
Error Handling Programming Guide For Cocoa

Data Management: Event Handling



2009-10-16



Apple Inc.
© 2005, 2009 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, Carbon, Cocoa, eMac, Mac, Mac OS, Macintosh, and Objective-C are trademarks of Apple Inc., registered in the United States and other countries.

iPhone is a trademark of Apple Inc.

UNIX is a registered trademark of The Open Group

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE

ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction Introduction to Error Handling Programming Guide For Cocoa 7

Organization of This Document 7

See Also 8

Chapter 1 Error Objects, Domains, and Codes 9

Why Have Error Objects? 9

Error Domains 9

Error Codes 10

The User Info Dictionary 12

 Localized Error Information 12

 The Recovery Attempter 14

 Underlying Error 14

 Domain-Specific Keys 14

Chapter 2 Using and Creating Error Objects 15

Handling Error Objects Returned From Methods 15

 Error-Handling Alternatives in Mac OS X 16

Displaying Information From Error Objects 17

 Propagating Errors for Display by the Application Object (Mac OS X) 18

Creating and Returning NSError Objects 19

 A Note on Errors and Exceptions 20

Chapter 3 Error Responders and Error Recovery 23

The Error-Responder Chain 23

Error Customization 25

Error Recovery 26

Chapter 4 Handling Received Errors 29

Passing Errors Up the Error-Responder Chain 29

Customizing an Error Object 30

Chapter 5 Recovering From Errors 33

Document Revision History 37

Index 39

Figures, Tables, and Listings

Chapter 1 **Error Objects, Domains, and Codes** 9

- Figure 1-1 The localized strings of an `NSError` object 13
- Table 1-1 Header files for error codes in major domains 10
- Table 1-2 Header files declaring error codes 10
- Listing 1-1 Part of the POSIX error-code declarations (`errno.h`) 11
- Listing 1-2 Testing for particular error codes in a specific domain 11

Chapter 2 **Using and Creating Error Objects** 15

- Listing 2-1 Handling an `NSError` object returned from a `UIKit` method 15
- Listing 2-2 Displaying an alert composed mostly from error-object attributes 17
- Listing 2-3 Assigning custom message strings based on error domain and code 17
- Listing 2-4 Displaying a document-modal error alert 18
- Listing 2-5 Modal delegate handling the user response 19
- Listing 2-6 Directly displaying an error alert dialog 19
- Listing 2-7 Implementing a method that returns an `NSError` object 19

Chapter 3 **Error Responders and Error Recovery** 23

- Figure 3-1 The error-responder chain—part one 24
- Figure 3-2 The error-responder chain — part two 24
- Figure 3-3 Error-responder chain for document-based applications 25
- Figure 3-4 Error-responder chain for non-document applications with window controllers 25
- Figure 3-5 Error-responder chain for simple (non-document) applications 25

Chapter 4 **Handling Received Errors** 29

- Listing 4-1 Handling an error passed up the error-responder chain 29
- Listing 4-2 Customizing an `NSError` object 30

Chapter 5 **Recovering From Errors** 33

- Listing 5-1 Preparing for error recovery 34
- Listing 5-2 Recovering from the error and informing the modal delegate 34
- Listing 5-3 Modal delegate responding to recovery attempter 35

Introduction to Error Handling Programming Guide For Cocoa

Every program must deal with errors as they occur at runtime. The program, for example, might not be able to open a file, or perhaps it cannot parse an XML document. Often errors such as these require the program to inform the user about them. And perhaps the program can attempt to get around the problem causing the error.

Cocoa (and Cocoa Touch) offer developers programmatic tools for these tasks: the `NSError` class in Foundation and new methods and mechanisms in the Application Kit to support error handling in applications. An `NSError` object encapsulates information specific to an error, including the domain (subsystem) originating the error and the localized strings to present in an error alert. With an application there is also an architecture allowing the various objects in an application to refine the information in an error object and perhaps to recover from the error. This document describes this API and architecture and explains how to use them.

Important: The `NSError` class is available on both Mac OS X and iPhone OS. However, the error-responder and error-recovery APIs and mechanisms are available only in the Application Kit (Mac OS X).

Although the `NSError` class was introduced in Mac OS X v10.3, several methods have been added to the class and to the Application Kit in Mac OS X v10.4 to support error handling as described in this document.

Organization of This Document

Error Handling Programming Guide for Cocoa has the following articles:

- [“Error Objects, Domains, and Codes”](#) (page 9) describes the attributes of an `NSError` object, particularly its domain and error code, and discusses the possible contents of an error object’s “user info” dictionary, including localized message strings and underlying errors.
- [“Using and Creating Error Objects”](#) (page 15) explains how to evaluate an error, how to display an error message using an `NSError` object, and how to implement methods that return an `NSError` object by reference.
- [“Error Responders and Error Recovery”](#) (page 23) describes the Application Kit architecture for passing error objects up a chain of objects in an application, giving each object a chance to customize the error before it is presented. It also discusses the role of the recovery attempter, an object designated to attempt a recovery from an error if the user requests it.
- [“Handling Received Errors”](#) (page 29) discusses how, in the chain of error-responder objects, you handle a received error and customize it.
- [“Recovering From Errors”](#) (page 33) explains the procedure for attempting a user-requested recovery from an error.

The two chapters relevant to iPhone OS are [“Error Objects, Domains, and Codes”](#) (page 9) and [“Using and Creating Error Objects”](#) (page 15).

See Also

“Error Handling in the Document Architecture” in *Document-Based Applications Overview* offers valuable advice for subclasses that override methods with a by-reference `NSError` parameter (Mac OS X).

“Types of Dialogs and When to Use Them” in *Apple Human Interface Guidelines* offers advice on the form and content of alerts in Mac OS X. *Alerts, Action Sheets, and Modal Views* offers similar advice for alerts on iPhone OS. You should consult these guidelines before composing your error messages. Also take a look at the following documents discussing areas of Cocoa programming related to error handling and the presentation of error messages:

- *Assertions and Logging* (both platforms)
- *Dialogs and Special Panels* (alerts on Mac OS X)
- *Sheet Programming Topics for Cocoa* (Mac OS X)
- *UIAlertView Class Reference* (iPhone OS)

Exception Programming Topics for Cocoa discusses how to raise and handle exceptions. Exception Handling in *The Objective-C Programming Language* describes the compiler directives `@try`, `@catch`, `@throw`, and `@finally`, which are used in exception handling.

Error Objects, Domains, and Codes

Cocoa programs use `NSError` objects to convey information about runtime errors that users need to be informed about. In most cases, a program displays this error information in a dialog or sheet. But it may also interpret the information and either ask the user to attempt to recover from the error or attempt to correct the error on its own.

The core attributes of an `NSError` object—or, simply, an error object—are an error domain, a domain-specific error code, and a “user info” dictionary containing objects related to the error, most significantly description and recovery strings. This chapter explains the reason for error objects, describes their attributes, and discusses how you use them in Cocoa code.

Why Have Error Objects?

Because they are objects, instances of the `NSError` class have several advantages over simple error codes and error strings. They encapsulate several pieces of error information at once, including localized error strings of various kinds. `NSError` objects can also be archived and copied, and they can be passed around in an application and modified. And although `NSError` is not an abstract class (and thus can be used directly) you can extend the `NSError` class through subclassing.

Because of the notion of layered error domains, `NSError` objects can embed errors from underlying subsystems and thus provide more detailed and nuanced information about an error. Error objects also provide a mechanism for error recovery by holding a reference to an object designated as the recovery attempter for the error.

