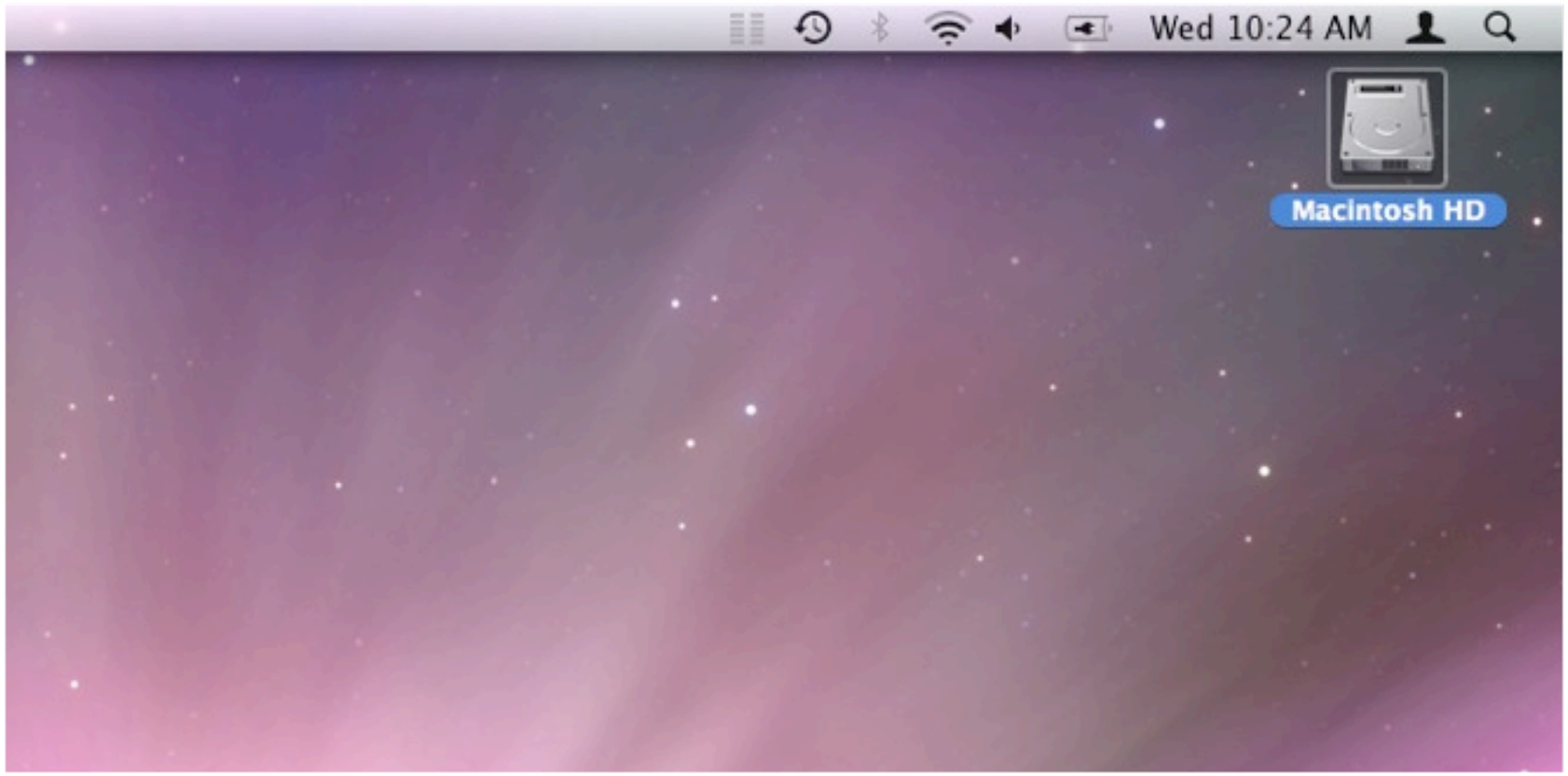


Intro to Cocoa: Part Two

A crash course in programming Macs, iPhones, and iPod Touches

Stephen Voida, August 2009

In Our Last Episode…

- A bit of history and perspective on OS X
- Intro to the Apple development tools (Xcode, Interface Builder)
- Intro to the Objective-C programming language
- Intro to the Cocoa frameworks
- Putting it all together: A simple UI app from start to finish

This Week

- Tidy up loose ends from last week (questions, corrections)
- Moving to the iPod: Using the developer tools and working with provisioning profiles
- Intro to the UIKit framework, including widgets, views, and event handling
- Other goodies: multi-touch, animation, OpenGL
- Four example programs demonstrating different aspects of the APIs and interaction techniques

Last Week's Example (Fixed!)

