

**ECOOP 2001 Tutorial**

**Squeak:  
An Open Source Smalltalk  
for the 21st Century**





# **Squeak: An Open Source Smalltalk for the 21st Century!**

Andrew P. Black

Professor, Oregon Graduate Institute



## **What is Squeak?**

- An “Open Source” Smalltalk
- A pure Object-Oriented language for research, experimentation, prototyping, building applications
- Support for e-mail, web, sound, video, 3-D modeling, scripting, presentations...
- An active community of people who are getting excited about programming again!
- A “place” to experiment with objects



## What we *won't* do this afternoon...

- Listen to a 3-hour lecture telling you everything about Squeak

Why?

- I don't know *everything* about Squeak!
- Even if I did, I couldn't tell you everything in 3 week, let alone 3-hours!!
- What you learn would be out of date in a month anyway!!!



## What we *will* do:

Learn how to *learn* about Squeak

- Focus on showing you how to find out more
- Explore objects
- Explore source code
- Try things out — learn by doing
- Know to use the Swiki and the mailing list



## Outline

14:00 —Introduction

14:30 —Basic Smalltalk (Worksheet 1)

- Worksheet on Squeak syntax, creating and browsing classes, instances and methods
- Using the paragraph editor and command keys
- Using Workspaces
- Filling-in code



15:00 —How to learn more (demo)

- Finding classes, exploring objects
- Finding methods, senders, implementors
- Fixing a bug
- “Showing off” your code: The Dandelion system

15:20 —Morphic User Interface (worksheet 2)

- 15:30—16:00 Refreshment break
- Drawing on the Screen
- Morphic Events
- Animation and updating



## 16:20 — More on Morphic

- What's so neat about Morphic anyway?
- Any object is a window
- Relative addressing

## 16:30 —Morphic Programming Project (hands on)

- Build your own Morphic project



## 17:00 —Distribution and MultiMedia (Demo and lecture)

- Sound
- MPEG
- s2s: Remote Message Send
- PWS: Pluggable Web Server
- The SuperSwiki

## 17:30 — The End



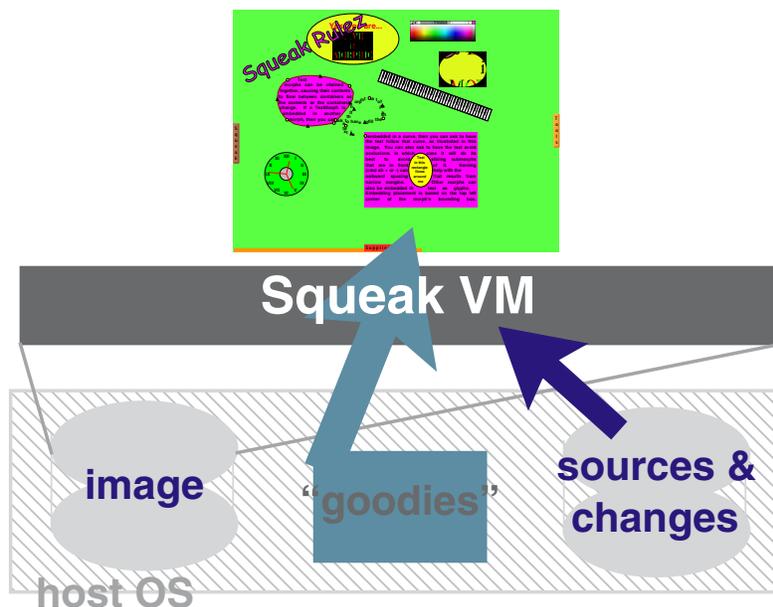
# The Squeak Environment

A “place” to experiment with objects

- Forget applications, files, compilers, data...
- Focus on objects



# The Squeak World



## Smalltalk Syntax

- No syntax for classes, packages, *etc.*
  - Class creation and method categorization are done *imperatively* using the development tools
- The method syntax is simple, but different

### **>= aString**

"Answer whether the receiver sorts after or equal to aString. The collation order is simple ascii (with case differences)."

`^ (self compare: self with: aString collated: AsciiOrder) >= 2`



## Smalltalk — The Language

### Literal Objects

27	The unique object 27
18.5	The floating point number 18.5
1.85e1	same as above
'a string'	a string
#request	the symbol <i>request</i> . It is unique; two symbols with the same name denote the same object
\$r	the single character <i>r</i>
(3, 2.7, 'a string')	an array literal. This is a heterogeneous array containing an integer, a float, and a string



# Sending Messages

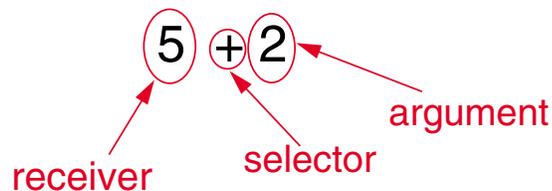
## Unary Message (no arguments)



- selector is a keyword-like *symbol*
  - examples: 3 factorial  
7 negated  
\$c asInteger
  - note: no colon at the end of the symbol



## Binary Message (one argument!)



- selector is one or two special characters

7 = 5	message = 5 sent to object 7
2 + 3	message + 3 sent to object 2
17 // 3	message // 3 sent to integer object 17 (result is 5)
17 / 3	message / 3 sent to integer object 17 (result is )



## Keyword Messages

- one or more arguments
  - Examples:  
#(3 5 7 9 11) at: 2  
game movefrom: pinA to: pinB using: pinC  
5 between: 0 and: 9
- The colon ':' indicates to the parser that an argument follows the keyword.



## Order of Evaluation

- The receiver (or an argument) can be another invocation (message expression)
- Evaluation order is
  - parenthesized invocations
  - unary invocation, evaluated *left to right*
  - binary invocations, evaluated *left to right*
  - keyword invocations
- No “priorities” for particular operators
  - \* does not bind more tightly than +



## Cascaded Messages (syntactic sugar)

```
anArray at: 1 put: 9.  
anArray at: 2 put: 11.  
anArray at: 3 put: 13.
```

- This can be abbreviated as

```
anArray at: 1 put: 9; at: 2 put: 11; at: 3 put: 13
```

receiver for all  
3 messages

“receiverless messages”

- Result is that of the last message send

```
Transcript show: 'Hello World'; cr
```



## Variables

### Instance Variables

- The names of the “slots” in an object, which make up its representation.
- declared in the class

```
instanceVariableNames: 'name1 name2'
```

### Temporaries

- Names local to a method body or block

```
| student professorAtOGI |
```



## Assignment

$x \leftarrow 3 + 5$

- make  $x$  name the object resulting from the evaluation of the expression  $3 + 5$

$y := \text{Array new: } 1000000$

- make  $y$  name a new 1MB array

- Variables name objects
  - They do not provide storage for objects
- Assigning to a variable makes it name a different object
  - no object is created or copied by assignment



## Enter the Squeak World!

- If you have loaded Squeak, but not really figured out how to do anything:
  - do worksheet 1A: [An Introduction to the Squeak World](#).
- If you are familiar with another Smalltalk, or have already done 1A, instead:
  - do worksheet 1B: [Building Applications by Direct Manipulation](#).
- If you haven't got Squeak on your computer:
  - Come and see me or one of the student assistants.



## If you get Stuck...

- If you get stuck, yell for help.
- Save your brainpower for the hard stuff!
- The reason to do a hands on workshop is to quickly get past the initial learning “bump”

14:30 – 15:00 — “Hands On”  
Worksheet 1A or 1B



## Learning More

- Finding Classes
  - By name or fragment of a name
    - `command-f` in the Class-category pane of a browser
  - By selecting a morph and choosing `browse morph class` from the debug menu



- Finding methods

- By name fragment or by example — with the **method finder**
- **Smalltalk browseMethodsWhoseNamesContain:** 'screen'
- **Smalltalk browseMethodsWithString:** 'useful', or highlight the string and type *command-E*
- highlight a selector, choose **implementors of ...** (*command-m*) or **senders of ...** (*command-n*)



## Finding Answers

Some invaluable resources:

- The Squeak “Swiki”
  - a wiki is a website where anyone is free to contribute to editing and maintenance
  - <http://minnow.cc.gatech.edu/squeak>
  - snapshot at <http://swikimirror.squeakspace.com/>
- Squeak.org
  - Documentation, tutorials, swikis, other sites, books and papers, downloads, and information on ...



- The Squeak mailing list

- a friendly place where “newbies” are made welcome

- [squeak-request@cs.uiuc.edu](mailto:squeak-request@cs.uiuc.edu)

- Archive of [FIX]es, [ENH]ancements, [GOODIE]s...

- <http://swiki.gsug.org:8080/SQFIXES>

- Searchable archive of whole list

- <http://groups.yahoo.com/group/squeak>



## Example—Finding a Bug

- Let your mouse linger over the collapse icon in a window.

- What does the balloon say?

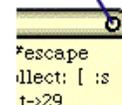
- Now click the icon and collapse the window.

- Let your mouse linger over the *same* icon in the collapsed version of the window.

- What does the balloon say?

- **Let's fix this right now!**

Collapse icon



## Showing off your Code

- Smalltalk programmers normally look at code using their Smalltalk System
  - code file browser, system browser ...
- But what about showing code to others?
- Enter **Dandelion**: a code analysis framework for Squeak.
  - produces web pages for each class
  - will eventually generate other forms of documentation, *e.g.*, UML class diagrams



## Morphic User Interface

15:20–16:20 — Worksheet 2

- Remember:  
refreshment break from 15:30 to 16:00



## More on Morphic

- A new way of thinking about graphical Interfaces
- Morphic *reifies* the UI
  - reify = “to make real”
  - no separation between *building* the UI and *using* the UI



- display objects—**Morphs**—are full-fledged Squeak objects
  - they have state and behaviour
    - you can send them messages
  - you can add or subtract to their structure
    - they can be nested to arbitrary depth
  - they react to events (like mouse clicks)



- You can change the look and behaviour of Morphs *after* they are instantiated
  - Example: Buttons
    - Place a button on the desktop
    - Give it behavior so that when clicked it opens a workspace.
    - Try [preferences >> annotationPanels](#)
  - Example
    - “Ant” has a few simple rules for wandering about, dropping pheromones
    - [StarSqueakAntColony new openInWorld](#)



## Hierarchy of Morphs

- The “world”—the desktop or background—is a Morph (a [PasteUpMorph](#))
- Any morph can be embedded inside any other morph
  - But not inside more than [one](#)
  - Thus we have a pure tree structure:
    - each morph has zero or one parents, called [owners](#)



- Example: building a “Launcher” [launcher.5.cs](#)

### Launcher >> initializelayout

```
initializeLayout  
self layoutPolicy: TableLayout new.  
self listDirection: #leftToRight.  
self layoutInset: 2.  
self borderWidth: 0.  
self hResizing: #shrinkWrap.  
self vResizing: #shrinkWrap.  
self color: Color gray.
```

- listDirection can be changed dynamically, *e.g.*, to *#topToBottom*



## What about Model-View-Controller?

- The Model-View separation was pioneered by Smalltalk.
- It's an excellent idea for *many* applications ... but not for all.
- The Ant world, for instance, is much simpler without the separation.
  - We can actually give behavior to the dots on the screen.
- You can use Morphs to implement View and Controller with a separate Model.



## A Word about MVC

- Squeak at present has **two** mostly incompatible GUI frameworks:
  1. Morphic — my focus today
  2. MVC — the traditional UI framework since 1980
- There are a number of classes that really work only in MVC, *e.g.*, Spline, Line, CurveFitter ...
- We will not talk about MVC during this tutorial
  - There are many references available on MVC



## Morphic Programming Project

- Build your own Morphic Clock
- Play the Reflex Game

**16:30 – 17:00 — Worksheet 3**

- **Warning: thinking is required!**
  - this worksheet is less of a cookbook
  - more discovery is required



## Distribution and Multimedia

Squeak is network and web aware

- Mail agent (Celeste)
- Web browser (Scamper)
- Online Updates
- The “SuperSwiki”
  - Squeak projects that can be loaded directly
- Netscape plugin
- PWS: Pluggable Web Server



## Distributed Programming

- TCP and UDP, FTP, HTTP, SMTP and POP3: all the “P”s!
- s2s: a remote message send system for Squeak.
  - Creates proxies for remote objects.
  - Messages sent to proxies are relayed to the real object on the remote machine.
  - Answer is relayed back.
  - Like CORBA or Java RMI.



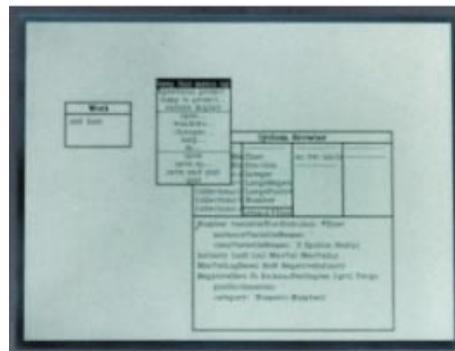
## Multimedia Support

- Has support for MIDI.
  - Sampled sounds as well as synthetic waveforms.
  - Look at “fun with Music” project.
- Has MPEG decoder and player
  - but decoder is file-based plug-in.



## A Squeak Machine

- SoftComputing “is using Squeak to program an embedded microprocessor “from the ground up”
  - 160MHz combined DSP and RISC chip
  - See <http://www.softcomp.com>
  - I’m not endorsing this product!





## What Next?

- Explore the projects in the image and the SuperSwiki
- Buy Mark Guzdial's books!
  - O-O Design with Multimedia Apps
  - Squeak: Open Personal Computing and Multimedia
- Join the squeak mailing list
- Contribute to the evolution of Squeak.
  - It's an open source project
  - *You can drive it wherever you want to take it*



## Acknowledgements

- All of the people on the Squeak mailing list
- “Squeak Central”



without whom I would not be here today!



## The End



# **Worksheets**



## Worksheet 1A

# An Introduction to the Squeak World

---

Based on Introduction to Squeak by Harry Porter  
Version 1.0, 17 April 2001, by Andrew P. Black, [black@cse.ogi.edu](mailto:black@cse.ogi.edu)

---

## Getting Started

If you don't already have the Squeak system on a computer that you can share, download it now! Launch Squeak.

## Using the Mouse

Smalltalk, from which Squeak is derived and with which it is still somewhat compatible, was designed assuming a three-button mouse. If your mouse has fewer buttons, you must press extra keys with the mouse button or buttons to simulate the mouse buttons that you are missing.

For platform independence, the mouse buttons are usually referred to by colors in the Squeak software and documentation. Consider buying a 3-button mouse; they are quite inexpensive, and make the Squeak world much more friendly. (They are pretty useful in many other applications too!) Software that comes with the mouse, or that is available on the Internet, will let you set up the buttons so that they do the right things. The following chart shows what keys must be held down when mouse-clicking to simulate Squeak's buttons.

Symbolic	MacOS	Windows	3-button	Use
Red	Mouse button	Left-button	Left-button	Selecting, moving the insertion cursor
Yellow	Option-button	Right-button	Middle-button	Application-specific menus
Blue	⌘-button	Alt-Left-button	Right-button	Window and graphics manipulation

My mouse also has a scrolling wheel. I have this set up so that "wheel up" maps to `⌘-upArrow` (on my Macintosh) and "wheel down" maps to `⌘-downArrow`. This lets me use the wheel to control scrolling in my Squeak windows.

One of the advantages of referring to the buttons by colors is that you may choose to map them to different physical buttons from those shown above. For example, if you are left-handed, you might choose to reverse the red and blue buttons. On my mouse, the middle button is actually the scrolling wheel. Because the yellow button is used very frequently, I prefer to put the yellow button on the right, and to put the blue button on the wheel in the middle.

Placing some colored labels on the mouse buttons will help your fingers to follow these directions.

## The World Menu

Red-click outside of any window; you will see the *world menu*. Notice that most Squeak menus are not modal; you can leave them on the screen for as long as you wish by selecting *keep this menu up*. Do this. Also, notice that menus appear when you click the mouse, but do not disappear when you release it; they stay visible until you make a selection, or until you click outside of the menu. You can even move the menu around by grabbing its title bar.

Bring up the *open...* submenu and select *workspace*. You should get a large window labeled *Workspace*.

## Terminating and Restarting a Squeak Session

A Squeak session is normally terminated by writing out all of your objects to a disk file called a snapshot or *image* file. When you first start Squeak, the Squeak Virtual Machine is loaded with an initial set of objects, which includes a vast amount of pre-existing code and programming tools (all of which are objects). You will modify these objects during your Squeak session.

When you terminate Squeak, you will *save* a snapshot of your memory containing all of your objects. The next time you use Squeak, memory will be reloaded from this file and the state of the system will be exactly as you left it. The snapshot file will be named *xxxxx.image*, where *xxxxx* is something you will choose, like *ECOOP.image*. There will be a corresponding "changes" file, with a name such as *ECOOP.changes*.

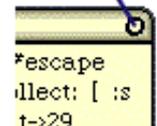
- From the world menu, select *new morph ... >> Demo >> BlobMorph*. A "Blob" will attach itself to the mouse cursor. (The jargon is: the blob is "in hand"). Put the Blob down (by red-clicking) somewhere on the screen.
- Make your first snapshot by selecting *save as...* from the world menu. Give a name to your image file when requested, such as *ECOOP.image*. (Saving may take a second or two.) You may save an image any time you wish, by selecting the *save* option from the system menu. If the Squeak system crashes, and you restart it, the state will revert to the last snapshot you made. You can use this feature to go back to an earlier state if you get your Squeak machine into a really big mess.
- Another file called *ECOOP.changes* will also be created. All the source code that you compile or execute is saved there. For the most part the changes can be ignored, but if a crash occurs you can use it to recover changes made after the last snapshot. See [the box](#) (at the end of this worksheet) for instructions.
- Select *quit* on the world menu to exit the Squeak environment.
- Restart Squeak from your saved image. The blob should still be ... well, behaving like a blob! Yes, running processes are saved and restored too.

## Collapse and / or close unneeded windows

Each window has a title bar with a "close" icon (an X) in the upper left and a "collapse" icon in the upper right.

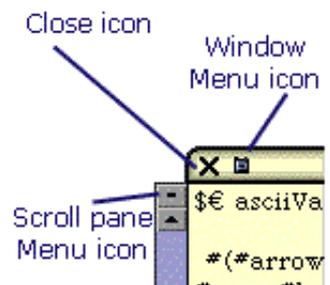
- In the initial state, you will see several windows on the screen. Collapse the windows called *Getting Started...* and *Welcome To...* Then expand them back to their original size.
- Close the Workspace window that you created earlier.
- Blue-click on the blob. You will see a collection of colored dots. Click in the pink  handle. The blob should go away. (You may have to try several times as the Blob squirms around and tries to get away from the mouse.)

Collapse icon



## Workspaces

- **Open another workspace.** This time, instead of using the world menu, mouse over to the *Tools* flap at the right hand edge of the screen. The flap will pull out, like a drawer. Inside are a collection of tools. The off-white rectangle third from the top is a workspace. If you let the mouse linger over the tools, ballons will identify them. *Drag* a workspace out of the flap.
- **Practice resizing and moving windows.** To move the workspace window, grab it in the title bar at the top. To make the window larger or smaller, move the mouse to the lower right corner. A yellow dot will appear; this is a draggable resize handle. Note that a scroll bar appears when the cursor is in the workspace. Ignore the scroll bar for now.
- **The window menu.** Every window has menu of items that apply specifically to that window. Bring up the window menu for your workspace by red-clicking the small *menu* icon at the left of the title bar, just to the right of the *close* icon. Select *change title ...* to change the label in the title tab of the workspace. This brings up a prompter—this is the same kind of prompter used when saving a snapshot. (It's actually an object of class *FillInTheBlank*.) Experiment with *full screen*, and with changing the stacking order. Select *window color*; the system is now waiting for you to select a color for the window. Position the mouse anywhere on the screen and click it to select a color.



- **Close the workspace** window by pressing the "close" icon in the upper left corner.
- **Window Selection.** At any time only one window is active. The active window is in front and has its label highlighted. The mouse cursor must be in the window in which you wish to type. Create a second workspace and experiment with changing the active window with the red button. Experiment with overlapping views.
- **Text Editing in a Workspace.** Type a sentence or two into a workspace. Use the *return*, *tab* and *backspace* keys. Drag with the red button to select some characters. (Press, drag, and release.) Type characters to replace the highlighted characters. Try inserting and deleting characters after repositioning the cursor. Click the mouse button twice without moving it. This is called double-clicking, regardless of how long a time elapses between the clicks. What does double-click do when the cursor is in the middle of a word, at the beginning of a line, at the beginning of the workspace or after quotes, brackets or parentheses? (The bracket/quote feature will come in handy in matching nested brackets and parentheses when you are writing code.) Bring up the yellow button menu and practice using *copy*, *cut* and *paste*. (There is a "copy buffer" for text that has been cut. "Copy" saves into this buffer without cutting. You can also copy and paste bwtween Squeak and other applications.
- **Command keys** can be used as short-cuts keys for *copy*, *cut*, and *paste*, as well as for many other editing operations—see the [Squeak Language Reference Sheet](#) for a full list.
  - In this worksheet, we will write *command-c* to mean that you type c while holding down the shortcut key. But the actual key that you press to get a shortcut depends on your operating system. On the Macintosh, hold down the  $\text{⌘}$  key. On an IBM-compatible PC, hold down *alt*.
  - Some command key shortcuts are written with a capital letter. For example, *command-T* inserts the text `ifTrue: .` To type these capitalized shortcuts you can hold down *shift* and the shortcut key while you type *t*, or, more conveniently, you can hold down *ctrl* while you type *t*.
  - Here are some of the more commonly used editing shortcuts.