Error Domains

For largely historical reasons, error codes in Mac OS X are segregated into domains. For example, Carbon error codes, which are typed as `OSStatus`, have their origin in versions of the Macintosh operating system predating Mac OS X. On the other hand, POSIX error codes derive from the various POSIX-conforming “flavors” of UNIX, such as BSD. The Foundation framework declares in `NSError.h` the following string constants for the four major error domains:

```
NSMachErrorDomain
NSPOSIXErrorDomain
NSOSStatusErrorDomain
NSCocoaErrorDomain
```

The above sequence of domain constants indicates the general layering of the domains, with the Mach error domain at the lowest layer. You get the domain of an error by sending an `NSError` object a `domain` message.

In addition to the four major domains, there are error domains that are specific to frameworks or even to groups of classes or individual classes. For example, the Web Kit framework has its own domain for errors in its Objective-C implementation, `WebKitErrorDomain`. Within the Foundation framework, the URL classes have their own error domain (`NSURLErrorDomain`) as do the XML classes (`NSXMLParserErrorDomain`). The `NSStream` class itself defines two error domains, one for SSL errors and the other for SOCKS errors.

The Cocoa error domain (`NSCocoaErrorDomain`) includes all error codes for the Cocoa frameworks—except, of course, for error codes in class-specific domains of those frameworks. These frameworks include not only Foundation, UIKit, and Application Kit, but Core Data and potentially other Objective-C frameworks. (Error domains within the Cocoa frameworks that are separate from the Cocoa error domain were defined before the latter was introduced.)

Domains serve several useful purposes. They give Cocoa programs a way to identify the Mac OS X subsystem that is detecting an error. They also help to prevent collisions between error codes from different subsystems with the same numeric value. In addition, domains allow for a causal relationship between error codes based on the layering of subsystems; for example, an error in the `NSOSStatusErrorDomain` may have an underlying error in the `NSMachErrorDomain`.

You can create your own error domains and error codes for use in your own frameworks, or even in your own applications. It is recommended that the string constant for the domain be of the form `com.company.framework_or_app.ErrorDomain`.

Error Codes

An error code identifies a particular error in a particular domain. It is a signed integer assigned as the value of a program symbol. You get the error code by sending an `NSError` object a `code` message. As listed in Table 1-1, error codes are declared and documented in one or more header files for each major domain.

Table 1-1 Header files for error codes in major domains

Domain	Header file
Mach	<code>/usr/include/mach/kern_return.h</code>
POSIX	<code>/usr/include/sys/errno.h</code>
Carbon (OSStatus)	<code>/System/Library/Frameworks/CoreServices.framework/Frameworks/CarbonCore.framework/Headers/OSStatus.h</code>
Cocoa	See Table 1-2.

Table 1-2 lists the frameworks and header files where error codes in the Cocoa domain are currently declared.

Table 1-2 Header files declaring error codes

Framework/Header	Description
<code><Foundation/FoundationErrors.h></code>	Generic Foundation error codes
<code><AppKit/AppKitErrors.h></code>	Generic Application Kit error codes
<code><CoreData/CoreDataErrors.h></code>	Core Data error codes

To give an idea of how you might test for and act upon errors, let's say you want to test for underlying POSIX errors during an operation that writes to a file. (Underlying errors are explained in [“Underlying Error”](#) (page 14).) If you consulted the POSIX error codes declared in `/usr/include/sys/errno.h`, you would see a list similar to Listing 1-1.

Listing 1-1 Part of the POSIX error-code declarations (`errno.h`)

```
#define EPERM      1      /* Operation not permitted */
#define ENOENT     2      /* No such file or directory */
#define ESRCH     3      /* No such process */
#define EINTR     4      /* Interrupted system call */
#define EIO       5      /* Input/output error */
#define ENXIO     6      /* Device not configured */
#define E2BIG     7      /* Argument list too long */
#define ENOEXEC   8      /* Exec format error */
#define EBADF     9      /* Bad file descriptor */
#define ECHILD   10     /* No child processes */
#define EDEADLK  11     /* Resource deadlock avoided */
                        /* 11 was EAGAIN */
#define ENOMEM   12     /* Cannot allocate memory */
#define EACCES   13     /* Permission denied */
#define EFAULT   14     /* Bad address */#H
```

You could choose the error conditions you want to test for and use them in code similar to that in Listing 1-2.

Listing 1-2 Testing for particular error codes in a specific domain

```
// underError is underlying-error object of a Cocoa-domain error
if ( [[underError domain] isEqualToString:NSPOSIXErrorDomain] ) {
    switch([underError code]) {
        case EIO:
        {
            // handle POSIX I/O error
        }
        case EACCES:
        {
            // handle POSIX permissions error
        }
        // etc.
    }
}
```

You may declare your own error codes for use by your own applications or frameworks, but the error codes should belong to your own domain. You should never add error codes to an existing domain that you do not “own.”

The User Info Dictionary

Every `NSError` object has a “user info” dictionary to hold error information beyond domain and code. You access this dictionary by sending a `userInfo` message to an `NSError` object. The advantage of an `NSDictionary` object over another kind of container object is that it is flexible; it can even carry custom information about an error. But all user info dictionaries contain (or can contain) several predefined string and object values related to an error.

Localized Error Information

An important role for `NSError` objects is to contain error information that programs can display in an alert dialog or sheet. This information is usually stored in the user info dictionary as strings in several categories: description, failure reason, recovery suggestion, and recovery options. (See [Figure 1-1](#) (page 13) for the placement of these strings on an alert.) When you create an `NSError` object, you should insert localized strings into the dictionary, unless you want to compute them lazily.

Note: Don't expect the user info dictionary of every error object to contain localized strings. A subclass of `NSError`, for example, could override `localizedDescription` to compose these strings on-the-fly from the error domain, code, and context instead of storing them.

You can usually access the localized information associated with an `NSError` object in one of two ways. You can send `objectForKey:` to the user info dictionary, specifying the appropriate key. Or you can send an equivalent message to the `NSError` object. However, you should send the message rather than use the dictionary key to access a localized string. The error object might not store the string in the dictionary, instead choosing to compose it dynamically. The dictionary is designed to be a fallback mechanism, not the sole repository of error strings. Use the dictionary keys instead to store your own strings in the user info dictionary.

The following summaries include both the dictionary key and the method used to access the localized string:

Error description

The main description of the error, appearing in a larger, bold type face. It often includes the failure reason. If no error description is present in the user info dictionary, `NSError` either constructs one from the error domain and code or (when the domain is well-known, such as `NSCocoaErrorDomain`), attempts to fetch a suitable string from a function or method within that domain. .

User info key: `NSLocalizedStringKey`

Method: `localizedDescription` (never returns `nil`)

Failure reason

A brief sentence that explains the reason why the error occurred. It is typically part of the error description. Methods such as `presentError:` do not automatically display the failure reason because it is already included in the error description. The failure reason is for clients that only want to display the reason for the failure.

User info key: `NSLocalizedFailureReasonErrorKey`

Method: `localizedFailureReason` (can return `nil`)

Note: An example can help to clarify the relationship between error description and failure reason. An error object has an error description of “File could not be saved because the disk is full.” The accompanying failure reason is “The disk is full.”

Recovery suggestion

A secondary sentence that ideally tells users what they can do to recover from the error. It appears beneath the error description in a lighter type face. If the recovery suggestion refers to the buttons of the error alert, it should use the same titles as specified for recovery options (`NSLocalizedStringRecoveryOptionsErrorKey`). You may use this string as a purely informative message, supplementing the error description and failure reason.

User info key: `NSLocalizedStringRecoverySuggestionErrorKey`

Method: `localizedRecoverySuggestion` (can return nil)

Recovery options

An array of titles (as strings) for the buttons of the error alert. By default, alert sheets and dialogs for error messages have only the “OK” button for dismissing the alert. The first string in the array is the title of the rightmost button, the next string is the title of the button just to the left of the first, and so on. Note that if a recovery attempter is specified for the error object, the recovery-options array should contain more than one string. The recovery attempter accesses the recovery options in order to interpret user choices.