- The header file…

```
#import <Foundation/Foundation.h>

@interface LuckyNumbers : NSObject {
    NSMutableString *personName;
    int firstNumber;
    int secondNumber;
}
- (void)prepareRandomNumbers;
- (void)setPersonName:(NSString *)name;
- (NSString *)personName;
- (int)firstNumber;
- (int)secondNumber;
@end
```

- And the implementation… (part 1 of 3)

```
#import “LuckyNumbers.h”

@implementation LuckyNumbers

// Override the default initializer, providing default values
- (id)init {
    return [self initWithName:@“Jane Doe”];
}

// Provide a “designated initializer” to do all the work
- (id)initWithName:(NSString *)name {
    self = [super init];
    if (self) {
        personName = [name retain];              // need to retain it!
        firstNumber = random() % 100 + 1;
        secondNumber = random() % 100 + 1;
    }
    return self;
}

- (void)dealloc {
    [personName release];          // Deallocate what we retained
    [super dealloc];               // Before dealloc’ing ourselves
}
```

- And the implementation… (part 2 of 3)

```
- (void)prepareRandomNumbers {
    firstNumber = random() % 100 + 1;
    secondNumber = random() % 100 + 1;
}

- (void)setPersonName:(NSString *)name {
    [name retain];                  // “Retain-then-release” prevents
    [personName release];           // inadvertent deallocation
    personName = name;              // if someone sends a ref for the
}                                   // active personName to the setter

- (NSString *)personName {
    return personName;
}

- (int)firstNumber {
    return firstNumber;
}

- (int)secondNumber {
    return secondNumber;
}
```

- And the implementation… (part 3 of 3)

```
- (NSString *)description {
    NSString *result;
    result = [[NSString alloc] initWithFormat:@"%@'s lucky numbers
            are %d and %d", personName, firstNumber, secondNumber];
    [result autorelease];           // decrements retain count but
    return result;                  // doesn't deallocate right away
}

@end
```

Properties (A Recent Addition!)

- "getters" and "setters" comprise quite a lot of object-oriented code, and they're methods where most of the memory management headaches tend to arise
- Instead of defining the getters and setters, use a special syntax
  - In the header: `@property (attributes) type varName;`

    Example: `@property (retain) NSDictionary* myHashtable;`
  - In the implementation file: `@synthesize varName;`

    Example: `@synthesize myHashtable;`

Property attributes

- `readwrite` generates both getter and setter (default)
- `readonly` generates only the getter
- `assign` stores the variable by simple assignment (default)
- `retain` stores the variable and increases its retain count (the right thing to do for most object types)
- `copy` stores a copy of the new value, ensuring that the new item is retained (good for `NSStrings`)
- `nonatomic` doesn't bother with using a lock in the setter method

Updating the Class Example

- The header file…

```
#import <Foundation/Foundation.h>

@interface LuckyNumbers : NSObject {
    NSMutableString *personName;
    int firstNumber;
    int secondNumber;
}
- (void)prepareRandomNumbers;
- (void)setPersonName:(NSString *)name;
- (NSString *)personName;
- (int)firstNumber;
- (int)secondNumber;
@end
```

Updating the Class Example

- The header file…

```
#import <Foundation/Foundation.h>

@interface LuckyNumbers : NSObject {
    NSMutableString *personName;
    int firstNumber;
    int secondNumber;
}
- (void)prepareRandomNumbers;
- (void)setPersonName:(NSString *)name;
- (NSString *)personName;
- (int)firstNumber;
- (int)secondNumber;
@end
```

