iOS Dev Accelerator
Week6 Day2

- Review
- NSCoding, NSCopying
- NSPredicate and Searching Collection classes
- Key Value Coding, Key Value Observing
- Class Clusters

Review

Private properties/methods in ObjC

- Declare private properties inside an anonymous class category
- A what?
- An anonymous class category, with no name, only matching parenthesis.
- It's like a normal @interface declaration, but no superclass is declared.
- Please… just show me an example!

Private properties/methods in ObjC

ViewController.h

```
@interface ViewController : UIViewController

@property (nonatomic, strong) APILoader *loader;

@end
```

loader is public

ViewController.m

```
@interface ViewController ()

@property (nonatomic, strong) APILoader *loader;

@end
```

loader is private

enum in Swift and ObjC

- In Swift, the enum type is powerful. In ObjC, it is not.
- ObjC enums as basically wrappers around Integers.
- Let’s you reference a “magic number” by name
- Swift enums extend way beyond simple Integers.
- In short, only similarity is the enum keyword.

instancetype vs. id

- instancetype offers extra type checking to ensure
- instancetype can only be used as the result type to a method declaration
- id is useful for opting out of type safety, but sometimes loosing type safety isn't awesome.

```
[[NSArray array] mediaPlaybackAllowsAirPlay];
```
Fails

```
[[[NSArray alloc] init] mediaPlaybackAllowsAirPlay];
```
Fails, but compiles

Conforming to Protocols

- In Swift, Superclass and Protocols are comma separated, defined together
- In ObjC, protocol defined after Superclass, between <> brackets, and comma separated

Conforming to Protocols

```
@interface ViewController : UIViewController <UITableViewDataSource, UITableViewDelegate>

   // TODO: method prototypes and properties go here

@end
```

Objective-C

```
class ViewController: UIViewController, UITableViewDataSource, UITableViewDataSource {

   // TODO: full class implementation goes here

}
```

Swift

Casting and isKindOfClass

- Use `isKindOfClass:` to check the class of an object
- Declare another variable, and cast the first object to an instance of the second.

```
if ([responseObj isKindOfClass:[NSHTTPURLResponse class]]) {
    NSHTTPURLResponse *httpResponse = (NSHTTPURLResponse *)responseObj;

    if (httpResponse.statusCode == 200) {
        NSLog(@"success is mine!");
    }
}
```

@property and generated accessors

- When declaring instance variables using @property syntax, the compiler generates accessors for you.
- Accessors are methods used to access (get, set) an instance variable
- Accessors promote abstraction, by exposing variables through methods rather than direct pointers.
- The compiler also generates a hidden instance variable; _property

@property and generated accessors

```
@interface PASPass : NSObject {
    NSString *_title;
    NSUInteger _uniqueId;
}
```

Declaring two instance variables

Older style, but still compiles

```
- (void)setTitle:(NSString *)title {
    _title = [title copy];
}

- (NSString *)title {
    return _title;
}
```

Using @property, you don’t have to do this anymore

Still likely to see examples in the wild.

@property and generated accessors

- When using @property, you are likely to see the _ prefixed instance variables show up in Xcode autocompletion.
- It’s fine to pass them in as arguments to a method (function)
- You should use the self.property style when setting property.

```
@property (nonatomic, strong) WKWebView *webView;

- (void)viewDidLoad {
    [super viewDidLoad];
    self.webView = [[WKWebView alloc] init];
    [self.view addSubview:_webView];
}
```

__block modifier

- Variables in the calling scope of the block are captured, but they cannot be modified by default.
- Use the double underscore __block modified to change a variable within a block.

```
NSMutableArray *questions = [NSMutableArray array];
__block BOOL foundTopQuestion = NO;

[objects enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {
    Question *question = [Question questionFromDictionary:obj];
    [questions addObject:question];

    if ([question isTopQuestion]) {
        foundTopQuestion = YES;
    }
}];
```

Completion Handler

```
@implementation APILoader

- (void)loadData: (void (^)(NSArray *, NSError *)) completion {
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        // Do some expensive work on a background thread
        // If your using NSURLSession, you can let that API handle threading.

        // load some expensive data, takes a long time.
        NSArray *garbage = @[@"Garbage", @"Data", @"Goes", @"Here"];

        // TODO: implement real error checking
        NSError *error = nil;

        dispatch_async(dispatch_get_main_queue(), ^{

            // Call completion on the main thread
            // Note completion is called as a C function.
            // i.e. functionName(parameter1, parameter2);
            completion(garbage, error);
        });
    });
}

@end
```

NSCoding

- Protocol defines how objects will encode and decode themselves
- Translation from one representation to another
- Allows custom objects to be serialized to disk
- Provides an alternative persistence solution vs Core Data

NSCoding

```
- (id)initWithCoder:(NSCoder *)decoder
```

Your implementation should initialize self, using the data in the decoder

```
- (void)encodeWithCoder:(NSCoder *)encoder
```

Your implementation should encode itself, by saving data into the encoder.