User info key: `NSLocalizedStringRecoveryOptionsErrorKey`

Method: `localizedRecoveryOptions` (if returns nil, implies a single “OK button”)

Figure 1-1 The localized strings of an NSError object



Note: Beginning with Mac OS X version 10.4, you can use the `alertWithError:` class method of `NSAlert` as a convenience for creating `NSAlert` objects to use when displaying alert dialogs or sheets. The method extracts the localized information from the passed-in `NSError` object for its message text, informative text, and button titles. You may also use the `presentError:` message to display error alerts.

To internationalize your error strings, create a `.strings` file for each localization and place the file in an appropriately named `.lproj` subdirectory of your bundle’s `Resources` directory. Then use one of the `NSLocalizedString` macros to add localized strings to the user info dictionary of an `NSError` object. For more on internationalization and string localization, see *Internationalization Programming Topics*.

Note: For many error objects in the Cocoa error domain, the localization is performed on demand; in these cases, the localized values are not stored in the user info dictionary.

The Recovery Attempter

An `NSError` object's user info dictionary can also contain a recovery attempter. A recovery attempter is an object that implements one or more methods of the `NSErrorRecoveryAttempting` informal protocol. In many cases, this is the same object that creates the `NSError` object, but it can be any other object that may know how to recover from a particular error.

If a recovery attempter has been specified for an `NSError` object, and multiple recovery options have also been specified, when the error alert is displayed and the user selects a recovery option, the recovery attempter is given a chance to recover from the error. You access the recovery attempter by sending `recoveryAttempter` to the `NSError` object. You can add an recovery attempter to the user info dictionary using the key `NSRecoveryAttempterErrorKey`.

For more on the recovery-attempter object and its role in error handling, see [“Error Responders and Error Recovery”](#) (page 23).

Underlying Error

The user info dictionary can sometimes include another `NSError` object that represents an error in a subsystem underlying the error represented by the containing `NSError`. You can query this underlying error object to obtain more specific information about the cause of the error.

You access the underlying error object by using the `NSUnderlyingErrorKey` dictionary key.

Domain-Specific Keys

Many of the various error domains specify keys for accessing particular items of information from the user info dictionary. This information supplements the other information in the error object. For example, the Cocoa domain defines the keys `NSStringEncodingErrorKey`, `NSURLLErrorKey`, and `NSFilePathErrorKey`.

Check the header files or documentation of error domains to find out what domain-specific keys they declare.

Using and Creating Error Objects

The following sections describe how to deal with `NSError` objects returned from framework methods, how to display error messages using error objects, how to create error objects, and how to implement methods that return error objects by reference.

Handling Error Objects Returned From Methods

Many methods of the Cocoa and Cocoa Touch classes include as their last parameter a direct or indirect reference to an `NSError` object. In some Foundation and UIKit methods you find `NSError` objects as arguments of delegation methods. The following declaration is from the UIKit framework's `UIWebViewDelegate` protocol; a delegate would implement a method such as this to find out if an operation failed:

```
- (void)webView:(UIWebView *)webView didFailLoadWithError:(NSError *)error;
```

Some methods of the Cocoa frameworks that you call include an indirect reference to an `NSError` object; these methods typically perform an operation such as creating a document, writing a file, or loading a URL. For example, the following method declaration is from the `NSDocument` class header file:

```
- (BOOL)writeToURL:(NSURL *)absoluteURL
    ofType:(NSString *)typeName
    error:(NSError **)outError;
```

If a method such as this encounters an error in its implementation, it directly returns `NO` to indicate failure and indirectly returns (if the client code requests it) an `NSError` object in the last parameter to describe the error.

If you want to evaluate the error, declare an `NSError` object variable before calling a method such as `writeToURL:ofType:error:`. When you invoke the method, pass in a pointer to this variable. (If you are not interested in the error, just pass `NULL`.) If the method directly returns `nil` or `NO`, inspect the `NSError` object to determine the cause of the error or simply display an error alert. Listing 2-1 illustrates this approach.

Listing 2-1 Handling an `NSError` object returned from a AppKit method

```
NSError *theError = nil;
BOOL success = [myDoc writeToURL:[self docURL] ofType:@"html" error:&theError];

if (success == NO) {
    // Maybe try to determine cause of error and recover first.
    UIAlertView *theAlert = [UIAlertView alertWithError:theError];
    [theAlert runModal]; // Ignore return value.
}
```

Important: Success or failure is indicated by the return value of the method. Although Cocoa methods that indirectly return error objects in the Cocoa error domain are guaranteed to return such objects if the method indicates failure by directly returning `nil` or `NO`, you should always check that the return value is `nil` or `NO` before attempting to do anything with the `NSError` object.

This code in Listing 2-1 uses the returned `NSError` to display an error alert to the user immediately. (`UIAlertView`, the UIKit class corresponding to `NSAlert`, has no equivalent method for `alertWithError:`.) Error objects in the Cocoa domain are always localized and ready to present to users, so they can often be presented without further evaluation.

Instead of merely displaying an error message based on an `NSError` object, you could examine the object to determine if you can do something else. For example, you might be able to perform the operation again in a slightly different way that circumvents the error; if you do this, however, you should first request the user's permission to perform the modified operation.

When evaluating an `NSError` object, always use the object's domain and error code as the bases of tests and not the strings describing the error or how to recover from it. Strings are typically localized and are thus likely to vary. With a few exceptions (such as the `NSURLErrorDomain` domain), pre-existing errors returned from Cocoa framework methods are always in the `NSCocoaErrorDomain` domain; however, because there are exceptions you might want to test whether the top-level error belongs to that domain. Error objects returned from Cocoa methods can often contain underlying error objects representing errors returned by lower subsystems, such as the BSD layer (`NSPOSIXErrorDomain`).

Of course, to make a successful evaluation of an error, you have to anticipate the errors that might be returned from a method invocation. And you should ensure that your code deals adequately with new errors that might be returned in the future.

Important: You should always special-case test for the `NSUserCancelledError` error code (in the `NSCocoaErrorDomain`). This code indicates that the user cancelled the operation (for example, by pressing Command-period). In this case, you should not display any error dialog.

Error-Handling Alternatives in Mac OS X

If you are developing a Mac OS X application, there are many other things you can do upon receiving an `NSError` object:

- If you know how to recover from the error, but require the user's approval, you could create a new version of the error object that adds a recovery attempter to it (see [“Recovering From Errors”](#) (page 33)).
- Send the error object up the error-responder chain so that other objects in the application can add to it or try to recover from the error.

Before doing this, you might be able supplement the information in the error from the current programming context and then create a new error object that contains this enriched information.

- If you use the returned `NSError` object as the basis of a new error object, either by adding a recovery attempter or supplementary information, you can either:
 - ❑ Display the message immediately.
 - ❑ Pass the error on to the next error responder.

For more on customizing errors passed up the error-responder chain, see [“Handling Received Errors”](#) (page 29).

Displaying Information From Error Objects

There are several different ways to display the information in `NSError` objects. You could extract the localized description (or failure reason), recovery suggestion, and the recovery options from the error object and use them to initialize the tiles and message text of an `NSAlert`, `UIAlertView`, or `UIActionSheet` object (or a Mac OS X modal document sheet). This universal approach gives you a large degree of control over the content and presentation of the error alert.

For example, the code in Listing 2-2 composes the message text of an `UIAlertView` object from the localized description and failure reason taken from the passed-in `NSError` object.

