iOS Development

Adam May

Sam Kirchmeier

Goals

Goals

1. Learn iOS

Goals

1. Learn iOS
2. Be ambitious

Goals

1. Learn iOS
2. Be ambitious
3. Build an app

Build an app

Connects to a web service
Displays hierarchical information
Has custom views
Has custom animations
Uses object persistence
Uses concurrency

Smarticle

NY Times
http://developer.nytimes.com/docs

Goodreads
http://www.goodreads.com/api

Instagram
http://instagram.com/developer

Schedule

Mondays

6-9 p.m.

No class on Memorial Day, May 26th

Final class is June 23rd

iOS TC Hack

Here every 2nd and 4th Wednesday

7-9 p.m.

5/14, 5/28, 6/11, 6/25

http://tchacknight.com

Topics

Week 1: Objective-C
Week 2: Cocoa Touch
Week 3: Networking
Week 4: Animations & Layout
Week 5: Performance
Week 6: Deployment

Class Structure

Lecture

Labs

Homework

Class Kickoff

Introductions

Today

Toolset Overview

Crash Course in Objective-C

Build an Old-School Adventure Game

Toolset

Adventure.xcodeproj — AdventureViewController.xib

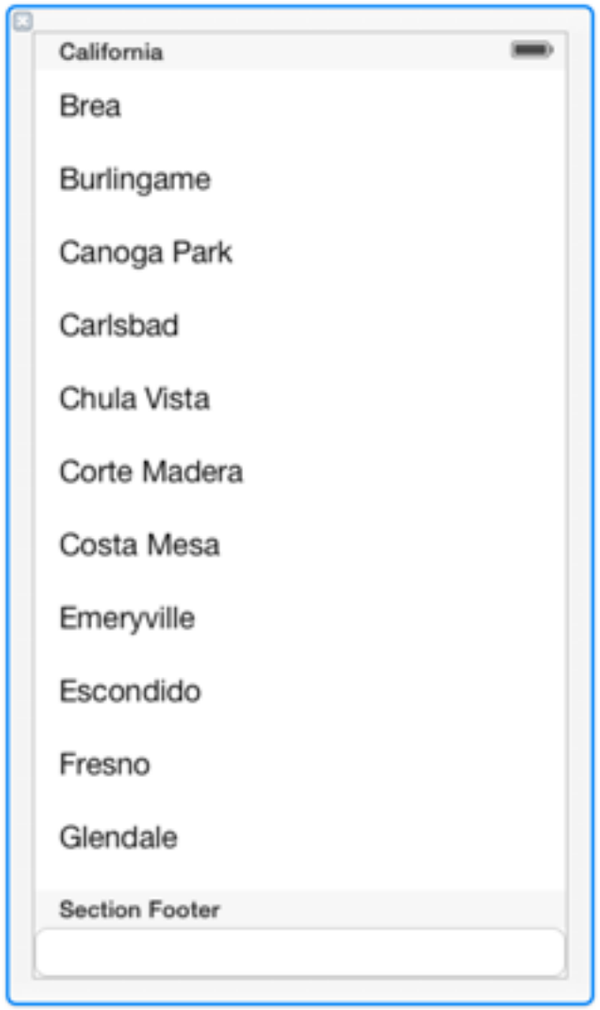
Adventure › iPhone › iOS 6.1 | Adventure: Ready | Today at 2:03 AM | No Issues

Adventure › Adventure › AdventureViewController.xib › View

Adventure
2 targets, iOS SDK 7.0
- Adventure
  - AdventureCell.h
  - AdventureCell.m
  - AdventureCell.xib
  - AdventureResponder.h
  - AdventureViewController.h M
  - AdventureViewController.m
  - AdventureViewController.xib
  - AppDelegate.h
  - AppDelegate.m
  - FactoryAdventure.h
  - FactoryAdventure.m
  - Room.h
  - Room.m
  - Supporting Files
- Adventure Tests
- External
- Frameworks
- Products

Placeholders
- File's Owner
- First Responder

View



View

Show: Frame Rectangle

X: 0 Y: 0
Width: 320 Height: 568
Origin

Content Hugging Priority
Horizontal: 250
Vertical: 250

Content Compression Resistance Priority
Horizontal: 750
Vertical: 750

Intrinsic Size: Default (System Defined)

Constraints
- Top Space to: Table View
- Trailing Space to: Table View
- Leading Space to: Table View
- Trailing Space to: Round Styl...
- Leading Space to: Round Styl...

