Intro to Objective C on the NeXT Machine

by Gerrit Huizenga

gerrit@mentor.cc.purdue.edu

Prerequisites:

Basic understanding of the C Programming Language

Helpful Extras:

Some experience with the NeXT Machine
Some familiarity with SmallTalk
Experience with Ansi C or the Gnu C compiler
Familiarity with Objective Oriented Programming

Outside Reading

Object Oriented Programming: An Evolutionary Approach
by Brad Cox
Objective-C Compiler Version 4.0 Reference Manual
The Stepstone Corporation
Display PostScript System
Adobe Systems Incorporated
NeXT Technical Documentation
NeXT Incorporated

Goal

This course will provide an introduction to Object Oriented Programming using Objective C on the NeXT machine. If you are interested in doing any software development on the NeXT machine, this course will help get you started. I will discuss the terminology and concepts used in Objective Oriented Programming as well as describe the syntax used in Objective C. I will show how to read an Objective C spec sheet and describe some of standard Classes of Objects provided by NeXT and how to use some of them.

Future Courses

I plan to teach the following courses in the future:

Introduction to the NeXT Interface Builder
Intermediate Objective C on the NeXT machine
Advanced Interface Builder

I may also teach a class a more advanced class on Objective C and the NeXT Object Classes.

Object Oriented Programming

What is it?

Grouping of functionality and data

Separation of interface and implementation

Type-dependent operations

Code Sharing

Hierarchical partitioning of functionality

A different way of solving problems

How does it work?

Syntactical support for abstract data types

Message-sends instead of function calls

Compiler support for inheritance

Programmer thinks differently

Abstract Data Types

Computer programs manipulate data.
What they do to that data is important.
How they do it or what form the data takes is not.

So...create new data types and and declare exactly what the program can do to manipulate a variable of that type.

This provides a form of data integrity...

And provides flexibility for future modification of the data’s form (i.e. for efficiency, bug-fixing, or enhancement).

Terminology:

In object-oriented languages, these data types are called **Classes**.

A variable of a given *class* is an **instance** of that class.

The word **object** is often used to refer to an *instance* of a *class* (e.g., a Button *object* is an *instance* of the Button *class*).

The operations that a given class implements are called **methods** (not functions).

Messaging

We can ask an object to perform a method without knowing what its class is.

Instead of calling a function to perform some operation on an object, we send that object a message asking it to perform that operation on itself.

Depending on the class of the object, different code will be executed by the computer.

This is useful since a given message may be meaningful to different classes of objects (e.g. "size" is meaningful both to a stack and a queue even though each may calculate its size in a very different way.)

Terminology:

The messages sent to an *object* are called **messages**.

Instead of calling a function, we send a *message* to an *object* to get it to **perform** one of its *methods*.

Example of Object Oriented design

The Unix File System is Object Oriented. Most of you are familiar with the following Unix system calls:

open()
close()
read()
write()
lseek()
ioctl()

The open() call takes a file name as one of its arguments and returns a handle to a file. The open call may perform some device specific initialization, such as supplying DTR to a terminal line or locating a disk drive which contains the named file. In the kernel, there is a switch statement which decides which device specific routines be called based on the filename that you have provided. Each of the other routines listed above have similar code to decide how to do buffering and actually locate the data that you are working with, either by sending the appropriate commands to the disk drive or tape drive, or waiting for serial input from a terminal line, or sending data to a printer.

As the person using the routines though, you generally don't have to know which set of operations will be performed by the kernel, but that the kernel will do whatever is necessary to fulfill your request.

In this example, the *class* might be FileManipulation, the *methods* would be open(), close(), etc, and an *object* would be the file descriptor that the open() routine returns.

Inheritance

Related classes can share common code.

> A new class can be created which inherits all the functionality and data of some other class (i.e. instances of the new class understand all the messages that an instance of the class it inherits from understands).
>
> The new class can choose to respond to any given message in a different way than the class it inherits from does and can also define new methods.
>
> The new class can modify the data used to implement the class it inherits from and can add to that data.

Terminology

A *class* which inherits from another *class* is a **subclass** of the *class* it inherits from. A *class* is a *subclass* of its **superclass**.

When a *subclass* responds to a message in a different way than its *superclass* does, the *subclass* is said to have **overridden** its *superclass's method*.

Subclass is often used as a verb, "I *subclassed* the Vehicle *class* to create the Car *class*."

Programming

The thought processes of the programmer are as much a part of object-oriented programming as any of the above.

Programs must be thought of as a collection of cooperating pieces of data (objects) rather than a thread of control.

Program design is data-oriented rather than process-oriented.

User-interface programming is fundamentally object-oriented.

Terminology:

Object-oriented programmers use phrases like:

"When the user presses this button, it sends a *message* to this *object* which calculates something and then sends a *message* to this other *object* which updates this and..."

instead of

"We wait for the user to press a button and then we decide which one it was, and, based on that, we decide what to do, then we wait for the user to do something else..."

