Older Release Notes
Cocoa Foundation Framework

The Foundation Framework is a library of Objective-C classes that provide the infrastructure for object-based applications without graphical user interfaces as well as for most other Cocoa frameworks, most notably the Application Kit.

This document describes the changes in Foundation framework in macOS 10.12/iOS 10, followed by earlier release notes. Always refer to the current releases notes first as the more recent changes could have obsoleted some of the earlier notes discussed further below.

- Notes specific to macOS 10.12 Sierra and iOS 10
- Notes specific to OS X 10.11 El Capitan and iOS 9
- Notes specific to OS X 10.10 Yosemite and iOS 8
- Notes specific to OS X 10.9 Mavericks and iOS 7
- Notes specific to OS X 10.8 Mountain Lion and iOS 6
- Notes specific to Mac OS X 10.7 Lion
- Notes specific to Mac OS X 10.6 Snow Leopard
- Notes specific to Mac OS X 10.5 Leopard
- Notes specific to Mac OS X 10.4 Tiger
- Notes specific to Mac OS X 10.3 Panther
- Notes specific to Mac OS X 10.2 Jaguar
- Notes specific to Mac OS X 10.1 Puma

Notes specific to macOS 10.12 and iOS 10

Overall API Updates

This release brings many continued API refinements to Foundation, taking advantage of new API features in Objective-C and enabling APIs to come across more naturally in Swift.

Some of the key API update areas include:
- Adoption of Swift 3 API design guidelines. Please refer to Swift Evolution document SE-0023, "API Design Guidelines."
- Value Types, where many Foundation classes are exposed as structs with value type semantics for Swift. This is described in Swift Evolution document SE-0069, "Mutability and Foundation Value Types."
- Introduction and adoption of string enumerations, with NS_STRING_ENUM and NS_EXTENSIBLE_STRING_ENUM.
- Introduction and adoption of NS_NOESCAPE to mark blocks whose execution completes before the API the block is passed in as an argument to.
- Dropping of "NS" prefix on certain Foundation classes and types, in both Swift and Objective-C.
- Continued fixes to nullability of values.
- Use of class properties, in both Swift as well as Objective-C, latter using the new "@property (class)" declaration.

The above topics and more were covered during the following sessions at WWDC 2016, which you can watch on videos at Apple's Developer Site:
- Swift API Design Guidelines
- What's New in Cocoa
- What's New in Foundation for Swift

API Availability Macros

Foundation has switched to new availability macros for ObjC that enable much more flexibility in identifying each platform independently. These new macros are API_AVAILABLE, API_DEPRECATED, API_DEPRECATED_WITH_REPLACEMENT, and API_UNAVAILABLE. Sample use cases:

```
FOUNDATION_EXPORT NSURLResourceKey const NSURLCanonicalPathKey API_AVAILABLE(macosx(10.12), ios(10.0), watchos(3.0), tvos(10.0
```

```
@property BOOL usesStrongWriteBarrier API_DEPRECATED("Garbage collection no longer supported",
                                                     macosx(10.5, 10.12), ios(2.0, 10.0), watchos(2.0, 3.0), tvos(9.0, 10
```

```
static const NSWindowStyleMask NSResizableWindowMask API_DEPRECATED_WITH_REPLACEMENT("NSWindowStyleMaskResizable",
                                                       macosx(10.0, 10.12)) = NSWindowStyleMaskResizable;
```

Measurements and Units

This release brings a number of new APIs to Foundation that enable representing measured amounts. The primary APIs are:

NSUnit - The abstract superclass of units, such as miles, km/h, degrees Fahrenheit.

NSDimension - Subclass of NSUnit representing unit families, such as length, speed, temperature. This class has many subclasses such as NSUnitLength, representing the different unit families.

NSUnitConverter - Helper class to make it easy to convert between units in a unit value.

NSMeasurement - A unit and a value, representing a measured quantity. For instance 10 miles, 42 km/h, 72 degrees Fahrenheit.

NSMeasurementFormatter - A new formatter to display measurements in localized fashion.

NSISO8601DateFormatter

A new formatter to format and parse ISO8601 dates. By default uses RFC3339, but can be configured by setting various options.

NSDateInterval

A new class to represent a date interval. Used by NSDateIntervalFormatter.

NSPersonNameComponentsFormatter

Introduced in the previous release for formatting person names, this formatter can now also parse them.

NSKeyedArchiver

New property encodedData, which returns the data by first invoking finishEncoding if needed to wrap-up the encoding. With the addition of this property, NSKeyedArchiver can also now be created without supplying a data, with just -init, which makes it more convenient to use.

NSKeyedUnarchiver

Ensure that the array of classes provided to -decodeObjectOfClasses:key: is actually an array of classes

Fix to ensure that we preserve the original exception stack trace during object decode.

Various performance improvements during decode. Also, numerous leak fixes during error/exception throws.

We now enforce the set-once-to-YES convention of setRequiresSecureCoding.

NSKeyedUnarchiver no longer uses mmap for the data in + (id)unarchiveObjectWithFile:(NSString *)path.

NSKeyedUnarchiverDelegate

Note in Swift/ARC it is unsafe to return nil from -unarchiver:didDecodeObject: when the decoded object is non-nil.

Use of strong, copy, weak, or assign on readonly object properties

Although at first glance it may seem unnecessary, Cocoa APIs decorate readonly properties with one of strong, copy, weak, or assign. For one thing, the attribute indicates how the receiver object holds on to the value, for instance when the receiver has an init method that takes the properties, and readonly getters. In addition, these provide additional info on the return value:

- copy: either the receiver has copied the incoming value, or effectively gives you a new value each time
- strong: the value is effectively a singleton, you will get back the same value each time
- weak: this one isn't too applicable, but effectively a "singleton" that is cached, so may go away (for instance, NSFont will do this for named font requests)

NSURL

NSURL has a number of new properties:

NSURLCanonicalPathKey - URL's path as a canonical absolute file system path (Read-only, value type NSString)
NSURLVolumeIsEncryptedKey - true if the volume is encrypted. (Read-only, value type boolean NSNumber)
NSURLVolumeIsRootFileSystemKey - true if the volume is the root filesystem. (Read-only, value type boolean NSNumber)
NSURLVolumeSupportsCompressionKey - true if the volume supports transparent decompression of compressed files using decmpfs. (Read-only, value type boolean NSNumber)
NSURLVolumeSupportsFileCloningKey - true if the volume supports clonefile(2) (Read-only, value type boolean NSNumber)
NSURLVolumeSupportsSwapRenamingKey - true if the volume supports renamex_np(2)'s RENAME_SWAP option (Read-only, value type boolean NSNumber)
NSURLVolumeSupportsExclusiveRenamingKey - true if the volume supports renamex_np(2)'s RENAME_EXCL option (Read-only, value type boolean NSNumber)

The NSURLFileResourceIdentifierKey, NSURLVolumeIdentifierKey, and NSURLGenerationIdentifierKey file system properties are now correctly documented to conform to NSSecureCoding.

The NSURLAttributeModificationDateKey file system property is now correctly documented to be read-only.

Relative file URLs with a file reference baseURL are now handled correctly.

NSURL objects are immutable and must have a URL string. So, [NSURL init] makes no sense because there is no URL string supplied. In past versions of the OS, [NSURL init] released self and returned nil. However, to keep [NSURL init] from failing, [NSURL init] will now initialize the NSURL object with an empty string. This behavior change (obviously) also affects [NSURL new].

The NSURL pathExtension and URLByDeletingPathExtension properties, and the -[NSURL(NSURLPathUtilities) URLByAppendingPathExtension:] method now determine valid path extensions using the same logic as the rest of the OS. This may lead to small changes in behavior. A valid extension cannot contain the space character (\u0020), the forward slash character (\u002F), or Unicode characters which change the direction of the text. A path component ending with a period does not have an extension. A path component starting with a period and containing no other periods does not have an extension.

NSURLComponents

The NSURLComponents path and percentEncodedPath properties now return an empty path instead of nil.

NSURLSessionTaskMetrics

NSURLSessionTaskMetrics and related APIs provide performance information about the loading of network resources.

NSPathUtilities

The NSPathUtilities pathExtension and stringByDeletingPathExtension properties, and the -[NSString(NSStringPathExtensions) stringByAppendingPathExtension:] method now determine valid path extensions using the same logic as the rest of the OS. This may lead to small changes in

behavior. A valid extension cannot contain the space character (\u0020), the forward slash character (\u002F), or Unicode characters which change the direction of the text. A path component ending with a period does not have an extension. A path component starting with a period and containing no other periods does not have an extension.

Property List Serialization

Now properly supports round tripping of +/- INF values.

Fixes -xml / -json decode to allow dot-escaping. In particular you should again be able to mutate deeply nested json/xml plists via plutil again.

Various crash fixes.

Deprecate GC NSPointerFunctions

The 'usesStrongWriteBarrier' and 'usesWeakReadAndWriteBarriers' properties have been deprecated.

NSLog / CFLog

NSLog is now just a shim to os_log in most circumstances.

Array / Dictionary / OrderedSet / Set

Fix nullability of the C-array based designated initializers for all collections.

NSArray and NSDictionary now have high performance special-cased one-element instances.

Special case empty ranges in subarrayWithRange.

NSMutableArray

Performance improvements for creation and mutable copies, especially for internally contiguous arrays.

NSDictionary

Ensure that the sharedKey based dictionaries have reasonably performant -hash functions. Also, make sure that the sharedKey implementation works well on 64-bit platforms. Also make sharedKey based dictionaries fully support NSSecureCoding.

NSMutableDictionary

Detect and HALT on degenerate cases where we would otherwise infinite-loop during rehash. This should be very rare, we've only seen evidence of it happening once.

Fixed a few potential crashes.

NSFastEnumeration

Should once again be possible to fast-enumerate a given collection on multiple threads at the same time (please don't mutate on multiple threads though, there lies pain!)

NSCache

When deallocating an NSCache, we no longer pass a moribund object to delegate -willEvictObject: call outs during -dealloc. Instead we now send out a special sentinel NSCache instance (to keep in spirit of nullability of the delegate).

NSRunLoop / CFRunLoop

Performance optimization for runloop-less threads when checking if a given run-loop is current. In previous releases we'd create a RunLoop if it hadn't been before just to check if it was current. We now treat the non-existence of a run-loop as != without needing to create it.

CFSocket

Removed SOCK_SEQPACKET; Darwin platforms have never supported it.

CFPasteboard

Various changes to support new continuity features, including new pasteboard subsystem: com.apple.CFPasteboard that can be useful to explore when debugging pasteboard related issues.

Localized String Formatting

Localized formatting of strings (localizedFormatWithString: and variants, as well as initWithFormat:locale: and CFStringCreateWithFormat() when a non-nil locale argument is provided) now surround %@ arguments with the Unicode isolate formatting characters FSI and PDI, if the resulting string contains any strong right-to-left characters. This addresses number of issues in the layout and display of text with mixed directionality.

This happens only for applications linked against the macOS 10.12 / iOS 10 SDKs.

NSCharacterSet / NSMutableCharacterSet

They adopt NSSecureCoding.

NSDateInterval

New API to create a date interval object.

NSDateIntervalFormatter

New method that takes in an NSDateInterval object.

NSDateComponentsFormatter

If allowedUnits is not set, it will default to NSCalendarUnitYear | NSCalendarUnitMonth | NSCalendarUnitWeekOfMonth | NSCalendarUnitDay | NSCalendarUnitHour | NSCalendarUnitMinute | NSCalendarUnitSecond.

Fix related to handling negative values. This affects stringFromTimeInterval: and stringFromDateComponents:.

NSDateComponentsFormatterUnitsStyleShort is now accurately documented in the header.

Added new enum value NSDateComponentsFormatterUnitsStyleBrief.

CoreFoundation <-> Foundation Bridging Performance

Many types in Foundation and CF are bridged such that they have a C interface exposed by CF, and an Objective-C class interface exposed by Foundation. In previous releases of OS X and iOS, which of these has better performance has been somewhat unintuitive, and has varied over the years. In Mac OS X 10.12 and iOS 10.0, we've simplified this: the recommended API (Foundation) is also the fastest API in nearly all cases now.

In particular, places to re-evaluate performance tradeoffs in your code include:

- Creating arrays and dictionaries of Objective-C objects is significantly faster, especially for objects that don't override the -retain and -release methods
- The speed of -retain, -release, -isEqual:, and -hash on CF types have improved to the point of actually being faster than the corresponding CF functions in most cases.
- Using NSArray/NSMutableArray methods to operate on objects created via CFArrayCreate* on iOS is significantly faster now.
- The performance penalty for passing Objective-C objects to CF functions has been significantly reduced, and in some cases eliminated
- Objective-C collection literals now correctly special-case small and empty collections as the regular allocation methods do
- NSMutableDictionary's memory footprint and performance have been improved vs CFMutableDictionaryRef
- NSDictionary's memory usage was already significantly better than CFDictionaryRef's, but it's now further improved

Additionally, most methods and functions on the core collection and value types in both CF and Foundation are faster in general (rather than relative to each other), in some cases significantly so. As always, using performance tools such as Instruments is essential to making informed performance-related decisions about your code.

NSUserDefaults detects folder deletion

Deleting a folder of preferences (say, the container of an application you're testing), should now correctly update the defaults system, rather than leaving it in an invalid and difficult to recover from state.

NSXMLParser fix with compatibility implications

When running in applications built using the 10.12 SDK, NSXMLParser will no longer artificially extend the lifetime of the 'attributes' dictionary passed to the -parser:didStartElement:namespaceURI:qualifiedName:attributes: method. This change reduces memory footprint significantly, but may cause incorrectly written applications to crash after switching to the new SDK and rebuilding. If you see a crash like this, make sure you're keeping a strong reference to the attributes dictionary if you need to use it after the method returns.

Key-Value Observing is less sensitive to certain combinations of arguments

In Mac OS X 10.11.0, And to a lesser extent in 10.6-10.10, certain combinations of arguments to the -addObserver:forKeyPath:withOptions:context: method could cause unusually bad performance. Many of these cases were fixed in 10.11.3, and all known remaining ones are fixed in 10.12. If you found that you needed to avoid NSKeyValueObservingOptionInitial, or avoid using particular context parameters, you can revisit those decisions.

Key-Value Observing and NSUserDefaults

In previous releases, KVO could only be used on the instance of NSUserDefaults returned by the +standardUserDefaults method. Additionally, changes from other processes (such as defaults(1), extensions, or other apps in an app group) were ignored by KVO. These limitations have both been corrected. Changes from other processes will be delivered asynchronously on the main queue, and ignore NSKeyValueObservingOptionPrior.

Key-Value Coding and weak properties

KVC now supports accessing weak object-typed properties correctly, as well as being somewhat faster.

-[NSUserDefaults registerDefaults:] and NSURLs

Calling -registerDefaults: with a dictionary containing NSURLs now encodes them the same way as -setURL:forKey:, so they'll work properly with -URLForKey:.

-[NSLocale calendar] bug fix

The NSCalendar returned by -[NSLocale calendar] is now correctly copy-on-write and thread-safe, like NSCalendar instances created directly.

Notes specific to OS X 10.11 and iOS 9

Swiftification

Framework APIs in both OS X 10.11 and iOS 9 have adopted new Objective-C features such as nullability, lightweight generics, and “kindof.” These features enable API intent to be expressed more precisely and APIs to come across better in Swift.

For instance we can now indicate that nil is a valid value for the destination view when converting an NSRect between coordinate systems:

```
- (NSRect)convertRect:(NSRect)rect toView:(nullable NSView *)view;
```

and that the array returned by splitting an NSString with componentsSeparatedByString: is an NSArray of NSStrings, and it will never be nil:

```
- (NSArray<NSString *> *)componentsSeparatedByString:(NSString *)separator;
```

You will notice many API changes in Foundation and other frameworks due the adoption of these new features. When building your projects against the new SDKs you may get build warnings or errors; these are important to react to since in some cases they may point out actual issues. In addition, adopting these new features in your own code should allow better compile time detection of bugs.

Please see WWDC 2015 Session 202, “What’s New in Cocoa,” for more info on this “Swiftification” effort, as well as other changes in Foundation as well as AppKit.

Variable Width Presentation Strings

Framework level support for variable width strings. To exemplify the problem, imagine an iOS app meant to run on different screen sizes and/or different orientations. Now, say there was some text, “Welcome to the Apple Store.” At times, that text may fit perfectly well in the available space. For others, it may be too long and it’d be desirable to shorten it to something like “Welcome."

Our solution is to allow developers/localizers to provide different width-variations for a string in a stringsdict file, the same file meant for different cardinalities (“1 file is selected” vs. “2 files are selected”). The -[NSBundle localizedStringForKey:value:table:] method reads the stringsdict file and creates an NSString that knows about the different variations.

We have a single method, variantFittingPresentationWidth: on NSString. It selects the appropriate variation for the given width. The width is just an integer identifying one of the variations. The strings are sorted such that if the width does not exist, the string at the next smaller width is returned or the smallest available if none are smaller. The application can decide on their own contexts, but iOS uses em’s; the number of ‘M' characters for the current system-font that will fit the width of the window.

After obtaining a string from variantFittingPresentationWidth:, the application can invoke the method again on the returned string object for a different width. It effectively “preserves the magic.”

NSString API exposed in NSBundle:

```
- (NSString *)variantFittingPresentationWidth:(NSInteger)width;
```

For strings with width variations, such as from a stringsdict file, this method returns the variant at the given width. If there is no variant at the given width, the one for the next smaller width is returned. And if there are none smaller, the smallest available is returned. For strings without variations, this method returns self. The unit that width is expressed in and what width is a measurement of is decided by the application or framework. But it is intended to be some measurement indicative of the context a string would fit best to avoid truncation and wasted space.

The format of the stringsdict file looks like this. We’re also introducing NSStringVariableWidthRuleType.

…

```
<key>Welcome</key>
<dict>
    <key>NSStringVariableWidthRuleType</key>
    <dict>
        <key>100</key>
        <string>Hi</string>
        <key>200</key>
        <string>Welcome</string>
        <key>300</key>
        <string>Welcome to the store!</string>
    </dict>
</dict>
```

…

NSNotificationCenter

In OS X 10.11 and iOS 9.0 NSNotificationCenter and NSDistributedNotificationCenter will no longer send notifications to registered observers that may be deallocated. If the observer is able to be stored as a zeroing-weak reference the underlying storage will store the observer as a zeroing weak reference, alternatively if the object cannot be stored weakly (i.e. it has a custom retain/release mechanism that would prevent the runtime from being able to store the object weakly) it will store the object as a non-weak zeroing reference. This means that observers are not required to un-register in their deallocation method. The next notification that would be routed to that observer will detect the zeroed reference and automatically un-register the observer. If an object can be weakly referenced notifications will no longer be sent to the observer during deallocation; the previous behavior of receiving notifications during dealloc is still present in the case of non-weakly zeroing reference observers. Block based observers via the -[NSNotificationCenter addObserverForName:object:queue:usingBlock] method still need to be un-registered when no longer in use since the system still holds a strong reference to these observers. Removing observers (either weakly referenced or zeroing referenced) prematurely is still supported. CFNotificationCenterAddObserver does not conform to this behavior since the observer may not be an object.

NSNotificationCenter and NSDistributedNotificationCenter will now provide a debug description when printing from the debugger that will list all registered observers including references that have been zeroed out for aiding in debugging notification registrations. This data is only valid per the duration of the breakpoint since the underlying store must account for multithreaded environments. Wildcard registrations for notifications where the name or object that

was passed to the addObserver method family was null will be printed in the debug description as *.

NSNumber

In OS X 10.11 and iOS 9 integral floating point and double numbers will now be created as tagged pointers when possible. This only applies to 64 bit architectures and integral values that can be stored in 56 bits of storage.

```
NSNumber *tagged = [NSNumber numberWithDouble:5.0];
NSNumber *notTagged = [NSNumber numberWithDouble:4.2];
```

Low Power Mode

iOS 9.0 introduced a new feature for reducing battery usage across applications and system settings called Low Power Mode. This setting can be read by accessing the newly introduced NSProcessInfo property lowPowerModeEnabled and can respond to changes of the state by registering for the notification NSProcessInfoPowerStateDidChangeNotification. When Low Power Mode is enabled applications should reduce power intense operations such as tasks that have heavy cpu load, network activity that may bring either WiFi or cellular radios online, or keeping the screen on.

The notification for NSProcessInfoPowerStateDidChangeNotification is posted on a global dispatch queue from the default NSNotificationCenter. The notification's object is the NSProcessInfo instance.

```
- (void)setup {
    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(lowPowerModeChanged:)
                                                 name:NSProcessInfoPowerStateDidChangeNotification object:nil];
}

- (void)lowPowerModeChanged:(NSNotification *)notification {
    if ([[NSProcessInfo processInfo] isLowPowerModeEnabled]) {
        // change behavior to consume less power
    } else {
        // restore behavior to normal
    }
}
```

NSProgress

NSProgress now has a -resume method, which parallels the behavior of the existing -pause method.

NSProgress now supports composition of progress objects into trees both implicitly and explicitly. The implicit approach uses the existing API, such as -becomeCurrentWithPendingUnitCount: and -resignCurrent. The explicit approach uses these new methods:

```
/* Directly add a child progress to the receiver, assigning it a portion of the receiver's total unit count.
 */
- (void)addChild:(NSProgress *)child withPendingUnitCount:(int64_t)inUnitCount;

/* Return an instance of NSProgress that has been attached to a parent progress with the given pending unit count.
 */
+ (NSProgress *)progressWithTotalUnitCount:(int64_t)unitCount parent:(NSProgress *)parent
                          pendingUnitCount:(int64_t)portionOfParentTotalUnitCount;

/* Return an instance of NSProgress that has been initialized with -initWithParent:userInfo:. The initializer is passed nil fo
   resulting in a progress object that is not part of an existing progress tree. The value of the totalUnitCount property is a
*/
+ (NSProgress *)discreteProgressWithTotalUnitCount:(int64_t)unitCount;
```

This additional API makes it appropriate to expose a progress object as a property of a class, because the progress object can now be retrieved and directly added as a child of another progress object. A new protocol has also been added, NSProgressReporting, which can be adopted to mark a class which reports progress in this way. When creating a progress object that will be vended from your class, you most likely want to use the new +discreteProgressWithTotalUnitCount: method to ensure you are creating a progress object which does not get implicitly attached to the current progress.

The rules about unit counts remain the same: completed and total unit count are in the units of the receiver NSProgress object. When adding a child, the pending unit count given to that child is also in the units of the receiver NSProgress object. NSProgress objects should either have no children and update their completedUnitCount directly ("leafy"), or they should have children and hand out all of their totalUnitCount as pendingUnitCount to those children.

In general, use the explicit addChild methods to compose trees when it is possible to get a progress object as the result of a method or a property of a class. Use the implicit methods to compose trees when the progress object is not directly accessible, for example calling through a proxy object (as NSProgress support over NSXPCConnection works today), or through an existing method which cannot gain an NSProgress return value (eg -[NSData dataWithContentsOfFile:]).

Starting in 10.11 and iOS 9.0, implicitly composed progress objects will no longer split pending unit count amongst themselves. This change was made to ensure that progress did not unintentionally move backwards, due to unanticipated child NSProgress objects being attached to the parent. Instead, the first child will consume the entire pending unit count. This means it is important for a method which supports implicitly attaching NSProgress objects by using +[NSProgress progressWithTotalUnitCount:] should make this call at the very first opportunity inside the implementation of the method. If other methods are invoked first, and they also use implicit progress, then there is the possibility that they will instead consume the pending unit count of the current progress. For example:

```
/* This method supports implicitly-created progress, and -becomeCurrentWithPendingUnitCount: was called right before this metho
- (void)myMethod {
    [object doSomething];
    [object doSomethingElse];
    NSProgress *p = [NSProgress progressWithTotalUnitCount:10]; // WRONG - doSomething or doSomethingElse may have already crea
                                                                //   and attached it to the currentProgress. This progress obj
    // work, where p.completedUnitCount is updated
}

- (void)myMethod {
    NSProgress *p = [NSProgress progressWithTotalUnitCount:10]; // RIGHT - by creating this first, we ensure that 'p' is attac
                                                                //    regardless of what the following two methods do.
    [object doSomething];
    [object doSomethingElse];
    // work, where p.completedUnitCount is updated
```

```
}
```

NSUserActivity

Foundation has new API to support the Handoff features of OS X Yosemite and iOS 8.0.

The NSUserActivity class provides a means for your application to store a small amount of information about what a user may be doing and to send that information from your application to another application of your creation when the user indicates that they would like to "hand off" this user activity from one device to another. The intention is that the application running on the second device, upon being given the information you created and put into a user activity, is able to reproduce the same state, showing the same information to the user, as is on the first device.

User activities allow a user to exchange information between applications created by the same developer. Applications that wish to receive user activities must be signed with the same developer certificate (and must share the same developer team identifier) as the application which created those user activities.

User activity type strings should follow a reverse-DNS style format (for example, "com.companyname.activityType".)

An application may have at most one "current" user activity, which is the NSUserActivity that the system would choose to show on a second device when this application is the frontmost application and the system has determined that a user activity can be handed off. Many factors contribute to whether a system makes a user activity available for Handoff, including whether the user appears to be interacting with the system, whether Bluetooth or Wi-Fi networking is available, and other factors. If you want to support Handoff, do not attempt to determine whether a system is capable of supporting Handoff. Simply create a user activity for whatever the user is doing and respond to any delegate messages that you receive. On devices where Handoff is not supported, the overhead of creating a user activity is quite low.

If you have a current user activity and wish to make it no longer current, but don't have another user activity that you want to make current, you should invalidate the user activity. If you need to make that user activity current again, you should recreate it.

Applications should fill in the title and userInfo properties of user activity objects they create. Its is acceptable to leave the title property empty, but if there is a reasonable value for it (for example, the name of the document the user is editing) then set it. It is not useful to set the title to the name of the application that creates the user activity, or to some other static string. The user info dictionary can contain any value or values of use to the application of the following classes: NSArray, NSData, NSDate, NSDictionary, NSNumber, NSString, NSURL, and NSUUID. The dictionary is generally transmitted unchanged between the creating application and the receiving application with the exception of NSURL objects representing files in the Cloud Documents container on the sending device. These NSURLs will be changed, on the receiving device, so that they point to the same file in the Cloud Documents folder on the receiving device, and so may have different paths or names than they did on the originating device.

Because Handoff is suppose to be a quick operation between two devices, it is important that the data placed into the user info dictionary be as small as possible. Your goal should be that the dictionary represent less than 3 KB of information. The exact method of encoding is an implementation detail, but the following code can give you a rough estimate of your dictionary's size:

```
NSInteger roughGuessAtDataSize = [NSKeyedArchiver archiveDataWithRootObject:userActivity.userInfo];
NSLog(@"The user info dictionary is roughly %lu bytes in size", (unsigned long)roughGuessAtDataSize.length);
```

Mac OS X 10.11 and iOS 9.0 add additional properties to NSUserActivity, for use by the App Search and History features.

Several new properties are added. Three new properties control when a NSUserActivity object will be available for Handoff, Spotlight, and public App Search. These properties are eligibleForHandoff, eligibleForSearch, and eligibleForPublicIndexing. All activities will initialize .eligibleForHandoff to YES and .eligibleForSearch and .eligibleForPublicIndexing to NO.

For an NSUserActivity which should be added to the local spotlight index, set eligibleForSearch to YES, add any appropriate keywords that describe this object to the keywords set, and set the contentAttributeSet property (which is declared as a category in CoreSpotlight.h, so you must #import <CoreSpotlight/CoreSpotlight.h>. When a user chooses at item from the Spotlight UI, your application will be opened with the application:continueUserActivity:restorationHandler: delegate.

For an NSUserActivity which should be found by other users via Siri, you should also set the requiredUserInfoKeys set to include only those keys in the top level of the .userInfo dictionary which contain values that are appropriate for the public indexing. See the NSUserActivity and App Search documentation for more information on how to decide this for your application.

A new property, expirationDate, can be set and which indicates a point in time after which an activity, even if still the current activity, should no longer be considered eligible for handoff, search, or public indexing.

If an NSUserActivity has a delegate set and if .eligibleForSearch or .eligibleForPublicIndexing set to YES, then periodically the delegate will get a userActivityWillSave: callback, to update the information in the NSUserActivity. The userActivityWasContinue: callback will only be called when the activity is sent via Handoff to another device.

NSURL New API

Several new file system resource property keys have been added for OSX and/or iOS.

Several new methods have been added:

```
-initFileURLWithPath:isDirectory:relativeToURL:
-initFileURLWithPath:relativeToURL:
+fileURLWithPath:isDirectory:relativeToURL:
+fileURLWithPath:relativeToURL:
-initWithDataRepresentation:relativeToURL:
+URLWithDataRepresentation:relativeToURL:
-initAbsoluteURLWithDataRepresentation:relativeToURL:
+absoluteURLWithDataRepresentation:relativeToURL:
-dataRepresentation
-hasDirectoryPath
```

NSURL Deprecations

In OSX 10.9 and iOS 7.0, we introduced replacement API for the NSURL methods and CFURL functions:

```
-stringByAddingPercentEscapesUsingEncoding:
-stringByReplacingPercentEscapesUsingEncoding:
CFURLCreateStringByAddingPercentEscapes();
CFURLCreateStringByReplacingPercentEscapesUsingEncoding();
```

The old "PercentEscapes" API was deficient and needed to be replaced for four reasons:

1 - Text encoding in URLs should be UTF8 because that is the text encoding the internet standards organizations have standardized on (see). When the "PercentEscapes" methods and functions were introduced over a decade ago, that wasn't always the case. Because the text encoding is not embedded in the

URL strings, using a non-standard text encoding will likely cause interoperability problems unless you know all code using the URL is using the same non-standard text encoding.

2 - With the older APIs, text encodings that are not a superset of ASCII encoding don't always work correctly (a text encoding that is a superset of ASCII encoding is where all 7-bit ASCII characters can be stored in a single 8-bit byte -- those text encodings include MacRoman, WindowsLatin1, ISOLatin1, NextStepLatin, ASCII, and UTF8) -- if you use one of those text encodings to percent-encode a string and then percent-decode the encoded string with the same text encoding, you won't always get the original string. Even with text encodings which are a superset of ASCII encoding, you can run into problems when one text encoding is used to percent-encode and a different text encoding is used to percent-decode because of characters which are not available in both text encodings.

3 - The "AddingPercentEscapes" methods and functions were originally documented in a way which made it seem you could pass an unpercent-encoded URL string in and get a correctly percent-encoded URL string out. That is not the case because each URL component has it's own rules about what unencoded characters are legal, and which illegal characters must be percent-encoded. The old "AddingPercentEscapes" code has these problems:
    The characters '/', ':', '?' and '@' in the user, password, and host sub-components are not being percent-encoded.
    The characters '[' and ']' in an IPv6 host address in the host sub-component are being percent-encoded.
    The characters ':', ';' and '?' in the path components are not being percent-encoded.
    The characters '"' and '#' are being percent-encoded everywhere in the string.

4 - It's never been documented what characters the "AddingPercentEscapes" methods and functions by default consider legal. That makes it hard to determine what they actually do, and in the case of CFURLCreateStringByAddingPercentEscapes(), how to correctly use the charactersToLeaveUnescaped and legalURLCharactersToBeEscaped arguments.The only ways to find out are to either look at the source code in CFURL.c (which is available open source), or to test by feeding all 7-bit ASCII characters through the API to see what characters are not percent-encode. FWIW: Here are the characters considered legal by default:

```
!$&'()*+,-./0123456789:;=?@ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyz~
```

Those characters are not the correct set for any URL component.

The replacement API for those two methods and two functions are:

```
-stringByAddingPercentEncodingWithAllowedCharacters:
-stringByRemovingPercentEncoding
```

With the predefined character sets:

```
+URLUserAllowedCharacterSet
+URLPasswordAllowedCharacterSet
+URLHostAllowedCharacterSet
+URLPathAllowedCharacterSet
+URLQueryAllowedCharacterSet
+URLFragmentAllowedCharacterSet
```

So, we're deprecating the old API. Because the old API has existed since OSX 10.3 and has always existed in iOS, there is a lot of code using it. What follows is a short guide to replacing the old API with the new API.

• "I was using a text encoding other than kCFStringEncodingUTF8 or NSUTF8StringEncoding."

If the input string only contained 7-bit ASCII characters and the text encoding was a superset of ASCII encoding, the deprecated API using one of those text encodings didn't do anything wrong and switching to kCFStringEncodingUTF8 or NSUTF8StringEncoding will work the same (only it'll be faster); otherwise, you should ask yourself why you thought that text encoding was the correct text encoding to use.

• "I was percent-encoding an entire URL string."

If you thought -stringByAddingPercentEscapesUsingEncoding: or CFURLCreateStringByAddingPercentEscapes() were working for you, they might have been working only by accident. If you are creating an URL from separate component values, you should use NSURLComponents to construct the URL. NSURLComponents knows now to correctly percent-encode the individual URL component and sub-components. If you need to percent-encode a single URL component or sub-component string (i.e., you aren't going to create a URL or URL string), you should use -stringByAddingPercentEncodingWithAllowedCharacters:. If you need to percent-encode an entire URL string, you can use this code to encode a NSString intended to be a URL (in urlStringToEncode):

```
NSString *percentEncodedURLString =
   [[NSURL URLWithDataRepresentation:[urlStringToEncode dataUsingEncoding:NSUTF8StringEncoding] relativeToURL:nil] rela
```

(The CoreFoundation equivalent to URLWithDataRepresentation: is CFURLCreateWithBytes() using the encoding kCFStringEncodingUTF8 with a fall back to kCFStringEncodingISOLatin1 if kCFStringEncodingUTF8 fails.)

• "I wasn't using the percent-escaping functions/methods for anything having to do with URLs -- they just looked like a good way to encode my data as ASCII so I used them."

-stringByAddingPercentEncodingWithAllowedCharacters: will also work for your purposes. You can easily create your own NSCharacterSet and then you'll know exactly what characters are being encoded and what characters are not being encoded. Or if one of the predefined URL character sets works for your purposes, you can use them.

• "My code cannot link with Foundation so it cannot use the new Foundation replacement API."

Unfortunately, you're going to have to keep using the old deprecated API. You will have to be careful to not trip over the problems which made a replacement API necessary. You should always use kCFStringEncodingUTF8. You should never use a text encoding that isn't a superset of ASCII encoding. You must not encode an entire URL string. You must make sure the characters you want percent-encoded are encoded and those you do not want percent-encoded are not (you may have to pass in strings in the charactersToLeaveUnescaped and legalURLCharactersToBeEscaped arguments to get the results you expect). You can disable deprecation warnings around your use of deprecated functions by using these #pragmas:

```
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wdeprecated-declarations"
    // use deprecated functions
#pragma GCC diagnostic pop
```

Other deprecations:
  -initWithScheme:host:path:

NSURLComponents New API

The following properties were added to NSURLComponents to return the character range of a component in the URL string returned by -[NSURLComponents string].

```
rangeOfScheme
rangeOfUser
rangeOfPassword
rangeOfHost
rangeOfPort
rangeOfPath
```

```
rangeOfQuery
rangeOfFragment
```

NSFileManager New API

NSFileManager now provides a method for unmounting and optionally ejecting file system volumes.

```
-unmountVolumeAtURL:options:completionHandler:
```

The options for unmountVolumeAtURL:options:completionHandler are:

```
NSFileManagerUnmountAllPartitionsAndEjectDisk
NSFileManagerUnmountWithoutUI
```

If unmountVolumeAtURL:options:completionHandler: fails, the process identifier of the dissenter can be found in the NSError's userInfo dictionary with the NSFileManagerUnmountDissentingProcessIdentifierErrorKey.

NSAffineTransform now conforms to NSSecureCoding

In OS X 10.11 and iOS 9, NSAffineTransform now conforms to NSSecureCoding. Invalid archives of NSAffineTransform will now raise an NSGenericException if they do not strictly match the expected layout.

Dictionaries created with dictionaryWithSharedKeySet now fully supports NSSecureCoding

Dictionaries created via NSSharedKey dictionary can now survive unarchive even if the the underlying hash implementation of the contained objects changes between archival/unarchival.

NSCache deadlock

On iOS 8 it was possible for NSCache to cause an application to deadlock if managed to retain itself during an UIApplicationDidEnterBackgroundNotification notification. This no longer occurs in iOS 9.

NSPropertyListSerialization now populates error on validation failures

On OS X 10.10 and iOS 8 and earlier, invalid property lists passed to -[NSPropertyListSerialization dataWithPropertyList:format:options:error] could return nil, but not populate the error (it would log instead). Beginning in OS X 10.11 and iOS 9, this API will now populate the error appropriately if provided.

NSCoder gains support for expressing decode failures via NSError

In OS X 10.10 and iOS 8 and earlier, instances of NSCoder communicated errors primarily through NSException, even if the underlying problem was not a programmer error (i.e.. corrupt archive). New to OS X 10.11 and iOS 9, all NSCoder implementations gain support for error notification through NSError out-parameters. These APIs should make it much easier to unarchive data in Swift.

You can opt-in to this behavior by calling any of these new top-level decode methods:

NSCoder

```
- (nullable id)decodeTopLevelObjectAndReturnError:(NSError **)error;
- (nullable id)decodeTopLevelObjectForKey:(NSString *)key error:(NSError **)error;
- (id)decodeTopLevelObjectOfClass:(Class)aClass forKey:(NSString *)key error:(NSError **)error;
- (nullable id)decodeTopLevelObjectOfClasses:(nullable NSSet<Class> *)classes forKey:(NSString *)key error:(NSError **)error;
```

NSKeyedUnarchiver

```
+ (id)unarchiveTopLevelObjectWithData:(NSData *)data error:(NSError **)error API_AVAILABLE(macos(10.11), ios(9.0), watchos(2.0
```

The top-level distinction is important, as NSCoder implementations still use exceptions internally to communicate failure. Any other decode-related API on NSCoder can still throw. The idea is that you would make use of one of the top-level methods, and then use the non-NSError variants as you would have previously.

In addition to the new top-level APIs, we additionally introduce a method that can be used to signal to your coder that the decode has failed.

```
- (void)failWithError:(NSError *)error API_AVAILABLE(macos(10.11), ios(9.0), watchos(2.0), tvos(9.0));
```

Typically, you would want to call this method in your -initWithCoder: implementation when you detect situations like:
- lack of secure coding
- corruption of your data
- domain validation failures

Before calling -failWithError: within your -initWithCoder: implementation, you should clean up any resources that are not automatically recovered (malloc's, etc.).

Once an error has been signaled to a decoder, it remains set until it has handed off to the first TopLevel decode invocation above it. For example, consider the following call graph:

```
A    -decodeTopLevelObjectForKey:error:
B        -initWithCoder:
C            -decodeObjectForKey:
D                -initWithCoder:
E                    -decodeObjectForKey:
F                        -failWithError:
```

In this case the error provided in stack-frame F will be returned via the outError in stack-frame A. Furthermore the result object from -decodeTopLevelObjectForKey:error: will be nil, regardless of the result of stack-frame B.

The call-stack-unwinding from frame F to A utilizes NSException which matches the historical behavior of decode failures for NSCoder based unarchival.

To differentiate between missing values and invalid archives, the following NSError code can be examined for: NSCoderValueNotFoundError when the decode method returns nil. Note that this error will not be encountered in Swift, where the decode methods do return an optional.

It is still possible for the top-level APIs to raise NSExceptions in the presence of programmer error (e.g. calling -decodeObjectForKey: on an unarchiver that you have already called -finishDecoding).

NSUndoManager Block-based Undo

In the interest of providing an easier way to use implement undo operations in Swift, NSUndoManager gains the following API:

```
/*! @abstract records single undo operation for the specified target
    @param target non-nil target of the undo operation
    @param undoHandler non-nil block to be executed for the undo operation
    @discussion
      As with other undo operations, this does not strongly retain target. Care should be taken to avoid introducing
      retain cycles by other references captured by the block.
 */
- (void)registerUndoWithTarget:(id)target handler:(void (^)(id target))undoHandler;
```

Additionally, like the other undo operations (target/selector and invocation), the target can be used with the -removeAllActionsWithTarget:.

Single Object NSSet Performance Improvements

Changes have were made to dramatically increase the performance of single-object immutable sets.

Various Performance Improvements in our collections

Improved immutable NSArray performance across the board, especially around access, creation and deletion.

NSDictionary -isEqual: is now faster, along with numerous other dictionary related operations.

Safer Buffer API for NSDictionary

In OS X 10.11 and iOS 9, NSDictionary gains the following method:

```
- (void)getObjects:(ObjectType __unsafe_unretained [])objects andKeys:(KeyType __unsafe_unretained [])keys count:(NSUInteger)co
```

It is intended to replace the following method, which is soft-deprecated this release, and will be officially deprecated in the future.

```
- (void)getObjects:(ObjectType __unsafe_unretained [])objects andKeys:(KeyType __unsafe_unretained [])keys;
```

Safer Buffer API for NSIndexPath

Similarly, NSIndexPath gains the following method:

```
- (void)getIndexes:(NSUInteger *)indexes range:(NSRange)positionRange;
```

Replacing the older, less safe variant (which is now soft-deprecated):

```
- (void)getIndexes:(NSUInteger *)indexes;
```

NSFileVersion hasLocalContents Bug Fix

In OS X 10.10 and iOS 8, NSFileVersion exposed the "hasLocalContents" property to indicate whether a particular NSFileVersion originated locally, or from some non-local source, like iCloud. Unfortunately, the property returned the opposite value than intended. This is has been fixed in OS X 10.11 and iOS 9.

NSMutableDictionary subscript syntax change

In OS X 10.11 and iOS 9, NSMutableDictionary now allows you to use its subscript syntax to assign nil for a key. Like Swift's Dictionary type, doing so will remove an existing object for that key from the dictionary. In effect, the following lines of code are now equivalent:

```
[dictionary removeObjectForKey:@"Key"];

dictionary[@"Key"] = nil;
```

These new semantics exist only when building with the OS X 10.11 or iOS 9 SDKs. If your application's deployment target is earlier operating system, then runtime support will be implicitly linked into your binary by Xcode to ensure the behavior works on any targeted operation system.

NSFileCoordinator and NSFilePresenter usage on iOS (New since WWDC Seed)

Prior to iOS 9, NSFileCoordinator and NSFilePresenter were documented as unsafe to use for coordinating access to files in iCloud or group containers. If an application or extension left an NSFilePresenter registered for a file in such a container or had an active coordinated read or write at the time the application was backgrounded and suspended, future attempts to access these files would deadlock.

In iOS 9, NSFileCoordinator and NSFilePresenter were improved to avoid these problems. In general, applications and extensions should typically remove their NSFilePresenters via -[NSFileCoordinator removeFilePresenter:] when the application enters the background. For applications that fail to do so, the system will now automatically kill a suspended application or extension with an NSFilePresenter that File Coordination would need to message and get a response (i.e. relinquishPresentedItemToWriter:) to avoid deadlock. If such an application or extension got "lucky" and was not killed while suspended, a message will be logged in the application when it becomes active again to help you discover this problem.

One-way NSFilePresenter messages are not vulnerable to deadlock, so starting in iOS 9, it is now safe to leave an NSFilePresenter registered if it does not implement any NSFilePresenter methods that require a reply via a block (i.e. a completionHandler or relinquisher block).

Additionally, a coordinated read or write, when granted, will now automatically begin a background task similar to -[UIApplication beginBackgroundTaskWithExpirationHandler:]. This should ensure that your application has enough time to finish the operation and unblock other processes's attempts to do coordinated access of the file. If the background task expires, the process will be killed. If a process is suspended while waiting for a coordinated read or write to be granted, the request will be cancelled, returning an NSError with NSUserCancelledError.

NSURLConnection

NSURLConnection is deprecated as of OS X 10.11 and iOS 9. Please use NSURLSession.

Application Transport Security

Application Transport Security is a new security feature whose goal is to protect customer data. If your app is linked on or after OS X 10.11 or iOS 9, all NSURLSession and NSURLConnection based cleartext HTTP loads (http://) will be denied by default. Encrypted HTTPS (https://) connections will also require "best practice" TLS behaviors, such as TLS version and cipher suite requirements. Temporary exceptions can be configured via your app's Info.plist file. For more information, see the WWDC sessions covering Application Transport Security.

WatchOS

The symbols NSURLSessionTaskPriorityDefault, NSURLSessionTaskPriorityHigh and NSURLSessionTaskPriorityLow are missing from the WatchOS SDK.

NSLocaleMeasurementSystem Support for UK

In appropriate locales, NSLocale may now return @"U.K." for the NSLocaleMeasurementSystem key. This indicates "imperial" values for non-metric measurements, e.g. a pint is 568.3 mL (vs a US pint's 473.2 mL); can also serve to indicate the mix of metric and non-metric measurements used in UK, such as miles for road distances but litres for volume of gasoline and other liquids.

NSString

NSString now exposes API for transliteration. This was available in CFString previously.

```
- (nullable NSString *)stringByApplyingTransform:(NSString *)transform
                                         reverse:(BOOL)reverse;
```

The indicated transformation is applied to the receiver (see below for the predefined set of transforms provided in NSString). reverse indicates that the inverse transform should be used instead, if it exists. Attempting to use an invalid transform identifier or reverse an irreversible transform will return nil; otherwise the transformed string value is returned (even if no characters are actually transformed). In addition to the predefined transforms, you can pass in any valid ICU transform ID as defined in the ICU User Guide. Arbitrary ICU transform rules are not supported.

There's also API on NSMutableString for the same thing:

```
- (BOOL)applyTransform:(NSString *)transform
               reverse:(BOOL)reverse
                 range:(NSRange)range
          updatedRange:(nullable NSRangePointer)resultingRange;
```

Similar to the NSString API, except that only the specified range will be modified (however the transform may look at portions of the string outside that range for context). If supplied, resultingRange is modified to reflect the new range corresponding to the original range.

The following transforms are provided:

```
NSString * const NSStringTransformLatinToKatakana;
NSString * const NSStringTransformLatinToHiragana;
NSString * const NSStringTransformLatinToHangul;
NSString * const NSStringTransformLatinToArabic;
NSString * const NSStringTransformLatinToHebrew;
NSString * const NSStringTransformLatinToThai;
NSString * const NSStringTransformLatinToCyrillic;
NSString * const NSStringTransformLatinToGreek;
NSString * const NSStringTransformToLatin;
NSString * const NSStringTransformMandarinToLatin;
NSString * const NSStringTransformHiraganaToKatakana;
NSString * const NSStringTransformFullwidthToHalfwidth;
NSString * const NSStringTransformToXMLHex;
NSString * const NSStringTransformToUnicodeName;
NSString * const NSStringTransformStripCombiningMarks;
NSString * const NSStringTransformStripDiacritics;
```

The following two new methods are the most appropriate methods for doing user-level string searches, similar to how searches are done generally in the system. The search is locale-aware, case and diacritic insensitive. As with other APIs, "standard" in the name implies "system default behavior," so the exact list of search options applied may change over time. If you need more control over the search options, please use the rangeOfString:options:range:locale: method. You can pass [NSLocale currentLocale] for searches in user's locale.

```
- (BOOL)localizedStandardContainsString:(NSString *)str;
- (NSRange)localizedStandardRangeOfString:(NSString *)str;
```

The following are new convenience methods that do the case mappings in a locale-aware fashion.

```
- (NSString *)localizedUppercaseString;
- (NSString *)localizedLowercaseString;
- (NSString *)localizedCapitalizedString;
```

NSStringGenderRuleType Bugfix

We fixed a longstanding bug in stringsdict's NSStringGenderRuleType. A properly-formatted example of this is below:

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
```

```
<plist version="1.0">
<dict>
    <key>test</key>
    <dict>
        <key>NSStringLocalizedFormatKey</key>
        <string>%@ is %#@pronoun_gender@ name</string>
        <key>pronoun_gender</key>
        <dict>
            <key>NSStringFormatSpecTypeKey</key>
            <string>NSStringGenderRuleType</string>
            <key>NSStringFormatValueTypeKey</key>
            <string>d</string>
            <key>0</key>
            <string>his</string>
            <key>1</key>
            <string>her</string>
            <key>2</key>
            <string>their</string>
        </dict>
    </dict>
</dict>
</plist>
```

Usage:

```
NSString *gendered = [NSString localizedStringWithFormat:NSLocalizedStringFromTable(@"test", @"gender", @"comment"), @"Nat", 0
```

Outputs:
Nat is his name

If instead you call:

```
NSString *gendered = [NSString localizedStringWithFormat:NSLocalizedStringFromTable(@"test", @"gender", @"comment"), @"Nat", 1
```

You get:
Nat is her name

And:

```
NSString *gendered = [NSString localizedStringWithFormat:NSLocalizedStringFromTable(@"test", @"gender", @"comment"), @"Nat", 2
```

You get:
Nat is their name

Tagged Pointer NSStrings

Starting with iOS 9 certain strings (ones with a suitable length and encoding) on 64 bit architectures will now use a “tagged pointer” format where the string contents are stored directly in the pointer. This matches behavior introduced on OS X in 10.10 Yosemite.

Similarly to the OS X targets, passing a tagged NSString to functions such as CFStringGetCStringPtr or CFStringGetCharactersPtr will return NULL in cases where it may not have before. As before it is important to check for the NULL return value and use the corresponding buffer fetching function:

```
char buffer[BUFSIZE];
const char *ptr = CFStringGetCStringPtr(str, encoding);
if (ptr == NULL) {
    if (CFStringGetCString(str, buffer, BUFSIZE, encoding)) ptr = buffer;
}
```

In addition, this change enables index and range checking to be performed more often when working with strings. So you may see runtime exceptions due to this change. In almost all cases these exceptions point to bugs in code, so please take them seriously.

This change will also break any code which treats NS/CFStrings objects as pointers and attempts to dereference them.

NSString's +stringEncodingForData: API Enhancements

In OS X 10.11 and iOS 9, if you called +[NSString stringEncodingForData:encodingOptions:convertedString:usedLossyConversion:] with a suggested encoding via NSStringEncodingDetectionSuggestedEncodingsKey and allowed lossy detection via NSStringEncodingDetectionAllowLossyKey you will always get a result. Previously in OS X 10.0 and iOS 8, the API could fail to find an encoding in such circumstances.

NSPersonNameComponentsFormatter

NSPersonNameComponentsFormatter is a new formatter which enables formatting and displaying person names properly in different localizations. It formats objects of type NSPersonNameComponents, which allows specifying different parts of names.

NSError Value Provider

We recommend that NSErrors are as rich as possible, with localized description, failure reason, recovery attempter, etc. This in turn may make the creation of NSErrors more expensive, and tight loops where expected errors are being encountered could be an issue. To help with this, NSError now allows you to specify a block as the userInfoValueProvider for a domain. With this, at the time you create an NSError, you will provide the minimal amount of info needed, often just the domain and the code, and in some cases a userInfo value or two. Then, the provider can provide the other info to present great NSErrors lazily, on-demand:

```
+ (void)setUserInfoValueProviderForDomain:(NSString *)errorDomain
                                 provider:(id __nullable (^ __nullable)(NSError *err, NSString *userInfoKey))provider;
+ (id __nullable (^ __nullable)(NSError *err, NSString *userInfoKey))userInfoValueProviderForDomain:(NSString *)errorDomain;
```

Specify a block which will be called from the implementations of localizedDescription, localizedFailureReason, localizedRecoverySuggestion, localizedRecoveryOptions, recoveryAttempter, and helpAnchor, when the underlying value for these is not present in the userInfo dictionary of NSError instances with the specified domain. The provider will be called with the userInfo key corresponding to the queried property: For instance, NSLocalizedDescriptionKey for localizedDescription. The provider should return nil for any keys it is not able to provide and, very importantly, any keys it does

not recognize (since we may extend the list of keys in future releases).

It's important that the provider return nil if an appropriate result for the requested key cannot be provided, rather than choosing to manufacture a generic fallback response such as "Operation could not be completed, error 42." A nil return will enable NSError to look for other ways to present the needed info (for instance, asking for NSLocalizedFailureReasonErrorKey in addition to NSLocalizedDescriptionKey). If nothing succeeds NSError will generate a properly localized fallback message.

The specified block will be called synchronously at the time when the above properties are queried. The results are not cached.

This provider is optional. It enables localization and formatting of error messages to be done lazily; rather than populating the userInfo at NSError creation time, these keys will be fetched on-demand when asked for.

It is expected that only the "owner" of an NSError domain specifies the provider for the domain, and this is done once. This facility is not meant for consumers of errors to customize the userInfo entries. This facility should not be used to customize the behaviors of error domains provided by the system.

NSError and Swift

Swift 2 introduces a new error handling mechanism which uses terminology such as throw, catch, etc. Although these terms are similar to terms used with exceptions in other contexts, Swift's error handling is not exception handling in the traditional Objective-C or Cocoa sense. Swift's error handling is a mechanism for reporting expected errors and properly cleaning up after them. As such, Swift's error handling is designed to interoperate well with NSError and Cocoa error handling patterns, and Swift's ErrorType is bridged to NSError using "as NSError," for instance in catch statements:

```
do {
    try fileManager.removeItemAtURL(URL)
} catch let error as NSError {
    NSApp.presentError(error)    // A simplified approach to presenting an error to the user on OS X
}
```

Objective C and Swift code can still raise exceptions in the traditional sense (using NSException.raise()), and in fact APIs will continue to react to programming errors (such as "array index out of bounds") by raising traditional exceptions. Such exceptions are often not meant to be dealt with, and are also not caught by the Swift error handling mechanism.

NSError

There have been some changes in the ways NSErrors are formatted. The most significant change is that -[NSError description] and NSErrors formatted with %@ no longer make an extra effort to localize, although they may still return localized results in cases where only a localized value is available. Since the results of -[NSError description] and %@ are already not user presentable, unless your application parses the output of -[NSError description] or NSErrors formatted with %@, this change shouldn't impact anything. Note that APIs such as -[NSError localizedDescription] and presentError: will of course continue to still localize where localizations are available.

Notes specific to OS X 10.10 and iOS 8

Swift

Swift is a new programming language for Cocoa and Cocoa Touch that provides many innovative features while also seamlessly operating with Cocoa APIs and Objective-C. You can find documentation, release notes, and other resources for Swift development on the Apple developer website.

Some items of note regarding the interaction of Swift with existing Cocoa APIs:

• Instance methods in Cocoa APIs come across to Swift pretty much as-is. A method such as:

```
- (BOOL)setResourceValue:(id)value forKey:(NSString *)key error:(NSError **)error;
```

in Swift looks like:

```
func setResourceValue(value:AnyObject?, forKey key:String?, error:NSErrorPointer) -> Bool
```

The first part of the Objective-C name appears as the first part of the Swift name, outside the parens, and the labels on the second and subsequent arguments appear as labels in Swift as well.

In both cases, "value" and "key" are simply the names of the local variable corresponding to the first and second arguments, and are not part of the method name.

Important to note that the Swift name of such a method includes the labels, so the name of such a method would be read as in ObjC, as "setResourceValue forKey error", not "setResourceValue." Removing the names of the arguments, the method signature is:

```
func setResourceValue(AnyObject?, forKey:String?, error:NSErrorPointer) -> Bool
```

So basically Objective-C instance method names are exposed automatically and without any changes in Swift.

init methods are also exposed automatically, but there is a mapping: Both init methods and convenience constructors are exposed in Swift as constructors. In such cases if present, "with" is dropped, and all the arguments, including the first one, have an explicit label. So a method such as:

```
- (instancetype)initWithTimeIntervalSinceNow:(NSTimeInterval)secs;
```

is exposed as:

```
convenience init(timeIntervalSinceNow secs: NSTimeInterval)
```

and the same thing happens with a convenience constructor such as:

```
+ (NSColor *)colorWithPatternImage:(NSImage *)image;
```

which appears as:

```
init(patternImage image: NSImage?)
```

• As you can see in the above examples, some Objective-C Cocoa types are mapped to their Swift counterparts, such as NSString/String above. Also note that

by-ref NSError return arguments (NSError **) appear as NSErrorPointer in Swift. id appears in Swift as AnyObject, and NSArray appears as AnyObject[]. NSInteger is mapped to Int, and in most cases so is NSUInteger.

• Cocoa structs such as NSRange and NSSize are available to Swift with custom constructors, such as:

```
let size = NSSize(width: 20, height: 40)
```

• To localize strings in your applications you can use the NSLocalizedString(key:tableName:bundle:value:comment:) function. This function provides default values for the tableName, bundle, and value arguments, so you can use it in a variety of forms, shortest being:

```
result = NSLocalizedString("Update", comment:"Title of button the user can click to update account info")
```

• Enums in Cocoa which follow the "common prefix" naming guideline (which applies to the more recent enums) appear in Swift with the common prefix removed. For instance:

```
typedef NS_ENUM(NSInteger, NSByteCountFormatterCountStyle) {
    NSByteCountFormatterCountStyleFile,
    NSByteCountFormatterCountStyleMemory,
    NSByteCountFormatterCountStyleDecimal,
    NSByteCountFormatterCountStyleBinary
};
```

is exposed in Swift as:

```
enum NSByteCountFormatterCountStyle : Int {
    case File
    case Memory
    case Decimal
    case Binary
}
```

and you can refer to the individual values as NSByteCountFormatterCountStyle.File or just .File in appropriate contexts.

• Objective-C selectors appear using the Selector type in Swift. You can construct a selector with a string literal, such as

```
let action: Selector = "updateAccountInfo:"
```

String Encoding Detection

NSString has added an API that can be used to detect the string encoding of an array of bytes. The API is:

```
+ (NSStringEncoding)stringEncodingForData:(NSData *)data
                          encodingOptions:(NSDictionary *)opts
                          convertedString:(NSString **)string
                      usedLossyConversion:(BOOL *)usedLossyConversion;
```

This API is used to detect the string encoding of a given raw data. It can also do lossy string conversion. It converts the data to a string in the detected string encoding. The data object contains the raw bytes, and the option dictionary contains the hints and parameters for the analysis. The opts dictionary can be nil. If the string parameter is not NULL, the string created by the detected string encoding is returned. The lossy substitution string is emitted in the output string for characters that could not be converted when lossy conversion is enabled. The usedLossyConversion indicates if there is any lossy conversion in the resulted string. If no encoding can be detected, 0 is returned.

The possible key for the option dictionary and their values are:

• Key: NSStringEncodingDetectionSuggestedEncodingsKey; Value: an array of suggested string encodings (without specifying the 3rd option in this list, all string encodings are considered but the ones in the array will have a higher preference; moreover, the order of the encodings in the array is important: the first encoding has a higher preference than the second one in the array)
• Key: NSStringEncodingDetectionDisallowedEncodingsKey; Value: an array of string encodings not to use (the string encodings in this list will not be considered at all)
• Key: NSStringEncodingDetectionUseOnlySuggestedEncodingsKey; Value: a boolean option indicating whether only the suggested string encodings are consider
• Key: NSStringEncodingDetectionAllowLossyKey; Value: a boolean option indicating whether lossy is allowed
• Key: NSStringEncodingDetectionFromWindowsKey; Value: an option that gives a specific string to substitute for mystery bytes
• Key: NSStringEncodingDetectionLossySubstitutionKey; Value: the current user's language
• Key: NSStringEncodingDetectionLikelyLanguageKey; Value: a boolean option indicating whether the data is generated by Windows

If the values in the dictionary have wrong types (for example, the value of NSStringEncodingDetectionSuggestedEncodingsKey is not an array), an exception is throw. If the values in the dictionary are unknown, (for example, the value in the array of suggested string encodings is not a valid encoding), the values will be ignored.

If the string encoding is known, +[NSString initWithData:encoding:] should be used to create the string. Use this new API to create a string only if the string encoding is unknown or lossy conversion is needed.

Date Interval Formatter

NSDateIntervalFormatter has been added in 10.10. It is used to format a date interval in a locale-sensitive way. A date interval is defined by two dates, for example, Sept 1st, 2013 - Oct 1st, 2013. In different countries and different regions, people use different languages and different formats to express a date interval. NSDateIntervalFormatter is designed to fulfill the need that we can easily format a date interval into a localized string.

NSDateIntervalFormatter is a subclass of NSFormatter. However, it does not support parsing. In addition, NSDateIntervalFormatter requires two objects to work on, so the base methods in NSFormatter such as stringForObjectValue: do not apply.

There is a single method in NSDateIntervalFormatter: -stringFromData:toDate:. How the dates are formatted is controlled by either the dateTemplate property or the dateStyle and timeStyle properties (just like NSDateFormatter). The dateTemplate property is different from the dateFormat property in NSDateFormatter. The position information of each unit in dateTemplate is ignored, and the value of the dateTemplate may be changed and normalized.

Date Components Formatter

Foundation now provides localized formatting of durations and quantities of time via the new NSDateComponentsFormatter class. The header file NSDateComponentsFormatter.h is extensively commented with examples, information, and usage guidelines. The 'formattingContext' property mentioned below is not yet implemented in the WWDC seed.

Unit Formatters

Three new unit formatters have been added in 10.10. They are NSMassFormatter, NSLengthFormatter, and NSEnergyFormatter. These formatters do not support either parsing or unit conversion. These formatters can format a combination of a value and a unit into a localized string. They can also be used to get a localized string of a unit, and if the unit is singular or plural is based on the given value.

NSMassFormatter has a special property called isForPersonMassUse. This property should be set to YES if the formatter is used to format a person's mass. NSEnergyFormatter has a special property called isForFoodEnergyUse. This property should be set to YES if the formatter is used to format values for food energy.

Formatting Context

A new property called formattingContext has been added to NSDateFormatter, NSNumberFormatter, NSDateComponentsFormatter, and NSByteCountFormatter to express the context information for data formatting. The context information is where the data will be displayed so that it can be capitalized appropriately. For example, a date or the date element could be for the beginning of a sentence, the middle of a sentence, for display in a UI list, or for standalone display. The date or the date element may have different capitalization for different context. The formatting context API gives a better control of how the data is capitalized.

There are 6 different values for formattingContext, and they are defined in NSFormatter.h.

NSFormattingContextDynamic should be used in most of the time. When NSFormattingContextDynamic is used, the capitalization context is determined dynamically from the set {NSFormattingContextStandalone, NSFormattingContextBeginningOfSentence, NSFormattingContextMiddleOfSentence}. For example, if a date is placed at the beginning of a sentence, NSFormattingContextBeginningOfSentence is used to format the string automatically. When this context is used, the formatter will return a string proxy that works like a normal string in most cases. After returning from the formatter, the string in the string proxy is formatted by using NSFormattingContextUnknown. When the string proxy is used in -[NSString stringWithFormat:], we can determine where the %@ is and then set the context accordingly. With the new context, the string in the string proxy will be formatted again and be put into the final string returned from -[NSString stringWithFormat:].

NSDateFormatter Bug Fix

The older 10.0-style NSDateFormatter implementation could log error messages even when being used correctly. This has been corrected, though it's still preferable to transition to 10.4-style formatters.

NSCalendar

Two Islamic calendar variants have been added in 10.10:

```
// A simple tabular Islamic calendar using the astronomical/Thursday epoch of CE 622 July 15
NSString * const NSCalendarIdentifierIslamicTabular;
// The Islamic Umm al-Qura calendar used in Saudi Arabia. This is based on astronomical calculation, instead of tabular behavi
NSString * const NSCalendarIdentifierIslamicUmmAlQura;
```

They can be used to create NSCalendar objects with initWithCalendarIdentifier:.

Bug fix:
The following methods now work correctly in 10.10 when the NSDateComponents argument has isLeapMonth set for the Chinese calendar:
- enumerateDatesStartingAfterDate:matchingComponents:options:usingBlock:
- nextDateAfterDate:matchingComponents:options:

Behavior change:
The behavior of the following methods has been changed in 10.10 with the NSCalendarMatchPreviousTimePreservingSmallerUnits option or the NSCalendarMatchNextTimePreservingSmallerUnits option:

- enumerateDatesStartingAfterDate:matchingComponents:options:usingBlock:
- nextDateAfterDate:matchingComponents:options:

In 10.9, if the date that matches the giving components is missing, the methods will return the previous (or the next, depending on which option is specified) existing value of the highest unit with exists and preserves the lower units' values. For example, giving 2014-01-01 11:11:11, if the components are Feb. 29, the methods will return Jan. 29 00:00:00 in 2014 giving the NSCalendarMatchPreviousTimePreservingSmallerUnits option.

In 10.10, the behavior has been changed: if the date that matches the giving components is missing, the methods will return the previous (or the next, depending on which option is specified) existing value of the missing unit and preserves the lower units' values of the giving date. For example, giving 2014-01-01 11:11:11, if the components are Feb. 29, the methods will return Feb 28th 11:11:11 in 2014 giving the NSCalendarMatchPreviousTimePreservingSmallerUnits option.

NSXPCConnection and NSProgress

In OS X 10.10 and iOS 8.0, NSXPCConnection and NSProgress have been integrated to work together seamlessly.

To receive progress updates from work done in another process, simply make an NSProgress object current before calling out to your remoteObjectProxy. If the other side supports reporting progress, then the NSProgress object in your process will be updated as work is completed in the remote process.

For example, the following code is part of an application which asks an XPC helper service to download a file.

```
- (IBAction)fetch:(id)sender
{
    NSURL *urlToFetch = ...;
    NSProgress *progress = [NSProgress progressWithTotalUnitCount:1];
    [progress becomeCurrentWithPendingUnitCount:1];
    // Create a connection to our fetch-service and ask it to download for us. The fetch-service will implement the 'Fetch' pr
    NSXPCConnection *fetchServiceConnection = [[NSXPCConnection alloc] initWithServiceName:@"com.apple.SandboxedFetch.fetch-se
    fetchServiceConnection.remoteObjectInterface = [NSXPCInterface interfaceWithProtocol:@protocol(Fetch)];
    [fetchServiceConnection resume];
    // Send a message to the fetch service. Because there is a current progress object ('progress'), it will be updated if the
    [[fetchServiceConnection remoteObjectProxy] fetchURL:urlToFetch withReply:^(BOOL ok) {
        // This block will be invoked when the helper service replies.
    }];
    // Even if the work is done asynchronously, we can resignCurrent here.
    [progress resignCurrent];
}
```

In the service, which implements the fetchURL:withReply: method, a current progress object is already setup for us and we simply need to attach a new progress to it to represent the work that we will be doing.