Updating the Class Example

- The header file…

```
#import <Foundation/Foundation.h>

@interface LuckyNumbers : NSObject {
    NSMutableString *personName;
    int firstNumber;
    int secondNumber;
}
- (void)prepareRandomNumbers;

@property (copy) NSMutableString* personName;
@property (readonly, assign) int firstNumber;
@property (readonly, assign) int secondNumber;

@end
```

- And the implementation… (part 2 of 3)

```
- (void)prepareRandomNumbers {
    firstNumber = random() % 100 + 1;
    secondNumber = random() % 100 + 1;
}

- (void)setPersonName:(NSString *)name {
    [name retain];
    [personName release];
    personName = name;
}

- (NSString *)personName {
    return personName;
}

- (int)firstNumber {
    return firstNumber;
}

- (int)secondNumber {
    return secondNumber;
}
```

- And the implementation… (part 2 of 3)

```
- (void)prepareRandomNumbers {
    firstNumber = random() % 100 + 1;
    secondNumber = random() % 100 + 1;
}

- (void)setPersonName:(NSString *)name {
    [name retain];
    [personName release];
    personName = name;
}

- (NSString *)personName {
    return personName;
}

- (int)firstNumber {
    return firstNumber;
}

- (int)secondNumber {
    return secondNumber;
}
```

- And the implementation… (part 2 of 2)

```
- (void)prepareRandomNumbers {
    firstNumber = random() % 100 + 1;
    secondNumber = random() % 100 + 1;
}

@synthesize personName;
@synthesize firstNumber;
@synthesize secondNumber;

- (NSString *)description {
    NSString *result;
    result = [[NSString alloc] initWithFormat:@"%@'s lucky numbers
            are %d and %d", personName, firstNumber, secondNumber];
    [result autorelease];
    return result;
}

@end
```

- And the implementation… (part 1 of 2)

```
#import “LuckyNumbers.h”

@implementation LuckyNumbers

// Override the default initializer, providing default values
- (id)init {
    return [self initWithName:@“Jane Doe”];
}

// Provide a “designated initializer” to do all the work
- (id)initWithName:(NSString *)name {
    self = [super init];
    if (self) {
        personName = [name retain];
        firstNumber = random() % 100 + 1;
        secondNumber = random() % 100 + 1;
    }
    return self;
}

- (void)dealloc {
    [personName release];
    [super dealloc];
}
```

- And the implementation… (part 1 of 2)

```
#import “LuckyNumbers.h”

@implementation LuckyNumbers

// Override the default initializer, providing default values
- (id)init {
    return [self initWithName:@“Jane Doe”];
}

// Provide a “designated initializer” to do all the work
- (id)initWithName:(NSString *)name {
    self = [super init];
    if (self) {
        personName = [name retain];
        firstNumber = random() % 100 + 1;
        secondNumber = random() % 100 + 1;
    }
    return self;
}

- (void)dealloc {
    [personName release];
    [super dealloc];
}
```

- And the implementation… (part 1 of 2)

```
#import “LuckyNumbers.h”

@implementation LuckyNumbers

// Override the default initializer, providing default values
- (id)init {
    return [self initWithName:@“Jane Doe”];
}

// Provide a “designated initializer” to do all the work
- (id)initWithName:(NSString *)name {
    self = [super init];
    if (self) {
        self.personName = name;
        self.firstNumber = random() % 100 + 1;
        self.secondNumber = random() % 100 + 1;
    }
    return self;
}

- (void)dealloc {
    self.personName = nil;
    [super dealloc];
}
```

Properties: Pros and Cons

- PROS
  - Less code to write/manage/maintain
  - Reduces memory-related programming overhead
  - Introduces new syntax (.) that more closely mirrors C++, Java
- CONS
  - Creates potential for overlooking memory management details
  - Removes explicit method declarations from header file
  - Adds another new syntax to the language (.)