Objective C

In Objective C, we describe the interface to a *class* using using the **@interface** declaration:

```
@interface Stack : Object
```

followed by the variables used to implement the *object* (these are called the *class's* **instance variables,** and each *instance* of this *class* has its own copy of these variables):

```
{
	StackLink *top;
	unsigned int size;
}
```

Then we list the methods that this class implements (i.e. the messages that objects of this class understand):

```
- free;

- push: (int) anInt;

- (int) pop;

- (unsigned int) size;
```

and finish up with the **@end** declaration:

```
@end
```

This would normally be stored in a header file called "Stack.h". The definitions for the *superclass* of the *class* you are *subclassing* are imported at the beginning of the file. An import uses the **#import** directive which is similar to the **#include** directive, but ensures that the file is only include by your file once. With all of this in place, the file would look like this:

```
#import <objc/Object.h>

@interface Stack : Object

{
        StackLink *top;
        unsigned int size;
}

- free;
- push: (int) anInt;
- (int) pop;
- (unsigned int) size;

@end
```

Note that in the **@interface** declaration, we specify the *class* that this *class* inherits from (its *superclass*). The *superclass* of the Stack *class* is the Object *class* (all *classes* are at least a *subclass* of the Object *class*).

Method names always contain a colon (":") before any of the arguments that are passed along in the *message* (e.g. **push:**).

Any number of parameters may be sent in a *message*, each separated by a keyword ending in colon. For example, if we wanted to be able to push two integers on the stack with one *message*, we could define the *method*:

- push: (int) first and: (int) second;

The name of this method is "**push:and:**".

Parameters passed in the *message* are declared using the C "type cast" notation. Any type that is valid as a C function parameter is valid as a *method* parameter. The return type of a *method* is also specified using the "type cast" notation. If no return type is specified, then the *method* is assumed to return a pointer to an *object*.

We define the implementation of the class (in a sepearate file from the interface) using the **@implementation** directive:

@implementation Stack

Objective C is just like normal ANSI-C (as implemented by the GNU C compiler), except that it provides the ability to define *classes*, create *instances* of *objects*, and send *messages* to *objects*. Two new fundamental types are added to the language:

id pointer to an *object*
SEL a *message* (we sometimes call *messages* **selectors**, thus the abbreviation).

Both variables of type **id** and type **SEL** are valid parameters that can be sent in a *messages* or passed to a C function.

Messages are sent using a Smalltalk-like syntax:

```
id s;
int i;

s = [Stack new];
[s push:34];
i = [s pop];
```

In the implementation of a *class*, *methods* can manipulate the *class's instance variables* (e.g. the linked list pointed to by top in the Stack *class*), can generally do anything normally allowed in C, and can send *messages* to *objects*.

What *objects* can an *object* send *messages* to?

itself (this is very common)

itself, but use its *superclass's* implementation!

objects pointed to by one of its *instance variables* (i.e. *instance variables* of type **id**).

objects passed to it as a parameter in a *message* sent to us by some other *object*

factory objects (factory objects are sometimes knowns as *class objects)*

How does an *object* send a *message* to itself?

Example: **[self push:34.0];**

self is a special variable which is a pointer to the *object* which received the *message* which invoked the currently executing *method*(!). In other words, it is the **receiver** of the message.

Example: Remember **push:and:** ? Here is probably how that *method* would be implemented:

```
- push: (int) first and: (int) second
{
      [self push: first];
      [self push: second];
      return self;
}
```

Notice that the **push:and:** *method* returns **self**. Remember that if a *method* does not specify the type of its return value, the default is to return a pointer to an *object* (i.e. something of type **id**). This means that, barring any other sensible thing to return, ***methods* should always return self!**

How can an *object* send a *message* to itself but use its *superclass's* implementation? And why would someone ever want to do that?

Why? ...because in the implementation of a *method* which is overridden (i.e. the *superclass* implements it, and the *subclass* implements it differently, a *class* may want (and often does want) to perform its *superclass's* implementation as part of its own implementation.

How? ...any *message* an *object* sends to the pseudo-variable **super** will cause its *superclass's* implementation of the *method* to be performed.

An *object* can send a *message* to **super** in any *method*. **super** implicitly means "**self**, but use superclass's implementation"

There is one and only one **factory object** per *class*. There is a global **id** variable which points to it, and that variable's name is the same as the *class's* name. You send *messages* to it to create new *instances* of that *class* or to query *class*-specific (as opposed to *instance*-specific) information.

Example:

id s;

s = [Stack new];

The file which contains the implementation of the *class* ends with the **@end** directive (just like the interface file does).

Example:

```
@implementation Stack

- free
{
	StackLink *next;

	while (top != (StackLink*) 0) {
		next = top->next;
		free( (char *) top);
		top = next;
	}

return [super free];
}

< other methods >

@end
```

File: Stack.h

```
#import <objc/Object.h>

typedef struct StackLink {
	int data;
	struct StackLink *next;
} StackLink;

@interface Stack : Object
{
	StackLink *top;
	unsigned int size;
}

- free;
- push: (int) value;
- (int) pop;
- (unsigned int) size;

@end
```

```
#import "Stack.h"

@implementation Stack

#define NULL_LINK (StackLink *) 0

- free
{
	StackLink *next;

	while (top != NULL_LINK) {
		next = top->next;
		free ((char *) top);
		top = next;
	}
	return [super free];
}

- push: (int) value
{
	StackLink *newLink;

	newLink = (StackLink *)malloc(sizeof StackLink);
	if (newLink == 0) {
		fprintf(stderr, “Out of memory\n”);
		return nil;
	}
	newLink->data = value;
	newLink->next = top;
	top = newLink;
	size++;

	return self;
}
```

```
- (int) pop
{
	int value;
	StackLink *topLink;

	if (0 != size) {
		topLink = top;
		top = top->next;
		value = topLink->data;
		free (topLink);
		size--;
	} else {
		value = 0;
	}
	return value;
}

- (unsigned int) size
{
	return size;
}

@end
```

Factory Objects

Objective-C automatically creates a **factory object** for each *class* used in an application.

