AppKit Release Notes for macOS 10.13

Document Generated: 2017-08-24 16:34:04 -0700

Developer Release Notes
Cocoa Application Framework (macOS 10.13)

The Cocoa Application Framework (also referred to as the Application Kit, or AppKit) is one of the core Cocoa frameworks. It provides functionality and associated APIs for applications, including objects for graphical user interfaces (GUIs), event-handling mechanisms, application services, and drawing and image composition facilities.

Some of the major topics covered in this document:

These are the AppKit release notes for macOS 10.13.

Release notes for AppKit in macOS 10.12 and earlier releases can be found here.

Note that even though some APIs and descriptions refer to the APIs in Objective-C, and some others in Swift, unless otherwise specified all AppKit APIs are available in both Objective-C and Swift.

Marking updated APIs in headers

New APIs in headers are marked with decorations that include references to "10_13":

```
 NS_AVAILABLE_MAC(10_13), NS_AVAILABLE(10_13, <iOS Release>), NS_CLASS_AVAILABLE(10_13, <iOS Release>), NS_ENUM_AVAILABLE(10_13)
```

Deprecated APIs are marked with:

```
 NS_DEPRECATED_MAC(<Release when introduced>, 10_13) or  NS_DEPRECATED_MAC(<Release when introduced>, 10_13, "Suggested alternative")
```

Runtime Version Check

There are several ways to check for new features provided by the Cocoa frameworks at runtime. One is to look for a given new class or method dynamically, and not use it if it is not there.

In Swift, you can use #available, and as of Xcode 9, you can use @available from Objective-C to check the system version at runtime.

Swift

```
if #available(macOS 10.13, *) {
      // macOS 10.13 or later code path
} else {
      // code for earlier than 10.13
}
```

Objective-C

```
if (@available(macOS 10.13, *)) {
      // macOS 10.13 or later code path
} else {
      // code for earlier than 10.13
}
```

You can also use the global variable NSAppKitVersionNumber from Objective-C(or, in Foundation, NSFoundationVersionNumber):

```
typedef double NSAppKitVersion NS_TYPED_EXTENSIBLE_ENUM;
. . .
static const NSAppKitVersion NSAppKitVersionNumber10_7 = 1138;
static const NSAppKitVersion NSAppKitVersionNumber10_8 = 1187;
static const NSAppKitVersion NSAppKitVersionNumber10_9 = 1265;
static const NSAppKitVersion NSAppKitVersionNumber10_10 = 1343;
static const NSAppKitVersion NSAppKitVersionNumber10_10_2 = 1344;
static const NSAppKitVersion NSAppKitVersionNumber10_10_3 = 1347;
static const NSAppKitVersion NSAppKitVersionNumber10_10_4 = 1348;
static const NSAppKitVersion NSAppKitVersionNumber10_10_5 = 1348;
static const NSAppKitVersion NSAppKitVersionNumber10_10_Max = 1349;
static const NSAppKitVersion NSAppKitVersionNumber10_11 = 1404;
static const NSAppKitVersion NSAppKitVersionNumber10_12 = 1504;
static const NSAppKitVersion NSAppKitVersionNumber10_12_1 = 1504.60;
static const NSAppKitVersion NSAppKitVersionNumber10_12_2 = 1504.76;
```

One typical use of this is to floor() the value, and check against the values provided in NSApplication.h. For instance:

```
if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_9) {
  /* On a 10.9.x or earlier system */
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_10) {
  /* On a 10.10 system */
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_10_x) {
  /* on a 10.10.x system */
. . .
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_11) {
  /* on a 10.11 or 10.11.x system */
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_12) {
  /* on a 10.12 or 10.12.x system */
} else {
  /* on a 10.13 or later system */
}
```

Note that unlike most AppKit software updates, 10_10 software updates incremented the AppKit major version, which accounts for the specific advice for 10_10.x. Other special cases or situations for version checking are discussed in the release notes as appropriate. For instance some individual headers may also declare the versions numbers for NSAppKitVersionNumber where some bug fix or functionality is available in a given update, for example:

```
static const NSAppKitVersion NSAppKitVersionWithSuchAndSuchBadBugFix 1138.42
```

Backward Compatibility

One backward compatibility mechanism that is occasionally used in the frameworks is to check for the version of the system an application was built against, and if an older system, modify the behavior to be more compatible. This is done in cases where bad incompatibility problems are predicted or discovered; and most of these are listed below in these notes.

Typically we detect where an application was built by looking at the version of the System, Cocoa, AppKit, or Foundation frameworks the application was linked against. Thus, as a result of relinking your application against the latest SDK, you might notice different behaviors, some of which might cause incompatibilities. In these cases because the application is being rebuilt, we expect you to address these issues at the same time as well. For this reason, if you are doing a small incremental update of your application to address a few bugs, it's usually best to continue building on the same build environment and libraries used originally.

In some cases, we provide defaults (preferences) settings which can be used to get the old or new behavior, independent of what system an application was built against. Often these preferences are provided for debugging purposes only; in some cases the preferences can be used to globally modify the behavior of an application by registering the values (do it somewhere very early, with -[NSUserDefaults registerDefaults:]).

NSCollectionView Responsive Scrolling and Prefetching

Like UICollectionView on iOS, NSCollectionView now prepares content outside of its immediate visibleRect, to facilitate responsive, reduced-latency scrolling.

The internal mechanics differ from those on iOS, adopting the Responsive Scrolling model introduced in macOS 10.9 as the means of "prefetching" content (content is cached on the main thread and scrolling allowed to proceed concurrently on a background thread), but the end result as seen by the user and NSCollectionView client code are nearly the same:

When Responsive Scrolling is enabled for an NSCollectionView, its dataSource may now be sent -collectionView:itemForRepresentedObjectAtIndexPath: and -collectionView:viewForSupplementaryElementOfKind:atIndexPath: for content that resides outside the NSCollectionView's current visibleRect.

The NSCollectionView's delegate will also receive -collectionView:willDisplayItemforRepresentedObjectAtIndexPath: and -collectionView:willDisplaySupplementaryView:forElementKind:atIndexPath: for content outside the visibleRect. The sending of these and the corresponding "…didEndDisplaying…" messages differs subtly from iOS, because Responsive Scrolling caches content outside the visibleRect. The delegate therefore receives these messages when content enters and exits the prefetched content area that is being cached.

To ensure compatibility in light of these subtle behavior changes, Responsive Scrolling in NSCollectionViews is enabled only for apps linked on or after macOS 10.13. If your app's NSCollectionViewDataSource and NSCollectionViewDelegate implementations don't contain any subtle assumptions about instantiated items and supplementary views being inside the NSCollectionView's visibleRect, a simple recompile on macOS 10.13 should be all you need to do to automatically enable Responsive Scrolling for your NSCollectionViews.

If you suspect trouble due to the enabling of Responsive Scrolling for NSCollectionViews, please file a Radar describing the problem, providing an example project that demonstrates the issue if possible. You can experimentally disable Responsive Scrolling as described in the NSView.h header comment for -prepareContentInRect:, by overriding that method to invoke [super prepareContentInRect:[self visibleRect]] and then return.

If fetching the data needed to display each item in your CollectionView is potentially high-latency, you may wish to assign your CollectionView a prefetchDataSource. The CollectionView will message this object as early as possible after determining that items at particular index paths are likely to be displayed soon (e.g. due to scrolling). It can also notify your prefetchDataSource when previously requested data is no longer expected to be needed.