**General Editing Commands**

Key	Description	Notes
z	Undo	
x	Cut	
c	Copy	
v	Paste	
a	Select all	
D	Duplicate. Paste the current selection over the prior selection, if it is non-overlapping.	1
e	Exchange. Exchange the contents of current selection with the contents of the prior selection	1
y	Swap. If there is no selection, swap the characters on either side of the insertion cursor, and advance the cursor. If the selection has 2 characters, swap them, and advance the cursor.	
w	Delete preceding word	

1. These commands are a bit unusual: they concern and affect not only the current selection, but also the immediately preceding selection.
- **Scrolling.** Using *copy* and *paste*, type in more text than will fit in the window. The scroll bar works as you expect; whether scroll bars are on the right or the left, and whether they flop out when they are needed or are permanently on view, can be controlled by preferences. (*World menu*>> *help*>>*preferences*). Notice that there is a tiny menu icon at the top of the scroll bar. Press the red mouse button here to get the yellow button menu that applies to the scrolling text pane. This is a nice short-cut, especially when using a stylus or a one-button mouse.
  - **Message Sending.** Type `2+3` in the workspace, select it and then select the *print it* menu option, or type *command-p*. (*Print it* and *do it* work on the selected text, but if there is no selection, they work on the whole of the current line.) Both commands evaluate the selected expression. The *print it* command displays

the result, but the *do it* option doesn't display the returned value. The short-cut key for *do it* is *command-d*. These are very common actions, so practice the short-cut keys.

Key	Description
d	Do "it" (where "it" is a Squeak expression)
i	Inspect "it": evaluate "it" and open an inspector on the result. ("it" is a Squeak expression). Exception: in a method list pane, i opens an inheritance browser.
p	Print "it". Evaluate "it" and insert the results immediately after "it." (where "it" is a Smalltalk expression)
I	Open the Object Explorer on "it" (where "it" is an expression)

- **Undo, Searching.** Experiment with *undo*, to undo the last change. Experiment with searching the workspace for a string. Use the shortcut keys as well as the menu items:

Key	Description
f	Find. Set the search string from a string entered in a dialog. Then, advance the cursor to the next occurrence of the search string.
g	Find again. Advance the cursor to the next occurrence of the search string.
h	Set Search String from the selection.
j	Replace the next occurrence of the search string with the last replacement made
A	Advance argument. Advance the cursor to the next keyword argument, or to the end of string if no keyword arguments remain.
J	Replace all occurrences of the search string with the last replacement made
S	Replace all occurrences of the search string with the present change text

- **Accept, Cancel and Confirmers.** Figure out how the *accept* and *cancel* commands work. Here are the shortcut keys.

Key	Description
l	Cancel (also "revert"). Cancel all edits made since the pane was opened or since the last save
s	Accept (also "save"). Save the changes made in the current pane.
o	Spawn. Open a new window containing the present contents of this pane, and then reset this window to its last saved state (that is, cancel the present window).

- Workspaces keep a back-up copy. *Accept* writes into this copy.
- Make some changes. *Cancel* takes the workspace back to the last copy saved by *accept*. Do an *accept* and then close the workspace.
- In another workspace, make some changes without accepting them and try closing this workspace. A confirmer should appear; try ignoring the question—what happens?

## Graphics Demonstrations

- In a workspace, type *Spline example*, select it and *do it*. Use the red button to click off several points on the screen. Then click another button: you should see a curve. What does *restore display* do to the curve? Can the curve lie inside a window? What do scroll bars do to the curve? Why?
- Type in *Spline example1*, then select *do it*. This is a typo and you should see a window pop up. Push the

*Abandon* button. (*example1* is a valid message, just not for Splines.)

- Type in *Spline exampl* then *do it*. This is also a typo, but this time Squeak suggests some possible corrections. Select *cancel* or *example*. (Here *exampl* is not a valid message on any object, so the system looks for messages with similar spellings.)

## Accessing Files

- From the world menu, select *open...>>file list*. Alternatively, drag a *file list* browser from the *Tools* flap. You should see a pink window with several "sub-windows" known as *panes*. This is a file list browser, and gives you a view of the current directory.
  - In the upper-left window pane you'll see the directory hierarchy, directories, so you know where you are; the current directory is at the bottom.
  - In the upper-right pane, you'll see all the files in this directory.
  - Select a file name in the upper-right pane. (Choose a small text file.) You should see its contents in the bottom pane.
  - Try using the yellow-button menu in the upper-right pane to change the sort order.
- The bottom pane is similar to a workspace. You may edit files this way. Actually, you are editing an in-memory copy. Use *accept* (or *command-s*) to write the in-memory copy back to disk. Use *cancel* to revert to the old version without updating.
- The middle-left pane, containing just a \*, defines a filtering pattern. You can change the pattern by *accepting* new text into this pane. Only directories and file names matching this pattern will be listed in the upper-right pane.
- The file list browser also supports ftp. Servers appear as if they were volumes at the root directory level. Yellow-click in the upper-left pane to add a new ftp server.

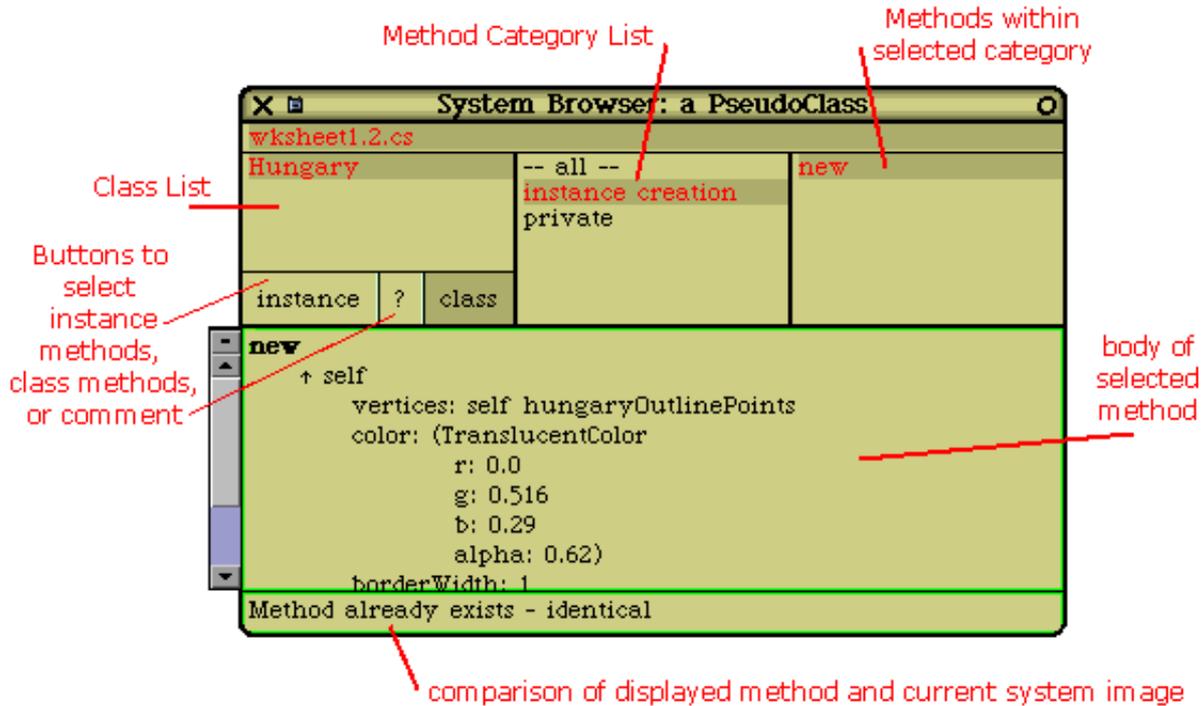
## File Creation and Removal

- In the upper-right pane, select *add new file* and give it a name. This creates a zero-length file.
- Make some changes to the file, write them out, and read the changes back in.
- Use *delete* to remove this and other unneeded files. Select the *more...* menu option and take a look at the submenu options.

## Importing Code using *file In*

- *file In* is important: it is used for reading and compiling Squeak source code.
- Select the file *wkSheet.cs* from the list in the upper-right pane.
- Choose *browse code* from the yellow-button menu. This brings up a tan-colored *file contents browser*, which

lets you examine the code in this file before you import it into Squeak.



- o Click on the comment button ? and the *class* button, and explore the code in this file.
- o Note how selections in one list affect the lists to its right.
- o Dismiss the file contents browser.
- Return to the file list browser, make sure that the file *wkSheet.cs* is still selected, and choose *file In* from the yellow-button menu.

## Forms and Points

Forms are rectangular blocks of pixels, like mini-bitmaps.

- Expressions such as `100@100` specify Points; the @ character is a binary infix operator. For example `20@500` means the point with `x=20` and `y=500`.
- The upper-left corner of the screen is position `0@0`. The `y` coordinate increases *down* the screen.
- Experiment with different display locations and variable names by re-sending the above messages.
- Use `f ← Form extent: 30@40` to create a blank form. The `extent: message` creates a form that is 30 pixels wide (`x`) and 40 pixels high (`y`).
- Execute `f displayAt: Sensor waitButton`. Click the mouse button.

## Using a System Browser

The System Browser is the main tool used to read and write code in Smalltalk. It is OK to have many System Browsers open at once. Normally, if you change code in one Browser, the changes will be visible in any other. (*Preferences >> smartUpdating* controls this).

- Open a *System Browser* from the tool flap (or the world menu). The upper 4 panes in the browser contain lists. These lists work just like the lists in the file contents browser. Practice scrolling and selecting items from the lists. Note how selections in one list affect the lists to its right.
- Select *ECOOP-Tutorial* from the upper-left pane; it should be right at the bottom. This pane is the *class category* pane; because there are so many classes in Squeak, it is convenient to organize them into categories. Once you have selected *ECOOP-Tutorial*, you should see the class *Hungary* (the one that you just filed in) in the *class* pane, just to the right.
- Look at some methods in the lower pane.
- Examine the menus in each pane of the browser. See if you can figure out what some of the menu options

do. (Selecting *accept* in the lower pane will re-compile the method if you made any changes. Select *cancel* if wish to avoid a compile and revert to the previous text.)

- If you inadvertently change a method, use the versions menu item in the message list (top right) pane to get back the old one.
- Locate the even method in class Integer. How does this method determine its result?
- Create a new Class category for the classes that you will define during this Workshop. Call it something like *ECOOP-MyStuff*.

## Executing Code: Sending Messages to Objects

Paths are collections of points used to represent geometric shapes.

There are two ways to type the assignment operator. You may either type colon-equal (like Pascal) or you may use the left arrow, which in Squeak replaces the underscore character. In other words, type the `_` key and see `←` on the display.

- In a workspace, execute `p := Path fromUser`. Then choose a blank peice of screen and click on several points in a rough circle using the red mouse button. Each time that you click, you will leave behind a small red dot. When you have enough points, click with one of the other mouse buttons. (The dots will disappear.)
- What's going on here? This expression sends the message `fromUser` to the class object `Path`. The answer is a new instance of `Path`, which we assign to the temporary variable `p`.
- Execute `p ← Path fromUser`. (Same statement, just use the left-arrow this time.)
- Execute `p displayAt: 10 @ 0`. The red dots should reappear, but only for a moment, and offset 10 pixels to the right of where you first drew them.
- OK, now let's make something that stays on the screen. Type the following text into the workspace, select it and *do it*. Be careful with the capitalization. For example, in `Color blue`, `Color` is a class, and so is capitalized, while `blue` is a message sent to that class, and so is lower case. Also, `vertices: color: borderWidth: borderColor:` is a bit of a mouthful, so type `vertices` and then use *command-q* completion. *ctrl-a* will move the cursor from one argument position to the next.

```
m ← PolygonMorph vertices: p color: Color blue borderWidth: 1 borderColor:
Color black.
```

```
m openInWorld
```

- The first line, `m ← PolygonMorph ...`, creates a new PloygonMorph and names it `m`. The seond line tells `m` to display itself in the `World`, that is, on the screen. A blue PolygonMorph will appear on the screen.
- Pick up your PolygonMorph with the mouse, and put it down somewhere else on the screen.
- Bring up the window menu of the workspace (using the window menu icon), and select the last item: *start accepting dropping morph for reference*.
- Pick up your PolygonMorph again, and putit down in the workspace. What happens?
- You should now have a textual name in your workspace for the PolygonMorph — something like `polygon879`. This provides you with a way of referring to the morph. Of course, you already happend to have a name for your morph—`m`. So `m` and `polygon879` should both be names for the same object. Let's check this: do a *print it* on

```
polygon879 == m
```

`a == b` tests whether `a` and `b` are the very same object. This expression should print as *true*.

- Now execute

```
m color: Color red
```

The morph will change color.

- Pick up the (red) morph with the mouse, and drag it into the *trash can* in the Squeak flap. (If you like, you can drag the trash can out of the Squeak flap and put it somewhere more convenient.) Now execute `m`

openInWorld again. The morph (still red) will reappear.

- What's going on here? What do you think dropping a Morph into the trash really does?

## What if you Crash Squeak?

It is quite possible to crash Squeak—as an experimental system, Squeak lets you change *anything*, including things that are vital to make Squeak work! For example, try `Smalltalk := nil`.

The good news is that you need never lose work, even if you crash and go back to the last saved version of your image, which might be hours old. This is because all of the code that you executed is saved in the changes file. All of it! That includes one liners that you evaluate in a workspace, as well as code that you add to a class while programming.

So here are the instructions on how to get your code back. There is no need to read this until you need it. But when you do need it, you'll find it here waiting for you.

In the worst case, you can use a text editor on the changes file, but since it is many megabytes in size, this can be slow. Squeak gives you better ways.

### How to get your code back

Restart Squeak, and select *help>>useful expressions* from the *world* menu. This will give you a workspace full of useful expressions. The first three,

```
Smalltalk recover: 10000.
ChangeList browseRecentLog.
ChangeList browseRecent: 2000.
```

are most useful for recovery.

If you execute `ChangeList browseRecentLog`, you will be given an opportunity to decide how far back in history you wish to browse. Normally, it's sufficient to browse changes as far back as the last Snapshot. (You can get much the same effect by editing `ChangeList browseRecent: 2000` so that the number 2000 is something else, using trial and error.)

Once you have a *recent changes* browser, say, back as far as your last snapshot, you will have a list of *everything* that you have done to Squeak during that time. You can delete items from this list using the yellow-button menu. When you are satisfied, you can *file in* what is left. It's a good idea to start a new change set, using the ordinary change set browser, before you do the *file in*, so that all of your recovered code will be in a new change set. You can then file out this change set.

One useful thing to do in the Recent changes browser is to *remove doIts*. Usually, you won't want to *file in* (and thus re-execute) *doIts*. But there is an exception. Creating a class shows up as a *doIt*. **Before you can file in the methods for a class, the class must exist.** So, if you have created any new classes, **first file in** the class creation *doIts*, **then remove doIts** and *file in* the methods.

When I am done, I like to *file out* my new ChangeSet, quit Squeak without saving the image, restart, and make sure that my new ChangeSet files back in cleanly.

## Worksheet 1B

# Building Applications by Direct Manipulation

---

Closely Based on A Morphic Rolodex Tutorial using Direct Manipulation by John Hinsley, [jhinsley@telinco.co.uk](mailto:jhinsley@telinco.co.uk)  
Version of 22 April 2001. This in turn was inspired by Bolot Kerimbaev's Rolodex System and Dan Shafer's Counter Tutorial, as well as by other sources.  
Version 1.1, 24 April 2001, by Andrew P. Black, [black@cse.ogi.edu](mailto:black@cse.ogi.edu)

---

## Introduction

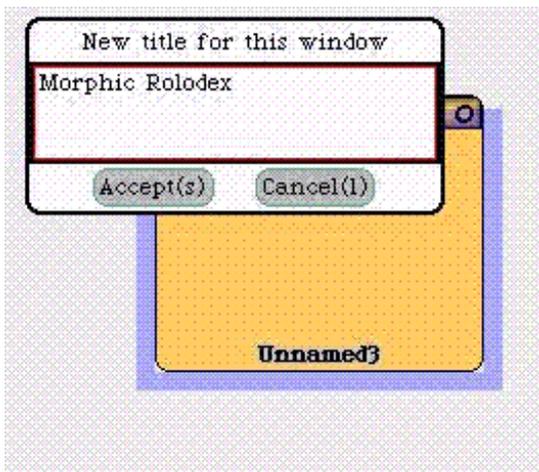
Those already familiar with other Smalltalks will find Worksheet 1A rather boring. It is primarily for those people that I am including this Tutorial. It is about a new and still rather experimental technique for building applications by direct manipulation and scripting.

There is one part of Worksheet 1A that will be useful for those who are new to Squeak even if they are very familiar with other Smalltalks. It is the final subsection entitled [Executing Code: Sending Messages to Objects](#). I suggest that you take a few minutes to do this now. If you have difficulty, it will probably pay you to go back to the [beginning](#) of Worksheet 1A

This Tutorial was written by John Hinsley and tested under Squeak 3.0.

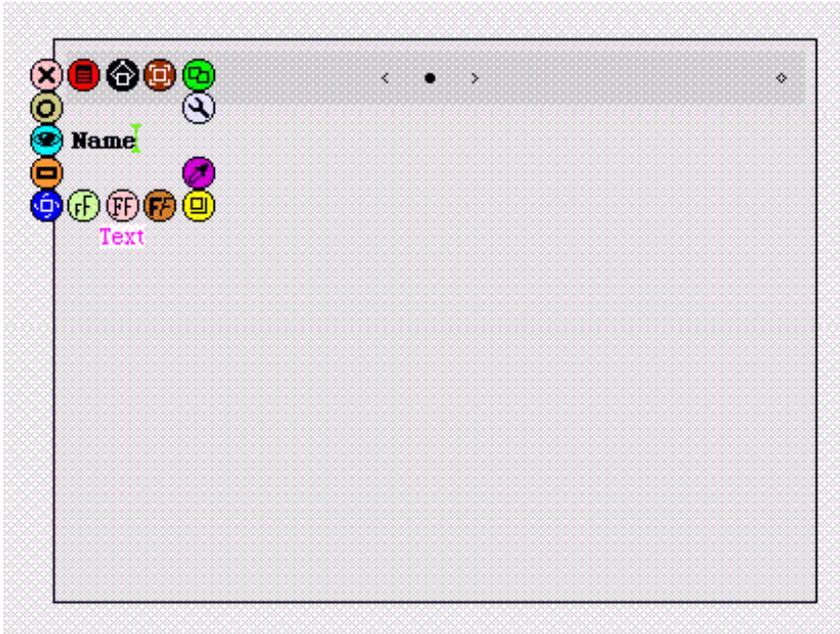
## Getting Started

From Squeak's opening page, red click to bring up the *world* menu, select *open...* and from the *open...* menu select *morphic project*. An orange window will appear. Click on its menu and change the title. (Alternatively, you can just highlight and delete the text at the bottom and type in your title directly.)

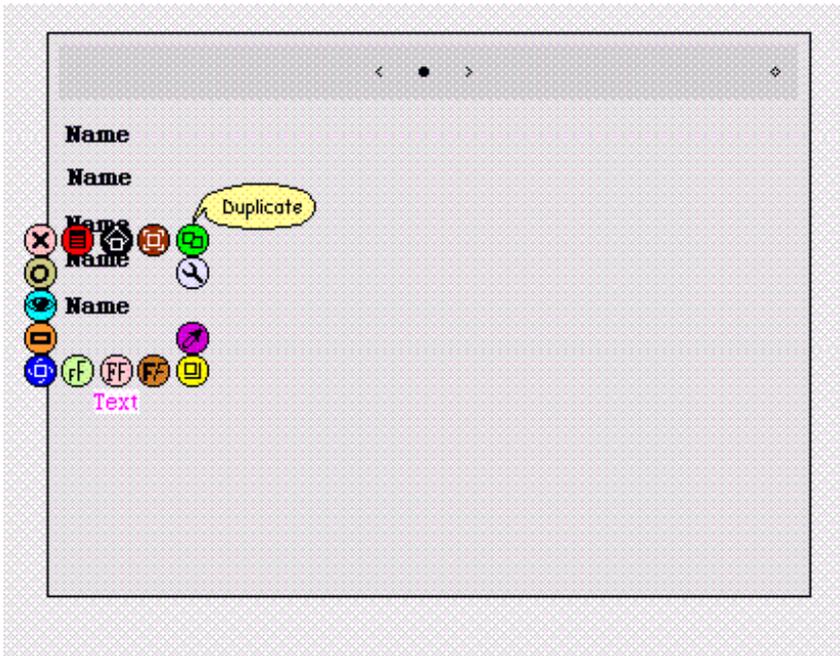


Click in the main pane to enter the project. You'll find yourself inside a blank project.

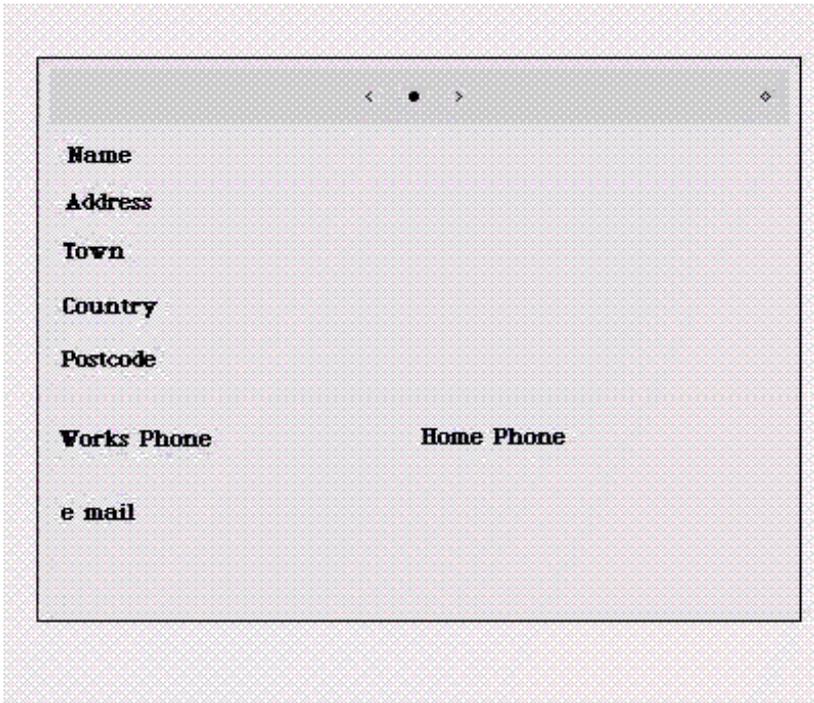
From the supplies tab at the bottom, drag out a blank *BookMorph*. Next, drag out a *TextMorph* ("Text for Editing") from the supplies flap. Blue click until the halo appears around the *TextMorph* (you'll see a little label *Text* at the bottom), highlight the "Text for Editing" and delete it. Type in what you want as your field (for example, "Name").



