Defobj Library

Documentation and Implementation Status

Overview

The defobj library supports the style of object-oriented programming that is used throughout Swarm. It defines a specific style for using the Objective C language that includes its own standard conventions for creating objects and for storage allocation, error handling, and debugging support.

Documentation

The defobj library follows the Swarm `library interface conventions`

`(defobj/library.html)`
. It has following standard sections of documentation:

- `Usage Guide`
  `(defobj/guide.html)`
- `Advanced Usage Guide`
  `(defobj/adv.html)`
- `Interface Reference`
  `(defobj/ref.html)`
- `Subclassing Reference`
  `(defobj/subclass.html)`
- `Interface Design Notes`
  `(defobj/design.html)`
- `Implementation Notes`
  `(defobj/impl.html)`

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Defobj Library: Documentation and Implementation Status

The current interfaces defined in defobj are expected to remain stable, with the exception of those relating to package structure and custom-generated classes. Those facilities are in process of being totally replaced, and will not be documented further until that replacement is complete. Details of support for defining a customized type will also be provided later.

All the interfaces defined in the header file are currently implemented, except for the various types of zones that would hold all allocated storage in a collection of local pages, and that would provide automatic reclaim of unused storage. The current zone implementation supports all defined allocation messages, and also maintains the population of the zone that consists of all objects created directly within it.

The Usage Guide section is only a start, but it does include subsections that discuss the style of Objective C programming adopted for use throughout Swarm. There is a full set of Interface Reference sections, though some details remain to be filled in.

Throughout the documentation, a parenthesized comment that starts with (.. indicates an editorial comment on the current status of implementation or documentation.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Library Interface Conventions

Overview

Some of the basic Swarm libraries are implemented using object definition conventions established by the defobj library

(../defobj.html)
. This document explains how to read and interpret the header files and interface definitions published by such libraries. It also explains the typical structure of documentation provided for libraries that follow these conventions.

These conventions are documented under the following subsections:

- library header file
- object type definitions
- GridTurtle example
- global object symbols
- interface design conventions
- documentation structure

Library Header File

For a library that adopts these conventions, a library is not merely a collection of source files that are compiled into a library archive under control of a make file. Instead, Objective C source files in the library are processed in a special way to publish their definitions not only as header files, but as generated objects available at runtime to make full use of the library. A library processed in this way is referred to as as a "package." (Note: some of the newer Swarm libraries, such as objectbase and random, don't undergo this special processing and yet still follow all the library interface conventions described in this document.)

The defobj
(../defobj.html)
library documents the full details of special processing performed on a package. For simply using a library, the key fact to keep in mind is that the entire public interface to a library is declared in the one header file having the same name as the package itself, plus a trailing .h suffix. Additionally, the header file of a library normally documents only its public interface, in a way that is completely separated from the implementation of the objects it specifies.

The separation of implementation means that a library publishes its interface entirely without reference to any Objective C classes which implement its objects. Even though classes are often thought of as separating the interface of an object from its implementation, this separation is far from complete. Not only do classes typically contain many internal methods not intended for external use, but they also define a particular storage format for an object defined as instance variables.

A library instead publishes its interface as a set of public object types. These object types may also be supplemented by global object constants called symbols. Both these kinds of definitions normally appear only in the header file of a library. The remaining source files in a library normally contain the classes which implement the object types.

In a library, the files which implement classes (including class header files), need not ever be referenced simply to make use of the implemented capabilities of the library object types. Documentation for the library is normally expressed entirely in terms of the types and symbols published in the library header file. If a feature does not appear in the library header file, it should not be considered part of a supported public interface.

Individual class header files are required to subclass from existing implementations, but interfaces for subclassing are an entirely separate issue from normal public use of an object library. Class inheritance can be a powerful implementation technique, but extension of an existing class framework is typically safe only if performed in explicitly permitted ways. If a library supports subclassing at all, it must carefully state which classes may be subclassed and in what ways. For a library package, this information is supplied outside the library header file. The library header file specifies only the interfaces by which objects are intended to be used, whether implemented by a local class or an external subclass.

Remaining sections of this document explain the declarations which appear in a library header file, and end with a suggested structure of documentation to be provided for a library. The libraries of Swarm mostly follow this structure.

Object Type Definitions

The interfaces to objects defined by a library are specified by object type definitions. An object type defines only a set of messages which may be sent to an object over various phases of its lifetime. An object type makes no commitment to the classes that might be used to implement the object.

Multiple classes may all implement the same messages belonging to a type. The independence of types and classes means that different classes can provide alternate implementations of the same object type. For example, a particular implementing class might be selected to optimize the implementation for a particular case.

Object types are similar to protocols defined by Objective C, and the declarations appearing in a library header file are a minor adaptation of Objective C protocol syntax. A key difference from protocols is that object types are published as real external objects that may be used at runtime to create instances of a type. A futher difference is that the object types of the defobj library are divided into separate interfaces that define distinct phases of an object's life cycle.

GridTurtle example

An example will help illustrate the features of type definitions supported by the defobj library. Throughout documentation of the basic Swarm libraries, a series of running examples will be based on a simple type of object belonging to sample Swarm simulations. This type of object is an agent that move around on a two-dimensional grid, always moving in a current direction that it maintains internally. This agent is like a "turtle" of the original Logo system, except that its position is constrained to discrete integer values of its X-Y coordinates, and its direction is always one of the four orthogonal directions north, east, south, or west.

Following is a complete library header file for a library which defines such an object, called GridTurtle:

```
/*
Name:         GridTurtle.h
Description:  object type for Swarm example programs
Library:      grid
*/

#import <defobj.h>

@deftype GridTurtle <Create, Drop, CREATABLE>CREATING
- (void) setXLocMax: (int)xLocMax;
- (void) setYLocMax: (int)yLocMax;
SETTING
- (void) setXLoc: (int)xLoc;
- (void) setYLoc: (int)yLoc;
- (void) setXLoc: xLoc setYLoc: yLoc;

- (void) setDirection: direction;
USING
- (int) getXLoc;
- (int) getYLoc;

- getDirection;

- (void) move: (int)distance;
- (void) turn: (int)angle;  // angle measured in units of pi/2 (90 deg.)
- (void) print;
@end
```

```
id <Symbol> North, East, South, West;

#import "grid.xt"
```

An object type is defined by an @deftype declaration. (Note: newer libraries, including objectbase and random, now follow the library interface conventions without using this special @deftype tag. Instead they use just an ordinary @protocol declaration, but otherwise they follow all the structure explained in this document.) The syntax of such declaration is identical to that of an Objective C @protocol definition, except for the entirely uppercase keywords (CREATABLE, CREATING, SETTING, USING) appearing in the GridTurtle example above. All these modifications of Objective C syntax are accomplished by simple preprocessor macros; no extensions to the language compiler are involved.

When this library header file is processed (by a special rule in a make file), an external object id with the name GridTurtle is automatically published. The name of a defined type becomes an ordinary object that accepts specific messages defined by the defobj library. The defobj library
(../defobj.html)
explains the details of such messages; the only purpose here is to explain the basic sections of a deftype declaration.

deftype declarations follow the syntax as Objective C protocols for inheriting messages from each other: a list of names enclosed in angle brackets (e.g., <Create, Drop, ...> above) gives the names of other declared types containing messages to be supported by the new type as well. (These types referenced here are defined by the imported file <defobj.h>.) Like protocols, full multiple inheritance of types is supported. The same messages may be inherited any number of times through any path with no different effect than if inherited or declared just once, so long as no conflicts occur in any of their argument or return types.

The CREATABLE tag appearing in the inherited type list above is a special type which defines no messages of its own, but merely marks the type as one which supports direct creation of instances of the type. Without this tag, the only role of a type is to define messages for inheritance by other types. With this tag, the global type object has a particular implementation that supports object creation using standard messages defined in defobj.

The declared messages of the type may be separated into sections marked by the special uppercase tags such as CREATING, SETTING, and USING above. (Currently, these are the only such tags which may occur.) These sections each define messages belonging to a particular defined "interface" of the object type, which are further combined into distinct "phases" of an object lifecycle supported by defobj messages. Further explanation of the interfaces and phases defined by this example are provided in the Usage Guide
(guide.html)
of the defobj library.

