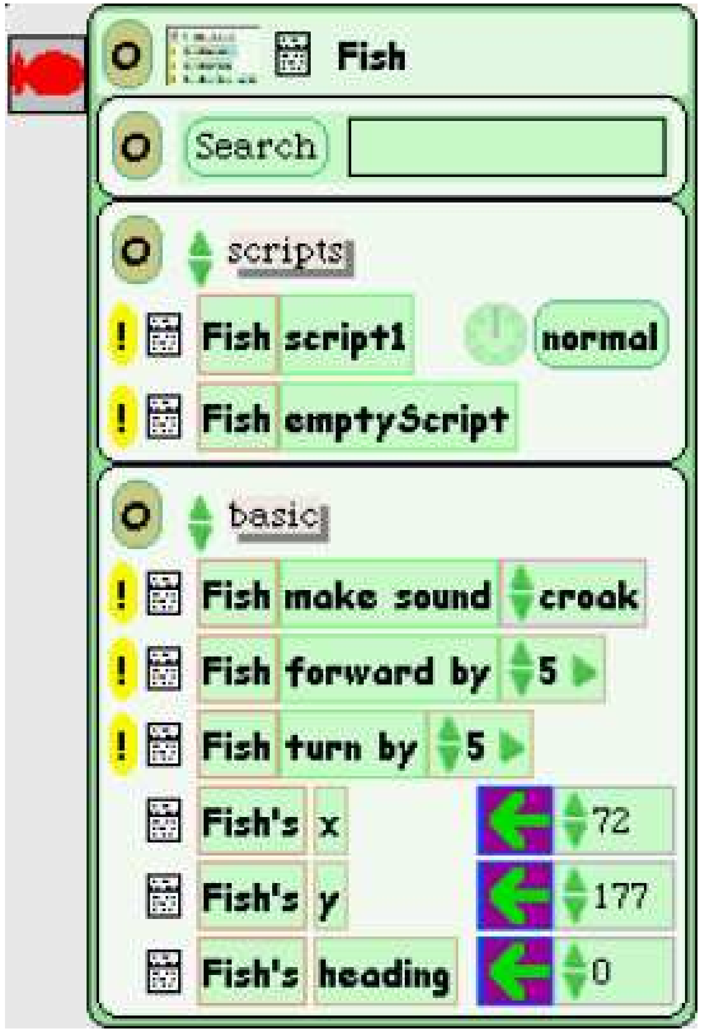
```
[] in UndefinedObject>>DoIt
Array(Collection)>>detect:ifNone:
UndefinedObject>>DoIt
Compiler>>evaluate:in:to:notifying:ifFail:
```

```
DoIt
    ↑ #(1 2 3 )
        detect: [:x | x < 0]
        ifNone: [ Transcript show: 'none...'; cr. 0 ]
```

