API Design for Cocoa and Cocoa Touch

Ali Ozer
Manager, Cocoa Frameworks

What's "API"?

- Not
  - Active Pharmaceutical Ingredient
  - Armor-Piercing Incendiary
  - American Pirate Industries
  - American Pain Institute
- But
  - Application Programming Interface

Why Is API Design Important?

- APIs allow your code to interface with the system
- Well-designed APIs:
  - Make you more productive
  - Allow you to leverage existing code
  - Let you write code that works as the user expects
- APIs live for a long time
- API design is UI design for developers

Features of Good APIs

- Consistency
- Performance
- Safety
- Reusability
- Convenience

Consistency

Features of Good APIs

- Consistency
  - Naming conventions
- Performance
- Safety
- Reusability
- Convenience

Consistency

Why is it important?

- Eliminates need to refer to documentation for every single API
- Allows different subsystems to plug-in to each other more easily
- Improves performance

Naming Conventions

Classes

```
NSString
UIView
CLLocation
```

- Use prefixes
  - Protects against collisions
  - Differentiate functional areas

Naming Conventions

Methods

```
– subviews
– isEditable
– insertObject:atIndex:
```

- Focus on readability

```
if (myTextField.isEditable) ...
```

- Use camel case
- Choose clarity over brevity
- Name all the arguments

It's Better to Be Clear Than Brief

- Use

```
– removeObjectAtIndex:
```

- Rather than

```
– removeObject:
```

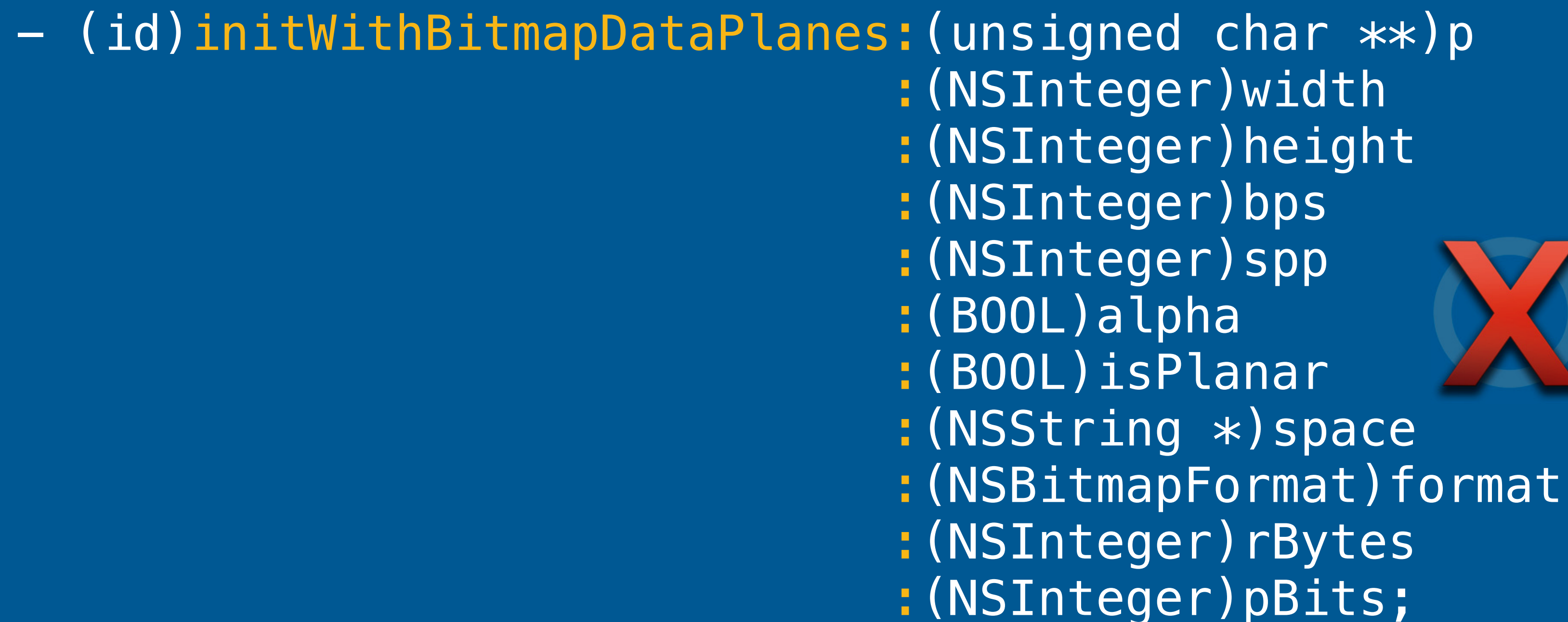
- Or

```
– remove:
```

Name All the Arguments

```
- (id)initWithBitmapDataPlanes:(unsigned char **)p
                    pixelsWide:(NSInteger)width
                    pixelsHigh:(NSInteger)height
                 bitsPerSample:(NSInteger)bps
               samplesPerPixel:(NSInteger)spp
                      hasAlpha:(BOOL)alpha
                      isPlanar:(BOOL)isPlanar
                colorSpaceName:(NSString *)space
                  bitmapFormat:(NSBitmapFormat)format
                   bytesPerRow:(NSInteger)rBytes
                  bitsPerPixel:(NSInteger)pBits;
```