Listing 2-2 Displaying an alert composed mostly from error-object attributes

```
- (void)webView:(UIWebView *)webView didFailLoadWithError:(NSError *)error {
    NSString *titleString = @"Error Loading Page";
    NSString *messageString = [error localizedDescription];
    NSString *moreString = [error localizedFailureReason] ?
        [error localizedFailureReason] :
        NSLocalizedString(@"Try typing the URL again.", nil);
    messageString = [NSString stringWithFormat:@"%@. %@", messageString,
moreString];

    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:titleString
        message:messageString delegate:self
        cancelButtonTitle:@"Cancel" otherButtonTitles:nil];
    [alertView show];
    [alertView release];
}
```

However, you don't have to use the localized error strings made available by the frameworks. For example, if you think they're not descriptive enough or want to supplement them with context-specific information, you can identify the error by its domain and code and then substitute your own string values. Take the delegate method used in the prior example; the error object passed to the delegate in `webView:didFailLoadWithError:` is almost always of the `NSURLErrorDomain` domain. You could then find out which code of this domain is associated with the error and substitute your own string for the strings contained in the error object. (`NSURLErrorDomain` and its codes are declared in `NSURLError.h`.) Listing 2-3 gives an example of this.

Listing 2-3 Assigning custom message strings based on error domain and code

```
- (void)webView:(UIWebView *)webView didFailLoadWithError:(NSError *)error {
    NSString *errorMsg = nil;

    if ([[error domain] isEqualToString:NSURLErrorDomain]) {
        switch ([error code]) {
            case NSURLErrorCannotFindHost:
                errorMsg = NSLocalizedString(@"Cannot find specified host. Retype
URL.", nil);
```

```

        break;
    case NSErrorCannotConnectToHost:
        errorMsg = NSLocalizedString(@"Cannot connect to specified host.
Server may be down.", nil);
        break;
    case NSErrorNotConnectedToInternet:
        errorMsg = NSLocalizedString(@"Cannot connect to the internet.
Service may not be available.", nil);
        break;
    default:
        errorMsg = [error localizedDescription];
        break;
    }
} else {
    errorMsg = [error localizedDescription];
}

UIAlertView *av = [[UIAlertView alloc] initWithTitle:
    NSLocalizedString(@"Error Loading Page", nil)
    message:errorMsg delegate:self
    cancelButtonTitle:@"Cancel" otherButtonTitles:nil];
[av show];
[av release];
}

```

Propagating Errors for Display by the Application Object (Mac OS X)

The Application Kit provides a few shortcuts for displaying error alerts. The `presentError:` and the `presentError:modalForWindow:delegate:didPresentSelector:contextInfo:` methods permit you to originate an error alert that is eventually displayed by the global application object, `NSApp`; the former method requests an application-modal alert and the latter a document-modal alert. You must send either of these present-error messages to an object in the error-responder chain (see [“The Error-Responder Chain”](#) (page 23)): a view object, a window object, an `NSDocument` object, an `NSWindowController` object, an `NSDocumentController` object, or `NSApp`. (If you send the message to a view, it should ideally be a view object associated in some way with the condition that produced the error.) Listing 2-4 illustrates how you might invoke the document-modal `presentError:modalForWindow:delegate:didPresentSelector:contextInfo:` method.

Listing 2-4 Displaying a document-modal error alert

```

NSError *theError;
NSData *theData = [doc dataOfType:@"xml" error:&theError];
if (!theData && theError)
    [anyView presentError:theError
     modalForWindow:[doc windowForSheet]
     delegate:self
     didPresentSelector:
         @selector(didPresentErrorWithRecovery:contextInfo:)
     contextInfo:nil];

```

After the user dismisses the alert, `NSApp` invokes a method (identified in the `didPresentSelector:` keyword) implemented by the modal delegate. As Listing 2-5 shows, the modal delegate in this method checks whether the recovery-attempter object (if any) managed to recover from the error and responds accordingly.

Listing 2-5 Modal delegate handling the user response

```
- (void)didPresentErrorWithRecovery:(BOOL)recover contextInfo:(void *)info {
    if (recover == NO) { // Recovery did not succeed, or no recovery attempted.
        // Proceed accordingly.
    }
}
```

For more on the recovery-attempter object, see [“Recovering From Errors”](#) (page 33).

Sometimes you might not want to send an error object up the error-responder chain to be displayed by NSApp. You would rather show an error alert to the user immediately, and not have to construct it yourself. The `UIAlertView` class provides the `alertWithError:` method for this purpose.

Listing 2-6 Directly displaying an error alert dialog

```
UIAlertView *theAlert = [UIAlertView alertWithError:theError];
NSInteger button = [theAlert runModal];
if (button != UIAlertViewFirstButtonReturn) {
    // handle
}
```

Note: The `presentError:` and `presentError:modalForWindow:delegate:didPresentSelector:contextInfo:` methods silently ignore `NSUserCancelledError` errors in the `NSCocoaErrorDomain` domain.

Creating and Returning NSError Objects

You can declare and implement your own methods that indirectly return an `NSError` object. Methods that are good candidates for `NSError` parameters are those that open and read files, load resources, parse formatted text, and so on. In general, these methods should not indicate an error through the existence of an `NSError` object. Instead, they should return `NO` or `nil` from the method to indicate that an error occurred. Return the `NSError` object to describe the error.

If you are going to return an `NSError` object by reference in an implementation of such a method, you must create the `NSError` object. You create an error object either by allocating it and then initializing it with the `initWithDomain:code:userInfo:` method of `NSError` or by using the class factory method `errorWithDomain:code:userInfo:`. As the keywords of both methods indicate, you must supply the initializer with a domain (string constant), an error code (a signed integer), and a “user info” dictionary containing descriptive and supporting information. (See [“Error Objects, Domains, and Codes”](#) (page 9) for full descriptions of these data items.) You should ensure that all strings in the user info dictionary are localized. If you create an `NSError` object with `initWithDomain:code:userInfo:`, you should send `autorelease` to it before you return it to the caller.

Listing 2-7 is an example of a method that, for the purpose of illustration, calls the POSIX-layer `open` function to open a file. If this function returns an error, the method creates an `NSError` object of the `NSPOSIXErrorDomain` that is used as the underlying error of a custom error domain returned to the caller.

Listing 2-7 Implementing a method that returns an `NSError` object

```
- (NSString *)fooFromPath:(NSString *)path error:(NSError **)anError {
```

```

const char *fileRep = [path fileSystemRepresentation];
int fd = open(fileRep, O_RDWR|O_NONBLOCK, 0);

if (fd == -1) {

    if (anError != NULL) {
        NSString *description = nil;
        NSDictionary *uDict = nil;
        int errCode;

        if (errno == ENOENT) {
            description = NSLocalizedString(@"No file or directory at requested
location", @"");
            errCode = MyCustomNoFileError;
        } else if (errno == EIO) {
            // Continue for each possible POSIX error...
        }

        // Make underlying error.
        NSError *underlyingError = [[[NSError alloc] initWithDomain:NSPOSIXErrorDomain
            code:errno userInfo:nil] autorelease];
        // Make and return custom domain error.
        NSArray *objArray = [NSArray arrayWithObjects:description, underlyingError,
path, nil];
        NSArray *keyArray = [NSArray arrayWithObjects:NSLocalizedDescriptionKey,
            NSUnderlyingErrorKey, NSFilePathErrorKey, nil];
        NSDictionary *eDict = [NSDictionary dictionaryWithObjects:objArray
            forKeys:keyArray];

        *anError = [[[NSError alloc] initWithDomain:MyCustomErrorDomain
            code:errCode userInfo:eDict] autorelease];
    }
    return nil;
}
// ...

```

In this example, the returned error object includes in its user info dictionary the path that caused the error.

As the example in Listing 2-7 shows, you can use errors that originate from underlying subsystems as the basis for error objects that you return to callers. You can use raised exceptions that your code handles in the same way. An `NSException` object is compatible with an `NSError` object in that its attributes are a name, a reason, and a user info dictionary. You can easily transfer information in the exception object over to the error object.

A Note on Errors and Exceptions

It is important to keep in mind the difference between error objects and exception objects in Cocoa and Cocoa Touch, and when to use one or the other in your code. They serve different purposes and should not be confused.