Now, you could keep dragging TextMorphs out of the supplies flap until all your fields were complete, but it's easier to use the duplicate handle in the halo to do it. All you do is click on the green duplicate handle and use the black pincers handle to relocate the copy. Very occasionally it's possible for something strange to happen during this process, as though you've got one TextMorph on top of another. Just delete it using the pink "x" handle.

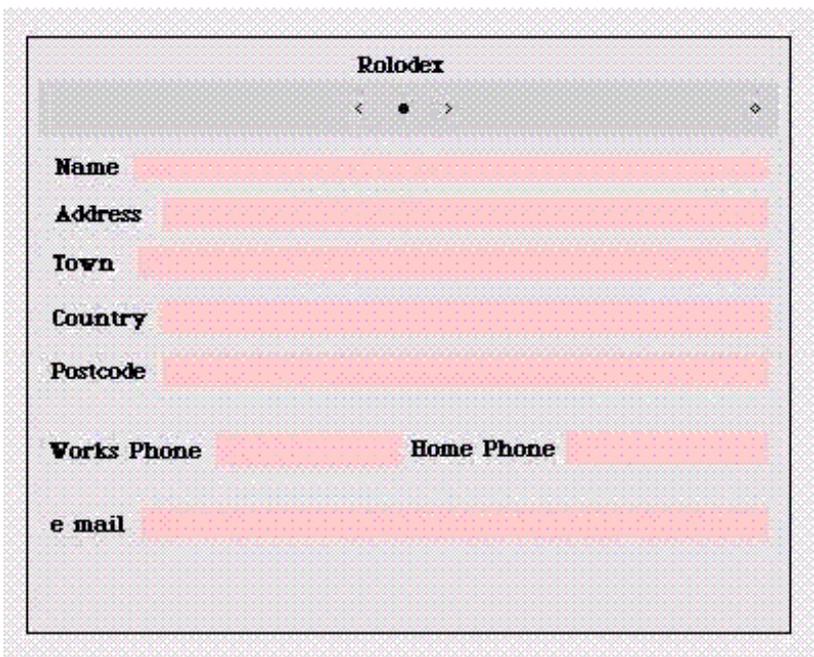


(Incidentally, the duplicate handle is very powerful, much more so than this example can demonstrate.) Change the text to whatever you want to call the field. If the field wraps onto a second line and you don't want it to, use the yellow "change size" handle to drag the string out onto one line by moving it to the right. You may need to use the blue button again to bring up the halo so that you can move something into a better position using the black "pincers" handle.



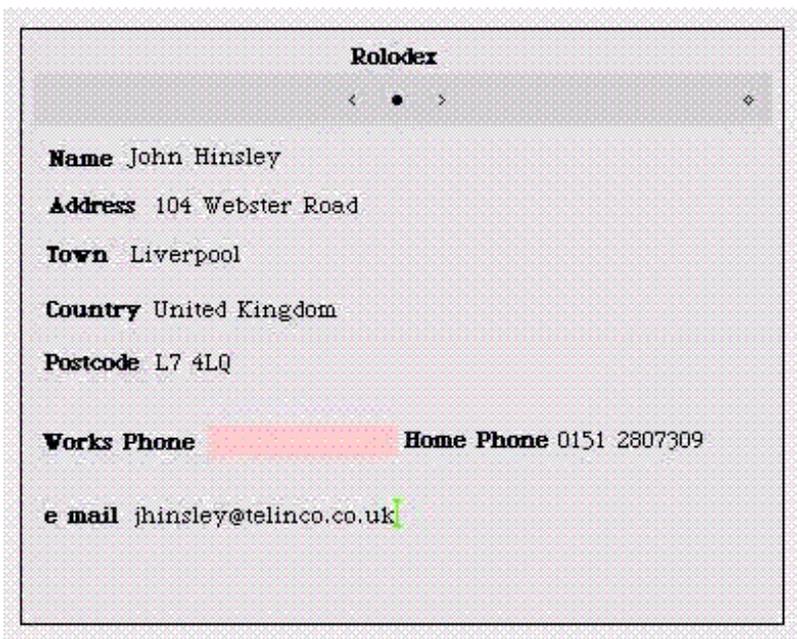
Now try moving the BookMorph about with its pincers. If any of the fields "fall out" (I'm unsure why this seems to happen to some and not others!) you'll need to embed them. Pick them up and drag them back into place and choose *embed* from each one's red handle menu. You'll be given a choice of where to embed them. *aPasteUpMorph* is the white background, so that's no use, while *book* will embed them as the title of the book. The remaining choice is *page*. But since we've discovered how to create a nice title for the book, we may as well use it. Duplicate one more TextMorph, place it anywhere on the book, and change the text to Rolodex. Use the *embed* menu to embed it in the *book*, and voilà! there it is, nicely centered, too. The BookMorph even makes room for it. At this point, red click to bring up the world menu and choose *save*, or *save as...*

So far, we have the labels for the text fields, but no way of entering the text. Guess what we'll use? That's right, blank TextMorphy! So, duplicate one that's already there, delete the entry ("Name", or whatever) entirely, and place it alongside a field label. Use the duplicate handle to make copies, so that every text field has its own blank TextMorph. Here you will need to drag them out to the right with the change size handle. (This time I'm taking no risks and embedding everything, remember, we want to embed it in page, OK?) You should end up with something like this:



Thus far we've done no coding, and we don't need to do any coding at this stage. (In fact, here we're going to use the

"direct manipulation" style throughout.) But we still have a working Rolodex! Now, before you do anything else, click the central dot in the grey bar in the book and choose *save as new-page prototype*: this is really important! Save again, and then red click in the pink bars to enter the details for the text fields for one page:



Now, at the right of the grey bar on the book is a little diamond icon (a bubble help *more controls* pops up). Click on that and then on the + icon (*add another page*) and fill in the name of a friend or two. You'll not only discover that the pages are all there, but that you can use the find dialogue in the book's main menu to find an item.

Our next task is to try and use a button to pop up that dialogue.

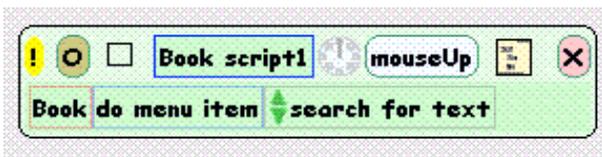
Blue click until the book's halo comes up and click on the blue eye handle. You'll see a menu of scripts appear. Click the top menu icon (it actually looks rather more like a unfurled Roman scroll) and pick *miscellaneous*.



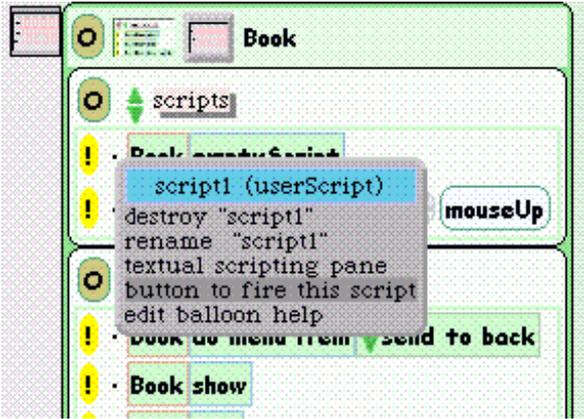
The script that we want is called *Book do menu item*. Drag this script out onto a convenient place in the workspace and click on the up and down arrows on the third element until *search for text* appears.

We can test that this script does, indeed, launch the search dialogue by clicking on the yellow exclamation mark at extreme left. Wow! Now we need to associate that script with a button, and to make sure that the script responds to mouse up (that is to say, it will be fired by the release of the mouse button after the button has been clicked on).

All we need to do to set the mouse up behaviour is to click on the normal tab of this script to bring up a menu from which we select mouse up.



Now, take a look back at the viewer and you'll see that a new item corresponding to our *Book script 1* has appeared. We get our button simply by clicking on the dot to the left of this item and selecting *button to fire this script* from the menu that pops up.

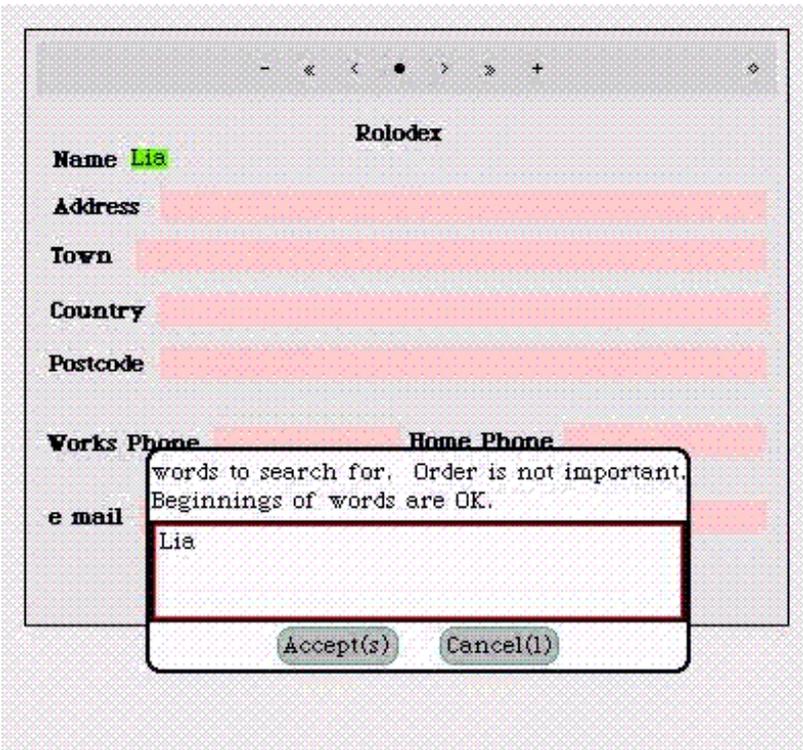


A green button will appear.



We just need to change the label on this script (*Search* or *Find* seem the obvious candidates) and, since I like yellow buttons, change the colour, and embed it in the page. Now, test it by pressing on the button.

Now we need to undo all the text that we have entered to test our searches. From the central *dot* menu of the book, select the page with the Search button in it as the *save as new-page prototype*. Now, enter some names, and try it out!

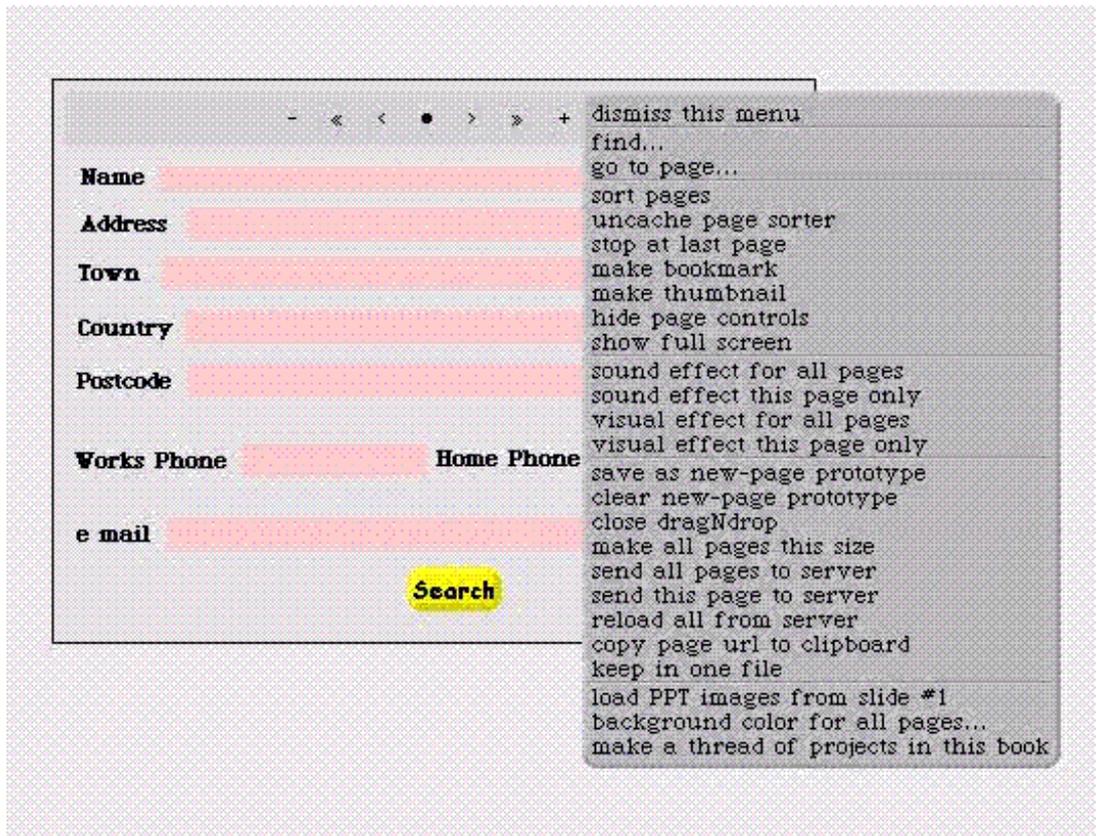


That's it. You've just programmed a fully functional Rolodex with a search function in less than an hour (considerably less if this wasn't your first or second Morph) without entering a line of code.

Now, I'm not going to tell you that it's a particularly difficult project, or that this is all there is to Morphic (and it's only my fourth Morphic project), but I believe that in any other language this would take at least twice, and probably three times, as long.

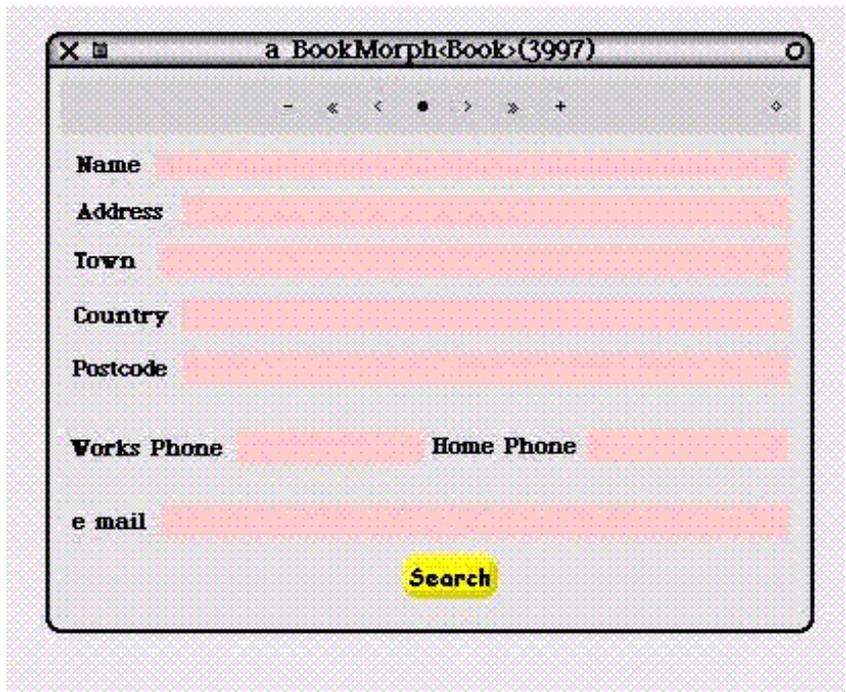
# A Postscript

Eventually I got fed up with the strange way that "Rolodex", although embedded in the book, comes back as a sort of page heading. So let's remove that. Before we once more *save as new-page prototype* we should really get rid of all the little bits of text we've put in.



For our next trick, we'll put the book in a window. We do this simply by opting to do that from the book's red menu handle.

Fortunately the book will auto-embed in the window! Now we just need to change the window's title (again, through the red menu handle). That's it for now.

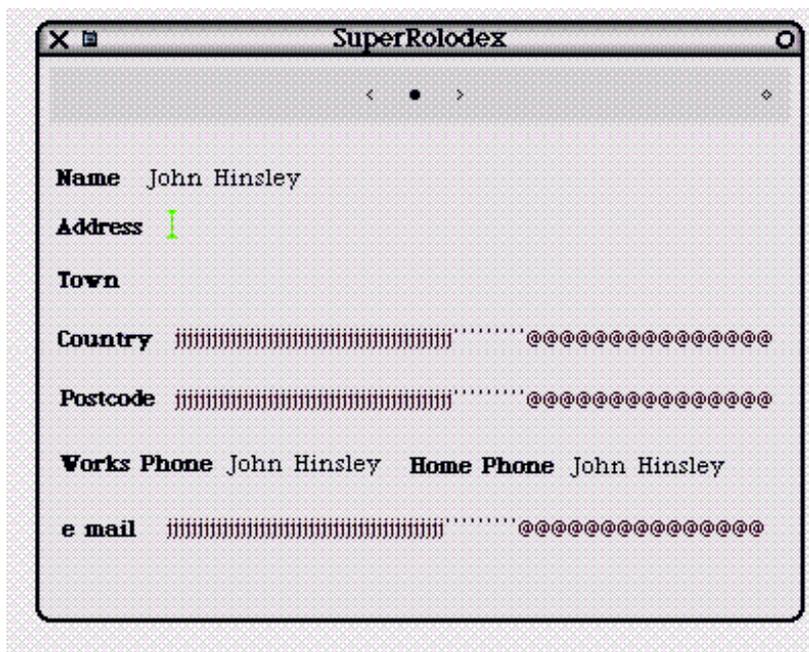


Don't forget to save your image now, and each time you put a new entry in your Rolodex.

## An Enhancement: Visibility and Affordance!

One of the people who responded to the first version of this tutorial was having problems entering text in the empty TextMorphs (the pink bar). I wasn't sure whether this was due to my description of the process (since revised) or an inherent fault in the interface. Now, on the course I'm doing at the Open University we're taught to assess interfaces in terms of visibility, affordance and feedback. I won't go into the details of that (because I tend to get confused between one and the other!), but if we look at the pink bar, it's not at all clear that it will accept focus or that it will take text. The only feedback we get is that if we do get focus on a TextMorph which is blank and start typing, text appears. I messed about (as one must) with some of the other text morphs from the *Add a new morph* menu and finally found ShowEmptyTextMorph (under S in the *from alphabetical list* sub menu). This both allows focus and shows you have it by a flashing cursor. (If you like, the cursor "affords" that we can enter text.)

If you like the sound of this, all you have to do is to delete the blank TextMorphs and replace them with ShowEmptyTextMorphs. It's easier to manipulate a ShowEmptyTextMorph if you actually fill it in with some text. The illustration shows one field completed, some just full of text for the purpose of manipulation/justification, and another with focus (you should have been able to see the green cursor next to "Address") waiting for input.



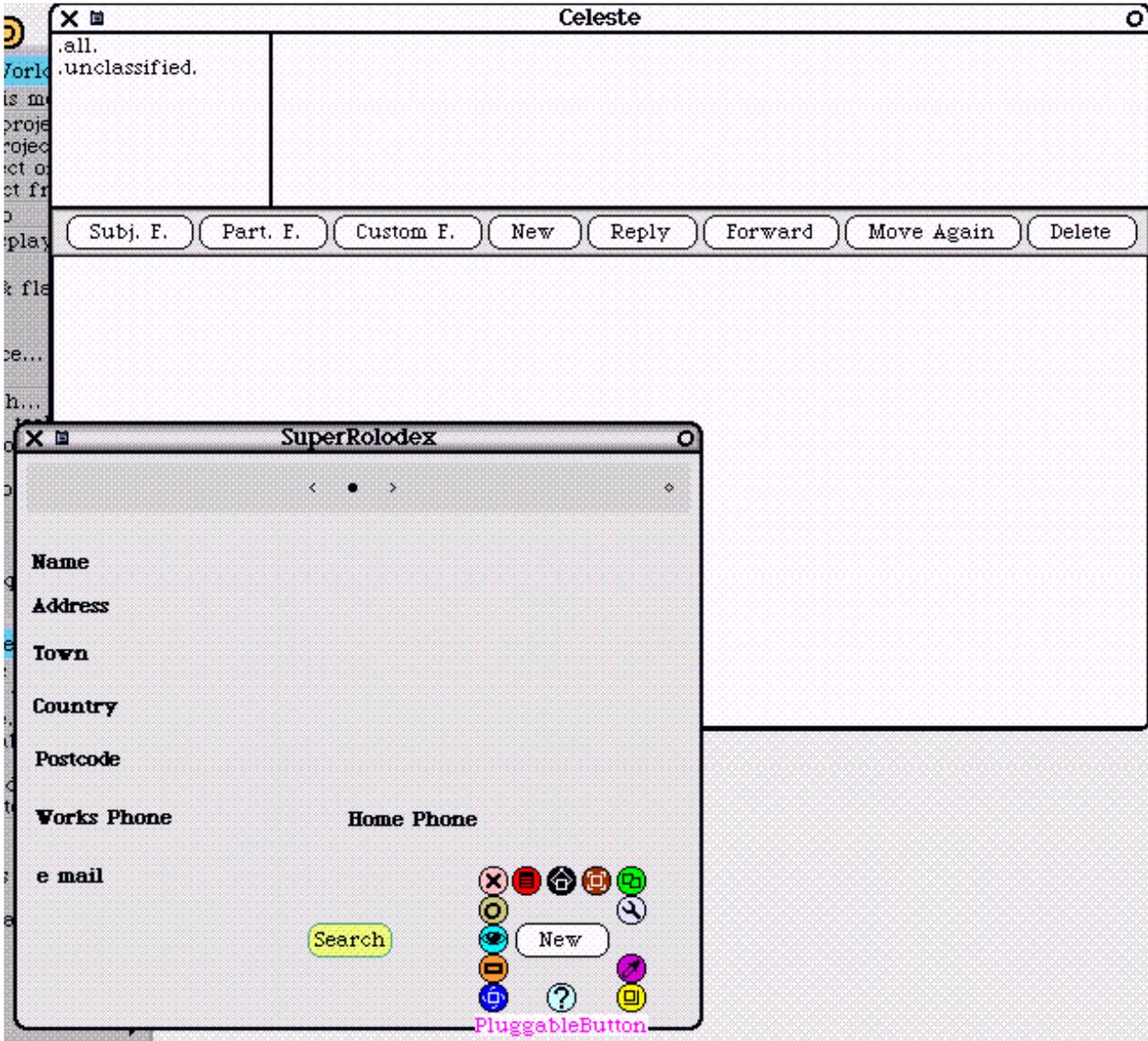
You'll note that on this SuperRolodex version, I've yet to put the "Search" button.

