Exception Programming Topics

Developer

Contents

Figures, Tables, and Listings

Introduction to Exception Programming Topics for Cocoa

This document discusses how to raise and handle exceptions: special conditions that interrupt the normal flow of program execution. The Objective-C directives and Foundation API for exceptions are available on iOS and Mac OS X.

> **Important** You should reserve the use of exceptions for programming or unexpected runtime errors such as out-of-bounds collection access, attempts to mutate immutable objects, sending an invalid message, and losing the connection to the window server. You usually take care of these sorts of errors with exceptions when an application is being created rather than at runtime.
>
> If you have an existing body of code (such as third-party library) that uses exceptions to handle error conditions, you may use the code as-is in your Cocoa application. But you should ensure that any expected runtime exceptions do not escape from these subsystems and end up in the caller's code. For example, a parsing library might use exceptions internally to indicate problems and enable a quick exit from a parsing state that could be deeply recursive; however, you should take care to catch such exceptions at the top level of the library and translate them into an appropriate return code or state.

Instead of exceptions, error objects (`NSError`) and the Cocoa error-delivery mechanism are the recommended way to communicate expected errors in Cocoa applications. For further information, see *Error Handling Programming Guide*.

Organization of This Document

This document contains the following articles:

- "Exceptions and the Cocoa Frameworks" (page 6) describes `NSException` objects and their general use with the Cocoa frameworks.
- "Handling Exceptions" (page 7) describes how to handle an exception using the compiler directives `@try`, `@catch`, and `@finally` and the legacy macros `NS_DURING`, `NS_HANDLER`, and `NS_ENDHANDLER`.
- "Throwing Exceptions" (page 17) describes how to throw (raise) an exception.
- "Nesting Exception Handlers" (page 19) describes how exception handlers can be nested.
- "Predefined Exceptions" (page 24) describes where to find exceptions defined by Cocoa.
- "Uncaught Exceptions" (page 23) describes what happens to an exception not caught by an exception handler.

- "Controlling a Program's Response to Exceptions" (page 25) describes how to use the Exception Handling framework for monitoring and controlling the behavior of Cocoa programs in response to various types of exceptions.
- "Exceptions in 64-Bit Executables" (page 31) describes zero-cost `@try` blocks and C++ interoperability in 64-bit executables.

See Also

For information on originating, handling, and recovering from expected runtime errors, see *Error Handling Programming Guide* . Also see the related document,*Assertions and Logging Programming Guide* , for information on the Foundation framework's support for making assertions and logging error information.

Exceptions and the Cocoa Frameworks

Exceptions in Cocoa are represented by objects of the `NSException` class, which is part of the Foundation framework. The methods of this class allow you to create exception objects, raise (throw) exceptions with them, and get the call return addresses related to an exception. The attributes of an `NSException` object are the following:

- A `name` — a short string that is used to uniquely identify the exception. The name is required.
- A `reason` — a longer string that contains a "human-readable" reason for the exception. The reason is required.
- An optional dictionary (`userInfo`) used to supply application-specific data to the exception handler. For example, if the return value of a method causes an exception to be raised, you could pass the return value to the exception handler through `userInfo`.

You may extract the information in an exception object and, if appropriate, present to the user in an alert dialog, perhaps using an `NSError` object. See "Handling Exceptions" (page 7) for information on this subject.

The Cocoa frameworks require that all exceptions be instances of `NSException` or its subclasses. Do not throw objects of other types.

The Cocoa frameworks are generally not exception-safe. The general pattern is that exceptions are reserved for programmer error only, and the program catching such an exception should quit soon afterwards.

Handling Exceptions

The exception handling mechanisms available to Objective-C programs are effective ways of dealing with exceptional conditions. They decouple the detection and handling of these conditions and automate the propagation of the exception from the point of detection to the point of handling. As a result, your code can be much cleaner, easier to write correctly, and easier to maintain.

The following sections describe how to handle exceptions using compiler directives or, for appropriate projects, the legacy mechanism of exception-handling macros.

Important The Objective-C compiler directives discussed below were introduced in Mac OS X v10.3. An application that uses these directives for exception handling cannot run on earlier versions of the operating system.

Handling Exceptions Using Compiler Directives

Starting with version 3.3 of the GNU Compiler Collection (GCC), the compiler provides runtime support for exception handling. To turn on this support, make sure the `-fobj-exceptions` flag is turned on; this is enabled through the Enable Objective-C Exceptions build option in Xcode.

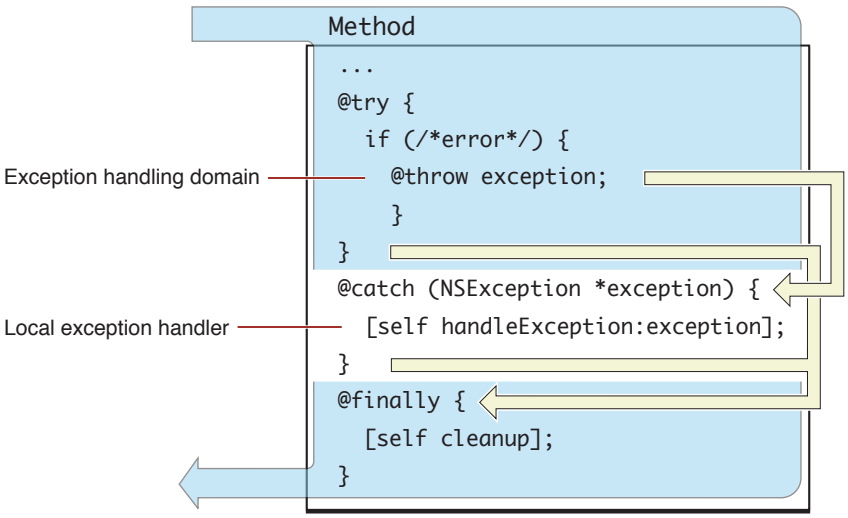
The compiler support for exceptions is based on four compiler directives:

- `@try` —Defines a block of code that is an exception handling domain: code that can potentially throw an exception.
- `@catch()` —Defines a block containing code for handling the exception thrown in the `@try` block. The parameter of `@catch` is the exception object thrown locally; this is usually an `NSException` object, but can be other types of objects, such as `NSString` objects.
- `@finally` — Defines a block of related code that is subsequently executed whether an exception is thrown or not.
- `@throw` — Throws an exception; this directive is almost identical in behavior to the `raise` method of `NSException`. You usually throw `NSException` objects, but are not limited to them. For more information about `@throw`, see "Throwing Exceptions" (page 17).

Important Although you can throw and catch objects other than `NSException` objects, the Cocoa frameworks themselves might only catch `NSException` objects for some conditions. So if you throw other types of objects, the Cocoa handlers for that exception might not run, with undefined results. (Conversely, non-`NSException` objects that you throw could be caught by some Cocoa handlers.) For these reasons, it is recommended that you throw `NSException` objects only, while being prepared to catch exception objects of all types..

The `@try`, `@catch`, and `@finally` directives constitute a control structure. The section of code between the braces in `@try` is the exception handling domain; the code in a `@catch` block is a local exception handler; the `@finally` block of code is a common "housekeeping" section. In Figure 1, the normal flow of program execution is marked by the gray arrow; the code within the local exception handler is executed only if an exception is thrown—either by the local exception handling domain or one further down the call sequence. The throwing (or raising) of an exception causes program control to jump to the first executable line of the local exception handler. After the exception is handled, control "falls through" to the `@finally` block; if no exception is thrown, control jumps from the `@try` block to the `@finally` block.

Figure 1 Flow of exception handling using compiler directives



Where and how an exception is handled depends on the context where the exception was raised (although most exceptions in most programs go uncaught until they reach the top-level handler installed by the shared `NSApplication` or `UIApplication` object). In general, an exception object is thrown (or raised) within the domain of an exception handler. Although you can throw an exception directly within a local exception handling domain, an exception is more likely thrown (through `@throw` or `raise`) indirectly from a method invoked from the domain. No matter how deep in a call sequence the exception is thrown, execution jumps to the local exception handler (assuming there are no intervening exception handlers, as discussed in "Nesting Exception Handlers" (page 19)). In this way, exceptions raised at a low level can be caught at a high level.

Listing 1 illustrates how you might use the `@try`, `@catch`, and `@finally` compiler directives. In this example, the `@catch` block handles any exception thrown lower in the calling sequence as a consequence of the `setValue:forKeyPath:` message by setting the affected property to `nil` instead. The message in the `@finally` block is sent whether an exception is thrown or not.

Listing 1 Handling an exception using compiler directives

```
- (void)endSheet:(NSWindow *)sheet
{
    BOOL success = [predicateEditorView commitEditing];
    if (success == YES) {

        @try {
            [treeController setValue:[predicateEditorView predicate]
forKeyPath:@"selection.predicate"];
        }

        @catch ( NSException *e ) {
            [treeController setValue:nil forKeyPath:@"selection.predicate"];
        }

        @finally {
            [NSApp endSheet:sheet];
        }
    }
}
```

One way to handle exceptions is to "promote" them to error messages that either inform users or request their intervention. You can convert an exception into an `NSError` object and then present the information in the error object to the user in an alert panel. In Mac OS X, you could also hand this object over to the Application Kit's error-handling mechanism for display to users. You can also return them indirectly in methods that include an error parameter. Listing 2 shows an example of the latter in an Automator action's implementation of `runWithInput:fromAction:error:` (in this case the error parameter is a pointer to an `NSDictionary` object rather than an `NSError` object).

Listing 2 Converting an exception into an error

```
- (id)runWithInput:(id)input fromAction:(AMAction *)anAction error:(NSDictionary
**)errorInfo {


    NSMutableArray *output = [NSMutableArray array];
    NSString *actionMessage = nil;
    NSArray *recipes = nil;
    NSArray *summaries = nil;


    // other code here....


    @try {
        if (managedObjectContext == nil) {
            actionMessage = @"accessing user recipe library";
            [self initCoreDataStack];
        }
        actionMessage = @"finding recipes";
        recipes = [self recipesMatchingSearchParameters];
        actionMessage = @"generating recipe summaries";
        summaries = [self summariesFromRecipes:recipes];
    }
    @catch (NSException *exception) {
        NSMutableDictionary *errorDict = [NSMutableDictionary dictionary];
        [errorDict setObject:[NSString stringWithFormat:@"Error %@: %@",
actionMessage, [exception reason]] forKey:OSAScriptErrorMessage];
        [errorDict setObject:[NSNumber numberWithInt:errOSAGeneralError]
forKey:OSAScriptErrorNumber];
        *errorInfo = errorDict;
        return input;
    }


    // other code here ....
}
```

> **Note** For more on the Application Kit's error-handling mechanisms, see *Error Handling Programming Guide*. To learn more about Automator actions, see *Automator Programming Guide*.

