Objective-C APIs Are Available in Swift

```objective-c
// RunData.h
// RoadRunner
//
// Copyright (c) 2014 Apple, Inc. All rights reserved.
//
#import <Foundation/Foundation.h>
#import <CoreLocation/CoreLocation.h>

NS_ASSUME_NONNULL_BEGIN

extern NSString * const ImageDataImageName;
extern NSString * const ImageDataImage;
extern NSString * const ImageDataName;
extern NSString * const ImageDataDate;

extern CGFloat MAX_NUM_SAMPLES;

@interface RunData : NSObject

@property (readonly) NSString *name;
@property (readonly) NSMutableArray<NSDictionary<NSString *, id> *> *trackPoints;
@property (readonly, nonatomic) NSMutableArray<NSDictionary<NSString *, id> *> *imageInfo;

-(instancetype)initWithName:(NSString *)name trackPoints:(NSMutableArray<NSDictionary<NSString *, id> *> *)trackPoints;
-(NSMutableArray<NSNumber *> *)fetchVelocityDataForRoute:(NSUInteger)numSamples;
@end

NS_ASSUME_NONNULL_END
```
Objective-C APIs Are Available in Swift
Objective-C APIs Are Available in Swift
Roadmap

Working with Objective-C
Error Handling
Nullability Annotations
Lightweight Generics
“Kindof” Types
Working with Objective-C
When Are Methods Exposed to Objective-C?
When Are Methods Exposed to Objective-C?

Subclasses of NSObject

```swift
class MyController : UIViewController {
    func refresh() {
        // ...
        // ...
    }
}
```
When Are Methods Exposed to Objective-C?

Subclasses of NSObject

- Not private

```swift
class MyController: UIViewController {
    private func refresh() {
        // ...
        // ...
    }
}
```
When Are Methods Exposed to Objective-C?

Subclasses of NSObject
- Not `private`
- Not using Swift features

```swift
class MyController : UIViewController {
    func refresh() -> (Int, String)? {
        // ...
        return (status, response)
    }
}
```
Subclasses of NSObject
- Not private
- Not using Swift features
- Not for `@objc` protocols

```swift
class MyController: UIWebViewDelegate {
    func webViewDidStartLoad(v: UIWebView) {
        // ...
        // ...
    }
}
```
Subclasses of NSObject

- Not private
- Not using Swift features
- Not for @objc protocols

```swift
class MyController : UIWebViewDelegate {
    func webViewDidStartLoad(v: UIWebView) {
        // ...
        // ...
    }
}
```

warning: non-@objc method 'webViewDidStartLoad' cannot satisfy optional requirement of @objc protocol 'UIWebViewDelegate'
When Are Methods Exposed to Objective-C?

Being explicit
When Are Methods Exposed to Objective-C?

Being explicit

```swift
class MyController : UIViewController {
    private func refresh() {
        // ...
        // ...
    }
}
```
When Are Methods Exposed to Objective-C?

Being explicit

```swift
@IBOutlet, @IBAction, @NSManaged

class MyController: UIViewController {
    @IBAction private func refresh() {
        // ...
        // ...
    }
}
```
When Are Methods Exposed to Objective-C?

Being explicit

@IBOutlet, @IBAction, @NSManaged
dynamic

class MyController : UIViewController {
    dynamic private var title: String? {
        get { /* ... */ }
        set { /* ... */ }
    }
}


When Are Methods Exposed to Objective-C?

Being explicit

```swift
@IBOutlet, @IBAction,
@NSManaged
dynamic
@objc

class MyController : UIViewController {
    @objc private func refresh() {
        // ...
        // ...
    }
}
```
When Are Methods Exposed to Objective-C?

Being explicit

```
@IBOutlet, @IBAction,
@NSManaged

class MyController : UIViewController {
    @objc func refresh() -> (Int, String)? {
        // ...
        // ...
    }
}
```
When Are Methods Exposed to Objective-C?