> Exactly 1 instance of a *factory object* per *class* exists at runtime

> The name of the *factory object* is the name of the *class*

The primary purpose of a *factory object* is to provide a mechanism to create instances of the *class*:

```
id myStack;

myStack = [Stack new];
```

Factory objects respond to **factory methods**.

Factory methods are indicated by a "**+**" preceding the name when declared and defined.

```
+ new
{
	self = [super new];
	top = (StackLink *) 0;
	return self;
}
```

Factory objects are not *instances* of the *class* and therefore do not have access to the *instance variables* associated with an *instance* of the *class*, so *factory methods* typically redefine **self** before accessing *instance variables*.

File: CalculatorBrain.h

```
#import <appkit/appkit.h>
#import <objc/Object.h>

@interface CalculatorBrain : Object
{
	id stack;
	int accumulator;
	BOOL accumulatorEntered;
	id display;
}

/* Private */

- handleDigit: (int) digit;

/* Public */

- add:sender;
- digit:sender;
- divide:sender;
- enter:sender;
- multiply:sender;
- subtract:sender;
- zero:sender;

@end
```

File CalculatorBrain.m

```
#import "CalculatorBrain.h"
#import "Stack.h"

@implementation CalculatorBrain

+ new
{
        self = [super new];
        stack = [Stack new];
        accumulatorEntered = YES;
        return self;
}

- setDisplay:anObject
{
        display = anObject;
        return self;
}

- handleDigit: (int) digit
{
        if (accumulatorEntered == YES) {
                accumulator = digit;
                accumulatorEntered = NO;
        } else {
                accumulator = accumulator * 10 + digit;
        }

        [display setIntValue:accumulator];
        return self;
}
```

```
- add: sender
{
	if (accumulatorEntered == NO) {		/* Hit plus sign, pretend Enter hit */
		[self enter:self];
	}
	accumulator = [stack pop] + [stack pop];
	[self enter:self];
	[display setIntValue:accumulator];
	return self;
}

- digit:sender
{
	int operand;

	return [self handleDigit:[sender selectedTag]];
}

- divide:sender
{
	int operand;

	if (accumulatorEntered == NO) {
		[self enter:self];
	}
	operand = [stack pop];
	if (operand != 0) {
		accumulator = [stack pop] / operand;
	} else {
		[stack pop];
		accumulator  = MAXINT;
	}
	[self enter:self];
	[display setIntValue:accumulator];
	return self;
}
```

```
- enter:sender
{
	[stack push:accumulator];
	accumulatorEntered = YES;
	return self;
}

- multiply:sender
{
	if (accumulatorEntered == NO) {
		[self enter:self];
	}
	accumulator = [stack pop] * [stack pop];
	[self enter:self];
	[display setIntValue:accumulator];
	return self;
}

- subtract:sender
{
	int tmp;		/* needed because order of eval not defined for "+" below */
	if (accumulatorEntered == NO) {
		[self enter:self];
	}
	tmp = - [stack pop];
	accumulator = tmp + [stack pop];
	[self enter:self];
	[display setIntValue:accumulator];
	return self;
}

- zero:sender
{
	return [self handleDigit:0];
}
@end
```

Reading A Class Specification

There are three software kits currently available on the NeXT Machine. These three kits are the Application Kit, the Sound Kit and the Music Kit. There are also some classes that come with the Objective C compiler. These kits are documented in detail in Chapter 21 of the *NeXT Technical Documentation* which is available on any NeXT machine. The documentation for the classes provided by these kits are prepared in a standard form, called a Class Specification.

The class specifications are grouped according to kit; within each kit they are arranged in alphabetical order by class. Each class specification details the instance variables the class declares, the methods it defines, and any special constants and defined types it uses. There's also a general description of the class and its place in the inheritance hierarchy. However, you won't find a discussion of any kit's design or an explanation of how to go about using the kit to program an application. You may occasionally encounter terms that assume some prior knowledge about the kits, Mach, the Display PostScript system, or object-oriented programming. This information is briefly described in Chapter 1 of the *NeXT Technical Documentation*, and described more thoroughly in the volume of the *NeXT Technical Documentation* entitled **Concepts.**

A more complete guide to reading the specifications is available at the beginning of Chapter 21 of the *NeXT Technical Documentation*.

Organization

Information about a class is presented under the following headings:

INHERITS FROM

The first line of a class specification lists the classes that the class being described inherits from. For example the **Menu** class in the application kit has the following inheritance hierarchy:

Panel : Window: Responder : Object

The first class listed (Panel, in this example) is the class's superclass. The last class listed is always Object, the root of all Objective-C inheritance hierarchies. The classes between show the chain of inheritance from Object to the superclass.

REQUIRES HEADER FILE

Each kit is identified by a master header file that includes almost all the other header files you need to program with the kit:

Kit	Header File
Application Kit	/usr/include/appkit/appkit.h
Music Kit	/usr/include/musickit/musickit.h
Sound Kit	/usr/include/soundkit/soundkit.h

Each of these header files include the master header file for classes which come with the compiler. If you aren't using one of these header files, you should include <objc/objc.h>.

Occasionally, a class will also list a UNIX header file not included by the master header file.

Because the kits are written in Objective-C, they make use of constants and types defined in the principal header file for Objective-C, objc.h. Only a handful of these constants and types are used by the kits, but they're used pervasively. For convenience, they're listed below.