The prefetchDataSource should conform to the new NSCollectionViewPrefetching protocol, which has one required method and one optional method:

```
@protocol NSCollectionViewPrefetching <NSObject>

@required

/* Notifies your 'prefetchDataSource' that items at the specified 'indexPaths' are likely to be instantiated and displayed soon.  The CollectionVie
*/
- (void)collectionView:(NSCollectionView *)collectionView prefetchItemsAtIndexPaths:(NSArray<NSIndexPath *> *)indexPaths;

@optional
```

```
/* Notifies your 'prefetchDataSource' that items at the specified 'indexPaths', for which the CollectionView previously sent -collectionView:prefet
*/
- (void)collectionView:(NSCollectionView *)collectionView cancelPrefetchingForItemsAtIndexPaths:(NSArray<NSIndexPath *> *)indexPaths;

@end
```

NSCollectionView Drag-and-Drop

On macOS 10.12, dragging one or more contiguous items within a CollectionView and dropping them in the same place with a proposedDropOperation of NSCollectionViewDropBefore was treated as a no-op by NSCollectionView (-collectionView:acceptDrop:indexPath:dropOperation: would not be sent to the NSCollectionView's delegate). This unintentionally suppressed operations such as copying items in place. On macOS 10.13, NSCollectionView only treats the drop as a no-op if the NSDragOperation is also NSDragOperationMove.

An NSCollectionViewDelegate that wants to provide file promises on macOS 10.12 and later should implement -collectionView:pasteboardWriterForItemAtIndexPath: to return a fully configured NSFilePromiseProvider. The NSFilePromiseProvider's delegate provides resolution of the corresponding file name and writing of the promised file on demand, eliminating the need for -collectionView:namesOfPromisedFilesDroppedAtDestination:forDraggedItemsAtIndexPaths:, which is now considered a legacy method and does not need to be implemented unless targeting macOS < 10.12.

NSCollectionView viewForSupplementaryElementOfKind Fix (New Since Seed 2)

On macOS 10.11 and 10.12, an NSCollectionView dataSource's implementation of -collectionView:viewForSupplementaryElementOfKind:atIndexPath: would sometimes be invoked with an undeclared kind. This no longer occurs on macOS 10.13. Implementations of this method that must support macOS < 10.13 should return nil when passed an unrecognized kind. This is OK to do as a workaround, even though this method is declared and documented as requiring its return value to be nonnull.

NSCollectionView -draggingImage... Methods and Out-of-View Items (New Since Seed 2)

-draggingImageForItemsAtIndexPaths:withEvent:offset: and its legacy counterpart, -draggingImageForItemsAtIndexes:withEvent:offset:, can only image parts of items that are within the CollectionView's visibleRect. Invoking either method for items that lie entirely outside the visibleRect can cause an exception to be raised due to the empty area being imaged.

On macOS 10.13, invoking these methods for items entirely outside the visibleRect no longer raises an exception, and instead simply returns a zero-sized NSImage. Apps targeting previous versions of macOS should verify that the items they're asking to image are within the CollectionView's visibleRect before they invoke either of these methods. For each indexPath being considered, this can be done by obtaining the corresponding NSCollectionViewItem "item" using:

```
NSCollectionViewItem *item = [collectionView itemAtIndexPath:indexPath];
```

and then, if item != nil, confirming that [[item view] frame] intersects [collectionView visibleRect].

NSCollectionView Setup

When preparing to use an NSCollectionView with the 10.11-and-later API, it's important to ensure that you assign a collectionViewLayout to the CollectionView as early as possible, as this lets the CollectionView know definitively that it will be operating using the 10.11-and-later API model. Certain configuration operations, such as invoking the "-registerClass:..." or "-registerNib:..." methods, can fail if attempted before the CollectionView has a collectionViewLayout, causing unexpected odd behavior later when the CollectionView attempts to instantiate items. (An example of this is attempting to register item classes or nibs in an -initWithFrame: override in an NSCollectionView subclass. This problem primarily applies to programmatic instantiation and configuration of NSCollectionViews, since an NSCollectionViews unarchived from a .nib file or storyboard typically has a collectionViewLayout configured already.)

Ensuring that a CollectionView has a non-nil collectionViewLayout first, and then performing further configuration such as class and nib registrations, will ensure that your configuration code has the intended effect. This advice applies to macOS 10.11 through High Sierra.

NSApplicationDelegate URL Handling

NSApplicationDelegate now has -application:openURLs:. This will be called for any URLs your application is asked to open, including URL types and Document types defined in your Info.plist (under the CFBundleURLTypes and CFBundleDocumentTypes keys). This method will not be called for URLs that have an associated NSDocument class, which will be opened through NSDocumentController. If this is implemented, -application:openFiles: and -application:openFile: will not be called.

NSTreeController arrangedObjects

NSTreeController arrangedObjects now returns an NSTreeNode. On previous releases, the object returned responded to the -children and -descendantNodeAtIndexPath: selectors, but was not an NSTreeNode. Because of this, applications deployed to previous versions of macOS should check the version before calling any methods aside from -children and -descendantNodeAtIndexPath:.

Asynchronous Restorable State Encoding

Objects that need to encode restorable state can now do so in the background by overriding encodeRestorableStateWithCoder:backgroundQueue: on their NSResponder and NSDocument subclasses. The given backgroundQueue is a serial queue, and the given coder may be used to encode on multiple threads. The callout still occurs on the main thread, however encoding will be considered finished once all operations enqueued are finished. You can use it like so:

```
- (void)encodeRestorableStateWithCoder:(NSCoder *)coder backgroundQueue:(NSOperationQueue *)queue {
    [super encodeRestorableStateWithCoder:coder backgroundQueue:queue];
    [coder encodeObject:self.documentIdentifier forKey:@“documentIdentifier”];
    [queue addOperationWithBlock:^{
        [coder encodeObject:[self URLForIdentifier:self.documentIdentifier createIfNeeded:YES] forKey:@“documentURL”]; // This might hit the networ
    }];
}
```

NSWindow Lifecycle Changes (Updated since Seed 2)

If your application is linked on macOS 10.13 SDK or later, NSWindows that are ordered-in will be strongly referenced by AppKit, until they are explicitly ordered-out or closed (not including hide or minimize operations). This means that, generally, an on-screen window will not be deallocated (and close/order-out as a side-effect of deallocation).

NSPasteboard Leak Reporting

NSPasteboard objects created with non-standard names must have releaseGlobally invoked on them in order for their system-wide resources to be cleaned up. Starting in macOS 10.13, such pasteboards that are otherwise unreferenced by an application will show up in leaks.

NSPasteboard Secure Coding (Updated since Seed 2)

If your application is linked on macOS 10.13 SDK or later, classes that return NSPasteboardReadingAsKeyedArchive from readingOptionsForType:pasteboard: must support secure coding, and will be decoded with a coder that requires secure coding. When linked on prior SDKs, only classes that claim to support secure coding will be decoded with a coder that requires secure coding.

NSToolbar

NSToolbar's -init method now calls through to -initWithIdentifier: with an empty string identifier. This makes sure the toolbar is properly setup when not using -initWithIdentifier:. User customizable toolbars are not supported when using -init and must be created by using -initWithIdentifier: and a unique identifier. Setting allowsUserCustomization to YES on a toolbar created using -init will cause an assert. This behavior is new to 10.13, using -init will have undefined behavior on previous versions.

