Objective-C Language and GNUstep Base Library Programming Manual

Francis Botto (Brainstorm)
Richard Frith-Macdonald (Brainstorm)
Nicola Pero (Brainstorm)
Adrian Robert

Copyright © 2001-2004 Free Software Foundation

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

Table of Contents

1 Introduction ... 3

1.1 What is Object-Oriented Programming? ... 3
1.1.1 Some Basic OO Terminology ... 3
1.2 What is Objective-C? ... 4
1.3 History ... 5
1.4 What is GNUstep? ... 6
1.4.1 GNUstep Base Library ... 6
1.4.2 GNUstep Make Utility ... 7
1.4.3 A Word on the Graphical Environment ... 7
1.4.4 The GNUstep Directory Layout ... 7
1.5 Building Your First Objective-C Program ... 8

2 The Objective-C Language ... 11

2.1 Non OO Additions ... 11
2.2 Objects ... 11
2.2.1 Id and nil ... 12
2.2.2 Messages ... 12
2.2.3 Polymorphism ... 13
2.3 Classes ... 13
2.3.1 Inheritance ... 13
2.3.2 Inheritance of Methods ... 14
2.3.3 Overriding Methods ... 14
2.3.4 Abstract Classes ... 14
2.3.5 Class Clusters ... 15
2.4 NSObject: The Root Class ... 15
2.4.1 The NSObject Protocol ... 16
2.5 Static Typing ... 16
2.5.1 Type Introspection ... 16
2.5.2 Referring to Instance Variables ... 17
2.6 Working with Class Objects ... 17
2.6.1 Locating Classes Dynamically ... 18
2.7 Naming Constraints and Conventions ... 18
2.8 Strings in GNUstep ... 19
2.8.1 Creating NSString Static Instances ... 19
2.8.2 NSString +stringWithFormat: ... 19
2.8.3 C String Conversion ... 20
2.8.4 NSMutableString ... 20
2.8.5 Loading and Saving Strings ... 21

3 Working with Objects ... 23

3.1 Initializing and Allocating Objects ... 23
3.1.1 Initialization with Arguments ... 23

3.1.2 Memory Allocation and Zones 24
3.1.3 Memory Deallocation 25
3.2 Memory Management 25
3.2.1 Explicit Memory Management 25
3.2.2 OpenStep-Style (Retain/Release) Memory Management ... 25
3.2.2.1 Autorelease Pools 27
3.2.2.2 Avoiding Retain Cycles 28
3.2.2.3 Summary 28
3.2.3 ObjC-2 and Automated Reference Counting 29

4 Writing New Classes 31
4.1 Interface 31
4.1.1 Interface Capabilities 31
4.1.2 Including Interfaces 33
4.1.3 Referring to Classes - @class 33
4.2 Implementation 34
4.2.1 Writing an Implementation 35
4.2.2 Super and Self 35
4.2.3 Instance Initialization 36
4.2.4 Flexible Initialization 39
4.2.5 Instance Deallocation 40
4.3 Protocols 41
4.3.1 Declaring a Formal Protocol 41
4.3.2 Implementing a Formal Protocol 41
4.3.3 Using a Formal Protocol 42
4.4 Categories 43
4.4.1 Category Overrides 44
4.4.2 Categories as an Implementation Tool 44
4.4.3 Categories and Protocols 44
4.5 Simulating Private and Protected Methods 45
4.6 Simulating Class Variables 46

5 Advanced Messaging 49
5.1 How Messaging Works 49
5.2 Selectors 50
5.2.1 The Target-Action Paradigm 50
5.2.2 Obtaining Selectors 51
5.2.3 Avoiding Messaging Errors when an Implementation is Not Found 51
5.3 Forwarding 52
5.4 Implementations 53

6 Exception Handling, Logging, and Assertions . . 55

6.1 Exceptions . . . 55
6.1.1 Catching and Handling Exceptions . . . 55
6.1.2 Passing Exceptions Up the Call Stack . . . 57
6.1.3 Where do Exceptions Originate? . . . 57
6.1.4 Creating Exceptions . . . 57
6.1.5 When to Use Exceptions . . . 58
6.2 Logging . . . 59
6.2.1 NSLog . . . 59
6.2.2 NSDebugLog, NSWarnLog . . . 59
6.2.3 Last Resorts: GSPrintf and fprintf . . . 61
6.2.4 Profiling Facilities . . . 61
6.3 Assertions . . . 61
6.3.1 Assertions and their Handling . . . 61
6.3.2 Custom Assertion Handling . . . 62
6.4 Comparison with Java . . . 62

7 Distributed Objects . . . 63

7.1 Object Interaction . . . 63
7.2 The GNUstep Solution . . . 63
7.2.1 Code at the Server . . . 64
7.2.2 Code at the Client . . . 65
7.2.3 Using a Protocol . . . 67
7.2.4 Complete Code for Telephone Directory Application . . . 68
7.2.5 GNUstep Distributed Objects Name Server . . . 69
7.2.6 Look Ma, No Stubs! . . . 70
7.3 A More Involved Example . . . 70
7.3.1 Protocol Adopted at Client . . . 71
7.3.2 Protocol Adopted at Server . . . 71
7.3.3 Code at the Client . . . 71
7.3.4 Code at the Server . . . 74
7.4 Language Support for Distributed Objects . . . 79
7.4.1 Protocol Type Qualifiers . . . 79
7.4.2 Message Forwarding . . . 81
7.5 Error Checking . . . 82
7.5.1 Vending the Server Object . . . 82
7.5.2 Catching Exceptions . . . 82
7.5.3 The Connection Fails . . . 82

8 Base Library . . . 83

8.1 Copying, Comparing, Hashing Objects . . . 83
8.2 Object Containers . . . 84
8.3 Data and Number Containers . . . 85
8.3.1 NSData . . . 85
8.3.2 NSValue . . . 86

8.3.3 NSNumber 86
8.3.4 NSRange, NSPoint, NSSize, NSRect 87
8.4 Date/Time Facilities 87
8.5 String Manipulation and Text Processing 87
8.5.1 NSScanner and Character Sets 87
8.5.2 Attributed Strings 87
8.5.3 Formatters 87
8.6 File Handling 88
8.7 Persistence and Serialization 88
8.7.1 Property List Serialization 88
8.7.2 Archives 89
8.8 Utility 92
8.9 Notifications 92
8.10 Networking and RPC 93
8.10.1 Basic Networking 93
8.10.2 Remote Process Communications 94
8.11 Threads and Run Control 94
8.11.1 Run Loops and Timers 94
8.11.2 Tasks and Pipes 95
8.11.3 Threads and Locks 95
8.11.4 Using `NSConnection` to Communicate Between Threads 97
8.12 GNUstep Additions 98

Appendix A The GNUstep Documentation System 99

A.1 Quick Start 99
A.2 Cross-Referencing 100
A.3 Comment the Interface or the Implementation? 100
A.4 Comparison with OS X Header Doc and Java JavaDoc 101

Appendix B Application Resources: Bundles and Frameworks 103

Appendix C Differences and Similarities Between Objective-C, Java, and C++ 105

C.1 General 105
C.2 Language 105
C.3 Source Differences 106
C.4 Compiler Differences 106
C.5 Developer's Workbench 106
C.6 Longevity 106
C.7 Databases 106
C.8 Memory 106
C.9 Class Libraries 107

Appendix D Programming GNUstep in Java and Guile 109

Appendix E GNUstep Compliance to Standards .. 111

E.1 Conditional Compilation 111
E.2 User Defaults 111

Appendix F Using the GNUstep Make Package .. 113

F.1 Makefile Contents 113
F.1.1 Makefile Example 113
F.1.2 Makefile Structure 114
F.1.3 Debug and Profile Information 115
F.1.4 Static, Shared and DLLs 115
F.2 Project Types 115

Concept Index 117

The aim of this document is to provide a GNUstep/Objective-C programming manual (primarily tutorial in style) for the language, the GNUstep Base library, and the GNUstep Make package. While to focus is on Objective-C, the GNUstep libraries can also be used from Java and Guile, and some information on this usage is also included.

The manual does not cover installation instructions as these vary from system to system, and are documented fairly well in the GNUstep HOWTO (`../../../User/GNUstep/gnustep-howto_toc.html`).

The target audience for this manual is the C, C++, or Java programmer that wishes to learn to use Objective-C effectively. We assume that, while the reader is able to understand English, it is quite possibly not their native language.

For detailed class reference documentation the reader is directed to the GNUstep Base Library documentation, and to the Apple Cocoa Objective-C Foundation documentation (available through `http://www.apple.com`).

1 Introduction

The aim of this manual is to introduce you to the Objective-C language and the GNUstep development environment, in particular the Base library. The manual is organised to give you a tutorial introduction to the language and APIs, by using examples whenever possible, rather than providing a lengthy abstract description.

While Objective-C is not a difficult language to learn or use, some of the terms may be unfamiliar, especially to those that have not programmed using an object-oriented programming language before. Whenever possible, concepts will be explained in simple terms rather than in more advanced programming terms, and comparisons to other languages will be used to aid in illustration.

1.1 What is Object-Oriented Programming?

There are several object-oriented (OO) programming languages in common use today and you have probably heard of some of them: C++ and Java for example, and of course Objective-C. OO languages all have one thing in common: they allow you to design and write programs in a different way than if you used a traditional procedural language like C or Pascal.

Procedural languages provide the programmer with basic building blocks that consist of data types, (integers, characters, float etc) and functions that act on that data. This forces the program designer to design the program using these same building blocks. Quite often this requires quite a leap in imagination between what the program must do and how it can be implemented.

Object-oriented languages allow the program designer to think in terms of building blocks that are closer to what the program will actually do. Rather than think in terms of data and functions that act on that data, OO languages provide you with objects and the ability to send messages to those objects. Objects are, in a sense, like mini programs that can function on their own when requested by the program or even another object.

For example, an object may exist that can draw a rectangle in a window; all you need to do as a programmer is send the appropriate messages to that object. The messages could tell the object the size of the rectangle and position in the window, and of course tell the object to draw itself. Program design and implementation is now reduced to sending messages to the appropriate objects rather than calling functions to manipulate data.

1.1.1 Some Basic OO Terminology

OO languages add to the vocabulary of more traditional programming languages, and it may help if you become familiar with some of the basic terms before jumping in to the language itself.

Objects

As stated previously, an object is one of the basic building blocks in OO programming. An object can receive messages and then act on these messages to alter the state of itself (the size and position of a rectangle object for example). In software an object consists of instance variables (data) that represent the state of the object, and methods (like C functions) that act on these variables in response to messages.

Rather than 'calling' one of its methods, an object is said to 'perform' one of its methods in response to a message. (A method is known as a 'member function' in C++.)

Classes

All objects of the same type are said to be members of the same class. To continue with the rectangle example, every rectangle could belong to a rectangle class, where the class defines the instance variables and the methods of all rectangles.

A class definition by itself does not create an object but instead acts like a template for each object in that class. When an object is created an 'instance' of that class is said to exist. An instance of a class (an object) has the same data structure (instance variables) and methods as every other object in that class.

Inheritance

When you define a new class you can base it on an existing class. The new class would then 'inherit' the data structure and methods of the class that you based it on. You are then free to add instance variables and methods, or even modify inherited methods, to change the behavior of the new class (how it reacts to messages).

The base class is known as the 'superclass' and the new class as the 'subclass' of this superclass. As an example, there could be a superclass called 'shapes' with a data structure and methods to size, position and draw itself, on which you could base the rectangle class.

Polymorphism

Unlike functions in a procedural program such as C, where every function must have a unique name, a method (or instance variable) in one class can have the same name as that in another class.

This means that two objects could respond to the same message in completely different ways, since identically named methods may do completely different things. A draw message sent to a rectangle object would not produce the same shape as a draw message sent to a circle object.

Encapsulation

An object hides its instance variables and method implementations from other parts of the program. This encapsulation allows the programmer that uses an object to concentrate on what the object does rather than how it is implemented.

Also, providing the interface to an object does not change (the methods of an object and how they respond to received messages) then the implementation of an object can be improved without affecting any programs that use it.

Dynamic Typing and Binding

Due to polymorhism, the method performed in response to a message depends on the class (type) of the receiving object. In an OO program the type, or class, of an object can be determined at run time (dynamic typing) rather than at compile time (static typing).

The method performed (what happens as a result of this message) can then be determined during program execution and could, for example, be determined by user action or some other external event. Binding a message to a particular method at run time is known as dynamic binding.

1.2 What is Objective-C?

Objective-C is a powerful object-oriented (OO) language that extends the procedural language ANSI C with the addition of a few keywords and compiler directives, plus one

syntactical addition (for sending messages to objects). This simple extension of ANSI C is made possible by an Objective-C runtime library (libobjc) that is generally transparent to the Objective-C programmer.

During compilation of Objective-C source code, OO extensions in the language compile to C function calls to the runtime library. It is the runtime library that makes dynamic typing and binding possible, and that makes Objective-C a true object-oriented language.

Since Objective-C extends ANSI C with a few additional language constructs (the compiler directives and syntactical addition), you may freely include C code in your Objective-C programs. In fact an Objective-C program may look familiar to the C programmer since it is constructed using the traditional `main` function.

```
#include <stdio.h>
#include <objc/objc.h>

int main (void)
{

  /* Objective C and C code */

  return(0);
}
```

Objective-C source files are compiled using the standard GNU **gcc** compiler. The compiler recognises Objective-C source files by the `.m` file extension, C files by the `.c` extension and header files by the `.h` extension.

As an example, the command **$gcc -o testfile testfile.m -lobjc** would compile the Objective-C source file `testfile.m` to an executable named `testfile`. The `-lobjc` compiler option is required for linking an Objective-C program to the runtime library. (On GNU/Linux systems you may also need the `-lpthreads` option.)

The GNUstep **make** utility provides an alternative (and simple) way to compile large projects, and this useful utility is discussed in the next section.

Relative to other languages, Objective-C is more dynamic than C++ or Java in that it binds all method calls at runtime. Java gets around some of the limitations of static binding with explicit runtime "reflection" mechanisms. Objective-C has these too, but you do not need them as often as in Java, even though Objective-C is compiled while Java is interpreted. More information can be found in Appendix Appendix C [Objective-C Java and C++], page 105.

1.3 History

Objective-C was specified and first implemented by Brad Cox and his company Stepstone Corporation during the early 1980's. They aimed to minimally incorporate the object-oriented features of Smalltalk-80 into C. Steve Jobs's NeXT licensed Objective-C from StepStone in 1988 to serve as the foundation of the new NeXTstep development and operating environment. NeXT implemented its own compiler by building on the *gcc* compiler, modifications that were later contributed back to gcc in 1991. No less than three runtime libraries were subsequently written to serve as the GNU runtime; the one currently in use was developed by Danish university student Kresten Krab Thorup.

Smalltalk-80 also included a class library, and Stepstone's Objective-C implementation contained its own library based loosely on it. This in turn influenced the design of the NeXTstep class libraries, which are what GNUstep itself is ultimately based on.

After NeXT exited the hardware business in the early 1990s, its Objective-C class library and development environment, *NeXTstep*, was renamed *OpenStep* and ported to run on several different platforms. Apple acquired NeXT in 1996, and after several years figuring out how to smooth the transition from their current OS, they released a modified, enhanced version of the NeXTstep operating system as Mac OS X, or "10" in 1999. The class libraries in OS X contain additions related to new multimedia capabilities and integration with Java, but their core is still essentially the OpenStep API.

This API consists of two parts: the *Foundation*, a collection of non-graphical classes for data management, network and file interaction, date and time handling, and more, and the *AppKit*, a collection of user interface widgets and windowing machinery for developing full-fledged graphical applications. GNUstep provides implementations of both parts of this API, together with a graphical engine for rendering AppKit components on various platforms.

1.4 What is GNUstep?

GNUstep is an object-oriented development environment that provides the Objective-C programmer with a range of utilities and libraries for building large, cross-platform, applications and tools. It is split into three components: **Base**, non-graphical classes corresponding to the NeXTstep *Foundation* API, **GUI**, consisting of graphical classes corresponding to the NeXTstep *AppKit* API, and **Back**, a modular framework for rendering instance of the GUI classes on multiple platforms.

GNUstep is generally compatible with the OpenStep specification and with recent developments of the MacOS (Cocoa) API. Where MacOS deviates from the OpenStep API, GNUstep generally attempts to support both versions. See Appendix Appendix E [Compliance to Standards], page 111, for more detailed information.

This manual does not discuss the full functionality of GNUstep but concentrates on using the GNUstep Base library to create non-graphical programs, and the GNUstep **make** utility to compile these programs. Further information about GNUstep can be found at `http://gnustep.org`.

1.4.1 GNUstep Base Library

The GNUstep base library contains a powerful set of non-graphical Objective-C classes that can readily be used in your programs. At present there are approximately 70 different classes available, including classes to handle strings and arrays, dates and times, distributed objects, URLs and file systems (to name but a few). It is similar to but more stable than the non-graphical portion of the Java Development Kit (JDK) API (see Appendix Appendix C [Objective-C Java and C++], page 105, for more information).

Classes in the base library are easily identified since they begin with the upper case characters 'NS', as in `NSString`. Some examples in this manual use classes from the base library, but for complete documentation on the base library see the API documentation (`../Reference/index.html`).

1.4.2 GNUstep Make Utility

The GNUstep **make** utility is the GNU version of the UNIX make utility, plus a number of predefined rules specialized for building GNUstep projects. So what does it do? It simplifies the process of building (compiling and linking) a large project. You simply type **make** at the command prompt and the make utility takes care of file dependencies, only re-compiling source files that have changed, or that depend on files that have changed, since the last 'make' (a header file for example). It also takes care of including the proper GNUstep header and library references automatically.

Before using **make** you must first create a 'makefile' that lists all the files and file dependencies in your project. The easiest way to do this is to copy an existing makefile and change it to suit your own project.

The make utility will be used to build the Objective-C examples shown in this manual, and when an example can be compiled then the makefile will also be shown. For a full description of the make utility see its documentation (`../../Make/Manual/make_toc.html`).

1.4.3 A Word on the Graphical Environment

The GNUstep **GUI** component is discussed elsewhere (`../../Gui/ProgrammingManual/manual_toc.html`), but a brief overview is useful here. GNUstep GUI provides a collection of classes for developing graphical applications, including windows, controls (also known as widgets, and back-end components for event handling and other functions. Internally, the implementation is divided into two components, the *back end* and the *front end*. The front end provides the API to the developer, and makes display postscript (DPS) calls to the back end to implement it. The back-end converts the DPS calls into calls to the underlying window system. If you install GNUstep from source, you must first compile and install the front end, then compile and install the back end.

Implementations of the back-end have been produced for both X11 (Linux/UNIX systems), and Windows. There is also a quasi-native display postscript system similar to what was available on the NeXT but using Ghostscript to render to X11. This implementation is largely complete, but proved to be inefficient and difficult to optimize relative to the current back-end framework (which converts the DPS from the front end to window drawing commands immediately rather than relying on a postscript stack).

1.4.4 The GNUstep Directory Layout

The directories of a GNUstep installation are organized in a fashion that balances compatibility with NeXTstep/OpenStep/OS X with traditional Unix filesystem conventions. The highest level of organization consists of four *domains* - the System, Local, Network, and Users. *System* holds the main GNUstep installation, including the Base and GUI libraries and documentation. *Local* holds third party applications, custom extension libraries, etc., analogously to `/usr/local` on a Unix system. *Network* mounts shared files in a networked environment. *Users* usually exists as `$HOME/GNUstep` and holds preferences files and personal application data. There is further documentation (`../../../User/GNUstep/filesystem_toc.html`) available on the complete directory layout.

Usually, on a Unix-type system, the GNUstep installation will be found under `/usr/lib/GNUstep`.

1.5 Building Your First Objective-C Program

The following example will show you how to create and compile an Objective-C program. The example simply displays a text message on the screen, and there are easier ways to to do this, but the example does demonstrate a number of object-oriented features of Objective-C, and also demonstrates the use of **make** to compile an Objective-C program.

1. Create a new project directory to hold your project.

2. Create the following Objective-C source code using your favourite text editor and save it in the project directory with the filename `source.m`.

```
#include <stdio.h>

/*
 * The next #include line is generally present in all Objective-C
 * source files that use GNUstep.  The Foundation.h header file
 * includes all the other standard header files you need.
 */
#include <Foundation/Foundation.h>

/*
 * Declare the Test class that implements the class method (classStringValue).
 */
@interface Test
+ (const char *) classStringValue;
@end

/*
 * Define the Test class and the class method (classStringValue).
 */
@implementation Test
+ (const char *) classStringValue;
{
  return "This is the string value of the Test class";
}
@end

/*
 * The main() function: pass a message to the Test class
 * and print the returned string.
 */
int main(void)
{
  printf("%s\n", [Test classStringValue]);
  return 0;
}
```

The text between comment markers (/* */) is ignored by the compiler but indicates to someone reading the source file what each part of the program does. The program is an example of a (class) method responding to a message. Can you see how it works?

A message is sent to the `Test` class as an argument to `printf()`, requesting the string value of that class. The `Test` class performs its `classStringValue` method in response to this message and returns a string that is finally printed. No object is created in this program since a class method does not require an instance of a class in order to respond to a message.

You will learn more about class methods in the next chapter.

3. Now create the makefile, again using your favourite text editor, and save it in the same project directory with the filename `GNUmakefile`.

```
include $(GNUSTEP_MAKEFILES)/common.make

TOOL_NAME = LogTest
LogTest_OBJC_FILES = source.m

include $(GNUSTEP_MAKEFILES)/tool.make
```

If you look at the makefile above you will notice the two lines that tell the make utility to build a tool with the filename `LogTest` from the Objective-C source file `source.m`. You could copy and modify this makefile for later projects you may have: just change the tool name and list the new source files.

The two 'include' lines are just a way of keeping your makefile simple, by including two 'ready-made' makefiles that someone else created.

4. Before you can execute this makefile you must first set your GNUstep environment variables. Among other things this defines the `GNUSTEP_MAKEFILES` variable referenced above. The simplest way to do this is to execute one of the following commands (you must first locate your GNUstep installation manually):

   C shell:

   `source <GNUstep root>/System/Library/Makefiles/GNUstep.csh`

   Bourne shell:

   `. <GNUstep root>/System/Library/Makefiles/GNUstep.sh`

   On most Unix systems, GNUstep is installed in `/usr/lib/GNUstep`. (Directory layout documentation (`../../../User/GNUstep/filesystem_toc.html`).)

5. You can now compile the project using make. At the system command prompt, change to the project directory and enter the **make** command.

6. Run the program (on Unix enter **./obj/LogTest** at the command prompt). The message "This is the string value of the Test class" will be displayed (assuming there were no errors).

You have now compiled and run your first Objective-C program. Hungry for more? Then read on.

2 The Objective-C Language

In the previous chapter you were introduced to some basic object-oriented programming terms. This chapter will expand on these terms, and introduce you to some new ones, while concentrating on how they apply to the Objective-C language and the GNUstep base library. First let us look at some non OO additions that Objective-C makes to ANSI C.

2.1 Non OO Additions

Objective-C makes a few non OO additions to the syntax of the C programming language that include:

- A boolean data type (`BOOL`) capable of storing either of the values `YES` or `NO`.
  A `BOOL` is a scalar value and can be used like the familiar `int` and `char` data types.
  A `BOOL` value of `NO` is zero, while `YES` is non-zero.
- The use of a pair of slashes (`//`) to mark text up to the end of the line as a comment.
- The `#import` preprocessor directive was added; it directs the compiler to include a file only if it has not previously been included for the current compilation. This directive should only be used for Objective-C headers and not ordinary C headers, since the latter may actually rely on being included more than once in certain cases to support their functionality.

2.2 Objects

Object-oriented (OO) programming is based on the notion that a software system can be composed of objects that interact with each other in a manner that parallels the interaction of objects in the physical world.

This model makes it easier for the programmer to understand how software works since it makes programming more intuitive. The use of objects also makes it easier during program design: take a big problem and consider it in small pieces, the individual objects, and how they relate to each other.

Objects are like mini programs that can function on their own when requested by the program or even another object. An object can receive messages and then act on these messages to alter the state of itself (the size and position of a rectangle object in a drawing program for example).

In software an object consists of instance variables (data) that represent the state of the object, and methods (like C functions) that act on these variables in response to messages.

As a programmer creating an application or tool, all you need do is send messages to the appropriate objects rather than call functions that manipulate data as you would with a procedural program.

The syntax for sending a message to an object, as shown below, is one of the additions that Objective-C adds to ANSI C.

```
[objectName message];
```

Note the use of the square [ ] brackets surrounding the name of the object and message.

Rather than 'calling' one of its methods, an object is said to 'perform' one of its methods in response to a message. The format that a message can take is discussed later in this section.