Exceptions (represented by `NSException` objects) are for programming errors, such as an array index that is out of bounds or an invalid method argument. User-level errors (represented by `NSError` objects) are for runtime errors, such as when a file cannot be found or a string in a certain encoding cannot be read. Conditions

giving rise to exceptions are due to programming errors; you should deal with these errors before you ship a product. Runtime errors can always occur, and you should communicate these (via `NSError` objects) to the user in as much detail as is required.

Although exceptions should ideally be taken care of before deployment, a shipped application can still experience exceptions as a result of some truly exceptional situation such as “out of memory” or “boot volume not available.” It is best to allow the highest level of the application—the global application object itself—to deal with these situations.

Error Responders and Error Recovery

As “[Why Have Error Objects?](#)” (page 9) points out, `NSError` objects bring considerable advantages to Cocoa programming. But the Cocoa frameworks also give `NSError` objects a prominent role to play in architectures for error presentation and error recovery. These architectures enhance the usefulness of error objects. They make it possible for Cocoa applications to present users with a richer and more customizable range of messages, and to attempt recovery from errors as well as informing users of them.

Note: The error-presentation and error-recovery architectures described in this chapter are available only on the Mac OS X platform.

The Error-Responder Chain

The Application Kit, largely through the `NSResponder` class, defines a mechanism known as the responder chain by which events and action messages in an application are passed up the view hierarchy to windows and eventually to the application object. The Application Kit defines a similar chain of objects for error handling and presentation.

To initiate the journey of an `NSError` object up the error-responder chain, you can send one of two messages to any object in the chain:

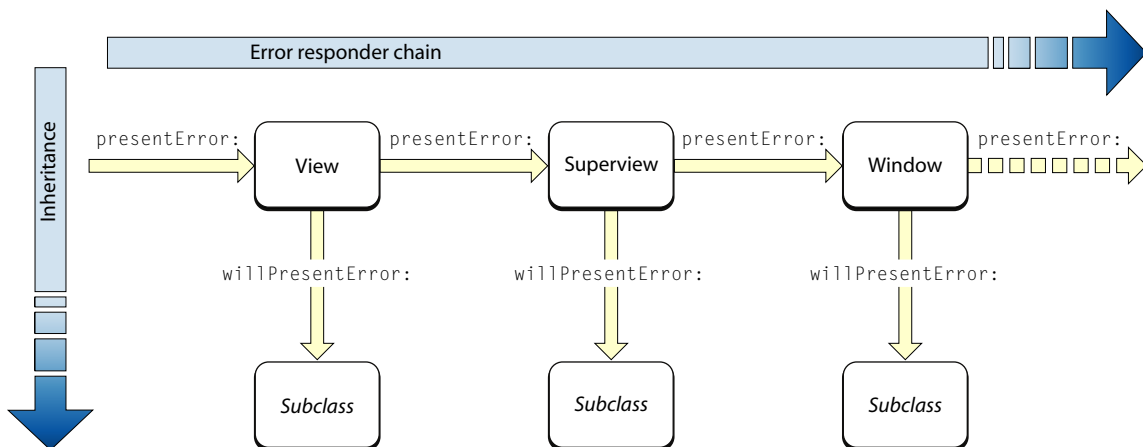
- `presentError:` — for error messages displayed in application-modal alert dialogs
- `presentError:modalForWindow:delegate:didPresentSelector:contextInfo:` — for error messages displayed in document-modal alert sheets

Although these methods are declared by the `NSResponder` class, you may also send them to objects of the `NSDocument` and `NSDocumentController` classes. (The `NSResponder` class is the superclass, of course, of the `NSView`, `NSWindow`, `NSApplication`, and `NSWindowController` classes.)

The default behavior of both the `presentError:...` methods—except for the `NSApplication` implementation—is to send `willPresentError: to self` before forwarding the `presentError:..` message to the next object in the chain. Subclasses can implement the `willPresentError: to self` method to inspect the passed-in `NSError` object and return a customized object. Subclasses might want to do this if they know more than their superclass about the conditions giving rise to the error or if they know best how to recover from it.

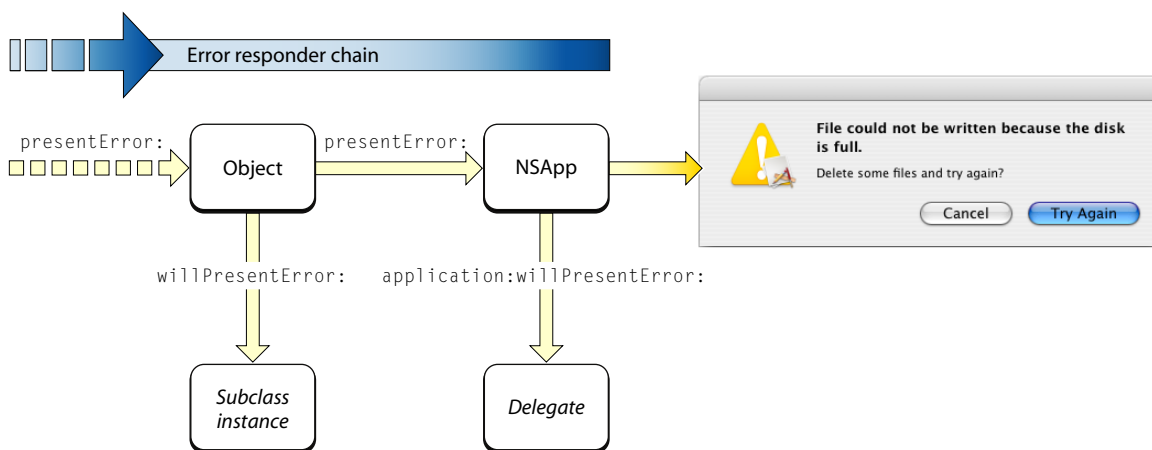
For the purposes of illustration, assume that a view object well down the view hierarchy receives the `presentError: message`. As Figure 3-1 shows, it sends `willPresentError: to self` and then sends `presentError: to its superview`, passing it any modified `NSError` object. The `superviews` of the originating view do the same thing until finally the window’s content view sends the `presentError: message` to its window object.

Figure 3-1 The error-responder chain—part one



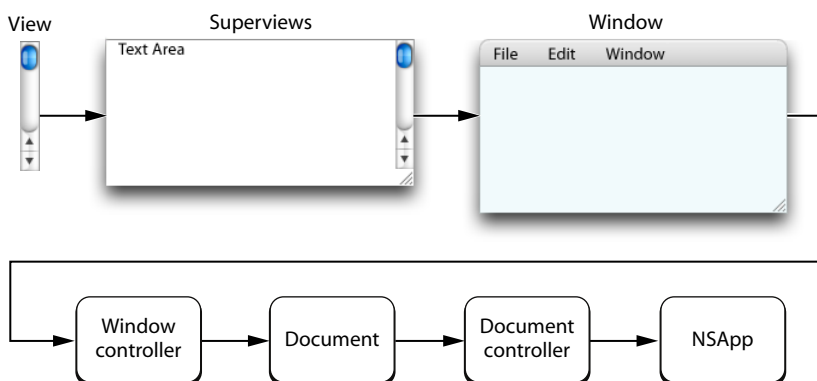
The `presentError:` message proceeds up the chain of error responders in this fashion until it reaches the global application object, `NSApp`. As Figure 3-2 depicts, `NSApp` sends the `application:willPresentError:` message to its delegate, giving it the same opportunity as the subclass objects in the chain to inspect the error object and possibly modify it—but without the need for a custom subclass. When the delegate returns, `NSApp` displays the error as (in this case) an alert dialog.