NSCoding

```
- (id)initWithCoder:(NSCoder *)coder {
    if (self = [self init]) {
        self.URL = [coder decodeObjectForKey:@"URL"];
        self.title = [coder decodeObjectForKey:@"title"];
        self.enabled = [coder decodeBoolForKey:@"enabled"];
        self.count = [coder decodeIntegerForKey:@"count"];
        self.position = [coder decodeCGPointForKey:@"position”];
    }

    return self;
}
```

NSCoding

```
- (void)encodeWithCoder:(NSCoder *)coder {
    if (self.URL != nil) [coder encodeObject:self.URL forKey:@"URL"];
    if (self.title != nil) [coder encodeObject:self.title forKey:@"title"];

    [coder encodeBool:self.enabled forKey:@"enabled"];
    [coder encodeInteger:self.count forKey:@"count"];
    [coder encodeCGPoint:self.position forKey:@"position"];
}
```

Archiving Objects

- `NSCoding` compliant objects can be easily persisted to disk
- Use `NSKeyedArchiver` for saving objects
- Use `NSKeyedUnarchiver` for retrieving objects
- Collection classes are `NSCoding` compliant
- Storing `NSCoding` compliant objects in Collection classes and save to disk

Archiving Objects

- Use **`NSKeyedArchiver`** for saving objects
- Pass in any **`NSCoding`** compliant object as the root object
- If the root object is a collection, all objects within must be **`NSCoding`** compliant
- The **`NSData`** returned can be written directly to disk

```
NSArray *options = [self selectedOptions];
NSData *data = [NSKeyedArchiver archivedDataWithRootObject:options];
[data writeToFile:path atomically:NO];
```

Unarchiving Objects

- Use **`NSKeyedUnarchiver`** for retrieving objects
- Pass in **`NSData`** that was previously archived
- The top level object returned will match the type of original encoding

```
NSString *path = [self pathForSavedData];

if ([[NSFileManager defaultManager] fileExistsAtPath:path]) {
    NSData *data = [NSData dataWithContentsOfFile:path];
    NSArray *options = [NSKeyedUnarchiver unarchiveObjectWithData:data];
    return options;
}
```

NSSecureCoding

- Object handles encoding and decoding in a manner that protects against substitution attacks.

```
+ (BOOL)supportsSecureCoding
```

- Your implementation should encode self, by saving data into the encoder.

NSSecureCoding

- Potentially unsafe, where object is decoded before class type is checked.

```
if let object = decoder.decodeObjectForKey("myKey") as MyClass {
    // TODO: handle success
} else {
    // TODO: handle failure
}
```

- Safer using NSSecureCoding

```
let obj = decoder.decodeObjectOfClass(MyClass.self, forKey: "myKey")
```

NSCopying

- Protocol for defining functional copies of an object
- Independent objects produced
- Protocol defines one method: `copyWithZone:`
- It is common practice to provide a copy method, for convenience
- What is an `NSZone`? It is a unit of memory.

NSCopying

```
- (id)copyWithZone:(NSZone *)zone {
    id copy = [[[self class] alloc] init];
    if (copy) {
        // Copy NSObject subclasses
        [copy setVendorID:[self.vendorID copyWithZone:zone]];
        [copy setAvailableCars:[self.availableCars copyWithZone:zone]];

        // Set primitives
        [copy setCount:self.count];
    }

    return copy;
}
```

NSPredicate

- Query language describing how data should be fetched and filtered
- Describes logic conditions to use when searching collections
- Used in many Foundation classes; Core Data too!
- Familiar with SQL? Think of the WHERE clause

NSPredicate

- Variable substitution is supported too, using $ sign for variable names
- Cache predicates and perform substitutions at runtime, for better performance.

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"id = $user"];
NSDictionary *substitutions = @{@"user": currentUser}

NSPredicate *filter = [predicate
predicateWithSubstitutionVariables:substitutions];
```

NSPredicate

- Use predicateWithFormat for building simple predicates
- Use `%K` and `%@` for substituting keys and values

```
[NSPredicate predicateWithFormat:@"name = 'Clarus'"];
[NSPredicate predicateWithFormat:@"species = %@", @"Dogcow"];
[NSPredicate predicateWithFormat:@"%K like %@", attributeName, attributeValue];
```

- Predicates will traverse key paths in a query

```
[NSPredicate predicateWithFormat:@"department.name like %@", department];
[NSPredicate predicateWithFormat:@"ANY employees.salary > %f", salary];
```

Searching Collections

- Collection classes can be searched and filtered using **`NSPredicate`***
- **`NSFetchRequest`** exposes a predicate property, for filtering entities