2.2.1 Id and nil

Objective-C defines a new type to identify an object: `id`, a type that points to an object's data (its instance variables). The following code declares the variable '`button`' as an object (as opposed to '`button`' being declared an integer, character or some other data type).

```
id button;
```

When the button object is eventually created the variable name '`button`' will point to the object's data, but before it is created the variable could be assigned a special value to indicate to other code that the object does not yet exist.

Objective-C defines a new keyword `nil` for this assignment, where `nil` is of type `id` with an unassigned value. In the button example, the assignment could look like this:

```
id button = nil;
```

which assigns `nil` in the declaration of the variable.

You can then test the value of an object to determine whether the object exists, perhaps before sending the object a message. If the test fails, then the object does not exist and your code can execute an alternative statement.

```
if (anObject != nil)
  ... /* send message */
else
  ... /* do something else */
```

The header file `objc/objc.h` defines `id`, `nil`, and other basic types of the Objective-C language. It is automatically included in your source code when you use the compiler directive `#include <Foundation/Foundation.h>` to include the GNUstep Base class definitions.

2.2.2 Messages

A message in Objective-C is the mechanism by which you pass instructions to objects. You may tell the object to do something for you, tell it to change its internal state, or ask it for information.

A message usually invokes a method, causing the receiving object to respond in some way. Objects and data are manipulated by sending messages to them. Like C-functions they have return types, but function specific to the object.

Objects respond to messages that make specific requests. Message expressions are enclosed in square brackets and include the receiver or object name and the message or method name along with any arguments.

To send a message to an object, use the syntax:

```
[receiver messagename];
```

where `receiver` is the object.

The run-time system invokes object methods that are specified by messages. For example, to invoke the display method of the mySquare object the following message is used:

```
[mySquare display];
```

Messages may include arguments that are prefixed by colons, in which case the colons are part of the message name, so the following message is used to invoke the `setFrameOrigin::` method:

```
[button setFrameOrigin: 10.0 : 10.0];
```

Labels describing arguments precede colons:

```
[button setWidth: 20.0 height: 122.0];
```

invokes the method named `setWidth:height:`

Messages that take a variable number of arguments are of the form:

```
[receiver makeList: list, argOne, argTwo, argThree];
```

A message to `nil` does NOT crash the application (while in Java messages to `null` raise exceptions); the Objective-C application does nothing.

For example:

```
[nil display];
```

will do nothing.

If a message to `nil` is supposed to return an object, it will return `nil`. But if the method is supposed to return a primitive type such as an `int`, then the return value of that method when invoked on `nil`, is undefined. The programmer therefore needs to avoid using the return value in this instance.

2.2.3 Polymorphism

Polymorphism refers to the fact that two different objects may respond differently to the same message. For example when client objects receive an alike message from a server object, they may respond differently. Using Dynamic Binding, the run-time system determines which code to execute according to the object type.

2.3 Classes

A **class** in Objective-C is a *type of object*, much like a structure definition in C except that in addition to variables, a class has code – method implementations – associated with it. When you create an *instance* of a class, also known as an *object*, memory for each of its variables is allocated, including a pointer to the class definition itself, which tells the Objective-C runtime where to find the method code, among other things. Whenever an object is sent a message, the runtime finds this code and executes it, using the variable values that are set for this object.

2.3.1 Inheritance

Most of the programmer's time is spent defining classes. Inheritance helps reduce coding time by providing a convenient way of reusing code. For example, the `NSButton` class defines data (or instance variables) and methods to create button objects of a certain type, so a subclass of `NSButton` could be produced to create buttons of another type - which may perhaps have a different border colour. Equally `NSTextField` can be used to define a subclass that perhaps draws a different border, by reusing definitions and data in the superclass.

Inheritance places all classes in a logical hierarchy or tree structure that may have the `NSObject` class at its root. (The root object may be changed by the developer; in GNUstep

it is `NSObject`, but in "plain" Objective-C it is a class called "`Object`" supplied with the runtime.) All classes may have subclasses, and all except the root class do have superclasses. When a class object creates a new instance, the new object holds the data for its class, superclass, and superclasses extending to the root class (typically `NSObject`). Additional data may be added to classes so as to provide specific functions and application logic.

When a new object is created, it is allocated memory space and its data in the form of its instance variables are initialised. Every object has at least one instance variable (inherited from `NSObject`) called `isa`, which is initialized to refer to the object's class. Through this reference, access is also afforded to classes in the object's inheritance path.

In terms of source code, an Objective-C class definition has an:

- *interface* declaring instance variables, methods and the superclass name; and an
- *implementation* that defines the class in terms of operational code that implements the methods.

Typically these entities are confined to separate files with `.h` and `.m` extensions for Interface and Implementation files, respectively. However they may be merged into one file, and a single file may implement multiple classes.

2.3.2 Inheritance of Methods

Each new class inherits methods and instance variables from another class. This results in a class hierarchy with the root class at the core, and every class (except the root) has a superclass as its parent, and all classes may have numerous subclasses as their children. Each class therefore is a refinement of its superclass(es).

2.3.3 Overriding Methods

Objects may access methods defined for their class, superclass, superclass' superclass, extending to the root class. Classes may be defined with methods that overwrite their namesakes in ancestor classes. These new methods are then inherited by subclasses, but other methods in the new class can locate the overridden methods. Additionally redefined methods may include overridden methods.

2.3.4 Abstract Classes

Abstract classes or abstract superclasses such as `NSObject` define methods and instance variables used by multiple subclasses. Their purpose is to reduce the development effort required to create subclasses and application structures. When we get technical, we make a distinction between a pure abstract class whose methods are defined but instance variables are not, and a semi-abstract class where instance variables are defined).

An abstract class is not expected to actually produce functional instances since crucial parts of the code are expected to be provided by subclasses. In practice, abstract classes may either stub out key methods with no-op implementations, or leave them unimplemented entirely. In the latter case, the compiler will produce a warning (but not an error).

Abstract classes reduce the development effort required to create subclasses and application structures.

2.3.5 Class Clusters

A class cluster is an abstract base class, and a group of private, concrete subclasses. It is used to hide implementation details from the programmer (who is only allowed to use the interface provided by the abstract class), so that the actual design can be modified (probably optimised) at a later date, without breaking any code that uses the cluster.

Consider a scenario where it is necessary to create a class hierarchy to define objects holding different types including **chars**, **ints**, **shorts**, **longs**, **floats** and **doubles**. Of course, different types could be defined in the same class since it is possible to **cast** or **change** them from one to the next. Their allocated storage differs, however, so it would be inefficient to bundle them in the same class and to convert them in this way.

The solution to this problem is to use a class cluster: define an abstract superclass that specifies and declares components for subclasses, but does not declare instance variables. Rather this declaration is left to its subclasses, which share the **programmatic interface** that is declared by the abstract superclass.

When you create an object using a cluster interface, you are given an object of another class - from a concrete class in the cluster.

2.4 NSObject: The Root Class

In GNUstep, `NSObject` is a root class that provides a base implementation for all objects, their interactions, and their integration in the run-time system. `NSObject` defines the `isa` instance variable that connects every object with its class.

In other Objective-C environments besides GNUstep, `NSObject` will be replaced by a different class. In many cases this will be a default class provided with the Objective-C runtime. In the GNU runtime for example, the base class is called `Object`. Usually base classes define a similar set of methods to what is described here for `NSObject`, however there are variations.

The most basic functions associated with the `NSObject` class (and inherited by all subclasses) are the following:

- allocate instances
- connect instances to their classes

In addition, `NSObject` supports the following functionality:

- initialize instances
- deallocate instances
- compare self with another object
- archive self
- perform methods selected at run-time
- provide reflective information at runtime to queries about declared methods
- provide reflective information at runtime to queries about position in the inheritance hierarchy
- forward messages to other objects.

2.4.1 The NSObject Protocol

In fact, the `NSObject` class is a bit more complicated than just described. In reality, its method declarations are split into two components: essential and ancillary. The essential methods are those that are needed by *any* root class in the GNUstep/Objective-C environment. They are declared in an "`NSObject` protocol" which should be implemented by any other root class you define (see Chapter 4 [Protocols], page 31). The ancillary methods are those specific to the `NSObject` class itself but need not be implemented by any other root class. It is not important to know which methods are of which type unless you actually intend to write an alternative root class, something that is rarely done.

2.5 Static Typing

Recall that the `id` type may be used to refer to any class of object. While this provides for great runtime flexibility (so that, for example, a generic `List` class may contain objcts of any instance), it prevents the compiler from checking whether objects implement the messages you send them. To allow type checking to take place, Objective-C therefore also allows you to use class names as variable types in code. In the following example, type checking verifies that the `myString` object is an appropriate type.

```
 // compiler verifies, if anObject's type is known, that it is an NSString:
NSString *myString = anObject;
 // now, compiler verifies that NSString declares an int 'length' method:
int len = [myString length];
```

Note that objects are declared as pointers, unlike when `id` is used. This is because the pointer operator is implicit for `id`. Also, when the compiler performs type checking, a subclass is always permissible where any ancestor class is expected, but not vice-versa.

2.5.1 Type Introspection

Static typing is not always appropriate. For example, you may wish to store objects of multiple types within a list or other container structure. In these situations, you can still perform type-checking manually if you need to send an untyped object a particular message. The `isMemberOfClass:` method defined in the `NSObject` class verifies that the receiver is of a specific class:

```
if ([namedObject isMemberOfClass: specificClass] == YES)
  {
    // code here
  }
```

The test will return false if the object is a member of a subclass of the specific class given - an exact match is required. If you are merely interested in whether a given object *descends* from a particular class, the `isKindOfClass:` method can be used instead:

```
if ([namedObject isKindOfClass: specificClass] == YES)
  {
    // code here
  }
```

There are other ways of determining whether an object responds to a particular method, as will be discussed in Chapter 5 [Advanced Messaging], page 49.

2.5.2 Referring to Instance Variables

As you will see later, classes may define some or all of their instance variables to be *public* if they wish. This means that any other object or code block can access them using the standard "`->`" structure access operator from C. For this to work, the object must be statically typed (not referred to by an `id` variable).

```
Bar *bar = [foo getBar];
int c = bar->value * 2;   // 'value' is an instance variable
```

In general, direct instance variable access from outside of a class is not recommended programming practice, aside from in exceptional cases where performance is at a premium. Instead, you should define special methods called *accessors* that provide the ability to retrieve or set instance variables if necessary:

```
- (int) value
{
    return value;
}

- (void) setValue: (int) newValue
{
    value = newValue;
}
```

While it is not shown here, accessors may perform arbitrary operations before returning or setting internal variable values, and there need not even be a direct correspondence between the two. Using accessor methods consistently allows this to take place when necessary for implementation reasons without external code being aware of it. This property of *encapsulation* makes large code bases easier to maintain.

2.6 Working with Class Objects

Classes themselves are maintained internally as objects in their own right in Objective-C, however they do not possess the instance variables defined by the classes they represent, and they cannot be created or destroyed by user code. They do respond to class methods, as in the following:

```
id result = [SomeClassName doSomething];
```

Classes respond to the class methods their class defines, as well as those defined by their superclasses. However, it is not allowed to override an inherited class method.

You may obtain the class object corresponding to an instance object at runtime by a method call; the class object is an instance of the "`Class`" class.

```
  // all of these assign the same value
id stringClass1 = [stringObject class];
Class stringClass2 = [stringObject class];
id stringClass3 = [NSString class];
```

Classes may also define a version number (by overriding that defined in `NSObject`):

```
int versionNumber = [NSString version];
```

This facility allows developers to access the benefits of versioning for classes if they so choose.

2.6.1 Locating Classes Dynamically

Class names are about the only names with global visibility in Objective-C. If a class name is unknown at compilation but is available as a string at run time, the GNUstep library `NSClassFromString` function may be used to return the class object:

```
if ([anObject isKindOf: NSClassFromString("SomeClassName")] == YES)
  {
    // do something ...
  }
```

The function returns `Nil` if it is passed a string holding an invalid class name. Class names, global variables and functions (but not methods) exist in the same name space, so no two of these entities may share the same name.

2.7 Naming Constraints and Conventions

The following lists the full uniqueness constraints on names in Objective-C.

- Neither gGlobal variables nor function names may share the same name as classes, because all three entities are allocated the same (global) name space.
- A class may define methods using the same names as those held in other classes. (See Chapter 2 [Overriding Methods], page 11, above.)
- A class may define instance variables using the same names as those held in other classes.
- A class category may have the same name as another class category.
- An instance method and a class method may share the same name.
- A protocol may have the same name as a class, category, or any other entity.
- A method and an instance variable may share the same name.

There are also a number of conventions used in practice. These help to make code more readable and also help avoid naming conflicts. Conventions are particularly important since Objective-C does not have any namespace partitioning facilities like Java or other languages.

- Class, category and protocol names begin with an uppercase letter.
- Methods, instance variables, and variables holding instances begin with a lowercase letter.
- Second and subsequent words in a name should begin with a capital letter, as in "ThisIsALongName", not "Thisisalongname". As can be seen, this makes long names more readable.
- Classes intended to be used as libraries (Frameworks, in NeXTstep parlance) should utilize a unique two or three letter prefix. For example, the Foundation classes all begin with 'NS', as in "NSArray, and classes in the OmniFoundation from Omni Group (a popular library for OpenStep) began with "OF".
- Classes and methods intended to be used only be the developers maintaining them should be prefixed by an underscore, as in "_SomePrivateClass" or "_somePrivateMethod". Capitalization rules should still be followed.
- Functions intended for global use should beging with a capital letter, and use prefixing conventions as for classes.

2.8 Strings in GNUstep

Strings in GNUstep can be handled in one of two ways. The first way is the C approach of using an array of `char`. In this case you may use the "`STR`" type defined in Objective-C in place of `char[]`.

The second approach is to rely on the `NSString` class and associated subclasses in the GNUstep Base library, and compiler support for them. Using this approach allows use of the methods in the NSString API (`../Reference/NSString.html`). In addition, the `NSString` class provides the means to initialize strings using printf-like formats.

The `NSString` class defines objects holding raw **Unicode** character streams or **strings**. Unicode is a 16-bit worldwide standard used to define character sets for all spoken languages. In GNUstep parlance the Unicode character is of type **unichar**.

2.8.1 Creating NSString Static Instances

A **static** instance is allocated at compile time. The creation of a static instance of `NSString` is achieved using the `@"..."` construct and a pointer:

```
NSString *w = @"Brainstorm";
```

Here, `w` is a variable that refers to an `NSString` object representing the ASCII string "Brainstorm".

2.8.2 NSString +stringWithFormat:

The class method `stringWithFormat:` may also be used to create instances of `NSString`, and broadly echoes the `printf()` function in the C programming language. `stringWithFormat:` accepts a list of arguments whose processed result is placed in an `NSString` that becomes a return value as illustrated below:

```
int qos = 5;
NSString *gprsChannel;

gprschannel = [NSString stringWithFormat: @"The GPRS channel is %d",
                qos];
```

The example will produce an `NSString` called `gprsChannel` holding the string "The GPRS channel is 5".

`stringWithFormat:` recognises the `%@` conversion specification that is used to specify an additional `NSString`:

```
NSString *one;
NSString *two;

one = @"Brainstorm";
two = [NSString stringWithFormat: @"Our trading name is %@", one];
```

The example assigns the variable `two` the string "Our trading name is Brainstorm." The `%@` specification can be used to output an object's description - as returned by the `NSObject -description` method), which is useful when debugging, as in:

```
NSObject *obj = [anObject aMethod];

NSLog (@"The method returned: %@", obj);
```

2.8.3 C String Conversion

When a program needs to call a C library function it is useful to convert between NSStrings and standard ASCII C strings (not fixed at compile time). To create an NSString using the contents of the returned C string (from the above example), use the NSString class method stringWithCString::

```
char *function (void);

char *result;
NSString *string;

result = function ();
string = [NSString stringWithCString: result];
```

To convert an NSString to a standard C ASCII string, use the cString method of the NSString class:

```
char *result;
NSString *string;

string = @"Hi!";
result = [string cString];
```

2.8.4 NSMutableString

NSStrings are immutable objects; meaning that once they are created, they cannot be modified. This results in optimised NSString code. To modify a string, use the subclass of NSString, called NSMutableString. Use a NSMutableString wherever a NSString could be used.

An NSMutableString responds to methods that modify the string directly - which is not possible with a generic NSString. To create a NSMutableStringuse stringWithFormat::

```
NSString *name = @"Brainstorm";
NSMutableString *str;
str = [NSMutableString stringWithFormat: @"Hi!, %@", name];
```

While NSString's implementation of stringWithFormat: returns a NSString, NSMutableString's implementation returns an NSMutableString.

Note. Static strings created with the @"..." construct are always immutable.

NSMutableStrings are rarely used because to modify a string, you normally create a new string derived from an existing one.

A useful method of the NSMutableString class is appendString:, which takes an NSString argument, and appends it to the receiver:

```
NSString *name = @"Brainstorm";
NSString *greeting = @"Hello";
NSMutableString *s;

s = AUTORELEASE ([NSMutableString new]);
[s appendString: greeting];
[s appendString: @", "];
```

```
[s appendString: name];
```

This code produces the same result as:

```
NSString *name = @"Brainstorm";
NSString *greeting = @"Hello";
NSMutableString *s;

s = [NSMutableString stringWithFormat: @"%@, %@", greeting, name];
```

2.8.5 Loading and Saving Strings

The the GNUstep Base library has numerous string manipulation features, and among the most notable are those relating to writing/reading strings to/from files. To write the contents of a string to a file, use the `writeToFile:atomically:` method:

```
#include <Foundation/Foundation.h>

int
main (void)
{
  CREATE_AUTORELEASE_POOL(pool);
  NSString *name = @"This string was created by GNUstep";

  if ([name writeToFile: @"/home/nico/testing" atomically: YES])
    {
      NSLog (@"Success");
    }
  else
    {
      NSLog (@"Failure");
    }
  RELEASE(pool);
  return 0;
}
```

`writeToFile:atomically:` returns YES for success, and NO for failure. If the atomically flag is YES, then the library first writes the string into a file with a temporary name, and, when the writing has been successfully done, renames the file to the specified filename. This prevents erasing the previous version of filename unless writing has been successful. This is a useful feature, which should be enabled.

To read the contents of a file into a string, use `stringWithContentsOfFile:`, as shown in the following example that reads `@"/home/Brainstorm/test"`:

```
#include <Foundation/Foundation.h>

int
main (void)
{
  CREATE_AUTORELEASE_POOL(pool);
  NSString *string;
```

```
  NSString *filename = @"/home/nico/test";

  string = [NSString stringWithContentsOfFile: filename];
  if (string == nil)
    {
      NSLog (@"Problem reading file %@", filename);
      /*
       * <missing code: do something to manage the error...>
       * <exit perhaps ?>
       */
    }

  /*
   * <missing code: do something with string...>
   */

  RELEASE(pool);
  return 0;
}
```

3 Working with Objects

Objective-C and GNUstep provide a rich object allocation and memory management framework. Objective-C affords independent memory allocation and initialization steps for objects, and GNUstep supports three alternative schemes for memory management.

3.1 Initializing and Allocating Objects

Unlike most object-oriented languages, Objective-C exposes memory allocation for objects and initialization as two separate steps. In particular, every class provides an '`+alloc`' method for creating blank new instances. However, initialization is carried out by an instance method, not a class method. By convention, the default initialization method is '`-init`'. The general procedure for obtaining a newly initialized object is thus:

```
id newObj = [[SomeClass alloc] init];
```

Here, the call to `alloc` returns an uninitialized instance, on which `init` is then invoked. (Actually, `alloc` *does* set all instance variable memory to 0, and it initializes the special `isa` variable mentioned earlier which points to the object's class, allowing it to respond to messages.) The `alloc` and `init` calls may be collapsed for convenience into a single call:

```
id newObj = [SomeClass new];
```

The default implementation of `new` simply calls `alloc` and `init` as above, however other actions are possible. For example, `new` could be overridden to reuse an existing object and just call `init` on it (skipping the `alloc` step). (Technically this kind of instantiation management can be done inside `init` as well – it can deallocate the receiving object and return another one in its place. However this practice is not recommended; the `new` method should be used for this instead since it avoids unnecessary memory allocation for instances that are not used.)

3.1.1 Initialization with Arguments

In many cases you want to initialize an object with some specific information. For example a `Point` object might need to be given an *x, y* position. In this case the class may define additional initializers, such as:

```
id pt = [[Point alloc] initWithX: 1.5 Y: 2.0];
```

Again, a `new` method may be defined, though sometimes the word "new" is not used in the name:

```
id pt = [Point newWithX: 1.5 Y: 2.0];
  // alternative
id pt = [Point pointAtX: 1.5 Y: 2.0];
```

In general the convention in Objective-C is to name initializers in a way that is intuitive for their classes. Initialization is covered in more detail in Chapter 4 [Instance Initialization], page 31. Finally, it is acceptable for an `init...` method to return `nil` at times when insufficient memory is available or it is passed an invalid argument; for example the argument to the `NSString` method `initWithContentsOfFile:` may be an erroneous file name.

3.1.2 Memory Allocation and Zones

Memory allocation for objects in GNUstep supports the ability to specify that memory is to be taken from a particular region of addressable memory. In the days that computer RAM was relatively limited, it was important to be able to ensure that parts of a large application that needed to interact with one another could be held in working memory at the same time, rather than swapping back and forth from disk. This could be done by specifying that particular objects were to be allocated from a particular region of memory, rather than scattered across all of memory at the whim of the operating system. The OS would then keep these objects in memory at one time, and swap them out at the same time, perhaps to make way for a separate portion of the application that operated mostly independently. (Think of a word processor that keeps structures for postscript generation for printing separate from those for managing widgets in the onscreen editor.)

With the growth of computer RAM and the increasing sophistication of memory management by operating systems, it is not as important these days to control the regions where memory is allocated from, however it may be of use in certain situations. For example, you may wish to save time by allocating memory in large chunks, then cutting off pieces yourself for object allocation. If you know you are going to be allocating large numbers of objects of a certain size, it may pay to create a zone that allocates memory in multiples of this size. The GNUstep/Objective-C mechanisms supporting memory allocation are therefore described here.

The fundamental structure describing a region of memory in GNUstep is called a *Zone*, and it is represented by the `NSZone` struct. All `NSObject` methods dealing with the allocation of memory optionally take an `NSZone` argument specifying the Zone to get the memory from. For example, in addition to the fundamental `alloc` method described above, there is the `allocWithZone:` method:

```
+ (id) alloc;
+ (id) allocWithZone: (NSZone*)zone;
```

Both methods will allocate memory to hold an object, however the first one automatically takes the memory from a default Zone (which is returned by the `NSDefaultMallocZone()` function). When it is necessary to group objects in the same area of memory, or allocate in chunks - perhaps for performance reasons, you may create a Zone from where you would allocate those objects by using the `NSCreateZone` function. This will minimise the paging required by your application when accessing those objects frequently. In all normal yuse however, you should confine yourself to the default zone.

Low level memory allocation is performed by the `NSAllocateObject()` function. This is rarely used but available when you require more advanced control or performance. This function is called by `[NSObject +allocWithZone:]`. However, if you call `NSAllocateObject()` directly to create an instance of a class you did not write, you may break some functionality of that class, such as caching of frequently used objects.

Other `NSObject` methods besides `alloc` that may optionally take Zones include `-copy` and `-mutableCopy`. For 95% of cases you will probably not need to worry about Zones at all; unless performance is critical, you can just use the methods without zone arguments, that take the default zone.

3.1.3 Memory Deallocation

Objects should be deallocated from memory when they are no longer needed. While there are several alternative schemes for managing this process (see next section), they all eventually resort to calling the `NSObject` method `-dealloc`, which is more or less the opposite of `-alloc`. It returns the memory occupied by the object to the Zone from which it was originally allocated. The `NSObject` implementation of the method deallocates only instance variables. Additional allocated, unshared memory used by the object must be deallocated separately. Other entities that depend solely on the deallocated receiver, including complete objects, must also be deallocated separately. Usually this is done by subclasses overriding `-dealloc` (see Chapter 4 [Instance Deallocation], page 31).

As with `alloc`, the underlying implementation utilizes a function (`NSDeallocateObject()`) that can be used by your code if you know what you are doing.

3.2 Memory Management

In an object-oriented environment, ensuring that all memory is freed when it is no longer needed can be a challenge. To assist in this regard, there are three alternative forms of memory management available in Objective-C:

– Explicit
  You allocate objects using `alloc`, `copy` etc, and deallocate them when you have finished with them (using `dealloc`). This gives you complete control over memory management, and is highly efficient, but error prone.

– Retain count
  You use the OpenStep retain/release mechanism, along with autorelease pools which provide a degree of automated memory management. This gives a good degree of control over memory management, but requires some care in following simple rules. It's pretty efficient.

The recommended approach is to use some standard macros defined in `NSObject.h` which encapsulate the retain/release/autorelease mechanism, but which permit efficient use of Automated reference Counts (ARC) if you build your software with that. We will justify this recommendation after describing the three alternatives in greater detail.

