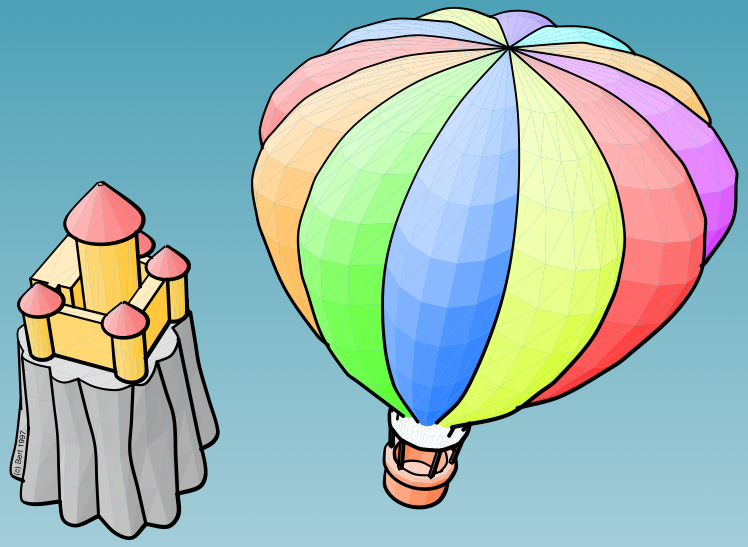


Smalltalk - Pure OOP

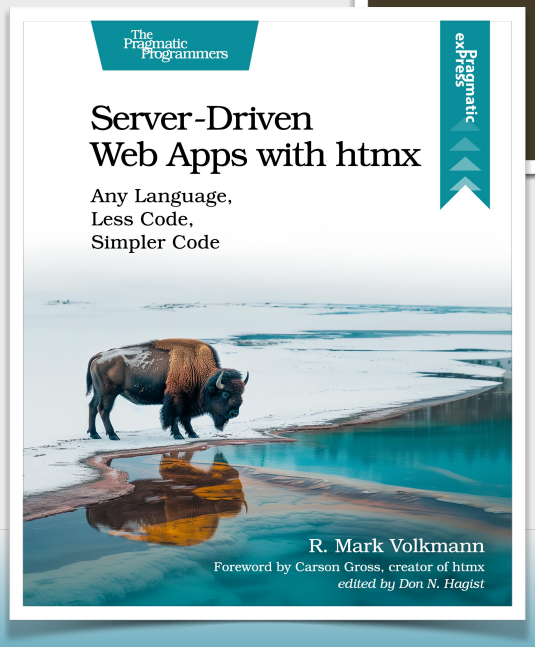
R. Mark Volkmann
Object Computing, Inc.
https://objectcomputing.com
mark@objectcomputing.com
@mark_volkmann

OBJECT COMPUTING
YOUR OUTCOMES ENGINEERED

Slides at **https://github.com/mvolkmann/talks/**

About Me

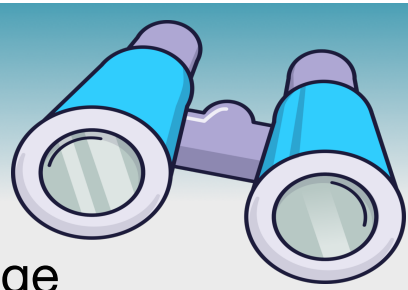
- Partner and Distinguished Software Engineer at Object Computing, Inc. in St. Louis, Missouri USA
- 44 years of professional software development experience
- Writer and speaker
- Blog at https://mvolkmann.github.io/blog/
- Author of Manning book “Svelte ... in Action”
- Author of Pragmatic Bookshelf book “Server-Driven Web Apps with htmx”



Why Learn Smalltalk?