The following line at the end of the <grid.h> header file above is required as part of the standard conventions for coding a library header file: This line adds no information to the header file, but is required as part of the system by which definitions are automatically generated.

```
#import "grid.xt"
```

Global Object Symbols

The <grid.h> header file above also contains the declaration:

```
id <Symbol>  North, East, South, West;
```

Lines that declare global id variables of type Symbol, EventType, Warning, or Error (using the angle bracket syntax of id variables conforming to a protocol) are processed somewhat like deftype declarations in that they also produce global id variables initialized to support particular messages defined by defobj. These global variables, however, are not used to define or implement other message interfaces, but only to define certain fixed capabilities referenced through their global object names.

If declared as a Symbol, as in the case here, the generated objects have no particular behavior of their own (other than the character string of their name), but only serve to define unique global id constants which may be used as distinct named values in messages. In this example, the current direction of a GridTurtle object is represented by one of the symbol names North, East, South, or West. These values are like the enum constants of the C language, except that they are defined as full Objective C objects, and may be used with further restrictions. An EventType, or a Warning or Error, defines a further subtype of a Symbol constant with further specialized messages documented in defobj.

Interface Design Conventions

A variety of rules on naming and declaration of object types, symbols, and messages are followed by many of the Swarm libraries. These rules help establish a basic consistency on the library interfaces. Following is a list of such conventions that apply to a public library interface, some but not all of which are derived from standard Smalltalk or Objective C coding practice:

- Names of global object constants are capitalized. In the public interface, such names include types and symbols.
- Recapitalization separates words of a compound name (e.g., GridTurtle). Underscores are generally not used.
- Message names start with a lower-case character, and are named using verbs. Nouns that represent gettable or settable components of object state (e.g., Direction of a GridTurtle), are prefixed by get or set to indicate the action being performed.
- The Smalltalk convention that a message return the receiver of a message if there

is no other specific return value is generally *not* followed. If there is no specific return value needed from a message the return type is declared (void).

Documentation Structure

The following standard sections of documentation are suggested for libraries that follow the strict interface vs. implementation separation of library packages, each with purpose and typical contents as given:

- Usage Guide
  Tutorial introduction to a library. Focuses on the most common uses in an order reflecting the needs of a first-time user. Most explanation by means of progressively elaborated examples. Serves as a guided tour of major library capabilities. No attempt at reference-style completeness.
- Advanced Usage Guide
  Continues overview of all significant capabilities of a library, including those which might be needed by advanced users customizing or extending built-in capability. Provides examples of specialized uses extending beyond normal, basic usage, but still using built-in features of the library framework. Helps to simplify the Usage Guide by providing a location for overflow of more advanced features.
- Interface Reference
  Complete, concise summary of all features of a library. Fills the role a Unix "man page" in providing comprehensive definition of library services. Little or no concern to provide path of tutorial introduction. Does not contain extended examples, at most only fragments of examples which serve needs of specification.
- Subclassing Reference
  Documents the rules for writing new classes which subclass from classes that implement the library. Because a library might use complex combinations of classes to implement the range of behaviors defined by its types, the classes in a library are not automatically usable as superclasses of user-defined classes. Each library documents which classes are available for use as superclasses, and the specific rules that must be followed when subclassing from these classes.
- Interface Design Notes
  Explains why choices were made the way they were in the design of the public interface to a library, including other alternatives considered. Serves as a "Rationale" document for the documented interface. May include historical background deliberately left out of the preceding sections. May also include references to other work.
- Implementation Notes
  Explains the structure of classes by which the types of a library are implemented. Summarizes the status of implementation if still incomplete and lists items of possible future work. Provides overview and high-level structure of the implementation in whatever ways would best guide a reader of implementation source code. May

also discuss tradeoffs considered along with references to related or supporting work.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Defobj Library: Usage Guide

Contents

Why Swarm uses Objective C

Swarm uses object-oriented programming not only because this is a good way to build general-purpose software libraries, but because the very concept of an object is at the base of how you build a model in Swarm. To build a simulation in Swarm, the first step to define the various kinds of objects that can inhabit some real or artificial world, and the second step is to define the kinds of events that can occur to these objects, including all the different ways that the objects can interact with each other.

The basic concept of an object is that it responds to external events, and that the only thing that really matters about an object are the ways it can be observed responding to these events. In an object-oriented programming system, an object is represented by some uniquely identified chunk of data, and the events are the dynamic operations that can be made to occur on these objects.

Different object-oriented programming systems define the operations that occur on objects in different ways. For example, in C++ the operations are called "member functions" and look much like an ordinary function call in the C language. In Objective C, an operation on an object is called a "message" and has a special syntax by which you call it, but a message also has arguments and behaves in many respects like a call to a function.

The key requirement for Swarm is to be able to make all these operations happen to any object at any time, whenever they're supposed to occur within some simulated world. It stores these operations inside its own data structures, and when you run a simulation in Swarm, the Swarm system itself traverses these data structures to make the necessary operations happen at the proper time.

Swarm needs to be able to make any kind of action happen to any kind of object at any time, and to do this it needs very general-purpose structures that hold some kind of representation of the actions themselves. That's a special requirement that not every object language provides. You already have enough work to define the kinds of objects that can exist inside your model, and Swarm doesn't want you to have to define an even larger number of objects for every kind of event that might occur to your objects. Instead, it wants the object language to provide it with a representation of the operations on objects that you've already defined. Swarm can then store these representations in its own data structures, and make the operations happen whenever it needs to.

In Swarm, the representation of events that happen on objects are just as fundamental to the model as the objects themselves. That's why Swarm is a "discrete-event" simulation system. But if the object system doesn't provide a general-purpose enough representation of events as well as objects, there would be a lot more work to do. The Objective C system, through its special data type called a "message selector," provides a representation of operations that is flexible enough to do this, but the C++ language does not. (In C++, the compiler requires more information about an operation than the event structures in Swarm would be able to provide.)

There's a host of additional reasons why Objective C has also been a good match for the requirements of the Swarm system. Like C++, the operations on objects can be compiled to a very efficient form (though Objective C does require a little more overhead than C++ to get the operation started). This efficiency can be very important for a simulation, since simulations can run for very long periods of time to explore all the behavior that might occur within their simulated worlds. Other languages, such as Lisp, Smalltalk, and Java, also have the "dynamic message dispatch" feature that would make general-purpose event structures possible, but they still carry significant added overhead compared to C.

Objective C is also a very simple extension to the C language. Basic knowledge of C is already widespread, and a few days are typically all that is needed to learn the few additions of Objective C. (Really learning the concepts of objects, however, can take much longer, no matter what object language you try to use.)

Objective C was also a good choice for Swarm because it has a high quality, freely distributable implementation in the GNU C compiler. One of the main concerns about Objective C is that it isn't nearly as widely known or used as other languages like C++, but at least the GNU C compiler assures it will be available on machines where Swarm needs to run. The OpenStep system of the NeXT corporation, and the parallel GNUStep project, also help assure that Objective C is a living language.

There are even more powerful aspects of Objective C that Swarm takes advantage of. Some of these are described in the Advanced Usage Guide
(adv.html)

of the defobj library. But many others are at the heart of how Swarm uses Objective C, and so are dealt with in the rest of this Usage Guide. The entire purpose of the defobj library is to define a standard style for the use of Objective C in Swarm. This style is backed up by a library of foundation classes that Swarm provides to support this style.

Swarm provides its own foundation classes even for such basic operations as creating an object, and other support that user classes ordinarily receive from a builtin Object superclass. A good understanding of the defobj library is essential for Objective C programming in Swarm. Like any programming language, Objective C requires learning not only the rules of the language itself, but also a standard library of initial capabilities that you build from. Objective C doesn't really have a single official standard library (or language definition either, for that matter), but the NextStep foundation libraries (and the NextStep language definition) come very close to this status.

For a variety of reasons explained here and in the Advanced Usage Guide
(adv.html)
, Swarm doesn't use a standard library based on the NextStep foundation interface. You can still learn the Objective C language from other available sources, but you have to be careful to sort out the language level from the standard object and library support they also discuss.

The index page
(../../index.html)
of the Swarm documentation provides a variety of links to other Objective C resources. In particular, it provides a link to a complete on-line reference for the Objective C language as defined by NextStep and implemented by the GNU C compiler. None of the Swarm documentation attempts to duplicate this coverage of the basic language; Swarm assumes that you learn Objective C from other available sources. The NextStep reference(*Object-Oriented Programming and the Objective C Language*
(http://www.next.com/Pubs/Documents/OPENSTEP/ObjectiveC/objctoc.htm)
) is the single best available source, and it can also be ordered as a hardcopy book from the NeXT publications page
(//www.next.com/Pubs/Documents/OPENSTEP)
. Don't try to program in Objective C without it.