```
- (NSArray *)filteredArrayUsingPredicate:(NSPredicate *)predicate;
- (NSSet *)filteredSetUsingPredicate:(NSPredicate *)predicate;
```

*Most collection classes; NSDictionary and NSIndexSet don’t directly accept an NSPredicate

NSKeyValueCoding

- Protocol defines mechanisms for indirectly accessing object properties
- Object properties are accessed using the property name rather than directly using the accessor method
- Object properties can be accessed in a consistent manner
- Dynamic and flexible piece of Foundation architecture

NSKeyValueCoding

- Two basic methods for accessing and setting values

```
- (void)setValue:(id)value forKey:(NSString *)key;

- (id)valueForKey:(NSString *)key;
```

- Variations for setting nil values and supporting undefined keys

Why is this useful?

- Promotes loose coupling between two objects
- Helps to hide implementation details
- Useful for generating model objects from JSON
  - Maintain a mapping dictionary between external, internal names

Key-value Observing

- Object properties can be observed indirectly by their key
- One object can observe the property of another object, taking action any time the property value changes
- In MVC, Controller can observe Model and View properties
- Closely related within Key-value Coding

NSKeyValueObserving

- Protocol that defines a way for objects to be notified about property changes on other objects.
- Observe any object properties including simple attributes, to-one relationships, and to-many relationships
- NSObject implements this protocol; you get for free!

NSKeyValueObserving

- When properties are set, using generated setter, these methods are called before and after the change occurs.
- If you mutate properties outside of the generated setter, you'll need to call these methods manually to be KVO compliant

```
- (void)willChangeValueForKey:(NSString *)key;
- (void)didChangeValueForKey:(NSString *)key;
```

Key-value Observing

- Add an observer for an object property, using the key or key path

```
[webView addObserver:self
          forKeyPath:”estimatedProgress"
             options:NSKeyValueObservingOptionNew
             context:0];
```

- Respond to changes by implementing one method on your observer

```
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context
```

KVO Gotchas

- Must call super implementation of observeValueForKeyPath
- Recommended to use a valid context pointer instead of NULL
- Your implementation of observeValueForKeyPath will be called for subclasses. Use context pointer to differentiate
- Remember to remove observers.
  - Use `deinit()` in Swift, `dealloc` in ObjC

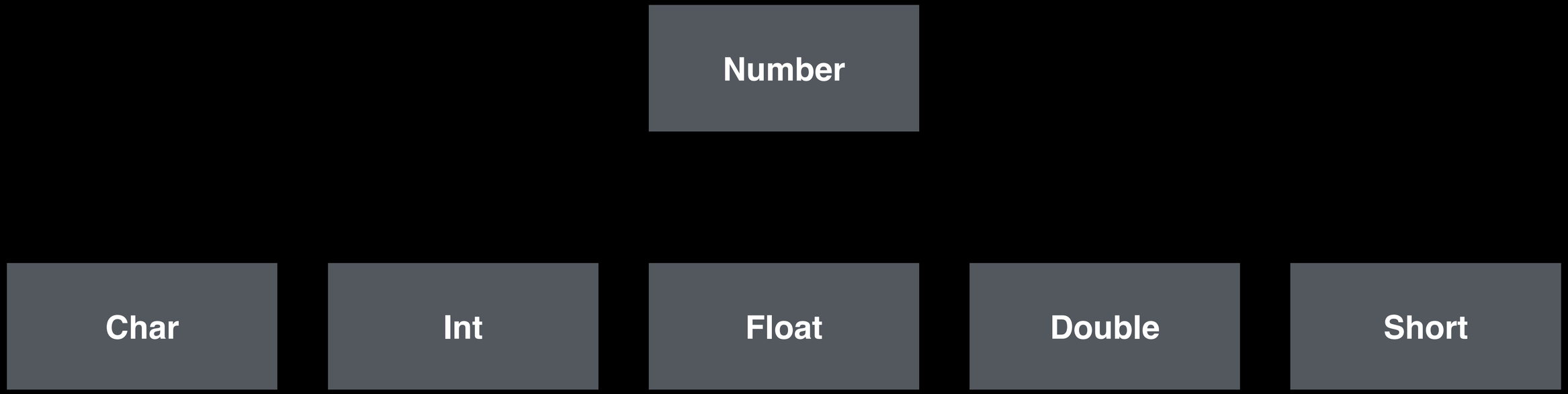
http://stackoverflow.com/a/14162363

Class Clusters

- Design pattern used by Foundation
- Groups a number of private classes underneath an abstract superclass
- Simplifies publicly visible architecture without sacrificing functional richness
- Based on the Abstract Factory design pattern

Class Clusters

- Number is an abstract superclass
- Internal implementation is hidden behind the scenes



Class Clusters

- Collection classes like `NSArray` and `NSDictionary` are clusters
- Might see `__NSArrayM_` or `__NSDictionaryI_` in the call stack
- These are internal implementation details bleeding through
- Subclass class clusters can be risky

  https://mikeash.com/pyblog/friday-qa-2010-03-12-subclassing-class-clusters.html