Udacity Course

Objective-C vs. Swift

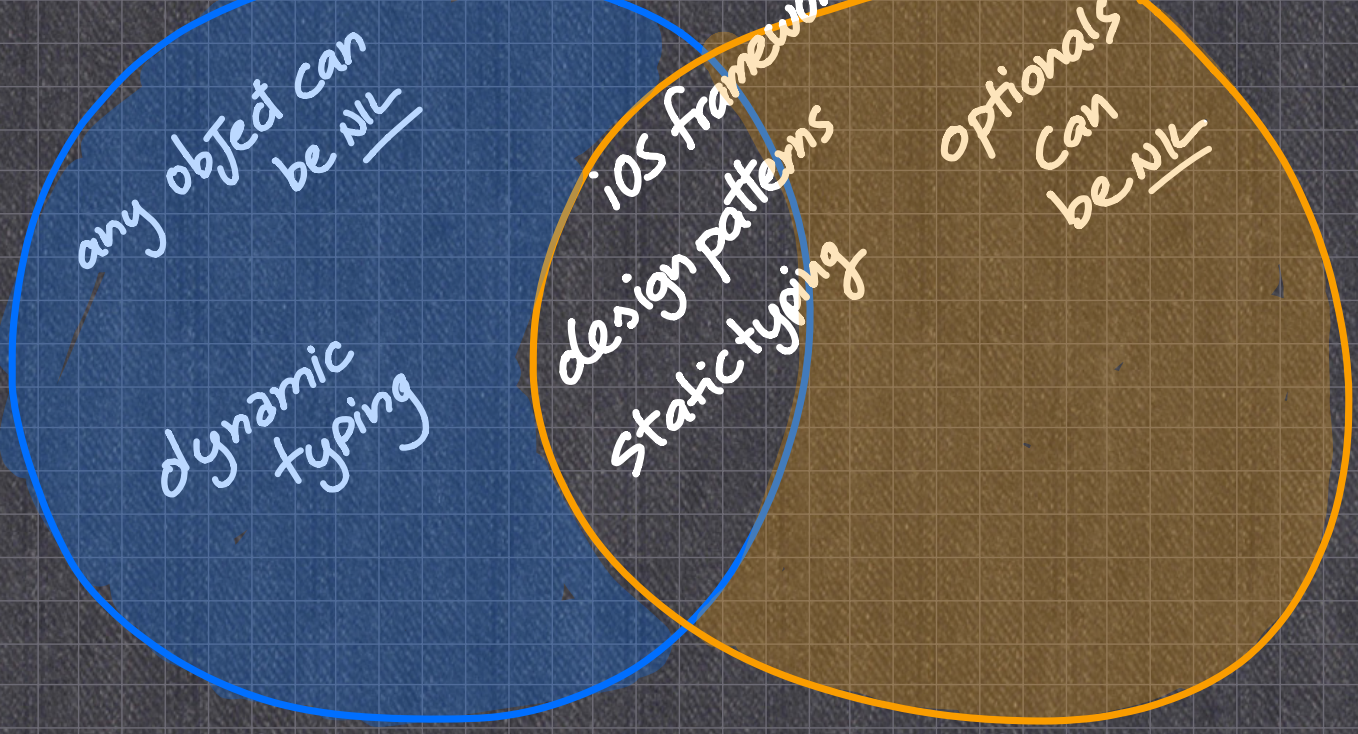
- goal: dive into an unfamiliar Obj-C codebase and migrate features from Obj-C to Swift. We'll be focusing on:

- Compare and contrast
- writing classes
- methods and messages
- port an app from Obj-C to Swift

! Obj-C will stick around

it's good to know the basis of obj-C - for example (all variables can be nil in obj-C)