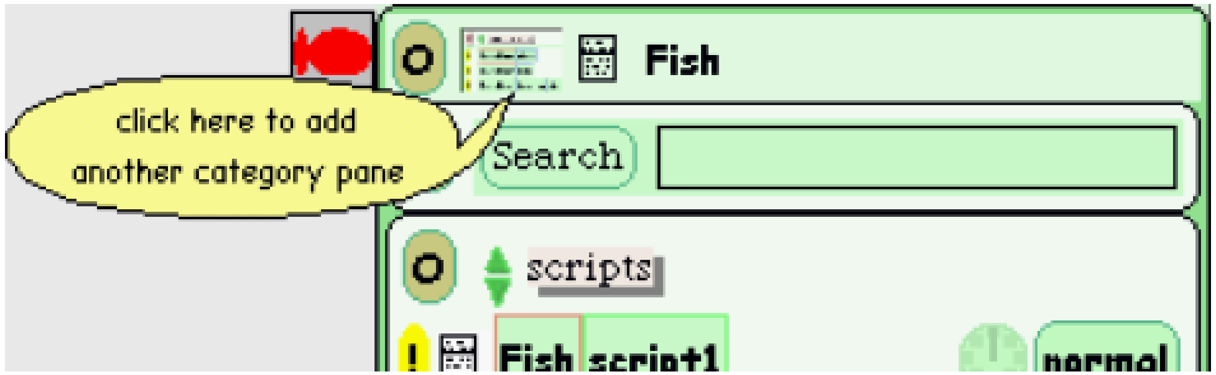
Tile-based scripting

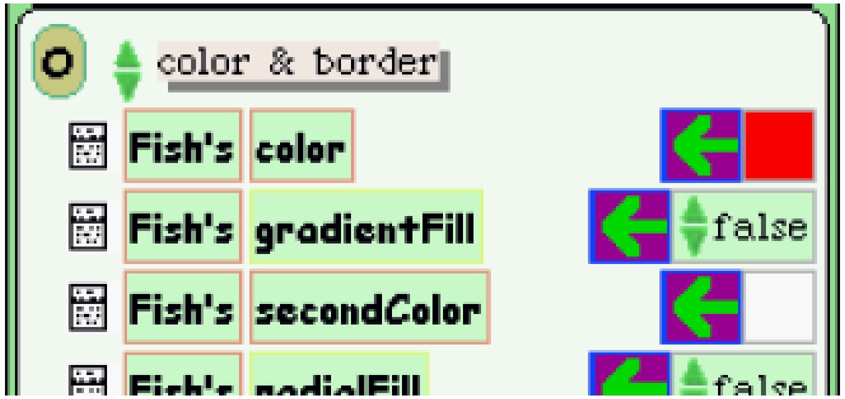
- **Tiles:** graphical representations of Squeak objects and code
- Use **Viewer** halo to obtain tile scripting interface for a morph