```
- (void)fetchURL:(NSURL *)url withReply:(void (^)(BOOL))reply {
    // Create a progress object to handle progress below
    NSProgress *progress = [NSProgress progressWithTotalUnitCount:10];
    progress.cancellationHandler = ^{
        // handle a cancellation from the application
    };
    // do work here, e.g.
    progress.completedUnitCount = 1;
    // more work
    progress.completedUnitCount = 5;
    // etc.
   reply(YES);
}
```

Changing Filename Case with NSFileManager

In previous releases, the NSFileManager methods -moveItemAtPath:toPath:error: and -moveItemAtURL:toURL:error: would fail with NSFileWriteFileExistsError when both parameters refer to the same item on disk. This operation now succeeds, which on case-insensitive but case-preserving file systems (like HFS+ on OS X) will allow changing the case of a file's name.

More NSSecureCoding Adoption

NSIndexSet and NSIndexPath now both implement the NSSecureCoding protocol, which allows them to be used with NSXPC APIs.

NSFilePresenter File Package Attribute Change Notifications

In previous releases, -presentedItemDidChange was not sent to NSFilePresenters of file packages when only attributes of that package were changed. This has been fixed on OS X 10.0 and iOS 8.0.

Bug Fix in NSMetadataItem

In OS X 10.9, creating an NSMetadataItem with a URL for which no file exists would cause a subsequent -valueForAttribute: message to crash. Instead of crashing, -initWithURL: will now return nil in this situation.

Increased Leniency for NSDataBase64DecodingIgnoreUnknownCharacters

In OS X 10.9, decoding a base 64 string with "=" characters before the end of the string would cause the entire decoding operation to fail. However, RFC 4648 specifically allows implementations to ignore these characters, treating them like other unknown characters. The NSData base 64 decoding algorithm has been updated to allow for this when NSDataBase64DecodingIgnoreUnknownCharacters is used.

NSFileCoordinatorReadingForUploading

NSFileCoordinator has a new reading option that is meant to facilitate uploading user documents to web services like web pages or mail servers that only understand regular files and not file packages.

To use this API, request a coordinated read on any user document, the same way you would for normal reading, except use NSFileCoordinatorReadingForUploading. Before your accessor block is invoked, NSFileCoordinator will check if the file is a directory or not. If it is, it will create a zip archive of that directory in a temporary folder that is accessible by your application. If the file is not a directory, then it will be copied to that directory instead. The URL to the newly created temporary file will be accessible within your accessor block.

For normal coordinated reads, other coordinated writes are made to wait until you return from the accessor block. However, when using this option, other processes will be able to resume accessing that file immediately after the temporary file is created. This will allow you to do whatever is necessary to prepare the file for uploading without blocking other processes for arbitrary lengths of time.

The temporary file NSFileCoordinator creates will be unlinked as soon as you return from the accessor block. If you need to retain access to that file outside of the block, then you can copy, move, or hardlink the file to another location, or keep an open file descriptor to it.

This API is not intended to be a general-use facility for creating zip archives of files. Use it only for uploading.

Asynchronous NSFileCoordinator Waiting

Prior to OS X 10.10 and iOS 8.0, the recommended way to asynchronously wait for a coordinated read or write was to dispatch a block to another queue that then calls -coordinateReadingItemAtURL:options:error:byAccessor: (or one of the three other similar methods), which then waits synchronously on that queue. However, this pattern was wasteful of threads and dispatch queues.

There is a new API available on NSFileCoordinator called -coordinateAccessWithIntents:queue:byAccessor: that will wait asynchronously for access to the requested URLs without consuming additional queues. When access has been granted, the method will invoke your accessor block on a queue that you specify.

Like the older methods, the accessor block passed to this method is still run synchronously—when you return from the block, coordinated access to the file will end. Unlike the older methods, this new API allows you to perform coordination for an arbitrary number of files by passing an array of NSFileAccessIntent objects. NSFileAccessIntent encapsulates the URL, whether the access is a read or a write, and the reading or writing options. When your accessor block is invoked, you should use the URL property on the NSFileAccessIntent object, not the NSURL object you used to create that NSFileAccessIntent. If a document you are trying to access is moved or renamed, then NSFileCoordinator will update the URL property of the NSFileAccessIntent with the new value. NSFileCoordinator is unable to modify any other URLs that you have captured with the accessor block. Here's an example of proper usage of this API:

```
NSFileAccessIntent *srcIntent = [NSFileAccessIntent readingIntentWithURL:url1 options:NSFileCoordinatorWritingForMoving];
NSFileAccessIntent *dstIntent = [NSFileAccessIntent writingIntentWithURL:url2 options:NSFileCoordinatorWritingForReplacing];
[fileCoordinator coordinateAccessWithIntents:@[srcIntent, dstIntent] queue:myQueue byAccessor:^(NSError *error) {
    BOOL success = error == nil;
    if (success) {
        success = [fileManager removeItemAtURL:dstIntent.URL error:&error];
    }
    if (success) {
```

```
        success = [fileManager moveItemAtURL:srcIntent.URL toItemAtURL:dstIntent.URL error:&error];
    }
    if (!success) {
        // present error on main queue
    }
}];
```

Note that if an error occurs, the NSError is passed to the accessor block instead of being returned by reference via an NSError ** parameter.

Debugging File Coordination Delays

With File Coordination, applications have several opportunities to block other applications from being able to gain coordinated access to files. For example, they can delay returning from an NSFileCoordinator accessor block, or delay invoking blocks passed to NSFilePresenter messages. Sometimes, bugs in these applications can cause these delays to become indefinitely long. Due to the asynchronous and inter-process nature of File Coordination, it can often be difficult to understand and debug these issues.

In OS X 10.10 and iOS 8.0, the NSFileCoordinator class now responds to a method called +(void)_printDebugInfo. This method will print all available information about each file or directory involved in File Coordination on your system, and each process that is interacting with it. This method is intended for use in the debugger, so do not use this method in shipping code. It is not declared in any headers and may be removed or renamed without notice.

The first section of the output will show a compressed hierarchy of files that the File Coordination daemon is tracking. It shows what processes have NSFilePresenters registered for these files, and which ones have claims outstanding or pending for them. When a claim is delayed for any reason, you will be able to see whether that claim is waiting for another conflicting claim to finish, or whether it is waiting for an NSFilePresenter to respond to the claim.

The second section of the output will show each process that is interacting with File Coordination. It shows NSFilePresenter activity, like what NSFilePresenter messages have been sent but not yet responded to, as well as NSDocument serialization info, if applicable.

Using all of this information, it should become much easier to understand and debug issues you may be experiencing with File Coordination. On OS X, all of this same information is provided in the archive created by the ‘sysdiagnose’ tool, which will aid you in debugging problems that occur on customer’s computers. Note that this information includes file and process names, but does not include any file contents.

NSFileCoordinator purpose identifiers

An NSFileCoordinator’s purpose identifier is a string that uniquely identifies the file access that will be done by the NSFileCoordinator. Every NSFileCoordinator has a unique purpose identifier that is created during initialization. Coordinated reads and writes performed by NSFileCoordinators with the same purpose identifier never block each other, even if they originate from different processes. If you are coordinating file access on behalf of an NSFilePresenter, you should use -initWithFilePresenter: and should not attempt to set a custom purpose identifier. Every NSFileCoordinator instance initialized with the same NSFilePresenter will have the same purpose identifier.

When using the NSFileProviderExtension API, you need to supply a ‘providerIdentifier’. Whenever the NSFileProviderExtension needs to do file coordination, it needs to set the NSFileCoordinator’s purpose identifier to the NSFileProviderExtension’s providerIdentifier. Otherwise, the coordination could trigger additional unexpected actions by your NSFileProviderExtension.

If you have multiple subsystems cooperating to perform one high-level operation, and each subsystem performs its own coordinated reads and writes, they should all use the same purpose identifier to avoid blocking on one another.

When creating custom purpose identifiers, you can use a reverse DNS style string, such as "com.mycompany.myapplication.mypurpose", or a UUID string. Nil and zero-length strings are not allowed.

Purpose identifiers can be set only once per NSFileCoordinator. If you attempt to set the purpose identifier of an NSFileCoordinator that you initialized with -initWithFilePresenter: or that you already assigned a purpose identifier, an exception will be thrown.

NSMetadataQuery and NSMetadataItem enhancements for ubiquitous items

When using one of the “ubiquitous” NSMetadataQuery scopes, you are now allowed to pass [NSPredicate predicateWithValue:YES] to easily match all files in the scope. This predicate is now also the default for ubiquitous scopes. An explicit predicate is still required on OS X for non-ubiquitous scopes.

The NSMetadataItemContentTypeKey and NSMetadataItemContentTypeTreeKey attributes are now available to use on NSMetadataItems created by queries with ubiquitous scopes.

The symbols NSMetadataQueryUpdateAddedItemsKey, NSMetadataQueryUpdateChangedItemsKey, and NSMetadataQueryUpdateRemovedItemsKey were originally declared to be available on iOS 7.0. However, the symbols did not exist on iOS until 8.0. The declaration in the header has been updated to reflect this fact.

Updating your iCloud application for OS X 10.10 and iOS 8.0

Prior to OS X 10.10 and iOS 8.0, iCloud files that the system knows about but hasn’t yet downloaded would be represented in an application’s ubiquity container with a special user-visible file. That file would have the same name as the file in iCloud, and could be used to get various metadata about the file. That file could also be moved, renamed, or deleted locally, and those changes would be reflected in the iCloud server state. This is no longer the case on OS X 10.10 or iOS 8.0 SDKs. Instead, unfulfilled files are represented with an invisible file in the same directory, the naming and contents of which are implementation details and subject to change. This change has two implications for applications:

1) It is not recommended to use directory enumeration APIs (e.g. NSDirectoryEnumerator, CFURLEnumerator, readdir, etc.) to enumerate the contents of a ubiquity container. Applications should instead use NSMetadataQuery with NSMetadataQueryUbiquitousDataScope or NSMetadataQueryUbiquitousDocumentsScope, which will list all known data or document files in the container. If you must enumerate the contents of the container, you must ignore any hidden files, or files with extensions your application does not recognize.

2) It is not possible to use the standard APIs for getting file attributes (e.g. NSURL resource values, NSFileManager, stat, getattrlist, etc.) on URLs of unfulfilled files, because there is no file present at the URL until the file has been downloaded. Applications should instead use new APIs designed for this purpose, which are explained below.

If your application already uses NSMetadataQuery to list iCloud files, only gets file attributes from NSMetadataItem, and always uses NSFileCoordinator to access iCloud files, then you shouldn’t need to change anything about your application to allow it to continue working with iCloud on OS X 10.10 and iOS 8.0.

If your application needs to access file metadata directly from the file system for files that may not yet be downloaded, then you can use the following APIs:

1. -[NSURL getPromisedItemResourceValue:forKey:error:] and -[NSURL promisedItemResourceValuesForKeys:error:]

These methods are versions of the existing NSURL APIs that understand how to get attributes from unfulfilled iCloud files. They can be used with any NSURL resource values that are not dependent on file contents being present (for example, NSURLGenerationIdentifierKey, NSURLContentAccessDateKey, and

NSURLFileResourceIdentifierKey).

A coordinated read is not required to invoke this method. If you choose not to, you must be prepared for this method to fail. Listening to NSMetadataQuery or NSFilePresenter notifications will help you know when to retry invoking these methods.

2. NSFileCoordinatorReadingImmediatelyAvailableMetadataOnly and NSFileCoordinatorWritingMetadataOnly

Normally, using NSFileCoordinator to perform a coordinated read will cause an unfulfilled iCloud file to be downloaded before the accessor block is invoked. If you just want to read file metadata, you can avoid waiting for the file to download by using this option in your coordinated read. When using this option you must still use the "PromisedItem" NSURL methods inside your accessor block in order to safely read metadata for files that are unfulfilled.

Similarly, when all you want to do is set some metadata on potentially unfulfilled iCloud files, you must use NSFileCoordinatorWritingContentIndependentMetadataOnly. There are no "PromisedItem" APIs for setting attributes, so you can use normal APIs like -[NSURL setResourceValue:forKey:error:]. As the name indicates, again setting only content independent metadata is supported,

To ease adoption of these new APIs, applications built using the OS X 10.9 or iOS 7.0 SDKs (or earlier) will have all files in their ubiquity container downloaded greedily.

NSFilePresenters for ubiquity container directories will continue to receive -presentedSubitemDidChangeAtURL: and -presentedSubitemAtURL:didMoveToURL: notifications for unfulfilled files. However, your application needs to be aware that for these unfulfilled files, there is nothing in the file system at those URLs, so you need to use NSFileCoordinator or the "PromisedItem" APIs to interact with them.

Finally, NSMetadataQuery objects with ubiquitous scopes will now report directories in the query results. If needed, you can use NSMetadataItemContentTypeKey in your query to filter out anything with the type 'public.folder'.

iCloud Versions

iCloud applications on OS X and iOS can now access previous document versions in iCloud. These versions are automatically created by iCloud when changes are uploaded to the server. To discover these versions, invoke +[NSFileVersion getNonlocalVersionsOfItemAtURL:completionHandler:]. If successful, the completion handler will be passed an array of NSFileVersions.

Versions returned by this API will not initially have the version contents available locally. -[NSFileVersion hasLocalContents] will return NO to indicate this. Accessing these URLs is very much like accessing regular iCloud files. You can use NSURL's "promisedItem" APIs to get metadata about the file, like its NSURLFileSizeKey or NSURLTypeIdentifierKey. In order to trigger the contents to be downloaded, you must use NSFileCoordinator to perform a coordinated read on the NSFileVersion's URL property.

When a version's contents are downloaded, it will be cached in a local NSFileVersion, meaning it will start appearing in the array returned by +[NSFileVersion otherVersionsOfItemAtURL:]. This local NSFileVersion will have a -persistentIdentifier identical that of the NSFileVersion previously returned by +getNonlocalVersionsOfItemAtURL:completionHandler:. You can take advantage of this fact to avoid duplicate versions.

If you want to present all versions available, both local and non-local, for a particular document, you need to do the following things to avoid potential races, which could result in duplicate or missing versions:

1. Perform a coordinated read on the desired file.
2. Register an NSFilePresenter with +[NSFileCoordinator addFilePresenter:]
3. Invoke +[NSFileVersion otherVersionsOfItemAtURL:] to get local versions
4. Invoke +[NSFileVersion getNonlocalVersionOfItemAtURL:completionHandler:] to get non-local versions.
5. In your NSFilePresenter's implementation of -presentedItemDidGainVersion:, look for a version with the same -persistentIdentifier value as the passed-in NSFileVersion (or that matches with -isEqual:) and replace that version with the new one. In your NSFilePresenter's implementation of -presentedItemDidLoseVersion:, look for an NSFileVersion matching the passed-in NSFileVersion, and remove it.

Ubiquitous external documents

If your iOS application uses UIDocumentPickerViewController, your application will be able to access documents from outside of your application's ubiquity container. When a user grants your application access to a file from another application's ubiquity container, your application will be allowed to reopen that document on any device using the same iCloud account without requiring the user to go through the document picker again. These previously opened external ubiquitous documents can be found with NSMetadataQueryAccessibleUbiquitousExternalDocumentsScope.

NSMetadataItem results in this scope will return YES for the NSMetadataUbiquitousItemIsExternalDocumentKey attribute to indicate that they from outside the application's container. Because these documents exist outside your application's sandbox, the NSMetadataItemURLKey attribute of these items is a security-scoped URL. In order to access these documents, you must first invoke -[NSURL startAccessingSecurityScopedResource]. If this method returns YES, then when you're done accessing the document you must invoke -[NSURL stopAccessingSecurityScopedResource].

If your iOS application wants to show external documents in its document picker UI together with other documents, you can use NSMetadataUbiquitousItemContainerDisplayNameKey to show the name of the container the document comes from.

External ubiquitous documents are also represented by a special file on disk. The location of these files is available with NSMetadataUbiquitousItemURLInLocalContainerKey. You may allow users to move these files into other directories within your application's ubiquity container.

iCloud download requested state

You can now use NSURLUbiquitousItemDownloadRequestedKey or NSMetadataQueryUbiquitousItemDownloadRequestedKey to detect whether a download for an iCloud file has already been requested with either a coordinated read or -[NSFileManager startDownloadingUbiquitousItemAtURL:error:]. You can use this state in your UI to show users that the document has been requested and will be delivered as soon as iCloud can deliver it.

NSProcessInfo Operating System Version Checking

NSProcessInfo has a new API:

```
- (BOOL) isOperatingSystemAtLeastVersion:(NSOperatingSystemVersion)version
```

which allows programs to test whether they're on a particular version of the OS or newer. Additionally, there's a new property to get the version of the OS that the application is currently running on:

```
@property (readonly) NSOperatingSystemVersion operatingSystemVersion
```

+[NSLocale autoupdatingCurrentLocale] Bug Fix

Auto-updating NSLocales were not completely toll-free bridged with CFLocaleRef, resulting in crashes when passed to some methods (in particular lowercaseStringWithLocale:). This has been fixed.

-[NSUserDefaults synchronize] Changes

In earlier releases, the various ‘set’ methods on NSUserDefaults would wait several seconds before publishing the changes to other applications unless synchronized (though it has always been instant within a program). This delay has been removed. Additionally, iOS now matches Mac OS X’s behavior of automatically picking up external changes without needing to synchronize.

If you:

- Synchronized to avoid losing data if your app crashed
- Synchronized to read changed values from outside your process
- Synchronized because it made things work and you weren’t sure why

Then please don’t do so now. It should just work, and synchronizing will unnecessarily slow down your program.

If you synchronized because you immediately send out a notification to re-read preferences in another program after setting, and you wanted to be sure it will pick it up, then it may still be worth calling synchronize. You should probably restructure your program to use a real IPC mechanism (such as NSXPCConnection) to send the data to the other process though, rather than indirecting through the preferences system.

NSUserDefaults Plist Files

The on-disk property list files used by NSUserDefaults have always been a private implementation detail, but in previous releases of iOS, and significantly older releases of Mac OS X, directly modifying them has mostly worked (though there are some potential data-loss issues for applications that do so, even on previous systems). In iOS 8 and later, the NSUserDefaults daemon process (cfprefsd) will cache information from these files and asynchronously write to them. This means that directly modifying plist files is unlikely to have the expected results (new settings will not necessarily be read, and may even be overwritten). You should use the NSUserDefaults or CFPreferences APIs, or (on Mac OS X) the defaults(1) command, to interact with the preferences system.

NSUserDefaults Performance Tradeoffs

Reading from NSUserDefaults is extremely fast. It will cache values to avoid reading from the disk, and takes about 0.5 microseconds to do [[NSUserDefaults standardUserDefaults] boolForKey:] on a 2012 MacBook Pro. It’s generally unnecessary (and even undesirable since it prevents picking up new values) to cache the result of reading from preferences.

However, writing to NSUserDefaults is a bit slower. In general, expect it to take roughly as long as using NSPropertyListSerialization to convert your key and value to plist data, plus a few 10s of microseconds. For this reason, as well as memory usage, it’s generally best to store relatively small data in CFPreferences.

As with any and all performance guidelines, use Instruments to profile your application and evaluate the impact of various alternatives.

defaults(1) enhancements

The ‘defaults import’ and ‘defaults export’ commands now support stdin and stdout as targets, by passing ‘-‘ as the path.

NSQualityOfService

Several new Quality of Service (QoS) classifications have been added which map to the new underlying OS concept in OS X 10.10 and iOS 8. They are used to indicate to the system the nature and importance of work, and are used by the system to manage a variety of resources. Higher QoS classes receive more resources than lower ones during resource contention.

NSQualityOfServiceUserInteractive: this QoS is used for work directly involved in providing an interactive UI such as processing events or drawing to the screen.

NSQualityOfServiceUserInitiated: this QoS is used for performing work that has been explicitly requested by the user and for which results must be immediately presented in order to allow for further user interaction. For example, loading an email after a user has selected it in a message list.

NSQualityOfServiceUtility: this QoS is used for performing work which the user is unlikely to be immediately waiting for the results. This work may have been requested by the user or initiated automatically, does not prevent the user from further interaction, often operates at user-visible timescales and may have its progress indicated to the user by a non-modal progress indicator. This work will run in an energy-efficient manner, in deference to higher QoS work when resources are constrained. For example, periodic content updates or bulk file operations such as media import.

NSQualityOfServiceBackground: this QoS is used for work that is not user initiated or visible. In general, a user is unaware that this work is even happening and it will run in the most efficient manner while giving the most deference to higher QoS work. For example, pre-fetching content, search indexing, backups, and syncing of data with external systems.

NSQualityOfServiceDefault: this NSQualityOfService value indicates the absence of QoS information. Whenever possible QoS information will be inferred from other sources. If such inference is not possible, a QoS between UserInitiated and Utility will be used.

NSTask qualityOfService

NSTask has a new qualityOfService property.

You can change the qualityOfService property before launching the task, as much as you like, but it becomes immutable when the task is launched, and the value it has at that time is the value that is used:

- If the property has not been set, you get a default behavior from the OS.
- If the property has been set to NSQualityOfServiceDefault, you get a default behavior from the OS.
- If the property has been set to another value, that value is given to the OS when launching the new process.

NSTasks do not infer any QOS from any execution context, though the underlying OS behaviors might.

NSThread qualityOfService

NSThread has a new qualityOfService property.

You can change the qualityOfService property before starting the thread, as much as you like, but it becomes immutable when the thread is started, and the value it has at that time is the value that is used:

- If the property has not been set, you get a default behavior from the OS.
- If the property has been set to NSQualityOfServiceDefault, you get a default behavior from the OS.
- If the property has been set to another value, that value is given to the OS when starting the new thread.

Reading a thread's qualityOfService after the thread has started will read out its current value.

NSThreads do not infer any QOS from any execution context, though the underlying OS behaviors might.

If the deprecated NSThread threadPriority property is set, then any value that has been set in the qualityOfService property is cleared, as if the qualityOfService property had not been set. If the qualityOfService property is set, the threadPriority property is set back to its default state.

NSOperationQueue qualityOfService

NSOperationQueue has a new qualityOfService property.

You can change the qualityOfService property at any time.

When an operation is added to a queue, the queue's qualityOfService value at that time may affect the effective QOS that the operation will be run at:

- If the queue property has not been set, the operation is unaffected.
- If the queue property has been set to NSQualityOfServiceDefault, the operation is unaffected.
- If the queue property is set to another value, the operation is promoted to the queue's qualityOfService if its promotion QOS is not already at least that level.

If the qualityOfService property of a queue is changed while there are operations in the queue, the effective QOS of operations in the queue will be affected, just as the operations were when added to the queue. Thus, when the qualityOfService property of a queue is changed, all operations in the queue, running or not, have their effective QOS raised up to that level (if they were lower), and future additions to the queue are raised to that level (when they are lower). If the qualityOfService property is lowered from one level to another, only future additions will be affected by that new lower value. Operations' promotion or effective QOS is never lowered by the setting of, or the set value of, a queue's qualityOfService.

When an operation is added to a queue, the operation's effective QOS values at that time may affect the effective QOS of the operations which are already in the queue ("ahead of it"):

- The operations already ahead in the queue of the newly added operation, running or not, are promoted to the effective QOS of the operation being added.

Thus, if a high QOS operation is added to a queue, operations already in the queue are raised up to that level (if they were lower). Operations added after that high-QOS operation are not affected by its presence in the queue.

The meaning and interaction of operation promotion QOS and effective QOS is discussed in the section on NSOperation qualityOfService.

NSOperationQueues do not infer any QOS from any execution context.

If the (dispatch_queue_t) underlyingQueue property of an NSOperationQueue is set, qualityOfService property values of NSOperationQueues and NSOperations have no effect. The effective QOS of operations run by that queue is determined by the state of the dispatch_queue_t.

NSOperation qualityOfService

NSOperation has a new qualityOfService property.

You can change the qualityOfService property at any time.

There are various real and virtual QOS values that relate to how an operation runs:

- The qualityOfService property value
- An inferred QOS
- The promotion QOSes
- The effective QOS

When an operation object is created, an inferred QOS value is computed from the execution context:

- If either:
  - the operation is being created in the execution context of another operation (already running on that thread); or
  - the operation is being created in the execution context of a certain NSProcessInfo API;

then the nearest one of those to the current activation frame of the call stack of the current thread is used as the inferred QOS of the new operation:
  - that operation's effective QOS at the time it started running;
  - the NSProcessInfo API's values are mapped to a QOS value.
- If the operation is being created on the main thread, the inferred QOS is NSQualityOfServiceUserInitiated.
- Otherwise, the current thread's QOS (which may be none) is read and used as the inferred QOS of the new operation.

An operation can be promoted (have promotion QOSes applied to it) in several contexts:

- When the operation is added to a queue, or when the qualityOfService property of the queue the operation is in is changed
  - (as discussed in the NSOperationQueue section)
- When a different operation is added to a queue that the operation (in question) is already in
  - (as discussed in the NSOperationQueue section)
- When a different later (after the operation in question) operation in the same queue has its effective QOS raised
  - the effective QOS of the other operation promotes the operation
- When a different operation (a dependee) becomes dependent upon the operation in question
  - the effective QOS of the dependee promotes the operation
- When a dependee operation has its effective QOS raised
  - the new effective QOS of the dependee promotes the operation
- When the operation is waited upon, with the -waitUntilFinished method, or indirectly when the operation's queue's -waitUntilAllOperationsAreFinished method, is used
  - if the waiting thread is the main thread, the promotion QOS is taken to be NSQualityOfServiceUserInteractive;
  - otherwise if the waiting is done in the execution context of another operation, its effective QOS promotes the operation;
  - otherwise the QOS of the current thread promotes the operation.

These are all collectively called the promotion QOSes; or for the MAX() of all of them, just promotion QOS.

These various values are put together into the effective QOS. The effective QOS is the MAX() of all of these QOS values: {the inferred QOS, the promotion QOSes, the qualityOfService property value}, with these qualifications:

- If the operation's qualityOfService property has been explicitly set to anything, even NSQualityOfServiceDefault, the inferred QOS is ignored.
- If the operation's qualityOfService property has not been explicitly set to anything, it is ignored (as if no value exists).
- All QOS values of NSQualityOfServiceDefault are ignored.
- If there are no QOS values after all that ignoring, the effective QOS is NSQualityOfServiceDefault.

Thus, for example, if an operation is waited upon, its effective QOS may be raised by the waiting context, which may recursively raised all of its dependent operations and all operations ahead of it in the queue (and so on recursively outward in the tree of these relationships).

An operation's qualityOfService property value has no effect if the operation is started manually, rather than being put in an NSOperationQueue, unless the code which is starting it reads out that value and makes some appropriate use of it; that is outside the purview of Foundation.

NSOperation's threadPriority property

The threadPriority property has been deprecated. Move to using the "qualityOfService" property and concepts instead.

NSOperation name

NSOperation has a new name property. The operation's name will show up as part of the name of the underlying dispatch queue that the operation is running on, which shows up in some tools.

NSOperationQueue underlyingQueue

NSOperationQueue has a new underlyingQueue property. When set to, to run operations, the NSOperationQueue will dispatch a block to this dispatch queue which starts the operation, rather than NSOperationQueue making a dispatch queue choice itself.

An exception will be raised if the dispatchQueue is changed while operations are in the operation queue. It is just not advisable to attempt to do that. Set the value once, after creating the NSOperationQueue and before using it otherwise.

There is and will be no way to "find" an NSOperationQueue with a given dispatch queue.

For operations put into an NSOperationQueue with an assigned dispatch_queue_t, the qualityOfService property has no effect; the effective QOS of execution comes from the dispatch_queue_t. The name property of the NSOperationQueue and NSOperation has no effect (the dispatch_queue_t has the name).

Note that this is not a queue which NSOperationQueue will use for its own "thread-safety" serialization; this is only "the dispatch queue to run the operations on".

NSString

NS/CFStrings now use a “tagged pointer” format where the string contents are stored in the pointer directly in some cases. This happens automatically thru the existing APIs that create strings, without need to use any new API.

This change will break any code which treats NS/CFStrings objects as pointers.

In addition, this change enables index and range checking to be performed more often when working with strings. So you may see runtime exceptions due to this change. In almost all cases these exceptions point to bugs in code, so please take them seriously.

There are other related changes in behavior due to this change. For instance, -UTFString and -cStringUsingEncoding: now need to manufacture the buffer that is returned as opposed to possibly returning an internal pointer (which was sometimes possible). The operation has been optimized so there isn’t a significant performance impact, but there is a behavior change nonetheless.

Do not make any assumptions on the kind and size of strings which can fit into the tagged pointer.

This change is enabled only for applications that link against the 10.10 SDK.

NSString now has the following two convenience methods:

```
- (BOOL)containsString:(NSString *)str;
- (BOOL)localizedCaseInsensitiveContainsString:(NSString *)str;
```

containsString: returns YES if the target string is contained within the receiver. This is the same as calling rangeOfString:options: with no options, thus doing a case-sensitive, non-literal search. localizedCaseInsensitiveContainsString: is the case-insensitive variant. Note that it takes the current locale into effect as well. Locale-independent case-insensitive operation, and other needs can be achieved by calling rangeOfString:options:range:locale: directly.

Note that as is the case with the existing rangeOfString: method and variants, these methods will return NO if the target string is the empty string.

NSByteCountFormatter

A bug where non-adaptive, zero-padding formatter displaying byte counts would use decimal places (for instance displaying “12.0 bytes”) has been fixed.

Notes specific to OS X 10.9 and iOS 7

NSXPCConnection

In OS X 10.8 Mountain Lion, if a programmer error caused an exception when a message is received on a connection, the message would be dropped and no further action would be taken. As of OS X 10.9, NSXPCConnection will also automatically invalidate the connection. This ensures that the error handler will be invoked on the calling side with a ‘connection interrupted’ or ‘connection invalidated’ error.
The CPU and memory performance of sending large amounts of data over an NSXPCConnection is significantly increased.
NSXPCConnection will now clear the invalidation handler, interruption handler, and exported object after the connection is invalidated. This helps prevent accidental retain cycles, for example when an invalidation handler block captures the NSXPCConnection instance. NSXPCConnection will also stop retaining any additional exported objects.

NSKeyedArchiver

In previous releases, archiving a string with the exact contents "$null" would cause it to be unarchived as a nil object. Archives with this string created on OS X 10.9 will unarchive correctly on OS X 10.9 and previous releases.
NSKeyedArchiver supports the NSSecureCoding feature added in OS X 10.8 Mountain Lion. The support is available on OS X 10.8 Mountain Lion and later. To use secure coding, create the NSKeyedUnarchiver or NSKeyedArchiver instance and use the new method. This code sample assumes ARC is enabled.

```
NSString *myString = @"Hello world";
NSMutableData *data = [NSMutableData data];
NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc] initForWritingWithMutableData:data];
[archiver setRequiresSecureCoding:YES];
[archiver encodeObject:myString forKey:@"MyKey"];
[archiver finishEncoding];
// ...
NSData *dataToUnarchive = ...;
NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc] initForReadingWithData:dataToUnarchive];
[unarchiver setRequiresSecureCoding:YES];
NSString *decodedString = [unarchiver decodeObjectOfClass:[NSString class] forKey:@"MyKey"];
```

If the class of the decoded object is not an NSString or NSString subclass, then the decodeObjectOfClass:forKey: method will thrown an exception.
In OS X 10.9, Foundation exports a new key: NSKeyedArchiveRootObjectKey. This is the root key used for archives created with the existing NSKeyedArchiver method +archivedDataWithRootObject: on all previous releases.

NSBundle

Although the documentation allows NSBundle and CFBundle to insert new keys in a bundle’s Info.plist, doing so resulted in a common crash when the Info.plist dictionary was read on one thread and modified on another. NSBundle and CFBundle will no longer set any additional keys in the Info.plist dictionary, and applications are highly discouraged from attempting to modify the Info.plist dictionary returned from the -infoDictionary method.

NSOperation and NSOperationQueue

The performance of -addOperationWithBlock:, -addOperation:, starting execution of operations, and starting execution of completion blocks is significantly faster.

NSUserNotification

Calling -removeDeliveredNotification: and -removeAllDeliveredNotifications: will now remove notifications that are displayed.
A new property on NSUserNotification, -identifier, can be used to uniquely identify an individual notification even across separate launches of an application. If a notification is delivered with the same identifier as an existing notification, it will replace the preexisting notification. This can be used to update existing notifications.
NSUserNotification has a new property to specify the image shown in the content of a notification.
NSUserNotification allows for a ‘quick reply.’ The property responsePlaceholder can be used to set a placeholder string and the response can be retrieved from the response property.

App Nap

App Nap is a new feature in OS X 10.9 which focuses system resources like CPU, I/O, and battery energy on the most important work done for the user. The system uses heuristics to determine when an application is doing important work, and when the work it is doing is not critical. These heuristics include (but are not limited to): visibility on screen, drawing activity, event processing, audio playback, foreground vs background, and application type.
When an application enters app nap mode, the system applies up to three kinds of effects: CPU priority lowering, I/O priority lowering, and timer throttling. CPU priority lowering will make an application lower priority than other apps, but not as low as most system daemons. I/O priority lowering will allow I/O to proceed as fast as possible, but allows high priority I/O to go first. This is designed to improve responsiveness in foreground applications that the user is actively interacting with. Timer throttling will reduce the frequency of most kinds of timers in an application. These combined effects provide a significant increase in battery life when an application is performing frequent unnecessary work.
The greatest benefit to the user comes when every application on their system is doing as little work as possible. Therefore, App Nap is an opt-out feature. The user can opt an application out of App Nap manually with a checkbox in the Finder “Get Info...” pane. Developers can temporarily opt an application out by bracketing user-initiated activities with new NSProcessInfo API. Please consider the power usage of your application before opting out of App Nap.

App Nap - User Activities

Applications can help improve the result of the App Nap heuristics by using new API on NSProcessInfo to distinguish user-initiated activities from background or other maintenance work. This API is called automatically by AppKit for user event handling. You should call it when your application begins long-running or asynchronous work.
The new API has two forms. The first form is a begin/end pairing. Call -[NSProcessInfo beginActivityWithOptions:reason:] when your application begins a user initiated activity. The returned object should then passed into -[NSProcessInfo endActivity:] when the activity is finished.
The second form is block-based: -[NSProcessInfo performActivityWithOptions:reason:block:]. With this API, you specify the kind of activity and do the work inside the block. The method will run the block synchronously and automatically begin and end the activity around the block.
The options parameter describes the kind of activity your application is performing. If the work is user initiated, use NSActivityUserInitiated. If the work is background or other maintenance work then use NSActivityBackground. The options can also be used to prevent the system from entering idle system sleep or idle display sleep, with the NSActivityIdleSystemSleepDisabled and NSActivityIdleDisplaySleepDisabled constants. If your app is performing user initiated work that should not prevent the system from idle sleeping, then use NSActivityUserInitiated. You should be careful to choose the right kind of activity any time you use the new API. Preventing the computer from idling and going to sleep may result in an empty battery.
User initiated activities should be limited to work explicitly started by the user. Examples include exporting files or recording audio. Application-initiated activities like performing maintenance should either not use this API or use the NSActivityBackground type. There is new API in XPC for performing regular maintenance activities at more appropriate times (like when connected to A/C power). See <xpc/activity.h>.

App Nap - Eligibility

Applications may be opted out of App Nap automatically based on their application type. Currently, only applications which can become the frontmost app are eligible. This behavior may change in the future. If you have an application which is LSUIElement or LSBackgroundOnly, you can opt your application in to App Nap by setting the NSSupportsAppNap key in your Info.plist to be a Boolean type with a value of YES. A value of NO is ignored for all applications.

If your application shows a user interface, then the system will use the application visibility and other heuristics to move your application in and out of App Nap automatically. If your app does background work at various times but shows no UI, it can use the User Activities API to inform the system when it is appropriate for it to be eligible for App Nap.

App Nap - Debugging

There are many new tools available on the system to investigate the root cause of unexpected battery drain. Xcode now includes a view of the energy usage when running your application. Activity Monitor includes a new column that shows if your app is in App Nap or not, plus a synthesized “power score” which indicates the overall power usage of your app. The battery menu extra will now report apps using an excessive amount of power. The new command line tool timerfires (“man timerfires” for more information) can be used to investigate what timers your application has scheduled.
You should strive to have a power score of 0.0 and 0 idle wake ups. Remember that even a small amount of CPU usage can have a significant effect on battery life.

Timer Tolerance

It is now possible to inform the system of the potential “tolerance” for NSTimer or CFRunLoopTimerRef objects. The tolerance is an allowable delay after the scheduled fire date of a timer. For example, a timer is setup with a fire date of 5 seconds from now, repeating every 7 seconds, and with a tolerance of 3 seconds. The timer may then fire between times 5 to 8s, 12 to 15s, 19 to 22s, 26 to 29s, and so forth.
Adding a tolerance allows the system to schedule timers in a significantly more power-friendly fashion. Most timers should have a tolerance set. A value of at least 10% of the interval is recommended, but the exact value will be application-specific.

Progress Reporting and Cancellation

Mac OS X 10.9 and iOS 7.0 include a new mechanism for progress reporting. It allows code that does work to report the progress of that work, and user interface code to observe that progress so the progress can be presented to the user. Specifically, it can be used to show the user a progress bar and explanatory text, both updated properly as progress is made. It also allows work to be cancelled or paused by the user. This mechanism takes the form of a new class in the Foundation framework named NSProgress. Some design goals of this class were:
• Loose coupling. Code that does work can report the progress of that work regardless of what is observing it, or even whether it is being observed at all. To a lesser degree, code that observes progress and presents it to the user does not have to account for how the code that does the work is structured. Most of this goal is achieved simply by there being a single NSProgress class that can be used by a wide variety of progress reporters and observers.
• Composability. Code that does work can report progress without taking into account whether that work is actually just part of a larger operation whose total progress is what's really interesting to the user. To this end every NSProgress can have a single parent and multiple children. An NSProgress representing the total progress of an operation that's interesting to the user typically has no parent. If there are suboperations then their progress is represented by child NSProgresses. Reports of progress being made propagate from children to parents. Requests for cancellation propagate from parents to children. The subdivision of progress into a tree of NSProgresses enables solutions to problems like how the progress of work performed by disparate pieces of code should be used to calculate one overall progress number worth presenting to the user. For example, see -[NSProgress fractionCompleted], which returns a value that takes into account both the receiver and its children.
• Reusability. NSProgress is meant to be used by virtually all code that can link to the Foundation framework and that makes user-presentable progress. On Mac OS X, NSProgress includes a mechanism for publishing progress in one process and observing the progress in others.
• Usability. In many cases a substantial obstacle to using NSProgress would be arranging for code that does work to find the exact instance of NSProgress it should use to report its progress. The size of this obstacle depends on many things, like how layered your code is (would you have to pass the NSProgress as an argument through many layers of functions and methods?), how it is already being used by multiple projects (can you even add NSProgress parameters without breaking things?), how it is divided between framework and application code (does all of this code ship at the same time?), and so on. To help surmount this obstacle there is a notion of current progress, which is the instance of NSProgress that should be the parent for any new progress objects that represent a subdivision of work. You can set a progress object as the current progress, then call into a framework or other section of code. If it supports progress reporting, it can find the current progress object using the currentProgress method, attach its own children if required, and do its work.
The NSProgress.h header has more detailed information about the methods available.

Reporting Progress

Using NSProgress in simple situations is not too complicated, though getting progress reporting and error handling right at the same time requires some care. Here's an example of doing that in a method that is supposed to be reusable regardless of whether the code that's invoking it will actually present its progress to the user, and even regardless of whether the work it's doing is just part of a larger operation.

```
- (BOOL)readFromData:(NSData *)data error:(NSError **)outError {
    // If there is already current progress, make a child of it to report progress about the part
    // of the work that's being done by this method. In this example we simply use "bytes of input"
    // as the unit of progress.
    NSUInteger length = [data length];
    NSProgress *progress = [NSProgress progressWithTotalUnitCount:length];
    // A loop that does something with each byte of data.
    NSError *error = nil;
    for (NSUInteger index = 0; index < length; index++) {
        // For the most part cancellation is just another kind of error to Cocoa.
        // In real code there might be some cleanup to do here, but it should be just more of the
        // same thing that you would do after any kind of error.
        if ([progress isCancelled]) {
            error = [NSError errorWithDomain:NSCocoaErrorDomain code:NSUserCancelledError userInfo:nil];
            break;
        }
        // Do some work with each byte. This code also might set the error to something, clean up, and break out of the loop.
        // ...
        // Report progress. We add one to the index so that the completed unit count will equal the total unit count at the end
        [progress setCompletedUnitCount:(index + 1)];
    }
    // Finish up the error handling.
    if (error && outError) {
        *outError = error;
    }
    return error ? NO : YES;
}
```

If your method does work asynchronously, then you can create a new child object synchronously, capture it in a block, and update it from another thread or queue. Creating the child object first allows it to be attached to the currentProgress, if set by the caller of the method.

```
- (void)readAsynchronouslyFromData:(NSData *)data withCompletionHandler:(void (^)(NSError *error))completionHandler {
    NSUInteger length = [data length];
    // By creating the child progress before we move the work to another queue, we can
    // capture the currentProgress object if set by the caller of this method.
    NSProgress *progress = [NSProgress progressWithTotalUnitCount:length];
    // We use a pre-existing NSOperationQueue here, but this approach will also apply if you use
    // dispatch_async or another technique to move the work onto another thread or queue.
    NSOperationQueue *queue = _queue;
    [queue addOperationWithBlock:^{
        NSError *error = nil;
        for (NSUInteger index = 0; index < length; index++) {
```

```
            if ([progress isCancelled]) {
                error = [NSError errorWithDomain:NSCocoaErrorDomain code:NSUserCancelledError userInfo:nil];
                break;
            }
            // Do some work with each bytes, as before
            // ...
            // Report progress. We add one to the index so that the completed unit count will equal the total unit count at the
            [progress setCompletedUnitCount:(index + 1)];
        }
        completionHandler(error);
    }];
}
```

Creating a Tree of Progress Objects

NSProgress is designed to promote loose coupling of its objects, to free each section of code from having to concern itself with the implementation details of code that it calls or code that calls it. This is accomplished by using the notion of a current progress to implicitly create a tree of progress objects.
For example, imagine that application code intends to read many data objects using the above methods. Assume that this example method is called when there is no current progress object and an array with 3 objects.

```
- (void)readAllData:(NSArray *)arrayOfData {
    NSProgress *overallProgress = [NSProgress progressWithTotalUnitCount:[arrayOfData count]];
    for (NSUInteger index = 0; index < [arrayOfData count]; index++) {
        [overallProgress becomeCurrentWithPendingUnitCount:1];
        [self readAsynchronouslyFromData:arrayOfData[index] withCompletionHandler:^(NSError *error) {
            // Perform error handling here
        }];
        [overallProgress resignCurrent];
    }
}
```

Because the readAsynchronously... method supports NSProgress, this will create a tree of objects. The overallProgress object will be the parent, and it will have 3 children. As each child NSProgress is updated, it will report its progress to the overallProgress, which will reflect the updates via Key-Value Observing and its fractionCompleted property. When each child asynchronously finishes its work, the overallProgress object will update its own completedUnitCount to include the pendingUnitCount that was assigned to that child. The calling code should not attempt to set completedUnitCount on the overallProgress object manually. If multiple children are attached to a parent progress during one period of becoming current, then the pending unit count will be divided equally among them. This can potentially cause the progress to move backwards (because it can not be known in advance how many children there will be), so if you have the choice, prefer to become current several times as this example method does.
It is likely that some methods you invoke do not yet support NSProgress. The class is designed so that the calling code does not need to know if the called code will create a child or not. For example, let's assume the writeData method in the following example has not yet been updated to use NSProgress:

```
- (void)writeAllData:(NSArray *)arrayOfData {
    NSProgress *overallProgress = [NSProgress progressWithTotalUnitCount:[arrayOfData count]];
    for (NSUInteger index = 0; index < [arrayOfData count]; index++) {
        [overallProgress becomeCurrentWithPendingUnitCount:1];
        [self writeData:arrayOfData[index]];
        [overallProgress resignCurrent];
    }
}
```

In this case, NSProgress will determine that no children have been attached between the time that overallProgress became current and the time it resigned current. In this case NSProgress will simply assume that the pendingUnitCount assigned to the work in between the two calls has been completed and update its own completedUnitCount and fractionCompleted. The caller does not have to manually increment the completed unit count of the overallProgress.

Observing Progress

For the most part, observing progress means creating an NSProgress and adding key-value observers of properties like "indeterminate," "fractionCompleted," and "localizedDescription" to it. These KVO notifications are sent out on the thread that changes the value of the property on the NSProgress object. If your code requires that the notification be posted on the main thread (for example, it updates your user interface in AppKit or UIKit), then it can either change the properties on on the main thread or the controller object can be the observer and assume responsibility for moving the UI work onto the main thread when it receives the KVO notification. This approach allows for maximum flexibility when deciding how to structure your code.
Here is an example of how one might implement a controller method that is invoked when the user chooses to open a file in our application:

```
- (void)openFromFileAtURL:(NSURL *)url {
    // This is a method for some sort of controller object that reads model objects from a file, presenting a progress panel i
    // and presenting an error panel if that fails.
    // Certain properties of the NSProgress object may be interesting to display to the user.
    // This controller object may use Key Value Observing to watch the values of those properties and update an AppKit or UIKi
    // It is important to note that the KVO notifications will be posted on the thread that changes the value.
    // If the UI view must be updated on the main thread, it is your responsibility to move that work to the main thread from
    // your observeValueForKeyPath:ofObject:change:context: implementation.
    // You may have to make a decision about how useful it is to show a progress panel. For example, if the length of the
    // operation is short enough it may make sense not to put up a panel only to immediately dismiss it.
    // That human interface decision is encapsulated in the following fictional method implemented on this controller object.
    [self setUpProgressPanel];
    // The total unit count we use for this operation is arbitrary. What matters is that the counts of units we attribute to su
    _progress = [NSProgress progressWithTotalUnitCount:10];
    // It is important to remember not to block the main thread with the work. Otherwise the progress would not even be visible
    [_concurrentQueue addOperationWithBlock:^(void) {
        // To take advantage of NSData's progress reporting, make our progress the current one before we ask NSData to read the
        // NSData will add a child NSProgress and report its reading progress through that. After a little bit of theoretical c
        // we theoretically determined that in this example it's reasonable to call the actual reading of the file 20% of the w
        // parsing of the file and the creation of objects from it the other 80%. Some files, like those on network drives, tak
        // to read than others though. This is not an exact science.
        NSError *error = nil;
        BOOL didRead = NO;
        [_progress becomeCurrentWithPendingUnitCount:2];
        NSData *data = [NSData dataWithContentsOfURL:url options:0 error:&error];
        [_progress resignCurrent];
        // -readFromData:error:, shown above in "Reporting Progress," does 80% of the work here but its progress reporting is i
        // where it's some other fraction of the work. This approach is most useful when the code involved is spread all over a
        // large code base built for reusability.
        [_progress becomeCurrentWithPendingUnitCount:8];
        didRead = [self readFromData:data error:&error];
        [_progress resignCurrent];
        // Get back on the main thread to do whatever it is this app does with the objects it just created.
        [[NSOperationQueue mainQueue] addOperationWithBlock:^(void) {
            // Success or failure, if our progress panel was presented then tear it down, and clean up.
```

```
            [self tearDownProgressPanel];
            _progress = nil;
            // If there was a failure, you may want to present it to the user.
            if (!didRead) {
                // You never have to tell the user they hit the Cancel button! On Mac OS X, AppKit's error presentation methods
                // its subclasses take care of not doing that for you. On iOS you are responsible for checking the kind of erro
                BOOL errorIsCocoa = [[error domain] isEqualToString:NSCocoaErrorDomain];
                if (!(errorIsCocoa && [error code] == NSUserCancelledError)) {
                    [self presentError:error];
                }
            }
        }];
    }];
}
```

Reporting Progress to Other Processes and Observing Progress in Other Processes

There are a variety of reasons why a process might require the progress of work it's doing to be presented by another process. For example, in Mac OS X 10.9, the progress of Safari downloading a file is presented by Finder, on the file's icon, and by the Dock, on the containing folder's Dock item. To make that sort of thing possible, instances of NSProgress can be published.
The published NSProgress must have a value for NSProgressFileURLKey in its userInfo dictionary. The other process can then subscribe to the progress by invoking +addSubscriberForFileURL:withPublishingHandler:. The first argument is the file URL that the subscribing process wishes to observe and the second argument is a block which is invoked when the publishing process calls -publish. The block has an NSProgress argument, which is a proxy that can be observed like any other NSProgress. The KVO notifications for updates on this NSProgress proxy are sent on the main thread. The block passed to +addSubscriberForFileURL:withPublishingHander: and the block it returns (for use when the progress is unpublished) are also invoked on the main thread.
Not every process in a running OS X system is necessarily running with the same localization, so care must be taken that text about progress is localized in the process that will present it to the user. To help with this, NSProgress encapsulates the creation of text describing the progress made, formatted and localized such that it is suitable for presenting to the user. See the localizedDescription, localizedAdditionalDescription, and kind properties.

NSProgresses Have User Info Dictionaries

NSProgress follows the pattern established by NSError, in which all localized text generated by the Foundation framework is derived, at least in part, from values found in a user info dictionary attached to each instance. See user info keys like NSProgressFileOperationKindKey. Entries in the user info dictionary also affect how an NSProgress is published, so see NSProgressFileURLKey for example too.
You can put entries in an NSProgress' user info dictionary for your own purposes. If the values are property list objects then they will be preserved when an NSProgress published by one process is observed by another.

NSMetadata

NSMetadata now supports additional attributes and scopes as well as API that was formerly available only through the lower level MDQuery API. Availability of these new symbols and methods is indicated in the corresponding header.
Attributes that are defined in the MDQuery API are now also available as equivalents in NSMetadata. For example, NSMetadataItemRecipientAddressesKey is equivalent to kMDItemRecipientAddressesKey. The full set of attributes can be seen in the new header NSMetadataAttributes.h, which is included by NSMetadata.h. The string value of these symbols is identical to the lower level MDQuery equivalents.
Two additional scopes are available to complement the existing ones by limiting the scope to volumes which are indexed.
NSMetadataQueryIndexedLocalComputerScope represents all indexed locally mounted volumes plus the user's home even if remote.
NSMetadataQueryIndexedNetworkScope represents all indexed user-mounted remote volumes.
Additionally, NSMetadataQuery now supports scoping a search to an existing array of items, which can be specified as an arbitrary mixture of NSURLs and NSMetadataItems. -setSearchItems: is the setter, and -searchItems is the getter. The getter returns the same mixture as was set.
NSMetadataQuery now supports setting an operation queue on which query result notifications will occur. This makes it easier to synchronize query result processing with other related operations, such as updating the data model, and decouples it from the thread used to execute the query. -setOperationQueue: is the setter, and -operationQueue is the getter.
NSMetadataItem can now be created from a NSURL using -initWithURL:. This makes it easier to get NSMetadata attributes on arbitrary URLs without resorting to the lower-level APIs.
NSMetadataQuery now provides a simpler way of enumerating results using a block that automatically disables the query at the start of the iteration and re-enables it upon completion. -enumerateResultsUsingBlock: provides basic iteration, and -enumerateResultsWithOptions:usingBlock: provides additional options for concurrent or reverse iteration.
NSMetadataQuery notifications now provide a notification info dictionary. The new keys NSMetadataQueryUpdateAddedItemsKey, NSMetadataQueryUpdateChangedItemsKey, and NSMetadataQueryUpdateRemovedItemsKey provide access to corresponding arrays of NSMetadataItems that indicate which items have been added, changed, or removed since the last update.

Bug Fixes in NSFilePresenter Messaging

File coordination allows you to register an NSFilePresenter for any file or directory, even one inside a file package. For example, when you double-click on an embedded image in TextEdit to open it in Preview, Preview registers an NSFilePresenter for the image file that is inside the TextEdit .rtfd file package. In this example you might expect that when TextEdit does a coordinated write of the file package as a whole, which it does during document saving, that Preview's NSFilePresenter would be sent a -relinquishPresentedItemToWriter:. Since file coordination's introduction in Mac OS 10.7 this has not been the case. This bug has been fixed in Mac OS 10.9.
An NSFilePresenter may signal an error when invoking the completion handler passed to -savePresentedItemChangesWithCompletionHandler: or -accommodatePresentedItemDeletionWithCompletionHandler:. Since file coordination’s introduction in Mac OS 10.7, however, doing so would often cause file coordination to not invoke the reacquisition block provided by your NSFilePresenter in its response to a previous invocation of -relinquishPresentedItemToReader: or -relinquishPresentedItemToWriter: for the same coordinated reading or writing. This bug has been fixed in Mac OS 10.9. Now your NSFilePresenter’s reacquisition blocks are always invoked, even when your NSFilePresenter has signaled a failure to save changes or accommodate deletion.

Bug Fix in Key Value Observing (KVO)

Mac OS 10.8 and earlier had a bug in which KVO could send an -observeValueForKeyPath:ofObject:change:context: to an observer after it was no longer an observer, and in some cases even after the observer was deallocated. This could happen, for example, if an object had multiple observers of key paths, and the implementation of -observeValueForKeyPath:ofObject:change:context: for one observer attempted to removed one of the other observers. The removal would not take effect immediately enough to prevent the messaging of the other observer's zombie. This was possible even when just one thread was involved. This bug has been fixed in Mac OS 10.9.

NSURL

The "file" URL scheme is defined so that no authority, an empty host, and "localhost" all mean the end-user's machine. To reduce memory use, file URL objects created with file system paths or from file system representation no longer include the host string "localhost".
This change also means -host no longer returns the string "localhost" for file URLs and will instead return nil.
-initFileURLWithFileSystemRepresentation:isDirectory:relativeToURL:, +fileURLWithFileSystemRepresentation:isDirectory:relativeToURL:, -getFileSystemRepresentation:maxLength:. and -fileSystemRepresentation were added to allow easy conversion from file system representation to URL and URL to file system representation.
-removeCachedResourceValueForKey:, and -removeAllCachedResourceValues were added to allow removal of cached resource values from a URL object. -setTemporaryResourceValue:forKey: provides a way to set temporary resource values on a URL object.

URL objects created from URL strings where the URL string length was exactly 1 can no longer be created with characters not allowed in URL strings. This change affects CFURL's CFURLCreateWithString(), and the NSURL methods -initWithString:, -initWithString:relativeToURL:, +URLWithString:, and +URLWithString:relativeToURL:.
Added -stringByAddingPercentEncodingWithAllowedCharacters: and -stringByRemovingPercentEncoding. -stringByAddingPercentEncodingWithAllowedCharacters: is intended to percent-encode an URL component or subcomponent string, NOT the entire URL string. The predefined NSCharacters sets returned by +URLUserAllowedCharacterSet, +URLPasswordAllowedCharacterSet, +URLHostAllowedCharacterSet, +URLPathAllowedCharacterSet, +URLQueryAllowedCharacterSet, and +URLFragmentAllowedCharacterSet are intended to be passed to -stringByAddingPercentEncodingWithAllowedCharacters:.
The descriptions of long URLs with the data scheme may be truncated if the data URL is very large.

NSURLComponents

NSURLComponents is a new class which encapsulates the components of a URL in an object-oriented manner. When a NSURLComponents is created from a NSURL or URL string, it provides IETF STD 66 (rfc3986) parsing of the URL string. When it is used to create a NSURL by providing the components and subcomponents which make up a URL (scheme, host, port, path, query and so on), the URL is created using IETF STD 66 (rfc3986) rules. NSURLComponents makes it easy to correctly parse an existing NSURL or URL string, makes it easy to create a correctly formed NSURL from URL components and makes it easy to create a new NSURL from an existing URL with modifications.

Custom NSData Deallocator for No-Copy Buffers

NSData has long provided the ability to create instances that use a buffer provided by you, instead of copying its contents to an internally created buffer. The APIs to do this are -[NSData initWithBytesNoCopy:length:] and -[NSData initWithBytesNoCopy:length:freeWhenDone:]. One limitation of these APIs is that if you wish to defer deallocation of the no-copy buffer to the NSData instance, then you must use malloc(3) (or one if its variants), because NSData will use free(3).
To eliminate this limitation, NSData has added an API In OS X 10.9 that allows you to specify a custom method of deallocation via a block parameter. That API is -[NSData initWithBytesNoCopy:length:deallocator:].
There are some important things to be aware of when using this API. NSData will, when necessary, copy the deallocator block onto the heap. As a byproduct, any Objective-C object pointers that are captured in that block will be retained. In order to avoid any inadvertent retain cycles, you should avoid capturing pointers to any objects that may in turn retain the NSData object, including explicit references to self and implicit ones via instance variable referencing. To assist in this, the deallocator block has two parameters, the buffer's pointer its length. You should always use these values instead of trying to use references from outside the block.

NSData API for Base 64 Encoding and Decoding

NSData has added an API in OS X 10.9 for encoding and decoding Base 64. The encoding algorithm can be configured to automatically insert line breaks with customizable line ending characters. The decoding algorithm by default will reject the entire input if it encounters any non-Base 64 characters (including whitespace), but there is an option available to make it more lenient. Depending on your needs, you can use one of the two variants of the encoding and decoding methods. One pair uses Base 64 encoded NSStrings and the other uses Base 64 encoded NSDatas.
If your application needs to target an operating system prior to OS X 10.9, you can use NSData's -initWithBase64Encoding: and -base64Encoding instead. These methods have existed since OS X 10.6, but were not exposed until OS X 10.9. These methods behave like the new methods, except -initWithBase64Encoding: will ignore unknown characters. When your application no longer needs to target an operation system prior to OS X 10.9, you should transition to the new methods.

NSPurgeableData Thread Safety

Prior to OS X 10.9, NSPurgeableData was not fully thread-safe. If multiple threads were to invoke certain NSPurgeableData methods simultaneously, an exception or a crash could result. This has been resolved on OS X 10.9.

NSString -stringByAppendingPathExtension: Behavior Change

Prior to OS X 10.9, if the receiver of NSString's -stringByAppendingPathExtension: method started with a tilde (~) character, the method would silently fail to append the extension and return the string unmodified. This behavior has been changed for applications linked against the 10.9 SDK to allow this method to behave as expected for home-relative paths (e.g. "~/user/path/to/file") and terminal file names (e.g. "~tempfile").

NSFileProtectionKey Bug Fixed

Prior to iOS 7.0, when the device was locked, -[NSFileManager attributesOfItemAtPath:error:] would return a dictionary without NSFileProtectionKey. This has been fixed on iOS 7.0 so that the returned dictionary will report the proper NSFileProtectionKey value.

NSFileManager Delegate Method Bug Fixed

Prior to OS X 10.9, delegate implementations of the NSFileManagerDelegate method -fileManager:shouldProceedAfterError:removingItemAtURL: would never be invoked. This has been fixed on OS X 10.9. To work around the problem, you can use the -fileManager:shouldProceedAfterError:removingItemAtPath: method instead. If implemented, that method will be invoked regardless of whether you invoked -removeItemAtPath:error: or -removeItemAtURL:error:.

NSData Copying Behavior Change

Prior to OS X 10.9 and iOS 7.0, the +[NSData initWithData:] method would always return a new NSData instance when the existing NSData object was immutable (and not a custom subclass). Additionally, -[NSData copyWithZone:] would always retain the receiver under the same circumstances. These behaviors have changed for applications linked against the OS X 10.9 or iOS 7.0 SDKs.
For applications linked against the OS X 10.9 or iOS 7.0 SDKs, +dataWithData: now calls -copyWithZone: for immutable objects, allowing it to avoid unnecessary memory allocations. In addition, -[NSData copyWithZone:], when sent to an immutable object, will create a new NSData instance if the receiver was created with a "NoCopy" initializer and does not own its bytes buffer (meaning freeWhenDone was NO, or the deallocator block was nil). This latter change helps to avoid problems where an NSData object created via -copyWithZone: could end up with an invalid bytes pointer.
If you need to create a new NSData object that contains a copy of the contents of another NSData object, the only supported and guaranteed way to do ensure this has been to create it explicitly with something like [NSData dataWithBytes:[otherData bytes] length:[otherData length]].

NSUserDefaults support for Security Application Groups (Sandboxing)

For applications that are part of a Security Application Group, the NSUserDefaults "suite" APIs (-initWithSuiteName:, -addSuiteNamed: and -removeSuiteNamed:) will operate on a suite shared by applications in the group and stored in the group container, if the suite identifier is the identifier of the group.

Changes to NSUserDefaults "compatibility" keys (Mac)

The deprecated constant user defaults keys at the bottom of NSUserDefaults.h are no longer functional in applications built with the 10.9 or later SDK.

Improved safety of NSUserDefaults values

Validation that values set in NSUserDefaults are valid Property List types has been made more thorough, and easier to debug. Crash logs from failing to do this will now include additional information noting the problem.

Collections returned from NSUserDefaults have always been documented to be immutable, but for applications built with the 10.9 or later SDK, actually are now. Attempting to mutate these collections on earlier releases led to extremely hard to diagnose bugs.
Nonsensical parameters to "Suite" methods on NSUserDefaults will now be rejected with a logged message. Currently the cases rejected are NSGlobalDomain and the bundle identifier of the main bundle.

-[NSUserDefaults synchronize] and non-current-application domains (Mac)

In earlier releases, -[NSUserDefaults synchronize] synchronized only the current application's search list. In 10.9 and later it also synchronizes any other domains with un-synchronized changes.

-[NSUserDefaults synchronize] is not generally useful (Mac)

You should only need to call -synchronize if a separate application will be reading the default that you just set, or if a process that does not use AppKit is terminating. In most applications neither of these should ever occur, and -synchronize should not be called. Note that prior to Mac OS X 10.8.4 there was a bug that caused AppKit to automatically synchronize slightly prematurely during application termination, so preferences set in response to windows closing while the application is terminating might not be saved; this has been fixed.

dispatch_data_t -> NSData bridging

In 64-bit apps using either manual retain/release or ARC, dispatch_data_t can now be freely cast to NSData *, though not vice versa. Note that one implication of this is that NSData objects created by Cocoa may now contain several discontiguous pieces of data. You can efficiently work with discontiguous ranges of data by using the new

```
- (void) enumerateByteRangesUsingBlock:(void (^)(const void *bytes, NSRange byteRange, BOOL *stop))block
```

API on NSData. This will be roughly the same speed as -bytes on a contiguous NSData, but avoid allocation and copying for a discontiguous one. Once a discontiguous NSData is compacted to a contiguous one (generally by calling -bytes, other NSData API will do discontiguous accesses), future accesses to the contiguous region will not require additional copying.
Various system APIs (in particular NSFileHandle) have been updated to use discontiguous data for improved performance, so it's best to structure your code to handle it unless it absolutely needs contiguous bytes.

Empty NSData objects

Immutable NSData objects with a length of zero are now all a single object. This shouldn't matter for most applications, but relying on the == operator or methods like -indexOfObjectIdenticalTo: to distinguish between empty data objects will no longer work.

Immutable NSData objects may be vm-backed

Starting in 10.9, NSData may choose to allocate its backing store with vm_allocate, rather than malloc. This should not impact behavior at all, but may be useful to know when examining your program with memory analysis tools (vmmap, Instruments, etc...). This also means that transferring NSData objects over NSXPCConnection can remap the pages rather than copying them, which can improve performance.

NSData -copy, -copyWithZone: and -initWithData:

Methods that create copies of NSData have been improved to simply retain if it's safe to do so in more cases. This shouldn't matter for most applications, but relying on the == operator or methods like -indexOfObjectIdenticalTo: to distinguish between copied data objects will be less likely to work than in the past.

plutil(1) plist editing

plutil(1) has four additional verbs: insert, replace, extract, and remove, which can be used to manipulate plist files. Run 'plutil -help' for more information.

defaults(1) additions for dealing with arbitrary plist files

defaults(1) has two additional verbs: import and export. These can be used to add the contents of a property list file to a defaults domain, or save the contents of a defaults domain as a property list file.

NSCalendar performance and thread-safety

Prior to 10.9, NSCalendar was not thread-safe, which required the +currentCalendar and -copy/-copyWithZone: methods to make time consuming full copies. This has been improved so that expensive copy operations will be deferred until the object is mutated. If you can avoid calling setter methods on NSCalendar instances, you'll likely get better performance.

NSTask performance and thread-safety

NSTask is now thread-safe, and significantly more efficient, especially when launching tasks from host processes with very large VSIZEs and/or unusual memory mappings.

NSDateFormatter and NSNumberFormatter thread-safety

When set to the modern formatter behavior (NSNumberFormatterBehavior10_4), NSNumberFormatter instances are now thread-safe. When set to the modern formatter behavior and using the non-fragile ABI (64 bit applications, on OSX), NSDateFormatter instances are also thread-safe.

NSNetServices

A new includesPeerToPeer property was added to NSNetService and NSNetServiceBrowser to enable Bonjour discovery and connectivity over peer-to-peer Wi-Fi and Bluetooth.
A new NSNetServiceListenForConnections option was added to -publishWithOptions: that allows a published NSNetService to listen for incoming TCP connections. New connections are delivered in the form of NSStreams via the new -netService:didAcceptConnectionWithInputStream:outputStream: delegate method.

NSURLSession

NSURLSession is a replacement API for NSURLConnection. It supports out-of-process downloads and uploads that notify your app on completion. In addition, it provides a new way to configure your networking requests (NSURLRequest) so that they don't conflict with other networking code in your application.

NSURLCredential

Use the new NSURLCredentialPersistenceSynchronizable persistence policy flag to sync the credential across devices through iCloud.
The new -[NSURLCredentialStorage removeCredential:forProtectionSpace:options:] API can be used to remove a credential that has a persistence policy

of NSURLCredentialPersistenceSynchronizable. If the passed in NSURLCredential object has a persistence policy of NSURLCredentialPersistenceSynchronizable, and if the options dictionary is present and includes the NSURLCredentialStorageRemoveSynchronizableCredentials key set to YES, the remove will attempt to delete the credential from iCloud, which will result in the credential being removed from all synchronized devices. If the passed in NSURLCredential object has a persistence policy of NSURLCredentialPersistenceSynchronizable and no options dictionary is provided,
or NSURLCredentialStorageRemoveSynchronizableCredentials does exist but is set to NO, then the call will fail. Passing in an NSURLCredential with a persistence policy other than NSURLCredentialPersistenceSynchronizable is the same as calling the existing -[NSURLCredentialStorage removeCredential:forProtectionSpace:] method.

NSPredicate, NSExpression, and NSSortDescriptor

NSPredicate, NSExpression, and NSSortDescriptor now support NSSecureCoding.
While it is safe to unarchive these objects using NSSecureCoding, it is not safe to blindly evaluate anything you get out of the archive, and as such, evaluation of securely decoded objects will be disabled. Any process receiving predicates/expressions/sortDescriptors should preflight the content of the archive by validating keypaths, selectors, etc contained within it to ensure no erroneous or malicious code will be executed. Once preflighting has been done, evaluation can be enabled by invoking allowEvaluation on the decoded object to recursively enable evaluation on the object graph rooted at the receiver.

NSCalendar

In OS X 10.9, Foundation provides many new APIs to simplify calendrical calculations. This also helps to avoid mistakes from complex calendrical calculations. The new APIs are:

```
+ (id)calendarWithIdentifier:(NSString *)calendarIdentifierConstant;
- (void)getEra:(out NSInteger *)eraValuePointer year:(out NSInteger *)yearValuePointer
          month:(out NSInteger *)monthValuePointer day:(out NSInteger *)dayValuePointer fromDate:(NSDate *)date;
- (void)getEra:(out NSInteger *)eraValuePointer yearForWeekOfYear:(out NSInteger *)yearValuePointer
          weekOfYear:(out NSInteger *)weekValuePointer weekday:(out NSInteger *)weekdayValuePointer fromDate:(NSDate *)date;
- (void)getHour:(out NSInteger *)hourValuePointer minute:(out NSInteger *)minuteValuePointer second:(out NSInteger *)secondVal
          nanosecond:(out NSInteger *)nanosecondValuePointer fromDate:(NSDate *)date;
- (NSInteger)component:(NSCalendarUnit)unit fromDate:(NSDate *)date;
- (NSDate *)dateWithEra:(NSInteger)eraValue year:(NSInteger)yearValue month:(NSInteger)monthValue day:(NSInteger)dayValue
          hour:(NSInteger)hourValue minute:(NSInteger)minuteValue second:(NSInteger)secondValue nanosecond:(NSInteger)nanosec
- (NSDate *)dateWithEra:(NSInteger)eraValue yearForWeekOfYear:(NSInteger)yearValue weekOfYear:(NSInteger)weekValue
          weekday:(NSInteger)weekdayValue hour:(NSInteger)hourValue minute:(NSInteger)minuteValue second:(NSInteger)secondVal
- (NSDate *)startOfDayForDate:(NSDate *)date;
- (NSDateComponents *)componentsInTimeZone:(NSTimeZone *)timezone fromDate:(NSDate *)date;
- (NSComparisonResult)compareDate:(NSDate *)date1 toDate:(NSDate *)date2 toUnitGranularity:(NSCalendarUnit)unit;
- (BOOL)isDate:(NSDate *)date1 equalToDate:(NSDate *)date2 toUnitGranularity:(NSCalendarUnit)unit;
- (BOOL)isDate:(NSDate *)date1 inSameDayAsDate:(NSDate *)date2;
- (BOOL)isDateInToday:(NSDate *)date;
- (BOOL)isDateInYesterday:(NSDate *)date;
- (BOOL)isDateInTomorrow:(NSDate *)date;
- (BOOL)isDateInWeekend:(NSDate *)date;
- (BOOL)rangeOfWeekendStartDate:(out NSDate **)datep interval:(out NSTimeInterval *)tip containingDate:(NSDate *)date;
- (BOOL)nextWeekendStartDate:(out NSDate **)datep interval:(out NSTimeInterval *)tip options:(NSCalendarOptions)options afterD
- (NSDateComponents *)components:(NSCalendarUnit)unitFlags fromDateComponents:(NSDateComponents *)startingDateComp
          toDateComponents:(NSDateComponents *)resultDateComp options:(NSCalendarOptions)options;
- (NSDate *)dateByAddingUnit:(NSCalendarUnit)unit value:(NSInteger)value toDate:(NSDate *)date options:(NSCalendarOptions)opti
- (void)enumerateDatesStartingAfterDate:(NSDate *)start matchingComponents:(NSDateComponents *)comps options:(NSCalendarOption
          usingBlock:(void (^)(NSDate *date, BOOL exactMatch, BOOL *stop))block;
- (NSDate *)nextDateAfterDate:(NSDate *)date matchingComponents:(NSDateComponents *)comps options:(NSCalendarOptions)options;
- (NSDate *)nextDateAfterDate:(NSDate *)date matchingUnit:(NSCalendarUnit)unit value:(NSInteger)value options:(NSCalendarOptio
- (NSDate *)nextDateAfterDate:(NSDate *)date matchingHour:(NSInteger)hourValue minute:(NSInteger)minuteValue second:(NSInteger
- (NSDate *)dateBySettingUnit:(NSCalendarUnit)unit value:(NSInteger)v toDate:(NSDate *)date options:(NSCalendarOptions)opts;
- (NSDate *)dateBySettingHour:(NSInteger)h minute:(NSInteger)m second:(NSInteger)s toDate:(NSDate *)date options:(NSCalendarOp
```

More details about these new APIs can be found in NSCalendar.h. The Date and Time Programming Guide also has more detail discussion about how these APIs would be used.
In addition to the new APIs, Foundation also exports a new notification: NSCalendarDayChangedNotification. This new notification is posted through [NSNotificationCenter defaultCenter] when the system day changes. Note that the notification is not posted to observers on any particular or defined thread or queue, so if you need the handling of the notification to do things that must only be done by a specific thread or queue, you need to then schedule that work in response to the notification.
CoreFoundation and Foundation are going to deprecate several existing calendar-related APIs in the next release. If the preprocessor macro NS_ENABLE_CALENDAR_DEPRECATIONS is set to a non-zero value in a project, then usage of those APIs will be indicated by the compiler.

NSDateComponents

NSDateComponents has four new APIs for convenience:

```
- (void)setValue:(NSInteger)value forComponent:(NSCalendarUnit)unit;
- (NSInteger)valueForComponent:(NSCalendarUnit)unit;
- (BOOL)isValidDate;
- (BOOL)isValidDateInCalendar:(NSCalendar *)calendar;
```

More details about these new APIs can be found in NSCalendar.h.

instancetype

Many new core APIs of Foundation use the 'instancetype' return type. These are class methods which return an instance of the receiving class, even if the receiving class is a subclass of the class which declares the method. Typically these methods would have returned 'id' in the past.
This information can help the compiler identify basic kinds of errors, like the NSArray type in the following line:

```
NSArray *set = [NSSet setWithObject:@"A String"];
```

But conversely, note that just because a method is a class method and returns an instance of the class, it does not necessarily follow that it will return an instance of an arbitrary subclass if the method is sent to the subclass. Thus, not all class methods which return instances are necessarily true candidates for the 'instancetype' return value.

NSArray -firstObject

The -firstObject method returns the first object in the array, or nil if the array is empty. This method is available back to OS X 10.6 and iOS 4.0.

NSString Errors

NSString out-of-bounds index and ranges now generate better error messages.
Formatting localized numbers with NSStrings using the "+" flag will now will no longer display an extra "+" for negative numbers. However, the "+" is not localized.

NSScanner

NSScanner now provides the following API to scan unsigned long long values:
- (BOOL)scanUnsignedLongLong:(unsigned long long *)value;
This works exactly like other existing NSScanner methods, including the caveats:
- it doesn't scan localized numbers or numbers with thousands separators
- returns YES on overflow, with a clamped result

NSAttributedString

For applications linked against the 10.9 SDK or later, copying a standard instance of NSAttributedString now retains it instead.

Localized Property List File

Many languages including English have grammatical rules for selecting a word form depending on contextual conditions such as the gender or plurality. For example, it is common to manually implement the logic for selecting singular or plural forms in formatting localized strings.

```
NSString *localizedString;
NSUInteger numberOfItems; // assume the number of selected item is stored
if (numberOfItems == 1) { // single item
    localizedString = NSLocalizedString(@"A file is selected", @"Message displayed when showing a single file selection.");
} else { // multiple item
    localizedString = [NSString localizedStringWithFormat:NSLocalizedString(@"%d files are selected", @"Message displayed when
}
```

The rule gets more complex when dealing with languages requiring the gender-based form selection and/or plural form condition. For languages such as Russian and Arabic with multiple plural forms, properly formatting by conditional logic like above becomes extremely hard to maintain.
To show the complexity, the following code snippet illustrates what would happen to support the Arabic plural logic for the same message selection logic shown above.

```
NSString *localizedString;
NSUInteger numberOfItems; // assume the number of selected item is stored
NSUInteger modValue = numberOfItems % 100;
if (numberOfItems == 0) { // no item
    localizedString = NSLocalizedString(@"No file is selected", @"Message displayed when showing no selection.");
} else if (numberOfItems == 1) { // single item
    localizedString = NSLocalizedString(@"A file is selected", @"Message displayed when showing a single file selection.");
} else if (numberOfItems == 2) { // dual items
    localizedString = NSLocalizedString(@"Two files are selected", @"Message displayed when showing two files selected.");
} else if ((modValue >= 3) && (modValue <= 10)) { // paucal items
    localizedString = [NSString localizedStringWithFormat:NSLocalizedString(@"A few files (%d items) are selected", @"Message
} else { // multiple items
    localizedString = [NSString localizedStringWithFormat:NSLocalizedString(@"%d files are selected", @"Message displayed when
}
```

Further complicating the issue, the logic for choosing from multiple plural forms varies from locale to locale.
In order to solve this issue, the localizers need to be able to specify more sophisticated data structure than the simple key-value pair from the strings file format. There is a new file format with ".stringsdict" suffix introduced. -[NSBundle localizedStringForKey:value:table:] now accesses two files: .stringsdict and .strings. It queries the .stringsdict file first, then, .strings file. Note that if a .stringsdict file is provided, a .strings file with the same name also needs to be there, even if empty.
The .stringsdict file contains the original key (i.e. @"%d files are selected") with the value as an arbitrary property list (mostly dictionary or string). When the value is a dictionary, it must contain a key for the localized format string value and its format specifier configuration dictionaries.
When -localizedStringForKey:value:table: returns an NSString instantiated from an entry in a .stringsdict file, the string object carries the additional information stored in the file. The information can be retained and copied across -copy and -mutableCopy. It gets discarded when a mutable copy gets mutated.

NSString %@ format specifier enhancement

For representing a new dictionary based formatting rule, %@ format directive is enhanced with the alternate form flag '#'.
The extended %@ format syntax is:

```
%[n$]#[FLAGS][FIELD WIDTH][.PRECISION]@<CONFIGURATION KEY>@
```

A configuration key between two @s is used to query an item in the external format configuration dictionary. Only characters in [a-zA-Z_0-9] are allowed. The name space is independent for each format string.
For example, @"%d files are selected" can become @"%#@num_files_are@ selected" referencing an item for "num_files_are" in the external format configuration dictionary.

The format specifier configuration dictionary

The format specifier configuration dictionary contains key-value pairs of configuration data. Each value is represented by itself a dictionary.
There are a couple of dictionary keys predefined.
- NSStringFormatSpecTypeKey
This key defines the type of format specifier configuration. This is a mandatory key. Currently we're supporting NSStringPluralRuleType and NSStringGenderRuleType. The rest of dictionary contents is determined by the type.
- NSStringFormatValueTypeKey
This is an optional key describing the original argument type on the stack. The value is the original format specifier. For mapping @"%d files are selected" to @"%#@num_files_are@ selected", the configuration dictionary for num_files_are should contain an entry "NSStringFormatValueTypeKey = d". If absent, %@ is assumed.
The NSStringPluralRuleType configuration maps the argument number into a plural choice using the Unicode CLDR Plural mapping rule. The configuration dictionary could contain keys for "zero", "one", "two", "few", "many", and "others". They correspond to the Unicode CLDR Plural supplement data items. The value for "others" is mandatory. If "zero" is present, the value is used for mapping the argument value zero regardless of what CLDR rule specifies for the numeric value. If a mapping key is absent, the value for "others" is used as the fallback.
The NSStringGenderRuleType configuration maps the argument number to a variant form. As with most other gender variation handling systems including CLDR, we're assigning index numbers to genders. For example, male is 0, female is 1, and neutral is 2. Since the number of gender types varies greatly from languages to languages (i.e. Polish has 5). The key for this configuration info type is represented by a positive integer number. The key 0 is mandatory and used for absence of other gender keys.
The mapped value itself can be a format string. The format string is evaluated with the value used to select it.
Sample configuration information for the selected file example.

```
@"%d files are selected" = @"%#@num_files_are@ selected" // localized string dynamically substitute "%d files are"
```

The configuration dict is:

```
{
    "NSStringFormatSpecTypeKey" = "NSStringPluralRuleType"; // plural type
    "NSStringFormatValueTypeKey" = "d"; // int argument
    "zero" = "No file is";
    "one" = "A file is";
```

```
    "other" = "%d files are";
}
```

Since there are cases where the condition of a plural/gender rule substitution is depending on another substitution (i.e. French articles like le, la, and les), the recursive formatting capability is essential for this feature. For example, with @"%d in %d files are selected", giving localizers options to substitute the whole sentence depending on the number of total files is necessary. We're allowing the recursive formatting by applying the entire argument list to each substituted format specifier.

```
@"%d in %d files are selected" = @"%2$#@d_in_d_files_are_selected@"
```

The configuration dictionary can contain

```
"d_in_d_files_are_selected" = {
    "NSStringFormatSpecTypeKey" = "NSStringPluralRuleType"; // plural type
    "NSStringFormatValueTypeKey" = "d"; // int argument
    "zero" = "There is no file";
    "one" = "There is a file, and %1$#@it_is_selected@";
    "other" = "%1$d in %2$d files are selected";
};
"it_is_selected" = {
    "NSStringFormatSpecTypeKey" = "NSStringPluralRuleType"; // plural type
    "NSStringFormatValueTypeKey" = "d"; // int argument
    "zero" = "it is not selected";
    "other" = "it is selected";
};
```

The results are "There is no file" for 0 files, "there is a file, and it is not selected" for 0 in 1, and "3 in 5 files are selected" for 3 in 5.
This is a sample stringsdict contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>%d files are selected</key>
    <dict>
        <key>NSStringLocalizedFormatKey</key>
        <string>%#@num_files_are@ selected</string>
        <key>num_files_are</key>
        <dict>
            <key>NSStringFormatSpecTypeKey</key>
            <string>NSStringPluralRuleType</string>
            <key>NSStringFormatValueTypeKey</key>
            <string>d</string>
            <key>zero</key>
            <string>No file is</string>
            <key>one</key>
            <string>A file is</string>
            <key>other</key>
            <string>%d files are</string>
        </dict>
    </dict>
</dict>
</plist>
```

Notes specific to OS X 10.8 and iOS 6

NSPointerFunctions zeroing weak changes

The NSPointerFunctionsZeroingWeakMemory option has been deprecated. This option was for holding zeroing weak objects under garbage collection, and non-retained object pointers under manual reference counting.
This option has been superseded by the NSPointerFunctionsWeakMemory option, which causes zeroing weak behavior under manual reference counting, automatic reference counting, and garbage collection. Note that this is not entirely equivalent to and compatible with the previous option's behavior: objects must be weak-reference-safe under manual and automatic reference counting; not all objects are.

NSHashTable zeroing weak changes

The NSHashTableZeroingWeakMemory hash table option has been deprecated. This option was for holding zeroing weak objects under garbage collection, and non-retained object pointers under manual reference counting.
This option has been superseded by the NSHashTableWeakMemory option, which causes zeroing weak behavior under manual reference counting, automatic reference counting, and garbage collection. Note that this is not entirely equivalent to and compatible with the previous option's behavior: objects must be weak-reference-safe under manual and automatic reference counting; not all objects are.
Similarly, the convenience creation method +hashTableWithWeakObjects has been deprecated, and a new one which uses the new option, +weakObjectsHashTable, has been created.
Note that, analogously to what happens in garbage collection when a weak storage location was read, and an object thus read is "enlivened" so as to not be collected in the immediate future, a similar thing happens with weak references under manual and automatic reference counting: the object gets retained and autoreleased, so that it continues to live for at least the current scope of the reading code. This affects weak hash tables as well, and weakly held objects are returned with an extra retain and autoreleased, unlike typical collection behavior.

NSMapTable zeroing weak changes

The NSMapTableZeroingWeakMemory map table option has been deprecated. This option was for holding zeroing weak objects under garbage collection, and non-retained object pointers under manual reference counting.
This option has been superseded by the NSMapTableWeakMemory option, which causes zeroing weak behavior under manual reference counting, automatic reference counting, and garbage collection. Note that this is not entirely equivalent to and compatible with the previous option's behavior: objects must be weak-reference-safe under manual and automatic reference counting; not all objects are.
Similarly, the 4 convenience creation methods +mapTableWith(Weak|Strong)To(Weak|Strong)Objects have been deprecated, and 4 new ones which uses the new option, +(weak|strong)To(Weak|Strong)ObjectsMapTable, have been created.
However, weak-to-strong NSMapTables are not currently recommended, as the strong values for weak keys which get zero'd out do not get cleared away (and released) until/unless the map table resizes itself.
Note that, analogously to what happens in garbage collection when a weak storage location was read, and an object thus read is "enlivened" so as to not be collected in the immediate future, a similar thing happens with weak references under manual and automatic reference counting: the object gets retained and

autoreleased, so that it continues to live for at least the current scope of the reading code. This affects weak map tables as well, and weakly held objects are returned with an extra retain and autoreleased, unlike typical collection behavior.

NSPointerArray zeroing weak changes

Along with the deprecation of the NSPointerFunctions' NSPointerFunctionsZeroingWeakMemory option, the two +pointerArrayWith(Weak|Strong)Objects convenience creation methods have been deprecated.
Two new methods, +weakObjectsPointerArray and +strongObjectsPointerArray, have been added which create NSPointerArrays with the NSPointerFunctionsWeakMemory behavior.

NSHashTable, NSMapTable, NSPointerArray available in iOS 6

The NSHashTable, NSMapTable, and NSPointerArray classes, and some NSPointerFunctions APIs, are now available in iOS 6.