Name All the Arguments



Name All the Arguments

```
id image = [[NSBitmapImageRep alloc]
    initWithBitmapDataPlanes:planes
                  pixelsWide:32
                  pixelsHigh:32
               bitsPerSample:32
             samplesPerPixel:4
                    hasAlpha:YES
                    isPlanar:NO
              colorSpaceName:NSDeviceRGBColorSpace
                bitmapFormat:NSAlphaFirstBitmapFormat
                 bytesPerRow:0
                bitsPerPixel:0];
```

Name All the Arguments

Java version, circa 2000

```
NSBitmapImageRep image = new NSBitmapImageRep(
                             planes,
                             32,
                             32,
                             32,
                             4,
                             true,
                             false,
                             NSDeviceRGBColorSpace,
                             NSAlphaFirstBitmapFormat,
                             0,
                             0);
```

Accessor Naming

```
– color                – setColor:
– isEditable           – setEditable:
– drawsBackground      – setDrawsBackground:
```

- For boolean properties that are adjectives, use “is” on the getter
- Do not embellish the getter with “get” or other verbs

```
– (UIColor *)getColor;
– (NSString *)computeFullName;
– (CGImageRef)computeThumbnailImage;
```

Accessors

Acceptable use of “get”

- Use “get” on accessors that return values by reference

```
– (void)getBytes:(void *)buffer range:(NSRange)range;
```

```
– (void)getLineDash:(CGFloat *)pattern
              count:(NSInteger *)count
              phase:(CGFloat *)phase;
```

- Callers pass NULL in for the arguments they don’t want

Accessors

Using @property

```
@property (copy) UIColor *color;
@property (getter=isEditable) BOOL editable;
@property BOOL drawsBackground;
```

Functions

```
CFRangeMake()
NSRectFill()
CGPathAddLines()
```

- Framework prefix, followed by the type or functionality area
    - "CF" + "Range" + "Make"
    - "CG" + "Path" + "Add Lines"
- The common prefix allows easier searching and sorting

Enum Values, Constants

```
UITouchPhaseBegan
NSTextCheckingCityKey
UIKeyboardWillHideNotification
```

- Similar to functions
- We also use some common suffixes
    - …Notification
    - …Key

Functions, Enum Values, and Constants

Naming conventions have evolved over time

```
CFRangeMake()
UIRectFrame()
NSRectFill()
```

```
NSMakeRange()
NSFrameRect()
NSHeight()
```

Do Not Abbreviate Arbitrarily

- Good

```
– setFloatingPointFormat:
```

- Bad

```
– setFloatingPntFormat:
```

- Ugly

```
– setFltPntFmt:
```

Acceptable Abbreviations

Be consistent

- Acceptable abbreviations

```
alloc, allocWithZone:, dealloc
int
max, min
```

- Commonly used acronyms are fine

```
PDF
USB
ASCII
URL
XML
```

Stick to Consistent Terminology

- Use

```
remove
```

- Rather than

```
delete
takeOut
doAwayWith
eliminate
exterminate
obliterate
vaporize
```

Avoid Names That Are Ambiguous

```
- sendPort
- displayName
- center
```

```
- portForSending
- localizedName
- middle
```

Block Parameter Naming

"Block" can be ambiguous

- Use only in the generic cases

```
– enumerateObjectsUsingBlock:
```

- Alternatives

```
– indexOfObjectsPassingTest:
– sortedArrayUsingComparator:
+ addLocalMonitorForEventsMatchingMask:handler:
– recycleURLs:completionHandler:
– beginBackgroundTaskWithExpirationHandler:
```

Object Ownership Across APIs

Memory management

- Object ownership is not transferred across calls
- Except return values from methods
  - Whose names begin with
    - alloc
    - new
    - copy, mutableCopy
  - retain
- You should never have to ask the question

  "Do I have to release the result from calling … ?"

Object Ownership Across APIs

Marking memory management mistakes in APIs

```
NS_RETURNS_RETAINED
```

```
- (NSString *)fullName NS_RETURNS_RETAINED;
```

- Use this for static analysis purposes, not to define or justify bad APIs

Performance

Features of Good APIs

- Consistency
- Performance
  - Impedance matching
  - Mutability
  - Concurrency
- Safety
- Reusability
- Convenience

Performance

Why is it important?