Again, don't forget to save the page as a *new-page prototype*, and then to *save* your image.

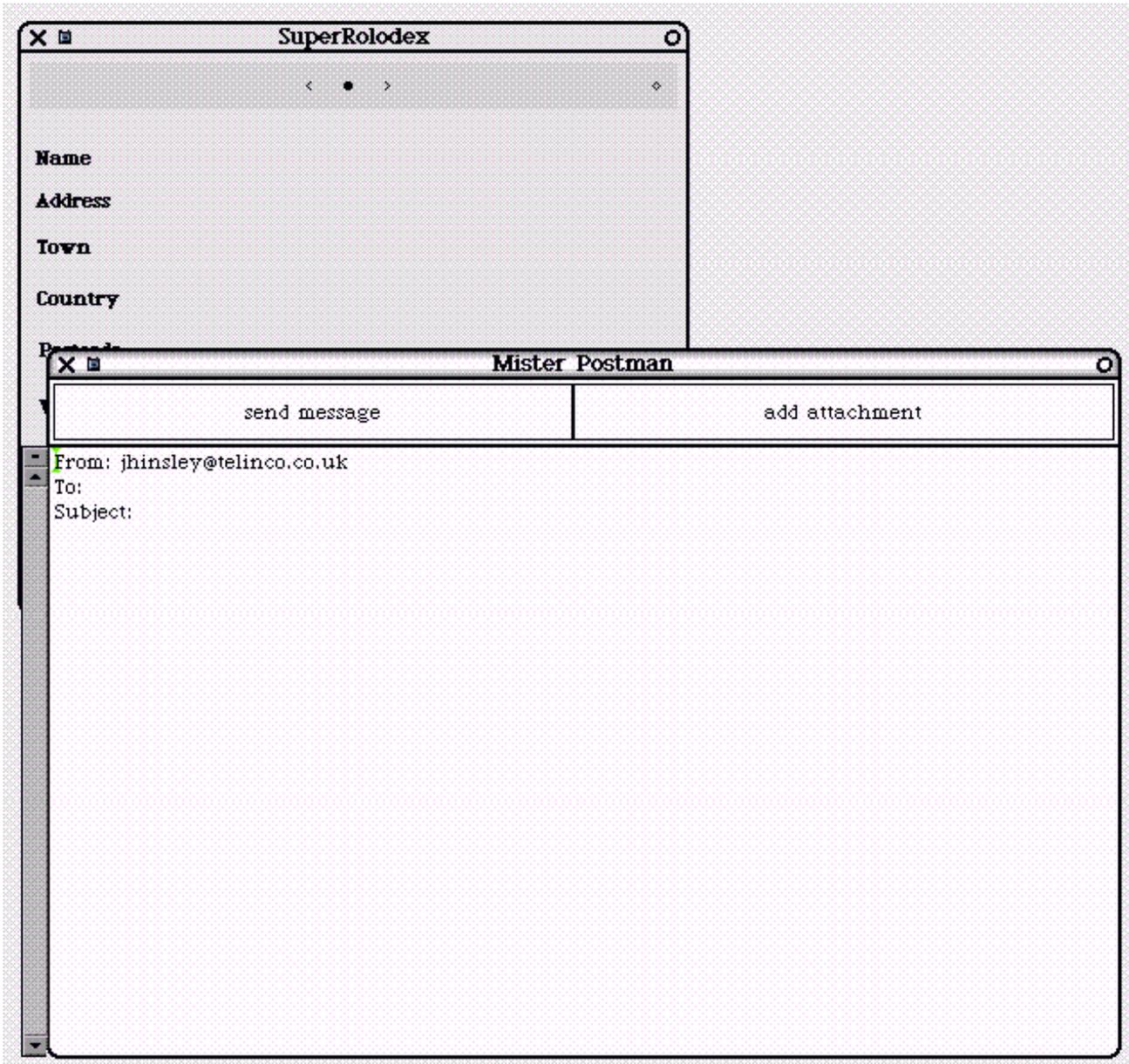
## Onwards and Upwards: adding an email function

Squeak already has a nice little e mailer called Celeste. It's somewhat minimal (folk are working on enhancements such as a mail sorter as I write), but is extremely elegant and just what we need to add a little functionality to our Rolodex.

I have to admit that how to get Rolodex to call Celeste's "compose" page (wonderfully named Mister Postman), puzzled me for a good few days. In the end, I decided to try the simplest solution I could think of. All you need to do is to call Celeste from the *world menu* >> *open...* >> *email reader*.



You'll see that Celeste has a "New" button. Try it out and you'll find that this button calls Mister Postman. (While you're in Celeste you might like to fill in your e-mail address and the SMTP and POP3 servers) We can simply get a halo around this "New" button, duplicate it, drag it down to the Rolodex and embed it (in Page, remember). Close down Celeste and try your new "New" button. There you go, an e-mail function!



What we need to do now is to sort out how to get Mister Postman's To: field to contain the text in the ShowEmptyTextMorph for the email field in that page. Bear with me, I may be gone for some time...

## For the Future...

So, what remains to be done? I'm still not sure how to change the colour or the font of the buttons on the search dialogue (and have them stay changed, that is). It would be nice if clicking on a 'phone number brought up an autodial option. I'd like to be able to do stand alone versions (but these would presumably need a save dialogue too, so that new entries were not lost). Also I think that a finished, stand alone version shouldn't have any superfluous menus. Removing them without wrecking everything may be a little tricky!

In effect, the idea would be to create a program that has all the functionality of the Card Index program that came with Windows 3.1 (surely one of the best Windows programs ever!) and more, but to do it entirely by direct manipulation.

Any comments, criticisms or ideas on how to improve the functionality of this little project will be gratefully received, please [e mail](#) me.

In the meantime, may the Morph be with you!

## Worksheet 2

# The Morphic Graphics System

---

Based on Tutorial: Fun with the Morphic Graphics System by John Maloney  
Version 1.0, 17 April 2001, by Andrew P. Black, [black@cse.ogi.edu](mailto:black@cse.ogi.edu)

---

## Introduction

In this worksheet you will build a Morphic object by writing code (as opposed to direct manipulation). We're going to start with an empty class that is a subclass of Morph. Morph is the generic graphical class in the Morphic object system.

Open a Browser (from the world menu >> open submenu, or drag one from the tools flap) select the category for your stuff that you created previously. (If you don't have such a category, create one now — see the box.) Make sure that none of the other browser panes have stuff selected. The bottom pane of the browser will be showing the template for creating a new class. Make it look like this:

```
Morph subclass: #TestMorph
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'ECOOP-MyStuff'
```

First select Object (double-click it) and type the word Morph with a capital M. The new class will be a subclass of Morph. Select NameOfSubClass" and type in "TestMorph". The category name will already be filled in, and will probably be different from mine!. Type *Command-s* to accept (save), or select *accept (s)* from the yellow button menu.

Now open a workspace (or switch to an existing workspace) and create an instance of your new class. Type

```
TestMorph new openInWorld.
```

Hit *Command-d* or choose "do It" from the yellow button menu. Since nothing was selected, the "doIt" applies to the whole of the line that the cursor is on.

Your new Morph (graphical object), a blue square, is in the upper left corner of the display. You can pick this object up and move it around. Just grab it with the mouse. Notice how, when you picked up this Morphic object, it threw a dark shadow behind it. That helps you to see that you've actually lifted it up above everything else. If there were overlapping objects, grabbing this object would automatically pull it to the front, and the shadow would show that it's in front of everything else.

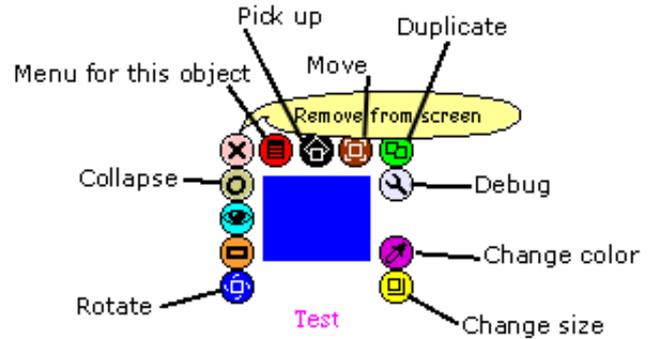
You'll notice the odd capitalization of 'openInWorld'. Smalltalkers place a high value on readability, but in Smalltalk a message selector must be a single string without any spaces in it. The convention is to use capital letters at the beginning of each English word except for the first, and to stitching them all together without spaces. Other languages often use underbars or dashes between the words. Squeak does not. You will probably grow to like this convention, but you should adopt it even if you don't!

## The Morphic Halo

Now click the blue mouse button on your Morph, and you will see a surrounding array of colored circles, which

we call a "Morphic halo". Some of these circles are buttons, and some are handles; each of them is a quick way to send a command to the Morph. If you linger over any of the dots with the mouse, a help balloon will pop up telling you what the dot does.

The black handle's balloon says: "Pick up". In this case, if you drag the black handle, the effect is the same as if you dragged the blue rectangle itself. That may seem like a useless function, but the reason that it is there is that sometimes Morphs will have an interactive behavior, like a button. The pick up handle gives you a way of directly manipulating something that has behavior associated with mouse clicks.



The basic idea behind the halo is that there is no need to resort to "modes" for changing the size of an object, etc.

That is, Morphic does not have a separate mode for editing an object as opposed to using it. In Morphic, you can get a halo on anything regardless of how active it is. The halo is a safe way of dealing with a Morph while it is running. For example, there is no need to turn off the function of a button in order to change its appearance.

Let's look at what the halo can do. The green duplicate handle is a button that makes a copy, and the pink handle with the x deletes the Morph (moves it to the trash). The yellow handle is used to resize the object—try dragging it. Resist the temptation to do too many other things to it yet; in particular, don't use the blue rotate handle.

## Adding Custom Behaviour

Now let's start making this our own kind of object, by writing some methods to customize it.

The first thing you will do is make your object do something when you click on it with the mouse. This involves writing two methods. Click on the Browser. Click on "no messages" in the third pane. Type (or paste) the following text into the bottom pane, and *Accept*. (Beware: cutting text from some web browsers includes some invisible garbage characters; if the Squeak compiler doesn't like this text, that may be the problem).

```
handlesMouseDown: evt
    ^ true
```

If this is the first time that you have accepted a method in this image, you will be asked to type your initials, which will be recorded along with the methods that you change.

The fact that this method, `handlesMouseDown:`, answers `true` tells Squeak that this object wants to receive a message when a mouse button is pressed over it. (Pressing a mouse button generates a `mouseDown` event; guess what you get when a mouse button is released!)

So now we better define a method corresponding to that message. Go ahead and type (or paste) the following text right over the other method.

```
mouseDown: evt
    self position: self position + (10 @ 0).
```

This will move the object ten pixels to the right when you click on it. How's that? Well, the infix `@` notation is used to build a `Point` from two numbers. For example, `2@3` is a `Point`, with x coordinate 2 and y coordinate 3. The message `position`, when sent to a `Morph`, answers a `Point` that is the `Morph`'s current position. And `+` is defined on `Points` to do elementwise addition. Note that `position:` is a different method; this one takes a `Point` as argument and *changes* the position of the receiver.

Try clicking on the object; see what happens.

To summarize: at this point we have two methods. One of them returns `true` to say that we want to handle `mouseDown:` events. The second one responds to the `mouseDown` event by changing the object's position.

Now lets give this Morph some behavior for redrawing itself. Type or paste this method into the bottom pane of the browser and *accept* it. Rather than worrying about indentation, if you wish you can choose *pretty print* from the yellow-button menu *more...* menu.

```
drawOn: aCanvas
  | colors |
  colors := Color wheel: 10.
  colors withIndexDo: [:c :i |
    aCanvas fillOval: (self bounds insetBy: self width // 25 * i + 1)
      color: c].
```

Now when you click on the object it looks totally different. When the screen needs to be redrawn, the `drawOn:` message is sent by the world to all of the objects in it. So, by defining this method for our Morph, we are defining its appearance. Let's go back and look at what this method is doing.

## Understanding the drawOn: method

The first line, `drawOn: aCanvas`, is a heading that gives the name of the method `drawOn:` and a name for the formal parameter, `aCanvas`. After the heading, the second line of the method, `| colors |`, declares a local variable `colors`. Actually, there is no need to type this line; if you omit it, the compiler will offer to add it for you. Remember, Smalltalk has no type declarations — any variable can name any kind of object. Smalltalk programmers often choose identifiers for variables based on the types of objects that they will name.

The third line of the method is the first line of executable code: it assigns to `colors` the result of the expression `Color wheel: 10`. To see what that does, select `wheel:` and type *Command-m* for *immMMMMplementers*. You will see that there are two implementations of `wheel:` select the one in `Color` class.

One thing we often do in Squeak is we put a little comment at the head of a method. The first line says what `wheel:` does. It returns a collection of `thisMany` colors, where `thisMany` is the argument. The colors are evenly spaced around the color wheel.

When it's easy to give an example of the method in action, Smalltalkers do that too. The second line here is an expression, which you can actually execute. Select the quoted text (by double-clicking just inside the quotes) and *doIt* (*command-d*). Across the top of the screen Squeak just sprays out the color wheel—a collection of colors spaced around the spectrum. (But not with total saturation nor full brightness, so they won't be too garish.)

Notice that you can scribble directly on the screen in Squeak—you don't have to be in some sort of window to draw. To clean up such scribbles, use the *restore display* item in the world menu (*command-r*).

This was a digression to figure out what `wheel:` meant. We now know that it's a collection of Colors. Close the *Implementors of wheel:* window by clicking in the "X" on the left side of the title bar.

The next part of the `drawOn:` method steps through that set of colors.

*This is a fragment of the method drawOn:, which you have already typed:*

```
colors withIndexDo: [:c :i |
  aCanvas fillOval: (self bounds insetBy: self width // 25 * i + 1)
    color: c].
```

The `withIndexDo:` method evaluates the block between `[` and `]` ten times. It supplies as arguments to the block both an element of `colors`, `c`, and an index, `i`. The index `i` is needed to keep track of where we are in the collection of colors; `i` will be 1 the first time around, 2 the second, and so on, all the way through to 10.

Inside the block, we send `aCanvas` the message, `fillOval:color:` with two arguments: the mess in parens and `c`, our color. What about that mess in parens? It is a Rectangle inset by  $((self\ width) // 25) * i + 1$  pixels inside the bounding Rectangle returned by `(self bounds)`. As the `i` increases, the inset also increases, returning ever smaller nested Rectangles, in each of which we inscribe an oval and fill it with the color `c`.

When we look at our object, we see up to ten bands of color. Because the `drawOn:` method is using the result of the message `width` being sent to the Morph itself, this code "knows" how big the Morph is. Bring up the halo, and drag on the yellow dot. As you resize the Morph, it just keeps redrawing itself at the new size.

## Animating the Morph

The next thing to do is make clicking on our Morph invoke a more exciting kind of motion—some animation. This means that we need to create a list of points that will be the path over which the object moves. When you click on the object, we want it to move to each point in the path in turn. So, we will need some state in our object, so that it can remember its path.

```
Morph subclass: #TestMorph
  instanceVariableNames: 'path'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'ECOOP-MyStuff'
```

Go back and click in the Browser, and then on the "instance" button (at the bottom of the second pane). That will bring up the TestMorph class definition. Then click in the place for instance variable names, between the single quotes, and type `path`. *Accept* it.

## Initializing an Instance Variable

Since you added an instance variable, you need to make sure that it's initialized to something. Let's define an initialization message for TestMorph. Click on the message category whose name has changed to "as yet unclassified", and then type (or paste) the following text in the bottom pane of the browser.

```
initialize
  super initialize.
  path := OrderedCollection new.
```

Why is the line `super initialize` here? Even though the definition of class TestMorph is very simple, `path` is not the only instance variable that it has. In the second pane of the Browser, bring up the yellow button menu and choose *inst var refs*. You see a list of all the instance variables in TestMorph. `path` is at the bottom, but above that is a list of other instance variables, which TestMorph inherits from Morph. There is no need right now to select any of them, but if you do, Squeak will show you all the methods that use that instance variable.

So, since Morph has a bunch of instance variables, there is a good bet that it will have an `initialize` method that sets some or all of them to good values. (Take a look at it if you wish; it's fine to open several more browsers). When we define our own `initialize` method, we *override* the one in our superclass, class Morph. So unless we do something special, the `initialize` method in class Morph will no longer execute. This would be bad!

The "something special" is the `super initialize`. This executes the `initialize` method of our superclass, class Morph. When you send the message `initialize` to the receiver `super`, the message is really sent to `self`, but Squeak makes sure that the *inherited* version of `initialize` is invoked instead of the one that is executing.

After taking care of that small but vital detail, the guts of our `initialize` method is the assignment `path := OrderedCollection new`. This creates a new, empty, Collection and puts it into `path`. So, when our TestMorph receives the `initialize` message, it first does all the initialization that Morph would do, and then it does its own initialization. The result is that all those instance variables inherited from class Morph will be initialized as well as the new one that you added.

We are actually using a convention here, since a new object is not automatically sent the `initialize` message. But if you look at the new method for class Morph, you will see that it always sends `initialize` to each new Morph instance. This is how many, many objects are coded.

Now, our original TestMorph, here on the screen, was not initialized with our new code: we made it before we defined our version of `initialize`. We can fix this in two ways: with an *inspector*, or by deleting the Morph and making a new one.

## Using an Inspector

The simplest thing is just to delete the TestMorph. But before we do that, lets see what an inspector can do for us. Inspectors are really useful tools for looking at and changing individual objects, so it is worth getting to know what they can do.

Bring up the Halo on our TestMorph, click on the debug handle  and select *inspect morph* from the debug menu. An *inspector* will attach itself to the mouse cursor. Put the inspector down in a convenient place. The left pane contains a list of all of the instance variables in our TestMorph, including `path` and those inherited from Morph. Click on `path`, and you will see (in the right pane) that its value is `nil`, that is, `path` has not been initialized.

You can fix this right now by selecting `nil` and replacing it with `OrderedCollection new`. *Accept*; what you typed will be evaluated, and the result, a new empty collection, will be assigned to `path`. The text will change to an `OrderedCollection()`, which is the way that empty collections print themselves.

By the way, the bottom pane of the inspector is a little workspace. You can type a Smalltalk expression there and *do it* or *print it*. The useful thing about this workspace is that, here, the pseudo-variable `self` names the object that you are inspecting. So you can type `self color` and *print it* and see the result of sending the `color` message to this very TestMorph.

## Meanwhile, back to the Animation...

OK, so we are done with the inspector, and with our original TestMorph. Dismiss them both (using the `x` in the menu bar of the inspector, and the pink button with the `x` in the TestMorph's halo). Now get a new TestMorph (properly initialized this time) by executing:

```
TestMorph new openInWorld.
```

We're going to make our object do something different when you click the mouse on it. We've got the `Collection` in `path`. Type into the bottom pane of the browser and *accept* this method:

```
startAnimation
  path reset.
  0 to: 9 do: [:i | path add: self position + (0@(10 * i))].
  path := path, path reverse.
  self startStepping.
```

We are resetting our `OrderedCollection` in line 2, just to make sure that it is empty. In the next line, we're doing a block ten times. See if you can figure out what it is doing! Remember, `0@30` is a `Point`, with `x` coordinate 0 and `y` coordinate 30.

In line 3, we change the `path` to be `(path, path reverse)`. I'm sure that you can guess what sending `reverse` to an `OrderedCollection` does! The comma `,` is just another message; it concatenates two collections.

What we've done is taken our list of ten points which are going from zero to ninety, and added on a list that goes from ninety back to zero. So now we've got twenty `Points` that start out and end up at the original position, and go to a bunch of places in between.

The last line of this method enrolls our morph in an engine that keeps sending it the message `step`. This is the "tick" of the animation engine. So now we must make our morph understand the `step` message.

```
step
  path size > 0 ifTrue: [self position: path removeFirst].
```

This one is pretty easy to understand. It says that as long as there is something in that list of points, that is, as long

as the size of `path` is greater than zero, then move myself. `path removeFirst` does two things: it *removes* the first point from `path`, making `path` a collection with one less element, and it *answers* the Point that it just removed. So, the argument to `self position:` is the first point that was in the path. Sending myself the `position:` message changes my position.

Graphically, the effect is that our TestMorph it will jump to the next point; you will see the change on the next frame of the animation.

## Starting the Animation

All that we need do now is actually kick our TestMorph off off by sending it the message `startAnimation`. One way to do this is to use a temporary variable in the Workspace. Type these lines in the Workspace, select them, and *doIt*.

```
t := TestMorph new openInWorld.
t startAnimation
```

You could also bring up an inspector on your TestMorph, and *accept self startAnimation* in the bottom pane.