Subscripting syntax capabilities added to some collections

The NSArray/NSMutableArray and NSOrderedSet/NSMutableOrderedSet classes have gained indexed subscripting syntax support. So, for example, one can write myArray[6] to get the seventh element from an array or ordered set (with at least 7 elements!), instead of using the -objectAtIndex: method.
For arrays and ordered sets, the subscript operator [] used for reading behaves just like objectAtIndex:, and indexes outside the range [0 ... count-1] cause an exception. The subscript operator [] used for assignment can only be applied to mutable objects, and behaves like the NSMutableOrdededSet -setObject:atIndex: method, and indexes outside the range [0 ... count] cause an exception. When used to create an l-value to be assigned to, [] replaces the object if the index is less than the count (same behavior as -replaceObjectAtIndex:withObject:), and adds the element to the end of the collection, growing the collection, if the index is equal to count (same behavior as -insertObject:atIndex:).
Similarly, the NSDictionary/NSMutableDictionary classes have gained keyed subscripting syntax support. So, for example, one can write myDict[@"key"] to get the value for the key @"key" from a dictionary, instead of using the -objectForKey: method.
For dictionaries, the subscript operator [] behaves just like the -objectForKey: and -setObject:forKey: methods.

NSDictionary methods now declare <NSCopying> conformance requirement for keys

Methods in NSDictionary and NSMutableDictionary which take key arguments now declare that the argument must conform to the NSCopying protocol. If this causes a compile warning or error for you, you should look into that, as you may have a bug, or you may simply need to add a cast (to tell the compiler you know what you are doing). One way this occurs is when Class objects are used as dictionary keys: the Objective C language has no way for Class objects to declare their conformance to protocols, so Class objects cannot formally conform to the NSCopying protocol.

Weakly unavailable classes

The NSConnection, NSMachPort, and NSMessagePort classes have been marked weak-unavailable. Instances of those classes cannot be stored into weak references.

NSRealMemoryAvailable() function deprecated

The NSRealMemoryAvailable() function has been deprecated. In 32-bit, its return value cannot express the complete range of the amount of memory which may actually be available (i.e., greater than 4GB). Use NSProcessInfo's -physicalMemory method instead.

NSCopyObject() function deprecated

The NSCopyObject() function has been deprecated. It has always been a dangerous function to use, except in the implementation of copy methods, and only then with care. It was not available under automatic reference counting in 10.7, and now it has been formally deprecated.

NS_RETURNS_INNER_POINTER

Methods which return pointers (other than Objective C object type) have been decorated with the clang compiler attribute objc_returns_inner_pointer (when compiling with clang) to prevent the compiler from aggressively releasing the receiver expression of those messages, which no longer appear to be referenced, while the returned pointer may still be in use.

New NSObject protocol optional method

A declaration of the -debugDescription optional method has been added to the NSObject protocol. This allows this method to be implemented in developer classes without causing an issue to be flagged in the iOS and Mac app stores, where otherwise it might look like the developer is overriding a private method in Foundation. Remember that optional methods are not required to actually be implemented on any class which conforms to the protocol, so one must test that an object responds to the optional method before sending it.

New NSDateComponents methods

Two new methods have been added to NSDateComponents, -isLeapMonth and -setLeapMonth:.
The Chinese calendar can have a leap month inserted after any regular month. Other lunar calendars can also have leap months inserted before a regular month, and have regular months skipped, as well. The numbering of the leap months repeats (or "pre-peats") the number of a regular month. So, a month number of "10" is ambiguous -- in a given year, it could possibly mean one of two different (adjacent) months. So the interpretation of which month is specified by an NSDateComponents is really a combination of the month number and isLeapMonth states.
There is no "Unit" constant for this new state. It will be calculated and returned along with the month, requested by NSMonthCalendarUnit. It is not useful to set this new field in an NSDateComponents without also setting the month property.
Another aspect of the Chinese calendar is that years are numbered from 1 - 60, in a repeating cycle. Thus two dates 60 years apart likely have the same Year number. It is crucial to set the Era field appropriately for Chinese calendar calculations.

NSNumberFormatter, NSDateFormatter default behavior

A change in the NSNumberFormatter and NSDateFormatter classes means that instances now default to 10.4 behavior (NSNumberFormatterBehavior10_4).
Developers can ensure formatters have 10_0 behavior, if desired, by using: [theFormatter setFormatterBehavior:NSNumberFormatterBehavior10_0];
That method has been available since 10.4.

NSXPCConnection

NSXPCConnection is a new feature in the Foundation framework for interprocess communication (IPC) between applications, XPC services, daemons and agents. It builds on top of XPC to simplify IPC in Cocoa apps. NSXPCConnection takes care of the details of remote messaging, argument marshaling, secure deserialization, message dispatch, and reply handling.

NSXPCConnection Getting Started

The design of NSXPCConnection is based around the idea of defining specific interfaces that an application or daemon implements on behalf of a remote

caller. An interface is composed of methods, with optional arguments and an optional reply. All methods in the interface are called asynchronously, and arguments are copied (turning an argument into a proxy is supported, see Advanced Topics below). The interface is formalized by using an Objective C @protocol and wrapped with a Foundation NSXPCInterface object.
For example, an XPC service that compresses data on behalf of an application may have an interface like this:

```
@protocol CompressingServer
- (oneway void)compressData:(NSData *)data withReply:(void (^)(NSData *compressedData))reply;
@end
```

The method implementation on the server would receive an NSData object with the data to compress, and when it is finished it would call the reply block with the resulting compressed data. The data is sent over the connection and the reply block provided by the client application will be invoked with the result. Communication is bi-directional, so the client may implement its own interface that the server can call.
Once the interfaces are defined, a channel of communication needs to be opened between the two applications. One side should be listening for new connections, and the other will connect to it. In our example, the CompressingServer is an XPC service that listens for new connections. It may implement its main method like this:

```
NSXPCListener *listener = [NSXPCListener serviceListener];
listener.delegate = [ListenerDelegate new];
[listener resume]; // does not return
```

At this point, the service is waiting to receive new connections. The delegate is consulted when a new connection is created, giving the service a change to configure the new connection with the interface we defined earlier. Each connection also exports an object. The exported object will receive the messages defined in the interface when they are sent by the remote object. In this example, the delegate also implements the CompressingServer protocol.

```
@interface ListenerDelegate : NSObject <NSXPCListenerDelegate, CompressingServer>
@end
...
- (BOOL)listener:(NSXPCListener *)listener shouldAcceptNewConnection:(NSXPCConnection *)newConnection {
    newConnection.exportedObject = self;
    newConnection.exportedInterface = [NSXPCInterface interfaceWithProtocol:@protocol(CompressingServer)];
    [newConnection resume];
    return YES;
}
```

Next, the client application will connect to the service. It looks it up by name and sets up a few properties:

```
NSXPCConnection *connection = [[NSXPCConnection alloc] initWithServiceName:@"com.yourcompany.CompressingServer"];
connection.remoteObjectInterface = [NSXPCInterface interfaceWithProtocol:@protocol(CompressingServer)];
[connection resume];
```

Now that we have the connection configured, the client may send a request to the server by simply sending a normal message to a proxy object supplied by the connection.

```
id <CompressingServer> proxy = [c remoteObjectProxy];
[proxy compressData:bigData withReply:^(NSData *compressedData) {
    [compressedData writeToURL:myURL atomically:YES];
}];
```

In summary, to create a basic connection between two applications:
1. Create an NSXPCInterface to define the interface that each part will implement
2. Create an NSXPCListener on one side to listen for new connections
3. Connect to the service using NSXPCConnection
4. Send messages to the proxy object

NSXPCConnection Error Handling

Of course, not every message send is guaranteed to work. NSXPCConnection provides several levels of error handling. On the connection object itself, you may setup blocks to be called in case of connection invalidation or interruption. Also, for each message send, you can supply an error block to be called. If, due to an error on the connection like the remote process crashing, the reply block will not be called, then the error block will be invoked instead. The error handling block is a property of the proxy you send the message through.

```
id <CompressingServer> proxy = [c remoteObjectProxyWithErrorHandler:^(NSError *err) {
    // handle error
}];
[proxy compressData:bigData withReply:^(NSData *compressedData) {
    [compressedData writeToURL:myURL atomically:YES];
}];
```

Sometimes, it is convenient to nest these methods:

```
[[c remoteObjectProxyWithErrorHandler:^(NSError *err) {
    // handle error
}] compressData:bigData withReply:^(NSData *compressedData) {
    [compressedData writeToURL:myURL atomically:YES];
}];
```

NSXPCConnection Object Arguments

NSXPCConnection uses a new subclass of the NSCoding protocol to provide a layer of security when decoding objects. It is called NSSecureCoding. When using secure coding, an object may specify an expected class for every object it decodes. This means that the remote process can not cause an unexpected kind of object to be instantiated by simply sending it over the wire.
Many Foundation classes already adopt NSSecureCoding. If you want to send your own objects over the wire, they should adopt it as well.
1. Adopt the NSSecureCoding protocol, including implementing -initWithCoder:, and -encodeWithCoder:. In your -initWithCoder, use the new NSCoder methods, for example: -decodeObjectOfClass:forKey:.
2. Implement +(BOOL)supportsSecureCoding on your class and return YES.
We have extended the metadata generated by the clang compiler and understood by the Objective C runtime about methods in @protocols. This allows NSXPCInterface to automatically whitelist any object classes specified as arguments. However, if the argument is a collection (for example, NSArray or NSDictionary), the secure coding protocol requires some more information about the expected classes of objects in that collection. For example, for this protocol:

```
@protocol SpellChecker
- (oneway void)spellCheckWord:(NSString *)word withReply:(void (^)(NSArray *suggestions))reply;
@end
```

NSXPCInterface can automatically determine the first argument should be of class NSString. It needs additional information about the objects in the ‘suggestions’ argument in the reply block. Here is how you would set it up to say that only NSString objects are allowed in the array:

```
NSXPCInterface *ifc = [NSXPCInterface interfaceWithProtocol:@protocol(SpellChecker)];
[ifc setClasses:[NSSet setWithObject:[NSString class]] forSelector:@selector(spellCheckWord:withReply:) ofReply:YES];
```

NSXPCConnection Advanced Topics

Additional Proxy Objects
By default, all object arguments to your methods are sent ‘by copy’ over the connection. This is usually the best approach for performance, because then the receiving side has a completely local object graph to use. Sometimes, however, you may want an object to become an additional proxy object (like the root remoteObjectProxy provided by NSXPCConnection). This is possible by simply telling the NSXPCInterface that a specific argument of a method in its protocol should be a proxy object, by providing the NSXPCInterface for that new proxy.

NSUserNotification and NSUserNotificationCenter

NSUserNotification and NSUserNotificationCenter are new APIs to interact with the system-wide Notification Center. Your application can use this API to deliver notifications immediately, or schedule them to appear at some time in the future. Notifications can be configured in many ways, including repeat intervals, custom titles, and button names. Your application will be notified when a user clicks on a notification. If your application is not running, then it will be launched and given the opportunity to respond to the user action.
There are two key classes in the API: NSUserNotification, which is a model object representing one notification, and NSUserNotificationCenter, which is a singleton controller object that represents your application’s interface with the Notification Center.
A simple notification delivered immediately may look like this:

```
NSUserNotification *note = [[NSUserNotification alloc] init];
note.title = @"Hello";
[[NSUserNotificationCenter defaultUserNotificationCenter] deliverNotification:note];
```

A simple notification scheduled for a time in the future may look like this:

```
NSUserNotification *note = [[NSUserNotification alloc] init];
note.title = @"Hello";
note.deliveryDate = [NSDate dateWithTimeIntervalSinceNow:5];
[[NSUserNotificationCenter defaultUserNotificationCenter] scheduleNotification:note];
```

This will schedule a notification to be presented in 5 seconds. Note that if your application is frontmost when the notification is delivered or when the scheduled time arrives, it will not be shown in the user interface with an animation. This is because if your application is frontmost, it is expected that your application will present whatever UI is appropriate. It will still appear in the list of delivered notifications in the Notification Center, however. If you want to override this behavior, see the section on the NSUserNotificationCenter delegate below.
It is very important to use the appropriate calendar, date, and time APIs when calculating the delivery date. Failure to correctly create the NSDate value for deliveryDate will result in unexpected behavior. Please consult the Date and Time Programming Guide for more information on calculating dates.
NSUserNotificationCenter maintains two lists of notifications for your application. The first list is scheduled notifications. Scheduled notifications will be delivered to the notification center at some point in the future. You can schedule notifications individually or bulk schedule notifications by setting the scheduledNotifications property of NSUserNotificationCenter.
The second list is delivered notifications. These are notifications which already appear in the Notification Center. You can deliver a notification to the center immediately using the deliverNotification: method. The remove methods let your application manage the list of delivered notifications. It is appropriate to remove old delivered notifications when they are no longer useful, instead of requiring your user to manually clean up the list.
Note that these two lists are stored in the notification center daemon, and require blocking interprocess communication to retrieve the contents. You should not call this method excessively.
The NSUserNotificationCenter will most often be configured with a delegate. This delegate will be notified when the notification center delivers a notification, using the delegate method userNotificationCenter:didDeliverNotification:.
The delegate will also be notified when the user activates a notification by clicking on it. If your application is running when the notification is activated, your delegate’s implementation of userNotificationCenter:didActivateNotification: will be called. At this time you should take appropriate action, including displaying the relevant data to the user in your UI. If your application is not running, then it will be launched. Because the delegate will not be set immediately on application launch, the NSUserNotification object representing the activated notification will be delivered as an object in the user info dictionary of the applicationDidFinishLaunching method on your application delegate, using the NSApplicationLaunchUserNotificationKey key.
The delegate also has a method called userNotificationCenter:shouldPresentNotification:. As noted above, if your application is frontmost then notifications will be suppressed. If you want the notification to be presented anyway, implement this method and return YES. This feature should be used judiciously to avoid annoying your users.
The NSUserNotification object has several properties that you may configure before scheduling a notification. The most common properties your application will set are the string values (title, subtitle, etc). Remember to localize these strings as appropriate. Other important properties include the deliveryDate and deliveryRepeatInterval. The repeat interval is specified using an NSDateComponents. If you want the notification to repeat every hour, for example, then set the ‘hour’ property of the NSDateComponents object to 1. This ensures correct behavior across daylight saving time boundaries and other transitions.
If your notification has an absolute time (for example, an egg timer), you may choose to set the deliveryTimeZone to the current time zone. If the default value of nil is used, then the delivery time will be adjusted if the user changes time zones.
The NSUserNotification object also has several properties that are set by the notification center after a notification has been delivered, including a BOOL value indicating if your notification was actually presented and an NSDate recording the actual delivery date (useful for repeating notifications).
The user has ultimate control over what notifications are displayed, and the style (banner, alert, etc). There is no mechanism to override the user preferences.

New iCloud APIs

In Mac OS 10.7, the only way to check whether the user is logged into iCloud with Data & Documents enabled is to invoke -[NSFileManager URLForUbiquityContainerIdentifier:] and check for a non-nil URL. However this method can sometimes block for a significant amount of time, making it unsuitable for calling from the main thread.
In Mac OS 10.8 there is a new method called -[NSFileManager ubiquityIdentityToken], which can return much more quickly than -URLForUbiquityContainerIdentifier: and is suitable for invoking on the main thread. This method will return nil if Data & Documents is disabled or the user is logged out of iCloud entirely, otherwise it will return non-nil.
This API can also be used to detect changes in the current iCloud account, if that is important to your application. When a user is logged into iCloud with Data & Documents enabled, this method returns an opaque token that can be copied, compared with -isEqual:, and encoded. Each iCloud account will have a different identity token associated with it. You can detect changes in the current iCloud account by registering an observer for NSUbiquityIdentityDidChangeNotification—a new notification sent via [NSNotificationCenter defaultCenter]—and upon receiving that notification comparing the old and new values of -ubiquityIdentityToken. If the tokens are equal, then the account definitely did not change. If the tokens are not equal, then the current account may be, and usually is, different.

File Coordination for -setUbiquitous:itemAtURL:destinationURL:error:

In Mac OS 10.7, -[NSFileManager setUbiquitous:itemAtURL:destinationURL:error:] would create its own NSFileCoordinator instance and perform its own file coordination so that callers were not required to. Unfortunately, this created problems when used in the context of NSDocument and other NSFilePresenters that do not want to receive -relinquishPresentedItemToWriter: for that write. Since NSFileManager created the NSFileCoordinator itself, there was no way filter out the -relinquishPresentedItemToWriter: method by initializing the NSFileCoordinator with the right NSFilePresenter.
In Mac OS 10.8, this method will now detect if file coordination is being performed on the current thread, and if so will not do any file coordination of its own. You are encouraged to start doing your own file coordination around this method, like so:

```
NSFileCoordinator *fc = [[NSFileCoordinator alloc] initWithFilePresenter:myFilePresenter];
[fc coordinateWritingItemAtURL:sourceURL options:NSFileCoordinatorWritingForMoving
              writingItemAtURL:destinationURL options:NSFileCoordinatorWritingForReplacing
                         error:errorPtr byAccessor:^(NSURL *newSourceURL, NSURL *newDestinationURL) {
    success = [fileManager setUbiquitous:YES itemAtURL:newSourceURL destinationURL:newDestinationURL error:errorPtr];
    if (success) {
        [fc itemAtURL:newSourceURL didMoveToURL:newDestinationURL];
    }
];
[fc release];
```

You must invoke -setUbiquitous:itemAtURL:destinationURL:error: on the same thread that initiated the file coordination.
Take note that when you start doing your own file coordination for -setUbiquitous:itemAtURL:destinationURL:error:, you need to invoke -[NSFileCoordinator

itemAtURL:didMoveToURL:] when the operation is successful. Otherwise the system will not be able to reliably inform other NSFilePresenters of the file's new URL.

NSFileManager Trash APIs

In Mac OS 10.8, NSFileManager has new methods to manage the Trash. The -[NSFileManager trashItemAtURL:resultingItemURL:error:] will attempt to move the item at the given URL to the trash, returning the resulting URL by reference. As part of this operation, the system may rename the file to avoid name conflicts; if so, the resultingItemURL will reflect this new name.
You should use this API within File Coordination like so:

```
NSFileCoordinator *fc = [[[NSFileCoordinator alloc] init] autorelease];
[fc coordinateWritingItemAtURL:urlToMoveToTrash options:NSFileCoordinatorWritingForMoving error:&error byAccessor:^(NSURL *url
    success = [[NSFileManager defaultManager] trashItemAtURL:url resultingItemURL:&resultingURL error:&error];
}];
```

If you need to find the URL of a Trash folder, you can use NSTrashDirectory, a new NSSearchPathDirectory enum value. You can combine this value with NSUserDomainMask, NSLocalDomainMask or a URL when calling -[NSFileManager URLForDirectory:inDomain:appropriateForURL:create:error:], to get a specific Trash directory appropriate for the given domain or URL.
Some volumes may not support a Trash folder, so these methods will report failure by returning NO or nil and an NSError with NSFeatureUnsupportedError. NSFeatureUnsupportedError is a new error code in the NSCocoaErrorDomain that indicates a failure to perform the requested operation because the feature is not supported, either because the file system lacks the feature, or required libraries are missing, or other similar reasons.

-[NSObject autoContentAccessingProxy]

Prior to Mac OS 10.8, -[NSObject autoContentAccessingProxy] returned an object that did not properly implement message forwarding. This proxy now behaves correctly on Mac OS 10.8.

NSDataWritingWithoutOverwriting

By default -[NSData writeToURL:options:error:] will try to overwrite the file at the specified path, if it exists. Prior to Mac OS 10.8 it was not possible to use this method to write to the specified file only if it doesn't already exist in an atomic manner. In Mac OS 10.8 the NSDataWriting enum now includes NSDataWritingWithoutOverwriting to provide this ability.
It currently is not possible to do an atomic write that prevents overwriting, so specifying both NSDataWritingAtomic and NSDataWritingWithoutOverwriting will cause NSData to throw an exception.

NSIndexSet Enumeration

In Mac OS 10.7, -[NSIndexSet enumerateIndexesInRange:options:usingBlock:] would throw an exception when given an empty range. This is fixed in Mac OS 10.8 so that the method simply does nothing and the given block is never invoked.

NSUUID

NSUUID is a new class in Foundation. It includes support for creating RFC 4122 version 4 UUIDs.
NSUUID complements CFUUID in CoreFoundation. It is not toll-free bridged with CFUUID, but you can use the UUIDString method to convert between the two.
If you need to compare the value of two NSUUIDs, use isEqual: instead of ==.

NSLinguisticTagger

As of 10.8, NSLinguisticTagger provides parts-of-speech tags and lemmas for Spanish and Italian, in addition to English, French, and German.

NSError

NSErrors created with a nil userInfo no longer return nil when -userInfo is called; instead an empty dictionary is returned. This is effective for apps linked against 10.8 and later.

NSUserScriptTask

NSUserScriptTask and its subclasses are new classes intended to execute user-supplied scripts, and will execute them outside of the application's sandbox, if any. They are not intended to execute scripts built into an application; for that, use NSTask, NSAppleScript, or AMWorkflow. If the application is sandboxed, then the script must be in the "application scripts" folder, which you can get using +[NSFileManager URLForDirectory:NSApplicationScriptsDirectory ...]. A sandboxed application may read from, but not write to, this folder.
If you simply need to execute scripts without regard to input or output, use NSUserScriptTask, which can execute any of the specific types. If you need specific control over the input to or output from the script, use one of the subclasses, which have more detailed "execute" methods.

NSByteCountFormatter

NSByteCountFormatter is an NSFormatter subclass for use in displaying localized byte counts such as file, disk, and memory sizes. It has default behaviors that match the system and should be good enough in many cases; however, the behaviors can also be explicitly set if needed.
One important feature of this class is its ability to specify whether 1000 or 1024 bytes make up a kilobyte in the output. By using the default setting of NSByteCountFormatterCountStyleFile you can display file and disk sizes in your apps in a way that is consistent with the rest of the system. For displaying memory sizes, you can choose NSByteCountFormatterCountStyleMemory.
In 10.8, this class is used just for formatting, and not parsing. Parsing APIs inherited from NSFormatter will fail with NO return if invoked.

NSString

NSString and NSMutableString -initWithUTF8String: methods no longer go through -initWithBytes:length:encoding:. Ideally this shouldn't affect anyone; however, it is possible that some subclassers may depend on this behavior. For that reason, subclassers in apps linked before 10.8 will be treated compatibly and go through the old code path. The app may crash when rebuilt against 10.8 SDKs, and the subclass may need to be fixed.

NSString localized formatting

In 10.8, in apps linked against the 10.8 SDK, -localizedStringWithFormat:, and -initWithFormat:locale: (and friends) when supplied with a non-nil locale, will now do localized formatting of numbers. Previously these calls already did decimal point handling; so in some locales comma would be used for decimal point separator. This new behavior builds upon this to use localized digits as well as thousands separators and proper placement of the sign symbol.
The localization applies in the following cases:
- Only e, E, f, F, g, G, d, D, i, u, U are localized; the upper case variants are ignored (and treated as if they were lower case). a, A, o, O, x, X are not localized, and neither are all other non-numeric format characters.
- %@ used with NSNumber/CFNumber is localized.
- The flags +, -, 0, width, and precision are honored, but note that extra characters (beyond the specified width) may be inserted to accommodate thousands separators.
Potential pitfalls include getting localized output where unlocalized was intended, and use of unintended thousands separators (for instance if someone was

showing a year with %d, now they may get "2,012").

NSString localized case mapping methods

NSString now has methods for performing localized case mapping: -uppercaseStringWithLocale:, -lowercaseStringWithLocale:, and -capitalizedStringWithLocale:. To perform localized mapping based on the user's preference, pass +[NSLocale currentLocale] to these methods. Passing nil indicates non-localized canonical behavior.

Dictionary shared key set API

NSDictionary now offers a method to create a shared key set object, +sharedKeySetForKeys:. You give it an array of keys, and it creates an object which represents those keys and can be passed into the new NSMutableDictionary creation method +dictionaryWithSharedKeySet:. All NSMutableDictionaries created with the same shared key set object share the key objects in it, and do not need to store them or pointers to them, saving memory. So, this is useful if you are creating a lot of dictionaries with more or less the same keys.
Since a shared key set is created once, and then used many times, it is worthwhile to expend some additional effort during its creation, and it can then speed up the hash probing for key lookups and sets in the mutable dictionary as well. The shared key object calculates a minimal perfect hash of the keys you give it which, for keys in the ket set, then eliminates any need for probe looping during a dictionary lookup.
Don't get hung up on the term "set" in "shared key set", it's not an NSSet. The word "set" here is being used more abstractly, in the sense of "collection" or "group".
When you have a situation where you know you will have a lot of dictionaries with lots of the same keys, and you know what those keys are, what you would typically do is create a shared key set object once, with the most common keys, save it away somewhere, and then create dictionaries using +dictionaryWithSharedKeySet: with that object as the argument. These dictionaries behave just like ordinary dictionaries, keys can be any copyable object, and you can store key-value pairs in them for keys which are not in the shared key set object. In fact, it isn't advisable to create the shared key set object with the whole universe of possible keys; that may just end up wasting memory that could have been better used elsewhere. The expense of creating the shared key set object also depends on how many keys with which you are creating it. You want to create the shared key object with the most popular keys in these dictionaries you are going to be creating. Keys which aren't in the shared key set, should they be added to one of these special mutable dictionaries, will be handled in a similar fashion to an ordinary Foundation-provided NSMutableDictionary.
As with any object used as a key to a dictionary, the key objects given to +[NSDictionary sharedKeySetForKeys:] must implement the NSCopying protocol and must have -hash and -isEqual: methods which obey the hash-isEqual invariant and must have hash values that don't change over time. The shared key set object is more effective (more memory and cpu savings) when the keys (or rather, the class(es) of the keys) have good hash functions, and the -hash method for each of the keys returns a different value. However, that is not required, just as with ordinary dictionaries (and would be problematic for a developer to ensure anyway!).

Notes specific to Mac OS X 10.7

JSON Support in Foundation

The new NSJSONSerialization class in Foundation has support for serializing Foundation objects to JSON and deserializing JSON into Foundation objects. Reading and writing from both data objects and streams is supported. Please see the NSJSONSerialization.h header for more information about the new class and methods.

Distributed Objects now supports keyed archiving

When using Distributed Objects between processes running on OS X 10.7 or later, keyed archiving is now used by NSPortCoder. Messages sent between OS X 10.7 and any earlier version of OS X will use non-keyed archiving. Call allowsKeyedCoding on the coder in encodeWithCoder and the decoder in initWithCoder to determine if keyed archiving is being used.

Leak of objects passed in structures over Distributed Objects fixed

In OS X 10.6 Snow Leopard and earlier, an object in a structure that was passed over Distributed Objects had an extra retain and was leaked. This leak has been fixed in OS X 10.7 for applications linked with the OS X 10.7 SDK or later.

NSValue objects with common structure types now support keyed archiving

An NSValue object with an NSPoint, NSRect, NSSize, NSRange, CGAffineTransform, NSEdgeInsets, or UIEdgeInsets structure type may now be archived and unarchived using NSKeyedArchiver. The value will only unarchive on OS X 10.7 or later.

New Availability Macros in Foundation and AppKit headers

The header files for classes that are new in OS X 10.7 use a new macro, NS_CLASS_AVAILABLE(_MacOSIntro, _iOSIntro), to indicate the availability of the class. Classes decorated in this manner may be nil-checked for existence in future versions of OS X or iOS, like this:

```
if ([NSNewClass class]) { /* ... */ }
```

Methods, functions, and exported values also use a new macro, NS_AVAILABLE(_MacOSIntro, _iOSIntro).
Availability macros for symbols introduced in OS X 10.4 Tiger or earlier have been removed.

NSFilePresenter

OS X 10.7 includes a new mechanism that allows file presenters, which are objects that present the contents of files or directories to the user for viewing or editing, to take an active role in operations that access those files or directories, even operations performed by other processes in the system. It's an important part of the implementation of the OS X 10.7 modernized document model that is described in the AppKit release notes.
An application uses this mechanism by creating and registering a file presenter for each document file or file package whose contents is being presented to the user. When some other process uses the file coordination mechanism described in the next section to read from or write to the presented file system item, that coordinated reading or writing is made to wait while the file presenter is given a chance to do various things first. The set of things that a file presenter can do while coordinated reading and writing is made to wait is defined by the NSFilePresenter protocol, which is declared in <Foundation/NSFilePresenter.h>. You use this protocol in your application by instantiating classes that conform to it and registering those instances with the NSFileCoordinator class, which is introduced in the next section and declared in <Foundation/NSFileCoordinator.h>. See the comments in those files for details.
For example, in a typical "shoebox" application like iPhoto, which doesn't readily expose the notion of files to the user but nonetheless stores the user's data in files somewhere on the user's computer, you might make the class of the application delegate object conform to this protocol if the files are all in one directory tree. The application delegate would register itself as an NSFilePresenter in its -applicationWillFinishLaunching: method. If the files are in multiple directory trees then you would probably make some class of your own design conform to this protocol and instantiate it on the basis of one per directory tree, and each of those instances would be registered.
For the benefit of document-centric applications, NSDocument has been made to conform to this protocol because each instance of NSDocument presents the contents of a document file or file package to the user. NSDocument automatically registers and deregisters instances of itself with the NSFileCoordinator class.

Every NSFilePresenter is responsible for keeping track of what file system item it's presenting, and providing a URL for it on command:

```
@property (readonly) NSURL *presentedItemURL;
```

When your application registers an NSFilePresenter it asserts "ownership" of the presented file or directory. Nothing else in the system that uses file coordination appropriately will be able to read from or write to the presented item without your NSFilePresenter being given the opportunity to prepare for the file access beforehand and notified of its completion afterward. There are two main methods your NSFilePresenter can implement to get that opportunity:

```
- (void)relinquishPresentedItemToReader:(void (^)(void (^reacquirer)(void)))reader;
- (void)relinquishPresentedItemToWriter:(void (^)(void (^reacquirer)(void)))writer;
```

What sort of preparation needs to be done depends on the application. In particular, it depends on how strong of an ownership model your application requires. A very weak ownership model is possible if your application is able to use coordinated reading and writing for all of its access to the presented file or directory, instead of merely depending on other processes to do coordinated reading and writing.
Asserting ownership of a presented file or directory is just one of several reasons for an application to register a file presenter. For example, there's a method your NSFilePresenter can implement to be given the opportunity to make sure the presented file or directory is up to date before another process using coordinated reading reads from it:

```
- (void)savePresentedItemChangesWithCompletionHandler:(void (^)(NSError *errorOrNil))completionHandler;
```

NSDocument's implementation of this method autosaves the document if it has been changed since the last time it was saved or autosaved. This is the implementation of the part of the OS X 10.7 modernized document model that allows the user to not have to know and take care to save document changes before causing the document’s file to be read by another application.
NSFilePresenter has other methods. See the comments in <Foundation/NSFilePresenter.h> for details.

NSFileCoordinator

OS X 10.7 includes a new mechanism called file coordination. In addition to triggering work by file presenters, as described in the previous section and the comments in <Foundation/NSFilePresenter.h>, it allows your application to access files and directories in a way that is serialized with other process' access of the same files and directories so that inconsistencies due to overlapping reading and writing don't occur.
File coordination is performed by instances of the new NSFileCoordinator class. There are two main NSFileCoordinator methods:

```
- (void)coordinateReadingItemAtURL:(NSURL *)url
                           options:(NSFileCoordinatorReadingOptions)options
                             error:(NSError **)outError
                        byAccessor:(void (^)(NSURL *newURL))reader;
- (void)coordinateWritingItemAtURL:(NSURL *)url
                           options:(NSFileCoordinatorWritingOptions)options
                             error:(NSError **)outError
                        byAccessor:(void (^)(NSURL *newURL))writer;
```

You use these methods by putting your application's reading or writing of some files and directories (definitely not all, more on this in below) in blocks and passing the blocks to invocations of these methods. Each time you invoke one of these methods it will synchronously wait if appropriate until other processes that also use NSFileCoordinator finish accessing the same file system item, and in some cases other file system items. Your block will be invoked. While your block is being invoked other processes that also use NSFileCoordinator and access the same file system item, and in some cases other file system items, will be made to wait. When your block returns, one or more of those other processes will stop waiting as appropriate. See the comments in <Foundation/NSFileCoordinator.h> for details.
Your application should use file coordination for files that the user thinks are files, and opens knowingly in applications, or manipulates with the Finder, or makes attachments out of, as in Mail. The default implementations of the right NSDocument and NSDocumentController methods already use file coordination so you may not have to do anything to your application to make it use file coordination if it's NSDocument-based.
Your application need not use file coordination when the serialization of its file access with other process' file access would not be useful. For example, there is typically no benefit to using file coordination for private cache and temporary files because typically there are no other processes with which your application must coordinate.
NSFileCoordinator is designed to accommodate the fact that files and directories may be moved or renamed while your process is waiting to access them; notice the NSURL that is passed to file accessor blocks. If your application renames a file while saving it then then your application must announce that it is doing so, using this NSFileCoordinator method:

```
- (void)itemAtURL:(NSURL *)oldURL didMoveToURL:(NSURL *)newURL;
```

The need to use this method is rare but arises, for example, in TextEdit when a file's name extension must be changed from .rtf to .rtfd because the user has added attachments to a rich text document and a different file format must be used for it. If your application is NSDocument-based you probably do not have worry about this because the default implementation of the right NSDocument method already invokes this.
NSFileCoordinator has other methods, and there are options you can choose to control coordinated reading and writing. See the comments in <Foundation/NSFileCoordinator.h> for details.
This mechanism is different from traditional BSD advisory file locking in that:
• It does not require that your application do everything it needs to do to a file for a particular user operation within one open()/flock()/reading or writing/close() sequence to enforce consistency. This is important in the context of a Cocoa application because most Cocoa methods deal in NSURLs, not file descriptors or even NSFileHandles.
• It has a few features that BSD advisory file locking does not have (and would not even be appropriate to put at that level). One important example of this is that coordinated reading of a file can actually trigger writing to that file by another process, while your reading waits, to help implement the OS X 10.7 modernized document model that is described in the AppKit release notes. Coordinated writing can also trigger actions in other processes, while your writer waits. These are described in the previous section.
• It takes file packages into account automatically.
• There is no equivalent of flock()'s LOCK_NB option or EWOULDBLOCK error code. This mechanism is for use when the file access is being commanded by a user and presenting the user with an error about an inability to take a lock would be unacceptably bad UI because that is such a technical concept.
• The methods we have published to expose this mechanism are structured so as to make it difficult for you to accidentally make other processes wait to access files longer than you intend. You don't lock and unlock, you do coordinated reading or writing, passing our methods blocks of code that do the reading or writing. When one of those blocks returns or throws an Objective-C exception, or your application crashes of course, your application definitely stops making other processes wait.
Several components of OS X 10.7 use file coordination. For example:
• NSDocumentController's document opening code does a single coordinated read of each document file that’s opened. In other words, all of the work it does to actually read a document file is within a block passed to -[NSFileCoordinator coordinateReadingItemAtURL:options:error:byAccessor:], even if the work is on a background thread. There’s more to opening a document than just reading the file, but the other work is done before and after the reading is done, on the main thread.
• NSDocument's document saving code does a single coordinated write of each document file that’s saved. In other words, all of the work it does to actually write a document file is within a block passed to -[NSFileCoordinator coordinateWritingItemAtURL:error:options:byAccessor:], even if that’s on a background thread. “All of the work” typically includes writing a new revision of the document to a file in a temporary directory, perhaps setting some file attributes on it, and then safely moving it into place to replace the old revision on disk. There’s more to saving a document than just that, but the other work is done before and after the writing is done, on the main thread.
• Mail does coordinated reading of files it's making into message attachments so, if any of those files are open documents, changes already made by the user to those documents get reliably written to disk in time for Mail to read them. In other words, all of the work it does to read each dropped file is within a block passed to coordinateReadingItemAtURL:options:error:byAccessor:.
• Finder does coordinated reading of files that it’s copying, to make sure changes that users have made to documents get copied too.

New Handling of Ambiguous Scripting Location Specifiers in NSPositionalSpecifier

In all previous versions of OS X Cocoa’s scripting support would interpret location specifiers that specify replacement literally so if one was used as an argument to a command the command could replace the specified object even when that was not what the vast majority of people writing scripts would expect or want. For example, telling System Events to ‘make new folder at folder “A Folder to Add To“ of disk "Stuff" with properties {name:”Oh No!”}’ would replace the folder named “A Folder to Add To” with an empty folder named “Oh No!” instead of creating a new folder named “Oh No!” inside of “A Folder to

Add To." In general this kind of problem applied to the make, duplicate, and move commands in applications having scripting models where a class' elements can be nested objects of the same class, like System Events' folders or Xcode's groups.
In OS X 10.7, NSPositionalSpecifier, the class where this behavior is implemented, has been changed so that location specifiers like the one in the above example specify insertion into the specified object instead of replacement of the specified object. It only does this when the class of object being inserted matches one of the element classes of the class of specified object. For backward binary compatibility the old behavior remains in applications linked against OS X 10.6 or older. If you choose to you can mitigate the risk of breaking existing scripts by forcing NSPositionalSpecifier to retain the old behavior, regardless of what version of OS X it's linked against, by setting the value of the NSPositionalSpecifierPrefersInsertionOverReplacement user default to false. (The correct way for an application to set a user default that isn't actually a user preference like this one is to use -[NSUserDefaults registerDefaults:] during the application's launching.)

More Precise Removal of Key-Value Observers

Since the introduction of key-value observing (KVO) in OS X 10.3 there have been methods to register and deregister one object as the observer of a keyed value in another, but the NSObject(NSKeyValueObserverRegistration) deregistration method, -removeObserver:forKeyPath:, suffered from the flaw of not allowing you to specify the same context pointer that you had passed to -addObserver:forKeyPath:options:context:. When the same observer is registered for the same key path multiple times, but with different context pointers each time, -removeObserver:forKeyPath: has to guess at the context pointer when deciding what exactly to remove, and it can guess wrong. This is particularly galling given our advice to check the context pointer when receiving a KVO notification to discriminate among different purposes for observing in the first place. A surprising and difficult to debug example of this is code in a subclass observing the same keyed value in the same object as code in a superclass.
To let you more precisely specify your intent when removing an observer a new NSObject(NSKeyValueObserverRegistration) method has been added in OS X 10.7:

```
- (void)removeObserver:(NSObject *)observer forKeyPath:(NSString *)keyPath context:(void *)context;
```

The same issue applies to batched key-value observer deregistration, so a new NSArray(NSKeyValueObserverRegistration) method has also been added:

```
- (void)removeObserver:(NSObject *)observer fromObjectsAtIndexes:(NSIndexSet *)indexes
            forKeyPath:(NSString *)keyPath context:(void *)context;
```

Bug Fix in Key-Value Observing

In OS X 10.6 there was a bug in key-value observing in which one observer observing a key path like "commonObject.value" from two different objects, where the value for "commonObject" of the two objects was the same object, could cause a variety of crashes, exceptions, and malfunctions. Whether or not the bug actually manifested itself depended on a variety of factors, like the order of observer adding and removing and whether the context pointer for the two observations was the same. This bug has been fixed in OS X 10.7.

Improved Error Handling and Reporting in NSFileManager

Several changes have been made to NSFileManager in OS X 10.7 to improve error handling and reporting.
For applications linked on or after OS X 10.7, the delegate method -fileManager:shouldProceedAfterError:movingItemAtURL:toURL: will now be called when an item at the destination URL already exists. If the delegate returns YES, the operation will proceed, which will likely result in overwriting the item at the destination. Errors reported by NSFileManager's copy, move, remove, and link methods are now more consistent about returning NSErrors in the Cocoa error domain with POSIX domain underlying errors. NSFileManager's copy, move, and link methods now report errors with the new NSFileWriteFileExistsError Cocoa domain error code when the destination file already exists.
For applications linked on or after OS X 10.7, the delegate method -fileManager:shouldProceedAfterError:linkingItemAtURL:toURL: will now properly be called when NSFileManager encounters an error during a link operation. Before OS X 10.7, NSFileManager mistakingly called -fileManager:shouldProceedAfterError:copyingItemAtURL:toURL: instead.

New NSURL-based NSFileManager API

In OS X 10.6, many NSFileManager APIs were modified to take NSURLs as parameters instead of NSString paths to promote file system efficiency. In OS X 10.7, two additional methods now have NSURL-taking versions. The new methods are:

```
- (BOOL)createDirectoryAtURL:(NSURL *)url
        withIntermediateDirectories:(BOOL)createIntermediates
                         attributes:(NSDictionary *)attributes
                              error:(NSError **)error;
- (BOOL)createSymbolicLinkAtURL:(NSURL *)url
             withDestinationURL:(NSURL *)destURL
                         error:(NSError **)error;
```

The semantics of these new methods are identical to their NSString-taking counterparts.
In order to create a relative symbolic link, the destinationURL should be created using [NSURL URLWithString:relativePath relativeToURL:nil].
Range Enumeration of NSIndexSet NSIndexSet is an abstraction of, as the name indicates, a set of indexes. It is also often used as a set of non-contiguous ranges. Before OS X 10.7, there was no API for accessing the ranges of indexes stored in an index set; ranges could only be accessed by using the index-based primitives. Since this is not trivial, OS X 10.7 has new convenience API for enumerating an NSIndexSet's non-contiguous ranges.

```
- (void)enumerateRangesUsingBlock:(void (^)(NSRange range, BOOL *stop))block;
- (void)enumerateRangesWithOptions:(NSEnumerationOptions)opts usingBlock:(void (^)(NSRange range, BOOL *stop))block;
- (void)enumerateRangesInRange:(NSRange)range options:(NSEnumerationOptions)opts usingBlock:(void (^)(NSRange range, BOOL *sto
```

If the range passed into the last method intersects a range of the receiver's indexes, then that intersection will be passed to the block.
These methods are only guaranteed to perform as well as if they were implemented with -enumerateIndexesInRange:options:usingBlock:. However, if the receiver's implementation permits, it may perform better than that.
Safe File Mapping for NSData Before OS X 10.7, specifying NSDataReadingMapped (or NSMappedRead) for -initWithContentsOfFile:options:error: would cause NSData to always attempt to map in the specified file. However, in some situations, mapping a file can be dangerous. For example, when NSData maps a file from a USB device or over the network the existence of said file can't be guaranteed for the lifetime of the object. Accessing the NSData's bytes after the mapped file disappears will likely cause unpredictable crashes in your application.
For applications linked on OS X 10.7 or later, NSDataReadingMapped will now only map the requested file if it can determine that mapping the file will be relatively safe. To reflect this change, NSDataReadingMapped has been deprecated and is now an alias for NSDataReadingMappedIfSafe.
The methods -dataWithContentsOfMappedFile: and -initWithContentsOfMappedFile: have been deprecated. In the very rare event that your application needs to map a file regardless of safety, you may use the NSDataReadingMappedAlways option.

More flexible version of NSIntegralRect

Foundation already provides NSIntegralRect, which takes a rectangle and pushes each edge outward to the nearest integer. For 10.7 we add NSIntegralRectWithOptions, a version that gives you some control over how the rect is massaged into being integral.
Each edge of the rect or its width and height can be pushed inward, outward, or to the nearest integer.
The primary purpose of this function is to provide the options (and implementation) used in the new AppKit method, -[NSView backingAlignedRect:options:].

NSUserDefaults and Threading

In earlier releases of OS X, NSUserDefaults (and CFPreferences) serialized all access; if you've been avoiding using the preferences system in heavily threaded apps due to contention, this should be greatly improved.
Additionally, automatic synchronization is now non-blocking in most cases. You should consider any calls to NSUserDefaults -synchronize carefully to see if they're really necessary, as you can avoid blocking IO by removing them.

NSUserDefaults euid/uid handling

Prior to OS X Lion, CFPreferences (and therefore NSUserDefaults) would change the owner of the plist files it wrote to the uid/gid of the application writing to preferences. In Lion, for apps linked against Lion or later, the file will be owned by the euid/egid instead. You can temporarily revert to the old behavior for testing purposes by setting __CFPREFERENCES_USE_OLD_UID_BEHAVIOR to a non-null value in your environment. This will primarily impact applications with privileged helper tools that write to the preferences of non-privileged apps such as the Dock or the parent process.

NSProcessInfo Thread Safety

NSProcessInfo is now threadsafe. Additionally it now correctly protects its encapsulation; this may lead to different object lifetimes for the return values of -arguments, -environment, etc... If your code is following the normal Cocoa memory management rules correctly, this won't matter for you.

NSProcessInfo Support for Automatic Termination

Automatic Termination is a new facility to allow processes to be terminated by the system automatically. For instance, automatic termination can used to kill hidden document-based apps, or apps with no visible windows, if the system encounters memory pressure. Note that automatic termination is usually used only in cases where there is no obvious state change to the user, and the state can be restored automatically if needed. So a hidden document based app which was automatically terminated would be brought back with its state intact if the user caused the app to be reactivated.
NSProcessInfo has two additional methods to allow fine-grained control of the new system-managed termination of applications feature.

```
- (void) disableAutomaticTermination:(NSString *)reason;
- (void) enableAutomaticTermination:(NSString *)reason;
```

NSXMLParser Stream API

NSXMLParser has a new method, -initWithStream:. This allows constructing an NSXMLParser that will parse data incrementally as it appears, rather than collecting all available data and then parsing it all at once. This can very significantly reduce the peak amount of memory used by the parser. -initWithContentsOfURL: has been modified to use streams internally when parsing file:// URLs.

NSXMLParser Thread Safety

NSXMLParser is now threadsafe. However, it is not reentrant on a given thread; don't call -parse on an NSXMLParser from within a delegate callback of another NSXMLParser.

NSXMLDocument External Entity Restriction API

External entities in XML files can be a security concern (see http://www.securityfocus.com/archive/1/297714/2002-10-27/2002-11-02/0 for an example). To mitigate this concern, NSXMLDocument has new API for controlling how external entities are loaded.
The following mutually exclusive options can be passed when creating the NSXMLDocument:

`NSXMLNodeLoadExternalEntitiesAlways`

This is identical to the behavior in OS X 10.6 and earlier

`NSXMLNodeLoadExternalEntitiesSameOriginOnly`

This only applies when a URL has been provided. It loads entities with target URLs that match the host, scheme, and port of the document URL.

`NSXMLNodeLoadExternalEntitiesNever`

This disables loading external entities

If no options are specified, the system-default behavior is used. For applications linked on OS X 10.6 and earlier, this is NSXMLNodeLoadExternalEntitiesAlways. For applications linked on 10.7 or later, all entities that don't require network access are loaded.
If an external entity fails to load, the document is invalid and the parse is aborted with an error. Since many uses of NSXMLDocument expect a small set of known external entities (DTDs being the most common), a new init method has been added that accepts an NSSet of URLs that always load regardless of the above options:

```
- (id)initWithData:(NSData *)data options:(NSUInteger)mask validExternalEntityURLs:(NSSet *)validURLs error:(NSError **)error;
```

NSBundle -bundlePath/-bundleURL fixes

For applications linked on Lion or later, NSBundle now returns full paths for -bundlePath/-bundleURL even if the bundle was created with a relative path. Additionally, NSBundle no longer directly returns internal state from -bundlePath, which results in different object lifetimes for those return values. If your code is following the normal Cocoa memory management rules correctly, this won't matter for you.

NSBundle App Store Receipt Support

NSBundle now has API for retrieving the URL to the location for the receipt file from the App Store:

```
- (NSURL *)appStoreReceiptURL;
```

This allows you to use the receipt file without hard-coding relative paths in your bundle. Note that this will always return a URL, even if the receipt file is not present.

NSFileHandle block-based readability/writeability API

NSFileHandle now has the following new API:

```
@property (copy) void (^readabilityHandler)(NSFileHandle *);
@property (copy) void (^writeabilityHandler)(NSFileHandle *);
```

The handler blocks will be called when the underlying file descriptor for the NSFileHandle in question becomes readable or writeable respectively. They will be called on an implementation-defined queue, so if you need them to execute in a particular context you should arrange for them to do that (via dispatch_sync or similar). Note that as soon as the handler returns, it will be called again if the file descriptor being monitored continues to be writeable; so, if you only want to write once, you should set the writeabilityHandler to nil before returning from the block. To avoid retain cycles, handler blocks should access their associated NSFileHandle via their argument, not by capturing it.

NSRegularExpression

NSRegularExpression is a new class used to represent and apply regular expressions. An instance of this class is an immutable representation of a compiled regular expression pattern and various option flags. The pattern syntax currently supported is that specified by ICU (described at http://userguide.icu-project.org/strings/regexp). The supported options are as follows:

```
enum {
   NSRegularExpressionCaseInsensitive             = 1 << 0, // Match letters in the pattern independent of case.
   NSRegularExpressionAllowCommentsAndWhitespace = 1 << 1, // Ignore whitespace and #-prefixed comments in the pattern.
   NSRegularExpressionIgnoreMetacharacters        = 1 << 2, // Treat the entire pattern as a literal string.
   NSRegularExpressionDotMatchesLineSeparators    = 1 << 3, // Allow . to match any character, including line separators.
```

```
   NSRegularExpressionAnchorsMatchLines          = 1 << 4, // Allow ^ and $ to match the start and end of lines.
   NSRegularExpressionUseUnixLineSeparators      = 1 << 5, // Treat only \n as a line separator (otherwise, all standard
                                                           //   line separators are used).
   NSRegularExpressionUseUnicodeWordBoundaries   = 1 << 6  // Use Unicode TR#29 to specify word boundaries (otherwise,
                                                           //   traditional regular expression word boundaries are used).
};
typedef NSUInteger NSRegularExpressionOptions;
```

The fundamental matching method on NSRegularExpression is a block iterator. There are several additional convenience methods, for returning all matches at once, the number of matches, the first match, or the range of the first match. Each match is specified by an instance of NSTextCheckingResult (of type NSTextCheckingTypeRegularExpression) in which the overall match range is given by the range property (equivalent to rangeAtIndex:0) and any capture group ranges are given by rangeAtIndex: for indexes from 1 to numberOfCaptureGroups. {NSNotFound, 0} is used if a particular capture group does not participate in the match. Note that some regular expressions may successfully match zero-length ranges, in which case the location of the range would be significant; this differs from the case of matching literal strings, in which a length of 0 would imply a failure to match. The options that may be supplied to matching methods are as follows:

```
enum {
   NSMatchingReportProgress           = 1 << 0,  // Call the block periodically during long-running match operations.
   NSMatchingReportCompletion         = 1 << 1,  // Call the block once after the completion of any matching.
   NSMatchingAnchored                 = 1 << 2,  // Limit matches to those at the start of the search range.
   NSMatchingWithTransparentBounds    = 1 << 3,  // Allow matching to look beyond the bounds of the search range.
   NSMatchingWithoutAnchoringBounds   = 1 << 4   // Prevent ^ and $ from automatically matching the beginning and end of the se
};
typedef NSUInteger NSMatchingOptions;
```

By default, the block iterator method calls the block precisely once for each match, with a non-nil result and appropriate flags. The client may then stop the operation by setting the contents of stop to YES. If the NSMatchingReportProgress option is specified, the block will also be called periodically during long-running match operations, with nil result and NSMatchingProgress set in the flags, at which point the client may again stop the operation by setting the contents of stop to YES. If the NSMatchingReportCompletion option is specified, the block will be called once after matching is complete, with nil result and NSMatchingCompleted set in the flags, plus any additional relevant flags from among NSMatchingHitEnd, NSMatchingRequiredEnd, or NSMatchingInternalError. NSMatchingReportProgress and NSMatchingReportCompletion have no effect for methods other than the block iterator. The flags that may be passed to the block are as follows:

```
enum {
   NSMatchingProgress      = 1 << 0,  // Set when the block is called to report progress during a long-running match operation
   NSMatchingCompleted     = 1 << 1,  // Set when the block is called after completion of any matching.
   NSMatchingHitEnd        = 1 << 2,  // Set when the current match operation reached the end of the search range.
   NSMatchingRequiredEnd   = 1 << 3,  // Set when the current match depended on the location of the end of the search range.
   NSMatchingInternalError = 1 << 4   // Set when matching failed due to an internal error.
};
typedef NSUInteger NSMatchingFlags;
```

NSMatchingHitEnd is set in the flags passed to the block if the current match operation reached the end of the search range. NSMatchingRequiredEnd is set in the flags passed to the block if the current match depended on the location of the end of the search range. NSMatchingInternalError is set in the flags passed to the block if matching failed due to an internal error (such as an expression requiring exponential memory allocations) without examining the entire search range.

NSMatchingAnchored, NSMatchingWithTransparentBounds, and NSMatchingWithoutAnchoringBounds can apply to any match or replace method. If NSMatchingAnchored is specified, matches are limited to those at the start of the search range. If NSMatchingWithTransparentBounds is specified, matching may examine parts of the string beyond the bounds of the search range, for purposes such as word boundary detection, lookahead, etc. If NSMatchingWithoutAnchoringBounds is specified, ^ and $ will not automatically match the beginning and end of the search range (but will still match the beginning and end of the entire string). NSMatchingWithTransparentBounds and NSMatchingWithoutAnchoringBounds have no effect if the search range covers the entire string.

NSRegularExpression is designed to be immutable and threadsafe, so that a single instance can be used in matching operations on multiple threads at once. However, the string on which it is operating should not be mutated during the course of a matching operation (whether from another thread or from within the block used in the iteration).

NSRegularExpression also provides find-and-replace methods for both immutable and mutable strings. The replacement is treated as a template, with $0 being replaced by the contents of the matched range, $1 by the contents of the first capture group, and so on. Additional digits beyond the maximum required to represent the number of capture groups will be treated as ordinary characters, as will a $ not followed by digits. Backslash will escape both $ and itself. For clients implementing their own replace functionality, there is also a method to perform the template substitution for a single result, given the string from which the result was matched, an offset to be added to the location of the result in the string (for example, in case modifications to the string moved the result since it was matched), and a replacement template.

Regular Expression Additions to NSString

In addition to the NSRegularExpression class, some convenience functionality has been added for using regular expressions with NSString directly. This takes the form of an additional option in NSStringCompareOptions, NSRegularExpressionSearch. The NSRegularExpressionSearch option applies only to methods of the form rangeOfString:..., stringByReplacingOccurrencesOfString:..., or replaceOccurrencesOfString:..., and it precludes all other options except for NSCaseInsensitiveSearch and NSAnchoredSearch.

The behavior is equivalent to that of NSRegularExpression with no options specified, except NSRegularExpressionCaseInsensitive (if NSCaseInsensitiveSearch is specified), or NSMatchingAnchored (if NSAnchoredSearch is specified). Note that some regular expressions may successfully match zero-length ranges, in which case the location of the range would be significant; this differs from the case of matching literal strings, in which a length of 0 would imply a failure to match.

In the stringByReplacingOccurrencesOfString:... and replaceOccurrencesOfString:... methods, the replacement string is treated as a template, as in the corresponding NSRegularExpression methods, with $0 being replaced by the contents of the matched range, $1 by the contents of the first capture group, and so on. +[NSRegularExpression escapedTemplateForString:] can be used to escape a string so that template characters will be treated literally.

NSDataDetector

NSDataDetector is a specialized subclass of NSRegularExpression. Instead of finding matches to regular expression patterns, it matches items identified by Data Detectors, such as dates, addresses, and URLs. The checkingTypes argument should contain one or more of the types NSTextCheckingTypeDate, NSTextCheckingTypeAddress, NSTextCheckingTypeLink, NSTextCheckingTypePhoneNumber, and NSTextCheckingTypeTransitInformation. The NSTextCheckingResult instances returned will be of the appropriate types from that list.

Additions to NSTextCheckingResult

As part of the effort to support NSRegularExpression and NSDataDetector, there are some additions to NSTextCheckingResult. First, there is a new type, NSTextCheckingTypeRegularExpression, used for regular expression results. Second, results may now optionally have additional ranges beyond the overall range they have always had. For this, there is a new property and method:

```
@property (readonly) NSUInteger numberOfRanges;
- (NSRange)rangeAtIndex:(NSUInteger)idx;
```

The numberOfRanges should always be at least 1, and the rangeAtIndex:0 should always match the existing range property.
NSTextCheckingTypeRegularExpression uses rangeAtIndex:n to represent the range corresponding to the nth capture group.
There is a new method

```
- (NSTextCheckingResult *)resultByAdjustingRangesWithOffset:(NSInteger)offset;
```

that allows clients to produce a result modified by offseting all of its ranges, for example to adjust for the offset of a substring within a larger string.

NSTextCheckingTypePhoneNumber and NSTextCheckingTypeTransitInformation are new types for NSTextCheckingResult, for use with NSDataDetector. These have associated with them new creation methods

```
+ (NSTextCheckingResult *)phoneNumberCheckingResultWithRange:(NSRange)range phoneNumber:(NSString *)phoneNumber;
+ (NSTextCheckingResult *)transitInformationCheckingResultWithRange:(NSRange)range components:(NSDictionary *)components;
```

Phone number results have a property

```
@property (readonly) NSString *phoneNumber;
```

and transit information results have a property

```
@property (readonly) NSDictionary *components;
```

with keys currently NSTextCheckingAirlineKey and NSTextCheckingFlightKey for airline flight information. The old addressComponents property for address results is also being replaced by the new components property, although addressComponents will continue to work for compatibility.

NSLinguisticTagger

NSLinguisticTagger is a new class used to automatically segment natural-language text and tag it with information such as parts of speech. An instance of this class is assigned a string to tag, and clients can then obtain tags and ranges for tokens in that string appropriate to a given tag scheme.

A number of tag schemes are currently available: NSLinguisticTagSchemeTokenType, which is a tag scheme that classifies tokens according to their broad type: word, punctuation, whitespace, etc.; NSLinguisticTagSchemeLexicalClass, which classifies tokens according to class: part of speech for words, type of punctuation or whitespace, etc.; NSLinguisticTagSchemeNameType, which classifies tokens as to whether they are part of named entities of various types or not; NSLinguisticTagSchemeNameTypeOrLexicalClass, which follows NSLinguisticTagSchemeNameType for names, NSLinguisticTagSchemeLexicalClass for all other tokens; NSLinguisticTagSchemeLemma, which supplies a stem form for each word token (if known); NSLinguisticTagSchemeLanguage, which tags tokens according to their most likely language (if known); and NSLinguisticTagSchemeScript, which tags tokens according to their script. NSLinguisticTagSchemeTokenType, NSLinguisticTagSchemeLexicalClass, NSLinguisticTagSchemeNameType, and NSLinguisticTagSchemeNameTypeOrLexicalClass use tags from a specified list of string constants (clients may use == comparison). Tags for NSLinguisticTagSchemeLemma are lemmas from the language. Tags for NSLinguisticTagSchemeLanguage are standard language abbreviations. Tags for NSLinguisticTagSchemeScript are standard script abbreviations.

Not all languages can be handled by the tagger, and not all tag schemes are available for any given language. availableTagSchemesForLanguage: is provided for runtime determination of the schemes available for a particular language. The current languages for which parts-of-speech tagging is available are English, French, and German.

An instance of NSLinguisticTagger is created using initWithTagSchemes:options: with an array of tag schemes. The tagger will be able to supply tags corresponding to any of the schemes in this array. Once a string has been provided to the tagger using setString:, the tagger will segment the string as needed into sentences and tokens, and return those ranges along with a tag for any scheme in its array of tag schemes. The fundamental tagging method on NSLinguisticTagger is a block iterator, enumerateTagsInRange:scheme:options:usingBlock:, that iterates over all tokens intersecting a given range, supplying tags and ranges. There are several additional convenience methods, for obtaining a sentence range, information about a single token, or for obtaining information about all tokens intersecting a given range at once, in arrays.

Option flags are available that allow clients to control which types of tokens are returned in the results. By default all tokens will be returned, but if the options flag corresponding to omitting a given type is specified, then tokens of that general type (word, punctuation, whitespace, or other) will be omitted from the tokens returned in the results. For NSLinguisticTagSchemeNameType and NSLinguisticTagSchemeNameTypeOrLexicalClass, there is an option controlling whether a multi-word name will be returned as a single token, or as a separate token for each word of the name (the default).

If clients know the orthography for a given portion of the string, they may supply it to the tagger via setOrthography:range:. Otherwise, the tagger will infer the language from the contents of the text. If the string attached to the tagger is mutable, the client must inform the tagger whenever the string changes, via stringEditedInRange:changeInLength:. A given instance of NSLinguisticTagger should not be used from more than one thread simultaneously.

There are also some related convenience APIs on NSString. Clients wishing to analyze a given string once may use these NSString APIs without having to create an instance of NSLinguisticTagger. If more than one tagging operation is needed on a given string, it is more efficient to use an explicit NSLinguisticTagger instance.

NSURLConnection

The NSURLConnection class now has queue support, enabling delegate callbacks without requiring the running of a runloop.
This method supports performing asynchronous loading of URL requests where the results of the request are delivered to a block via an NSOperationQueue:

```
- (void)setDelegateQueue:(NSOperationQueue *)queue;
```

This is a convenience routine that allows for asynchronous loading of a URL-based resource.

```
+ (void)sendAsynchronousRequest:(NSURLRequest *)request
             queue:(NSOperationQueue *)queue
               completionHandler:(void (^)(NSURLResponse *, NSData *, NSError *))handler;
```

There is a new delegate method for NSURLConnection:

```
- (void)connection: (NSURLConnection *) connection
        willSendRequestForAuthenticationChallenge:(NSURLAuthenticationChallenge *)challenge;
```

This new delegate combines the functionality of the existing three delegate methods didReceiveAuthenticationChallenge:, canAuthenticateAgainstProtectionSpace:, and connectionShouldUseCredentialStorage: . This delegate is always called when a challenge is given, even when credentials exist on the system that could be used to satisfy the current challenge. This gives the delegate the opportunity to validate the credentials being used (if any) and allow, deny, or provide alternate credentials for the current transaction.
If a delegate implements this new callback, the delegate callbacks didReceiveAuthenticationChallenge:, canAuthenticateAgainstProtectionSpace:, and connectionShouldUseCredentialStorage: will not be called.
In association to the new delegate callback on NSURLConnection, the protocol NSURLAuthenticationChallengeSender has been extended to include two new optional methods:

```
@optional
- (void)performDefaultHandlingForAuthenticationChallenge:(NSURLAuthenticationChallenge *)challenge;
- (void)rejectProtectionSpaceAndContinueWithChallenge:(NSURLAuthenticationChallenge *)challenge;
```

These two new protocol methods of NSURLAuthenticationChallengeSender allow implementors of NSURLConnection's delegate callback, willSendRequestForAuthenticationChallenge:, to direct NSURLConnection to perform default processing of the given authentication challenge. This allows NSURLConnection to handle new authentication protocols and delegates of NSURLConnection do not have to have special knowledge of the new authentication protocols. Additionally, calling rejectProtectionSpaceAndContinueWithChallenge: from willSendRequestForAuthenticationChallenge: will skip the current NSURLProtectionSpace included in the offered NSURLAuthenticationChallenge object. If another NSURLProtectionSpace exists for the NSURLAuthenticationChallenge, the willSendRequestForAuthenticationChallenge: will be called again with the new NSURLProtectionSpace in the offered challenge parameter. If there are no more NSURLProtectionSpace objects, NSURLConnection will treat this similar to calling cancel: on the challenge.

NSURLRequest

NSURLRequest has a new method for network service types:

```
- (NSURLRequestNetworkServiceType)networkServiceType;
```

```
- (void)setNetworkServiceType:(NSURLRequestNetworkServiceType)networkServiceType;
```

Network Service Types provide the networking subsystems a hint as to the intended purpose of the request. The current list of network services types includes voice-control, voice-data, video, and background.
Support for HTTP pipelining has been added:

```
- (BOOL)HTTPShouldUsePipelining;
- (void)setHTTPShouldUsePipelining:(BOOL)shouldUsePipelining;
```

HTTP Pipelining allows for the transmission of multiple concurrent HTTP requests without waiting for a response, which improves performance over high-latency networks.
Enabling pipelining is a hint which may be discarded when the server or proxy involved does not support pipelining.

NSHTTPCookieStorage

There is a new method

```
- (NSArray *)sortedCookiesUsingDescriptors:(NSArray *)sortOrder;
```

that avoids expensive string conversion and supports sorting the cookies based on a sort order. This method is preferred over the more generic:

```
- (NSArray *)cookies;
```

Legacy OS X 10.4 predicate parser removed

The legacy predicate parser shipped with OS X 10.4 (Tiger) has been removed; this will not affect applications linked on or after OS X 10.5 (Leopard). In applications linked against OS X 10.4, this could result in slightly different behavior for predicates using the BETWEEN and CONTAINS operators, and addition of several reserved words to the predicate grammar. Please file a bug if this negatively affects you or an application you use.

New collection class

There is a new collection class in Foundation, NSOrderedSet. An NSOrderedSet is an ordered collection, accessed by index, like an NSArray, but also offers NSSet-like operations.
NSOrderedSet has 3 primitives, two of which are the same as NSArray:

```
- (NSUInteger)count;
- (id)objectAtIndex:(NSUInteger)idx;
```

and adds a third:

```
- (NSUInteger)indexOfObject:(id)object;
```

which is the basis of the Set operations. The latter returns the index of the given object, as you might guess. All of these [in a subclass] must be quick "constant time" operations for NSOrderedSet to perform well.
The -isEqual: method is the basis for object comparison (e.g., is an object in the ordered set or not). The -hash method must also be well-implemented on the objects put in an ordered set, and the hash/isEqual invariant maintained, since some ordered set implementations may wish to use hashing techniques to implement the primitives.
NSMutableOrderedSet has 3 additional primitives, which mostly act like NSMutableArray's primitives:

```
- (void)insertObject:(id)object atIndex:(NSUInteger)idx;
- (void)replaceObjectAtIndex:(NSUInteger)idx withObject:(id)object;
- (void)removeObjectAtIndex:(NSUInteger)idx;
```

The differences are that insertObject:atIndex: does nothing if a matching object is already in the NSMutableOrderedSet, and replaceObjectAtIndex:withObject: does nothing if a matching object is already in the ordered set at a different index.
The other available methods operate by using the primitives, of course, in a way that should seem to naturally follow from the behavior of the primitive.
There are two methods in NSOrderedSet that warrant special attention:

```
- (NSArray *)array;
- (NSSet *)set;
```

These two methods return proxy or facade objects for the underlying ordered set, acting like, respectively, an immutable array or immutable set. These are useful in situations where you have an NSOrderedSet but need an NSArray to pass into some API. Note that these returned facades are not copies of the ordered set, and while you cannot change an ordered set through them, if the original ordered set is mutable, direct mutations to it will show through the facade objects and the "immutable" collection will appear to spontaneously be changing. The effect of apparent changes to those objects, to code which has received them, can be subtle. A simple example would be this:

```
[aMutableOrderedSet removeObjectsInArray:[aMutableOrderedSet array]]
```

Here, the array object given as parameter to removeObjectsInArray: will be changing while removeObjectsInArray: is iterating over it, possibly causing a crash or plain mis-behavior. The safer thing, when you know a mutable ordered set is the receiver of -array or -set, and you know it will be mutating further, is to make a copy of the ordered set into an array or set.
Currently the simplest way to make an NSArray from an NSOrderedSet is to copy the return value of -array:

```
[[anOrderedSet array] copy];
```

Currently the simplest way to make an NSSet from an NSOrderedSet is to copy the return value of -set:

```
[[anOrderedSet set] copy];
```

Distributed notification delivery

If you want a posted distributed notification to be received immediately, be sure you are passing the NSNotificationSuspensionBehaviorDeliverImmediately suspension behavior flag when registering for the notification, or using the NSNotificationDeliverImmediately flag when posting. Bugs in OS X releases prior to 10.7 meant that sometimes a distributed notification would get delivered through to suspended observers, and not be properly queued, even when those flags weren't used.

NSCalendar

The NSCalendar algorithms have several differences depending on which OS SDK version you build your app against. -rangeOfUnit:inUnit:forDate: can produce materially different results for OS X 10.4, OS X 10.5, and OS X 10.6/10.7. -ordinalityOfUnit:inUnit:forDate: can produce materially different results for pre-OS X 10.6 and OS X 10.6/10.7. -dateFromComponents: can produce different results on OS X 10.7 than previously when working with Week units.
But of course, most of the algorithms can produce different results between OS releases, as bugs are fixed or changes made, either in Foundation or CoreFoundation or in libraries beneath them. The differences noted in the previous paragraph are the significant ones.
There are 3 new Week-related unit/component constants in NSCalendar:
NSWeekOfMonthCalendarUnit, NSWeekOfYearCalendarUnit, NSYearForWeekOfYearCalendarUnit
and the values of these quantities are week numbers or amounts relating to the month or the year that the week is in, and the year number for week-based interpretations of a calendar, such as the ISO 8601 calendar.
For many operations, there is no difference in meaning between NSWeekOfMonthCalendarUnit and NSWeekOfYearCalendarUnit, but for some there is a difference. Obviously getting the ordinality of a week with the month will usually give a different answer than the ordinality of the week within the year. The NSWeekCalendarUnit constant is not yet officially deprecated, but its use in new code is discouraged. It has a behavior like either NSWeekOfMonthCalendarUnit or NSWeekOfYearCalendarUnit, as the case may be, to give it behavior like it had prior to OS X 10.7.

iCloud

Foundation in 10.7 includes APIs to enable applications to store configuration information and documents in the iCloud:
NSFileManager has APIs to put a document in the cloud or take it out, and to explicitly initiate a download.
NSMetadataQuery provides APIs to enumerate documents in the cloud.
NSURL has additional keys to query attributes of documents in the cloud, even if the documents have not yet been downloaded.
NSFileVersion enables querying information about different versions of a document, including those which had conflicts as a result of synchronizing changes from the cloud. Applications can use choose to do more sophisticated conflict resolution. Also note that the NSDocument versions browser shows these versions, including versions which had conflicts but were determined to be older and thus set aside. In apps which support Auto Save, users can utilize the versions browser and restore these "conflict losers" partly or wholesale.
Finally, NSUbiquitousKeyValueStore allows saving configuration information in the cloud. This should typically be limited to small amount of data, such as high scores, user settings, etc.

Strings File Handling

In OS X Lion, strings files in the system have been converted to binary property list format for performance reasons. It's rare for strings files to be accessed directly (without going through CF/NSBundle), and it's even less common for the system strings files to be accessed directly by third party applications; however, if they are, then they will no longer open properly if accessed with -[NSString propertyListFromStringsFileFormat] or -[NSString propertyList], which first require loading the file in as an NSString, the converting to a property list. Instead the APIs in NSPropertyListSerialization should be used to open strings files directly.
NSString methods -propertyList and -propertyListFromStringsFileFormat will be deprecated in a future release.

Notes specific to Mac OS X 10.6

NSKeyedArchiver

Prior to 10.6, the archiver parameter to replacementObjectForKeyedArchiver: was always nil. For applications linked on or after 10.6, this parameter will be populated with the archiver as expected.

NSPropertyList

The following methods on NSPropertyListSerialization are now obsolete and will be deprecated in a future release:

```
+ (NSData *)dataFromPropertyList:(id)plist format:(NSPropertyListFormat)format errorDescription:(NSString **)errorString;
+ (id)propertyListFromData:(NSData *)data mutabilityOption:(NSPropertyListMutabilityOptions)opt
                    format:(NSPropertyListFormat *)format errorDescription:(NSString **)errorString;
```

The replacement methods are:

```
+ (NSData *)dataWithPropertyList:(id)plist format:(NSPropertyListFormat)format options:(NSPropertyListWriteOptions)opt error:(
+ (id)propertyListWithData:(NSData *)data options:(NSPropertyListReadOptions)opt format:(NSPropertyListFormat *)format error:(
+ (NSInteger)writePropertyList:(id)plist toStream:(NSOutputStream *)stream format:(NSPropertyListFormat)format
                       options:(NSPropertyListWriteOptions)opt error:(NSError **)error;
+ (id)propertyListWithStream:(NSInputStream *)stream options:(NSPropertyListReadOptions)opt
                      format:(NSPropertyListFormat *)format error:(NSError **)error;
```

The new methods provide better error handling, better conformance with the standard memory management rules, and better support for localization.

On the new methods, if the error parameter is non-NULL and an error occurs then it will be set to an autoreleased NSError describing the problem.

On the dataWithPropertyList:format:options:error: and writePropertyList:toStream:format:options:error: methods, the NSPropertyListWriteOptions parameter is currently unused and should be set to 0. The format option should be set to a NSPropertyListFormat.

On the propertyListWithData:options:format:error: and propertyListWithStream:options:format:error: methods, the NSPropertyListReadOptions parameter should be set to one of the NSPropertyListMutabilityOptions. If the format parameter is non-NULL, it will be filled out with the format of the read property list.

Bug Fix in NSAppleScript Running Garbage-Collected

In Mac OS 10.5 there was a bad bug in NSAppleScript that rendered it effectively incompatible with garbage collection. The symptoms were frequent crashes and spurious returning of errors. This bug has been fixed since Mac OS 10.5.3.

Bug Fix in -[NSUndoManager prepareWithInvocationTarget:] (New since November seed)

In earlier versions of Mac OS -[NSUndoManager prepareWithInvocationTarget:] always returned the receiving NSUndoManager. Because of the way Objective-C message forwarding works this meant that undo actions would not be recorded for messages which NSUndoManager itself can handle including, importantly, all messages implemented by NSObject or any category of NSObject. Instead the messages would just be handled by the NSUndoManager. This bug has been fixed in Mac OS 10.6.

Bug Fix in .sdef-Declared Scriptability

In Mac OS 10.4 and Mac OS 10.5 Cocoa Scripting's support for the "number" type was incomplete even though this type is one of the .sdef primitive types. The symptom of this was exceptions with descriptions that said things like "This instance of the class 'NSCFNumber' doesn't respond to -scriptingNumberDescriptor messages" and "+[NSNumber scriptingNumberWithDescriptor:]: unrecognized selector sent to class 0x26a7560." This bug has been fixed.

Bug Fix in General Scriptability

In all previous versions of Mac OS X -[NSTextStorage(NSScripting) words] was easily confused by numbers followed immediately by punctuation. The result was that invoking the method wouldn't return all of the words that it should, which of course affected the results of telling apps like TextEdit to return things like the "words of the front document." This bug has been fixed.

Sudden Termination - Fast Killing Of Applications (Updated since November seed)

Mac OS 10.6 includes a new mechanism that allows the operating system to log out or shut down more quickly by, whenever possible, killing applications instead of requesting that they quit themselves. Two new methods have been added to NSProcessInfo:

```
- (void)disableSuddenTermination;
- (void)enableSuddenTermination;
```

These methods disable or reenable the ability to be quickly killed. The default implementations of these methods increment or decrement, respectively, a counter whose value is 1 when the process is first created. When the counter's value is 0 the application is considered to be safely killable and may be killed by the operating system without any notification or event being sent to the process first. If an application's Info.plist has an NSSupportsSuddenTermination entry whose value is true then NSApplication invokes -enableSuddenTermination automatically during application launch, which typically renders the process killable right away. You can also manually invoke -enableSuddenTermination right away in, for example, agents or daemons that don't depend on AppKit. After that, you can invoke these methods whenever the process has work it must do before it terminates.

For example:
- NSUserDefaults uses these to prevent process killing between the time at which a default has been set and the time at which the preferences file including that default has been written to disk.
- NSDocument uses these to prevent process killing between the time at which the user has made a change to a document and the time at which the user's change has been written to disk.
- You can use these whenever your application defers work that must be done before the application terminates. If for example your application ever defers writing something to disk, and it has an NSSupportsSuddenTermination entry in its Info.plist so as not to contribute to user-visible delays at logout or shutdown time, it must invoke -disableSuddenTermination when the writing is first deferred and -enableSuddenTermination after the writing is actually done.

Notice that -enableSuddenTermination is used to inform the operating system that the process can participate in the fast killing mechanism in the first place, and that it will be automatically invoked once for that purpose if you put an NSSupportsSuddenTermination entry whose value is true in your application's Info.plist. Notice that it is also used to balance previous invocations of -disableSuddenTermination.