3.2.1 Explicit Memory Management

This is the standard route to memory management taken in C and C++ programs. As in standard C when using `malloc`, or in C++ when using `new` and `delete`, you need to keep track of every object created through an `alloc` call and destroy it by use of `dealloc` when it is no longer needed. You must make sure to no longer reference deallocated objects; although messaging them will not cause a segmentation fault as in C/C++, it will still lead to your program behaving in unintended ways.

This approach is generally *not* recommended since the Retain/Release style of memory management is significantly less leak-prone while still being quite efficient.

3.2.2 OpenStep-Style (Retain/Release) Memory Management

The standard OpenStep system of memory management employs retain counts. When an object is created, it has a retain count of 1. When an object is retained, the retain count

is incremented. When it is released the retain count is decremented, and when the retain count goes to zero the object gets deallocated.

```
Coin *c = [[Coin alloc] initWithValue: 10];

  // Put coin in pouch,
[c retain]; // Calls 'retain' method (retain count now 2)
  // Remove coin from pouch
[c release]; // Calls 'release' method (retain count now 1)
  // Drop in bottomless well
[c release]; // Calls 'release' ... (retain count 0) then 'dealloc'
```

One way of thinking about the initial retain count of 1 on the object is that a call to `alloc` (or `copy`) implicitly calls `retain` as well. There are a couple of default conventions about how `retain` and `release` are to be used in practice.

- *If a block of code causes an object to be allocated, it "owns" this object and is responsible for releasing it. If a block of code merely receives a created object from elsewhere, it is* **not** *responsible for releasing it.*
- *More generally, the total number of* `retain`*s in a block should be matched by an equal number of* `release`*s.*

Thus, a typical usage pattern is:

```
NSString *msg = [[NSString alloc] initWithString: @"Test message."];
NSLog(msg);
  // we created msg with alloc -- release it
[msg release];
```

Retain and release must also be used for instance variables that are objects:

```
- (void)setFoo:(FooClass *newFoo)
{
    // first, assert reference to newFoo
  [newFoo retain];
    // now release reference to foo (do second since maybe newFoo == foo)
  [foo release];
    // finally make the new assignment; old foo was released and may
    // be destroyed if retain count has reached 0
  foo = newFoo;
}
```

Because of this retain/release management, it is safest to use accessor methods to set variables even within a class:

```
- (void)resetFoo
{
  FooClass *foo = [[FooClass alloc] init];
  [self setFoo: foo];
    // since -setFoo just retained, we can and should
    // undo the retain done by alloc
  [foo release];
}
```

Exceptions

In practice, the extra method call overhead should be avoided in performance critical areas and the instance variable should be set directly. However in all other cases it has proven less error-prone in practice to consistently use the accessor.

There are certain situations in which the rule of having retains and releases be equal in a block should be violated. For example, the standard implementation of a container class `retain`s each object that is added to it, and `release`s it when it is removed, in a separate method. In general you need to be careful in these cases that retains and releases match.

3.2.2.1 Autorelease Pools

One important case where the retain/release system has difficulties is when an object needs to be transferred or handed off to another. You don't want to retain the transferred object in the transferring code, but neither do you want the object to be destroyed before the handoff can take place. The OpenStep/GNUstep solution to this is the *autorelease pool*. An autorelease pool is a special mechanism that will retain objects it is given for a limited time – always enough for a transfer to take place. This mechanism is accessed by calling `autorelease` on an object instead of `release`. `Autorelease` first adds the object to the active autorelease pool, which retains it, then sends a `release` to the object. At some point later on, the pool will send the object a second `release` message, but by this time the object will presumably either have been retained by some other code, or is no longer needed and can thus be deallocated. For example:

```
- (NSString *) getStatus
{
  NSString *status =
    [[NSString alloc] initWithFormat: "Count is %d", [self getCount]];
   // set to be released sometime in the future
  [status autorelease];
  return status;
}
```

Any block of code that calls `-getStatus` can also forego retaining the return value if it just needs to use it locally. If the return value is to be stored and used later on however, it should be retained:

```
...
NSString *status = [foo getStatus];
  // 'status' is still being retained by the autorelease pool
NSLog(status);
return;
  // status will be released automatically later

...
currentStatus = [foo getStatus];
  // currentStatus is an instance variable; we do not want its value
  // to be destroyed when the autorelease pool cleans up, so we
  // retain it ourselves
[currentStatus retain];
```

Convenience Constructors

A special case of object transfer occurs when a *convenience* constructor is called (instead of `alloc` followed by `init`) to create an object. (Convenience constructors are class methods that create a new instance and do not start with "new".) In this case, since the convenience method is the one calling `alloc`, it is responsible for releasing it, and it does so by calling `autorelease` before returning. Thus, if you receive an object created by any convenience method, it is autoreleased, so you don't need to release it if you are just using it temporarily, and you DO need to retain it if you want to hold onto it for a while.

```
- (NSString *) getStatus
{
    NSString *status =
        [NSString stringWithFormat: "Count is %d", [self getCount]];
    // 'status' has been autoreleased already
    return status;
}
```

Pool Management

An autorelease pool is created automatically if you are using the GNUstep GUI classes, however if you are just using the GNUstep Base classes for a nongraphical application, you must create and release autorelease pools yourself:

```
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
```

Once a pool has been created, any autorelease calls will automatically find it. To close out a pool, releasing all of its objects, simply release the pool itself:

```
[pool release];
```

To achieve finer control over autorelease behavior you may also create additional pools and release them in a nested manner. Calls to `autorelease` will always use the most recently created pool.

Finally, note that `autorelease` calls are significantly slower than plain `release`. Therefore you should only use them when they are necessary.

3.2.2.2 Avoiding Retain Cycles

One difficulty that sometimes occurs with the retain/release system is that cycles can arise in which, essentially, Object A has retained Object B, and Object B has also retained Object A. In this situation, neither A nor B will ever be deallocated, even if they become completely disconnected from the rest of the program. In practice this type of situation may involve more than two objects and multiple retain links. The only way to avoid such cycles is to be careful with your designs. If you notice a situation where a retain cycle could arise, remove at least one of the links in the chain, but not in such a way that references to deallocated objects might be mistakenly used.

3.2.2.3 Summary

The following summarizes the retain/release-related methods:

Method	Description
`-retain`	increases the reference count of an object by 1
`-release`	decreases the reference count of an object by 1
`-autorelease`	decreases the reference count of an object by 1 at some stage in the future

`+alloc` and `+allocWithZone:`	allocates memory for an object, and returns it with retain count of 1
`-copy`, `-mutableCopy`, `copyWithZone:` and `-mutableCopyWithZone:`	makes a copy of an object, and returns it with retain count of 1
`-init` and any method whose name begins with `init`	initialises the receiver, returning the retain count unchanged. `-init` has had no effect on the reference count.
`-new` and any method whose name begins with `new`	allocates memory for an object, initialises it, and returns the result.
`-dealloc`	deallocates object immediately (regardless of value of retain count)
convenience constructors	allocate memory for an object, and returns it in an autoreleased state (retain=1, but will be released automatically at some stage in the future). These constructors are class methods whose name generally begins with the name of the class (initial letter converted to lowercase).

The following are the main conventions you need to remember:

- If a unit of code allocates, retains, or copies an object, the same unit, loosely speaking, is responsible for releasing or autoreleasing it at some future point. It is best to balance retains and releases within each individual block of code.
- If you receive an object, it should remain valid until the object which provided it is sent another message or until the autorelease pool which was in use at the point when you received it is emptied. So you can usually expect it to remain valid for the rest of the current method call and can even return it as the result of the method. If you need to store it away for future use (e.g. as an instance variable, or to use after emptying/destroying an autorelease pool, or to be used after sending another message to the object's owner), you must retain it.
- The retain counts mentioned are guidelines only ... more sophisticated classes often perform caching and other tricks, so that `+alloc` methods may retain an instance from a cache and return it, and `-init` methods may release their receiver and return a different object (possibly obtained by retaining a cached object). In these cases, the retain counts of the returned objects will obviously differ from the simple examples, but the ownership rules (how you should use the returned values) remain the same.

3.2.3 ObjC-2 and Automated Reference Counting

When using a compiler and runtime supporting ObjC-2 and ARC, the reference counting for objects is handled by the compiler. To enable easy development (to ObjC-1) code, a number of macros are provided which encapsulate the manual reference counting required when ARC is not available.

Macro	Functionality
`RETAIN(foo);`	`[foo retain];`

`RELEASE(foo);`	`[foo release];`
`AUTORELEASE(foo);`	`[foo autorelease];`
`ASSIGN(foo, bar);`	`[bar retain]; [foo release]; foo = bar;`
`ASSIGNCOPY(foo, bar);`	`[foo release]; foo = [bar copy];`
`ASSIGNMUTABLECOPY(foo, bar);`	`[foo release]; foo = [bar mutableCopy];`
`DESTROY(foo);`	`[foo release]; foo = nil;`
`ENTER_POOL`	Like @autoreleasepool this introduces a code block with a manually created autorelease pool.
`LEAVE_POOL`	This ends a block created by an ENTER_POOL destroying the autorelease pool created at the start of the block
`DEALLOC`	`[super dealloc];`
`IF_NO_ARC(...)`	Code to be compiled only if ARC is not in use

In the assignment "convenience" macros, appropriate `nil` checks are made so that no retain/release messages are sent to `nil`.

4 Writing New Classes

Objective-C class definitions are always divided into two parts: an *interface* and an *implementation*. This division mirrors the common C library division into a header file with function declarations, which is distributed to all users of a library, and a source file with the implementations, which is only used to compile the library and is generally not distributed to users. A class interface declares instance variables, methods and the superclass name, while the implementation file holds the operational code that implements those methods. Typically the interface and implementation are held in separate files, using the `.h` and `.m` extensions, respectively. They may, however, be merged into one file, and a single file may implement many classes.

4.1 Interface

The interface is included in the source using `#include`:

```
#include "SomeClass.h"
```

To ensure that a Header file is included only once, it is usual to protect it with pre-compiler defines:

```
#ifndef _MY_CLASS_H_INCLUDED
#define _MY_CLASS_H_INCLUDED

/* HEADER FILE */
#endif
```

This is the standard C technique to protect header files from being included more than once. A cleaner alternative, introduced in Objective-C, is to use the `#import` directive instead of `#include`. The compiler will automatically include `#import`ed files no more than once, even if multiple `import` statements are encountered. Thus, you can do away with the messy preprocessor conditionals in the header file.

You should be careful, however, to only use `#import` for Objective-C interface headers, and continue using `#include` for standard C files. It is possible, though not likely, that regular C headers may rely on being included multiple times in some cases. Also, you may need to include the compiler directive `-Wno-import` to gcc to avoid a didactic warning to this effect.

4.1.1 Interface Capabilities

The interface file declares new classes that can be used by source code, holding all the information necessary to use the classes from other Objective-C code. Firstly, the file reveals to the programmer the position of the class in the class hierarchy by defining exactly which is the superclass. Secondly, it informs programmers of what variables are inherited when they create subclasses. Finally, the interface file may inform other software entities of the messages that can be sent to the class object and to the instances of the class.

Interface files use the `.h` extension as for ordinary C header files. (If you use emacs, put a line "`// -*-ObjC-*-`" at the top of your file to use the correct mode.)

Here is an example of a class interface declaration:

```
#import <Foundation/NSObject.h>
```

```
@interface Point : NSObject
{
@private
    // instance variables only accessible to instances of this class ...
@protected
    // instance variables accessible to instances of this class or subclasses
  float x;
  float y;
@public
    // instance variables accessible by all code ...
}

// class methods
+ (id) new;
+ (id) newWithX: (float)x0 Y: (float)y0;
+ (Point*) point;
+ (Point*) pointWithX: (float)x0 Y: (float)y0;

// instance methods
- (id) init;
- (id) initWithX: (float)x0 Y: (float)y0;
- (float) x;  // (field accessor)
- (float) y;
- (void) setX: (float)newX;
- (void) setY: (float)newY;
@end
```

- The interface file should import the interface of the superclass of the class it is defining.
- The interface is enclosed between the compiler directives `@interface` and `@end`.
- `@interface Point : Object` names the class and links it to the superclass. If no superclass is named, and the directive is without a colon, the compiler assumes that a root class is being created. You more than likely don't want to do this.
- Braces enclose declared instance variables; each class's instance will have all these instance variables including instance variables inherited from the superclass, and from the superclass of the superclass, extending to the root class.
- Instance variables may be declared as `private`, `protected`, or `public`. An instance's *private* variables may only be accessed by instances of this class. An instance's *protected* variables may be accessed by instances of this class or instances of subclasses of this class. *Public* variables may be accessed by any code. This is analogous to the usage in C++ and Java. If you do not mark your instance variable declaration explicitly, it is made `protected` by default.
- Method declarations that begin with a `"+"` sign are class methods, and are defined for the class object. Thus, you can call them without creating an instance, and their implementations do not have access to any instance variables. A class object inherits class methods from superclasses.
- Method declarations that begin with a `"-"` sign are instance methods, and are defined

for class instances. Class instances inherit instance methods from superclasses.

- A method may share the name of an instance variable.
- A method return type is declared using the C syntax for type casts:

```
- (float) x;
```

  which is a method returning a float.

- Argument types can be declared in the same way as method return types:

```
- (void) setX: (float)newX;
```

  which is a method that returns nothing, and takes a single float as its argument.

  Note. The default type for methods and messages (`id`) is assumed when a return or argument type is not explicitly declared. For example, '`-name`' implicitly means a method returning `id` (i.e. an object). It is usually better to avoid this and use explicit typing as in

```
- (NSString*) name;
```

4.1.2 Including Interfaces

Source code (including Objective-C implementation and interface files) may integrate interfaces using `#import` (or `#include`). Thereafter the source module may utilize the classes in those interfaces so as to:

- Make instances of them.
- Send messages to invoke methods declared for them.
- Refer to instance variables in them.

With the exception of the root class, all working interfaces integrate a superclass using either `#import` or `#include` – as was seen in the previous simplified interface file example. As a result the vast majority of class files begin with a standard form that includes their superclasses, and thus places them in the class hierarchy:

```
#import "SomeSuperclass.h"

@interface SomeClass : SomeSuperclass
{
  // instance variables ...
}
  // method declarations ...
@end
```

4.1.3 Referring to Classes - @class

It is possible for a source module to refer to classes without including their interface files. This is useful when you just need to tell the compiler that a certain word is a class name, but you want to avoid the overhead of including the whole interface file for that class.

For example, to inform the compiler that `Border` and `Square` are classes without including their full interface file, the following syntax is used:

```
@class Border, Square;
```

Class names may also appear in interface files at times when instance variables, return values and arguments are statically typed:

```
#import "Foundation/NSObject.h"

@class Point

@interface Square : NSObject
{
@protected
  Point *lowerLeft;
  float sideLength;
}
+ (id) newWithLowerLeft: (Point *)lowerLeft sideLength: (float)sideLength;

- (id) initWithLowerLeft: (Point *)lowerLeft sideLength: (float)sideLength;

- (Point *) lowerLeft;
- (float) sideLength;
- (void) setLowerLeft: (Point *)newLowerLeft;
- (void) setSideLength: (float)newSideLength;
@end
```

Here, we see the `Point` class we declared earlier being used as a component in `Square`'s definition. Because this class is only referred to here to declare variables and method signatures, it suffices to reference it only using the `@class` directive. On the other hand, the implementation file may need to send messages to `Point` instances and would be better of importing the interface in this case.

The compiler will produce a warning if you don't include it, and no type checking can be performed (to see if class instances respond to the messages you send to them), but compilation will succeed. It is best to take advantage of type-checking when you can, however, and include interfaces that messages are to be sent to.

There is one situation where you *must* include the interface however. If you are implementing a new class, you always need to include the interface of the superclass; `@class` cannot be used in this case because the compiler needs to know the details of the superclass and its instance variables etc., so as to create a fully working new class. If you try using `@class` in this situation, compilation will abort.

4.2 Implementation

An interface file declares a class, while an implementation file implements it. The separation between the interface and implementation file yields a black box concept where the programmer using the class need only be concerned with the interface and its declared methods, superclasses, and instance variables. The implementation of classes is transparent to the programmer who may use them without detailed knowledge of their structures.

Implementation files use the `.m` extension, to distinguish them from ordinary C files.

4.2.1 Writing an Implementation

An implementation file contents are encapsulated between `@implementation` and `@end` directives:

```
#import "Point.h"
@implementation Point
// method implementations
+ (id)new
{
  // statements ...
}

+ (id)newWithX: (float)x Y: (float)y
{
  // statements ...
}

// ...

- (void)setY: (float)newY
{
  // statements ...
}

@end
```

The implementation file uses `#import` to include a named interface file holding all declarations. Then it places method implementations for the class between `@implementation` and `@end` directives. Each method declared in the interface must be implemented. Instance variables may be referred to in instance methods (the ones with a "-" in front of them) but not class methods (the ones with a "+").

```
- (float) x
{
  return x;
}

- (void) setX: (float)newX
{
  x = newX;
}
```

4.2.2 Super and Self

To assist in writing instance methods, Objective-C provides the two reserved words `self` and `super`. `Self` is used to refer to the current instance, and is useful for, among other things, invoking other methods on the instance:

```
- (Foo *) foo
{
```

```
  if (![self fooIsInitialized])
    [self initializeFoo];
  return foo;
}
```

`Super` is used to refer to method implementations in the superclass of the instance. It is useful when overriding methods and when writing initializers, as discussed in the next section.

4.2.3 Instance Initialization

Instance initialization is one of the trickier aspects of getting started in Objective-C. Recall that instances of a class are created by use of the class `alloc` method (inherited from `NSObject`) but are initialized by instance methods. This works a little differently than in C++ and Java, where constructors are special methods that are neither class nor instance methods. In particular, since initializer methods are inherited instance methods, they may still be called even if you have not implemented them in your class. For example, it is always valid to invoke

```
SomeComplexClass *c = [[SomeComplexClass alloc] init];
```

Even if you have not implemented `init` in `SomeComplexClass`, the superclass's implementation will be invoked, or, ultimately, `NSObject`'s if no other ancestors implement it. Obviously, this could result in some of `SomeComplexClass`'s internal state being left uninitialized. For this reason, you should always either provide an `init` implementation, or document whether it should be used. We will return to this concern below.

Typically, a class will also provide one or more `initWith...` methods for initialization with arguments, and it may optionally also provide `+new` methods and convenience class methods that act like constructors. The general approach to implementing these is illustrated here for the `Point` class.

```
+ new
{
  Point *point;

  // note "self" refers to the "Point" _class_ object!
  point = [[self allocWithZone: NSDefaultMallocZone()] init];
  return point;
}

+ newWithX: (float)x0 Y: (float)y0
{
  Point *point;

  point = [[self allocWithZone: NSDefaultMallocZone()] initWithX: x Y: y];
  return point;
}

+ point
{
  Point *point;
```

```
  // note "self" refers to the "Point" _class_ object!
  point = [self new];
  return AUTORELEASE(point);
}

+ pointWithX: (float)x0 Y: (float)y0
{
  Point *point;

  point = [self newWithX: x Y: y];
  return AUTORELEASE(point);
}

- init
{
  return [self initWithX: 0.0 Y: 0.0];
}

// this is the "designated" initializer
- initWithX: (float)x0 Y: (float)y0
{
  self = [super init];
  if (self != nil)
    {
      x = x0;
      y = y0;
    }
  return self;
}
```

Notice that, first, the convenience constructors (`new` and `newWithX:Y:`) execute `[self allocWithZone:]` to begin with. The "`self`" here refers to the *class* object, since it is used inside a *class* method. Thus the effect is the same as if "`[Point alloc]`" had been executed in external code. Second, notice that the other convenience constructors (`point` and `pointWithX:Y:`) autorelease the new instance before returning it. This is to follow the rules of memory management discussed in Chapter 3 [Memory Management], page 23. Third, note that the `new..` methods each call a corresponding `init...` method. It is not necessary to maintain such a one to one correspondence but it is a common convention to have the convenience implementations rely on instance `init` methods as shown. Fourth, note that the use of `[self allocWithZone: NSDefaultMallocZone()]` rather than `[self alloc]` is generally unnecessary, but provides a slight efficiency gain since `+alloc` is implemented by calling `+allocWithZone:` on the default zone.

Designated Initializer

Finally, notice that the `initWithX:Y:` method is marked as the "designated" initializer. This concept is important to ensure proper initialization for classes within a hierarchy. The designated initializer should be the one with the most control over the nature of the new

instance, and should be the one that all other initializers "ground out" in. In other words, all other initializers should be chained so that they either call the designated initializer, or they call another initializer that (eventually) calls it.

The importance of having a designated initializer is this: when a subclass is created, it need only override the designated initializer to ensure that all of its instances are properly initialized. If this is not done, external code could invoke an initializer that initializes only the superclass's instance variables, and not the subclass's. To avoid this, each class designates a "ground out" initializer to which other initializers ultimately delegate. Then the subclass overrides this initializer, and in its own designated initializer, makes a call to it, to ensure that the superclass is initialized properly. Thus:

```
@implementation SuperClass
- initWithA: (int)a
{
  return [self initWithA:a B:0];  // 0 is default value
}

// designated init for SuperClass
- initWithA: (int)a B: (int)b
{
  self = [super init];
  myA = a;
  myB = b;
  return self;
}
@end

@implementation SubClass

// overrides SuperClass's designated init
- initWithA: (int)a B: (int)b
{
  return [self initWithA: (int)a B: (int)b C: (int)c];
}

// designated init for SubClass
- initWithA: (int)a B: (int)b C: (int)c
{
  self = [super initWithA: a B: b];
  myC = c;
  return self;
}
@end
```

Note, as shown above, unlike in some other object-oriented languages, '`self`' is a variable that can be redefined. For example, we could have written the `new` constructor above like this:

```
{
```

```
  self = [[self alloc] init];
    // note "self" now refers to the new instance!
  [self setX: 1.0];
  return self;
}
```

Another point to note is that Objective-C does not enforce calling superclass initializers before carrying out subclass initialization. Although above the first call in the designated initializer was always `[super ...]`, this was not required, and if you need to set something up before `super` acts, you are free to do so.

4.2.4 Flexible Initialization

As mentioned before, it is possible for an initialization process to, if desired, return not a new object but an existing object. This may be done in one of two ways. If you are doing it from a convenience class method like `new`, then use something like the following:

```
+ new
{
  if (singleton == nil)
    singleton = [[self alloc] init];
  return singleton;
}
```

Note this example presupposes the existence of a class variable, '`singleton`'. Class variables as such don't exist in Objective-C but can be simulated, as discussed below.

If you want to possibly return an existing instance from an init instance method like `init`, the procedure is slightly more complicated:

```
- init
{
  if (singleton != nil)
    {
      RELEASE(self);
      self = RETAIN(singleton);
    }
    else
    {
      singleton = self;
    }
  return self;
}
```

Here, we explicitly **deallocate** the current instance and replace it with the desired existing instance. Because this might happen, you should always be careful to use the returned value from an `init` method:

```
id anObject = [SomeClass alloc];
  // this is bad:
[anObject init];
  // anObject might have been deallocated!
  // do this instead:
```

```
anObject = [anObject init];
```

One scenario where this actually occurs in the GNUstep libraries is with the class `NSConnection`. It only permits one connection to exist between any two ports, so if you call `initWithReceivePort:sendPort:` when a connection for the ports exists, the method will deallocate the newly allocated instance, and return the current conflicting object, rather than the receiver.

In general, it is better to catch this type of requirement in a "`new`" class method rather than an instance "`init`" method so as to avoid the unnecessary allocation of instances that will not be used, however this is not always possible given other design constraints.

4.2.5 Instance Deallocation

As described in Chapter 3 [Memory Management], page 23, objects should be deallocated when they are no longer needed. When ARC is not being used, this is done through explicit calls to the `dealloc` method. When ARC *is* being used, `dealloc` is still called implicitly, and should be implemented. However the tasks of the `dealloc` method are fewer in this case.

When ARC is *not* used, the `dealloc` method must release all other objects that this instance has retained. Usually these are those instance variables that are objects rather than primitive types. In certain cases such as container classes, other objects must be released as well. In addition, if the instance has acquired any external resources, such as a network connection or open file descriptor, these should be relinquished as well. Likewise, any memory that has been directly allocated through use of `malloc` or other functions should be released.

When ARC *is* active, the `dealloc` method is still responsible to relinquish external resources, but other GNUstep objects must not be released, since they will be managed by ARC.

If you cannot be sure whether your class will be running in an ARC environment, you should use the RETAIN/RELEASE macros instead of the function calls.