Swarm style of Objective C Programming

No matter where you've learned to program in Objective C, don't try to use Objective C in Swarm without understanding the special ways in which Swarm adapts its use of the language and the language runtime system. Your place to find this information is right here in the defobj library. Defobj not only provides the most basic layer of foundation class libraries for Swarm, but also serves as the place where all the rules and guidelines for the use of Objective C in Swarm are gathered together.

Some of these rules and guidelines don't even require any specific software to implement them, but are just conventions and recommended style that anyone programming in Objective C can follow when their goals are similar to those of Swarm, and to a large extent often do. These kinds of conventions are the focus of this section; following sections discuss aspects of Swarm Objective C style that depend on specific software support.

Lots more to come... For now, see Library Interface Conventions
(library.html)
for some of this information, and Swarm tutorials for much of the rest.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Defobj Library: Advanced Usage Guide

(This section of documentation is not yet available.)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Defobj Library: Interface Reference

Dependencies

The defobj library is defined and documented using the library interface conventions

(library.html)
established by the defobj library. It imports the Object superclass and other standard type definitions of the GNU Objective C runtime system.

> The collections library must always be linked along with the defobj library. Even though the interface definitions of defobj do not depend directly on collections, the implementations of these libraries both require the presence of the other. Only the collections library should be initialized directly; initialization of the collections library automatically initializes the defobj library as well.

Contents

- module definition conventions (module.html)
- create combination messages (combo.html)
- global portability assumptions (globalport.html)
- global definitions (globaldef.html)

Marcus G. Daniels <mgd@santafe.edu> (mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Name

DefinedObject – standard support for all objects

Synopsis

```
//
// DefinedObject -- object with defined type and implementation
//
@deftype DefinedObject
- getZone;
- (BOOL) respondsTo: (SEL)aSel;
- getClass;

- (ref_t) addRef: (notify_t)notifyFunction withArgument: (void *)arg;
- (void) removeRef: (ref_t)refVal;

- (int) compare: anObject;

- perform: (SEL)aSel;
- perform: (SEL)aSel with: anObject1;
- perform: (SEL)aSel with: anObject1 with: anObj2;
- perform: (SEL)aSel with: anObject1 with: anObj2 with: anObj3;

- (void) setDisplayName: (const char *)displayName;
- (const char *)getDisplayName;

- (void) describe: outputCharStream;
- (void) xprint;
- (void) xprintid;
@end
```

Description

DefinedObject is the top-level supertype for all objects that follow the object programming conventions of the defobj library. The messages defined by this type are the only messages which should be assumed to be automatically available on objects that follow these conventions. In particular, use of messages defined by the Object superclass of the GNU Objective C runtime should not generally be assumed because future implementations of some objects might not give continued access to them.

The DefinedObject type defines a minimum of standard messages, and leaves to other types the definition of message that might or might not apply in any general way to particular objects.

Inherited behavior

The DefinedObject type inherits no behavior from any other type. It is the top of the interface type hierarchy for objects that follow the defobj programming conventions. All objects that follow these conventions should inherit from DefinedObject, either directly or indirectly through other types.

Defined behavior

```
- getZone;
- (BOOL) respondsTo: (SEL)aSel;
- getClass;
```

The getZone message returns the zone in which the object was created. The respondsTo: message returns true if the object implements the message identified by the selector argument. To implement a message means only that some method will receive control if the message is sent to the object. (The method could still raise an error.) The respondsTo: message is implemented by direct lookup in a method dispatch table, so is just as fast as a normal message send. It provides a quick way to test whether the type of an object includes a particular message. getClass returns the class that implements the current behavior of an object.

Defined behavior

```
- (ref_t) addRef: (notify_t)notifyFunction withArgument: (void *)arg;
- (void) removeRef: (ref_t)refVal;
```

These messages add or remove an external reference to an object that is notified whenever the object is changed or relocated. (More details to follow.)

```
- (int) compare: anObject;
```

```
- perform: (SEL)aSel;
- perform: (SEL)aSel with: anObject1;
- perform: (SEL)aSel with: anObject1 with: anObj2;
- perform: (SEL)aSel with: anObject1 with: anObj2 with: anObj3;
```

These messages are have implementations in the Object superclass, excppt for the three-argument form of perform:. They have local implementations in defobj, however, just to make subclassing from Object unnecessary. They are currently the only messages defined in Object that are recommended for use in Swarm.

```
- (void) setDisplayName: (const char *)displayName;
- (const char *)getDisplayName;
```

These messages assign a character string as a name that identifies an object for display or debug purposes. A global function, xsetname(), is also available to set the name by calling the function from the debugger.

```
- (void) describe: outputCharStream;
- (void) xprint;
- (void) xprintid;
```

The describe: message prints a brief description of the object for debug purposes to the object passed as its argument. The object passed as the outputCharStream argument must accept a catC: message as defined in String and OutputStream in the collections library. Particular object types may generate object description strings with additional information beyond the built-in default, which is just to print the hex value of the object id pointer along with the name of its class, and the display name of the object, if any. xprint prints the standard describe string to standard output; it is also called by the xprint function that may be called from the debugger. xprintid prints just a one-line describe string, consisting of the built-in default.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Name

Create – create a customized instance of a type

Synopsis

```
//
// Create -- create a customized instance of a type
//
@deftype Create <DefinedObject>
CREATING
+ create: aZone;
+ createBegin: aZone;
- createEnd;

+ customizeBegin: aZone;
- customizeEnd;
- customizeCopy: aZone;
@end
```

Description

The Create supertype defines standard messages that provide a general-purpose protocol for creating new objects. These messages may be used either to create a new instance of a type in one message, or to bracket a series of messages that customize available options for an object to be created. The separation of create-time specifications from later behavior of an object gives substantial flexibility to adapt a generic type to particular needs.

These create messages may be implemented either by a type that hides its implementing classes, or directly by a class that adopts these messages as a uniform interface for creating objects. If implemented directly by a class, then the class object serves as the type object in all message descriptions that follow. Otherwise, a type object might be implemented in a variety of ways that guarantee only that published messages on a type are accepted.

In addition to the create messages defined by Create, an object type may support any other messages of any other names or calling conventions. These messages define only a standard method for creating new objects that other types are free to inherit and implement in conformance with a uniform convention. Further conventions are established elsewhere (create combination messages
(combo.html)

) for standard ways in which create messages that combine several steps can be combined.

Inherited behavior

The Create type has no inherited behavior beyond the universal messages of the DefinedObject

(DefinedObject.html)
type.

Defined behavior

```
+ create: aZone;
```

The create: message creates a new instance of a type with default options. The zone argument specifies the source of storage for the new object. The receiving object of this message is a previously defined type object. The message is declared as a class message (with a + declaration tag) to indicate that the message is accepted only by the type object itself rather than an already created instance of the type (which a - declaration tag otherwise defines).

The create: message returns the new object just created.

Defined behavior

```
+ create: aZone;
```

The create: message creates a new instance of a type with default options. The zone argument specifies the source of storage for the new object. The receiving object of this message is a previously defined type object. The message is identified as a class message (a + declaration tag) to indicate that the message is accepted only by the type object itself rather than an already created instance of the type, which most messages of a type (those with a - declaration tag) otherwise define.

The create: message returns the new object just created. This object is an instance of some class selected to implement the type. The class which a type selects to implement an object may be obtained by the getClass message, but is not otherwise visible to the calling program. A caller never refers to any class name when creating objects using these messages, only to type names, which are automatically published as global constants from any @deftype declaration.

```
+ createBegin: aZone;
- createEnd;
```

The createBegin: and createEnd messages bracket a series of messages that specify options for an object being created. The intermediate messages can set values defined as parameters of the type, or provide other forms of specification using available messages. A particular object type defines the specific messages that are valid for sending during this interim creation phase.

createBegin: returns an interim object intended only for receiving create-time messages. If a type was defined by a @deftype declaration, these messages are those appearing in either the CREATING or SETTING sections. Otherwise, the messages valid as create-time messages are defined by the type without any specific syntactic marker.