- Users like it when applications are fast
- Improves battery life

Impedance Matching

Small number of basic data types in APIs

- Improves consistency
- Allows code to fit together more easily
- Eliminates need to do conversions

Impedance Matching

Small number of basic data types in APIs

- NSString
- NSDate
- NSURL
- NSArray, NSDictionary
- UIColor/NSColor
- UIFont/NSFont
- UIImage/NSImage
- …

Impedance Matching

Not all basic data types are objects

- Use C types for numeric values
  - NSInteger, NSUInteger
  - CGFloat
  - NSNumber only to wrap these where necessary
- For widely used types where abstraction is not important, structs OK
  - CGPoint/NSPoint
  - CFRange/NSRange
  - …

Equivalent Types

Multiple types for the same concept

- What's up with…
  - CGPoint and NSPoint?
    - Equivalent typedefs
  - NSInteger, NSUInteger?
    - Enable moving to 64-bit
  - CFStringRef and NSString?
    - "Toll-free" bridged

Mutability

Mutable means changeable

- Many objects are by nature only mutable
  - NSWindow
  - UIScrollView
- Others, usually "value" objects, can exist in immutable forms
  - NSString
  - UIColor
- In some cases both make sense

Mutability

NSString versus NSMutableString

- NSString

```
NSString *str = ...;
NSString *result = [str stringByAppendingString:@“!”];
```

- NSMutableString

```
NSMutableString *str = ...;
[str appendString:@“!”];
```

Mutability

Why have immutable variants at all?

- Performance
- Simpler implementation
- Thread safety
- Easier analysis of program logic

Mutability

Which one to use in APIs?

- Immutable

```
- (NSString *)title;
```

- Mutable

```
- (NSMutableString *)title;
```

- Immutable version is almost always the right one to use
- Using @property:

```
@property (copy) NSString *title;
```

Mutability

Very few exceptions

- NSAttributedString

```
- (NSString *)string;
```

- NSMutableAttributedString

```
- (NSMutableString *)mutableString;
```

Concurrency

Achieve higher performance on multi-core machines

- Blocks are a good fit for representing concurrent work
  - They can be processed by GCD or NSOperationQueue
  - They can capture state
- Not all block usage is necessarily concurrent

Concurrency

Often there is an explicit option for concurrency

- Enumeration, sorting, and searching in collections

```
- (void)enumerateObjectsWithOptions:(NSEnumerationOptions)opts
                         usingBlock:(void (^)(id obj,
                                              NSUInteger idx,
                                              BOOL *stop))block;
```

```
[myArray enumerateObjectsWithOptions:NSEnumerationConcurrent
                    usingBlock:^(id obj, NSUInteger i, BOOL *stop) {
                        // ... process obj ...
                    }];
```

Concurrency

Often there is an explicit option for concurrency

- Receiving NSNotifications

```
  - (id)addObserverForName:(NSString *)name
                    object:(id)object
                     queue:(NSOperationQueue *)queue
                usingBlock:(void (^)(NSNotification *))block;
```

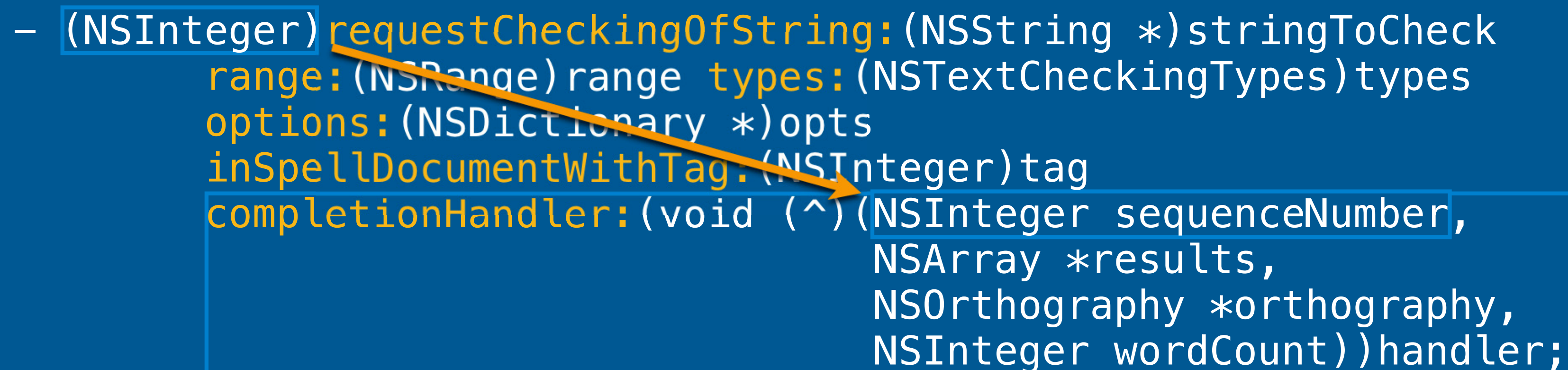
- Non-nil queue argument enables concurrent posting to observer