Debugging tip: you can find out the value of the counter mentioned in the above comment by executing 'print (long)[[NSClassFromString(@"NSProcessInfo") processInfo] _suddenTerminationDisablingCount]' in gdb (using the Xcode Debugger Console, for instance). Do not attempt to invoke or override -_suddenTerminationDisablingCount in your application. It is there just for this debugging purpose, and may disappear at any time.

Instruments also has an instrument that lets you track invocations of these methods.

Automatic Removal of Finalized Key-Value Observers When Running Garbage-Collected (New since November seed)

With the addition of support for Objective-C garbage collection to Foundation in Mac OS 10.5 one of the basic rules of key-value observing (KVO) remained, and still applied to garbage-collected applications: all invocations of KVO's -addObserver:forKeyPath:options:context: method must be balanced by invocations of -removeObserver:forKeyPath: or KVO will leak memory. (KVO logs when it senses failure to follow this rule but only when running non-garbage-collected.) The same rule applied to use of KVO's NSArray batched observer registration methods. This resulted in situations in which classes of observers had to have -finalize methods just to do observer deregistration when they otherwise would not have to. -finalize methods are supposed to be rarer than that. In Mac OS 10.6, explicit removal of observers when they're finalized is now never necessary. KVO automatically removes observers as they're collected. Actually, in Mac OS 10.6 all invocations of -[NSObject(NSKeyValueObserverRegistration) removeObserver:forKeyPath:] and -[NSArray(NSKeyValueObserverRegistration) removeObserver:fromObjectsAtIndexes:forKeyPath:] do virtually nothing when either the receiver or the observer is being finalized (so it's not very bad for performance to leave them there in applications that still have to run on Mac OS 10.5).

One thing has not changed in Mac OS 10.6: for performance reasons you should not leave an observer registered with objects that no longer matter at all to the observer. When observing the properties of objects in a collection whose membership changes as the application runs, it's best to follow a pattern of making that collection the value for a to-many relationship of some parent object and adding and removing your observer of the individual objects as they become related and unrelated. See for example Sketch's SKTGraphicView class, in particular when it invokes its own -startObservingGraphics: and -stopObservingGraphics: methods. If you instead assume that observed objects will just go away when there are no references to them other than observation then your application might use more memory than you expect because observed objects will not be collected as soon as you expect. You can depend on KVO to cleanly handle observers and observed objects being collected, and you can depend on KVO to not cause observers to go uncollected, but you cannot depend on an observed object being collected before its observers, even if no other objects reference the observed object.

Bug Fixes in Debugging of Objects Being Deallocated With Observers Still Registered When Not Running Garbage-Collected (Updated since March 2009 Seed)

In Mac OS 10.3 through 10.5 there was a bug in which a valuable debugging feature of KVO was not triggered when an object observing itself did not remove itself as an observer of itself during deallocation. This bug has been fixed in Mac OS 10.6. When this happens Foundation now logs something like "An instance 0x100771010 of class MySelfObservingClass was deallocated while key value observers were still registered with it. Observation info was leaked, and may even become mistakenly attached to some other object. Set a breakpoint on NSKVODeallocateBreak to stop here in the debugger. Here's the current observation info:..."

There was another bug in which this logging was done spuriously when an object observed by a second object was deallocated, its deallocation caused the release of the second object, and the second object correctly unregistered itself as an observer of the first object at that time due to its own deallocation. This bug has also has been fixed in Mac OS 10.6.

To summarize, KVO's test for objects being deallocated with observers still registered exhibited both false negatives and false positives, and both kinds of mistake have been fixed.

New NSPurgeableData class

Mac OS 10.6 includes an NSMutableData subclass called NSPurgeableData that takes advantage of the new purgeable memory feature. NSPurgeableData objects conform to the NSDiscardableContents protocol, whose description is forthcoming.

For NSPurgeableData, if -beginContentAccess returns NO, then the NSPurgeableData's bytes have been purged and the bytes are effectively inaccessible. Note that NSPurgeableData objects are "accessed" upon creation, so -endContentAccess must be called after creation to make the data purgeable.

NSData copyWithZone: changed (New since WWDC 2008)

Prior to Mac OS 10.6, when -copyWithZone: was invoked on any CFData instance through toll-free bridging, the method would always return a new NSData instance containing a copy of the original bytes. For applications linked on or after 10.6, this method will retain and return the original instance, instead of copying, when invoked on an immutable CFData. If you need to create an actual copy of a CFData instance, use CFDataCopy() or -dataWithData:.

-[NSData getBytes:range:] and NSRangeException (New since WWDC 2008)

Prior to Mac OS 10.6, -[NSData getBytes:range:] did not properly raise an NSRangeException for ranges which start inside the NSData's bytes, but end outside them. Instead it filled the provided buffer up to the end of the NSData. For applications linked on or after Mac OS 10.6, this method will now properly raise an NSRangeException.

NSNumberFormatter, NSDateFormatter, and -getObjectValue:forString:errorDescription: (New since WWDC 2008)

Prior to Mac OS 10.6, both NSNumberFormatter's and NSDateFormatter's implementation of -getObjectValue:forString:errorDescription: would return YES and a parsed object value if only part of the string could be parsed. This is problematic because with this API you cannot be sure what portion of the string was parsed. For applications linked on or after Mac OS 10.6, this method instead returns an error if part of the string cannot be parsed. You can use -getObjectValue:forString:range:error: to get the old behavior; this method returns the range of the substring that was successfully parsed.

Formal Protocol Adoption (New since WWDC 2008)

In Mac OS 10.6, Foundation switched to using formal protocols for all delegates to provide better compile-time type checking. Required protocol methods are marked with @required, where possible. The rest are marked with @optional.

The affected classes are
- NSConnection
- NSKeyedArchiver
- NSMetadataQuery
- NSNetService
- NSNetServiceBrowser
- NSPort
- NSMachPort
- NSSpellServer
- NSStream
- NSXMLParser

The changes will introduce new warnings in code using these classes' delegates. Fortunately, the changes needed to correct these warnings are fairly simple.

- Your delegate classes need to declare conformance to the new protocols. For example:

```
@interface MyDelegate : NSObject <NSMetadataQueryDelegate> ... @end
```

- Sending messages to [foo delegate] that are not in the delegate's protocol will generate a warning. This is not generally recommended since there are no guarantees about the delegate's class. However, you may work around this by casting the result of [foo delegate] to id. Also, you should always perform a respondsToSelector: check before invoking a method on [foo delegate] that is not an @required method in the protocol.

- If you have a subclass of one of these classes that adds additional delegate methods you must also create a subprotocol and override -delegate and -setDelegate:. For example:

```
@protocol MyStreamDelegate;

@interface MyStream : NSStream

- (id <MyStreamDelegate)delegate;
- (void)setDelegate:(id <MyStreamDelegate>)delegate;

@end

@protocol MyStreamDelegate <NSStreamDelegate>
- (void)additionalDelegateMethod;
@end

@implementation MyStream

- (id <MyStreamDelegate)delegate {
    return (id <MyStreamDelegate>)[super delegate];
}

- (void)setDelegate:(id <MyStreamDelegate>)delegate {
    [super setDelegate:delegate];
}

@end
```

- If you need to target Leopard or Tiger with the same sources, you should conditionally declare empty protocols, or else the compiler will complain about missing protocols declarations. For example:

```
#if MAC_OS_X_VERSION_10_6 > MAC_OS_X_VERSION_MAX_ALLOWED

@protocol NSConnectionDelegate <NSObject> @end

#endif
```

`

Deprecating unsafe buffer-taking methods (New since November seed)

The following methods will deprecated in the next release of Mac OS.

```
-[NSString getCharacters:]
-[NSData getBytes:]
-[NSArray getObjects:]
```

These have been identified as unsafe. Typically, the buffers passed in to these methods are sized relative to the result from -count or -length. However, it is possible for the receiver to be mutated from another thread between these method calls. This may potentially cause a buffer overrun. To prevent this, you should use the following methods instead:

```
-[NSString getCharacters:range:]
-[NSData getBytes:length:] or -[NSData getBytes:range:]
-[NSArray getObjects:range:]
```

NSXMLParser fatal errors (New since November seed)

Before Mac OS 10.6, NSXMLParser would sometimes try to continue parsing after a fatal error. For applications linked on SnowLeopard or later, NSXMLParser will now abort parsing after reporting a fatal error.

For compatibility's sake, you can restore the pre-SnowLeopard behavior by setting the environment variable NSXMLParserContinueParsingAfterFatalError to YES. However, you should eliminate any dependency on this behavior as soon as possible.

NSData rangeOfData:withOptions:range: (New since November seed)

NSData now has a method that will efficiently search its contents:

```
- (NSRange)rangeOfData:(NSData *)dataToFind
              options:(NSDataSearchOptions)mask
                range:(NSRange)searchRange;
```

The semantics are nearly identical to NSString's rangeOfString:withOptions:range:. The only significant difference is the lack of 'case insensitive' and 'literal' search options, which are only applicable to strings.

NSFileManager (New since November seed)

The following methods on NSFileManager now throw exceptions in Mac OS 10.6:

```
- (BOOL)copyItemAtPath:(NSString *)srcPath toPath:(NSString *)dstPath error:(NSError **)error;
- (BOOL)moveItemAtPath:(NSString *)srcPath toPath:(NSString *)dstPath error:(NSError **)error;
- (BOOL)linkItemAtPath:(NSString *)srcPath toPath:(NSString *)dstPath error:(NSError **)error;

- (BOOL)copyItemAtURL:(NSURL *)srcURL toURL:(NSURL *)dstURL error:(NSError **)error;
- (BOOL)moveItemAtURL:(NSURL *)srcURL toURL:(NSURL *)dstURL error:(NSError **)error;
- (BOOL)linkItemAtURL:(NSURL *)srcURL toURL:(NSURL *)dstURL error:(NSError **)error;
```

For all applications, these methods will throw an NSInvalidArgumentException if dstPath or dstURL is nil. For applications linked on Mac OS 10.6 or later, these methods will throw an NSInvalidArgumentException if srcPath or srcURL nil.

NSString path completion (New since WWDC 2008)

NSString's completePathIntoString:caseSensitive:matchesIntoArray:filterTypes: method previously did not correctly take the case-sensitive flag into account (specifically, much of the behavior defaulted to case-sensitive even if the flag was set to NO). This is now fixed; the entire path is matched case-insensitively if specified. (Note that on a case-insensitive filesystem like most HFS+ drives, case-sensitive completion will still match the path case-insensitively; only the last component will be matched case-sensitively. This is a limitation of the filesystem.)

NSDecimal (New since WWDC 2008)

NSDecimal and NSDecimalNumber now correctly report the zeroth power of a negative number (e.g. [someNegative decimalNumberByRaisingToPower:0]) to be 1, instead of -1.

NSIndexSet (New since WWDC 2008)

NSIndexSet can now safely be used on multiple threads. Like other collection classes, though, mutations to index sets are not thread-safe, so you must still synchronize access to any index sets that may change. If you want to ensure an index set is immutable, you should make an immutable copy.

Text Checking (Updated since WWDC 2008)

Snow Leopard includes a new facility known as text checking, which provides a unified interface for a variety of types of checking, including spell checking, grammar checking, URL detection, smart quotes, date and address detection via Data Detectors, and others. Among other things, this facility allows for the automatic identification of the languages used in a piece of text, so that spellchecking can proceed without the user having to label text as to language.

The basic interface for this facility is now available in AppKit via NSSpellChecker (see the AppKit release notes for details). Foundation contains two new classes intended to be used as part of this feature: NSTextCheckingResult and NSOrthography.

An instance of NSTextCheckingResult represents something that has been found during checking--a misspelled word, a sentence with grammatical issues, a detected URL, a straight quote to be replaced with curly quotes, and so forth. This is an immutable value class intended to be passed back as elements of the array returned to clients of the text checking APIs. Each instance has at a minimum a checking type showing the sort of item noted, and a range within the string being checked to which the result applies. There are currently 10 system-defined checking types, with space reserved for 22 more, and for 32 additional custom text checking types to be defined by clients. NSTextCheckingResult itself is a semi-abstract superclass; any custom result types would usually define a suitable subclass to hold the appropriate information.

NSOrthography is a class used to describe the linguistic content of a piece of text, especially for the purposes of spelling and grammar checking. It describes (a) which scripts the text contains, (b) a dominant language and possibly other languages for each of these scripts, and (c) a dominant script and language for the text as a whole. This is an immutable value class intended to be passed back as part of the results of automatic language identification, or passed in as a starting point for that identification.

For the purposes of NSOrthography, scripts are uniformly described by standard four-letter tags (Latn, Grek, Cyrl, etc.) with the supertags Jpan and Kore typically used for Japanese and Korean text, Hans and Hant for simplified and traditional Chinese text; the tag Zyyy is used if a specific script cannot be identified. Languages are uniformly described by BCP-47 tags, preferably in canonical form; the tag und is used if a specific language cannot be determined.

In addition, there is a new delegate method on NSSpellServer, which an NSSpellServer delegate in a spelling checker can use to perform spelling and grammar checking simultaneously when so requested by the new text checking methods on NSSpellChecker, as well as specifying autocorrection results.

```
- (NSArray *)spellServer:(NSSpellServer *)sender
```

```
checkString:(NSString *)stringToCheck
     offset:(NSUInteger)offset
      types:(NSTextCheckingTypes)checkingTypes
    options:(NSDictionary *)options
orthography:(NSOrthography *)orthography
  wordCount:(NSInteger *)wordCount;
```

The return value should be an array of NSTextCheckingResult objects, and the other arguments generally correspond to those for the new NSSpellChecker methods. The results should be of the orthography, spelling, grammar, or correction types, as specified by the checkingTypes. The string passed in to this method may be a substring of the string passed in to NSSpellChecker, and the offset argument represents the offset of that substring within the entire string; it should be added to the origin of the range for any NSTextCheckingResult returned. The orthography argument represents the identified orthography of the string being passed in to this method.

Learned Words (New since WWDC 2008)

The format for the learned words lists used by the spellchecker has not previously been documented. There are two types of learned word lists, both of which are stored as UTF-8 plain text documents in ~/Library/Spelling. The first type consists of those words learned specifically in the context of a particular language; these are stored in files named by the abbreviation for that language as specified in the NSLanguages array in the spellchecker's Info.plist. The second type consists of those words learned outside of the context of a particular language; these are stored in a single file named LocalDictionary. In any case, the files consist of lists of learned words (case-insensitive), one per line, separated by newlines (\n, U+000A). (Embedded nulls can also be used in place of the newlines, and were so used prior to Snow Leopard, but newlines are now preferred.) If a particular word appears more than once in a given file, it is treated as learned if and only if it appears an odd number of times. The spell checking machinery will occasionally update these files to be automatically normalized into a standard form, with each string appearing at most once.

URL-Based Methods for NSBundle

To reduce impedance mismatch with other methods taking or returning URLs, NSBundle now has URL-based equivalents of its original path-based methods. The arguments generally parallel those of the existing path-based methods, but the names in some cases have been changed to reflect current standards or to avoid terminology that has been found to be confusing. The new methods are:

```
+ (NSBundle *)bundleWithURL:(NSURL *)url;
- (id)initWithURL:(NSURL *)url;

- (NSURL *)bundleURL;
- (NSURL *)resourceURL;
- (NSURL *)executableURL;
- (NSURL *)URLForAuxiliaryExecutable:(NSString *)executableName;

- (NSURL *)privateFrameworksURL;
- (NSURL *)sharedFrameworksURL;
- (NSURL *)sharedSupportURL;
- (NSURL *)builtInPlugInsURL;

+ (NSURL *)URLForResource:(NSString *)name withExtension:(NSString *)ext subdirectory:(NSString *)subpath inBundleWithURL:(NSU
+ (NSArray *)URLsForResourcesWithExtension:(NSString *)ext subdirectory:(NSString *)subpath inBundleWithURL:(NSURL *)bundleURL
```

```
- (NSURL *)URLForResource:(NSString *)name withExtension:(NSString *)ext;
- (NSURL *)URLForResource:(NSString *)name withExtension:(NSString *)ext subdirectory:(NSString *)subpath;
- (NSURL *)URLForResource:(NSString *)name withExtension:(NSString *)ext subdirectory:(NSString *)subpath localization:(NSStri

- (NSArray *)URLsForResourcesWithExtension:(NSString *)ext subdirectory:(NSString *)subpath;
- (NSArray *)URLsForResourcesWithExtension:(NSString *)ext subdirectory:(NSString *)subpath localization:(NSString *)localizat
```

NSXMLNode and -setObjectValue:

NSXMLNode defines the -setObjectValue: method which would automatically transform non-NSString arguments into NSStrings to use as the value of the NSXMLNode object.

A long-standing bug in the number transformation code has been fixed in Mac OS X 10.6 which will affect output of XML files. Prior to SnowLeopard, NSXMLNode would improperly and inconsistently format NSNumbers passed in to -setObjectValue:. For applications linked on or after Mac OS X 10.6, NSXMLNode will use correct scientific notation for all NSNumbers passed in to -setObjectValue:. Applications linked against SDKs prior to Mac OS X 10.6 will get the original (possibly incorrect) behavior.

As a rule, if you require a particular format for any value in your XML document, you should format the data yourself as a string and then use -[NSXMLNode setStringValue:]. This guarantees that the text generated is in a format you control directly.

+[NSXMLNode namespaceWithName:stringValue:] (New since November seed)

For applications linked on SnowLeopard or later, +[NSXMLNode namespaceWithName:stringValue:] will now throw if the 'name' parameter is nil.

NSURL

There are significant API additions to NSURL to enable more efficient file property manipulation as well as additional behaviors. More description for this is forthcoming.

We have added some NSURL-based parallel APIs to places where we had only NSString-based APIs for referencing files. We intend to continue with this to make sure that all file referencing can take place via NSURLs, without the need to convert to other types such as strings or FSRefs.

NSFileManager URL-based file operations

In Mac OS X 10.6 "SnowLeopard", NSFileManager offers implementations of the common file operations that are based on NSURLs rather than paths represented by NSStrings. This eliminates some API mismatch when working primarily with AppKit APIs which primarily use NSURLs.

These methods are now available:

```
- (BOOL)copyItemAtURL:(NSURL *)srcURL toURL:(NSURL *)dstURL error:(NSError **)error;
- (BOOL)moveItemAtURL:(NSURL *)srcURL toURL:(NSURL *)dstURL error:(NSError **)error;
- (BOOL)linkItemAtURL:(NSURL *)srcURL toURL:(NSURL *)dstURL error:(NSError **)error;
- (BOOL)removeItemAtURL:(NSURL *)URL error:(NSError **)error;
```

The corresponding delegate methods are also available - see the NSFileManager.h header for more details.

NSFileManager mounted volume discovery (New since November seed)

It is now possible to get a list of the currently mounted volumes and at the same time get properties of those volumes using the following instance method on NSFileManager:

```
- (NSArray *)mountedVolumeURLsIncludingResourceValuesForKeys:(NSArray *)propertyKeys
                                                     options:(NSVolumeEnumerationOptions)options;
```

The property keys are those listed in the NSURL.h header as being volume property keys.

The return value is an NSArray of NSURL instances whose resource caches have been pre-populated with the requested resource values.

The options are:

```
enum {
    NSVolumeEnumerationSkipHiddenVolumes = 1L << 1,
    NSVolumeEnumerationProduceFileReferenceURLs = 1L << 2
}
typedef NSUInteger NSVolumeEnumerationOptions;
```

This call may block if I/O is required to determine values for the requested keys.

NSFileManager directory enumeration changes (New since November seed)

NSFileManager now offers URL-based directory enumeration similar to the mounted volume discovery above.

An NSDirectoryEnumerationOptions typedef describes the options:

```
enum {
    NSDirectoryEnumerationSkipsSubdirectoryDescendants = 1L << 0,
    NSDirectoryEnumerationSkipsPackageDescendants      = 1L << 1,
    NSDirectoryEnumerationSkipsHiddenFiles             = 1L << 2
};
typedef NSUInteger NSDirectoryEnumerationOptions;
```

For shallow directory enumerations, the following method is now available:

```
- (NSArray *)contentsOfDirectoryAtURL:(NSURL *)url
              includingPropertiesForKeys:(NSArray *)keys
                                 options:(NSDirectoryEnumerationOptions)mask
                                   error:(NSError **)error;
```

which returns an NSArray of NSURL instances representing the contents of the directory rooted at url. You can also provide an NSArray of keys of attributes to retrieve during the enumeration (these are the keys specified in <Foundation/NSURL.h>). Because this method provides a shallow enumeration, the only flag which makes sense to pass as an option is NSDirectoryEnumerationSkipsHiddenFiles. If an error occurs, the return value will be a nil array and the error parameter will be set to an appropriate NSError in the Cocoa error domain.

For deep directory enumerations, there is a new method on NSFileManager to return an NSDirectoryEnumerator which vends NSURLs:

```
- (NSDirectoryEnumerator *)enumeratorAtURL:(NSURL *)url
            includingPropertiesForKeys:(NSArray *)keys
                               options:(NSDirectoryEnumerationOptions)mask
                          errorHandler:(BOOL (^)(NSURL *url, NSError *error))handler;
```

The url and keys parameters are as above. The mask parameter can take any of the flags defined in the NSDirectoryEnumerationOptions typedef. If an error occurs during enumeration, the handler block is invoked, which will be passed the url on which the error occurred and the error. If no handler is provided, the error will be skipped and enumeration will continue.

For both of these methods, if you wish to only receive the URLs and no other attributes, then pass '0' for 'options' and an empty NSArray ('[NSArray array]') for 'keys'. If you wish to have the property caches of the vended URLs pre-populated with a default set of attributes, then pass '0' for 'options' and 'nil' for 'keys'.

NSFileManager file system item replacement (New since November seed)

In Mac OS X 10.6 "SnowLeopard", NSFileManager provides an NSURL-based mechanism for replacing one filesystem object with another:

```
- (BOOL)replaceItemAtURL:(NSURL *)originalItemURL
          withItemAtURL:(NSURL *)newItemURL
         backupItemName:(NSString *)backupItemName
                options:(NSFileManagerItemReplacementOptions)options
       resultingItemURL:(NSURL **)resultingURL error:(NSError **)error;
```

This method returns YES if the replacement operation was successful. resultingURL will be populated with the URL which points at the new item. resultingURL may be the same as originalItemURL if the replacement could be made without having to create a new filesystem object. resultingURL may be different than originalItemURL if the replacement could not be made without having to create a new object (e.g. going from an rtf document to an rtfd requires the creation of a new item - in this case, resultingURL would locate the newly-created rtfd).

By default, the creation date, permissions, Finder label and color, and Spotlight comments of the original item will be preserved on the resulting item.

If backupItemName is provided, that name will be used to create a backup of the original item. The backup will be placed in the same directory as the original item. Should an error occur during the creation of the backup item, the operation will fail. If there is already an item with that name, the item will be removed.

NSFileManagerItemReplacementOptions is defined as follows:

```
enum {
    NSFileManagerItemReplacementUsingNewMetadataOnly = 1L << 0,
    NSFileManagerItemReplacementWithoutDeletingBackupItem = 1L << 1
}
typedef NSUInteger NSFileManagerItemReplacementOptions;
```

Pass 0 to get the default behavior which uses only the metadata from the new item, adjusting some properties from the original item (as above). In most cases, 0 should be passed.

NSFileManagerItemReplacementUsingNewMetadataOnly means metadata on the resulting item will be taken entirely from the new item.

NSFileManagerItemReplacementWithoutDeletingBackupItem flag causes -replaceItemAtURL:withItemAtURL:backupItemName:options:resultingItemURL:error: to leave the backup item in place if the operation is successful.

A new domain selector, NSItemReplacementDirectory, has been added for use with -[NSFileManager URLForDirectory:inDomain:appropriateForURL:create:error:]. It returns an NSURL locating the most appropriate directory for use with this method. You should write your new item into this directory.

If an error occurs in replacing a filesystem item and the original item has been left in neither the original location nor the temporary location, the NSError returned will contain a user info dictionary with the NSFileOriginalItemLocationKey key and its value will be an NSURL instance which locates the item. The error code is one of the various NSFile* errors already present in <Foundation/FoundationErrors.h>.

NSTask (New since WWDC 2008)

In SnowLeopard, the following method has been added to NSTask:

```
- (NSTaskTerminationReason)terminationReason;
```

which returns one of the following:

```
enum {
    NSTaskTerminationReasonExit = 1,
    NSTaskTerminationReasonUncaughtSignal = 2
};
typedef NSInteger NSTaskTerminationReason;
```

This allows a client to distinguish between the child process exiting cleanly and the child process ending because it received a signal it did not or could not handle.

If -[NSTask terminationReason] is called prior to task termination, an exception is thrown.

NSURL keyed archiving (New since February 2009 seed)

Mac OS X 10.6 "SnowLeopard" introduces file reference URLs; URLs which track by a file system item's identity rather than its path. Because file reference URLs do not function after certain file system or other events (e.g. login/logout or remount of a file system), file reference URLs must be used sparingly and encoded carefully.

When asked to perform keyed archiving, file reference URLs will encode their minimal bookmark data and when decoded will resolve to a file path URL (if you wish to receive a file reference URL from this, please call -[NSURL fileReferenceURL] on the resulting item).

NSURL Path Utilities (New since November seed)

NSURL has new methods for common path manipulations, defined in the NSURLPathUtilities category on NSURL:

```
+ (NSURL *)fileURLWithPathComponents:(NSArray *)components;
- (NSArray *)pathComponents;
- (NSString *)lastPathComponent;
- (NSString *)pathExtension;
- (NSURL *)URLByAppendingPathComponent:(NSString *)pathComponent isDirectory:(BOOL)isDirectory;
- (NSURL *)URLByAppendingPathComponent:(NSString *)pathComponent;
- (NSURL *)URLByDeletingLastPathComponent;
- (NSURL *)URLByAppendingPathExtension:(NSString *)pathExtension;
- (NSURL *)URLByDeletingPathExtension;
```

With path-based file: scheme URLs, the above methods operate similarly to the NSPathUtilities category methods on NSString. When file reference URLs (e.g. file:///.file/id=103.3747951) are sent these messages, a file reference URL will be returned which has been appropriately modified. fileURLWithPathComponents: will always return a path-based file: scheme URL.

For example, when sending the URLByAppendingPathComponent:isDirectory: message to a file reference URL, you would get the file reference URL as the base with the relative portion being the string passed as the pathComponent parameter. This creates a file reference URL which tracks the parent directory by filesystem identity, but locates a file of a specific name within that directory.

These conversions occur only when necessary. If the operation can be safely performed by manipulating the string of the relative portion of the NSURL instance, then these methods will do so. These conversions guarantee that the NSURL instance that is returned has the same base type as the original NSURL. If the base of the original NSURL instance was a file reference URL, then the base of the new NSURL instance will be a file reference URL.

If there is no last path component or path extension, the first two methods return the empty string (@"").

The following methods work only on file: scheme path-based URLs; for file reference URLs or for non-file: scheme URLs, these methods return the URL unchanged:

```
- (NSURL *)URLByStandardizingPath;
- (NSURL *)URLByResolvingSymlinksInPath;
```

lastPathComponent is not suitable for display to the user. You should use NSURL's getResourceValue:forKey:error: and pass NSURLLocalizedNameKey for the key.

pathExtension should not be used to determine the type of the file. You should instead use NSURL's getResourceValue:forKey:error: and pass NSURLTypeIdentifierKey as the key.

NSHost (New since November seed)

NSHost now provides a method to retrieve the name of the computer as specified in the "Sharing" preferences pane.

```
- (NSString *)localizedName;
```

This is the name that is used in the Finder sidebar and as the default name when publishing NSNetServices. This method only returns an NSString when sent to the +[NSHost currentHost] instance; all other instances currently return nil.

NSUserDefaults (new since February seed)

NSUserDefaults now has methods for setting and reading NSURLs as values.

```
- (void)setURL:(NSURL *)url forKey:(NSString *)key;
- (NSURL *)URLforKey:(NSString *)key;
```

When an NSURL is stored using -[NSUserDefaults setURL:forKey:], some adjustments are made:

1. Any non-file URL is written by calling +[NSKeyedArchiver archivedDataWithRootObject:] using the NSURL instance as the root object.

2. Any file reference file: scheme URL will be treated as a non-file URL, and information which makes this URL compatible with 10.5 systems will also be written as part of the archive as well as its minimal bookmark data.

3. Any path-based file: scheme URL is written by first taking the absolute URL, getting the path from that and then determining if the path can be made relative to the user's home directory. If it can, the string is abbreviated by using stringByAbbreviatingWithTildeInPath and written out. This allows pre-10.6 clients to read the default and use -[NSString stringByExpandingTildeInPath] to use this information.

When an NSURL is read using -[NSUserDefaults URLForKey:], the following logic is used:

1. If the value for the key is an NSData, the NSData is used as the argument to +[NSKeyedUnarchiver unarchiveObjectWithData:]. If the NSData can be unarchived as an NSURL, the NSURL is returned otherwise nil is returned.

2 If the value for this key was a file reference URL, the file reference URL will be created but its bookmark data will not be resolved until the NSURL instance is later used (e.g. at -[NSData initWithContentsOfURL:]).

3. If the value for the key is an NSString which begins with a ~, the NSString will be expanded using -[NSString stringByExpandingTildeInPath] and a file: scheme NSURL will be created from that.

Notes about persistence of NSURL and file reference URLs

When using NSURL instances to refer to files within a process, it's important to make the distinction between location-based tracking (file: scheme URLs that are basically paths) versus filesystem identity tracking (file: scheme URLs that are file reference URLs). When persisting an NSURL, you should take that behavior into consideration. If your application tracks the resource being located by its identity so that it can be found if the user moves the file, then you should explicitly write the NSURL's bookmark data or encode a file reference URL.

If you want to track a file by reference but you require explicit control over when resolution occurs, you should take care to write out bookmark data to NSUserDefaults rather than rely on -[NSUserDefaults setURL:forKey:]. This allows you to call +[NSURL URLByResolvingBookmarkData:options:relativeToURL:bookmarkDataIsStale:error:] at a time when you know your application will be able to handle the potential I/O or required user interface interactions.

defaults command

The defaults command-line tool is now able to print the by-host identifier used for a particular computer. This identifier is the string that is used for preferences filenames in the user's Library/Preferences/ByHost directory. You may print this identifier by using the following command-line invocation:

```
    defaults printHostIdentifier
```

NSXMLParser (new since February 2009 seed)

For applications linked on or after SnowLeopard, NSXMLParser will stop parsing when it encounters a fatal error; by not stopping applications would sometimes crash due to inconsistent parser state.

Applications linked on SDKs prior to 10.6 will continue to get the old behavior. New applications should expect that parsing will end when an error occurs.

New NSCalendarUnit

A new "quarter" calendar unit (NSQuarterCalendarUnit) has been added to the set of NSCalendarUnits (NSCalendar.h). A new pair of accessor methods (-quarter/-setQuarter:) have also been added to NSDateComponents.

New calendars (Updated since November seed)

Four new calendar constants (NSRepublicOfChinaCalendar, NSPersianCalendar, NSIndianCalendar, NSISO8601Calendar) have been added (NSLocale.h). The ISO8601 calendar is not yet implemented. A Chinese calendar can be created, and one can do calendrical calculations with it, but it should not be used for formatting as the necessary underlying functionality is not functioning correctly yet.

New NSTimeZone class method

A new method, +setAbbreviationDictionary:, has been added (NSTimeZone.h). This corresponds to the CFTimeZoneSetAbbreviationDictionary() function in the CFTimeZone API.

NSConnection method deprecation

The +defaultConnection method in NSConnection has been deprecated. This provided a singleton connection object per thread. For what should be obvious reasons, it was never a particularly good idea to use this method/connection unless you had absolute certainty that no one else was using it (or going to use it), which was problematic. Just use [[NSConnection new] autorelease] instead (in conjunction with NSThread's thread dictionary if you must have one stored per-thread, though as a design point that may cause you problems in the future, so it would be best to avoid a per-thread connection).

NSMutableArray method deprecation

The -removeObjectsFromIndices:numIndices: method has been deprecated. Use -removeObjectsAtIndexes: instead.

NSDate method deprecations and additions

The -addTimeInterval: method of NSDate has been deprecated and replaced with -dateByAddingTimeInterval: (NSDate.h). The new method is available from 10.5 onward.

The methods +dateWithTimeInterval:sinceDate: and -initWithTimeIntervalSince1970: have been added to NSDate (NSDate.h). These new methods are available from 10.4 onward.

Method declarations moved

The declarations for the keyed archiving geometry struct category methods on NSCoder – -encodePoint:forKey:, -encodeSize:forKey:, -encodeRect:forKey:, -decodePointForKey:, -decodeSizeForKey:, -decodeRectForKey: – moved from NSKeyedArchiver.h to NSGeometry.h.

The declaration for the NSDate method -descriptionWithLocale: moved from a category in NSCalendarDate.h to NSDate.h.

New convenience methods on NSDateFormatter, NSNumberFormatter

A new class method, +localizedStringFromDate:dateStyle:timeStyle:, has been added to NSDateFormatter to produce a localized formatted date string representation in fewer lines of code than the previous sequence involving creation of an NSDateFormatter.

A new class method, +localizedStringFromNumber:numberStyle:, has been added to NSNumberFormatter to produce a localized formatted number string representation in fewer lines of code than the previous sequence involving creation of an NSNumberFormatter.

Privatized instance variables

In several classes, the declarations of the instance variables have been fixed to mark them private, protected or package where previously the permissions were more permissive. This includes the classes NSIndexPath, NSMutableIndexSet, NSNotificationCenter, and NSSimpleCString.

Autorelease pool debugging/performance DTrace probes added

Several DTrace probe points have been added for autorelease pool performance analysis and debugging using DTrace scripts:

```
provider Cocoa_Autorelease {
   probe pool_push(unsigned long value);    // arg is a token representing pool
   probe pool_pop_start(unsigned long value);    // arg is a token representing pool
   probe pool_pop_end(unsigned long value);    // arg is a token representing pool
   probe autorelease(unsigned long value);    // arg is object pointer
   probe error_no_pool(unsigned long value);    // arg is object pointer
   probe error_freed_object(unsigned long value);    // arg is object pointer
};
```

These are triggered at what should be the obvious points.

NSAutoreleasePool debugging helper methods deprecated

These methods in NSDebug.h in a category on NSAutoreleasePool have been deprecated:
+enableFreedObjectCheck:
+enableRelease:
+resetTotalAutoreleasedObjects
+totalAutoreleasedObjects
+autoreleasedObjectCount
+topAutoreleasePoolCount
+poolCountHighWaterMark
+setPoolCountHighWaterMark:
+poolCountHighWaterResolution
+setPoolCountHighWaterResolution:

This function no longer exists to set a breakpoint on:
_NSAutoreleaseHighWaterLog

There is no longer any direct way to do what these used to do.

These two breakpoint functions have been renamed:
_NSAutoreleaseNoPool to __NSAutoreleaseNoPool
_NSAutoreleaseFreedObject to __NSAutoreleaseFreedObject

System uptime

A new method, +systemUptime, has been added to NSProcessInfo (NSProcessInfo.h). This returns the amount of time the system has been awake since last restart.

New system clock change notification (Updated since November seed)

A new NSSystemClockDidChangeNotification notification is now available (NSDate.h), which is sent after the calendar clock of the machine changes (the clock used by NSDate, gettimeofday(), etc.).

NSJavaSetup.h removed

The NSJavaSetup.h header has been removed.

NSCondition availability

NSCondition is currently marked as available in Mac OS X 10.0 and later. However, there is a bug in the implementations on Mac OS X 10.0 - 10.4 that can make it not work properly in some usage patterns. So we will probably be marking it "available in 10.5 and later" at some point for the next release.

NSAssertionHandler changes

When compiling with a C99-compliant compiler, the NSAssert macro now accepts a variable number of arguments (including zero as before) for the description, and can be used instead of the argument-number-specific NSAssert1, NSAssert2, etc.

A new constant - NSAssertionHandlerKey - has been added which is the key in the thread dictionary of the per-thread assertion handler object.

NSObject changes

The -forwardingTargetForSelector: method is now properly declared on NSObject in NSObject.h.

New Block-based collection enumeration methods

New methods to enumerate collections have been added. These methods take Blocks which are invoked with each element in the collection (or subset thereof). These methods are currently not available to ObjC++, since Blocks are not yet available to C++.

```
NSArray:      -enumerateObjectsUsingBlock:, -enumerateObjectsWithOptions:usingBlock:, -enumerateObjectsAtIndexes:options:usingB
NSDictionary: -enumerateKeysAndObjectsUsingBlock:, -enumerateKeysAndObjectsWithOptions:usingBlock:
NSSet:        -enumerateObjectsUsingBlock:, -enumerateObjectsWithOptions:usingBlock:
NSIndexSet:   -enumerateIndexesUsingBlock:, -enumerateIndexesWithOptions:usingBlock:, -enumerateIndexesInRange:options:usingBl
```

The type signatures for the Block arguments are:

```
NSArray:      void (^)(id obj, NSUInteger idx, BOOL *stop)
NSDictionary: void (^)(id key, id obj, BOOL *stop)
NSSet:        void (^)(id obj, BOOL *stop)
NSIndexSet:   void (^)(NSUInteger idx, BOOL *stop)
```

The 'stop' argument is an out-only argument, though not currently marked as such, and also, really, it is a pointer to a "volatile" BOOL. If you are going to use it at all, you should only ever set this boolean to YES, and never to NO, from within the Block.

See below for an explanation of the options flags.

New Block-based collection searching methods

New methods to test or search the elements of some collections have been added. These methods take Blocks which are invoked with each element in the collection (or subset thereof). The Block should return YES if the element matches the test being performed. These methods are currently not available to ObjC++, since Blocks are not yet available to C++.

These methods return the index of the first found matching object:

```
NSArray:      -indexOfObjectPassingTest:, -indexOfObjectWithOptions:passingTest:, -indexOfObjectAtIndexes:options:passingTest:
```

These methods return the first found matching index:

```
NSIndexSet:  -indexPassingTest:, -indexWithOptions:passingTest:, -indexInRange:options:passingTest:
```

These methods return an index set of all indexes of matching objects:

```
NSArray:      -indexesOfObjectsPassingTest:, -indexesOfObjectsWithOptions:passingTest:, -indexesOfObjectsAtIndexes:options:pass
```

The type signatures for the Block arguments are the same as for the enumeration methods, for each class:

```
NSArray:      void (^)(id obj, NSUInteger idx, BOOL *stop)
NSIndexSet:  void (^)(NSUInteger idx, BOOL *stop)
```

The 'stop' argument is an out-only argument, though not currently marked as such, and also, really, it is a pointer to a "volatile" BOOL. If you are going to use it at all, you should only ever set this boolean to YES, and never to NO, from within the Block.

See below for an explanation of the options flags.

New Block-based sorting methods

New methods to test the elements of some collections have been added. These methods take Blocks which are invoked with each element in the collection (or subset thereof). The Block is used to compare pairs of elements. These methods are currently not available to ObjC++, since Blocks are not yet available to C++.

```
NSArray:        -sortedArrayUsingComparator:, -sortedArrayWithOptions:usingComparator:
NSMutableArray: -sortUsingComparator:, -sortWithOptions:usingComparator:
NSDictionary:   -keysSortedByValueUsingComparator:, -keysSortedByValueWithOptions:usingComparator:
```

The type signatures for the Block arguments are NSComparator:

```
NSComparisonResult (^)(id obj1, id obj2)
```

See below for an explanation of the options flags.

Options on the new Block-based collection enumeration, searching, and sorting methods

These options are available on the new Block-based collection enumeration and searching methods:

NSEnumerationConcurrent: invoke the Block on the selected elements concurrently; the order of invocation is nondeterministic and undefined; this flag is a hint and may be ignored by the implementation under some circumstances; the code of the Block must be safe against concurrent invocation

NSEnumerationReverse: invoke the Block on the selected elements in reverse of the natural order; available for NSArrays and NSIndexSets; undefined for NSDictionarys and NSSets, or when combined with the NSEnumerationConcurrent flag

These options are available on the new Block-based collection sorting methods:

NSSortConcurrent: invoke the Block concurrently on the pairs of elements to be compared; this flag is a hint and may be ignored by the implementation under some circumstances; the code of the Block must be safe against concurrent invocation

NSSortStable: use a stable sorting algorithm, so that equal elements remain in their original relative order; without this flag, it is undefined whether the sort algorithm will be stable or not

NSDiscardableContent protocol added

The NSDiscardableContent protocol has been added to NSObject.h. This protocol enables your objects to declare that their content may go away at any point, and allows users of your content to explicitly access the content in order to keep it around.

NSDiscardableContent-related NSObject method added

The -autoContentAccessingProxy method has been added in a category on NSObject. This method creates and returns an autoreleased proxy for the receiving object, if the receiver adopts the NSDiscardableContent protocol and still has undiscarded content. The proxy calls -beginContentAccess on the receiver to keep the content available as long as the proxy lives, and calls -endContentAccess when the proxy is deallocated (or finalized). The wrapper object is otherwise a subclass of NSProxy and forwards messages to the original receiver object as an NSProxy does. This method can be used to hide an NSDiscardableContent object's content volatility by creating an object that responds to the same messages but holds the contents of the original receiver available as long as the created proxy lives. Thus hidden, the NSDiscardableContent object (by way of the proxy) can be given out to unsuspecting recipients of the object who would otherwise not know they might have to call -beginContentAccess and -endContentAccess around particular usages (specific to each NSDiscardableContent object) of the NSDiscardableContent object.

NSCache class added

The NSCache class (NSCache.h) has been added to Foundation. This object acts somewhat like a mutable dictionary, except that objects can disappear out of it when there is memory pressure or in response to the sizing property hints. Objects put in a cache are retained by the cache (while they are in the cache).

Do not use NSCopyObject() (New since WWDC 2008)

This function is dangerous and very difficult to use correctly. It's use as part of -copyWithZone: by any class that can be subclassed, is highly error prone. This function is known to fail for objects with embedded retain count ivars, singletons, and C++ ivars, and other circumstances. Additionally, under GC or under ObjC 2.0, the zone is completely ignored.

Here is an implementation close to what is present in Mac OS X to illustrate:

```
id NSCopyObject(id object, NSUInteger extraBytes, NSZone *zone) {
    if (object == nil) return nil;
    id result = nil;
#if !__OBJC2__
    if (!(objc_collecting_enabled() && auto_zone_size((malloc_zone_t *)NSDefaultMallocZone(), object))) {
        if (!zone) zone = NSDefaultMallocZone();
        result = object_copyFromZone(object, extraBytes, (void *)zone);
    } else
#endif
    {   // ignore zone completely
        result = class_createInstance([object class], extraBytes);
        NSUInteger size = class_getInstanceSize([object class]) + extraBytes;
        objc_memmove_collectable(result, object, size);
    }
    return result;
}
```

Obviously, the objc_memmove_collectable() (essentially, memmove()) is not the right thing for C++ ivars, and object_copyFromZone() is known not to work for C++ ivars as well. More cautions on using NSCopyObject() can be found here: http://developer.apple.com/documentation/LegacyTechnologies/WebObjects/WebObjects_3.5/Reference/Frameworks/ObjC/Foundation/Protocols/NSCopying/Descrip

NSCopyObject() is likely to be deprecated after Mac OS X 10.6.

NSOperation, NSOperationQueue changes (New since WWDC 2008)

There are several new methods on NSOperation:

```
- (void (^)(void))completionBlock;
- (void)setCompletionBlock:(void (^)(void))block;
```

You can set a Block to be invoked when the NSOperation is finished. The execution context of this Block is undefined, and the Block should shunt things appropriately if this or that thing it does needs to be done in a particular context.

```
- (void)waitUntilFinished;
```

Blocks execution of the current thread until after the receiving operation has finished. Use this method with care, as it can be an easy way to deadlock your applications.

```
- (double)threadPriority;
- (void)setThreadPriority:(double)p;
```

Specifies what the priority of the thread should be while executing -main. Only applies to non-concurrent (isConcurrent returns NO) operations. The priority is changed on a best-effort basis and can't be guaranteed.

There are several new methods on NSOperationQueue:

```
- (void)addOperations:(NSArray *)ops waitUntilFinished:(BOOL)wait;
```

Bulk adder, with option of waiting each all of those operations has finished. The thread is blocked during the wait. Adding operations with -addOperation: is also much faster than in 10.5.

```
- (NSUInteger)operationCount;
```

A faster way to get the current number of operations in the queue, without getting the entire array of operations. Of course, this information may be out-of-date by the time you get and use the return value (just as the operations array can be out-of-date), so it should only be used for approximate guidance.

```
- (void)setName:(NSString *)n;
- (NSString *)name;
```

NSOperationQueues can now be named, and some tools may be able to see this name.

```
+ (id)currentQueue;
```

The currentQueue is the one running the current operation's code -- if the code being executed is being done so in the context of an NSOperation, of course. The return value is often nil.

```
+ (id)mainQueue;
```

This method returns a queue that represents the main dispatch queue. This is a serial (one thing at a time) queue and only serviced on the main thread. It is never serviced re-entrantly.

New class NSBlockOperation (New since WWDC 2008)

This is a subclass of NSOperation to which Blocks can be added, and the object will execute the Blocks as its work. If multiple Blocks are added (-addExecutionBlock:), they are executed concurrently. The operation will become Finished when all the Blocks are done executing. This may be a convenient way to get concurrent fan-out in some cases without having to create multiple NSOperations.

NSOperationQueue and Concurrent Operations and the "Current Thread" in 10.5 (New since WWDC 2009)

In attempting to explain concurrent NSOperations in the 10.5 documentation, the documentation contained this comment: "For a concurrent operation, the queue simply calls the object's start method on the current thread." Some people have taken this to be a defined behavior of the NSOperationQueue API, but as the documentation did not say or make any committments about what the "current thread" was, this was not information that could be usefully acted or relied upon.

What the documentation was describing was a 4-line sequence like this in the 10.5.x NSOperationQueue implementation:

```
if ([op isConcurrent]) {
   [op start];
} else {
   [NSThread detachNewThreadSelector:@selector(start) toTarget:op withObject:nil];
}
```

and from that point of view the documentation was literally correct. However, one step farther out the question becomes: "on what thread is that code invoked?" Or similarly, "when is that code invoked?" Without that information, the fact that -start was "called on the current thread" cannot obviously and seriously be relied upon to achieve any particular effect.

In 10.5 what actually happened was that that chunk of code was run whenever an NSOperation finished, on whatever thread posted the isFinished KVO notification for that operation, which the operation queue would listen for and react to by starting the next operation. The code was also invoked at other times when it was determined that "capacity exists" and "there is some work to do" (such as after an addOperation:) and an operation to run had been chosen from the queue. [Note that although an addOperation: on some thread might cause another operation to be run, since NSOperationQueues are not FIFO queues it would not necessarily be that specific new operation which would be chosen.] So, the "current thread" could be potentially any thread, including ones which would shortly thereafter terminate.

These comments and some example code have been removed from the 10.6 documentation.

Hashing-based collections changes (New since WWDC 2008)

The CoreFoundation and Foundation framework-provided implementations of hashing-based collections such as dictionaries have changed how they store elements, so elements may be retrieved in a different order than in previous releases. The order of elements in hashing-based collections is undefined and can change at any time, so developers must never rely on the order that elements are enumerated, or returned from a function like CFDictionaryGetKeysAndValues(). This is true even for cases where a developer may be trying to manipulate the hash codes of the objects in order to achieve some particular ordering.

The hashing algorithm is also different, so very carefully constructed -hash methods to produce perfect hash functions for a given set of objects may cause more collisions than in Leopard. On the other hand, the algorithm is more robust against middling or mediocre hash functions, though also somewhat slower in terms of cpu usage.

The maximum load factor of hashing collections now varies with the size of the collection, but for medium and large collections (say, over 100 elements, though not necessarily that specific number) is around 62% versus the 75% of Leopard. Other changes have made hashing collections grow more slowly than previously, as a memory-saving measure, which may produce more growth-induced rehashes than in Leopard (mainly impacts cpu time).

NSNotificationCenter new API (New since November seed)

This method has been added to NSNotificationCenter:

```
- (id)addObserverForName:(NSString *)name object:(id)obj queue:(NSOperationQueue *)queue usingBlock:(void (^)(NSNotification *
```

You can use it to specify a block to handle notifications, instead of the traditional target object and selector. An object to act as the observer – of unspecified type, though it will respond to methods in the NSObject protocol – is created for you and returned. You use -removeObserver: with this returned object as the parameter, to cancel a registration. The system maintains a retain on this object (until it is removed). Under GC, as with observers registered with the old addObserver:... method, the system does not keep a reference to those observers, and registrations are automatically cleaned up when an observer is collected. So under GC, you need to hang-on to the returned observer object somewhere (which is probably the case if you intend to explicitly call removeObserver: on it), as long as you want the notification registration to remain.

This new API also accomplishes 3 other things:
- the unregistration is exact; this registration does not suffer from the ambiguity of -removeObserver:name:object: as to which registration will be removed
- the execution context of the notification handler can be specified, at least to the degree of an NSOperationQueue; if the queue parameter is non-nil, the notification is handled in the context of that operation queue. If the queue parameter is nil, the block is invoked as traditionally, in the context of the posting thread.
- the execution of observer handlers added this way is performed concurrently with other observers' notification handlers. Execution order of notification handlers has always been unpredictable, and observers should not depend on another observer handling a notification before or after it. The concurrency may make such issues more apparent. Note that although observers are concurrently invoked, posting still waits until all observers have finished executing their handlers.

Using NSNotificationQueue Warning (New since WWDC 2009)

NSNotificationQueue is an API and mechanism related to run loops (NSRunLoop), and the running of run loops. In particular, posting of notifications via the notification queue is driven by the running of its associated run loop. However, there is no definition for what run loop a notification queue uses, and there is no way for a client to configure that. In fact, any given notification queue might be "tickled" into posting by nearly any run loop and different ones at different times (and given that enqueued notifications are also mode-specific, what mode(s) the run loop(s) are running in any any given time also come into play). Further, although each notification queue is "thread-specific" (see the +defaultQueue documentation), there has never been any relationship between that thread and the run loop used by a notification queue. This is all more and more problematic as more and more things move to happening on different and new threads.

The practical upshot is:
- Notifications posted through a queue might not be posted in any particular "timely" fashion;
- Notifications posted through a queue might not ever be posted (the run loop of the thread that the queue chose to ask to poke it might not be run again; for example, the thread of that run loop might exit and the notification queue discarded);
- Notifications can be posted by any given queue on a different thread than the thread the queue is the default queue for (when a queue is a default queue for some thread);
- Notifications can be posted by any given queue on different threads over time;
- There is no necessary/guaranteed relationship between the thread(s) on which notifications are enqueued and those on which the notifications eventually get posted (delivered) for any notification queue

This is true for all releases of Mac OS X. Foundation and AppKit do not use NSNotificationQueue themselves, partly for these reasons.

Concurrency, GCD, and Run Loop Cautions (New since November seed)

Many Foundation and AppKit APIs are still run loop-based. Two examples are NSTask and NSFileHandle. These APIs have no queue-targetting or completion-block-taking APIs yet. To get the completion notifications (e.g., about task termination or a read in background finishing), the run loop of the thread which started the asynchronous operation (e.g., launched the task or started the background read) must still be run, as historically, at least until that notification is delivered.

If you are switching code to NSOperation or GCD, or in general when code is moved from one execution context where it is nice and comfortable to another execution context, there can be indirect effects. Of course, if you are introducing concurrency or threads, the code may need to be audited for concurrency safety and adjusted. If the code was depending on per-thread data existing (perhaps something else created it), that per-thread data may not exist. Or, the code may have been putting sources or timers in the current thread's run loop, and relying on something else to run the run loop, which may not happen in the new execution environment. Relatedly, but conversely, if some code was putting sources in the current run loop, and that code moves elsewhere, the current run loop of the original thread may no longer have anything in it, and so other code which attempts to run that run loop may just end up spinning or doing nothing. And certainly if the running of that run loop was supposed to service something that would cause the running of that run loop to stop, those conditions may get changed any more which may mean a run loop running loop may not terminate, and spin forever.

NSIndexPath changes (New since WWDC 2008)

On Pre-Leopard systems, NSIndexPath's -indexPathByRemovingLastIndex would return nil if the receiver had a length of 1. On Leopard and newer systems, this method will return an empty indexPath instead and will never return nil.

NSAttributedString

NSAttributedString now has two methods to allow enumerations using blocks.

```
- (void)enumerateAttributesInRange:(NSRange)enumerationRange
            options:(NSAttributedStringEnumerationOptions)opts
            usingBlock:(void (^)(NSDictionary *attrs, NSRange range, inout BOOL *stop))block;
```

This method will execute the provided block with every attribute run in enumerationRange, passing it the attributes dictionary and the range over which is applies.

Ranges are by default the longest effective range, clipped to enumerationRange. If NSAttributedStringEnumerationLongestEffectiveRangeNotRequired option is supplied, then the longest effective range computation is not performed; the blocks may be invoked with consecutive attribute runs that have the same value.

The block can stop the enumeration by setting *stop = YES; it shouldn't touch *stop otherwise.

If this method is sent to an instance of NSMutableAttributedString, mutation (deletion, addition, or change) is allowed, as long as it is within the range provided to the block; after a mutation, the enumeration continues with the range immediately following the processed range, after the length of the processed range is adjusted for the mutation. (The enumerator basically assumes any change in length occurs in the specified range.) For example, if the block is called with a range starting at location N, and the block deletes all the characters in the supplied range, the next call will also pass N as the index of the range.

```
- (void)enumerateAttribute:(NSString *)attrName
                   inRange:(NSRange)enumerationRange
                   options:(NSAttributedStringEnumerationOptions)opts
                   usingBlock:(void (^)(id value, NSRange range, inout BOOL *stop))block;
```

This method is similar to the above, but enumeration takes place on a single attribute.

NSString (Updated since November seed)

NSString now has API for enumerating a variety of substring types using blocks:

```
- (void)enumerateSubstringsInRange:(NSRange)range
        options:(NSStringEnumerationOptions)options
        usingBlock:(void (^)(NSString *substring, NSRange substringRange, NSRange enclosingRange, BOOL *stop))block;
```

This method enumerates the substrings of the specified type (lines, words, sentences, etc) in the specified range of the receiver.

The options argument specifies the substring type to enumerate, as well as the following:
NSStringEnumerationSubstringNotRequired can be used as a way to indicate that the block does not need substring, in which case nil will be passed. This is simply a performance shortcut.
NSStringEnumerationReverse causes enumeration to occur from the end of the specified range to the start.
NSStringEnumerationLocalized causes the enumeration to occur using user's default locale. This will never make a difference in line, paragraph, or composed character sequence enumeration, but it may for words or sentences.

In the block that is executed, substring is the enumerated string, substringRange is the range of the enumerated string in the receiver, and enclosingRange is the range that includes the substring as well as any separator/filler characters that follow.

For instance, for lines, enclosingRange will contain the line terminators. The enclosingRange for the first string enumerated will also contain any characters that occur before the string. Consecutive enclosing ranges are guaranteed not to overlap, and every single character in the enumerated range will be included in one and only one enclosing range.

The block can stop the enumeration by setting *stop = YES; it shouldn't touch *stop otherwise.

If this method is sent to an instance of NSMutableString, mutation (deletion, addition, or change) is allowed, as long as it is within enclosingRange. After a mutation, the enumeration continues with the range immediately following the processed range, after the length of the processed range is adjusted for the mutation. (The enumerator basically assumes any change in length occurs in the specified range.) For example, if the block is called with a range starting at location N, and the block deletes all the characters in the supplied range, the next call will also pass N as the index of the range. Note that this is the case even if mutation of the previous range changes the string in such a way that the following substring would have extended to include the already enumerated range. (For example, if the string "Hello World" is enumerated via words, and the block changes "Hello " to "Hello", thus forming "HelloWorld", the next enumeration will return "World" rather than "HelloWorld".

This next method is a convenience method to enumerate all the lines in a string. The passed in line contains just the contents of the line, without the line terminators:

```
- (void)enumerateLinesUsingBlock:(void (^)(NSString *line, BOOL *stop))block;
```

As of WWDC 2009, word and sentence enumeration are implemented, but word enumeration is not yet correct in Japanese, Chinese, and Thai.

NSString (New Since WWDC2008)

NSString has a bunch of convenience methods for comparison, but none correspond directly to the way file names are sorted in the system by Finder or the open panel. This comparison is possible in Leopard with:

```
return [str compare:otherStr
            options:NSCaseInsensitiveSearch|NSNumericSearch|NSWidthInsensitiveSearch|NSForcedOrderingSearch
            range:NSMakeRange(0, [str length])
            locale:[NSLocale currentLocale]];
```

but SnowLeopard adds a new API for this:

```
- (NSComparisonResult)localizedStandardCompare:(NSString *)string;
```

This method should be used whenever file names or other strings are presented in lists and tables where Finder-like sorting is appropriate. The exact behavior and implementation of this method may be tweaked in future releases, and will be different under different localizations, so clients should not depend on the exact sorting order of the strings, and should not assume the implementation will remain as shown above.

The methods lengthOfBytesUsingEncoding: and maximumLengthOfBytesUsingEncoding: now return 0 if the amount of memory required for storing the results of the encoding conversion would exceed NSIntegerMax. Note that the former would already return 0 for any conversion errors.

The methods substringWithRange:, getLineStart:end:contentsEnd:forRange:, rangeOfString:, rangeOfCharacterFromSet:, and variants (more specialized forms) will now detect all invalid ranges (including those with negative lengths). For apps linked against SnowLeopard, this error will cause an exception; for apps linked against earlier releases, this error causes a warning, which is displayed just once per app execution.

The methods -UTF8String and cStringUsingEncoding: are now declared as __strong, which means they return pointers that are safe to keep around when running with garbage collection.

Foundation APIs which take string format arguments (APIs such as initWithFormat:, NSLog(), etc) are now decorated with attributes which allow the compiler to generate warnings on potentially unsafe usages. This currently includes usage such as:

```
NSString *str = ...;
NSString *formattedStr = [[NSString alloc] initWithFormat:str];
```

where the format is not a constant and there are no arguments. In a case like above, the call to initWithFormat: is unnecessary, so to avoid warnings, just drop the call. In a case like NSLog(str), you can instead use NSLog(@"%@", str).

NSData (New Since WWDC2008)

The following enum values have been renamed as shown:

```
    NSMappedRead   -> NSDataReadingMapped
    NSUncachedRead -> NSDataReadingUncached
    NSAtomicWrite  -> NSDataWritingAtomic
```

The old names are still available but deprecated and will be removed in a future release.

NSError

NSError has a new method and key to enable displaying a help button to accompany the error when it's displayed to the user:

```
NSString *const NSHelpAnchorErrorKey;

- (NSString *)helpAnchor;
```

If -[NSError helpAnchor] returns a non-nil value for an error being used with +[NSAlert alertWithError:], the alert panel will include a help anchor button with that value. The various presentError: variants in the kit go through the NSAlert method and will thus automatically generate a help button.

The easy way to get a value set for -helpAnchor is to specify it as the value of NSHelpAnchorErrorKey in the NSError's userInfo dictionary; or the method can be overridden.

Although this functionality is publicized in 10.6, it is available back to 10.4.

We have also added a new error code, NSFileWriteVolumeReadOnlyError, to represent the error attempting to write to a read only volume.

Error messages for validation errors in NSNumberFormatter have been improved; instead of "Formatting error" the messages will now try to indicate the invalid value and reason for validation failure. Note that NSNumberFormatter uses NSFormattingError for indicating validation errors; this value is a bit unfortunately named for now, and this could be addressed in the future.

NSComparator warnings note (New since November seed)

In trying to use NSComparator blocks like so:

```
[myArray sortUsingComparator:^(id obj1, id obj2) {
    return ([obj1 doubleValue] < [obj2 doubleValue]) ? NSOrderedAscending : NSOrderedDescending;
 }];
```

You may get an error from the compiler:

```
error: incompatible block pointer types initializing ‘int (^)(struct objc_object *, struct objc_object *)’, expected ‘NSCompa
```

This is because the block thinks its return value is int. One solution is to cast the return values:

```
[myArray sortUsingComparator:^(id obj1, id obj2) {
    return ([obj1 doubleValue] < [obj2 doubleValue]) ? (NSComparisonResult)NSOrderedAscending : (NSComparisonResult)NSOrdere
 }];
```

A better solution is to declare the block as returning NSComparisonResult:

```
[myArray sortUsingComparator:^NSComparisonResult(id obj1, id obj2) {
    return ([obj1 doubleValue] < [obj2 doubleValue]) ? NSOrderedAscending : NSOrderedDescending;
 }];
```

Collection comparisons with isEqual: vs isEqualTo<collection>:

In Leopard and earlier releases, isEqual: and isEqualToArray: (or Dictionary/Set) went through different code paths, which could sometimes give different results. In SnowLeopard, in apps linked against SnowLeopard, isEqual: now goes through the corresponding isEqualTo... methods for the three collection types. This should cause these methods to give consistent results. One noteworthy change here is that the two numbers [NSNumber numberWithInteger:1] and [NSNumber numberWithBool:YES] (or numbers created with 0 and NO) will now compare equal when in collections (just like they do when compared with -[NSNumber isEqual:]).

Foundation API not available in iPhone OS 2.0

The Mac OS X APIs in these headers are not available in iPhone OS 2.0:
NSAffineTransform.h, NSAppleEventDescriptor.h, NSAppleEventManager.h, NSAppleScript.h, NSArchiver.h, NSAttributedString.h, NSCache.h, NSCalendarDate.h, NSClassDescription.h, NSComparisonPredicate.h, NSCompoundPredicate.h, NSConnection.h, NSDistantObject.h, NSDistributedLock.h, NSDistributedNotificationCenter.h, NSExpression.h, NSGarbageCollector.h, NSGeometry.h, NSHFSFileTypes.h, NSHashTable.h, NSHost.h, NSMapTable.h, NSMetadata.h, NSObjectScripting.h, NSPointerArray.h, NSPointerFunctions.h, NSPortCoder.h, NSPortMessage.h, NSPortNameServer.h, NSPredicate.h, NSProtocolChecker.h, NSScriptClassDescription.h, NSScriptCoercionHandler.h, NSScriptCommand.h, NSScriptCommandDescription.h, NSScriptExecutionContext.h, NSScriptKeyValueCoding.h, NSScriptObjectSpecifiers.h, NSScriptStandardSuiteCommands.h, NSScriptSuiteRegistry.h, NSScriptWhoseTests.h, NSSpellServer.h, NSTask.h, NSURLDownload.h, NSURLHandle.h, NSUndoManager.h, NSValueTransformer.h, NSXMLDTD.h, NSXMLDTDNode.h, NSXMLDocument.h, NSXMLElement.h, NSXMLNode.h, NSXMLNodeOptions.h

Various other APIs which are either related to (or use types defined by) the above, or which are deprecated as of 10.5, are also unavailable.

Additionally, these APIs in the available headers are not available:
- The garbage collector-related allocation APIs (NSAllocateCollectable(), NSReallocateCollectable(), and their option flags) in NSZone.h
- The NSCoder -encodePropertyList: and -decodePropertyList methods (just use -encodeObject: and -decodeObject)
- 10.0-behavior-specific methods on NSDateFormatter and NSNumberFormatter

Otherwise, iPhone OS 2.0 contains a Foundation API matching that in Mac OS X 10.5.

Foundation API not available in iPhone OS 3.0

The Mac OS X APIs in these headers are not available in iPhone OS 3.0:
NSAffineTransform.h, NSAppleEventDescriptor.h, NSAppleEventManager.h, NSAppleScript.h, NSArchiver.h, NSAttributedString.h, NSCache.h, NSCalendarDate.h, NSClassDescription.h, NSConnection.h, NSDistantObject.h, NSDistributedLock.h, NSDistributedNotificationCenter.h, NSGarbageCollector.h, NSGeometry.h, NSHFSFileTypes.h, NSHashTable.h, NSHost.h, NSMapTable.h, NSMetadata.h, NSObjectScripting.h, NSPointerArray.h, NSPointerFunctions.h, NSPortCoder.h, NSPortMessage.h, NSPortNameServer.h, NSProtocolChecker.h, NSScriptClassDescription.h, NSScriptCoercionHandler.h, NSScriptCommand.h, NSScriptCommandDescription.h, NSScriptExecutionContext.h, NSScriptKeyValueCoding.h, NSScriptObjectSpecifiers.h, NSScriptStandardSuiteCommands.h, NSScriptSuiteRegistry.h, NSScriptWhoseTests.h, NSSpellServer.h, NSTask.h, NSURLDownload.h, NSURLHandle.h, NSXMLDTD.h, NSXMLDTDNode.h, NSXMLDocument.h, NSXMLElement.h, NSXMLNode.h, NSXMLNodeOptions.h

Various other APIs which are either related to (or use types defined by) the above, or which are deprecated as of 10.5, are also unavailable.

Additionally, these APIs in the available headers are not available:
- The garbage collector-related allocation APIs (NSAllocateCollectable(), NSReallocateCollectable(), and their option flags) in NSZone.h
- The NSCoder -encodePropertyList: and -decodePropertyList methods (just use -encodeObject: and -decodeObject)
- 10.0-behavior-specific methods on NSDateFormatter and NSNumberFormatter

Otherwise, iPhone OS 3.0 contains a Foundation API matching that in Mac OS X 10.5.

Notes specific to Mac OS X 10.5

64-Bit

Leopard contains 64-bit versions of system frameworks, enabling building and running many Cocoa apps as 64-bit.

Please see the 10.5 AppKit release notes for details on how 64-bit impacts the Cocoa APIs.

Fast enumeration

Leopard introduces the NSFastEnumeration protocol which allows a fast, safe method for objects to provide object enumerations:

```
@protocol NSFastEnumeration
- (NSUInteger)countByEnumeratingWithState:(NSFastEnumerationState *)state
                                 objects:(id *)stackbuf
                                   count:(NSUInteger)len;
@end

typedef struct {
    unsigned long state;
    id *itemsPtr;
    unsigned long *mutationsPtr;
    unsigned long extra[5];
} NSFastEnumerationState;
```

This protocol is implemented by Foundation collections as well as NSEnumerator, and is used by the new language "for...in" feature for enumerating:

```
for (id myObj in myArray) { ... do something with myObj ... }
```

Mutation of the object is forbidden during iteration, and there can be several iterations executing concurrently.

The following rules must be obeyed by any implementation:

1. On first entry, the "state" field will be 0, otherwise the client is neither allowed or required to alter state. State must be set to a non-zero before returning a non-zero count (otherwise the client will go into an infinite loop expecting a zero return, and the object will always be starting a new enumeration). For complicated enumerations "extra" is supplied to hold extra state.
2. On exit, a count of 0 means that there are no objects available for enumeration. This may reflect a bad or inconsistent internal state, or more normally it may simply reflect that no more objects are available. But if the value is zero, the client can make no assumptions about the content of state.
3. A non-zero count return implies that itemsPtr has been set to point to the first of the non-zero count objects. This is encouraged to be a pointer to internal object state where feasible. If not feasible, the client has supplied a stack buffer at "objects" and the length in the "count" parameter, and the object pointers to be iterated should be copied there (no GC primitives required). In addition, mutationsPtr is set to a valid address that should track some form of change counter for this object. If the object is immutable, this pointer could point to the object itself.

This protocol is designed to work when sent to nil objects.

Warning about mutations during enumerations

As mentioned above, in the discussion for fast enumeration, it is a significant programming error to enumerate a mutable collection and mutate that same collection. This has always been the case, but in Tiger, many applications got away with this due to an implementation detail of most NSEnumerators. A link check has been provided for these applications in order to allow them to continue to function safely.
For applications linked on Leopard, when a mutation is detected during enumeration, an NSInvalidArgumentException will be thrown:

```
2006-05-23 13:46:08.945 MyTestApp[756] *** Uncaught exception: <NSInvalidArgumentException>
                                       Collection <NSCFArray: 0x303d70> was mutated while being enumerated.
```

Garbage Collection

In Leopard it is now possible to write Cocoa applications which are garbage collected. Application developers choose (in Xcode, or through compiler flags) whether they want their applications to run garbage collected or not. System frameworks are designed to work with both garbage collected and non-garbage collected apps. Note that all binaries (bundles, frameworks, plug-ins) loaded by a garbage collected app also need to be garbage collected, otherwise they are not loaded.

Please refer to developer documentation for more details on garbage collection.

Foundation includes a new class, NSGarbageCollector, which provides a default shared instance that can be messaged to control garbage collection behavior.

NSMapTable, NSHashTable

NSMapTable and NSHashTable are now available as objects. They are generally usable, but are especially useful in garbage collected applications. The existing

functional API continues to work as well.

As objects, these classes provide new APIs for accessing them as if they always and only contain objects. We also provide common and interesting options for construction - to be created with an option to copy in its arguments, to treat the objects as pointers with regard to hashing and equality, and option to hold them in a non-retained zero-ing weak manner. These, and especially the latter, are the predominant uses that have turned up as code has been converted to be GC-compatible.

Note that even as objects NSMapTable and NSHashTable are distinct from NSDictionary and NSSet, which have rigorous semantic definitions that a lot of code relies upon. NSMapTable and NSHashTable are not plist types.

NSMapTable and NSHashTable conform to NSFastEnumeration, like other collections.

Exceptions

Zero-overhead exceptions in 64-bit

In 64-bit, Objective C uses the same technology as C++ to implement exception throwing, which makes the cost of entering a @try block fall to zero, and the cost of raising an exception much higher.

New NSException API

The NSException class now offers this new API:

```
- (NSArray *)callStackReturnAddresses;
```

This method returns an array of NSNumbers with the return addresses from the thread's stack, with the first values in the array being from the lowest frames on the stack, at the time and on the thread on which the exception was first raised. Re-raising the exception does not reset this value. There is no way to set this value.

NSException macro changes

The _NSSETJMP macro is no longer defined. It was not for your use.
The NSHandler and NSHandler2 and struct _NSHandler2 types are no longer defined. They were not for your use.
The _NSAddHandler2, _NSRemoveHandler2, and _NSExceptionObjectFromHandler2 functions are no longer declared. They were not for your use.

The NS_DURING, NS_HANDLER, and NS_ENDHANDLER macros have been redefined in terms of @try/@catch Objective C syntax. The NS_VALUERETURN and NS_VOIDRETURN macros have been updated to match.

Throwing objects other than NSExceptions in Cocoa apps

Do not throw objects other than NSExceptions (or, perhaps, a subclass thereof) in Cocoa apps. Doing so is a recipe for potential disaster if the exception is seen by Cocoa code. Just don't do it. Cocoa code may also completely and silently squash any non-object type, and possibly any non-NSException * type, exceptions that get to the Cocoa code.

Setting a breakpoint to catch exception raises

To trap exceptions being thrown in the debugger, break on objc_exception_throw. This will catch all exception throwing, in 32-bit and 64-bit, and whether the exception was thrown with @throw or thrown with -[NSException raise]. Setting a breakpoint on the latter has never caught exception throwing due to @throw.

Exceptions caught from outside callbacks no longer dropped

Foundation used to catch exceptions in certain places and not re-raise them, just log a message like "Exception ignored while..." or "Exception raised during... Ignored. ...". For apps linked on 10.5 and later, Foundation no longer does this, it just lets the exception pass through those points.

Note that letting exception escape from application code back up into system framework/library code (Cocoa or non-Cocoa code) in inheritently unpredictable. C code, for example, cannot react to exceptions being raised across a C language activation frame on the stack, partly because the syntax isn't available and partly because exceptions are not part of the language so there should be no need for C code to be aware of exceptions. So, data structures, locks, and other things may be left in an unpredictable/inconsistent state, and memory or other resources leaked. Even existing Objective C code is not necessarily robust in these regards. It is perfectly possible -- just as one example which isn't a crash -- that a raised exception will cause the application UI to freeze up and be completely unresponsive after the exception, which means the user will not be able to save any documents or other unsaved changes.

NSOperation, NSOperationQueue

NSOperation is an object which performs some encapsulated task. NSOperationQueue holds a priority queue of NSOperation instances to execute, and provides a flow regulation mechanism for operation execution.

NSOperations need not be put in NSOperationQueues to be executed - they can be executed at any time, and NSOperations which are designed as "active" objects may in fact start themselves when the appropriate conditions exist.

NSOperation has dependency management; an operation is not ready to be executed until all of the other operations in its dependency list are finished.

NSOperation provides methods to enable declaring whether it can be executed asynchronously; NSOperationQueue in turn provides methods to specify how many concurrent operations to allow. These features make NSOperation/NSOperationQueue a good fit for specifying encapsulated tasks in a way that will take advantage of multi-processor machines.

Please refer to the documentation for more information on NSOperation and NSOperationQueue.

NSThread

NSThread now enables setting a name. The primary purpose of this method is debugging:

```
- (void)setName:(NSString *)n;
- (NSString *)name;
```

You now can specify a stack size for a thread. This is obeyed if it's set only before the thread has been started. The underlying operating system may impose constraints on the value:

```
- (void)setStackSize:(NSUInteger)s;
- (NSUInteger)stackSize;
```

The following methods enable finding out about the main thread:

```
- (BOOL)isMainThread;
+ (BOOL)isMainThread; // reports whether current thread is main
+ (NSThread *)mainThread;
```

NSThread now has init methods to enable creation without starting:

```
- (id)init;    // designated initializer
- (id)initWithTarget:(id)target selector:(SEL)selector object:(id)argument;
```

If the target object does not implement the selector, the -initWithTarget:selector:object: method returns nil. The target object and argument are retained for the life of the NSThread.

The following methods enable more control over NSThread execution:

```
- (BOOL)isExecuting;
- (BOOL)isFinished;

- (BOOL)isCancelled;
- (void)cancel;

- (void)start;

- (void)main;    // thread body method
```

The -start method creates an underlying thread object and starts it executing; it can fail due to OS resource limits being reached or the thread being previously cancelled. -start causes -main to be called in the context of the new thread. The default behavior of -main is to do nothing if the thread was not initialized with a target/selector/argument, otherwise it invokes the method on the target. A subclass of NSThread can override -main to perform whatever work the thread is supposed to do (analogous to the thread function given to pthread_create()).

The -start method is the one that should be called by clients of NSThread to start a thread running; -main should not be called from outside the thread object.

The executing pthread has a logical retain on the thread object, once -start has been called. When the control flow of a thread terminates, the backing pthread is destroyed, but the NSThread object may continue to exist if there are additional retains.

The -main method is implemented to perform the function of the operation. The -main method must properly maintain the isExecuting and isFinished states. For example, -main must capture exceptions raised back to it and change the states if before returning or re-raising the exception (actions which imply that the operation is transitioning from executing to finished). However, it is legitimate for the -main method to never return and continue to execute indefinitely.

The "cancel" state is advisory: this state can be polled by code to find out if it should stop what it is doing, but it does not force stoppage in any way. This is not the same as pthread cancellation. 'Cancelled' is a one-way state transition, you cannot set a operation object back to "uncancelled". The code being executed by the operation can terminate the execution by causing control flow to exit the -main method (by returning from that method or raising an exception from that method). Because the cancelled state is advisory, it is orthogonal to the "executing" and "finished" states.