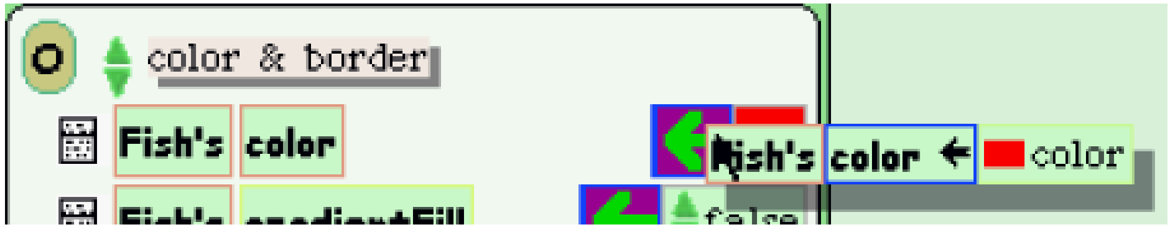
Script categories



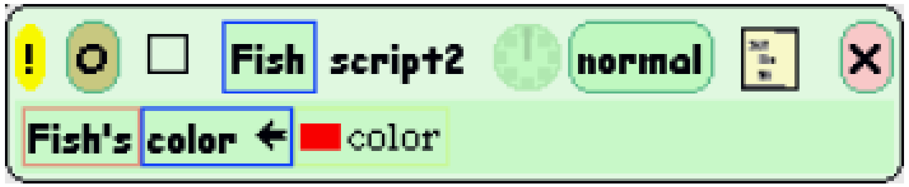


Making scripts

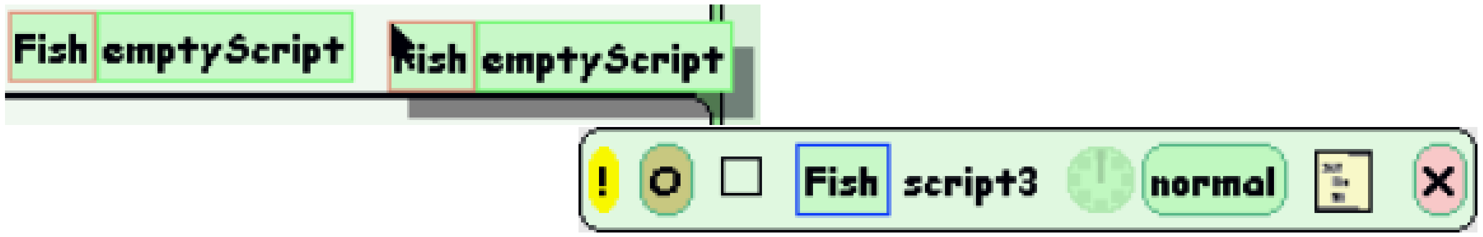
- Drag from script tile to start a script:



- Dropping onto desktop makes standalone script:

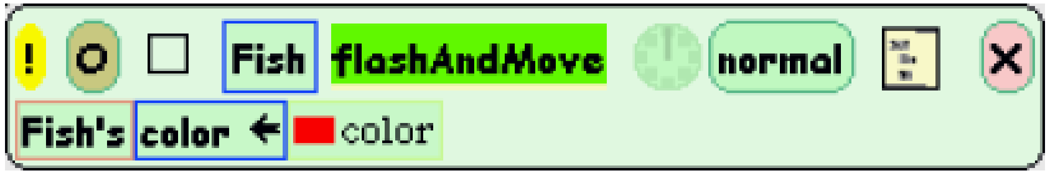


- Drag from empty script to start with no code:

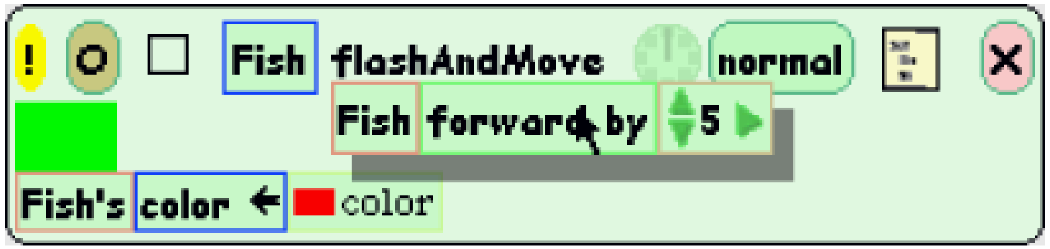


Editing scripts

- To change name, click on title

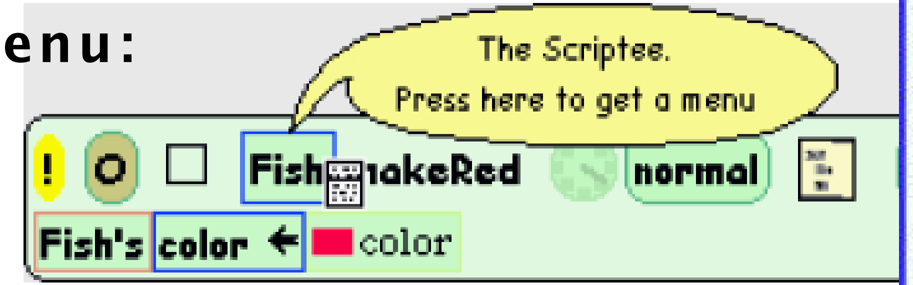


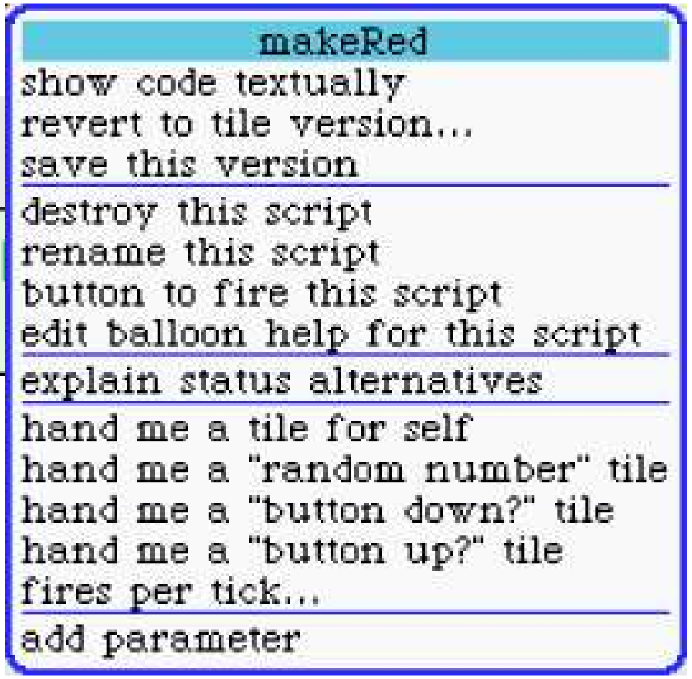
- To add lines, drop more tiles onto script



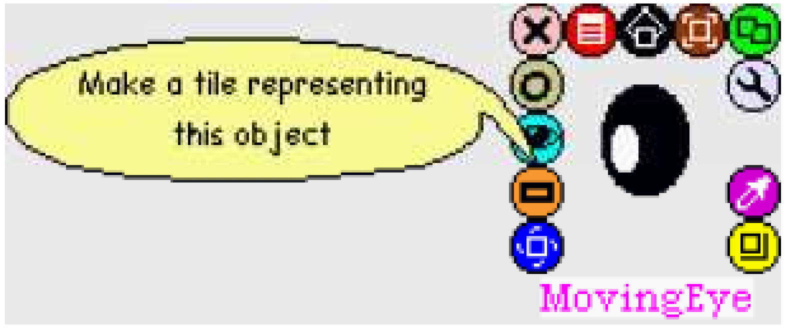
Editing scripts, ct'd.

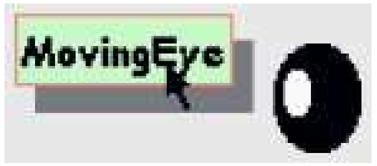
- Click on morph name to get menu:



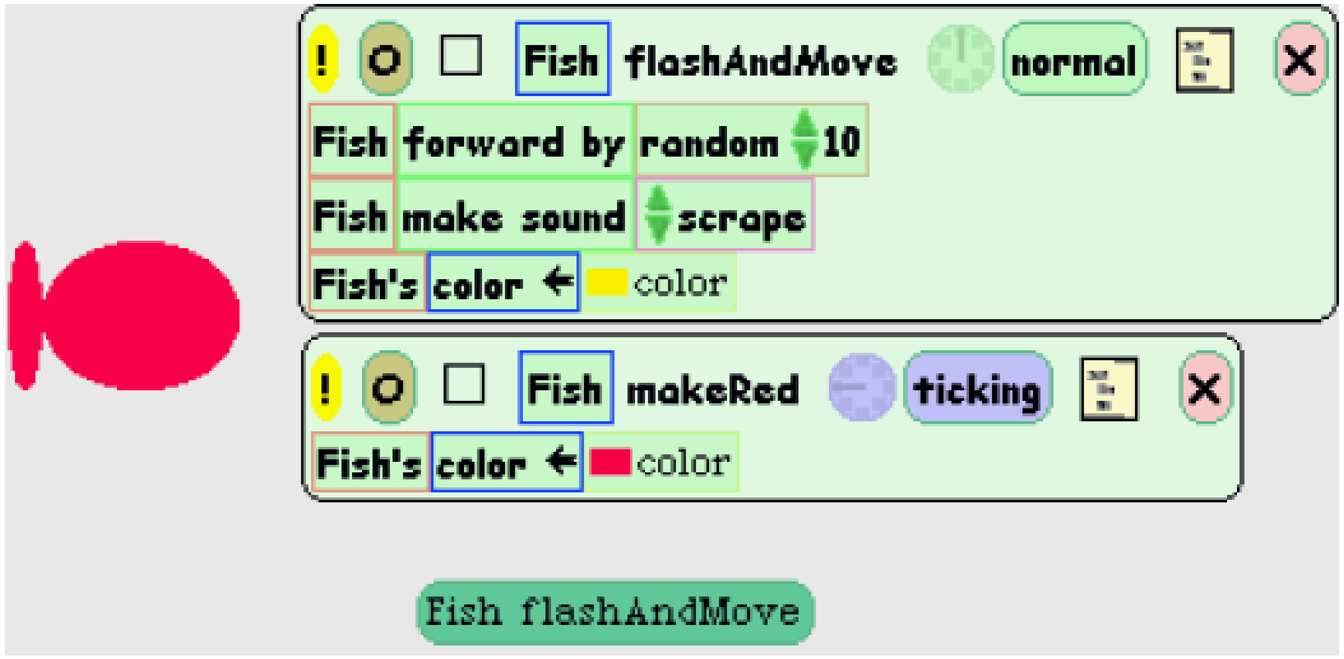

makeRed
show code textually
revert to tile version...
save this version
destroy this script
rename this script
button to fire this script
edit balloon help for this script
explain status alternatives
hand me a tile for self
hand me a "random number" tile
hand me a "button down?" tile
hand me a "button up?" tile
fires per tick...
add parameter

- Can get tiles from here, or many other places (e.g., Tile halo of other Morph)





Assembled scripts

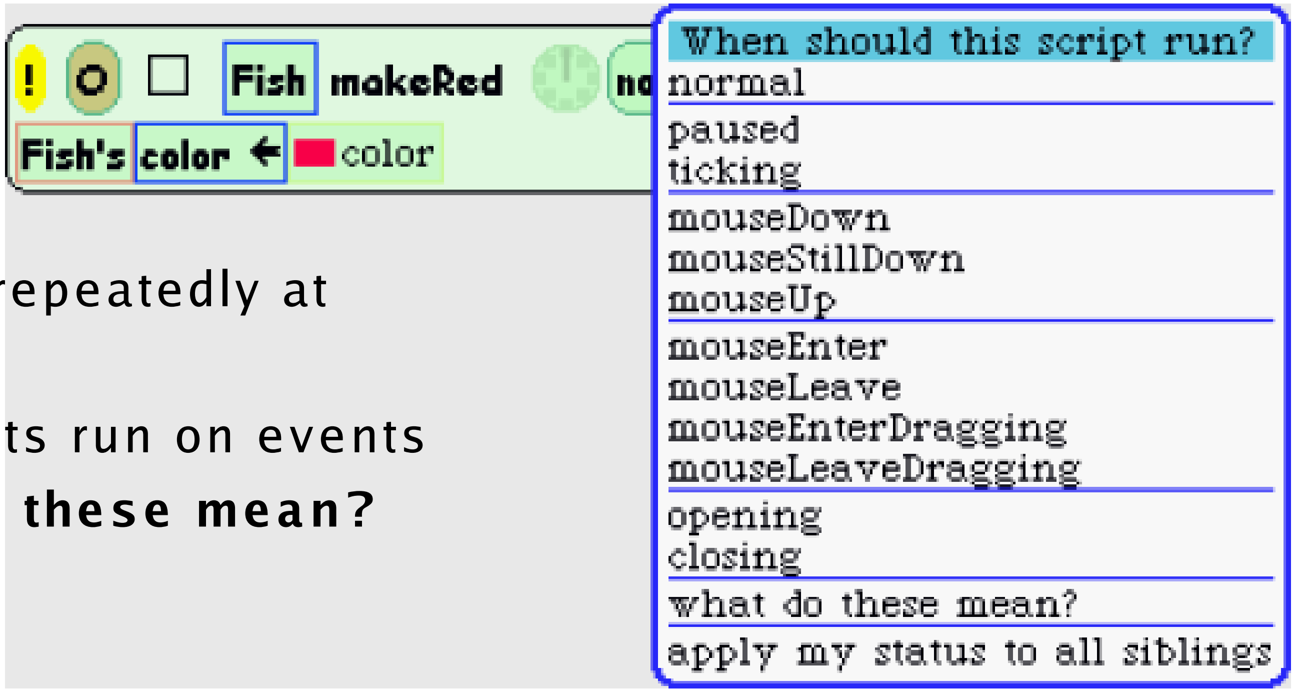


```
! ⭘ ☐ Fish flashAndMove normal ✕
Fish forward by random 10
Fish make sound scrape
Fish's color ← color

! ⭘ ☐ Fish makeRed ticking ✕
Fish's color ← color
```