NSTouchBar RTL

NSTouchBar gains a new property in 10.13 named groupUserInterfaceLayoutDirection which will allow the items in the group to be laid out in the order appropriate for the current localization. This property defaults to NSUserInterfaceLayoutDirectionLeftToRight which means nothing will flip automatically. To flip the layout direction of the group, set this property to [NSApp userInterfaceLayoutDirection]. When creating Touch Bar controls that support RTL layouts, NSGroupTouchBarItem has the benefit of supporting customization whereas container views like NSStackView do not.

NSTouchBarItem Identifiers

NSTouchBarDelegate's -touchBar:makeItemForIdentifier: method has always warned if an item is returned with an identifier that does not match the identifier AppKit is asking for. AppKit will now assert instead.

NSGroupTouchBarItem

NSGroupTouchBarItem gains the ability to dynamically size items in the form of NSUserInterfaceCompressionOptions which allow hiding images or text, among other behaviors, which are applied when items would otherwise be dropped in Touch Bar. This behavior has been enabled by default for all NSGroupTouchBarItems in 10.13. To opt out, set the -prioritizedCompressionOptions property to an empty array, or set the allowedCompressionOptions parameter of the +groupItemWithIdentifier:items:allowedCompressionOptions: constructor to an empty option.

Additionally, +groupItemWithIdentifier:items: will create an instance of self starting in 10.13 rather than an instance of NSGroupTouchBarItem specifically.

NSTableView (New since Seed 1)

In 10.13 when a cell view or row view is removed from a NSTableView and added to the reuse queue, AppKit will usually hide it and keep it inside its superview rather than removing it. When -makeViewWithIdentifier:owner: returns one of these views, we will call -prepareForReuse on the view which will unhide it. AppKit will keep track of views returned from the reuse queue from -makeViewWithIdentifier:owner: and if they are not returned to the table from a call to one of the row view or cell view creation delegate methods, AppKit will remove them from the superview and add them back to the reuse queue. Please do not rely on these views being removed from the superview when added to the reuse queue.

NSTouchBar Esc Key Replacements (New since Seed 1)

AppKit will now notice changes in the size of an esc key replacement item after initial display and appropriately resize the esc key region. In previous versions, the esc key region would remain the same size and the item's view may have become clipped.

NSWindowController, NSTabViewController, NSSplitViewController

Changing the delegate of an NSWindow, NSTabView, or NSSplitView managed by a NSWindowController, NSTabViewController, or NSSplitViewController respectively is not supported and will assert. When building using Xcode 9 or later, Xcode will now automatically set the delegate outlet on these objects to their owning controllers. If the application was previously changing the delegate of these objects programmatically after view/window loading, the application will now assert.

New Enumerations

NSWindowDepth and NSTextMovement have adopted NS_ENUM; NSFontPanelModeMask and NSWindowNumberListOptions have adopted NS_OPTIONS. Overrides of

API that deals with these types and previously declared the type as NSInteger will result in a build warning and should be updated to use the new enumeration types.

Many new closed and extensible string typedefs are introduced in AppKit using NS_STRING_ENUM and NS_EXTENSIBLE_STRING_ENUM attributes.
NS_STRING_ENUM is used for sets of strings where AppKit defines all possible values – custom values cannot be defined. For example, AppKit defines all possible NSAccessibilityRole values, and external systems cannot add additional values.
NS_EXTENSIBLE_STRING_ENUM is used for types where arbitrary strings can be used as their values, even if AppKit defines standard values. For example, AppKit has standard values for NSToolbarItemIdentifier, but custom ones are intended to be defined.
API that is intended to deal with one of these specific sets of strings have adopted these typedefs in place of NSString to indicate that relationship. For example, NSImageName is a new extensible typedef that is used as the type of the NSImageNameActionTemplate constant as well as the type of the argument passed to +[NSImage imageNamed:].
There is no binary or source impact to Objective-C, and literal or dynamic NSStrings can continue to be used with this API.

This same technique is used with non-NSString types, such as NSFontWeight, NSLayoutPriority, NSStackViewVisibilityPriority, NSToolbarItemVisibilityPriority, NSModalResponse, NSWindowLevel, and NSAppKitVersion. These are all extensible types as custom values can be used alongside AppKit's standard values.

Nested enumerations and globals in Swift

Enumerations, options, and global C constants and functions are nested into associated types in Swift 4.
For instance: NSAlertStyle is imported as NSAlert.Style, NSStackViewSpacingUseDefault is imported as NSStackView.useDefaultSpacing, NSAvailableWindowDepths() are imported as NSWindow.Depth.availableDepths, and NSRectFill(_:) is imported as NSRect.fill().
When an enumeration or global is not tied to a single type, it is left global, e.g. NSBackgroundStyle is left as-is.

NSColorPickerTouchBarItem's allowedColorSpaces

NSColorPickerTouchBarItem has a new property, allowedColorSpaces, that allows for restricting the color picker to only display and emit colors within those color spaces.
If the color picker encounters a color, such as in the user's favorite swatches, outside of the allowed color spaces, it will convert the color to the first allowed color space before displaying it or notifying the target of its selection.
This can be set to a single color space, such as the color profile of a document, to clarify to the user the exact color that will be added to the document on selection.
By default it is nil, which means the color picker can present any possible color.

NSStoryboard.mainStoryboard

NSStoryboard now has a mainStoryboard class property that can be used to refer to the application's main UI storyboard as configured in the Xcode project.
This and explicit creation of the main storyboard using +storyboardWithName:bundle: is a cached singleton.

print() methods in Swift

NSWindow, NSView, NSDocument's print() instance methods have been renamed to printWindow(), printView(), and printDocument() respectively in Swift 4. This fixes the unexpected experience where adding debug logging to a subclass of one of these instances shows a print panel instead.

NSSplitViewController should conform to NSUserInterfaceValidations

NSSplitViewController now publicly conforms to NSUserInterfaceValidations, which it did internally since 10.11. Any overrides of -validateUserInterfaceItem: should call through to super to get standard validation of the -toggleSidebar: action method.

NSPasteboardTypeURL and NSPasteboardTypeFileURL (New since Seed 1)

New NSPasteboardTypes have been added: NSPasteboardTypeURL and NSPasteboardTypeFileURL, which correspond to UTIs of kUTTypeURL and kUTTypeFileURL respectively. These replace the deprecated NSURLPboardType and NSFilenamesPboardType pasteboard types and should be used in their place when registering for drag types, etc. When used with pre-10.13 deployment targets, kUTTypeURL and kUTTypeFileURL can be used as pasteboard type values instead.

NSColors from Asset Catalogs

Asset catalogs can now store colors. Once a color has been added to an asset catalog and given a name, an instance of NSColor can be instantiated using one of the following new methods:

```
+ (nullable NSColor *)colorNamed:(NSColorName)name NS_AVAILABLE_MAC(10_13);
+ (nullable NSColor *)colorNamed:(NSColorName)name bundle:(nullable NSBundle *)bundle NS_AVAILABLE_MAC(10_13);
```

A new extensible string enum has been added, NSColorName, that replaces NSColorListColorKey:

```
typedef NSString * NSColorName NS_EXTENSIBLE_STRING_ENUM;
```

NSAccessibility

Three new text attributes are now available:
- NSAccessibilityAnnotationTextAttributes
- NSAccessibilityCustomTextAttribute
- NSAccessibilityLanguageTextAttribute

The annotation text attribute is for annotation-like UI in the text document.
The custom text attribute is to cover all application's own custom text attributes.
The language text attribute is to allow the app to specify the language associated with a range of text.