Label — **Label** – A variably sized amount of static text.

Button — **Button** – Intercepts touch events and sends an action message to a target object when it's tapped.

1 2 — **Segmented Control** – Displays multiple segments, each of which functions as a discrete button.


Carrier
3:40 PM
Photos
Contacts
Settings
Game Center
Tuesday
12
Calendar
280
Maps
Newsstand
Passbook
Safari

Instruments

Stop | Target: PreciouStatusSta... | Inspection Range | 00:00:20 Run 1 of 1 | View | Library | Filter: Instrument Detail

Instruments

Allocations

VM Tracker

Allocations | Statistics | Allocation Summary

Generation Analysis

Mark Generation

Allocation Lifespan

- All Objects Created
- Created & Still Living
- Created & Destroyed

Allocation Type

- All Heap & Anonymous VM
- All Heap Allocations
- All VM Regions

Call Tree

- Separate by Category
- Separate by Thread
- Invert Call Tree
- Hide Missing Symbols
- Hide System Libraries
- Show Obj-C Only
- Flatten Recursion

Call Tree Constraints

Specific Data Mining

Graph	Category	Live Bytes	# Living	# Transient	Overall Bytes	# Overall	Bytes Allocated (Net / Overall)
✓	All Heap & Anonymous VM	9.36 MB	18,862	59,545	23.19 MB	78,407	
	All Heap Allocations	1.73 MB	18,822	59,519	10.41 MB	78,341	
	All Anonymous VM	7.63 MB	40	26	12.78 MB	66	
	VM: ImageIO_PNG_Data	4.39 MB	7	0	4.39 MB	7	
	VM: CG raster data	2.55 MB	7	1	2.57 MB	8	
	VM: SQLite page cache	256.00 KB	2	0	256.00 KB	2	
	Malloc 2.00 KB	172.00 KB	86	153	478.00 KB	239	
	VM: CoreAnimation	132.00 KB	5	0	132.00 KB	5	
	VM: Allocation 128.00 KB	128.00 KB	1	0	128.00 KB	1	
	Malloc 62.50 KB	125.00 KB	2	0	125.00 KB	2	
	Malloc 1.00 KB	109.00 KB	109	268	377.00 KB	377	
	VM: CoreUI image data	108.00 KB	13	2	116.00 KB	15	
	CFString (immutable)	101.72 KB	3,156	7,874	370.09 KB	11,030	
	Malloc 4.00 KB	96.00 KB	24	105	516.00 KB	129	
	CFString (store)	94.52 KB	667	1,108	231.22 KB	1,775	
	CFData (store)	66.61 KB	82	305	89.80 KB	387	
	CFBufferConstantSize<327...	65.00 KB	2	0	65.00 KB	2	
	Malloc 32 Bytes	59.44 KB	1,902	13,972	496.06 KB	15,874	
	sfnt_name_t	57.42 KB	614	141	76.55 KB	755	
	CFBasicHash (value-store)	53.06 KB	644	1,446	156.64 KB	2,090	
	Malloc 1.50 KB	52.50 KB	35	83	177.00 KB	118	
	CFBasicHash (key-store)	50.03 KB	579	1,195	147.16 KB	1,774	
	Malloc 16 Bytes	44.48 KB	2,847	10,330	205.89 KB	13,177	
	CFData	38.50 KB	186	467	68.94 KB	653	
	Malloc 8.00 KB	32.00 KB	4	4	64.00 KB	8	
	VM: Allocation 32.00 KB	32.00 KB	1	0	32.00 KB	1	
	CFDictionary (mutable)	28.45 KB	607	587	55.97 KB	1,194	
	Malloc 64 Bytes	24.62 KB	394	2,553	184.19 KB	2,947	
	SecCertificate	23.00 KB	12	0	23.00 KB	12	

Objective-C

History

History

1981-1983: Brad Cox and Tom Love working at Stepstone, develop a new language based on C, heavily influenced by Smalltalk
1985: Steve leaves Apple to found NeXT
1988: NeXT licenses Objective-C from Stepstone and uses it to build NeXTSTEP, a forerunner of OS X and iOS
1996: Apple acquires NeXT
2001: Mac OS X released
2007: iOS released

Why Objective-C?