Concurrency

Often there is an explicit option for concurrency

• Synchronous and asynchronous checking in NSSpellChecker

```
- (NSArray *)checkString:(NSString *)stringToCheck
        range:(NSRange)range types:(NSTextCheckingTypes)types
        options:(NSDictionary *)opts
        inSpellDocumentWithTag:(NSInteger)tag
        orthography:(NSOrthography **)orthography
        wordCount:(NSInteger *)wordCount;
```



Safety

Features of Good APIs

- Consistency
- Performance
- Safety
  - Runtime errors
  - Programming errors
  - Atomicity
- Reusability
- Convenience

Safety

Why is it important?

- Reduces chance of crashes
- Allows easier debugging
- Makes more for robust and user-friendly applications

Dealing with Errors

- Two major categories of errors:
  - Runtime
  - Programming

Runtime Errors

Errors that are expected to occur

- Unreadable file
- Out of disk space
- Lost network connection
- Invalid user input
- …

Runtime Errors

Handle with return values and optional NSError

- A basic return value (BOOL or an object value) to indicate success
- In cases where reporting the error is interesting, an NSError

```
- (id)initWithContentsOfURL:(NSURL *)url
                   encoding:(NSStringEncoding)enc
                      error:(NSError **)error;
```

- NSError is often returned "by reference," as last argument
  - NULL may be passed in

Runtime Errors

Sometimes NSError is passed via other means

- Sent to delegate

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

- Passed to completion block

```
- (void)duplicateURLs:(NSArray *)URLs
    completionHandler:(void (^)(NSDictionary *newURLs,
                                NSError *error))handler;
```

Runtime Errors

Not all APIs need an NSError

- NSError returns best confined to APIs where:
  - The caller may want to take conditional action based on type of error
  - The error may be reported to the user
    - In AppKit, use NSResponder API:

```
– presentError:
– presentError:modalForWindow:delegate:
```

Programming Errors

Errors often caused by misuse of APIs

- Out-of-bounds access

```
obj = [myArray objectAtIndex:[myArray count]];
```

- Invalid argument type

```
[someView setBackgroundColor:@“0.4, 0.1, 0.1”];
```

- Invalid parameter value

```
[myTextField setStringValue:nil];
```

Programming Errors

- Not indicated via return values

```
- (ErrorCode)setBackgroundColor:(UIColor *)color;
```

- But by exceptions
  - NSException

Programming Errors

Handling exceptions

- Typically exceptions are not meant to be caught

Programming Errors

Handling exceptions

- You may still consider registering a top level exception handler
  - Alert the user that something bad happened
  - Give them a chance to save their work and quit the app

Atomicity

Thread safety for a single property

- Objective-C 2.0 properties are by default "atomic"
  - But they can be made non-atomic:

```
@property (nonatomic, copy) NSString *title;
```

- Atomic guarantees that in the presence of multiple threads that get or set a property, the property is set or retrieved fully
  - Provides a basic, low-level of thread safety for a single property
  - But it does not provide consistency between properties

Atomicity

Often a good idea to leave properties atomic

- When to consider non-atomic
  - Performance critical usages
    - Look for objc_getProperty() or objc_setProperty() in samples
  - Where you might already use a higher level of synchronization
    - Locks
    - Queues
    - Single-threaded access

Reusability

Features of Good APIs

- Consistency
- Performance
- Safety
- Reusability
  - Subclassing
  - Categories
  - Patterns for communicating changes
- Convenience

Reusability

Why is it important?

- No need to reinvent the wheel
- You only write the code that distinguishes your application
- Pieces of your application can be reused elsewhere

Reusability

Subclassing

- Fundamental object-oriented programming feature
- Not very commonly used in Cocoa and Cocoa Touch for customization
  - Many classes are "concrete" and usable as-is
  - Some classes are meant for subclassing: "abstract" or "semi-abstract"

Reusability

Some classes are meant for subclassing

- NSObject, UI/NSView, UI/NSViewController, UI/NSResponder, NSDocument
- Identify methods meant for overriding

```
- [NSObject init]
- [UIView drawRect:]
- [UIResponder canBecomeFirstResponder]
- [NSDocument readFromURL:ofType:error:]
  ...
```

Subclassing

- Some Foundation value classes are abstract, but usable
  - NSString, NSData, NSArray, NSDictionary, …
- alloc/init methods return proper instances
- But subclasses don’t work unless some methods are overridden
  - “Primitives”