Being explicit

@IBOutlet, @IBAction, @NSManaged
dynamic
@objc

class MyController: UIViewController {
    @objc func refresh() -> (Int, String)? {
        // ...
        // ...
    }
}

error: method cannot be marked @objc because its result type cannot be represented in Objective-C
When Are Methods Exposed to Objective-C?

Being explicit

```swift
@IBOutlet, @IBAction, @NSManaged

dynamic
@objc

class MyController: UIViewController {

    @objc func refresh() -> (Int, String)? {

        // ...

        // ...

    }

}
```

error: method cannot be marked @objc because its result type cannot be represented in Objective-C
class CalculatorController : UIViewController {
    func performOperation(op: (Double) -> Double) {
        // ...
    }

    func performOperation(op: (Double, Double) -> Double) {
        // ...
    }
}
class CalculatorController : UIViewController {
    func performOperation(op: (Double) -> Double) {
        // ...
    }

    func performOperation(op: (Double, Double) -> Double) {
        //
    }
}

error: method 'performOperation' with Objective-C selector 'performOperation:' conflicts with previous declaration with the same Objective-C selector
class CalculatorController : UIViewController {
    func performOperation(op: (Double) -> Double) {
        // ...
    }

    @objc(performBinaryOperation:)
    func performOperation(op: (Double, Double) -> Double) {
        // ...
    }
}
class CalculatorController : UIViewController {
    func performOperation(op: (Double) -> Double) {
        // ...
    }

    @nonobjc
    func performOperation(op: (Double, Double) -> Double) {
        // ...
    }
}
Function Pointers
Function Pointers

Used in C for callbacks
Function Pointers

Used in C for callbacks
Like closures, but can’t carry state
Function Pointers

Used in C for callbacks
Like closures, but can’t carry state

let fd = funopen(nil, nil, {
    [weak self] ctx, data, length in
    self?.appendData(data, length)
    return length
}, nil, nil)
Function Pointers

Used in C for callbacks
Like closures, but can’t carry state

let fd = funopen(nil, nil, {
    [weak self] ctx, data, length in
    self?.appendData(data, length)
    return length
}, nil, nil)

error: C function pointer cannot be formed from a closure that captures context
 typedef void (*dispatch_function_t)(void *);

typealias dispatch_function_t = CFunctionPointer<(UnsafeMutablePointer<Void>) -> Void>
Function Pointers
In Swift 2.0

typedef void (*dispatch_function_t)(void *);

typealias dispatch_function_t =
@convention(c) (UnsafeMutablePointer<Void>) -> Void
Error Handling
Error Handling

Objective-C

- (id)contentsForType:(NSString *)typeName
  error:(NSError **)outError;

Swift

func contentsForType(typeName: String) throws -> AnyObject
Error Handling

Objective-C

- (id)contentsForType:(NSString *)typeName
  error:(NSError **)outError;

Swift

func contentsForType(typeName: String) throws -> AnyObject
Error Handling

Objective-C

- (id)contentsForType:(NSString *)typeName
    error:(NSError **)outError;

Swift

func contentsForType(typeName: String) throws -> AnyObject

What's New in Swift
Error Handling

Objective-C

- (id)contentsForType:(NSString *)typeName
  error:(NSError **)outError;

Swift

func contentsForType(typeName: String) throws -> AnyObject
Error Handling
Return types

Objective-C
- (id)contentsForType:(NSString *)typeName
  error:(NSError **)outError;

Swift
func contentsForType(typeName: String) throws -> AnyObject
Error Handling

Return types

Objective-C

- (BOOL)readFromURL:(NSURL *)url
  error:(NSError **)outError;

Swift

func readFromURL(url: NSURL) throws -> Void
Error Handling
"AndReturnError"

Objective-C
- (BOOL)checkResourceIsReachableAndReturnError:(NSError **)error;

Swift
func checkResourceIsReachable() throws -> Void
Error Handling

Callbacks?