Figure 3-2 The error-responder chain — part two

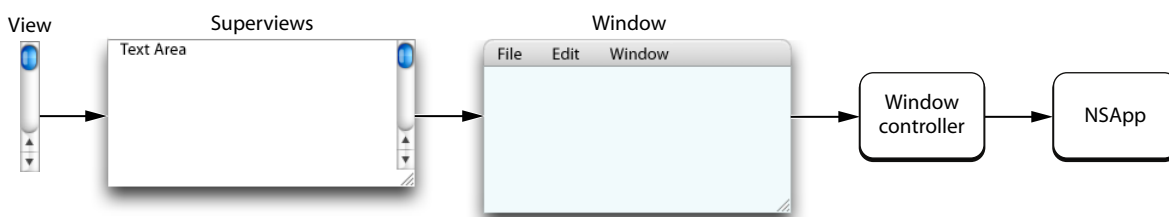


Important: Overriding the `presentError:modalForWindow:delegate:didPresentSelector:contextInfo:` method or `presentError:` method is not recommended.

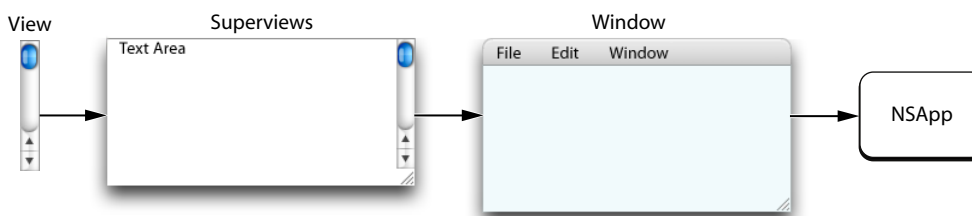
The exact sequence of objects in the error-responder chain varies according to the type of application. For document-based applications, the error-responder chain includes document objects, window controllers, and document controllers as well as views, windows, and `NSApp` (Figure 3-3).

Figure 3-3 Error-responder chain for document-based applications

Some Cocoa applications are not document-based but still use one or more window controllers. Figure 3-4 shows the sequence of objects in this error-responder chain.

Figure 3-4 Error-responder chain for non-document applications with window controllers

Finally, simple Cocoa applications—those that are not document-based and that don't use window controllers—have an error-responder sequence as depicted in Figure 3-5.

Figure 3-5 Error-responder chain for simple (non-document) applications

Error Customization

As described in the preceding section, all along the error-responder chain custom subclasses of objects in the chain are given the opportunity to inspect and customize an `NSError` object if they implement the `willPresentError:` method. Near the end of the chain the application delegate has the same opportunity in `application:willPresentError:..` What kind of tests and customizations can take place in these methods?

In either method, you probably first want to determine what the error is. When doing this, test the `NSError` object's domain and error code against the constants that are probably related to the error condition. Do not evaluate the description or recovery strings as these can vary, especially when they are localized. You might also narrow down the cause of the error by using domain-specific keys to extract various pieces of information from the user info dictionary.

Important: You should always special-case test for the `NSUserCancelledError` error code (in the `NSCocoaErrorDomain`). This code indicates that the user cancelled the operation (for example, by pressing Command-period). In this case, you should not display any error dialog.

You might also want to find out if the error object has an underlying error; you can access this object from the user info dictionary with the `NSUnderlyingErrorKey` key. If there is an underlying error, and this object has a failure reason in its user info dictionary, you can append this localized string to the error description to create a more informative description.

If you decide that you know how to recover from the error, you can add an object to the user info dictionary as the recovery attempter. For a recovery attempter to be effective, it must satisfy the requirements summarized in “[Error Recovery](#)” (page 26).

If you are customizing a received `NSError` object to have a custom error domain and error code, you may choose to store the original error in the user info dictionary as an underlying error. Use the key `NSUnderlyingErrorKey` for this purpose (or override the `recoveryAttempter` method).

You cannot modify a received `NSError` object because the class provides no setter methods and the user info dictionary is immutable. When customizing an error, you must create a new `NSError` object, initializing with new data plus data from the old error object that you want to carry over. See “[Using and Creating Error Objects](#)” (page 15) for explicit instructions and examples.

Error Recovery

A recovery attempter is an object designated to attempt, upon user request, a recovery from a specific error. For example, say that a program cannot save a file because it is locked. The recovery attempter could try to unlock it first before overwriting it.

The error recovery mechanism is similar to the delegation design pattern in that a designated object—the recovery attempter—is asked to respond to a user action. An `NSError` object can encapsulate a recovery attempter and recovery options, which is an array of button titles to display in the error alert. Among the button titles is one requesting error recovery. When an error alert is displayed and the user clicks a button, the application sends a message to the recovery attempter, passing it the index of the button that was clicked. If the the “recover” button was clicked, the recovery attempter tries to complete the operation in a way that avoids the error or fixes the condition that gives rise to it. Finally, the recovery attempter informs either the application object or the document-modal sheet delegate whether it was successful.

There are three requirements for error recovery to occur as a result of a user choice:

- The recovery-attempter object must implement one of the `NSErrorRecoveryAttempting` informal protocol methods:
`attemptRecoveryFromError:optionIndex:delegate:didRecoverSelector:contextInfo:`
or `attemptRecoveryFromError:optionIndex:`, depending on whether the error alert is document-modal (sheet) or application-modal (dialog), respectively.

- The `recoveryAttempter` method must return a suitable object. To ensure this, you can add the recovery attempter to the user info dictionary as the value of `NSRecoveryAttempterErrorKey`, or you can override the `recoveryAttempter` method.
- The `localizedRecoveryOptions` must return an array of button titles (including the title of the button that requests error recovery). To ensure this, you can add the array to the user info dictionary as the value of `NSLocalizedRecoveryOptionsErrorKey`, or you can override the `localizedRecoveryOptions` method.

For the complete procedure for error recovery, including sample code, see [“Recovering From Errors”](#) (page 33).

Handling Received Errors

Note: The error-presentation and error-responder architecture described in this chapter is available only on the Mac OS X platform.

When you send a `presentError:` or

`presentError:modalForWindow:delegate:didPresentSelector:contextInfo:` message to certain eligible objects, the message travels up a sequence of objects in an application called the error-responder chain (see “[The Error-Responder Chain](#)” (page 23)). The default implementation for most objects in this chain is to send the `willPresentError:` method to `self` before sending the `presentError:` message to the next object. The `willPresentError:` message gives instances of custom subclasses an opportunity to look at the error object being passed up the chain and possibly customize it. When the error object reaches the end of the chain, the global application object, `NSApp`, displays an error alert to users; but before `NSApp` displays the error alert, it invokes the method `application:willPresentError:`, giving its delegate the same opportunity.

The following sections discuss strategies for implementing the `willPresentError:` and `application:willPresentError:` methods.

Passing Errors Up the Error-Responder Chain

If you have a subclass of `NSDocument`, `NSDocumentController`, `NSWindowController`, `NSWindow`, `NSPanel`, or any view class, you can override the `willPresentError:` method to customize the presentation of errors. This might be something you want an instance of your subclass to do if it knows more about the context of a particular error than other objects in the application. Generally, an implementation of `willPresentError:` examines the passed-in `NSError` object and if, for example, its localized description is insufficient, or if the subclass knows how to recover from the error, it creates a new `NSError` object and returns it. In most cases, the customized error object retains some information from the passed-in object.

An implementation of the `willPresentError:` method should always use the error domain and error code as the basis for deciding whether to return a customized error object. Do not base the decision on the strings in the user info dictionary for these can be localized and may vary between invocations. If your implementation decides not to customize the error, don't return the passed-in object directly; instead, send the `willPresentError:` message to `super`. Listing 4-1 illustrates some of these strategies.

