# Intro to Objective C on the NeXT Machine

*by Gerrit Huizenga*
gerrit@cc.purdue.edu

## Abstract

This course will provide you 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 Object 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 the standard Classes of Objects provided by NeXT and how to use them.

# References

## 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 Object Oriented Programming

## Outside Reading

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

# Object Oriented Programming

## What is Object Oriented Programming?

1. **Grouping of functionality and data**

   A single module contains a group of data and all the routines that act on that data

2. **Separating the interface from the implementation**

   Interface defines only the routines and can be exported to any other modules. The implementation can then be hidden and modified without affecting other modules.

3. **Type-dependent operations**

   An operation such as “sort” can be defined for each type of data (or *object*), each version of which may be implemented differently, but all of which do something similar.

4. **Code sharing**

   Code can be easily reused and/or extended without rewriting through *inheritance*.

5. **Hierarchical partitioning of functionality**

   Code modules can have specific, limited interactions with other modules. Each module is responsible for its own set of data. Several independent modules can then be put together like building blocks to perform a specific task.

6. **Solving problems a different way**

   Allows the programmer to solve a problem using a possibly different strategy which may easier to understand in some applications.

## How is Object Oriented Programming Different?

1. **Syntactical support for abstract data types**

   Abstract data types are a grouping of data and the functions which operate on that data. An Object Oriented Programming Language typically supplies support for this grouping of data and functions.

2. **Message-sends instead of function calls**

   The programmer typically thinks in terms of one data objects sending a message to another data object rather than in terms of which function calls what other functions.

3. **Compiler support for inheritance**

   The compiler for an Object Oriented Language tries to make it as easy as possible to resuse code through the use of inheritance.

4. **Programming from a different perspective**

   Program design is usually viewed in terms of how the data interacts, rather than in terms of how the various procedures interact.

## What is an Abstract Data Type?

In an Object Oriented Programming Language such as Objective C, an Abstract Data Type consists of a definition of data (similar to structs in C or records in Pascal) and the entire set of functions which can modify that data. Anyone who uses an Abstract Data Type needs to know how s/he can access or modify that data. However, no one except the implementor of the Abstract Data Type needs to know the details of the implementation.
Computer programs manipulate data. What they do to that data is important. How they do it, or what form the data takes is not. Therefore the programmer is free to create new data types and declare exactly what the program can do to manuipulate a variables of those new types.

## Why Use an Abstract Data Type?

1. **They provide a form of data integrity.**

   You provide the only methods which are allowed to modify your data. You can then tightly check any attempted modifications to that data, as well as rest assured that your data will not contain unexpected values.

2. **They allow flexibility for future modifications to the data's form**

   You are free to change the underlying implementation to use faster algorithms or more appropriate data storage methods. You can also provide many types of bug fixes without changing the interface. And you are free to make enhancements to the data structures and add new methods without affecting existing code which uses your module.

## Terminology Used in Objective C

1. **Class**

   In object-oriented languages, an Abstract Data Types is called a *Class*.

2. **Instance**

   A variable of a given *Class* is called an *Instance*.

3. **Object**

   An *Object* is an *Instance* of a *Class* (e.g., a Button *Object* is an *Instance* of the Button *Class*).

4. **Method**

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

## How Objects Interact

Instead of calling a function of another object directly, you send a message from your object to another object which asks the recipient to *perform* a particular method. You reference the destination object through the use of a local instance variable which points to that object.

You do not need to know the Class of the destination object. This allows you to send a generic message such as a request for the size of an object to any of a set of objects. The object will then either calculate in its own way the size of itself or return a message saying that it did not understand your request. This might be useful if one of your modules uses a data storage object. You can then use either a stack object or a queue object, but your code won't care and will simply tell the object to store or fetch an item or perhaps return the number of items currently being stored (such as with a "size" message).