Objective-C

```c
-(void)moveToURL:(NSURL *)url
completionHandler:(void (^)(NSError *))handler;
```

Swift

```swift
func moveToURL(url: NSURL,
              completionHandler handler: (NSError?) -> Void)
```
“What if I call a Swift method from Objective-C and it throws an error?”
Error Handling
The secret life of ErrorType

@objc enum RequestError : Int, ErrorType {
    case Incomplete = 9001
}

@objc enum RequestError : Int, ErrorType {
    case Incomplete = 9001
}

func sendRequest(request: Request) throws {
    if !request.isComplete {
        throw RequestError.Incomplete
    }
    // ...
}
Error Handling

The secret life of ErrorType

@objc enum RequestError : Int, ErrorType {
    case Incomplete = 9001
}

NSError *error;
id result = [controller sendRequest:request error:&error];
Error Handling
The secret life of ErrorType

@objc enum RequestError : Int, ErrorType {
    case Incomplete = 9001
}

NSError *error;
id result = [controller sendRequest:request error:&error];
Error Handling

The secret life of ErrorType

@objc enum RequestError : Int, ErrorType {
    case Incomplete = 9001
}

NSError *error;
NSString *result = [controller sendRequest:request error:&error];
if (!result) {
    NSLog(@"failure %@: %ld", error.domain, error.code);
    return nil;
}
@objc enum RequestError : Int, ErrorType {
    case Incomplete = 9001
}

NSError *error;
$id result = [controller sendRequest:request error:&error];
if (!result) {
    NSLog(@"failure %@: %ld", error.domain, error.code);
    return nil;
}
@objc enum RequestError : Int, ErrorType {
    case Incomplete = 9001
}

// Generated by Swift 2.0.
typedef NS_ENUM(NSInteger, RequestError) {
    RequestErrorIncomplete = 9001
};
@objc enum RequestError : Int, ErrorType {
    case Incomplete = 9001
}

// Generated by Swift 2.0.
typedef NS_ENUM(NSInteger, RequestError) {
    RequestErrorIncomplete = 9001
};
static NSString * const RequestErrorDomain = @"...";
Handling Cocoa Errors
func preflight() -> Bool {
    do {
        try url.checkResourceIsReachable()
        resetState()
        return true
    } catch NSURLError.FileDoesNotExist {
        return true // still okay
    } catch {
        return false
    }
}
func preflight() -> Bool {
    do {
        try url.checkResourceIsReachable()
        resetState()
        return true
    } catch NSURLError.FileDoesNotExist {
        return true // still okay
    } catch {
        return false
    }
}
Handling Cocoa Errors

```swift
func preflight() -> Bool {
    do {
        try url.checkResourceIsReachable()
        resetState()
        return true
    } catch NSURLError.FileDoesNotExist {
        return true // still okay
    } catch {
        return false
    }
}
```

- NSCocoaError
- NSURLError
- AVErrror
- CKErrorCode
- CLError
- GKErrorCode
- HMErrorCode
- POSIXError
- WKErrorCode
- WatchKitErrorCode
Nullability for Objective-C
Which Pointers Can Be nil?

Objective-C
@interface UIView
@property(nonatomic,readonly) UIView *superview;
@property(nonatomic,readonly,copy) NSArray *subviews;
- (UIView *)hitTest:(CGPoint)point withEvent:(UIEvent *)event;
@end
Which Pointers Can Be `nil`?

Objective-C
```objc
@interface UIView
@property(nonatomic,readonly) UIView       *superview;
@property(nonatomic,readonly,copy) NSArray *subviews;
- (UIView *)hitTest:(CGPoint)point withEvent:(UIEvent *)event;
@end
```

Swift 1.0
```swift
class UIView {
    var superview: UIView!
    var subviews: [AnyObject]!
    func hitTest(point: CGPoint, withEvent: UIEvent!) -> UIView!
}
```
Nullability Audit

Objective-C
@interface UIView
@property(nonatomic,readonly) UIView *superview;
@property(nonatomic,readonly,copy) NSArray *subviews;
- (UIView *)hitTest:(CGPoint)point withEvent:(UIEvent *)event;
@end

Swift 1.1
class UIView {
    var superview: UIView?
    var subviews: [AnyObject]
    func hitTest(point: CGPoint, withEvent: UIEvent?) -> UIView?
}