You can have a sequence of `@catch` error-handling blocks. Each block handles an exception object of a different type. You should order this sequence of `@catch` blocks from the most-specific to the least-specific type of exception object (the least specific type being `id`), as shown in Listing 3. This sequencing allows you to tailor the processing of exceptions as groups.

Listing 3 Sequence of exception handlers

```
@try {
    // code that throws an exception
    ...
}
@catch (CustomException *ce) { // most specific type
    // handle exception ce
    ...
}
@catch (NSException *ne) { // less specific type
    // do whatever recovery is necessary at his level
    ...
    // rethrow the exception so it's handled at a higher level
    @throw;
}
@catch (id ue) { // least specific type
    // code that handles this exception
    ...
}
@finally {
    // perform tasks necessary whether exception occurred or not
    ...
}
```

Note You cannot use the `setjmp` and `longjmp` functions if the jump entails crossing an `@try` block. Since the code that your program calls may have exception-handling domains within it, avoid using `setjmp` and `longjmp` in your application. However, you may use `goto` or `return` to exit an exception handling domain.

Exception Handling and Memory Management

Using the exception-handling directives of Objective-C can complicate memory management, but with a little common sense you can avoid the pitfalls. To see how, let's begin with the simple case: a method that, for the sake of efficiency, creates an object, uses it, and then releases it explicitly:

```
- (void)doSomething {
    NSMutableArray *anArray = [[NSMutableArray alloc] initWithCapacity:0];
    [self doSomethingElse:anArray];
    [anArray release];
}
```

The problem here is obvious: If the `doSomethingElse:` method throws an exception there is a memory leak. But the solution is equally obvious: Move the `release` to a `@finally` block:

```
- (void)doSomething {
    NSMutableArray *anArray = nil;
    array = [[NSMutableArray alloc] initWithCapacity:0];
    @try {
        [self doSomethingElse:anArray];
    }
    @finally {
        [anArray release];
    }
}
```

This pattern of using `@try...@finally` to release objects involved in an exception applies to other resources as well. If you have `malloc`'d blocks of memory or open file descriptors, `@finally` is a good place to free those; it's also the ideal place to unlock any locks you've acquired.

Another, more subtle memory-management problem is over-releasing an exception object when there are internal `autorelease` pools. Almost all `NSException` objects (and other types of exception objects) are created autoreleased, which assigns them to the nearest (in scope) autorelease pool. When that pool is released, the exception is destroyed. A pool can be either released directly or as a result of an autorelease pool further down the stack (and thus further out in scope) being popped (that is, released). Consider this method:

```
- (void)doSomething {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSMutableArray *anArray = [[[NSMutableArray alloc] initWithCapacity:0]
autorelease];
    [self doSomethingElse:anArray];
    [pool release];
}
```

This code appears to be sound; if the `doSomethingElse:` message results in a thrown exception, the local autorelease pool will be released when a lower (or outer) autorelease pool on the stack is popped. But there is a potential problem. As explained in "Throwing Exceptions" (page 17), a re-thrown exception causes its associated `@finally` block to be executed as an early side effect. If an outer autorelease pool is released in a `@finally` block, the local pool could be released *before* the exception is delivered, resulting in a "zombie" exception.

There are several ways to resolve this problem. The simplest is to refrain from releasing local autorelease pools in `@finally` blocks. Instead let a pop of a deeper pool take care of releasing the pool holding the exception object. However, if no deeper pool is ever popped as the exception propagates up the stack, the pools on the stack will leak memory; all objects in those pools remain unreleased until the thread is destroyed.

An alternative approach would be to catch any thrown exception, retain it, and rethrow it . Then, in the `@finally` block, release the autorelease pool and autorelease the exception object. Listing 4 shows how this might look in code.

Listing 4 Releasing an autorelease pool containing an exception object

```
- (void)doSomething {
    id savedException = nil;
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSMutableArray *anArray = [[[NSMutableArray alloc] initWithCapacity:0]
autorelease];
    @try {
        [self doSomethingElse:anArray];
```

```
    }
    @catch (NSException *theException) {
        savedException = [theException retain];
        @throw;
    }
    @finally {
        [pool release];
        [savedException autorelease];
    }
}
```

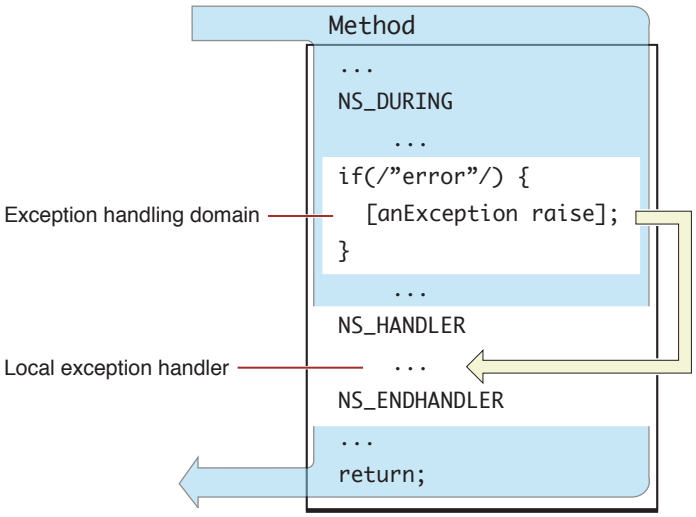
Doing this retains the thrown exception across the release of the interior autorelease pool—the pool the exception was put into on its way out of `doSomethingElse:`—and ensures that it is autoreleased in the next autorelease pool outward to it in scope (or, in another perspective, the autorelease pool below it on the stack). For things to work correctly, the release of the interior autorelease pool must occur before the retained exception object is autoreleased.