Defined Types:

id An object.

STR A C string. **STR** is a shorthand for **(char *)**. It's used only for an array of characters that's terminated by the null character.

SEL A method selector. **SEL** is another shorthand for **(char *)**, where the character string can be thought of as a method name. However, **SEL** is used only as a unique code for a method name, rather than as a pointer to an actual occurrence of the name in memory. Values should be assigned to **SEL** variables only with the **@selector** operator:

```
SEL   aMethod;
aMethod = @selector(moveTo::);
```

This allows selectors to be tested by matching the value of a **SEL** code, rather than by comparing all the characters in a string.

BOOL	A char that holds one of two values: **YES** (true) or **NO** (false).

Constants:

nil	A null object **id**, **(id)0**.
YES	Boolean true, **(BOOL)1**.
NO	Boolean false, **(BOOL)0**.

DEFINED IN

The name of the kit is given after this heading, along with the version number of the software release that's documented. The table below lists the libraries where the kits are defined:

Kit	Library
Application Kit	libNeXT_s.a
Sound Kit	libNeXT_s.a
Music Kit	libmusickit.a

The common classes that come with the Objective-C compiler are defined in libsys_s.a. Since these classes aren't part of a kit, they're introduced by a slightly different heading, "DEFINED AS," and are identified as common classes.

The "_s" suffix on libNeXT_s.a and libsys_s.a designates them as shared libraries. All three libraries reside in the /usr/lib directory.

CLASS DESCRIPTION

This section gives a general description of the class. It tells how the class fits into the general design of its kit and how your application can make use of it.

INSTANCE VARIABLES

The instance variables that are incorporated into each object belonging to the class, including instance variables inherited from other classes, are listed next. The first instance variable in all the lists is one inherited from the Object class, isa. isa identifies the class that an object belongs to for the Objective-C run-time system; it should never be altered or read directly.

After all the instance variables are listed, those declared in the class being described are explained.

However, instance variables that are for the internal use of the class are neither listed nor explained. These instance variables all begin with an underscore (_) to prevent collisions with names that you might choose for instance variables in a subclass you define.

METHOD TYPES

Methods are next listed by name and grouped by type—for example, methods used to draw are listed separately from methods used to handle events. This directory includes all the principal methods defined in the class and some that are defined in classes it inherits from. Inherited methods are followed by the name of the class where they're defined; they're included in the directory to let you know which inherited methods you might commonly use with instances of the class and where to look for a description of those methods.

CLASS METHODS
INSTANCE METHODS

A detailed description of each method defined in the class follows the classification by type. Methods that are used by class objects (factory objects) are presented first; methods that are used by instances (the objects produced by the class, instance methods) are presented next. The descriptions within each group are ordered alphabetically by method name.

Each description begins with the syntax of the method's arguments and return values, continues with an explanation of the method, and ends, where appropriate, with a list of other related methods. Where a related method is defined in another class, it's followed by the name of the other class within parentheses.

All methods have reliable return values. Unless the method description mentions otherwise, every method returns self. This allows you to chain messages together:

```
[[[receiver message1] message2] message3];
```

Internal methods used to implement the class aren't listed. Since you shouldn't override any of these methods, or use them in a message, they're excluded from both the method directory and the method descriptions. However, you may encounter them when looking at the call stack of your program from within the debugger. A private method is easily recognizable by the underscore (_) that begins its name.

There are a couple of other sections in some of the class specifications. However, they are beyond the scope of this class.

List

INHERITS FROM	Object
REQUIRES HEADER FILES	<objc/List.h>
DEFINED AS	A common class

CLASS DESCRIPTION

List allows easy manipulations of collections of objects. Collections can be manipulated as fixed or variable size lists, sets, or ordered collections.

INSTANCE VARIABLES

Inherited from Object	struct _SHARED	*isa;
Declared in List	id	*dataPtr;
	unsigned	numElements;
	unsigned	maxElements;
	unsigned	growAmount;

dataPtr	data of the List object
numElements	Actual number of elements
maxElements	Total allocated elements
growAmount	Number of elements to grow or shrink the array by

METHOD TYPES

Creating and freeing a List object	- free - freeObjects + new + newCount:
Manipulating objects by index	- addObject: - count - insertObject:at: - lastObject - objectAt: - removeLastObject - removeObjectAt: - replaceObject:with:
Manipulating objects by id	- addObjectIfAbsent:

	- indexOf: - removeObject: - replaceObjectAt:with:
Emptying the List	- empty
Sending messages to the objects	- makeObjectsPerform: - makeObjectsPerform:with:
Managing the storage capacity	- capacity - capacity: - setGrowAmount:
Archiving	- read: - write:

FACTORY METHODS

new

+ (List *)**new**

Returns a new List.

newCount:

+ (List *)**newCount:**(unsigned)*numSlots*

Returns a new List object large enough to hold *numSlots* objects.