Subclassing

Primitive methods

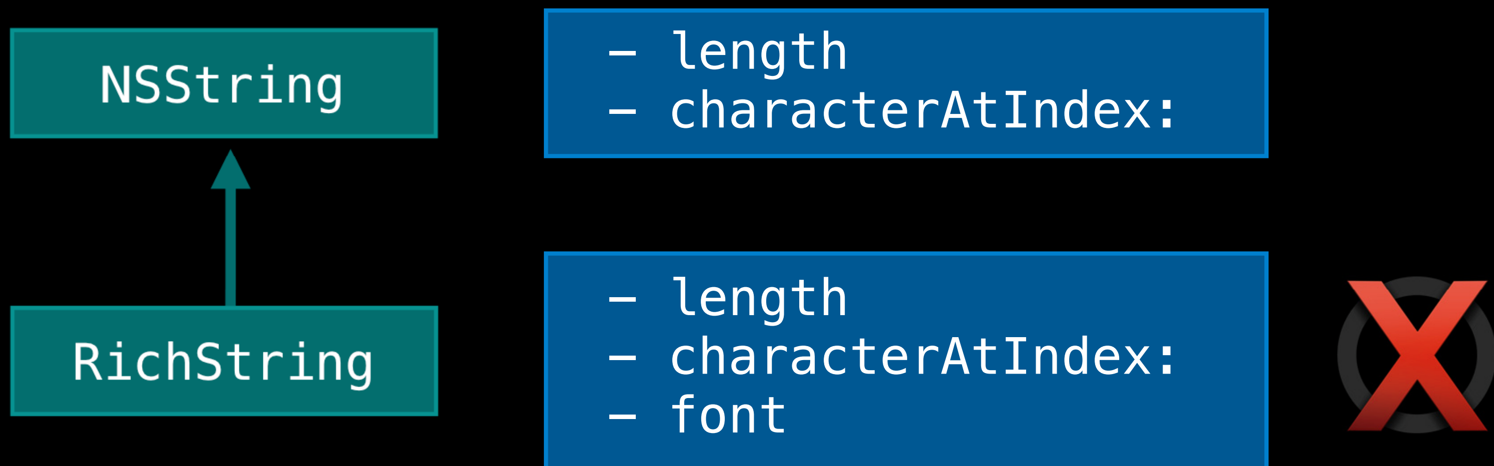
- Minimal API to implement a new subclass

```
@interface NSString : NSObject

- (NSUInteger)length;
- (unichar)characterAtIndex:(NSUInteger)index;

@end
```

Class Clusters

Why are these classes abstract?

- We do not want to encourage subclassing these classes to add additional properties
  - Changes the fundamental meaning of what the object stores



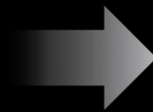
Class Clusters

Why are these classes abstract?

```
NSString *string = ...;           // "Hello"
```

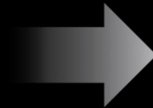
```
RichString *richString = ...;   // "Hello", Helvetica
```

```
[string isEqual:richString]
```



Sure!

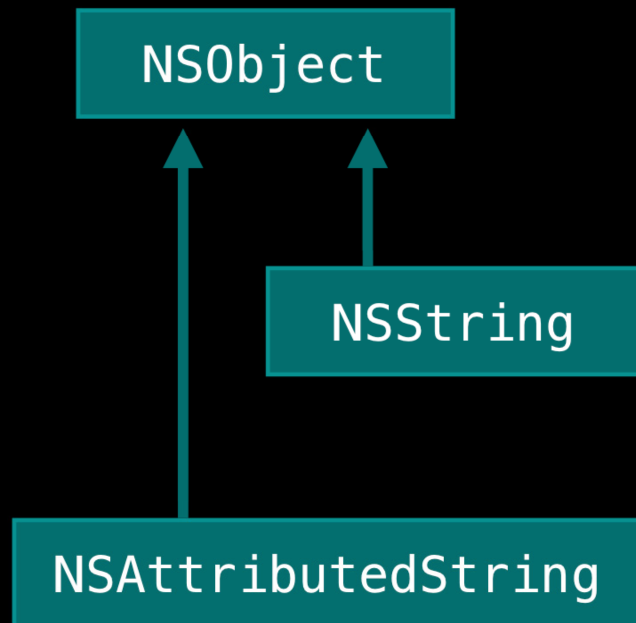
```
[richString isEqual:string]
```



No way!

Class Clusters

NSString versus NSAttributedString



- length
- characterAtIndex:

- string
- attributesAtIndex:

Class Clusters

Why would you subclass NSString, NSData, NSArray, …?

