iPhone Programming
Objective-C, Part 2
Spring 2019

- Custom Objective-C Classes
- Properties
Custom Classes
Design Phase - Tracking Voters

- 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
Or, in recent version of Xcode, we can simply declare a PROPERTY to create the ivar, the getter and the setter!

```objc
@property (nonatomic) age;
```

Automatically creates
1) a getter named `age`
2) a setter called `setAge` and
3) an ivar called `_age`
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:@"Barack Obama"];  
[person setAge:57];

[person castBallot];

// What do we do with person when we’re done?
# Memory management

<table>
<thead>
<tr>
<th></th>
<th>Allocation</th>
<th>Destruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>malloc</td>
<td>free</td>
</tr>
<tr>
<td>Objective-C</td>
<td>alloc</td>
<td>release</td>
</tr>
</tbody>
</table>
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:@"Barack Obama"];  
[person setAge:57];

[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
#import "Person.h"

@interface Person

- (NSString *)name {
    return name;
}

- (void)setName:(NSString *)newName {
    if (name != newName) {
        [name release];
        name = [newName retain];
        // name’s retain count has been bumped up by 1
    }
    if (name != newName) {
        [name release];
        name = [newName copy];
        // name has retain count of 1, we own it
    }
@end
#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;
- (void)setName:(NSString *)value;
+ (int)age;
- (void)setAge:(int)age;
- (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
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

- Mix and match synthesized and implemented properties
  
  ```swift
  @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
Using 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?

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

This is equivalent to:

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