INSTANCE METHODS

addObject:

– addObject:*anObject*

Puts *anObject* at the end of the List.

addObjectIfAbsent:

– addObjectIfAbsent:*anObject*

Searches the List for *anObject* and, if it isn't already in the List, adds it at the end. If *anObject* is already in the list, this method does nothing.

capacity

– (unsigned)**capacity**

Returns the maximum number of objects that can be stored in the List without increasing its current capacity.

capacity:

– capacity:(unsigned)*numSlots*

Sets the storage capacity of the List to *numSlots* objects. It's best not to use this method.

count

– (unsigned)**count**

Returns the number of objects currently in the List.

empty

– empty

Empties the List of all its objects.

free

– free

Deallocates the List object, but not the objects that are in the List.

freeObjects

– freeObjects

Deallocates storage for the List object and for every object in the List. Does not free argument itself. Since free methods are performed, no side effect should be produced on the List object itself during these performs.

indexOf:

– (unsigned)**indexOf:***anObject*

Returns the index of the first occurrence of *anObject* in the List, or –1 if *anObject* isn't in the List.

insertObject:at:

– insertObject:*anObject*
at:(unsigned)*index*

Puts *anObject* into the List at *index*, moving objects down one slot to make room, and returns **self**. However, if an object isn't already located at *index* —that is, if *index* is greater than the value returned by **count**—this method just returns **nil**.

lastObject

– lastObject

Returns the last object in the List, or **nil** if there are no objects in the List. This method doesn't remove the object that's returned.

makeObjectsPerform:

– makeObjectsPerform:(SEL)*aSelector*

Sends an *aSelector* message to each object in the List, starting with the first and continuing through the List to the last object. The *aSelector* method must be one that takes no arguments. List should not be modified by side effects during the execution of this method.

makeObjectsPerform:with:

– makeObjectsPerform:(SEL)*aSelector*
with:*anObject*

Sends an *aSelector* message to each object in the List, starting with the first and continuing through the List to the last object. The *aSelector* method must be one that takes a single argument of type **id**. The message is sent with *anObject* as the argument. List should not be modified by side effects during the execution of this method.

objectAt:

– objectAt:(unsigned)*index*

Returns the **id** of the object located at slot *index,* or **nil** if *index* is beyond the end of the List.

read:

– read:(NXTypedStream *)*stream*

Reads the List object from an archive

removeLastObject

– removeLastObject

Removes the object occupying the last position in the List and returns it. If there are no objects in the List, this method returns **nil**.

removeObject:

– removeObject:*anObject*

Removes the first occurrence of *anObject* from the List, and returns it. If *anObject* isn't in the List, this method returns **nil**.

The positions of the remaining objects in the List are adjusted so there's no gap.

removeObjectAt:

– removeObjectAt:(unsigned)*index*

Returns the object located at *index* and removes it from the list. If there is no object at *index*, this method returns **nil**.

The positions of the remaining objects in the List are adjusted so there's no gap.

replaceObject:with:

– replaceObject:*anObject*
with:*newObject*

Returns the object at *index* and replaces it with *newObject*. If there is no object at *index* or *newObject* is **nil**, this method simply returns **nil**.

replaceObjectAt:with:

– replaceObjectAt:(unsigned)*index*
with:*newObject*

Replaces the first occurrence of *anObject* in the List with *newObject* and returns *anObject.* However, if *newObject* is **nil** or *anObject* isn't in the List, this method does nothing but return **nil**.

setGrowAmount:

– setGrowAmount:(unsigned)*numSlots*

Sets the amount of memory that the List should grow or shrink by. The argument, *numSlots*, is a number of objects.

write:

– write:(NXTypedStream *)*stream*

Stores the List object in an archive

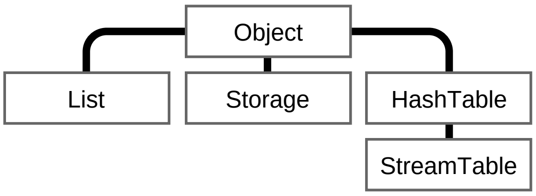


Figure 1. Inheritance Hierarchy of the Common Classes

OpenPanel

INHERITS FROM	SavePanel : Panel : Window : Responder : Object
REQUIRES HEADER FILES	appkit.h
DEFINED IN	The Application Kit, version 0.9

CLASS DESCRIPTION

The OpenPanel provides a convenient way for an application to query the user for the name of a file to open. It can only be run modally (the user should use the directory browser in the Workspace for non-modal opens). It allows the specification of certain types (i.e. file name extensions) of files to be opened. Every application has one and only one OpenPanel, and the method **new** returns a pointer to it.

See the class description for SavePanel for more information.

INSTANCE VARIABLES

Inherited from Object	struct _SHARED	*isa;
Inherited from Responder	id	nextResponder;
Inherited from Window	NXRect	frame;
	id	contentView;
	id	delegate;
	id	firstResponder;
	id	lastLeftHit;
	id	lastRightHit;
	id	counterpart;
	id	fieldEditor;
	int	winEventMask;
	int	windowNum;
	float	backgroundGray;
	struct _wFlags	wFlags;
	struct _wFlags2	wFlags2;
	id	miniWindowView;
Inherited from Panel	(none)	
Inherited from SavePanel	id	form;
	id	browser;
	id	okButton;
	id	accessoryView;
	char	filename[];
	char	directory[];
	char	**filenames;

	const char	*requiredType;
	struct _spFlags	spFlags;
Declared in OpenPanel	NXFileFilterFunc	fileFilterFunc;
	const char *const	*filterTypes;

fileFilterFunc	function to filter files
filterTypes	types allowed to open

METHOD TYPES

Creating an OpenPanel	+ newContent:style:backing:buttonMask:defer:
Filtering files	- allowMultipleFiles: - fileFilterFunc - setFileFilterFunc:
Querying the chosen files	- filenames
Running the OpenPanel	- runModalForDirectory:file: - runModalForDirectory:file:types: - runModalForTypes:

FACTORY METHODS

newContent:style:backing:buttonMask:defer:

+ **newContent:**(const NXRect *)*contentRect*
style:(int)*aStyle*
backing:(int)*bufferingType*
buttonMask:(int)*mask*
defer:(BOOL)*flag*

Creates a new OpenPanel (actually every app has no more than one OpenPanel, this returns a pointer to it). A simpler interface to creating the OpenPanel is via the inherited **new** method which calls this method with all the appropriate arguments.

