

# CS193P - Lecture 3

## iPhone Application Development

Custom Objective-C Classes  
Properties

# Announcements

- Office Hours

- Troy - Monday 4 - 6 PM, Meyer Library 2nd floor
- Kayvon/Joe - Tuesday 2:30 - 4:30 PM

- Class Calendar

- Available on Google Calendars

<http://tinyurl.com/CS193P-Calendar>

- or search for "CS193P" in public Google calendars
- Shows lectures, office hours, due dates and guest speakers

# Announcements

- **Enrollment**
  - Emails sent to the 50 students that got enrolled
  - If you didn't receive an email, please drop the class in Axess
  - Auditors still welcome!
- Keep an eye on the wiki, we'll start adding common Q&A there
- Still working out details of university team...
  - For the time being, stick with the simulator

# Announcements

- Both Assignment 2A and 2B due next Weds 10/8
  - 2A: Continuation of Foundation tool
    - Add custom class
    - Basic memory management
  - 2B: Beginning of our first iPhone application
    - We'll cover this on Thursday

# Today's Topics

- Questions from Assignment 1A or 1B?
- Defining and Using Custom Classes
- Objective-C Properties
- Dot Syntax Addendum

# Custom Classes

# Design Phase

- Create a Person class
- Determine the superclass
- Determine properties of the class
  - Name, age, whether they can vote
- Decide on actions the class can perform
  - Cast a ballot

# Defining a class

A public header and a private implementation



Header File



Implementation File

# Class interface declared in header file

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

@interface Person : NSObject
{
    // instance variables
    NSString *name;
    int age;
}

// method declarations
- (NSString *)name;
- (void)setName:(NSString *)value;

- (int)age;
- (void)setAge:(int)age;

- (BOOL)canLegallyVote;
- (void)castBallot;
@end
```

# Defining a class

A public header and a private implementation



Header File



Implementation File

# Implementing custom class

- Implement setter/getter methods
- Implement action methods

# Class Implementation

```
#import "Person.h"

@implementation Person

- (int)age {
    return age;
}

- (void)setAge:(int)value {
    age = value;
}

... and other methods

@end
```

# Calling your own methods

```
#import "Person.h"

@implementation Person

- (BOOL)canLegallyVote {
    return ([self age] >= 18);
}

- (void)castBallot {
    if ([self canLegallyVote]) {
        // do voting stuff
    } else {
        NSLog(@"I'm not allowed to vote!");
    }
}

@end
```

# Using Classes

# Using Classes

- Creating objects
- Basic memory management
- Destroying objects

# Object Creation

- Two step process
  - allocate memory
  - initialize object state
  
- + `alloc`
  - class method that allocates memory for the object
  
- `init`
  - instance method to complete initialization

# Creating objects

```
Person *person = nil;
```

```
person = [[Person alloc] init];
```

# Init method

```
#import "Person.h"

@implementation Person

- (id)init {
    // allow superclass to initialize its state first
    if (self = [super init]) {
        age = 0;
        name = nil;

        // do other initialization...
    }

    return self;
}

@end
```

# Multiple init methods

- Classes may define multiple init methods

- `(id)init;`
- `(id)initWithName:(NSString *)name;`
- `(id)initWithName:(NSString *)name age:(int)age;`

- Less specific ones typically call more specific with default values

- `(id)init {  
 return [self initWithName:@"No name"];  
}`
- `(id)initWithName:(NSString *)name {  
 return [self initWithName:name age:0];  
}`

# Creating objects

```
Person *person = nil;

person = [[Person alloc] init];

[person setName:@"Paul Marcos"];
[person setAge:24];
[person setWishfulThinking:YES];

[person castBallot];

// What do we do with person when we're done?
```

# Memory management

	Allocation	Destruction
C	malloc	free
Objective-C	alloc	release

# Reference counting

- Every object has a “retain count”
  - defined in NSObject
  - as long as retain count is  $> 0$ , object is alive and valid
- `+alloc` and `-copy` create objects with retain count == 1
- `-retain` increments retain count
- `-release` decrements retain count
- When retain count reaches 0, object is destroyed
  - `-dealloc` method invoked
  - dealloc is a one-way street, once you're in it there's no turning back

# Creating objects

```
Person *person = nil;

person = [[Person alloc] init];

[person setName:@"Paul Marcos"];
[person setAge:24];
[person setWishfulThinking:YES];

[person castBallot];

// When we're done with person, release it
[person release];    // person will be destroyed here
```

# Object deallocation

```
#import "Person.h"