Here is an example of a `dealloc` implementation:

```
- dealloc
{
  RELEASE(anInstanceVariableObject);
  NSZoneFree(NULL, myMemory);
  DEALLOC
}
```

Here, we use the `RELEASE` macro to release an instance variable, and the `NSZoneFree` function to free memory that was earlier allocated with `NSZoneMalloc` or a related function. (See Chapter 8 [Base Library], page 83, for discussion of GNUstep's raw memory allocation functions.) The `NULL` used indicates that the memory was from the default zone, and is equivalent to saying '`NSDefaultMallocZone()`' instead.

Finally, notice we end with the DEALLOC macro to call the superclass implementation (but not for ARC). This should always be done in `dealloc` implementations, and you should never concern yourself with deallocating structures that are associated with a superclass, since it will take care of this itself.

4.3 Protocols

Protocols in Objective-C provide a level of flexibility beyond class structure in determining what messages objects respond to. They are similar to interfaces in Java but more flexible.

There are two types of protocol in Objective-C: **informal** protocols, where we document methods to which objects will respond, and specify how they should behave, and **formal** protocols, where we provide a list of methods that an object will support in a format where the compiler can check things, and the runtime can also check that an object conforms to the protocol. Informal protocols are merely convention, but are useful where we want to say that some system will work as long as it (or its delegate) implements some subset of a group of methods. Formal protocols are of more use when we want the compiler or runtime to check that an object implements all of a group of methods itself. Formal protocols form an inheritance hierarchy like classes, and a given class may conform to more than one protocol. Thus, formal protocols are identical in many respects to Java *interfaces*.

As in Java, a particularly important use of protocols is in defining the methods that an object in a remote process can respond to ... by setting the protocol used by a local proxy object, you can avoid having to send messages to the remote process to check what methods are available - you can simply check the local protocol object. This will be covered later in Chapter 7 [Distributed Objects], page 63.

Informal protocols are closely associated with *Categories*, another Objective-C language facility, and will be discussed in the next section.

4.3.1 Declaring a Formal Protocol

A formal protocol is declared as a series of method declarations, just like a class interface. The difference is that a protocol declaration begins with `@protocol` rather than `@interface`, and has an optional **super** protocol specified in angle brackets.

```
@protocol List
- (void) add:        (id) item;
- (void) remove:     (id) item;
- getAtIndex:        (int)idx;
- (void) clear;
@end

@protocol LinkedList <List>
- (void) addFirst:   (id)item;
- (void) addLast:    (id)item;
- getFirst;
- getLast;
@end
```

4.3.2 Implementing a Formal Protocol

If you want your class to conform to a protocol, you declare it in your interface, and implement the methods in your declaration:

```
@interface BiQueue <LinkedList>
{
  // instance variables ...
```

```
}
  // method declarations ...
  // [don't need to redeclare those for the LinkedList protocol]
- takeFirst
- takeLast
@end

...

@implementation BiQueue
  // must implement both List's and LinkedList's methods ...
- add:        (id) item
{
  // ...
}

- addFirst:   (id)item
{
  // ...
}
@end
```

To declare conformance to multiple protocols, do something like this:

```
@interface ContainerWindow < List, Window >
  ...
@end
```

The implementation must include all methods in both protocols.

4.3.3 Using a Formal Protocol

To use a formal protocol, simply send objects the messages in the protocol. If you want type-checking, you must either use the type of a class implementing the protocol, or use a special syntax:

```
...
BiQueue queue = [[BiQueue alloc] init];
  // send a LinkedList message
[queue addFirst: anObject];

  // alternatively, we may stipulate only that an object conforms to the
  // protocol in the following way:
id<LinkedList> todoList = [system getTodoList];
task = [todoList getFirst];
...
```

In the last part of this example, we declare that `todoList` is an object that conforms to the `LinkedList` protocol, but do not specify what class it may be an instance of.

If you are not sure the returned object does indeed conform to the protocol you are interested in, you can check it:

```
if ([anObject conformsToProtocol: aProtocol] == YES)
  {
    // We can go ahead and use the object.
  }
else
  {
    NSLog(@"Object of class %@ ignored ... does not conform to
      protocol", NSStringFromClass([anObject class]));
  }
```

Finally, you can specify an object conforming to *multiple* protocols in the same way you declare it in an interface:

```
id <LinkedList, Window>   windowContainerOfUnknownClass;
```

4.4 Categories

Categories provide a way in Objective-C to add new methods to an existing class, without declaring a subclass. Once the category is declared and implemented, all instances of the existing class that are created will include the capability to respond to the new methods. Furthermore, subclasses of the class will inherit these methods. However, it is not possible to add instance variables to a class using a category. Categories do not have an obvious parallel in other major object-oriented languages (with the exception of Ruby), but it is well worth taking the trouble to understand them and the benefits they can provide.

A category is declared in connection with the class it is going to modify. (You can think of it as a new "category" of instances of this class.)

```
#import "Point.h"

@interface Point (Transformable)
- translateByX: (float)tx Y: (float)ty;
- rotateByAngle: (float)radians;
- scaleByAmountX: (float)xscale Y: (float)yscale;
@end
```

You then provide an implementation file more or less analogously to that for a class, where you implement just the new methods:

```
#import "PointTransformable.h"

@implementation Point (Transformable)
- (void) translateByX: (float)tx Y: (float)ty
{
  x += tx;
  y += ty;
  return self;
}

- (void) rotateByAngle: (float)radians
{
  // ...
```

```
}

- (void) scaleByAmountX: (float)xscale Y: (float)yscale
{
  // ...
}
@end
```

Notice that you have access to instance variables of the class you are creating a category of; this includes private and protected variables.

One of the primary uses of categories is illustrated by this example. Suppose you are working with a third party drawing package that uses some geometrical classes such as `Point` and `Line`. You are developing an animation program based on the package and need the ability to move things around. Rather than employing a complex subclassing or aggregation scheme to add these capabilities, you simply define the `Transformable` category for each of the geometrical entities. At runtime, all instances of these entities, whether created by you or the package itself, have the additional methods. The presence of these methods does not affect the existing operation of this or any third party package, but allows you to conveniently implement the enhanced functionality you need.

4.4.1 Category Overrides

You can also use categories to override methods that a class already has. If you do so, you cannot access an existing implementation in the class itself, however you can still call `[super someMethod]` to access an implementation inherited from a superclass. You obviously need to be careful not to break existing functionality.

You can add multiple categories to a class by declaring them and implementing them separately. Instances of the class will then implement *all* of the categories declared. The order in which the category implementations are searched for methods is not defined, therefore you cannot override a method implemented in one category with an implementation in another.

4.4.2 Categories as an Implementation Tool

Categories are not just useful for extending an existing class. Another major use for categories is to separate the implementation of a *new* class into a number of source files. (Each file implements one category of the new class, and at runtime instances of the class respond to the methods in all the categories.) The benefits of this program development strategy include: grouping subject-oriented methods; incremental compilation for large classes; helping to logically divide the class when being created by a number of developers; and, permitting configuration-specific classes targeting particular applications.

4.4.3 Categories and Protocols

As described in the previous section, in addition to the *formal* protocol facility described, Objective-C provides for *informal* protocols. An informal protocol is essentially a category declaration without an implementation. Usually, the informal protocol is declared as a category for a high-level object, such as `NSObject`, then each class that actually wishes to implement something in the protocol lists the methods it chooses to implement in its interface and provides implementations in its implementation.

4.5 Simulating Private and Protected Methods

Unlike most object-oriented languages Objective-C does not provide for method scoping. Instead, all methods are effectively public. Often, however, it is useful to have internal "utility" methods that help a class do its job but are hidden from external use. Rather than cluttering up the class's API with a bunch of methods marked "do not use", one wants to make these methods visible only to subclasses, or only to the class itself. Categories can help in this regard.

Using Categories

One common approach is to define a category within a class's *implementation* file:

```
#import "Point.h"

@interface Point (Private)
-(BOOL) isPositiveQuadrant;
@end

@implementation Point
  // public method implementations ...
@end

@implementation Point (Private)
-(BOOL) isPositiveQuadrant
{
  return (x > 0) && (y > 0) ? YES : NO;
}
@end
```

All of this code would appear in the file `Point.m`. What this does is add a category to Point defining the private methods. Since external code only "knows about" `Point` through its interface file, these additional methods are effectively invisible. However, you should be aware that external code is not prevented from actually calling the private methods, if it happens to know about them. However the compiler will produce a warning if you try to do this with a typed variable:

```
Point *p = [[Point alloc] init];
  // works, but produces a compile warning
BOOL b = [p isPositiveQuadrant];
```

The bright side of this is it allows you to simulate *protected* methods as well. For this, the writer of a subclass must be informed in some way about the protected methods, and they will need to put up with the compiler warnings. Alternatively, you could declare the Protected category in a separate interface file (e.g., "`PointProtected.h`"), and provide this interface file with the understanding that it should only be imported and used by a subclass's interface file.

Using Convention

Another approach to providing *protected* methods that the class or subclass can use is to prefix these methods with an underscore ('_'). These methods will still be visible publicly,

but programmers will know, by convention, not to use them externally, and the Appendix A [GSDoc], page 99, will automatically mark these in API documentation as off-limits.

An alternative approach to providing *private* methods is to simply declare them as functions within the implementation file itself. The catch to this is that these functions will *not* have access to the class's instance variables. You will need to pass these in manually whenever you invoke them from an ordinary method.

4.6 Simulating Class Variables

While Objective-C does not provide for variables that are associated with the class as a whole rather than an instance, these are often useful. It is possible to simulate them to a limited extent by declaring static variables in the implementation file for the class (inside the `@implementation` block). The variables will not be available to subclasses, unless they explicitly declare them "`extern`" and are compiled at the same time.

...

5 Advanced Messaging

Objective-C provides some additional possibilities for message routing besides the capabilities described so far (inheritance and categories). One of the most important is that it is possible for an object, upon receiving a message it has not been set up to respond to, to *forward* that message to another object. A second important capability, which forwarding relies on, is the ability to represent method implementations directly in code. This supports various reflective operations as well as optimization where messages are sent many times.

5.1 How Messaging Works

Sending an Objective-C message requires three types of information:

- The message **receiver** - the object which is to perform the request.
- The message **selector** - this identifies the message, and is used to locate the excecutable code of the corresponding **method** by searching the structure of the class, and if necessary its superclasses, for an implementation.
- The message **arguments** - once the implementation has been found, these are simply passed to the method on the stack as in an ordinary function call.

In the message '`[taskArray insertObject: anObj atIndex: i]`', the receiver is "`taskArray`", the selector is "`insertObject:atIndex:`", and the arguments are "`anObj`" and "`i`". Notice that the selector includes the argument titles and both colons, but not the argument names. In other words, this method might have been declared as '`- (void) insertObject: (id)anObject atIndex: (unsigned)index;`', but the "`anObject`" and "`index`" are just used for tracking the arguments within the method implementation code and not for looking up the method itself.

The following sequence of events would occur on sending this message at runtime:

1. The internal `isa` pointer of the **receiver** (`taskArray`) is used to look up its class.
2. The class representation is searched for a method implementation matching the **selector** (`insertObject:atIndex:`). If it is not found, the class's superclass is searched, and recursively its superclass, until an implementation is found.
3. The implementation is called, as if it were a C function, using the **arguments** given (`anObj` and `i`), and the result is returned to the code sending the message.

In fact, when the method implementation is actually called, it additionally receives two *implicit* arguments: the **receiver** and the **selector**. These additional hidden arguments may be referred to in the source code by the names `self` and `_cmd`.

The process of looking up the method implementation in the receiver at runtime is known as dynamic binding. This is part of what makes the language powerful and flexible, but it is inevitably (despite clever caching strategies used in the runtime library) a little slower than a simple function call in C. There are, however, ways of short-circuiting the process in cases where performance is at a premium. Before discussing this, we must first cover the concepts of selectors and implementations in greater detail.

5.2 Selectors

So far we have been using the following syntax to send messages to objects:

```
[myArray removeObjectIdenticalTo: anObject];
```

The example sends the message named `removeObjectIdenticalTo:` to `myArray` with the argument `anObject`.

An alternative method of writing this is the following:

```
SEL removalSelector = @selector(removeObjectIdenticalTo:);
[myArray performSelector: removalSelector withObject: anObject];
```

Here, the first line obtains the desired method selector in the form of a compiled representation (not the full ASCII name), and the second line sends the message as before, but now in an explicit form. Since the message that is sent is now effectively a variable set at runtime, this makes it possible to support more flexible runtime functioning.

5.2.1 The Target-Action Paradigm

One conventional way of using selectors is called the *target-action* paradigm, and provides a means for, among other things, binding elements of a graphical user interface together at runtime.

The idea is that a given object may serve as a flexible signal sender if it is given a receiver (the *target*) and a selector (the *action*) at runtime. When the object is told to send the signal, it sends the selector to the receiver. In some variations, the object passes itself as an argument.

The code to implement this paradigm is simple -

```
- (id) performAction
{
  if (target == nil || action == 0)
    {
      return nil; // Target or action not set ... do nothing
    }
  if ([target respondsToSelector: action] == NO)
    {
      return nil; // Target cannot deal with action ... do nothing
    }
  return [target performSelector: action withObject: self];
}
```

As an example, consider a graphical button widget that you wish to execute some method in your application when pressed.

```
[button setTarget: bigMachine]
[button setAction: @selector(startUp:)];
```

Here, `button` stores the given target and action in instance variables, then when it is pressed, it internally calls a method like `performAction` shown above, and sends the message "`[bigMachine startUp: button]`".

If you are used to programming with events and listeners in Java, the target-action paradigm provides a lighter-weight alternative for the most common case where only one object needs to be informed when an event occurs. Rather than writing or extending a special-purpose

adaptor class, you just register the method you want called directly with the actuating element. If you need to send the event to multiple objects, however, you would need to write a special method to multiplex the event out. This would be approximately comparable effort to what is always required in Java, and is only needed in the minority of cases.

5.2.2 Obtaining Selectors

In addition to using the compile-time `@selector` operator, there are a couple of other ways of obtaining selectors.

- In a method implementation, you can always obtain the current selector from the variable `_cmd`:

```
- (void) removeObjectIdenticalTo: (id)anObject
{
  SEL  mySelector = _cmd;
    // ...
}
```

- At any point, you can use the `NSSelectorFromString()` function -

```
SEL  mySelector = NSSelectorFromString(@"removeObjectIdenticalTo:");
```

  In reality, you would never use `NSSelectorFromString` for a constant string as shown; `@selector` would do and is more efficient, since is a compile-time operator. Its chief utility lies in the case where the selector name is in a variable value (for whatever reason).

If you ever need to test the contents of a `SEL` variable for equality with another, you should use the function `sel_eq()` provided as part of the GNU Objective-C runtime library. This is necessary because, while the compiler tries to ensure that compile-time generated references to selectors for a particular message point to the same structure, selectors produced at runtime, or in different compilation units, will be different and a simple pointer equality test will not do.

5.2.3 Avoiding Messaging Errors when an Implementation is Not Found

Using **typed** objects as shown below, the compiler would forewarn you if the `anObject` was unable to respond to the `alert:` message, as it knows what type of object `anObject` is:

```
SomeClass *anObject; // an instance of the 'SomeClass' class

anObject = [[SomeClass alloc] init]; // build and initialize the object
[anObject alert: additionalObject]; // compiler warns if 'alert:' not
                                    // defined in SomeClass or a superclass
```

However at times the compiler will not forewarn you that a message will attempt to invoke a method that is not in the **receiver's** repertoire. For instance, consider the code below where `anObject` is not known to implement the `alert:` message:

```
id anObject; // arbitrary object;

anObject = [[SomeClass alloc] init]; // build and initialize object
[anObject alert: additionalObject]; // compiler cannot check whether
                                    // 'alert' is defined
```

In this case, the compiler will not issue a warning, because it only knows that `anObject` is of type `id` . . . so it doesn't know what methods the object implements.

At runtime, if the Objective-C runtime library fails to find a **method implementation** for the `alert:` message in the `SomeClass` class or one of its superclasses, an exception is generated. This can be avoided in one of two ways.

The first way is to check in advance whether the method is implemented:

```
if ([anObject respondsToSelector: @selector(alert:)] == YES)
  {
    [anObject alert: additionalObject]; // send it a message.
  }
else
  {
    // Do something else if the object can't be alerted
  }
```

The second way is for the object the message was sent to to *forward* it somewhere else.

5.3 Forwarding

What actually happens when the GNU Objective-C runtime is unable to find a method implementation associated with an object for a given selector is that the runtime instead sends a special `forwardInvocation:` message to the object. (Other Objective-C runtimes do the same, but with a slightly different message name and structure.) The object is then able to use the information provided to handle the message in some way, a common mechanism being to forward the message to another object known as a **delegate**, so that the other object can deal with it.

```
- (void) forwardInvocation: (NSInvocation*)invocation
{
  if ([forwardee respondsToSelector: [invocation selector]])
    return [invocation invokeWithTarget: forwardee];
  else
    return [self doesNotRecognizeSelector: [invocation selector]];
}
```

- `invocation` is an instance of the special `NSInvocation` class containing all the information about the original message sent, including its **selector** and its arguments.
- `forwardee` is an instance variable containing the `id` of an object which has been determined to be likely to implement methods that this object does not.
- The `NSInvocation` class has a convenience method that will pass the message on to a target object given as argument.
- The `doesNotRecognizeSelector` method is a fallback which is implemented in `NSObject`. Unless it has been overidden, its behavior is to raise a runtime exception (a `NSInvalidArgumentException` to be exact), which generates an error message and aborts.

Forwarding is a powerful method for creating software patterns. One of these is that forwarding can be used to in effect provide a form of multiple inheritance. Note, however that, unlike inheritance, a forwarded method will not show up in tests like `respondsToSelector`

and `isKindOfClass:`. This is because these methods search the inheritance path, but ignore the forwarding path. (It is possible to override them though.)

Another pattern you may come across is *surrogate object*: surrogates forward messages to other objects that can be assumed to be more complex. The `forwardInvocation:` method of the surrogate object receives a message that is to be forwarded; it determines whether or not the receiver exists, and if it does not, then it will attempt to create it. A **proxy** object is a common example of a surrogate object. A proxy object is useful in a remote invocation context, as well as certain scenarios where you want one object to fulfill functions of another.

5.4 Implementations

Recall that when a message is sent, the runtime system searches for a method implementation associated with the recipient object for the specified selector. (Behind the scenes this is carried out by a function "`objc_msgSend()`".) This may necessitate searches across multiple superclass objects traversing upwards in the inheritance hierarchy, and takes time. Once the runtime finds an implementation for a class, it will cache the information, saving time on future calls. However, even just checking and accessing the cache has a cost associated with it. In performance-critical situations, you can avoid this by holding on to an implementation yourself. In essence, implementations are function pointers, and the compiler provides a datatype for storing them when found at runtime:

```
SEL  getObjSelector = @selector(getObjectAtIndex:);
  // get the 'getObjectAtIndex' implementation for NSArray 'taskArray'
IMP  getObjImp = [taskArray methodForSelector: getObjSelector];
  // call the implementation as a function
id obj = (getObjImp)( taskArray, getObjSelector, i );
```

Here, we ask the runtime system to find the '`taskArray`' object's implementation of '`getObjectAtIndex`'. The runtime system will use the same algorithm as if you were performing a method call to look up this code, and then returns a function pointer to it. In the next line, this pointer is used to call the function in the usual C fashion. Notice that the signature includes both the object and the selector – recall that these are the two implicit arguments, `self` and `_cmd`, that every method implementation receives. The actual type definition for `IMP` allows for a variable number of additional arguments, which are the explicit arguments to the method call:

```
typedef id (*IMP)(id, SEL, ...);
```

The return type of `IMP` is `id`. However, not all methods return `id`; for these others you can still get the implementation, but you cannot use an `IMP` variable and instead must cast it yourself. For example, here is such a cast for a method taking a double and returning '`double`':

```
double (*squareFunc)( id, SEL, double );
double result;

squareFunc = (double (*)( id, SEL, double ))
     [mathObj methodForSelector: @selector(squareOf:)];

result = squareFunc(mathObj, @selector(squareOf:), 4);
```

You need to declare such a function pointer type for any method that returns something besides `id` or `int`. It is not necessary to declare the argument list (`double`) as we did above; the first line could have been "`double (*squareFunc)( id, SEL, ... )`" instead.

An excellent exposition of the amount of time saved in using `methodForSelector` and other details of the innards of Objective-C and the Foundation may be found here: `http://www.mulle-kybernetik.com/artikel/Optimization/opti-3.html`.

You should realize that it is only worth it to acquire the `IMP` if you are going to call it a large number of times, and if the code in the method implementation itself is not large compared with the message send overhead. In addition, you need to be careful not to call it when it might be the wrong function. Even when you are sure of the class of the object you are calling it on, Objective-C is sufficiently dynamic that the correct function could change as a program runs. For example, a new category for a class could be loaded, so that the implementation of a method changes. Similarly, a class could be loaded that poses as another, or one that was posing stops doing so. In general, `IMPs` should be acquired just before they are to be used, then dropped afterwards.

6 Exception Handling, Logging, and Assertions

No matter how well a program is designed, if it has to interact with a user or other aspect of the outside world in any way, the code is bound to occasionally meet with cases that are either invalid or just plain unexpected. A very simple example is when a program asks the user to enter a filename, and the user enters the name of a file that does not exist, or does not enter a name at all. Perhaps a valid filename *is* entered, but, due to a previous disk write error the contents are garbled. Any number of things can go wrong. In addition, programmer error inevitably occurs and needs to be taken account of. Internal functions may be called with invalid arguments, either due to unexpected paths being taken through the code, or silly things like typos using the wrong variable for something. When these problems happen (and they *will* happen), it is better to handle them gracefully than for the program to crash, or worse, to continue processing but in an erroneous way.

To allow for this, many computer languages provide two types of facilities. The first is referred to as *exception handling* or sometimes *error trapping*. The second is referred to as *assertion checking*. Exceptions allow the program to catch errors when they occur and react to them explicitly. Assertions allow a programmer to establish that certain conditions hold before attempting to execute a particular operation. GNUstep provides both of these facilities, and we will cover each in turn. The assertion facility is tied in with the GNUstep *logging* facilities, so we describe those as well.

To use any of the facilities described in this chapter requires that you include `Foundation/NSException.h`.

6.1 Exceptions

GNUstep exception handling provides for two things:

1. When an error condition is detected during execution, control is passed to a special error-handling routine, which is given information on the error that occurred.
2. This routine may itself, if it chooses, pass this information up the function call stack to the next higher level of control. Often higher level code is more aware of the context in which the error is occurring, and can therefore make a better decision as to how to react.

6.1.1 Catching and Handling Exceptions

GNUstep exception handling is implemented through the macros `NS_DURING`, `NS_HANDLER`, and `NS_ENDHANDLER` in conjunction with the `NSException` class. The following illustrates the pattern:

```
NS_DURING
  {
    // do something risky ...
  }
NS_HANDLER
  {
    // a problem occurred; inform user or take another tack ...
  }
NS_ENDHANDLER
```

```
  // back to normal code...
```

For instance:

```
- (DataTree *) readDataFile: (String *)filename
{
  ParseTree *parse = nil;
  NS_DURING
    {
      FileHandle *handle = [self getFileHandle: filename];
      parse = [parser parseFile: handle];
      if (parse == nil)
        {
          NS_VALUERETURN(nil);
        }
    }
  NS_HANDLER
    {
      if ([[localException name] isEqualToString: MyFileNotFoundException])
        {
          return [self readDataFile: fallbackFilename];
        }
      else if ([[localException name] isEqualToString: NSParseErrorException])
        {
          return [self readDataFileInOldFormat: filename];
        }
      else
        {
          [localException raise];
        }
    }
  NS_ENDHANDLER
  return [[DataTree alloc] initFromParseTree: parse];
}
```

Here, a file is parsed, with the possibility of at least two different errors: not finding the file and the file being misformatted. If a problem does occur, the code in the `NS_HANDLER` block is jumped to. Information on the error is passed to this code in the `localException` variable, which is an instance of `NSException`. The handler code examines the name of the exception to determine if it can implement a work-around. In the first two cases, an alternative approach is available, and so an alternative value is returned.

If the file is found but the parse simply produces a nil parse tree, the `NS_VALUERETURN` macro is used to return nil to the `readDataFile:` caller. Note that it is *not* allowed to simply write "`return nil;`" inside the NS_DURING block, owing to the nature of the behind-the-scenes C constructs implementing the mechanism (the `setjmp()` and `longjmp()` functions). If you are in a void function not returning a value, you may use simply "`NS_VOIDRETURN`" instead.

Finally, notice that in the third case above the handler does not recognize the exception type, so it passes it one level up to the caller by calling `-raise` on the exception object.

6.1.2 Passing Exceptions Up the Call Stack

If the caller of `-readDataFile:` has enclosed the call inside its own `NS_DURING ... NS_HANDLER ... NS_ENDHANDLER` block, it will be able to catch this exception and react to it in the same way as we saw here. Being at a higher level of execution, it may be able to take actions more appropriate than the `-readDataFile:` method could have.