Fast
Object-oriented
Based on C
Dynamic
Well-maintained

Objective-C

C

objects

Objective-C

C

[[[[]]][]]][]]]]]][[]]

Objective-C is a strict superset of C.

Code you write in C just works.

C

C, the good parts

Primitives
Operators
Low-level data types
Functions
Pointers

Primitives

Integer

```
char c = ‘a’;
short mentors = 2;
int students = 9;
unsigned long count = 11L;
```

Floating Point

```
float pi = 3.14f;
double percentage = 0.75;
```

Casting

```
double ratio = 1.5;
int i = (int)ratio;  // truncates
```

Precision

```
sizeof(int); // Always 4, for us
```

Precision

```
sizeof(NSInteger) // 4 on 32-bit,
                  // 8 on 64-bit
```

Booleans

```
BOOL shouldWeUseBools = YES;
BOOL areBoolsAnythingButIntegers = NO;
```

Operators

Arithmetic

a + b

a - b

a * b

a / b

a % b

a++

a--

Boolean

a == b

a != b

a >= b

a < b

!a

a || b

a && b

Bitwise

~a

a & b

a | b

a ^ b

a << b

a >> b

Assignment

a = b

a+=b

a-=b

a*=b

a/=b

a%=b

Data types

Enums

```
typedef enum : NSInteger {
    AnimationFade,
    AnimationFlip,
} Animation;

Animation myFade = AnimationFade;
```

Structs

```
struct Point {
   float x;
   float y;
};
typedef struct Point Point;

Point p;
p.x = 20.f;
p.y = 10.f;
```

Arrays

```
int ratings[5] = {1, 2, 3, 4, 5};
```

```
int bestRating = ratings[4];
```

C-Strings

```
char[] cString = “Hello, world!”;
```

Functions

Functions

```
int sum(int a, int b)
{
   return a + b;
}

int nine = sum(4, 5);
```

Pointers

Pointers

```
int justAnInt;
int *pointerToAnInt;
```

Objects

```
NSString *name;
```

```
NSString *name = @“Adam”;
```

```
NSString *name = @“Adam”;
NSUInteger length = [name length];
```

Messages

```
[name length];
```

When name receives the message length,

```
[name length];
```

its length method is called.

Messages can be sent with parameters

```
[name substringFromIndex:2];
```

Messages can be sent with parameters

```
[name substringFromIndex:2];
```

and return values

```
length = [name length];
```

A message with multiple parameters

```
[name stringByPaddingToLength:9
withString:@“.” startingAtIndex:0];
```

A message with multiple parameters

```
[name stringByPaddingToLength:9
                   withString:@“.”
              startingAtIndex:0];
```

and a cleaner look.

We would refer to this method as

```
[name stringByPaddingToLength:9
                   withString:@“.”
              startingAtIndex:0];
```

stringByPaddingToLength:withString:startingAtIndex:

The colons are part of the name

```
[name stringByPaddingToLength:9
                   withString:@“.”
              startingAtIndex:0];
```

indicating where the parameters go.

```
NSString *name = @“Adam”

// This will return Adam.....
[name stringByPaddingToLength:9
                   withString:@“.”
              startingAtIndex:0];
```

Favors clarity over brevity

We might imagine an NSString method that also takes 3 parameters:

```
[name :9 :@“.” :0];
```

The name of this made-up, but syntactically correct , method would be :::.