- Beautifully minimal syntax
- Excellent development environment
- Gain understanding of pros and cons compared to other languages
- Get ideas for features that can be added to other languages and their development environments
- Actually use it as an alternative to other languages

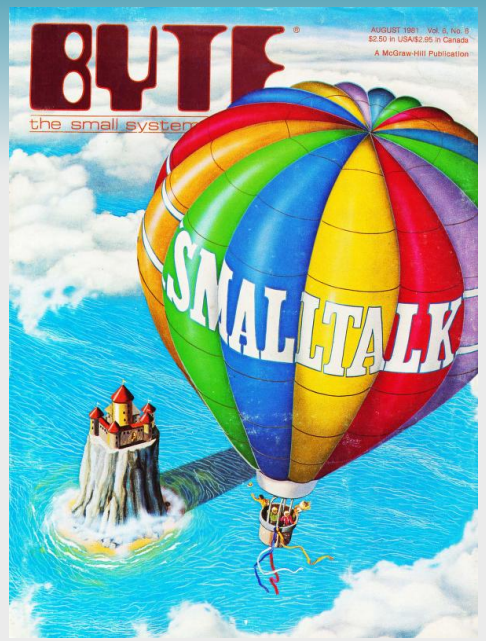
Smalltalk Overview



- Purely object-oriented, dynamically typed programming language
  - first popular OOP
  - everything is represented by an object that is an instance of some class
  - includes classes themselves and all development environment GUI elements
- Everything happens by sending messages to objects
  - objects decide whether and how to act on messages
- Duck typing
  - types of variables, method parameters, and method return types are never specified
  - use any object as long as it responds to all messages sent to it
    - determined at run-time (late binding)

Origin

- Invented at Xerox Palo Alto Research Center (PARC)
  - by Alan Kay, Dan Ingall's, Adele Goldberg, and others in 1970's
- First version in 1972
- Gained popularity in 1981 due to Byte magazine cover
- Was popular alternative to C++ in early 1990's
- But there were no free, open source implementations and licenses for commercials versions were expensive
- The free Java language was released in 1995 and the wind was removed from Smalltalk sails



Current Implementations

- **Free, Open-Source**
  - Squeak (1996) - fork of Smalltalk-80
  - Pharo (2008) - fork of Squeak
  - Cuis (2009) - fork of Squeak

Pharo and Cuis use Squeak VM

- **Commercial**
  - Cincom Smalltalk - VisualWorks and ObjectStudio
  - Instantiations VA Smalltalk - VAST Platform
  - GemTalk Systems - GemStone/S
  - Dolphin Smalltalk

Notable ways in which Cuis differs from Squeak and Pharo:

- ships with minimal set of Smalltalk-80-inspired classes for simplicity
- supports Unicode
- supports TrueType fonts
- suports high-quality vector graphics
- supports SVG

Just-in-Time Compilation

- Not interpreted
- First programming language to use just-in-time (JIT) compilation
- Code is compiled to optimized bytecode
  that is executed by a virtual machine (VM)
- Compilation occurs when method code is saved
- Results in better performance than interpreting code

VMs and Images

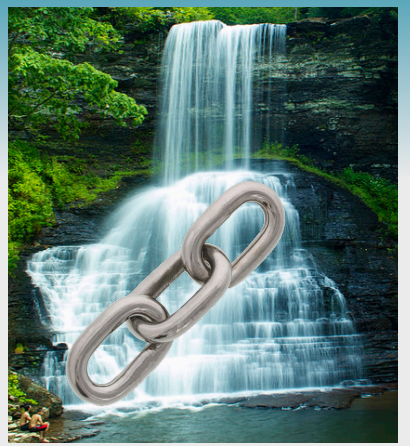
- Running Smalltalk programs requires two parts, VM and image file
  - VM reads and executes Smalltalk code found in image file
  - VM is specific to operating system and CPU architecture being used
- Image files can be moved between operating systems on different CPU architectures
  - displays same (pixel for pixel) across Windows, Linux, and MacOS, only differing based on screen size
  - no need to recompile code for different environments; makes code highly portable
- Image files are snapshots of current environment state
  - describes collection of all active objects
  - during development, changes can be saved to current image or a new image

Message Syntax ...

