

*Swarm Idioms:
Little Things Worth Knowing*

Paul E. Johnson
University of Kansas
Swarmfest 2005
Torino, Italy June 5, 2005

Item #1: What is a “Swarm”?

- First guess: Group of interacting Agents
 - First answer: First Guess completely wrong
 - Swarm is a protocol/class that has access to the scheduling framework
 - Particularly, it has access to
 - (id <Activity>)activateIn: SwarmContext
 - ActivateIn does vital work of weaving different levels together in the time chain
-
-

Item #2: Objective-C syntax

- Message: brackets [] indicate an object or Class is sent a message
 - `returnValue = [messageReceiver
theMessage: anArgument];`
 - `returnValue = [messageReceiver
messageArg1: argument1
arg2: argument2];`
 - `[fred getGroceriesMeat: steak Fish: cod];`
 - The message must “match up” against the methods listed in the class's *.h file
-
-

Item #3: Classes and Protocols

- `MyThing = [SomeClass createSomething:
someOption];`

`myThing` is an “instance” of class `SomeClass`

- `myThing` inherits all variables and methods in `SomeClass` (and its ancestors)



Super and Self

- **self** is defined as the Objective-C object in which code is executing.
- If it is not “in an object,” there is no “self”
- If you say:

[super doThisAndThat];

does it mean “super” does something?

- **No!** It means the method 'doThisAndThat' is found in a class from which the object inherited the method
-
-

Object Declarations

- Generic declaration:

`id anAgent;`

- Specific declaration:

`Agent * anAgent;`

- Protocol offers “hint”:

`id <myProtocol> anAgent;`

- Specific declaration preferred to help compiler find the methods you want (avoids confusion over duplicate method names)
-
-

Why Protocol?

- Protocol: list of methods an agent can answer to
- If you can't
 - directly subclass
 - create an instance

Protocol usage is the “best you can do” to let compiler check your work



Item #4: Memory Zones

- C requires explicit memory allocation
- Swarm uses a Zone concept
 - Zone is an object that can allocate memory when you need it.
 - Objects can be grouped by Zones (debugging).
- If a SwarmObject wants memory, it has to find its own Zone to ask for some:

```
id <Zone> myZone = [self getZone]
```

- Usually implicit:

```
id <List> myList = [List create: [self getZone]];
```

But there's a counterexample in Model Swarm!

- In ModelSwarm.m, one often finds:
`id <List> myList = [List create: self];`
 - Explanation: In a Swarm object:
`self = [self getZone];`
 - Swarm objects are subclassed from Zone, so they are Zones.
 - GUISwarm (like ObserverSwarm) is also a Zone
 - Read old Swarm programs, see this was not always true.
-
-

Item #5: Creating Objects

- Swarm designers conceptualized the creation/use of objects as 3 phases
 - **Creating:** permanently fixing attributes that are “once and final”
 - **Setting:** methods that can be called during the creating phase or later
 - **Using:**
 - This paradigm causes a particular Swarm style of writing programs.
-
-

createBegin, createEnd

```
anObject = [SomeClass createBegin: self];  
[anObject setThisVariable: 5];  
[anObject setThatVariable: 22];  
anObject = [anObject createEnd];
```

- +createBegin: is a “Class method”. We ask the class to carry out the first phase of creation
 - -createEnd is an “instance method”. An object carries “closes off” its CREATING phase.
 - After createEnd is called, only SETTING and USING methods can be used
-
-

createEnd: good chance to initialize

- C programs react badly when “uninitialized” variables are used.
- Example: suppose and IVAR x is not initialized

```
int y = 3 + x;
```

will produce gibberish.

- the createEnd method is a good place to set variables like x.



createEnd

Common usage:

```
- createEnd  
{  
  x = 0;  
  return [super createEnd];  
}
```

What's that [super createEnd] ?? super's createEnd

Why return [super createEnd] ?? just “self”?

createEnd: maybe better to:

- - createEnd
{
 [super createEnd];
 x=0; //put after to undo super's behavior
 return self;
}

Create: is a shorthand

- If you use the “create:” method, the Swarm library will (behind the scenes) run

```
createBegin:  
createEnd
```

- In other words, these are the same:

```
id myObject = [SwarmClass create: self];
```

- and

```
id myObject = [SwarmClass createBegin: self];  
myObject = [myObject createEnd];
```

Forget createEnd: big problem!