Messages can also be nested, like so

```
[[name substringFromIndex:2] length];
```

Creating Objects

It turns out this

```
NSString *name = @“Adam”;
```

Is actually something like this

```
NSString *name =
    [[NSString alloc] initWithUTF8String:”Adam”];
```

Or for an empty string, simply

```
NSString *empty = [[NSString alloc] init];
```

alloc returns an instance of the receiving class

```
NSString *empty = [[NSString alloc] init];
```

init prepares the instance for use

Always always always

```
NSString *empty = [[NSString alloc] init];
```

call alloc and init together.

Or if you’re already tired of typing brackets

```
NSString *empty = [NSString string];
```

call a convenience factory method

But know that alloc/init

```
NSString *empty = [[NSString alloc] init];
```

is always behind the scenes.

Variables

When declaring a local primitive variable, it's uninitialized.

```
int value;          // Yikes!
```

Best off initializing it, like so:

```
int value = 3;      // OK
```

Pointers are always automatically initialized to nil.

```
NSArray *anArray; // OK
```

nil

nil receives any message and returns a default value based on the method's return type

```
NSString *nilString = nil;
NSUInteger count = [nilString length];
```

No exception thrown here, count will be set to 0.

`NO` for BOOL return types

`nil` for object return types

`0` for numeric return types

`nil` evaluates to false

```objc
if (aString != nil)
{
   NSString *upper = [aString uppercaseString];
}
```

```objc
if (aString)
{
   NSString *upper = [aString uppercaseString];
}
```

```
NSString *name = @“”;

if ([name length] != 0)
{
    // name is not empty
}
else
{
    // name is empty
}
```

```objc
NSString *name = nil;

if ([name length] != 0)
{
    // name is not empty
}
else
{
    // name is empty
}
```

Foundation

Helpful classes that you will use in every iOS app

NSString

Just a bunch of characters

NSNumber

An Objective-C wrapper for scalar C types

```
NSNumber *satScore =
   [[NSNumber alloc] initWithInt:2400];

NSNumber *gpa =
   [[NSNumber alloc] initWithDouble:4.0];

NSNumber *smartyPants =
   [[NSNumber alloc] initWithBool:YES];
```

```
NSNumber *satScore =
   [NSNumber numberWithInt:2400];

NSNumber *gpa =
   [NSNumber numberWithDouble:4.0];

NSNumber *smartyPants =
   [NSNumber numberWithBool:YES];
```

```
NSNumber *satScore = @2400;
NSNumber *gpa = @4.0;
NSNumber *smartyPants = @YES;
```

```
int scalarSatScore = [satScore intValue];
double scalarGpa = [gpa doubleValue];
BOOL scalarSmartyPants = [smartyPants boolValue];
```

```
NSNumber *zero = @0;

if (zero)
{
   // Will this code get executed?
}
else
{
   // Or will this code?
}
```

```
NSNumber *zero = @0;

if ([zero boolValue])
{
   // Will this code get executed?
}
else
{
   // Or will this code?
}
```

Collections

NSArray

NSDictionary

NSArray

An ordered collection of objects

```
NSArray *subjects =
    [[NSArray alloc] initWithObjects:
        @"English", @"Science", @"Math", nil];
```

This is literally a much better way to create arrays

```
NSArray *subjects = @[@"English",
@"Science",
@"Math"];
```

Mix and match types

```
NSNumber *aNumber = @3;
NSString *aString = @"Three";
```

```
NSArray *anArray = @[aNumber, aString];
```

No problem! (At least until you access them).

Retrieve objects using objectAtIndex:

```
NSArray *subjects =
   @[@"English", @"Science", @“Math”];

NSString *science = [subjects objectAtIndex:1];
```

or with subscripting

```
NSString *math = subjects[2];
```

Enumeration

```
for (int i = 0; i < anArray.length; i++)
{
   NSString *string = [anArray objectAtIndex:i];
}

for (NSString *string in anArray)
{
   // Fast enumeration, really fast
}
```

NSDictionary

Maps keys to values

```
NSDictionary *favoriteColors =
   [NSDictionary dictionaryWithObjectsAndKeys:
@"Blue", @"Sam",
@"Green", @"Adam",
nil];
```

So literal

```
NSDictionary *favoriteColors = @{
    @"Sam" : @"Blue"
    @"Adam" : @"Green"
};
```