The following is a slightly less expensive version of the +sleepUntilDate: method, since an NSDate object need not be created. Although it has just been publicized, it has been in Foundation since 10.0 and can be used on older systems:

```
+ (void)sleepForTimeInterval:(NSTimeInterval)ti;
```

The following new method on NSObject enables executing in the context of a specific NSThread:

```
- (void)performSelector:(SEL)aSelector
               onThread:(NSThread *)thr
             withObject:(id)arg
          waitUntilDone:(BOOL)wait
                  modes:(NSArray *)array;
```

There is also a version without the modes: argument, which is equivalent to the above method called with kCFRunLoopCommonModes.

The naming matches the existing performSelectorOnMainThread:... methods, which become a special case of these methods. These new methods have the same semantics as the existing performSelectorOnMainThread:... methods, except that a particular NSThread object can be specified.

Finally the following method is a convenience to encapulate the process of creating a new thread to run the given method. No 'modes' argument is needed, as the run loop is not involved here. Semantically similar to other existing performSelector-style methods.

```
- (void)performSelectorInBackground:(SEL)aSelector withObject:(id)arg;
```

Another new method in NSThread is:

```
+ (NSArray *)callStackReturnAddresses;
```

This method returns an array of NSNumbers with the return addresses from the current thread's stack, with the first values in the array being from the lowest frames on the stack, at the time of the call to this method.

NSProcessInfo

NSProcessInfo now exposes a new method for returning the amount of physical memory installed in the computer.

```
- (unsigned long long)physicalMemory;
```

returns the amount of memory in bytes physically installed in the computer. Developers may use this method as a way of determining memory allocation patterns particularly in 64 bit applications.

The following new methods return the number of processors as well as the number of active processors in the machine:

```
- (NSUInteger)processorCount;
- (NSUInteger)activeProcessorCount;
```

Note that the number of available processors can vary over time, for example, if CPUs are turned off or on by the user.

NSCondition

NSCondition is a new class in Foundation. An NSCondition is a pthread-style condition variable, combined with the associated mutex lock. NSCondition operates just like pthread_conds with the same usage patterns and potential pitfalls.

The existing NSConditionLock class is a more limited version of this. An NSConditionLock defines only an integer state that the condition test tests against. So, NSCondition, by allowing the while predicate test to be moved outside the object, allows for greater flexibility, beyond simple integer equality.

The usage pattern is almost always this:

```
[cond lock];
while (TEST) {
    [cond wait];
}
...
[cond signal]; // or broadcast
[cond unlock];
```

NSCondition already existed inside Foundation, and has since before 10.0. So it can be used on older operating systems.

NSLock logging

NSLock and subclasses will now indicate erroneous lock and unlock attempts with logs such as:

```
2006-07-14 09:17:00.729 MyTestApp *** -[NSConditionLock unlockWithCondition:]: lock (<NSConditionLock: 0x183b3800>)
                        unlocked when not locked
2006-07-14 09:17:00.729 MyTestApp *** Break on _NSLockError() to debug.
```

These almost always indicate problems that should be fixed. In many cases the problems were there in 10.4 as well, but not being reported.

NSLock naming

NSLock, NSRecursiveLock, NSConditionLock, and NSCondition objects can now be named with two new methods in each class:

```
- (void)setName:(NSString *)n;
- (NSString *)name;
```

The name is emitted in NSLogs about the objects and in their descriptions.

Distributed Objects

Distributed Objects and 64 bit

In a distributed objects environment prior to Leopard, method signatures of the local and remote objects were assumed to be the same; if they were different an exception would be thrown. With the introduction of 64 bit Foundation, the signatures may differ. For example, a 32 bit spell server communicating with a 64 bit text editor will result in different NSRange types - the server will consider it to be a struct of two 32 bit ints, and the client will consider it to be a struct of two 64 bit longs.

To provide for DO communication between these processes, the DO signature equality requirement has been relaxed to method signature "compatibility." Two method signatures are compatible if the type of each argument (and return value) in the signature can be converted to the corresponding type in the other signature. Conversions within (but not between) the following groups are allowed:

- int, long, long long
- unsigned int, unsigned long, unsigned long long
- float, double

Aggregate types such as structs are compatible if their members are compatible, recursively. For example, float can be converted to or from double, and therefore NSPoints and NSSizes can be passed between 32 bit and 64 bit processes because they are composed of floats and doubles, and therefore NSRects can be passed as well because they are composed of NSPoints and NSSizes.

Distributed Objects overflow and underflow behavior

If a conversion would result in an out of range value, the result is set to the closest value supported by that type. For example, if the 64 bit signed integer value 5000000000 (five billion) is passed via DO to an object expecting a 32 bit signed integer value, it will receive the value INT_MAX. The possible results of integer overflow or underflow are therefore:

```
INT_MAX
INT_MIN
UINT_MAX
```

Double values smaller than the smallest floating point value are always rounded towards zero. The possible results of floating point overflow or underflow are therefore:

```
FLT_MAX
-FLT_MAX
0.f
-0.f
```

Be aware that this behavior may not preserve the semantics of certain special values. For example, -[NSArray indexOfObject:] called on a 64 bit process will return UINT_MAX to a 32 bit process instead of the 32 bit value of NSNotFound.

If overflow or underflow occurs for any type, a message is logged by the server to the console.

Distributed Objects local signature search sequence

When a method to be forwarded is called on an NSDistantObject, the object needs information about how the method was called to properly construct the NSInvocation - specifically, it needs the NSMethodSignature corresponding to the method signature that the compiler used. NSDistantObject searches for the method signature in the following places, in order:

• Within the Protocol set on the NSDistantObject via setProtocolForProxy:

• Within the local class of the same name as the remote object's class
• Within the remote object

Therefore, if the call may have been compiled against a different method signature than that of the remote object, you must ensure that the class is available locally, or better yet, that the methods you call are present in the NSDistantObject's Protocol. If the method signature is only available remotely, and it differs from what the compiler saw, parameters and return values will not be passed correctly. For example, if you compile a 32 bit client against a method signatures that uses NSInteger, but the actual server is 64 bit, and no protocol is set on the object, and the class is not present in the client, the parameters will not be passed correctly.

Distributed Objects Tiger compatibility

These conversions are not supported if either the client or server is running Tiger or earlier. To use distributed objects on a pre-Leopard version of the OS, you must ensure that the client and server method signatures match exactly in both size and type.

NSNotFound

As part of the 64-bit porting, NSNotFound is now defined as NSIntegerMax. Formerly it was 0x7fffffff, effectively the same thing on 32-bit. Note that since the value is different in 32-bit and 64-bit, it is a bad idea to [continue to] save it in files or archives, if you are today, and sending it between 32-bit and 64-bit processes via Distributed Objects will not get you NSNotFound on the other side. This applies to any Cocoa methods invoked over D.O. as well, which might return NSNotFound, such as -indexOfObject: in NSArray (specifically, when sent to a proxy to an array, of course).

Leaks in Distributed Objects plugged

Some leaks, and places where objects could leak if the timing was right, in Distributed Objects have been fixed in 10.5. This may mean that you may no longer get away with something you used to get away with due to the leak. Not retaining the root proxy you get from the connection, if you're going to hold onto it, seems to be a common pitfall.

Key Value Coding and Observing (KVC, KVO)

New KVO Options

Two new key-value observing options that can be used when invoking -[NSObject(NSKeyValueObserverRegistration) addObserver:forKeyPath:options:context:] have been added in Mac OS 10.5. The first merely allows to you specify that the observer should receive a notification about the observed property's value right away:

```
NSKeyValueObservingOptionInitial = 0x04,
```

Specifies whether a notification should be sent to the observer immediately, before the observer registration method even returns. The change dictionary in the notification will always contain an NSKeyValueChangeNewKey entry if NSKeyValueObservingOptionNew is also specified but will never contain an NSKeyValueChangeOldKey entry. (In an initial notification the current value of the observed property may be old, but it's new to the observer.) You can use this option instead of explicitly invoking, at the same time, code that is also invoked by the observer's -observeValueForKeyPath:ofObject:change:context: method. When this option is used with -addObserver:toObjectsAtIndexes:forKeyPath:options:context: a notification will be sent for each indexed object to which the observer is being added.

The second allows you to specify that the observer should receive change notifications before and after changes, instead of just after:

```
NSKeyValueObservingOptionPrior = 0x08
```

Specifies whether separate notifications should be sent to the observer before and after each change, instead of a single notification after the change. The change dictionary in a notification sent before a change always contains an NSKeyValueChangeNotificationIsPriorKey entry (also declared in <Foundation/NSKeyValueObserving.h>) whose value is [NSNumber numberWithBool:YES], but never contains an NSKeyValueChangeNewKey entry. When this option is specified the change dictionary in a notification sent after a change contains the same entries that it would contain if this option were not specified. You can use this option when the observer's own KVO-compliance requires it to invoke one of the -willChange... methods for one of its own properties, and the value of that property depends on the value of the observed object's property. (In that situation it's too late to easily invoke -willChange... properly in response to receiving an -observeValueForKeyPath:ofObject:change:context: message after the change.)

Support for Key Paths in KVO's Dependency Mechanism (Updated since WWDC 2007)

A simple property dependency mechanism was included in KVO when it was first published in Mac OS 10.3, in the form of the +[NSObject(NSKeyValueObservingCustomization) setKeys:triggerChangeNotificationsForDependentKey:] method and KVO's use of the information recorded by your application's invocation of it. This mechanism however:
• Did not support key paths.
• Was not very easy to program with, because it was difficult to determine when exactly to invoke +setKeys:triggerChangeNotificationsForDependentKey:.

In Mac OS 10.5, +setKeys:triggerChangeNotificationsForDependentKey: has been deprecated and a new method, +keyPathsForValuesAffectingValueForKey: has been published. It doesn't suffer from either of the just-described problems. See the comments for it in <Foundation/NSKeyValueObserving.h> for more information.

Note: This was not yet true in the WWDC 2007 seed, even though this enhancement was mentioned in the seed's version of this release note.

Better Support in KVO for Properties Added By Categories (Updated since WWDC 2007)

+automaticallyNotifiesObserversForKey: did not lend itself well to situations where the getter for the keyed property was added by a class category. To fix this problem, the default implementation of +automaticallyNotifiesObserversForKey: has been updated to find methods whose names follow the pattern +automaticallyNotifiesObserversOf<Key>. (This is consistent with the way the new +keyPathsForValuesAffectingValueForKey: method works.) Such methods can be easily implemented in categories right next to the corresponding getter methods. See the comments for it in <Foundation/NSKeyValueObserving.h> for more information.

Note: This was not yet true in the WWDC 2007 seed, even though this enhancement was mentioned in the seed's version of this release note.

Support in KVO for Cascading Property Dependencies (New since WWDC 2007)

In Mac OS 10.3 and Mac OS 10.4 you could use the now-deprecated +setKeys:triggerChangeNotificationsForDependentKey: method to declare that property X of a class depends on property Y, and that property Y depends on property Z, but changes to the value of Z would not result in notifications sent to observers of X. In Mac OS 10.5 changes to the value of Z now result in notifications sent to observers of both X and Y, regardless of whether the dependencies were

registered using +setKeys:triggerChangeNotificationsForDependentKey: or the newer +keyPathsForValuesAffectingValueForKey: mechanism described above.

Bug Fix in KVO's Dependency Mechanism

In Mac OS 10.3 and Mac OS 10.4 there was a bug in which KVO's dependency mechanism did not interoperate well with observing by key path. For example, given a class Foo with properties "visible" and "visibleThing," where -visibleThing would always return nil if visible was set to NO, or always return an object that itself had a "name" attribute if visible was set YES, and given that the dependency of "visibleThing" on "visible" was declared by invoking [Foo setKeys: [NSArray arrayWithObject:@"visible"] triggerChangeNotificationsForDependentKey:@"visibleThing"], observers of a Foo's "visibleThing.name" would not be notified when the value of the Foo's "visible" property changed. This bug has been fixed in Mac OS 10.5.

Better Interoperability Between KVO and Class Categories Loaded From Bundles

In Mac OS 10.3 and 10.4 there was a bug in which methods added to classes by categories loaded from bundles would not work in the variant of the class that's created by the KVO autonotification isa-swizzling machinery. So, it was possible to add a KVO observer to an object, load a bundle that was supposed to add methods to the class of observed object, and then have those methods not be responded to. This bug has been fixed in Mac OS 10.5.

Support for the Class Type in KVC and KVO

In Mac OS 10.5, KVC now works with the Class type. For example, -valueForKey: will invoke methods like -(Class)key, and -setValue:forKey: will invoke methods like -(void)setKey:(Class)aClass. Both methods work with appropriately named instance variables whose type is Class. KVO's automatic observer notification machinery also handles Class properties properly.

Support for Arbitrary Types in KVC and KVO

In Mac OS 10.5, KVC now work with arbitrary types. For example, if you have a class like this:

```
typedef struct {
    float x, y, z;
} ThreeFloats;

@interface MyClass

- (void)setThreeFloats:(ThreeFloats)threeFloats;
- (ThreeFloats)threeFloats;

@end
```

[anInstanceOfMyClass valueForKey:@"threeFloats"] will invoke -[MyClass threeFloats] and return the result wrapped in an NSValue. Likewise [anInstanceOfMyClass setValue:anNSValueWrappingThreeFloats forKey:@"threeFloats"] will invoke -[MyClass setThreeFloats:] with the result of sending -getValue: to anNSValueWrappingThreeFloats. Both methods work with appropriately named instance variables of arbitrary type. KVO's automatic observer notification machinery also handles arbitrarily-typed properties properly. This mechanism doesn't take reference counting or garbage collection into account, so take care when using with object-pointer-containing structure types.

Less Exception Throwing for Zero-Length Keys and Paths in KVC

In Mac OS 10.5, the overrides of -valueForKey: and -valueForKeyPath: in the NSDictionary, NSArray, NSSet, and NSUserDefaults classes no longer throw spurious "Range or index out of bounds" exceptions for zero-length keys and key paths.

Support for Debugging of Bad KVO Removal

In Mac OS 10.3 and 10.4, -[NSObject(NSKeyValueObserverRegistration removeObserver:forKeyPath:] would do one of two things when passed an object that was not registered as an observer of the receiver at that moment:
1) Crash
2) Nothing
In Mac OS 10.5 KVO has been update to always throw an exception for bad key-value observer removal. For backward binary compatibility no exception is thrown in case #2, in applications linked against Mac OS 10.4 or earlier.

Support for Debugging of Bad KVO Compliance

The most common mistake that people make when trying to make a class key-value observing (KVO) compliant for a particular key is neglecting to arrange for the sending of appropriate observer notifications when the value for that key changes. This is especially easy to do when you forgo automatic key-value observing in your class. The most readily visible symptom of this mistake is that views that are bound in some way to the keyed property don't update to reflect changed values as you would expect. When you see problems like this review the "Ensuring KVO Compliance" page and the related pages of the Key-Value Observing Programming Guide.

A more serious symptom of bad KVO compliance appears when the key is used in a key path. In Mac OS 10.3 and 10.4 bad KVO-compliance can cause crashes in KVO's own path-observing machinery. Such a crash usually looks something like this:

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x00000006
0x90285e24 in CFRetain ()
(gdb) bt 4
#0  0x90285e24 in CFRetain ()
#1  0x90b8165c in _NSKeyValueObservationInfoCreateByRemoving ()
#2  0x90b81434 in -[NSObject(NSKeyValueObserverRegistration) _removeObserver:forProperty:] ()
#3  0x90b81324 in -[NSObject(NSKeyValueObserverRegistration) removeObserver:forKeyPath:] ()
#4  0x90b81274 in -[NSObject(NSKeyValueObserverRegistration) removeObserver:forKeyPath:] ()
(More stack frames follow...)
(gdb)
```

In Mac OS 10.5 KVO has been updated to throw an exception when this crash would have happened. Now, for example, executing this code:

```
- (void)demonstrateExceptionsForBadKVOCompliance {

    /* Allocate an object that can be used as a key-value observer.
       In a typical Cocoa application using Bindings the observers are usually private
       objects created by the key-value bindings machinery, but any class can be used as a
       key-value observer as long as it implements the -observeValueForKeyPath:ofObject:change:context:
       method properly.
```

```
*/
KeyValueObserver *observer = [[KeyValueObserver alloc] init];

/* Allocate an instance of a class that has a to-one relationship to a mutable dictionary, keyed
   by "toOneRelationshipWithBadKVOCompliance," but doesn't properly notify key-value observers when
   the dictionary itself is replaced by another dictionary. The use of a mutable dictionary as the
   related object in this example is arbitrary; key-value coding (KVC) and key-value observing work
   with all sorts of objects as long as they are KVC and KVO compliant for the keys in use.
   (Dictionaries are automatically KVC and KVO compliant for all keys.)
*/
PlentyOProperties *plentyOProperties = [[PlentyOProperties alloc] init];

/* Start observing an entry in that mutable dictionary. This causes KVO to not only start
   observing the dictionary, but also to register itself as an observer of the plentyOProperties
   object, so it knows when the dictionary has changed and it therefore needs to stop observing
   the old dictionary and start observing the new dictionary.
*/
[plentyOProperties addObserver:observer forKeyPath:@"toOneRelationshipWithBadKVOCompliance.anyOldKey"
                       options:0 context:NULL];

/* Change the pointed-to dictionary. For this demonstration we made the PlentyOProperties
   class KVO noncompliant by overriding +automaticallyNotifiesObserversForKey: to turn off
   automatic observer notification and by neglecting to implement a -setToOneRelationshipWithBadKVOCompliance:
   method that does manual observer notification using -willChangeValueForKey: and -didChangeValueForKey:.
   Because PlentyOProperties is not KVO compliant, the KVO mechanism that's used when observing
   key paths doesn't get notified that it should stop observing the old dictionary and start observing
   the new one here.
*/
[plentyOProperties setValue:[NSMutableDictionary dictionaryWithObject:@"anyOldValue" forKey:@"anyOldKey"]
                     forKey:@"toOneRelationshipWithBadKVOCompliance"];

/* Stop observing what we started observing up above. In Panther and Tiger: Crash! Because KVO itself tries
   to stop observing the newly-related dictionary, but it never started observing it in the first place.
   In Mac OS 10.5: A helpful exception is thrown.
*/
[plentyOProperties removeObserver: observer forKeyPath:@"toOneRelationshipWithBadKVOCompliance.anyOldKey"];

/* Etc, etc. */

}
```

will cause something like this to appear in the console:

```
Cannot remove an observer <KeyValueObserver 0x40fdb0> for the key path
"toOneRelationshipWithBadKVOCompliance.anyOldKey" from <PlentyOProperties 0x410a30>,
most likely because the value for the key "toOneRelationshipWithBadKVOCompliance" has
changed without an appropriate KVO notification being sent. Check the KVO-compliance
of the PlentyOProperties class.
```

If you break on objc_exception_throw in the debugger, you'll probably see two exceptions go by. In addition to the just-described exception, you'll see something like this, first:

```
Cannot remove an observer <NSKeyValueObservationForwarder 0x3125c0> for the key path "anyOldKey"
from <NSCFDictionary 0x312250> because it is not registered as an observer.
```

The shorter exception is the result of KVO sensing the symptom of our KVO-compliance problem. The longer exception is the result of KVO catching that exception and throwing one with, hopefully, more descriptive information about the problem.

For the curious: What is NSKeyValueObservationForwarder? It's the private class of objects that KVO uses to implement observing of key paths. Your application shouldn't depend on it. It may go away in future releases of Mac OS X.

Advice for Fixing One Kind of Bad KVO Compliance

One internal Apple application was throwing the kind of exception described above. It looked like this:

```
2005-09-04 17:42:16.146 MyTestApp[936] *** -[NSAutoreleasePool dealloc]: Exception ignored while
releasing an object in an autorelease pool: Cannot remove an observer <NSArrayDetailBinder 0x3e48890>
for the key path "myBook.indexMarkersInBook" from <MyDocument 0x439930>, most likely because
the value for the key "myBook" has changed without an appropriate KVO notification being sent.
Check the KVO-compliance of the MyDocument class.
```

Here's what the accessor for the myBook property in the MyDocument class looked like:

```
- (Book *) myBook {
    return [[self myChapter] myBook];
}
```

That looks pretty simple, and the MyDocument class was KVO-compliant for the myChapter property. So what was the problem? The problem was that KVO can't infer that the value of myBook changes when the value of myChapter changes. Problems like this can be fixed in at least two ways:

- Use a different key path in bindings. In this example binding an NSArrayController to the document's "myChapter.myBook.indexMarkersInBook" instead of it's myBook.indexMarkersInBook" fixed the problem. Because MyDocument is KVO-compliant for myChapter in this example, and the Chapter class is KVO-compliant for myBook, everything works. This kind of solution isn't always desirable though; perhaps the author of the MyDocument class doesn't want to publish the fact that instances of it even have "chapters" to other parts of the program, for design reasons.
- Use +[NSObject(NSKeyValueObservingCustomization) setKeys:triggerChangeNotificationsForDependentKey:] to tell KVO that the value of MyDocument's myBook depends on the value of its myChapter. That way, with the implementation of -myBook above, MyDocument is KVO-compliant for myBook.

Bug Fix in Observing A Key Path Of Self

In Mac OS 10.3 and 10.4 there was a bug in which a debugging feature of KVO made it difficult for an object to observe one of its own values using a multicomponent key path: right before the class' -dealloc method would be invoked, with the object still as an observer of itself, Foundation would log something like "An instance 0x123456 of class MySelfObservingClass is being deallocated while key value observers are still registered with it. Break on _NSKVODeallocateLog to start debugging." Then if the object correctly removed itself as an observer of itself, an exception would be thrown or a crash would occur. This bug has been fixed in Mac OS 10.5. KVO still has a feature in which it logs a warning if an object is being deallocated with observers registered with it, but it now reliably ignores the object itself as an observer.

NSBundle

NSBundle Load Errors, Preflighting, and Architecture Detection

NSBundle now contains methods to allow it to return an NSError if bundle loading fails, to allow preflight testing of bundle loading without actually causing the bundle to be loaded, and to determine the processor architectures that a Mach-O executable contains. The method -loadAndReturnError: behaves like the existing -load method, except that if the optional by-reference error parameter is non-NULL and the load fails, then it will be filled in with a suitable NSError providing a description of the problem suitable for presenting to the user. If the method returns YES--that is, if the bundle was already loaded, or has now been successfully loaded--then the contents of the error parameter will not be touched. All of the errors returned are in the Cocoa error domain, and their codes are described in FoundationErrors.h. The caller does not gain a reference to a returned NSError, and should retain it if it wishes to hold on to it beyond the lifetime of the current autorelease pool. Note that if the error parameter is non-NULL and the load fails, then this method may do additional work over and beyond what -load would do, in order to determine the precise error type.

The current possible error codes are: NSFileNoSuchFileError, if the bundle's executable cannot be located; NSExecutableNotLoadableError, if the bundle's executable exists but is not a loadable executable; NSExecutableArchitectureMismatchError, if the bundle's executable exists but does not provide a version for the processor architecture of the current process; NSExecutableRuntimeMismatchError, if the bundle's executable exists but contains Objective-C runtime information that is not compatible with the current process; NSExecutableLoadError, if the bundle's executable fails to be loadable for some other reason detectable prior to linking, notably the lack of a required library, or the lack of an architecture- and runtime-compatible version of a required library; and NSExecutableLinkError, if the bundle's executable passes all other checks but fails to load due to link errors. Note that NSExecutableNotLoadableError will be returned if the bundle's executable is PEF/CFM but the current process does not support CFM. Other error codes may be added in future releases. More specific information may be shown in the error's debug description, available via -description or the gdb print-object command.

Similar information about load errors can also be obtained without actually loading the bundle, using the method -preflightAndReturnError:. This method returns YES if the bundle is already loaded, or if it appears to be loadable by all tests short of actual linking. The optional by-reference error parameter behaves like the error parameter to -loadAndReturnError:, except that it cannot return NSExecutableLinkError.

The -executableArchitectures method can be used to determine the list of processor architectures that a bundle's executable supports. If the executable is Mach-O, then this function returns an array of NSNumbers of integer type, each of which represents a cpu type for which the executable has a version. NSBundle declares constants for certain well-known types (NSBundleExecutableArchitecturePPC, NSBundleExecutableArchitecturePPC64, NSBundleExecutableArchitectureI386, and NSBundleExecutableArchitectureX86_64) but the values are taken directly from the executable, so other values may occur, and other values may be added in the future. If the executable is not Mach-O, then this method returns nil.

NSBundle Loading vs. CFBundle Loading

Previously it has been recommended that bundles containing Objective-C code or Cocoa Java code should be loaded using NSBundle rather than CFBundle. In Leopard, there are fewer differences between CFBundle and NSBundle loading, and bundles containing Objective-C code may be loaded with either. However, there are still a few remaining differences. First, bundles with an executable type of MH_BUNDLE (that is, loadable bundles rather than frameworks) loaded using CFBundle are loaded privately and with immediate binding, while the same bundles loaded using NSBundle are loaded globally and with lazy binding. Bundles with an executable type of MH_DYLIB (that is, frameworks) are loaded globally and with lazy binding in either case. Second, loading of bundles using NSBundle will cause NSBundleDidLoadNotification notifications to be sent, while loading of the same bundle using CFBundle will not produce these notifications. Third, bundles containing Cocoa Java code should still be loaded using NSBundle.

NSBundle Unloading

In Leopard, NSBundle exposes a - (BOOL)unload method. This method attempts to unload the bundle if it is loaded, and returns YES if the bundle is not loaded at the end of the call, i.e., if it was successfully unloaded or if it was not loaded in the first place. Success or failure in unloading depends on the underlying dynamic loader, usually dyld. (Note that in Leopard, dyld now supports the unloading of MH_DYLIB as well as MH_BUNDLE executables.) Bundles may also be unloaded using CFBundle, but it is recommended that bundles loaded with NSBundle be unloaded with NSBundle and those loaded with CFBundle be unloaded with CFBundle. As usual, it is the responsibility of the client to make sure that there are no dangling pointers to the code that is being unloaded, and in particular that there are no extant instances of classes defined in the bundle. This method exists in previous versions of Mac OS X, but always returns NO on pre-Leopard systems, because unloading of NSBundles is not supported on those systems.

NSFileManager

NSFileManager delegates and instances (Updated since WWDC 2007)

In versions of Mac OS X prior to Leopard, the only supported NSFileManager instance was the instance returned from +[NSFileManager defaultManager] (a singleton instance). Calling [[NSFileManager alloc] init] would return a new object instance of NSFileManager, but this instance might behave strangely in the face of multiple threads.

In Leopard, calling +[NSFileManager defaultManager] still gets you the same singleton instance no matter when or in what thread you call it, but calling [[NSFileManager alloc] init] is now explicitly supported and returns a new instance. In addition, NSFileManager instances can now have delegate objects. The delegate is set in the usual way:

```
- (void)setDelegate:(id)delegate;
- (id)delegate;
```

The delegate is not retained in usual operation. In GC, a strong reference is kept to the delegate.

NSFileManager copy/move/link/remove API (Updated since WWDC 2007)

Leopard introduces new API for copying, moving, linking and removing filesystem items.

```
- (BOOL)copyItemAtPath:(NSString *)srcPath toPath:(NSString *)dstPath error:(NSError **)error;
- (BOOL)moveItemAtPath:(NSString *)srcPath toPath:(NSString *)dstPath error:(NSError **)error;
- (BOOL)linkItemAtPath:(NSString *)srcPath toPath:(NSString *)dstPath error:(NSError **)error;
- (BOOL)removeItemAtPath:(NSString *)path error:(NSError **)error;
```

which replace the similarly-named methods currently extant in NSFileManager.

Events are passed to the delegate using the following methods defined in a category on NSObject:

```
- (BOOL)fileManager:(NSFileManager *)fileManager shouldCopyItemAtPath:(NSString *)srcPath toPath:(NSString *)dstPath;
- (BOOL)fileManager:(NSFileManager *)fileManager shouldProceedAfterError:(NSError *)error
                                                    copyingItemAtPath:(NSString *)srcPath toPath:(NSString *)dstPath;

- (BOOL)fileManager:(NSFileManager *)fileManager shouldMoveItemAtPath:(NSString *)srcPath toPath:(NSString *)dstPath;
- (BOOL)fileManager:(NSFileManager *)fileManager shouldProceedAfterError:(NSError *)error
                                                      movingItemAtPath:(NSString *)srcPath toPath:(NSString *)dstPath;

- (BOOL)fileManager:(NSFileManager *)fileManager shouldLinkItemAtPath:(NSString *)srcPath toPath:(NSString *)dstPath;
- (BOOL)fileManager:(NSFileManager *)fileManager shouldProceedAfterError:(NSError *)error
                                                    linkingItemAtPath:(NSString *)srcPath toPath:(NSString *)dstPath;
```

```
- (BOOL)fileManager:(NSFileManager *)fileManager shouldRemoveItemAtPath:(NSString *)path;
- (BOOL)fileManager:(NSFileManager *)fileManager shouldProceedAfterError:(NSError *)error removingItemAtPath:(NSString *)path;
```

Delegates in the "shouldXXXItemAtPath" methods have the opportunity to return "NO" to skip a particular item. Doing so does not trigger an error return at the top-level XXXItemAtPath:toPath:error: method. The default behavior is to act as if these methods return YES.

Delegates in the shouldProceedAfterError:XXXingItemAtPath:toPath: methods have the opportunity to return YES. This will suppress the error return from the top-level XXXItemAtPath:toPath:error:. Delegates in the shouldProceedAfterError: method can take the opportunity to attempt to fix the problem in order to let the operation continue. The default behavior is to act as if these methods return NO.

-[NSFileManager copyPath:toPath:handler:] and ACLs

Mac OS X 10.4 introduced Access Control Lists (ACLs), a sophisticated mechanism for access control on the filesystem. NSFileManager's copyPath:toPath:handler: method does not copy ACLs when copying a filesystem object. But the new Leopard API, copyItemAtPath:toPath:error:, does copy ACLs over.

NSFileManager APIs which return errors (Updated since WWDC 2007)

NSFileManager's error reporting has been improved in Leopard with the addition of a number of new methods which take a by-reference NSError parameter.

The following methods are new in Leopard and are intended as replacements for their non-error-taking counterparts:

```
- (BOOL)setAttributes:(NSDictionary *)attributes ofItemAtPath:(NSString *)path error:(NSError **)error;
- (BOOL)createSymbolicLinkAtPath:(NSString *)path withDestinationPath:(NSString *)destPath error:(NSError **)error;
- (NSArray *)contentsOfDirectoryAtPath:(NSString *)path error:(NSError **)error;
- (NSArray *)subpathsOfDirectoryAtPath:(NSString *)path error:(NSError **)error;
- (NSDictionary *)attributesOfItemAtPath:(NSString *)path error:(NSError **)error;
- (NSDictionary *)attributesOfFileSystemForPath:(NSString *)path error:(NSError **)error;
- (NSString *)destinationOfSymbolicLinkAtPath:(NSString *)path error:(NSError **)error;
- (BOOL)createDirectoryAtPath:(NSDictionary *)path
             withIntermediateDirectories:(BOOL)createIntermediates
             attributes:(NSDictionary *)attributes
             error:(NSError **)error;
```

Some methods have not been enhanced; for those methods there is no useful user-presentable error to be generated or the general utility was questionable. One example of a method of this type is -isWritableFileAtPath:. Race windows exist between testing for writability and actually writing; a far better approach is to attempt writing to the file and handling the errors that may occur gracefully.

Grammar Checking

Grammar checking is a new feature associated with the existing spellchecking functionality. Any spellchecking server has the option of also providing grammar checking, by implementing the NSSpellServer delegate method

```
- (NSRange)spellServer:(NSSpellServer *)sender checkGrammarInString:(NSString *)stringToCheck
             language:(NSString *)language details:(NSArray **)details;
```

The return value is intended to be the range of the next sentence or other grammatical unit that contains sections to be flagged for grammar, since grammar checking generally must be performed sentence by sentence. The details argument optionally returns by reference an array of dictionaries, each one describing a grammatical issue detected in the sentence (since a single sentence may contain more than one problem). In these dictionaries the following keys will be recognized:

```
NSString *NSGrammarRange;
NSString *NSGrammarUserDescription;
NSString *NSGrammarCorrections;
```

The value for the NSGrammarRange key should be an NSValue containing an NSRange, a subrange of the sentence range used as the return value, whose location should be an offset from the beginning of the sentence--so, for example, an NSGrammarRange for the first four characters of the overall sentence range should be {0, 4}. The value for the NSGrammarUserDescription key should be an NSString containing descriptive text about that range, to be presented directly to the user; it is intended that the user description should provide enough information to allow the user to correct the problem. A value may also be provided for the NSGrammarCorrections key, consisting of an NSArray of NSStrings representing potential substitutions to correct the problem, but it is expected that this may not be available in all cases. It is recommended that NSGrammarUserDescription be supplied in all cases; in any event, either NSGrammarUserDescription or NSGrammarCorrections must be supplied in order for something to be presented to the user. If NSGrammarRange is not present, it will be assumed to be equal to the overall sentence range. Additional keys may be added in future releases.

Immutable collections and copy behavior

In Mac OS X 10.4 and earlier, the default copy behavior of custom subclasses of NSArray, NSSet, and NSDictionary (the immutable variants) was to deep copy the contents of the collection. This meant that sending copy or copyWithZone: to any of these collections copied each element in the collection. This required that the contents of the collection conformed to the NSCopying protocol.

For applications linked on Tiger or earlier when running on Leopard, these collections will continue to perform a deep copy. For applications linked on Leopard or later, these collections will perform a shallow copy (and be more consistent with their mutable and CF equivalents).

If on Tiger you were using code like this:

```
NSArray *deepCopyOfArray = [someArray copyWithZone:nil];     // deep copy on Tiger
```

and relying on the deep copy behavior to get a deep copy, you would use the following code on both Tiger and Leopard:

```
NSArray *deepCopyOfArray = [[NSArray alloc] initWithArray:someArray copyItems:YES];   // explicitly deep copy
```

as a drop-in replacement (as the ownership rules are the same).

Collection construction methods and NS_REQUIRES_NIL_TERMINATION

There are six collection creation methods in Foundation which require nil termination to function properly. They are:

```
+[NSArray arrayWithObjects:]
-[NSArray initWithObject:]
+[NSDictionary dictionaryWithObjectsAndKeys:]
-[NSDictionary initWithObjectsAndKeys:]
```

```
+[NSSet setWithObjects:]
-[NSSet initWithObjects:]
```

These methods have been decorated with the NS_REQUIRES_NIL_TERMINATION macro, which adds an additional check to invocations of those methods to make sure the nil has been included at the end of the argument list. This warning is only available when compiling with -Wformat.

Order of values in hashing-based collections

The CoreFoundation and Foundation framework-provided implementations of hashing-based collections such as dictionaries have changed how they store elements, so elements may be retrieved in a different order than in previous releases. The order of elements in hashing-based collections is undefined and can change at any time, so developers must never rely on the order that elements are enumerated, or returned from a function like CFDictionaryGetKeysAndValues(). This is true even for cases where a developer may be trying to manipulate the hash codes of the objects in order to achieve some particular ordering.

NSSet

NSSet and NSMutableSet now have predicate-based methods for filtering. These are similar to the existing filtering methods on NSArray/NSMutableArray.

NSSet has the following new methods, which should be self-explanatory from the name:

```
- (NSSet *)setByAddingObject:(id)anObject;
- (NSSet *)setByAddingObjectsFromSet:(NSSet *)other;
- (NSSet *)setByAddingObjectsFromArray:(NSArray *)other;
```

NSIndexSet

Has the following new method for returning the number of indexes in the specified range:

```
- (NSUInteger)countOfIndexesInRange:(NSRange)range;
```

Advice for Invokers of -[NSUndoManager removeAllActions]

-[NSUndoManager removeAllActions] resets the "isUndoing" and "isRedoing" state of the undo manager. If your application invokes it during an undo or redo operation, the undo manager object will be left in an inconsistent state, causing hard-to-debug errors or crashes. Design your application so that this doesn't happen. There's no known good reason to send an undo manager a -removeAllActions message when it's in the middle of an undo operation. If necessary, you can use -isUndoing and -isRedoing to protect against doing such a thing.

NSGeometry.h and converting between CGRect and NSRect

In 32-bit code, CGRects and NSRects are distinct types. In order to aid in converting between CGRects and NSRects, NSGeometry.h defines 6 new functions:

```
NSRectFromCGRect
NSRectToCGRect
NSPointFromCGPoint
NSPointToCGPoint
NSSizeFromCGSize
NSSizeToCGSize
```

which do the pointer conversion between NSRects and CGRects.

-[NSData getBytes:range:] and NSRangeExceptions

-[NSData getBytes:range:] does not properly raise an NSRangeException for ranges which start inside the NSData, but end outside the NSData. Instead, it fills the provided buffer up to the end of the NSData. This will continue to be the case for Mac OS X 10.5 Leopard, but a future release of the operating system will start throwing range exceptions for applications linked later than Mac OS X 10.5.

NSSearchPathForDirectoriesInDomains and Developer directories

In Mac OS X 10.5 "Leopard" it is possible to have multiple versions of the Developer Tools software (Xcode, Interface Builder, etc.) installed on your system at one time. It is also possible to have them installed in a location that is not rooted at the "/Developer" path in the filesystem. As a result, the directories returned from NSSearchPathForDirectoriesInDomains by passing the NSDeveloperApplicationDirectory or NSDeveloperDirectory constants may not return something which corresponds to the actual location of the Developer Tools that are installed.

+[NSInputStream inputStreamWithData:] (New since WWDC 2007)

In Mac OS X 10.4 "Tiger", the NSData passed to +[NSInputStream inputStreamWithData:] would not be retained by the created stream. In Mac OS X 10.5 "Leopard" the NSData is now retained by the stream.

NSNetServices

It is an error to call -[NSNetService publish] on an NSNetService initialized with -[NSNetService initWithDomain:type:name:]. For applications linked on Mac OS X 10.5 "Leopard" or later, doing so will cause an exception to be thrown.

Applications which have been linked on Mac OS X 10.5 Leopard or later will raise an exception when passing nil to +[NSNetService dictionaryFromTXTRecordData:] or +[NSNetService dataFromTXTRecordDictionary:].

NSNetServices (New since WWDC 2007)

NSNetServices' behavior in handling timeouts is slightly different than what is provided by CFNetServices. In Leopard, CFNetServices will unconditionally report a timeout error when the timeout chosen in CFNetServiceResolveWithTimeout() is hit, regardless of whether or not any addresses were actually resolved.

NSNetServices will only report an NSNetServicesTimeoutError to the delegate's netService:didNotResolve: callback if no addresses were resolved by a call to -[NSNetService resolve] or -[NSNetService resolveWithTimeout:]. If addresses have been resolved, when the timeout is reached the delegate will be notified on its -netServiceDidStop: delegate method.

NSDateFormatter

NSDateFormatter has new methods:

```
- (NSArray *)longEraSymbols;
- (void)setLongEraSymbols:(NSArray *)array;

- (NSArray *)veryShortMonthSymbols;
- (void)setVeryShortMonthSymbols:(NSArray *)array;

- (NSArray *)standaloneMonthSymbols;
- (void)setStandaloneMonthSymbols:(NSArray *)array;

- (NSArray *)shortStandaloneMonthSymbols;
- (void)setShortStandaloneMonthSymbols:(NSArray *)array;

- (NSArray *)veryShortStandaloneMonthSymbols;
- (void)setVeryShortStandaloneMonthSymbols:(NSArray *)array;

- (NSArray *)veryShortWeekdaySymbols;
- (void)setVeryShortWeekdaySymbols:(NSArray *)array;

- (NSArray *)standaloneWeekdaySymbols;
- (void)setStandaloneWeekdaySymbols:(NSArray *)array;

- (NSArray *)shortStandaloneWeekdaySymbols;
- (void)setShortStandaloneWeekdaySymbols:(NSArray *)array;

- (NSArray *)veryShortStandaloneWeekdaySymbols;
- (void)setVeryShortStandaloneWeekdaySymbols:(NSArray *)array;

- (NSArray *)quarterSymbols;
- (void)setQuarterSymbols:(NSArray *)array;

- (NSArray *)shortQuarterSymbols;
- (void)setShortQuarterSymbols:(NSArray *)array;

- (NSArray *)standaloneQuarterSymbols;
- (void)setStandaloneQuarterSymbols:(NSArray *)array;

- (NSArray *)shortStandaloneQuarterSymbols;
- (void)setShortStandaloneQuarterSymbols:(NSArray *)array;
```

These set and return:
- "long" era names, for example "Anno Domini" instead of "AD"
- "very short" names for months and weekdays; for example, "F" instead of "Friday"
- "standalone" names for months and weekdays; for some locales/languages, a month name displayed in isolation needs to be written differently than a month name within a displayed date
- names of quarters; for example, "Q2" for a short quarter name

There is also two new methods to set and get the switchover date to be used for the Gregorian calendar:

```
- (NSDate *)gregorianStartDate;
- (void)setGregorianStartDate:(NSDate *)date;
```

This is used to specify the start date for the Gregorian calendar switch from the Julian calendar. Different locales switched at different times. Normally you should just accept the locale's default date for the switch.

New NSNumberFormatter API

There is some new API in NSNumberFormatter in 10.5:

```
- (NSString *)currencyGroupingSeparator;
- (void)setCurrencyGroupingSeparator:(NSString *)string;
```

This is the grouping separator for currency values, which could be different than the usual grouping separator in some locales.

```
- (BOOL)isLenient;
- (void)setLenient:(BOOL)b;
```

May enable greater leniency in parsing strings into numbers in some operating system releases when true.

```
- (BOOL)usesSignificantDigits;         // whether to use the next two or not
- (void)setUsesSignificantDigits:(BOOL)b;

- (NSUInteger)minimumSignificantDigits;
- (void)setMinimumSignificantDigits:(NSUInteger)number;

- (NSUInteger)maximumSignificantDigits;
- (void)setMaximumSignificantDigits:(NSUInteger)number;
```

These are useful for scientific notation formatting.

```
- (BOOL)isPartialStringValidationEnabled;
- (void)setPartialStringValidationEnabled:(BOOL)b;
```

These methods are used to enable partial string validation, which by default NSNumberFormatters do not do (and have never done). However, the implementation in 10.5 doesn't do anything with this property.

Default Formatter behavior

When a new NSNumberFormatter or NSDateFormatter is created in apps linked on 10.5 and later, the default behavior setting for that formatter is the "10_4" behavior. It is still a good idea to set the desired behavior of a formatter explicitly just after creating it (when doing so in code) to make the intent clear.

New NSMachPort API

You can now specify some deallocation options to NSMachPort when you create one with a Mach port:

```
enum {
    NSMachPortDeallocateNone = 0,
    NSMachPortDeallocateSendRight = (1 << 0),
    NSMachPortDeallocateReceiveRight = (1 << 1)
};
```

```
+ (NSPort *)portWithMachPort:(uint32_t)machPort options:(NSUInteger)f;
- (id)initWithMachPort:(uint32_t)machPort options:(NSUInteger)f;
```

These options let you specify whether the NSMachPort should effectively take ownership of a send right reference and/or a receive right references for the port when you create the port object. When the port object is deallocated, it will deallocate the specified rights. You should of course understand what all this means for a Mach port, and understand what type of references you may have, before using these new options. Plus, they come with a major caveat: your call must be the first the create a port object with that Mach port, or your options will be ignored, and there is no good way to tell if your options were ignored.

New NSCalendar APIs

NSCalendar has the following new API to compute the time range for a given unit surrounding a given moment:

```
- (BOOL)rangeOfUnit:(NSCalendarUnit)unit startDate:(NSDate **)datep
           interval:(NSTimeInterval *)tip forDate:(NSDate *)date;
```

The second and third parameters are out parameters.

For example, to find the starting moment and length of the week around the current moment, for a given calendar, you might do:

```
NSDate *start;
NSTimeInterval length;
Boolean ret = [calendar rangeOfUnit:NSCalendarUnitWeek startDate:&start interval:&length forDate:[NSDate date]];
if (ret) { ...
```

Keep in mind that the time at (start + length) will usually be in the next occurrence of the unit, not in the current one, so you may need to fudge the arithmetic slightly to work with the end date of the unit. Also, some values have no defined end point (like usually, the current era), so an arbitrary point in the future is chosen.

```
+ (id)autoupdatingCurrentCalendar;
```

Returns a special NSCalendar instance which always reflects the current state of the current user's calendar settings. The existing currentCalendar method will continue to return the snapshot that was current at the time it's called.

Chinese calendar still not available

Although there is a constant for it, the Chinese calendar is not available from NSCalendar or related APIs yet. In 10.5, trying to create a NSCalendar with that constant will return nil (fail).

Calendrical computations

Some calendrical computations have been corrected in 10.5. In particular -[NSCalendar rangeOfUnit:inUnit:forDate:] has changed to a more literalist interpretation of "range" for applications linked on or after 10.5.

Use of NSCalendarDate discouraged

Use of the NSCalendarDate class is again discouraged in 10.5, but it is not deprecated yet. It may be in the next major OS release after 10.5. NSDateFormatter should be used for date formatting, and NSCalendar should be used for calendrical calculations. Where formatting differs between NSCalendarDate and NSDateFormatter (10.4-style) differ, the NSDateFormatter result will be considered to be the correct result (absent a bug being exercised). Where NSCalendar and NSCalendarDate calendrical calculations differ and the NSCalendar result is reasonable, we define its value to be the correct value. Developers should abandon hope for NSCalendarDate bug fixes.

New NSTimeZone API

```
- (NSTimeInterval)daylightSavingTimeOffsetForDate:(NSDate *)aDate;
- (NSDate *)nextDaylightSavingTimeTransitionAfterDate:(NSDate *)aDate;
```

The first method returns the daylight saving time offset from GMT at the given moment, or 0.0 if there is no data for that time. The second method returns the next moment after the given time at which a daylight saving time occurs, or nil if there is no data after that time.

```
- (NSTimeInterval)daylightSavingTimeOffset;
```

```
- (NSDate *)nextDaylightSavingTimeTransition;
```

These two are convenience methods for the two above, with the date taken to be the current time.

```
enum {
    NSTimeZoneNameStyleStandard,
    NSTimeZoneNameStyleShortStandard,
    NSTimeZoneNameStyleDaylightSaving,
    NSTimeZoneNameStyleShortDaylightSaving
};
typedef NSInteger NSTimeZoneNameStyle;

- (NSString *)localizedName:(NSTimeZoneNameStyle)style locale:(NSLocale *)locale;
```

This API returns the localized display string for the given time zone and locale, in the given style. If the data does not exist for that combination of parameters, returns nil.

NSTimeZone has the following new notification name API:

```
NSString * const NSSystemTimeZoneDidChangeNotification;
```

This notification is posted when the system time zone changes. The object of the notification is the previous system time zone object. This notification carries no user info. Keep in mind that there is no order in how notifications are delivered to observers, and frameworks or other parts of your code may be observing this notification as well to take their own actions, which may not have occurred by the time you receive the notification.

New NSPortNameServer API

```
- (NSPort *)servicePortWithName:(NSString *)name;
```

This method can be used to look up an auto-launched service port. This method is used by the service itself (if anybody) at launch time to check in with launchd.

New NSConnection API

```
+ (id)serviceConnectionWithName:(NSString *)name rootObject:(id)root usingNameServer:(NSPortNameServer *)server;
+ (id)serviceConnectionWithName:(NSString *)name rootObject:(id)root;
```

These new methods are used to create a new NSConnection by an autolaunched service process, for the service(s) they are providing. The connection is configured with the given root object. -registerName: should not be used with such an NSConnection, and -setRootObject: is unnecessary. This also checks the service in with launchd.

New NSRunLoop APIs

```
+ (NSRunLoop *)mainRunLoop;
```

Returns the run loop object of the main thread. At the present time, this method is of dubious value, since some NSRunLoop functionality is not thread-safe (all "perform" functionality in NSRunLoop.h, for example).

```
NSString * const NSRunLoopCommonModes;
```

This is a new constant for NSRunLoop that corresponds to CFRunLoop's kCFRunLoopCommonModes constant. Refer to CFRunLoop's documentation for more information.

NSURLConnection (Updated since WWDC 2007)

NSURLConnection now supports scheduling on individual run loops. If the delegate wishes to receive the delegation messages on a thread other than the current thread, they can create the connection using -initWithRequest:delegate:startImmediately: specifying NO for the last argument (startImmediately). They can then choose where to schedule the delegation methods using -scheduleInRunLoop:forMode: and -unscheduleFromRunLoop:forMode:. Once the connection has been scheduled, it can be started using -start. Once a connection has been started, its scheduling cannot be changed.

New constants NSURLRequestReloadIgnoringLocalAndRemoteCacheData and NSURLRequestReloadRevalidatingCacheData have been added to the list of caching policies on NSURLRequest. However, they are not implemented; they are simply placeholders at this time.

The delegate method connection:willSendRequest:redirectResponse: has changed behavior in 10.5. First, prior to 10.5, its behavior was inconsistent if nil was returned; sometimes a nil return would cancel the connection, but sometimes, a nil return would cause the connection to use the given request unmodified. In 10.5., a nil return will always cancel the connection; this matches the documented behavior. Second, prior to 10.5, NSURLConnection would often modify the incoming NSURLRequest prior to transmission without notifying the delegate. In 10.5, the delegate is always notified via the delegation method connection:willSendRequest:redirectResponse:. This means that the delegate will often receive a connection:willSendRequest:redirectResponse: message before the connection has even properly begun, prior to transmitting the request to the remote server. This may be surprising to older code, written assuming connection:willSendRequest:redirectResponse: will only be called if a redirection has occurred. To fix such code, simply check whether the redirectResponse is nil. If it is, no redirection has occurred and NSURLConnection is simply informing the delegate that it modified the request prior to transmission. If the delegate does not want to intervene, it should return the request unmodified. Here's an example:

```
- (NSURLRequest *)connection:(NSURLConnection *)connection
             willSendRequest:(NSURLRequest *)request
            redirectResponse:(NSURLResponse *)redirectResponse {
  if (redirectResponse) {
    // Handle the redirection; older code goes here
    ...
  } else {
    // Just being shown the final request prior to transmission
    return request;
  }
}
```

New NSLocale APIs

```
+ (NSArray *)commonISOCurrencyCodes;
```

Returns an array of CFStrings that represents ISO currency codes for currencies in common use, which is a more generally useful set than that returned by +ISOCurrencyCodes.

```
+ (NSArray *)preferredLanguages;
```

Returns an array of NSStrings giving the user's preferred language identifiers, canonicalized, in order.

NSTimeZone has the following new notification name API:

```
NSString * const NSCurrentLocaleDidChangeNotification;
```

This is a local notification posted when the user changes locale information in the System Preferences panel. Keep in mind that there is no order in how notifications are delivered to observers, and frameworks or other parts of your code may be observing this notification as well to take their own actions, which may not have occurred by the time you receive the notification. There is no object or user info for this notification.

```
+ (id)autoupdatingCurrentLocale;
```

Returns a special NSLocale instance which always reflects the current state of the current user's locale settings. This is useful in assigning to NSDateFormatter, for instance. This autoupdating instance will appear to have changed by the time NSCurrentLocaleDidChangeNotification is sent out.

The existing currentLocale method will continue to return the snapshot that was current at the time it's called.

Methods which take locale arguments

Some methods in Foundation used to take an NSDictionary * as a locale: (or Locale:) argument. The types of these arguments on those methods have been changed to id. For example:

```
- (NSString *)descriptionWithLocale:(NSDictionary *)locale;        // previously
- (NSString *)descriptionWithLocale:(id)locale;                 // now
```

Such methods that are implemented in Foundation now accept either an old-style NSDictionary locale dictionary, or an NSLocale * locale. Use of NSLocales in new code is encouraged.

Note that some existing methods in Cocoa are typed to take an NSLocale * locale argument, and these still only accept an NSLocale object.

Methods which take locale dictionaries which are not in Foundation, CoreData, or AppKit (Cocoa itself) have not necessarily been improved with this capability. This includes overrides of these methods in subclasses which are not in Cocoa.

NSDate's -descriptionWithLocale: accepts either, but ignores locale dictionary arguments in 10.5 and later, giving results based on the current user's locale for non-nil locale dictionaries. NSLocale objects as the argument are honored.

Finally note that the NSCalendarDate class still only takes the old-style locale dictionary style locale parameters for its methods which take locales.

Avoid NSMessagePort

There's little reason to use NSMessagePort rather than NSMachPort or NSSocketPort. There's no particular performance or functionality advantage. We recommend avoiding it. It may be deprecated in the next major OS release after 10.5.

Avoid 'long double' and '_Complex' types

We strongly recommend avoiding all use of the "long double" type and the various C99 "_Complex" types other than -- at most -- purely local computational usage. In particular, we recommend not using it for parameter types; not using it for return values; not using or passing pointers to long double or _Complex types; not using arrays of long doubles or _Complex types or arrays of pointers to long doubles or _Complex types as parameters; not embedding long doubles or _Complex types or pointers to long doubles or _Complex types inside structures which are passed as parameters or returned as return values, nor using pointers to such structures; not using long doubles or _Complex types as object instance variable types; and so on. Do not mix Objective C method calls or variables with any long double or _Complex type usage; in other words, and do not attempt to use long doubles or _Complex types with any Cocoa API. You may notice that Cocoa does not offer any "long double" or "_Complex" type APIs (such as in NSNumber) -- we are avoiding them too.

Part of the issue is that the compiler emits "d" (double) as the type encoding string for @encode(long double), so there is no way to distinguish between the two, and this affects everything which uses Objective C type encoding strings, including key-value coding, forwarding and archiving. If the compiler were ever to fix that now, an app using long double will have binary compatibility problems.

Similarly for the _Complex types, the compiler emits "" (an empty string) for @encode(_Complex {float,double,long double}). Thus these types are completely invisible in method parameters and structure encodings and so on. This causes no end of havoc.

NSMethodSignature, NSInvocation

NSMethodSignature and NSInvocation were rewritten in 10.5. This probably means that there are some things behave differently on 10.5 than they did on 10.4. In particular, there were some bugs fixed, but there may be other subtle changes, undocumented behaviors that you may have been relying on. If you want your app to run well on 10.4 and 10.5, be sure to test on both.

NSInvocation notes on setting and retention of arguments and return values (New since WWDC 2007)

In 10.5, NSInvocation does not retain arguments or return values, unless -retainArguments is called on it. The primary purpose of -retainArguments is to cause the receiving invocation to copy or hold onto certain types of arguments as long as the invocation survives. You would do this if you are saving invocations for later use, or any use on another thread. For example, you would generally want to use this even if synchronously waiting on one thread for an invocation to finish being invoked on another, since the autorelease pools and garbage collector on the other thread are cleaning up objects asynchronously with the waiting thread.

Unfortunately, there is too much ambiguity and historical compatibility in the Objective C language and libraries for NSInvocation to "get this right" all the time for everybody. For example, for some methods, you may want a pointer return value preserved and returned unchanged, and for others, you might want the pointed-to items to be copied shallowly, or at other times deeply, in order to preserve the data being pointed to. In different contexts, you may even want different behaviors from NSInvocation for the same method being invoked. So NSInvocation implements some generic generally useful behaviors. Some of

these behaviors may vary from those on 10.4 and earlier, which were a little more ad hoc.

In 10.5, the candidate types for retention are Objective C object [pointer] types, fixed-length arrays, and C strings. Note that there is an inherent ambiguity in the type "char *", which can variously mean, in Objective C, "pointer to single char", "pointer to array of char of some known length", and "pointer to null-terminated array of char". The Objective C compiler and runtime have historically chosen to interpret "char *" (and variations) as a C string (null-terminated array) pointer, and this tradition continues. You cannot use the other two interpretations with anything that uses the runtime type metadata, such as forwarding (NSInvocations) or key-value coding.

For Objective C object [pointer] types, like "NSArray *", the parameter and return objects are retained when -retainArguments has been invoked. For fixed-length arrays not embedded in a struct, the array is copied, and the pointer to the copy becomes the new argument or return value. For C strings, the C string is copied, and the pointer to the copy becomes the new argument or return value. NSInvocation also, recursively, retains (or copies) the Objective C object and C string types within C struct and C array arguments and return values (fixed-length arrays embedded within a struct are part of the struct).

Other pointer uses are not candidates for retention. For example, if a method takes an "int *" argument, NSInvocation takes no action (even when -retainArguments has been called) to copy/preserve the int which that pointer is pointing to. If you manually set that argument (-setArgument:atIndex:), you set the pointer only, not [also] the value being pointed to (and of course, since the first argument to -setArgument:atIndex: is a pointer to the value to be set, it should be a pointer to the pointer-to-int to be set, not the pointer-to-int itself). This non-copying behavior is generally desireable, since you usually don't want NSInvocation to attempt to copy a FILE * or a C++ object (which, as far as the Objective C runtime is concerned, is just a pointer to a struct, and no copy constructor would be invoked) parameter or return value.

Arithmetic types are naturally preserved/copied due to their simple nature, when set or captured as part of forwarding or NSInvocation invocation. Structs passed by value are also preserved/copied when -retainArguments has been invoked.

Note that for an invocation object that has been told to -retainArguments, which has arguments set on it multiple times, or is invoked multiple times, accumulates those retainable arguments and return values, and holds them for the lifetime of the invocation.

For a somewhat higher degree of safety/compatibility, for applications linked on or before 10.4, NSInvocations automatically retain return values after -invoke.

New NSMethodSignature API

```
+ (NSMethodSignature *)signatureWithObjCTypes:(const char *)types;
```

This method, available since Mac OS X 10.0, has been exposed. The method creates an NSMethodSignature for the given Objective C method type string. It is entirely the caller's responsibility to pass in type strings which are either from the current runtime data or match the style of type string in use by the runtime that the application is running on. Only type encoding strings of the style of the runtime that the application is running against are supported. If you hard-code method type strings into your framework or application and try to use this method, you may be unpleasantly surprised by any changes in type encoding strings. In exposing this method there is no commitment to binary compatibily supporting any "old-style" type encoding strings after such changes occur.

New NSObject class methods for resolving methods at runtime

Objective-C now gives a class the opportunity to dynamically resolve method implementations at runtime. A class can override +resolveInstanceMethod: and use class_addMethod() to add a method implementation to the class. +resolveClassMethod: is also provided for dynamically resolving class methods. These methods are invoked before the forwarding machinery is invoked, when an object does not implement a method.

Both methods return a BOOL used to indicate whether the selector in question was resolved. If the method returns NO, the Objective-C runtime will pass control to the method forwarding mechanism. If it is resolved, the method will be invoked. Once resolved, all future invocations will directly execute the methods in the same fashion that any other method is executed. Generally you should invoke super's implementation of this method first to give the super class a chance to resolve the method itself. If that returns YES, you shouldn't need to do anything yourself, just return the YES back to your caller. The NSObject class implements these methods with a default implementation that returns NO today, though you should still invoke it even if you're just directly subclassing NSObject.

New forwarding fast path

When the forwarding mechanism is invoked (that is, when an object does not respond to a message sent to it), the forwarding mechanism in 10.5 first attempts to send this message to the original receiver object of the message:

```
- (id)forwardingTargetForSelector:(SEL)sel;
```

If the object implements (or inherits) this method, and returns a non-nil and non-self result, that returned object is used as the new receiver object and the message dispatch resumes to that new object. (Obviously if 'self' were allowed to be returned from the method, the code would just fall into an infinite loop.) Thus this method gives an object a chance to redirect an unknown message sent to it before the much more expensive forwardInvocation: machinery takes over. This is useful in basic proxying situations and can be an order of magnitude faster than regular forwarding. It is not useful where the goal of the forwarding is to capture the NSInvocation, or manipulate the arguments or return value during the forwarding.

Non-root classes which implement this method should invoke super's implementation of this method if the class decides that it has nothing to return for the given selector, and return super's result.

Old forward:: Objective C method

The Objective C runtime no longer attempts to invoke the -forward:: method on an object when an object does not respond to a message which is sent to it. Cocoa replaces the runtime forwarding mechanism entirely with its own version in 10.5, which propagates the well-known methodSignatureForSelector: and forwardInvocation: methods to objects so they can forward methods.

Changes to NSZombie debug mechanism

NSZombieEnabled now works for general CFType objects, including toll-free bridged ones. If NSZombieEnabled is set to "YES", then the CFZombieLevel environment variable is ignored.

In 10.5, when zombies are enabled, a message something like this is logged and a debugger trap is invoked:

```
2006-09-09 19:24:31.122 MyTestApp[402] *** -[NSURLConnection release]: message sent to deallocated instance 0x2141e30
```

Thus you do not have to set breakpoints on lots of potential methods and/or re-run the application (assuming you ran it under the debugger the first time) and try to reproduce the problem in order to find out where it is happening.

NSPropertyListSerialization (Updated since WWDC 2007)

A previous release note in the Leopard Foundation release notes said that a leak has been fixed in the two NSPropertyListSerialization methods that return an NSString * error description by reference. This fix has been reverted, and will never be done. As stated in the documentation for those methods, and as was true in 10.4 and earlier, it is the client's responsibility to release that string (if either method returns nil) in 10.5 and beyond as well.

NSTimeZone names cannot be nil

The NSTimeZone methods:

```
- (id)initWithName:(NSString *)tzName;
- (id)initWithName:(NSString *)tzName data:(NSData *)aData;
```

and indirectly, any class methods which call them, now insist on the tzName and aData parameters not being nil. In prior OS releases this would have caused a crash. In 10.5 and later, this is now more explicitly an invalid argument exception.

NSDate method behavior changes (Updated since WWDC 2007)

NSDate methods which take an NSDate * argument have historically had undefined/arbitrary behavior when passed nil. In particular they would tend to use an unpredictable floating point value for nil's timeIntervalSinceReferenceDate. Now they behave as if nil's time interval is NaN.

When comparing against nil or an NSDate with NaN as its time interval, NSOrderedSame is consistently returned. Previously the return value was a little arbitrary. However, nil is not equal to any date object, and only a NaN-holding date can be equal to a NaN-holding date.

The -earlierDate: and -laterDate: methods tended to return their arguments when passed nil, a NaN-holding NSDate, or a date which was equal to the receiver, but not in all cases. In 10.5 and later, these methods return the receiver object for these ambiguous cases.

NSString

There are several new NSStringEncoding values:

```
NSUTF16StringEncoding
NSUTF16BigEndianStringEncoding
NSUTF16LittleEndianStringEncoding
NSUTF32StringEncoding
NSUTF32BigEndianStringEncoding
NSUTF32LittleEndianStringEncoding
```

NSUTF16StringEncoding is simply an alias for the existing NSUnicodeStringEncoding.

When creating an NSString from bytes, NSUTF16BigEndianStringEncoding and NSUTF16LittleEndianStringEncoding enable you to specify the endianness of a UTF-16 stream of bytes explicitly, without consulting a BOM character. Note that as a consequence, if there is a BOM character in the stream, it will be interpreted as an actual character and included in the created NSString.

The NSUTF32StringEncoding, NSUTF32BigEndianStringEncoding, and NSUTF32LittleEndianStringEncoding values are similar to their UTF16 counterparts, except they work with 4-byte Unicode characters.

Although these encoding names are being introduced in Leopard headers, the actual values do work back to Tiger.

NSProprietaryStringEncoding has been deprecated, since it has not actually been used at all since 10.0.

A clarification: The maxLength argument to getCString:maxLength:encoding: needs to accomodate both the returned bytes and the additional NULL byte. In this regard this method behaves differently than the deprecated getCString:maxLength: method.

The previous documentation and comment in NSString.h were both wrong in this regard. Rather than changing the implementation and introducing compatibility concerns, we are updating the documentation.

NS and CFString now check most immutable creation requests against a set of "popular" strings, and in some cases will return a pre-created string. What this means is that it's now more likely that two distinct creation requests for an NS or CFString with the same contents might return the same exact string; don't count on pointer inequality in those cases. It's also possible that in those cases any unbalanced retain/release requests will go unnoticed.

Note that the set of "popular" strings will change between releases, so do not make any assumptions about what strings are shared based on observations on Leopard. In fact, the set has been updated significantly between the WWDC 2007 seed and final release of Leopard.

The following new method enables convenient way grow a range to include all composed character sequences it overlaps. This can be used with substringWithRange:, etc, to get back "proper" substrings that don't break composed character sequences:

```
- (NSRange)rangeOfComposedCharacterSequencesForRange:(NSRange)range;
```

For convenience, a zero-length range is treated like a 1-length range, except when the range is at the end of the string (so, {length,0}), in which case it will return {length,0}. So for the string "abc", {0,0} will return {0,1}, and {3,0} will return {3,0}.

The following method enables incremental encoding conversion NSStrings. Pass the desired range to be converted, and get back the remaining range. This method returns YES if it was able to convert any characters, even if it had to stop at an unconvertible character. (This would mean the next conversion would return NO.)

```
enum {
    NSStringEncodingConversionAllowLossy = 1,
    NSStringEncodingConversionExternalRepresentation = 2
};
typedef NSUInteger NSStringEncodingConversionOptions;

- (BOOL)getBytes:(void *)buffer
       maxLength:(NSUInteger)maxBufferCount
      usedLength:(NSUInteger *)usedBufferCount
```

```
        encoding:(NSStringEncoding)encoding
         options:(NSStringEncodingConversionOptions)options
           range:(NSRange)range
  remainingRange:(NSRangePointer)leftover;
```

Although this method has been publicized in 10.5, it does exist back to 10.4.

The following method returns an array of strings resulting from dividing the receiver like componentsSeparatedByString: does, but at boundaries identified by any of the characters in the argument.

```
- (NSString *)componentsSeparatedByCharactersInSet:(NSCharacterSet *)set;
```

The following methods are for various convenient replacements in NSString. They already exist in NSMutableString.

```
- (NSString *)stringByReplacingOccurrencesOfString:(NSString *)target
                                        withString:(NSString *)replacement
                                           options:(NSUInteger)optionMask
                                             range:(NSRange)searchRange;

- (NSString *)stringByReplacingOccurrencesOfString:(NSString *)target
                                        withString:(NSString *)replacement;

- (NSString *)stringByReplacingCharactersInRange:(NSRange)range withString:(NSString *)replacement;
```

longLongValue parses a long long out of the string, using "loose" rules much like the existing intValue and floatValue methods. Will skip initial white characters (whitespaceSet) and ignore characters after the number. Use NSScanner for more formalized parsing:

```
- (long long)longLongValue;
```

boolValue will parse a boolean. Skips initial space characters (whitespaceSet), or optional -/+ sign followed by zeroes. Returns YES on encountering one of "Y", "y", "T", "t", or a digit 1-9. It ignores any trailing characters.

```
- (BOOL)boolValue;
```

The following new typedef is used in NSString APIs to indicate arguments which take a combination of compare options (such as NSCaseInsensitiveSearch, NSBackwardsSearch, etc):

```
typedef NSUInteger NSStringCompareOptions;
```

NSString now supports the formatting characters %L, %a, %A, %F, %z, %t, and %j when formatting strings. %n, which never worked properly (it always returned value of zero), is now disabled for applications linked on Leopard. %n in the format string is still processed as before, but the corresponding argument is no longer written to.

The locale argument for -compare:options:range:locale: now accepts NSLocale. When nil, it performs non-localized comparison. For backward compatibility, it treats non-NSLocale objects as if it is +[NSLocale currentLocale].

There is a new searching method taking a locale argument. When nil, it performs non-localized search.