# Example of Object Oriented Design

Access to the Unix filesystem is *Object Oriented*. Most of you are familiar with the following Unix system calls:

**fd = open("/tmp/output", O_RDWR, 0666);**

**close(fd)**

**read(fd, &buffer[0], 256)**

**write(fd, &buffer[0], cc)**

**lseek(fd, offset, whence);**

**ioctl(fd, request, arg);**

The **open()** call takes a file name as one of its arguments and returns a handle to a file. **Open()** 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 is a switch() statement which decides which device specific routines should 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 with which you are working, either by sending the appropriate commands to the disk drive or tape drive, by waiting for serial input from a terminal line, or by sending data to a printer.

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

In Objective C, the filesystem access routines might look like the following:

```
fd = [File open:"/tmp/output" flags:O_RDWR modes:0666];
[fd close];
cc = [fd read:&buffer[0] size:256];
cc = [fd write:&buffer[0] size:cc];
offset = [fd lseek:offset whence:0];
retval = [fd ioctl:request argument:arg];
```

# Inheritance

## Related classes can share common code.

1. **Always inherits all previous methods and data**

   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 from which it inherits understands).

2. **Can add new data**

   A new class can define additional data for this class.

3. **Can define new methods**

   A new class can define additional methods to which this class will respond.

4. **Can redefine methods**

   A new class can redefine a method which it has inherited. The new method can perform differently from the inherited method even though it has the same name.

5. **Can utilize functionality of previous methods and enhance them**

   A new class can implement a new method with the same name as a method it has inherited and also invoke the inherited method as part of its new implementation.

## Terminology

1. **Subclass**

   A class which inherits from another class is a subclass of the class from which it inherits. Subclass is often used as a verb, "I subclassed the Vehicle class to create the Car class."

2. **Superclass**

   A superclass is a class from which some subclass inherits.

3. **Override**

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

## Programming Philosophy

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

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

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

4. **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 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 might 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 from which this class inherits (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 with 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 you don’t spcify a return type, 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.

## Sending Messages

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.

## To what objects can an object send messages?

1. **itself (this is very common)**

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

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

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

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

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

```
[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 **id** of the object which received the current message.

Remember **push:and:** ? Here is how that method might 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 send a message to your superclass?

Because you typically want to inherit much of the functionality that the superclass's method already provides. You don't want to have to reimplement that code, but you usually want to enhance it by adding new code such as error checking or more initialization.

## How do you send a message to the superclass?

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. For example,

```
id s;
s = [Stack new];
```

This creates a new instance of a Stack and **s** will contain a pointer to that instance.

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

```
@implementation Stack      /* Definition of a Class to implement a Stack */

- free          /* release all space associated with an instance of class Stack */
{
     StackLink *next;  /* temporary for pointer to a Stack element */

     while (top != (StackLink*) 0) {  /* free each element in */
          next = top->next;                /* the linked list */
          free( (char *) top);
          top = next;
     }

return [super free];  /* perform the Object class's free method which */
                              /* frees all space for this instance */
}

< other methods >

@end
```

## File: Stack.h

```
#import <objc/Object.h> /* include the method definitions for our parent class */

typedef struct StackLink {
     int data;
     struct StackLink *next;
} StackLink; /* This declares a local typedef for use inside this Object */

/* The typedef StackLink may be used outside the interface if passed in as
   argument to a method */


@interface Stack : Object /* declare our superclass */

{      /* instance variables which are local to each instance of Stack */
     StackLink *top; /* Pointer to top of Stack (implemented as a linked list) */
     unsigned int size; /* Current number of elements on the stack */
}

/* Methods implemented by our Class. These methods are either new or they override
   the methods that we inherited from our superclass */

- free;                      /* method to free the entire stack */
- push: (int) value;         /* Put an integer on the stack */
- (int) pop;                 /* Return the current value on the top of stack */
- (unsigned int) size;       /* Return the current number of elements on the stack */