- Three kinds
  - unary - **`receiver unaryMsg`**
    - example: **`'Hello, World!' print`**
  - binary - **`receiver binaryMsg argument`**
    - example: **`width * height`**
  - keyword - **`receiver kw1: arg1 kw2: arg2 kw3: arg3`**
    - example: **`dogs at: 'Comet' put: 'Whippet'`**
- Precedence is unary, binary, then keyword
- Evaluated left to right
- Use parentheses to change evaluation order

Binary message names can only contain one or more of the following characters:
`+ - * / \ ~ < > = @ % | & ? ,`

... Message Syntax



- An object sends a message to itself using **`self`** as receiver
- Statements in method bodies are separated by periods
- Message cascade (`;`)
  - sends multiple messages to same receiver
  - ex.
```
Transcript show: 'Hello'; newLine; show: ' World!'
```
- Message chain (`::`)
  - sends message to result of previous message
  - removes need to surround previous expression with parentheses

```
1 + 2 squared -> 5
(1 + 2) squared -> 9
1 + 2 :: squared -> 9
```

Blocks ...

- Deferred set of message sends that are not evaluated until block value is requested
- Value is that of the last expression

```
[1 + 2] value
```
3

- Can take arguments

```
[:a :b | a + b] value: 1 value: 2
```
3

OR

```
[:a :b | a + b] valueWithArguments: #(1 2)
```
3

... Blocks

- Can be assigned to variables
- Can be passed to methods
- Can be returned from methods
- Can declare temporary (local) variables
- Can contain multiple statements
- Are closures, so can access in-scope variables

```
average := [:a :b |
    | sum |
    sum := a + b.
    sum / 2.0
]
```

More Syntax

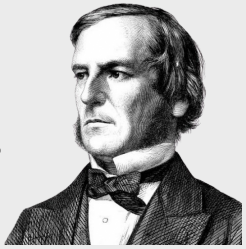
- Literal strings are delimited by single quotes

```
'Hello, World!'
```

- Comments are delimited by double quotes

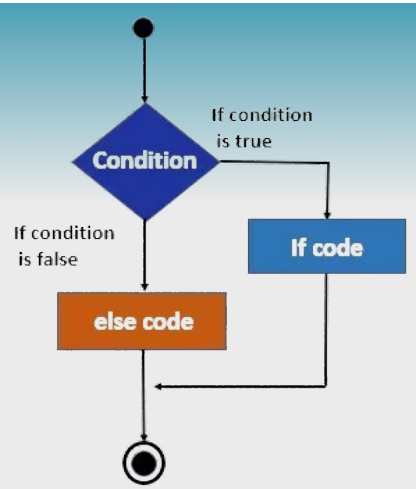
```
"This is a greeting."
```

- Boolean values **`true`** and **`false`** are singleton instances of **`True`** and **`False`** classes which are subclasses of **`Boolean`**



George Boole

Conditional Logic

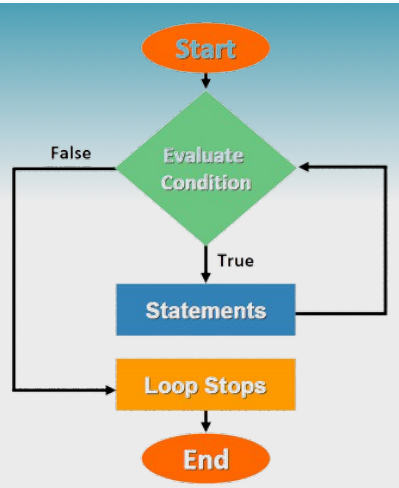


- Implemented as methods in **`Boolean`** class
- Examples

```
result := a < b ifTrue: ['less'] ifFalse: ['more']

color := 'blue'.
assessment := color caseOf: {
    ['red'] -> ['hot'].
    ['green'] -> ['warm'].
    ['blue'] -> ['cold']
}
```

associations

Iteration



- Many methods in provided classes support iteration
  - such as **`Integer`** and **`Interval`**
- Examples

```
3 timesRepeat: ['Ho' print]

interval := 1 to: 10 by: 2.
interval do: [:n | n print]

1 to: 10 by: 2 do: [:n | n print]
```

```
n := 0.
[n = 10] whileFalse: [
    n := 10 atRandom.
    n print.
]
```