Two new attributes are now available:
- accessibilityChildrenInNavigationOrder
- accessibilityCustomRotors

accessibilityChildrenInNavigationOrder allows the application to specify the order of VoiceOver navigation for the children elements of an element.
accessibilityCustomRotors allows the application to provide its own results for various searches, and to construct its own custom lists to be used for VoiceOver rotors.

One new role is now available:
- NSAccessibilityPageRole

The page role is useful for document type applications that have page-like structure for the document.

Three new subroles are now available:
- NSAccessibilityTabButtonSubrole
- NSAccessibilityCollectionListSubrole
- NSAccessibilitySectionListSubrole

The TabButton subrole is used for tabs in the window tab bar and NSTabView; these tabs report a role of NSAccessibilityRadioButtonRole, but can now be distinguished from ordinary radio buttons by inspecting the subrole.
The CollectionList and SectionList subroles are used to identify certain parts of an NSCollectionView.

Two new properties of NSWorkspace are now available:
- voiceOverEnabled
- switchControlEnabled

These NSWorkspace properties allow the developer to know if VoiceOver and Switch Control are running or not.

NSAccessibilityCustomAction

The custom action class allows an application to construct its own accessibility action that's not defined by the OS.
To see an implementation of using NSAccessibilityCustomAction, please check out the developer sample code project AccessibilityUIExamples.

NSAccessibilityCustomRotor

This new class allows apps to vend their own custom rotors, similar to the UIAccessibilityCustomRotor class in UIKit.
Many assistive technologies, such as VoiceOver, provide ways to quickly search applications for content of a given type. For example, in a web browser, a list of links can quickly be explored using VoiceOver's content menus. On macOS and iOS, these content menus are referred to as rotors.
Apps can use the NSAccessibilityCustomRotor class to create custom rotors. As an example, Pages can create a "Headings" custom rotor that allows assistive technologies to search the Pages document for all headings, which are then presented to the user.
To see an implementation of using NSAccessibilityCustomRotor, please check out the developer sample code project AccessibilityUIExamples.

Nullability

We fixed a number of places where arguments and return values had incorrect nullability. These changes have no effect on Swift 3, in order to maintain source compatibility.

Some nullability changes have special considerations if you support running on releases older than 10.13:

The NSBrowser property selectionIndexPath can return nil, and is now nullable to reflect that. In 10.13, it also accepts nil as a set value. However on releases older than 10.13, setting nil will throw an assertion.

The NSText property string incorrectly accepted nil values but threw an exception when attempting to set the property to nil. The property now requires a nonnull value in Objective-C and Swift 4, while the incorrect nullability is retained for Swift 3 to maintain source code compatibility.

The NSTextField convenience constructor +textFieldWithString: incorrectly accepted nil values. Since NSTextField's -stringValue requires a nonnull value, this made it possible in Swift to create a NSTextField with a nil state that it was not possible to return to. The convenience constructor now requires a nonnull value in Objective-C and Swift 4, while the incorrect nullability is retained for Swift 3 to maintain source code compatibility.

NSWindow Implicit Full Screen

NSWindow no longer checks if a resizable window can be resized to the current screen's frame to make the window implicitly full screen capable. It now relies on NSWindowStyleMaskResizable without the additional min/max size check. As in 10.11, the exact semantics of this may change over time, and it is recommended to explicitly opt in primary document windows by including NSWindowCollectionBehaviorFullScreenPrimary in the collectionBehavior.

Zoom button in Full Screen

When an window is in a full screen split view, option clicking the zoom button will now maximize the window. This puts both windows into their own full screen space.

NSSegmentedControl alignment and distribution properties

NSSegmentedControl now has alignment and distribution properties. Distribution controls how the segments are laid out, and alignment controls how the contents of a segment are laid out.

Possible values for distribution are:

```
    NSSegmentDistributionFit
    NSSegmentDistributionFill
    NSSegmentDistributionFillEqually
    NSSegmentDistributionFillProportionally
```

Fit matches the distribution that segmented control has always used until now. Fill distributes extra space equally among all the segments. FillEqually uses the extra space to try to make all the segments equal width. FillProportionally distributes the extra space in proportion to each segment's fitting width.

For older apps the value of this property defaults to Fit, but for apps linked on 10.13 the default is Fill.

Alignment can be control for each segment:

```
- (void)setAlignment:(NSTextAlignment)alignment forSegment:(NSInteger)segment;
- (NSTextAlignment)alignmentForSegment:(NSInteger)segment;
```

The 'natural' alignment of segment content is centered, which is the default value for this property.

NSSavePanel/NSOpenPanel

Layout support for accessory views in Open/Save panels has been consolidated and clarified for macOS 10.13. An app-provided accessory view that does not use auto layout and does not have NSViewWidthSizable set on its autoresizingMask will use legacy behavior and be centered horizontally in the panel. If it does not use auto layout but is specified to be NSViewWidthSizable, the initial size will be used as a minimum width but otherwise the accessory view will be stretched horizontally to fit the panel. If an auto-layout-compliant accessory view is specified, it will be stretched to fit the panel regardless of the initial width, and it is up to the developer to specify any minimum width via constraints. The initial height for auto-layout-compliant accessories is also ignored, and accessory views should be provided with complete and non-ambiguous constraints. As a corollary, accessory views that change height based on constraint modifications are now fully supported.
Open panels raised by NSDocument are now always modeless regardless of whether the user has iCloud Drive enabled, with the exception of any apps still overriding the legacy [NSDocumentController runModalOpenPanel:] method. Apps with more specific requirements for the Open panel (such as requiring the Open panel to open above a welcome screen) should override the [NSDocumentController beginOpenPanel:forTypes:completionHandler:] API and make adjustments to the supplied panel as necessary.

Drag & Drop File Promises

The various namesOfPromisedFilesDroppedAtDestination: APIs have all been deprecated. Please migrate to the item based NSFilePromiseReceiver and NSFilePromiseProvider replacement API introduced in 10.12. In cases where you need to add additional data to each promised item on the pasteboard, you can either add it directly to the pasteboard (which puts it on the first item) or subclass NSFilePromiseProvider and implement the NSPasteboardWriting protocol.

Drag & Drop Auto Image Scaling

Large drag and drop images are automatically scaled to smaller sizes as the drag leaves the source view of the drag. The following general advice now applies to all sources of a drag.
1. The drag should start with an image of the item being dragged sized as the user currently experiences it in the UI.
2. Carefully choose the source view for the drag. Auto scaling occurs once the drag exits the bounds of the source view. For example, if re-arranging items in a table, the source view should be the table and not the cell view. In this manner the drag image won't change size during the re-order operation unless the user drags outside of the table view.

Class properties

A number of class setter and getter methods have been converted to class properties. These properties can still be accessed as methods from Objective-C, or you may use the property dot syntax. The methods are still exposed in Swift 3, but Swift 4 exposes the properties.

Objective-C:

```
@property (class, readonly, strong) __kindof NSApplication *sharedApplication;
```

Swift 3:

```
class func shared() -> NSApplication
```

Swift 4:

```
class var shared: NSApplication { get }
```

NSTouchBar support for NSDocument-based applications

In macOS 10.13, NSDocument-based applications automatically gain a "New Document" button in the Touch Bar when no documents are registered via NSDocumentController.addDocument(_:). The creation of this button is dependent on the existence of a visible and validated menu item in the File menu with an action of newDocument:. Tapping on this button will send newDocument: to a nil target, just like the menu item.