@end /* End of the @interface section */
```

## File: Stack.m

```
#import "Stack.h" /* include our interface spec, typedefs, etc */

@implementation Stack /* Begin the definition of the implementation */

#define NULL_LINK (StackLink *) 0 /* Local, unexported definition */

- free /* Definition of instance method which free's all resources related to Stack */
{
     StackLink *next; /* Local variable, NOT an instance variable */

     while (top != NULL_LINK) {
          next = top->next; /* Free the local elements */
          free ((char *) top);
          top = next;
     }
     return [super free]; /* Free this instance of Stack */
}

- push: (int) value /* push: takes an integer as an argument */
{
     StackLink *newLink;
     /* Allocate space for an element here; our Stack is a linked list */
     newLink = (StackLink *)malloc(sizeof StackLink);
     if (newLink == 0) {
          fprintf(stderr, "Out of memory\n");
          /* Add better error recovery here */
          return nil;
     }
     newLink->data = value;
     newLink->next = top;
     top = newLink;
     size++; /* Keep our bookkeeping straight */
```

```
    return self; /* Return self on success (non-nil) */
}

- (int) pop /* Pop takes no arguments and returns the int on the top of stack*/
{
    int value;
    StackLink *topLink;

    if (0 != size) { /* Locate the value, remove from Linked List */
        topLink = top;
        top = top->next;
        value = topLink->data;
        free (topLink);
        size--; /* Again, keep bookkeeping straight */
    } else {
        value = 0; /* return 0 is stack is empty */
    }
    return value; /* otherwise, return the value we found */
}

- (unsigned int) size /* return the number of elements on the Stack */
{
    return size;
}
```

/* We could also have a method called -setSize: which we could use internatlly to verify the value we put on the stack. Here we don't want the value to become negative */

```
@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> /* The Kitchen Sink, avoid when possible*/
#import <objc/Object.h> /* Include definitions for our superclass */
/* A Class definition for an HP-like calculator */
@interface CalculatorBrain : Object
{
    id stack; /* To hold an instance of the Stack Class we just defined */
    int accumulator; /* A copy of the value currently being displayed */
    BOOL accumulatorEntered; /*Has value of accumulater been pushed?*/
    id display; /* Some form of display object */
}

/* Private */ /* Objective C doesn't really have a good way to make methods private */

- handleDigit: (int) digit; /* Internal method to process a digit */

/* Public */

- digit:sender; /* Method invoked when a digit is pressed on the calculator */
- add:sender; /* Calculator keys are bound to these methods */
- 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]; /* Create a new instance of CalculatorBrain */
     stack = [Stack new]; /* Create a Stack for our PostFix calculator */
     accumulatorEntered = YES; /* Do local initialization */
     return self; /* Return self so other Classes can inherit from us */
}

- setDisplay:anObject /* Gives our Object a way to name its output */
{
     display = anObject;
     return self;
}

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

     [display setIntValue:accumulator]; /* Redisplay new value */
     return self;
}
```

```
- add: sender /* sender is the id of the object which sent the original message */
{
     if (accumulatorEntered == NO) {
          [self enter:self]; /* push accumulator if necessary */
     }
     accumulator = [stack pop] + [stack pop];
     [self enter:self]; /* Add top two values, push result on stack */
     [display setIntValue:accumulator]; /* Display new result */
     return self;
}