Listing 4-1 Handling an error passed up the error-responder chain

```
- (NSError *)willPresentError:(NSError *)error {
    if ([[error domain] isEqualToString:NSCocoaErrorDomain]) {

        switch([error code]) {
            case NSFileLockingError:
            case NSFileReadNoSuchFileError:
            { // private method of custom subclass
                return [self customizeError:error];
            }
        }
    }
}
```

```

        }
        default:
            return [super willPresentError:error];
    }
}
}

```

You don't have to make a subclass in order to customize an `NSError` object for presentation. Instead, your application delegate can implement the `application:willPresentError:` method. The same observations and guidelines given for `willPresentError:` above apply to the implementation of `application:willPresentError:`, except that you can return the original error object directly if you decide not to customize it.

Customizing an Error Object

In the `willPresentError:` example in [Listing 4-1](#) (page 29), a private method is invoked to customize the error object. This is done to clarify the structure of the implementation. But if the customizing code was in-line, it might look some like the `willPresentError:` implementation in [Listing 4-2](#). This code checks if the passed-in object has a failure reason and, if it does, it creates a more application-specific error description, appending the failure reason. Then it creates a new `NSError` object with this different description.

Listing 4-2 Customizing an NSError object

```

- (NSError *)willPresentError:(NSError *)error {
    if ([[Error domain] isEqualToString:NSCocoaErrorDomain]) {

        switch([error code]) {
            case NSFileLockingError:
            case NSFileReadNoSuchFileError:
            {
                NSString *locFailure = [error localizedFailureReason];
                if (locFailure) {
                    NSMutableDictionary *newUserInfo = [NSMutableDictionary
                        dictionaryWithCapacity:[[[error userInfo] allKeys] count]];
                    [newUserInfo setDictionary:[error userInfo]];
                    NSString *errorDesc = [NSString stringWithFormat:
                        NSLocalizedString(@"MyGreatApp cannot open the file. %@, @"),
                        locFailure];
                    [newUserInfo setObject:errorDesc
                        forKey:NSLocalizedStringKey];
                    NSError *newError = [NSError errorWithDomain:[error domain]
                        code:[error code] userInfo:newUserInfo];
                    return newError;
                } else {
                    return [super willPresentError:error];
                }
            }
            default:
                return [super willPresentError:error];
        }
    }
}

```

In this example, the original error object is essentially cloned to make the new one. The new error object contains a more specific error description and appends the failure reason to it.

As noted in [“Passing Errors Up the Error-Responder Chain”](#) (page 29), there is no difference in implementing `willPresentError:` and the delegate method `application:willPresentError:`, except that in the latter method you can return the passed-in error object directly if you do not customize it.

Note: For another example of error-object customization, see [Listing 5-1](#) (page 34) in [“Recovering From Errors”](#) (page 33). In this case, the original object is changed to include recovery options, a recovery suggestion, and a recovery-attempter object.

Recovering From Errors

Note: The error-presentation, error-responder, and error-recovery architectures described in this chapter are available only on the Mac OS X platform.

As described in “[The Recovery Attempter](#)” (page 14), an `NSError` object can have a designated recovery attempter, an object that attempts to recover from the error if the user requests it. The error object holds a reference to the recovery attempter in its user info dictionary, so if the error object is passed around within an application, the recovery attempter stays with it. The user info dictionary must also contain recovery options, an array of localized strings for button titles, one or more of which requests recovery. When the error is presented in an alert and the user selects the recovery option, a message is sent to the recovery attempter, requesting it to do its job.

Ideally, the recovery attempter should be an independent object that knows something about the conditions of an error and how best to circumvent those conditions. An application could even have an object whose role is to recover from errors of various kinds. A recovery attempter must implement at least one of the two methods of the `NSErrorRecoveryAttempting` informal protocol:

`attemptRecoveryFromError:optionIndex:delegate:didRecoverSelector:contextInfo:` or `attemptRecoveryFromError:optionIndex:.` It implements the former method for error alerts presented document-modally, and the latter method for application-modal alerts.

You must also prepare an error object so that error recovery can take place. To do this, add three items to the user info dictionary of the error object:

- The recovery-attempter object under the key `NSRecoveryAttempterErrorKey`
- The recovery options, an array of localized strings, under the key `NSLocalizedRecoveryOptionsErrorKey`
- A recovery-suggestion string, also localized, under the key `NSLocalizedRecoverySuggestionErrorKey`. Although this property is not strictly required, an error alert that offers recovery as an option should display this string, as stipulated by the human-interface guidelines. And if the string refers to particular button titles, it should use the same titles in the recovery-options array.

For guidelines about error alerts on Mac OS X, see *Apple Human Interface Guidelines* (section on dialogs in “Windows”).

Listing 5-1 illustrates a case involving the `NSXMLDocument` class. In this example, an `NSDocument` object attempts to create an internal tree representing an XML document using the `initWithContentsOfURL:options:error:` method of `NSXMLDocument`. If the attempt fails, the usual cause is that the source XML is malformed—for example, there is a missing end tag for an element, or an attribute value is not quoted. If the source XML is “tidied” first to fix the structural problems, it may be possible to create the XML tree.

In the example in Listing 5-1 if the invocation of `initWithContentsOfURL:options:error:` returns an error object by reference, the document object customizes the error object, adding (among other things) a recovery-attempter object, localized recovery options, and a localized recovery suggestion to its user info dictionary. Then it sends `presentError:modalForWindow:delegate:didPresentSelector:contextInfo:` to `self`.

Listing 5-1 Preparing for error recovery

```
- (BOOL)readFromURL:(NSURL *)url ofType:(NSString *)type error:(NSError **)anError {
    NSError *err = nil;

    // xmlDoc is an NSXMLDocument instance variable.
    if (xmlDoc != nil) {
        [xmlDoc release];
        xmlDoc = nil;
    }

    xmlDoc = [[NSXMLDocument alloc] initWithContentsOfURL:url
              options:NSXMLNodeOptionsNone error:&err];

    if (xmlDoc == nil && err) {
        NSString *newDesc = [[err localizedDescription] stringByAppendingString:
                             ([err localizedFailureReason] ? [err localizedFailureReason] : @"")];

        NSMutableDictionary *newDict = [NSMutableDictionary dictionaryWithCapacity:4];
        [newDict setObject:newDesc forKey:NSLocalizedStringKey];
        [newDict setObject:
         NSLocalizedString(@"Would you like to tidy the XML and try again?", @"")
         forKey:NSLocalizedStringRecoverySuggestionErrorKey];
        [newDict setObject:self forKey:NSRecoveryAttempterErrorKey];
        [newDict setObject:[NSArray arrayWithObjects:
         NSLocalizedString(@"Try Again", @""), NSLocalizedString(@"Cancel", @""),
         nil] forKey:NSLocalizedStringRecoveryOptionsErrorKey];
        [newDict setObject:url forKey:NSURLErrorKey];
        NSError *newError = [[NSError alloc] initWithDomain:[err domain]
                          code:[err code] userInfo:newDict];
        [self presentError:newError modalForWindow:[self windowForSheet]
         delegate:self
         didPresentSelector:@selector(didPresentErrorWithRecovery:contextInfo:)
         contextInfo:nil];
    }
}
// ...
```

Note that the document object also adds to the user info dictionary the URL identifying the source of XML. The recovery attempter will use this URL when it attempts to create a tree representing the XML.

The error object is passed up the error-responder chain and `NSApp` displays it. When the user clicks any button of the error alert, `NSApp` checks to see if the error object has both a recovery attempter and recovery options. If both of these conditions are true, it invokes the method implemented by the recovery attempter that corresponds to the mode of the alert (that is, document-modal or application-modal).

Listing 5-2 shows how the recovery attempter for the XML document implements the `attemptRecoveryFromError:optionIndex:delegate:didRecoverSelector:contextInfo:` method.