Handling Exceptions Using Macros

An exception handler is contained within a control structure created by the macros `NS_DURING`, `NS_HANDLER`, and `NS_ENDHANDLER`, as shown in Figure 2.

Important The exception macros are a legacy mechanism that should only be used when binary compatibility with versions of the operating system prior to Mac OS X v10.3 is a concern.

Figure 2 Flow of exception handling using macros



The section of code between `NS_DURING` and `NS_HANDLER` is the exception handling domain; the section between `NS_HANDLER` and `NS_ENDHANDLER` is the local exception handler. The normal flow of program execution is marked by the gray arrow; the code within the local exception handler is executed only if an exception is raised. Sending a `raise` message to an exception object causes program control to jump to the first executable line following `NS_HANDLER`.

Although you can raise an exception directly within the exception handling domain, `raise` is more often invoked indirectly from a method invoked from the domain. No matter how deep in a call sequence the exception is raised, execution jumps to the local exception handler (assuming there are no intervening exception handlers, as discussed in "Nesting Exception Handlers" (page 19)). In this way, exceptions raised at a low level can be caught at a high level.

For example, in the following program excerpt, the local exception handler displays an alert dialog after detecting an exception having the name MyAppException. The local exception handler has access to the raised exception object through a local variable `localException`.

```
NS_DURING
    ...
    if (someError)
        [anException raise];
    ...
```

```
NS_HANDLER
    if ([[localException name] isEqualToString:MyAppException]) {
       NSRunAlertPanel(@"Error Panel", @"%@", @"OK", nil, nil,
            localException);
    }
    [localException raise]; /* Re-raise the exception. */
NS_ENDHANDLER
```

You may leave the exception handling domain (the section of code between `NS_DURING` and `NS_HANDLER`) by:

- Raising an exception.
- Calling `NS_VALUERETURN()`
- Calling `NS_VOIDRETURN`
- "Falling off the end"

The above example raises an exception when `someError` is `YES`. Alternatively, you can return control to the caller from within the exception handling domain by calling either `NS_VALUERETURN()` or `NS_VOIDRETURN`. "Falling off the end" is simply the normal path of execution—after all statements in the exception handling domain are executed, execution continues on the line following `NS_ENDHANDLER`.

> **Note** You cannot use `goto` or `return` to exit an exception handling domain—errors will result. Nor can you use the `setjmp` and `longjmp` functions if the jump entails crossing an `NS_DURING` statement. Since the code that your program calls may have exception-handling domains within it, avoid using `setjmp` and `longjmp` in your application.

Similarly, you can leave the local exception handler (the section of code between `NS_HANDLER` and `NS_ENDHANDLER`) by raising an exception or simply "falling off the end".

Throwing Exceptions

Once your program detects an exception, it must propagate the exception to code that handles it. This code is called the exception handler. This entire process of propagating an exception is referred to as "throwing an exception" (or "raising an exception"). You throw (or raise) an exception by instantiating an `NSException` object and then doing one of two things with it:

- Using it as the argument of a `@throw` compiler directive
- Sending it a `raise` message

Important The `@throw` compiler directives was introduced in Mac OS X v10.3. An application that uses this directive for throwing exceptions cannot run on earlier versions of the operating system.

The following example shows how you throw an exception using the `@throw` directive (the `raise` alternative is commented out):

```
NSException* myException = [NSException
        exceptionWithName:@"FileNotFoundException"
        reason:@"File Not Found on System"
        userInfo:nil];
@throw myException;
// [myException raise]; /* equivalent to above directive */
```

An important difference between `@throw` and `raise` is that the latter can be sent only to an `NSException` object whereas `@throw` can take other types of objects as its argument (such as string objects). However, because higher-level handlers in the Application Kit might use the exception-handling macros—and thus can only deal with `NSException` objects—Cocoa applications should `@throw` only `NSException` objects.

Typically you throw or raise an exception inside an exception-handling domain, which is a block of code marked off by one of the two sets of Cocoa APIs intended for exception handling:

- The `NS_DURING` and `NS_HANDLER` macros (when developing for Mac OS X v10.2 and earlier).
- The block of code marked off by the `@try` compiler directive. (`@catch` and `@finally` are the other directives in this set.)

See "Handling Exceptions" (page 7) for details.

Within exception handling domains you can re-propagate exceptions caught by local exception handlers to higher-level handlers either by sending the `NSException` object another `raise` message or by using it with another `@throw` directive. Note that in `@catch` exception-handling blocks you can rethrow the exception without explicitly specifying the exception object, as in the following example:

```
@try {
    NSException *e = [NSException
        exceptionWithName:@"FileNotFoundException"
        reason:@"File Not Found on System"
        userInfo:nil];
    @throw e;
}
@catch(NSException *e) {
    @throw; // rethrows e implicitly
}
```

There is a subtle aspect of behavior involving re-thrown exceptions. The `@finally` block associated with the local `@catch` exception handler is executed before the `@throw` causes the next-higher exception handler to be invoked. In a sense, the `@finally` block is executed as an early side effect of the `@throw` statement. This behavior has implications for memory management (see "Exception Handling and Memory Management " (page 12)).