INSTANCE METHODS

allowMultipleFiles:

– allowMultipleFiles:(BOOL)*flag*

If *flag* is YES, then the user can select more than one file in the browser. If multiple files are allowed, then the **filename** method will be non-NULL only if one and only one file was selected. The **filenames** method will always return the selected files (even if only one file was selected). Note further that, though **filename** always returns a fully-specified path, **filenames** never returns a fully-specified path (the files in the list are always relative to the path returned by **directory**).

fileFilterFunc

– (NXFileFilterFunc)**fileFilterFunc**

Sets the function that will be called to filter files that match the list of suffixes.

filenames

– (const char *const *)**filenames**

Returns a NULL terminated list of files (relative to the path returned by **directory**). This will be valid even if allowMultipleFiles is NO. This is the preferred way to get the name(s) of the file(s) that the user has chosen.

runModalForDirectory:file:

– (int)**runModalForDirectory:**(const char *)*path*
file:(const char *)*name*

Initializes the panel to the file specified by *path* and *name*, then displays it and begins its event loop.

runModalForDirectory:file:types:

– (int)**runModalForDirectory:**(const char *)*path*
file:(const char *)*name*
types:(const char *const *)*fileTypes*

Loads up the directory specified in *path* and optionally set *name* as the default file to open. *fileTypes* is a NULL-terminated list of suffixes (not including the ".”'s) to be used to filter which files the user is given the opportunity to open. If the FIRST item in the list is a NULL, then all ASCII files will be included.

runModalForTypes:

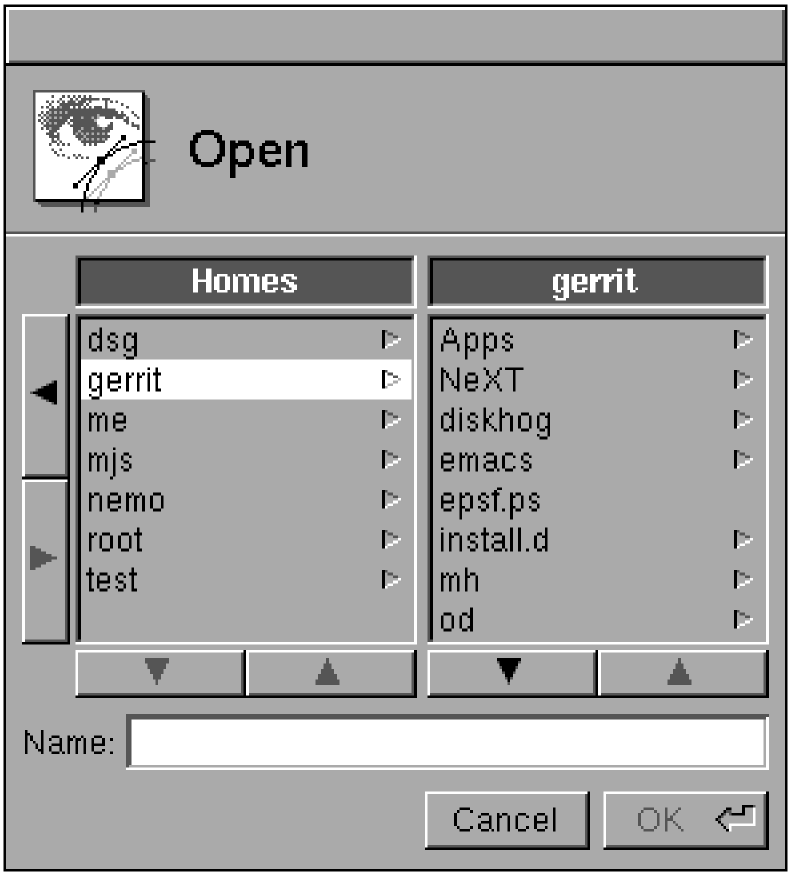
– (int)**runModalForTypes:**(const char *const *)*fileTypes*

Same as **runModalForDirectory:file:types:** except that the last directory from which a file was chosen is used.

setFileFilterFunc:

– setFileFilterFunc:(NXFileFilterFunc)*aFunc*

Sets the function that will be called to filter files that will be displayed in the browser. The file filter function should return YES if it wants the file to be included in the list of chooseable files, NO otherwise.