NSDocumentController provides this NSTouchBar via NSApp.touchBar, but will not override NSTouchBars set manually via the same property. Note that as of macOS 10.13, this NSTouchBar will supersede any provided by NSApp.delegate.touchBar.

To opt out of this NSTouchBar, you can override NSApplication.touchBar to return nil, or manually set an alternative NSTouchBar on NSApp.

NSLevelIndicator

The Level Indicator control sports a brand new look in macOS 10.13! Accompanying the new look are several new APIs on NSLevelIndicator to support custom colors, tiered capacity styles, and rating indicator customization.

Custom Colors:

The colors used to draw the normal, warning, and critical states of the level indicator can now be read and customized via the following properties:

```
@property (copy, null_resettable) NSColor *fillColor;
@property (copy, null_resettable) NSColor *warningFillColor;
@property (copy, null_resettable) NSColor *criticalFillColor;
```

The fillColor property is also used when drawing rating-style indicators. These properties are null_resettable and can be used to read out the standard colors used by the control. Avoid archiving these colors explicitly as they may change across control styles and releases of macOS.

Tiered Capacity Styles:

The Continuous Capacity and Discrete Capacity styles now support the option to draw all three threshold colors stacked in sequence, similar to an audio VU meter. This behavior is enabled through the drawsTieredCapacityLevels property.

Rating Indicator Options:

The visibility of Rating indicator placeholders ("empty stars") can now be customized via the placeholderVisibility property.

The images used by the Rating indicator can now be customized via the following properties:

```
@property (strong, nullable) NSImage *ratingImage;
@property (strong, nullable) NSImage *ratingPlaceholderImage;
```

Template images are rendered using the fillColor of the control. If ratingImage is set, but ratingPlaceholderImage is not, then a faded version of the ratingImage is automatically used to draw placeholders.

NSLevelIndicatorStyle

In Objective-C and Swift 4, this enumeration has been updated to use common-prefix naming, bringing it in line with modern naming conventions and improving its representation in Swift.

NSButton Content Animations

NSButton supports animation of its content in 10.13 with use of its animator proxy. e.g.:

```
button.animator.title = @"New Title";
button.animator.imagePosition = NSImageOnly;
```

Many NSButton properties can be animated: title, attributedTitle, image, imagePosition, bezelColor, alternateTitle, alternateImage. In order to use these built in animations, the NSButton or subclass must be layer-backed and not have custom cell drawing.

NSFontAssetRequest

This new class has been added to allow downloading system-provided font assets asynchronously without blocking user interactions. This new API provides a NSProgress based API to track the download progress and sync with UI.

```
@interface NSFontAssetRequest : NSObject <NSProgressReporting>

- (instancetype)initWithFontDescriptors:(NSArray<NSFontDescriptor *> *)fontDescriptors options:(NSFontAssetRequestOptions)options;
@property (readonly, copy) NSArray<NSFontDescriptor *> *downloadedFontDescriptors;
@property (readonly, strong) NSProgress *progress;
- (void)downloadFontAssetsWithCompletionHandler:(BOOL (^)(NSError * _Nullable error))completionHandler;

@end
```

For a new download you will create a new NSFontAssetRequest instance with an array of font descriptors with -initWithFontDescriptors:options:, then start the download with -downloadFontAssetsWithCompletionHandler:. All the font assets matching these descriptors will be downloaded and the entire progress will be reported through NSProgressReporting protocol via the progress property of this NSFontAssetRequest instance.

When an error happens (due to network failure or other reasons), the completionHandler block will be called to decide whether the download should continue (return YES) or be aborted (return NO), and the error argument will provide detailed information on why it happened. At the end of the download, when all the font descriptors have been processed, the completionHandler block will be called again with a nil argument to indicate the completion. After that you can release the NSFontAssetRequest instance.

NSFontAssetRequest also provides a way to directly trigger system standard UI for font asset downloading, when developers do not want to provide their custom UI for it.

```
typedef NS_OPTIONS(NSUInteger, NSFontAssetRequestOptions) {
    NSFontAssetRequestOptionUsesStandardUI = 1 << 0,
}
```

Pass this option as options to -initWithFontDescriptors:options: to show the download progress with the standard system UI. When you use this option, you can still receive the progress information from the progress property.

This new API is a replacement for the low-level CoreText API CTFontDescriptorMatchFontDescriptorsWithProgressHandler().

NSFontDescriptor

NSFontDescriptor has a new property to determine whether this font descriptor should be downloaded as asset before use.

```
@property (readonly) BOOL requiresFontAssetRequest NS_AVAILABLE_MAC(10_13);
```

YES indicates that any fonts matching the descriptor needs to be downloaded prior to instantiating a font. To ensure that the matching fonts are available before use, use NSFontAssetRequest to download. NO indicates that the descriptor is not available for download, has already been downloaded, or is backed by an installed font.

More predictable -updateConstraints

As of macOS 10.13, views with -needsUpdateConstraints=YES will receive -updateConstraints prior to a layout pass, even if the view hierarchy contains no layout constraints. This means that -setNeedsUpdateConstraints:YES will reliably cause -updateConstraints to be called, and that views can place constraint logic in -updateConstraints without fear that it will never be invoked. Remember that -updateConstraints is not a reliable indicator that auto layout is being used in a window. In uncommon cases, view subclasses may need to avoid adding constraints from within -updateConstraints unless auto layout is already in use elsewhere in the window. The recommendation for these cases is to add constraints only if translatesAutoresizingMaskIntoConstraints is set to NO, and to operate in "non-auto-layout mode" otherwise.

On a related note, on releases prior to macOS 10.13, there were cases where constraints could be updated for only a portion of the view hierarchy before performing layout. This issue has now been resolved, so if -layoutSubtreeIfNeeded is called on a view, constraints will be updated for all views that could potentially impact the layout of the receiver or its descendants.

Stricter allocation of view-backing layers

macOS 10.13 is stricter in some cases about when it allocates view-backing layers for certain views. As before, if your view relies on having a layer (e.g. for the purpose of setting layer properties), it should receive [view wantsLayer:YES] and should probably also return YES from -wantsUpdateLayer.

Ownership of view-backing layer properties

macOS 10.13 is more consistent around updates to properties of views' layers (that is, layers that are the .layer of some NSView, however created). As before, if an application modifies such a layer by changing any of the following properties, the behavior of the application is undefined: bounds, position, zPosition, anchorPoint, anchorPointZ, transform, affineTransform, frame, hidden, geometryFlipped, masksToBounds, opaque, compositingFilter, filters, shadowColor, shadowOpacity, shadowOffset, shadowRadius, shadowPath, layoutManager.

In addition, it is illegal (and ineffective) to change the delegate of a view's layer unless that view either returns a custom layer from -makeBackingLayer or had a custom layer set via -setLayer:. If a view does not have such a custom layer, the identity of its layer's delegate is undefined and should not be relied upon.

As before, for best results, a view that modifies any other properties of its layer should also faithfully implement drawRect: to reflect such modifications, since not all layer properties can be automatically represented in all drawing circumstances. For instance, if a view's -updateLayer implementation changes the border properties of its layer, its -drawRect: should draw an equivalent border.

Ordering of a view's layer's sublayers

macOS 10.13 is more consistent with regard to ordering of the sublayers of the layer of a view's layer. Specifically, if a view's layer has sublayers that do not belong to a view, it will maintain their order with respect to one another, but they will collectively be ordered behind any subviews' layers. Another way to think about this is that any custom sublayers of a view's layer are considered to be part of the view's appearance and thus appear below any sublayers. A view that wishes to intersperse custom CALayer content with its subviews should create one or more subviews to hold those sublayers as appropriate and then order the subviews as needed.