@implementation Person

- (void)dealloc {
    // Do any cleanup that's necessary

    // when we're done, call super to clean us up
    [super dealloc];
}

@end
```

# Object deallocation

- When retain count == 0, object is deallocated
  - dealloc method called automatically
- You **never** call dealloc explicitly in your code
  - You only deal with alloc, copy, retain, release to manage an object's lifetime

# Object ownership

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

@interface Person : NSObject
{
    // instance variables
    NSString *name; // Person class "owns" the name
    int age;
}

// method declarations
- (NSString *)name;
- (void)setName:(NSString *)value;

- (int)age;
- (void)setAge:(int)age;

- (BOOL)canLegallyVote;
- (void)castBallot;
@end
```

# Object ownership

```
#import "Person.h"

@implementation Person

- (NSString *)name {
    return name;
}

- (void)setName:(NSString *)newName {
    if (name != newName) {
        [name release];
        name = [newName copy];
        // name's retain count has been bumped by 1
    }
}

@end
```

# Object deallocation

```
#import "Person.h"

@implementation Person

- (void)dealloc {
    // clean up any objects we own
    [name release];

    // when we're done, call super to clean us up
    [super dealloc];
}

@end
```

# Objective-C Properties

# Properties

- Alternative mechanism for providing access to object attributes
- Typically replace boilerplate getter/setter methods
- Also allow for defining:
  - read-only versus read-write access
  - memory management policy for object properties

# Properties

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

```
@interface Person : NSObject  
{  
    // instance variables  
    NSString *name;  
    int age;  
}
```

```
// method declarations  
+(NSString *)name;  
@property(nonatomic) NSString *name;  
@property(nonatomic) NSString *age;  
@property(nonatomic) BOOL  
- (BOOL)canLegallyVote;
```

```
- (void)castBallot;  
@end
```

# Properties

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

@interface Person : NSObject
{
    // instance variables
    NSString *name;
    int age;
}

// property declarations
@property int age;
@property (copy) NSString *name;
@property (readonly) BOOL canLegallyVote;

- (void)castBallot;
@end
```

# Properties

```
@implementation Person
```

```
@synthesize age; ;  
@synthesize name; ;  
}  
- (void)setAge:(int)value {  
    age = value;  
}  
- (NSString *)name {  
    return name;  
}  
- (void)setName:(NSString *)value {  
    if (value != name) {  
        [value release];  
        name = [value copy];  
    }  
}
```

```
- (BOOL)canLegallyVote {  
    return (age > 17);  
}  
@end
```

# Property attributes

- Read-only versus read-write

```
@property int age; // read-write by default  
@property (readonly) BOOL canLegallyVote;
```

- Memory management policies (only for object properties)

```
@property (assign) NSString *name; // pointer assignment  
@property (retain) NSString *name; // retain called  
@property (copy) NSString *name; // copy called
```

- Different setter/getter methods

```
@property (getter=getAge, setter=setAge) int age;
```

# Properties

- Property name can be different than instance variable

```
@interface Person : NSObject {  
    int numberOfYearsOld;  
}
```

```
@property int age;
```

```
@end
```

- 

```
@implementation Person
```

```
@synthesize age = numberOfYearsOld;
```

```
@end
```

# Properties

- Mix and match synthesized and implemented properties

```
@implementation Person

@synthesize age;
@synthesize name;

- (void)setAge:(int)value {
    age = value;

    // now do something with the new age value...
}

@end
```

- Setter method explicitly provided
- Getter method still synthesized

# Properties - In Practice

- Newer APIs use @property
- Older APIs use getter/setter methods
- Properties used heavily throughout UIKit APIs
  - Not so much with Foundation APIs
- You can use either approach
  - Properties are more “magic”, but can be non-obvious
  - Getter/setters are clear, but requires manual memory management

# Dot Syntax

# Dot Syntax and self

- When used in custom methods, be careful with dot syntax for properties defined in your class
- References to properties and ivars behave very differently

```
@interface Person : NSObject
{
    NSString *name;
}
@property (copy) NSString *name;
@end

@implementation Person
- (void)doSomething {
    name = @"Fred";           // accesses ivar directly!
    self.name = @"Fred";     // calls accessor method
}
}
```

# Common Pitfall with Dot Syntax

What will happen when this code executes?

```
@implementation Person
- (void)setAge:(int)newAge {
    self.age = newAge;
}
@end
```

This is equivalent to:

```
@implementation Person
- (void)setAge:(int)newAge {
    [self setAge:newAge];
}
@end
```

# Further Reading

- Objective-C 2.0 Programming Language
  - “Defining a Class”
  - “Declared Properties”

Questions?