```
- (NSRange)rangeOfString:(NSString *)aString options:(NSUInteger)mask range:(NSRange)searchRange locale:(NSLocale *)locale;
```

The following new constants are introduced for the search/compare option flag.

Ignore diacritics (o-umlaut == o):

```
NSDiacriticInsensitiveSearch = 128
```

Ignore width differences ('a' == UFF41):

```
NSWidthInsensitiveSearch = 256
```

Force comparisons to return either kCFCompareLessThan or kCFCompareGreaterThan if the strings are equivalent but not strictly equal, for stability when sorting (e.g. "aaa" > "AAA" with kCFCompareCaseInsensitive specified):

```
NSForcedOrderingSearch = 512
```

There is now a method "folding" characters:

```
- (NSString *)stringByFoldingWithOptions:(NSUInteger)options locale:(NSLocale *)theLocale;
```

Folding string returns a canonical representative of the class of strings that are equivalent with respect to the options passed. For example, if you fold two strings that differ only with respect to case, you will get the same string back. This is useful for making dictionaries keyed by case-insensitive strings or similar.

NSString plain text file encoding support

The methods writeToFile:atomically:encoding:error:, writeToURL:atomically:encoding:error:, the deprecated writeToFile:atomically:, and writeToURL:atomically: now store the specified encoding with the file, in an extended attribute. The methods initWithContentsOfFile:usedEncoding:error:, initWithContentsOfURL:usedEncoding:error:, and their stringWith... counterparts use this information to open the file using the right encoding.

The extended attribute is stored under the name "com.apple.TextEncoding". The value contains the IANA name for the encoding and the CFStringEncoding value for the encoding, separated by a semicolon. The CFStringEncoding value is written as an ASCII string containing an unsigned 32-bit decimal integer. The string is not terminated by a NUL character.

One or both of these values may be missing. If the IANA name is missing but the CFStringEncoding value is present, the semicolon should still be there. Foundation consults the encoding number first, then the IANA name.

Examples of the value written to extended attributes include:

```
MACINTOSH;0
UTF-8;134217984
UTF-8;
;3071
```

Note that in the future the attribute may be extended compatibly by adding additional information after what's there now, so any readers should be prepared for an arbitrarily long value for this attribute, with stuff following the CFStringEncoding value, separated by a non-digit.

initWithContentsOfURL:encoding:error:, initWithContentsOfURL:usedEncoding:error:, and their stringWith... counterparts now work with data that is http compressed, decompressing it first if needed. initWithContentsOfURL:usedEncoding:error: and its stringWith... counterpart now try to pick up the text encoding from http headers.

In the absence of all other encoding information, where they would have returned nil in Tiger, initWithContentsOfFile:usedEncoding:error:, initWithContentsOfURL:usedEncoding:error:, and their stringWith... counterparts now try UTF-8 as a fallback. So any ASCII file or file which looks like a valid UTF-8 stream will be loaded and reported as UTF-8.

NSString deprecations

NSString cString and related methods which have been deprecated since 10.4 are now formally marked as deprecated. All the deprecated methods have newer counterparts (added 10.4 or earlier) which take explicit encoding arguments.

NSMaximumStringLength is now deprecated. It does not work in 64bit applications. Use NSUIntegerMax instead.

NSAttributedString

Creation methods initWithString:attributes: and initWithString: now check for nil string argument and log (only once per session). The intent is to have this raise an exception for apps linked post-Leopard.

NSCharacterSet

NSCharacterSet now has a new factory method, +newlineCharacterSet.

The return type for all the factory methods is now (id) instead (NSCharacterSet *) eliminating the need for casting when creating mutable instances.

NSScanner

The following method is analogous to scanHexInt::

```
- (BOOL)scanHexLongLong:(unsigned long long *)result;
```

The following scan hex-format floating point values written out with %a or %A format characters in NSString or printf(). These require a 0x or 0X prefix:

```
- (BOOL)scanHexFloat:(float *)result;
- (BOOL)scanHexDouble:(double *)result;
```

CFError

A new CoreFoundation type, CFError, has been added for error management. It is toll-free bridged with NSError and provides the same level of functionality.

We don't expect Cocoa applications to make direct use of CFError, but we mention it here since it does enable lower-level non-Foundation based APIs to provide errors that can be automatically presented by Cocoa's error presentation and handling machinery.

As with NSError, providers of CFErrors are encouraged to make sure errors have user-presentable error messages that enable them to be presented with little or no extra work.

NSError

NSError now has two additional codes to for additional error conditions on file reading: NSFileReadTooLargeError, NSFileReadUnknownStringEncodingError.

NSValue

NSValue will now provide better descriptions for often-used structs such as NSRect, NSSize, NSPoint, and NSRange.

NSPredicate

Two new operators have been added: NSContainsPredicateOperatorType is intended to complement the NSInPredicateOperatorType, since IN and CONTAINS are not directly invertible when predicate modifiers are considered (ie 'ANY X in Y' does not achieve the same effect as 'ANY Y contains X'). NSBetweenPredicateOperatorType allows for more simplistic construction and evaluation of a common type of compound predicate that has a direct mapping in some external technologies to which predicates are mappable (ie X BETWEEN {LOWERBOUND, UPPERBOUND}). Between also allows for more efficient database operations than (x >= lower && x <= upper) due to the handling of indices.

The following new method is optimized for situations which require repeatedly evaluating a predicate with substitution variables with different variable substitutions:

```
- (BOOL)evaluateWithObject:(id)object substitutionVariables:(NSDictionary *)variables;
```

Evaluates the specified object, substituting in the values in the variables dictionary for any replacement tokens. This method returns the same result as the two step process of first invoking predicateWithSubstitutionVariables:, and then invoking evaluateWithObject:.

NSCompoundPredicate

For applications linked on Mac OS X 10.5 "Leopard" or later, initializing an NSCompoundPredicate now copies the subpredicates array rather than retaining it. Applications linked on Mac OS X 10.4 "Tiger" continue to only retain the subpredicates array for binary compatibility.

NSExpression

New expression types NSSubqueryExpressionType, NSAggregateExpressionType, NSUnionExpressionType, NSIntersectExpressionType, and NSMinusExpressionType permit CoreData to generate much more efficient SQL. On 10.4, working around the absence of these operations requires developers fetch intermediate results (wrapped as objects) and perform some operations in memory, which has impacted scalability. With these expressions, CoreData can offload substantially more work on the underlying database and avoid bringing otherwise unnecessary rows into memory.

NSExpression now also has a constructor for NSFunctionExpressionType which enables creating your own functions for use in predicate evaluation.

Scripting

Improved Error Sensing and Reporting in Scriptability

In Mac OS 10.5 many improvements have been made to Cocoa Scripting's sensing and reporting of errors. Among them:
• Cocoa's mechanism for converting Apple event object specifiers to NSScriptObjectSpecifiers and evaluating them now limits itself to using the standard error codes listed in the AppleScript documentation when reporting errors, and always tries to choose the most descriptive one. Your scripters won't have to figure out what "NSReceiverEvaluationScriptError: 4" means anymore. You can do the same in your application's code; it's completely valid to pass error codes declared in <CarbonCore/MacErrors.h> to -[NSScriptCommand setScriptErrorNumber:] or -[NSScriptObjectSpecifier setEvaluationErrorNumber:].
• NSIndexSpecifier evaluation now does much better range checking, and will always return an error for an invalid index instead of sometimes returning nothing.
• NSPositionalSpecifier construction and evaluation now checks for nonsense, and returns errors instead of letting the the command implementations that depend on it fail or malfunction in inexplicable ways. For example, telling TextEdit to 'make new window at before words of front document' no longer creates a new window and attempts to insert it in the words of the front document (!). Now it results in "TextEdit got an error: Can’t make or move that element into that container."
• NSSpecifierTest has improved type checking. For example telling TextEdit to 'get every word of the front document where it is the window of the front document' used to return nothing. Now it results in "TextEdit got an error: Can’t make window of document 1 into type word."
• NSWhoseSpecifier evaluation now returns errors for a number of constructs that are invalid, instead of returning gibberish. For example, when the front TextEdit document contains the text "The quick brown fox jumped over the lazy dog," telling TextEdit to 'get the fourth word of the front document whose third character is "e"' used to return "the," even though there was actually no fourth word whose third character is "e" at all. Now it results in "TextEdit got an error: Can’t get word 4 of document 1 whose character 3 = "e". Invalid index."
• Because the object specifier machinery is now much better at recording when errors have happened, command execution machinery now behaves more predictably. For example, telling TextEdit to 'move window "There's no window with this name!" to beginning of every window' used to do nothing. Now it results in "TextEdit got an error: Can’t get window "There's no window with this name!"."

Bug Fixes in .sdef-Declared Scriptability

Support for parsing of .sdef files was added to Cocoa's Scripting support in Mac OS 10.4. Some substantial bugs were fixed in Mac OS 10.4.3:
• Handling of whose clauses having unmatched but compatible object specifiers now works. For example, sending "the first paragraph of theDocument whose last word is equal to paragraph 3 of theDocument" to a version of TextEdit whose scriptability is declared with an .sdef file no longer results in "TextEdit got an error: Can't make paragraph 3 of document 1 into type word."
• Make commands with a "with data" parameter whose value is an object specifier now work. For example, sending "make new word at end of document 2 with data first word of document 1" to a version of TextEdit whose scriptability is declared with an .sdef file no longer results in "TextEdit got an error: Can't make word 1 of document 1 into type word."
• Count commands now return the correct results, instead of lists containing the correct results.

In Mac OS 10.4 the machinery that converts Objective-C objects to Apple event descriptors didn't take into account nesting of objects in arrays properly in application whose scriptability is declared in an .sdef file, so telling Sketch 2 to get "every graphic of every document" returned an error. This bug has been fixed in Mac OS 10.5.

In Mac OS 10.4 the machinery that converts Apple event descriptors to Objective-C objects didn't always take into account the possibility of evaluating an object specifier immediately to get a value. For example, telling a Sketch document to 'set fill color of graphic 1 to fill color of graphic 2' always failed, regardless of how many graphics there were in the document. For another example, this simple script:

```
tell application "Sketch"
    set theDocument to make new document
    set theWindow to the first window whose name is the name of theDocument
end tell
```

Returned "Sketch got an error: Can't make name of document "Untitled" into type string, even though the type of a document's name is of course a string. This bug has been fixed in Mac OS 10.5.

Bug Fix in -.scriptSuite/.scriptTerminology-Declared Scriptability

In all previous versions of Mac OS X there was a bug in which the determination of whether or not a command had a result that should be put in the reply Apple event was made using the return type of the script command handling method of the first receiver, instead of the .scriptSuite-declared result type of the command. This would cause, for example, the result of telling TextEdit to "close every window" to be a list of missing values, one per window. This bug has been fixed in Mac OS 10.5. Telling TextEdit to close every window now puts no result in the reply Apple event. For backward binary compatibility the old behavior remains in applications linked against Mac OS 10.4 or earlier (because the bug could mask missing result type declarations in .scriptSuite files).

Bug Fixes in General Scriptability (Updated since WWDC 2007)

In all previous versions of Mac OS X, the count command didn't work properly when the receivers were deeply nested in other objects. For example, while Sketch has always returned the correct number when asked to return the "count of every text area of every document," it has always returned the exact same number when asked to return the "count of words of every text area of every document," which was incorrect. This bug has been fixed in Mac OS 10.5.

The index specifiers resulting from the use of the AppleScript repeat with…in… construct did not handle deep nesting either. For example, telling Sketch to do something with theGraphic in a "repeat with theGraphic in every graphic of every document" clause would throw an exception, and return a very unfriendly error to the scripter. This bug has been fixed in Mac OS 10.5.

In all previous versions of Mac OS X, whose specifier evaluation did not take into account the fact that there's nothing really wrong with a test that uses a property that is not present on every tested object. For example, telling a Sketch document to get 'every graphic whose text contents is "Some text."' returned an error if any of the graphics didn't have text contents, like circles. In Mac OS 10.5 this sort of whose specifier now returns the matching objects instead of an error.

Likewise, whose specifier evaluation did not take into account the fact that there's nothing really wrong with a test that uses an out-of-range index specifier.

For example, telling a TextEdit document to get 'every paragraph where the tenth word of it is "foo"' returned an error if any of the paragraphs did not have ten words. In Mac OS 10.5 this sort of whose specifier now returns the matching objects instead of an error.

Since -[NSObject(NSScripting) scriptingProperties]'s introduction in Mac OS 10.2 it has gratuitously caught and silently swallowed exceptions thrown by its invocations of -[NSObject(NSKeyValueCoding) valueForKey:]. This made it easy to overlook missing KVC-compliance in scriptable classes; getting the "properties" of a scriptable object with no error and no incorrect entries in the returned record didn't mean that everything was really working properly (and completely missing entries are easy to overlook!). -[NSObject(NSScripting) setScriptingProperties:] suffered from a similar problem. In Mac OS 10.5 these bugs are fixed. For backward binary compatibility the old behavior remains in applications linked against Mac OS 10.4 or earlier.

In recent versions of Mac OS X there was a bug that prevented NSPropertySpecifiers that specify all objects in a to-many relationship, NSMiddleSpecifiers, and NSRandomSpecifiers from being properly converted into Apple event descriptors. As a result AppleScript could not handle the result of returning one of these kinds of object specifiers from implementations of the -objectSpecifier method. This bug has been fixed in Mac OS 10.5.

Changed Behavior of -[NSSObject(NSScripting) setScriptingProperties:] (New since WWDC 2007)

Since -[NSSObject(NSScripting) setScriptingProperties:] was introduced in Mac OS 10.2, its default implementation has invoked [self coerceValue:theValue forKey:theKey] for each entry in the passed-in properties dictionary before setting the value of the individual property in the receiver. Cocoa Scripting's implementations of standard commands (Duplicate, Make, and Set, in particular) did not coerce the individual property values before invoking it. Considering that -setScriptingProperties: is just the setter of the "scriptingProperties" property, this behavior was inconsistent with how setting the value of properties is generally done by Cocoa; in every other case the setter is passed the result of applying coercion. Starting in Mac OS 10.5, this inconsistency has been fixed. -setScriptingProperties: is now passed a dictionary that contains values that are already coerced, and the default implementation of -setScriptingProperties: does no further coercion. For backward binary compatibility the old behavior remains in applications linked against Mac OS 10.4 or earlier.

Support for the Hidden Attribute in .sdef-Declared Scriptability

Support for parsing of .sdef files was added to Cocoa Scripting in Mac OS 10.4, but Cocoa's .sdef parser ignored all uses of the hidden attribute ('man 5 sdef' to see what elements could be hidden). This did not affect the appearance of an application's scriptability in 10.4's Script Editor, because Script Editor's dictionary viewer does its own .sdef parsing. It did however affect scripting applications that parsed a Cocoa application's reply to the Get Apple Event Terminology event (kASAppleScriptSuite/kGetAETE). In Mac OS 10.5, Cocoa Scripting now generates 'aete' data that takes hiding into account, by putting declarations of hidden scripting terminology elements into the implicitly hidden "Type Names" suite, where they can be found by the AppleScript interpreter but aren't to be presented to users by scripting dictionary viewers.

Make Commands with No "At" Parameter and Support for the Insert-At-Beginning Attribute in .sdef-Declared Scriptability (New since WWDC 2007)

The .sdef file format has no equivalent of the old .scriptSuite format's LocationRequiredToCreate entry, which means that the Make command's "at" parameter is always optional in applications with .sdef-declared scriptability. When a script sends a Make command with no "at" parameter to an application, NSCreateCommand's default implementation sends the container of the newly-created object an -[NSObject(NSScriptKeyValueCoding) insertValue:inPropertyWithKey:] message. In Mac OS 10.4 this would result in an exception if the class of the container did not implement an -insertIn<Key>: method. As a result every class of scriptable container had to implement an -insertIn<Key>: method for every to-many relationship (or, less commonly, override insertValue:inPropertyWithKey:) to be correct. This was very easy to overlook. In Mac OS 10.5, in applications with .sdef-declared scriptability, the default implementation of -[NSObject(NSScriptKeyValueCoding) insertValue:inPropertyWithKey:] now invokes [self insertValue:theNewObject atIndex:anIndex inPropertyWithKey:theKey] when no -insertIn<Key>: method is implemented. The value of the index is controlled by a new "insert-at-beginning" attribute in the <cocoa> subelement of the declaring <element> .sdef element. The default value of the attribute is "no," indicating that the insertion index to use is the same as the current count of related objects. If the value is "yes" the insertion index will be 0. So, by default new objects are added to the ends of lists of elements. In cases where that is not appropriate you can use "insert-at-beginning" to declare that by default new objects should be inserted at the beginning of lists of elements. See for example the declaration of the "graphic" element of Sketch's "document" class, at /Developer/Examples/AppKit/Sketch/Sketch.sdef.

Updated Support for the Responds-To Element in .sdef-Declared Scriptability

The .sdef file format has been changed in Mac OS 10.5. "responds-to" elements now have a "command" attribute instead of a "name" attribute. For backward binary compatibility Cocoa's .sdef parser recognizes either name for the attribute.

Support for Dynamic .sdef-Declared Scriptability

Cocoa did not handle Info.plist OSAScriptingDefinition entries whose values were "dynamic" in Mac OS 10.4. In Mac OS 10.5, Cocoa now uses the OSACopyScriptingDefinition() function to get the application's own .sdef data. For "dynamic" OSAScriptingDefinition Info.plist entries, this function sends an 'ascr'/'gsdf' Apple event, so you can return .sdef scripting declarations that are computed at run-time, to take plug-in scriptability into account, for instance. You have to register a handler for the Apple event. For example, you can do this in your application delegate's -applicationWillFinishLaunching: method:

```
// Register to provide .sdef data when asked by apps like Script Editor, or this app's own scripting machinery.
[[NSAppleEventManager sharedAppleEventManager] setEventHandler:self
                                                    andSelector:@selector(handleGetSDEFEvent:withReplyEvent:)
                                                  forEventClass:'ascr'
                                                     andEventID:'gsdf'];
```

And then implement the matching handler method:

```
- (void)handleGetSDEFEvent:(NSAppleEventDescriptor *)event withReplyEvent:(NSAppleEventDescriptor *)replyEvent {

    // Our dynamic sdef isn't really that dynamic, but you can see that you have a great deal of flexibility here.
    NSData *sdefData = [NSData dataWithContentsOfFile:[[NSBundle mainBundle] pathForResource:@"Sketch" ofType:@"sdef"]];
    [replyEvent setDescriptor:[NSAppleEventDescriptor descriptorWithDescriptorType:typeUTF8Text data:sdefData]
                   forKeyword:keyDirectObject];

}
```

Support for Included .sdef-Declared Scriptability (New since WWDC 2007)

In Mac OS 10.5 there is a new file at /System/Library/ScriptingDefinitions/CocoaStandard.sdef that you can import from your application's .sdef file so it doesn't have to redeclare the standard classes and commands. You import it using Xinclude, like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dictionary SYSTEM "file://localhost/System/Library/DTDs/sdef.dtd">

<dictionary title="Sketch Terminology" xmlns:xi="http://www.w3.org/2001/XInclude">

    <xi:include href="file:///System/Library/ScriptingDefinitions/CocoaStandard.sdef"
                xpointer="xpointer(/dictionary/suite)"/>
```

```
    […Stuff that's not in the Standard Suite, including stuff in the Text Suite, if that's applicable…]

<dictionary>
```

XPointer provides a great deal of flexibility in selecting which elements from CocoaStandard.sdef to actually include in your application's scripting declaration. See the documentation for that standard.

The declaration of the Save command in CocoaStandard.sdef requires that the "saveable file format" type be declared somewhere in your .sdef. See the 'Support for the "As" Parameter of the Save Command' section below. See also the next section for information about reusing CocoaStandard.sdef's declaration of the document class while still specifying a specific subclass of NSDocument, which is necessary.

Support for Class Extension Elements in .sdef-Declared Scriptability (Updated since WWDC 2007)

In Mac OS 10.5, Cocoa's .sdef parser understands a new <class-extension> element, which allows the declaration of additional features of an existing class. Its contents and placement are identical to those of the "class" element, but the only valid attribute is "extends," which is the name of the class being extended. A <class-extension> element may and typically will occur in a different suite than the original class.

A <class-extension> element can have a <cocoa> subelement that has a "class" attribute. For example:

```
<class-extension extends="document">
    <cocoa class="SKTDrawDocument"/>
    …
    </class-extension>
```

The value of the "class" attribute must be the name of an Objective-C class that is a subclass of the one declared for the scripting class being extended. You can use this in apps whose .sdef imports CocoaStandard.sdef to declare the subclass of NSDocument to be be instantiated when a script tells your app "make new document." (CocoaStandard.sdef's declaration of the document class can't possibly name an Objective-C class more specific than NSDocument.) The result of multiple <class-extension> elements changing the Objective-C class of the same scripting class is undefined.

Support for Synonym Elements in .sdef-Declared Scriptability (New since WWDC 2007)

Support for parsing of .sdef files was added to Cocoa Scripting in Mac OS 10.4, but Cocoa's .sdef parser ignored all uses of the <synonym> element. In Mac OS 10.5 Cocoa parses <synonym> elements and uses the Apple event codes declared in them when converting incoming Apple events to NSScriptCommands.

Support for Custom Value Types in .sdef-Declared Scriptability

In Mac OS 10.4, you could put <value-type> elements in your application's .sdef and Cocoa would use them, but this support for custom value types was lacking in two ways:

- Corresponding hidden classes weren't put in 'aete' generated by Cocoa Scripting. This meant that AppleScript wouldn't recognize the name of the type unless it was one declared by AppleScript itself. In Mac OS 10.5 Cocoa now generates the right 'aete' data for <value-type> declarations. (In Mac OS 10.5, AppleScript asks apps with .sdef-declared scriptability for their sdef data instead of their 'aete' data so this fix is less important than it was, but it still might affect some scripting tools.)
- There was no documentation about what code had to be written to make custom value types work (though some people deduced it based on the exceptions thrown when required methods weren't implemented). Here's a little documentation.

When you put a <value-type> element in your application's .sdef, you also have to provide methods that Cocoa can use to convert Apple event descriptors of that type to Objective-C objects and vice versa. There must be a class method whose name matches the pattern +scripting<CondensedTypeName>WithDescriptor:, where CondensedTypeName is the type name with the first letter of each word capitalized and the spaces removed. Cocoa sends +scripting<CondensedTypeName>WithDescriptor: messages to the class named in the <cocoa> subelement of the <value-type> element in question when it needs to convert an Apple event descriptor into an Objective-C object. The method should return nil for failure, and take advantage of regular Apple event coercion where applicable. There must also be an instance method whose name matches the pattern -scripting<CondensedTypeName>Descriptor. Cocoa sends -scripting<CondensedTypeName>Descriptor messages to objects that it must convert into Apple event descriptors. This method should also return nil for failure. Neither kind of method needs to record error information in the current script command when it returns nil.

See NSColor_SKTScripting.m in /Developer/Examples/AppKit/Sketch for an example. It's Sketch's implementation of the "RGB color" value type.

Support for the Missing Value Type in .sdef-Declared Scriptability

In Mac OS 10.5 you can now use "missing value" as a type name. It corresponds to the Objective-C NSNull class. You will typically use it as an alternative in a complex type. For example, the declaration of the fill color property of the graphic class in Sketch.sdef now looks like this:

```
<property name="stroke thickness" code="slwd">
    <type type="real"/>
    <type type="missing value"/>
    <cocoa key="scriptingStrokeWidth"/>
</property>
```

That means at least four things:

- An AppleScript can set the stroke thickness of a graphic to either a number or the missing value.
- The method that provides KVC compliance for setting the value for the "scriptingStrokeWidth" key must be prepared to handle either a floating point number or nil. See Sketch's -[SKTGraphic setScriptingStrokeWidth:] to see how it does this.
- The getter method for the "scriptingStrokeWidth" key is allowed to return either a floating point number or nil. See Sketch's -[SKTGraphic scriptingStrokeWidth].
- An AppleScript shouldn't be surprised if it asks for the stroke thickness of a graphic and the result is the missing value (which is distinct by the way from no value).

Most application code isn't supposed to have to deal with NSNull, so whenever Cocoa Scripting invokes one of your application's KVC setter methods, it always converts NSNull to nil first. For backward binary compatibility however it doesn't do this in applications linked against Mac OS 10.4 or earlier (because there was limited, unpublished support for NSNull values even before Mac OS 10.5, and there are likely to be shipping applications with setters that expect to be passed NSNull and would malfunction when passed nil). Likewise, your application's KVC getter methods can return nil instead of NSNull.

Also, Cocoa Scripting now does a good job of returning missing values in result lists. For example, telling Sketch to get the "text contents of every graphic of the front document," when the front document contains a text area and a circle (circles don't have text contents), now returns something like {"the contents of the text area", missing value} instead of an error.

How Support for Complex Types in .sdef-Declared Scriptability Works, and a Bug Fix (New since WWDC 2007)

When you declare that a class property or a command parameter is of complex type, like this example from iChat's .sdef file:

```
        <enumeration name="InviteType" code="invt">
            <enumerator name="audio invitation" code="acon">
```

```
            <cocoa name="AudioInvitation"/>
        </enumerator>
        <enumerator name="text chat invitation" code="tcon">
            <cocoa name="TextInvitation"/>
        </enumerator>
        <enumerator name="video invitation" code="vcon">
            <cocoa name="VideoInvitation"/>
        </enumerator>
    </enumeration>
    <command name="send" code="ichtsend" description="Sends a message or file to a buddy or to a chat.">
        <cocoa class="SendCommand"/>
        <direct-parameter>
            <type type="text"/>
            <type type="file"/>
            <type type="InviteType" hidden="yes"/>
        </direct-parameter>
        […]
    </command>
```

The machinery in Cocoa Scripting that converts incoming Apple event descriptors to Objective-C objects tries to do so using information about each type, in turn, until a successful conversion has been done, in which case it stops trying and ignores the rest of the types. In this example, Cocoa will first try to convert an Apple event descriptor to an NSString (the Objective-C class that corresponds to the "text" type), then an NSURL (corresponding to "file"), and then an NSNumber (the default class corresponding to enumerators).

That was true in Mac OS 10.4, and is still generally true in Mac OS 10.5, with one new refinement: to accommodate the fact that Apple event coercions built into the Mac OS X frameworks sometimes make it possible to convert to a type that is not necessarily the best one out of the alternatives, Cocoa may now try to still convert to other types in the list, and choose the resulting value of one of those other types. In Mac OS 10.5 it does this by favoring any other value class over NSString or NSURL, because those are the two classes to which conversion is often successful only because of inadvertent coercion. When an Apple event descriptor can be converted to either an NSString or an NSURL according to the declared complex type, it favors NSURL if the original Apple event descriptor's type is one that explicitly indicates files (typeFileURL, typeAlias, etc.), NSString otherwise. Given these rules, in our example telling iChat to 'send video invitation to aBuddy' now results in a send command whose direct parameter is an NSNumber wrapping the four character code 'vcon', which is correct. On Mac OS 10.4 it would have resulted in a send command whose direct parameter is an NSString containing "vcon," which is incorrect. Similarly, telling iChat to 'send POSIX file "/Applications/Chess.app"' now results in a send command whose direct parameter is an NSURL containing "file://localhost/Applications/Chess.app," which is correct. On Mac 10.4 it would have resulted in a send command whose direct parameter is an NSString containing "YourRootVolumeName:Applications:Chess.app:," which is incorrect.

Support for Nondefault Enumerator Values in .sdef-Declared Scriptability

In Mac OS 10.5 you can now add an attribute to the "cocoa" subelement of .sdef-declared enumerator declarations to specify what value your code is passed when a script uses that enumerator. For example:

```
<enumeration name="printing error handling" code="enum">
    <enumerator name="standard" code="lwst" description="Standard PostScript error handling">
        <cocoa boolean-value="NO"/>
    </enumerator>
    <enumerator name="detailed" code="lwdt" description="print a detailed report of PostScript errors">
        <cocoa boolean-value="YES"/>
    </enumerator>
</enumeration>
```

The "boolean-value" attributes, in combination with this property in the standard print settings record declaration:

```
<property name="error handling" code="lweh" type="printing error handling" description="how errors are handled">
    <cocoa key="NSDetailedErrorReporting"/>
</property>
```

Results in an entry whose key is "NSDetailedErrorReporting" and whose value is a boolean NSNumber with a value of NO or YES being put in the NSDictionary to which a print settings record Apple event descriptor is converted. (And that's convenient because such a dictionary can be used as the attributes dictionary of an NSPrintInfo without further processing, while the scripter just works with the kind of print settings record described in Technical Note 2082, "The Enhanced Print Apple Event.")

In addition to "boolean-value," "string-value" and "integer-value" attributes are supported. "string-value" not surprisingly corresponds to NSString, and the other two correspond to NSNumber. You can only use one per "cocoa" subelement. If you don't use any, the default behavior is the same as Mac OS 10.4's, which is to make the programmatic value associated with the enumerator an NSNumber containing the four character code of the enumerator.

This works with any kind of enumerator. For example, you can use "integer-value" attributes in an enumeration type that's used as the parameter type of a command, and deal with simple small integers in your code instead of four character codes. (Just make sure to keep the .sdef and the code consistent!)

Support for the "As" Parameter of the Save Command

In all previous versions of Mac OS X -[NSDocument handleSaveScriptCommand:] ignored the "as" parameter, even though Foundation's own declaration of the save command includes such a parameter. In Mac OS 10.5 -[NSDocument handleSaveScriptCommand:] now uses the "as" parameter if its value is an NSString, interpreting it as a type name of the sort used by NSDocument. In .sdef-declared scriptability you can take advantage of the enumerator value mechanism described in the previous section to allow for writing scripts that don't have to mention NSDocument type names, which can be too technical to make users deal with. For example, Sketch 2 (which uses UTIs as its document type names; see the AppKit release notes for information about doing that) declares a "saveable file format" enumeration and uses it as the type of the "as" parameter:

```
<enumeration name="saveable file format" code="savf">
    <enumerator name="Sketch" code="sktc" description="The native Sketch 2 file format">
        <cocoa string-value="com.apple.sketch2"/>
    </enumerator>
    <enumerator name="PDF" code="PDF " description="Portable Document Format">
        <cocoa string-value="com.adobe.pdf"/>
    </enumerator>
    <enumerator name="TIFF" code="TIFF" description="Tagged Image File Format">
        <cocoa string-value="public.tiff"/>
    </enumerator>
</enumeration>

<command name="save" code="coresave" description="Save a document.">
    <direct-parameter type="specifier" description="The document(s) or window(s) to save."/>
    <parameter name="in" code="kfil" type="file" optional="yes" description="The file in which to save the document.">
        <cocoa key="File"/>
    </parameter>
    <parameter name="as" code="fltp" type="saveable file format" optional="yes" description="The file format to use.">
        <cocoa key="FileType"/>
    </parameter>
```

```
</command>
```

This allows people to write this sort of thing in their scripts:

```
save theDocument in theFile as TIFF
```

Better Behavior of the Save Command in Unsaved Documents

In Mac OS 10.4 it was not possible to export a document to a non-native file format if the document had never been saved. (An application of a too-literal reading of the Scripting Interface Guidelines' "save in with an unsaved file acts like Save As.") In Mac OS 10.5, if the file format specified by a save command, either implicitly with an "as" parameter or implicitly with a file name extension, is not a native file format for the document, but it is one to which the document can be exported, the save command acts like Save a Copy (also known as NSSaveToOperation, also known as Export As, in the Human Interface Guidelines nowadays).

Support in NSCloneCommand for Making the "To" Parameter of the Duplicate Command Optional

NSCloneCommand's default implementation now duplicates the receivers of the command in place when a script does not specify a "to" parameter, inserting each duplicate after the original in the element sequence. For example, you can now tell a Sketch document to 'duplicate every graphic'. For apps with .scriptSuite/.scriptTerminology-declared scriptability, Cocoa's own declaration of the duplicate command has been updated to take advantage. For apps with .sdef-declared scriptability, this feature will be disabled until you've updated the application's .sdef to declare the "to" parameter optional.

Ending of Undo Manager Groups During Handling of Scripting Commands

In previous versions of Mac OS X the mechanism that automatically ends undo manager groups during event handling was not triggered during handling of Apple events, including those containing scripting commands. This meant that scripted changes were sometimes put in the same undo manager group as whatever change the user happened to make after switching back to the application. It also meant that things like documents' modification status weren't always updated in a timely fashion. Starting in Mac OS 10.5, any open undo manager groups are ended after the dispatch of each Apple event.

New Methods You Can Override to Customize Creation and Copying of Scripting Objects (New since WWDC 2007)

Historically Cocoa Scripting has created new scripting objects by sending +alloc to a class and -init to the resulting object. It's copied scripting objects by sending -copyWithZone:. People have discovered many situations where this is not sufficient. So that you can take more control over what happens when your application is sent a Make or Duplicate command, new methods have been added to NSObject(NSScripting) for you to override. These methods are invoked on the prospective container of the new or copied object. The returned objects or objects are then inserted into the container using key-value coding:

```
- (id)newScriptingObjectOfClass:(Class)class
                forValueForKey:(NSString *)key
             withContentsValue:(id)contentsValue
                    properties:(NSDictionary *)properties;
```

Create a new instance of a scriptable class to be inserted into the relationship identified by the key, set the contentsValue and properties of it, and return it. The contentsValue and properties are derived from the "with contents" and "with properties" parameters of a Make command. The contentsValue may be nil.

```
- (id)copyScriptingValue:(id)value forKey:(NSString *)key withProperties:(NSDictionary *)properties;
```

Create one or more scripting objects to be inserted into the relationship identified by the key by copying the passed-in value, set the properties in the copied object or objects, and return it or them. The value is, for example, derived from the receivers of a Duplicate command. Its type must match the type of the property identified by the key. For example, if the property is a to-many relationship, the value will always be an array of objects to be copied, and an array of objects must therefore be returned. The properties are derived from the "with properties" parameter of a Duplicate command.

New Method You Can Override to Customize Object Specifier Evaluation Without Making Up Object Indexes (New since WWDC 2007)

There have been several requests for an alternative to implementing -[NSObject(NSScriptObjectSpecifiers) indicesOfObjectsByEvaluatingObjectSpecifier:] to customize the evaluation of object specifiers, one that doesn't require the scripting container to make up indexes for contained objects that don't naturally have indexes. A new method has been added to NSObject(NSScripting) for you to override:

```
- (id)scriptingValueForSpecifier:(NSScriptObjectSpecifier *)objectSpecifier;
```

Given an object specifier return the specified object or objects in the receiving container. This might successfully return an object, an array of objects, or nil, depending on the kind of object specifier. Because nil is a valid value, failure is signaled by sending the object specifier -setEvaluationError: before returning. Your override doesn't also have to invoke any of the NSScriptCommand error signaling methods, though it can, to record very specific error information. The NSUnknownKeySpecifierError and NSInvalidIndexSpecifierError numbers are special, in that Cocoa may continue evaluating an outer specifier if they're encountered, for the convenience of scripters.

New NSScriptObjectSpecifier Methods for Accessing the Underlying Apple Event Descriptor (New since WWDC 2007)

Several people have requested a way to create an NSScriptObject specifier from an Apple event descriptor and a way to get the Apple event descriptor out of an NSScriptObjectSpecifier. New methods have been added to NSScriptObjectSpecifier:

```
+ (NSScriptObjectSpecifier *)objectSpecifierWithDescriptor:(NSAppleEventDescriptor *)descriptor;
```

Given a typeObjectSpecifier Apple event descriptor, create and return an object specifier, or nil for failure. If this is invoked and fails during the execution of a script command, information about the error that caused the failure is recorded in [NSScriptCommand currentCommand].

```
- (NSAppleEventDescriptor *)descriptor;
```

Return an Apple event descriptor that represents the receiver. If the receiver was created with +objectSpecifierWithDescriptor: the passed-in descriptor is returned. Otherwise a new one is created and returned (autoreleased, of course).

New NSScriptClassDescription Methods for Accessing Information from the Class Declaration (New since WWDC 2007)

Because the "name" of an NSScriptClassDescription resulting from an .sdef class declaration is its human-readable name, the -name method cannot be used to find out the Objective-C class that must be instantiated to make objects of that scriptable class. To let you get at that information, an existing method in NSScriptClassDescription has been published:

```
- (NSString *)implementationClassName;
```

Return the name of the Objective-C that implements the described scriptable class. This method also works on Mac OS 10.4.

NSScriptClassDescription hasn't had enough accessor methods to allow for some of the customizations that people want to do. Also, overriders of -[NSObject(NSScripting) scriptingValueForSpecifier:] may have to do some of the same error checking that the default implementation of that method does.

New methods have been added to NSScriptClassDescription:

```
- (BOOL)hasPropertyForKey:(NSString *)key;
- (BOOL)hasOrderedToManyRelationshipForKey:(NSString *)key;
- (BOOL)hasReadablePropertyForKey:(NSString *)key;
- (BOOL)hasWritablePropertyForKey:(NSString *)key;
```

Return whether the described class has a property identified by the key, whether it's a to-many relationship, whether it's readable, or whether it's writable, respectively. For example, -[NSObject(NSScripting) scriptingValueForSpecifier:] uses -hasPropertyForKey: to make sure that the scripted object actually has the specified property. (Because Cocoa Scripting lets you do things like telling Sketch to get the "text contents of graphic 1 of document 1," and something has to make sure that graphic 1 is not actually a circle, which doesn't have text contents.) It then uses -isPropertyReadableForKey: to make sure that the property has not been declared to be write-only.

Because the existing -[NSScriptClassDescription isReadOnlyKey:] does not fit the pattern established by these new methods, and because it is a little dangerous (a result of NO could mean writing to that property is permitted, or it could just mean that the key is just unrecognized), it is deprecated in Mac OS 10.5.

New NSScriptCommand Methods for Error Reporting (New since WWDC 2007)

Cocoa Scripting's error reporting is much improved in Mac OS 10.5. One of the big improvements is that it now populates the reply Apple event with the standard kOSAErrorOffendingObject and kOSAErrorExpectedType parameters when those are applicable. So that your code can do the same, new methods have been added to NSScriptCommand:

```
- (void)setScriptErrorOffendingObjectDescriptor:(NSAppleEventDescriptor *)errorOffendingObjectDescriptor;
- (void)setScriptErrorExpectedTypeDescriptor:(NSAppleEventDescriptor *)errorExpectedTypeDescriptor;
- (NSAppleEventDescriptor *)scriptErrorOffendingObjectDescriptor;
- (NSAppleEventDescriptor *)scriptErrorExpectedTypeDescriptor;
```

Set or get the offending object or expected type descriptor that should be put in the reply to the Apple event from which this command was constructed, when execution of the command is completed, if the sender of the event requested a reply.

New NSPositionalSpecifier Accessor Methods (New since WWDC 2007)

So that you can get the values with which an NSPositionalSpecifier (a "location specifier," in AppleScript-speak, like the "at" parameter of a Make command) was initialized, new accessor methods have been added:

```
- (NSInsertionPosition)position;
- (NSScriptObjectSpecifier *)objectSpecifier;
```

Return the position or object specifier that was specified at initialization time.

Results Now Returned by the Open Command (New since WWDC 2007)

In previous versions of Mac OS X AppKit's default handling of the Open command never returned a result. Starting in Mac OS 10.5, it returns an NSDocument or an NSArray of NSDocuments, depending on whether the direct parameter of the command is a file or a list of files. For applications with .scriptSuite/.scriptTerminology-declared scriptability, Foundation's declaration of the standard suite (in its NSCoreSuite.scriptSuite resource file) has been updated to match. For applications with .sdef-declared scriptability, the CocoaStandard.sdef file mentioned elsewhere in these notes also includes a declaration of the Open command with appropriate result types.

NSURL methods

NSURL has new creation methods:

```
- initFileURLWithPath:(NSString *)path isDirectory:(BOOL)isDir;
+ (id)fileURLWithPath:(NSString *)path isDirectory:(BOOL)isDir;
```

These methods enable NSURL to avoid an extra I/O to check whether the path is a directory or not.

Note that NSURL.h lists these two methods as being available back to 10.4. This is incorrect; these methods were added in 10.5 and don't exist back on 10.4.

Deprecated NSURLHandle

NSURLHandle has been deprecated, as have all APIs that reference them. NSURLConnection (introduced in 10.3) should be used in its place.

NSJavaSetup.h deprecated

All API in NSJavaSetup.h has been deprecated in 10.5. The header will likely not be available in the next major OS release after 10.5.

NSInvocation.h API deprecations

The enum _NSObjCValueType type and the NSObjCValue type have been deprecated. These things will likely not be available in this header in the next major OS release after 10.5.

+poseAsClass: deprecated

The +poseAsClass: method in NSObject has been deprecated. Posing has been deprecated in the Objective C runtime, and this change is a reflection of that.

NSCoder.h API deprecated

The function NXReadNSObjectFromCoder() and the methods encodeNXObject: and decodeNXObject, which appear in NSCoder.h, have been deprecated in 10.5. If you've been putting up with the logged messages since 10.0, it's time for you to move on.

Deprecated NSRunLoop API

The -configureAsServer method is deprecated in 10.5. It never did anything, so there was never a point in calling it in Mac OS X.

Deprecated defaults constants in NSUserDefaults.h

As mentioned in the 10.4 release notes for Foundation, the following user defaults are deprecated in 10.5:

```
NSString * const NSWeekDayNameArray;
NSString * const NSShortWeekDayNameArray;
NSString * const NSMonthNameArray;
NSString * const NSShortMonthNameArray;
NSString * const NSTimeFormatString;
NSString * const NSDateFormatString;
NSString * const NSTimeDateFormatString;
NSString * const NSShortTimeDateFormatString;
NSString * const NSCurrencySymbol;
NSString * const NSDecimalSeparator;
NSString * const NSThousandsSeparator;
NSString * const NSDecimalDigits;
NSString * const NSAMPMDesignation;
NSString * const NSHourNameDesignations;
NSString * const NSYearMonthWeekDesignations;
NSString * const NSEarlierTimeDesignations;
NSString * const NSLaterTimeDesignations;
NSString * const NSThisDayDesignations;
NSString * const NSNextDayDesignations;
NSString * const NSNextNextDayDesignations;
NSString * const NSPriorDayDesignations;
NSString * const NSDateTimeOrdering;
NSString * const NSInternationalCurrencyString;
NSString * const NSShortDateFormatString;
NSString * const NSPositiveCurrencyFormatString;
NSString * const NSNegativeCurrencyFormatString;
```

Developers should use NSLocale, NSDateFormatter and NSNumberFormatter APIs instead.

Where possible, NSUserDefaults will supply compatible values for these keys, derived from the above sources. For the following keys, localized values are not available from the above sources and values will be provided in English for all localizations:

```
NSDecimalDigits, NSHourNameDesignations, NSYearMonthWeekDesignations, NSEarlierTimeDesignations, NSLaterTimeDesignations,
NSThisDayDesignations, NSNextDayDesignations, NSNextNextDayDesignations, NSPriorDayDesignations
```

Removed headers

The following previously deprecated headers have been removed from Foundation in 10.5:

```
NSCompatibility.h
NSUtilities.h
NSSerialization.h
```

Foundation profile binary

Foundation no longer provides a version of the binary built for profiling. Of course, Foundation used to be one of the few libraries that actually did provide one. Use dtrace instead.

Notes specific to Mac OS X 10.4

NSMetadata

NSMetadata.h contains several classes which expose the Spotlight API to Cocoa developers. The principle class is NSMetadataQuery. The NSMetadata classes are bindings-compliant. Cocoa keys for the well-defined Spotlight attributes of MDItemRef are not available; use the keys of the Spotlight layer (such as kMDItemContentType) directly.

There is a new example, /Developer/Examples/AppKit/Spotlighter, which demonstrates the use of the NSMetadataQuery.

NSXMLNode, NSXMLDocument, NSXMLElement, NSXMLDTD, NSXMLDTDNode (Section updated since WWDC)

These are new classes to represent XML objects. There is reference documentation at Cocoa->Internet and Web->Tree-based XML Processing.

To provide a fast implementation NSXML does not check the validity of data when using setStringValue or setObjectValue. This means that, for example, the content of a comment could be set to "foo--bar" and NSXML will output an invalid comment: <!--foo--bar-->. To provide a robust implementation when handling unknown input, clients should check for "--" anywhere and "-" at the end of the string; processing instructions should be checked for "?>". NSXML breaks CDATA sections up on output if they contain "]]>". For example, if the content of a cdata node is set to "foo]]>bar", NSXML will output "<!CDATA[foo]]]><!CDATA[>bar]]>". NSXML does not normalize data (change linebreaks, merge text nodes) when setting a node's value. Invalid XML

characters such as control characters will be output if a node's value includes them.

The XQuery/XPath implementation conforms to the October 2004 draft specification. It supports the optional features "Full Axis" and "Module Feature". Regular expressions use pcre and do not support Unicode character classes such as \p{Lu} or block escapes.

NSLocale (Section updated since WWDC)

The new NSLocale class is a cover over the CFLocale API in CoreFoundation. NSLocales and CFLocales are toll-free bridged.

NSLocales cannot be passed into Cocoa APIs which take a locale: parameter -- these are expecting a dictionary for that parameter. It is unclear whether such APIs will be enhanced to take either NSDictionary * or NSLocale * in the future, as there are compatibility issues, such as the possible existence of subclasses which have overridden those methods and will fail if given an NSLocale *. There is no API for converting an NSLocale * to an NSDictionary *, and although both implement -objectForKey:, the two use different sets of keys.

There are some additional notes on CFLocale in the CoreFoundation Release Notes, which may be useful to users of the NSLocale API.

NSCalendar (Section added since WWDC)

The new NSCalendar class is a cover over the new CFCalendar API in CoreFoundation. NSCalendar allows you to ask for numeric properties of a calendar, convert between NSDates and decomposed unit representations, and perform some types of calendrical arithmetic. NSCalendars and CFCalendars are toll-free bridged.

NSCalendars are created using the calendar identifiers in NSLocale.h, and only those identifiers are supported. You cannot define your own calendars with NSCalendar, though you can subclass and override every instance method, and you would also have to create a custom NSDateFormatter to get date formatting using your calendar, as NSDateFormatter only understands the well-defined built-in calendars in Mac OS X 10.4 and does not necessarily invoke methods on its NSCalendar attribute to compute answers. The "current calendar" returned by +currentCalendar is the user's preferred calendar configured with the user's locale and the default time zone. The default locale for a NSCalendar is the System Locale (+[NSLocale systemLocale]).

Note: the Chinese calendar is currently unimplemented in the NSCalendar, and related, APIs. (Results are undefined.)

You can change some properties of an NSCalendar (locale, time zone, first day of the week, and the minimum number of days for a week to be the first week of the year). However, these are just a convenience for setting parameters to the calendrical calculations, and do not change the NSCalendar, say, for purposes of equality testing, and are not archived.

The values/computations/results of the NSCalendar API do not necessarily agree with the values/computations/results from the NSCalendarDate API in NSCalendarDate.h. The NSCalendarDate API is taking a secondary place to the NSCalendar and NSDateFormatter APIs as of Mac OS X 10.4, and may be deprecated in the next release.

There are many additional notes on CFCalendar in the CoreFoundation Release Notes, which will be useful to users of the NSCalendar API. The following notes describe the main difference between the CFCalendar and NSCalendar APIs.

Instead of variable-argument methods and component description strings as in the CFCalendar API, NSCalendar uses the simple NSDateComponents object to hold arbitrary sets of components for the calendrical calculation methods. NSDateComponents is basically just an object with fields that can be get and set, but can also be extended later, and results in an API which is more Cocoa-like than the CFCalendar API. Component fields which are not initialized or which should be ignored have the value NSUndefinedDateComponent. NSDateComponents can be used to represent dates/times, like, 26 March to variable degrees of specificity (e.g., the example doesn't specify which year, which could represent any), or quantities of components, like "5 months and 1 hour". It is important to note that NSDateComponents (1) does not do any calendrical calculations (it doesn't return "correct" answers for fields which have not been set), and (2) it does not check the fields being set for (a) consistency amongst the fields or (b) range validity. NSDateComponents is also not tied to any particular calendar, so in some situations, both an NSCalendar and an NSDateComponents may need to be kept together to make sense of the date later.

When an NSDateComponents is the return value, a unitFlags argument specifies which fields of the NSDateComponents are to be initialized/used. The unit constants are bitwise OR'd together to make the unitFlags argument.

The NSDateComponents approach does not allow for components to be specified in a particular order to the -dateByAddingComponents:toDate:options: and -components:fromDate:toDate:options: calendrical operations. NSCalendar always uses the components in the order that the unit constants are listed in NSCalendar.h, so to effect a particular computation order to the components, multiple calls to the computation methods may have to be made, and computation of intermediate values. Difference computations with the Week unit will be particularly troublesome in this regard.

Note that the NSDateComponents class does not implement the NSCopying or NSCoding protocols, which is an oversight in Mac OS X 10.4.

NSDateFormatter (Section updated since WWDC)

For Mac OS X 10.4, the NSDateFormatter class gets a major functionality overhaul. The new implementation is based on CFDateFormatter, which is based on the open-source ICU (International Components for Unicode) library, and should produce much better localized formatted dates. NSDateFormatter and CFDateFormatter, however, are not toll-free bridged.

There are basically two modes in which an NSDateFormatter can operate, called the "formatter behavior". The 10.0 (to 10.3) compatibility mode operates like NSDateFormatter has in previous Mac OS X releases, including the limitations and bugs. The new 10.4 behavior mode allows more configurability and better localization.

Developers are encouraged to use the NSDateFormatterStyle constants to specify how much information is displayed for the date and time parts of a formatted NSDate. The styles are NoStyle (omit this part, either date or time or both), ShortStyle, MediumStyle, LongStyle, and FullStyle (displaying the most information). These are styles that the user can configure in the International preferences panel in System Preferences. If you as a developer decide that you just don't like any of the styles, and set your own format string, then you can just just the components you want to be displayed, but you lose user-configurability.

In addition to the methods inherited from NSFormatter, NSDateFormatter adds the getObjectValue:forString:range:error: method. This method allows you to specify a subrange of the string to be parsed, and returns the range of the string which was actually parsed (which, on failure, indicates where the failure occurred). The method also returns an NSError object, which can contain richer information than the failure string returned by the inherited getObjectValue:... method. NSDateFormatter also adds two convenience methods, -stringFromDate: and -dateFromString:.

Note that NSCell, since it works with general NSFormatters, only invokes the NSFormatter getObjectValue:... method in Mac OS X 10.4. For a 10.4-style date formatter, that method calls the new getObjectValue:... method.

But, NSDateFormatters don't have to just be attached to cells. The new methods are provided to make it nicer to use an NSDateFormatter directly in code, and that is the direction in which the Foundation framework is moving. Developers wanting to format dates into strings will use date formatters.

There are several new attributes one can get and set on a 10.4-style date formatter, including the date style, time style, locale, time zone, calendar, format string, the two-digit-year cross-over date, the default date which provides unspecified components, and there is also access to the various textual strings,

like the month names. But it will generally be atypical to change most of these attributes from their default values.

The new methods in 10.4 do not do anything when invoked on a 10.0-style formatter, and return a generic return value when required to return something. The new methods should not be invoked on a 10.0-style formatter.

The biggest change in a 10.4-style formatter is the format of the format string. The "%A %B %y" style of NSDateFormatter formats in 10.3 and earlier and NSCalendarDate formats has been replaced with the format string structure of CFDateFormatter (and ICU). These format strings are similar to those found in C#/.Net and Java APIs. See the documentation for CFDateFormatter or NSDateFormatter for more information. When the formatter is in "10.0" mode, the old-style format strings are required. One thing to note with the CFDateFormatter format string format is that literal text should be enclosed inside single quotes (') in the format string -- this is easy to forget.

The object type for "10.0" date formatters is still NSCalendarDates, but for new-style formatters, it is NSDates. You can configure a new-style formatter to generate NSCalendarDates with setGeneratesCalendarDates: if you want them. You are encouraged to switch to NSDates now, if possible, for new-style formatters.

The default behavior for NSDateFormatters in Mac OS X 10.4 is the 10.0 behavior, for backwards binary compatibility. Developers are encouraged to try out 10.4-style formatters, and switch to them if possible, to get the better localization support. Obviously that has to be conditionally done if the app is shipping on releases prior to 10.4 as well.

You can call [formatter setFormatterBehavior:NSDateFormatterBehavior10_4] on each individual formatter instance you want to change. Note that the old -initWithDateFormat:allowNaturalLanguage: method always initializes one to 10.0 behavior. But you may have to update the behavior of your app as well when you do that; for example, the new-style formatter will start returning NSDates instead of NSCalendarDates by default, and if your app is expecting NSCalendarDates later, there will be trouble.

There is a new default you can also set to have date formatters converted to new-style automatically (which could cause other trouble in the app, be warned). Set the "NSMakeDateFormatters10_4" default to a boolean YES/true value in the app's preferences using the 'defaults' command. The preference has to be set before any use of the NSDateFormatter class is made. The default has two effects: (1) date formatters which are created with +alloc/-init will be 10.4-style; (2) date formatters which are unarchived from either non-keyed or keyed archives will be converted to 10.4-style if the archived formatter has uncustomized format string -- most of those found in IB's NSDateFormatter inspector -- at unarchive time.

The main down-side that may be noticed with a new 10.4-style date formatter is that the lenient parsing mode is not as forgiving as the "natural language" parsing of NSDateFormatter when "allowsNaturalLanguage" was turned on in the formatter. This has a bad and a good side. Users will have to be a bit more careful and perhaps thorough when typing in dates, but they are more likely to find that the value they were trying to input was correctly set to the value they wanted rather than what the "natural language" parsing guessed they meant.

NSNumberFormatter (Section updated since WWDC)

For Mac OS X 10.4, the NSNumberFormatter class gets a major functionality overhaul. The new implementation is based on CFNumberFormatter, which is based on the open-source ICU (International Components for Unicode) library, and should produce much better localized formatted numbers. NSNumberFormatter and CFNumberFormatter, however, are not toll-free bridged.

There are basically two modes in which an NSNumberFormatter can operate, called the "formatter behavior". The 10.0 (to 10.3) compatibility mode operates like NSNumberFormatter has in previous Mac OS X releases, including the limitations and bugs. The new 10.4 behavior mode allows more configurability and better localization.

Developers are encouraged to use the NSNumberFormatterStyle constants to specify pre-canned sets of attributes which determine how a formatted number is displayed. The styles are NoStyle (use when you're going to set the format string yourself), DecimalStyle, CurrencyStyle, PercentStyle, ScientificStyle, and SpellOutStyle (textual number representation). These are styles that the user can configure in the International preferences panel in System Preferences. If you as a developer decide that you just don't like any of the styles, and set your own format string, then you can just just the components you want to be displayed, but you lose user-configurability.

In addition to the methods inherited from NSFormatter, NSNumberFormatter adds the getObjectValue:forString:range:error: method. This method allows you to specify a subrange of the string to be parsed, and returns the range of the string which was actually parsed (which, on failure, indicates where the failure occurred). The method also returns an NSError object, which can contain richer information than the failure string returned by the inherited getObjectValue:... method. NSNumberFormatter also adds two convenience methods, -stringFromNumber: and -numberFromString:.

Note that NSCell, since it works with general NSFormatters, only invokes the NSFormatter getObjectValue:... method in Mac OS X 10.4. For a 10.4-style number formatter, that method calls the new getObjectValue:... method.

But, NSNumberFormatters don't have to just be attached to cells. The new methods are provided to make it nicer to use an NSNumberFormatter directly in code, and that is the direction in which the Foundation framework is moving. Developers wanting to format numbers into strings in ways more complex or perhaps more convenient than NSString formatting allows will use number formatters.

There are several new attributes one can get and set on a 10.4-style number formatter, including the number style, locale, negative-number and positive-number format strings, various strings for special values, text attribute sets for created attributed strings, and various other configuration attributes. But it will generally be atypical to change most of these attributes from their default values.

The new methods in 10.4 do not do anything when invoked on a 10.0-style formatter, and return a generic return value when required to return something. The new methods should not be invoked on a 10.0-style formatter. On a 10.4-style formatter, the old methods map approximately to one or more new methods/attributes, but the new methods should be used directly when possible.

There are slight changes in a 10.4-style formatter to the format of the format string. The 10.0-style of NSNumberFormatter formats in 10.3 and earlier has been replaced with the format string structure of CFNumberFormatter (and ICU). These format strings are similar to those found in C#/.Net and Java APIs. See the documentation for CFNumberFormatter or NSNumberFormatter for more information. One thing to note with the CFNumberFormatter format string format is that literal text should be enclosed inside single quotes (') in the format string -- this is easy to forget. The main difference is that the '_' (underbar) format character of the 10.0-style formatter is not accepted by a 10.4-style formatter, and that the position of the comma(s), if they occur in the 10.4 format string (and they need not) is significant and determines where the grouping separators will be placed (where it is not significant in a 10.0-style format).

The object type for "10.0" number formatters is still NSDecimalNumbers, but for new-style formatters, it is NSNumbers. You can configure a new-style formatter to generate NSDecimalNumbers with setGeneratesDecimalNumbers: if you want them. You are encouraged to switch to NSNumbers now, if possible, for new-style formatters.

The default behavior for NSNumberFormatters in Mac OS X 10.4 is the 10.0 behavior, for backwards binary compatibility. Developers are encouraged to try out 10.4-style formatters, and switch to them if possible, to get the better localization support. Obviously that has to be conditionally done if the app is shipping on releases prior to 10.4 as well.

You can call [formatter setFormatterBehavior:NSNumberFormatterBehavior10_4] on each individual formatter instance you want to change. But you may have to update the behavior of your app as well when you do that; for example, the new-style formatter will start returning NSNumbers instead of NSDecimalNumbers by default, and if your app is expecting NSDecimalNumbers later, there will be trouble.

There is a new default you can also set to have number formatters converted to new-style automatically (which could cause other trouble in the app, be warned). Set the "NSMakeNumberFormatters10_4" default to a boolean YES/true value in the app's preferences using the 'defaults' command. The preference has to be set before any use of the NSNumberFormatter class is made. The default has two effects: (1) number formatters which are created with +alloc/-init will be 10.4-style; (2) number formatters which are unarchived from either non-keyed or keyed archives will be converted to 10.4-style if the archived formatter has uncustomized format string -- most of those found in IB's NSNumberFormatter inspector -- at unarchive time.

NSIndexPath

NSIndexPath is a new class for representing a sequence of indexes; its primary purpose is to encapsulate the information needed to navigate down a tree of objects. NSTreeController, a new class in the AppKit, makes use of NSIndexPath.

The designated initializer for NSIndexPath is:

```
- (id)initWithIndexes:(unsigned int *)indexes length:(unsigned int)length;
```

and the primitive accessors are:

```
- (unsigned int)indexAtPosition:(unsigned int)position;
- (unsigned int)length;
```

Please refer to NSIndexPath.h for the rest of the API.

Key-Value Coding and Observing for Sets (Section added since WWDC)

Mac OS 10.3 introduced key-value observing (KVO), a mechanism by which one object can observe the properties, including ordered to-many relationships, of another. It also introduced explicit support for ordered to-many relationships to the existing key-value coding (KVC) mechanism. Because some applications, especially those that use CoreData, need to represent unordered to-many relationships, support for unordered to-many relationships has been added to KVC/KVO. This support takes the form of additions to KVC's -valueForKey: method, new KVC methods, and new KVO methods.

To explicitly support nonmutating access of unordered to-many relationships, -[NSObject(NSKeyValueCoding) valueForKey:], an existing method, now finds KVC-compliance methods that correspond to the NSSet primitives. After looking for array accessor methods (as in Mac OS 10.3) but before looking for instance variables (as in Mac OS 10.3), it now looks for set accessor methods. From NSKeyValueCoding.h's comments for -valueForKey:

"3 (introduced in Mac OS 10.4). Otherwise (no simple accessor method or set of array access methods is found), searches the class of the receiver for a threesome of methods whose names match the patterns -countOf<Key>, -enumeratorOf<Key>, and -memberOf<Key>: (corresponding to the primitive methods defined by the NSSet class). If all three such methods are found a collection proxy object that responds to all NSSet methods is returned. Each NSSet message sent to the collection proxy object will result in some combination of -countOf<Key>, -enumeratorOf<Key>, and -memberOf<Key>: messages being sent to the original receiver of -valueForKey:."

Your class' implementations of such KVC-compliance methods should have the same signatures as:

```
- (unsigned int)countOf<Key>;
- (NSEnumerator *)enumeratorOf<Key>;
- (id)memberOf<Key>;
```

As has been the case for ordered relationships, it is reasonable and usually more convenient to merely implement a KVC-compliance accessor method for the entire collection. For an unordered relationship the accessor method should have the same signature as:

```
- (NSSet *)<key>;
```

To support mutation of unordered to-many relationships, two new methods have been added to NSObject(NSKeyValueCoding):

```
- (NSMutableSet *)mutableSetValueForKey:(NSString *)key;
- (NSMutableSet *)mutableSetValueForKeyPath:(NSString *)keyPath;
```

Given the key or key path that identifies a relationship, return an object that can be used to mutate the relationship. Several KVC-compliance method name patterns are recognized:

```
- (void)add<Key>Object:(id)objectToAdd;
- (void)remove<Key>Object:(id)objectToRemove;
- (void)add<Key>:(NSSet *)objectsToAdd;
- (void)remove<Key>:(NSSet *)objectsToRemove;
- (void)intersect<Key>:(NSSet *)intersectionObjects;
- (void)set<Key>:(NSSet *)replacementObjects;
```

Typically your class will implement one add and one remove method for a particular key. There may be substantial performance benefits to implementing the NSSet-taking ones. See NSKeyValueCoding.h's comments for details.

-mutableSetValueForKeyPath: follows the pattern established by existing KVC key path-taking methods; basically it just invokes [[self valueForKey:firstKeyPathComponent] mutableSetValueForKeyPath:theRestOfTheKeyPath].

To support observation of unordered to-many relationship mutations, a new enumeration has been added, and a pair of methods have been added to NSObject(NSKeyValueObserverNotification):

```
typedef enum {
    // The set representing an unordered to-many relationship is being changed using NSMutableSet's
       -unionSet:, -minusSet:, -intersectSet:, or -setSet: method, or something that has the same
       semantics.
    NSKeyValueUnionSetMutation = 1,
    NSKeyValueMinusSetMutation = 2,
    NSKeyValueIntersectSetMutation = 3,
    NSKeyValueSetSetMutation = 4
} NSKeyValueSetMutationKind;

- (void)willChangeValueForKey:(NSString *)key withSetMutation:(NSKeyValueSetMutationKind)mutationKind
    usingObjects:(NSSet *)objects;
- (void)didChangeValueForKey:(NSString *)key withSetMutation:(NSKeyValueSetMutationKind)mutationKind
    usingObjects:(NSSet *)objects;
```

Given a key that identifies an unordered to-many relationship, prepare to send, and send, respectively, an -observeValueForKeyPath:ofObject:change:context: message. The passed-in mutation kind corresponds to an NSMutableSet method. The passed-in set must contain the set that would be passed to the corresponding NSMutableSet method. Invocations of these methods must always be paired, with identical arguments. The change dictionaries in notifications resulting from use of these methods always contain an NSKeyValueChangeKindKey entry. Its value will depend on the passed-in mutationKind value:
- NSKeyValueUnionSetMutation -> NSKeyValueChangeInsertion
- NSKeyValueMinusSetMutation -> NSKeyValueChangeRemoval
- NSKeyValueIntersectSetMutation -> NSKeyValueChangeRemoval
- NSKeyValueSetSetMutation -> NSKeyValueChangeReplacement
The change dictionary may also contain optional entries:
- The NSKeyValueChangeOldKey entry, if present (only for for NSKeyValueChangeRemoval and NSKeyValueChangeReplacement), contains the set of objects that were removed.
- The NSKeyValueChangeNewKey entry, if present (only for NSKeyValueChangeInsertion and NSKeyValueChangeReplacement), contains the set of objects that were added.

New Overrides of Public Key-Value Coding Methods By NSSet (Section added since WWDC)

For consistency with NSArray's existing KVC behavior, NSSet now overrides two public KVC methods:

```
- (id)valueForKey:(NSString *)key;
```

Return a set containing the results of invoking -valueForKey: on each of the receiver's members. The returned set might not have the same number of members as the receiver. The returned set will not contain any elements corresponding to instances of -valueForKey: returning nil (in contrast with -[NSArray(NSKeyValueCoding) valueForKey:], which may put NSNulls in the arrays it returns).

For backward binary compatibility, this method will merely invoke NSObject's default implementation of -valueForKey: in applications linked against Mac OS 10.3 or earlier.

```
- (void)setValue:(id)value forKey:(NSString *)key;
```

Invoke -setValue:forKey: on each of the receiver's members.

For backward binary compatibility, this method will merely invoke NSObject's default implementation of -setValue:forKey: in applications linked against Mac OS 10.3 or earlier.

Support for New Method Name Patterns in Key-Value Coding for Arrays (Section added since WWDC)

-[NSObject(NSKeyValueCoding) valueForKey:], an existing method, will now find methods whose names conform to the pattern:

```
- (NSArray *)<key>AtIndexes:(NSIndexSet *)indexes;
```

in addition to the -objectIn<Key>AtIndex: pattern supported in Mac OS 10.3. If the same class has both a -<key>AtIndexes: and an -objectIn<Key>AtIndex: method, automatic collection proxies returned by -valueForKey: will invoke whichever is best for performance, depending on the message that was sent to the collection proxy.

-[NSObject(NSKeyValueCoding) mutableArrayValueForKey:], an existing method, will now find methods whose names conform to the patterns:

```
- (void)insert<Key>:(NSArray *)objectsToAdd atIndexes:(NSIndexSet *)indexes;
- (void)remove<Key>AtIndexes:(NSIndexSet *)indexes;
- (void)replace<Key>AtIndexes:(NSIndexSet *)indexes with<Key>:(NSArray *)replacementObjects;
```

in addition to the -insertObject:in<Key>AtIndex:, -removeObjectFrom<Key>AtIndex:, and -replaceObjectIn<Key>AtIndex:withObject: patterns supported in Mac OS 10.3. If the same class has both an index-set-taking and an index-taking insertion, removal, or replacement method, automatic mutable collection proxies returned by -valueForKey: will invoke whichever is best for performance, depending on the message that was sent to the mutable collection proxy.

New NSIndexSet-Taking Method in NSArray

A new method has been added to NSArray:

```
- (NSArray *)objectsAtIndexes:(NSIndexSet *)indexes;
```

Return an array of the objects at the indexes, or throw an NSRangeException if the largest index in the set is greater than the receiving array's count.

New NSIndexSet-Taking Methods in NSMutableArray

New methods have been added to NSMutableArray:

```
- (void)insertObjects:(NSArray *)objects atIndexes:(NSIndexSet *)indexes;
```

Insert the objects at the indexes, or throw an NSRangeException if the largest index in the set is not less than the sum of the counts of the receiving array and the passed-in array, or throw an NSInvalidArgumentException if the passed-in NSIndexSet and NSArray don't have the same count. The passed-in indexes are the indexes at which the inserted objects are to be located after the entire operation is complete.

```
- (void)removeObjectsAtIndexes:(NSIndexSet *)indexes;
```

Remove the indexed objects, or throw an NSRangeException if the largest index in the set is greater than the receiving array's count. The passed-in indexes are the indexes at which the removed objects were before the operation.

```
- (void)replaceObjectsAtIndexes:(NSIndexSet *)indexes withObjects:(NSArray *)objects;
```

Replace the indexed objects with other objects, or throw an NSRangeException if the largest index in the set is greater than the receiving array's count., or throw an NSInvalidArgumentException if the passed-in NSIndexSet and NSArray don't have the same count.

Explicit Overriding of Key-Value Observer Registration Methods by NSArray and NSSet

NSArrays and NSSets are not observable, so these methods:

```
- (void)addObserver:(NSObject *)observer forKeyPath:(NSString *)keyPath
    options:(NSKeyValueObservingOptions)options context:(void *)context;
- (void)removeObserver:(NSObject *)observer forKeyPath:(NSString *)keyPath;
```

raise exceptions when invoked on NSArrays and NSSets. Instead of observing an array or set, observe the ordered or unordered to-many relationship for which the array or set is the collection of related objects. For NSArrays, this behavior is unchanged since Mac OS 10.3; the only change is the explicit declaration of the exception-throwing method overrides in NSKeyValueObserving.h. For NSSets, this behavior is new since Mac OS 10.3; for backward binary compatibility, the exception-throwing method overrides merely invoke NSObject's default implementations in applications linked against Mac OS 10.3 or earlier.

Deprecation of Stored Key-Value Coding Methods

In Mac OS 10.3, the documentation for each of these methods:

```
+ (BOOL)useStoredAccessor;
- (id)storedValueForKey:(NSString *)key;
- (void)takeStoredValue:(id)value forKey:(NSString *)key;
```

claimed:

Note: This method is not deprecated as of Mac OS X v10.3, but may be deprecated in favor of a new method in a future release of Mac OS X.

These methods are now deprecated. Their implementations are virtually unchanged since Mac OS 10.3, and you may continue to use them, but their implementations will not be improved to keep pace with improvements to key-value coding and observing. -storeValueForKey: and -takeStoredValue:forKey: are not invoked from anywhere within Cocoa, as in Mac OS 10.3 and earlier. +useStoredAccessor is only invoked from within -storeValueForKey: and -takeStoredValue:forKey:, as in Mac OS 10.3 and earlier.

If you are using the new NSManagedObject class, use its -primitiveValueForKey: and -setPrimitiveValue:forKey: methods instead.

Publication of String Constants for Key-Value Coding Array Operator Names

Strings for the names of array operators, which have been supported by key-value coding since Mac OS 10.3, have been published:

```
NSString *NSAverageKeyValueOperator;
NSString *NSCountKeyValueOperator;
NSString *NSDistinctUnionOfArraysKeyValueOperator;
NSString *NSDistinctUnionOfObjectsKeyValueOperator;
NSString *NSMaximumKeyValueOperator;
NSString *NSMinimumKeyValueOperator;
NSString *NSSumKeyValueOperator;
NSString *NSUnionOfArraysKeyValueOperator;
NSString *NSUnionOfObjectsKeyValueOperator;
```

The values of these do not include '@' array operator prefixes.

Bug Fix in Key-Value Coding

In Mac OS 10.3, -[NSArray valueForKeyPath:]'s support for operators had bugs that caused crashing, hanging, and exception throwing when the @sum, @max, or @min operator was applied to an array containing an element that returned nil for the keyed value. For example:

```
[[NSArray arrayWithObject:[NSDictionary dictionary]] valueForKeyPath:@"@sum.anyOldKey"]
```

would crash or hang (because the nil that the dictionary returned from -valueForKey:@"anyOldKey" was mishandled). This problem has been fixed.

In both Mac OS 10.3 and Mac OS 10.4 the support for the @count and @avg operators do not take into account returned nils. This is true regardless of both -[NSArray valueForKeyPath:] and -[NSSet valueForKeyPath:].

Bug Fix in Key-Value Observing for -description Methods (Section added since WWDC)

In Mac OS 10.3, -description messages sent to any observed object would always result in the invocation of a private method that behaved more-or-less like -[NSObject description], even if -description was overridden by the receiver's class, unless key-value observing autonotification was disabled for all observed properties by overriding of +automaticallyNotifiesObserversForKey:. This was a bug and has been fixed. Your class' override of -description will now be invoked even if the receiver is observed and its "isa is swizzled" by KVO's autonotification machinery.

-[NSObject description] itself, which is primarily for use in debugging, does not attempt to hide isa-swizzling from you. For example, when -[NSObject description] would otherwise return something like "<Foo: 0x301780>" it will now return something like "<NSKVONotifying_Foo: 0x301780>" if the object's isa has been swizzled. If you see this while debugging and are surprised to see that an object is being observed you can send the object an -observationInfo message and view the description of the results. For example, a GDB command like 'po [mySurprisinglyObservedObject observationInfo]' will output a list of observances on that object.

Bug Fix in Key-Value Observing for Nested WillChange/DidChange Sequences (Section added since WWDC)

In Mac OS 10.3 there was a bug in which nested willChange/didChange sequences would result in observer notifications being sent too early. For example:

```
    [observedObject addObserver:observer forKeyPath:@"someProperty" options:0 context:NULL];
    [observedObject willChangeValueForKey:@"someProperty"];
    [observedObject willChangeValueForKey:@"someProperty"];
    [observedObject didChangeValueForKey:@"someProperty"];
            // Observer is sent two -observeValueForKeyPath:... messages. It should be sent one.
    [observedObject didChangeValueForKey:@"someProperty"];
            // Observer is sent zero -observeValueForKeyPath:... messages.
            // It should be sent one (unless the first -observeValueForKeyPath:... invoked
            // [observedObject removeObserver:observer forKeyPath:@"someProperty"]).
```

This bug has been fixed. It was most noticeable when the observer did something like removing itself as an observer in response to the first observer notification, with the expectation that it would not receive a second.

NSAppleEventDescriptor Changes

+appleEventWithEventClass:eventID:targetDescriptor:returnID:transactionID: and -initWithEventClass:eventID:targetDescriptor:returnID:transactionID:] have each been updated to accept a nil target descriptor argument. The resulting Apple event descriptor has no keyAddressAttr attribute.

Key Value Observing and Cocoa Scripting

In Mac OS 10.3, Cocoa's scripting support did not use Key Value Coding methods introduced in Panther like -setValue:forKey: and -mutableArrayValueForKey:, so changes to model objects made by AppleScripts were not observable using automatic Key Value Observing. This has been fixed. Cocoa's scripting support now invokes -setValue:forKey: instead of -takeValue:forKey: unless the container in question overrides -takeValue:forKey:, in which case -takeValue:forKey: will be invoked for backward binary compatibility. The implementations of -insertValue:atIndex:inPropertyWithKey:, -removeValueAtIndex:fromPropertyWithKey:, and -replaceValueAtIndex:inPropertyWithKey:withValue: have been updated to invoke -mutableArrayValueForKey: and mutate the result if no corresponding scripting-KVC-compliant method (insertIn<Key>:atIndex:, -removeFrom<Key>AtIndex:, or replaceIn<Key>:atIndex:, respectively) is found.

Bug Fixes in NSScriptCommand Suspending and Resuming

In Mac OS 10.3, -[NSScriptCommand suspendExecution] could malfunction if the corresponding invocation of resumeExecutionWithResult:] was done very quickly in a different thread, leading to crashes. This bug has been fixed.

Execution of multiple-receiver commands could result in the command being sent to the same receiver repeatedly, if the receiver's command handler suspending the command. This bug has been fixed.

Cocoa Scripting Support for .sdef Files

Cocoa now supports declaration of scriptability in .sdef files instead of .scriptSuite/.scriptTerminology files. This support is incomplete in several ways in the WWDC 2004 seed of Tiger, but here are some interesting facts:

• 'man sdef' to find out what the file format is all about. See the .sdef files in Foundation and AppKit's Resources directories for examples. Also, check the Apple Developer Connection site at <connect.apple.com> for information about WWDC 2004 Session 430, "Advances in Cocoa Scripting," for more information.

• .sdefs are loaded instead of .scriptSuite/.scriptTerminology files only if the app is linked against Tiger or better and the app's main bundle includes at least one .sdef file. This will likely change so that Cocoa's decision to load .sdef instead of .scriptSuite/.scriptTerminology files depends on a new "OSAScriptingDefinition" Info.plist entry.

• A big difference between Cocoa's .sdef parsing and the version of sdp that came with Panther is that the name of the Cocoa attribute that identifies a property's KVC key is "key," not "method."

• Other big differences: use "specifier" now, not "object," and "location specifier" instead of "location."

• The color class in NSCoreSuite.[scriptSuite|scriptTerminology] won't be declared as a class in Cocoa's own .sdef files. One of Cocoa's .sdef files just declares a simple value type, "color." Apple event descriptors of several types are convertable to NSColors.

• The "file" type that .sdef has always had corresponds to the NSURL class.

• The "any" type that .sdef has always had corresponds to the NSAppleEventDescriptor class, not NSObject.

• The Objective-C implementation class of the "item" scriptable class is NSObject, not the artificial AbstractObject type that was used in .scriptSuite files.

• The standard for document classes has changed, to be consistent with the Scripting Interface Guidelines. Every document class should now have a read-only "file" property of type "file," instead of a read-write "path" property of string type. Document class' "name" property should now be read-only instead of read-write.

• The Cocoa key for the "name" property should be "displayName," not "lastComponentOfFileName." -[NSDocument(Scripting) lastComponentOfFileName] is probably going to be deprecated.

• Various window class properties have been removed to bring Cocoa in line with the new Scripting Interface Guidelines. Window names are now read-only. "Miniaturizable" and "miniaturized" have been renamed to "minimizable" and "minimized."

• The text class' "size" property's type is now real instead of integer.

• The text attachment class now has a read-only "file" property of type "file" instead of read-write "file name" string property.

• The open command's direct parameter is of type "list of file."

• The print command's direct parameter is of type "list of file | specifier," so you can either print files or documents (this may not work very well in the WWDC 2004 seed of Tiger).

• The close command's "saving in" parameter and the save command's "in" parameter are now of type "file."

• Various script command handler methods in AppKit have been updated to deal with NSURL arguments, because that's what "file" Apple event descriptors get converted to.

• The "with data" parameter of the make command has been renamed to "with contents."

• NSScriptClassDescription's -suiteName and -className return the human-readable strings by which .sdefs are keyed, when the class is declared in an .sdef.

• Ditto for NSScriptCommand's -suiteName and -commandName.

• NSScriptSuiteRegistry's -suiteNames returns the same sort of human-readable strings for .sdef-declared stuff, and -appleEventCodeForSuite:, -bundleForSuite:, -classDescriptionsInSuite:, and -commandDescriptionsInSuite: take the same sort of strings.

NSNetServices finite resolves

Leaving resolves on NSNetServices open generates (largely) unnecessary network traffic - resolves tend to happen either very quickly, or not at all. To this end, we are deprecating the original open-ended -resolve method in favor of:

```
/* Starts a resolve for the NSNetService of a finite duration. If your delegate is called before the
timeout expires, the resolve can be considered successful. If the resolve times out, your
netService:didNotResolve:  callback will be called with the appropriate error dictionary.
*/
- (void)resolveWithTimeout:(NSTimeInterval)timeout;
```

This will allow a resolve of finite duration, limiting network traffic. For applications linked on Tiger, the now-deprecated -resolve will call -resolveWithTimeout with a timeout of 5 seconds (generally if something is going to happen, it'll happen within the first half-second or so). For applications linked on Panther running on Tiger, -resolve will call -resolveWithTimeout: with a timeout in the distant future.

NSNetServices TXT record support

In Tiger, we have deprecated the protocolSpecificInformation calls that work with NSStrings in favor of:

```
/* Allows the developer to use an NSData containing arbitrary bytes as the TXT record.
Returns YES if the data is successfully set as the TXT record. Returns NO if it cannot be set.
*/
```

```
- (BOOL)setTXTRecordData:(NSData *)recordData;
- (NSData *)TXTRecordData;
```

Developers checking to see if this is a key-value style text record can use the following API for conversion attempts:

```
/* TXT record data can be presented either as an NSData or an NSDictionary of key-value pairs.
It's very useful to be able to convert between the two. NSNetService provides a pair of class methods
to do this conversion. Each returns nil in the event that the conversion cannot take place.
*/
+ (NSDictionary *)dictionaryFromTXTRecordData:(NSData *)txtData;
+ (NSData *)dataFromTXTRecordDictionary:(NSDictionary *)txtDictionary;
```

The primary utility of this is for use in netService:didUpdateTXTRecordData:.

Applications linked on Panther or Jaguar and running on Tiger will continue to be able to interoperate with the same application running on other Panther or Jaguar machines on the network.

NSNetServices TXT record update observation

In the old behavior, an additional netService:didResolve: delegate method would be called when the protocolSpecificInformation updated. This requires leaving the resolve open, which generates unnecessary network traffic.

On Tiger, if a delegate implements the appropriate delegate method, they will get the new behavior instead, which does not require that a resolve be open at all (it will generate some network traffic, but the intent is that it will be less traffic).

First, on NSNetService itself, two new methods:

```
/* These calls control whether an NSNetService will watch for TXT record updates, notification
of which would be delivered to the delegate on the netServiceDidUpdateTXTRecord: method in the
NSNetServiceDelegateMethods category.
*/
- (void)startMonitoring;
- (void)stopMonitoring;
```

And on the NSNetServiceDelegateMethods category:

```
/* Called to inform the delegate that the TXT record associated with the sending NSNetService
object has updated. txtData contains the new TXT record as an NSData.
*/
- (void)netService:(NSNetService *)sender didUpdateTXTRecordData:(NSData *)txtData;
```

This method delivers the updated TXT record as an NSData to the delegate.

NSNetServices Publishing

There is a new delegate method indicating that the net service was successfully published:

```
- (void)netServiceDidPublish:(NSNetService *)sender;
```

An error (delivered on the current error delegate method) is triggered in one of two cases: (1) Timeout has been reached, or (2) Name collision has occurred.

There is a new error code for the timeout case:

```
NSNetServicesTimeoutError = -72007
```

NSDirectoryEnumerator (Section added since WWDC)

A long-standing bug in NSDirectoryEnumerator's -directoryAttributes method has been fixed. In Tiger, it now behaves as documented, returning the attributes of the directory at which the enumeration began.

For the common case of enumerating a directory without asking for attributes, performance is much improved over previous versions of OS X.

Performance Improvement in Keyed Archiving

In Mac OS X 10.4, there has been a significant performance improvement in creating keyed archives which have lots of array objects.

Performance Improvement in Notification Posting

In Mac OS X 10.4, there has been a significant performance improvement in posting notifications for which there are no observers. The performance of all posting has also generally improved.

NSNumber (Section added since WWDC)

Since NSNumbers are immutable, we now cache and reuse some "popular" numbers in order to reduce allocation activity. This means that some distinct NSNumber allocation calls might return the same exact object, with a incremented reference count. Applications should not rely on getting distinct objects from separate NSNumber creation requests.

In Tiger "popular" numbers include -1..12, although this might very well change at any point, including in a software update.

Comparisons against NAN (not a number) now work correctly in that no number is equal to NAN (including +[NSDecimalNumber notANumber]), except NAN itself (which has to be special case, since object equality must hold). Comparison against NAN will also give deterministic results; however, no assumptions should be made on the ordering, and the results may vary between system releases.

NSString

NSString's getCString:maxLength: and getCString:maxLength:range:remainingRange: methods will no longer raise if the provided buffer size is not large enough. Note that these did not raise in all cases.

More NSString and NSMutableString methods now detect out of bounds indexes and invalid ranges. One case which wasn't properly being detected so far was where the location and/or length were so large that the sum ended up being less than the length of the string. For applications linked on Tiger, this error will cause an exception after logging the problem in the console; for older apps, for compatibility, we just log once, but don't raise. Note that even if the old behavior seems work under a set of circumstances, it's actually just getting lucky and in many cases might end up crashing at some point. So any instances of these logs should be fixed.

If a call to componentsSeparatedByString: resulted in an array with one string, equal to the receiver, the one string was sometimes not copied and its value could change when the receiver string was subsequently edited. This has been fixed.

NSString cString API deprecation

In the continuing effort to deprecate cString methods without encoding arguments, NSString now has the following new API. Note that this new API also provides NSError returns as a way to provide more info about failures in cases where it might matter to the user, for instance, for reading/writing.

