iOS Dev Accelerator Week6 Day1

- Objective-C
- Properties, Header Files
- Pointers
- Comparison to Swift
- Using Swift and ObjC together

Objective-C

- A subset of C
- An object-oriented language
- Uses message passing, inspired by SmallTalk
- Dynamically typed (with optional static typing)

```
super.viewDidLoad()
```

```
[super viewDidLoad];
```

```
var persons: [People] = []
```

```
NSMutableArray *persons = [[NSMutableArray alloc] init];
```

main.m

- Entry Point for Objective-C application
- UIApplication is initialized with a reference to your App Delegate

```
#import <UIKit/UIKit.h>
#import "PASAppDelegate.h"

int main(int argc, char * argv[])
{
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil,
                                 NSStringFromClass([PASAppDelegate class]));
    }
}
```

```objc
@interface Person : NSObject

- (NSString *)fullName;

@end
```

Header

```objc
@implementation Person

- (NSString *)fullName {
    // TODO: implement
    return nil;
}

@end
```

Implementation

Demo

Method Prototypes

- Objective-C requires methods to be defined in header files
- Have same signature as functions, but without a body

```
- (void)updateDataStore;

- (void)setPersons:(NSArray *)persons;

- (BOOL)shouldTerminateWithOptions:(id)options;
```

Methods

- Instance method definitions prefaced with **-** sign.

```
- (NSString *)fullName;

- (void)startAnimations;
```

- Class method definitions prefaced with **+** sign.

```
+ (UIApplication *)sharedApplication;

+ (NSString *)formattedString;
```

Methods

- Return type placed before method name, in parenthesis
- Method name follows return type
- A optional colon : follows the name, before any parameters
- Each parameter is named, prefaced by its type

```
- (NSDictionary *)parseResponseData:(NSData *)response;

- (NSString *)contextAtXPath:(NSString *)xpath parser:(TFHpple *)parser;
```

Properties

- Defined using the `@property` keyword
- Defined in header file, within `@interface` blocks
- Modified using attributes, following keyword
- By default, properties are `(atomic, readwrite, retain)`
- Setters and Getters automatically generated
- Can specify custom names for getter/setter

Property Modifiers

- Property modifiers specify:
  - Thread restrictions (nonatomic, atomic)
  - Access restrictions (readwrite, readonly)
  - Memory management (retain, copy, assign, strong, unsafe_retained)

```
@property (nonatomic, copy) NSString *title;
@property (nonatomic, strong, readonly) NSArray *cars;
@property (atomic, assign, getter=isActive) BOOL *active;
```

Properties and Dot Notation

- Objective-C allows the use of donation, instead of [] brackets, when accessing and setting properties. This is the only time you can use it

```
self.view.backgroundColor = [UIColor redColor];
```

—Same As—

```
[[self view] setBackgroundColor:[UIColor redColor]];
```

Creating new instances

- Create instances using the `alloc/init` pattern
- First an `alloc` message is sent to the Class
- This returns an instance of the Class
- Then an `init` message is sent to new instance
- This returns an initialized instance of the Class

Creating new instances

```
[[NSArray alloc] init];
```

- Call to **`alloc`** will allocate enough memory for instance and its properties
- `alloc` returns an `id` type— could be anything
- Call the `init` will set suitable initial values for properties.

Implementing init

- Called [super init] and assign the result to self
- If self is not nil setup initial state and return self

```
- (instancetype)init {
    if ((self = [super init])) {
        // TODO: implement
    }

    return self;
}
```

dealloc

- Since ARC, you don’t need to explicitly release memory in dealloc
- But dealloc is still a good place to handle teardown operations
- Do not call super

```
- (void)dealloc {
    //
}
```

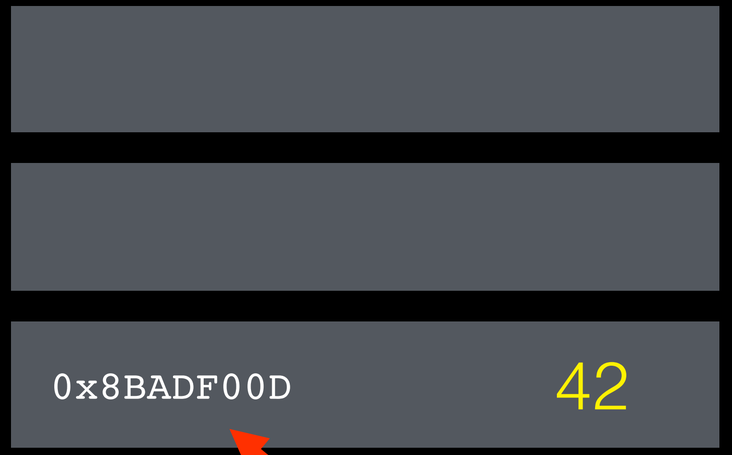
Demo

Pointers

- Creating variables, or allocating objects, stores values in memory.
- The location of a value in memory is called the *memory address*
- A pointer is a variable with the memory address of *another* value in memory
- Thus, a pointer *points to* the location in memory where a value is stored.

