

Morphic

- **A highly unusual graphical toolkit!**
- **Originates in the Self project at Sun**
 - Self: a prototype-based programming language
 - No classes---objects inherit/instantiate by “cloning”
- **Self design strongly reflected in Morphic**
 - Can create Morphic objects, add properties & behavior without defining classes
 - All Morphic objects have uniform, “concrete” feel---e.g., “shadows” when dragging

Morphic, Smalltalk-style

- Smalltalk is class-based, so Squeak Morphic generates classes “under the hood”
- You can also use Morphic in traditional (non-prototype-based) style.
- This tutorial will use a traditional class-based programming style.

Squeak Morphic programming

- **Goal:** to get you coding something “interesting” as quickly as possible.
- **Steps:**
 - (Enter a Morphic world)
 1. Define & instantiate your own Morph
 2. Customizing your Morph
 3. Animating Morphs
 4. Toolkits and hierarchical composition

Morphs in the class browser

The image shows two screenshots from a software interface. The left screenshot shows a 'new morph...' menu with a pink arrow pointing to a submenu titled 'Add a new morph'. This submenu lists various options like 'dismiss this menu', 'from paste buffer', and 'from alphabetical list', followed by a list of packages: Basic, Books, Components, Demo, Experimental, Games, Kernel, Keunwoo, Menus, Palettes, Scripting, and Morphic-Scripting. The 'BasicButton' class is highlighted in a pink oval at the bottom of the submenu. The right screenshot shows a 'System' class browser window with a list of packages on the left and a list of classes on the right. The 'Morphic-Scripting' package is highlighted in red. The classes listed are BasicButton, CategoryViewer, MethodMorph, PhraseWrapperMorph, and Player. Below the class list, there are three columns: 'instance', '?', and 'class'. The 'BasicButton' class is highlighted in red.

new morph...

Add a new morph
dismiss this menu
from paste buffer
from a file...
from alphabetical list ▶
grab patch from screen
make new drawing
make link to project...

Basic
Books
Components
Demo
Experimental
Games
Kernel
Keunwoo
Menus
Palettes
Scripting
Morphic-Scripting