The createEnd message completes the process of specifying available options for an object being created. Typically it validates that requested options are valid and consistent with one another, and raises an error if they are not. The standard, predefined error InvalidCombination
(Error.html#stderr)
may be raised by createEnd to indicate an invalid combination of requests, or other, more specific forms of error handling may be used.

If all requests received since the initial createBegin: are valid, both individually and in combination with each other, then createEnd determines a finalized form of object that satisfies all requests received and then returns this object. Any additional storage required for the finalized object is taken from the same zone originally passed to createBegin. The object may have whatever implementation is selected to best satisfy a particular request. Different requests may result in entirely different implementations being returned. The only guarantee is that a returned object supports the messages defined for further use of the finalized object. If a type was defined by a @deftype declaration, these messages are those appearing in either the SETTING or USING sections.

On return from createEnd, the id of the interim object returned by createBegin: is no longer guaranteed to be valid for further use, and should no longer be referenced. A variable which holds this such an id can be reassigned the new id returned by createEnd, so that the same variable holds successive versions of the object being created. Following is a fragment that illustrates typical coding practice using createBegin: and createEnd:

```
newArray = [Array createBegin: aZone];
  [newArray setInitialValue: aList];
  [newArray setDefaultMember: UnsetMember];
  [newArray setCount: [aList getCount] * 2 );
  newArray = [newArray createEnd];   // ! note reassignment of newArray
```

By resetting the variable to the returned value of createEnd, there is no possibility that the id value first returned by createBegin: can ever be used again. Sometimes, the id value returned by createEnd might be the same value first returned by createBegin:, if createBegin: was able to allocate the same object returned later by createEnd. In other cases, however, the object returned by createBegin: might be strictly a temporary object that is thrown away after createEnd determines a final object to be created and returned. To protect against changes in implementation, only the final createEnd return value should ever be used again after createEnd has returned.

```
+ customizeBegin: aZone;
- customizeEnd;
- customizeCopy: aZone;
```

Some types accept create-time messages not only when creating a new instance, but to customize a new version of the type itself. Objects created from a customized type will have all options preset that create-time messages sent to the customized type object have already set. If many objects all need the same create-time options, it is often simpler (and can also be faster) to create a customized version of a type first, and then create further instances from that type.

Customizing a type object does not modify the original type object, but instead creates a new type object that has the customizations built-in. A create: message on the new type object creates a new instance as if the same sequence of create-time messages had been sent to the original type object using createBegin: and createEnd. A type is customized by bracketing the sequence of create-time messages not with the createBegin: and createEnd messages used to create a new instance, but with customizeBegin: and customizeEnd messages instead. customizeBegin: returns an interim value for receiving create-time messages much like createBegin:, and customizeEnd returns the new, customized version of the original type. Following is sample code sequence that illustrates the use of these messages:

```
newArrayType = [Array customizeBegin: aZone];
  [newArrayType setCount: 100];
  newArrayType = [newArrayType customizeEnd];
  array1 = [newArrayType create: aZone];
  array2 = [newArrayType create: aZone];
  ... // [array1 getCount] and [array2 getCount] are both 100
```

The customizeCopy: message creates a new copy of the interim object returned by customizeBegin: which may be used for further customizations that do not affect the customization already in progress. It may be used to branch off a path of a customization in progress to create an alternate final customization. Following is a code fragrment which illustrates this usage:

```
newArrayType1 = [Array customizeBegin: aZone];
  [newArrayType1 setCount: 100];
  newArrayType2 = [newArrayType2 customizeCopy: aZone];
  [newArrayType2 setDefaultMember: UnsetMember];

  newArrayType1 = [newArrayType1 customizeEnd];
  newArrayType2 = [newArrayType2 customizeEnd];
  array1 = [newArrayType1 create: aZone];  // no DefaultMember option
  array2 = [newArrayType create: aZone];   // DefaultMember option set
```

customizeCopy may be used only on an interim object returned by customizeBegin: and not yet finalized by customizeEnd. The new version of the interim object being customized may be allocated in the same or different zone as the original version, using the zone argument required by customizeCopy:

The zone passed to customizeBegin: is the same zone from which storage for the new, finalized type object will be taken. This zone need not be the same as any instance later created from that type, since a new zone argument is still passed in any subsequent create message on that type.

Whether a customized version of a type can be created depends on the implementation of the type itself. If a type does not support customization, a customizeBegin: message on the type raises an error. All types defined by an @deftype declaration may be relied on to support at least one cycle of customization to create a new type object. Whether an already customized type object (returned by customizeEnd) supports a further cycle of customization (by another sequence of customizeBegin:/customizeEnd) depends on the implementation of the original starting type. A type should not be relied on to support more than one cycle of customization unless it is specifically documented to do so.

Any interim object returned by either createBegin: or customizeBegin: supports the getZone and drop messages that a finalized instance may also support. These messages are defined by the Drop
(Drop.html)
type, which is normally inherited by a type to declare these messages on a finalized instance. This type is not inherited by the Create type because the messages would then apply to the finalized instance, not to the interim object. Even though not declared, the messages are available on the interim objects nonetheless. The drop message on an interim object may be used if it turns out that a finalized version is no longer required after creation or customization has already begun.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Name

Drop – end existence of an object

Synopsis

```
//
// Drop -- end existence of an object
//
@deftype Drop
- (void) drop;
@end
```

Description

The Drop supertype defines the drop message, which is a standard message for indicating that an object no longer exists and will never again be referenced. Any future attempt to reference a dropped object is an error. This error may or not produce predictable effects depending on the level of debug checking and other factors.

Immediate effects of the drop message depends on the subtype of Zone
(Zone.html)
used to provide storage for the object. For some zone types, the drop message immediately deallocates storage for the object and makes the freed storage available for other use. Subsequent use could include the allocation of a new object at precisely the same location, resulting in a new object id identical to a previously dropped one. Other zone types (..none implemented yet..) might delay any reclamation of freed storage until a more comprehensive check can be made that other references no longer exist to the dropped object. When running in debug mode (see debug support
(globaldef.html#debug)
) the storage for a dropped object is filled with predictable contents so that an error is raised if there is any immediate attempt to us it again.

The Drop type may be inherited by any type that provides drop support for its instances. In addition to freeing the storage and invalidating the object, a drop message may release other resources acquired or held within the object. Not every object which can be created can also be dropped, and some objects can be dropped which are not directly creatable. Some objects may be created as a side effect of other operations and still be droppable, and some objects may be created with links to other objects and not droppable on their own. A type independently inherits Create or Drop types, or both, to indicate its support of these standard interfaces to define the endpoints of an object lifecycle.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Name

Zone – modular unit of storage allocation

Synopsis

```
//
// Zone -- modular unit of storage allocation
//
@deftype Zone <Create, Drop, GetOwner, CREATABLE>
CREATING
- (void) setReclaimPolicy: reclaimPolicy;
- (void) setStackedSubzones: (BOOL)stackedSubzones;
- (void) setPageSize: (int)pageSize;
USING
- getReclaimPolicy;
- (BOOL) getStackedSubzones;
- (int) getPageSize;

- allocIVars: aClass;
- copyIVars: anObject;
- (void) freeIVars: anObject;

- allocIVarsComponent: aClass;
- copyIVarsComponent: anObject;
- (void) freeIVarsComponent: anObject;

- getComponentZone;

- (void *) alloc: (size_t)size;
- (void) free: (void *)aBlock;

- (void *) allocBlock: (size_t)size;
- (void) freeBlock: (void *)aBlock blockSize: (size_t)size;

-               getPopulation;
- getSubzones;
- (void) mergeWithOwner;

- (BOOL) containsAlloc: (void *)alloc;
- getSubzone: (void *)alloc;

- (void) reclaimStorage;
- (void) releaseStorage;
```

```
- (void) xfprint;
- (void) xfprintid;
@end
```

Description

A zone is a source of storage for objects or other allocated data. Whenever a new object is created, a zone must be identified from which the storage for its instance variables, or other internal data, is obtained. A program may establish multiple zones to ensure that objects with similar lifetime or storage needs are allocated together, and in general to optimize allocation and reuse of storage.

Zones also maintain a collection of all objects allocated within the zone. This collection, referred to as the "population" of a zone, is a set of all objects which have been created but not yet dropped within the zone. Collections maintained automatically by zones can eliminate a need for other, separately maintained collections in applications that need to keep track of entire populations of objects. Collections of allocated objects can provide support for object query, external object storage, and automatic storage reclamation.