Nullability Qualifiers for Objective-C

Indicate whether Objective-C/C pointers can be nil

- Better communicates the intent of APIs
- Allows improved static checking
- Improves usability of APIs in Swift
Nullability Qualifiers

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Usage</th>
<th>Swift</th>
</tr>
</thead>
<tbody>
<tr>
<td>nullable</td>
<td>Pointer may be nil</td>
<td>UIView?</td>
</tr>
<tr>
<td>nonnull</td>
<td>nil is not a meaningful value</td>
<td>UIView</td>
</tr>
<tr>
<td>null_unspecified</td>
<td>Neither nullable nor nonnull applies</td>
<td>UIView!</td>
</tr>
</tbody>
</table>
Nullability in the SDK

Nullability qualifiers used throughout the SDKs
New warnings for misuses of APIs with non-null parameters

```swift
[tableView deselectRowAtIndexPath: nil animated: false];
```
Nullability in the SDK

Nullability qualifiers used throughout the SDKs
New warnings for misuses of APIs with non-null parameters

```swift
[tableView deselectRowAtIndexPath: nil animated: false];
```

**warning:** null passed to a callee that requires a non-null argument
Audited Regions

Objective-C

```objective-c
NS_ASSUME_NONNULL_BEGIN
@interface UIView
@property(nonatomic,readonly) UIView *superview;
@property(nonatomic,readonly,copy) NSArray *subviews;
- (UIView *)hitTest:(CGPoint)point withEvent:(UIEvent *)event;
@end
NS_ASSUME_NONNULL_END
```

Audited regions make default assumptions about some pointers:

- Single-level pointers are assumed to be `nonnull`
- `NSError**` parameters are assumed to be nullable for both levels
Audited Regions

Objective-C

NS_ASSUME_NONNULL_BEGIN
@interface UIView
@property(nonatomic,readonly,nullable) UIView *superview;
@property(nonatomic,readonly,copy) NSArray *subviews;
- (nullable UIView *)hitTest:(CGPoint)point withEvent:(nullable UIEvent *)event;
@end

NS_ASSUME_NONNULL_END

Audited regions make default assumptions about some pointers:

- Single-level pointers are assumed to be nonnull
- NSError** parameters are assumed to be nullable for both levels

Only annotate the nullable or null_unspecified cases
Double-underscored variants of nullability qualifiers can be used anywhere

Write the nullability qualifier *after* the pointer

```c
CFArrayRef __nonnull CFArrayCreate(
    CFAllocatorRef __nullable allocator,
    const void * __nonnull * __nullable values,
    CFIndex numValues,
    const CFArrayCallBacks * __nullable callBacks);
```
Double-underscored variants of nullability qualifiers can be used anywhere

Write the nullability qualifier *after* the pointer

```
CFArrayRef __nonnull CFArrayCreate(  
    CFAllocatorRef __nullable allocator,  
    const void * __nonnull * __nullable values,  
    CFIndex numValues,  
    const CFArrayCallBacks * __nullable callBacks);
```
Double-underscored variants of nullability qualifiers can be used anywhere.

Write the nullability qualifier *after* the pointer.

```c
CFArrayRef __nonnull CFArrayCreate(
    CFAllocatorRef __nullable allocator,
    const void __nonnull* __nullable values,
    CFIndex numValues,
    const CFArrayCallBacks __nullable* __nullable callBacks);
```
Nullability for Objective-C

Used throughout the SDKs
Use it to improve your Objective-C APIs
Lightweight Generics for Objective-C
Objective-C
@interface UIView
@property(nonatomic,readonly,copy) NSArray *subviews;
@end
Collections

Objective-C
@interface UIView
@property(nonatomic,readonly,copy) NSArray *subviews;
@end

Swift
class UIView {
    var subviews: [AnyObject] { get }
}
Typed Collections

Allow collections to be parameterized by element type

- “An array of views”
- “A dictionary mapping strings to images”
Typed Collections

Allow collections to be parameterized by element type

• “An array of views”
• “A dictionary mapping strings to images”