Compare AND Contrast



- many classes in the iOS framework are written in ObJ-C but are bridged

A FAMILIAR METHOD FROM UIKIT

Swift

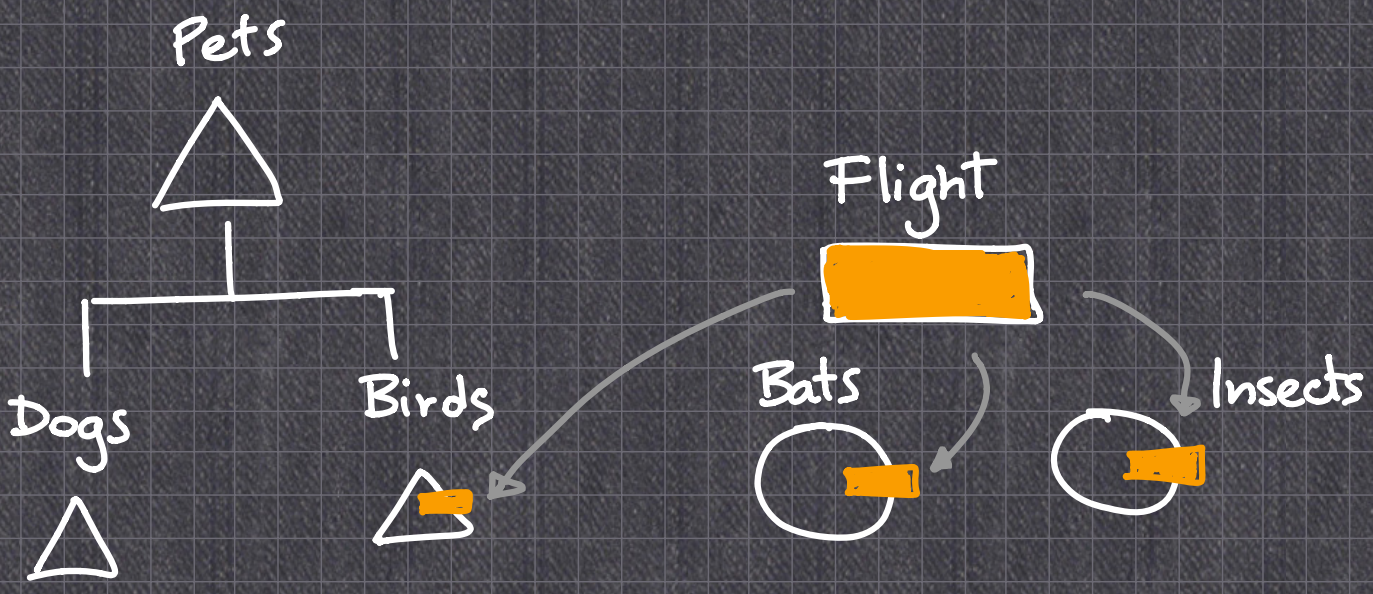
```
override func viewWillAppear(animated:Bool) {
    → super.viewWillAppear(animated)
    → displayResult()
}
```

ObJective-C

```
- (void) viewWillAppear:(BOOL) animated {
```

```
    → [super viewWillAppear:animated];
    → [self displayResult];
}
```

- Also, both obj-c and swift protocols enable shared functionality Across classes



- both languages have single inheritance

Differences

- handling nil

- Swift treats nil values with a great

deal of care

- ObJ-C is much less concerned about nil values. The compiler just doesn't care. In general, we don't want our program to do nothing when it's supposed to do something.

- Mutability

- in Swift, constants are constants and we are encouraged to favor constants.

- in ObJ-C, mutability is limited by certain classes and property attributes. By default, all objects are mutable.

ObJ-C classes with limited mutability include all Foundation collections. For example, NSstrings, NSArray, NSDictionary, NSSet are not mutable. Mutable types are NSMutableString, NSMutableArray, NSMutableDictionary, NSMutableSet, etc.

EXAMPLE:

- NSArray: I cannot append an element to the array because it's not mutable.

- we need to use NSMutableArray

- in Swift there's also a preference to use structs instead of classes.

- **Typing** [in Obj-C objects are NOT typed]

  - Swift is statically typed (= the type of every entity is known at Compile time).

- objective-C uses a combination of static and dynamic typing. It's required for every object to have a type. But sometimes the type that's declared is id. (= "to be determined")

- **Dynamic Method Resolution**

  - Swift method binding happens at

• In Swift, method binding happens at compile-time.

• In obj-C methods and the objects invoking them are not bound until runtime.

• SEE: method swizzling

BUT invoking a method that doesn't exist is not usually a good idea.
unrecognized selector sent to instance :—
[run-time error] instead do:

```
if ([item respondsToSelector:@selector(play)])
                                              {
    item.play();
}
```

CLASSES

• Swift: structs, classes, enums

• Obj-C: primarily classes
all obj- classes descend from NSObject.