Nesting Exception Handlers

Exception handlers can be nested so that an exception raised in an inner domain can be treated by the local exception handler *and* any number of encompassing exception handlers. This design allows an exception to be handled by code that, although it is further from the code actually generating the exception, might have more knowledge about the conditions leading to the exception.

You can nest exception handlers using both the `@try...@catch...@finally` directives and the `NS_DURING...NS_HANDLER...NS_ENDHANDLER` macros. There are some subtle differences in these mechanisms between the flow of program control from inner exception handler to outer exception handler, so the following sections discuss them separately.

Important The compiler directives discussed below were introduced in Mac OS X v10.3. An application that uses these directives for exception handling cannot run on earlier versions of the operating system.

Nested Exception Handlers With Compiler Directives

To understand how nested exception handlers defined with the compiler directives are invoked, consider the code fragment in Listing 1.

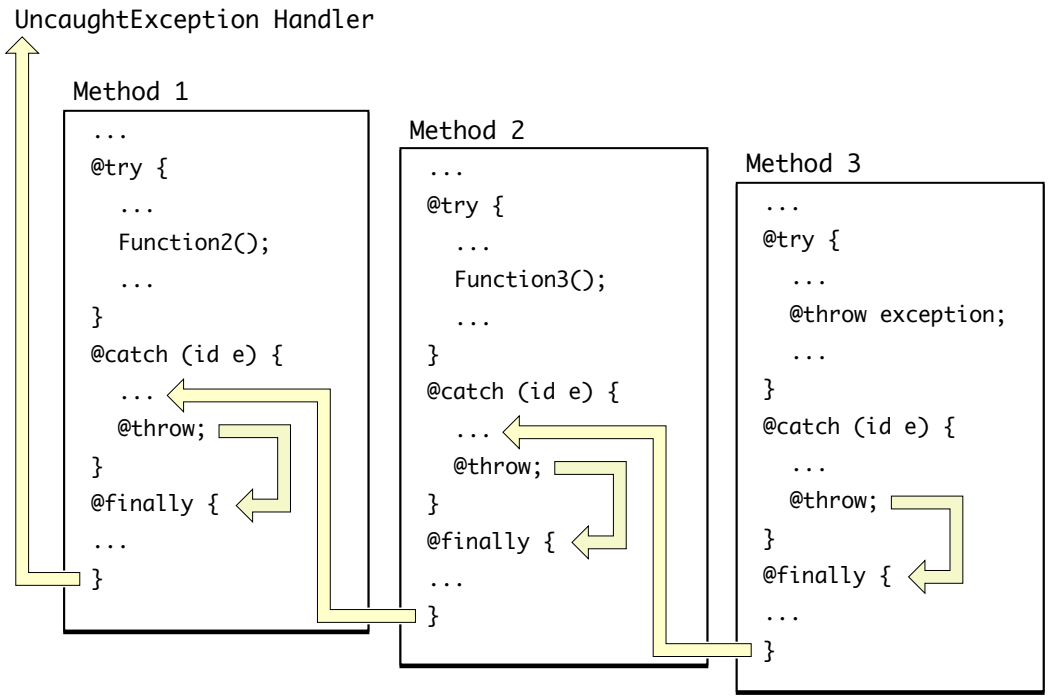
Listing 1 Throwing and re-throwing an exception

```
@try {
    // ...
    if (someError) {
        NSException *theException = [NSException exceptionWithName:MyAppException
 reason:@"Some error just occurred!" userInfo:nil];
        @throw theException;
    }
}
@catch (NSException *exception) {
    if ([[exception name] isEqualToString:MyAppException]) {
        NSRunAlertPanel(@"Error Panel", @"%@", @"OK", nil, nil,
                localException);
```

```
    }
    @throw; // rethrow the exception
}
@finally {
    [self cleanUp];
}
```

In this code the exception (`exception`) is thrown again at the end of the local handler, allowing an encompassing exception handler to take some additional action. Figure 1 illustrates the flow of program control between nested exception handlers created with the `@catch` directive.

Figure 1 Control flow with nested exception handlers—using directives



An exception raised within Method 3's domain causes execution to jump to its local exception handler. In a typical application, this exception handler queries the exception object to determine the nature of the exception. The local handler may handle exception types that it recognizes and then may rethrow the exception object to pass notification of the exception to the handler nested above it—that is, the handler in Method 2. However, before the next outer exception handler is invoked, the code in the `@finally` block associated with the local exception handler is executed. (This has implications for memory management, as discussed in "Exception Handling and Memory Management " (page 12).)

An exception that is re-thrown appears to the next higher handler just as if the initial exception had been raised within its own exception handling domain. Method 2's exception handler again may handle the exception and may rethrow the exception to Method 1's exception handler; Method 1's handler does not receive the re-thrown exception until Method 2's `@finally` block completes its task. Finally, Method 1's handler rethrows the exception. Because there is no exception handling domain above Method 1, the exception passes to the uncaught exception handler (see "Uncaught Exceptions" (page 23)).