A zone may be used to obtain storage not only for objects, but also for raw storage blocks like those provided by the C malloc function. All objects and storage blocks allocated in a zone remain local to that zone. This means that allocation of storage in other zones does not affect the efficiency of storage allocation within a particular zone. For most zone types, individual allocations may still be freed within a zone, and total storage of a zone may grow and shrink according to aggregate needs. In addition to freeing individual allocations, an entire zone may also dropped. Dropping a zone automatically frees all allocations made within it, including final drop processing on any allocated objects that need it. Release of an entire zone can be much faster than individual release of each object within it.

The Zone type is a fully implemented type that provides default storage management support for objects and other allocated storage. It is also a supertype for other zones that implement alternative policies for use in specialized situations.

(..The current Zone implementation is a temporary one based on a simple pass-through of all allocation requests to the C library malloc function. Complete support of the zone population is already provided, however, as is support for internal storage mapping that will eventually be required for automatic storage reclamation Considerable improvements in storage allocation are possible when the current implementation is replaced by more efficient allocation contained in pages local to each zone. ..)

Inherited behavior

```
@deftype Zone <Create, Drop, CREATABLE>
```

A zone is created using standard create messages just like other objects. This means that a zone must identify another zone from which it obtains its storage. Storage is typically obtained from this other zone in large units called pages, which are then managed by the local zone to support internal allocations. The getZone message of the Drop
(Drop.html)
type returns the zone which provides these base pages.

Since a new zone always requires that an existing zone be identified, no new zones could be created unless there were some zones that already existed. Several such zones are predefined as part of the defobj library; see predefined zones
(globaldef.html#zones)
for a summary.

Create-time behavior

```
- (void) setReclaimPolicy: reclaimPolicy;
- (void) setStackedSubzones: (BOOL)stackedSubzones;
- (void) setPageSize: (int)pageSize;
USING
- getReclaimPolicy;
- (BOOL) getStackedSubzones;
- (int) getPageSize;
```

ReclaimPolicy

(.. Not implemented yet. Will define the various types of zone implementations, all the way to zones that support full automatic storage reclamation. ..)

StackedSubzones

(.. Not implemented yet. Specifies that no further allocations are made in the zone after a subzone is created, until that subzone itself is dropped. Supports an especially efficient of implementation. ..)

PageSize

PageSize specifies the size of pages within which a zone manages its internal allocation. Its default is typically a natural page size (perhaps 4K) for the local machine architecture. The default should be overridden only when tuning storage allocation for very specific situations. Allocations within a zone are not limited to the page size, since any requests that exceed the page size are simply passed up to the owner zone within which the zone was allocated.

Object allocation

```
- allocIVars: aClass;
- copyIVars: anObject;
- (void) freeIVars: anObject;
```

allocIVars: allocates the instance variable structure for a new object. The initial word of this structure is set to class id passed as its argument. The class also determines the size of the structure allocated. All remaining contents of this structure are initialized to binary zeroes.

copyIVars: creates copies an existing instance variable structure into a new allocation made within the local zone. The existing instance variable structure may be in any zone, but must contain a class pointer in its first word that correctly describes the size of the structure.

freeIVars: releases storage that was previously allocated to hold the instance variable structure of an object. The first word of the object must be a class pointer that correctly describes the size of the structure. Storage allocated by allocIVars: or copyIVars: may be freed only by freeIVars:, and freeIVars: may be used only to free storage allocated by one of these messages.

The object allocation functions are intended for use only within the implementation of an object class. Users of an object type should create a new object only using create: messages or other messages provided for this purpose.

Internal component objects

```
- allocIVarsComponent: aClass;
- copyIVarsComponent: anObject;
- (void) freeIVarsComponent: anObject;

- getComponentZone;
```

These messages allocate, copy, and free the storage that holds the instance variables for an object, but they allocate the object as an internal component of the zone that is not included in the zone population. They are used by classes that allocate additional objects as part of the implementation of another object, and that control the mapping of this storage separately from the zone level objects. getComponentZone returns a specially qualified version of the zone that automatically allocates all its objects with the internal component qualification, even if allocated with allocIVars: or copyIVars:. This qualified zone may be passed as an argument to a create: or createBegin: message so that it will create the new object as an internal component object.

Raw storage block allocation

```
- (void *) alloc: (size_t)size;
- (void) free: (void *)aBlock;

- (void *) allocBlock: (size_t)size;
- (void *) copyBlock: (void *)aBlock blockSize: (size_t)size;
```

```
- (void) freeBlock: (void *)aBlock blockSize: (size_t)size;
```

alloc: allocates a new storage block much like the malloc function of the C library. The storage is aligned according to the most restrictive requirements for any data type on the local machine architecture. The storage is not initialized to any known contents.

free: releases a block of storage previously allocated using alloc:. The size of the block is not required as an argument because alloc: has saved this size as necessary as part of the initial allocation. free: may be used only to free a block allocated by alloc:, and a block allocated by alloc: may be freed only by free:.

allocBlock: allocates a new storage block similar to alloc:, except that the size of the block allocated must be passed as an argument when freeing the block. copyBlock:blockSize: allocates a new storage with the same requirements on free as allocBlock:, but also fills the new storage with the contents of a block passed as the first argument, up to the size of the newly allocated block. freeBlock:blockSize: must be used to free any block previously allocated by allocBlock: or copyBlock:blockSize.

Subzone structure

```
-               getPopulation;
- getSubzones;
- (void) mergeWithOwner;

- (BOOL) containsAlloc: (void *)alloc;
- getSubzone: (void *)alloc;
```

getPopulation returns a collection all objects allocated in a zone using either allocIVars: or copyIVars: and not yet freed using freeIVars:. getObjects returns nil if the ObjectCollection option is false. The collection returned has the type OrderedSet
(../collections/OrderedSet.html)
as defined in the collections library
(../collections.html)
, with the ReadOnly option set true and the IndexSafety option set to SafeAlways. The members of this collection may change as objects are allocated and freed, but may not added or removed directly within the collection.

getSubzones returns a collection of other zones that all obtain their storage from the local zone. mergeWithOwner drops the local zone, but reassigns all the zone allocations to the zone from which all local storage was obtained. (..neither message currently implemented..)

containsAlloc: tests if a particular allocation was made by the local zone, or any zone which obtains storage from that zone. The pointer argument must point to an allocation previously made by any zone. The message returns true if the allocation was made by either the local zone or any of its subzones. getSubzone is similar, except that it returns the particular zone in which the pointer was allocated, or nil if the pointer was not allocated in the local zone or one of its subzones. (..neither message currently implemented..)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Name

Symbol – object defined as a distinct global id constant

Synopsis

```
//
// Symbol -- object defined as a distinct global id constant
//
@deftype Symbol <Create, GetName, CREATABLE>
CREATING
+ create: aZone setName: (const char *)name;
- (void) setName: (const char *)name;
@end
```

Description

A Symbol is an object created with a fixed name. It has no behavior except to get the name with which it was created. A Symbol is typically used to define unique id values which are assigned to global constant names. These names, capitalized according to the recommended convention for global object constants, are used by some libraries as flags or enumerated value codes in arguments or return values of messages.

Ordinarily, a symbol is created with its character string name matching the global id constant to which it is assigned. These global program constants can then provide a minimal level of self documentation as objects. Subtypes of Symbol can extend the base of a named, global id constant to establish further components of a global, constant definition.

Inherited behavior

A symbol is fully creatable using standard Create messages. A character string name must be supplied for any new symbol; there is no default. Symbol inherits the getName

(misc.html#GetName)
message, which returns the symbol name.

Create combination messages

```
CREATING
+ create: aZone setName: (const char *)name;
```

create:setName: is a combination message defined as a caller convenience. See combination messages
(combo.html)
for a summary of conventions on combination messages.

Create-time options

```
CREATING
- (void) setName: (const char *)name;
```

The Name option may be set only at create time. Its value is a null-terminated character string that remains fixed for the life of the object. The inherited getName message returns this name from a created instance.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Name

Warning, Error – report of some condition occurring during program execution

    (.. This section of documentation is only a start.)