A graphical alternative to all this typing is to make our TestMorph send itself the `startAnimation` message when the mouse button goes down.

```
mouseDown: evt
  self startAnimation.
```

There are two ways you can define this. You can paste it over any method showing in the bottom pane of the Browser. Or you can click on *mouseDown:* in the right pane of the Browser, and modify the existing `mouseDown:` method. Either way, the same thing happens when you *accept*.

When you've done that, go over and click on your morph, and you should see something happen. It's animating, which is great, but it's going very, very slowly. Don't be too disappointed; Squeak is faster than this! What is actually happening is that, by default, `step` is sent to an object once every second. But the object can decide how often it wants to be told to `step` by implementing the method `stepTime`.

```
stepTime
  ^ 50
```

This answers the time in milliseconds between `step` messages that the object is requesting. Now, obviously, you can ask for zero time between `step` messages, but you won't get it. Basically, `stepTime` is an assertion of how often you would *like* to be stepped.

Now if we click on our object we get a much faster animation. Change it to 10 milliseconds, and try to get a hundred frames a second. Let's see if that works. If you are really interested in knowing how fast your animation is running, see if you can find out how to record the times at which your Morph receives the `step` message.

## More Realistic Motion

Now, this tutorial was originally written by John Maloney, who works for Disney, "The Animation Company". Our animation starts and stops instantly, but everyone at Disney knows that "slow in" and "slow out" are needed to make it look really good. Here is a modification that uses the square of `i` to start the motion off slowly.

```
startAnimation
  path reset.
  0 to: 29 do: [:i | path add: self position + (0@(i squared / 5.0))].
  path := path, path reversed.
  self startStepping.
```

Only one line in the `startAnimation` method needs to be changed. This code puts thirty points in the path, and then appends the reverse version to make sixty points. The other change is that instead of saying times ten, we say "i squared divided by five". At the end, the expression will be  $841 (29^2)$  divided by 5.

This gives us an animation that starts down slowly and speeds up, and then slows down again as it comes back up. In fact, it looks a bit like a ball bouncing.

## More Morphs

Now would be a good time to play with some pre-made Morphic objects on your own. Let's get a parts bin full of objects. Click on the "Supplies" tab at the bottom of the screen. (If there is no tab at the bottom of the screen, use the World menu and choose "authoring tools...". Choose "standard parts bin".) In either case you have a bunch of objects that you can drag onto the screen. After you have dragged one out, blue-click to get a halo, and hold down on the red circle to get the object's content menu. Here are some of the Morphs to play with:

- A `RectangleMorph` (from the contents menu, try *raised bevel*, *inset bevel* and *rounded corners*)
- An `EllipseMorph`
- A `StarMorph`
- A `CurveMorph` (from the contents menu, try *show handles*)
- A `PolygonMorph` (from the contents menu, try *show handles*)
- A `TextMorph` (be sure to use the yellow handle to make it wider)
- The artist's palette starts a new painting where you drop it. After drawing, press *Keep* to turn it into a Morph.

In this worksheet, you have been introduced to morphic programming. You probably noticed some of the other parts of the Morphic Halo that we did not talk about, like the eye and the rectangle buttons. These give access to an entirely different way of creating applications: not by programming, but by direct manipulation. Feel free to experiment! Worksheet 1b gives an example of building a complete application by direct manipulation — if you haven't had a chance to try it out yet, do so! There are also other tutorials available on the web that will introduce you to this style of application building.

## A Final Example

Use a file list browser to find the file *Hungary.gif*. From the yellow-button menu, choose *open image in a window*. The resulting map is a full-fledged Morph (an instance of *SketchMorph*).

You should already have filed-in *wkSheet.cs*—if you have not, file it in now. In a workspace, type and execute

```
Hungary new openInWorld.
```

The result will be an instance of *HungaryMorph*! Superimpose it on the map. Use the Supplies tab to drag in other Morphs that help you to complete Hungary. For example, you can drag in a *Text for Editing*, change the text to read *Budapest*, and place it in the appropriate position on the map. Then use the red handle in the morphic halo to bring up a menu on the *TextMorph*, and *embed ...* it into *Hungary*. Now you will be able to move *Hungary*, and *Budapest* will move with it.

---

## Worksheet 3

# A Small Programming Project

Version 1.0, 23 April 2001, by Andrew P. Black, [black@cse.ogi.edu](mailto:black@cse.ogi.edu)

## Introduction

This worksheet is less of a step-by-step "cook book" than the previous ones. Instead, we suggest two projects, and give you some hints as to how to program them in Squeak using Morphic. You probably won't have time to do both, so pick the one that looks more interesting to you. Of course, if you and your partner wish, you could tackle some completely different project.

I'm not going to talk about design at all. This is not because design is unimportant—quite the contrary! But I'm assuming that you are proficient in OO design and OO programming in another language.

## Project 3A: An Analogue Clock

The goal in this project is to implement a clock, the kind with a face and two (or three) hands. You can make it look pretty much as you like, but it should, of course, keep time! Here are some suggestions to make the project more interesting.



- Rather than showing local time, you might give your clock an offset, controllable from the UI, so that it shows the time in your home town. Add a label that identifies the town.
- Provide an option to make it a 24 hour clock, that is, make the hour hand go around once every 24 hours rather than twice. Or, change the numerals on the face in the afternoon.
- Allow the clock face to be a shape that is not round. The hands will need to change length as they go around!

## Getting Started

OK, how might you get started? It is always good to "shop" for functionality that others have already built, which you can reuse. In fact, the normal way of figuring out how to use some piece of a Smalltalk system is to do a *senders of ...* and see how others are already using it.

In this case, if you do a *find ... "clock"* in a system browser, you will find that there is already a `ClockMorph`. It's a digital clock, but worth examining nonetheless. `ClockMorph` uses a number of techniques that you have already seen, for example, a `step` method that is called once every second. Notice the use of `Time now` to obtain the time of day from the system clock. Why is this preferable to simply adding one second to the previous time?

`ClockMorph` is a subclass of `StringMorph`, which already handles the updating of the display. So, when the contents of the Morph is changed (by sending the message `self contents: time` in the method for `step`), the display will change immediately.

Incidentally, you can make a new instance of any of the Morph classes using the `world>>new morph...` menu. Some of the more common Morphs are also available from the `supplies` flap at the bottom of the screen

It turns out that there is also a `WatchMorph` in Squeak, which already implements an analogue clock. But try not to look at it just yet! See how far you can get figuring things out on your own.

The best bet for a clock face would seem to be an ellipse; you can drag an `EllipseMorph` out of the supplies flap. (When you blue-click on a Morph, the name of its class appears on the screen below it, without the *Morph* suffix.)

Think back to the bouncing colored ball in worksheet 2. In that worksheet, we changed the appearance of our `TestMorph` in arbitrary ways by overriding its `drawOn:` method. Could you use the same idea to draw the hands on the face of your ellipse? Previously, we used the `fillOval:color:` method to do the painting. Use the *method finder* to open a system browser on that method—you will see that it is in class `Canvas`. The browser will show you similar methods in other categories that draw rectangles and polygons.

What about the face of the clock? Remember that you can embed one Morph inside another. String Morphs can be used to place numerals in the right places. Or, perhaps you just want to draw tick marks every 30° around the face.

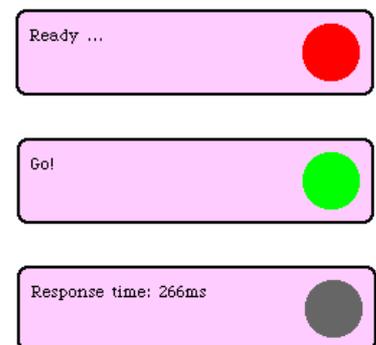
You know that positions on the display are represented by `Points`, like `100@50`. But points also understand polar coordinates, which are sometimes more convenient. For example, `Point r: 2 sqrt degrees: 45` prints as `1.0@1.0`.

I think that this should be enough to get you started ...

## Project 3B: The Reflex game

The reflex game is like one of those machines that you might have seen in a science museum or an arcade; it measures the speed of your reflexes.

1. The game starts by showing you a red light.
2. After a random time interval, the light turns green. This is an invitation to click the mouse button as quickly as you can.
3. The game measures how quickly you respond. Then it turns out the green light, and shows your reaction time on an LED display.



What if the user never clicks the mouse at all? Eventually, the game should time out, and reset itself back to the initial state. What if the user cheats, and clicks the mouse before the light turns green? The game should detect this.

This very simple game can be embellished in a number of ways.

- Have it record the results of three trials per user, and report the best.
- In addition to, or instead of, having the light turn green, have the game make a sound. Or have it say "Go!". Which gives the shortest reaction time?
- Turn this into a competition. Have two users on opposite sides of the keyboard, one on each shift key. When the light turns green, see which user presses their key first.

The display for this game shown here is quite simple—just an `EllipseMorph` and a `StringMorph` embedded in a `RectangleMorph`. I built this first by direct manipulation. Once I had a layout that I liked, copied the coordinates out of an explorer into my `initialize` method. `Morphic` supports quite complex dynamic 2D tabular layouts, but I won't discuss them here.

Most of the complexity of the reflex game is in the testing and timing. For this reason I separated those aspects from the user interface part. I used `Morphic` to provide the "view" and the "controller", and a separate model object to implement the timing part. Of course, there are many other valid architectures for this game.

My model was a state machine. Messages to the model caused it to change state. One might use the *state pattern* to implement this, or just a simple instance variable whose value (a `Symbol`) represents the state. In the latter case, the methods implementing the state transitions will have bodies that do a case selection on the state. (Yes, Squeak does have a case statement; it is described on the [Squeak Language Reference](#) sheet.)

Squeak has a class `Random` whose instances are random number *generators*. As with many classes, there is an `example` method in `Random` class (that's on the *class side*) that tells you how to use `Randoms`.

The class `Time` has many useful methods, including `Time millisecondClockValue`, and `Time milliseconds: since:`, which computes the difference between two times. Another useful class is `Delay`. A `Delay` is an object on which the current process can wait. Once again, there is an example method on `Delay` class that shows how a `Delay` can be used.

Along with delays, you will need to create processes. This could hardly be easier: just send a block the message `fork`. The `Delay` class example also illustrated the use of `fork`, or rather `forkAt:`, which allows you to specify a priority.

## Events

I found it convenient to create a new class, `Event`, which has a single method on the class side:

```
Event class>>afterMilliseconds: anInteger then: aBlock

    "evaluate aBlock (in a new process) after waiting for anInteger
    milliseconds. This method returns immediately, answering the new
    process."
    | d |
    d := self forMilliseconds: anInteger.
    ^ [d wait. aBlock value] fork
```

Then an object could cause its `timeOut` method to be invoked asynchronously at a time 3 seconds in the future by executing

```
Event afterMilliseconds: 3000 then: [self timeOut]
```

## Linking the Model and the View

Those of you who are familiar with the Model-View-Controller paradigm also know that it is traditional to separate the model from the view. In this example, such a separation makes it hard to do accurate timing in the model, because the model will only affect the view (*e.g.*, turn on the green light) indirectly. For this reason, I felt that it was appropriate for the model and the view to be closely coupled. My model has direct access to the display and the light of the reflex game morph.

## Making a Noise

If you have sound on your Squeak platform, you might try *speaking* the word 'Go'. Look at the examples in the class `Speaker`, or just try `Speaker default say: 'Go!'` There is a problem here, though, because the text-to-speech system takes a few moments to convert the string into sound. So, you will have to probe around a bit to find out where when to start measuring the reaction time.

More simply, try `Smalltalk beep`. Look at *Implementors of beep* to see other ways to make a sound.

# Reference Pages



# Squeak Smalltalk: Language Reference

---

Version 0.0, 20 November 1999, by Andrew C. Greenberg, [werdna@muco.com](mailto:werdna@muco.com)

Version 1.0, 17 April 2001, by Andrew P. Black, [black@cse.ogi.edu](mailto:black@cse.ogi.edu)

Based on:

Smalltalk-80: The Language and Its Implementation, Author: Adele Goldberg and David Robson  
Squeak Source Code v. 2.6  
and the readers of the Squeak mailing list.

Squeak site: <http://www.squeak.org>

---

## Contents

- [Using Squeak](#)
- [Squeak Smalltalk Syntax](#)

See also the [Squeak Classes Reference](#) page

---

## Using Squeak: the Basics

- [Mousing Around](#)
- [System Menus](#)
- [System Key Bindings](#)

### Mousing Around

Squeak (and the Smalltalk-80 from which it was spawned) assumes a machine with a three-button mouse (or its equivalent). These buttons were referred to as "red," "yellow" and "blue." The red button was conventionally used for selecting "things," the yellow button was conventionally used for manipulating "things" within a view or window and the blue button was conventionally used for manipulating windows themselves. Conventions, of course, are not always followed and your mileage may vary.

Since many modern mice no longer have three buttons, let alone colored ones, various "mapping" conventions are used:

For uncolored three-button mice, the mapping is:

```
left-mouse    -> red
middle-mouse  -> yellow
right-mouse   -> blue
```

Windows machines with three button mice can be made to conform with this mapping by right clicking the Windows title bar of Squeak, and selecting "VM Preferences -> Use 3 button mouse mapping." Otherwise, for Windows machines, the mapping is:

```
left-mouse    -> red
right-mouse   -> yellow
alt-left-mouse -> blue
```

MacOS Systems generally have one mouse button. The mapping is:

```
mouse         -> red
option-mouse  -> yellow
cmd-mouse     -> blue
```

If you purchase a 3-button mouse for your computer, you will be pleased that you did so! I put colored sticky labels on my buttons when I was first training my fingers.

If you have a mouse with a scrolling wheel, map "wheel up" to cmd-upArrow and "wheel down" to cmd-downArrow, and you will be able to use the wheel to control scrolling in your Squeak windows.

### System Menus

Squeak provides access to certain Smalltalk services through its system menus, some of which are depicted below:



**The Main Menu.** The World menu, sometimes called the "main menu," can be reached by clicking the red button while the mouse points to the background of a system project. From this menu, you can save the image and changes files, save them in files with a different name, and terminate execution of the Squeak virtual machine. You can also access many other menus ... including the four shown here.

**The open Menu** provides access to many system tools, including system browsers, workspaces, change sorters, transcripts and file lists, as well as end user tools such as an email agent (*Celeste*) and web browser (*Scamper*).

**The help Menu** provides access to certain on-line help facilities as well as a preferences dialog, some environment enquiries, a dictionary and facilities for updating your version of Squeak.

**The windows and flaps Menu** provides access to services for manipulating system windows and (in Morphic only) *flaps*. Flaps are small tabs at the side of the screen that pull out like drawers and provide quick access to whatever you place there. Try them! The *Tools* flap is a very convenient way of getting new system tools (rather than using the *open* menu).

**The appearance menu** lets the user change various aspects of the systems appearance. In particular, it provides a way of adjusting the display depth and going in and out of full screen mode.

## System Key Bindings

Applications that use standard Squeak text container widgets, including System Browsers, Workspaces, File Lists and Transcripts, provide facilities for manipulating the text and providing access to other system functionality. Many of these facilities can be reached by using the red-button menus, but many are more conveniently accessed using special key sequences. Of course, particular applications can use some, all or of none of these. In the following tables, a lower-case or numeric command "key" can be typed by simultaneously pressing the key and the Alt key (on Windows) or the  $\text{⌘}$  key (on MacOS). Upper-case keys are typed by simultaneously pressing either Shift-Alt (or Shift- $\text{⌘}$ ) and the indicated key, *or* ctrl and the indicated key. Other special key presses are indicated below in square brackets.

### General Editing Commands

---

Key	Description	Notes
z	Undo	
x	Cut	
c	Copy	
v	Paste	
a	Select all	
D	Duplicate. Paste the current selection over the prior selection, if it is non-overlapping and legal	1
e	Exchange. Exchange the contents of current selection with the contents of the prior selection	1
y	Swap. If there is no selection, swap the characters on either side of the insertion cursor, and advance the cursor. If the selection has 2 characters, swap them, and advance the cursor.	
w	Delete preceding word	

## Notes

1. These commands are a bit unusual: they concern and affect not only the current selection, but also the immediately preceding selection.

**Search and Replace**

Key	Description	Notes
f	Find. Set the search string from a string entered in a dialog. Then, advance the cursor to the next occurrence of the search string.	
g	Find again. Advance the cursor to the next occurrence of the search string.	
h	Set Search String from the selection.	
j	Replace the next occurrence of the search string with the last replacement made	
A	Advance argument. Advance the cursor to the next keyword argument, or to the end of string if no keyword arguments remain.	
J	Replace all occurrences of the search string with the last replacement made	
S	Replace all occurrences of the search string with the present change text	

**Cancel/Accept**

Key	Description	Notes
l	Cancel (also "revert"). Cancel all edits made since the pane was opened or since the last save	
s	Accept (also "save"). Save the changes made in the current pane.	
o	Spawn. Open a new window containing the present contents of this pane, and then reset this window to its last saved state (that is, cancel the present window).	

**Browsing and Inspecting**


---

Key	Description	Notes
b	Browse "it" (where "it" is a class name). Opens a new browser.	1
d	Do "it" (where "it" is a Squeak expression)	1
i	Inspect "it": evaluate "it" and open an inspector on the result. ("it" is a Squeak expression). Exception: in a method list pane, i opens an inheritance browser.	1
m	Open a browser of methods implementing "it" (where "it" is a message selector)	1,2
n	Open a browser of methods that send "it" (where "it" is a message selector).	1,2
p	Print "it". Evaluate "it" and insert the results immediately after "it." (where "it" is a Smalltalk expression)	1
B	Set the <i>present</i> browser to browse "it" (where "it" is a class name)	1
E	Open a browser of methods whose source contain strings with "it" as a substring.	1
I	Open the Object Explorer on "it" (where "it" is an expression)	1
N	Open a browser of methods using "it" (where "it" is an identifier or class name)	1
O	Open single-message browser (in selector lists)	1
W	Open a browser of methods whose selectors include "it" as a substring.	1

## Notes:

1. A null selection will be expanded to a word, or to the whole of the current line, in an attempt to do what you want.
2. For these operations, "it" means the *outermost* keyword selector in a large selection.

## Special Conversions and Processing

Key	Description	Notes
C	Open a workspace showing a comparison of the selection with the contents of the clipboard	
U	Convert linefeeds to carriage returns in selection	
X	Force selection to lowercase	
Y	Force selection to uppercase	
Z	Capitalize all words in selection	

## Smalltalk Program Data Entry

Key	Description	Notes
q	Attempt to complete the selection with a valid and defined Smalltalk selector. Repeated commands yield additional selectors.	
r	Recognizer. Invoke the Squeak glyph character recognizer. (Terminate recognition by mousing out of the window)	
F	Insert 'ifFalse:'	
T	Insert 'ifTrue:'	
V	Paste author's initials, date and time.	
L	Outdent (move selection or line one tab-stop left)	
R	Indent (move selection or line one tab stop right)	
[Ctl-return]	Insert return followed by as many tabs as the previous line (with a further adjustment for additional brackets in that line)	
[shift-delete]	Forward delete. Or, deletes from the insertion point to the beginning of the current word.	

## Bracket Keys

These keys are used to enclose (or unenclose if the selection is already enclosed) the selection in a kind of "bracket". Conveniently, double clicking just inside any bracketed text selects the entire text, but not the brackets.

Key	Description	Notes
Control-(	Enclose within ( and ), or remove enclosing ( and )	
Control- [	Enclose within [ and ], or remove enclosing [ and ]	
Control- {	Enclose within { and }, or remove enclosing { and }	
Control- <	Enclose within < and >, or remove enclosing < and >	
Control- '	Enclose within ' and ', or remove enclosing ' and '	
Control- "	Enclose within " and ", or remove enclosing " and "	

### Special Keys for Changing Text Style and Emphasis

Key	Description	Notes
k	Set font	
u	Align	
K	Set style	
1	10 point font	
2	12 point font	
3	18 point font	
4	24 point font	
5	36 point font	
6	color, action-on-click, link to class comment, link to method, url. Brings up a menu. To remove these properties, select more than the active part and then use command-0.	
7	bold	
8	italic	
9	narrow (same as negative kern)	
0	plain text (resets all emphasis)	
- (minus)	underlined (toggles it)	
=	struck out (toggles it)	
_ (a.k.a. shift -)	negative kern (letters 1 pixel closer)	
+ (a.k.a. shift =)	positive kern (letters 1 pixel larger spread)	

## Squeak Smalltalk Syntax: the Basics

- [Pseudo-variables](#)
- [Identifiers](#)
- [Comments](#)
- [Literals](#)
- [Assignments](#)
- [Messages](#)
- [Expression Sequences](#)
- [Cascades](#)
- [Expression Blocks](#)
- [Control Structures](#)
- [Brace Arrays](#)
- [Class Definition](#)
- [Method Definition](#)

### Pseudo-Variables

Pseudo-variable	Description
<b>nil</b>	The singleton instance of Class UndefinedObject
<b>true</b>	The singleton instance of Class True
<b>false</b>	The singleton instance of Class False
<b>self</b>	The current object, that is, the receiver of the current message.
<b>super</b>	As the receiver of a message, <b>super</b> refers to the same object as <b>self</b> . However, when a message is sent to <b>super</b> , the search for a suitable method starts in the superclass of the class whose method definition contains the word <b>super</b> .
<b>thisContext</b>	The active context, that is, the "currently executing" MethodContext or BlockContext.

- Pseudo-variables are reserved identifiers that are similar to keywords in other languages.
- **nil**, **true** and **false** are constants.
- **self**, **super** and **thisContext** vary dynamically as code is executed.
- It is not possible to assign to any of these pseudo-variables.

## Identifiers

letter (letter | digit )\*

- Smalltalk identifiers (and symbols) are **case-sensitive**.
- It is a Smalltalk convention for identifiers (instance and temporaries) of several words to begin with a lower case character, and then capitalize subsequent words. (e.g., thisIsACompoundIdentifier).
- Certain identifiers, for example, globals (e.g., Smalltalk) and class variables, are by convention initially capitalized. The names of all classes are also global variables (e.g., SystemDictionary).