Pointers

```
int x = 42;
```




```
int *y = &x;
```

* is the **value at address** operator

& is the **address of** operator

Pointers

- Allow functions to modify values by passing a memory address of that value
- Accessing the value referenced by a pointer is called *dereferencing a pointer*
- Allows varying sizes of memory to be referred to by single value; starting address

The @ sign

- Used to denote certain keywords:
  - `@interface, @implementation, @property, @protocol, @selector, @end`
- Used for `NSArray`, `NSDictionary`, and `NSNumber` shorthand
  - `@[view1, view2, view3], @{@“key”,@(42)}`
- Used to define instances of `NSString`
  - `NSString *organization = @“Code Fellows”;`

Messages

```
[NSUserDefaults standardUserDefaults];
```

```
[receiver message]
```

```
[NSArray arrayWithObject:@42];
```

```
[receiver message:argument]
```

Message Sending

- ObjC runtime keeps a list of method names and address locations
- The method name is the **`Selector`**
- When message is sent, ObjC checks if the receiver responds to the message (**`respondsToSelector:`**)
- All messages sent to **`nil`** are ignored.

Selectors

- Name used to select a method to execute
- Used to identify a method
- When used within ObjC, acts like dynamic function pointer

```
SEL first = @selector(methodName);
SEL second = NSSelectorFromString(@"methodName");
```

Demo

Blocks

- Introduced in ObjC 2.0
- Similar to Closures in Swift
- Used extensively in Cocoa APIs that accept completion blocks
- Can be anonymous or stored local variables or properties
- Very easy to introduce retain cycle, if you’re not careful.
- Syntax not easily to remember: http://goshdarnblocksyntax.com

Blocks

- Begin a block with the caret operator ^
- Declare a return type; typically void on completion blocks
- Include types of parameters. Do not need to be named in method prototype

```
- (void)loadRoutes:(void (^)(NSArray *, NSError *))success;
```

```
[[RFRoutesModel sharedModel] loadRoutes:^(NSArray *routes, NSDictionary *err) {
    //
}];
```

Precompiled Header (PCH)

- A Header file compiled an into intermediate form that is faster for the compiler to process
- Reduces compilation time when applied to large header files
- Xcode generates one automatically: ProjectName-Prefix.pch
- Use it to include Libraries or 3rd party code in your App

Precompiled Header (PCH)

```
//
//  Prefix header
//
//  The contents of this file are implicitly included at the beginning
//  of every source file.
//

#import <Availability.h>

#ifndef __IPHONE_5_0
#warning "This project uses features only available in iOS SDK 5.0 and later."
#endif

#ifdef __OBJC__
    #import <UIKit/UIKit.h>
    #import <Foundation/Foundation.h>
#endif
```

#import or #include

- #include comes from C
- #import was added to Objective-C
- Using #import ensures a file is only ever included once; No recursive imports
- You a free to decide which to use.
- Good rule of thumb: #import for ObjC, #include for C libraries

#if, #ifdef and friends

- Preprocessor Macros in the C language
- #if works like a normal if statement. If someOption, do X
- #ifdef looks for a existing value or macro that is #defined
- #ifndef check if a value of macro is NOT #defined

#if, #ifdef and friends

- At compile time, only one branch is taken, depending on #defines
- For the branch not taken, the code won't even be compiled
- Compile will rewrite source code, at compile time, based on which preprocessor macro is evaluated

```
#ifdef DEBUG
// App is running in DEBUG mode
#else
// App is a real application, in production mode
#endif
```

Using const

- Declare constants using the const keyword
- Any type can be a constant
- Value cannot change, one declared.
- Think of let keyword in Swift

```
NSString * const ReuseIdentifier = @“ReuseIdentifier";

int const limit = 100;
```