Synopsis

```
//
// EventType -- report of some condition occurring during program execution
//
@deftype EventType <Symbol, RecordType>
- (void) raiseEvent;
- (void) raiseEvent: (void *)eventData, ...;
@end

//
// Warning -- a condition of possible concern to a program developer
//
@deftype Warning <EventType, CREATABLE>
- (void) setMessageString: (const char *)messageString;
- (const char *)getMessageString;
@end

//
// Error -- a condition which prevents further execution
//
@deftype Error <Warning, CREATABLE>
@end
```

Description

Warning and Event are object types (both subtypes of EventType) that may be used to report conditions detected during program execution. A warning is a condition of possible interest to a developer or user, but one that does not prevent further execution. An error is a condition which prevents further execution of the program.

    Warning and Event also inherit from Symbol, so that each type of warning or error
    may be given a unique global name. Any library may publish its own warning and event
    names as part of its documented library interface (see module definition conventions

(module.html)
for standard methods of declaring interface symbols in a library header file.)

These predefined warning and event types are typically used not directly by the messages defined above, but by the following macro defined in defobj.h:

```
#define raiseEvent( eventType, formatString, args... )
```

This macro raises an event of the type identified by the first argument. Default handling of the event is to print a formatted message including standard header and trailer lines, and then either to continue (in the case of a Warning) or to terminate the program in the case of Error). The standard header lines include the file, function, and line number where the macro that raised the event was executed.

If the formatString argument is nil, a default message string already established as part of the warning or event type is used to report the error. Otherwise, the formatString argument must be a null-terminated C character string that overrides the default string.

Either the default or replacement message string may contain printf-style formatted field definitions. For each formatted field definition, an argument value must be supplied in the args... section of the macro. (The support of the GNU C compiler for macros with variable argument lists is used by the macro.)

(.. Methods for defining new event types as part of a library implementation are not currently documented. Instead, just use the raiseEvent macro on one of the predefined Warning or Event Types if you want to raise an event using this standard mechanism. Eventually, this standard mechanism will be used to handle warnings and errors under various forms of logging exception handling support.)

Standard Error and Warning Types

The following standard error and warning types are defined by the defobj library. Thesy may also be used by other libraries for errors or warnings that fit within these standard predefined categories:

```
//
// standard errors
//
extern id <Error>
  SourceMessage,          // message in the source defines error
  NotImplemented,         // requested behavior not implemented by object
  SubclassMustImplement,  // requested behavior must be imple-
mented by subclass
  InvalidCombination,     // invalid combination of set messages for create
  InvalidOperation,       // invalid operation for current state of receiver
  InvalidArgument,        // argument value not valid
  CreateSubclassing,      // improper use of Create subclassing framework
  CreateUsage,            // incorrect sequence of Create protocol messages
```

```
  OutOfMemory,            // no more memory available for allocation
  InvalidAllocSize,       // no more memory available for allocation
  InternalError,          // unexpected condition encountered in program
  BlockedObjectAlloc,     // method from Object with invalid allocation
  BlockedObjectUsage;     // method inherited from Object superclass

//
// standard warnings
//
extern id <Warning>
  WarningMessage,         // message in the source defines warning
  LibraryUsage,           // invalid usage of library interface
  DefaultAssumed,         // non-silent use of default
  ObsoleteMessage,        // using feature which could be removed in future
  ObsoleteFeature,        // using feature which could be removed in future
  ResourceAvailability;   // resource from runtime environment not available
```

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1997-01-08

Miscellaneous Operations

Object Types

The following object types are documented below:

- Copy
- GetName
- GetOwner
- SetInitialValue

Summary of Miscellaneous Operations

An assortment of object types in the defobj library serve just to establish a single declaration of messages that many other object types adopt as part of their declared interface. The centralized definition of the messages in these types avoids their repetition throughout many other types. It also provides a single place where basic rules for these messages can be summarized for all other implementations to follow. This reference documentation provides these summaries along with intended purpose of each message.

> None of these types inherits from any other types; their whole purpose is just to package the definition of one or more messages. The type name is typically the same as a message declared by the type, only with the uppercase convention of a type name. These types may be freely mixed into the inheritance lists of other types to build whatever combinations of messages those other types need to define.

Copy Interface Description

Synopsis

```
//
// Copy -- copy all state defined as part of object
//
@deftype Copy
- copy: aZone;
```

```
@end
```

Description

The copy message creates a new object that has the same contents and resulting behavior as a starting object, except that an independent copy of the contents of the starting object is created so that further changes to one object do not affect the other. The zone argument specifies the source of storage for the new object. The message returns the id of the new object created.

> An object type that supplies the copy operation defines what it includes as the contents of an object copied. There is no global rule for what is considered "inside" a copied object vs. merely referenced by it. (There is no fixed notion of "shallow" vs. "deep" copy found in some object libraries.) After copying, the new object may still contain some references to other elements also referenced by the starting object, but in general the new object minimizes any dependencies shared with the starting object. Any object type supplying the copy message should also supply documentation on its rules for copied objects.

GetName Interface Description

Synopsis

```
//
// GetName -- get name which identifies object in its context of use
//
@deftype GetName
- (const char *) getName;
@end
```

Description

The getName message returns a null-terminated character string that identifies an object in some context of use. This message is commonly used for objects that are created once in some fixed context where they are also assigned a unique name. Constant objects defined as part of a program or library are examples. This message is intended only for returning a name associated with an object throughout its lifetime. It does not return any data that ever changes.

GetOwner Interface Description

Synopsis

```
//
// GetOwner - get object on which existence of object depends
//
@deftype GetOwner
- getOwner;
@end
```

Description

The getOwner message returns another object which is considered as the owner of an initial object. What is considered as an owner depends on its specific object type, but might be a larger object of which the local object is a part, or an object that has exclusive control over the local object. The principal constraint established by an ownership structure is that a given object can have only a single other object as its unambiguous owner.

> Ownership hierarchies arrange themselves in a strict, single-rooted tree. The top-level node of an ownership hierarchy typically returns nil as its owner. If an object is regarded merely as one part of another object defined as its owner, then copying or dropping the owner object should copy or drop the member object as well. Owner and member are neutral terms for a generic relationship sometimes called parent vs. child, but it is up to a particular object type to define specifically what it means by a getOwner relationship.

SetInitialValue Interface Description

Synopsis

```
//
// SetInitialValue - create using initial value from an existing object
//
@deftype SetInitialValue
CREATING
+ create: aZone setInitialValue: initialValue;
+ create: aZone setReadOnlyValue: readOnlyValue;

- (void) setInitialValue: initialValue;
- (void) setReadOnlyValue: readOnlyValue;
SETTING
- (void) setReadOnly: (BOOL)readOnly;
USING
- (BOOL) getReadOnly;
@end
```

Description

The SetInitialValue type defines a variety of messages relating to an initial or unmodifiable value established as part of an object. This message is typically provided when creation of a new object might be more easily accomplished by copying the value of an existing object rather than establishing a new value from scratch. As with the copy message, precisely what is considered the value of an existing object to copy is defined only by the particular object type that supplies these messages.

If an object has a value which can be established at create time, it is often useful (and can also enable significant optimization) to declare that no further modification will occur to this value during further use of the object. A restriction against modifying a value is referred to as a "read-only" restriction. This type supplies messages to declare a read-only restriction along with any initial value. For some object types, a read-only restriction can also be added or removed after an object has already been created.

The setInitialValue: message requires another object as its argument, from which the value of a newly created object is to be taken. Unlike a copy message, the object used as the source of the new value need not have the identical type as the new object to be created. A particular object type defines the types of initial value objects which it can accept, along with any special conversion or interpretation it might apply to such a value.

The ReadOnly flag specifies that the value of an object may not be modified further by other messages on an object. If such messages are attempted then an error is raised. If the ReadOnly flag is set at create time, the object is first created in read-only mode. Typically an initial value is always provided for an initial read-only object, since otherwise the created object would have only an empty or initial default value. The setReadOnlyValue: message is equivalent to a combination of a setInitialValue: message with an added ReadOnly restriction.

If a read-only restriction is established at create time, then for some object types this restriction may be required to remain for the entire lifetime of an object. For other object types, the read-only restriction may be added or removed any time after initial object creation. If a particular type does not support a change to read-only status, a request to do so raises an error.

Standard forms of create combination messages
(combo.html)
are provided to create an object in one message expression from the value of another object, with or without a read-only restriction.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Module Definition Conventions

Overview

Some of the basic Swarm libraries are implemented using object definition conventions established by the defobj library

(../defobj.html)
. These conventions include some minor special syntax to declare the public interfaces to Objective C objects. This document explains how to read and interpret the header files and interface definitions published by such libraries. It also explains the typical structure of documentation provided for libraries that follow these conventions.