CALayerDelegate callbacks

macOS 10.13 is more strict with regard to the relationship between a view and its layer. As previously, if a view relies on being the delegate of its layer, it should (a) declare conformance to the protocol <CALayerDelegate>, and (b) override -makeBackingLayer to create a custom layer, set its delegate to self, and return it.

Layer layout and -[NSView viewWillDraw]

In previous versions of macOS, changing the geometry of a layer-backed view resulted in its (or, if applicable, its layer's layoutManager's) automatically being sent -layoutSublayersOfLayer:. This is no longer always the case on macOS 10.13. First, as above, the behavior of an application that modifies the layoutManager property of a view's layer is undefined. Second, unless a view has a custom layer (returned from an override of -makeBackingLayer or set via -setLayer:), the identity of its delegate is both immutable and undefined, so a view with a non-custom layer may not rely on any CALayerDelegate methods. (Use normal view methods like -updateConstraints, -layout, -

drawRect:, -updateLayer, and -animationForKey: instead.)

Even if a view has a custom layer, its -layoutSublayers (or its delegate's -layoutSublayersOfLayer:) may not adjust the geometry (bounds, position, zPosition, anchorPoint, anchorPointZ, transform, affineTransform, frame, hidden, geometryFlipped) of any sublayer that is some view's layer. It may only do so for custom sublayers. To influence the geometry of views' layers, modify the view's geometry directly, ideally in an -[NSView layout] override.

Furthermore, in previous versions of macOS, changing the geometry of a layer-backed view resulted in its layer automatically being sent -viewWillDraw. This was incidental and is no longer always the case on macOS 10.13. To cause a view to receive -viewWillDraw, call setNeedsDisplay:YES on it. Alternatively, switch to -layout and call setNeedsLayout:YES.

More consistent behavior of diverted view drawing

The behavior of -[NSView cacheDisplayInRect:toBitmapImageRep:] and -[NSView displayRectIgnoringOpacity:inContext:] are slightly more consistent in macOS 10.13, especially with regard to flipped views. Here is a summary of the behavior. -cacheDisplayInRect:toBitmapImageRep: will draw the given rect of the bounds space of the receiving view at the origin of the given bitmap. Flippedness is taken into account (i.e., we'll automatically flip before calling the view's drawRect: if necessary). -displayRectIgnoringOpacity:inContext: will draw the given rect of the bounds space of the receiving view into the same rect of user space of the provided context. Again, flippedness is taken into account, but no other transform is applied to the given context. The gstate of the context after return is unmodified.

For flippedness to be taken into account correctly, the semantic flippedness of the context needs to be properly set. The flippedness of a context is set upon creation with +graphicsContextWithCGContext:flipped:.

As before, for best results when being drawn via either of these two methods, a view should faithfully implement drawRect:, since not all layer properties can be faithfully represented when drawing into a CGContext.

Performance of text field rendering

macOS 10.13 improves performance of rendering of NSTextField. As before, to achieve optimal text appearance—including font smoothing—and performance, it is recommended to use NSTextField for displaying text rather than doing manual string drawing.

NSProgressIndicator

The NSProgressIndicator API has changed slightly in macOS 10.13. First, the enumerators in NSProgressIndicatorStyle are renamed to match the more modern common-prefix-based naming scheme and to come across more naturally in Swift. The old versions are still usable, but do not rely on them—they will be removed in a future release. In addition, use the enumerators of the NSProgressIndicatorThickness type is highly discouraged. These values do not accurately represent the geometry of NSProgressIndicator, and they will be removed in a future release.

Non-deferred drawing discouraged

Use of any of the following means of non-deferred drawing is discouraged: the -lockFocus, -lockFocusIfCanDraw, -unlockFocus, and -canDraw methods on NSView (subclass NSView and implement -drawRect: instead); the -graphicsContext method on NSWindow (add views instead); direct use of the -display, -displayIfNeeded, -displayIfNeededIgnoringOpacity, -displayRect:, -displayIfNeededInRect:, -displayRectIgnoringOpacity:, and -displayIfNeededInRectIgnoringOpacity: methods on NSView (call setNeedsDisplay:YES on a view instead; AppKit will then automatically display views for you); the -scrollRect:by: method on NSView (use NSScrollView).

Don't modify the view hierarchy during -drawRect: or -updateLayer

As before, if an application modifies a view hierarchy (or the layer hierarchy of a layer attached to a view) during a -drawRect: or -updateLayer callout to a view, the behavior of the application is undefined. A better time to make such adjustments is in a view's -layout method, where the view is free to modify its subviews. (It should not modify its own geometry there.)

NSBackingStoreType deprecations

The NSBackingStoreRetained and NSBackingStoreNonretained enumerators of NSBackingStoreType have been deprecated in macOS 10.13. The only non-deprecated choice is NSBackingStoreBuffered. Use NSBackingStoreBuffered when creating an NSWindow. (The other options have effectively been synonyms of NSBackingStoreBuffered since OS X Mountain Lion, as the macOS window server has not supported retained or nonretained windows.)

NSMenu

A new NSMenuItem property, allowsKeyEquivalentWhenHidden, will cause a menu item's key equivalent to be considered for keyboard shortcut matching even when the menu item is hidden. This is useful for adding extra shortcuts that don't need to be visible in the menu bar.

-[NSMenuItem setTitleWithMnemonic:] has been deprecated. Use -setTitle: instead. Note that all other mnemonic API was previously deprecated in macOS 10.6 and macOS 10.8.

-[NSMenu indexOfItemWithRepresentedObject:] now accepts a nil object, and will return the index of the first item that it finds that has a nil representedObject property. In this case, it ignores menu items that are separators, or that have submenus, since these typically have no represented object but are also uninteresting to return from this API.

Improved orphan control for short Chinese and Japanese strings

We've reduced the occurrence of orphan characters when laying out short Chinese and Japanese strings in macOS 10.13. This new layout logic is used with the Cocoa string drawing convenience methods when the preferred language is set to Chinese or Japanese and when drawing attributed strings tagged as Chinese or Japanese with NSLanguageAttributeName. The orphan reduction is most noticeable for strings that span 2 lines.

NSTextField adopts the improved orphan control automatically; you get these improvements for free if you are using NSTextField.

These improvements are not adopted by NSTextView, since NSTextView is generally used for drawing longer strings and does not use the string drawing convenience methods to draw its text.

Sharing Service Picker for NSTouchBar

NSSharingServicePickerTouchBarItem can now have the delegate be set to an instance of itself.

NSWindow

The NSWindowStyleMaskMiniaturizable style mask bit is properly respected for applications that link on 10.13 or higher. Previously, it would only include showing or hiding the miniaturize button, but the window would potentially still be miniaturizable via menu items or double clicking on the titlebar.

Window updates are automatically batched together and disableScreenUpdatesUntilFlush no longer needs to be called; it is effectively a no-op.

Previously NSWindow would make the titlebar transparent for certain windows with NSWindowStyleMaskTexturedBackground set, even if titlebarAppearsTransparent was NO. When linking against the 10.13 SDK, textured windows must explicitly set titlebarAppearsTransparent to YES for this behavior.

Automatic NSWindow Tabbing

A window with titlebarAppearsTransparent is normally opted-out of automatic window tabbing (when window.tabbingMode==NSWindowTabbingModeAutomatic). This is because any content can be placed under the titlebar area and show through, and the system's automatic window tabs would potentially conflict with this area. In macOS 10.13 applications can explicitly allow full automatic window tabbing by setting window.tabbingMode=NSWindowTabbingModeAutomatic, and setting an explicit tabbingIdentifier, such as: window.tabbingIdentifier = @"MyUniqueWindowID".