House Class (Header file)

```
@interface House : NSObject
@property (nonatomic) NSString *address;
@property (nonatomic) int numberOfBedrooms;
@property (nonatomic) bool hasHotTub;
```

Can be atomic or nonatomic
this distinction relates to how properties are handled in multithreading [default is atomic]

- "readwrite" in obJ-C ≈ var in Swift
- "readonly" in obJ-C ≈ let in Swift
- use "copy" for strings → we don't want to leave properties of our classes vulnerable to unintentional changes.

```
House *myHouse = [[House alloc] init];
```

- alloc and init are always written together-

allocates memory space

initialize object instance

Writing a custom initializer:

instance of the class in which the method is found

```
@implementation House

-(instancetype)initWithAddress:(NSString*)address
                                                {
    self = [super init];
    if (self) {
        _address = [address copy];
        _numberOfBedrooms = 2;
        _hasHotTub = false;
    }
    return self;
}
```

we call the init method of the superclass.

default values

@end

General Form of Method:

```
- (returnType) methodName: (parameterType*)
                              parameterName
                                  {

    body;
}
```

Now make the initializer visible outside of the class:

```
@interface House: NSObject
- (instancetype) initWithAddress: (NSString*)
                                    address;
@end
```

What is self?

Self is a pointer to the object at hand. When you are writing code inside of a class, self is a pointer to the object at hand.

Enums in ObJ-C

macro → NS_Enum
type → NSInteger
enum name → Direction

```
typedef NS_Enum (NSInteger, Direction) {
        North,
        South,
        East,
        West
};
```

strong, weak references

- strong [default] references are used for properties that are the primary responsibility of the class = owned by the class (e.g. house owns bedrooms).

! use weak references for

1) delegates
2) subviews of the main view

this is very important to avoid memory cycles

Sample declaration:

```
RPSTurn *playersTurn = [[RPSTurn alloc]
                        initWithMove:
                        playersMove];
```

notice that the objective-C compiler doesn't infer types the way the Swift compiler does, so every time we create a new variable we need to first indicate its type.

Calling Methods

```
[receiver message];
```

e.g. `[RPSTurn alloc];`

Behind the Scenes: every time a message is sent to an object, the

`objc_msgSend()` method is called -

In ObJ-C the message and the receiver are not bound until Compile time [this is opposed as to what happens in C++]. This is why we say:

in Swift]

"Send a message"

Dot notation / Getters and setters

- obj-c generates getters/setters automatically Just like Swift.

CONTROL FLOW

- Switch Statements

- they only work with integers
- you need a break statement.

```
switch (integer) {
case 0:
        statement;
        break;
case 1:
        statement;
        break;
default:
        statement;
        break;
}
```

}

- **if statement**

always have parenthesis around your condition.

```
if (condition) {
    statements;
} else {
    statement;
}
```

QUIZ: what is the principal difference between a function call and a message? How does messaging work in Objective-C?

The principal difference is that a function and a particular method implementation are coupled at compile time, but a message and its receiver are not linked until runtime. At runtime a call to the NSObject method, `objc_msgSend()` makes a connection between receiver

and message-

OBJECTIVE C MIGRATION TO SWIFT

what's analogous to a category in objC?
an extension!

Handling Untyped Arrays

- consider the following piece of code:

```
required init?(coder aDecoder: NSCoder){
  self.url = aDecoder.decode .... as? URL
  -----                          as? URL
  --------                       as? String
  ----------                     as? UIImage
```

! if we inspect the documentation for decodeObjectForKey we see that it returns an object of type AnyObject?

- in Swift we want objects to have specific types. Compensating for differences like this in typing is a common interoperability challenge —

Objective-C Preprocessor Macros

- It's not part of the compiler but is a separate step in the compilation process.
- It instructs the compiler to do required pre-processing before actual compilation —

Example:

pound symbol

```
#define GIFURL
[[NSSearchPathForDirectoriesInDomains(NSDo
cumentDirectory, NSUserDomainMask, YES)
objectAtIndex:0]
stringByAppendingPathComponent:@"savedGifs
"]
```

which becomes this in Swift:

```
var gifsFilePath: String {
    let directories = NSSearchPathForDirectoriesInDomains(.documentDirectory, .userDomainMask, true)
    let documentsPath = directories[0]
    let gifsPath = documentsPath.appendingFormat("/savedGifs")
    return gifsPath
}
```

Common Interoperability Challenges

- There are some problems that you may encounter most frequently:

Objective-C consuming Swift code

- Swift → Objective-C. Xcode doesn't make it possible to create headers to use structs and enums (which are specific to Swift) in Objective-C. So in order to have your objective-C consuming your Swift code you have to create wrapper classe which leads to a larger and more complex codebase.