## Comments

"a comment comprises any sequence of characters, surrounded by double quotes"  
 "comments can include the 'string delimiting' character"  
 "and comments can include embedded double quote characters by ""doubling"" them"  
 "comments can span many  
 many  
 lines"

## Literals (Constant Expressions)

### Numbers (Instances of class Number)

In the following, ==> means "prints as".

Decimal integer: **1234**, **12345678901234567890**  
 Octal integer: **8r177**, **8r17777777777777777777**  
 Hex integer: **16rFF**, **16r123456789ABCDEF012345**  
 Arbitrary base integer: **2r1010** ==> 10  
 Integer with exponent: **123e2** ==> 12300, **2r1010e2** ==> 40  
 Float (double precision): **3.14e-10**  
 Arbitrary base float: **2r1.1** ==> 1.5  
 Float with exponent: **2r1.1e2** ==> 6.0

- Squeak supports SmallInteger arithmetic (integers between  $-2^{30}$  and  $2^{30-1}$ ) with fast internal primitives.
- Squeak supports arbitrary precision arithmetic seamlessly (automatically coercing SmallInteger to LargePositiveInteger and LargeNegativeInteger where appropriate), albeit at a slight cost in speed.
- Squeak supports several other kinds of "numeric" value, such as Fractions (arbitrary precision rational numbers) and Points. While there are no literals for these objects, they are naturally expressed as operations on built-in literals. ( "2/3" and "2@3", respectively)
- Numbers may be represented in many radices, but the radix specification itself is always expressed in base 10. The base for the exponent part is the same as the radix. So: **2r1010** ==> 10, **10e2** ==> 1000 (=10 x 10<sup>2</sup>), but **2r1010e2** ==> 40 (=10 x 2<sup>2</sup>)

### Characters (Instances of class Character)

**\$x** "A character is any character (even unprintable ones), preceded by a dollar sign"  
**\$3** "Don't be shy about characters that are digits"  
**\$<** "or symbols"  
**\$\$** "or even the dollar sign"

### Strings (Instances of class String)

'a string comprises any sequence of characters, surrounded by single quotes'  
 'strings can include the "comment delimiting" character'  
 'and strings can include embedded single quote characters by doubling" them'  
 'strings can contain embedded  
 newline characters'  
 "" "and don't forget the empty string"

- A string is very much like ("isomorphic to") an array containing characters. Indexing a string answers characters at the corresponding position, starting with 1.

### Symbols (Instances of class Symbol)

#'A string preceded by a hash sign is a Symbol'  
 #orAnyIdentifierPrefixedWithAHashSign  
 #orAnIdentifierEndingWithAColon:  
 #or:several:identifiers:each:ending:with:a:colon:  
 #- "A symbol can also be a hash followed by '-' or any special character"  
 #+< "or a hash followed by any pair of special characters"

- Symbol is a subclass of String, and understands, in large part, the same operations.
- The primary difference between a symbol and a string is that all symbols comprising the same sequence of characters are the same instance. Two different string instances can both have the characters 'test one two three', but every symbol having the characters #'test one two three' is the same instance. This "unique instance" property means that Symbols can be efficiently compared, because equality (=) is the same as identity (==).
- "Identifier with colon" Symbols (e.g., #a:keyword:selector:) are often referred to as keyword selectors, for reasons that will be made clear later.
- "Single or dual symbol" Symbols (e.g., #\* or #++) are often referred to as binary selectors.
- The following are permissible special characters: +/\*\~<=>@%|&?!  
 Note that #-- is not a symbol (or a binary selector). On the other hand, #'--' is a symbol (but not a binary selector).

### Constant Arrays (Instances of class Array)

#( 1 2 3 4 5 ) "An array of size 5 comprising five Integers (1 to 5)"  
 #'( 'this' #is \$a #'constant' array ) "An array of size 5 comprising a String ('this'), a Symbol (#is), a Character (\$a) and two Symbols (#constant and #array)."  
 #( 1 2 ( 1 #(2) 3 ) 4 ) "An array of size 4 comprising two Integers (1 and 2), an Array of size 3, and another Integer (4)."  
 #( 1 + 2 ) "An array of size 3 comprising 1, #+, and 2. It is *not* the singleton array comprising 3."

- Constant arrays are constants, and their elements must therefore be constants. "Expressions" are not evaluated, but are generally parsed as sequences of symbols as in the example above.
- Constant arrays may contain constant arrays. The hash sign for internal constant arrays is optional.
- Identifiers and sequences of characters in constant arrays are treated as symbols; the hash sign for internal symbols is optional.
- Arrays are indexed with the first element at index 1.

### Assignments

identifier ← expression  
 identifier := expression " := is always a legal alternative to ← , but the pretty printer uses ← "

foo ← 100 factorial  
 foo ← bar ← 1000 factorial

- The identifier (whether instance variable, class variable, temporary variable, or otherwise) will thereafter refer to the object answered by the expression.
- The "←" glyph can be typed in Squeak by keying the underbar character (shift-hyphen).
- Assignments are expressions; they answer the result of evaluating the right-hand-side.
- Assignments can be cascaded as indicated above, resulting in the assignment of the same right-hand-side result to each variable.

### Messages

#### Unary Messages

theta sin  
 quantity sqrt  
 nameString size  
 1.5 tan rounded asString "same result as (((1.5 tan) rounded) asString)"

- Unary messages are messages without arguments.
- Unary messages are the most "tightly parsed" messages, and are parsed left to right. Hence, the last example answers the result of sending #asString to the result of sending #rounded to the result of sending #tan to 1.5

#### Binary Messages

```

3 + 4 " ==> 7 "
3 + 4 * 5 " ==> 35 (not 23) "
3 + 4 factorial " ==> 27 (not 5040) "
total - 1
total <= max "true if total is less than or equal to max"
(4/3)*3 = 4 " ==> true — equality is just a binary message, and Fractions are exact"
(3/4) == (3/4) " ==> false — two equal Fractions, but not the same object"

```

- Binary messages have a receiver, the left hand side, and a single argument, the right hand side. The first expression above sends 3 the message comprising the selector #+ with the argument 4.
- Binary messages are *always* parsed left to right, without regard to precedence of numeric operators, unless corrected with parentheses.
- Unary messages bind more tightly than binary messages

## Keyword Messages

```

12 between: 8 and: 15 " ==> true "
#($t $e $s $t) at: 3 " ==> $s "
array at: index put: value " ==> answers value, after storing value in array at index"
array at: index factorial put: value "same, but this time stores at index factorial"
1 to: 3 do: aBlock "This sends #to:do: (with two parameters) to integer 1"
(1 to: 3) do: aBlock "This sends #do: (with one parameter) to the Interval given by evaluating '1 to: 3'"

```

- Keyword messages have 1 or more arguments
- Keyword messages are the least "tightly parsed messages." Binary and unary messages are resolved first unless corrected with parentheses.

## Expression Sequences

```
expressionSequence ::= expression (. expression)* (.)opt
```

```

box ← 20@30 corner: 60@90.
box containsPoint: 40@50

```

- Expressions separated by *periods* are executed in sequence.
- Value of the sequence is the value of the final expression.
- The values of all of the other expressions are ignored.
- A final period is optional.

## Cascade Expressions

```

receiver
  unaryMessage;
+ 23;
at: 23 put: value;
yourself

```

- messages in a cascade are separated by *semicolons*; each message is sent to *receiver* in sequence.
- Intermediate answers are ignored, but side-effects on *receiver* will be retained.
- The cascade answers the result of sending the last message to *receiver* (after sending all the preceding ones!)

## Block Expressions

Blocks, actually instances of the class BlockContext. They are used all the time to build control structures. Blocks are created using the `[]` syntax around a sequence of expressions.

```

[ expressionSequence ] "block without arguments"
[ (: identifier)+ | expressionSequence ] "block with arguments"
[ (: identifier)+ || identifier+ | expressionSequence ] "block with arguments and local variables"

```

```

[ 1. 2. 3 ] "a block which, when evaluated, will answer the value 3"
[ object doWithSideEffects. test ] "a block which, when evaluated, will send #doWithSideEffects to object, and answer the object test"
[ :param | param doSomething ] "a block which, when evaluated with a parameter, will answer the result of sending #doSomething to the parameter.

```

- A block represents a deferred sequence of actions.
- The value of a block expression is an object that can execute the enclosed expressions at a later time, if requested to do so. Thus
  - `[ 1. 2. 3 ] ==> [ ] in UndefinedObject>>DoIt`
  - `[ 1. 2. 3 ] value ==> 3`
- Language experts will note that blocks are roughly equivalent to lambda-expressions, anonymous functions, or closures.

## Evaluation Messages for BlockContext

Message	Description	Notes
<b>value</b>	Evaluate the block represented by the receiver and answer the result.	1
<b>value: arg</b>	Evaluate the block represented by the receiver, passing it the value of the argument, arg.	2
<b>valueWithArguments: anArray</b>	Evaluate the block represented by the receiver. The argument is an Array whose elements are the arguments for the block. Signal an error if the length of the Array is not the same as the the number of arguments that the block was expecting.	3

## Notes

1. The message #value, sent to a block, causes the block to be executed and answers the result. The block must require zero arguments.
2. The message #value: arg, causes the block to be executed. The block must require exactly one argument; the corresponding parameter is initialized to arg.
3. Squeak also recognizes #value:value:, #value:value:value and #value:value:value:value. If you have a block with more than four parameters, you must use #valueWithArguments.

## Control Structures

## Alternative Control Structures (Receiver is Boolean)

Message	Description	Notes
<b>ifTrue: alternativeBlock</b>	Answer nil if the receiver is false. Signal an Error if the receiver is nonBoolean. Otherwise, answer the result of evaluating alternativeBlock	1,2
<b>ifFalse: alternativeBlock</b>	Answer nil if the receiver is true. Signal an Error if the receiver is nonBoolean. Otherwise answer the result of evaluating the argument, alternativeBlock.	1,2
<b>ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock</b>	Answer the value of trueAlternativeBlock if the receiver is true. Answer the value of falseAlternativeBlock if the receiver is false. Otherwise, signal an Error.	1,2
<b>ifFalse: falseAlternativeBlock ifTrue: trueAlternativeBlock</b>	Same as ifTrue:ifFalse:.	1,2

## Notes

1. These are not technically control structures, since they can be understood as keyword messages that are sent to boolean objects. (See the definitions of these methods in classes True and False, respectively).
2. However, these expressions play the same role as control structures in other languages.

## Alternative Control Structures (Receiver is any Object)

Message	Description	Notes
<b>ifNil: nilBlock</b>	Answer the result of evaluating nilBlock if the receiver is nil. Otherwise answer the receiver.	
<b>ifNotNil: ifNotNilBlock</b>	Answer the result of evaluating ifNotNilBlock if the receiver is not nil. Otherwise answer nil.	
<b>ifNil: nilBlock ifNotNil: ifNotNilBlock</b>	Answer the result of evaluating nilBlock if the receiver is nil. Otherwise answer the result of evaluating ifNotNilBlock.	
<b>ifNotNil: ifNotNilBlock ifNil: nilBlock</b>	Same as #ifNil:ifNotNil:	

## Iterative Control Structures (receiver is aBlockContext)

Message	Description	Notes
<b>whileTrue</b>	Evaluate the receiver. Continue to evaluate the receiver for so long as the result is true.	
<b>whileTrue: aBlock</b>	Evaluate the receiver. If true, evaluate aBlock and repeat.	
<b>whileFalse</b>	Evaluate the receiver. Continue to evaluate the receiver for so long as the result is false.	
<b>whileFalse: aBlock</b>	Evaluate the receiver. If false, evaluate aBlock and repeat.	

## Enumeration Control Structures (Receiver is anInteger)

Message	Description	Notes
<b>timesRepeat:</b> aBlock	Evaluate the argument, aBlock, the number of times represented by the receiver.	
<b>to:</b> stop <b>do:</b> aBlock	Evaluate aBlock with each element of the interval (self to: stop by: 1) as the argument.	
<b>to:</b> stop <b>by:</b> step <b>do:</b> aBlock	Evaluate aBlock with each element of the interval (self to: stop by: step) as the argument.	

### Enumeration Control Structures (Receiver is Collection)

Message	Description	Notes
<b>do:</b> aBlock	For each element of the receiver, evaluate aBlock with that element as the argument.	1

1. Squeak collections provide a very substantial set of enumeration operators. See the section [Enumerating Collections](#) on the Classes Reference.

### Case Structures (Receiver is any Object)

Message	Description	Notes
<b>caseOf:</b> aBlockAssociationCollection	Answer the evaluated value of the first association in aBlockAssociationCollection whose evaluated key equals the receiver. If no match is found, signal an Error.	1
<b>caseOf:</b> aBlockAssociationCollection <b>otherwise:</b> aBlock	Answer the evaluated value of the first association in aBlockAssociationCollection whose evaluated key equals the receiver. If no match is found, answer the result of evaluating aBlock.	1

1. aBlockAssociationCollection is a collection of Associations (key/value pairs).  
Example: aSymbol caseOf: { [#a] -> [1+1] . ['b' asSymbol] -> [2+2] . [#c] -> [3+3] }

### Expression "Brace" Arrays

```
braceArray ::= { expressionSequence }
```

```
{ 1. 2. 3. 4. 5 } "An array of size 5 comprising five Integers (1 to 5)"
{ $a #brace array } "An array of size 3 comprising a Character ($a) a Symbol (#brace), and the present value of the variable array."
{ 1 + 2 } "An array of size 1 comprising the single integer 3."
```

- Brace arrays are bona-fide Smalltalk expressions that are computed at runtime.
- The elements of a brace array are the answers of its component expressions.
- They are a sometimes convenient and more general alternative to the clunky expression "Array with: expr1 with: expr2 with: expr3"
- Indexing is 1-based.

### Answer Expressions

```
answerExpression ::= ^ expression
```

```
^ aTemporary
```

```
^ 2+3
```

- Inside the body of a method, an answer expression is used to terminate the execution of the method and deliver the expression as the method's answer.
- Answer expressions inside a nested block expression will terminate the enclosing method.

### Class Definition

#### Class Definition

```
SuperClass subclass: #NameOfClass
  instanceVariableNames: 'instVarName1 instVarName2'
  classVariableNames: 'ClassVarName1 ClassVarName2'
  poolDictionaries: ''
  category: 'Major-Minor'
```

#### Variable Class Definition

These forms of class definition are used to create indexable objects, *i.e.*, those like Array, ByteArray and WordArray. They are included here for completeness, but are not normally used directly; instead, use an ordinary object with an instance variable whose value is an appropriate Array (or other collection) object.

```

SuperClass variableSubclass: #NameOfClass
  instanceVariableNames: 'instVarName1 instVarName2'
  classVariableNames: 'ClassVarName1 ClassVarName2'
  poolDictionaries: ''
  category: 'Major-Minor'

SuperClass variableByteSubclass: #NameOfClass
  instanceVariableNames: 'instVarName1 instVarName2'
  classVariableNames: 'ClassVarName1 ClassVarName2'
  poolDictionaries: ''
  category: 'Major-Minor'

SuperClass variableWordSubclass: #NameOfClass
  instanceVariableNames: 'instVarName1 instVarName2'
  classVariableNames: 'ClassVarName1 ClassVarName2'
  poolDictionaries: ''
  category: 'Major-Minor'

```

## Method Definition

All methods answer a value; there is an implicit `^ self` at the end of every method to make sure that this is the case. Here is an example (from class `String`).

### `lineCount`

"Answer the number of lines represented by the receiver, where every `cr` adds one line."

```

| cr count |
cr ← Character cr.
count ← 1 min: self size.
self do:
  [:c | c == cr ifTrue: [count ← count + 1]].
^ count

```

---

# Squeak Smalltalk: Classes Reference

---

Version 0.0, 20 November 1999, by Andrew C. Greenberg, [werdna@muco.wisc.edu](mailto:werdna@muco.wisc.edu)  
 Version 01., 10 April 2001, by Andrew P. Black, [black@cse.ogi.edu](mailto:black@cse.ogi.edu)

Based on:  
 Smalltalk-80: The Language and Its Implementation, Author: Adele Goldberg and David Robson  
 Squeak Source Code v. 2.6  
 and the readers of the Squeak mailing list.

Squeak site: <http://www.squeak.org>

---

## Contents

- [Fundamental Classes and Methods](#)
- [Numeric Classes and Methods](#)
- [Collection Classes and Methods](#)
- [Streaming Classes and Methods](#)
- [ANSI-Compatible Exceptions](#)
- [The Squeak Class Hierarchy](#)
- [Other Categories of Squeak Classes](#)

See also the [Squeak Language Reference](#) page

---

## Fundamental Classes and Methods

- [Class Object](#)
- [Class Boolean](#)
- [Class Magnitude](#)
- [Class Character](#)

### Class Object (Operations on all objects)

#### Instance Creation (Class Side)

Message	Description	Notes
<b>new</b>	Answer a new instance of the receiver (which is a class). This is the usual way of creating a new object. <b>new</b> is often overridden in subclasses to provide class-specific behavior.	1, 2
<b>basicNew</b>	This is the primitive that is ultimately called to implement <b>new</b> .	3
<b>new:</b> anInteger	Answer an instance of the receiver (which is a class) with size given by anInteger. Only allowed if it makes sense to specify a size.	4

Notes:

1. The usual body for a **new** method is `^ super new initialize`. Remember to put it on the class side, remember to type the `^`, and remember to say `super`, not `self`!
2. It's OK not to implement `new` if it makes no sense, For example, look at `Boolean class>>new` and `MappedCollection class>>new`.
3. **basicNew** is there so you can still make instances even if a superclass has overridden **new**. Consequently, never override **basicNew**, until you become a wizard.
4. If you need an initialization parameter other than a size, choose a more meaningful name than **new:** For example, look at the instance creation protocol for `Pen` class and `Rectangle` class.

### Comparing Objects

---

Message	Description	Notes
<code>==</code> <code>anObject</code>	Are the receiver and the argument the same object? (Answer is <b>true</b> or <b>false</b> )	1
<code>~~</code> <code>anObject</code>	Are the receiver and the argument different objects?	1
<code>=</code> <code>anObject</code>	Are the receiver and the argument equal? The exact meaning of equality depends on the class of the receiver.	2
<code>~=</code> <code>anObject</code>	Are the receiver and the argument unequal?	2
<code>hash</code>	Answer a <code>SmallInteger</code> whose value is related to the receiver's value.	2

Notes:

1. `==` and `~~` should not normally be redefined in any other class.
2. Since various classes (particularly `Sets` and `Dictionaries`) rely on the property that equal objects have equal hashes, you should override `#hash` whenever you override `#=`. It must be true that  $(a = b)$  implies  $(a \text{ hash} = b \text{ hash})$ . The inverse and the converse will not hold in general.

### Testing Objects

Message	Description	Notes
<code>isNil</code>	Is the receiver nil? (Answer is <b>true</b> or <b>false</b> )	
<code>notNil</code>	Is the receiver not nil?	
<code>ifNil: aBlock</code>	Evaluate <code>aBlock</code> if the receiver is nil, and answer the value of <code>aBlock</code> . Otherwise answers nil.	
<code>ifNotNil: aBlock</code>	Evaluate <code>aBlock</code> if the receiver is not nil, and answer the value of <code>aBlock</code> . Otherwise answers the receiver.	
<code>ifNotNilDo: aBlock</code>	If the receiver is not nil, evaluate <code>aBlock</code> with the receiver as argument.	1

- `ifNotNilDo: aBlock` is useful if the receiver is a complex expression, for example  

```
self leftChild rightChild ifNotNilDo: [ :node | node balance ]
```

### Copying Objects

Message	Description	Notes
<code>copy</code>	Answer another instance just like the receiver. Subclasses typically override this method; they typically do not override <code>shallowCopy</code> .	
<code>shallowCopy</code>	Answer a copy of the receiver which shares the receiver's instance variables.	
<code>deepCopy</code>	Answer a copy of the receiver with its own copy of each instance variable.	
<code>veryDeepCopy</code>	Do a complete tree copy using a dictionary. An object in the tree twice is copied once and shared by both referents.	

### Sending Messages to Objects

Message	Description	Notes
<code>perform: aSymbol</code>	Send the unary selector, <code>aSymbol</code> , to the receiver. Signal an error if the number of arguments expected by the selector is not zero.	
<code>perform: aSymbol with: anObject</code>	Send the selector <code>aSymbol</code> to the receiver with <code>anObject</code> as its argument. Fail if the number of arguments expected by the selector is not one.	1
<code>perform: selector withArguments: argArray</code>	Send the selector, <code>aSymbol</code> , to the receiver with arguments in <code>argArray</code> . Fail if the number of arguments expected by the selector does not match the size of <code>argArray</code> .	

1. Squeak objects also recognize `#perform:with:with:` and `#perform:with:with:with`

### Indexing Objects

---

Message	Description	Notes
<b>at:</b> index	Answer the value of an indexable element in the receiver. Signal an Error if index is not an Integer or is out of bounds.	
<b>at:</b> index <b>put:</b> anObject	Store the argument value in the indexable element of the receiver indicated by index. Signal an Error if index is not an Integer or is out of bounds. Or fail if the value is not of the right type for this kind of collection. Answer the value that was stored.	
<b>at:</b> index <b>modify:</b> aBlock	Replace the element at index of the receiver with that element transformed by the block.	
<b>size</b>	Answer the number of indexable variables in the receiver. This value is the same as the largest legal subscript. If the receiver does not have instance variables, then signal an Error.	

### Displaying and Storing Objects

Message	Description	Notes
<b>printString</b>	Answer a String whose characters describe the receiver.	
<b>printOn:</b> aStream	Append to the argument, aStream, a String whose characters describe the receiver.	
<b>storeString</b>	Answer a String from which the receiver can be reconstructed.	
<b>storeOn:</b> aStream	Append to the argument, aStream, a String from which the receiver can be reconstructed	

### Interrogating Objects

Message	Description	Notes
<b>class</b>	Answers the receiver's class (an object).	
<b>isKindOf:</b> aClass	Is the receiver an instance of aClass or one of its subclasses?	
<b>isMemberOf:</b> aClass	Is the receiver an instance of aClass? (Same as <code>rcvr class == aClass</code> )	
<b>respondsTo:</b> aSelector	Can the receiver find a method for aSelector, either in the receiver's class or in one of its superclasses?	
<b>canUnderstand:</b> aSelector	Does the receiver, which must be a class, have a method for aSelector? The method can belong to the receiver or to any of its superclasses.	