Nested Exception Handlers With Exception Macros

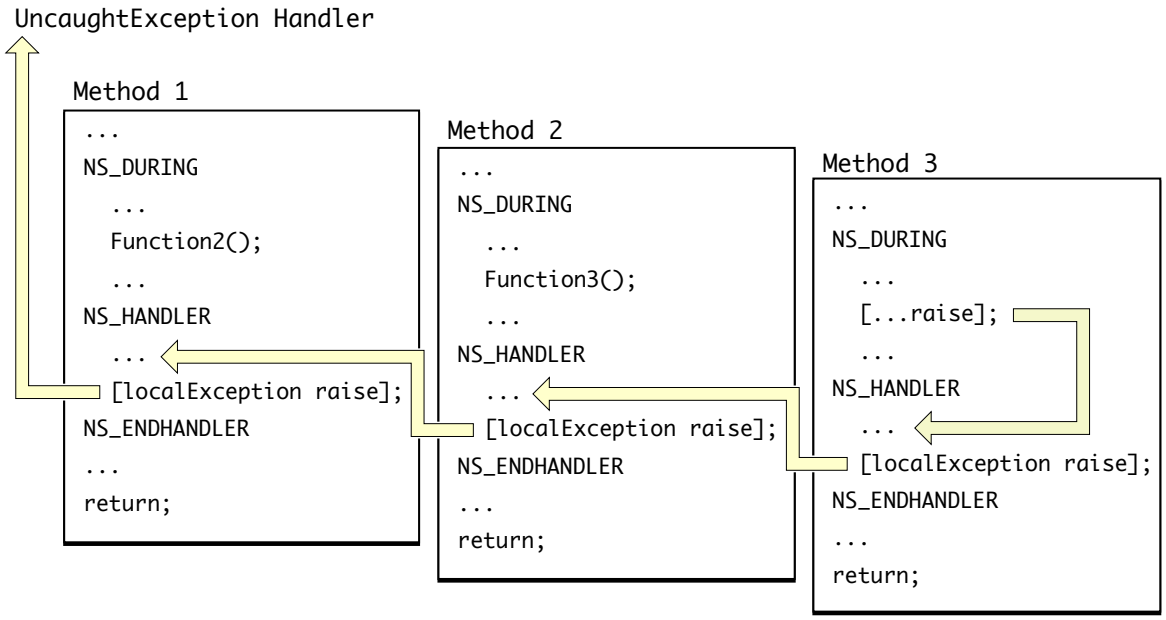
You program should use the exception-handling macros if it must be compatible with versions of the operating system prior to Mac OS X v10.3. If you are using the exception-handling macros, the code equivalent to that in Listing 1 (page 19) would look something like Listing 2.

Listing 2 Raising and re-raising an exception

```
NS_DURING
    ...
    if (someError) {
        NSException *theException = [NSException exceptionWithName:MyAppException
 reason:@"Some error just occurred!" userInfo:nil];
        [theException raise];
    }
    [self cleanUp];
NS_HANDLER
    if ([[localException name] isEqualToString:MyAppException]) {
        NSRunAlertPanel(@"Error Panel", @"%@", @"OK", nil, nil,
                localException);
    }
    [self cleanUp];
    [localException raise]; /* Re-raise the exception. */
NS_ENDHANDLER
```

In this code the exception (`exception` or `localException`) is raised again at the end of the local handler, allowing an encompassing exception handler to take some additional action. Figure 2 illustrates the use of nested exception handlers, and is discussed in the text that follows.

Figure 2 Control flow with nested exception handlers—using macros



An exception raised within Method 3's domain causes execution to jump to its local exception handler. In a typical application, this exception handler checks the object `localException` to determine the nature of the exception. For exception types that it recognizes, the local handler responds and then may send `raise` to `localException` to pass notification of the exception to the handler above, the handler in Method 2. (An exception that is re-raised appears to the next higher handler just as if the initial exception had been raised within its own exception handling domain.) Method 2's exception handler does the same and then re-raises the exception to Method 1's handler. Finally, Method 1's handler re-raises the exception. Since there is no exception handling domain above Method 1, the exception is transferred to the uncaught exception handler (see "Uncaught Exceptions" (page 23)).

Uncaught Exceptions

If an exception is not caught, it is intercepted by a function called the uncaught exception handler. The uncaught exception handler always causes the program to exit but may perform some task before this happens.

The default uncaught exception handler logs a message to the console before it exits the program. On Mac OS X, if the application was launched from the shell, the log messages are sent to the Terminal window.

You can set a custom function as the uncaught exception handler using the `NSSetUncaughtExceptionHandler` function; you can obtain the current uncaught exception handler with the `NSGetUncaughtExceptionHandler` function.

> **Note** Exceptions on the main thread of a Cocoa application do not typically rise to the level of the uncaught exception handler because the global application object catches all such exceptions.

Predefined Exceptions

Cocoa predefines a number of generic exception names to identify exceptions that you can handle in your own code or even raise and re-raise. You can also create and use custom exception names. The generic exception names are string constants defined in `NSException.h` and documented in *Foundation Constants Reference*. These constants include the following:

- `NSGenericException`
- `NSRangeException`
- `NSInvalidArgumentException`
- `NSInternalInconsistencyException`
- `NSObjectInaccessibleException`
- `NSObjectNotAvailableException`
- `NSDestinationInvalidException`
- `NSPortTimeoutException`
- `NSInvalidSendPortException`
- `NSInvalidReceivePortException`
- `NSPortSendException`
- `NSPortReceiveException`

In addition to the generic exception names, some subsystems of Cocoa define their own exception names, such as `NSInconsistentArchiveException` and `NSFileHandleOperationException`. These are also documented in *Foundation Constants Reference*.