- Also see **`do:`** method in collections on next slide

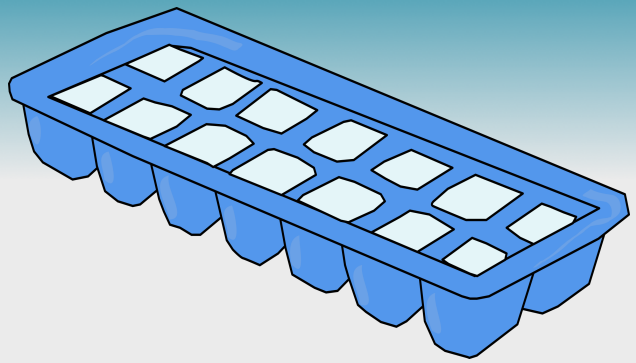
Collections

- Many provided collection classes in a class hierarchy (some shown here)
- **`Collection`** (abstract)
  - **`SequenceableCollection`** (abstract)
    - **`ArrayedCollection`** (abstract)
      - **`Array`**
    - **`Heap`**
    - **`Interval`**
    - **`LinkedList`**
    - **`OrderedCollection`**
      - **`SortedCollection`**
  - **`Bag`**
  - **`Set`**
    - **`Dictionary`**
      - **`OrderedDictionary`**

```
coll := OrderedCollection newFrom: #(1 2).
coll add: 3.
coll do: [ :n | (n * 2) print ]
```

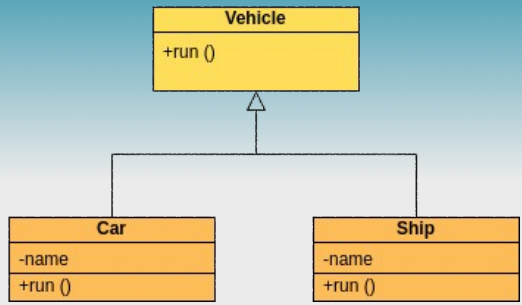
```
2
4
6
```

Arrays



- Literal syntax
  - list of literal values separated by spaces: `#{1 4 8}`
  - list of run-time expressions separated by periods: `{ score1. score2. score3 }`
- Indexed from `1` rather than `0`

Class Definitions



- Must be a subclass of some existing class
- Class name is a symbol
- 3 space-separated lists
  - instance variable names
  - class varianble names
  - pool dictionary names
- Category name

```
Object subclass: #Todo
    instanceVariableNames: 'done text'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'TodoApp'
```

Instance Variables

- Always private
- To expose, define accessor methods

```
text
    ^text

text: aString
    text := aString
```

^ is typically rendered as ↑

:= is typically rendered as ←

Methods

- Always public
- Methods that should only be used by others in the same class, should be in a method category whose name begins with "**`private`**"
  - ex. method **`emptyCheck`** is **`Collection`** class is in **`private`** method category
- Follow conventions for argument names that describes expected type

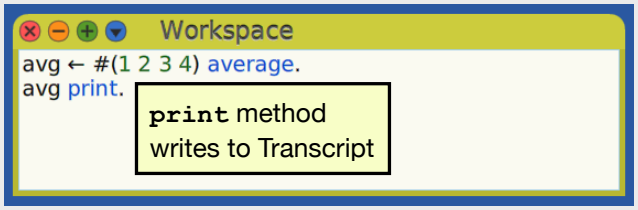
```
findFirst: aBlock startingAt: aNumber
    ...

from: fromNumber to: toNumber do: aBlock
    ...
```