Protocols

- Objective-C objects exhibit single-inheritance; protocols are like Java interfaces and help to lend a flavor of multiple inheritance
- A protocol is a list of methods that a class promises to implement
- Example:

```
@protocol WidgetDelegate

- (void) widgetDidChangePowerSettings:(float)newPowerLevel;

@required
- (void) widgetDidGetTurnedOn:(id)widget;

@end
```

Using Protocols

- Specifying that a class will conform to a protocol:

```
@interface WidgetManager : NSObject <WidgetDelegate>
```

- Testing to see if an object implements a protocol:

```
if ([anObject conformsToProtocol:@protocol(WidgetDelegate)])
{ ...
}
```

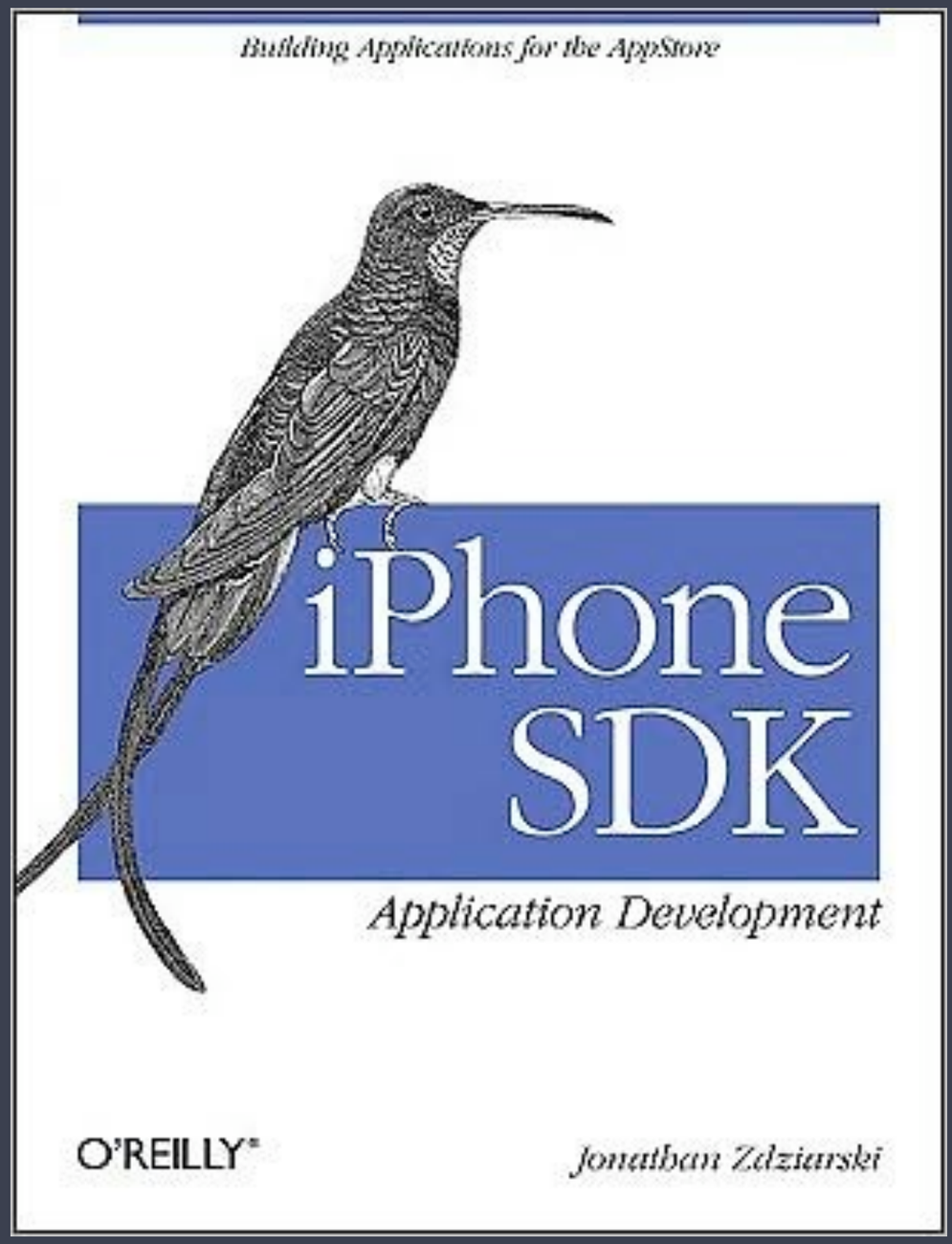
- Referring to classes as “instances” of a protocol:

```
id<WidgetDelegate> anObject;
[anObject widgetDidGetTurnedOn:self];
```

Questions at this point?