Retrieve objects by key

```
NSDictionary *favoriteColors = @{
    @"Sam" : @"Blue"
   @"Adam" : @"Green"
};
```

```
NSString *samsFave = [favoriteColors
                              objectForKey:@“Sam”];
```

or with subscripting

```
NSString *adamsFave = favoriteColors[@“Adam”];
```

Mutability

NSMutableString
NSMutableArray
NSMutableDictionary
NSMutableSet

NSMutableArray subclasses NSArray

```
NSMutableArray *breweries = [NSMutableArray array];

[breweries addObject:@"Fulton"];
[breweries addObject:@"Surly"];
[breweries addObject:@"Indeed"];

[breweries removeObject:@"Indeed"];
```

providing addObject: and removeObject:

Convert between mutable and immutable

```
NSArray *immutableBreweries = [breweries copy];
NSMutableArray *mutableBreweries =
[immutableBreweries mutableCopy];
```

by making a copy.

Lab 1.1

Lab 1.1

Write your first program with Objective-C

Debug your first program with log statements

Familiarize yourself with strings, arrays, dictionaries

Classes

.h

Header
Public Interface

.m

Implementation
Private Interface

Room.h

```
@interface Room : NSObject

@end
```

Room.h

```
@interface Room : NSObject

@property NSString *name;

@end
```

Room.h

```
@interface Room : NSObject

@property (nonatomic, strong) NSString *name;

@end
```

```
@property (nonatomic, strong) NSString *name;
```

```
@property  nonatomic
           atomic
```

@property

strong
weak
assign
copy

```objc
NSString *_name;

- (void)setName:(NSString *)name
{
    // Depends on property attributes
}

- (NSString *)name
{
    return _name;
}
```

```
[myRoom setName:@"Kitchen"];
```

```
[myRoom setName:@"Kitchen"];
```

```
myRoom.name = @"Kitchen";
```

Methods

Room.h

```
@interface Room : NSObject

@property (nonatomic, strong) NSString *name;

@end
```

Room.h

```
@interface Room : NSObject

@property (nonatomic, strong) NSString *name;

- (id)initWithName:(NSString *)name;

@end
```

```
- (NSUInteger)length;

- (NSComparisonResult)compare:(NSString *)aString
                      options:(NSStringCompareOptions)mask
                        range:(NSRange)range;

- (NSString *)stringByAppendingString:(NSString *)aString;
```

Implementation

Room.m

```
#import "Room.h"

@implementation Room


@end
```

Room.m

```
#import "Room.h"

@implementation Room

- (id)initWithName:(NSString *)name
{
    // Code for initWithName:
}

@end
```

```
- (id)initWithName:(NSString *)name
{
    self = [super init];

    if (self)
    {
        // Initialize the object here
    }

    return self;
}
```

```
- (id)initWithName:(NSString *)name
{
    self = [super init];

    if (self)
    {
        _name = name;
    }

    return self;
}
```