You can identify caught exceptions in your exception handler by comparing the exception's name with these predefined names. Then you can either handle the exception or, if it isn't one you are interested in, re-raise it. Note that all predefined exceptions begin with the prefix "NS", so you should avoid using the same prefix when creating new exception names.

Controlling a Program's Response to Exceptions

This document describes some user defaults and the API of the Exception Handling framework that you can use to control the behavior of applications in response to certain types of errors.

To use the services of the Exception Handling framework in your Cocoa project (whether application or non-application), add `ExceptionHandling.framework` in `/System/Library/Frameworks` to your Xcode project. Also insert the following import directive in the header or implementation file of the class that uses the framework:

```
#import <ExceptionHandling/NSExceptionHandler.h>
```

Important The Exception Handing framework is not available on iOS.

The services described below are made possible through an uncaught exception handler set by the Exception Handling framework. These services won't be available if a custom uncaught exception handler is set through the `NSSetUncaughtExceptionHandler` function.

Application Errors

Certain types of application errors typically cause Cocoa applications to exit abruptly. You can use a user default, `NSExceptionHandlingMask`, to control this behavior (for Application Kit-based applications only) for three of the most common classes of such errors:

- uncaught NSExceptions;
- system-level exceptions (such as invalid memory accesses)
- Objective-C runtime errors (such as messages sent to freed objects).

For these error types you can set `NSExceptionHandlingMask` to do one of the following actions:

- Print a descriptive log and a stack trace to the console when such an error occurs.
- Handle the error in such a way as to prevent the resulting abrupt termination,
- Do both of the above.

You construct the mask by adding the values corresponding to the types of errors to be logged or handled:

Table 1 Exception-handling constants and `defaults` values

Type of Action	Constant	Value for `defaults`
Log uncaught NSExceptions	`NSLogUncaughtExceptionMask`	1
Handle uncaught NSExceptions	`NSHandleUncaughtExceptionMask`	2
Log system-level exceptions	`NSLogUncaughtSystemExceptionMask`	4
Handle system-level exceptions	`NSHandleUncaughtSystemExceptionMask`	8
Log runtime errors	`NSLogUncaughtRuntimeErrorMask`	16
Handle runtime errors	`NSHandleUncaughtRuntimeErrorMask`	32

Thus, if you enter the following on the command line (in the Terminal application):

```
defaults write NSGlobalDomain NSExceptionHandlingMask 63
```

you cause the logging and handling behavior described above for all uncaught exceptions, system-level exceptions, and runtime errors in all applications.

The word "handle" in the exception-handling constants has a specific meaning depending on the type of exception. The Exception Handling framework handles system-level exceptions and runtime errors by converting them into `NSException` objects. These exception objects contain a stack trace in their `userInfo`dictionary under the key `NSStackTraceKey`. The framework handles uncaught `NSException` objects by terminating the thread in which they occur. Exceptions on the main thread of a Cocoa application are caught by the top-level handlers, which are usually installed by the Application Kit.

Instead of the `NSExceptionHandlingMask` user default, you can use the `setExceptionHandlingMask:` method of the Exception Handling framework to get the same exception-handling behavior. For both application and non-application Cocoa executables, link against the Exception Handling framework and send the following message:

```
[[NSExceptionHandler defaultExceptionHandler] setExceptionHandlingMask: aMask]
```

The `aMask` parameter is a bit mask composed by bitwise-ORing the constants listed in the table above. See the header files of the Exception Handling framework for more details on the NSExceptionHandler API.

Debugging Aids

For debugging purposes, it is also possible to use the same mechanisms to report on NSExceptions that would otherwise be caught. You can also use either the `NSExceptionHandlingMask` property of the `defaults` system for this purpose or the `setExceptionHandlingMask:` method of the NSExceptionHandler class. The related constants and values are listed in the following table:

Table 2 Debugging constants and `defaults` values

Type of Action	Constant	Value for `defaults`
Log exceptions that would be caught by the top-level exception handlers in NSApplication. See note below.	`NSLogTopLevel-ExceptionMask`	64
Handle exceptions that would be caught by the top-level exception handlers in NSApplication	`NSHandleTopLevel-ExceptionMask`	128
Log exceptions that will be caught at lower levels	`NSLogOtherExceptionMask`	256
Handle exceptions that will be caught at lower levels	`NSHandleOther-ExceptionMask`	512

Note When exception-handling domains are nested, log exceptions that make it to the top two levels. On the main thread of a Cocoa application, this means log exceptions caught by NSApp.

In these cases, handling an exception means nothing more than adding a stack trace to its `userInfo` dictionary under the key `NSStackTraceKey`. Note that caught exceptions should be logged or handled only for debugging, not under normal circumstances, because doing so may generate large amounts of output, or alter the normal behavior of applications.

For further debugging purposes, you can change the handling behavior for any condition handled by NSExceptionHandler so that the application is instead halted so a debugger can be attached. You can control this behavior with summed values for the `NSExceptionHangingMask` user default or with the bit mask passed into the `setExceptionHangingMask:` of the NSExceptionHandler class. The following table lists the valid constants and `defaults` values:

Type of Action	Constant	Value for `defaults`
Hang for uncaught exceptions	`NSHangOnUncaughtExceptionMask`	1

Type of Action	Constant	Value for `defaults`
Hang for system-level exceptions	`NSHangOnUncaught-SystemExceptionMask`	2
Hang for runtime errors	`NSHangOnUncaughtRuntimeErrorMask`	4
Hang for top-level caught exceptions	`NSHangOnTopLevelExceptionMask`	8
Hang for other caught exceptions	`NSHangOnOtherExceptionMask`	16

Printing Symbolic Stack Traces

As a aid to debugging, you can use the `atos` command-line utility to convert numeric stack traces into symbolic form. (See the `atos(1)` man page for details of this command-line utility.)

