Assertions and Logging Programming Guide

Data Management: Event Handling

2006-04-04

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

Simultaneously published in the United States and Canada.

Contents

Introduction to Assertions and Logging

Assertions are mechanisms that print error messages and raise exceptions when a given condition in code is false. Logging prints error or informational messages, typically to the standard error device.

Organization of This Document

This programming topic covers exception handling insofar as it relates to evaluations of given conditions and the logging of messages. For more sophisticated exception handling, namely techniques involving the raising (or throwing) and handling of exception objects, see *Exception Programming Topics*.

The articles covered in this programming topic are:

How Assertions Work

In your code you make an assertion using an assertion macro. These macros evaluate a condition and, if the condition evaluates to false, they pass a string (and possibly additional `printf`-style arguments formatted into the string) describing the failure to their NSAssertionHandler. Each thread has its own NSAssertionHandler object created for it. When invoked with an assertion, an NSAssertionHandler prints an error message that includes the method and class (or function) containing the assertion and then it raises an `NSInternalInconsistencyException`.

Using the Assertion Macros

This document describes how to use the `Assert` (and related) macros to evaluate a condition and create an assertion.

You use an assortment of macros to evaluate a condition—these macros serve as a front end to NSAssertionHandler. These macros fall into two categories: those you use in Objective-C methods, and those you use in C functions. For example, `NSAssert` is for use within methods and `NSCAssert` is for use within functions. Each macro has two arguments: the condition—an expression that evaluates to true or false—and the NSString describing the failure. Other macros are available if one or more arguments are needed for a `printf`-style description. For example, `NSAssert1` is used within methods if one argument is needed as in:

```
NSAssert1((0 <= component) && (component <= 255),
        @"Value %i out of range!", component);
```

For more details on these macros see `NSAssert`.

You create assertions only using the above macros—you rarely need to invoke NSAssertionHandler methods directly. The macros for use inside methods and functions send `handleFailureInMethod:object:file:lineNumber:description:` and `handleFailureInFunction:file:lineNumber:description:` messages respectively to the current assertion handler. The assertion handler for the current thread is obtained using the NSAssertionHandler `currentHandler` class method.

Assertions are not compiled into code if the preprocessor macro `NS_BLOCK_ASSERTIONS` is defined.

Logging Messages

You use the `NSLog` and `NSLogv` functions to log error and informational messages. These messages are written to `stderr`.

The message consists of a timestamp and the process ID prefixed to the string you pass in. You compose this string with a format string and one or more arguments to be inserted into the string. The format specification allowed by these functions is that which is understood by NSString's formatting capabilities (which is not necessarily the set of format escapes and flags understood by `printf`). For example, the following code fragment, outputs a string constructed from NSString and `int` arguments.

```
int recNum;
NSString *recName;
/* ... */
NSLog( @"Record %d is %@", recNum, recName );
```

See "Formatting String Objects" for a description of the various format specifiers.

In general, you should use the `NSLog` function instead of calling `NSLogv` directly. If you do call `NSLogv` directly, you must have prepared the required variable argument list by calling the standard C macro `va_start`. Upon completion, you must similarly call the standard C macro `va_end` for this list.

Using a Custom Assertion Handler

In some cases, you might want to define your own assertion handler to print error messages to a different error console or to raise custom exceptions, instead of the generic `NSInternalInconsistencyException`. To implement these features, you must define a subclass of NSAssertionHandler and override its `handleFailureInMethod:object:file:lineNumber:description:` and `handleFailureInFunction:file:lineNumber:description:` methods to handle assertions in methods and functions, respectively.

To add your assertion handler to a thread, you must add the assertion handler to the thread's attributes dictionary. Use the current NSThread's `threadDictionary` method to retrieve the dictionary. Add your assertion-handler object to the dictionary using the key `NSAssertionHandler`. This technique is used to specify a custom assertion handler on any thread, including the main thread. You must execute these steps in the thread which you wish to modify; one thread cannot modify the thread attributes dictionary of another.

Typically, you should add your assertion handler to the thread dictionary immediately after creating the thread. However, a default assertion handler is not created until an assertion macro is encountered and you can always replace the existing assertion handler in the thread dictionary. If your assertion handler already exists in the thread dictionary, it is used in place of the default assertion handler.

Document Revision History

This table describes the changes to *Assertions and Logging Programming Guide*.

Date	Notes
2006-04-04	Clarified that output of NSLog is to stderr.
2003-07-01	Clarified that the information provided in "Using a Custom Assertion Handler" (page 13) also applies to the main thread. Added that you must modify the dictionary in the same thread.
2003-06-03	"Logging Messages" (page 11) now includes a link to the article describing format specifiers.
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.