Porting from Objective-C to Swift

- it's not that difficult because porting anything from Objective-C to Swift is just a matter of changing

the syntax –

- The point of porting has also to do with software architectural changes (and not just with syntax per se). There can be a lot of whiteboarding involved because there are lots of possible solutions but likely only one best solution –

- The only reason someone could possibly write new objective-C code would probably be to do method swizzling. Though Swift gives you the power of map, flatMap, generics, structs and value types –

#1 NIL VALUES

- any object can be nil in objective-C –

If your Swift code is consuming Objective-C code you constantly need to check for nil values –

- by default, Swift treats every variable coming from objective-C as an implicitly unwrapped optional which are dangerous (we don't want to accidentally unwrap nil values).

Nullability Annotations: clarify which variable has the potential of carrying nil values -

- _Nullable → can have a nil value
- _Nonnull → not expected to be nil

Example:

```
@interface User : NSObject
@property (nonatomic) NSString * _Nonnull name;
@property (nonatomic) UIImage* _Nullable avatar;
@property (nonatomic) NSString* _Nonnull email;
```

-(instancetype _Nullable)initWithName ----

↑ this method can return nil
So we add _Nullable

@end

#2 The Hazards of 1D and AnyObject

- the type id is like saying "to be determined" or "this object can be of any type". At runtime a more specific type is returned—
- when Swift is consuming objective-C code, objects of type id are given the type AnyObject (which can hold a variety of types). It's not in the spirit of the Swift language to allow for such flexible types (although it may work).

Plus AnyObject can leave you vulnerable with unrecognized selector errors when dealing with untyped arrays.

- Delegates and objects inside of arrays

usually have type id.

- we can avoid unrecognized selector errors using optional chaining = place a question mark after the method (selector name) which is equivalent to the respondsToSelector: in Objective-C — use if-let to handle two different conditions:

```
let key = backpack.keys[0]
if let unlocked = key.openDoor?() {
    //success
} else {
    // object does not respond to selector
}
```

OR CAST TO A SPECIFIC TYPE:

```
let key = backpack.keys[0]
if let key = key as? key {
    key.openDoor()
    //success
} else {
    // object does not respond to selector
```

}

Lightweight Generics help us provide information about what a obj-c collection contains when they are defined. For example:

```
@property (nonatomic) NSArray<NSString *> * awards;
```

which is equivalent to:

```
open var awards: [String]
```

This also applies to dictionaries:

```
@property (nonatomic) NSDictionary<NSString *,
                                   NSUrl *>
                                   * quotes;
```

which is equivalent to:

```
open var quotes: [String : NSUrl]
```

- keep in mind, though, that lightweight generics are only supported by NSArray,

NSSet and NSDictionary. (They don't work with NSMutableArrays) -

#3 Error Handling

- in obj-C a pointer to an error is generally passed as the last parameter to a method that could fail. If a method fails a value can be assigned to the error pointer.
- Swift uses the keyword throws rather than an error parameter. You handle errors using do-catch.
- Errors travel between the two programming languages remarkably well.

```
- (BOOL) performStuffWithError: (NSError**) error;
```

(BOOL): removed in Swift

performStuffWithError: (NSError**) error: removed in Swift

pass with ampersand

Obj-C

Swift

```
public func performStuff() throws
```

- bottom line: the interoperability for

error handling is mostly handled for you-
To make Swift errors work in obj-C just
add @objc in front of the enum
that must have an integer type:

```
@objc enum MyError : Int, ErrorType {
    case TypicalReason = 997
    case WeirdReason = 998
}
```

#4 Is your Swift API visible to Obj-C?

- Which Swift code can Objective-C consume? How do we pick and choose the parts of our code to reveal to obj-C APIs?
- any class that inherits from NSObject will be visible in Obj-C with the exception of:
  - private properties and methods
  - language features that are specific to Swift (structs, enums, tuples).

! Structs, enums with non-integer values

and tuples are common examples of Swift features that are not available in Obj-C.

If you want to explicitly expose certain parts of a class to Obj-C and not others you can use the @objc and @nonobjc modifiers.

#5 Importing files into a mixed language project

- If you have a project in which files written in both Objective-C and Swift are embedded, you'll need to include some import statements in order to use both Swift and Obj-C classes. You'll be interacting with two types of header files that facilitate interoperability:

- **BRIDGING HEADER** - makes obj-C code visible to Swift.
- **GENERATED HEADER** - makes Swift code visible to objective-C.

To use Swift code in ObJ-C:

```
#import " ProJect_Name_ Header_File.h"
```

To use ObJ-C in Swift:

```
#import "House.h"
```

in bridging header file to make ObJ-C Code visible to Swift-