Development Environment

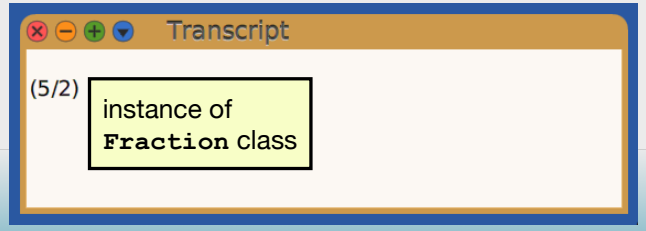
- Many kinds of windows
- Main ones are

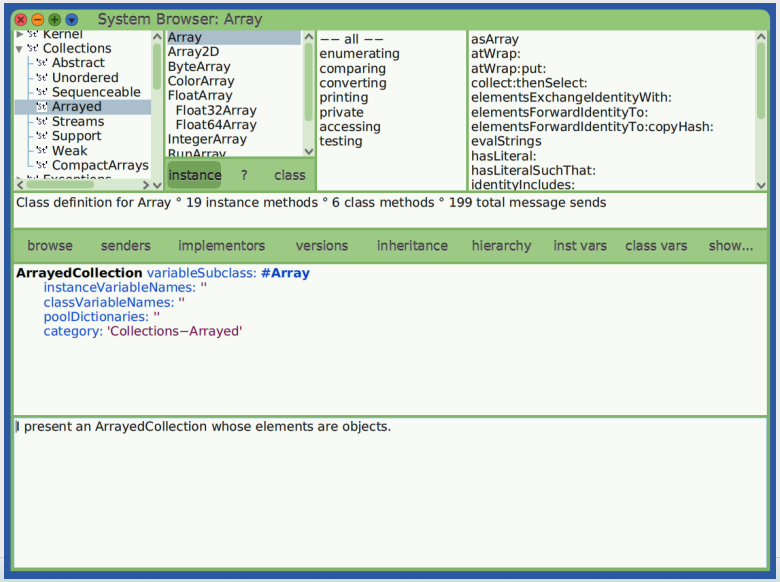
By default, focus moves with cursor position.

  - **System Browser** - for reading, writing, and modifying code
    - see next slide
  - **Workspace** - for experimenting with message sends