(Disclaimer)

Borrowing heavily here from Apple's example code and Zdziarski, J. *iPhone SDK Application Development*. O'Reilly, Sebastopol, CA, 2009.



Example 1:
“Hello World” for the iPhone

- Creating an iPhone UI using Interface Builder
- Differences in the UI framework and app classes
- Basic MVC: Targets
- Compiling for the iPhone Simulator
- Running and debugging iPhone code
- Bonus: Handling different device orientations

Receiving Orientation Changes

- Override the following default method in your `UIViewController`-derived object:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
                    (UIInterfaceOrientation)interfaceOrientation
{
    return (interfaceOrientation == UIDeviceOrientationPortrait);
}
```

- Possible enumeration values include:
  `UIDeviceOrientationPortrait`, `...PortraitUpsideDown`, `...LandscapeLeft`, `...LandscapeRight`, `...FaceUp`, `...FaceDown`

Receiving Orientation Changes

- Optionally, implement the following handler to receive notifications when the device has been re-oriented:

```
- (void)didRotateFromInterfaceOrientation:
                        (UIInterfaceOrientation)interfaceOrientation
{
    // Code here to deal with orientation changes
}
```

Example 2:
A Table-based iPhone App

- Extreme MVC: Delegates, and data sources
- Compiling for the iPhone Device
- Certificates and Provisioning Profiles (ugh)
- Running code on the device and debugging remotely

Example 3: Zdziarski's CALayer Demo

- Lifted directly from Chapter 5 of the *iPhone SDK* book
- Using CALayers to render complex images and animations
- Requesting data from the Internet
- Using a timer

Example 4:
Touches

- Displaying static images embedded in the application bundle
- Drawing to a UIView with Quartz
- Handling multi-touch input

Receiving Multi-Touch Events

- Override any or all of the following event-handling methods in a `UIView`-derived object:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)e;
```

- `touches` is an unordered list of `UITouch` objects, each corresponding to a touch on the screen; `event` is higher-level info

What does UITouch provide?

- `timestamp` timestamp relative to the previous event in seconds (type is NSTimeInterval = double)
- `phase` an enumerated value describing whether this touch just began, moved, ended, was stationary, or a cancel invoked the event
- `tapCount` the number of taps represented by this touch
- `locationInView` a `CGPoint` structure representing the floating point position of the touch

OpenGL ES

- Version 1.1 (which is supported by all iPhones/iPods) is a subset of desktop OpenGL 1.5 using a fixed-function pipeline
- Includes support for vertex arrays, [multi-]textures, renderbuffers, auto mipmap generation, clip planes, point sprites (for particles)
- Does not support `glBegin`–`glEnd` semantics, display lists, quad or polygon primitives, texgen, accumulation buffers, copy pixels, selection, or push and pop state attributes
- Utilizes standard OpenGL function calls: `glRotate`, `glOrtho`, `glLineWidth`, `glMaterial`, `glTexImage2D`, `glBindTexture`

Incorporating OpenGL into the app

- Special kind of CALayer that can receive OpenGL commands: `CAEAGLLayer`

Summing Up…

Workshop Objectives

- High-level understanding of how the OS X platform is organized and how Cocoa applications are implemented
- Introduction to the OS X and iPhone SDKs and development tools
- Quick-and-dirty overview of Objective-C
- Brief tour of the Cocoa and UIKit frameworks, focusing on rapid application development and UI software prototyping
- Illustration of how all the pieces come together with code samples

Good Places to Find Help

- Hillegass and Zdziarski books
- Apple Developer Documentation
  - Inside of Xcode
  - http://developer.apple.com
- Apple Code Samples — great place from which to borrow ideas
- CocoaDev (http://www.cocoadev.com)
- Me! (svoida@gmail.com, even after I'm gone)