BasicButton

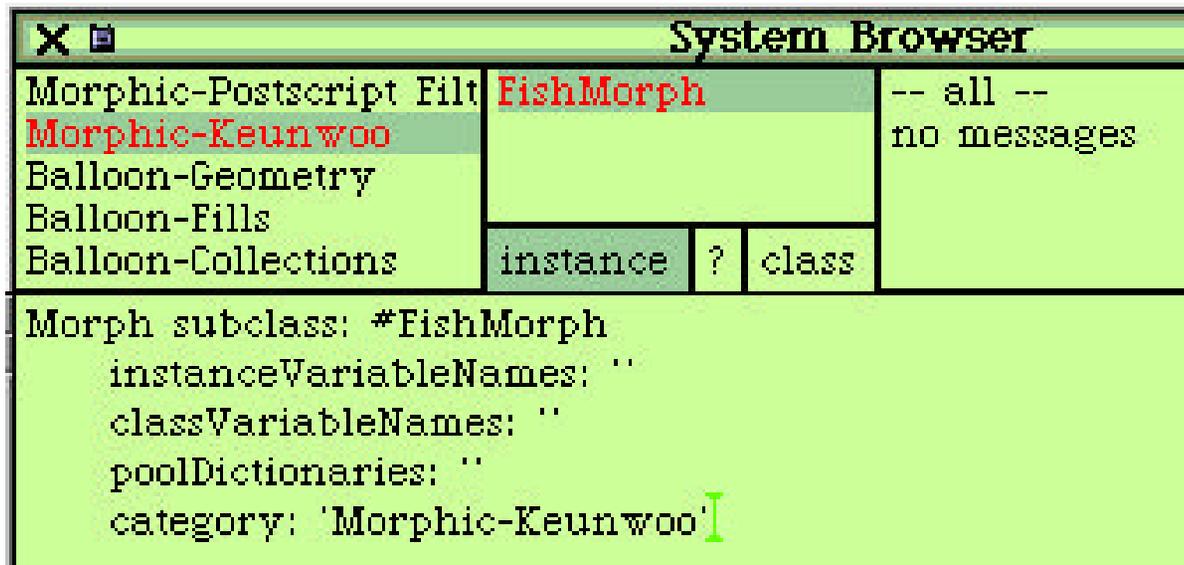
System

Morphic-Windows	BasicButton		
Morphic-Menus	CategoryViewer		
Morphic-Components	MethodMorph		
Morphic-Components B	PhraseWrapperMorph		
Morphic-Scripting	Player		
Morphic-Scripting Sup	instance	?	class
Morphic-Scripting Tile			

- All Morphic objects are instances of subclasses of **Morph**.
- Squeak looks for Morph subclasses in class categories starting with **Morphic-**
- Morphs found in these packages will be shown in the **new morph...** submenu

1. Defining your own Morph

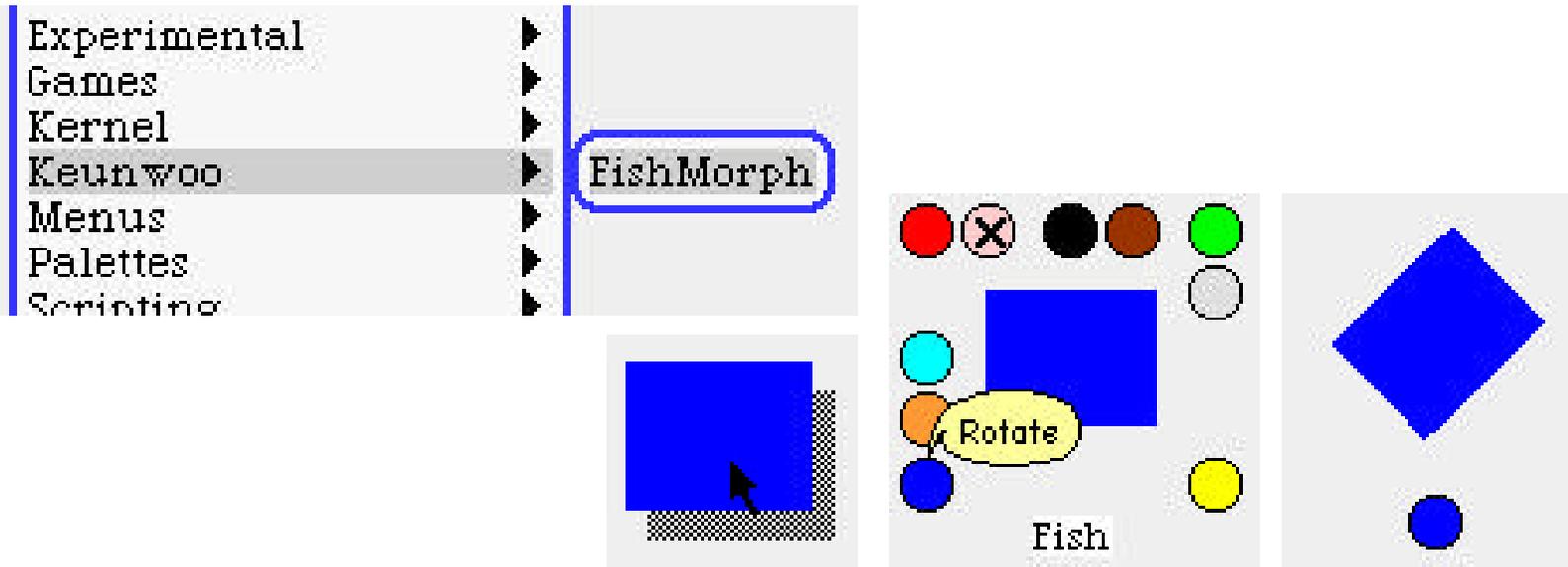
- You can add Morph subclasses anywhere.
- But, you will probably want to create a new class category for your Morphs, e.g. **Morphic-Keunwoo**
- In this category, define a new subclass of **Morph**:



The screenshot shows a 'System Browser' window with a tree view on the left and a code editor on the right. The tree view shows a hierarchy of categories: 'Morphic-Keunwoo' is selected and highlighted in red. Below it are 'Balloon-Geometry', 'Balloon-Fills', and 'Balloon-Collections'. The 'Balloon-Collections' category is expanded to show 'instance' and 'class' sub-items. The code editor displays the following Smalltalk code for the new subclass:

```
Morph subclass: #FishMorph
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Morphic-Keunwoo'
```

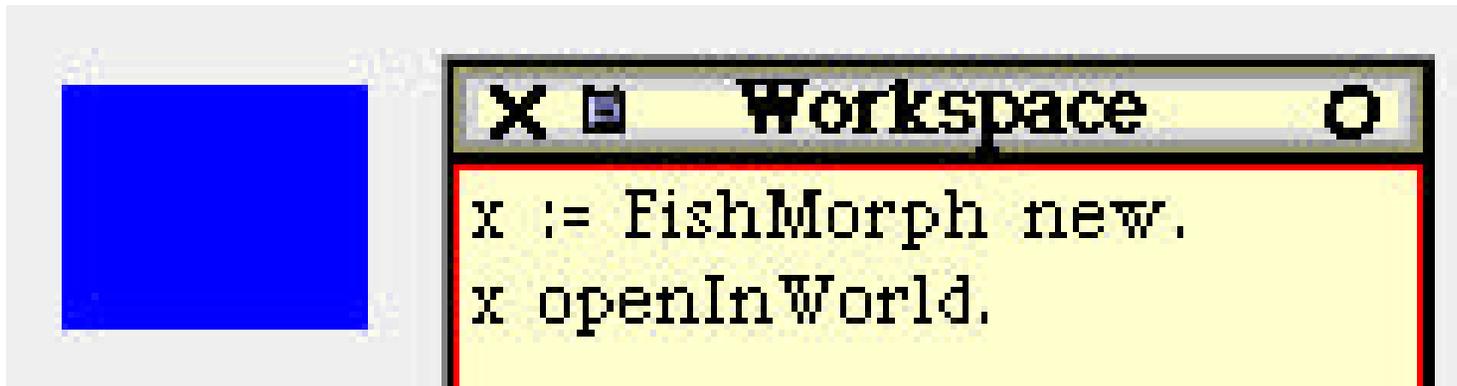
You're done! But...



- Your new morph category, and morph, should appear in the **new morph...** submenu.
- You inherit all the default Morph behaviors. (The default rendering is a blue rectangle.)
- Default behaviors are nice, but they're not *yours*...
- (Important: See various online tutorials for information on **halos**, direct manipulation of Morphs, etc.)

Alternate way to show instances

1. Open a workspace
2. Create an instance with **new**
3. Send the **openInWorld** message



What's the “world”?

- The global namespace* contains a variable named **World**.
- When you enter a Morphic “world”, **World** is set to point to the current “world”
- When you send the **openInWorld** message to a Morph, it gets the current World and adds itself.

* For the curious, the global namespace is a dictionary named **Smalltalk**. Do **Smalltalk inspect** in any Workspace to get a look at it.

Okay, but what's *a* “world”?

Q: What's a “world”?

A: An instance of a subclass of PasteUpMorph

Q: What's a PasteUpMorph?