- digit:sender /* sender is the id of the object which sent the original message */
{
     int operand;
     /* In this case, sender is an instance of the Matrix Class. We ask that class for
      *the tag on the associated element in the sending matrix. The tag matches the
      * the number on the button in this case.
      */
     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 /* The zero key is bound directly to this function */
{
    return [self handleDigit:0];
}
@end
```

# Chapter 22:
# Class Specifications

Menu
MenuCell
OpenPanel
PageLayout
Panel
Pasteboard
PopUpList
PrintInfo
PrintPanel
Responder
SavePanel
Scroller
ScrollView
SelectionCell
Slider
SliderCell
Speaker
Text
TextField
TextFieldCell
View
Window

### Sound Kit Classes

Sound
SoundMeter
SoundView

### Music Kit Classes

Conductor
Envelope
FilePerformer
FileWriter
Instrument
Midi
Note
NoteFilter
NoteReceiver
NoteSender
Orchestra
Part

Partials
PartPerformer
PartRecorder
PatchTemplate
Performer
Samples
Score
ScorefilePerformer
ScorefileWriter
ScorePerformer
ScoreRecorder
SynthData
SynthInstrument
SynthPatch
TuningSystem
UnitGenerator
WaveTable

# Chapter 22:
# Class Specifications

This chapter describes each of the classes defined in the three software kits–the Application Kit, Sound Kit, and Music Kit–as well as the classes that come with the Objective-C compiler. The classes that come with the compiler can be used with any kit (and in programs that don't use the kits).
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. For this information, turn to the chapters in Part 1 of this manual.

## How to Read the Specifications

The class specifications are grouped according to kit; within each kit, they're arranged in alphabetical order by class.

### 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:

```
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. (This particular example shows the inheritance hierarchy for the Menu class of the Application Kit.)

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 |

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

There's also a master header file for the classes that come with the compiler:

objc/objc.h

If you include a master header files for any of the software kits, you don't need to also include this file; it's included by the kit file.

See Chapter 21, ™Header Files,for more information on these files. <<*This chapter isn't available on-line. See the header files themselves in* ***/usr/include****.>>*

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 |

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. All three l ibraries reside in the **/usr/lib** directory. (The ™ssuffix indicates that these ar e shared libraries; see Chapter 18, ™Programming Tools,for more information about using share d libraries.)

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.

Some classes define ™off-the-shelfobjects: Your progr am can create direct instances of the class, or modify it in a subclass definition.

Other classes are ™abstract superclasses You wouldn't c reate an instance of the class itself, but only of its subclasses. The kits define some subclasses for each abstract superclass; others can be defined by your application.

Occasionally, the class description will recommend that you define a subclass of a kit class, even though the kit class isn't abstract. The subclass allows you to customize an object to the needs of your application.

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 are presented first; methods that are used by instances (the objects produced by the class) 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.

METHODS IMPLEMENTED BY THE DELEGATE

If a class lets you define another object–a delegate–that can intercede on behalf of instances of the class, the methods that the delegate can implement are described in a separate section.

These are not methods defined in the class; rather, they're methods that you can define to respond to messages sent to the delegate.

If you define one of these methods, the delegate will receive automatic messages to perform it at the appropriate time. For example, if you define a **windowDidBecomeKey:** method for a Window's delegate, the delegate will receive **windowDidBecomeKey:** messages whenever the Window becomes the key window.

Messages are sent only if the delegate has a method that can respond. If you don't define a **windowDidBecomeKey:** method, no message will be sent.
Only certain classes define delegates. In the Application Kit, they are:

Application
Listener
Speaker
Text
Window

The delegate is inherited by subclasses.

CONSTANTS AND DEFINED TYPES

If a class makes use of symbolic constants or defined types that are specific to the class, they're listed in the last section of the class specification. Defined types are likely to show up in instance variable declarations, and as return and parameter types in method declarations. Symbolic constants typically define permitted return and argument values.

## Method Descriptions

By far, the major portion of each class specification is the description of methods defined in the class. When reading these descriptions, be especially attentive to three kinds of information that affect how the method can be used:

Whether you should implement your own version of the metho d
Whether you should have your version of the method include the kit-defined version
Whether you should ever send a message to an object to perfo rm the method

The next three sections examine these questions.

### Implementing Your Own Version of a Method

For the most part, the methods in a class definition act as a private library for objects belonging to that class. Just as programmers generally don't replace functions in the standard C library with their own versions, you generally wouldn't write your own versions of the methods provided for a class.