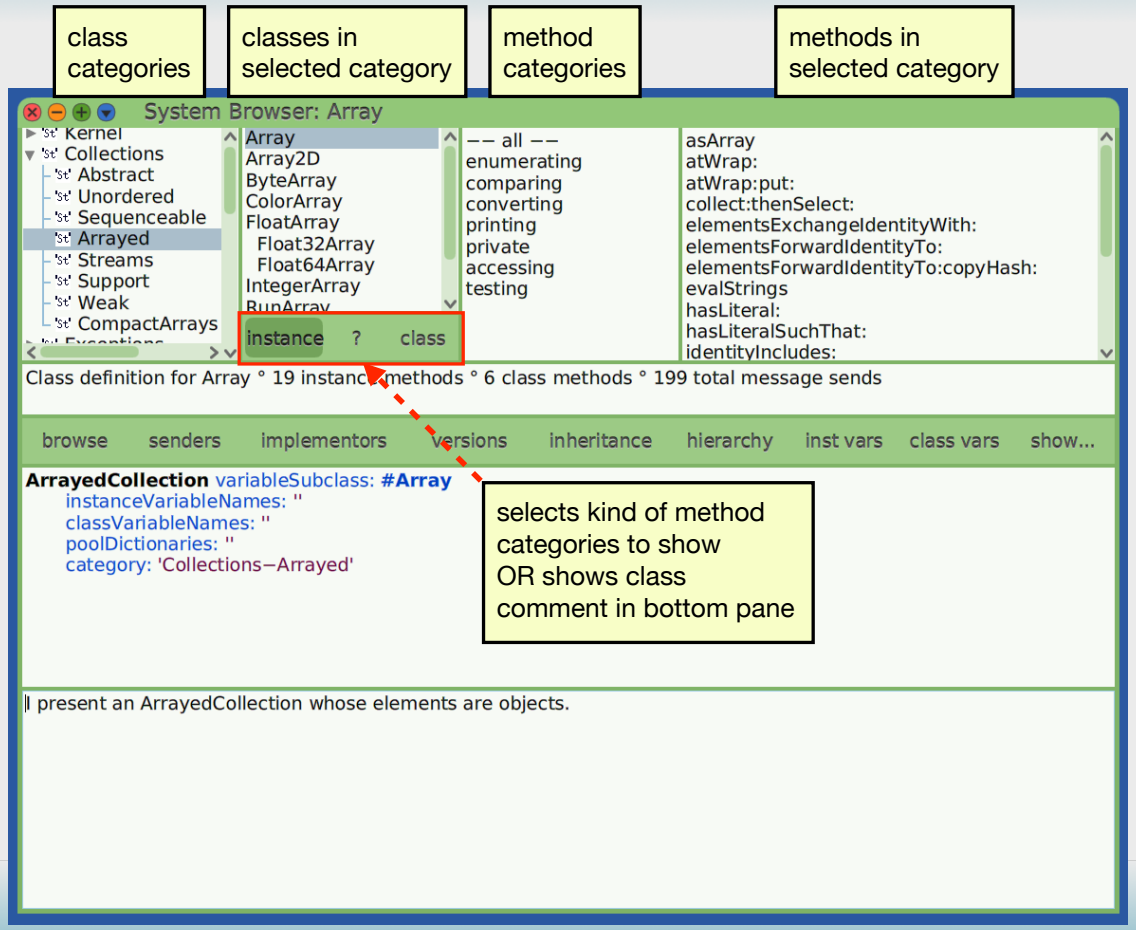


  - **Transcript** - for displaying output





System Browser



Code is written and read one method at a time, not in one source file per class.

#1

- Open a Browser
- Hover over class categories pane
- Press cmd-f and enter "`Stri`"
- Select "`String`" class
- Hover over methods pane
- Press "`i`" to scroll to first entry that begins with that
- Select "`isEmpty`" method
- Note that it sends the message "`size`" to `self`
- Click `size` method and discuss primitives (62 returns collection size)
- Click "?" button to see class comment
- Click "class" button to see list of class methods

#2

- Open Browser
- Right-click in class categories pane and select “add item...”
- Enter “**`Demo`**”
- In bottom code pane, change “**`NameOfSubclass`**” to “**`Greeter`**”
- Press cmd-s to save new class
- Add “**`message`**” to **`instanceVariableNames`** list
- In message categories pane, select “**`as yet unclassified`**”
- Add these methods →

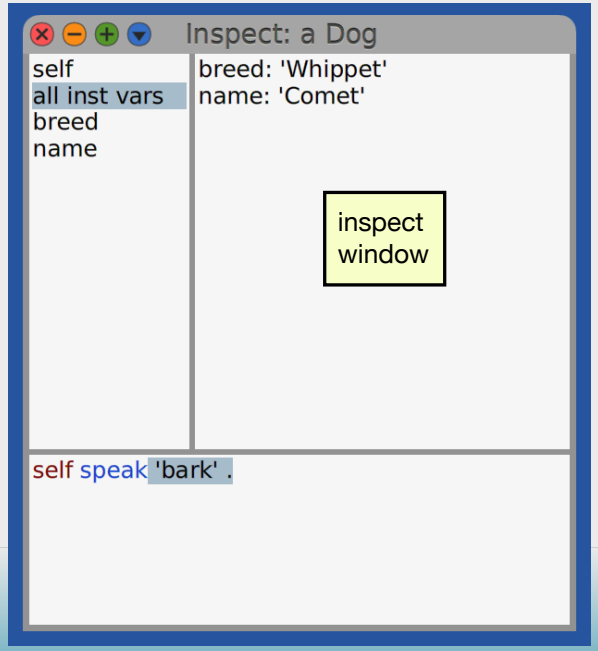
```
message
    ^message

message: aString
    message := aString

greet
    message print
```