Adding a window to a tab group with the API -addTabbedWindow:ordered: did not work correctly unless the window was first ordered in. This has been fixed for all applications on macOS 10.13. Applications targeting an older OS should order the window in before calling addTabbedWindow:ordered:.

It is now possible to provide your own menu items for the standard window tabbing methods:

```
- (IBAction)selectNextTab:(nullable id)sender NS_AVAILABLE_MAC(10_12);
- (IBAction)selectPreviousTab:(nullable id)sender NS_AVAILABLE_MAC(10_12);
- (IBAction)moveTabToNewWindow:(nullable id)sender NS_AVAILABLE_MAC(10_12);
- (IBAction)mergeAllWindows:(nullable id)sender NS_AVAILABLE_MAC(10_12);
- (IBAction)toggleTabBar:(nullable id)sender NS_AVAILABLE_MAC(10_12);
- (IBAction)toggleTabOverview:(nullable id)sender NS_AVAILABLE_MAC(10_13);
```

Simply create items in IB and add them to appropriate menus and the system will avoid adding its own menu items. This allows more customization of the placement of such menu items.

An NSWindow’s tab can be customized for a window with the new NSWindowTab class and the following property on NSWindow:

```
@property (strong, readonly) NSWindowTab *tab NS_AVAILABLE_MAC(10_13);
```

For example a title that appears only in the tab can be set with:

```
window.tab.title = @"Hello World";
```

Please see NSWindowTab.h for more customization properties and options.

NSTitlebarAccessoryViewController

NSTitlebarAccessoryViewController now supports the layoutAttribute being set to NSLayoutAttributeTop. This will only work on applications that link against macOS 10.13 or higher. The use of the styeMask NSWindowStyleMaskFullSizeContentView is required. The top attribute specifies the view to be placed at the location where the title would normally appear. The title will not appear when the top attribute is specified, as it is assumed that the use of the top attribute is for replacing the title. The viewController.view will be resized to take up the horizontal space between the "standard window buttons" (the traffic lights) and the outer edge. The vertical size will be set to the normal titlebar height. When using a window with the titleVisibility set to NSWindowTitleHidden, and a toolbar is shown, the view will be vertically sized to the toolbar height (if shown), and will overlap the toolbar. This allows you to place a custom view (such as a title text field or another control) in the dead center of a window title area, as seen in System Preferences.

Centering a view, such as an NSTextField, can easily be done with auto layout. One can subclass NSTitlebarAccessoryViewController, and center a view with code such as the following:

```
    @IBOutlet weak var titleTextField: NSTextField! // Setup to the view we want centered

    var centerConstraint: NSLayoutConstraint?

    override func updateViewConstraints() {
        if centerConstraint == nil {
            let contentView  = self.view.window!.contentView!
            // We want to be centered in the window
            centerConstraint = titleTextField.centerXAnchor.constraint(equalTo: contentView.centerXAnchor)
            centerConstraint!.isActive = true
        }
        super.updateViewConstraints()
    }

    override func viewWillAppear() {
        super.viewWillAppear()
        // Make sure our constraints are invalidated when we will appear in a window
        self.view.needsUpdateConstraints = true
    }

    override func viewDidDisappear() {
        super.viewDidDisappear()
        // Drop the center constraint when we move in or out of a window so we can create it relative to the content view
        // This might be different when we are in full screen and in a floating window above the full screen window.
        centerConstraint = nil
    }
```

NSTableView

Hiding rows with the method -hideRowsAtIndexes:withAnimation: did not work correctly prior to macOS 10.13 when using standard row heights. This has been fixed for all applications on macOS 10.13. If you are targeting an older OS, it is recommended to use "variable row heights" by implementing -tableView:heightOfRow: and returning the desired row height; this will work around the bug with hidden row indexes.

Unhiding rows with the method -unhideRowsAtIndexes:withAnimation: did not work correctly prior to macOS 10.13 when using non-contiguous sets of rows. This has been fixed for all applications on macOS 10.13.

NSTableView Automatic Variable Row Heights Utilizing Autolayout

View based table views now can automatically resize a row based on the cell's contents utilizing a new property called usesAutomaticRowHeights. When set to YES, the table will utilize autolayout for the row heights. It is recommended but not required that you set the rowHeight property to provide an estimated row height for views that are not yet loaded, in order to provide a proper estimate for the scroll bars. The delegate method -tableView:heightOfRow: can still be used to provide a more specific estimated row height. Note that a rowView's height is set to the rowHeight plus intercellSpacing.height, so an estimated rowHeight should have the intercellSpacing.height subtracted from it. The default value is NO and the value is encoded.

Using this API requires one to properly set up constraints inside the cell views.

The tableview will initially size itself based on the provided rowHeight (or the delegate implementation from tableView:heightOfRow:). The first time a particular NSTableRowView is added to the table the system will load each column's cell view and add it to the row view. Constraints will be added to the cellView that control its x/y position and width. After constraints are added the method - (void)didAddRowView:(NSTableRowView *)rowView forRow:(NSInteger)row is called, allowing subclassers (or the delegate) to dynamically add additional views or constraints as needed. The rowView's fittingSize.height is then queried, and this value is used for that row's height.
User added constraints inside the cell view can be modified at anytime. Modifying constraints that affect the rowView's height will cause it to automatically change.
The height is cached by the table, but will be re-queried any time the view is brought back in.

Column resizing will automatically be restricted based on constraints, in addition to respecting the column.minWidth and column.maxWidth properties.

It is easy to debug the automatically generated row heights if a row height isn't what you expect it to be. The most common mistake is to set up constraints that allow the view's fittingSize to shrink to a 0-cell height. This means the rowView.frame.size.height will probably be a small value, such as the intercellSpacing.height (such as the default value of 2.0).

To debug heights, implement the delegate method tableView:didAddRowView:forRow: and print out the fittingSize:

```
func tableView(_ tableView: NSTableView, didAdd rowView: NSTableRowView, forRow row: Int) {
   NSLog("Height: %g", rowView.fittingSize.height)
}
```

You can break on this in the debugger and find out the fitting size:

```
(lldb) p rowView.fittingSize
(NSSize) $R1 = (width = 0, height = 2)
```

This shows you that the calculated size is too small (2.0). You can easily see what constraints are causing it to go to this small height:

```
(lldb) po rowView.constraintsAffectingLayout(for: .vertical)
 1 element
  - 0 : <NSLayoutConstraint:0x608000084c90 'NSTableRowView-Encapsulated-Layout-Height' NSTableRowView:0x6080001a23e0.height == 44 priority:500   (a
```