However, to add specific behavior to your application, you must override some of the methods that are defined in the kits. Often, the kit-defined method will do little or nothing that's of use to your application, but it will appear in messages initiated by other methods. To give content to the method, your application must implement its own version.
To override a kit method with one of your own design, simply define a subclass of the appropriate class and redefine the method. For example, the interface declaration for the CircleView class illustrated below shows that it does nothing more than override the View class's **drawSelf::** method.

```
@interface CircleView : View
- drawSelf:(NWRect *)drawRects :(int)rectCount;
@end
```

CircleView objects will perform its version of **drawSelf::** rather than the empty default version defined in View.

In contrast to methods that must be overridden, some methods should never be changed by the application. The kit depends on these methods doing just what they're currently programmed to do–nothing more and nothing less. While your application can use these methods, it's important that you don't override them when defining a subclass.

Most methods fit between these two extremes: They can be overridden, but it's not necessary for you to do so. If a method description is silent on the question of overriding the kit method, you can be certain that it fits into this middle category. It's a method that you can override, but like a function in the C library, you normally would have no reason to.

If a method is designed to be overridden, or if it should never be overridden, the method description explicitly says so.

## Retaining the Kit's Version of a Method

Some methods can be overridden, but only to add behavior, not to alter the default actions of the kit-defined method. When your application overrides one of these methods, it's important that it incorporate the very method it overrides. This is done by messaging **super** to perform the kit-defined version of the method. For example, if you write a new version of the kit method that moves a Window, you'd most likely still want it to move a Window. The easiest way to have it do that is to include the old method in the new one through a message to **super**.

```
- moveTo:(NWCoord)x :(NWCoord)y
    [super moveTo:x :y];
    /* your code goes here */
```

The kits occasionally require you to implement a new version of a method while preserving the behavior of the method you override. An example is the **write:** method, which archives an object by writing it to a typed stream. When you define a kit subclass, you may need to implement a version of this method that can archive the instance variables your subclass declares. So that a **write:** message will archive all of an object's instance variables, not just those declared in the subclass, your version of the method should begin by incorporating the version used by its superclass.

```
- write:(NXTypedStream *)stream
    [super write:stream];
    /* your code goes here */
```

Method descriptions explicitly mention that you should incorporate a method you override only when it's not obvious that it would be a good idea to preserve the default behavior in the new method.

## Sending a Message to Perform a Method

Some methods should never appear as messages in the code you write; you should never directly ask an object to perform the method. Typically, these are methods that your application will use indirectly, through other methods.

Most of these methods begin with a underscore and are treated as class-internal methods. However, some don't have an underscore and are included in the method descriptions. These are methods that your application can implement, even though it won't directly use them in a message. The messages to perform these methods originate in the kit.

The most notable example of this is the **drawSelf::** method that draws a View. Although you must implement a **drawSelf::** method for each View subclass you define, your code should never send a **drawSelf::** message. Instead, you send a display message; the display method (such as **display**, **displayIfNeeded**, or **display:::**) sees to it that the drawing context is properly set before initiating a **drawSelf::** message to the View.

The methods that respond to event messages (such as **mouseUp:**, **keyDown:**, and **windowExposed:**) also fall into this category. Event messages are initiated by the Application Kit when it receives events from the Window Server; you shouldn't initiate them in your own code.

The **write:** and **read:** methods for archiving and unarchiving are other examples of methods that shouldn't be sent directly to objects. They're generated by functions, such as **NXWriteObject()** and **NXReadObject()**.

If a method is designed to respond to messages generated by other methods or by a kit, the method description will generally say so. If there's a penalty for generating the message within the code you write (as there is for **drawSelf::**), the description will include an explicit warning.