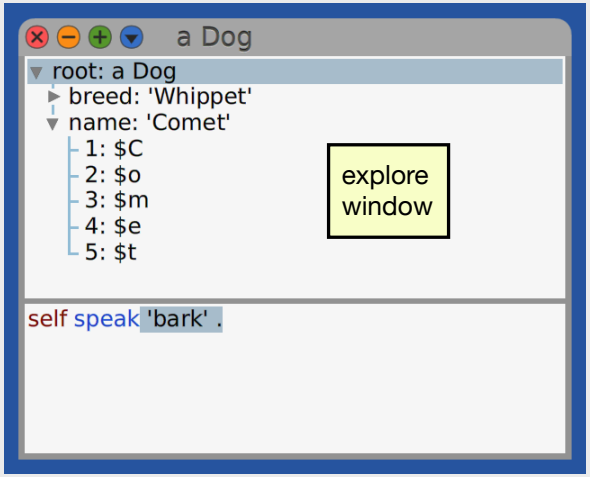
- Place first two methods in method category “accessing” and last one in “printing”
- Open Workspace
- Enter following lines, select them, and press cmd-d to “Do it”

```
g := Greeter new message: 'Hello, World!'.
g greet
```

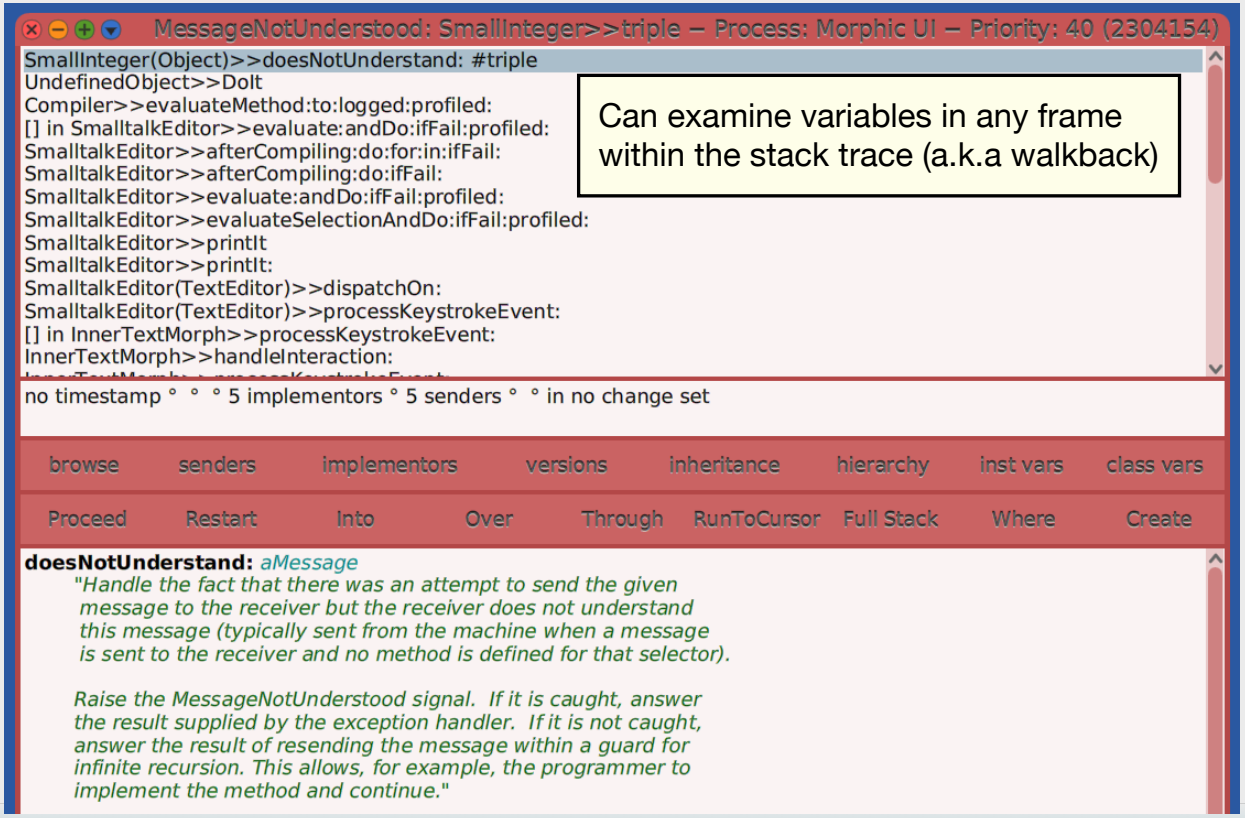
Inspect/Explore Windows

- Inspect windows examine a single object
- Explore window examine a tree of objects starting from one
- Both can send messages to selected object in bottom pane





Most Common Error



#3

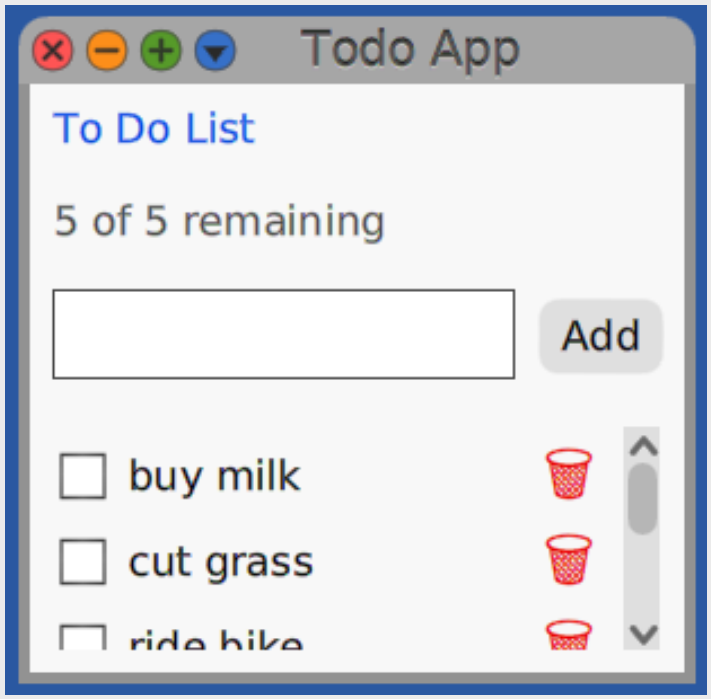
- In the Workspace, enter “g” (variable name)
- Press cmd-i to “Inspect it”
- Press cmd-I to “Explore it”
- In the bottom pane of either window, enter self message asUppercase
- Press cmd-p to “Print it”

#4

- Open a “Message Names” window
- Enter “**`at:put:`**”
- Select “**`Dictionary at:put:`**”
- Study signature, method comment, and method body
  - **`findElementOrNil`** is implemented in **`Set`**
    which is the superclass of **`Dictionary`**
  - **`ifNil:ifNotNil:`** is implemented in **`ProtoObject`**
    which is the superclass of **`Object`** which is a superclass of every class

Todo App

- Demonstrate app in Cuis
- Walk through code in Cuis



Resources

- **Smalltalk Wikipedia page**
  - https://en.wikipedia.org/wiki/Smalltalk
- **Cuis Smalltalk**
  - https://cuis.st/
- **Pharo Smalltalk**
  - https://pharo.org/
- **Squeak Smalltalk**
  - **https://squeak.org/**
- **Byte Magazine issue on Smalltalk**
  - https://archive.org/details/byte-magazine-1981-08

Wrap Up

- For me, Smalltalk has the most elegant syntax of any programming language
- Even if choose not to use Smalltalk, there are many ideas from it that can be applied to other programming languages
  - especially in its development environment
- Todo App code at https://github.com/mvolkmann/Cuis-Smalltalk-TodoApp