- Implement the primitives to provide alternate storage
  - Optimized for a given backing store
  - Load elements lazily

Cocoa Fundamentals Guide: "Class Clusters"   http://developer.apple.com

Extensibility

Categories

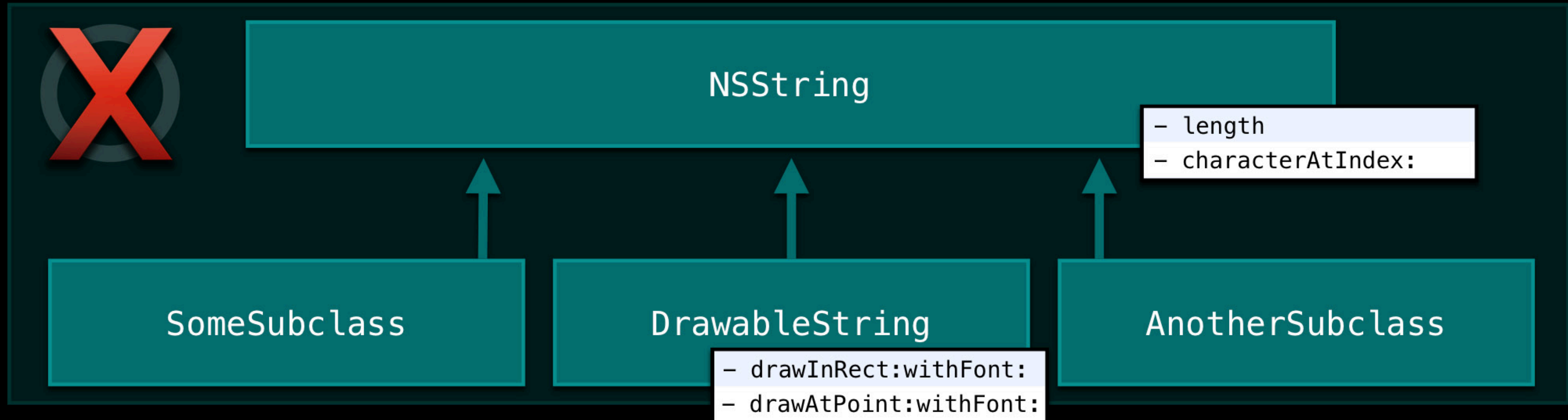
- Language feature
- Allows adding methods on existing classes
  - All instances are affected
- Enables
  - Declaring additional methods in other header files
  - Extending a class without subclassing

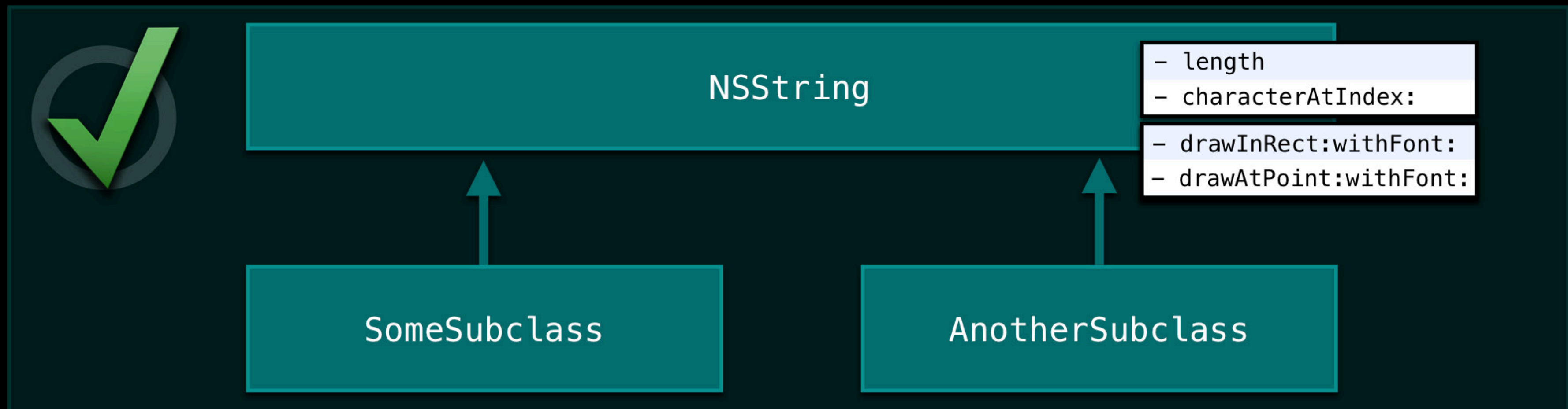
Categories

Example

- Both UIKit and AppKit add new functionality to NSString

```
@interface NSString (UIStringDrawing)

- (CGSize)drawInRect:(CGPoint)loc
              withFont:(UIFont *)font;
- (CGSize)drawAtPoint:(CGPoint)loc
               withFont:(UIFont *)font;
...
@end
```


NSString and DrawableString Subclass
NSString
– length
– characterAtIndex:
SomeSubclass
DrawableString
AnotherSubclass
– drawInRect:withFont:
– drawAtPoint:withFont:


NSString and UIStringDrawing Category
NSString
– length
– characterAtIndex:
– drawInRect:withFont:
– drawAtPoint:withFont:
SomeSubclass
AnotherSubclass

Extensibility

Patterns for communicating changes

- Delegation
- Notification
- Key-value observing
- Target-action

Cocoa Fundamentals Guide: "Cocoa Design Patterns" http://developer.apple.com

Extensibility

Delegation

- Allows an object to act on behalf of another
- Not a language feature
  - Classes explicitly support delegates

```
@interface UITextView

@property(assign) id<UITextViewDelegate> delegate;

@end
```

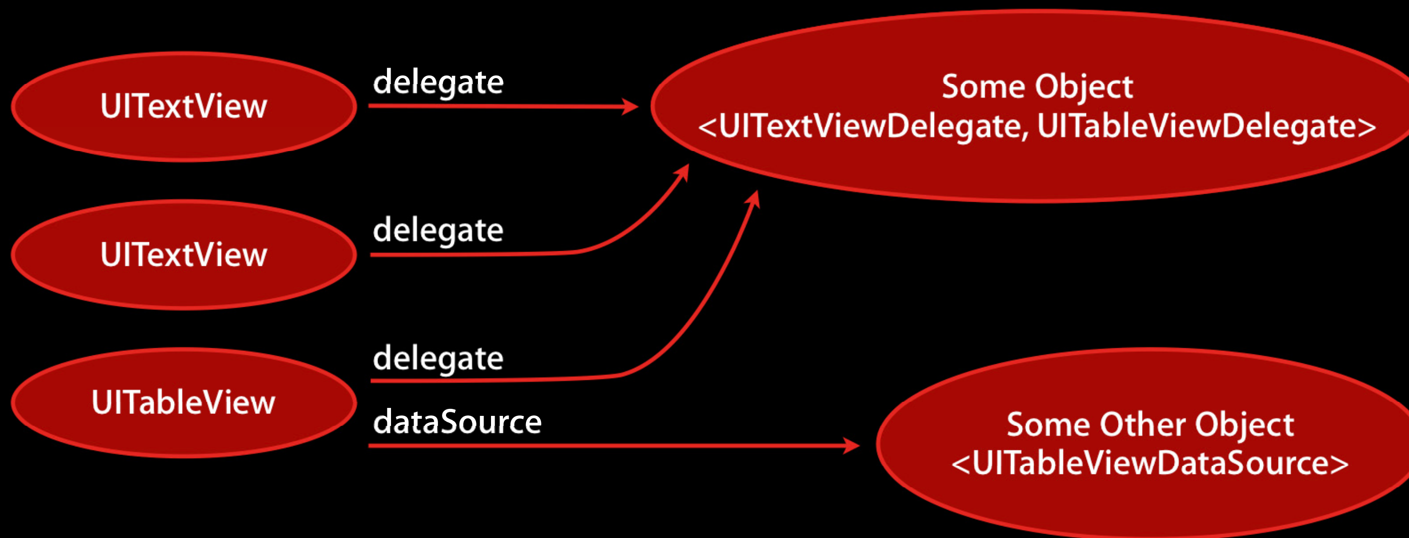
Delegation

Delegate methods declared as a protocol

```
@protocol UITextViewDelegate
@optional
- (BOOL)textViewShouldBeginEditing:(UITextView *)text;
- (BOOL)textViewShouldEndEditing:(UITextView *)text;
- (void)textViewDidChangeSelection:(UITextView *)text;
...
@end
```

Delegation

- Allows one object to help many others



- Allows proper subdivision of responsibility
- Flexible

Extensibility

Notifications

- Allows happenings to be broadcast to a set of unrelated observers
- Observers observe, but don’t interfere
- Not a language feature
  - NSNotificationCenter provides the facility
  - Classes declare the notifications they post

```
NSString *const UIPasteboardChangedNotification;
NSString *const NSWindowWillCloseNotification;
NSString *const UIKeyboardDidShowNotification;
```

Notifications

- Allows multiple observers
- Indirection between objects enables observing of
  - Any notification from a particular object
  - A specific notification from any object

Extensibility

Key-value observing

- Allows objects to broadcast updates to values of individual properties
- Not a language feature
  - Classes decide which properties they want to implement this for
  - Automatic for key-value coding compliant properties

Extensibility

Target-Action

- Allows UI controls to indicate user interaction
- Simple approach, well-suited for use in Interface Builder
  - Controls send their target a custom action
- Use of responder chain makes target-action flexible

Model View Controller

- Defines three clear functional roles for objects
- Each piece separately replaceable/customizable
- Used for overall app design, as well as at subsystem level:
  - Cocoa text system
  - Cocoa bindings architecture
  - UITableView, NSTableView
  - Cocoa Touch UIViewController

Model-View-Controller for iPhone OS	Russian Hill Wednesday 10:15AM
Advanced Cocoa Text Tips & Tricks	Russian Hill Wednesday 9:00AM

Convenience

Features of Good APIs

- Consistency
- Performance
- Safety
- Reusability
- Convenience
  - Convenience APIs
  - Blocks

WWDC Bash

There’s still time!