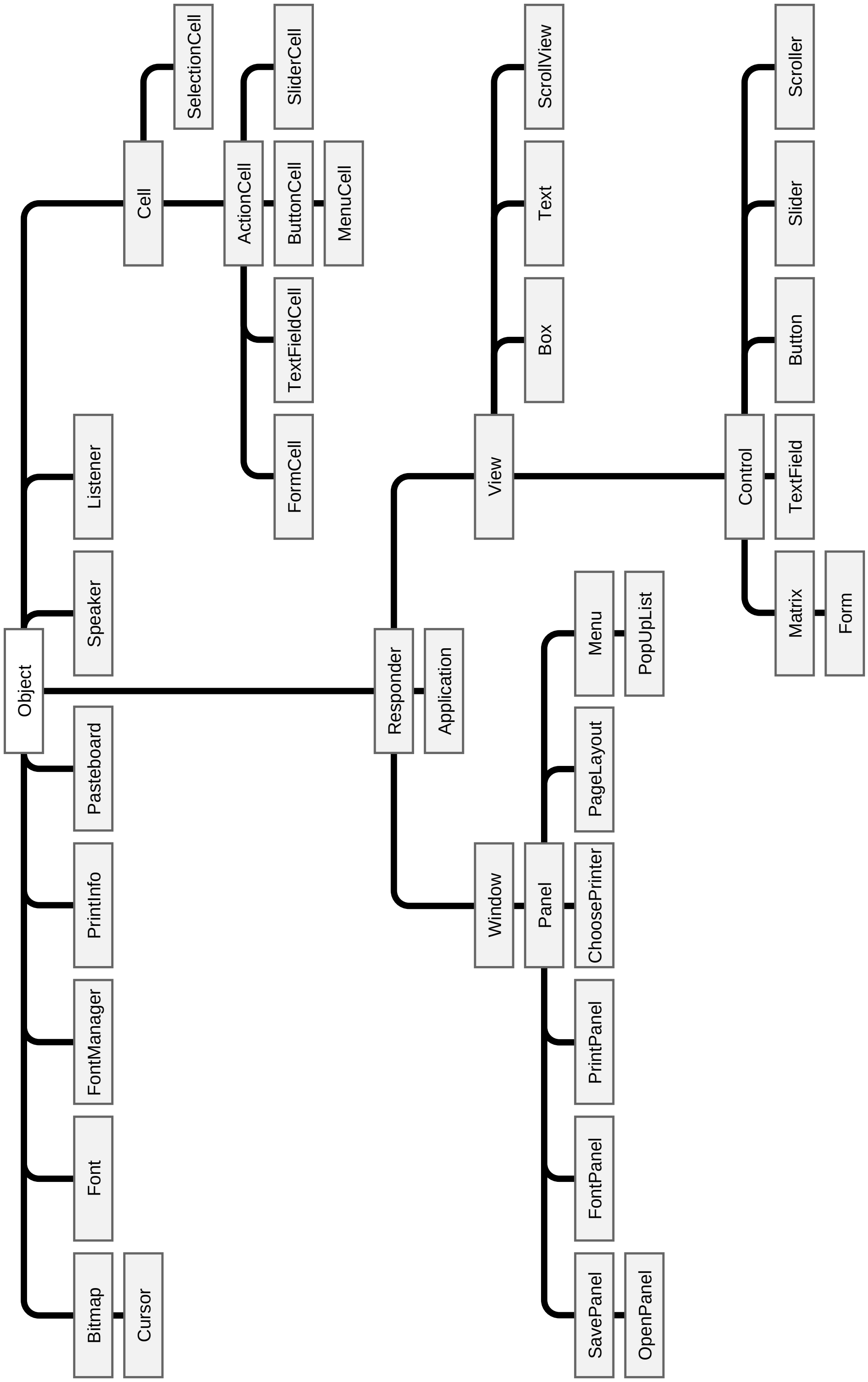


Figure 2. Application Kit Inheritance Hierarchy

Debugging using GDB - the GNU Source Level Debugger

To debug a program with GDB, type "gdb programname" to a shell. GDB commands include:

run *arguments...*
Start the program with the specified command line arguments.

break *linenumber*
break *function*
break *method*
break *filename:function*
break *filename:linenumber*
Place a breakpoint at the specified location. You can also specify an **if** clause with any of the above:

break *function* **if** *expression* (See *expression*, bottom of next page)

tbreak *args*
Place a one-time breakpoint. Takes same type of arguments as **break**.

info *breakpoints*
List all breakpoints, with their status and breakpoint numbers.

disable *pbnums...*
enable *pbnums...*
delete *bpnums...*
Temporarily disable/enable/delete breakpoints. Specify breakpoint numbers.

commands *bpnum*
Specify commands to be executed when breakpoint bpnum is reached.

list *args*
Lists source lines. Arguments are same as those for the **break** command.

step *count*
Run count lines of source. Number of lines defaults to one.

next *count*
Similar to **step**, but do not step into functions.

finish
Run until the current function/method returns.

backtrace
Show stack frames; useful in discovering where you are after a crash.

frame *framenumber*

Start examining the frame with the specified frame number.

print *expression*

Print the value of the *expression* (See *expression*, below)

set *variable* = *expression*

Assign value of *expression* to *variable* (See *expression*, below).

info classes *regexp*
info selectors *regexp*
info types *regexp*

Show info about the classes/selectors/types whose names match the regular expression *regexp.*

pclass *classname*

Show the methods defined for the specified class.

ptype *typename*

Show the type definition of the specified type.

whatis *expression*

Show the type of the specified expression. The expression is not evaluated.

expression

Any valid C or Objective-C expression, evaluated within the current stack frame. Expressions can contain the symbols $ (referring to the last value printed), $$ (the value before the last), $*n* (the *n*th value from value history), or $*var* (a convenience variable, created on the fly if necessary). Use **info history** to see the value history.

help *command*

GDB has plenty of help. Use this command to find out more about the above (and other) GDB commands.