alloc & init

```
Room *kitchen = [[Room alloc] initWithName:@“Kitchen”];
```


Room
alloc

Private Interface

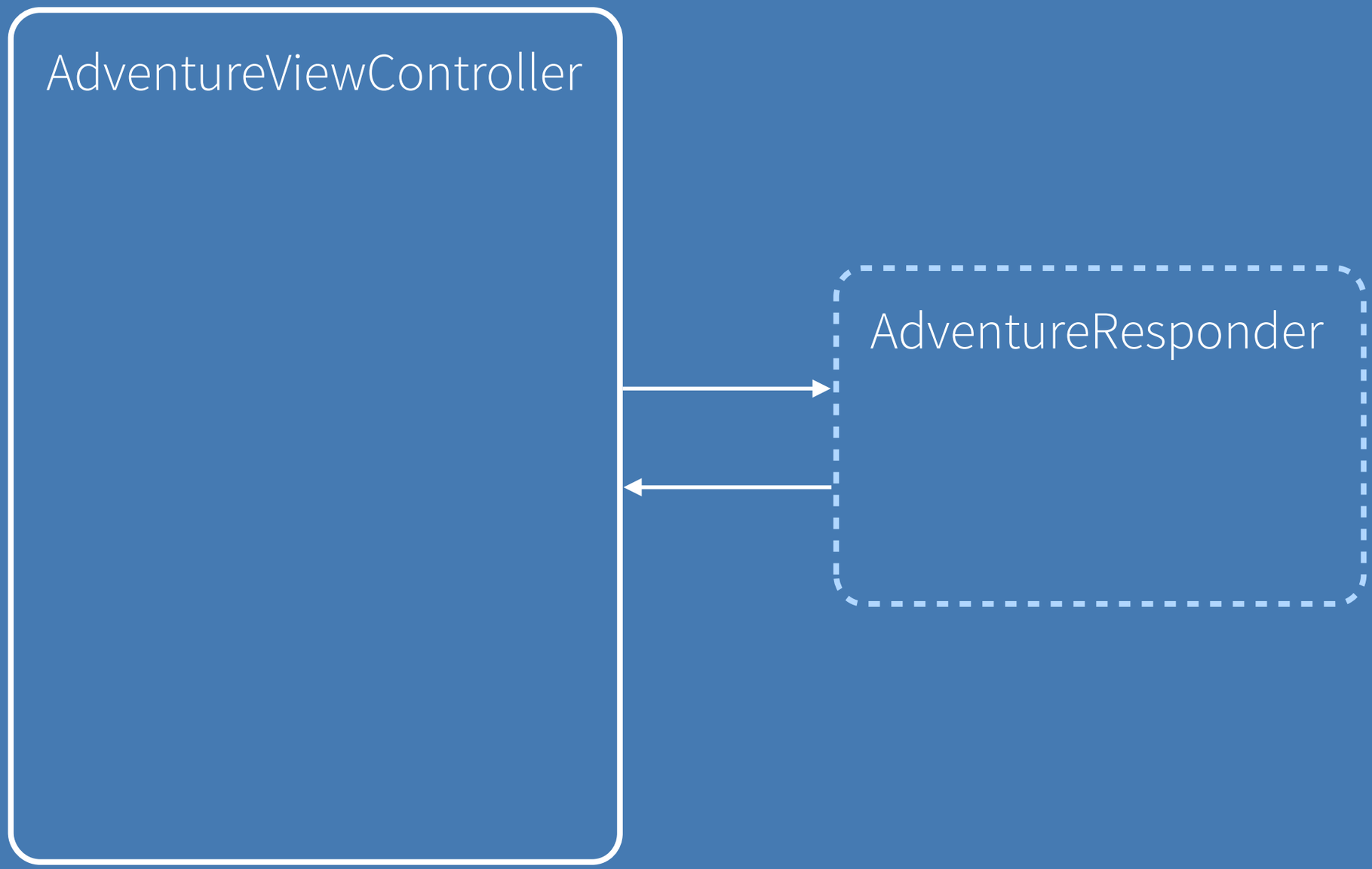
Room.m

```
#import "Room.h"

@interface Room ()

// Private interface goes here

@end

@implementation Room

// Public and private implementation goes here

@end
```