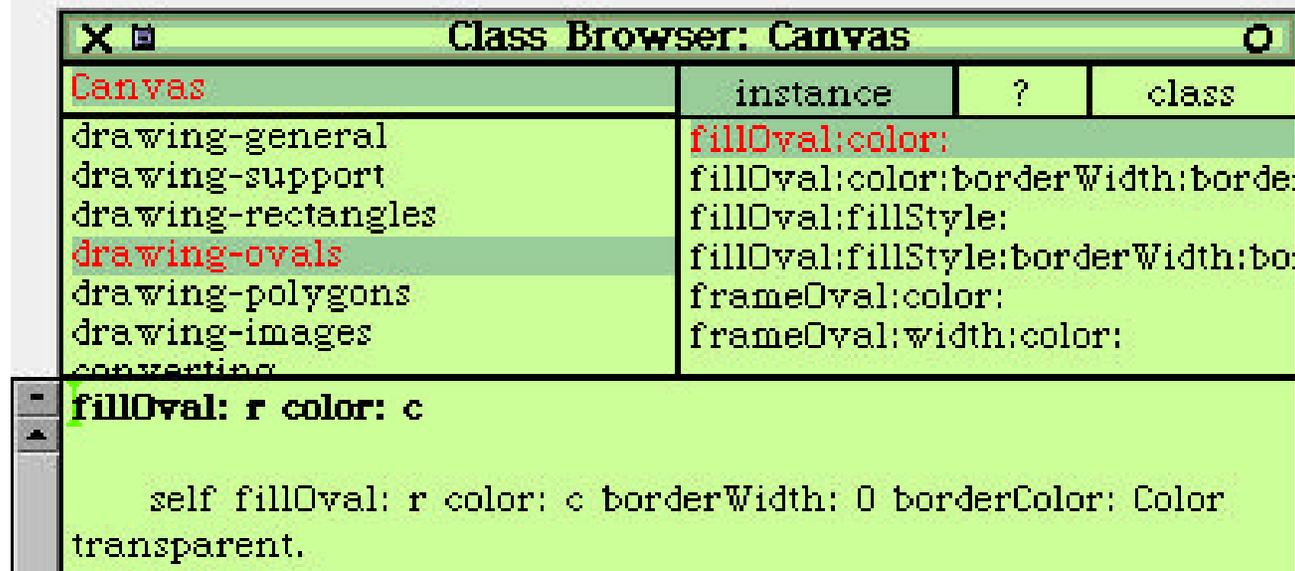
A: A Morph where you can drop other morphs, and they stick---think of it as a “desktop-like” morph.

2. Customizing your Morph

- Morphs are almost endlessly flexible
- For brevity, we will begin by customizing only two aspects:
 - Appearance (“look”)
 - Response to mouse input (“feel”)

2(a). Morph drawing [2]

- Like most graphics toolkits, components paint themselves onto a **graphics context** provided by the system.
- In Squeak, graphics contexts are instances of **Canvas**
- **Canvas** defines many methods for drawing...



Graphical environments: A question

Q: When should components paint themselves?

A: **Often. It's complicated...**

- When created
- Whenever onscreen area is covered, then uncovered
- Whenever it receives input that changes its state
 - (e.g., pressed button must change appearance)
- Whenever the state of the thing it represents changes
 - (e.g., an animation of a physics simulation)
- ...and more...

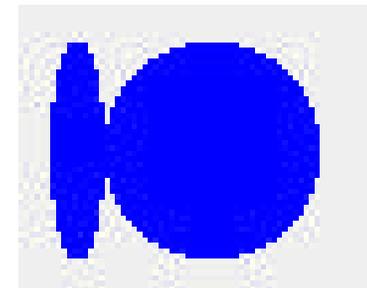
2(a) Drawing components [2]

- Therefore, components draw when asked by the system, onto the Canvas provided.
- When object needs a repaint, it will be sent the **drawOn:** message, which takes a **Canvas:**

System Browser			
Morphic-Kernel	Morph	drawing	drawOn:
Morphic-Basic	MorphExtension	geometry	drawOnCanvas:
Morphic-Worlds	MorphicModel	rotate scale and	drawPostscriptOn:
Morphic-Support		geometry testing	drawSubmorphsOn:
Morphic-Text Support	instance ? class	geometry eToy	flash
drawOn: aCanvas			
aCanvas fillRectangle: self bounds fillStyle: self fillStyle.			