If, on the other hand, the caller had *not* enclosed the call, it would not get a chance to react, but the exception would be passed up to the caller of *this* code. This is repeated until the top control level is reached, and then as a last resort `NSUncaughtExceptionHandler` is called. This is a built-in function that will print an error message to the console and exit the program immediately. If you don't want this to happen it is possible to override this function by calling `NSSetUncaughtExceptionHandler(fn_ptr)`. Here, `fn_ptr` should be the name of a function with this signature (defined in `NSException.h`):

```
void NSUncaughtExceptionHandler(NSException *exception);
```

One possibility would be to use this to save files or any other unsaved state before an application exits because of an unexpected error.

6.1.3 Where do Exceptions Originate?

You may be wondering at this point where exceptions come from in the first place. There are two main possibilities. The first is from the Base library; many of its classes raise exceptions when they run into error conditions. The second is that application code itself raises them, as described in the next section. Exceptions do *not* arise automatically from C-style error conditions generated by C libraries. Thus, if you for example call the `strtod()` function to convert a C string to a double value, you still need to check `errno` yourself in standard C fashion.

Another case that exceptions are *not* raised in is in the course of messaging. If a message is sent to `nil`, it is silently ignored without error. If a message is sent to an object that does not implement it, the `forwardInvocation` method is called instead, as discussed in Chapter 5 [Advanced Messaging], page 49.

6.1.4 Creating Exceptions

If you want to explicitly create an exception for passing a particular error condition upwards to calling code, you may simply create an `NSException` object and `raise` it:

```
NSException myException = [[NSException alloc]
                              initWithName: @"My Exception"
                                    reason: @"[Description of the cause...]"
                                  userInfo: nil];
[myException raise];
 // code in block after here is unreachable..
```

The `userInfo` argument here is a `NSDictionary` of key-value pairs containing application-specific additional information about the error. You may use this to pass arbitrary arguments within your application. (Because this is a convenience for developers, it should have been called `developerInfo`..)

Alternatively, you can create the exception and raise it in one call with `+raise`:

```
[NSException raise: @"My Exception"
```

```
                format: @"Parse error occurred at line %d.",lineNumber];
```

Here, the `format` argument takes a printf-like format analogous to `[NSString -stringWithFormat:]` discussed Chapter 2 [Strings in GNUstep], page 11. In general, you should not use arbitrary names for exceptions as shown here but constants that will be recognized throughout your application. In fact, GNUstep defines some standard constants for this purpose in `NSException.h`:

`NSCharacterConversionException`
: An exception when character set conversion fails.

`NSGenericException`
: A generic exception for general purpose usage.

`NSInternalInconsistencyException`
: An exception for cases where unexpected state is detected within an object.

`NSInvalidArgumentException`
: An exception used when an invalid argument is passed to a method or function.

`NSMallocException`
: An exception used when the system fails to allocate required memory.

`NSParseErrorException`
: An exception used when some form of parsing fails.

`NSRangeException`
: An exception used when an out-of-range value is encountered.

Also, some Foundation classes define their own more specialized exceptions:

`NSFileHandleOperationException (NSFileHandle.h)`
: An exception used when a file error occurs.

`NSInvalidArchiveOperationException (NSKeyedArchiver.h)`
: An archiving error has occurred.

`NSInvalidUnarchiveOperationException (NSKeyedUnarchiver.h)`
: An unarchiving error has occurred.

`NSPortTimeoutException (NSPort.h)`
: Exception raised if a timeout occurs during a port send or receive operation.

`NSUnknownKeyException (NSKeyValueCoding.h)`
: An exception for an unknown key.

6.1.5 When to Use Exceptions

As might be evident from the `-readDataFile:` example above, if a certain exception can be anticipated, it can also be checked for, so you don't necessarily need the exception mechanism. You may want to use exceptions anyway if it simplifies the code paths. It is also good practice to catch exceptions when it can be seen that an unexpected problem might arise, as any time file, network, or database operations are undertaken, for instance.

Another important case where exceptions are useful is when you need to pass detailed information up to the calling method so that it can react appropriately. Without the ability to raise an exception, you are limited to the standard C mechanism of returning a value

that will hopefully be recognized as invalid, and perhaps using an `errno`-like strategy where the caller knows to examine the value of a certain global variable. This is inelegant, difficult to enforce, and leads to the need, with void methods, to document that "the caller should check `errno` to see if any problems arose".

6.2 Logging

GNUstep provides several distinct logging facilities best suited for different purposes.

6.2.1 NSLog

The simplest of these is the `NSLog(NSString *format, ...)` function. For example:

```
NSLog(@"Error occurred reading file at line %d.", lineNumber);
```

This would produce, on the console (stderr) of the application calling it, something like:

```
2004-05-08 22:46:14.294 SomeApp[15495] Error occurred reading file at line 20.
```

The behavior of this function may be controlled in two ways. First, the user default `GSLogSyslog` can be set to "`YES`", which will send these messages to the syslog on systems that support that (Unix variants). Second, the function GNUstep uses to write the log messages can be overridden, or the file descriptor the existing function writes to can be overridden:

```
  // these changes must be enclosed within a lock for thread safety
NSLock *logLock = GSLogLock();
[logLock lock];

  // to change the file descriptor:
_NSLogDescriptor = <fileDescriptor>;
  // to change the function itself:
_NSLog_printf_handler = <functionName>;

[logLock unlock];
```

Due to locking mechanisms used by the logging facility, you should protect these changes using the lock provided by `GSLogLock()` (see Chapter 8 [Threads and Run Control], page 83, on locking).

The `NSLog` function was defined in OpenStep and is also available in Mac OS X Cocoa, although the overrides described above may not be. The next set of logging facilities to be described are only available under GNUstep.

6.2.2 NSDebugLog, NSWarnLog

The facilities provided by the `NSDebugLog` and `NSWarnLog` families of functions support source code method name and line-number reporting and allow compile- and run-time control over logging level.

The `NSDebugLog` functions are enabled at compile time by default. To turn them off, set `'diagnose = no'` in your makefile, or undefine `GSDIAGNOSE` in your code before including `NSDebug.h`. To turn them off at runtime, call `[[NSProcessInfo processInfo] setDebugLoggingEnabled: NO]`. (An `NSProcessInfo` instance is automatically instantiated in a running GNUstep application and may be obtained by invoking `[NSProcessInfo processInfo]`.)

At runtime, whether or not logging is enabled, a debug log method is called like this:

```
NSDebugLLog(@"ParseError", @"Error parsing file at line %d.", lineNumber);
```

Here, the first argument to `NSDebugLog`, "`ParseError`", is a string *key* that specifies the category of message. The message will only actually be logged (through a call to `NSLog()`) if this key is in the set of active debug categories maintained by the `NSProcessInfo` object for the application. Normally, this list is empty. There are three ways for string keys to make it onto this list:

- Provide one or more startup arguments of the form `--GNU-Debug=<key>` to the program. These are processed by GNUstep and removed from the argument list before any user code sees them.
- Call `[NSProcessInfo debugSet]` at runtime, which returns an `NSMutableSet`. You can add (or remove) strings to this set directly.
- The `GNU-Debug` user default nay contain a comma-separated list of keys. However, note that `[NSUserDefaults standardUserDefaults]` must first be called before this will take effect (to read in the defaults initially).

While any string can be used as a debug key, conventionally three types of keys are commonly used. The first type expresses a "level of importance" for the message, for example, "Debug", "Info", "Warn", or "Error". The second type of key that is used is class name. The GNUstep Base classes used this approach. For example if you want to activate debug messages for the `NSBundle`" class, simply add '`NSBundle`' to the list of keys. The third category of key is the default key, '`dflt`'. This key can be used whenever the specificity of the other key types is not required. Note that it still needs to be turned on like any other logging key before messasges will actually be logged.

There is a family of `NSDebugLog` functions with slightly differing behaviors:

`NSDebugLLog(key, format, args,...)`
: Basic debug log function already discussed.

`NSDebugLog(format, args,...)`
: Equivalent to `NSDebugLLog` with key "dflt" (for default).

`NSDebugMLLog(key, format, args,...)`
: Equivalent to `NSDebugLLog` but includes information on which method the logging call was made from in the message.

`NSDebugMLog(format, args,...)`
: Same, but use 'dflt' log key.

`NSDebugFLLog(key, format, args,...)`
: As `NSDebugMLLog` but includes information on a function rather than a method.

`NSDebugFLog(format, args,...)`
: As previous but using 'dflt' log key.

The implementations of the `NSDebugLog` functions are optimized so that they consume little time when logging is turned off. In particular, if debug logging is deactivated at compile time, there is NO performance cost, and if it is completely deactivated at runtime, each call entails only a boolean test. Thus, they can be left in production code.

There is also a family of `NSWarn` functions. They are similar to the `NSDebug` functions except that they do not take a key. Instead, warning messages are shown by default unless they are disabled at compile time by setting `'warn = no'` or undefining `GSWARN`, or at runtime by *adding* "`NoWarn`" to `[NSProcessInfo debugSet]`. (Command-line argument `--GNU-Debug=NoWarn` and adding "NoWarn" to the `GNU-Debug` user default will also work.) `NSWarnLog()`, `NSWarnLLog()`, `NSWarnMLLog`, `NSWarnMLog`, `NSWarnFLLog`, and `NSWarnFLog` are all similar to their `NSDebugLog` counterparts.

6.2.3 Last Resorts: GSPrintf and fprintf

Both the `NSDebugLog` and the simpler `NSLog` facilities utilize a fair amount of machinery - they provide locking and timestamping for example. Sometimes this is not appropriate, or might be too heavyweight in a case where you are logging an error which might involve the application being in some semi-undefined state with corrupted memory or worse. You can use the `GSPrintf()` function, which simply converts a format string to UTF-8 and writes it to a given file:

```
GSPrintf(stderr, "Error at line %d.", n);
```

If even this might be too much (it uses the `NSString` and `NSData` classes), you can always use the C function `fprintf()`:

```
fprintf(stderr, "Error at line %d.", n);
```

Except under extreme circumstances, the preferred logging approach is either `NSDebugLog`/`NSWarnLog`, due the the compile- and run-time configurability they offer, or `NSLog`.

6.2.4 Profiling Facilities

GNUstep supports optional programmatic access to object allocation statistics. To initiate collection of statistics, call the function `GSDebugAllocationActive(BOOL active)` with an argument of "`YES`". To turn it off, call it with "`NO`". The overhead of statistics collection is only incurred when it is active. To access the statistics, use the set of `GSDebugAllocation...()` functions defined in `NSDebug.h`.

6.3 Assertions

Assertions provide a way for the developer to state that certain conditions must hold at a certain point in source code execution. If the conditions do not hold, an exception is automatically raised (and succeeding code in the block is not executed). This avoids an operation from taking place with illegal inputs that may lead to worse problems later.

The use of assertions is generally accepted to be an efficient means of improving code quality, for, like unit testing, they can help rapidly uncover a developer's implicit or mistaken assumptions about program behavior. However this is only true to the extent that you carefully design the nature and placement of your assertions. There is an excellent discussion of this issue bundled in the documentation with Sun's Java distribution.

6.3.1 Assertions and their Handling

Assertions allow the developer to establish that certain conditions hold before undertaking an operation. In GNUstep, the standard means to make an assertion is to use the `NSAssert` or `NSCAssert` macros. The general form of these macros is:

```
NSAssert(<boolean test>, <formatString>, <argumentsToFormat>);
```

For instance:

```
NSAssert(x == 10, "X should have been 10, but it was %d.", x);
```

If the test '`x == 10`' evaluates to `true`, `NSLog()` is called with information on the method and line number of the failure, together with the format string and argument. The resulting console message will look like this:

```
Foo.m:126  Assertion failed in Foo(instance), method Bar.  X should have been
10, but it was 5.
```

After this is logged, an exception is raised of type '`NSInternalInconsistencyException`', with this string as its description.

If you need to make an assertion inside a regular C function (not an Objective-C method), use the equivalent macro `NSCAssert()`, etc..

Note, you can completely disable assertions (saving the time for the boolean test and avoiding the exception if fails) by putting `#define NS_BLOCK_ASSERTIONS` before you include `NSException.h`.

6.3.2 Custom Assertion Handling

The aforementioned behavior of logging an assertion failure and raising an exception can be overridden if desired. You need to create a subclass of `NSAssertionHandler` and register an instance in each thread in which you wish the handler to be used. This is done by calling:

```
[[[NSThread currentThread] threadDictionary]
    setObject:myAssertionHandlerInstance forKey:N̈SAssertionHandler"];
```

See Chapter 8 [Threads and Run Control], page 83, for more information on what this is doing.

6.4 Comparison with Java

GNUstep's exception handling facilities are, modulo syntax, equivalent to those in Java in all but three respects:

- There is no provision for a "finally" block executed after either the main code or the exception handler code.
- You cannot declare the exception types that could be raised by a method in its signature. In Java this is possible and the compiler uses this to enforce that a caller should catch exceptions if they might be generated by a method.
- Correspondingly, there is no support in the Appendix A [GSDoc], page 99, for documenting exceptions potentially raised by a method. (This will hopefully be rectified soon.)

The logging facilities provided by `NSDebugLog` and company are similar to but a bit more flexible than those provided in the Java/JDK 1.4 logging APIs, which were based on the IBM/Apache Log4J project.

The assertion facilities are similar to but a bit more flexible than those in Java/JDK 1.4 since you can override the assertion handler.

7 Distributed Objects

Until now we have been concentrating on using the Objective-C language to create programs that execute in a single process. But what if you want your program to interact with objects in other processes, perhaps running on different machines?

As a simple example, we may have a client process that needs to access a telephone directory stored on a remote server. The client process could send a message to the server that contained a person's name, and the server could respond by returning that person's number.

The GNUstep base library provides a powerful set of classes that make this type of remote messaging not only possible, but easy to program. So what do these classes do and how can we use them? To answer that we must first look at the way code interacts with objects in a single process, and then look at how we can achieve the same interaction with objects that exist in different processes.

7.1 Object Interaction

To continue with the example above, if the telephone directory existed in the same process as the code that was accessing it, then a simple message would return the wanted telephone number.

```
NSString *wantedNumber = [telephoneDirectory teleNumber: personName];
```

Now object and method names just hold pointers to memory addresses. The code executed at run time in response to the `teleNumber` message is located at an address held by the name of the responding method (a variable), while data in the telephone directory is located at an address held by the `telephoneDirectory` variable.

In a single process these addresses can be accessed by the client code at run time, but if the telephone directory is located on a remote server, then the address of the remote object is not known in the client process (the `telephoneDirectory` object and its responding method are said to exist in a separate 'address space').

The Objective-C run-time library was not designed for this inter-process communication or 'remote messaging'.

7.2 The GNUstep Solution

GNUstep overcomes these limitations by providing you with classes that form what is known as a 'distributed objects' architecture that extends the capabilities of the run-time system.

With the addition of a few lines of code in the client and server programs, these extensions allow you to send a message to a remote process by constructing a simple Objective-C statement. In the telephone directory example, the statement to retrieve the telephone number would now look something like this:

```
NSString *wantedNumber = [proxyForDirectory teleNumber: personName];
```

Compare this to the original statement:

```
NSString *wantedNumber = [telephoneDirectory teleNumber: personName];
```

Notice that the only difference between the two statements is the name of the object receiving the message, i.e. `proxyForDirectory` rather than `telephoneDirectory`. GNUstep makes it as simple as this to communicate with an object in another process.

The variable `proxyForDirectory` is known as a 'proxy' for the remote `telephoneDirectory` object. A proxy is simply a substitute for the remote object, with an address in the 'address space' of the local client process, that receives messages and forwards them on to the remote server process in a suitably coded form.

Let us now take a look at the additional lines of code required to make this 'remote messaging' possible.

7.2.1 Code at the Server

In order to respond to client messages, the responding server object must be set as the 'root object' of an instance of the `NSConnection` class, and this `NSConnection` must be registered with the network by name. Making an object available to client processes in this way is known as 'vending' the object. The registered name for the `NSConnection` is used by the client when obtaining a proxy for the responding server object over the network.

The only other code you need to consider is the code that listens for incoming messages. This 'runloop', as it is known, is started by sending a `run` message to an instance of the `NSRunLoop` class. Since an `NSRunLoop` object is created automatically for each process, there is no need to create one yourself. Simply get the default runloop, which is returned by the `+currentRunLoop` class method.

When the runloop detects an incoming message, the message is passed to the root object of the `NSConnection`, which performs a method in response to the message and returns a variable of the appropriate type. The `NSConnection` manages all inter-process communication, decoding incoming messages and encoding any returned values.

The code to vend the `telephoneDirectory` object and start the runloop would look something like this:

```
/*
 * The main() function: Set up the program
 * as a 'Distributed Objects Server'.
 */
int main(void)
{
  /*
   * Remember, create an instance of the
   * NSAutoreleasePool class.
   */
  CREATE_AUTORELEASE_POOL(pool);

  /*
   * Get the default NSConnection object
   * (a new one is automatically created if none exists).
   */
  NSConnection *connXion = [NSConnection defaultConnection];

  /*
   * Set the responding server object as
   * the root object for this connection.
   */
```

```
    [connXion setRootObject: telephoneDirectory];

    /*
     * Try to register a name for the NSConnection,
     * and report an error if this is not possible.
     */
    if ([connXion registerName: @"DirectoryServer"] == NO)
    {
      NSLog(@"Unable to register as 'DirectoryServer'");
      NSLog(@"Perhaps another copy of this program is running?");
      exit(1);
    }

    /* Start the current runloop. */
    [[NSRunLoop currentRunLoop] run];

    /* Release the pool */
    RELEASE(pool);
    return 0;
}
```

These additional lines of code turn a program into a distributed objects server, ready to respond to incoming client messages.

7.2.2 Code at the Client

At the client, all you need do is obtain a proxy for the responding server object, using the name that was registered for the `NSConnection` at the server.

```
    /* Create an instance of the NSAutoreleasePool class */
    CREATE_AUTORELEASE_POOL(pool);

    /* Get the proxy */
    id proxy = [NSConnection
      rootProxyForConnectionWithRegisteredName: registeredServerName];

    /* The rest of your program code goes here */

    /* Release the pool */
    RELEASE(pool);
```

The code that obtains the proxy automatically creates an NSConnection object for managing the inter-process communication, so there is no need to create one yourself.

The above example serves to establish a secure connection between processes which are run by the same person and are both on the same host.

If you want your connections to work between different host or between programs being run by different people, you do this slightly differently, telling the system that you want to use 'socket' ports, which make TCP/IP connections over the network.

```
int main(void)
```

```
{
  CREATE_AUTORELEASE_POOL(pool);

  /*
   * Create a new socket port for your connection.
   */
  NSSocketPort *port = [NSSocketPort port];

  /*
   * Create a connection using the socket port.
   */
  NSConnection *connXion = [NSConnection connectionWithReceivePort: port
  sendPort: port];

  /*
   * Set the responding server object as
   * the root object for this connection.
   */
  [connXion setRootObject: telephoneDirectory];

  /*
   * Try to register a name for the NSConnection,
   * and report an error if this is not possible.
   */
  if ([connXion registerName: @"DirectoryServer"
    withNameServer: [NSSocketPortNameServer sharedInstance]] == NO)
  {
    NSLog(@"Unable to register as 'DirectoryServer'");
    NSLog(@"Perhaps another copy of this program is running?");
    exit(1);
  }

  [[NSRunLoop currentRunLoop] run];

  RELEASE(pool);
  return 0;
}
```

In the above example, we specify that the socket port name server is used to register the name for the connection ... this makes the connection name visible to processes running on other machines.

The client side code is as follows

```
  /* Create an instance of the NSAutoreleasePool class */
  CREATE_AUTORELEASE_POOL(pool);

  /* Get the proxy */
  id proxy = [NSConnection
```

```
  rootProxyForConnectionWithRegisteredName: registeredServerName
  host: hostName
  usingNameServer: [NSSocketPortNameServer sharedInstance]];

/* The rest of your program code goes here */

/* Release the pool */
RELEASE(pool);
```

If the *hostName* in this statement is 'nil' or an empty string, then only the local host will be searched to find the *registeredServerName*. If *hostName* is `"*"`, then all hosts on the local network will be searched.

In the telephone directory example, the code to obtain the proxy from any host on the network would be:

```
id proxyForDirectory = [NSConnection
  rootProxyForConnectionWithRegisteredName: @"DirectoryServer"
  host: @"*"
  usingNameServer: [NSSocketPortNameServer sharedInstance]];
```

With this additional line of code in the client program, you can now construct a simple Objective-C statement to communicate with the remote object.

```
NSString *wantedNumber = [proxyForDirectory teleNumber: personName];
```

7.2.3 Using a Protocol

A client process does not need to know the class of a remote server object to avoid run-time errors, it only needs to know the messages to which the remote object responds. This can be determined by the client at run-time, by asking the server if it responds to a particular message before the message is sent.

If the methods implemented at the server are stated in a formal protocol, then the client can ask the server if it conforms to the protocol, reducing the network traffic required for the individual message/response requests.

A further advantage is gained at compile time, when the compiler will issue a warning if the server fails to implement any method declared in the protocol, or if the client contains any message to which the server cannot respond.

The protocol is saved to a header file and then included in both client and server programs with the usual compiler `#include` directive. Only the server program needs to implement the methods declared in the protocol. To enable compiler checking in the client program, extend the type declaration for the proxy to this protocol, and cast the returned proxy object to the same extended type.

In the telephone directory example, if the declared protocol was `TelephoneDirectory`, declared in header file `protocolHeader.h`, then the client code would now look like this:

```
#include "protocolHeader.h";

/* Extend the type declaration */
id<TelephoneDirectory> proxyForDirectory;
```

```
  /* Cast the returned proxy object to the extended type */
  proxyForDirectory = (id<TelephoneDirectory>) [NSConnection
    rootProxyForConnectionWithRegisteredName: @"DirectoryServer"
    usingNameServer: [NSSocketPortNameServer sharedInstance]];
```

Since class names and protocol names do not share the same 'address space' in a process, the declared protocol and the class of the responding server object can share the same name, making code easier to understand.

For example, `proxyForDirectory` at the client could be a proxy for an instance of the `TelephoneDirectory` class at the server, and this class could implement the `TelephoneDirectory` protocol.

7.2.4 Complete Code for Telephone Directory Application

Here we provide the rest of the code needed for client and server to actually run the above example.

Code At Server

```
#include <Foundation/Foundation.h>

/* Include the TelephoneDirectory protocol header file */
#include "TelephoneDirectory.h"

/*
 * Declare the TelephoneDirectory class that
 * implements the 'teleNumber' instance method.
 */
@interface TelephoneDirectory : NSObject <TelephoneDirectory>
@end

/*
 * Define the TelephoneDirectory class
 * and the instance method (teleNumber).
 */
@implementation TelephoneDirectory : NSObject
- (char *) teleNumber: (char *) personName
{
  if (strcmp(personName, "Jack") == 0) return " 0123 456";
  else if (strcmp(personName, "Jill") == 0) return " 0456 789";
  else return " Number not found";
}
@end

 /* main() function: Set up the program  as a 'Distibuted Objects Server'. */
 /*   [use code from server example above ...] */
```

Code at Client

```
#include <Foundation/Foundation.h>
```

```
/* Include the TelephoneDirectory protocol header file */
#include "TelephoneDirectory.h"

/*
 * The main() function: Get the telephone number for
 * 'personName' from the server registered as 'DirectoryServer'.
 */
int main(int argc, char *argv[])
{
  char *personName = argv[1];
  char *returnedNumber;
  id<TelephoneDirectory> proxyForDirectory;
  CREATE_AUTORELEASE_POOL(pool);

  /* Acquire the remote reference. */
  proxyForDirectory = (id<TelephoneDirectory>) [NSConnection
    rootProxyForConnectionWithRegisteredName: @"DirectoryServer"
    host: @"*"
    usingNameServer: [NSSocketPortNameServer sharedInstance]];

  if (proxyForDirectory == nil)
    printf("\n** WARNING: NO CONNECTION TO SERVER **\n");
  else printf("\n** Connected to server **\n");

  if (argc == 2) // Command line name entered
  {
    returnedNumber = (char *)[proxyForDirectory teleNumber: personName];
    printf("\n%s%s%s%s%s\n", "** (In client) The telephone number for ",
           personName, " is:",
           returnedNumber, "  **");
  }
  else printf("\n** No name entered **\n");
  printf("\n%s\n\n", "** End of client program **");
  RELEASE(pool);
  return 0;
}
```

To get this running, all you need do is create two directories, one for the client and one for the server. Each directory will hold a makefile, the client or server source code, and a copy of the protocol header file. When the files compile, first run the server and then the client. You can try this on the same machine, or on two different machines (with GNUstep installed) on the same LAN. What happens when you run the client without the server? How would you display a "No Server Connection" warning at the client?