These conventions are documented under the following subsections:

- library header file
- object type definitions
- GridTurtle example
- global object symbols
- interface design conventions
- documentation structure

Library Header File

For a library that adopts these conventions, a library is not merely a collection of source files that are compiled into a library archive under control of a make file. Instead, Objective C source files in the library are processed in a special way to publish their definitions not only as header files, but as generated objects available at runtime to make full use of the library. A library processed in this way is referred to as as a "module."

The defobj
(../defobj.html)
library documents the full details of special processing performed on a module. For simply using a library defined as a module, the key fact to keep in mind is that the entire public interface to a module is declared in the one header file having the same name as the module itself, plus a trailing .h suffix. Additionally, the header file of a library module normally documents only the public interface to a module, in a way that is completely separated from the implementation of the objects it specifies.

The separation of implementation means that a library module publishes its interface entirely without reference to any Objective C classes which implement its objects.

Even though classes are often thought of as separating the interface of an object from its implementation, this separation is far from complete. Not only do classes typically contain many internal methods not intended for external use, but they also define a particular storage format for an object defined as instance variables.

A library module instead publishes its interface as a set of public object types. These object types may also be supplemented by global object constants called symbols. Both these kinds of definitions normally appear only in the header file of a library. The remaining source files in a library normally contain the classes which implement the object types.

In a library module, the files which implement classes (including class header files), need not ever be referenced simply to make use of the implemented capabilities of the library object types. Documentation for the library is normally expressed entirely in terms of the types and symbols published in the library header file. If a feature does not appear in the library header file, it should not be considered part of a supported public interface.

Individual class header files are required to subclass from existing implementations, but interfaces for subclassing are an entirely separate issue from normal public use of an object library. Class inheritance can be a powerful implementation technique, but extension of an existing class framework is typically safe only if performed in explicitly permitted ways. If a library supports subclassing at all, it must carefully state which classes may be subclassed and in what ways. For a library module, this information is supplied outside the library header file. The library header file specifies only the interfaces by which objects are intended to be used, whether implemented by a local class or an external subclass.

Remaining sections of this document explain the declarations which appear in a library header file, and end with a suggested structure of documentation to be provided for a library module. The library modules of Swarm mostly follow this structure.

Object Type Definitions

The interfaces to objects defined by a library module are specified by object type definitions. An object type defines only a set of messages which may be sent to an object over various phases of its lifetime. An object type makes no commitment to the classes that might be used to implement the object.

Multiple classes may all implement the same messages belonging to a type. The independence of types and classes means that different classes can provide alternate implementations of the same object type. For example, a particular implementing class might be selected to optimize the implementation for a particular case.

Object types are similar to protocols defined by Objective C, and the declarations appearing in a library header file are a minor adaptation of Objective C protocol syntax. A key difference from protocols is that object types are published as real external objects that may be used at runtime to create instances of a type. A futher difference is that the object types of the defobj library are divided into separate interfaces that define distinct phases of an object's life cycle.

GridTurtle example

An example will help illustrate the features of type definitions supported by the defobj library. Throughout documentation of the basic Swarm libraries, a series of running examples will be based on a simple type of object belonging to sample Swarm simulations. This type of object is an agent that move around on a two-dimensional grid, always moving in a current direction that it maintains internally. This agent is like a "turtle" of the original Logo system, except that its position is constrained to discrete integer values of its X-Y coordinates, and its direction is always one of the four orthogonal directions north, east, south, or west.

Following is a complete library header file for a module which defines such an object, called GridTurtle:

```
/*
Name:         GridTurtle.h
Description:  object type for Swarm example programs
Library:      grid
*/

#import <defobj.h>
@deftype GridTurtle <Create, Drop, CREATABLE>CREATING
- (void) setXLocMax: (int)xLocMax;
- (void) setYLocMax: (int)yLocMax;
SETTING
- (void) setXLoc: (int)xLoc;
- (void) setYLoc: (int)yLoc;
- (void) setXLoc: xLoc setYLoc: yLoc;

- (void) setDirection: direction;
USING
- (int) getXLoc;
- (int) getYLoc;

- getDirection;

- (void) move: (int)distance;
- (void) turn: (int)angle;  // angle measured in units of pi/2 (90 deg.)
- (void) print;
@end

id <Symbol> North, East, South, West;
```

```
#import "grid.xt"
```

An object type is defined by an @deftype declaration. The syntax of such declaration is identical to that of an Objective C @protocol definition, except for the entirely uppercase keywords (CREATABLE, CREATING, SETTING, USING) appearing in the GridTurtle example above. All these modifications of Objective C syntax are accomplished by simple preprocessor macros; no extensions to the language compiler are involved.

When this library header file is processed (by a special rule in a make file), an external object id with the name GridTurtle is automatically published. The name of a defined type becomes an ordinary object that accepts specific messages defined by the defobj library. The defobj library
(../defobj.html)
explains the details of such messages; the only purpose here is to explain the basic sections of a deftype declaration.

deftype declarations follow the syntax as Objective C protocols for inheriting messages from each other: a list of names enclosed in angle brackets (e.g., <Create, Drop, ...> above) gives the names of other declared types containing messages to be supported by the new type as well. (These types referenced here are defined by the imported file <defobj.h>.) Like protocols, full multiple inheritance of types is supported. The same messages may be inherited any number of times through any path with no different effect than if inherited or declared just once, so long as no conflicts occur in any of their argument or return types.

The CREATABLE tag appearing in the inherited type list above is a special type which defines no messages of its own, but merely marks the type as one which supports direct creation of instances of the type. Without this tag, the only role of a type is to define messages for inheritance by other types. With this tag, the global type object has a particular implementation that supports object creation using standard messages defined in defobj.

The declared messages of the type may be separated into sections marked by the special uppercase tags such as CREATING, SETTING, and USING above. (Currently, these are the only such tags which may occur.) These sections each define messages belonging to a particular defined "interface" of the object type, which are further combined into distinct "phases" of an object lifecycle supported by defobj messages. Further explanation of the interfaces and phases defined by this example are provided in the Usage Guide
(guide.html)
of the defobj library.

The following line at the end of the <grid.h> header file above is required as part of the standard conventions for coding a module header file: This line adds no information to the header file, but is required as part of the system by which definitions are automatically generated.

```
#import "grid.xt"
```

Global Object Symbols

The <grid.h> header file above also contains the declaration:

```
id <Symbol>  North, East, South, West;
```

Lines that declare global id variables of type Symbol, EventType, Warning, or Error (using the angle bracket syntax of id variables conforming to a protocol) are processed somewhat like deftype declarations in that they also produce global id variables initialized to support particular messages defined by defobj. These global variables, however, are not used to define or implement other message interfaces, but only to define certain fixed capabilities referenced through their global object names.

If declared as a Symbol, as in the case here, the generated objects have no particular behavior of their own (other than the character string of their name), but only serve to define unique global id constants which may be used as distinct named values in messages. In this example, the current direction of a GridTurtle object is represented by one of the symbol names North, East, South, or West. These values are like the enum constants of the C language, except that they are defined as full Objective C objects, and may be used with further restrictions. An EventType, or a Warning or Error, defines a further subtype of a Symbol constant with further specialized messages documented in defobj.

Interface Design Conventions

A variety of rules on naming and declaration of object types, symbols, and messages are followed by many of the Swarm libraries. These rules help establish a basic consistency on the library interfaces. Following is a list of such conventions that apply to a public library interface, some but not all of which are derived from standard Smalltalk or Objective C coding practice:

- Names of global object constants are capitalized. In the public interface, such names include types and symbols.
- Recapitalization separates words of a compound name (e.g., GridTurtle). Underscores are generally not used.
- Message names start with a lower-case character, and are named using verbs. Nouns that represent gettable or settable components of object state (e.g., Direction of a GridTurtle), are prefixed by get or set to indicate the action being performed.
- The Smalltalk convention that a message return the receiver of a message if there is no other specific return value is generally *not* followed. If there is no specific return value needed from a message the return type is declared (void).