Convenience

Why is it important?

- Allows you to be more productive
- Makes coding fun

Convenience

Convenience APIs

- APIs that simplify or combine a number of other calls into one

```
- (NSComparisonResult)compare:(NSString *)string
                      options:(NSStringCompareOptions)opts
                        range:(NSRange)compareRange
                       locale:(id)locale;
```

```
- (NSComparisonResult)compare:(NSString *)string;
```

Convenience

Convenience APIs

- We usually consider convenience APIs only in cases where
  - The implementation is more than two lines,
  - There is additional value, or
  - There is valuable abstraction

Convenience

Additional value

```
- (NSComparisonResult)compare:(NSString *)string
                      options:(NSStringCompareOptions)opts
                        range:(NSRange)compareRange
                       locale:(id)locale;
```

```
- (NSComparisonResult)compare:(NSString *)string;
```

```
- (NSComparisonResult)compare:(NSString *)string {
    return [self compare:string
             options:0
               range:NSMakeRange(0, [self length])
              locale:nil];
}
```

Convenience

Additional value

- compare: allows easy use of sorting methods

```
[myArray sortUsingSelector:@selector(compare:)];
```

- In fact we have more NSString comparison conveniences:

```
- caseInsensitiveCompare:
- localizedCompare:
- localizedCaseInsensitiveCompare:
- localizedStandardCompare:
```

Convenience APIs

Valuable abstraction

```
– localizedStandardCompare:
```

- This is currently documented to use:

```
[string compare:otherStr
        options:NSCaseInsensitiveSearch |
                NSNumericSearch |
                NSWidthInsensitiveSearch |
                NSForcedOrderingSearch
          range:NSMakeRange(0, [str length])
         locale:[NSLocale currentLocale]];
```

- But the actual implementation will change over time

Convenience

Blocks

- Blocks do not enable anything that was impossible before
- But they bring a lot of convenience
  - Ability to specify a piece of code inline
  - Ability to capture state

Blocks as Callbacks

Make compare variants less necessary

- Instead of

```
[myArray sortUsingFunction:myNumericSort context:NULL];

...

NSComparisonResult myNumericSort(id str1, id str2, void *ctxt) {
    return [str1 compare:str2 options:NSNumericSearch];
}
```

- Can now do

```
[myArray sortUsingComparator:^(id str1, id str2) {
    return [str1 compare:str2 options:NSNumericSearch];
 }];
```

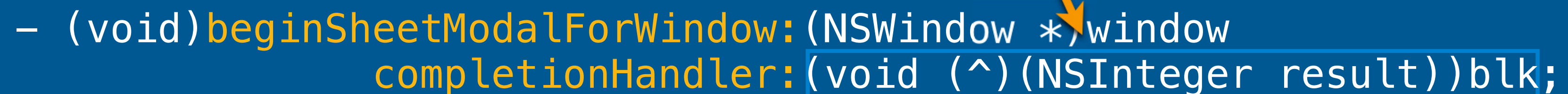
Blocks as Completions

New APIs for presenting sheets on NSSavePanel

- Leopard

```
- (void)beginSheetForDirectory:(NSString *)path
                          file:(NSString *)name
                modalForWindow:(NSWindow *)window
                 modalDelegate:(id)delegate
                didEndSelector:(SEL)didEndSelector
                   contextInfo:(void *)contextInfo;
```

- Snow Leopard



Blocks

New APIs for animations on UIView

```
+ (void)animateWithDuration:(NSTimeInterval)duration
                 animations:(void (^)(void))animations;
```

Blocks

New APIs for animations on UIView

- iPhone OS 3

```
[UIView beginAnimations:nil context:NULL];
[UIView setAnimationDuration:0.5];
view.alpha = 0.0;
[UIView commitAnimations];
```

- iOS 4

```
[UIView animateWithDuration:0.5
                 animations:^{ view.alpha = 0.0; }];
```

WWDC Bash

Almost there!

Summary

- Good API is a very important part of Cocoa and Cocoa Touch design
- Modeling your APIs as Apple's will allow your APIs to be
  - More predictable
  - Widely reusable
  - Better performing
- API Design is an evolving art

More Information

Bill Dudney
Frameworks Evangelist
dudney@apple.com

Documentation
Cocoa Fundamentals Guide
Introduction to Coding Guidelines for Cocoa
And many more!
http://developer.apple.com

Apple Developer Forums
http://devforums.apple.com

WWDC10