iOS Dev Accelerator Week 8

- Subclassing

Subclassing

- Create a subclass from the File > New menu
- Superclass is specified after a colon, following the class name
- Can modify a superclass manually

```
@interface HNCommentsModel : NSObject

@end
```

```
@interface SubclassName : SuperclassName

@end
```

Why use subclasses?

- It’s object-oriented programming; it’s what we do
- Modeling hierarchies of objects is a useful mental abstraction model
- Corresponding real-world taxonomy

When to create a subclass

- When creating a view or cell with a custom layout.
  - `UITableViewCell`, `UICollectionViewCell`
- When creating a custom model class
  - Subclass `NSObject` and leverage existing methods
- When abstracting away concepts to another level

When NOT to create a subclass

- If subclasses only share an interface, consider using protocols
- If a class will be modified and configured often, consider using a delegate model to dynamically configure.
- If adding simple functionality, consider using a Category instead
- If an existing API is available to achieve similar results.

What not to subclass

- `UIButton`
- Class clusters `NSString`
- Container view controllers
  - `UITabBarController`, `UINavigationController`

Limited Exposure and #import

- Should limit your classes footprint only #importing what you need
- You often don't need to #import dependancies in the `Header.h` file
- If the classes uses the dependancies internally, then #import dependencies inside `Implementaton.m`
- Forward declare classes using the `@class` directive
- Conform to protocols in an anonymous class category when possible

Forward Declaration

- Alternative to #importing in the **`Header.h`**
- Tells the compiler to ignore the missing class, for now.
- It will be there eventually, at runtime
- Syntax is @class followed by a class name

```
@class HNEntry;

@interface HNCommentsModel : NSObject

@property (nonatomic, copy) NSDictionary *info;
@property (nonatomic, strong) HNEntry *entry;

@end
```

Alternatives to Subclassing

- Protocols
- Delegates
- Categories
- Existing APIs

Declaring Protocols

- Define a set of behaviors an object is expect to have
- Specify methods an object should implement in certain situations
- Same concepts as from Swift
- Use @protocol keyword, followed by a name, and some number of methods
- Common for protocols to conform to <NSObject>, for compiler warnings.

```
@protocol HNEntryLoaderDelegate <NSObject>

- (void)controller:(UIViewController *)controller shouldLoadURL:(NSURL *)url;

@end
```

Categories

- Add new functionality to existing class, by implementing category methods
- Category methods will be available to all instances of the class
- Can modify classes that you don’t own and didn’t create
- Same concept as Swift extensions

Categories

- Categories required a Header and Impementation file
- Typically these are named in relation to class being modified
- Common to use a + plus between original class and category name
  - NSString+HTMLAdditions.h and NSString+HTMLAdditions.m
- Category methods are often namespaced to avoid collisions, because someone might implement a metod with that name in the future.

Using existing APIs

- A common pattern in Cocoa is to allow classes to be modified by providing other class instances
- This is because Cocoa objects are commonly composed of other objects.
- Some API or high level UI objects allow configuration to small elements

Using existing APIs

- A few examples from UIKit
- UINavigationController exposes a navigationBar property, which can be set (even though it's marked readonly)
- UIPopoverController exposes a popoverBackgroundViewClass property, to set as any subclass of UIPopoverView