## List

| | |
|---|---|
| INHERITS FROM | Object |
| REQUIRES HEADER FILES | objc/List.h |
| DEFINED AS | A common class, version 1.0 |

CLASS DESCRIPTION

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

INSTANCE VARIABLES

| | | |
|---|---|---|
| *Inherited from Object* | Class | isa; |
| *Declared in List* | @public id | *dataPtr; |
| | unsigned | numElements; |
| | unsigned | maxElements; |

| | |
|---|---|
| dataPtr | Data of the List object |
| numElements | Actual number of elements |
| maxElements | Total allocated elements |

METHOD TYPES

Creating and freeing a List object- copy
- 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

Comparing lists - isEqual:

Sending messages to the objects

- makeObjectsPerform:
- makeObjectsPerform:with:

Managing the storage capacity

- capacity
- setAvailableCapacity:

Archiving

- read:
- write:

CLASS METHODS

**new**

+ **new**

Returns a new List.

### newCount:

+ **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. No insertion is done and **nil** is returned if *anObject* is **nil**.

### capacity

-(unsigned)**capacity**

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

### copy

**- copy**

Creates a new List, with the same objects. Storage for the objects themselves is not copied and therefore the same pointers appear both lists.

### count

- (unsigned)**count**

Returns the number of objects currently in the List.

### empty

**- empty**

Empties the List of all its objects. Current capacity remains.

### 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 NX_NOT_IN_LIST 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**. No insertion is done and **nil** is also returned if *anObject* is **nil**.

### isEqual:

- (BOOL)**isEqual:***anObject*

Compares two lists and returns TRUE if both lists have the same number of elements and pointers to the same objects. It is assumed anObject is really a List.

### 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**.

### setAvailableCapacity:

-**setAvailableCapacity:**(unsigned)*numSlots*

Sets the storage capacity of the List to *numSlots*objects. Only a storage allocation hint, does not change list elements. If the list already contains more than *numSlots* elements, its capacity is left unchanged.

### write:

- **write:**(NXTypedStream *)*stream*

Stores the List object in an archive

## OpenPanel

| | |
|---|---|
| INHERITS FROM | SavePanel : Panel : Window : Responder : Object |
| REQUIRES HEADER FILES | appkit.h |
| DEFINED IN | The Application Kit, version 1.0 |

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* | Class | 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; |
| *Inherited from Panel* | (none) | |
| *Inherited from SavePanel* | id | form; |
| | id | browser; |
| | id | okButton; |
| | id | accessoryView; |
| | id | separator; |
| | char | *filename; |
| | char | *directory; |
| | const char | **filenames; |
| | char | *requiredType; |
| | struct _spFlags | spFlags; |
| | unsigned short | directorySize; |
| *Declared in OpenPanel* | NXFileFilterFunc | fileFilterFunc; |
| | char | **filterTypes; |

| | |
|---|---|
| fileFilterFunc | Function to filter files |
| filterTypes | Types allowed to open |

METHOD TYPES

| | |
|---|---|
| Creating or Freeing an OpenPanel | - free<br>+ newContent:style:backing:buttonMask:defer: |
| Filtering files | - allowMultipleFiles:<br>- fileFilterFunc<br>- setFileFilterFunc: |
| Querying the chosen files | - filenames |
| Running the OpenPanel | - runModalForDirectory:file:<br>- runModalForDirectory:file:types:<br>- runModalForTypes: |

CLASS METHODS

### newContent:style:backing:buttonMask:defer:

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

Don't call this method, perform **new** (inherited) instead.

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.

### free

- **free**

Frees the storage used by the OpenPanel object. The next time new is sent to the OpenPanel, it will be recreated. You probably never need to call this method since there is one shared instance of the OpenPanel.

### 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.

CONSTANTS AND DEFINED TYPES

```
typedef (*NXFileFilterFunc)(id self, NXDirEntry *dirEntry);

/* Tags of views in the SavePanel */
#define NX_OPICONBUTTON     NX_SPICONBUTTON
#define NX_OPTITLEFIELD     NX_SPTITLEFIELD
#define NX_OPBROWSER        NX_SPBROWSER
#define NX_OPCANCELBUTTONNX_SPCANCELBUTTON
#define NX_OPOKBUTTON       NX_SPOKBUTTON
#define NX_OPFORM           NX_SPFORM
```

# 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*)

**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.