Documentation Structure

The following standard sections of documentation are suggested for libraries that follow the strict interface vs. implementation separation of library modules, each with purpose and typical contents as given:

- Interface Reference
  Complete, concise summary of all features of a library. Fills the role a Unix "man page" in providing comprehensive definition of library services. Little or no concern to provide path of tutorial introduction. Does not contain extended examples, at most only fragments of examples which serve needs of specification.
- Usage Guide
  Tutorial introduction to a library. Focuses on the most common uses in an order reflecting the needs of a first-time user. Most explanation by means of progressively elaborated examples. Serves as a guided tour of major library capabilities. No attempt at reference-style completeness.
- Advanced Usage Notes
  Continues overview of all significant capabilities of a library, including those which might be needed by advanced users customizing or extending built-in capability. Provides examples of specialized uses extending beyond normal, basic usage, but still using built-in features of the library framework. Helps to simplify the Usage Guide by providing a location for overflow of more advanced features.
- Subclassing Notes
  Documents the rules for writing new classes which subclass from classes that implement the library. Because a library might use complex combinations of classes to implement the range of behaviors defined by its types, the classes in a library are not automatically usable as superclasses of user-defined classes. Each library documents which classes are available for use as superclasses, and the specific rules that must be followed when subclassing from these classes.
- Interface Design Notes
  Explains why choices were made the way they were in the design of the public interface to a library, including other alternatives considered. Serves as a "Rationale" document for the documented interface. May include historical background deliberately left out of the preceding sections. May also include references to other work.
- Implementation Notes
  Explains the structure of classes by which the types of a library are implemented. Summarizes the status of implementation if still incomplete and lists items of possible future work. Provides overview and high-level structure of the implementation in whatever ways would best guide a reader of implementation source code. May also discuss tradeoffs considered along with references to related or supporting work.

Marcus G. Daniels <mgd@santafe.edu>

(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Defobj Library: Create Combination Messages

The create phase of an object type may contain messages that are defined simply by identifying them as a combination of other defined messages. Such messages are referred to as "create combination messages." Typically, such messages have the following form:

```
-               create: aZone setOption: optionValue;
```

For this message to be a valid create combination message, the object type must already have the create: and setOption: messages defined (either locally or by inheritance). The meaning of this message is then identical to the following sequence (where SomeType is the name of a type implementing the combination message):

```
-   anObject = [SomeType createBegin: aZone];
-   [anObject setOption: optionValue];
-   anObject = [anObject createEnd];
```

Create combination messages are provided as a convenience for commonly used options. Some combination messages also have their own custom implementing methods so that creation is faster when using the combination message. Any such implementation, however, is required to implement the parts of the combination message so that its meaning is identical to the results that would be obtained if the separate parts were requested in separate message.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Defobj Library: Global Portability Assumptions

In a pure object-oriented system such as Smalltalk, the type of an object reference (an object id in Objective C) is the only fundamental data type used within the system. In a hybrid object system such as Objective C, both object id's and other data types are used together. The Swarm libraries make some basic assumptions about the compatibility of native C data types with the id pointer type of Objective C. These assumptions allow some of the more generic abstractions of the libraries (such as collections and actions) to provide one fundamental abstraction that can still be used under an identical implementation to support most of the other C data types.

The basic assumption is that all data types except double may be cast to and from a stored id type without losing the original value. This assumption is valid on all modern 32-bit or greater machines, including all machines on which Swarm is currently running or planning to run.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Defobj Library: Global Definitions

Contents

The following global definitions are documented below:

M() macro

Synopsis

```
//
// M() -- macro to abbreviate @selector()
//
#define M( messageName )  @selector( messageName )
```

The M() macro is currently nothing more than an abbreviation for the Objective C syntax used to obtain a selector for a particular message name. It is provided as a convenience for use in messages that combine a selector with other arguments.

---

func˙t type definition

Synopsis

```
//
// func_t -- function pointer type
//
typedef void (*func_t)( void );
```

func_t provides a type definition for a generic function pointer type (taking no arguments and returning no value). This type name is used in declaring messages that accept or return function pointers of various alternative types.

---

initModule() function

Synopsis

```
//
// initModule() -- module initialization macro
//
#define initModule( module )
```

The initModule function must be called to initialize all the definitions of a library module before any attempt is made to use the library. Initializing a particular library module will also initialize all other library modules on which the library depends. The macro has no effect if the library has already been initialized, so that multiple libraries that all depend on another library can all request initialization of the other library with no ill effect. The argument of the macro is the name of module to be initialized. For example, if just the collections library were being initialized, the macro would be coded:

Global Zones

Synopsis

```
#define globalZone
#define scratchZone
```

The identifiers globalZone and scratchZone refer to two standard zones created by initialization of the defobj library. All other created zones must obtain their storage from one or the other of these two initial zones. (These initial zones are defined as macros only to follow conventions on global external names.)

> globalZone may be used for all storage which must be allocated for the duration of the program, or until explicitly released. scratchZone should be used for storage which is allocated only for use during execution of a single function. It is typically used for objects which are strictly temporary as part of a local computation. (.. Currently, a scratch zone has the identical implementation as the global zone, but eventually a scratch zone may be optimized for temporary allocations, and such allocations may also be freed automatically at various points when functions are guaranteed to be complete.)

Output Files

Synopsis

```
extern FILE *_obj_xerror;     // output file for error messages
extern FILE *_obj_xdebug;     // output file for debugging messages
```

The file pointers _obj_xerror and _obj_xerror define standard output files used for error messages and debugging messages, respectively. If not set otherwise by the time of defobj module initialization, they both default to the standard error output defined by the C library.

Debug Support

Synopsis

```
extern BOOL _obj_debug;       // if true then perform all debug error check-
ing

extern void xsetname (id anObject, const char *name);  // debug set dis-
play name

extern void xprint (id anObject);              // debug object print
extern void xprintid (id anObject);            // debug object id print

extern void xfprint (id anObject);             // debug foreach object print
extern void xfprintid (id anObject);           // debug foreach id print

extern void xexec (id anObject, const char *name);  // debug method exec
extern void xfexec (id anObject, const char *name); // debug fore-
ach method exec
```

The _obj_debug variable enables debug-level error checking in libraries that support optional levels of checking. Its default value is true. The default may be overridden either before initialization of the defobj module, or by means of a compiler macro setting (-D_obj_debug=0). If set to zero by a compiler macro setting, optimized compilation of code that refers to it will see only the constant zero and so avoid even generating code to test the variable.

The xsetname function sets the display name for an object using its setDisplayName: message. This string may be used to identify the object in a user-meaningful way; if defined it will appear in the identification string generated by a standard describe: message.

The xprint function prints a standard object description to the standard debugging output file. It generates the string to be printed using the describe: message defined by the DefinedObject supertype, but it is defined as a function so that it may be called directly from a debugger. xprintid is similar, but it prints only a one-line description using the built-in default.

The xfprint function is similar to the xprint function, except that it prints a debug description for each member of a collection. The argument of xfprint must support enumeration of collection members using the messages of the Collection (../collections/Collection.html) type. xfprintid is similar, but only one line per member is printed.

The xexec function executes a message on an object passed as an argument. The name of message to be executed (the full name including arguments as would be used in naming a selector) must be given as the second argument. This function is equivalent to the "perform:" message of Objective C, but is defined as a function so that it may be called directly from a debugger.

The xfexec function is similar to the xexec function, except that it executes a message on each member of a collection. The first argument of xfprint must support enumeration of collection members using the messages of the Collection (../collections/Collection.html) type.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Defobj Library: Subclassing Reference

The defobj library currently includes the CreateDrop and Object_s superclasses used by other libraries. For user objects in Swarm simulations the SwarmObject superclass in the objectbase (../objectbase.html) library packages all required behavior of these superclasses into a single, simpler superclass.

The defobj library also provides support for custom-generated classes, such as those which implement separate phases of the standard create protocol. (See generated phase classes (adv.html#phase-classes) in the Advanced Usage Guide (adv.html) .) Subclassing from custom-generated classes is still not officially supported, because the framework for custom class generation is still being finalized. Custom-generated classes are currently used only in the defobj, collections, and activity libraries; these libraries document the specific classes that provide subclassing support.

In general, classes that implement a library are not automatically available for use as superclasses. Any library must document those specific superclasses that are valid for user classes to subclass from, along all the rules that a user subclass must follow when implementing new behavior.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Defobj Library: Interface Design Notes

(This section of documentation is not yet available.)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Defobj Library: Implementation Notes

(This section of documentation is not yet available.)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08