### Miscellaneous Messages on Objects

Message	Description	Notes
<b>yourself</b>	Answers self.	1
<b>asString</b>	Answers the receiver's printString.	
<b>doesNotUnderstand:</b> aSymbol	Report that the receiver does not understand aSymbol as a message.	
<b>error:</b> aString	Signal an Error.	
<b>halt</b>	Stops execution.	2

- the message `yourself` is mostly used as the last message in a cascade, when the previous message answered some object other than the receiver. For example,  

```
#(1 2 3 5 5) at: 4 put: 4           answers 4, the object that was put, whereas
#(1 2 3 5) at: 4 put: 4; yourself  answers #(1 2 3 4)
```
- `self halt` is the usual way of forcing entry to the debugger. The halt can be resumed.

### Class Boolean

This abstract class represents logical values, providing Boolean operations and conditional control structures. It has two subclasses, `True` and `False`, each of which have singleton instances represented by the Squeak keywords **true** and **false**, respectively.

### Evaluating and Non-Evaluating Logical Operations for Boolean

Message	Description	Notes
<b>&amp;</b> aBoolean	Evaluating conjunction (AND). Evaluate the argument. Then answer true if both the receiver and the argument are true.	
<b>eqv:</b> aBoolean	Answer true if the receiver is equivalent to aBoolean.	
<b>not</b>	Negation. Answer true if the receiver is false, answer false if the receiver is true.	
<b>xor:</b> aBoolean	Exclusive OR. Answer true if the receiver is not equivalent to aBoolean.	
<b> </b> aBoolean	Evaluating disjunction (OR). Evaluate the argument. Then answer true if either the receiver or the argument is true.	
<b>and:</b> alternativeBlock	Nonevaluating conjunction. If the receiver is true, answer the value of the argument, alternativeBlock; otherwise answer false without evaluating the argument.	
<b>or:</b> alternativeBlock	Nonevaluating disjunction. If the receiver is false, answer the value of the argument, alternativeBlock; otherwise answer true without evaluating the argument.	

## Class Magnitude

This abstract class embraces, among other classes, Numbers, Characters, Date and Time. It addresses classes whose instances can be linearly ordered.

Message	Description	Notes
<b>&lt;</b> aMagnitude	Answer whether the receiver is strictly less than the argument.	
<b>&gt;</b> aMagnitude	Answer whether the receiver is strictly greater than the argument.	
<b>&lt;=</b> aMagnitude	Answer whether the receiver is less than or equal to the argument.	
<b>&gt;=</b> aMagnitude	Answer whether the receiver is greater than or equal to the argument.	
<b>between:</b> min <b>and:</b> max	Answer whether the receiver is greater than or equal to the argument, min, and less than or equal to the argument, max.	
<b>min:</b> aMagnitude	Answer the receiver or the argument, whichever is the lesser magnitude.	
<b>max:</b> aMagnitude	Answer the receiver or the argument, whichever is the greater magnitude.	
<b>min:</b> firstMagnitude <b>max:</b> secondMagnitude	Take the receiver or the argument, firstMagnitude, whichever is the lesser magnitude, and answer that or the argument, secondMagnitude, whichever is the greater magnitude.	

## Class Character

Squeak has its own 256-character set, which may differ from that of the host platform. Instances of class Character store an 8-bit character code.

- The characters 0-127 are the same as the corresponding ASCII characters, with a few exceptions: the assignment arrow replaces underscore, and characters for the enter, insert, pageup, page down, home, and the 4 arrow keys replace some of the ACSII control characters. These characters can be accessed from Squeak using methods in class Character.
- The characters 128-255 are sparsely populated. Various symbols, such as bullets, trademark, copyright, cent, Euro and Yen, diphthongs and a fair number of accented characters as well as non-breaking space (Character nbsp) are available at the same codes as in the Macintosh character set, but fewer characters are assigned than on the Macintosh.
- The full character set can be viewed by doing a printIt on "Character allCharacters"

### Methods for instance creation (Class side)

Most of the time, characters literals \$a, \$b, *etc.* are used in preference to class methods. The principal exceptions are the non-printing characters listed here. Programs should never need to depend on the details of the character encoding.

Message	Description	Notes
<b>value:</b> n	n must be an integer in the range 0 to 255. Answer the Character with code n	1
<b>digitValue:</b> x	Answer the Character whose digit value is x. For example, answer \$9 for x=9, \$0 for x=0, \$A for x=10, \$Z for x=35.	
<b>arrowDown arrowLeft arrowRight arrowUp backspace cr delete end enter escape euro home insert lf linefeed nbsp newPage pageDown pageUp space tab</b>	Answer the appropriate character	

1. The invariant (Character value: n) `asciiValue = n` holds for all n in the range [0..255].

### Methods for accessing Characters

Message	Description	Notes
<code>asciiValue</code>	Answer the value used in the receiver's encoding. This is not really ASCII, despite the name!	1
<code>digitValue</code>	Answer 0-9 if the receiver is \$0-\$9, 10-35 if it is \$A-\$Z, and < 0 otherwise. This is used to parse literal numbers of radix 2-36.	

1. Character has the unique instance property, so that all equal ("=") instances of a character are identical ("=="). That is, a `asciiValue == b` `asciiValue` if and only if a == b.

### Methods for testing Characters

Message	Description	Notes
<code>isAlphaNumeric</code>	Answer whether the receiver is a letter or a digit.	
<code>isDigit</code>	Answer whether the receiver is a digit.	
<code>isLetter</code>	Answer whether the receiver is a letter.	
<code>isLowercase</code>	Answer whether the receiver is a lowercase letter.	
<code>isSeparator</code>	Answer whether the receiver is one of the separator characters: space, cr, tab, line feed, or form feed.	
<code>isSpecial</code>	Answer whether the receiver is one of the special characters	
<code>isUppercase</code>	Answer whether the receiver is an uppercase letter.	
<code>isVowel</code>	Answer whether the receiver is one of the vowels, AEIOU, in upper or lower case.	
<code>tokenish</code>	Answer whether the receiver is a valid token-character--letter, digit, or colon.	

### Methods for converting Characters

Message	Description	Notes
<code>asLowercase</code>	If the receiver is uppercase, answer its matching lowercase Character.	
<code>asUppercase</code>	If the receiver is lowercase, answer its matching uppercase Character.	

## Numeric Classes and Methods

### Class Number

This abstract class embraces Integers, Floats and Fractions. Number is a subclass of Magnitude.

### Methods for arithmetic on all Numeric Classes

Message	Description	Notes
+ aNumber	Answer the sum of the receiver and the argument.	
- aNumber	Answer the difference of the receiver and the argument.	
* aNumber	Answer the product of the receiver and the argument.	
/ aNumber	Answer the result of dividing the receiver and the argument, retaining as much precision as possible. If the answer is not exact, the result will be Fraction or Float, as appropriate. Signal ZeroDivide if the argument is Zero.	
// aNumber	Answer the result of dividing the receiver and the argument, truncating toward negative infinity. Signal ZeroDivide if the argument is Zero.	
\ aNumber	Answer the remainder of dividing the receiver and the argument, truncating toward negative infinity. This is the modulus operator. Signal ZeroDivide if the argument is Zero.	
quo: aNumber	Answer the result of dividing the receiver and the argument, truncating toward zero. Signal ZeroDivide if the argument is Zero.	
rem: aNumber	Answer the remainder of dividing the receiver and the argument, truncating toward zero. Signal ZeroDivide if the argument is Zero.	
abs	Answer the absolute value of the receiver.	
negated	Answer the negation of the receiver.	
reciprocal	Answer 1 divided by the receiver. Signal ZeroDivide if the receiver is zero.	

### Methods implementing mathematical functions for Numbers

Message	Description	Notes
exp	Answer a floating point number that is the exponential of the receiver	
ln	Answer the natural log of the receiver.	
log: aNumber	Answer the logarithm base aNumber of the receiver.	
floorLog: aNumber	Take the logarithm base aNumber of the receiver, and answer the integer nearest that value towards negative infinity.	
raisedTo: aNumber	Answer the receiver raised to the power of the argument, aNumber.	
raisedToInteger: anInteger	Answer the receiver raised to the power of the argument, anInteger. Signal an Error if anInteger is not an integer.	
sqrt	Answer a floating point number that is the positive square root of the receiver.	
squared	Answer the receiver multiplied by itself.	

### Methods for testing Numbers

Message	Description	Notes
even	Answer whether the receiver is even.	
odd	Answer whether the receiver is odd.	
negative	Answer whether the receiver is less than zero.	
positive	Answer whether the receiver is greater than or equal to zero.	
strictlyPositive	Answer whether the receiver is greater than zero.	
sign	Answer 1 if the receiver is strictly positive, zero if the receiver is zero, and -1 if the receiver is strictly negative.	
isZero	Answer whether the receiver is zero.	

### Methods for truncating and rounding Numbers

Message	Description	Notes
<b>ceiling</b>	Answer the Integer nearest the receiver toward positive infinity.	
<b>floor</b>	Answer the Integer nearest the receiver toward negative infinity.	
<b>truncated</b>	Answer an integer nearest the receiver toward zero.	
<b>truncateTo: aNumber</b>	Answer the next multiple of aNumber toward zero that is nearest the receiver.	
<b>rounded</b>	Answer the integer nearest the receiver.	
<b>roundTo: quantum</b>	Answer the nearest number that is a multiple of quantum.	
<b>roundUpTo: quantum</b>	Answer the next multiple of aNumber toward infinity that is nearest the receiver.	

### Methods for trigonometry on Numbers

Message	Description	Notes
<b>sin</b>	Answer the sine of the receiver taken as an angle in radians.	
<b>cos</b>	Answer the cosine of the receiver taken as an angle in radians.	
<b>tan</b>	Answer the tangent of the receiver taken as an angle in radians.	
<b>degreeSin</b>	Answer the sin of the receiver taken as an angle in degrees.	
<b>degreeCos</b>	Answer the cosine of the receiver taken as an angle in degrees.	
<b>arcSin</b>	Answer an angle in radians whose sine is the receiver.	
<b>arcCos</b>	Answer an angle in radians whose cosine is the receiver.	
<b>arcTan</b>	Answer an angle in radians whose tangent is the receiver.	
<b>arcTan: denominator</b>	Answer the angle in radians whose tan is the receiver divided by denominator.	
<b>degreesToRadians</b>	Answer the receiver in radians. Assumes the receiver is in degrees.	
<b>radiansToDegrees</b>	Answer the receiver in degrees. Assumes the receiver is in radians.	

## Class Integer

### Methods for arithmetic on Integers

Message	Description	Notes
<b>isPowerOfTwo</b>	Answer whether the receiver is a power of two.	
<b>factorial</b>	Answer the factorial of the receiver.	
<b>gcd: anInteger</b>	Answer the greatest common denominator of the receiver and the argument.	
<b>lcm: anInteger</b>	Answer the least common multiple of the receiver and the argument.	
<b>take: anInteger</b>	Answer the number of combinations of the receiver, taken the argument, anInteger, at a time.	

### Methods for bit manipulation on Integers

A range of bit manipulation operations are available on Integers. They are rarely needed, however, so they are not described here. Of course, they can be viewed using the browser.

## Collection Classes and Methods

### The Collection Hierarchy

Class	Description
Collection	Abstract Class for Collections
Bag	Unordered, unindexed collection of objects
Set	Unordered, unindexed collection of unique objects
Dictionary	Set of associations (values are indexable by keys)
IdentityDictionary	Dictionary, but comparisons are done using ==
IdentitySet	Set, but comparisons are done using ==
SequenceableCollection	Ordered collection, indexed by integers
OrderedCollection	Ordered according to manner elements are added and removed
SortedCollection	Ordered according to value of a "sortBlock"
LinkedList	Homogeneous SequenceableCollection of Links
Interval	Homogeneous sequence of arithmetic progression of Integers
ArrayedCollection	Ordered collection, indexed by fixed range of Integers
Array	ArrayedCollection of arbitrary Objects
Array2D	Homogeneous ArrayedCollection of Arrays
ByteArray	Homogeneous ArrayedCollection of Bytes (Integers -128..255)
FloatArray	Homogeneous ArrayedCollection of Floating point numbers
IntegerArray	Homogeneous ArrayedCollection of Signed 32-bit Integers
PointArray	Homogeneous ArrayedCollection of Points (with 32-bit values)
RunArray	Homogeneous ArrayedCollection of Integers (sparse RLE representation)
ShortIntegerArray	Homogeneous ArrayedCollection of Signed 16-bit Integers
ShortPointArray	Homogeneous ArrayedCollection of Points (with 16-bit values)
ShortRunArray	Homogeneous ArrayedCollection of Signed 16-bit Ints (sparse RLE rep)
String	Homogeneous ArrayedCollection of Characters
Symbol	Homogeneous ArrayedCollection of Characters (with unique instance property)
Text	Homogeneous ArrayedCollection of Characters with associated text attributes
WordArray	Homogeneous ArrayedCollection of Unsigned 32-bit Integers
Heap	Like SortedCollection, but stores information as a heap. (see Heapsort)
MappedCollection	Means for accessing an indexable Collection, using a mapping from a collection of "external" keys to the accessed collection's "indexing" keys. The MappedCollection can then be used directly, indexing and changing the accessed collection via the external keys.

## Class Collection

### Methods for creating Collections (Class Side)

Message	Description	Notes
<b>with:</b> anObject	Answer an instance of the receiver containing anObject	
<b>with:</b> firstObject <b>with:</b> secondObject	Answer an instance of the receiver containing all the arguments as elements. (Squeak recognizes instantiators of this type up to six "with:" clauses).	
<b>withAll:</b> aCollection	Answer an instance of the receiver containing all the elements from aCollection.	

### Methods for testing, adding and removing Collection elements

---

<b>Message</b>	<b>Description</b>	<b>Notes</b>
<b>anyOne</b>	Answer a specimen element of the receiver (any one at all). Signal an error if the receiver is empty.	
<b>isEmpty</b>	Answer whether the receiver contains any elements.	
<b>occurrencesOf: anObject</b>	Answer how many of the receiver's elements are equal to anObject.	
<b>anySatisfy: aBlock</b>	Evaluate aBlock with the elements of the receiver. If aBlock returns true for any element return true. Otherwise return false	
<b>includes: anObject</b>	Answer whether anObject is one of the receiver's elements.	
<b>includesAllOf: aCollection</b>	Answer whether all the elements of aCollection are in the receiver.	
<b>includesAnyOf: aCollection</b>	Answer whether any element of aCollection is one of the receiver's elements.	
<b>difference: secondCollection</b>	Answer a new collection that is computed by copying the receiver and removing all the elements in secondCollection.	
<b>add: newObject</b>	Include newObject as one of the receiver's elements. Answer newObject. ArrayedCollections cannot respond to this message.	
<b>addAll: newObject</b>	Include all the elements of aCollection as the receiver's elements. Answer aCollection.	
<b>remove: oldObject</b>	Remove oldObject as one of the receiver's elements. Answer oldObject unless no element is equal to oldObject, in which case, signal an Error.	
<b>remove: oldObject ifAbsent: anExceptionBlock</b>	Remove oldObject as one of the receiver's elements. If several of the elements are equal to oldObject, only one is removed. If no element is equal to oldObject, answer the result of evaluating anExceptionBlock. Otherwise, answer oldObject. SequenceableCollections cannot respond to this message.	
<b>removeAll: aCollection</b>	Remove each element of aCollection from the receiver. If successful for each, answer aCollection. Otherwise signal an Error.	
<b>removeAllFoundIn: aCollection</b>	Remove from the receiver each element of aCollection that is present in the receiver.	
<b>removeAllSuchThat: aBlock</b>	Apply the condition to each element and remove it if the condition is true.	

### Methods for enumerating Collections

---

Message	Description	Notes
<b>do:</b> aBlock	Evaluate aBlock with each of the receiver's elements as the argument.	
<b>do:</b> aBlock <b>separatedBy:</b> separatorBlock	Evaluate aBlock for all elements in the receiver, and if there is more than one element, evaluate the separatorBlock between each pair of elements in the receiver.	
<b>select:</b> aPredicateBlock	Evaluate aPredicateBlock with each of the receiver's elements as the argument. Collect into a new collection like the receiver, only those elements for which aPredicateBlock evaluates to true. Answer the new collection.	
<b>reject:</b> aPredicateBlock	Evaluate aPredicateBlock with each of the receiver's elements as the argument. Collect into a new collection like the receiver only those elements for which aPredicateBlock evaluates to false. Answer the new collection.	
<b>collect:</b> aMappingBlock	Evaluate aMappingBlock with each of the receiver's elements as the argument. Collect the resulting values into a collection like the receiver. Answer the new collection.	
<b>detect:</b> aPredicateBlock	Evaluate aPredicateBlock with each of the receiver's elements as the argument. Answer the first element for which aPredicateBlock answers true. Signal an Error if none are found.	
<b>detect:</b> aPredicateBlock <b>ifNone:</b> exceptionBlock	Evaluate aPredicateBlock with each of the receiver's elements as the argument. Answer the first element for which aPredicateBlock evaluates to true. If there is none, answer the result of evaluating exceptionBlock.	
<b>inject:</b> initialValue <b>into:</b> binaryBlock	Accumulate a running value associated with evaluating binaryBlock. The running value is initialized to initialValue. The current running value and the next element of the receiver are provided as the arguments to binaryBlock. For example, to compute the sum of the elements of a numeric collection, aCollection <b>inject: 0 into:</b> [:subTotal :next   subTotal + next].	
<b>collect:</b> aMappingBlock <b>thenSelect:</b> aPredicateBlock	Evaluate aMappingBlock with each of the receiver's elements as the argument. Collect the resulting values that satisfy aPredicateBlock into a collection like the receiver. Answer the new collection.	
<b>select:</b> aPredicateBlock <b>thenCollect:</b> aMappingBlock	Evaluate aMappingBlock with each of the receiver's elements for which aPredicateBlock answers true as the argument. Collect the resulting values into a collection like the receiver. Answer the new collection.	
<b>count:</b> aPredicateBlock	Evaluate aPredicateBlock with each of the receiver's elements as the argument. Return the number that answered true.	

## Bag

### Methods for accessing Bags

Message	Description	Notes
<b>add:</b> newObject <b>withOccurrences:</b> anInteger	Add the element newObject to the receiver. Do so as though the element were added anInteger number of times. Answer newObject.	

## Dictionary and IdentityDictionary

### Methods for Accessing Dictionaries

Dictionaries are homogenous Sets of key and value pairs. These pairs are called Associations: key and value can be any object. Instances of Association are created by sending the binary message "`key -> value`" (`->` is defined in Object). Dictionaries have the property that each key occurs at most once. IdentityDictionaries have the same property, but determine uniqueness of keys using `==` instead of `=`. In ordinary use, both kinds of Dictionary are indexed using the unique key to obtain the corresponding value.

Message	Description	Notes
<b>at:</b> aKey	Answer the value associated with aKey. Signal an Error if no value is associated with aKey.	
<b>at:</b> aKey <b>ifAbsent:</b> aBlock	Answer the value associated with aKey. If no value is associated with aKey, answer the value of aBlock.	
<b>associationAt:</b> aKey	Answer the association whose key is aKey. If there is none, signal an Error	
<b>associationAt:</b> aKey <b>ifAbsent:</b> aBlock	Answer the association whose key is aKey. If there is none, answer the value of aBlock.	
<b>keyAtValue:</b> aValue	Answer the key of the first association having aValue as its value. If there is none, signal an Error.	
<b>keyAtValue:</b> aValue <b>ifAbsent:</b> exceptionBlock	Answer the key of the first associaiton having aValue as its value. If there is none, answer the result of evaluating exceptionBlock.	
<b>keys</b>	Answer a Set containing the receiver's keys.	
<b>values</b>	Answer an Array containing the receiver's values.	
<b>includes:</b> aValue	Does the receiver contain a value equal to aValue?	
<b>includesKey:</b> aKey>	Does the receiver have a key equal to aKey?	
<b>do:</b> aBlock	Evaluate aBlock with each of the receiver's values as argument.	
<b>keysDo:</b> aBlock	Evaluate aBlock with each of the receiver's keys as argument.	
<b>valuesDo:</b> aBlock	same as <b>do:</b>	
<b>keysAndValuesDo:</b> aBinaryBlock	Evaluate aBinaryBlock with each of the receiver's keys and the associated value as the <i>two</i> arguments.	
<b>associationsDo:</b> aBlock	Evaluate aBlock with each of the receiver's elements (key/value associations) as the argument.	