> **Note** You must install the Developer Tools package to have the `atos` utility installed. Also, the `NSException` class provides the `callStackReturnAddresses`, which you can use for debugging in a manner similar to atos.

Instead of switching between Xcode an a Terminal shell, you can add code to your program that uses `atos` to print a symbolic stack trace to the console. Listing 1 shows how you do this. The method `printStackTrace:` extracts the (numeric) stack trace from the passed-in `NSException` object and then constructs an `NSTask` object that represents the `atos` command with the stack trace as a parameter. It launches the subtask and the resulting symbolic backtrace is printed to standard output (which is the run log in Xcode).

Listing 1 A method that prints a symbolic back trace of an exception

```
- (BOOL)exceptionHandler:(NSExceptionHandler *)sender shouldLogException:(NSException
 *)exception mask:(unsigned int)mask
{
    [self printStackTrace:exception];
    return YES;
}

- (void)printStackTrace:(NSException *)e
{
```

```
    NSString *stack = [[e userInfo] objectForKey:NSStackTraceKey];
    if (stack) {
        NSTask *ls = [[NSTask alloc] init];
        NSString *pid = [[NSNumber numberWithInt:[[NSProcessInfo processInfo]
processIdentifier]] stringValue];
        NSMutableArray *args = [NSMutableArray arrayWithCapacity:20];

        [args addObject:@"-p"];
        [args addObject:pid];
        [args addObjectsFromArray:[stack componentsSeparatedByString:@"  "]];
        // Note: function addresses are separated by double spaces, not a single
space.

        [ls setLaunchPath:@"/usr/bin/atos"];
        [ls setArguments:args];
        [ls launch];
        [ls release];

    } else {
        NSLog(@"No stack trace available.");
    }
}
```

In this example, the delegate invokes the `printStackTrace:` method in its implementation of `exceptionHandler:shouldLogException:mask:`; at this point, the exception is being handled, but has not yet caused termination of the debugged executable. The output of the `atos` utility, when combined with the `NSExceptionHandler` log information, looks similar to Listing 2.

Listing 2 Content of NSExceptionHandler log plus `atos` output

```
2006-08-21 12:18:19.727 ExceptionHandleTest[916] NSExceptionHandler has recorded
the following exception:
NSInvalidArgumentException -- *** -[NSCFString count]: selector not recognized
[self = 0x2a00c]
Stack trace:  0x9275c27b  0x92782fd7  0x9280b0be  0x9272f207  0x90a51ba1  0x0002995f
  0x00023f81  0x00001ca6  0x00001bcd  0x00000001
__NSRaiseError (in Foundation)
+[NSException raise:format:] (in Foundation)
```

```
-[NSObject doesNotRecognizeSelector:] (in Foundation)
-[NSObject(NSForwardInvocation) forward::] (in Foundation)
__objc_msgForward (in libobjc.A.dylib)
-[ExceptionTest testException] (in ExceptionHandleTest) (ExceptionTest.m:31)
_main (in ExceptionHandleTest) (ExceptionHandleTest.m:10)
start (in ExceptionHandleTest)
start (in ExceptionHandleTest)
0x00000001 (in ExceptionHandleTest)
```

There are other ways of accomplishing the same result. For example, the method that prints the symbolic stack trace could be on a category added to `NSException` instead of a method of the delegate's class.

Exceptions in 64-Bit Executables

The Objective-C runtime has reimplemented the exception mechanism for 64-bit executables to provide zero-cost `@try` blocks and interoperability with C++ exceptions.

Zero-Cost @try Blocks

64-bit processes that enter a zero-cost `@try` block incur no performance penalty. This is unlike the mechanism for 32-bit processes, which calls `setjmp()` and performs additional "bookkeeping". However, throwing an exception is much more expensive in 64-bit executables. For best performance in 64-bit, you should throw exceptions only when absolutely necessary.

C++ Interoperability

In 64-bit processes, Objective-C exceptions (`NSException`) and C++ exception are interoperable. Specifically, C++ destructors and Objective-C `@finally` blocks are honored when the exception mechanism unwinds an exception. In addition, default catch clauses—that is, `catch(...)` and `@catch(...)`—can catch and rethrow any exception

On the other hand, an Objective-C catch clause taking a dynamically typed exception object (`@catch(id exception)`) can catch any Objective-C exception, but cannot catch any C++ exceptions. So, for interoperability, use `@catch(...)` to catch every exception and `@throw;` to rethrow caught exceptions. In 32-bit, `@catch(...)` has the same effect as `@catch(id exception)`.

Document Revision History

This table describes the changes to *Exception Programming Topics* .

Date	Notes
2010-02-24	Removed misleading sentence from "Exceptions and the Cocoa Frameworks."
2009-07-28	Updated for iOS and added links to Cocoa Core Competencies. Added information on zero-cost @try blocks and C++ interoperability. Added "Exceptions and the Cocoa Frameworks" chapter.
2007-10-02	Made several minor corrections.
2007-01-08	Corrected typo.
2006-10-03	Added information on @try, @catch, @finally, and @throw and made various minor corrections.
2005-10-04	Fixed various bugs. Changed title from "Exceptions."
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.

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