The following return the maximum and exact number of bytes needed to store the receiver in the specified encoding in non-external representation. The first one is O(1), while the second one is O(n). These do not include space for a terminating null. The second one will return 0 if the conversion to the specified encoding is not possible; the first one doesn't check.

```
- (unsigned)maximumLengthOfBytesUsingEncoding:(NSStringEncoding)enc;
- (unsigned)lengthOfBytesUsingEncoding:(NSStringEncoding)enc;
```

Methods to convert NSString to a NULL-terminated cString using the specified encoding. Note, these are the "new" cString methods, and are not deprecated like the older cString methods which do not take encoding arguments.

```
- (const char *)cStringUsingEncoding:(NSStringEncoding)encoding;
- (BOOL)getCString:(char *)buffer maxLength:(unsigned)maxBufferCount encoding:(NSStringEncoding)encoding;
```

The following are new and improved cString methods which take explicit encoding arguments.

```
- (id)initWithCString:(const char *)nullTerminatedCString encoding:(NSStringEncoding)encoding;
+ (id)stringWithCString:(const char *)cString encoding:(NSStringEncoding)enc;
```

This following is exposed in Tiger, but has actually been around since 10.3 and so is available for use on 10.3.

```
- (id)initWithBytesNoCopy:(void *)bytes length:(unsigned)len
      encoding:(NSStringEncoding)encoding freeWhenDone:(BOOL)freeBuffer;
```

These use the specified encoding. If nil is returned, the optional error return indicates problem that was encountered (for instance, file system or encoding errors).

```
- (id)initWithContentsOfURL:(NSURL *)url encoding:(NSStringEncoding)enc error:(NSError **)error;
- (id)initWithContentsOfFile:(NSString *)path encoding:(NSStringEncoding)enc error:(NSError **)error;
+ (id)stringWithContentsOfURL:(NSURL *)url encoding:(NSStringEncoding)enc error:(NSError **)error;
+ (id)stringWithContentsOfFile:(NSString *)path encoding:(NSStringEncoding)enc error:(NSError **)error;
```

The following try to determine the encoding, and return the encoding which was used. Note that these methods might get "smarter" in subsequent releases of the system, and use additional techniques for recognizing encodings. If nil is returned, the optional error return indicates problem that was encountered (for instance, file system or encoding errors).

```
- (id)initWithContentsOfURL:(NSURL *)url usedEncoding:(NSStringEncoding *)enc error:(NSError **)error;
- (id)initWithContentsOfFile:(NSString *)path usedEncoding:(NSStringEncoding *)enc error:(NSError **)error;
+ (id)stringWithContentsOfURL:(NSURL *)url usedEncoding:(NSStringEncoding *)enc error:(NSError **)error;
+ (id)stringWithContentsOfFile:(NSString *)path usedEncoding:(NSStringEncoding *)enc error:(NSError **)error;
```

The following write to specified url using the specified encoding. The optional error return is to indicate file system or encoding errors.

```
- (BOOL)writeToURL:(NSURL *)url atomically:(BOOL)useAuxiliaryFile
      encoding:(NSStringEncoding)enc error:(NSError **)error;
- (BOOL)writeToFile:(NSString *)path atomically:(BOOL)useAuxiliaryFile
      encoding:(NSStringEncoding)enc error:(NSError **)error;
```

The following methods are deprecated and should not be used. They will be removed from the headers as soon as practical:

```
- (const char *)cString;
- (const char *)lossyCString;
- (unsigned)cStringLength;
- (void)getCString:(char *)bytes;
- (void)getCString:(char *)bytes maxLength:(unsigned)max;
- (void)getCString:(char *)bytes maxLength:(unsigned)max range:(NSRange)rg remainingRange:(NSRangePointer)rem;

+ (id)stringWithCString:(const char *)bytes length:(unsigned)length;
+ (id)stringWithCString:(const char *)bytes;
+ (id)stringWithContentsOfFile:(NSString *)path;
+ (id)stringWithContentsOfURL:(NSURL *)url;

- (id)initWithContentsOfFile:(NSString *)path;
- (id)initWithContentsOfURL:(NSURL *)url;

- (id)initWithCStringNoCopy:(char *)bytes length:(unsigned)length freeWhenDone:(BOOL)freeBuffer;
- (id)initWithCString:(const char *)bytes length:(unsigned)length;
- (id)initWithCString:(const char *)bytes;

- (BOOL)writeToFile:(NSString *)path atomically:(BOOL)flag;
- (BOOL)writeToURL:(NSURL *)url atomically:(BOOL)flag;
```

NSAttributedString

Attributed string methods such as attribute:atIndex:effectiveRange: used to raise NSInternalInconsistencyException on out-of-bounds accesses; they have now been switched to raising NSRangeException as documented.

-[NSMutableAttributedString initWithString:attributes:] could create a corrupt attributed string when initialized with a string of non-zero length. This has been fixed.

There is now a CFAttributedString, which is toll-free bridged to NSAttributedString.

NSAttributedString mutation of attributes warning (Section added since WWDC)

Once an attribute value is set in an attributed string, the attribute value should not be modified behind the attributed string. So any modification to the value should be performed by a new set operation (using any one of the attribute mutation methods in NSMutableAttributedString), with the new value.

One reason for this is that the attribute values are retained by the attributed string, and how the value propagates through an attributed string as the attributed string is further edited is not predictable. If you change the value, you might be editing more portions of the attributed string than you thought. In fact the value could appear on pieces of the attributed string in the undo stack, or might have even been copied to a totally different document. It's possible that this is not a concern in some instances.

Another reason for this limitation is that attributed strings do caching and uniquing of attributes (this actually occurs more at the AppKit level, but that is an implementation detail). The uniquing assumes attribute values aren't changing, that is, that isEqual: and hash on attribute values will not change as long as the attribute value is in an attributed string.

If you must change attribute values, two possible suggestions are:
- Use an attribute value whose isEqual: and hash do not depend on the values you are modifying. You might need to create a wrapper object for this. Or,
- Use indirection; use the attribute value as a lookup key into a table where the actual value can be changed. For instance, this might be the appropriate approach for having a "stylesheet" like attribute.

This warning has always been applicable, but it's more true in Tiger, since NSAttributedString now hashes more of the values inside an attribute dictionary when uniquing it. This has led to problems in a few applications where some attribute values were being mutated, or worse, corrupted or freed (but previously going unnoticed). As a temporary workaround to this issue, a default named NSPreTigerAttributedStringHash is available; setting this to YES for your application will cause NSAttributedString to use the same hashing algorithm as in Panther. This can help diagnose problems, and even bring an app back to life. This default should not be used as a long term solution though.

NSCocoaErrorDomain

In order to formalize the kind of errors returned by AppKit and Foundation frameworks, there is now a new error domain:

```
NSString *const NSCocoaErrorDomain;
```

This represents the domain for errors returned from most Cocoa subsystems. Errors in this domain have reasonable and localized user-readable error messages that are good enough to be displayed in alerts and other user interfaces. With a few existing exceptions, AppKit, Foundation, and CoreData APIs are expected to return NSErrors in this domain; in some cases lower level errors will be packaged as the underlying error.

NSError (Section updated since WWDC)

In order to enable presenting more reasonable user panels for various errors, NSError now has the ability to return the secondary message along with the titles of button(s) appropriate for an alert. With this, even a low level error (say, from NSData) is able to return an NSError that can be usefully presented to the user:
localizedDescription: "Could not save file 'Letter' in folder 'Documents' because the volume 'MyDisk' doesn't have enough space."
localizedRecoverySuggestion: "Remove files from the disk and try again."
localizedRecoveryOptions: nil (so, "OK")

A higher level (NSDocument or bindings, for instance), would do better by extending this, say by providing additional options (buttons) to try remedying the situation, retrying the save elsewhere, etc.

The following method returns the string that can be displayed as the "informative" (aka "secondary") message on an alert panel. Returns nil if no such string is available. Default implementation of this will pick up the value of the NSLocalizedRecoverySuggestionKey from the userInfo dictionary.

```
- (NSString *)localizedRecoverySuggestion;
```

The following method returns titles of buttons that are appropriate for displaying in an alert. These should match the string provided as a part of localizedRecoverySuggestion. The first string would be the title of the right-most and default button, the second one next to it, and so on. If used in an alert the corresponding default return values are NSAlertFirstButtonReturn + n. Default implementation of this will pick up the value of the NSLocalizedRecoveryOptionsKey from the userInfo dictionary. nil return usually implies no special suggestion, which would imply a single "OK" button.

```
- (NSArray *)localizedRecoveryOptions;
```

The following method returns just the reason for the failure, without mentioning the operation. This can be useful when the caller has a better idea of what the operation is, but still wants to leverage NSError's localized error string. Note that this may return nil, which indicates NSError had no idea why the operation failed:

```
- (NSArray *)localizedFailureReason;
```

In addition to the above, NSError now also has the ability to attempt recovery from the error. The following method returns an object that conforms to the NSErrorRecoveryAttempting informal protocol:

```
- (id)recoveryAttempter;
```

The default implementation of this method merely returns [[self userInfo] objectForKey:NSRecoveryAttempterErrorKey]:

```
NSString *const NSRecoveryAttempterErrorKey;
```

If non-nil, the recovery attempter must be an object that can correctly interpret an index into the array returned by -localizedRecoveryOptions.

The NSErrorRecoveryAttempting informal protocol has two methods:

```
- (void)attemptRecoveryFromError:(NSError *)error optionIndex:(unsigned int)recoveryOptionIndex
        delegate:(id)delegate didRecoverSelector:(SEL)didRecoverSelector contextInfo:(void *)contextInfo;
```

Given that an error alert has been presented document-modally to the user, and the user has chosen one of the error's recovery options, attempt recovery from the error, and send the selected message to the specified delegate. The option index is an index into the error's array of localized recovery options. The method selected by didRecoverSelector must have the same signature as:

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

The value passed for didRecover must be YES if error recovery was completely successful, NO otherwise.

```
- (BOOL)attemptRecoveryFromError:(NSError *)error optionIndex:(unsigned int)recoveryOptionIndex;
```

Given that an error alert has been presented applicaton-modally to the user, and the user has chosen one of the error's recovery options, attempt recovery from the error, and return YES if error recovery was completely successful, NO otherwise. The recovery option index is an index into the error's array of localized recovery options.

See the "NSResponder-Based Error Presentation" section in the AppKit release notes for information about how Cocoa itself uses error recovery attempters.

A change from Panther is that NSError will now pass by copy over distributed objects, unless the argument is specified to be "byref". It used to always go by reference.

Additional NSError notes (Section added since WWDC)

Cocoa's conventions for NSError returns include:
- NSError return should always be present in addition to a simpler way to indicate failure, for instance, via the return value of the function (NO, nil, NULL, or whatever's appropriate). NSErrors are typically returned via a "by-reference" argument, that is, pointer to an NSError object.
- The NSError argument is "optional." If the caller has specified NULL for the error pointer argument, then don't return it.
- If an failure is indicated in a call, and the caller has passed in a non-NULL error pointer, an NSError must be returned to indicate what went wrong. It's not acceptable to set *errorPtr to nil, say because the exact cause of the error couldn't be determined, or there was no real error.
- If a failure is not indicated, then do not use NSError as a way to communicate other state, such as warnings. On a successful return, *errorPtr is typically not modified; maybe set to NULL.
- NSErrors are not used for programming errors (such as array index out of bounds, invalid parameter value, attempt to mutate immutable object, etc). Cocoa uses exceptions for those.

Two "gotchas" worth pointing out about NSErrors:

The userInfo method might return nil in some cases, for instance if the NSError was created with nil. This means that if you are copying an NSError and adding some new keys to the userInfo dictionary, something like [[error userInfo] mutableCopy] will fail if the dictionary is nil. So, check for nil when doing this.

Another gotcha to point out is with the autoreleased return of NSErrors. If a method calls another method which returns an NSError, and then returns that NSError to its caller, you need to be careful if you happen to add an autorelease pool inside that method, since you might end up releasing the error you got from the nested call.

This of course is the same kind of problem you'd need to watch out for with normal return values; however, it's especially subtle when NSErrors are being returned. That's because NSErrors are secondary return values, so more easily missed. In addition, in most regular testing scenarios error code paths are not tested, which would mean the bug would not be encountered. In fact, you'd end up with a bug which causes a crash only when trying to report an error, which is unfortunate.

The solution of course is to extend the lifetime of the NSError around the destruction of the pool, for instance:

```
- (BOOL)aMethod:... error:(NSError **)errorPtr {
    BOOL success;
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    ...
    success = [someObj anotherMethod:...  error:errorPtr];
    ...
    if (!success && errorPtr) [*errorPtr retain];        // Extend the lifetime of the error
    [pool release];
    if (!success && errorPtr) [*errorPtr autorelease];
    return success;
}
```

NSData (Section added since WWDC)

The following new methods return optional NSErrors for file reading and writing. As with other NSError-returning methods, an NSError is returned in the case of failure (NO or NULL return), if errorPtr is not NULL. In many cases the returned error is good enough to be presented to the user:

```
enum {    // Options for NSData reading methods
    NSMappedRead = 1,        // Hint to map the file in if possible
    NSUncachedRead = 2        // Hint to get the file not to be cached in the kernel
};

enum {    // Options for NSData writing methods
    NSAtomicWrite = 1        // Hint to use auxiliary file when saving; equivalent to atomically:YES
};

+ (id)dataWithContentsOfFile:(NSString *)path options:(unsigned)readOptionsMask error:(NSError **)errorPtr;
+ (id)dataWithContentsOfURL:(NSURL *)url options:(unsigned)readOptionsMask error:(NSError **)errorPtr;

- (id)initWithContentsOfFile:(NSString *)path options:(unsigned)readOptionsMask error:(NSError **)errorPtr;
- (id)initWithContentsOfURL:(NSURL *)url options:(unsigned)readOptionsMask error:(NSError **)errorPtr;

- (BOOL)writeToFile:(NSString *)path options:(unsigned)writeOptionsMask error:(NSError **)errorPtr;
- (BOOL)writeToURL:(NSURL *)url options:(unsigned)writeOptionsMask error:(NSError **)errorPtr;
```

NSAffineTransform (Section added since WWDC)

The NSAffineTransform class implementation moved from AppKit to Foundation. -transformBezierPath:, -set, and -concat methods are now part of a category implemented in AppKit.

Using NSURLRequest for requests with large bodies

Historically, in order to use NSURLRequest and NSURLConnection to perform an HTTP transaction with a large HTTP body, the body had to first be loaded in its entirety in to memory, then the connection could be built and the HTTP transaction could take place. This unfortunately places a large memory burden on such transactions - when uploading large images or files, for instance, memory usage would mushroom for the length of the transaction to hold the entire body contents.

For Tiger, we have made it possible to set the body of an HTTP request as an NSInputStream, which then avoids the memory problem. When the connection is processed, the specified stream is opened and bytes are read a few at a time from the stream to be written to the HTTP server. The new methods both appear in NSURLRequest.h, and are -HTTPBodyStream (on the NSHTTPURLRequest category of NSURLRequest) and -setHTTPBodyStream: (on the NSMutableHTTPURLRequest category of NSMutableURLRequest). To make use of this feature, simply create an NSInputStream to wherever the body data resides (note that because NSInputStream is toll-free bridged to CFReadStream, you may pass a CFReadStreamRef if you prefer), then call -setHTTPBodyStream: on the relevant request.

There are a few caveats when using these methods. First and foremost, once a body stream has been set on a request, it becomes the property of that request and any ensuing connections made from that request. No one else should manipulate it any way. That includes anybody retrieving the stream from an extant NSURLRequest via the -HTTPBodyStream method. Second, the body stream and body data (as set by -setHTTPBody:) are mutually exclusive; setting one will clear the other.

NSUserDefaults (Section added since WWDC)

NSUserDefaults now saves preferences using the binary plist format, rather than XML. This makes preference files smaller, and speeds up read/write times.

Since the binary plist format is supported back to 10.2, this should work in home folders which are shared between systems running 10.4, 10.3, and 10.2. For cases where 10.1 compatibility is needed, there is a boolean default, CFPreferencesWritesXML, which will cause preferences to be saved using the XML format. As with most defaults, it can be set globally or on a per-app basis.

Should you need to rewrite a plist manually for some reason you can either convert it using the plutil command line tool or just open it up and edit it using the Property List Editor application.

Locale support in C library (Section added since WWDC)

In Panther, C library functions which do string manipulations (formatting, scanning, comparisons—functions such as atof(), strtod(), scanf(), printf(), etc) started paying attention to the current POSIX locale as set by setlocale(). For instance, for a European locale, strtod() treats "," as the decimal separator when converting its string argument to a floating point value. Since this change represented a potential compatibility problem for applications, any CoreFoundation (and higher) application had the locale hardwired back to the "C" locale to preserve the previous behavior for these functions.

In Tiger, this compatibility behavior holds true for AppKit-based applications that are linked on Panther or before. Other applications get the new behavior for the C library functions. However, since very few applications ever set the POSIX locale value to anything other than the default, the change in behavior is not likely to be a problem. Changing the user's language in Mac OS X does not automatically cause the POSIX locale to be changed in applications.

Where the new behavior might be more of an issue is frameworks and libraries that are loaded into arbitrary applications or executables; if an executable has called setlocale() to change the current locale in an app, any string manipulation calls from an unsuspecting library or framework might generate unexpected results.

As a solution, functions are being added to the C library in Tiger to enable the caller to explicitly specify the locale for string manipulations. Although the exact solution isn't finalized at the time of this writing, the functions are likely to take the form strtod_l(), that is, the original function name with an "_l" suffix. In addition, a new function uselocale() will likely be added, to enable changing the locale for the current thread only. This will enable clients to set and reset the locale around a bunch of calls in a thread-safe manner.

64-bit Note (Section added since WWDC)

The Foundation framework is not available for use in 64-bit processes in Mac OS X 10.4.

API Deprecations (Section added since WWDC)

The NSArchiver and NSUnarchiver classes and other APIs in NSArchiver.h are not deprecated in this release, but may be in the next release.

The NSCalendarDate class, NSDate categories, and other APIs in NSCalendarDate.h are not deprecated in this release, but may be in the next release.

The NSDecimalNumber and NSDecimalNumberHandler classes, the NSDecimalNumberBehaviors protocol, and other APIs in NSDecimalNumber.h are not deprecated in this release, but may be in the next release. The NSDecimal struct and functions in NSDecimal.h are not deprecated in this release, but may be in the next release. Native long double support in NSNumber in the next release will replace most of the need for NSDecimal and NSDecimalNumber (and be much much faster).

The date/time formatting "locale" constants in NSUserDefaults are not deprecated in this release, but may be in the next release: NSAMPMDesignation, NSDateFormatString, NSDateTimeOrdering, NSEarlierTimeDesignations, NSHourNameDesignations, NSLaterTimeDesignations, NSMonthNameArray, NSNextDayDesignations, NSNextNextDayDesignations, NSPriorDayDesignations, NSShortDateFormatString, NSShortMonthNameArray, NSShortTimeDateFormatString, NSShortWeekDayNameArray, NSThisDayDesignations, NSTimeDateFormatString, NSTimeFormatString, NSWeekDayNameArray, NSYearMonthWeekDesignations.

The number formatting "locale" constants in NSUserDefaults are not deprecated in this release, but may be in the next release: NSCurrencySymbol, NSDecimalSeparator, NSThousandsSeparator, NSDecimalDigits, NSInternationalCurrencyString, NSPositiveCurrencyFormatString, NSNegativeCurrencyFormatString.

If deprecated, API remains (for a while at least), but does not get any bug fixes, performance enhancements, or other enhancements. Deprecated API constants may not have any values or may not be used by the framework implementation in future versions. Also, deprecated API may not be available in any future 64-bit version of the Foundation framework.

Notes specific to Mac OS X 10.3

NSInputStream and NSOutputStream

Two new classes have been added to Foundation: NSInputStream and NSOutputStream, toll-free bridged to CFReadStream and CFWriteStream. Only subclasses of NSInputStream and NSOutputStream will be bridged.

Some things to be careful of:

CFReadStreamOpen() and CFWriteStreamOpen() when passed an instance of their respective bridged types will return TRUE regardless of state (-[NSStream open] is a void return). Calling CF{Read,Write}StreamSetClient() on an NS{Input,Output}Stream will work, but you may not have both a delegate and a client.

NSError

NSError is a new class which helps encapsulate richer and more extensible error information than just an error code or error string. Core attributes of an NSError are an error domain (represented by a name), and a domain-specific error code. Several well-known domains are defined out of the box, corresponding to Mach, POSIX, and OSStatus error domains. In addition, NSError allows an arbitrary userInfo dictionary to be specified, and provides the means to return a human-readable description for the error.

NSError is not abstract and can be used as-is, but subsystems wanting to do interesting things might want to subclass. One reason to subclass is to provide a mechanism to create better localized error strings; in many cases this can be done as a private subclass which overrides localizedDescription. We do not expect to see a proliferation of NSError subclasses.

In APIs, one recommended usage for this class is to appear as an by-reference (and hence "optional") argument in calls where it is useful to provide more info on errors. The presence of an error will usually still be indicated by other means, such as the simpler NO or nil return.

We do not expect NSError arguments to appear on many APIs; just a select few.

NSXMLParser

This is a new class for XML parsing. It requires the presence of a delegate object to function.

Note that at present namespacing is broken due to a bug in the underlying implementation. The validation error mentioned in the delegate interface will not be called in this version of the parser.

NSNotificationCenter

NSNotificationCenter has had a major overhaul since 10.2. Several bugs have been fixed. There may be small minor variations in behavior, such as different orders of posting notifications to the various observers, but you should not have been depending on one observer receiving a notification before another anyway. There will also be variations in the performance profile of the various operations, though their algorithmic performance characteristics haven't changed, so one should not assume (as with everything) that something that takes an amount of time in 10.3 will be the same speed on previous releases. Simultaneous multithreaded access to the notification center performance has increased as much as 5 times in some extreme cases.

New NSDistributedNotificationCenter API

With the addition of the Fast User Switching feature in 10.3 comes changes in the posting scope for distributed notifications. The new feature made it impossible to maintain binary compatibility for all application and application/server/helper app/whatever configurations. So the choice which seemed to present the most compatibility was made, that distributed notifications would only be posted by default to the other applications within the user's login session, and developers would have to use new API to get notifications posted outside the login session. A login session consists of the applications launched when the user logs in (including the Finder and Dock) and those launched by the user (or launched by an application the user launched). System daemons, servers launched at boot, and other things later launched by these exist in the system boot session, and are not part of the scope of the login session. Things launched via remote login to the machine exist in yet other sessions of their own. A notification posted by the old API does not cross a session boundary, except as noted below. Thus, a distributed notification posted by an application the user runs will not be seen by boot-time servers. Applications using distributed notifications, which is an anonymous posting service, as a substitute for peer-to-peer communication between and application and a server will be the most impacted by this.

A new method has been added to NSDistributedNotificationCenter:

```
- (void)postNotificationName:(NSString *)name
        object:(NSString *)object
        userInfo:(NSDictionary *)userInfo
        options:(unsigned)options;
```

This method allows a program to post a notification to processes in all sessions, when the NSNotificationPostToAllSessions option flag is used. The ordinary filtering mechanisms still apply, in that only processes which have registered to receive the named notification will still do so.

NSIndexSet

NSIndexSet and NSMutableIndexSet are new classes which efficiently and compactly represent a set of unsigned integers. In Panther, these are used in new NSTableView and NSArrayController APIs to represent selected indexes and such.

NSSortDescriptor

The NSSortDescriptor class represents criterion for sorting objects by key. Each instance has a property key, a direction to indicate whether the order should be ascending or descending, and a selector.

NSArray and NSMutableArray also have new methods to sort using sort descriptors:

```
@interface NSArray (NSSortDescriptorSorting)
- (NSArray *)sortedArrayUsingDescriptors:(NSArray *)sortDescriptors;
@end

@interface NSMutableArray (NSSortDescriptorSorting)
- (void)sortUsingDescriptors:(NSArray *)sortDescriptors;
@end
```

In addition, NSTableView has been enhanced with the ability to sort based on an array of sort descriptors.

NSValueTransformer

NSValueTransformer is a new class to perform automatic conversion of data types, very useful in the context of bindings technology. For instance, archived colors coming from user defaults as NSData instances can be converted automatically into NSColors using the NSUnarchiveFromDataTransformerName value

transformer.

URL Classes

Foundation now contains the following classes for richer URL support:

NSURLAuthenticationChallenge.h
NSURLCache.h
NSURLConnection.h
NSURLCredential.h
NSURLCredentialStorage.h
NSURLDownload.h
NSURLProtectionSpace.h
NSURLProtocol.h
NSURLRequest.h
NSURLResponse.h
NSHTTPCookie.h
NSHTTPCookieStorage.h

Please refer to documentation for more info on these classes.

Key Value Coding

Key value coding has been enhanced to support the controller/bindings technology described in the AppKit release notes. Please refer to <Foundation/NSKeyValueCoding.h> and <Foundation/NSKeyValueObserving.h> for info.

Different Behavior In -valueForKey: When Sent To NSArrays and NSDictionaries

In previous versions of Foundation's Key Value Coding, -valueForKey: messages sent to NSArrays would return the result of invoking the method whose named matched the passed-in key. For example, [anArray valueForKey:@"firstObject"] would return the same result as [anArray firstObject]. The behavior of -[NSArray valueForKey:] has changed. It now returns an array that contains the results of sending -valueForKey: to each element in the array, with NSNulls corresponding to each element returning nil from -valueForKey:. It is now possible however to access inherent properties of arrays by prefixing the key with a '@'. For example, [anArray valueForKey:@"@firstObject"] now returns the same result as [anArray firstObject].

-[NSDictionary valueForKey:] has been updated in a similar way. It is now possible to access inherent properties of dictionaries by prefixing the key with a '@'. For example, [aDictionary valueForKey:@"@allKeys"] now returns the same result as [aDictionary allKeys]. As before, [aDictionary valueForKey:@"someKeyNotStartingWithAnAtSign"] is synonymous with -[aDictionary objectForKey:@"someKeyNotStartingWithAnAtSign"].

For backward binary compatibility, -[NSArray valueForKey:] and -[NSDictionary valueForKey:] behave exactly as they did in Mac OS 10.2, when being used in applications that were linked against Mac OS 10.2 frameworks.

Support for NSValue Types and More NSNumber Types in -valueForKey: and -setValue:forKey:

-valueForKey: now supports get-accessor method with return types of NSPoint, NSRange, NSRect, NSSize, long long, and unsigned long long. Likewise, the new setValue:forKey: method supports set-accessor methods whose parameters are those types. (The behavior of -takeValue:forKey: is unchanged in this regard.) The complete list of conversions supported by -valueForKey: and -setValue:forKey: is now:

bool <-> NSNumber
char <-> NSNumber
double <-> NSNumber
float <-> NSNumber
int <-> NSNumber
long <-> NSNumber
long long <-> NSNumber
short <-> NSNumber
unsigned char <-> NSNumber
unsigned int <-> NSNumber
unsigned long <-> NSNumber
unsigned long long <-> NSNumber
unsigned short <-> NSNumber
NSPoint <-> NSValue
NSRange <-> NSValue
NSRect <-> NSValue
NSSize <-> NSValue

Warning: handling of accessor methods that get or set values of C99's bool type does not work for classes compiled with gcc 3.1.

Bug Fix in Conversion of kAENo and kAEYes Enumeration Descriptors to NSNumbers

In Mac OS 10.2's version of Cocoa Scripting, incoming enumeration Apple event descriptors were always converted to boolean NSNumbers if their values were kAENo or kAEYes, instead of the unsigned long NSNumbers more appropriate for enumerations. This was a bug, affecting in particular the "saving" arguments of the standard Close and Quit commands. This bug has been fixed. kAENo and kAEYes Apple event descriptors are now only converted to boolean NSNumbers if the declared type of the command argument or class property in question is not declared to be an enumeration type.

Bug Fixes in Automatic Loading of Script Suites

In previous versions of Cocoa Scripting, compilation or execution of a valid script using NSAppleScript would fail if the script sent commands to the application containing the NSAppleScript and the application had not already received a scripting Apple event sent by another application. The workaround for this problem was to force the loading of script suites and the registration of scripting Apple events handlers by invoking +[NSScriptSuitedRegistry

sharedScriptSuiteRegistry] before using NSAppleScript. This workaround is no longer necessary. Script suites are now loaded and corresponding event handlers are registered automatically the first time any unhandled Apple event resulting from use of NSAppleScript is dispatched.

In previous versions of Cocoa Scripting, script suites were loaded for all NSBundles instantiated at script suite loading time. This was a bug that has been fixed. Now, script suites are loaded only for bundles that have actually been loaded.

Bug Fix in Registration of Standard Suite Commands

In previous versions of Cocoa Scripting, commands that were declared in .scriptSuite files to have AppleEventCodes of 'oapp', 'odoc', 'pdoc', and 'quit' were never executed in AppKit applications. Incoming events with those codes were always handled by the AppKit's built-in event handling. This bug has been fixed.

Bug Fix in Evaluation of NSSpecifierTests

There was a bug in which one of the objects involved in the evaluation an NSSpecifierTest could be sent the wrong message. In some situations, objects would be sent -scriptingIsGreaterThanOrEqualTo: instead of -scriptingIsGreaterThan:, -scriptingIsGreaterThan: instead of -scriptingIsGreaterThanOrEqualTo:, -scriptingIsLessThanOrEqualTo: instead of -scriptingIsLessThan:, or -scriptingIsLessThan: instead of -scriptingIsLessThanOrEqualTo:. This bug has been fixed.

Bug Fixes in Support for Make Commands That Have No "At" Parameter

The support for Make commands having no "at" parameter, introduced in Mac OS 10.2, had a bug in which Make commands inside nested "tell" blocks malfunctioned. For example, the script:

```
tell application "Mail"
    set theNewMessage to make new outgoing message with properties {visible:true}
    tell theNewMessage
        make to recipient with properties {name:"Nobody", address:"nobody@nowhere.com"}
    end tell
end tell
```

would fail in the second make command with an NSArgumentsWrongScriptError, despite the fact that Mail's .scriptSuite file declares the "at" parameter optional when creating to-recipients in outgoing messages. This bug has been fixed.

The support for Make commands having no "at" parameter did not interoperate properly with the Implicitly Specified Subcontainers feature, also introduced in Mac OS 10.2. For example, the script:

```
tell application "TextEdit"
    set theFile to choose file
    tell front document
        make attachment with properties {file name:theFile}
    end tell
end tell
```

would fail in the make command with an NSArgumentsWrongScriptError, despite the fact that the AppKit framework's NSTextSuite.scriptSuite file now declares that insertion locations are not required when creating attachments in NSTextStorage containers, and TextEdit's .scriptSuite file declares the Document class' textStorage attribute to be the default subcontainer. This bug has been fixed.

More Useful Behavior of -[NSAppleEventDescriptor description]

As an aid to debugging, -[NSAppleEventDescriptor description] now returns a string that includes the output of the Apple Event Manager's AEPrintDescToHandle() function for the encapsulated AEDesc.

Accessing the Current Apple Event

There was no way in previous versions of Cocoa to easily access the Apple event currently being handled, or the corresponding reply event. New methods have been added to NSAppleEventManager:

```
- (NSAppleEventDescriptor *)currentAppleEvent;
```

If an Apple event is being handled on the current thread (i.e., a handler that was registered with -setEventHandler:andSelector:forEventClass:andEventID: is being messaged at this instant or -setCurrentAppleEventAndReplyEventWithSuspensionID: has just been invoked), return the descriptor for the event. Return nil otherwise. The effects of mutating or retaining the returned descriptor are undefined, though it may be copied.

```
- (NSAppleEventDescriptor *)currentReplyAppleEvent;
```

If an Apple event is being handled on the current thread (i.e., -currentAppleEvent would not return nil), return the corresponding reply event descriptor. Return nil otherwise. This descriptor, including any mutatations, will be returned to the sender of the current event when all handling of the event has been completed, if the sender has requested a reply. The effects of retaining the descriptor are undefined; it may be copied, but mutations of the copy will not be returned to the sender of the current event.

Accessing the Current Script Command

There was no way in previous versions of Cocoa to easily access the script command currently being executed. A new method has been added to NSScriptCommand:

```
+ (NSScriptCommand *)currentCommand;
```

If a command is being executed in the current thread by Cocoa Scripting's built-in Apple event handling (i.e., an instance of NSScriptCommand is handling an -executeCommand message at this instant, as the result of the dispatch of an Apple event), return the command. Return nil otherwise. -setScriptErrorNumber: and -setScriptErrorString: messages sent to the returned command object will affect the reply event sent to the sender of the event from which the command was constructed, if the sender has requested a reply.

Accessing a Script Command's Apple Event

There was no way in previous versions of Cocoa to easily access the Apple event from which a script command was constructed. A new method has been added to NSScriptCommand:

```
- (NSAppleEventDescriptor *)appleEvent;
```

If the receiver was constructed by Cocoa Scripting's built-in Apple event handling, the Apple event descriptor from which it was constructed. The effects of mutating or retaining this descriptor are undefined, though it may be copied.

Suspending and Resuming of Apple Event Handling

There was no way in previous versions of Cocoa to suspend the handling of Apple events. This means that there was no easy way to perform large amounts of processing in response to an Apple event without blocking the AppKit event loop and therefore rendering the program's user interface unresponsive. It also means that there was no way to properly spawn a separate thread for the processing associated with an Apple event, because doing so requires that the reply for the event must be sent when the processing is completed, instead of when the Apple event handler returns. New methods have been added to NSAppleEventManager to support Apple event suspending and resuming:

```
- (NSAppleEventManagerSuspensionID)suspendCurrentAppleEvent;
```

If an Apple event is being handled on the current thread (i.e., -currentAppleEvent would not return nil), suspend the handling of the event, returning an ID that must be used to resume the handling of the event. Return zero otherwise. The suspended event will no longer be the current event after this method has returned.

```
- (NSAppleEventDescriptor *)appleEventForSuspensionID:(NSAppleEventManagerSuspensionID)suspensionID;
```

Given a nonzero suspension ID returned by an invocation of -suspendCurrentAppleEvent, return the descriptor for the event whose handling was suspended. The effects of mutating or retaining the returned descriptor are undefined, though it may be copied. This method may be invoked in any thread, not just the one in which the corresponding invocation of -suspendCurrentAppleEvent occurred.

```
- (NSAppleEventDescriptor *)replyAppleEventForSuspensionID:(NSAppleEventManagerSuspensionID)suspensionID;
```

Given a nonzero suspension ID returned by an invocation of -suspendCurrentAppleEvent, return the corresponding reply event descriptor. This descriptor, including any mutatations, will be returned to the sender of the suspended event when handling of the event is resumed, if the sender has requested a reply. The effects of retaining the descriptor are undefined; it may be copied, but mutations of the copy will not be returned to the sender of the suspended event. This method may be invoked in any thread, not just the one in which the corresponding invocation of -suspendCurrentAppleEvent occurred.

```
- (void)setCurrentAppleEventAndReplyEventWithSuspensionID:(NSAppleEventManagerSuspensionID)suspensionID;
```

Given a nonzero suspension ID returned by an invocation of -suspendCurrentAppleEvent, set the values that will be returned by subsequent invocations of -currentAppleEvent and -currentReplyAppleEvent to be the event whose handling was suspended and its corresponding reply event, respectively. Redundant invocations of this method will be ignored.

```
- (void)resumeWithSuspensionID:(NSAppleEventManagerSuspensionID)suspensionID;
```

Given a nonzero suspension ID returned by an invocation of -suspendCurrentAppleEvent, signal that handling of the suspended event may now continue. This may result in the immediate sending of the reply event to the sender of the suspended event, if the sender has requested a reply. If the suspension ID has been used in a previous invocation of -setCurrentAppleEventAndReplyEventWithSuspensionID: the effects of that invocation will be completely undone. Redundant invocations of this method will be ignored. Subsequent invocations of other NSAppleEventManager methods using the same suspension ID are invalid. This method may be invoked in any thread, not just the one in which the corresponding invocation of -suspendCurrentAppleEvent occurred.

Suspending and Resuming of Script Command Execution

Just as it is necessary to suspend and resume the handling of Apple events, it is necessary to suspend and resume the execution of script commands that have been constructed from Apple events. New methods have been added to NSScriptCommand to support command execution suspending and resuming:

```
- (void)suspendExecution;
```

If the receiver is being executed in the current thread by Cocoa Scripting's built-in Apple event handling (i.e., the value of [[NSAppleEventManager sharedAppleEventManager] currentAppleEvent] is identical to the value of [self event]), suspend the execution of the command. Otherwise, do nothing. A matching invocation of -resumeExecutionWithResult: must be made.

```
- (void)resumeExecutionWithResult:(id)result;
```

If a successful, unmatched, invocation -suspendExecution has been made, resume the execution of the command. Otherwise, do nothing. The result is for the segment of command execution that was suspended:
- If -suspendExecution was invoked from within a command handler of one the command's receivers, the result is considered to be the return value of the handler. Unless the command has received a -setScriptErrorNumber: message with a nonzero error number, execution of the command will continue and the command handlers of other receivers will be invoked.
- If -suspendExecution was invoked from within an override of -performDefaultImplementation the result is treated as if it were the return value of the invocation of -performDefaultImplementation.
This method may be invoked in any thread, not just the one in which the corresponding invocation of -suspendExecution occurred.

NSNetServiceBrowser

NSNetServiceBrowser now autoreleases the NSNetService objects you receive on the delegate methods. Applications linked on Jaguar but running on Panther will continue to leak, but Panther applications will have to retain these if necessary.

NSFileManager

Copying paths with NSFileManager now no longer attempts to bring the entire file into RAM to do the copy. Users and developers seeing copies fail when there was enough space on the target disk, but not enough on the swap disk should now see the copy succeed. All applications will benefit from this change.

Obsolete public headers -- source compatibility

The following headers are being removed and may cause source files which were including them to no longer compile:

- NSCompatibility.h (import <Foundation/NSCoder.h>, if necessary, instead)
- NSUtilities.h (import other headers asnecessary)

Header #include changes -- source compatibility

The following changes may cause source files to no longer compile because the source files were getting some needed declarations via indirect header includes in other headers.

- NSArchiver.h no longer imports NSHashTable.h and NSMapTable.h
- NSConnection.h no longer imports NSHashTable.h and NSMapTable.h
- NSDecimal.h no longer imports NSDictionary.h
- NSFileManager.h no longer imports NSUtilities.h (and thus indirectly, NSString.h)

Added version number constant

The following version constant has been added for projects compiling for 10.3 or later:

```
#define NSFoundationVersionNumber10_2 462.0
```

NSPropertyListSerialization and irregular real number values

The property list parser does not parse real (floating point) NaN, +infinity, and -infinity values that were previously saved in the property list; there is code to do so, but it was wrong in 10.2. This has been fixed in 10.3.

NSTimeZone and obsolescent time zones

NSTimeZone canonicalizes the names of many obsolete (compatibility) time zones to their more modern equivalents in the Olson data, so the name with which you create a NSTimeZone is not necessarily the one you will get when you ask for the time zone's name. These old time zones are also filtered from the list of known time zones. This will be generally unnoticeable to all applications.

Byte swapping functions

NSSwapInt16() and NSSwapInt32() have a bug in 10.2 and previous releases where the register constraints for the inline asm were not completely specified, and so the routines would not work properly for certain expressions as parameters.

NSHost

NSHost has gotten improvements, and should report the addresses and names of a given host or the current host, more accurately. Note the operating system has started returning IPv6 addresses for the local machine, and so has NSHost. Note also that queries to NSHost (as to the functions that it itself calls) can result in auto-dialup (even operations on the currentHost object), which is generally annoying to users when it appears to be unnececessary to them, so it is not advisable to make frivolous calls to NSHost.

NSProcessInfo -globallyUniqueString

The -globallyUniqueString method in NSProcessInfo was rewritten for 10.3. Code should not depend on the structure of this string, but code which does is at risk for misbehavior. Note that this string is not necessarily a valid string fragment which can be used as part of a path name on all filesystems (for example, it may be too long, or contain disallowed characters) so we do not recommend using this for generating temporary file names.

Subclasses of keyed archiving better supported

NSArrays, NSDictionarys, NSSets, NSDatas, NSStrings, NSCharacterSets, and NSNumbers cannot be encoded and decoded with subclasses of NSKeyedArchiver and NSKeyedUnarchiver in 10.2, but can in 10.3.

NSCalendarDate functionality to avoid

As has been true for years, the documented NSCalendarDate format specs %c, %x, and %X all have the same effect, unlike what the documentation says. When not localizing to the user's locale, %c also produces a result with the year in an unusual position. Generally these format specs should not be used. We can never fix these due to binary compatibility constraints, so developers must deal with this themselves.

NSCalendarDate and the Japanese Imperial and Thai calendars

The NSCalendarDate description and init-with-string methods now produce and parse Japanese Imperial calendar dates and Thai calendar dates, when localizing to the user's locale, and when the user has chosen either of these calendars in the International Preferences panel. These are the same as the Gregorian calendar, except for the year designation. A two-digit year designation in the Thai calendar, for example, will show up as "46" for the Gregorian year 2003 (2546 Thai). (The confusion this may induce is yet another reason why using the full year %Y specification is better.) Years in the Japanese Imperial calendar are based upon the emperor's reign. This also applies to NSDateFormatters.

Note however that the natural language parsing facilities do not understand Japanese Imperial and Thai calendar dates. NSDateFormatters configured to allow natural language parsing will not parse such dates correctly, when the user's input does not match the formatter's format string.

Localized formats data improvements

The locale data in Foundation has been cleaned up and fixed in many ways. In particular, the values of the following keys contain much better localized values:

```
NSMonthNameArray, NSShortMonthNameArray, NSShortWeekDayNameArray,
NSWeekDayNameArray, NSAMPMDesignation, NSDateFormatString,
NSShortDateFormatString, NSTimeFormatString, NSTimeDateFormatString,
NSShortTimeDateFormatString, NSDecimalDigits, NSDecimalSeparator,
NSThousandsSeparator, NSCurrencySymbol, NSInternationalCurrencyString,
NSPositiveCurrencyFormatString, NSNegativeCurrencyFormatString
```

However, with any change like this, many assumptions that developers made about the contents of the values of these user defaults keys will be broken. You should never examine or modify the values of these defaults if possible, and if you do think you need to do either of those things, you should make a big effort to be robust about it. For example, if you assume that the NSTimeFormatString does not contain a %p, and just append one, two things will happen: (1) two AM/PM designations will appear in locales which do included it, and (2) in locales which use a 24-hour clock, the time will also appear with AM/PM, when it shouldn't. Another example is assuming that the NSThousandsSeparator, for example, is exactly 1 character long, and reading the first character from the string. If you're going to assume things about the values of these default keys, you might as well just hard-code your own value in, and forget about the localization or using these keys.

Foundation also now watches for changes in the International Preferneces panel, and updates these keys' values when the user makes changes (that is, if the values are fetched again afterwards, they may be different; previously returned object values aren't changed). Applications should generally not cache the values of these keys (and other defaults in general) for long periods, but on the other hand probably do want to just fetch the value once at the beginning of an operation, so that consistent results are produced. For example, in formatting many hundreds or thousands of dates for a report or display in the UI, it is probably undesirable for the format of the results to change in the middle of the operation, if the user makes changes in the panel during the operation.

Alternate NSCalendarDate format spec

Although as yet undocumented, the %i format specifier in a date format string will produce the hour of the day of the 12-hour clock, from 0 - 11. This format specifier works in 10.2 as well. The related spec %I produces the 12-hour clock hour number from 1 - 12.

The localized date and time format string values will contain %i when the user choose the "display 12th hour as 0" option in the International Preferences panel, and the application is linked on 10.3 or later. This format spec is not put in the format strings for applications which were linked on 10.2 or earlier, as some applications which were examining the contents of these format strings were discovered to behave badly when presented with %i.

Foundation language data obsolescent

The Language data files in Foundation.framework are obsolescent. You shouldn't be accessing them directly anyway, but if you are, be warned they will be going away in the next major OS release. And they don't necessarily contain the correct values today (the value you get from NSUserDefaults for the NSTimeDateFormatString key, for example, is not necessarily the one in the file corresponding to the user's chosen locale).

Future number, date, and time formatting changes

Developers should anticipate that there will probably be changes to number and date/time formatting in the next major OS release. However, whether this will be changes in the existing NSNumberFormatter, NSDateFormatter, and NSCalendarDate classes or not remains undetermined. There are severe binary compatibility constraints on those classes, so they may simply be obsolesced and new classes be added instead, that a developer would have to switch to explicitly. Any changes are likely to make use of CFDateFormatter and CFNumberFormatter, which are themselves covers over the open source ICU library.

Some leaks fixed for 10.3 applications

In any software release, leaks are fixed (and, unfortunately, new leaks occasionally introduced). These leaks in 10.2 were discovered in 10.2 but not fixed for apps linked on 10.2 due to compatibility problems the fixes would have introduced, do not occur for applications linked on 10.3 or later. Applications inadvertantly depending on the leak may have problems. These are leaks with linkage checks.

NSHost: the result of the +hostWithName: and +hostWithAddress: methods leaks for applications linked on 10.2, but not for applications linked on 10.1 or 10.3 and later.

NSMachPorts, NSMessagePorts, and NSSocketPorts get invalidated, but the memory leaked, when the last reference goes away for applications linked on 10.2 or earlier.

NSScanner

For apps linked on Panther and later, setScanLocation: will raise an exception if the argument is beyond the end of the string. Note that there was check previously, but due to a bug it could fail to check.

scanDouble: will properly deal with a NULL argument. However, for apps that need to run on earlier systems, you should still avoid passing NULL in.

NSString

The method stringByTrimmingCharacters: would sometimes return an empty string (if the only character not to be trimmed was the last one). This has been fixed.

Turns out range bounds checking added in 10.2 in methods such as replaceCharactersInRange:withString: didn't fully work for some ranges where the location is a very large number (usually achieved as a result of going below zero --- as the location is unsigned). This has now been remedied, but the change might cause some binary incompatibilities. Just like the range checking added in 10.2, this change is also effectively only for apps built on 10.2 or later.

commonPrefixWithString:options: has been sped up a great deal and now generates no autoreleased objects as a side-effect of operation.

NSString's copy method will no longer simply retain the string if the string was created with the NoCopy:freeWhenDone: methods (such as initWithCharactersNoCopy:freeWhenDone:), passing NO for the "freeWhenDone" value. Instead it will make a true copy.

NSStrings with C-string backing stores created from C-string contents which contain invalid bytes would return 0xFFFE as the Unicode character corresponding to those bad bytes. For post-10.2 apps, it now returns 0xFFFD, which is more correct and compatible with CFString. Note that this probably does not happen in practice anyway.

For comparing numeric subparts of strings using their numeric values (so, Foo2 < Foo7 < Foo25), we now expose NSNumericSearch in NSString.h. This works

the same way kCFCompareNumerically in CFString, and it works as far back as 10.2.

We now clearly make a distinction between paragraph separators and line separators. The following methods can be used to find paragraph boundaries in a string:

```
- (void)getParagraphStart:(unsigned *)startPtr
        end:(unsigned *)parEndPtr
        contentsEnd:(unsigned *)contentsEndPtr
        forRange:(NSRange)range;

- (NSRange)paragraphRangeForRange:(NSRange)range;
```

These parallel the existing line range equivalents, but which take into account only paragraph separators and not all line separators.

The methods getLineStart:end:contentsEnd:forRange: and lineRangeForRange: now treat U0085 (NEXT LINE) as a line separator character.

The incorrect mappings in lowercaseString, uppercaseString, and capitalizedString are fixed.

NSString now has the following methods, which should make it easier to avoid the cString methods. Note that although the first method is new, it has actually been in the runtime since 10.0:

```
- (id)initWithBytes:(const void *)bytes length:(unsigned)len encoding:(NSStringEncoding)enc;
- (id)initWithBytesNoCopy:(void *)bytes length:(unsigned)len encoding:(NSStringEncoding)enc freeWhenDone:(BOOL)flag;
```

Note that as usual, the "NoCopy" variant considers the not copying aspect as a hint, and might actually copy the data provided (in which case the provided bytes buffer will be freed immediately, if freeWhenDone flag is YES).

In some cases NSString's getCString:maxLength: and getCString:maxLength:range:remainingRange: methods might raise a NSCharacterConversionException exception if the provided buffer length isn't big enough to hold the resulting string; in other cases it will just return whatever fits. This discrepancy isn't new to 10.3, so it is not a regression. In addition, because it was discovered too late to fix, we're leaving it alone.

Avoid "cString" methods in NSString

We will also take this opportunity to restate that the cString methods (initWithCString:, cString, getCString:, and friends) in NSString are deprecated and should be avoided. Instead of these methods, use methods which take explicit encoding arguments, or the UTF8 methods. One additional questionable method is -[NSString initWithContentsOfFile:], which ends up using the user's cString encoding to read the file if the encoding cannot be determined.

NSString (NSPathUtilities)

The following changes have been made to the path utilities on NSString for applications linked on Panther or later:
- [NSString stringByAppendingPathExtension:] now raises an exception when passed an argument of nil.
- [NSString pathComponents] now correctly reports trailing slashes in paths.
- [NSString stringByDeletingLastPathComponents] no longer removes the only / in "/".

NSAttributedString

For apps linked on Panther or later, addAttribute:value:range: and addAttributes:range: will now raise NSInvalidArgumentException as documented if a nil argument is provided.

NSNumber

-[NSNumber boolValue] would sometimes inappropriately return NO --- for instance, for [NSNumber numberWithInt:256] and [NSNumber numberWithFloat:0.1]. For apps linked on Panther or later, these cases have been fixed --- boolValue returns NO only when the numerical value of the NSNumber is 0; and for floats, 0.0 exactly (without rounding).

Boolean NSNumber objects and archiving

NSNumber objects created with a BOOL (+numberWithBool: or -initWithBool:) do not retain their BOOL-ness when archived with old-style archiving or sent across a Distributed Objects connection. They get converted to an integer-valued NSNumber with 0 or 1. This has not been fixed due to binary compatibility constraints, so developers must deal with this themselves.

NSValue

[NSValue hash] now looks at upto 16 bytes from the contents NSValue. It used to just look at the first byte.

In Panther, [NSValue isEqual:] no longer crashes with a nil argument.

NSCharacterSet

The +controlCharacterSet predefined character set now consists of the Unicode general character categories Cc and Cf.

A new predefined character set type +symbolCharacterSet that consists of the Unicode general character category S* is added.

The whitespaceAndNewlineCharacterSet predefined character set now includes U0085 (NEXT LINE) to follow the upcoming XML 1.1 specification.

Notes specific to Mac OS X 10.2

NSNetServices

A new pair of classes are now available in Foundation - NSNetService and NSNetServiceBrowser in NSNetServices.h.

NSNetService - publishing

NSNetService is for the publication of services you're offering, or for the resolution of services that have been discovered. An NSNetService for publication is created using

```
- (id)initWithDomain:(NSString *)domain type:(NSString *)type name:(NSString *)name port:(int)port;
```

A delegate is then assigned, and the delegate object receives messages about events that have happened. For publication the interesting ones are

```
- (void)netServiceWillPublish:(NSNetService *)sender;
- (void)netService:(NSNetService *)sender didNotPublish:(NSDictionary *)errorDict;
- (void)netServiceDidStop:(NSNetService *)sender;
```

The error dictionary has two keys, NSNetServicesErrorCode and NSNetServicesErrorDomain. In most cases the NSNetServicesErrorCode should be enough for determining what when wrong (if anything).

NSNetService - resolving

NSNetServiceBrowsers (more on these in a bit) hand you an NSNetService object representing the service it's discovered. Objects given to you in this way can be resolved for connection purposes - in order to get information about progress or errors, you'll have to set a delegate for these received objects. These NSNetServices objects are not created by you, so if you want to keep it around, you'll need to retain it.

```
- (void)resolve;
```

is the message you send to one of these NSNetService objects. Your delegate object will receive the following messages:

```
- (void)netServiceWillResolve:(NSNetService *)sender;
- (void)netServiceDidResolveAddress:(NSNetService *)sender;
- (void)netService:(NSNetService *)sender didNotResolve:(NSDictionary *)errorDict;
```

Also, if resolution is taking a while, you can cancel it, in which case your delegate will get

```
- (void)netServiceDidStop:(NSNetService *)sender;
```

Once a service has been resolved, you can call

```
- (NSArray *) addressing;
```

Which returns an NSArray containing NSDatas, each of which contains a struct sockaddr_in suitable for socket binding, etc.

NSNetServiceBrowser

NSNetServiceBrowsers are for discovering domains and services. Creating NSNetService browser objects is accomplished through the standard alloc/init sequence. Once you've successfully created the object, you assign a delegate, which will get messages regarding events on the network.

Services and domains on the network are dynamic - they come and go. Your delegate object will get messages representing these events:

```
- (void)netServiceBrowserWillSearch:(NSNetServiceBrowser *)aNetServiceBrowser;
```

is sent to your delegate when if the search can actually occur (i.e. the network stack is ready for it). If some error occurred preventing searching from starting, you'll get

```
- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser didNotSearch:(NSDictionary *)errorDict;
```

Searches for domains are touched off by sending the NSNetServiceBrowser object one of the following messages:

```
- (void)searchForAllDomains;
- (void)searchForRegistrationDomains;
```

Registration domains are domains which you yourself are able to register in. Searching for all domains yields messages to your delegate about any domain that has been discovered on the local network. Once domains have been discovered, you can search for services of a specific type within a domain. Send an NSNetService object the following message:

```
- (void)searchForServicesOfType:(NSString *)type inDomain:(NSString *)domainString;
```

When the browser finds a domain or finds a service, your delegate object will receive one of these:

```
- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser didFindDomain:(NSString *)domainString moreComing:(BOOL)mo
- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser didFindService:(NSNetService *)aNetService moreComing:(BOO
```

Similarly, when services or domains disappear, you'll get

```
- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser didRemoveDomain:(NSString *)domainString moreComing:(BOOL)m
- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser didRemoveService:(NSNetService *)aNetService moreComing:(B
```

messages. The "moreComing" flag on each of these tells you whether or not there are more network events that you should be aware of. Postponing expensive UI updates until the moreComing flag is false may be advisable.

A note about type strings

This API exposes lower level OS features that rely on multicast DNS to do their work. The type strings are based on /etc/services style service types and the transport that the service uses. So "_lpr._tcp." would represent something offering lpr-style services over TCP (connection-oriented) sockets. The first part of the string can really be anything, but IANA eventually will be offering a way to register "well-known" service names as well as ports. For now, you can pretty much pass anything - one of the examples uses "_wwdcpic._tcp."

NSFileManager

New attribute constants

NSFileImmutable allows you to determine if the file is locked or has the immutable bit set.

NSFileAppendOnly allows you to determine if the file will only allow writing at the end of the file.

Note that depending on the runlevel of the system at the time, you may or may not be able to set these with changeFileAttributes:.

NSFileCreationDate exposes the Carbon file creation date attribute, which you may get and set. At the moment, NSFileManager copy and move operations do not preserve this attribute. The Mac OS X System Overview points out various differences between the BSD and Carbon layers, and in this respect Foundation chooses to preserve BSD behavior as before.

NSFileOwnerAccountID and NSFileGroupOwnerAccountID expose the raw numeric ID for the file's owner and group owner accounts, obviating the need to translate from symbolic to numeric IDs.

Fix for -[NSUndoManager removeAllActionsWithTarget:]

Previous versions of Cocoa did not remove undo groups that no longer contained any actions as a result of sending the undo manager a -removeAllActionsWithTarget: message. As a result, useless Undo and Redo items would appear in applications' Edit menus. This problem has been fixed.

NSBundle additions

In 10.2, NSBundle now responds to the following methods:

```
- (BOOL)isLoaded;
- (NSDictionary *)localizedInfoDictionary;
- (id)objectForInfoDictionaryKey:(NSString *)key;
- (NSString *)developmentLocalization;
+ (NSArray *)preferredLocalizationsFromArray:(NSArray *)localizationsArray forPreferences:(NSArray *)preferencesArray;
```

These generally correspond to functionality that was already available in CFBundle, and is now being exposed at the NSBundle level. The objectForInfoDictionaryKey: method should usually be used preference to other means of access to the info dictionary, because it will provide a localized value if one is available for the specified key. The developmentLocalization corresponds to the kCFBundleDevelopmentRegion key in the info dictionary, and preferredLocalizationsFromArray:forPreferences: corresponds to CFBundleCopyLocalizationsForPreferences (for which see the CFBundle release notes). In addition there is a new localized string macro,

```
NSLocalizedStringWithDefaultValue
```

which does not add any new functionality, but provides an additional (genstrings-recognized) cover for localizedStringForKey:value:table:.

NSSocketPortNameServer

A functioning version of NSSocketPortNameServer has been added to the system. Names are published and resolved via NSNetServices using the default domain, which implies that lookup is possible only within hosts reachable via that domain (i.e., link-local), and in addition that names must be unique within that domain. Lookup may specify a host by name or by IP address (either IPv4 or IPv6), or as @"" or nil (i.e., localhost), or as "*" (i.e., any host within the default domain).

Note that registration may currently fail silently if the name requested is already in use within the default domain. Future versions of NSSocketPortNameServer may have additional functionality to detect errors of this sort, provide finer control over publication, etc.

Miscellaneous New APIs in Foundation

Foundation.h

The master header for Foundation now imports the new public header <Foundation/NSKeyedArchiver.h>, which declares the two new classes NSKeyedArchiver and NSKeyedUnarchiver.

The master header for Foundation now also imports the new public header <Foundation/NSPropertyList.h>, which declares the new NSPropertyListSerialization class.

The master header for Foundation now also imports the new public header <Foundation/NSNetServices.h>, which declares the new NSNetServices APIs (also called "Rendezvous").

The master header now does not import the header <Foundation/NSSerialization.h> when the MAC_OS_X_VERSION_MAX_ALLOWED macro is set to MAC_OS_X_VERSION_10_2 or later. The API in NSSerialization.h is deprecated.

NSArray

- (id)initWithArray:(NSArray *)array copyItems:(BOOL)flag;

This is a method which exists in the implementation of Foundation in 10.0 and 10.1 as well. It is simply being exposed in 10.2.

```
- (void)exchangeObjectAtIndex:(unsigned)idx1 withObjectAtIndex:(unsigned)idx2;
```

This method was improperly declared as a method of NSArray, when in fact it is and has always been a mutating method on NSMutableArray. The location of the declaration has been fixed.

NSCoder

For information on new NSCoder APIs for keyed coding, see below.

NSData

```
+ (id)dataWithBytesNoCopy:(void *)bytes length:(unsigned)length freeWhenDone:(BOOL)b;
- (id)initWithBytesNoCopy:(void *)bytes length:(unsigned)length freeWhenDone:(BOOL)b;
```

Two new methods to allow the creator of the NSData to indicate whether or not to take ownership of the given bytes. NSData will attempt to free the bytes

with free() when the data is deallocated if freeWithDone:YES is used.

```
- (void)replaceBytesInRange:(NSRange)range withBytes:(const void *)replacementBytes
                     length:(unsigned)replacementLength;
```

This method allows the modifier of the data to replace a different number of bytes than the range being replaced.

NSObject

+ (BOOL)isSubclassOfClass:(Class)aClass;

Returns true if the receiving class is a subclass of or identical to the parameter class.

+ (void)cancelPreviousPerformRequestsWithTarget:(id)aTarget;

Cancels all outstanding delayed performSelector:withObject:afterDelay: operations which have the given target object. This differs from the existing +cancelPreviousPerformRequestsWithTarget:selector:object: method in that the matching is without regard to the selector and argument of the previously scheduled perform.

NSRunLoop

- (void)cancelPerformSelectorsWithTarget:(id)target;

Cancels all outstanding ordered performs scheduled with the run loop, without regard to the selector or argument of the scheduled perform operation. The existing -cancelPerformSelector:target:argument: requires matching on the selector and argument as well as the target.

NSSerialization

The API in NSSerialization is deprecated.

NSThread

+ (double)threadPriority;
+ (BOOL)setThreadPriority:(double)priority;

Get and set the current thread's priority. The priority is specified with a floating point number from 0.0 and 1.0, where 1.0 is highest priority. The priorities in this range are mapped to the operating system's priority values.

NSTimer

```
- (id)initWithFireDate:(NSDate *)date interval:(NSTimeInterval)ti
               target:(id)t selector:(SEL)s userInfo:(id)ui repeats:(BOOL)rep;
```

A way to initialize an NSTimer without creating an autoreleased object. This is also an initializer subclasses can override for initializing their timers. Also allows the first fire date to be distinct from the subsequent repeat interval, if any (rather than a multiple of the interval into the future from the point at which the timer is created).

```
- (void)setFireDate:(NSDate *)date;
```

This method allows a timer's next fire date to be set.

C99's bool type

Support for bool has been added to various places in Foundation (NSInvocation, key-value coding, NSValue, archiving), but developers are advised to NOT use bool at this time, as it is not clear that the compiler has chosen the final size that it is going to use for bool. The size of bool did in fact change in the switch from gcc 2.95 to gcc 3.1, and then was changed back again. bool will not necessarily be the same size as BOOL, or 1 byte (currently). It's also remotely possible that BOOL may change in the farther future to be the same as bool, possibly causing a change in size of BOOL as well, if there is a compatibility-break type transition (for example, all ObjC frameworks bump their major versions).

Property List Validation

The writeToFile:atomically: methods in NSArray and NSDictionary now recursively validate that all the contained objects are property list objects before writing out the file, and return NO if all the objects are not property list objects, or the file would not be a valid property list.

Methods and Functions which take SEL Arguments Now Check for NULL

The Foundation has been inconsistent in its handling of NULL SEL arguments in the past. Foundation now raises in all functions and methods, like NSArray's -sortedArrayUsingSelector:, that take SEL arguments if the SEL argument is 0, for applications linked on or after 10.2. The exception is the conversion function NSStringFromSelector(), which returns nil for a 0 SEL (as it did prior to 10.2).

NSClassFromString() and NSStringFromClass()

These two functions now accept a nil argument, and return nil when given a nil argument.

The NSMakeAllCollectionsMutable Preference

The NSMakeAllCollectionsMutable preference has been removed from Foundation, and is no longer examined to modify the behavior of the collections.

NSZone Functionality

The NSZone implementation has been moved onto the zone functionality available from the underlying operating system, and so one can now create new distinct zones and allocate memory from them. NSZoneCreate() used to simply return the default zone always. The NSRecycleZone() still has its documented behavior of not freeing zone memory in use, and right now all of a zone's memory is considered to be in use, so this method does nothing. To destroy a zone completely, without regard for any memory blocks it contains still being in use, you must use the provided function in the operating system API. An executable must be compiled on or after 10.2 for this change to take effect.

-[NSArray removeObjectsFromIndices:numIndices:]

Prior to 10.2, this method could remove the wrong objects or cause a crash, due to a bug in the algorithm.

The Methods -respondsToSelector: & -isKindOfClass: and the Collection Classes

The private (default) concrete subclasses of NSArray, NSDictionary, and NSSet now implement the -respondsToSelector: and -isKindOfClass: methods to simulate being a subclass of the mutable collection type, or not, depending on whether the receiving instance is mutable or not, even though there is only one concrete class for each of these abstract classes.

Dictionary Descriptions and Sorted Keys

NSDictionary now sorts the keys in -description and related methods only under more strict circumstances: namely, that all the keys are NSStrings. Before, NSDictionary would sort the keys in producing the description if all keys responded to the -compare: method, but a heterogeneous array of numbers, dates, and strings, all of which respond to that method, cannot be sorted properly since objects of two different types cannot be the receiver and parameter to -compare:.

XML Property List Version

The XML property list version has been changed to 1.0, from 0.9, but there have been no changes in the format.

Coding Clarified for NSDistantObject, NSInvocation, and NSPort

The NSDistantObject, NSInvocation, and NSPort classes (and subclasses of NSPort in Foundation) only encode and decode themselves properly for coders which are NSPortCoders. This has been made more apparent by having these classes enforce this by throwing an exception when an attempt is made to encode them with any other type of coder.

-[NSCalendarDate classForCoder]

Since NSCalendarDate is a concrete subclass of an abstract class-cluster class (NSDate), it must override -classForCoder to specify that instances of NSCalendarDate should not get unarchived as NSDate instances. However, prior to 10.2, the implementation of this method would return the class object for NSCalendarDate specifically, so subclasses of NSCalendarDate were forced to override the method again themselves so they could be archived properly. The implementation of this method now returns the class of the receiving object so that subclasses do not need to override this method.

NSProcessInfo and String Encodings

NSProcessInfo now tries to interpret environment variables and command-line arguments in the user's default C string encoding if they cannot be converted to Unicode as UTF-8 strings. If neither works, the values are ignored by NSProcessInfo.

NSArchiver and Posing Classes

NSArchiver used to archive instances of classes that were being posed as with the wrong class name, so that on unarchiving, if the same posing situation were not in place with the runtime, unarchiving would fail. Now the name of the class is recorded correctly so that the class need not be posed-as in order for the instances to be unarchived.

Parsing NSDecimalNumber from Strings Changed Slightly

The initialization and class creation method in NSDecimalNumber that take a string and produce an instance have been changed slightly to not produce the value "NaN" when given the string "-0". This change also affects NSNumberFormatters.

NSNumberFormatters and Zero

It used to be that NSNumberFormatters would produce zero when given a string which matched the attributed string for zero set in the formatter, even if zero was less than the minimum allowed value in the formatter, or greater than the maximum allowed value. NSNumberFormatter now refuses to accept zero (say, input by a user in a text field) if less than the minimum or greater than the maximum. An executable must be compiled on or after 10.2 for this change to take effect.

Leaks

Various leaks in NSNumberFormatter, NSFileHandle, and elsewhere have been fixed in 10.2.

-[NSURL initFileURLWithPath:]

This method used to compute the wrong result if the current working directory had been set to a symbolic link to a directory. This could also cause failures in methods using this method indirectly, including the NSArray, NSDictionary, NSString, and NSData read & write to file & URL methods.

-[NSCalendarDate years:months:days:hours:minutes:seconds:sinceDate:]

Prior to 10.2, this method would return a garbage answer if the last parameter was nil. Now, the method explicitly requires a non-nil final parameter, and throws an exception if it isn't non-nil.

NSHost & IPv6

NSHost now supports IPv6 addresses and machines with IPv6 addresses. This means that the +hostWithAddress: now accepts IPv6 addresses in addition to IPv4 addresses, to create a host (assuming that there is a host with that IPv6 address), and the -address and -addresses methods may return IPv6 addresses. The -addresses method has always returned the list in no particular order, and that remains true, so the -address method may return either an IPv4 or IPv6 address for a machine with both.

To test programs with IPv6 addresses, a developer can set the "NSHostIncludeIPv4MappedIPv6" default to YES, and NSHost will generate IPv4-mapped IPv6 addresses from the IPv4 addresses, and include them in the list of addresses, with the IPv4 addresses for the host object. This results in a complete list of addresses for a host. An IPv6 address will be returned from the -address method in this case.

A few bugs regarding +[NSHost currentHost] producing a host object that only knew the loopback address (127.0.0.1) for machines configured with DHCP have also been fixed.

New methods for performing selectors on the main thread

Sometimes a background thread must have the main thread perform some work on its behalf. These are often UI-related things, like updating a window's displayed state. Two new methods on NSObject, declared in <Foundation/NSThread.h>, have been added in 10.2.

```
- (void)performSelectorOnMainThread:(SEL)aSelector withObject:(id)arg waitUntilDone:(BOOL)wait modes:(NSArray *)array;
- (void)performSelectorOnMainThread:(SEL)aSelector withObject:(id)arg waitUntilDone:(BOOL)wait;
```

These behave as follows:

- performs the selector on the receiving object with the given argument, in the main thread; the main thread is the one considered 'main' by the Foundation and AppKit
- the method is performed at point time later when the main thread runs the run loop (if it isn't already) in one of the specified run loop modes
- the convenience form specifies all of the common modes (as defined in the CFRunLoop API)
- if the modes array is empty or nil, this method does nothing (this matches the behavior of existing similar methods in Foundation)
- if the waitUntilDone: parameter is true, the method blocks forever or until the main thread has performed the method; the blocking does not run that thread's run loop
- the target and argument object are retained until the perform is completed
- performs done using this method on the same thread will be done in the same order, if the lists of modes are the same
- the queueing is "loose FIFO", meaning those performs which want to be performed in the current mode (when the performing takes place) will be performed in FIFO order, but this may skip over performs, from the same thread, which don't want to be performed in that mode
- there is no ordering guarantee amongst performs done on different threads, including the main thread
- there is no mechanism for cancelling or coalescing these performs; coalescing should be effected by other state
- if this method is invoked on the main thread and waitUntilDone: is NO, the perform is queued as normal and the method returns immediately
- if this method is invoked on the main thread and waitUntilDone: is YES, the perform is done immediately and the modes argument is ignored
- performs that are queued while a perform scan of the queue is in progress are not performed until the next cycle of the run loop (which, however, is triggered immediately due to the performs having been added to the queue)
- note, as with other perform APIs, if the performed method itself sets up performs of itself, without any terminating conditions (conditions under which it stops doing that), then infinite recursion and/or run loop spinning can result

NSInvocation and Structure Types Containing Pointers

NSInvocation would sometimes mistakenly free pointers inside a structure which was the return value of the invoked method prior to 10.2.

NSInvocation and char and short and long long arguments

NSInvocation would sometimes improperly save and restore signed char and signed short parameters to methods being forwarded, and the symptom was usually that the values would loose their sign. A couple bugs in forwarding methods with long long or unsigned long long arguments have also been fixed.

+[NSUserDefaults resetStandardUserDefaults] Change

The +resetStandardUserDefaults method now synchronizes the standard user defaults object before throwing it and its contents away. An executable must be compiled on or after 10.2 for this change to take effect.

-[NSSocketPort initRemoteWithTCPPort:host:] and IP Addresses

This initialization method now allows the host: parameter string to be an IPv4-style and IPv6-style address in addition to an ordinary host name. An executable must be compiled on or after 10.2 for this change to take effect.

NSString is now a lot better about raising exceptions with bad arguments

The following methods now check for proper ranges and/or for nil arguments and raise for applications linked after 10.1.x. For apps linked on 10.1.x or earlier, the previous behavior is maintained. The previous behavior was often a crash, but not always. (And this latter is why we have to maintain compatibility.)

The methods are:

```
compare: and related methods
substringWithRange: and related methods
rangeOfCharacterFromSet: and related methods
rangeOfString: and related methods (including hasSuffix:, hasPrefix:)
getLineStart:end:contentsEnd:
characterAtIndex:
getCharacters:range:
initWithString: and initWithFormat:...
stringByAppendingString: and stringByAppendingFormat:
various NSMutableString mutation methods
```

The exceptions that are raised are NSRangeException or NSInvalidArgumentException. Note that, as usual with exceptions raise from Cocoa APIs, do not write code which checks for and deals with these exceptions; instead focus on fixing these as programming errors. Note that the exception that is raised might have a generic message which isn't enough to pinpoint the source of the problem; putting a breakpoint on -[NSException raise] and looking at the stack backtrace is a good way to find out where these problems are.

If for some reason these exceptions are getting in the way of your app development (for instance a framework you depend on has this problem), you can temporarily disable these exceptions (and fall back to the 10.1.x behavior) by defining the default NSDisableStringExceptions with the value "YES". Note that this default will be removed in a future release and should not be relied upon; the bugs causing the exceptions should be found and fixed.

NSString

NSString initWithContentsOfFile: and initWithContentsOfURL: now look for the UTF8 representation of the Unicode BOM character in beginning of files, and will treat the file as encoded in UTF8 (instead of the default C string encoding) if the sequence is found.

Expect NSString cString method to be deprecated and removed from public API at some point in the near future. This method was originally introduced as a convenient way to bridge the gap between NSStrings and original c-string APIs. However, this method is also rather troublesome, as it works in the user's default encoding, which is dependent on the user's language choice, and as it's not always capable of converting arbitrary Unicode strings to 8-bit.

We recommend you go through and remove your usages of the cString method from your code, in new versions of your apps or frameworks. (Note that binary compatibility will be maintained as long as appropriate --- we don't intend to break existing binaries which use this method.)

The most common 8-bit encoding needed these days is the UTF-8 encoding, for which you can use the method UTF8String. For file system strings, use fileSystemRepresentation (declared in NSPathUtilities.h).

The following two methods have been added to NSString. They are not available for 10.1.x or earlier; you can do respondsToSelector: check if needed.

The following method trims (removes) characters from the specified set at both ends of the string. Use [NSCharacterset whitespaceCharacterSet] or [NSCharacterSet whitespaceAndNewlineCharacterSet] to remove whitespace around strings. The set argument cannot be nil. If the string is composed solely of characters from the specified set, the empty string is returned.

```
- (NSString *)stringByTrimmingCharactersInSet:(NSCharacterSet *)set;
```

The following method grows or shrinks the string by either removing characters from the end, or by appending as many occurrences of padString as necessary. The first padString that is appended will start at padIndex rather than 0.

```
- (NSString *)stringByPaddingToLength:(unsigned)newLength withString:(NSString *)padString startingAtIndex:(unsigned)padIndex;
```

Examples of usage:

```
[@"abc" stringByPaddingToLength:9 withString:@"." startingAtIndex:0]  ->  "abc......"
[@"abc" stringByPaddingToLength:2 withString:@"." startingAtIndex:0]  ->  "ab"
[@"abc" stringByPaddingToLength:9 withString:@". " startingAtIndex:1] ->  "abc . . ."
```

NSString

All comparison and letter case mapping methods now uses the new updated character database based on the Unicode version 3.2 specification.

NSString now provides methods that return normalized strings as described in the Unicode version 3.2 specification standard annex #15 "Unicode Normalization Forms." -decomposedStringWithCanonicalMapping method returns a string that is normalized into the Normalization Form D, -precomposedStringWithCanonicalMapping returns the Normalization Form C, -decomposedStringWithCompatibilityMapping returns the Normalization Form KD, and -precomposedStringWithCompatibilityMapping returns the Normalization Form KC.

NSCharacterSet

The predefined character sets instantiated via the constructor methods such as +letterCharacterSet are now based on the Unicode version 3.2 specification. The sets covers the entire character range from U0000 to U10FFFF defined by the specification. The methods that take NSRange (i.e. -characterSetWithRange:) now interpret it as a UTF-32 character range.

A new accessor method -longCharacterIsMember: that takes UTF32Char type is added to support the 32bit character storage.

A new predefined character set type instantiated via +capitalizedLetterCharacterSet is added. That character set corresponds to the Unicode character category 'Lt' as defined in the specification. Now, the predefined character set +uppercaseLetterCharacterSet consists of both the uppercase (the Unicode character category 'Lu') and the titlecase (the Unicode character category 'Lt') letters.

Two new methods are added to help applications to access NSCharacterSet more efficiently.
-isSuperSetOfSet: examines if a character set is superset of another. You could easily achieve the same result with the following code, but this method tries not to allocate memory.

NSCharacterSet *setA;

```
NSCharacterSet *setB;
BOOL result;

result = [[[setA mutableCopy] formIntersectionWithCharacterSet:setB] isEqual:setB];
```

-hasMemberInPlane: helps applications that want to scan the character set. The previous version of NSCharacterSet only covered the range from U0000 to UFFFF so you only had to check 65536 times. With the new implementation, you need to check 1114112 (= 65536 * 17) times. Since most character set are expected to be scarce outside of the Basic Multilingual Plane (Plane 0), it is inefficient to query that many times. This method returns true if the character set has at least one member in a given plane.

NSKeyedUnarchiver, NSUnarchiver and Decoded Objects' Lifetime

The -decodeObject method in NSUnarchiver returns an object whose lifetime is the same as that of the unarchiver - it is not autoreleased. When the NSUnarchiver is destroyed, all decoded objects are destroyed as well, unless they were retained otherwise. This was an unusual exception to the general reference counting and autorelease conventions.