Lightweight generics for Objective-C

• Improve expressivity of APIs
• Make collections easier to use
• Enable better static type checking
Typed Collections

Objective-C

@interface UIView
@property(nonatomic,readonly,copy) NSArray *subviews;
@end
Typed Collections

Objective-C

@interface UIView
@property(nonatomic,readonly,copy) NSArray<UIView *> *subviews;
@end
Typed Collections

Objective-C

@interface UIView
@property(nonatomic,readonly,copy) NSArray<UIView *> *subviews;
@end
Typed Collections

Objective-C

@interface UIView
@property(nonatomic,readonly,copy) NSArray<UIView *> *subviews;
@end

Swift

class UIView {
    var subviews: [UIView] { get }
}
Type Safety for Typed Collections

```c
NSURL *url = ...;
NSArray<NSURL *> *components = url.pathComponents;
```
Type Safety for Typed Collections

```
NSURL *url = ...;
NSArray<NSURL *> *components = url.pathComponents;
```

warning: incompatible pointer types initializing ‘NSArray<NSURL *> *’ with ‘NSArray<NSString *> *’
Type Safety for Typed Collections

```
NSURL *url = ...;
NSArray<NSURL *> *components = url.pathComponents;
```

```
NSMutableArray<NSString *> *results = ...;
[results addObject: @17];
```

warning: incompatible pointer types initializing ‘NSArray<NSURL *> *’ with ‘NSArray<NSString *> *’
Type Safety for Typed Collections

```objective-c
NSURL *url = ...;
NSArray<NSURL *> *components = url.pathComponents;

warning: incompatible pointer types initializing ‘NSArray<NSURL *> *’ with ‘NSArray<NSString *> *’

NSMutableArray<NSString *> *results = ...;
[results addObject: @17];

warning: incompatible pointer types sending ‘NSNumber *’ to parameter of type ‘NSString *’
```
Type Safety for Typed Collections

```c
NSArray<UIView *> *views;
NSArray<UIResponder *> *responders = views;
```
Type Safety for Typed Collections

```objective-c
NSArray<UIView *> *views;
NSArray<UIResponder *> *responders = views;

NSMutableArray<UIView *> *storedViews;
NSMutableArray<UIResponder *> *storedResponders = storedViews;
```
NSArray<UIView *> *views;
NSArray<UIResponder *> *responders = views;

NSMutableArray<UIView *> *storedViews;
NSMutableArray<UIResponder *> *storedResponders = storedViews;
[storedResponders addObject: myViewController];
Type Safety for Typed Collections

```swift
NSArray<UIView *> *views;
NSArray<UIResponder *> *responders = views;

NSMutableArray<UIView *> *storedViews;
NSMutableArray<UIResponder *> *storedResponders = storedViews;
[storedResponders addObject: myViewController];
```
Defining Lightweight Generics

@interface NSArray : NSObject
@end
Defining Lightweight Generics

@interface NSArray<ObjectType> : NSObject
@end
Defining Lightweight Generics

@interface NSArray<ObjectType> : NSObject
@end

• Type parameters specified in <…>
Parameterized Classes

@interface NSArray<ObjectType> : NSObject
- (ObjectType)objectAtIndex:(NSUInteger)index;
@end

• Type parameters specified in <…> 

• Type parameters can be used throughout that interface
Parameterized Classes

@interface NSArray<ObjectType> : NSObject
- (ObjectType)objectAtIndex:(NSUInteger)index;
- (instancetype)initWithObjects:(const ObjectType [])objects
  count:(NSUInteger)cnt;
- (NSArray<ObjectType> *)arrayByAddingObject:(ObjectType)anObject;
@end

• Type parameters specified in <…>

• Type parameters can be used throughout that interface
@interface NSDictionary<KeyType, ObjectType> (Lookup)
- (nullable ObjectType)objectForKey:(KeyType)aKey;
@end

@interface NSDictionary (Counting)
@property (readonly) NSUInteger count;
@end
Backward Compatibility

Type erasure model provides binary compatibility

• No changes to the Objective-C runtime
• Zero impact on code generation
Backward Compatibility