Sharing const

- Expose constants in a header file to use from multiple points
- Define constants in an implementation file
- Useful technique to define constants and use across the app
- Using proper constants allows for equality tests using pointers vs #define

Sharing const

- Create Constants.h and Constants.m
- In Constants.h, expose the const variables

```
FOUNDATION_EXPORT NSString *const MyFirstConstant;
```

- In Constants.m, define the const variables

```
NSString *const MyFirstConstant = @“FirstConstant”;
```

ObjC vs Swift

- Swift uses static (rigid) types, ObjC uses dynamic (loose) types
- Swift uses function (vtable) lookup, ObjC uses messages passing
- ObjC wants implicit #import statements for dependencies; Swift doesn’t
- ObjC allows for nil messaging, Swift doesn’t and doing so causes a runtime error

Swift and ObjC together

- Use bridging header
- #import AppName-Swift.h, Xcode generates this
- Swift classes with @objc
- Careful of type conversions

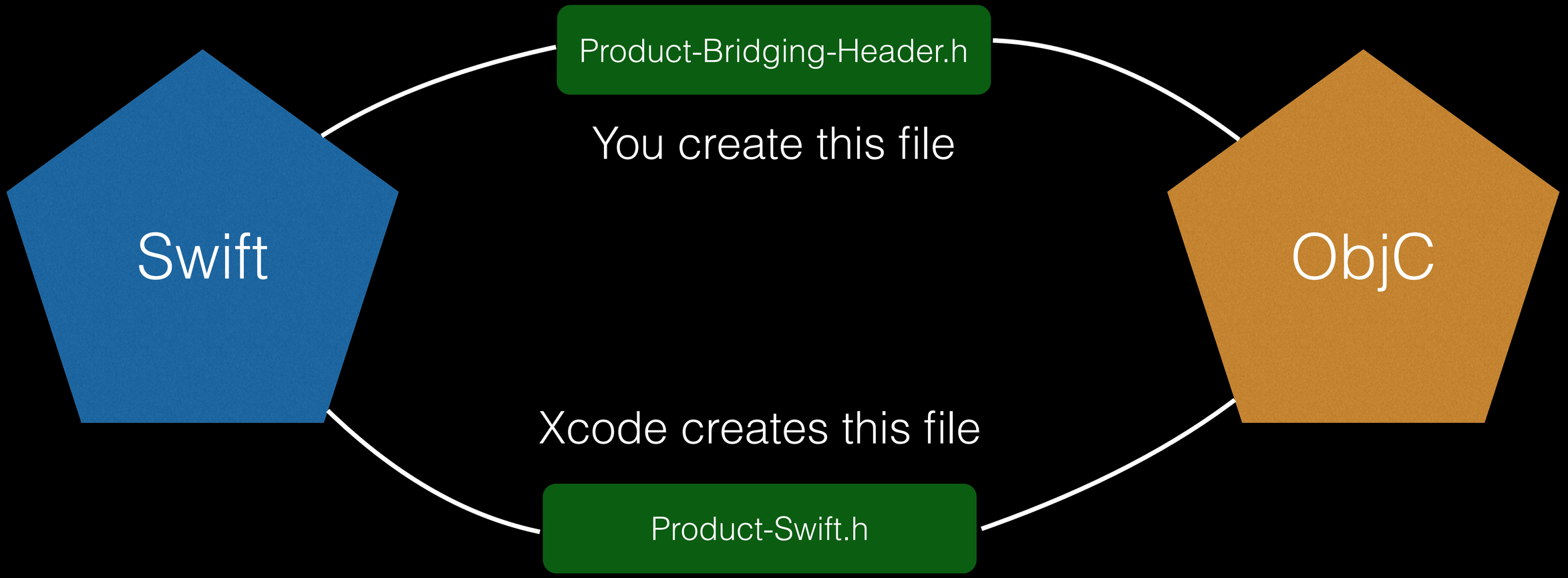
Type Bridging

Swift	Objective-C
`Bool`	`BOOL`
`Int`	`NSInteger`
`String`	`NSString *`
`Selector`	`SEL`
`AnyObject`	`id`
`[AnyObject]`	`NSArray *`
`[NSObject: AnyObject]`	`NSDictionary *`
`{}`	`^{}`
`AnyClass`	`Class`

Swift features not available to ObjC

- Generics
- Tuples
- Swift Enums
- Swift Structs
- Top-level Swift functions
- Global variables defined in Swift
- Typealiases defined in Swift
- Swift-style variadics
- Nested types
- Curried functions

Bridging Header files



Demo