7.2.5 GNUstep Distributed Objects Name Server

You might wonder how the client finds the server, or, rather, how it finds the directory the server lists itself in.

For the default connection type (a connection only usable on the local host between processes run by the same person), a private file (or the registry on ms-windows) is used to hold the name registration information.

For connections using socket ports to communicate between hosts, an auxiliary process will automatically be started on each machine, if it isn't running already, that handles this, allowing the server to register and the client to send a query behind the scenes. This *GNUstep Distributed Objects Name Server* runs as '`gdomap`' and binds to port 538. See the manual page or the HTML "GNUstep Base Tools" documentation (`../../Tools/Reference/index.html`) for further information.

7.2.6 Look Ma, No Stubs!

One difference you may have noticed in the example we just looked at from other remote method invocation interfaces such as CORBA and Java RMI was that there are *no stub classes*. The source of this great boon is described at the end of this chapter: Chapter 7 [Language Support for Distributed Objects], page 63.

7.3 A More Involved Example

Now we will look at an example called GameServer that uses distributed objects in a client/server game.

Actually the game itself is not implemented, just its distributed support structure, and while the code to vend an object and connect to a remote process is similar to that already shown, the code does show a number of additional techniques that can be used in other client/server programs. Here are the requirements we will implement:

- When the client attempts to join the game, the server checks that the client is entitled to join, based on the last time the client played. The rule is: if the client lost the last game, then they cannot re-play for the next 2 hours; but if the client won the last game, then they can re-play the game at any time (a reward for winning).

- The server also makes sure the client is not already connected and playing the game (i.e. they cannot play two games at the same time - that would be cheating).

- In addition to a proxy for the server being obtained at the client, a proxy for the client is received at the server. This allows two-way messaging, where the client can send messages to the server and the server can send messages to the client (e.g. the state of play may be affected by the actions of other players, or by other events at the server).

  Two protocols will therefore be required, one for the methods implemented at the server and one for those implemented at the client.

Have a look at the program code in the following sections and added comments. Can you work out what is happening at the server and client? If you have any difficulties then refer to the relevant sections in this manual, or to class documentation here (`../Reference/index.html`) or at the Apple web site.

7.3.1 Protocol Adopted at Client

We have chosen `GameClient` as the name of both the protocol adopted at the client and the class of the responding client object. The header file declaring this protocol will simply declare the methods that the class must implement.

```
@protocol GameClient
- (void) clientMessage: (bycopy NSString *)theMessage;
- (int) clientReply;

// Other methods would be added that
// reflect the nature of the game.

@end
```

The protocol will be saved as `GameClient.h`.

7.3.2 Protocol Adopted at Server

We have chosen `GameServer` as the name of both the protocol adopted at the server and the class of the responding server object. The header file declaring this protocol will simply declare the methods that the class must implement.

```
@protocol GameServer
- (BOOL) mayJoin: (id)client asPlayer: (bycopy NSString*)name;
- (int) startGame: (bycopy NSString*)name;
- (BOOL) endGame: (bycopy NSString*)name;

// Other methods would be added that
// reflect the nature of the game.

@end
```

The protocol will be saved as `GameServer.h`.

7.3.3 Code at the Client

The client code contains the `main` function and the `GameClient` class declaration and implementation.

The `main()` function attempts to connect to the server, while the `GameClient` class adopts the `GameClient` protocol.

```
#include <Foundation/Foundation.h>
#include "GameServer.h"
#include "GameClient.h"

/*
 * GameClient class declaration:
 * Adopt the GameClient protocol.
 */
@interface GameClient : NSObject <GameClient>
@end
```

```
/*
 * GameClient class implementation.
 */
@implementation GameClient

/*
 * Implement clientMessage: as declared in the protocol.
 * The method simply prints a message at the client.
 */
- (void) clientMessage: (NSString*)theMessage
{
  printf([theMessage cString]);
}

/*
 * Implement clientReply: as declared in the protocol.
 * The method simply returns the character entered
 * at the client keyboard.
 */
- (int) clientReply
{
  return getchar();
}
@end  // End of GameClient class implementation.

/*
 * The main function of the client program.
 */
int main(int argc, char **argv)
{
  CREATE_AUTORELEASE_POOL(pool);
  id<GameServer> server;
  int result;
  NSString *name;
  id client;

  /*
   * The NSUserName() function returns the name of the
   * current user, which is sent to the server when we
   * try to join the game.
   */
  name = NSUserName();

  /*
   * Create a GameClient object that is sent to
   * the server when we try to join the game.
   */
```

```
    client = AUTORELEASE([GameClient new]);

    /*
     * Try to get a proxy for the root object of a server
     * registered under the name 'JoinGame'. Since the host
     * is '*', we can connect to any server on the local network.
     */
    server = (id<GameServer>)[NSConnection
      rootProxyForConnectionWithRegisteredName: @"JoinGame"
      host: @"*"
      usingNameServer: [NSSocketPortNameServer sharedInstance]];
    if (server == nil)
      {
        printf("\n** No Connection to GameServer **\n");
        result = 1;
      }

    /*
     * Try to join the game, passing a GameClient object as
     * the client, and our user-name as name. The 'client'
     * argument will be received as a proxy at the server.
     */
    else if ([server mayJoin: client asPlayer: name] == NO)
      {
        result = 1; // We cannot join the game.
      }
    else
      {
        /*
         * At this point, we would actually start to play the game.
         */
        [server startGame: name]; // Start playing game.
        [server endGame: name]; // Finally end the game.
        result = 0;
      }
    RELEASE(pool);
    return result;
  }
```

To summarise the code at the client:

- We obtained a proxy for the server and can now communicate with the server using the methods declared in the `GameServer` protocol.

- We passed a `GameClient` object and our user-name to the server (the `GameClient` object is received as a proxy at the server). The server can now communicate with the client using the methods declared in the `GameClient` protocol.

- When the game is in progress, the server can alter the state of the client object to reflect the success of the player.

7.3.4 Code at the Server

The server code contains the `main` function and the `GameServer` class declaration and implementation.

The `main()` function vends the server's root object and starts the runloop, while the `GameServer` class adopts the `GameServer` protocol. The class also implements methods that initialise and deallocate the root object's instance variables (dictionaries that hold player information).

```
#include <Foundation/Foundation.h>
#include "GameServer.h"
#include "GameClient.h"

/*
 * GameServer class declaration:
 * Adopt the GameServer protocol and declare
 * GameServer instance variables.
 */
@interface GameServer : NSObject <GameServer>
{
  NSMutableDictionary *delayUntil; // Delays to re-joining GameServer.
  NSMutableDictionary *currentPlayers; // Proxies to each client.
  NSMutableDictionary *hasWon; // Success in game for each player.
}
@end

/*
 * GameServer class implementation.
 */
@implementation GameServer

/* Initialise GameServer's instance variables. */
- (id) init
{
  self = [super init];
  if (self != nil)
    {
      /*
       * Create a dictionary for a maximum of
       * 10 named players that will hold a
       * re-joining time delay.
       */
      delayUntil = [[NSMutableDictionary alloc]
                     initWithCapacity: 10];
      /*
```

```
        * Create a dictionary that will hold the
        * names of these players and a proxy for
        * the received client objects.
        */
       currentPlayers = [[NSMutableDictionary alloc]
                         initWithCapacity: 10];

       /*
        * Create a dictionary that will record
        * a win for any of these named players.
        */
       hasWon = [[NSMutableDictionary alloc]
                   initWithCapacity: 10];
     }
   return self;
}

/* Release GameServer's instance variables. */
- (void) dealloc
{
  RELEASE(delayUntil);
  RELEASE(currentPlayers);
  RELEASE(hasWon);
  [super dealloc];
}

/*
 * Implement mayJoin:: as declared in the protocol.
 * Adds the client to the list of current players.
 * Each player is represented at the server by both
 * name and by proxy to the received client object.
 * A player cannot join the game if they are already playing,
 * or if joining has been delayed until a later date.
 */
- (BOOL) mayJoin: (id)client asPlayer: (NSString*)name
{
  NSDate  *delay; // The time a player can re-join the game.
  NSString *aMessage;

  if (name == nil)
    {
      NSLog(@"Attempt to join nil user");
      return NO;
    }

  /* Has the player already joined the game? */
  if ([currentPlayers objectForKey: name] != nil)
```

```
    {
      /* Inform the client that they cannot join. */
      aMessage = @"\nSorry, but you are already playing GameServer!\n";
      [client clientMessage: aMessage];
      return NO;
    }

  /* Get the player's time delay for re-joining. */
  delay = [delayUntil objectForKey: name];

  /*
   * Can the player join the game? Yes if there is
   * no restriction or if the time delay has passed;
   * otherwise no, they cannot join.
   */
  if (delay == nil || [delay timeIntervalSinceNow] <= 0.0)
    {
      /* Remove the old restriction on re-joining the game. */
      [delayUntil removeObjectForKey: name];

      /* Add the player to the list of current players. */
      [currentPlayers setObject: client forKey: name];
      [hasWon setObject: @"NO" forKey: name]; // They've not won yet.

      /* Inform the client that they have joined the game. */
      aMessage = @"\nWelcome to GameServer\n";
      [client clientMessage: aMessage];
      return YES;
    }
  else
    {
      /* Inform the client that they cannot re-join. */
      aMessage = @"\nSorry, you cannot re-join GameServer yet.\n";
      [client clientMessage: aMessage];
      return NO;
    }
}

/*
 * Implement startGame: as declared in the protocol.
 * Simply ask the player if they want to win, and get
 * there reply.
 */
- (int) startGame: (NSString *)name
{
  NSString *aMessage;
  id client;
```

```
  int reply;

  client = [currentPlayers objectForKey: name];

  aMessage = @"\nDo you want to win this game? (Y/N <RET>) ... ";
  [client clientMessage: aMessage];

  reply = [client clientReply];
  if (reply == 'y' || reply == 'Y')
    [hasWon setObject: @"YES" forKey: name]; // They win.
  else [hasWon setObject: @"NO" forKey: name]; // They loose.
  return 0;
}

/*
 * Implement endGame: as declared in the protocol.
 * Removes a player from the game, and either sets
 * a restriction on the player re-joining or removes
 * the current restriction.
 */
- (BOOL) endGame: (NSString*)name
{
  id client;
  NSString *aMessage, *yesOrNo;
  NSDate *now, *delay;
  NSTimeInterval twoHours = 2 * 60 * 60; // Seconds in 2 hours.

  if (name == nil)
    {
      NSLog(@"Attempt to end nil user");
      return NO;
    }

  now = [NSDate date];
  delay = [now addTimeInterval: twoHours];
  client = [currentPlayers objectForKey: name];
  yesOrNo = [hasWon objectForKey: name];

  if ([yesOrNo isEqualToString: @"YES"]) // Has player won?
    {
      /*
       * Player wins, no time delay to re-joining the game.
       * Remove any re-joining restriction and send
       * a message to the client.
       */
      [delayUntil removeObjectForKey: name];
      aMessage = @"\nWell played: you can re-join GameServer at any time.\n";
```

```
        [client clientMessage: aMessage];

      }
    else // Player lost
      {
        /*
         * Set a time delay for re-joining the game,
         * and send a message to the client.
         */
        [delayUntil setObject: delay forKey: name];
        aMessage = @"\nYou lost, but you can re-join GameServer in 2 hours.\n";
        [client clientMessage: aMessage];
      }

    /* Remove the player from the current game. */
    [currentPlayers removeObjectForKey: name];
    [hasWon removeObjectForKey: name];
    return YES;
  }

  @end  // End of GameServer class implementation

  /*
   * The main function of the server program simply
   * vends the root object and starts the runloop.
   */
  int main(int argc, char** argv)
  {
    CREATE_AUTORELEASE_POOL(pool);
    GameServer *server;
    NSSocketPort *port;
    NSConnection *connXion;

    server = AUTORELEASE([GameServer new]);
    port = [NSSocketPort port];
    connXion = [NSConnection connectionWithReceivePort: port sendPort: port];
    [connXion setRootObject: server];
    [connXion registerName: @"JoinGame"
      withNameServer: [NSSocketPortNameServer sharedInstance]];
    [[NSRunLoop currentRunLoop] run];
    RELEASE(pool);
    return 0;
  }
```

To summarise the code at the server:

- We vend the server's root object and start a runloop, allowing clients to connect with

the server.

- When we receive a proxy for a client object, we communicate with that client using methods declared in the `ClientServer` protocol.

- We create three dictionary objects, each referenced by player name. `currentUsers` holds proxies for each of the current players; `delayUntil` holds times when each player can re-join the game; and `hasWon` holds a string for each player, which is set to `"YES"` if the player wins.

- When the game is in progress, the server can alter the state of each client object to reflect the success of each player.

I hope you managed to understand most of the code in this example. If you are reading the on-screen version, then you can copy and paste the code to suitably named files, create makefiles, and then make and run each program. What message is displayed if you immediately try to re-join a game after losing? And after winning?

Exercise: Modify the server code so that the server records the number of wins for each player, and displays this information at both the start and end of each game.

7.4 Language Support for Distributed Objects

Objective-C provides special 'type' qualifiers that can be used in a protocol to control the way that message arguments are passed between remote processes, while at run time, the run-time system transparently uses what is known as 'forward invocation' to forward messages to a remote process. (See Chapter 5 [Forwarding], page 49.)

7.4.1 Protocol Type Qualifiers

When message arguments are passed by value then the receiving method can only alter the copy it receives, and not the value of the original variable. When an argument is passed by reference (as a pointer), the receiving method has access to the original variable and can alter that variable's data. In this case the argument is effectively passed 'in' to the method, and then passed 'out' of the method (on method return).

When an argument is passed by reference to a remote object, the network must handle this two-way traffic, whether or not the remote object modifies the received argument.

Type qualifiers can be used in a protocol to control the way these messages are handled, and to indicate whether or not the sending process will wait for the remote process to return.

- The **oneway** qualifier is used in conjunction with a `void` return type to inform the run-time system that the sending process does not need to wait for the receiving method to return (known as 'asynchronous' messaging). The protocol declaration for the receiving method would look something like this:

```
- (oneway void)noWaitForReply;
```

- The **in, out** and **inout** qualifiers can be used with pointer arguments to control the direction in which an argument is passed. The protocol declaration for the receiving methods would look something like this:

```
/*
 * The value that 'number' points to will be passed in to the remote process.
 * (No need to return the argument's value from the remote process.)
 */
- setValue: (in int *)number;

/*
 * The value that 'number' points to will be passed out of the remote process.
 * (No need to send the argument's value to the remote process.)
 */
- getValue: (out int *)number;

/*
 * The value that 'number' points to is first passed in to the remote
 * process, but will eventually be the value that is passed out of the
 * remote process. (Send and return the argument's value.)
 */
- changeValue: (inout int *)number;
```

  Passing of arguments by reference is very restricted in Objective-C. it applies only to pointers to C data types, not to objects, and except for the special case of a pointer to a nul terminated C string (**char***) the pointer is assumed to refer to a single data item of the specified type.

```
/*
 * A method passing an unsigned short integer by reference.
 */
- updateCounter: (inout unsigned shortn *)value;

/*
 * A method passing a structure by reference.
 */
- updateState: (inout struct stateInfo *)value;

/*
 * As a special case, a char (or equivalent typedef) passed by reference
 * is assumed to be a nul terminated string ... there is no way to pass
 * a single character by reference:
 */
- updateBuffer: (inout char *)str;
```

- The **bycopy** and **byref** qualifiers can be used in a protocol when the argument or return type is an object.

An object is normally passed by reference and received in the remote process as a proxy. When an object is passed by copy, then a copy of the object will be received in the remote process, allowing the remote process to directly interact with the copy. Protocol declarations would look something like this:

```
/*
 * Copy of object will be received in the remote process.
 */
- sortNames: (bycopy id)listOfNames;

/*
 * Copy of object will be returned by the remote process.
 */
- (bycopy id)returnNames;
```

By default, large objects are normally sent **byref**, while small objects like `NSStrings` are normally sent **bycopy**, but you cannot rely on these defaults being adopted and should explicitly state the qualifier in the protocol.

The **bycopy** qualifier can also be used in conjunction with the **out** qualifier, to indicate that an object will be passed **out** of the remote process by copy rather than by proxy (no need to send the object).

```
/*
 * The object will not be received in the remote process, but the object
 * will be returned bycopy.
 */
- sortAndReturn: (bycopy out id *)listOfNames;
```

You should be aware that some classes ignore the **bycopy** qualifier and the object will be sent by reference. The **bycopy** qualifier will also be ignored if the remote process does not have the class of the object in its address space, since an object's instance variables are accessed through the object's methods.

When a copy of an object is sent to a remote process, only the object's instance variables are sent and received (an object's methods exist in the address space of the object's class, not in the address space of the individual object).

7.4.2 Message Forwarding

If you have used other remote invocation mechanisms such as CORBA or Java RMI, you may have noticed a big difference from these in the GNUstep Distributed Object paradigm – there are no "stub" classes, either on the client or the server. This tremendously simplifies

the use of remote invocation and is possible due to the Objective-C message-forwarding facility (Chapter 5 [Forwarding], page 49).

In GNUstep, there are proxies on the client and server side that handle network communications and serialization/deserialization of arguments and return values just as in CORBA and RMI, but when it comes to responding to the client and server protocol method calls themselves, they are intercepted through the use of the `forwardInvocation:` method, where they can be passed on to the registered client and server objects through the ordinary Objective-C message sending mechanism.

7.5 Error Checking

When dealing with distributed objects your code must be able to handle the following situations: failure to vend the server object, exceptions raised at run-time, and failure of the network connection.

7.5.1 Vending the Server Object

When vending the server object, your code must be able to handle the situation in which the network does not accept the proposed registered name for the server.

7.5.2 Catching Exceptions

There are two situations to consider.

- An `NSPortTimeoutException` is raised.

  This exception is raised if a message takes too long to arrive at the remote process, or if a reply takes too long to return. This will happen if the remote process is busy, has hung, or if there is a problem with the network. The best way to handle the exception is to close the connection to the remote process.

- An exception is raised in the remote process while the remote process is executing a method.

  In most cases you can deal directly with these exceptions in the process in which they were raised; i.e. without having to consider the network connection itself.

7.5.3 The Connection Fails

You can register an observer object to receive a notification, in the form of a `connectionDidDie:` message, when a registered connection fails. The argument to this message will be an `NSNotification` object that returns the failed connection when it receives an `object` message. See Chapter 8 [Event-Based Communications], page 83, for more information on notifications.

To receive this 'notification' the observer must implement the `connectionDidDie:` method, but can be an instance of any class. The observer can then handle the failure gracefully, by releasing any references to the failed connection and releasing proxies that used the connection. Registering an object to receive this notification is described in more detail in the `NSConnection` class documentation.

8 Base Library

The GNUstep Base library is an implementation of the OpenStep *Foundation*, a nongraphical API supporting for data management, network and file interaction, date and time handling, and more. Much of the API consists of classes with defined methods, but unlike many "class libraries" it also includes functions and macros when these are more appropriate to the functionality.

Note that many other APIs developed subsequently to OpenStep are also called "Foundation" – the Java standard classes and the Microsoft Windows C++ class library are two prominent examples. In OpenStep, however, the term only applies to a non-graphical library; the graphical component is referred to as the *Application Kit*, or "AppKit" for short.

Although the OpenStep API underwent several refactorings subsequent to its first release as part of NeXTstep, deprecated and superseded classes and functions have not been retained. Therefore the library still boasts a minimal footprint for its functionality.

In some cases, GNUstep has supplemented the OpenStep API, not to provide alternative means of achieving the same goals, but to add new functionality, usually relating to technology that did not exist when the OpenStep specification was finalized, but has not, for whatever reason, been added by Apple to the Cocoa APIs. These additions are called, appropriately enough, the *Base Additions* library, and include classes, functions, and macros. XML parsing facilities, for example, are provided as part of this library.

In addition, methods are sometimes added to Foundation classes. These are specially marked in the documentation and can even be excluded at compile time (a warning will be generated if you try to use them) if you are writing code intended to be ported to OpenStep or Cocoa compliant systems. In addition, Cocoa has made additions to OpenStep and these are marked as "MacOS-X". For information on how to set compile flags, see Appendix E [Compliance to Standards], page 111.

In deciding whether to use a given API, you need to weigh the likelihood you will need to port the application to a platform where it will not be available, and in that case, how much effort would be required to do without the API. If you are aiming for full portability from the start (only a recompile needed), then you should of course avoid APIs that will not be available. However in other cases it can be better to use whichever APIs are best suited initially so that early development and debugging will be as efficient as possible – as long as major redesign would not be required to later avoid these APIs.

Below, the Base and Base Additions APIs are covered in overview fashion, organized according to functionality. For detailed documentation on individual classes and functions, you should consult the GSDoc API references for Base (`../Reference/index.html`) and Base Additions (`../../BaseAdditions/Reference/index.html`). It may be helpful, when reading this chapter, to keep a copy of this open in another browser window for reference.

8.1 Copying, Comparing, Hashing Objects

Often in object-oriented code you need to make a duplicate copy of an existing object. The `NSObject` method `-(id) copy` provides a standard means of acquiring a copy of the object. The *depth* of the copy is not defined. That is, if an object has instance variables or other references to other objects, they may either themselves be copied or just the references to them will be copied. The root class `NSObject` does *not* implement the copy method

directly; instead it calls the `-copyWithZone` method, which is the sole method defined in the *NSCopying* informal protocol. `NSObject` does not implement this protocol. If you want objects of your class to support copying, you must implement this method yourself. If it is not implemented, the `-copy` method will raise an exception if you call it.

There is a related method `-(id) mutableCopy` (and an *NSMutableCopying* informal protocol with a `mutableCopyWithZone` method) which will be explained in the following section.

GNUstep, largely via the `NSObject` class, provides a basic framework for comparing objects for equality and ordering, used for sorting, indexing, and other programming tasks. These operations are also used in several crucial places elsewhere within the base library itself. For example, containers such as lists, sets, and hash maps are discussed in the next section utilize these methods.

The `- (BOOL) isEqual` method in `NSObject` is useful when you want to compare objects with one another:

```
if ([anObject isEqual: anotherObject])
  {
    // do something ...
  }
```

The default implementation returns `YES` only if the two objects being compared are the exact same object (which is the same as the result that would be returned using '`==`' to perform the comparison). Sometimes it is useful to have two objects to be equal if their internal state is the same, as reflected in instance variables, for example. In this case, you can override `isEqual` in your class to perform such a comparison.

The `-(unsigned int)hash` method is useful for indexing objects, and should return the same value for two objects of the same class that `isEqual` each other. The same reasoning applies as for the `isEqual` method – if you want this to depend on internal state rather than the identity of the object itself, override it. The default `hash` value is based on the memory address occupied by the object.

The `-(NSComparisonResult) compare: (id)object` method is used in Cocoa for comparing objects. It should return `NSOrderedAscending` if the receiver is less than the argument, `NSOrderedDescending` if it is greater, otherwise `NSOrderedSame`. Note that this is not meaningful for many types of objects, and is actually deprecated in GNUstep for this reason.

The `-(NSString *) description` method in `NSObject` returns a short description of the object, often used for debugging. The default implementation lists the object's class and memory location. If you want other information you can override it.

The methods discussed in this section are all very similar to counterparts in Java: the `equals` and `hashCode` methods, and the *Comparable* interface.

8.2 Object Containers

GNUstep defines three major utility classes for holding collections of other objects. `NSArray` is an ordered collection of objects, each of which may occur in the collection multiple times. `NSSet` is an unordered collection of unique objects (according to `isEqual` and/or `hash`). `NSDictionary` is an unordered collection of key-value pairs. The keys form a set (and must be unique), however there are no restrictions on the collection of values. The `-hash` and

`-isEqual` `NSObject` methods discussed above are used by collection instances to organize their members. All collections `retain` their members (see Chapter 3 [Objects], page 23).

Unlike container APIs in some other languages, notably Java, instances of these GNUstep classes are all *immutable* – once created, you cannot add or remove from them. If you need the ability to make changes (often the case), use the mutable classes `NSMutableArray`, `NSMutableSet`, and `NSMutableDictionary`. The `-mutableCopy` method mentioned in the previous section will return the mutable version of a container regardless of whether the original was mutable or not. Likewise, the `-copy` method returns an immutable version. You should generally use immutable variants of objects when you don't need to modify them, because their implementations are more efficient. Often it is worthwhile to convert a mutable object that has just been built into an immutable one if it is going to be heavily accessed.

Also unlike container objects in Java, GNUstep containers possess utility methods. For example, Arrays can sort their members, or send a message to each member individually (like the `map` function in Lisp). Sets can determine whether they are equal to or subsets of other sets. Dictionaries can save to and restore themselves from specially formatted files.

In addition to the three container types already mentioned, there is a fourth, `NSCountedSet`. This is an unordered collection whose elements need not be unique, however the number of times a given unique element has been added is tracked. This behavior is also known as *bag* semantics.