Glossary

abstract superclass:
In Objective-C, a class that's defined solely so that other classes can inherit from it. Programs don't use instances of an abstract class, only of its subclasses.

action message:
In the Application Kit, a message sent by a Control object (such as a Button or a Slider). The message translates the user's action in the Control into a specific instruction for the application. See also *target.*

active application:
The application currently associated with keyboard events. Menus are visible on-screen only for the active application, and only the active application can have the current key window and main window.

ancestor:
In the Application Kit, a View is said to be the ancestor of all the Views below it in the view hierarchy, including its subviews. See also *descendant.*

Application Kit:
The Objective-C classes and C functions available for implementing the NeXT window-based user interface in an application.

class:
In Objective-C, a particular kind of object. Objects that have access to the same methods and have the same types of instance variables belong to the same class. A class definition declares the instance variables and defines the methods for all members of the class.

class method:
In Objective-C, a method that can be used by the class object rather than by instances of the class..

class object:
In Objective-C, an object that knows how to create new objects (instances) of a class. Class objects are created by the compiler and have the same name as the class; they're the complied version of the class.

delegate:
In the Application Kit, an object that acts on behalf of another object. Window, Application, Text, Listener, and Speaker objects can be assigned delegates.

descendant:
In the Application Kit, a View is said to be the descendant of all the Views above it in the view hierarchy, including its superview. See also *ancestor*.

dispatch table:
In Objective-C, a table used to implement run-time messaging. Each object class has a dispatch table that associates method selectors with the addresses of the method in memory.

dynamic binding:
Binding an object data structure with the method the object is to perform at run time, rather than at compile time.

event:
A keyboard or mouse action or other occurrence that the application may want to respond to.

event dispatcher:
The part of the Window Server that accepts user input such as keyboard and mouse actions and decides which window to assign it to.

event message:
In the Application Kit, a message to perform a method named after an event or subevent. Event messages are used to dispatch events to the objects that will respond to them. See also *action message*.

factory:
Same as *factory object* or *class object.*

factory method:
Same as *class method.*

factory object:
Same as *class object*.

first responder:
In the Application Kit, the object that will have the first chance to respond to keyboard event messages, mouse-moved event messages, and action messages with user-selected targets. Each Window has its own first responder, which it changes in response to mouse-down events.

foundation class:
Any class defined by Objective-C and provided with the compiler. These classes are at the top of the inheritance hierarchy and provide a foundation for the classes defined in programs and the software kits.

id:
In Objective-C, an object type defined as a pointer to the object data structure.

inheritance:
In object-oriented programming, the ability of a superclass to pass its characteristics (methods and instance variables) on to its subclasses. In Mach, the transfer of address space access rights from a parent process to a child process.

inheritance hierarchy:
In object-oriented programming, the hierarchy of classes that's defined by the arrangement of superclasses and subclasses. Every class (except Object, which is at the root of the hierarchy) has a superclass, and any class may have an unlimited number of subclasses. Through its superclass, each class inherits from those above it in the hierarchy.

instance:
In Objective-C, any object that's not a class object is said to be an instance of its class.

instance method:
In Objective-C, any method that can be used by an instance of a class rather than by the class object.

instance variable:
In Objective-C, a variable that's part of an object's private data structure. Instance variables are declared in a class definition and become part of all the objects that are instances of the class.

Interface Builder:
A tool that lets you graphically specify your program's user interface. It sets up the corresponding objects for you and makes it easy for you to establish connections between these objects and your own code where needed.

key equivalent:
In the Application Kit, the character that can be used as the keyboard alternative for a given object.

makefile:
A specification file used by the program **make** to build an executable version of your application. A makefile details the files and dependencies on which your application is built.

message:
In object-oriented programming, a message is the method selector (name) and arguments that are sent to an object; it tells the receiving object what to do. In Mach, a message consists of a header and a variable-length body; operating system services are invoked by passing a message from a thread to the port representing the task that provides the desired service.

method:
In object-oriented programming, a procedure that can be executed by an object.

Music Kit:
The Objective-C classes and C functions available for music composition, manipulation, synthesis, and peformance.

next responder:
In the Application Kit, the object that will be sent event and action messages that the intended receiver can't handle. See also *responder chain*.

NextStep:
NeXT's application development and user environment, consisting of the Workspace Manager, Interface Builder, Application Kit, and Window Server.

nil:
In Objective-C, an object **id** with a value of 0.

object:
A programming unit that groups together a data structure (instance variables) and the operations (methods) that can use or affect that data; the central focus of object-oriented programming.

polymorphism:
In object-oriented programming, the ability of different objects to respond each in their own way to the same message..

receiver:
In object-oriented programming, the object that receives a message.

responder chain:
In the Application Kit, a linked list of Responder objects that's formed by initializing each object's next responder with the **id** of another object.

selector:
In Objective-C, the name of a method when it's used in a source-code message to an object, or the integer that replaces the name when the source code is compiled.

Sound Kit:
The Objective-C classes and C functions available for creating sound effects, doing speech analysis, and performing other sound manipulation.

subclass:
For any given class of objects, any class that's one step below it in the inheritance hierarchy.

superclass:
For any given class of objects, the class that's one step above it in the inheritance hierarchy.

supermenu:
A menu containing a command that controls another menu, its submenu.

target:
In the NeXT user interface, what the user selects to be acted on by a menu command or a control within a panel—for example, text that’s deleted by the Cut command. In the Application Kit, the object that’s receives action messages from a Control.

Window Server:
A process that dispatches user events to windows and enables applications to perform drawing operations with the PostScript language.