In this case there aren't any constraints actually affecting the height, so the size goes to the intercellSpacing.height. Adding a y position constraint, and/or height constraint will fix the problem. Running again shows the expected size and constraints that are affecting that height:

```
(lldb) p rowView.fittingSize
(NSSize) $R0 = (width = 0, height = 44)
(lldb) po rowView.constraintsAffectingLayout(for: .vertical)
 7 elements
  - 0 : <NSContentSizeLayoutConstraint:0x6200000a6b40 NSTextField:0x103428b00.height == 18 Hug:750 CompressionResistance:750   (active)>
  - 1 : <NSLayoutConstraint:0x608000087850 V:[NSTextField:0x103428b00]-(2)-[multilineLabel]   (active, names: multilineLabel:0x1034298f0 )>
  - 2 : <NSContentSizeLayoutConstraint:0x6200000a6c60 multilineLabel.height == 17 Hug:750 CompressionResistance:750   (active, names: multilineLabe
  - 3 : <NSLayoutConstraint:0x608000087710 V:|-(2)-[NSTextField:0x103428b00]   (active, names: '|':TableViewCircus.DesktopImageCellView:0x60800018
  - 4 : <NSLayoutConstraint:0x6080000877b0 V:[multilineLabel]-(3)-|   (active, names: multilineLabel:0x1034298f0, '|':TableViewCircus.DesktopImageC
  - 5 : <NSLayoutConstraint:0x60800008b720 V:|-(1)-[TableViewCircus.DesktopImageCellView:0x608000183260]   (active, names: '|':NSTableRowView:0x608
  - 6 : <NSLayoutConstraint:0x60800008b770 TableViewCircus.DesktopImageCellView:0x608000183260.bottom == NSTableRowView:0x6080001a2060.bottom - 1
```

Watch out for these common constraint problems:
* Setting a required bottom constraint in one column that forces the entire row to have a smaller height than expected.
* Forgetting to set any constraints that control the height.
* Adding constraints on the cell view that control its x/y position; this will potentially overwrite the constraints that the NSTableView adds and position the view at an incorrect

x-offset or y-offset. Interface Builder may restrict this at design time.

Increased usage of weak properties

In macOS 10.13 many object-typed properties use Objective-C weak references. This is especially true of delegate and delegate-like properties. We expect this to avoid a broad class of common pitfalls where clients forget to clear their delegate back pointers when those delegates are deallocated.

NSView.canDrawConcurrently

Introduced in 10.5, NSView.canDrawConcurrently has been nonfunctional since 10.7. In 10.13 it has been reenabled. Be on the lookout for crashes in views drawn on background threads.

NSColor

NSColor has a new property named type which aims to eventually replace the older colorSpaceName property (which was the pre-NSColorSpace way to refer to color spaces). The complete API to support this new concept is as follows:

```
typedef NS_ENUM(NSInteger, NSColorType) {
    NSColorTypeComponentBased,  // Colors with colorSpace, and floating point color components
    NSColorTypePattern,         // Colors with patternImage
    NSColorTypeCatalog          // Colors with catalogNameComponent and colorNameComponent
}

@property (readonly) NSColorType type NS_AVAILABLE_MAC(10_13);

- (nullable NSColor *)colorUsingType:(NSColorType)type NS_AVAILABLE_MAC(10_13);
```

The following APIs will be deprecated in a future release:

```
@property (readonly, copy) NSColorSpaceName colorSpaceName;
- (nullable NSColor *)colorUsingColorSpaceName:(NSColorSpaceName)name;
- (nullable NSColor *)colorUsingColorSpaceName:(nullable NSColorSpaceName)name
                        device:(nullable NSDictionary<NSDeviceDescriptionKey, id> *)deviceDescription;
```

NSColorSpaceName and NSColorType are directly convertible between each other like so:

type == NSColorTypeCatalog <—> colorSpaceName == NSNamedColorSpace
type == NSColorTypePattern <—> colorSpaceName == NSPatternColorSpace
type == NSColorTypeComponentBased <—> colorSpaceName == NSCustomColorSpace

Other colorSpaceName values, such namely NSCalibratedWhiteColorSpace, NSCalibratedRGBColorSpace, NSDeviceWhiteColorSpace, and NSDeviceRGBColorSpace, all also map back to type == NSColorTypeComponentBased.

Since colorSpaceName and colorUsingColorSpaceName:device: are among NSColor's primitive methods (methods required for overriding in subclasses), for compatibility default implementations of type and colorUsingType: in NSColor call these two older methods.

Moving forward, to support type and colorUsingType: as the new primitives, implementations of colorSpaceName and colorUsingColorSpaceName:device: check to see if they are in a subclass where type and colorUsingType: have been overridden. If so, then colorSpaceName and colorUsingColorSpaceName:device: will call those two instead.

In order to avoid recursion issues, it's important that in your custom subclasses of NSColor, implementations of type and colorUsingType: do not call super up to NSColor.

NSColor subclasses which want to work on pre-10.13 systems need to continue to implement colorSpaceName and colorUsingColorSpaceName:device:, and can also implement type and colorUsingType:. Those that need to work only on 10.13 or newer can get away with implementing just type and colorUsingType:.

New System Colors and Color Updates

NSColor exports new class properties that provide access to the various colors used by system applications where it's appropriate to include some color. These are available back to macOS 10.10:

```
@property (class, strong, readonly) NSColor *systemRedColor;
@property (class, strong, readonly) NSColor *systemGreenColor;
@property (class, strong, readonly) NSColor *systemBlueColor;
@property (class, strong, readonly) NSColor *systemOrangeColor;
@property (class, strong, readonly) NSColor *systemYellowColor;
@property (class, strong, readonly) NSColor *systemBrownColor;
@property (class, strong, readonly) NSColor *systemPinkColor;
@property (class, strong, readonly) NSColor *systemPurpleColor;
@property (class, strong, readonly) NSColor *systemGrayColor;
```

Like many other "meta" system colors, these return catalog colors whose values may vary between different appearances and releases. Do not make assumptions about the color spaces or actual colors used. If you do need to extract a CGColor (to use with a layer, say), always do it as lazily/late as possible, and update it whenever needed.

In addition, existing simple "standard" colors such as redColor, greenColor, etc, are now in the sRGB or generic gray 2.2 color spaces for applications linked against 10.13. They preserve their component values (so redColor is still 1.0, 0, 0), which means the actual color is going to appear slightly different than before.

One other change in applications linked against 10.13 is that colors returned by the older calibrated color initializers are now more likely to be autoreleased instances rather than long-lived singletons. This means apps that improperly failed to hang on to them may experience crashes. Affected methods are:

```
+ (NSColor *)colorWithCalibratedHue:(CGFloat)hue saturation:(CGFloat)saturation brightness:(CGFloat)brightness alpha:(CGFloat)alpha;
+ (NSColor *)colorWithCalibratedRed:(CGFloat)red green:(CGFloat)green blue:(CGFloat)blue alpha:(CGFloat)alpha;
+ (NSColor *)colorWithCalibratedWhite:(CGFloat)white alpha:(CGFloat)alpha;
```

NSTextAttachment

The built-in media support from NSTextAttachment now covers all file types supported by AVFoundation.

NSRulerMarker

The ruler markers for NSTextTab vended from NSLayoutManager now properly handle tab styles other than left-aligned.

Migration to CGGlyph from NSGlyph

The introduction of TextKit API for AppKit in OS X El Capitan 10.11 marked the first step of migration from NSGlyph to CGGlyph. In macOS 10.13, we're moving to the next step by soft deprecating NSGlyph and all API handling the type. Notable deprecated API includes NSGlyphGenerator and NSGlyphStorageInterface for NSTypesetter. Also, NSGlyphInfo has deprecated existing glyph types only supports the new CGGlyph-based API.

NSFont, NSBezierPath, and NSGlyphInfo introduce new CGGlyph-based API to help with the migration.

NSSpellingState

The enum type for NSSpellingStateAttributeName now has an explicit tag.

NSTextView

-readSelectionFromPasteboard:type: can now read multiple image NSPasteboardItems from the pasteboard. This allows pasting multiple images copied from iOS via iCloud pasteboard, for example.