2(a) Customized drawing [3]

- To customize drawing, simply override the **drawOn:** message



System Browser			
Morphic-Postscript Canvas	FishMorph	-- all --	drawOn:
Morphic-Postscript Filter		drawing	
Morphic-Keunwoo			
Balloon-Geometry			
Balloon-Fills			
Balloon-Collections	instance	?	class

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

Aside: a word about geometry

- Two natural screen coordinate systems:
 - “Text-like”: top left corner is $(0, 0)$
 - Y coordinate increases as you go down screen
 - “Math-like”: bottom left corner is $(0, 0)$
 - Y coordinate increases as you go up screen
- Morphic has both...
 - **x/x: and y/y: methods use math-like**
 - **position/position: methods use text-like**

2(b) Custom event handling [1]

- Input events are similar to painting events
- To define your own event action, override a message that handles the event, e.g. **mouseDown:**

Class Browser: Morph			
Morph	instance	?	class
event handling	keyboardFocusChange:		
pen	mouseDown:		
naming	mouseEnter:		
stepping and presenter	mouseEnterDragging:		
menus	mouseLeave:		
halos and balloon help	mouseLeaveDragging:		
mouseDown: evt			
"Handle a mouse down event. The default response is to let my eventHandler, if any, handle it."			
<pre>self eventHandler ifNotNil: [self eventHandler mouseDown: evt fromMorph: self].</pre>			

2(b) Custom event handling [2]

- An example of handling mouseDown event:



The screenshot shows a 'Class Browser: FishMorph' window. It displays a table with columns for 'instance', '?', and 'class'. The 'FishMorph' class is listed with 'instance' and 'class' columns. Below this, there is a section for 'event handling' and 'drawing'. The 'mouseDown:' event handler is shown with the code: `mouseDown: evt` and `self color: Color random.`

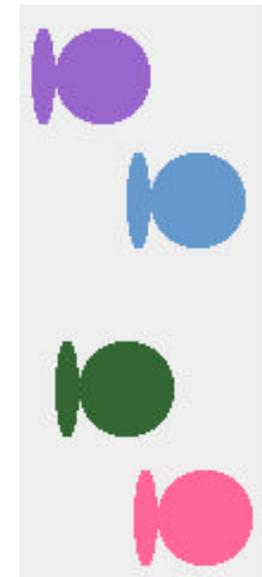
	instance	?	class
FishMorph			
-- all --			
event handling			
drawing			
mouseDown: evt			
self color: Color random.			

- However, this is not enough...

2(b) Custom event handling [3]

- Squeak does not want to dispatch all events to every Morph in the world (inefficient)
- To register interest in an event, you may have to override a **handlesXXX:** method, e.g.:

Class Browser: FishMorph			
FishMorph	instance	?	class
-- all --	handlesMouseDown:		
event handling	mouseDown:		
drawing			
handlesMouseDown: evt			
↑ true			



More about events...

- Event-driven programming is a big idea
- Good graphical toolkits provide a rich interface to send/receive/register interest in various events.
- Examine the “event handling” method category in the **Morph** base class for event handling methods.
- **MorphicEvent** (in class category Morphic-Support) is the class of the “evt” parameter received by the event handling methods.

3. Animating Morphs

- **Morph** defines a bunch of methods related to time-varying properties. Among the most important:
 - **step**
 - **stepTime**
 - **startStepping**
 - **stopStepping**
- These have the intuitively obvious meanings...
- As usual, override to make stuff happen

4. Hierarchical composition

- Most toolkits have a notion of “containers”, embodied in a class.
- **Container** is itself usually a subclass of the base **Component** class, so that Containers can recursively contain Containers.
 - (“Composite” design pattern – Gamma et al.)
- In this fashion, arbitrarily complex trees of components can be created.

Hierarchical composition in Morphic

- Morphic allows *all* Morphs to be containers
 - (some are better suited than others)
- Morph method **addMorph:** can be used to add any morph to any other.
- Note that **addMorph** alone does not constrain the position of submorphs!
 - A submorph may live outside its parent's physical area.
 - But, when this happens, painting often malfunctions

Composition, ct'd

- If you create your own specialized container (e.g., `BouncingAtomsMorph` in `Morphic-Demos`), you probably should not call `addMorph` directly
- Instead, create your own method, with a logical name, that calls **self addMorph**
 - (e.g., **addAtom:**)

Composition and delegation

- Adding components to containers allows the container to **delegate** responsibility for certain actions to its child objects
 - BouncingAtomsMorph need not explicitly define behavior of all atoms
- A fundamental principle of OOD: use hierarchical composition to build objects out of other objects.