The -decodeObject (a compatibility method) and -decodeObjectForKey: methods in NSKeyedUnarchiver return autoreleased objects, which may make the lifetime of such objects longer, or shorter, than with an NSUnarchiver. Essentially, one cannot assume a decoded object will be valid after -initWithCoder: returns, unless -initWithCoder: retains the object, which aligns the behavior of these methods with the usual returned-object conventions.

For compatibility, the NSUnarchiver behavior of -decodeValueOfObjCType:at: has not changed in NSKeyedUnarchiver - it is still an exception to general practice.

Keyed Archiving and Foundation Classes

Nearly all classes in Foundation that adopt NSCoding have been converted to do keyed archiving in addition to old-style archiving. [See also note above about classes which adopt NSCoding but only for Distributed Objects purposes.]

The New Archiving Mechanism in Cocoa

So, the Cocoa Foundation framework now has these new archiving classes NSKeyedArchiver and NSKeyedUnarchiver. What are they about, you say? They allow objects to give names to the data that the objects make persistent in an archive. The primary benefit of this is that when decoding, an object can ask for the data which it wants, when the object wants it, and can also ignore data it doesn't care about and ask for data which may not be in the archive. This lifts the restriction of the old NSArchiver and NSUnarchiver classes, where the data had to be asked for in the same order in which it was encoded, all data had to be consumed, and no testing of which pieces of data were in the archive was possible.

Why is this a benefit? For the most part, this ability to name the data makes it simpler for a class to archive and unarchive itself in such a way as to be both forwards and backwards compatible. That is, potentially, an archived object from the future can be unarchived on an ancient system, and an ancient systems' archived object can still be more easily readable on the future system.

Even if you don't need this capability in your classes today, converting today to use keyed-coding when the coder supports it will give you greater flexibility in the future, when you can make changes, and have older versions of your software still be able to decode your objects.

The new NSKeyedArchiver and NSKeyedUnarchiver classes provide the mechanism and data storage format for this, but much of the responsibility for this potential lies with the classes themselves, to adopt the new coding style and use it intelligently. That's what this document is about.

Making Classes Keyed-Codeable

For the most part, coding with a coder that supports keyed-coding operates the same as one that does not. For example, during encoding, an -encodeWithCoder: method still needs to call [super encodeWithCoder:], when necessary, to let the superclasses encode their state.

There are a few significant changes to the mechanisms of object archiving.

New Coding Methods

There are several new methods in NSCoder that a class may use in its NSCoding protocol methods.

For encoding:

```
- (void)encodeObject:(id)objv forKey:(NSString *)key;
- (void)encodeConditionalObject:(id)objv forKey:(NSString *)key;
- (void)encodeBool:(BOOL)boolv forKey:(NSString *)key;
- (void)encodeInt:(int)intv forKey:(NSString *)key;
- (void)encodeInt32:(int32_t)intv forKey:(NSString *)key;
- (void)encodeInt64:(int64_t)intv forKey:(NSString *)key;
- (void)encodeFloat:(float)realv forKey:(NSString *)key;
- (void)encodeDouble:(double)realv forKey:(NSString *)key;
- (void)encodeBytes:(const uint8_t *)bytesp length:(unsigned)lenv forKey:(NSString *)key;
```

And the parallel set of methods for decoding:

```
- (id)decodeObjectForKey:(NSString *)key;
- (BOOL)decodeBoolForKey:(NSString *)key;
- (int)decodeIntForKey:(NSString *)key;
- (int32_t)decodeInt32ForKey:(NSString *)key;
- (int64_t)decodeInt64ForKey:(NSString *)key;
```

- (float)decodeFloatForKey:(NSString *)key;
- (double)decodeDoubleForKey:(NSString *)key;
- (const uint8_t *)decodeBytesForKey:(NSString *)key returnedLength:(unsigned *)lengthp;

Generally the method used to decode a value must be for the same type as was used to encode the value.

Naming Values

Values that an object encodes to a coder can now be individually named with the new methods. The names can be any string. However, public classes, in a framework for example, which can be subclassed should prefix the name string with a prefix to avoid collisions with key names which may be used now or in the future by the subclasses of the class. A reasonable prefix is the full name of the class. Cocoa classes use the prefix "NS" on the key names, the same as the API prefi (and carefully makes sure that there are no collisions in the class hierarchy). Another possibility is to use the same string as the bundle identifier for the framework. The archiver and unarchiver use keys prefixed with "$" for internal values, and have to go to special trouble to allow user-defined keys with a "$" prefix. For best performance, you don't want the archiver and unarchiver to have to go to any trouble.

Subclasses also need to be somewhat aware of the prefix used by superclasses to avoid accidental collisions on key names. Subclasses of Cocoa classes should avoid unintentionally starting their key names with "NS". For example, don't name a key "NSString search options".

Of course, prefixes aren't the only way to "guarantee" uniqueness, but they are a simple way.

Coders Which Support Keyed-Coding

Not all NSCoder subclasses support, or will support, keyed-coding. For example, NSArchiver and NSUnarchiver. But someone may still use such a coder to encode and decode. Classes should make sure they only call the keyed-coding methods of NSCoder, on the NSCoder * parameter they receive in the NSCoding protocol methods, on encoders and decoders which support keyed-coding. A new method in NSCoder has been added to make this test:

- (BOOL)allowsKeyedCoding;

The implementation of this method in NSCoder returns NO, and classes which implement keyed-coding override this method to return YES. Classes which implement keyed-coding must also implement the other non-keyed-coding abstract methods of NSCoder.

Class Versioning

Versioning of encoded data is not longer handled through class versioning with keyed-coding. In fact, no automatic versioning is done for a class; this allows a class to at least get a look at the encoded values without the unarchiver deciding a priori that the versions fatally mismatch. A class is free to decide to encode some type of version information with its other values if it wishes, and this information can be of any type or quantity.

The Non-Keyed-Coding NSCoder Methods

Since the NSKeyedArchiver and NSKeyedUnarchiver classes subclass from NSCoder, the data encoding and decoding methods which do not take value name parameters are inherited, and are available for use. However, the use of these methods is subject to the same restrictions they have always had, primarily, (1) you must use the proper decode method for the encode method that was used, and (2) the decoding of items must be in the same order that they were encoded in. With NSKeyedArchiver and NSKeyedUnarchiver, you can intermix keyed and non-keyed methods, and on decode ask for the keyed values in any order, but the non-keyed values must be requested in the same order. If you do not do this, the unarchiver is free to give you the wrong values or throw exceptions (and lack of an exception does not mean you got the right value).

Use of the non-keyed methods within keyed coding is discouraged. Essentially, keyed coding with the non-keyed methods doesn't solve any of the problems or limitations of non-keyed coding, it's just writing the results to a different format.

Return Values for Missing Keys

When missing keyed-values are requested during decoding, defaults values are returned. These are nil, false, 0, and 0.0, depending on the return type of the decode method. During decoding, the existance of keyed-values can be tested with the NSKeyedUnarchiver method:

- (BOOL)containsValueForKey:(NSString *)key;

Type Coercions

The integer value decoding methods automatically perform coercions on the encoded data (if it was an integer which was encoded) to the requested size. For example, you can encode the number 35 as a 64-bit integer, but decode it, if you wish, as a 32-bit integer. If the integer is too large to fit in the destination type, an NSRangeException will be thrown. The two methods that encode and decode an int do so with whatever the natural size of the int type is on the platform on which the program is running.

The same applies to the floating point data methods. An encoded double which is decoded as a float will experience lost precision. A double whose magnitude is too large to be expressed in a float will result in an NSRangeException.

Data Types For Which There Is No Keyed-Coding Method

There are "forKey:" methods provided for the basic C data types, but not for aggregate types like structures and arrays, nor bitfields. In the future there may be more methods for these (so be careful how you name any category methods), but there aren't any today. What should be done with these? Here are some suggestions.

Array of Simple Types

If you are encoding an array of bytes, you can just use the provided methods to do so.

For other arithmetic types, create an NSData with the array. Encode an immutable data for slightly better performance. Note that in this case, dealing with

platform endianness issues is your responsibility, and this can be done in two general ways. The first is to convert the elements of the array (or likely rather, a temporary copy of the array) into a canonical endianness, either big or little, one at a time with the functions in Foundation/NSByteOrder.h, and give that result to the NSData as the buffer. At decode time, you have to reverse the process, converting from the big or little endian canonical form to the current host representation. The other technique is to use the array as-is, and record in a separate keyed-value (perhaps a boolean) which endianness the host was when the archive was created. During decoding, read the endian key and compare it to the endianness of the current host and swap the values only if different.

Or instead of wrapping the array in an NSData, you can use the bytes encoding and decoding methods.

You should of course not attempt to archive arrays of pointers this way.

If you wish, the simple approach of archiving each array element separately, perhaps by using key names inspired by the array syntax, like "theArray[0]", "theArray[1]", and so on. This is not a terribly efficient technique, but you can ignore endian issues.

Arrays of Objects

The simplest thing to do is to temporarily "wrap" the C array of objects in an NSArray with -initWithObjects:count: for the purposes of encoding, then get rid of the array. Since objects contain other information which has to be encoded, you can't just embed the array of pointers in an NSData; each object must be individually archived. During decoding, use -getObjects: on the retrieved array to get the objects back out into an allocated C array (of the correct size). Create an immutable array during encoding for slightly better performance.

Pointers

With one special exception, in the next paragraph, you can't encode a pointer and get back something useful at decode time. You have to encode the information that the pointer is pointing to, period. This is true in non-keyed-coding as well.

char * Pointers

Pointers to C strings can be saved by saving the characters, either as arrays of bytes, or by wrapping them with a temporary NSString, and archiving the NSString. Reverse the process during decoding. Create an immutable NSString for slightly better performance. If the characters aren't (or might not be) only 7-bit ASCII, be sure to keep in mind the character set encoding of the string when creating the NSString, and chose an appropriate creation method.

Structures

First, decide which fields of the structure you need to encode and decode to preserve an object's state. Not all fields might be required.

Under very few circumstances, and probably never, should one ever wrap a structure with an NSData and archive that. If the structure contains object or pointer fields, this isn't going to archive them correctly. You create a dependence on how the compiler decides to lay out the structure, which can change between versions of the compiler and on other factors. A compiler is not constrained to organize a structure just as you've specified it in the source code -- there may be arbitrary internal and invisible padding bytes between fields in the structure, for example, and the amount of these can change without notice and on different platforms. Plus, any fields which are multiple bytes in width aren't going to get treated correctly with respect to endianness issues. You will cause yourself all sorts of compatibility trouble. It's just a bad idea. Remember the old saying: a fool and his archive are soon parted.

The best technique is to archive the structure fields independently and chose the appropriate type of encoding/decoding method for each. The key names can be composed from the struct field names if you wish, like "theStruct.order", "theStruct.flags", etc., though this creates a slight dependency on the names of the fields in the source code, which over time may get renamed, but the archiving keys cannot if you want to maintain compatibility.

Arrays of Aggregates or Pointers

For more complex data types, you need to use a technique which combines the techniques above, or perhaps invent new ones of your own.

Bitfields

Bitfields are more trouble waiting to happen. You've carefully organized and sized the bitfields in your struct to be a multiple of the int size on the machine. Why can't you just encode that whole thing up now in operation as an int? Well, first of all, there are some requirements on compilers specified in the C standard (ISO/IEC 9899-1999, see sections 6.2.6.1, 6.2.6.2, and 6.7.2.1 for starters), but a compiler also has some freedom in how things are actually organized and which bits it chooses to store where, and what bits it may choose not to use (inter-field padding bits). So, assuming that you've actually done the math right and not accidentally created 33 bits of bitfield, and you never run into a buggy compiler, and the compiler doesn't change someday to notice that you're using one of the fields a lot and store it in a more optimal location (within the constraints), or simply change for no reason, you might get away with this. Then again, you might have later carefully reconstruct what happened in an older version and go to a lot of effort to interpret the value that was archived.

But there are also endianness issues to consider. What is the int value 2147483648 on one machine might be the value 128 on another, and loading an int into a struct with bitfields may give you a completely swapped set of 1 and 0 bits in the struct when you go to use the bitfields (the bits are swapped across all the bitfields, not within each bitfield). But, you could swap the value yourself, as discussed in the Arrays of Simple Types section above, going into or coming out of the archive.

Plus, it may not be appropriate to encode the values of all the fields - some may represent runtime state only, and not be something with which you want to create a new object.

Then there's potential compatibility issues with the sizes of the bitfields. Today you need two bits for that state. In the future, you add some features, and now you need 5 distinct states, or 3 bits. Having carefully packed your bitfields in prior archives, you now have problem. You can create a new key, and save just the new states in it, and put a compatible value in the old 2-bit bitfield for archiving (so that old versions can read your archive), or move the bitfield completely out of the old value to a new key and put some generic value in the old bitfield space (a value that will be at least somewhat satisfactory to an old -initWithCoder: method). On decode you need to make sure you ignore/mask out any bits of any bitfields no longer in use from an archived value (after you go to the effort of figuring out where the version of the compiler at that time was putting the bits).

More than likely what one would do is leave the old set of values for that bitfield alone (maybe it is a "button style", for example), and invent new separate state (e.g., "HasRoundCorners") rather than folding the feature into the set of styles. Here, the past archiving has constrained the implementation, and possibly the new API/feature.

If you really want to represent multiple values from bitfields or the like in one value, the safest thing to do (and yet, this document doesn't recommend it) is compute the integer representation yourself by packing it yourself. Decide how much space you are going to allocate to each field in the archived value (give yourself more bits if you think it possible you might want them later), initialize the value of the integer (it might be an int64_t instead of an int, even if you're starting with a 32-bit struct of bitfields, if you want to allocate more bits to any fields) to zero, and use the bitwise-or (|) operator and shifts to move the values into the integer, then encode the integer. Use bitwise-and (&) and shifts during decode to extract the values.

For example, consider these possible bits of instance variable state for a hypothetical button object:

```
struct {
    unsigned int isContinuous:1;
    unsigned int style:6;
    unsigned int isThrobbing:1;
    unsigned int hasImage:1;
    unsigned int hasText:1;
    unsigned int isBeingPressed:1;
    unsigned int isSwitch:1;
    unsigned int isRadio:1;
    unsigned int isSelected:1;
    unsigned int delegateRespondsToFooMethod:1;
    unsigned int delegateRespondsToBarMethod:1;
    unsigned int delegateRespondsToBazMethod:1;
    unsigned int windowIsMain:1;
    unsigned unused:14;
} _flags;
```

isThrobbing, isBeingPressed, and windowIsMain are definitely runtime states that should not be saved. If the delegate is encoded conditionally (which is common), the delegate state might come out of the archive during unarchiving as nil. Even if the delegate of the button is not encoded conditionally, it is not necessarily true that the delegate, if its implementation changes at all, will in the future (or past) have responded to the same set of methods. So those delegate state bits are really also runtime state bits that should be recomputed in -initWithCoder: if it gets a non-nil delegate out of the archive. hasImage and hasText can probably also be deduced at -initWithCoder: time when it successfully tries and retrieves those out of the archive. Of the remaining fields, all but style are probably truly boolean states, and will only need 1 bit. So we'll pack them into the low-order bits of the integer and put the style field above them so it can -- possibly grow (this assumes an old version of this button class wouldn't be completely confused by such an "undefined" - from its point of view - value).

```
#define CONT_SHIFT        0
#define SWITCH_SHIFT        1
#define RADIO_SHIFT        2
#define SELECTED_SHIFT    3
#define STYLE_SHIFT        4
#define STYLE_MASK        0x3f

int bits = 0;

bits |= (_flags.isContinuous << CONT_SHIFT);
bits |= (_flags.isSwitch << SWITCH_SHIFT);
bits |= (_flags.isRadio << RADIO_SHIFT);
bits |= (_flags.isSelected << SELECTED_SHIFT);
bits |= (_flags.style << STYLE_SHIFT);

[encoder encodeInt:bits forKey:@"com.mycompany.MyButton flags"];
```

At decode time, you reverse the process by shifting and masking:

```
int bits;

bits = [decoder decodeIntForKey:@"com.mycompany.MyButton flags"];
_flags.isContinuous = (bits >> CONT_SHIFT) & 0x1;
_flags.isSwitch = (bits >> SWITCH_SHIFT) & 0x1;
_flags.isRadio = (bits >> RADIO_SHIFT) & 0x1;
_flags.isSelected = (bits >> SELECTED_SHIFT) & 0x1;
_flags.style = (bits >> STYLE_SHIFT) & STYLE_MASK;
```

and then go on to set the states of the bitfields that weren't archived. A bonus of this abstracted technique is that endianness issues are not a problem - the arithmetic low-order bit of an int is the same low-order bit on another architecture, even if it is stored in memory in a different location.

Putting it together: Converting the Coding Methods

So how do you change your class to do keyed-coding? Here are our recommendations.

1. Find all occurrences of -encodeWithCoder: and -initWithCoder: methods in your classes.

2. Public classes which may have been subclassed should leave their old encoding and decoding methods alone. NSArchiver and NSUnarchiver still exist to allow applications to write and read old-style archives. If other developers may have used your class, or subclassed it, you should keep the old encoding and decoding code, and check the coder to see which code path you should execute.

3. Reassess what state of the object is being encoded. If the old code was archiving stuff that is no longer needed, just for the sake of compatibility, now is a good time to stop doing that. Nominally, also, an -initWithCoder: method might become much simpler, without the need to do old class version checking.

4. Convert old methods to new methods. There may not be exact matches for the old methods in the set of new keyed methods - choose something reasonable. Decide on names for each value, and don't forget to put some sort of prefix or other "uniqueness guarantor" into the key strings. Do not forget to retain the return value of decodeObjectForKey: (which is autoreleased), when converting code which used to decode the objects with the -decodeValueOfObjCType:at: or -decodeValuesOfObjCTypes: or -decodeArrayOfObjCType:count:at: methods. Also note that the new -decodeBytesForKey:returnedLength: returns a pointer to bytes which cannot be mutated - you must make a copy if you want to change the bytes.

5. The old encoding and decoding methods should be frozen, generally, along with the class version number. Any +setClassVersion: calls should be left in, so that the class version remains where it is. Of course, one can keep modifying the old non-keyed-coding algorithms, and changing the class version, as before, with the understanding that there will still be no easy forwards compatibility for people using such a coder to create archived objects. The Cocoa classes will not be updating their non-keyed coding algorithms, even when new features and state are added - non-keyed archives will simply not save or get the new features.

6. If you find, in your conversions, a class which doesn't really need to code, or shouldn't code, but is currently implementing NSCoding, you can simply remove the NSCoding conformance from the interface declaration and remove the methods, if the class is private and not encoded or decoded by any other class. For public classes, however, it is best to leave the encoding and decoding ability there. However, you may wish instead of actually encoding or decoding to raise an exception if the coder is a keyed-coder, to prevent instances of the class from ever being encoded into a keyed-coding archive.

7. On the other hand, new classes may not want to go to the trouble of encoding themselves with a non-keyed-coder, and so may choose to raise if the coder does not allow keyed coding.

This covers the basic conversion from non-keyed-coding to keyed-coding. The rest of this document discusses various additional considerations you may want to make with regards to how your classes encode and decode themselves.

Converting Coding Methods After a Class Has Been Archived to Keyed Archives

Special attention to the conversion of a class to keyed coding is required when instances of the class may have been previously archived, using old archiving methods, to a keyed archive. In this case, a keyed archive was created but the old-style archiving methods were used to encode the objects of the class. That is, at unarchiving time (for example), after the class is converted to keyed coding, even though -allowsKeyedCoding returns YES when a keyed coder is doing the coding, the state which was saved in the archive may still have been written using the old encoding methods, and MUST BE READ using the matching decoding methods. If this may have happened, and if it is interesting to be able to read these archives in the future, special care must be taken in the converted code to handle this case.

The simplest technique is to use a version key of some sort. When (after conversion), the object encodes itself, it needs to write a special keyed-value which indicates that the object was encoding using keyed coding methods. At decoding time, if the coder allows keyed coding AND this special key exists, then -initWithCoder: knows that not only is this a keyed archive, but keyed-coding methods were also used. If the key does not exist, -initWithCoder: must still use the old decoding algorithm.

That achieves backwards compatibility with old archives, but doesn't help forwards compatibility (that is, writing an archive now that can be read by the old implementation), since at encoding time you cannot know what versions of the class may be asked to unarchive the state.

Compatibility Strategies

The principle benefit of keyed-coding is that it makes being backwards compatible and forwards compatible easier. Backwards compatibility, in this document, is when something done in the past (such as an archive was created) can be dealt with without trouble at the present time (e.g., the archive read and interpreted correctly). Forwards compatibility is the opposite: software written today will be able to deal with something which is done in the future (for example, read an archive written by a future version of the operating system).

With non-keyed-coding, the burden of compatibility usually just fell on the -initWithCoder: method. When using keyed-coding, however, both coding methods often participate.

Anticipating Changes

Anticipating changes methods means accommodating future possibilities, or leaving yourself "back doors" or "fallback positions" so that you can make changes later. Of "planning for the future" and "dealing with the past", the former is definitely more challenging. Here are some ideas and advice on going about this; you may also think up some techniques of your own.

1. Perhaps the most significant ability for forwards compatibility actually occurs in -initWithCoder: - ignoring keys you don't know about. Since you are free not to read or use all the state that is available to you in the archive for a particular object, this helps the present in allowing for future changes, as well as the present in handling what was done in the past. Future versions of the class may encode new state with new keys without disrupting the present decoding algorithm.

2. Add some version info to the encoded state. This can be any variety of information you want, not just a integer (class version) as it generally was with non-keyed coding. You may just encode a "version" integer or string with some key or in some rare cases you may want a dictionary object full of goodies. Of course, adding some version info today presumes that you also have a plan for dealing with different versions in your -initWithCoder: today. If not, changing the version info in the future will not do the present version of the class any good.

3. Allow for 64-bit integers. Someday, your class's code may run on a machine with a processor with a 64-bit native word size. This seems to be the established trend in the industry at least. Encoding what is a 32-bit integer today as a 64-bit integer doesn't help you with NSKeyedArchiver and NSKeyedUnarchiver - they will only store as much information as required, and the extra high-order zero bits you're giving to the value as you give it to the archiver are useless. On the other hand, it isn't harmful either, and writing the code to understand 64-bit today while you're making changes may allow you to avoid upgrading it in the future (as long as you deal with it properly at decode time).

With the -initWithCoder: method, it is useful to reason today about what might happen on a 64-bit architecture. If you are using an encode/decode pair of explicit bit-size (32- and 64-bit) integer methods to handle each stored value, then you're OK. However, even so you should consider whether you might want to make that 32-bit integer a 64-bit integer value on a 64-bit architecture - for example, if it is a "count" of something, you might want to have more than 2^32 of them on a 64-bit machine. If you are using the generic -encodeInt:forKey: and -decodeIntForKey: methods, these read and write whatever the native int size is on the machine (actually, whatever the size was where and when and how the version of the Foundation framework that the application is using was compiled). On a 64-bit machine, int may be 64 bits wide (or it might not; the C language is flexible in this regard). So in the future its possible that values requiring more than 32 bits to represent may be written by an -encodeInt:forKey: method. If such an archive is transported to a 32-bit machine, the -decodeIntForKey: method may be unable to represent that integer in the int return value, and have to throw an NSRangeException.

Whether or not it is useful to attempt to handle this now by always decoding such integers as 64-bit is debatable. If the integer is a "count" of something, for example, it may be physically impossible to have more than 2^32 of whatever it is on a 32-bit machine, so further attempting to unarchive the file is probably a waste of time, and an exception is reasonable. Or you might want to throw a different exception, in which case you could ask for a 64-bit value, do the bounds check yourself, and raise an exception of your choosing (you could also do this by handling the NSRangeException and translating that to a different exception which is then raised). Or it might be better to avoid the exception and return nil from the -initWithCoder: method, and that is more conventional in Cocoa, though the caller of the -decodeObjectForKey: which is unpacking an instance of your class may not like the nil any more than the exception, and might end up raising an exception of their own which is less intelligible as to the cause of the problem than the range exception might have been.

4. Allow for missing keys when possible during decoding. You may have good reason in the future not to write out a particular value. Some things of course you may absolutely require today, and in the future, if you want to be forwards compatible, you have absolutely have to continue encoding even if you don't want to. But the more things you can be flexible about today, the more flexibility you will have in the future. If the rest of the class is prepared for an instance variable you may be trying to initialize to be uninitialized by the -initWithCoder: (or other -init methods), and willing to substitute some default value where a value is required, that is a good setup for allowing a keyed value to be missing from an archive. Or, -initWithCoder: can just check for the key having been missing (say, the value of the object-typed variable is still nil after -decodeObjectForKey:), and substitute some default value immediately. Extensive changes to class logic is likely not a good idea to accommodate coding compatibility, but it is something useful to keep in mind with new classes.

5. Build in fallback handling. Fallback handling is useful when one of a set of various possible keys for a value is encoded. Let's take a class representing images, Image, as an example (this example does not necessarily reflect the actual behavior of any image class, like NSImage). This is a simple example, not necessarily the best way to do this. Suppose the Image class wishes to write out a URL for an image created from a URL, but otherwise will write out either a JPEG or GIF representation of the image into an archive, whichever of the latter is more convenient at the time. So a value for only one of the following keys will be encoded (prefixes or other uniqueness has been omitted from the key names for simplicity in this example): @"URL", @"JPEG", @"GIF". The -initWithCoder: method checks for @"URL" first, and uses if there, then checks for @"JPEG", then if necessary @"GIF". In the future it might be that none of these are easy or convenient to archive (for example, taking whatever data the Image instance does have and converting it to JPEG might be fairly expensive).

An example of fallback handling would be to allow for an additional key (or group of keys) today, like @"rawdata", that wouldn't be ever written out by the -encodeWithCoder: method today, but is understood by the -initWithCoder: if none of the other keys for this value (the image data) are present. The value of the @"rawdata" key might be defined, for example, to be an NSData * containing 32-bit RGBA pixels, and in this case there might be auxiliary keys like @"pixelshigh" and @"pixelswide" that the decoding would look for to get a minimal set of information needed to produce an Image instance from the archived information. In the future, the encoding process for an Image might write out the convenient information, whatever that is at that time, and would also have to write out the @"rawdata" and other keys to allow old decoders to read the object.

Coping With The Past

The single most critical bit of information required to unarchive an object that was archived in the past is the historical record of what state class instances have archived about themselves in the past. If since converting to keyed-coding, the class has never changed the state it saves, you don't need to cope with the past, and you can ignore this section. Otherwise, you need to know what the set of state information was, written out by the class, for every time it has changed in the past. If you don't have this, you may be able to deduce some things from existing archives and the existing implementations of the NSCoding methods.

Again, the ability to read keyed values from the archive in any order, ignore keys you don't need, and add new keys without disrupting older versions of the class is the foundation for implementing backwards compatibility with keyed-coding. For specific key names, however, once a key is used, the "definition" of its possible values and what they mean generally cannot change in the future, and keyed-coding doesn't magically solve that restriction for you.

Here are some tips on being backwards compatible.

1. Ignore keys you no longer need. You don't have to read in information you don't want, so don't bother. The decoding will be slightly faster, too.

2. Read keys in any necessary order. With keyed-coding, your encoding method can write information to an archive any any order that is convenient, and the same applies to reading values during decoding. For example, though this will probably be uncommon, it is possible to read one or more of the object's keyed-values before even calling [super initWithCoder:] (when required because the superclass also implements NSCoding). Based on the information retrieved for example, you might decide it isn't even worth continuing to the superclass initialization, deallocate self, and return nil. (This assumes that your superclasses are robust in their -dealloc methods against partially initialized objects. Using NSDeallocateObject() might be better in such a case.)

3. Create and encode new keys for new state. New keyed-values are harmless to old unarchivers, as long as it is OK for them not to be initialized with such state. So this will influence how new state and functionality is added to a class, to some extent. For example, in one version of a button class, perhaps all the buttons are square; in a new version, they can optionally be square or have rounded corners, and this is recorded with a separate "HasRoundCorners" state now in a new archive. When an old version of the class reads the archived data for the object, it won't read the new key, and will make all the buttons square (even if it were to read the new information, since this is the old version of the class, the entire rest of the implementation only knows how to do square buttons, so there would be no point).

4. New values for a state, such as an enumeration, can be saved with the pre-existing key only if the old version of the class was written to allow for such undefined - from its point of view - values and have some reasonable behavior when it ran across such a thing. If you haven't made such allowances, or the allowances you did make now seem insufficient or unacceptable, you probably have to create a whole new key for the state, and see the next point.

5. Sometimes you may have to switch to a new key. In those cases, unless the decoding process in the old version will handle a missing key appropriately, you need to keep writing some value for the old key as well. The value should be something the old class will understand, and should probably be as close a simulation of the new state as possible. For example, if the old class came in "vanilla", "chocolate", and "butter pecan" flavors, and the class now has additional "double chocolate" and "caramel" flavors, to encode a value for the old key you can map "double chocolate" to the value for "chocolate" in the old class, and may have to write "caramel" as "vanilla". Of course with the new key you write the entire new set of values, and your -initWithCoder: method should now prefer to use the new key if available.

6. Whenever you have start writing out objects' state in new ways in the class's -encodeWithCoder: method, you need to update your -initWithCoder: method to understand the new keys. Almost always, decoding should prefer the new key; that is, look for new keys first, and use the value if available, and if not then look for old keys and do whatever mapping of old values to new values is required.

7. Take advantage of hooks you implemented in the past to allow for such changes now. See the Anticipating Changes section above. Or start doing some of them (like writing out version information, for example) so that you can distinguish, in the current and future versions of the class, archives written out by the past version, the current version, and future versions.

Other Keyed-Coding Notes

Performance Considerations

The less you encode for an object, the less you have to decode, and both writing and reading archives becomes faster. Stop writing out items which are no longer pertinent to the class (but which you may have had to continue writing under non-keyed-coding).

Encoding and decoding booleans is faster and cheaper than encoding and decoding individual 1-bit bitfields as integers. However, encoding many bitfields into a single integer value can be cheaper than encoding them individually. But that can also complicate compatibility efforts later.

Don't read keys you don't need. You aren't required to read all the information from the archive for a particular object, as you were with non-keyed-coding, and it is much cheaper not to. However, unread data still contributes to the size of an archive, so stop writing it out too if you don't need to read it.

It is faster to just decode a value for a key than to check if it exists then decode it if it exists. Do a contains check only when you need to distinguish the default return value (due to non-existance) from a real value which happens to be the same as the default.

It is more interesting for decoding to be fast than for encoding to be fast. If there is some tradeoff you can make to make decoding faster but which will make encoding slower, that is usually a reasonable tradeoff to make.

Root Objects

The concept of encoding a "root object", via -encodeRootObject:, does not apply to keyed archiving. As many objects or values as desired may be encoded "at the top level" after creating an archiver and before calling -finishEncoding.

Notes specific to Mac OS X 10.1

Versioning

A declaration for the global variable NSFoundationVersionNumber has been added to NSObjCRuntime.h. This variable existed with the same type in Mac OS X 10.0.x, so no special handling should be required to use it.

There's a symbolic value for the Mac OS X 10.0 version of the Foundation:

```
#define NSFoundationVersionNumber10_0 397.4
```

In general you should compare against a version number which you know fixes the problem you are checking for, rather than the exact version of the last external release.

NSFileManager

Three new entries have been defined for use in file attribute dictionaries in NSFileManager. Their keys and valid value types are:

NSFileExtensionHidden - an NSNumber containing a boolean
NSFileHFSCreatorCode - An NSNumber containing an unsigned long
NSFileHFSTypeCode - An NSNumber containing an unsigned long

Dictionaries returned by the -[NSFileManager fileAttributesAtPath:traverseLink:] method may contain one or more of these entries.

The -[NSFileManager createFileAtPath:contents:attributes:] method will now heed all of these entries in passed-in dictionaries.
The -[NSFileManager createDirectoryAtPath:attributes:] method will now heed the NSFileExtensionHidden entry in passed-in dictionaries.
The -[NSFileManager changeFileAttributes:atPath:] method will now always heed NSFileExtensionHidden entries, and will heed NSFileHFSCreatorCode and NSFileHFSTypeCode entries when the passed-in path specifies a file.

Dictionaries returned by the -[NSDirectoryEnumerator directoryAttributes] and -[NSDirectoryEnumerator fileAttributes] methods may contain one or more of these entries.

Three corresponding methods have been added to the category on NSDictionary declared in NSFileManager.h:

```
- (BOOL)fileExtensionHidden
```

Returns the value for the NSFileExtensionHidden key, or NO if the receiver doesn't have an entry for the key.

```
- (OSType)fileHFSCreatorCode
- (OSType)fileHFSTypeCode
```

Returns the value for the NSFileHFSCreatorCode or NSFileHFSTypeCode key, respectively, or zero if the receiver doesn't have an entry for the key.

Further support for the concept of hidden file name extensions is provided by the addition of a new method:

```
- (NSString *)displayNameAtPath:(NSString *)path
```

Returns the name of the file or directory at path, in a form appropriate for presentation to the user. If there is no file or directory at path, or an error occurs, [path lastPathComponent] is returned.

NSFileManager now properly supports copy, move, and deletion of resource forks.

NSFileManager copyPath:toPath:handler: and movePath:toPath:handler will automatically copy resource forks.

NSFileManager directoryContentsAtPath: when run on UFS or NFS volumes will no longer include the files representing resource forks in the enumeration result for applications compiled on 10.1. Applications compiled on 10.0.x will include the ._ resource fork file in the enumeration. (Unless the default NSFileManagerIsResourceSensitive is set --- see below.)

NSFileManager: removeFileAtPath:handler: will delete ._ (resource fork) files on UFS and NFS for applications compiled on 10.1. Applications compiled on 10.0.x will not delete these files. (Unless the default NSFileManagerIsResourceSensitive is set --- see below.)

If you need to check if the version of NSFileManager your app is running against has these changes for resource fork support, you can compare NSFoundationVersionNumber against the symbolic value NSFoundationVersionWithFileManagerResourceForkSupport.

A default has been added to allow application compiled against 10.0 to inherit these NSFileManager resource for changes. Set the default NSFileManagerIsResourceSensitive to 1 (actually any value does this). If you need to compile your app against 10.0, but need these new features, register this default in your application's initialization. Or this default can be set in the command line, for debugging purposes, to see if some bugs in 10.0-based apps are being caused by the changes in NSFileManager.

NSFileManager now allows trailing slashes on the path arguments to the movePath:toPath:handler:, copyPath:toPath:handler:, and removeFileAtPath: methods. Previously, the algorithms would get confused in some cases.

NSInvocation Excess Retain on Returned Value

A bug in Mac OS X 10.0(.x) where NSInvocation would retain the return value of an invoked method excess times, when the return value was an object, has been fixed. This most often manifests itself as an apparent leak of objects sent over Distributed Objects.

In 10.1, apps (and other executables) linked on 10.0.x will continue to over-retain the return value from NSInvocation, for compatibility. To get the fix, projects must be re-built on 10.1.

Changes to Immutable Collections

In 10.0.x, all NSArray, NSDictionary, and NSSet objects were mutable, even if created by using NSArray, NSDictionary, or NSSet as the class to which +alloc: was sent. Historically, this used to properly create immutable collections. This has been fixed.

However, some apps may have latent bugs due to NSMutable{Array,Dictionary,Set} methods having previously worked with collections that were created immutable. Sending a mutating method to an immutable collection now causes an exception to be raised.

Users and apps can temporarily work around this change, and restore the old behavior, by setting the default "NSMakeAllCollectionsMutable" to the boolean value true in the app's domain. However, this will cause some NSLogs to be emitted when the program is run, as a reminder that the default has been turned on. This default is not meant as a permanent solution, but a temporary one, until the apps can be fixed.

Nesting alloc and init

One highly discouraged programming practice is the failure to properly nest alloc and init; that is, doing

```
// Example of bad code! Always nest alloc and init!
dict = [NSDictionary alloc];
```

```
[dict init];
```

instead of the much better

```
dict = [[NSDictionary alloc] init];
```

or also acceptable (in case you have fear of nesting)

```
dict = [NSDictionary alloc];
dict = [dict init];
```

In cases where the "init" methods might return a new object, the bad code example above will not work properly. There are several examples in the AppKit and Foundation where you can encounter this alloc/init behavior. In 10.0, the bad code might have worked in some cases for NSDictionary, NSArray, and NSSet; in 10.1, the bad code is much more likely to fail, and you might get NSInvalidArgumentException exceptions of the form:

```
-setObject:forKey: sent to an uninitialized mutable dictionary object
```

If you see this error in your 10.1 testing, be sure to fix the code to nest the alloc and init. In general be sure to do this in all you allocations, not just those where you know it's needed, as more objects might follow this paradigm in future updates.

-initWithContentsOf methods

The NS{Mutable}Array and NS{Mutable}Dictionary methods -initWithContentsOfURL: and -initWithContentsOfFile: now do not leak the read-in data when a parse error occurs. The NSMutableArray and NSMutableDictionary methods now also correctly return a mutable collection. They didn't necessarily before, but due to the previous item this wasn't apparent.

Also, in Mac OS X 10.0, embedded collections within the top-level collection were always created immutable, regardless of whether or not the top-level collection was mutable or not. Again this was partly hidden by the previous item, but resulting memory corruption could sometimes be noticed. In Mac OS 10.1, embedded collections are created immutable if an immutable top-level collection is being initialized, and are created mutable if a mutable top-level collection is being initialized.

In any case, the safest thing to do whenever you don't know whether a collection is mutable or not is to create a mutable copy before mutating it. One should NEVER look at the class of a collection to see if it is mutable or not and mutate if so. Return and parameter types tell you if a collection is meant to be mutated.

Copying Collections Clarified

None of the following discussion represents a change from prior releases (including 10.0.x).

The abstract classes have -copyWithZone: methods that create a new instance with the immutable class's -initWith<collection>: copyItems: method, passing YES for copyItems:. The -mutableCopyWithZone: methods create a new instance in the same way, but pass NO to the copyItems: parameter. For example, the abstract copying methods for NSDictionary and NSMutableDictionary look like this:

```
- (id)copyWithZone:(NSZone *)zone {
    return [[NSDictionary allocWithZone:zone] initWithDictionary:self copyItems:YES];
}
```

```
- (id)mutableCopyWithZone:(NSZone *)zone {
    return [[NSMutableDictionary allocWithZone:zone] initWithDictionary:self copyItems:NO];
}
```

In some sense, the immutable copy is a deep copy, and the mutable copy is a shallow copy. It was, however, a historical mistake to tie these two concepts together.

Subclasses are free to override these methods to provide different behavior, and in fact they have to if they want a copy of an instance of their particular subclass to also be of their particular subclass.

Code that has obtained a collection from somewhere cannot really know what will be done as far as deep/shallow copying, or if any copying at all will occur, because subclasses may define their own semantics. Using an explicit allocation with an -initWith<collection>:copyItems: method may give more predictable results. On the other hand, explicit initialization requires the class of the result to be identified in the code, both through the receiver of the +alloc: method and through the choice of init method. For example:

```
    [[NSDictionary alloc] initWithDictionary:idToBeCopied copyItems:NO]
```

explicitly names NSDictionary and the -initWithDictionary:copyItems: method. Thus, you have to know some things about the idToBeCopied in order to do this. The naming of the receiver class can be abstracted a bit if you want the same class as the class of the idToBeCopied, and don't care about the mutability of the result; use [idToBeCopied class] instead of NSDictionary as the receiver of the +alloc message.

Copying Collections

Copying semantics of the concrete subclasses of the abstract NS collection classes has been modified slightly. Now that immutable collections are again being created, a historical optimization has been restored. When copyWithZone: is sent to an immutable collection, the receiver is simply retained and returned. Also, when copyWithZone: is sent to a mutable collection, the contained objects are just retained by the new collection (shallow copying), not copied. -mutableCopyWithZone: also just retains the contained objects in the new collection, as in the abstract implementation of the method.

NSFileHandle Descriptor Leak

NSFileHandle creates the new file handle for the NSFileHandleConnectionAcceptedNotification with closeOnDealloc:YES. Previously it created the new handle in such a way that the effect was of closeOnDealloc:NO, which would cause the file descriptor to be leaked. The new NSFileHandle properly owns the new file descriptor.

NSNumberFormatter Leak

NSNumberFormatters unarchived from nib files used to leak a couple of strings. They no longer do so.

NSZombieEnabled Issues

NSZombieEnabled has no effect on classes of objects which are toll-free bridged to CoreFoundation types.

NSLog() Thread-Safety

In 10.0.x, NSLog() was only thread-safe if Cocoa had been made thread-safe, through the use of NSThread to spawn a thread. Now, NSLog() itself is much more thread-safe. But use of %@ will still invoke ObjC messages on objects, which is not safe in such a circumstances, plus the objects messaged may not themselves be thread-safe.

NSLog() Banner Changed

The time stamp in the banner of NSLog() logs has changed. It is no longer localized, and consists of numbers and punctuation only. Of course, no one should ever depend on the format of the NSLog() banners, nor be parsing them.

NSTimeZone +timeZoneWithAbbreviation:

In 10.0.x, NSTimeZone's +timeZoneWithAbbreviation: method could crash when given an abbreviation which was not an abbreviation known to NSTimeZone (those in the abbreviation dictionary). Now it no longer crashes, but returns nil as it should.

NSSelectorFromString() Allows nil

NSSelectorFromString() now allows its parameter to be nil, and returns (SEL)0 for that. This parallels the existing behavior of NSStringFromSelector(), which would allow (SEL)0 and return nil. Note that in 10.0, nil will crash, so you shouldn't use it unless your app won't run on 10.0

NSData and CFData Hash Values

In 10.0.x, NSData and CFData returned different hash codes for the same byte sequences. Now they use the same algorithm.

Copied Booleans Keep Type

Now copying an NSNumber which was created with a boolean-typed value results in an NSNumber which has a boolean-typed value. Previously, the copy would end up with type "short". This is interesting because booleans are a distinct property list type from numbers, and the true type could be lost by copying.

NSMutableArray -removeObjectsInArray: fixed

In Mac OS X 10.0.x, the removeObjectsInArray: method only compared objects for pointer equality, but should have been using object equality. This has been fixed.

NSNumberFormatter

In Mac OS X 10.0, Interface Builder would crash in NSNumberFormatter if some formats containing spaces were set on a number formatter. This has been fixed. However, NSNumberFormatter is still for just formatting numeric values, not arbitrary strings which contain numbers.

Accessing User Preferences in setuid Processes

Setuid Cocoa apps that want to access the real user's defaults should call the public method +[NSUserDefaults resetStandardUserDefaults] then call seteuid() to change to the user's uid. +resetStandardUserDefaults will flush out any changes made to the user defaults to that point, and clear all the cached state. Any previously fetched standardUserDefaults instance is then invalid at that point and must be refetched with +[NSUserDefaults standardUserDefaults] (ie, be careful about storing that value in a local variable if you're switching euids in a function).

Similarly, before reverting to run as root, +[NSUserDefaults resetStandardUserDefaults] should again be called to push out any changes and reset all state. Essentially, this is something that should be done prior to calling seteuid(). (Even if the app isn't using defaults, since AppKit does and other frameworks might.)

NSUserDefaults removeVolatileDomainForName:

NSUserDefaults removeVolatileDomainForName: crashes when the volatile domain doesn't exist. The workaround is to ask for the volatile domain and test the result for null:

```
dict = [defaults volatileDomainForName: NSArgumentDomain];
if (nil != dict) {
    [defaults removeVolatileDomainForName: NSArgumentDomain];
}
```

Unicode 3.1 surrogate support

NSCharacterSet now supports Unicode 3.1 non-BMP plane characters accessible via surrogates as in CFCharacterSet (refer to the CoreFoundation release note for the detailed discussion).

Also, -[NSString rangeOfCharacterFromSet:options:range:] and its friend methods now maps a surrogate pair to its corresponding UCS-4 character. For example, NSCharacterSet with U+10000 matches with a character sequence U+D800U+DC00.

NSString

-[NSString rangeOfComposedCharacterSequenceAtIndex:] now correctly handles Hangul conjoining Jamos.

NSURL/NSURLHandle

Support for https has been added to NSURL/NSURLHandle; you can now download https URLs as you would http URLs. Also, it is now possible to configure the headers on the outgoing HTTP request via -writeProperty:forKey:. To do this, get an NSURLHandle from the relevant NSURL and prior to calling either -resourceData or -loadInBackground, message the handle with -writeProperty:forKey:; the key should be the HTTP header you wish to set and the property value should be an NSString containing the desired unquoted value for the header, for instance:

```
NSURL *url = [NSURL URLWithString:@"http://www.apple.com/"];
NSURLHandle *handle = [url URLHandleUsingCache:NO];
NSData *pageData;
[handle writeProperty:@"MyApp/1.0" forKey:@"User-Agent"];
pageData = [handle resourceData];
```

Notes specific to Mac OS X 10.0

API Changes Since Mac OS X Public Beta Summary

New Header Files

- NSAppleEventDescriptor.h
- NSAppleEventManager.h
- NSHFSFileTypes.h
- NSScriptClassDescription.h
- NSScriptCoercionHandler.h
- NSScriptCommand.h
- NSScriptCommandDescription.h
- NSScriptExecutionContext.h
- NSScriptKeyValueCoding.h
- NSScriptObjectSpecifiers.h
- NSScriptStandardSuiteCommands.h
- NSScriptSuiteRegistry.h
- NSScriptWhoseTests.h

New Methods

- NSProcessInfo: -processIdentifier
- NSTask: -processIdentifier
- NSTask: -suspend
- NSTask: -resume
- NSFormatter: -isPartialStringValid:proposedSelectedRange:originalString:originalSelectedRange:errorDescription:

Changed Methods

The removePortForName: method in NSPortNameServer now returns BOOL (used to return void), indicating success or failure. Subclasses of NSPortNameServer must have their removePortForName: methods updated.

Toll-free Bridging

CFTimeZone and NSTimeZone are now toll-free bridged.

Property Lists

Default Property List Written Format Changed to XML

Four Foundation methods have been changed to write the property lists out in XML format rather than the old OpenStep style. These are:

- NSArray
- -writeToFile:atomically:
- -writeToURL:atomically:
- NSDictionary
- -writeToFile:atomically:
- -writeToURL:atomically:

Setting the new default "NSWriteOldStylePropertyLists" with a boolean true value will cause these four methods to write the old format, in Mac OS X.

This now makes numbers (NSNumber), booleans (NSNumber), and dates (NSDate) valid plist objects for Foundation as well as CoreFoundation, for this and

future releases of Foundation. OpenStep-style plist files are still readable with NSArray's and NSDictionary's -initWithContentsOfFile: method.

WARNING: if your app is writing out object graphs with NSNumbers or NSDates (which were not property list types) with the plist writing methods listed above, and relying on those things to be read back in as NSStrings, then if your app rewrites that object graph to file, it will be written out as XML with number and date types, and will not be readable in Mac OS X Public Beta or any previous releases of Foundation, as they could read XML-format plists but did not allow the plist to contain objects of the new types. Similarly, if you start using NSNumbers and NSDates in your property lists, and writing those out, those files will not be readable by -initWithContentsOfFile: on NSArray or NSDictionary on Public Beta or previous releases. They can be parsed with CFPropertyListCreateFromXMLData(), found in CoreFoundation's CFPropertyList.h.

Using Strings Files and Legacy Property Lists

Strings files and legacy (non-XML) property lists now use UTF-8 as the default one-byte encoding; historically, they have used a legacy NeXTStep encoding. Note that the use of high-bit characters in one-byte encoded strings files and property lists has always been discouraged; if only low-bit characters are in use, this change has no effect. Moving forward, Apple continues to recommend that legacy property lists contain only low-bit characters (0-127), and that strings files be encoded using Unicode (UCS-2).

NSString

NSMutableString methods that used to accept nil in place of empty string as argument no longer do so. Previously they would always complain, but now they complain and crash. These methods are appendString:, replaceCharactersInRange:withString: and insertString:.

There is a known problem with -[NSString smallestEncoding] and -[NSString fastestEncoding] methods. These method could return a bogus value, 0xFFFFFFFF, until +[NSString defaultCStringEncoding] class method is invoked. Under normal circumstances in full-fledged applications this is not a problem; however, tools that use Foundation and the smallestEncoding/fastestEncoding methods directly might want to invoke this workaround early on to avoid any issues.

NSURL

NSURL methods -initWithString:, +URLWithString:, -initWithString:relativeToURL:, and +URLWithString:relativeToURL: will no longer be tolerant of strings that represent malformed URLs. If the string argument does not satisfy the requirements of RFC 2396 (which defines the syntax of URLs), these methods will return nil. This was done to avoid later failures, when the URL is messaged with messages that require the URL's string to be parsed.

Note that clients using NSURL merely to wrap file system paths should not be affected by this change; whenever creating an NSURL for a file system path, use -initFileURLWithPath: or +fileURLWithPath:. These methods will properly map file system paths to their URL equivalent. You need not worry about extracting the file system path vs. extracting the URL path; the semantic of -[NSURL path] has always been to return a file system path.

In order to help clients that need to be more tolerant of bad strings, there is a CFURL function CFURLCreateStringByAddingPercentEscapes() which will repair bad URL strings that suffer from syntactically insignificant characters that need to be percent-escaped for the string to be considered a valid URL string; see the CoreFoundation release notes for more details. You will need to take advantage of the toll-free bridging of NSStrings; that is, that any valid (NSString *) is also a valid CFStringRef, and vice versa.

NSFormatter

NSFormatter now has an additional method for partial string validation. This method is an alternate for the existing method. Subclasses may choose which of them to override. There is no good reason for a subclass to override both of them. The new method allows the formatter to control the selection when it decides to alter the proposed string and it also provides access to the string as it was prior to the proposed change. The new method is:

```
- (BOOL)isPartialStringValid:(NSString **)partialStringPtr
    proposedSelectedRange:(NSRangePointer)proposedSelRangePtr
    originalString:(NSString *)origString
    originalSelectedRange:(NSRange)origSelRange
    errorDescription:(NSString **)error;
```

The new method is slightly different from the existing method in that it uses the same argument both to pass in the proposed string and to pass back out the formatter's replacement. The proposed selection rage is handled similarly.

Formatters that implement this method should return YES if the proposed change should be taken as is and should return NO if the change should be rejected or altered. When you return NO, if the partialStringPtr was not changed then the entire edit is disallowed completely. If a new string was provided in partialStringPtr, then the change is allowed but that new string is used instead of the original partialStringPtr. If you replace the partialStringPtr you may also replace the proposedSelRangePtr if you wish to alter the selection.

HFS File Type Strings

To support an environment in which the type of a file may be indicated by either a file name extension or an HFS file type, a new form of file type string has been introduced. The file type strings that are accepted or returned by many Cocoa methods may now contain an encoded HFS file type surrounded by apostrophes, like so:

```
"'<HFS Four Character File Type Code>'"
```

For example, the type of a typical Mac OS 9 text file can be specified by the string "'TEXT'".

File type strings that contain a file name extension are still acceptable in every situation in which they were acceptable before.

Several new functions, declared in <Foundation/NSHFSFileTypes.h>, have been added to help manage these new file type strings:

```
NSString *NSFileTypeForHFSTypeCode(OSType hfsFileTypeCode);
```

Given an HFS file type code, this function returns a string that encodes the file type as described above. The string will have been autoreleased.

```
OSType NSHFSTypeCodeFromFileType(NSString *fileTypeString);
```

Given a string of the sort encoded by NSFileTypeForHFSTypeCode(), this function returns the corresponding HFS file type code. It returns zero otherwise.

```
NSString *NSHFSTypeOfFile(NSString *fullFilePath);
```

Given the name of a file, this function returns a string that encodes the file's HFS file type as described above, or nil if the operation was not successful. The string will have been autoreleased.

Cocoa Scripting

Merging of Frameworks

Since Mac OS X Public Beta, the Scripting and AppKitScripting frameworks have been merged into the Foundation and AppKit frameworks, respectively.

During this merge, some classes were renamed:

- NSCoercionHandler has become NSScriptCoercionHandler
- NSObjectSpecifier has become NSScriptObjectSpecifier
- NSWhoseTest has become NSScriptWhoseTest

12 new header files have been added to Foundation:

- NSScriptObjectSpecifiers.h, containing declarations of NSScriptObjectSpecifier and its 8 Foundation-declared subclasses
- NSScriptStandardSuiteCommands.h, containing declarations for the 10 subclasses of NSScriptCommand that provide our implementation of AppleScript's Standard Suite
- NSScriptWhoseTests.h, containing declarations of NSScriptWhoseTest and its 2 Foundation-declared subclasses
- NSAppleEventDescriptor.h
- NSAppleEventManager.h
- NSScriptClassDescription.h
- NSScriptCoercionHandler.h
- NSScriptCommand.h
- NSScriptCommandDescription.h
- NSScriptExecutionContext.h
- NSScriptKeyValueCoding.h
- NSScriptSuiteRegistry.h

There are still Scripting and AppKitScripting frameworks in Mac OS X, but these are mere stubs whose libraries import Foundation and AppKit libraries and whose header files import Foundation and AppKit header files. They should not be used for new development.

Binary Incompatibility for Subclassers of NSScriptSuiteRegistry

The sizeof(NSScriptSuiteRegistry) has changed since Mac OS X Public Beta. Subclasses of NSScriptSuiteRegistry have been broken and must be recompiled.

Removed NSAppleEventManager Method

The -[NSAppleEventManager replyFromSendingEvent:withSendMode:sendPriority:timeout:] method was not functional in Mac OS X Public Beta and has been removed.

Commands with Non-Object Direct Parameters

Script commands whose direct parameters are not object specifiers are now supported. To declare such a command, create appropriate entries in the Commands dictionaries of your program's script suite and script suite terminology property lists, as with any other scripting command, specifying a subclass of NSScriptCommand as the command's CommandClass. To handle the command, override -performDefaultImplementation in the specified NSScriptCommand subclass. This method will be invoked when an instance of the command is to be executed, if the command class is not specified in any of the suite's SupportedCommands dictionaries.

To give access to a command's direct parameter, a new method, - (id)directParameter, has been added to the NSScriptCommand class.

This method returns the object that corresponds to the keyDirectObject parameter of the Apple Event from which the command derives, or nil if the Apple Event contained no keyDirectObject parameter.

Enumeration Values

Cocoa scripting now supports the use of enumeration values as command arguments and command return values.

NOTE: It is not yet possible to use enumeration values as object attributes.

So that enumerations may be declared, a script suite property list may now contain a top-level Enumerations dictionary. Each entry shall be keyed by an enumeration name, and shall have as its value a dictionary containing an "AppleEventCode" entry and an "Enumerators" entry. The value of the "AppleEventCode" entry shall be a string containing a four-letter code representing the enumeration. The value of the "Enumerators" entry shall be a dictionary. Each entry in the "Enumerators" dictionary shall be keyed by an enumerator name, and shall have as its value a string containing a four-letter code representing the enumerator. For example, here is a property list segment declaring one enumeration, the enumerators of which are the possible values for the "saving" argument of the Standard Suite's Close command:

```
Enumerations = {
    SaveOptions = {
        AppleEventCode = savo;
        Enumerators = {Ask = "ask "; No = "no  "; Yes = "yes "; };
    };
};
```

The enumeration and enumerator names ("SaveOptions," "Ask," "No," and "Yes" in the above example) are only used as keys into the script suite's corresponding terminology property list. They will never be used in a user-visible string, and they need not match any identifier in your program's source code.

The four-letter codes in enumeration and enumerator declarations ('savo', 'ask ', 'no ', and 'yes ' in the above example) are used as enumeration and enumerator IDs, respectively, when 'aete' data is created. See http://developer.apple.com/techpubs/mac/IAC/IAC-314.html.

So that enumerations may be presented to the user (e.g., in a Script Editor dictionary window), a script suite terminology property list may now also contain a top-level Enumerations dictionary. Each entry shall be keyed by an enumeration name, matching an enumeration name in the corresponding script suite property list, and shall have as its value a dictionary containing one entry per enumerator. Each entry in such a dictionary shall be keyed by an enumerator name, matching an enumerator name in the corresponding script suite property list, and shall have as its value an enumerator dictionary. Each enumerator dictionary shall contain a "Name" entry, whose value is the user-visible name of the enumerator. Each enumerator dictionary may also contain an optional "Description" entry, whose value is the user-visible description of the enumerator. For example, here is a property list segment containing the terminology information for the enumeration declared in the above example:

```
Enumerations = {
    SaveOptions = {
        Ask = {Description = "Ask the user whether or not to save the file."; Name = ask; };
        No = {Description = "Do not save the file."; Name = no; };
        Yes = {Description = "Save the file."; Name = yes; };
    };
};
```

So that enumeration values may be used as arguments in Cocoa scripting commands, a string of the form NSNumber<enumerationName> may now be used as the value of a script suite command argument dictionary's "Type" entry, where enumerationName is the key for an entry in the script suite's "Enumerations" dictionary. For example, here is a property list segment declaring the Standard Suite's Close command:

```
        Close = {
            AppleEventClassCode = core;
            AppleEventCode = clos;
            Arguments = {
                File = {AppleEventCode = kfil; Optional = YES; Type = NSString; };
                SaveOptions = {AppleEventCode = savo; Optional = YES; Type = "NSNumber<SaveOptions>"; };
            };
            CommandClass = NSCloseCommand;
            Type = "";
        };
```

During the execution of a scripting command that takes an enumeration value as an argument, the argument can be treated as an NSNumber, whose possible values include the four-letter codes that were declared as possible enumerators. For example:

```
NSNumber *saveOptionsAsNumber = [[closeCommand evaluatedArguments] objectForKey:@"SaveOptions"];
OSType saveOptions = [saveOptionsAsNumber longValue];
switch (saveOptions) {
    case 'ask ':
        // Ask the user whether or not to save the file.
        break;
    case 'no  ':
        // Don't save the file.
        break;
    case 'yes ':
        // Save the file without asking.
        break;
}
```

So that enumeration values can be returned by Cocoa scripting commands, strings of the form NSNumber<enumerationName> may now be used as the value of a script suite command dictionary's "Type" entry, where enumerationName is the key for an entry in the script suite's Enumerations dictionary. When such an enumeration type is used, the command's "ResultAppleEventCode" entry must have a value that matches the named enumeration's "AppleEventCode" entry. For example, here is a property list segment declaring the Announce command from a test application named Cocoa Scripting Exerciser, and enumeration of the values it might return:

```
Commands = {
    Announce = {
        AppleEventClassCode = cses;
        AppleEventCode = annc;
        CommandClass = CSEAnnounceCommand;
        ResultAppleEventCode = reat;
        Type = "NSNumber<AnnouncementReaction>";
    };
};
Enumerations = {
    AnnouncementReaction = {
        AppleEventCode = reat;
        Enumerators = {Bad = "bad "; Good = good; Indifferent = indi; };
    };
```

At the end of the execution of the scripting command that returns an enumeration value, an NSNumber containing one of the possible enumerators should be returned. For example:

```
// The reaction to the announcement was good.
return [NSNumber numberWithLong:'good'];
```

Boolean Values

Cocoa scripting now supports the use of boolean values as command arguments and command return values.

NOTE: It is not yet possible to use boolean values as object attributes.

So that boolean values may be used as arguments in Cocoa scripting commands, a string of the form NSNumber<Bool> may now be used as the value of a script suite command argument dictionary's "Type" entry. For example, here is a property list segment declaring the Announce command from a test application named Cocoa Scripting Exerciser, with a boolean argument named "Beep":

```
        Announce = {
            AppleEventClassCode = cses;
```

```
        AppleEventCode = annc;
        Arguments = {
            Beep = {AppleEventCode = beep; Optional = YES; Type = "NSNumber<Bool>"; };
        };
        CommandClass = CSEAnnounceCommand;
        ResultAppleEventCode = reat;
        Type = "NSNumber<AnnouncementReaction>";
    };
```

During the execution of a scripting command that takes a boolean value as an argument, the argument can be treated as an NSNumber, whose possible values are YES and NO. For example:

```
BOOL beep;
NSNumber *beepAsNumber = [[announceCommand evaluatedArguments] objectForKey:@"Beep"];
    if (beepAsNumber) {
        beep = [beepAsNumber boolValue];
    }
```

So that boolean values can be returned by Cocoa scripting commands, strings of the form NSNumber<Bool> may now be used as the value of a script suite command dictionary's "Type" entry. When such a boolean type is used, the command's "ResultAppleEventCode" entry must be bool. For example, here is a property list segment declaring the Standard Suite's Exists command:

```
    Exists = {
        AppleEventClassCode = core;
        AppleEventCode = doex;
        CommandClass = NSExistsCommand;
        ResultAppleEventCode = bool;
        Type = "NSNumber<Bool>";
    };
```

At the end of the execution of the scripting command that returns a boolean value, an NSNumber containing YES or NO should be returned. For example:

```
// The object specified in an Exists command does exist.
return [NSNumber numberWithBool:YES];
```

Notes Specific to Mac OS X Public Beta

New Map and Hash Table Enumeration Functions

Two new functions have been added to the NSHashTable and NSMapTable APIs:

- void NSEndHashTableEnumeration(NSHashEnumerator *enumerator);
- void NSEndMapTableEnumeration(NSMapEnumerator *enumerator);

These functions must be used to clean up the hash or map enumerator state, if the enumerator is not exhausted (all objects read from it). If the enumeration state is not cleaned up, memory may be leaked. When a hash or map enumerator has reported all of the objects in the "get next" function, it automatically calls the appropriate function above to clean up the enumeration state. If the enumeration state has already been cleaned up, these functions do nothing.

NXZone APIs Undefined

NSObjCRuntime.h no longer #imports the obsolete header . All headers do not include it either anymore. Thus, sources which were getting declarations for the old NXZone APIs for free by including a Foundation header will no longer get those declarations. You should convert code off of all API in the objc/zone.h header.

NSSpellServer API Moved

The header NSSpellServer.h and implementation of the NSSpellServer class has been moved from AppKit to the Foundation framework. NSSpellServer is used in the implementation of spelling servers only, not to check spelling from applications.

Suspension APIs in NSDistributedNotificationCenter Note

The -setSuspended: and -suspended methods should not be used by most applications. Applications based on the AppKit should let the AppKit manage the suspension of distributed notification delivery. Foundation-only programs may have occasional need to use these methods.

New NSBundle API

NSBundle.h has a few new methods:

- - (NSString *)pathForAuxiliaryExecutable:(NSString *)executableName;
- - (NSString *)pathForResource:(NSString *)name ofType:(NSString *)ext inDirectory:(NSString *)subpath forLocalization:(NSString *)localizationName;
- - (NSArray *)pathsForResourcesOfType:(NSString *)ext inDirectory:(NSString *)subpath forLocalization:(NSString *)localizationName;
- - (NSArray *)localizations;
- - (NSArray *)preferredLocalizations;
- + (NSArray *)preferredLocalizationsFromArray:(NSArray *)localizationsArray;

Key-Value Coding in Foundation

Some of the key-value coding and related facilities from the Enterprise Object Framework have been added to Foundation, and are now available to Cocoa developers. The NS prefix is used rather than the EO prefix, on global symbols. The new headers are NSNull.h, NSKeyValueCoding.h, and NSClassDescription.h.

Notes Specific to Mac OS X Developer Preview 4

Changes to the Objective C runtime in Mac OS X Developer Preview 4 from Preview 3

The Objective C runtime is no longer embedded within the Foundation framework. The runtime is now contained in the library /usr/lib/libobjc.dylib. However, you should continue to link with the Foundation framework rather than the runtime, if possible. A number of APIs which have alternatives within the CoreFoundation or Cocoa frameworks have also been obsolesced and even removed.

Removed Functionality

The obsolete NXBundle, NXStringTable, Storage, and StreamTable classes, along with their headers, have been removed. The function objc_getModules() has also been removed from the runtime.

The command line utility objcunique, which used to help in prebinding binaries, no longer does anything interesting.

Obsolete Functionality

These APIs have been obsolesced. These APIs should not be used for new programs, and use of these APIs should be converted to alternatives as soon as possible.

- The List and HashTable classes, and their headers
- All API in the headers objc/error.h, objc/zone.h, objc/hashtable.h, objc/hashtable2.h, and objc/maptable.h (maptable.h is not a public header so you shouldn't have been using its API anyway)
- The function objc_getClasses(); use the function objc_getClassList() instead (objc/objc-runtime.h for more information)

NXZone APIs

All APIs which took or returned NXZone * parameters have been changed to take or return void * parameters; for return values, malloc_zone_t * are now returned (which can be used interchangeably with NXZone * currently); NXZone * parameters will continue to be accepted for now, but when the NXZone API is obsolesced so will that; these functions also accept malloc_zone_t * parameters. The List and HashTable classes now ignore all zone parameters.

Cautions for the Future of the Objective C Runtime API

Projects using Objective C are cautioned to identify uses of Objective C runtime APIs, and eliminate them if possible, in the event some are obsoleted after Mac OS X ships its first release. If you can localize your dependencies, you will be impacted less by possible future new versions of the runtime.

This includes all APIs in all objc/*.h headers, except for these types and constants in objc/objc.h: Class, id, SEL, IMP, BOOL, YES, NO, Nil, nil

You should also NOT:

- Assume that sizeof(BOOL) == 1
- Assume that the type of SEL is char * and that a SEL is a C string (this is not uncommon in log messages, for example)
- Access any of the fields of the Objective C runtime structures directly, including, but not limited to, the Class (struct objc_class) structure

Use of the @selector() Objective C language expression is a tricky issue with respect to possible future language directions, since in coding an @selector() expression, a programmer is effectively coding in some knowledge of how a method declaration is mangled into a selector. This should be the purview of the compiler and the runtime alone. However, there is no alternative currently, so we must live with this for now.

Finally, you should reconsider uses of the Objective C language @defs() expression on classes which are not your own.

Other Cautionary Notes

Do NOT traverse the method lists in an Objective C class yourself; use the class_nextMethodList() function if you have to do this at all.

Do not use Objective C from within a library initialization routine if you can avoid it. The Objective C runtime is not initialized at the time library initializers are executed.

Changes to Foundation in Mac OS X Developer Preview 4 from Preview 3

NSUserDefaults

In order to avoid naming collisions, NSUserDefaults now stores an application's defaults under a file named from the main bundle's identifier; if the bundle's identifier has not been set, the defaults are stored under the application's name, as always. For the most part, this change should be transparent to users, except that defaults will have been lost the first time the app is run under the new naming scheme. For more on the bundle identifier, see /System/Documentation/Developer/ReleaseNotes/InfoPlist.html.

The defaults Command

The defaults command (/usr/bin/defaults) has changed extensively in order to support the new options available for defaults. The old syntax continues to work as expected, but there are several new options available. Execute "defaults help" to get a full listing of the new options and syntax.

NSDebug.h API

Some API in the NSDebug.h header has been removed and obsolesced:

- The NSKeepAllocationStatistics global boolean in NSDebug.h is obsolete, and set to YES now be default
- The environment variables NSKeepAllocationStatistics and NSAllocationStatisticsOutputMask are obsolete, and not used
- Several allocation event constants in NSDebug.h have been eliminated
- The NSRecordAllocationEvent() function in NSDebug.h should be considered obsolescent, but as yet no alternative has been published

Miscellaneous API Changes

- The NSNextHashEnumeratorItem() function can now be called again safely after it has already returned NULL
- The NSNextMapEnumeratorPair() function can now be called again safely after it has already returned NO
- The NSNextMapEnumeratorPair() function now accepts NULL for the second and third return-by-reference arguments, if you are not interested in those values
- There is a new class NSSocketPortNameServer for the naming of NSSocketPorts
- There are some new methods in NSBundle:
- +bundleWithIdentifier:
- -executablePath
- -privateFrameworksPath
- -sharedFrameworksPath
- -sharedSupportPath
- -builtInPlugInsPath
- -bundleIdentifier

NSUndoManager

There is a known severe bug in NSUndoManager which causes undo to be unreliable at best. It may also cause some unusual side-effects, possibly including raises, when an undo group is closed. We expect the bug to be fixed in the next release of MacOS X.

Changes in Mac OS X Developer Preview 3 from Preview 2

New API in NSUserDefaults

The methods -searchList and -setSearchList: are now fully obsoleted; calling them will do nothing. Instead, we have added the methods -addSuiteNamed: and -removeSuiteNamed:. These add or remove the domains pertaining to the named suite, and are intended for use by framework and suite developers that wish to set and retrieve preferences which pertain to potentially several applications, but which do not wish to use NSGlobalDomain for the purpose. Once a named suite has been added, its defaults are placed in NSUserDefaults' search list, and will be found in the normal fashion. Suite defaults can be written by calling -setPersistentDomain:forName:, passing the suite's name as the domain name.

Changes in Mac OS X Developer Preview 2 from Preview 1

More Distributed Objects API changes

There is a new class, NSMachPort, which is a concrete subclass of the abstract NSPort class. NSMachPort is an object wrapper for a Mach port, and is only available on Mach.

There are two other new NSPort concrete subclasses, NSMessagePort and NSSocketPort, which can be used as endpoints for DO connections (or raw messaging). A subclass of NSPort represents a particular flavor of data transport from one process to another. Note that instances of port subclasses cannot be mixed on a particular communication channel. For example, a client cannot connect to a server using NSMessagePort if the server only supports connections made with NSSocketPort. Also, you cannot transfer instances of NSMessagePort in a message to another process over a channel which is using NSSocketPorts as its endpoints; you can only pass NSSocketPorts on such a channel. These restrictions apply to any subclasses of NSPort, not just NSMessagePort and NSSocketPort. However, you are free to create other connections to a server using other subclasses of NSPort (assuming the server supports multiple transports) and send instances of that other subclass on that channel.

NSMessagePort allows for local (on the same machine) communication only. NSSocketPort allows for both local and remote communication, but may be more expensive than NSMessagePort for the local case; there is also no name registry service for NSSocketPorts -- clients and servers must agree on the TCP port numbers to use beforehand.

The following NSPort methods are obsolete. If you want to create an NSPort wrapping a Mach port, use the new NSMachPort class.

- +portWithMachPort:
- -initWithMachPort:
- -machPort

The following NSPortMessage method is obsolete. Implement the -handlePortMessage: delegate method instead of -handleMachMessage: (which now only applies to NSMachPort instances) if you want messages sent in the default internal packaging format unpacked.

- - initWithMachMessage:

Subclasses of NSPort must now implement these two new methods to setup monitoring of the port when added to a run loop, and stop monitoring if needed when removed.

- - (void)scheduleInRunLoop:(NSRunLoop *)runLoop forMode:(NSString *)mode;
- - (void)removeFromRunLoop:(NSRunLoop *)runLoop forMode:(NSString *)mode;

There is a new concrete subclass, NSMessagePortNameServer, of the NSPortNameServer class. NSMessagePortNameServer takes and returns instances of the NSMessagePort class.

NSUserDefaults

NSUserDefaults now supports saving and retrieving NSNumber and NSDates, as well as the other traditional classes. One side-effect of this which may be unexpected is that -objectForKey: may now return objects of the new classes. This may cause older code that makes assumptions about the class of the

returned object to fail, or even crash. We recommend that clients verify that their code correctly checks the class of the returned object before messaging, as any of the various property list classes may be returned.

In addition, the following two methods in the NSUserDefaults class are being obsolesced:

- -searchList
- -setSearchList

This is in preparation for exporting the full feature set of CFPreferences through NSUserDefaults. Calling these methods now generates a console log. For now, the methods still function as well as they can, but their behavior is not guaranteed, and clients should move off them as soon as possible.

Other Changes

- Some API has been removed from NSDebug.h.
- A new method on NSRunLoop, -getCFRunLoop, returns the underlying CFRunLoop.

Changes between Mac OS X Server Release 1 and Mac OS X Developer Preview 1

"Toll-free" Bridging with CF Types

Foundation now links with the new CoreFoundation framework, and several classes in Foundation are now implemented in terms of the new APIs in CoreFoundation. Some classes are also "toll-free bridged" with their CoreFoundation counterparts. What this means is that the CoreFoundation type is interchangeable in function or method calls with the bridged Foundation object. For example, CFArray and NSArray are toll-free bridged, which means than in API where you see an NSArray * parameter, you can pass in a CFArrayRef, and in API where you see a CFArrayRef parameter, you can pass in an NSArray instance. This applies to all libraries, not just CoreFoundation and Foundation. It also applies to developers' concrete subclasses of abstract types like NSArray.

However, not all Foundation classes have counterparts in CoreFoundation, and even of those which seem to have counterparts, not all are (or will ever be) toll-free bridged. For example, there is a CFRunLoop type and an NSRunLoop class. These are not toll-free bridged, however, and are not interchangeable. NSRunLoop does however use CFRunLoop in its implementation.

Here is a list of types and classes which are toll-free bridged in this release. More may be bridged in the future.

- CFArray & NSArray
- CFCharacterSet & NSCharacterSet
- CFData & NSData
- CFDate & NSDate
- CFDictionary & NSDictionary
- CFRunLoopTimer & NSTimer
- CFSet & NSSet
- CFString & NSString
- CFURL & NSURL

Cross-host Distributed Objects Unavailable

The Mach Name Server is not present in Mac OS X. Distributed Objects relied on the Name Server to provide cross-host transport for Mach messaging (and Mach ports) and a string-name-to-port mapping service. The result is that there is no cross-host transport for Distributed Objects provided in Mac OS X Developer Preview 1, though Apple is considering adding a built-in cross-host transport facility in Foundation in the future.

In possible anticipation of a new transport facility and to assist developers writing their own D.O. transports or naming services, Distributed Objects has been modified somewhat in this release. The new class and list of new and obsolete methods can be seen in the following sections.

New Port Name Server Class

There is a new class NSMachBootstrapServer, a subclass of NSPortNameServer, which provides access to the Mach Bootstrap Server. This class is only available on Mac OS X. Note that the Bootstrap Server does not support port checkout capability (the port must be destroyed to remove a checked-in entry), and the inherited removePortForName: method does nothing.

In Mac OS X, NSMachBootstrapServer is the default port name server for D.O. The default used to be a private class which used the Mach Name Server.

Added Methods

- NSConnection: +connectionWithRegisteredName:host:usingNameServer:
- NSConnection: +rootProxyForConnectionWithRegisteredName:host:usingNameServer:
- NSConnection: -registerName:withNameServer:

These methods allow a port name server instance to be specified for the operation. The old methods of similar name, without the usingNameServer: parameter, continue to use the default port name server, returned by the systemDefaultPortNameServer method, below.

- NSPortNameServer: +systemDefaultPortNameServer

This method returns the hard-wired default port name server for Distributed Objects. In Mac OS X, this is an instance of NSMachBootstrapServer.

- NSPortCoder: -initWithReceivePort:sendPort:components:

This is simply the init method for the pre-existing class creation method which autoreleases its return value.

- NSPort: -sendBeforeDate:msgid:components:from:reserved:

The method which an NSPortMessage uses to send itself on a port. There is a method of similar name, which does not allow the msgh_id of a Mach message to be specified. If you were overriding that abstract method on NSPort, you should now override this one as well. The msgid parameter may not be used by some NSPort subclasses.

Deleted Methods

- NSPortNameServer: +defaultPortNameServer

Other Changes

- Foundation.h now imports CoreFoundation.h instead of the 15 ANSI C headers (which are imported by CoreFoundation.h).
- Foundation-based programs linked on Mac OS X Developer Preview 1 will not run on Mac OS X Server Release 1.
- The allocation statistics API in NSDebug.h is currently disabled and may change or be removed from Foundation in the future.