## Sequenceable Collection

### Methods for accessing SequenceableCollections

---

Message	Description	Notes
<b>atAll:</b> indexCollection	Answer a collection containing the elements of the receiver specified by the integer elements of the argument, indexCollection.	
<b>atAll:</b> aCollection <b>put:</b> anObject	Put anObject at every index specified by the integer elements of the argument, aCollection.	
<b>atAllPut:</b> anObject	Put anObject at every one of the receiver's indices.	
<b>first</b>	Answer the first element of the receiver. (Squeak also recognizes second, third, fourth, fifth and sixth). Signal an error if there aren't sufficient elements in the receiver.	
<b>middle</b>	Answer the median element of the receiver. Signal an error if the receiver is empty.	
<b>last</b>	Answer the last element of the receiver. Signal an error if the receiver is empty.	
<b>allButFirst</b>	Answer a collection equal to the receiver, but without the first element. Signal an error if the receiver is empty.	
<b>allButLast</b>	Answer a collection equal to the receiver, but without the last element. Signal an error if the receiver is empty.	
<b>indexOf:</b> anElement	Answer the index of anElement within the receiver. If the receiver does not contain anElement, answer 0.	
<b>indexOf:</b> anElement <b>ifAbsent:</b> exceptionBlock	Answer the index of anElement within the receiver. If the receiver does not contain anElement, answer the result of evaluating the argument, exceptionBlock.	
<b>indexOfSubCollection:</b> aSubCollection <b>startingAt:</b> anIndex	Answer the index of the receiver's first element, such that that element equals the first element of aSubCollection, and the next elements equal the rest of the elements of aSubCollection. Begin the search at element anIndex of the receiver. If no such match is found, answer 0.	
<b>indexOfSubCollection:</b> aSubCollection <b>startingAt:</b> anIndex <b>ifAbsent:</b> exceptionBlock	Answer the index of the receiver's first element, such that that element equals the first element of sub, and the next elements equal the rest of the elements of sub. Begin the search at element start of the receiver. If no such match is found, answer the result of evaluating argument, exceptionBlock.	
<b>replaceFrom:</b> start <b>to:</b> stop <b>with:</b> replacementCollection	This destructively replaces elements from start to stop in the receiver. Answer the receiver itself. Use <b>copyReplaceFrom:to:with:</b> for insertion/deletion that may alter the size of the result.	
<b>replaceFrom:</b> start <b>to:</b> stop <b>with:</b> replacementCollection <b>startingAt:</b> repStart	This destructively replaces elements from start to stop in the receiver starting at index, repStart, in the sequenceable collection, replacementCollection. Answer the receiver. No range checks are performed.	

### Methods for copying SequenceableCollections

---

Message	Description	Notes
, otherCollection	Answer a new collection comprising the receiver concatenated with the argument, otherCollection.	
<b>copyFrom:</b> start <b>to:</b> stop	Answer a copy of a subset of the receiver that contains all the elements between index start and index stop, inclusive of both.	
<b>copyReplaceAll:</b> oldSubCollection <b>with:</b> newSubCollection	Answer a copy of the receiver in which all occurrences of oldSubstring have been replaced by newSubstring.	
<b>copyReplaceFrom:</b> start <b>to:</b> stop <b>with:</b> replacementCollection	Answer a copy of the receiver satisfying the following conditions: If stop is less than start, then this is an insertion; stop should be exactly start-1, start = 1 means insert before the first character, start = size+1 means append after last character. Otherwise, this is a replacement; start and stop have to be within the receiver's bounds.	
<b>copyWith:</b> newElement	Answer a copy of the receiver that is 1 bigger than the receiver and has newElement at the last element.	
<b>copyWithout:</b> oldElement	Answer a copy of the receiver in which all occurrences of oldElement have been left out.	
<b>copyWithoutAll:</b> aList	Answer a copy of the receiver in which all occurrences of all elements in aList have been removed.	
<b>forceTo:</b> length <b>paddingWith:</b> anElement	Force the length of the collection to length, padding if necessary with elem. Note that this makes a copy.	
<b>reversed</b>	Answer a copy of the receiver in which the sequencing of all the elements has been reversed.	
<b>shuffled</b>	Answer a copy of the receiver in which the elements have been permuted randomly.	
<b>sortBy:</b> aBlock	Create a copy that is sorted. Sort criteria is the block that accepts two arguments. When the block is true, the first arg goes first ([a :b   a > b] sorts in descending order).	

### Methods for enumerating SequenceableCollections

Message	Description	Notes
<b>findFirst:</b> aBlock	Return the index of the receiver's first element for which aBlock evaluates as true.	
<b>findLast:</b> aBlock	Return the index of the receiver's last element for which aBlock evaluates as true.	
<b>keysAndValuesDo:</b> aBinaryBlock	Evaluate aBinaryBlock once with each valid index for the receiver in order, along with the corresponding value in the receiver for that index.	
<b>reverseDo:</b> aBlock	Evaluate aBlock with each of the receiver's elements as the argument, starting with the last element and taking each in sequence up to the first. For SequenceableCollections, this is the reverse of the enumeration for #do:.	
<b>with:</b> otherCollection <b>do:</b> binaryBlock	Evaluate binaryBlock with corresponding elements from this collection and otherCollection.	
<b>reverseWith:</b> otherCollection <b>do:</b> aBinaryBlock	Evaluate aBinaryBlock with each of the receiver's elements, in reverse order, along with the corresponding element, also in reverse order, from otherCollection.	

### OrderedCollections

#### Methods for accessing OrderedCollections

---

Message	Description	Notes
<b>add:</b> newObject <b>before:</b> oldObject	Add the argument, newObject, as an element of the receiver. Put it in the sequence just preceding oldObject. Answer newObject.	
<b>add:</b> newObject <b>after:</b> oldObject	Add the argument, newObject, as an element of the receiver. Put it in the sequence just succeeding oldObject. Answer newObject.	
<b>add:</b> newObject <b>afterIndex:</b> index	Add the argument, newObject, as an element of the receiver. Put it in the sequence just after index. Answer newObject.	
<b>addFirst:</b> anElement	Add newObject to the beginning of the receiver. Answer newObject.	
<b>addAllFirst:</b> anOrderedCollection	Add each element of anOrderedCollection at the beginning of the receiver. Answer anOrderedCollection.	
<b>addLast:</b> anElement	Add newObject to the end of the receiver. Answer newObject.	
<b>addAllLast:</b> anOrderedCollection	Add each element of anOrderedCollection at the end of the receiver. Answer anOrderedCollection.	
<b>removeAt:</b> anIndex	remove the element of the receiver at location anIndex. Answers the element removed.	
<b>removeFirst</b>	Remove the first element of the receiver and answer it. If the receiver is empty, signal an Error.	
<b>removeLast</b>	Remove the last element of the receiver and answer it. If the receiver is empty, signal an Error.	

## Strings

String is an extensive class, built over the ages in something of an ad hoc manner. We describe here only a small fraction of the methods provided (there are about 300!)

### Methods for accessing Strings

Message	Description	Notes
<b>findAnySubStr:</b> delimiters <b>startingAt:</b> start	Answer the index of the character within the receiver, starting at start, that begins a substring matching one of the delimiters; delimiters is an Array of Strings and/or Characters. If the receiver does not contain any of the delimiters, answer size + 1.	
<b>findBetweenSubStrs:</b> delimiters	Answer the collection of tokens that results from parsing the receiver. And of the Strings (or Characters) in the Array delimiters is recognized as separating tokens.	
<b>findDelimiters:</b> delimiters <b>startingAt:</b> start	Answer the index of the character within the receiver, starting at start, that matches one of the delimiters. If the receiver does not contain any of the delimiters, answer size + 1.	
<b>findString:</b> subString	Answer the first index of subString within the receiver. If the receiver does not contain subString, answer 0.	
<b>findString:</b> subString <b>startingAt:</b> start	Answer the index of subString within the receiver, starting at start. If the receiver does not contain subString, answer 0.	
<b>findTokens:</b> delimiters	Answer the collection of tokens that results from parsing the receiver. Any character in the argument, delimiters, marks a border. Several delimiters in a row are considered as just one separator	
<b>indexOf:</b> aCharacter	Answer the index of the first occurrence of aCharacter in the receiver. 0 Otherwise.	
<b>indexOf:</b> aCharacter <b>startingAt:</b> start	Answer the index of the first occurrence of aCharacter in the receiver, beginning at index start. 0 Otherwise.	
<b>indexOf:</b> aCharacter <b>startingAt:</b> start <b>ifAbsent:</b> aBlock	Answer the index of the first occurrence of aCharacter in the receiver, beginning at index start. If not present, answer the value of aBlock.	
<b>indexOfAnyOf:</b> aCharacterSet	Answers the index of the first occurrence in the receiver of any character in the given set. Returns 0 if none is found.	1

## Notes

- As with #indexOf, there are corresponding messages #indexOfAnyOf:ifAbsent:, #indexOfAnyOf:startingAt: and #indexOfAnyOf:startingAt:ifAbsent:)

### Methods for comparing Strings

Message	Description	Notes
= aString	Answer whether the receiver is equal to aString. The comparison is case-sensitive,	
< aString, <= aString > aString >= aString	Answer whether the receiver sorts as indicated with aString. The collation order is that of the Squeak character set, and therefore case-sensitive,	
sameAs: aString	Answer whether the receiver is equal to aString, ignoring differences of case.	
compare: aString	Answer a code defining how the receiver sorts relative to the argument, aString. 1 - receiver before aString; 2 - receiver equal to aString; and 3 - receiver after aString. The collation sequence is that of the Squeak character set and is case insensitive.	
match: text	Answer whether text matches the pattern in the receiver. Matching ignores upper/lower case differences. Where the receiver contains #, text may contain any character. Where the receiver contains *, text may contain any sequence of characters.	
beginsWith: prefix	Answer whether the receiver begins with the argument, prefix.	
endsWith: prefix	Answer whether the receiver ends with the argument, prefix.	
alike: aString	Answer a non-negative integer indicating how similar the receiver is to aString. 0 means "not at all alike". The best score is aString size * 2.	

### Methods for converting Strings

Message	Description	Notes
asLowercase	Answer a new String that matches the receiver but without any upper case characters.	
asUppercase	Answer a new String that matches the receiver but without any lower case characters.	
capitalized	Answer a copy of the receiver with the first character capitalized if it is a letter.	
asDisplayText	Answer a copy of the receiver with default font and style information.	
asInteger	Attempts to parse the receiver as an Integer. Answers the Integer, or nil if the receiver does not start with a digit.	
asNumber	Attempts to parse the receiver as a Number. It is an error if the receiver does not start with a digit.	
asDate	Attempts to parse the receiver as a date, and answers an appropriate instance of class Date. Many formats are recognized.	

## Streaming Classes and Methods

### The Stream Hierarchy

Class	Description
Stream	Abstract Class for Accessors
PositionableStream	Accessors for Collections Indexable by an Integer
ReadStream	Read-Only
WriteStream	Write-Only
ReadWriteStream	Read and/or Write
FileStream	Accessors for collections whose elements are "paged in"
StandardFileStream	Accessors for files accessed from a file system
CrLfFileStream	Automatically handles system-specific line endings
DummyStream	Like /dev/null

## Class Stream

Stream is an abstract class for an accessor to a sequence of objects, called the contents. The stream is said to be "advanced" when the stream is configured to access a later element of the contents.

### Methods for accessing Streams

Message	Description	Notes
<b>contents</b>	Answer the entire contents of the receiver.	
<b>next</b>	Answer the next object accessible by the receiver.	
<b>next: anInteger</b>	Answer the next anInteger number of objects accessible by the receiver.	
<b>next: n put: anObject</b>	Make the next n objects accessible by the receiver anObject. Answer anObject.	
<b>nextMatchAll: aColl</b>	Answer true if next N objects are the ones in aColl, else false. Advance stream if true, leave as was if false.	
<b>nextMatchFor: anObject</b>	Answer whether the next object is equal to the argument, anObject, advancing the stream.	
<b>nextPut: anObject</b>	Insert the argument, anObject, as the next object accessible by the receiver. Answer anObject.	
<b>nextPutAll: aCollection</b>	Append the elements of aCollection to the sequence of objects accessible by the receiver. Answer aCollection.	
<b>upToEnd</b>	Answer the remaining elements in the string	
<b>flush</b>	Ensure that any objects buffered in the receiver are sent to their final destination.	

### Methods for testing Streams

Message	Description	Notes
<b>atEnd</b>	Answer whether the receiver can access any more objects.	

### Methods for enumerating Streams

Message	Description	Notes
<b>do: aBlock</b>	Evaluate aBlock for each of the remaining objects accessible by receiver.	

## Class PositionableStream

PositionableStream is an abstract class for accessors to sequences of objects that can be externally named by indices so that the point of access can be repositioned. Concrete classes ReadStream, WriteStream and ReadWriteStream are typically used to instantiate a PositionableStream on Collections, depending upon the access mode. StandardFileStream and CRLFFileStream are typically used for instantiating PositionableStreams for Files.

### Methods for accessing PositionableStreams

Message	Description	Notes
<b>contentsOfEntireFile</b>	Answer a collection containing the remainder of the receiver.	
<b>last</b>	Return the final element of the receiver.	
<b>nextDelimited: terminator</b>	Answer the contents of the receiver, from the current position up to the next terminator character; provided, however, that doubled terminators will be included as a single element.	
<b>nextInto: buffer</b>	Given buffer, an indexable object of size n, fill buffer with the next n objects of the receiver.	
<b>nextLine</b>	Answer next line (may be empty), or nil if at end	
<b>originalContents</b>	Answer the receiver's actual contents collection. ( <b>contents</b> returns a copy)	
<b>peek</b>	Answer what would be returned if the message <b>next</b> were sent to the receiver, but don't advance the receiver. If the receiver is at the end, answer nil.	
<b>peekFor: anObject</b>	Answer false and do not move over the next element if it is not equal to anObject, or if the receiver is at the end. Answer true and advance the stream if the next element is equal to anObject.	
<b>upTo: anObject</b>	Answer a subcollection from the current access position to the occurrence (if any, but not inclusive) of anObject in the receiver. If anObject is not in the collection, answer the entire rest of the receiver.	
<b>upToAll: aCollection</b>	Answer a subcollection from the current access position to the occurrence (if any, but not inclusive) of aCollection. If aCollection is not in the stream, answer the entire rest of the stream.	

#### Methods for testing PositionableStreams

Message	Description	Notes
<b>isEmpty</b>	Answer whether the receiver's contents has no elements.	

#### Methods for positioning PositionableStreams

Message	Description	Notes
<b>match: subCollection</b>	Set the access position of the receiver to be past the next occurrence of the subCollection. Answer whether subCollection is found. No wildcards, case sensitive.	
<b>padTo: nBytes put: aCharacter</b>	Pad, using aCharacter, to the next boundary of nBytes.	
<b>padToNextLongPut: char</b>	Make position be on long word boundary, writing the padding character, char, if necessary.	
<b>position</b>	Answer the current position of accessing the sequence of objects.	
<b>position: anInteger</b>	Set the current position for accessing the objects to be anInteger, as long as anInteger is within the bounds of the receiver's contents. If it is not, create an error notification.	
<b>reset</b>	Set the receiver's position to the beginning of the sequence of objects.	
<b>resetContents</b>	Set the position and limits to 0.	
<b>setToEnd</b>	Set the position of the receiver to the end of the sequence of objects.	
<b>skip: anInteger</b>	Set the receiver's position to be the current position+anInteger. A subclass might choose to be more helpful and select the minimum of the receiver's size and position+anInteger, or the maximum of 1 and position+anInteger for the repositioning.	
<b>skipTo: anObject</b>	Set the access position of the receiver to be past the next occurrence of anObject. Answer whether anObject is found.	

### Class WriteStream

#### Methods for writing characters on WriteStreams

Message	Description	Notes
<b>cr</b>	Append a return character to the receiver.	
<b>crtab</b>	Append a return character, followed by a single tab character, to the receiver.	
<b>crtab: anInteger</b>	Append a return character, followed by anInteger tab characters, to the receiver.	
<b>space</b>	Append a space character to the receiver.	
<b>tab</b>	Append a tab character to the receiver.	

## ANSI-Compatible Exceptions

### Evaluating Blocks with Exceptions

#### Methods for handling Exceptions raised in a BlockContext

Message	Description	Notes
<b>ensure: aTerminationBlock</b>	Evaluate aTerminationBlock after evaluating the receiver, regardless of whether the receiver's evaluation completes.	
<b>ifCurtailed: aTerminationBlock</b>	Evaluate the receiver. If it terminates abnormally, evaluate aTerminationBlock.	
<b>on: exception do: handlerActionBlock</b>	Evaluate the receiver in the scope of an exception handler, handlerActionBlock.	

#### Examples

```
["target code, which may abort"]
  ensure:
    ["code that will always be executed
     after the target code,
     whatever whatever may happen"]

["target code, which may abort"]
  ifCurtailed:
    ["code that will be executed
     whenever the target code terminates
     without a normal return"]

["target code, which may abort"]
  on: Exception
  do: [:exception |
    "code that will be executed whenever
     the identified Exception is signaled."]
```

### Exceptions

Exception is an abstract class; instances should neither be created nor trapped. There are two common subclasses of Exception, Error and Notification, from which subclasses normally inherit. Errors are not resumable; a Notification is an indication that something interesting has occurred; if it is not handled, it will pass by without effect.

Exceptions play two distinct roles: that of the exception, and that of the exception handler.

#### Methods for describing Exceptions

Message	Description	Notes
<b>defaultAction</b>	The default action taken if the exception is signaled.	
<b>description</b>	Return a textual description of the exception.	
<b>isResumable</b>	Determine whether an exception is resumable.	
<b>messageText</b>	Return an exception's message text.	
<b>tag</b>	Return an exception's tag value.	

## Methods for signalling Exceptions

Message	Description	Notes
<b>signal</b>	Signal the occurrence of an exceptional condition.	
<b>signal:</b> signalerText	Signal the occurrence of an exceptional condition with a specified textual description.	

## Methods for dealing with a signaled Exception

Message	Description	Notes
<b>isNested</b>	Determine whether the current exception handler is within the scope of another handler for the same exception.	
<b>outer</b>	Evaluate the enclosing exception action for the receiver and return.	
<b>pass</b>	Yield control to the enclosing exception action for the receiver.	
<b>resignalAs:</b> replacementException	Signal an alternative exception in place of the receiver.	
<b>resume</b>	Return from the message that signaled the receiver.	
<b>resume:</b> resumptionValue	Return the argument as the value of the message that signaled the receiver.	
<b>retry</b>	Abort an exception handler and re-evaluate its protected block.	
<b>retryUsing:</b> alternativeBlock	Abort an exception handler and evaluate a new block in place of the handler's protected block.	
<b>return</b>	Return nil as the value of the block protected by the active exception handler.	
<b>return:</b> returnValue	Return the argument as the value of the block protected by the active exception handler.	

## Class ExceptionSet

An ExceptionSet is used to specify a set of exceptions for an exception handler.

### Creating ExceptionSet

Message	Description	Notes
<b>,</b> anException	Receiver may be an Exception class or an ExceptionSet. Answers an exception set that contains the receiver and anException.	

### Example

```
["target code, which may abort"]
  on: Exception, Error, ZeroDivide
  do:
    [:exception |
      "code that will be executed whenever
      one of the identified Exceptions is
      signaled."]
```

## The Squeak Class Hierarchy

In Smalltalk, "everything is an object." That is, everything is an instance of class Object or an instance of some subclass of class Object. Everything. Numbers, Classes, Metaclasses, everything. I refer to this as the "Object rule."

Actually, Squeak bends this rule a little bit; the Object rule does not apply for certain system objects, which derive from class ProtoObject. Nevertheless, except for these few system objects, the vast majority of Squeak objects, which I call, "proper objects," satisfy the Object Rule. Proper Objects and their classes and metaclasses, satisfy the following properties.

### The Laws of Proper (Smalltalk) Classes

- Every proper class is a subclass of class Object, except for Object itself, which has no proper superclass. In particular, Class is a subclass of ClassDescription, which is a subclass of Behavior which is a subclass of Object.

- Every object is an instance of a class.
- Every class is an instance of a metaclass.
- All metaclasses are (ultimately) subclasses of Class.
- Every metaclass is an instance of MetaClass.
- The methods of Class and its superclasses support the behavior common to those objects that are classes.
- The methods of instances of MetaClass add the behavior specific to particular classes.

## Class ProtoObject

Squeak additionally supports an improper class ProtoObject, from which object hierarchies other than proper instances and proper classes can inherit. ProtoObject is the superclass of class Object and has no instances. Presently, there are two subclasses of ProtoObject besides Object: ObjectOut and ImageSegmentRootStub, both of which are used to do magic involving objects that have been moved out of memory onto an external medium. You might need to subclass ProtoObject if you are doing something like implementing a remote message send system where you have proxies for remote objects (those on another computer).

However, as with proper classes, ProtoObject, is an instance of a metaclass, ProtoObject class, which in turn is an instance of class MetaClass.

## Categories of Squeak Classes

This quick reference only scratches the surface of the functionality available through Squeak. To assist the beginner in surveying the system, the following outline, or roadmap, of the Squeak system is provided.

Category	Description
Kernel	Primary Smalltalk classes for creating and manipulating Smalltalk objects, the Object hierarchy, coroutines and parallel processes. Subcategories include: Objects, Classes, Methods and Processes.
Numeric	Classes for numeric operations, including date and time operations. Subcategories include Magnitudes and Numbers
Collections	Classes for aggregations of Smalltalk objects.
Graphics	Core classes for Smalltalk graphic objects as well as facilities and applications for operating on graphic objects. Key classes include Form and BitBlT.
Interface	The "traditional" MVC User Interface Framework. Also found here are a number of useful Smalltalk applications, including: Squeak browsers, a mail client, a web browser, irc chat client and facilities for operating on "projects."
Morphic	Squeak's Morphic User Interface Framework
Music	Classes supporting Squeak's Sound Synthesis capabilities. Also found here are several useful facilities and applications for manipulating MIDI data and other representations of musical scores.
System	Key System Facilities. Subclasses include: Compiler (Smalltalk compiler); Object Storage (virtual memory for Smalltalk objects); File facilities; Compression; Serial Data Transmission; Basic network facilities.
Exceptions	Class supporting Squeak's ANSI-compliant exceptions facilities.
Network	Classes implementing various Internet and Squeak related network protocols.
PluggableWebServer	A complete web-server application, including an implementation of SWIKI, a collaborative world-wide-web environment. Key classes include: PWS
HTML	Classes for manipulating HTML data.
Squeak	Here lives the mouse. Key classes include: the Squeak VM and an interpreter; the Squeak Smalltalk Subset (Slang) to C translator; and facilities for developing native plugins (pluggable primitives).
Balloon	Classes for complex 2-D graphic objects and fast 2-D graphics rendering.
Balloon-3D	Classes for complex 3-D graphics objects and fast 3-D graphics rendering.
TrueType	Classes for manipulating and displaying TrueType data.
MM-Flash	Classes for manipulating and displaying Flash file data.
Alice & Wonderland	A remarkable interactive 3-D graphics environment.