Listing 5-2 Recovering from the error and informing the modal delegate

```
- (void)attemptRecoveryFromError:(NSError *)error
```

```

        optionIndex:(unsigned int)recoveryOptionIndex
        delegate:(id)delegate
    didRecoverSelector:(SEL)didRecoverSelector
    contextInfo:(void *)contextInfo {

BOOL success = NO;
NSError *err = nil;
NSInvocation *invoke = [NSInvocation invocationWithMethodSignature:
    [delegate methodSignatureForSelector:didRecoverSelector]];
[invoke setSelector:didRecoverSelector];

if (recoveryOptionIndex == 0) { // Recovery requested.
    xmlDoc = [[NSXMLDocument alloc] initWithContentsOfURL:[error userInfo]
        objectForKey:NSURLErrorKey] options:NSXMLDocumentTidyXML error:&err];
    if (xmlDoc != nil) {
        success = YES;
    }
}
[invoke setArgument:(void *)&success atIndex:2];
if (err)
    [invoke setArgument:&err atIndex:3];
[invoke invokeWithTarget:delegate];
}

```

The key part of the above example is the test the recovery attempter makes to determine if the user clicked the “Try Again” button: it checks the value of `recoveryOptionIndex`. If the user did click this button, the recovery attempter invokes the `initWithContentsOfURL:options:error:` method again, this time with the `NSXMLDocumentTidyXML` option. Then it creates and invokes an `NSInvocation` object, thereby sending the required message to the modal delegate of the error alert. The invocation object includes the two parameters required by the delegate’s selector: a Boolean indicating whether the recovery attempt succeeded and a “context info” parameter which, in this case, contains any error object returned from the recovery attempt.

Note: The example in Listing 5-2 shows the use of `NSInvocation` to send a message. However, if you have a reference to the modal delegate and know the name of the method it implements, you can send the message directly.

When the modal delegate receives the message from the recovery attempter, as in Listing 5-3, it can respond appropriately.

Listing 5-3 Modal delegate responding to recovery attempter

```

- (void)didPresentErrorWithRecovery:(BOOL)didRecover
    contextInfo:(void *)contextInfo {

    NSError *theError = (NSError *)contextInfo;
    if (didRecover) {
        [tableView reloadData];
    } else if (theError && [theError isKindOfClass:[NSError class]]) {
        [UIAlert alertWithError:theError];
    }
}

```


Document Revision History

This table describes the changes to *Error Handling Programming Guide For Cocoa*.

Date	Notes
2009-10-16	Initialized local variables in code samples.
2009-08-18	Mentioned that NSError objects are available on iPhone OS, but noted that the error-responder/recovery APIs and architecture are specific to the Mac OS X platform. Also made some minor corrections and added links to core-competency concepts.
2009-03-04	Corrected a link error.
2009-01-06	Added link to "Error Handling in the Document Architecture" in Document-Based Application Overview. Mentioned default behavior of presentError: with NSCocoaErrorDomain/NSUserCancelledError errors. Provided related reference, sample code, and documents.
2006-10-03	Corrected code in Listing 5-2 showing creation of NSInvocation object.
2006-04-04	Corrected code listing illustrating error recovery and discussed NSUserCancelledError code.
2005-04-29	New document that describes how to use NSError objects and related Application Kit support when handling user-level errors.

REVISION HISTORY

Document Revision History

Index

A

`alertWithError:` method 13, 19
application object 18
application-modal alerts 18, 26, 33
`application:willPresentError:` method 24, 29–31
architectures
 for error handling 23
`attemptRecoveryFromError:optionIndex:` method 26, 33
`attemptRecoveryFromError:optionIndex:delegate:didRecoverSelector:contextInfo:` method 26, 33

B

button titles 13

C

Carbon error codes 9
Cocoa error domain 10, 16
`code` method 10
creating error objects 19, 20
customizing errors 25, 26, 30–31

D

displaying errors 17, 19
document-based applications
 and error presentation 24
document-modal alerts 18, 26, 33
`domain` method 9

E

error alerts 16, 26
 application-modal 18
 document-modal 18
error codes 10, 19, 26
error customization 23, 25–26
error description 12
error domains 9, 19, 26
 reasons for 10
error evaluation 15, 29
error objects *See* NSError objects 15
error presentation 13, 17–19
 architecture for 23, 25
error recovery 26–27, 33, 35
error-responder chain 16, 18, 23–25, 34
 sequence of objects 24
errors
 testing for 26
 underlying 26
`errorWithDomain:code:userInfo:` method 19
evaluating errors 16

F

failure reason 12, 26

H

handling returned errors 15

I

`initWithDomain:code:userInfo:` method 19

L

localizedDescription **method** 12
 localizedFailureReason **method** 12
 localizedRecoveryOptions **method** 13, 27
 localizedRecoverySuggestion **method** 13
 localizing error strings 12, 13

M

methods
 returning errors from 19, 20
 with NSError parameters 15
 modal delegate
 of document-modal alerts 18, 35

N

NSAlert class 13
 NSAlert objects
 and error presentation 17
 NSApp 18, 24, 29, 34
 NSApplication class 23
 NSCocoaErrorDomain **constant** 9
 NSDictionary class 12
 NSDocument class 23, 29
 NSDocument objects 18
 NSDocumentController class 23, 29
 NSDocumentController objects 18
 NSError class 9
 NSError objects 15
 and error recovery 33, 35
 and the error-responder chain 29, 31
 as method parameters 15
 as parameters 19
 creating 19–20
 customizing 26
 description 9–14
 displaying 17, 19
 evaluating 16
 examining 16, 29
 reasons for 9
 versus exceptions 20
 NSErrorRecoveryAttempting informal protocol 14, 26, 33
 NSException objects 20
 NSFilePathErrorKey **constant** 14
 NSInvocation class 35
 NSLocalizedRecoveryOptionsErrorKey **constant** 27
 NSLocalizedDescriptionKey **constant** 12
 NSLocalizedFailureReasonErrorKey **constant** 12

NSLocalizedRecoverSuggestionErrorKey **constant** 13
 NSLocalizedRecoveryOptionsErrorKey **constant** 13, 33
 NSLocalizedRecoverySuggestionErrorKey **constant** 33
 NSLocalizedString **macros** 13
 NSMachErrorDomain **constant** 9
 NSOSStatusErrorDomain **constant** 9
 NSPanel class 29
 NSPOSIXErrorDomain **constant** 9, 19
 NSRecoveryAttempterErrorKey **constant** 14, 27, 33
 NSResponder class 23
 NSStringEncodingErrorKey **constant** 14
 NSUnderlyingErrorKey **constant** 14, 26
 NSURLErrorDomain **constant** 10
 NSURLErrorKey **constant** 14
 NSUserCancelledError **code** 16, 26
 NSView class 23
 NSView objects 18
 NSWindow class 23, 29
 NSWindow objects 18
 NSWindowController class 23, 29
 NSWindowController objects 18
 NSXMLDocument class 33
 NSXMLParserErrorDomain **constant** 10

O

objectForKey: **method** 12
 OSStatus **data type** 9

P

POSIX error codes 9
 presentError: **method** 12, 13, 18, 23, 29
 presentError:modalForWindow:delegate:
 didPresentSelector:contextInfo: **method** 18,
 23, 29
 programming errors 20

R

received errors 29, 31
 recovering from errors 26, 27, 33, 35
 recovery attempter 14, 16, 18, 26, 27, 33–35
 recovery options 13, 33
 recovery suggestion 13, 33
 recoveryAttempter **method** 14, 27

runtime errors [21](#)

U

underlying errors [14](#), [20](#), [26](#)

 in Cocoa errors [16](#)

user info dictionary [12](#), [19](#)

 and error recover [33](#)

 domain-specific keys [14](#)

userInfo method [12](#)

V

view classes [29](#)

W

WebKitErrorDomain constant [10](#)

willPresentError: method [23](#), [29–31](#)