- Perhaps the most frequent cause of program crashes and unexpected behavior:
-
- User forgets `createEnd`:



Sometimes I prefer to have an `-init:` method

- The Archiver takes objects out of storage, bypassing `createBegin:` and `createEnd`.
 - This creates an initialization problem.
 - “nil” objects may exist.
 - `init:` method is inserted in some models to make sure that variables & objects are initialized
 - Same actions could be in `createEnd`, except for Archiver issues.
-
-

Item #6: Variables

A. IVAR: “instance variable”

- declared in “*.h” file
 - Each “instance” of a class has its own instance variables
 - How individuals remember their unique attributes
-
-

B. Local (Automatic) Variables

- Declared inside methods in *.m file
- Value should be initialized in method

```
int x = 0;
```

- Values “forgotten” after method is finished
 - Can declare as “static” if you want value to be remembered
-
-

Item #7: Iterating over Collections

- Suppose myList is full of things.

```
id <List> myList= [List create: self ];
```

- Here's a bad way to iterate

```
int i;
```

```
for (i=0; i < [myList getCount]; i++)
```

```
{
```

```
    id anObject = [myList atOffset: i];
```

```
    {do something to anObject}
```

```
}
```

- Its slow! atOffset: in a List repeatedly counts up from 0.



Recommended way to iterate

```
id anObject;  
id <Index> index=[myList begin: [self getZone]];  
  
for ( anObject=[index next];  
      [index getLoc]==Member;  
      anObject=[index next])  
    {  
      [goes through whole collection, even nils];  
    }  
[index drop];
```

- Member is symbol for a valid collection element

Item #8: Swarm Arrays and Lists

- Array: allocate N “slots” for objects.
- Fast access
 - retrieve:
[anArray atOffset: 5];
 - insert:
[anArray atOffset: 5 put: anObject];
- Does not allow “addLast:” (as does List)
- index usage same as with Lists
 - but atOffset: not so slow as with Lists...



Item #9: Command Line Arguments

- Run a model with `–help` to see command line options
 - Short form (one dash, no equal sign)
`./heatbugs -b -S442432`
 - Long form (two dashes, one equal sign)
`./heatbugs –batch –seed=442432`
 - Several built in command line options
 - New command line options can be added by adding a user “Arguments” class
-
-

Item #10: Random Numbers

- pseudo random numbers (MT19937 is default)
 - Swarm Distributions
 - Uniform Double
 - Normal
 - Equally likely integers
 - Binomial
 - Same Seed = Same numbers every time
 - Random Re-Seed with Swarm models:
 - # ./heatbugs -s
 - or specify seed yourself:
 - # ./heatbugs -S2344322
-
-

Item #11: Runtime Crashes

Many possible causes of crashes

- Forget “createEnd”
- Accidental assignment of local variable (ex: color)
- Schedule an agent to do something impossible.
 - Obj-C is “run time” binding
 - Run will crash if you send a Message that agent can't carry out
 - Sometimes terminal output will reveal problem
 - Object does not respond to “xxx”

Here's a bad thing to do in sss

- [modelActions createActionTo: agentList
message: M(step)];

Changed to

- [modelActions createActionTo: agentList
message:
M(jumpOffBridge)];
 - That does compile and tries to run
 - Runtime crash says “Segmentation fault”
 - Very difficult to track down cause
 - Lesson: Be very careful in writing messages!
-
-

The Debugger: GDB

- gdb: GNU debugger
 - # gdb ./sss
 - > run
- when it crashes, type “bt” to get backtrace
- Or set a “breakpoint”
 - > break ModelSwarm.m:120
 - installs a “break point” at line 120 in ModelSwarm.m
 - run model, then “step” or “next” through code

GDB helps, sometimes

- If you have a crash, and you ask for help, the first thing we ask for is a “backtrace”
 - Sometimes frustrating because
 - none of “your model code” seems to cause the crash
 - debugging symbols are missing from pre-compiled libraries
 - doesn't help in finding “bogus selector” crash
 - Very helpful with some kinds of crashes:
 - accessing “out of bounds” points in grids
 - looping “out of bounds” in an array
-
-

Item #13: GUI is not just eye-candy

- Graphs may reveal coding mistakes
 - Clicks on Rasters may let you interact with agents and see their instance variables
 - sss-2.3: both right and left click
 - click & probe functionality is only “real reason” to link a ObjectGrid2d lattice of objects with the display grid on the screen.
 - could just let agents draw on screen
 - but then could not find them by clicking
-
-

Item #14: printf/fprintf

- `printf`
 - `printf("PJ says %d", aVariable)`
 - Ordinary C way of writing to the “screen”
 - Common way of finding out “what's going on”
 - `fprintf(stderr, "PJ says %d", aVariable);`
 - Does same thing
 - Better in case program crashes because output is forced through in sequence
-
-

Item #15: Langton's advice

- Chris Langton writes in the original Swarm tutorial
 - get a program that works.
 - make small, incremental changes.
 - make sure it does not break.



Item #16: Read Your Compiler Output

- Some models will run despite the presence of Warnings
 - Nevertheless, “good practice” is to fix code to eliminate all warnings.
 - Nobody in swarm-support will be interested in helping you if you send them a package of code that does not “at least” compile cleanly.
-
-