Namespaces

Protocols

AdventureViewController

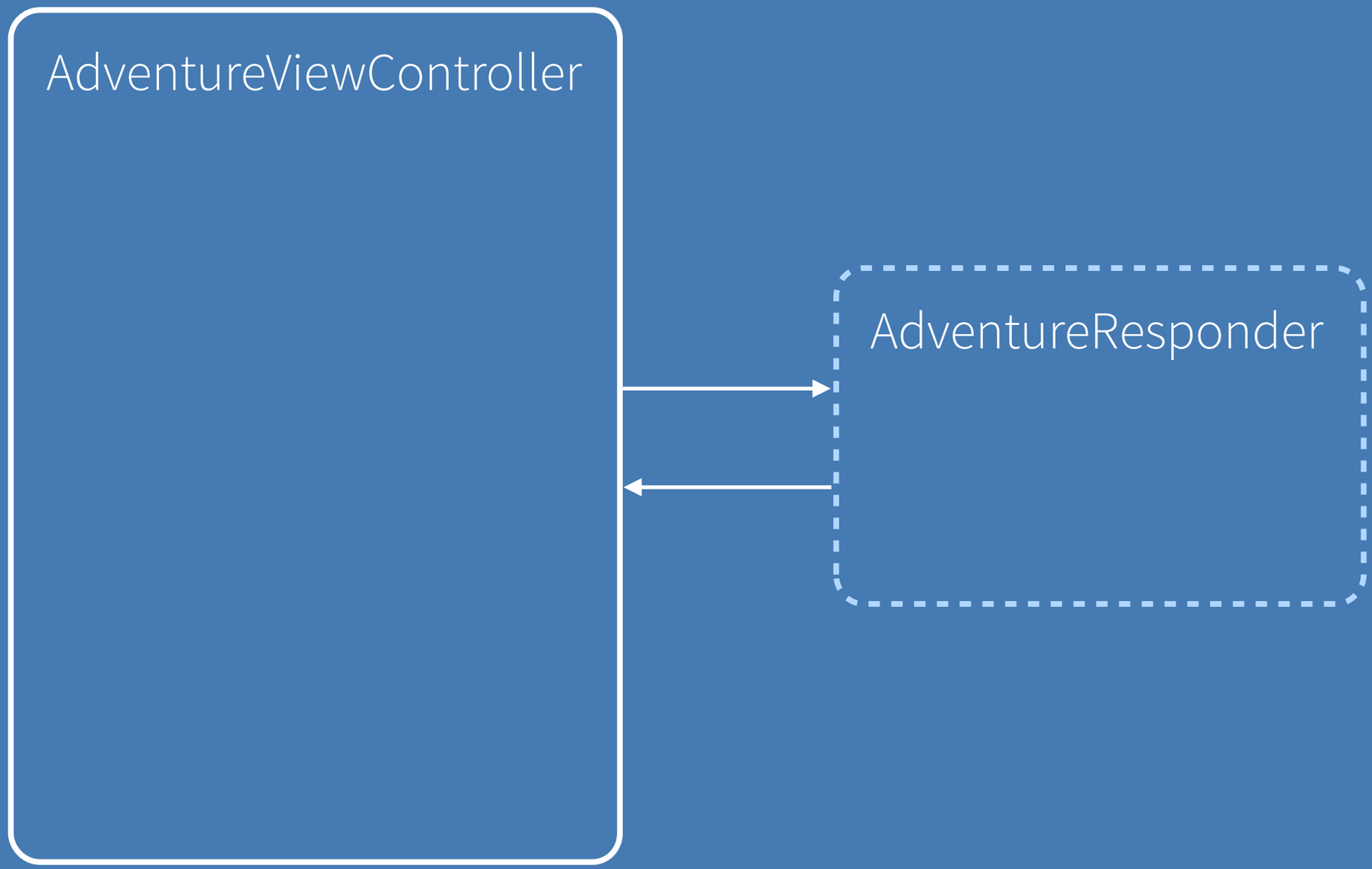

AdventureViewController
AdventureResponder

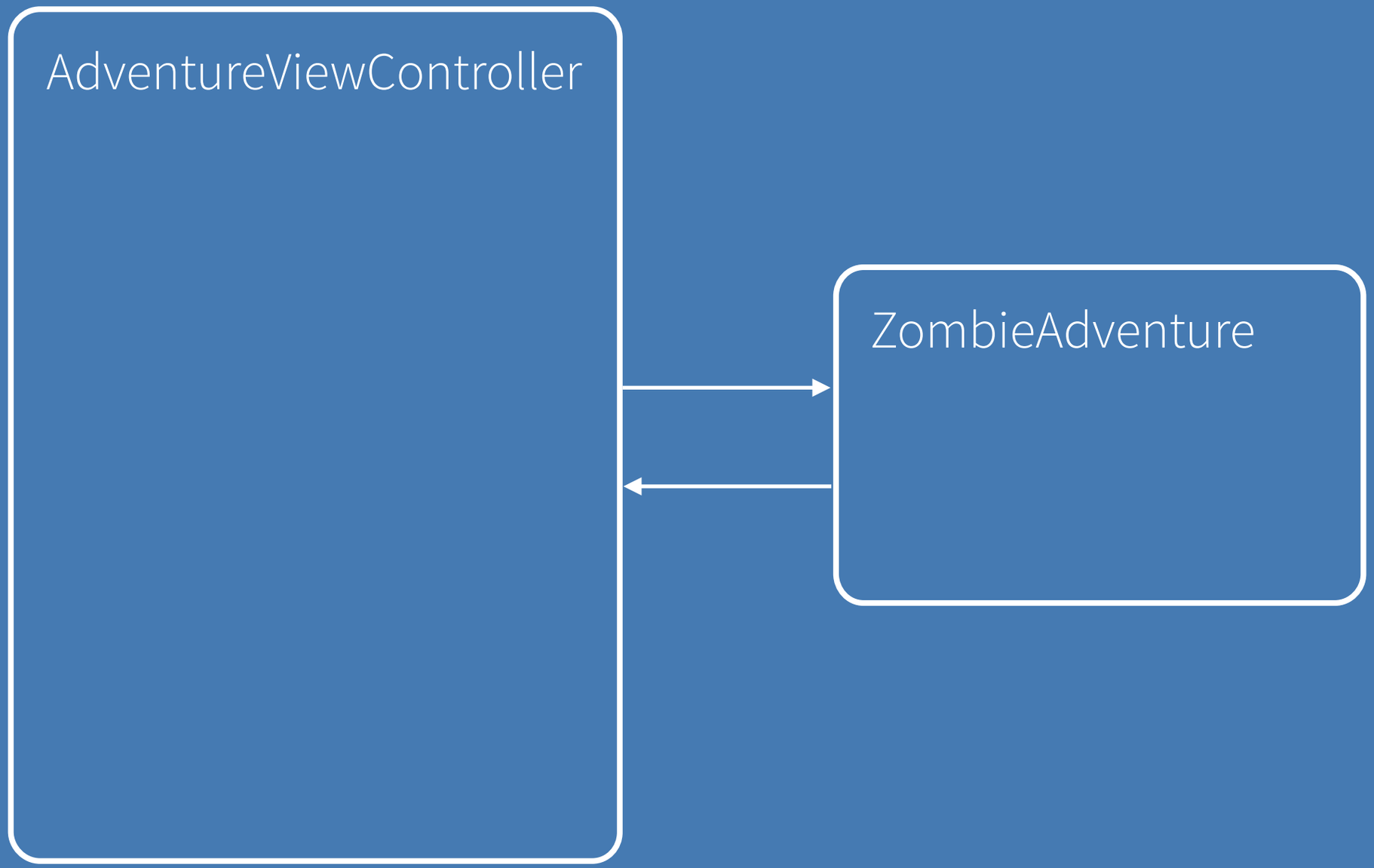
```
@protocol AdventureResponder

@end
```

```
@protocol AdventureResponder
- (NSString *)responseForInput:(NSString *)input;

@end
```

```
@protocol AdventureResponder <NSObject>
- (NSString *)responseForInput:(NSString *)input;

@end
```

```
@protocol AdventureResponder <NSObject>

@required
- (NSString *)responseForInput:(NSString *)input;

@end
```


AdventureViewController
AdventureResponder


AdventureViewController
ZombieAdventure

.h

Header
Public Interface

.m

Implementation
Private Interface

.h

```
@interface ZombieAdventure : NSObject <AdventureResponder>

@end
```

.m

```
@implementation ZombieAdventure

- (NSString *)responseForInput:(NSString *)input
{
}

@end
```

.m

```
@implementation ZombieAdventure

- (NSString *)responseForInput:(NSString *)input
{
    return @"You were eaten by zombies. Game over.";
}

@end
```

Lab 1.2

Lab 1.2

Build your own AdventureResponder

Create an Item class

Expand the adventure by adding rooms

Lab 1.2 Discussion

Next Week

Xcode projects
View Controllers
UINavigationController
UITableViewController

Resource to read ahead:

Apple’s Getting Started Docs