All collection classes support returning an `NSEnumerator` object which will enumerate over the elements of the collection. Note that if a mutable collection is modified while an enumerator is being used, the results are not defined.

Collections do not allow `nil` values or keys, but you can explicitly represent a nil object using the special `NSNull` class. You simply use the singleton returned from `[NSNull null]`.

The four container types just described handle objects, but not primitives such as `float` or `int`. For this, you must use an `NSHashTable` or `NSMapTable`. Despite their names, these are not classes, but data types. A set of functions is defined for dealing with them. Each can store and retrieve arbitrary pointers keyed by other arbitrary pointers. However you are responsible for implementing the hashing yourself. To create an `NSHashTable`, use the function `NSCreateHashtable`. `NSHashInsert` and `NSHashGet` are the major functions, but there are many others. There is a mostly parallel but more sophisticated set of functions dealing with `NSMapTables`.

8.3 Data and Number Containers

The data containers discussed in the previous section, with the exception of `NSHashTable` and `NSMapTable`, can store objects, but not primitive types such as `int`s or `float`s. The `NS...Table` structures are not always appropriate for a given task. For this case, GNUstep offers two alternatives.

8.3.1 NSData

The `NSData` and `NSMutableData` classes manage a buffer of bytes as an object. The contents of the buffer can be anything that can be stored in memory, a 4-dimensional array of `double` for example (stored as a linear sequence). Optionally, objects of these classes can take care

of the memory management for the buffer, growing it as needed and freeing it when they are released.

8.3.2 NSValue

The `NSValue` class can wrap a single primitive value as an object so it can be used in the containers and other places where an object reference is needed. Once initialized, an `NSValue` is immutable, and there is no `NSMutableValue` class. You initialize it by giving it a pointer to the primitive value, and you should be careful this does not get freed until after the `NSValue` is no longer used. You can specify to the `NSValue` what type the primitive is so this information can be accessed later:

```
int n = 10;
NSValue *theValue = [NSValue value: &n withObjCType: @encode(int)];
  // ...
int *m = (int *) [theValue pointerValue];
```

Here, `@encode` is a compile-time operator that converts the data type into a string (char *) code used at runtime to refer to the type. Object ids can also be stored within `NSValue`s if desired. Note that in the above case, the `NSValue` will be pointing to invalid data once the local variable *`n`* goes out of scope.

If you want to wrap `int` or other numeric values, you should use `NSNumber` (a subclass of `NSValue`) instead. This maintains its own copy of the data and provides convenience methods for accessing the value as a primitive.

```
int n = 10;
NSNumber *theNumber = [NSNumber numberWithInt: n];
  // ...
int m = [theNumber intValue];
float f = [theNumber floatValue];  // this is also valid
```

Notice that *`n`*'s value is used in the initialization, not a pointer to it.

8.3.3 NSNumber

`NSNumber` has a subclass called `NSDecimalNumber` that implements a number of methods for performing decimal arithmetic to much higher precision than supported by ordinary `long double`. The behavior in terms of rounding choices and exception handling may be customized using the `NSDecimalNumberHandler` class. Equivalent functionality to the `NSDecimalNumber` class may be accessed through functions, mostly named `NSDecimalXXX`. Both the class and the functions use a structure also called `NSDecimal`:

```
typedef struct {
  signed char exponent;   // Signed exponent - -128 to 127
  BOOL isNegative;         // Is this negative?
  BOOL validNumber;        // Is this a valid number?
  unsigned char length;     // digits in mantissa.
  unsigned char  cMantissa[2*NSDecimalMaxDigit];
}
```

Instances can be initialized using the `NSDecimalFromString(NSString *)` function.

8.3.4 NSRange, NSPoint, NSSize, NSRect

There are also a few types (not classes) for representing common composite structures. `NSRange` represents an integer interval. `NSPoint` represents a floating point 2-d cartesian location. `NSSize` represents a 2-d floating point extent (width and height). `NSRect` contains a lower-left point and an extent. A number of utility functions are defined for handling rectangles and points.

8.4 Date/Time Facilities

GNUstep contains the `NSDate` class and the `NSCalendarDate` classes for representing and handling dates and times. `NSDate` has methods just relating to times and time differences in the abstract, but not calendar dates or time zones. These features are added in the `NSCalendarDate` subclass. The `NSTimeZone` class handles time zone information.

8.5 String Manipulation and Text Processing

Basic string handling in the GNUstep Base library was covered in Chapter 2 [Strings in GNUstep], page 11. Here, we introduce a number of additional string and text processing facilities provided by GNUstep.

8.5.1 NSScanner and Character Sets

The `NSScanner` class can be thought of as providing a combination of the capabilities of the C `sscanf()` function and the Java `StringTokenizer` class. It supports parsing of NSStrings and extraction of numeric values or substrings separated by delimiters.

`NSScanner` works with objects of a class `NSCharacterSet` and its subclasses `NSMutableCharacterSet`, `NSBitmapCharSet`, and `NSMutableBitmapCharSet`, which provide various means of representing sets of unicode characters.

8.5.2 Attributed Strings

Attributed strings are strings that support the association of *attributes* with ranges of characters within the string. Attributes are name-value pairs represented by an `NSDictionary` and may include standard attributes (used by GNUstep GUI classes for font and other characteristics during rendering) as well as programmer-defined application specific attributes. The classes `NSAttributedString` and `NSMutableAttributedString` represent attributed strings. They are not subclasses of `NSString`, though they bundle an instance of one.

8.5.3 Formatters

Formatters are classes providing support for converting complex values into text strings. They also provide some support for user editing of strings to be converted back into object equivalents. All descend from `NSFormatter`, which defines basic methods for obtaining either an attributed string or a regular string for an object value. Specific classes include `NSDateFormatter` for `NSDate` objects, `NSNumberFormatter` for `NSNumber` objects. Instances of these classes can be customized for specific display needs.

8.6 File Handling

A number of convenience facilities are provided for platform-independent access to the file system. The most generally useful is the `NSFileManager` class, which allows you to read and save files, create/list directories, and move or delete files and directories. In addition to simply listing directories, you may obtain an `NSDirectoryEnumerator` instance from it, a subclass of `NSEnumerator` which provides a full listing of all the files beneath a directory and its subdirectories.

If you need to work with path names but don't need the full `NSFileManager` capabilities, `NSString` provides a number of path-related methods, such as `-stringByAppendingPathComponent:` and `-lastPathComponent`. You should use these instead of working directly with path strings to support cross-platform portability.

`NSFileHandle` is a general purpose I/O class which supports reading and writing to both files and network connections, including ordinary and encrypted (SSL) socket connections, and the standard in / standard out streams familiar from Unix systems. You can obtain instances through methods like `+fileHandleForReadingAtPath:(NSString *)path` and `+fileHandleAsServerAtAddress:(NSString *)address service:(NSString *)service protocol:(NSString *)protocol`. The network-related functions of `NSFileHandle` (which are a GNUstep extension not included in Cocoa) will be covered in a later section. Note this class also supports gzip compression for reading and writing.

Finally, GNUstep also provides some miscellaneous filesystem-related utility functions, including `NSTemporaryDirectory()` and `NSHomeDirectoryForUser()`.

8.7 Persistence and Serialization

GNUstep provides robust facilities for persisting objects to disk or sending them over a network connection (to implement Chapter 7 [Distributed Objects], page 63). One class of facilities is referred to as *property list serialization*, and is only usually used for `NSDictionary` and `NSArray` container objects, and `NSNumber`, `NSData`, `NSString`, and `NSDate` member objects. It utilizes primarily text-based formats.

Saving to and loading back from a serialized property list representation will preserve values but not necessarily the classes of the objects. This makes property list representations robust across platforms and library changes, but also makes it unsuitable for certain applications. *Archiving*, the second class of GNUstep persistence facilities, provides for the persistence of a *graph* of arbitrary objects, with references to one another, taking care to only persist each individual object one time no matter how often it is referred to. Object class identities are preserved, so that the behavior of a reloaded object graph is guaranteed to be the same as the saved one. On the other hand, the classes for these objects must be available at load time.

8.7.1 Property List Serialization

Serialized property list representations (sometimes referred to as "*plists*") are typically saved and restored using methods in collection classes. For example the `NSDictionary` class has `-writeToFile:atomically:` to save, `+dictionaryWithContentsOfFile` to restore, and `NSArray` has similar methods. Alternatively, if you wish to save/restore individual `NSData` or other objects, you can use the `NSPropertyListSerialization` class. (There are also

`NSSerializer` and `NSDeserializer` classes, but these are deprecated in Mac OS X and are not really needed in GNUstep either, so should not be used.)

Serialized property lists can actually be written in one of three different formats – plain text, XML, and binary. Interconversion amongst these is possible using the `pldes` and `plser` command-line tools (see the tools reference (`../../Tools/Reference/index.html`)).

8.7.2 Archives

Archiving utilizes a binary format that is cross-platform between GNUstep implementations, though not between GNUstep and Mac OS X Cocoa. Archiving, like serialization in Java, is used both for saving/restoring objects on disk and for interprocess communications with Chapter 7 [Distributed Objects], page 63. For an object to be archivable, it must adopt the `NSCoding` protocol. The coding process itself is managed by instances of the `NSCoder` class and its subclasses:

`NSCoder` Base class, defines most of the interface used by the others.

`NSArchiver, NSUnarchiver`
: Sequential archives that can only be saved and restored en masse.

`NSKeyedArchiver, NSKeyedUnarchiver`
: Random access archives that can be read from and written to on an individual-object basis and provide more robust integrity in the face of class changes.

`NSPortCoder`
: Used for Chapter 7 [Distributed Objects], page 63.

The basic approach to accessing or creating an archive is to use one of the convenience methods in an `NSCoder` subclass:

`+ (BOOL) archiveRootObject: (id)object toFile: (NSString *)file`
: Save object and graph below it to file. 'YES' returned on success. Both `NSArchiver` and `NSKeyedArchiver` support.

`+ (NSData *) archivedDataWithRootObject: (id)object`
: Save object and graph below it to a byte buffer. Both `NSArchiver` and `NSKeyedArchiver` suport.

`+ (id) unarchiveObjectWithFile: (NSString *)file`
: Load object graph from file. Both `NSUnarchiver` and `NSKeyedUnarchiver` support.

`+ (id) unarchiveObjectWithData: (NSData *)data`
: Load object graph from byte buffer. Both `NSUnarchiver` and `NSKeyedUnarchiver` support.

To obtain more specialized behavior, instantiate one of the classes above and customize it (through various method calls) before instigating the primary archiving or unarchiving operation.

From the perspective of the objects being archived, the `NSCoding` protocol declares two methods that must be implemented:

`-(void) encodeWithCoder: (NSCoder *)encoder`
: This message is sent by an `NSCoder` subclass or instance to request the object to archive itself. The implementation should send messages to `encoder` to save

its essential instance variables. If this is impossible (for whatever reason) an exception should be raised.

`-(id) initWithCoder: (NSCoder *)decoder`
: This message is sent by an `NSCoder` subclass or instance to request the object to restore itself from an archive. The implementation should send messages to `decoder` to load its essential instance variables. An exception should be raised if there is a problem restoring state.

Here is an example `NSCoding` implementation:

```
@interface City : PoliticalUnit
{
private
float      latitude;
float      longitude;
CensusData *censusData;
State      *state;
}

  // ...
@end

 ...

@implementation City

- (void) encodeWithCoder: (NSCoder *)coder
{
  [super encodeWithCoder:coder];  // always call super first

  if (![coder allowsKeyedCoding])
    {
      [coder encodeValueOfObjCType: @encode(float) at: &latitude];
      [coder encodeValueOfObjCType: @encode(float) at: &longitude];
      [coder encodeObject: censusData];
      [coder encodeConditionalObject: state];
    }
    else
    {
      [coder encodeFloat: latitude forKey: @"City.latitude"];
      [coder encodeFloat: longitude forKey: @"City.longitude"];
      [coder encodeObject: censusData forKey: @"City.censusData"];
      [coder encodeConditionalObject: state forKey: @"City.state"];
    }
  return;
}

- (id) initWithCoder: (NSCoder *)coder
```

```
{
  self = [super initWithCoder:coder];  // always assign 'self' to super init..

  if (![coder allowsKeyedCoding])
    {
      // Must decode keys in same order as encodeWithCoder:
      [coder decodeValueOfObjCType: @encode(float) at: &latitude];
      [coder decodeValueOfObjCType: @encode(float) at: &longitude];
      censusData = [[coder decodeObject] retain];
      state = [[coder decodeObject] retain];
    }
    else
    {
      // Can decode keys in any order
      censusData = [[coder decodeObjectForKey: @"City.censusData"] retain];
      state = [[coder decodeObjectForKey: @"City.state"] retain];
      latitude = [coder decodeFloatForKey: @"City.latitude"];
      longitude = [coder decodeFloatForKey: @"City.longitude"];
    }
  return self;
}

  // ...

@end
```

The primary wrinkle to notice here is the check to `[coder allowsKeyedCoding]`. The object encodes and decodes its instance variables using keys if this returns 'YES'. Keys must be unique within a single inheritance hierarchy – that is, a class may not use keys the same as its superclass or any of its ancestors or sibling classes.

Keyed archiving provides robustness against class changes and is therefore to be preferred in most cases. For example, if instance variables are added at some point to the `City` class above, this will not prevent earlier versions of the class from restoring data from a later one (they will just ignore the new values), nor does it prevent a later version from initializing from an earlier archive (it will not find values for the added instance variables, but can set these to defaults).

Finally, notice the use of `encodeConditionalObject` above for `state`, in contrast to `encodeObject` for census data. The reason the two different methods are used is that the `City` object *owns* its census data, which is an integral part of its structure, whereas the `state` is an auxiliary reference neither owned nor retained by `City`. It should be possible to store the cities without storing the states. Thus, the `encodeConditionalObject` method is called, which only stores the `State` if it is already being stored unconditionally elsewhere during the current encoding operation.

Note that within a given archive, an object will be written only once. Subsequent requests to write the same object are detected and a reference is written rather than the full object.

8.8 Utility

The GNUstep Base library provides a number of utility classes that don't fall under any other function category.

The `NSUserDefaults` class provides access to a number of system- and user-dependent settings that should affect tool and application behavior. You obtain an instance through sending `[NSUserDefaults standardUserDefaults]`. The instance provides access to settings indexed by string keys. The standard keys used are documented here (`../../../User/Gui/DefaultsSummary.html`). Users can adjust settings for particular keys using the `defaults (../../Tools/Reference/defaults.html)` command.

The `NSProcessInfo` class provides access to certain information about the system environment such as the operating system and host name. It also provides support for process logging (see Chapter 6 [Logging], page 55). You obtain an instance through sending `[NSProcessInfo processInfo]`.

The `NSUndoManager` class provides a general mechanism for supporting *undo* of user operations in applications. Essentially, it allows you to store sequences of messages and receivers that need to be invoked to undo or redo an action. The various methods in this class provide for grouping of sets of actions, execution of undo or redo actions, and tuning behavior parameters such as the size of the undo stack. Each application entity with its own editing history (e.g., a document) should have its own undo manager instance. Obtain an instance through a simple `[[NSUndoManager alloc] init]` message.

The `NSProtocolChecker` and `NSProxy` classes provide message filtering and forwarding capabilities. If you wish to ensure at runtime that a given object will only be sent messages in a certain protocol, you create an `NSProtocolChecker` instance with the protocol and the object as arguments:

```
id versatileObject = [[ClassWithManyMethods alloc] init];
id narrowObject = [NSProtocolChecker protocolCheckerWithTarget: versatileObject
                                                      protocol: @protocol(SomeSpecificProtocol)];
return narrowObject;
```

This is often used in conjunction with distributed objects to expose only a subset of an objects methods to remote processes. The `NSProxy` class is another class used in conjunction with distributed objects. It implements no methods other than basic `NSObject` protocol methods. To use it, create a subclass overriding `-(void) forwardInvocation:` and `-(NSMethodSignature) methodForSelector:`. By appropriate implementations here, you can make an `NSProxy` subclass instance act like an instance of any class of your choosing.

The `NSBundle` class provides support for run-time dynamic loading of libraries and application resources, usually termed "Bundles". A bundle consists of a top-level directory containing subdirectories that may include images, text files, and executable binaries or shared libraries. The ".app" directory holding a NeXTstep/OpenStep/GNUstep/Cocoa application is actually a bundle, as are the "Frameworks" containing shared libraries together with other resources. Bundles and frameworks are covered in Appendix B [Bundles and Frameworks], page 103.

8.9 Notifications

GNUstep provides a framework for sending messages between objects within a process called *notifications*. Objects register with an `NSNotificationCenter` to be informed whenever

other objects post NSNotifications to it matching certain criteria. The notification center processes notifications synchronously – that is, control is only returned to the notification poster once every recipient of the notification has received it and processed it. Asynchronous processing is possible using an NSNotificationQueue. This returns immediately when a notification is added to it, and it will periodically post the oldest notification on its list to the notification center. In a multithreaded process, notifications are always sent on the thread that they are posted from.

An NSNotification consists of a string name, an object, and optionally a dictionary which may contain arbitrary associations. Objects register with the notification center to receive notifications matching either a particular name, a particular object, or both. When an object registers, it specifies a message selector on itself taking an NSNotification as its sole argument. A message will be sent using this selector whenever the notification center receives a matching notification in a post.

Obtain a notification center instance using NSNotificationCenter +defaultCenter. An NSDistributedNotificationCenter may be used for interprocess communication on the same machine. Interprocess notification will be slower than within-process notification, and makes use of the gdnc (../../Tools/Reference/gdnc.html) command-line tool.

Notifications are similar in some ways to *events* in other frameworks, although they are not used for user interface component connections as in Java (message forwarding and the target-action paradigm are used instead). In addition, the GNUstep GUI (AppKit) library defines an NSEvent type for representing mouse and keyboard actions.

8.10 Networking and RPC

GNUstep provides some general network-related functionality, as well as classes supporting Chapter 7 [Distributed Objects], page 63, and related forms of inter-process communication.

8.10.1 Basic Networking

GNUstep provides the following classes handling basic network communications:

NSHost
: Holds and manages information on host names and IP addresses. Use the +currentHost, +hostWithName:, or +hostWithAddress: class methods to obtain an instance.

NSFileHandle
: On Unix, network connections are treated analogously to files. This abstraction has proven very useful, so GNUstep supports it in the NSFileHandle class, even on non-Unix platforms. You may use the class methods +fileHandleAsServerAtAddress:(NSString *)address service:(NSString *)service protocol:(NSString *)protocol and corresponding client methods to access one side of a connection to a port on a networked machine. (To use pipes, see the next section.)

NSURL
: Provides methods for working with URLs and the data accessible through them. Once an NSURL is constructed, data may be loaded asynchronously through NSURL -loadResourceDataNotifyingClient:usingCache: or synchronously through NSURL -resourceDataUsingCache:. It can also be obtained through NSString +stringWithContentsOfURL: or NSData +dataWithContentsOfURL:.

`NSURLHandle`
: This class provides additional control over the URL contents loading process. Obtain an instance through `NSURL -URLHandleUsingCache:`.

8.10.2 Remote Process Communications

GNUstep provides a number of classes supporting Chapter 7 [Distributed Objects], page 63, and related forms of inter-process communication. In most cases, you only need to know about the `NSConnection` class, but if you require additional control over distributed objects, or if you wish to use alternative forms of communications such as simple messaging, you may use the classes listed here.

`NSConnection`
: This is the primary class used for registering and acquiring references to distributed objects.

`NSDistantObject`
: When a client acquires a remote object reference through `NSConnection +rootProxyForConnectionWithRegisteredName:`, the returned object is an instance of this class, which is a subclass of `NSProxy`. Since usually you will just cast your reference to this to a particular protocol, you do not need to refer to the `NSDistantObject` class directly.

`NSPort, NSPortMessage`
: Behind the scenes in distributed objects, `NSPort` objects handle both network communications and serialization/deserialization for sending messages to remote objects and receiving the results. The actual data sent over the network is encapsulated by `NSPortMessage` objects, which consist of two ports (sender and receiver) and a body consisting of one or more `NSData` or `NSPort` objects. (Data in the `NSData` must be in network byte order.)

`NSSocketPort, NSMessagePort`
: If you want to send custom messages between processes yourself, you can use these classes. `NSSocketPort` can communicate to processes on the same or remote machines. `NSMessagePort` is optimized for local communications only.

`NSPortNameServer, NSSocketPortNameServer, NSMessagePortNameServer`
: The `NSPortNameServer` class and subclasses are used behind the scenes by the distributed objects system to register and look up remotely-accessible objects.

8.11 Threads and Run Control

A GNUstep program may initiate independent processing in two ways – it can start up a separate process, referred to as a *task*, much like a *fork* in Unix, or it may spawn multiple *threads* within a single process. Threads share data, tasks do not. Before discussing tasks and threads, we first describe the *run loop* in GNUstep programs.

8.11.1 Run Loops and Timers

`NSRunLoop` instances handle various utility tasks that must be performed repetitively in an application, such as processing input events, listening for distributed objects communications, firing `NSTimer`s, and sending notifications and other messages asynchronously. In general,

there is one run loop per thread in an application, which may always be obtained through the `+currentRunLoop` method, however unless you are using the AppKit and the [NSApplication] class, the run loop will not be started unless you explicitly send it a `-run` message.

At any given point, a run loop operates in a single *mode*, usually `NSDefaultRunLoopMode`. Other modes are used for special purposes and you usually need not worry about them.

An `NSTimer` provides a way to send a message at some time in the future, possibly repeating every time a fixed interval has passed. To use a timer, you can either create one that will automatically be added to the run loop in the current thread (using the `-addTimer:forMode:` method), or you can create it without adding it then add it to a run loop of your choosing later.

8.11.2 Tasks and Pipes

You can run another program as a subprocess using an `NSTask` instance, and communicate with it using `NSPipe` instances. The following code illustrates.

```
NSTask *task = [[NSTask alloc] init];
NSPipe *pipe = [NSPipe pipe];
NSFileHandle *readHandle = [pipe fileHandleForReading];
NSData *inData = nil;

[task setStandardOutput: pipe];
[task setLaunchPath: [NSHomeDirectory()
                          stringByAppendingPathComponent:@"bin/someBinary"]];
[task launch];

while ((inData = [readHandle availableData]) && [inData length])
  {
    [self processData:inData];
  }
[task release];
```

Here, we just assume the task has exited when it has finished sending output. If this might not be the case, you can register an observer for Chapter 8 [Notifications], page 83, named `NSTaskDidTerminateNotification`.

8.11.3 Threads and Locks

Threads provide a way for applications to execute multiple tasks in parallel. Unlike separate processes, all threads of a program share the same memory space, and therefore may access the same objects and variables.

GNUstep supports multithreaded applications in a convenient manner through the `NSThread` and `NSLock` classes and subclasses. `NSThread +detachNewThreadSelector:toTarget:withObject:` allows you to initiate a new thread and cause a message to be sent to an object on that thread. The thread can either run in a "one-shot" manner or it can sit in loop mode (starting up its own instance of the `NSRunLoop` class) and communicate with other threads using part of the Chapter 7 [Distributed Objects], page 63, framework. Each thread has a dictionary (accessed through `-threadDictionary` that allows for storage of thread-local variables.

Because threads share data, there is the danger that examinations of and modifications to data performed concurrently by more than one thread will occur in the wrong order and produce unexpected results. (Operations with immutable objects do not present this problem unless they are actually deallocated.) GNUstep provides the *NSLocking* protocol and the NSLock class and subclasses to deal with this. *NSLocking* provides two methods: -lock and -unlock. When an operation needs to be performed without interference, enclose it inside of lock-unlock:

```
NSArray *myArray;
NSLock *myLock = [[NSLock alloc] init];
  // ...
[myLock lock];
if (myArray == nil)
  {
    myAray = [[NSMutableArray alloc] init];
    [myArray addObject: someObject];
  }
[myLock unlock];
```

This code protects 'myArray' from accidentally being initialized twice if two separate threads happen to detect it is nil around the same time. When the lock method is called, the thread doing so is said to *acquire* the lock. No other thread may subsequently acquire the lock until this one has subsequently *relinquished* the lock, by calling unlock.

Note that the lock object should be initialized *before* any thread might possibly need it. Thus, you should either do it before any additional threads are created in the application, or you should enclose the lock creation inside of another, existing, lock.

The -lock method in the *NSLocking* protocol blocks indefinitely until the lock is acquired. If you would prefer to just check whether the lock can be acquired without committing to this, you can use NSLock -tryLock or NSLock -lockBeforeDate:, which return YES if they succeed in acquiring the lock.

NSRecursiveLock is an NSLock subclass that may be locked multiple times by the same thread. (The NSLock implementation will not allow this, causing the thread to deadlock (freeze) when it attempts to acquire the lock a second time.) Each lock message must be balanced by a corresponding unlock before the lock is relinquished.