Type erasure model provides binary compatibility

- No changes to the Objective-C runtime
- Zero impact on code generation

Implicit conversions provide source compatibility:

```c
NSArray<NSString *> *strings = ...;
NSArray *array = ...;
array = strings; // okay, drops type arguments
strings = array; // okay, adds type arguments
```
“Kindof” Types for Objective-C
@interface UIView
@property(nonatomic,readonly,copy) NSArray *subviews;
@end

$view.subviews[0].setTitle:@“Yes” forState: UIControlStateNormal;
A Problem of Evolution

@interface UIView
@property(nonatomic,readonly,copy) NSArray<UIView *> *subviews;
@end

[view.subviews[0] setTitle:@“Yes” forState:UIControlStateNormal];
A Problem of Evolution

@interface UIView
@property(nonatomic,readonly,copy) NSArray<UIView *> *subviews;
@end

[view.subviews[0] setTitle:@“Yes” forState: UIControlStateNormal];

warning: ‘UIView’ may not respond to ‘setTitle:forState:’
id is a Weak API Contract

Many id-producing APIs mean “some subclass of NSFoo”:

```python
extern id NSApp;  // NSApplication instance
```
id is a Weak API Contract

“Kindof” types express “some kind of X”

Many id-producing APIs mean “some subclass of NSFoo”:

```objective-c
extern id NSApp; // NSApplication instance
```
**id** is a Weak API Contract

"Kindof" types express "some kind of X"

Many **id**-producing APIs mean "some subclass of **NSFoo**":

```c
extern __kindof NSApplication *NSApp; // NSApplication instance
```
id is a Weak API Contract
“Kindof” types express “some kind of X”

Many id-producing APIs mean “some subclass of NSFoo”:

```c
extern __kindof NSApplication *NSApp; // NSApplication instance
```

__kindof types implicit convert to superclasses and subclasses:

```c
NSObject *object = NSApp;  // convert to superclass
MyApplication *myApp = NSApp; // convert to subclass
NSString *string = NSApp;
```
id is a Weak API Contract

“Kindof” types express “some kind of X”

Many id-producing APIs mean “some subclass of NSFoo”:

```c
extern __kindof NSApplication *NSApp; // NSApplication instance
```

__kindof types implicit convert to superclasses and subclasses:

```c
NSObject *object = NSApp;    // convert to superclass
MyApplication *myApp = NSApp; // convert to subclass
NSString *string = NSApp;
```

warning: incompatible pointer types initializing ‘NSString *’ from ‘__kindof NSApplication *’
**id** is a Weak API Contract

“Kindof” types express “some kind of X”

Many **id**-producing APIs mean “some subclass of **NSFoo**”:

```
extern __kindof NSApplication *NSApp; // NSApplication instance
```

__kindof types implicit convert to superclasses and subclasses:

```
NSObject *object = NSApp;       // convert to superclass
MyApplication *myApp = NSApp; // convert to subclass
NSString *string = NSApp;
```

Allows messaging subclass methods:

```
[NSApp praiseUser];          // invokes -[MyApplication praiseUser]
```
“Kindof” Types Are a More Useful `id`

**Objective-C**

```objective-c
@interface NSTableView : NSControl
-(nullable id)viewAtColumn:(NSInteger)column row:(NSInteger)row
                         makeIfNecessary:(BOOL)makeIfNecessary;
@end
```

**Swift**

```swift
class NSTableView : NSControl {
    func viewAtColumn(column: Int, row: Int, makeIfNecessary: Bool)
       -> AnyObject?
}
```
“Kindof” Types Are a More Useful id

Objective-C

@interface NSTableView : NSControl
-(nullable __kindof NSView *)viewAtColumn:(NSInteger)column row:(NSInteger)row makeIfNecessary:(BOOL)makeIfNecessary;
@end

Swift

class NSTableView : NSControl {
    func viewAtColumn(column: Int, row: Int, makeIfNecessary: Bool) -> AnyObject?
}
“Kindof” Types Are a More Useful `id`