(button made with **button to fire this script** selection of script menu)

Running scripts

- **To execute, click exclamation point button**
- **Can set to run on different events --- click button next to clock to edit:**

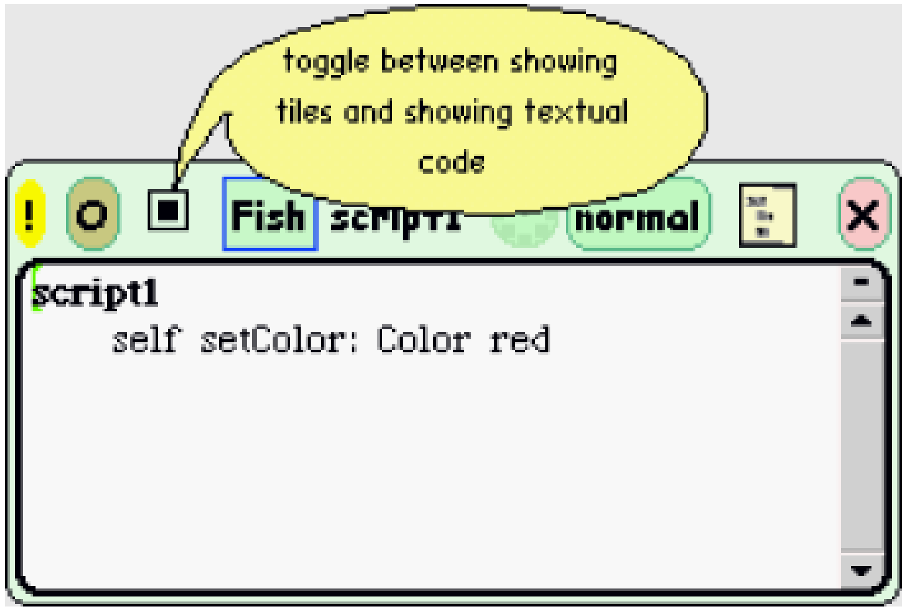


When should this script run?
normal
paused
ticking
mouseDown
mouseStillDown
mouseUp
mouseEnter
mouseLeave
mouseEnterDragging
mouseLeaveDragging
opening
closing
what do these mean?
apply my status to all siblings

- **ticking** runs repeatedly at clock ticks
- **mouse*** events run on events
- click **what do these mean?** for more info

Textual script editing

- Can always "drop down" into text-based code for Morphic scripts:



```
script1
	self setColor: Color red
```

- Useful for more sophisticated coding
- Also, can copy & paste script into method once script is debugged & mature

Conclusion

- emacs, Eclipse, and Visual Studio are not the last word in programming environments
- Demand more!
- You can build your own "inspector"-like programs for exploring objects in other languages/environments

  e.g.:
  - XML-RPC is recently developed protocol for objects communicating over network
  - Easy to build an XML-RPC inspector so you can interactively send messages, receive replies