NSConditionLock stores an int together with its lock status. The -lockWhenCondition:(int)value and related methods request the lock only if the condition is equal to that passed in. The condition may be changed using the unlockWithCondition:(int)value method. This mechanism is useful for, e.g., a producer-consumer situation, where the producer can "tell" the consumer that data is available by setting the condition appropriately when it releases the lock it acquired for adding data.

Finally, the NSDistributedLock class does not adopt the *NSLocking* protocol but supports locking across processes, including processes on different machines, as long as they can access a common filesystem.

If you are writing a class library and do not know whether it will be used in a multithreaded environment or not, and would like to avoid locking overhead if not, use the NSThread +isMultiThreaded method. You can also register to receive

`NSWillBecomeMultiThreadedNotification`s. You can also use the `GSLazyLock` and `GSLazyRecursiveLock` classes (see next section) which handle this automatically.

8.11.4 Using `NSConnection` to Communicate Between Threads

You can use the distributed objects framework to communicate between threads. This can help avoid creating threads repeatedly for the same operation, for example. While you can go through the full process of registering a server object by name as described in Chapter 7 [Distributed Objects], page 63, a lighter weight approach is to create `NSConnection`s manually:

```
  // code in Master...
- startSlave
{
  NSPort *master;
  NSPort *slave;
  NSArray *ports;
  NSConnection *comms;

  master = [NSPort port];
  slave  = [NSPort port];

  comms = [[NSConnection alloc] initWithReceivePort: master sendPort: slave];
  [comms setRootObject: self];

  portArray = [NSArray arrayWithObjects: slave, master, nil];

  [NSThread detachNewThreadSelector: @selector(newWithCommPorts:)
                           toTarget: [Slave class]
                         withObject: ports];
}

  // code in Slave...
+ newWithCommPorts: (NSArray *)ports
{
  NSConnection *comms;

  NSPort *slave  = [ports objectAtIndex: 0];
  NSPort *master = [ports objectAtIndex: 1];

  comms = [NSConnection connectionWithReceivePort: slave sendPort: master];

  // create instance and assign to 'self'
  self = [[self alloc] init];

  [(id)[comms rootProxy] setServer: self];
  [self release];
```

```
    [[NSRunLoop currentRunLoop] run];
  }
```

8.12 GNUstep Additions

The Base Additions (`../../BaseAdditions/Reference/index.html`) library consists of a number of classes, all beginning with 'GC' or 'GS', that are not specified in OpenStep or Cocoa but have been deemed to be of general utility by the GNUstep developers. The library is designed so that it can be built and installed on a system, such as OS X, where GNUstep is not available but an alternate Foundation implementation is.

It contains the following categories of classes:

`GSXMLxxx` Classes for parsing XML using DOM- or SAX-like APIs, for processing XPath expressions, and for performing XSLT transforms. These are implemented over the libxml2 (`http://xmlsoft.org`) C library, and using them should for the most part protect you from the frequent API changes in that library.

`GSHtmlXxx`
: Classes for parsing HTML documents (not necessarily XHTML).

`GSMimeXxx`
: Classes for handling MIME messages or HTML POST documents.

`GSLazyXxx`
: Classes implementing "lazy" locking that do not actually attempt any locking unless running in a multithreaded application. See Chapter 8 [Threads and Run Control], page 83.

All of these classes have excellent API reference documentation and you should look there (`../../BaseAdditions/Reference/index.html`) for further information.

Appendix A The GNUstep Documentation System

GNUstep includes its own documentation system for producing HTML, PDF, and other readable documents for developers and users. (It also includes facilities for "Help" accessed within applications, but these are not covered here.) It is based on *GSdoc*, an XML language designed specifically for writing documentation for the GNUstep project (`http://www.gnustep.org`). In practice, that means that it is designed for writing about software, and in particular, for writing about Objective-C classes.

It may be used to write narrative documentation by hand, and it can also be autogenerated by the *autogsdoc* tool, which parses Objective-C source files and documents classes, methods, functions, macros, and variables found therein, picking up on special comments when provided to enhance the documentation.

You can read more about GSdoc itself in this document (`../../Tools/Reference/gsdoc.html`).

The *autogsdoc* tool is described here (`../../Tools/Reference/autogsdoc.html`).

(Both of these documents are part of the Base Tools (`../../Tools/Reference/index.html`) documentation.)

A.1 Quick Start

The basic approach to using GSdoc is this: when writing source code, put comments that begin with "`/**`" instead of the usual C "`/*`" in your `@interface` or `@implementation` file above class, variable, and method declarations. If you have any functions or macros you are making available put such comments in front of them too. The comments still end with the regular "`*/`", no "`**/`" is necessary.

```
/**
 *  The point class represents 2-d locations independently of any graphical
 *  representation.
 */
@interface Point : NSObject
{
    // instance variables ...
}

/**
 *  New point at 0,0.
 */
+ new;

 // ...

/**
 *  Return point's current X position.
 */
- (float) x;
 // ...
```

```
@end
```

When you are finished, invoke *autogsdoc* giving it the names of all your header files. (It will find the implementation files automatically, as long as they have the same names; alternatively, give it the names of the implementation files as well.) This will produce a set of HTML files describing your classes. If you include the '`-MakeFrames YES`' argument, the HTML will be structured into frames for easy navigation.

(Autogsdoc, like all GNUstep command line tools, is found in the ${GNUSTEP_SYSTEM_ROOT}/Tools directory.)

You can also generate documentation automatically using the GNUstep make utility. Consult its primary documentation (`../../Make/Manual/make_toc.html`) for details. The short story is:

```
include $(GNUSTEP_MAKEFILES)/common.make

DOCUMENT_NAME = MyProject

MyProject_AGSDOC_FILES = <space-separated list of header files>
MyProject_AGSDOC_FLAGS = <flags, like MakeFrames YES>

include $(GNUSTEP_MAKEFILES)/documentation.make
```

Usually this is put into a separate makefile called "`DocMakeFile`" in the source directory.

A.2 Cross-Referencing

GSdoc provides the ability to reference entities both within the project and in external projects. When writing GSdoc comments in source code, references are particularly easy to create. To refer to an argument of the method or function you are documenting, just type it normally; it will be presented in a special type face in the final documentation to show it is an argument. To refer to another method within the same class you are documenting, just type its selector with the + or - sign in front. This will be converted into a hyperlink in output forms that support that. To refer to another class, you just type the class's name in [Brackets]. To refer to a method in another class, put the method selector after the name, as in [Class-methodWithArg1:andArg2:] (do not include a space). To refer to a protocol, use [(BracketsAndParentheses)] instead of just brackets. To refer to a category, use [Class(Category)]. For methods in these two cases, put the method name outside the parentheses. To refer to a function, simply type its name suffixed by parentheses().

A.3 Comment the Interface or the Implementation?

Since `autogsdoc` picks up comments both from interface/header files and implementation/source files, you might be wondering where it is best to put them. There is no consensus on this issue. If you put them in the interface, then anyone you distribute your library to (with the headers but not the source) will be able to generate the documentation. The header file carries all of the specification for the class's behavior. On the other hand, if you put the comments in the implementation, then people editing the source code will have the method descriptions handy when they need them. If *autogsdoc* finds comments for the same entity in both interface and implementation, they are concatenated in the result.

Nonetheless, the recommendation of this author is that you put the comments in the header, since this is more within the spirit of Objective-C, where the interface file declares the behavior of a class.

A.4 Comparison with OS X Header Doc and Java JavaDoc

The HTML output from all of these systems is roughly comparable. In terms of and comments needed in the source code to produce good class documentation, the GSdoc / autogsdoc system aims for maximal simplicity. In practice, requiring lots of special formatting makes developers less likely to document things, therefore, as described above, GSdoc does not require it, letting the parser do the work instead of the person.

In terms of non-HTML output formats and control over the HTML format, these are not provided with GSdoc, yet, but there are plans to provide them through the use of XSLT as a presentation layer.

Appendix B Application Resources: Bundles and Frameworks

TBD

Appendix C Differences and Similarities Between Objective-C, Java, and C++

This appendix explains the differences/similarities between Objective-C and Java. It does not cover the Java Interface to GNUstep (JIGS; see Appendix D [Java and Guile], page 109), but is included to help people who want to learn Objective-C and know Java already.

C.1 General

- C programmers may learn Objective-C in hours (though real expertise obviously takes much longer).
- Java has global market acceptance.
- Objective-C is a compiled OO programming language.
- Java is both compiled and interpreted and therefore does not offer the same run-time performance as Objective-C.
- Objective-C features efficient, transparent Distributed Objects.
- Java features a less efficient and less transparent Remote Machine Interface.
- Objective-C has basic CORBA compatibility through official C bindings, and full compatibility through unofficial Objective-C bindings.
- Java has CORBA compatibility through official Java bindings.
- Objective-C is portable across heterogeneous networks by virtue of a near omnipresent compiler (gcc).
- Java is portable across heterogeneous networks by using client-side JVMs that are software processors or runtime environments.

C.2 Language

- Objective-C is a superset of the C programming language, and may be used to develop non-OO and OO programs. Objective-C provides access to scalar types, structures and to unions, whereas Java only addresses a small number of scalar types and everything else is an object. Objective-C provides zero-cost access to existing software libraries written in C, Java requires interfaces to be written and incurs runtime overheads.
- Objective-C is dynamically typed but also provides static typing. Java is statically typed, but provides type-casting mechanisms to work around some of the limitations of static typing.
- Java tools support a convention of a universal and distributed name-space for classes, where classes may be downloaded from remote systems to clients. Objective-C has no such conventions or tool support in place.
- Using Java, class definitions may not be extended or divided through the addition of logical groupings. Objective-C provides categories as a solution to this problem.
- Objective-C provides delegation (the benefits of multiple inheritance without the drawbacks) at minimal programming cost. Java requires purpose written methods for any delegation implemented.
- Java provides garbage collection for memory management. Objective-C provides manual memory management, reference counting, and garbage collection as options.
- Java provides interfaces, Objective-C provides protocols.

C.3 Source Differences

- Objective-C is based on C, and the OO extensions are comparable with those of Smalltalk. The Java syntax is based on the C++ programming language.
- The object (and runtime) models are comparable, with Java's implementation having a subset of the functionality of that of Objective-C.

C.4 Compiler Differences

- Objective-C compilation is specific to the target system/environment, and because it is an authentic compiled language it runs at higher speeds than Java.
- Java is compiled into a byte stream or Java tokens that are interpreted by the target system, though fully compiled Java is possible.

C.5 Developer's Workbench

- Objective-C is supported by tools such as GNUstep that provides GUI development, compilation, testing features, debugging capabilities, project management and database access. It also has numerous tools for developing projects of different types including documentation.
- Java is supported by numerous integrated development environments (IDEs) that often have their origins in C++ tools. Java has a documentation tool that parses source code and creates documentation based on program comments. There are similar features for Objective-C.
- Java is more widely used.
- Objective-C may leverage investment already made in C based tools.

C.6 Longevity

- Objective-C has been used for over ten years, and is considered to be in a stable and proven state, with minor enhancements from time to time.
- Java is evolving constantly.

C.7 Databases

- Apple's EOF tools enable Objective-C developers to build object models from existing relational database tables. Changes in the database are automatically recognised, and there is no requirement for SQL development.
- Java uses JDBC that requires SQL development; database changes affect the Java code. This is considered inferior to EOF. Enterprise JavaBeans with container managed persistence provides a limited database capability, however this comes with much additional baggage. Other object-relational tools and APIs are being developed for Java (ca. 2004), but it is unclear which of these, if any, will become a standard.

C.8 Memory

- For object allocation Java has a fixed heap whose maximum size is set when the JVM starts and cannot be resized unless the JVM is restarted. This is considered to be a

disadvantage in certain scenarios: for example, data read from databases may cause the JVM to run out of memory and to crash.

- Objective-C's heap is managed by the OS and the runtime system. This can typically grow to consume all system memory (unless per-process limits have been registered with the OS).

C.9 Class Libraries

- Objective-C: Consistent APIs are defined by the OpenStep specification. This is implemented by GNUstep and Mac OS X Cocoa. Third-party APIs are available (called Frameworks).
- Java: APIs are defined and implemented by the Sun Java Development Kit distributions. Other providers of Java implementations (IBM, BEA, etc.) implement these as well.
- The Java APIs are complex owing to the presence of multiple layers of evolution while maintaining backwards compatibility. Collections, IO, and Windowing are all examples of replicated functionality, in which the copies are incompletely separated, requiring knowledge of both to use.
- The OpenStep API is the result of continuing evolution but backward compatibility was maintained by the presence of separate library versions. Therefore the API is clean and nonredundant. Style is consistent.
- The OpenStep non-graphical API consists of about 70 classes and about 150 functions.
- The equivalent part of the Java non-graphical API consists of about 230 classes.
- The OpenStep graphical API consists of about 120 classes and 30 functions.
- The equivalent part of the Java graphical API consists of about 450 classes.

Appendix D Programming GNUstep in Java and Guile

TBD

Appendix E GNUstep Compliance to Standards

GNUstep is generally compatible with the OpenStep specification and with recent developments of the MacOS-X (Cocoa) API. Where MacOS deviates from the OpenStep API, GNUstep generally attempts to support both versions. In some cases the newer MacOS APIs are incompatible with OpenStep, and GNUstep usually supports the richer version.

In order to deal with compatiblity issues, GNUstep uses two mechanisms - it provides conditionally compiled sections of the library header files, so that software can be built that will conform strictly to a particular API, and it provides user default settings to control the behavior of the library at runtime.

E.1 Conditional Compilation

Adding an option to a makefile to define one of the following preprocessor constants will modify the API visible to software being compiled -

`NO_GNUSTEP` [preprocessor]
: GNUstep specific extensions to the OpenStep and MacOS cocoa APIs are excluded from the headers if this is defined to a non-zero value.

`STRICT_MACOS_X` [preprocessor]
: Only methods and classes that are part of the MacOS cocoa API are made available in the headers if this is defined.

`STRICT_OPENSTEP` [preprocessor]
: Only methods and classes that are part of the OpenStep specification are made available in the headers if this is defined.

Note, these preprocessor constants are used in developer code (ie the code that users of GNUstep write) rather than by the GNUstep software itself. They permit a developer to ensure that he/she does not write code which depends upon API not present on other implementations (in practice, MacOS-X or some old OPENSTEP systems). The actual GNUstep libraries are always built with the full GNUstep API in place, so that the feature set is as consistent as possible.

E.2 User Defaults

User defaults may be specified ...

`GNU-Debug` [defaults]
: An array of strings that lists debug levels to be used within the program. These debug levels are merged with any which were set on the command line or added programmatically to the set given by the [NSProcessInfo-debugSet] method.

`GSLogSyslog` [defaults]
: Setting the user default GSLogSyslog to YES will cause log/debug output to be sent to the syslog facility (on systems which support it), rather than to the standard error stream. This is useful in environments where stderr has been re-used strangely for some reason.

`GSMacOSXCompatible` [defaults]

Setting the user default GSMacOSXCompatible to YES will cause MacOS compatible behavior to be the default at runtime. This default may however be overridden to provide more fine grained control of system behavior.

`GSOldStyleGeometry` [defaults]

Specifies whether the functions for producing strings describing geometric structures (NSStringFromPoint(), NSStringFromSize(), and NSStringFromRect()) should produce strings conforming to the OpenStep specification or to MacOS-X behavior. The functions for parsing those strings should cope with both cases anyway.

`GSSOCKS` [defaults]

May be used to specify a default SOCKS5 server (and optionally a port separated from the server by a colon) to which tcp/ip connections made using the NSFileHandle extension methods should be directed.

This default overrides the SOCKS5_SERVER and SOCKS_SERVER environment variables.

`Local Time Zone` [defaults]

Used to specify the name of the timezone to be used by the NSTimeZone class.

`NSWriteOldStylePropertyLists` [defaults]

Specifies whether text property-list output should be in the default MacOS-X format (XML), or in the more human readable (but less powerful) original OpenStep format.

Reading of property lists is supported in either format, but only if GNUstep is built with the libxml library (which is needed to handle XML parsing).

NB. MacOS-X generates illegal XML for some strings - those which contain characters not legal in XML. GNUstep always generates legal XML, at the cost of a certain degree of compatibility. GNUstep XML property lists use a backslash to escape illegal chatracters, and consequently any string containing either a backslash or an illegal character will be written differently to the same string on MacOS-X.

`NSLanguages` [defaults]

An array of strings that lists the users prefered languages, in order or preference. If not found the default is just English.

Appendix F Using the GNUstep Make Package

Reference/doc on the GNUstep make package; mainly examples of various types of projects.

F.1 Makefile Contents

Note. Type `man make` for assistance.

Make files comprise four key content types:

- **Comments**

  `#` prefixes comments.

- **Macros**

  `SRC=main.m` assigns a macro that is implemented as:`gcc $(SRC)` and is interpreted as: `gcc main.m`

- **Explicit rules** Explicit rules are used to indicate which files depend upon supporting files and the commands required for their compilation:

  ```
  targetfile : sourcefiles
  commands
  ```

  A Tab precedes each command that must be implemented on the source files in order to create the target file:

  ```
  main: main.m List.h
  gcc -o main main.m List.h
  ```

- **Implicit rules** Implicit rules broadly echo explicit variants but do not specify commands, rather the extensions indicate which commands are performed:

  `servertest.o: servertest.c io.h`
  is interpreted as:
  `$(CC) $(CFLAGS) -c servertest.c io.h`

F.1.1 Makefile Example

```
# The following two lines force the standard make to recognize the
# Objective-C .m suffix.

.SUFFIXES: .o .m
.m.o:
$(CC) -c $(CFLAGS) $<


# Macro declarations

CC = gcc
CFLAGS = -g
LIBS = -lobjc
SRC=main.m Truck.m Station.m Vehicle.m
OBJ=main.o Truck.o Station.o Vehicle.o
```

```
# Explicit rules

hist: $(OBJ)
$(CC) $(CFLAGS) -o main $(OBJ) $(LIBS)


# Implicit rules

Truck.o: Truck.h Truck.m
Station.o: Truck.h Station.h Station.m
Vehicle.o: Truck.h Vehicle.h Vehicle.m
main.o: Station.h Vehicle.h
```

F.1.2 Makefile Structure

The following Makefile defines a project:

```
#
# A GNUmakefile
#

# Check that the GNUSTEP_MAKEFILES environment variable is set
ifeq ($(GNUSTEP_MAKEFILES),)
GNUSTEP_MAKEFILES := $(shell gnustep-config --variable=GNUSTEP_MAKEFILES 2>/dev/null)
 ifeq ($(GNUSTEP_MAKEFILES),)
   $(error You need to set GNUSTEP_MAKEFILES before compiling!)
 endif
endif

# Include the common variables
include $(GNUSTEP_MAKEFILES)/common.make

# Build an Objective-C program
OBJC_PROGRAM_NAME = simple

# Objective-C files requiring compilation
simple_OBJC_FILES = source.m

-include GNUmakefile.preamble

# Include in the rules for making Objective-C programs
include $(GNUSTEP_MAKEFILES)/objc.make

-include GNUmakefile.postamble
```

To compile a package that uses the Makefile Package, type `make` in the top-level directory of the package. A non-GNUstep Objective-C file may be compiled by adding `-lobjc on` at the command line.

F.1.3 Debug and Profile Information

By default the Makefile Package does not flag the compiler to generate debugging information that is generated by typing:

```
make debug=yes
```

This command also causes the Makefile Package to turn off optimization. It is therefore necessary to override the optimization flag when running Make if both debugging information and optimization is required. Use the variable OPTFLAG to override the optimization flag.

By default the Makefile Package does not instruct the compiler to create profiling information that is generated by typing:

```
make profile=yes
```

F.1.4 Static, Shared and DLLs

By default the Makefile Package generates a shared library if it is building a library project type, and it will link with shared libraries if it is building an application or command-line tool project type. To tell the Makefile Package not to build using shared libraries but using static libraries instead, type:

```
make shared=no
```

This default is only applicable on systems that support shared libraries; systems that do not support shared libraries will always build using static libraries. Some systems support DLLs that are a form of shared libraries; on these systems DLLs are built by default unless the Makefile Package is told to build using static libraries instead.

F.2 Project Types

Projects are divided into different types. To create a project of a specific type, a make file is specified:

```
include $(GNUSTEP_MAKEFILES)/application.make
```

Each project type is independent, and if you want to create two project types in the same directory (e.g. a tool and a Java program), include both the desired make files in your main Make file.

- Aggregate - aggregate.make

  An Aggregate project holds several sub-projects that are of any valid project type (including the Aggregate type). The only project variable is the SUBPROJECTS variable:

  ```
  Aggregate project: SUBPROJECTS
  ```

  SUBPROJECTS defines the directory names that hold the sub-projects that the Aggregate project should build.

- Graphical Applications - application.make

  An application is an Objective-C program that includes a GUI component, and by default links in all the GNUstep libraries required for GUI development, such as the Base and GUI libraries.

- Bundles - bundle.make

  A bundle is a collection of resources and code that can be used to enhance an existing application or tool dynamically using the NSBundle class from the GNUstep base library.

- Command Line C Tools - ctool.make

  A ctool is a project that uses only C language files. Otherwise it is similar to the ObjC project type.

- Documentation - documentation.make

  The Documentation project provides rules to use various types of documentation such as texi and LaTeX documentation, and convert them into finished documentation like info, PostScript, HTML, etc.

- Frameworks - framework.make

  A Framework is a collection of resources and a library that provides common code that can be linked into a Tool or Application. In many respects it is similar to a Bundle.

- Java - java.make

  This project provides rules for building java programs. It also makes it easy to make java projects that interact with the GNUstep libraries.

- Libraries - library.make

  The Makefile Package provides a project type for building libraries that may be static, shared, or dynamic link libraries (DLLs). The latter two variants are supported only on some platforms.

Concept Index

A

abstract class 14
advanced messaging 49
allocating objects 23
AppKit 7
ARC 29
assertion facilities 55
assertion handling, compared with Java 62
assertions 61

B

base library 83
basic OO terminology 3
bundles 103
bycopy and byref type qualifiers 79

C

categories 43
class cluster 15
class, abstract 14
class, root 15
classes 13
client/server processes 63
cluster, classes 15
compilation, conditional 111

D

differences and similarities, Objective-C and C++ 105
differences and similarities, Objective-C and Java 105
directory layout 7
distributed objects 63
Distributed Objects Name Server, GNUstep 69
distributed objects, client code 64, 65
distributed objects, error checking 82
distributed objects, example (no error checking) 70
distributed objects, using a protocol 67

E

error checking, distributed objects 82
exception facilities 55
exception handling, compared with Java 62
exceptions 55

F

filesystem layout 7
forward invocation, distributed objects 81
forwarding 52
frameworks 103

G

game server example 70
gdomap 69
GNUstep base library 6
GNUstep Make package 113
GNUstep make utility 7
GNUstep, what is? 6
graphical programming 7
gsdoc 99
GUI 7

H

history of NeXTstep 5
history of Objective-C 5
history of OpenStep 5

I

in, out, and inout type qualifiers 79
inheritance 13
inheriting methods 14
instance variables, referring to 17
interface 31

J

Java and Guile, programming GNUstep 109

L

layout, filesystem 7
logging 59
logging facilities 55
logging, compared with Java 62

M

Make package, GNUstep 113
make utility, GNUstep 7
memory deallocation 25
memory management 25
memory management, explicit 25
memory management, OpenStep-style 25
memory management, retain count 25
message forwarding, distributed objects 81
messages 12
messaging, advanced techniques 49

N

naming constraints 18
naming conventions 18
NeXTstep, history 5
NS_DURING macro 55
NS_ENDHANDLER macro 55
NS_HANDLER macro 55
NSAssert macro 61
NSAssertionHandler class 61
NSConnection class 63
NSDebugLog function 59
NSException class 55
NSLog function 59
NSObject 15
NSProxy class 63
NSRunLoop class 63
NSUncaughtExceptionHandler 55
NSWarnLog function 59

O

ObjC-2 , automated reference counting 29
object interaction, remote objects 63
object-oriented programming 3
Objective-C and C++, differences
and similarities 105
Objective-C and Java, differences
and similarities 105
Objective-C, history 5
Objective-C, what is? 4
objects 11
objects, initalizing and allocating 23
objects, working with 23
oneway, type qualifier 79
OpenStep compliance 111
OpenStep, history 5
OS X compatibility 111
out, type qualifier 79
overriding methods 14

P

polymorphism 13
profiling facilities 59
protocol for distributed objects 67
protocol type qualifiers 79
protocols 41
protocols, formal 41

R

remote objects 63
resources, application 103
root class 15

S

standards compliance 111
standards, GNUstep compliance to 111
static typing 16

U

user defaults, API compliance 111

W

what is GNUstep? 6
what is Objective-C? 4
working with objects 23
writing new classes 31

Y

your first Objective-C program 8

Z

Zones 24