Objective-C

@interface NSTableView : NSControl
-(nullable __kindof NSView *)viewAtColumn:(NSInteger)column
    row:(NSInteger)row
    makeIfNecessary:(BOOL)makeIfNecessary;
@end

Swift

class NSTableView : NSControl {
    func viewAtColumn(column: Int, row: Int, makeIfNecessary: Bool)
        -> NSView?
}

"Kindof" Types Are a More Useful `id`
“Kindof” Types with Lightweight Generics

Objective-C

@interface UIView
@property(nonatomic,readonly,copy) NSArray<UIView *> *subviews;
@end

[view.subviews[0] setTitle:@“Yes” forState:UIControlStateNormal];
UIButton *button = view.subviews[0];
“Kindof” Types with Lightweight Generics

Objective-C

@interface UIView
@property(nonatomic,readonly,copy) NSArray<__kindof UIView *> *subviews;
@end

[view.subviews[0] setTitle:@“Yes” forState:UIControlStateNormal];
UIButton *button = view.subviews[0];
Should I Use `id` in an API?

Most idiomatic uses of `id` can be replaced with a more precise type

- `instancetype` for methods that return “self”
- Typed collections for most collections
- `__kindof X *` for “some subclass of X”
- `id<SomeProtocol>` for any type that conforms to `SomeProtocol`
Should I Use `id` in an API?

Most idiomatic uses of `id` can be replaced with a more precise type

- `instancetype` for methods that return “self”
- Typed collections for most collections
- `__kindof X *` for “some subclass of X”
- `id<SomeProtocol>` for any type that conforms to `SomeProtocol`

Use `id` when you truly mean “an object of any type”:

```swift
@property (nullable, copy) NSDictionary<NSString *, id> *userInfo;
```
Swift and Objective-C are co-designed to work together

- Xcode helps you move between the two languages

Modernize your Objective-C!

- New Objective-C language features improve API expressiveness
- Find problems faster with better type safety in Objective-C
- Makes your Objective-C interfaces beautiful in Swift
More Information

Swift Language Documentation
http://developer.apple.com/swift

Apple Developer Forums
http://developer.apple.com/forums

Stefan Lesser
Swift Evangelist
slesser@apple.com
## Related Sessions

<table>
<thead>
<tr>
<th>Session</th>
<th>Location</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>What's New in Swift</td>
<td>Presidio</td>
<td>Tuesday 11:00AM</td>
</tr>
<tr>
<td>What's New in Cocoa</td>
<td>Presidio</td>
<td>Tuesday 1:30PM</td>
</tr>
<tr>
<td>Improving Your Existing Apps with Swift</td>
<td>Pacific Heights</td>
<td>Tuesday 3:30PM</td>
</tr>
<tr>
<td>Swift in Practice</td>
<td>Presidio</td>
<td>Thursday 2:30PM</td>
</tr>
<tr>
<td>Building Better Apps with Value Types in Swift</td>
<td>Mission</td>
<td>Friday 2:30PM</td>
</tr>
<tr>
<td>Labs</td>
<td>Location</td>
<td>Time</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>Swift Lab</td>
<td>Developer Tools Lab A</td>
<td>Tuesday 1:30PM</td>
</tr>
<tr>
<td>Cocoa Lab</td>
<td>Frameworks Lab B</td>
<td>Tuesday 2:30PM</td>
</tr>
<tr>
<td>Swift Lab</td>
<td>Developer Tools Lab A</td>
<td>Wednesday 9:00AM</td>
</tr>
<tr>
<td>Foundation Lab</td>
<td>Frameworks Lab A</td>
<td>Wednesday 9:00AM</td>
</tr>
<tr>
<td>Swift Lab</td>
<td>Developer Tools Lab A</td>
<td>Wednesday 1:30PM</td>
</tr>
<tr>
<td>Swift Lab</td>
<td>Developer Tools Lab A</td>
<td>Thursday 9:00AM</td>
</tr>
<tr>
<td>Swift Lab</td>
<td>Developer Tools Lab A</td>
<td>Thursday 1:30PM</td>
</tr>
</tbody>
</table>