CS193E
Lecture #3

Categories and Protocols
Cocoa Memory Management

Winter 2008, Dempsey/Marcos

Today’s Topics

- Questions from Assignment 1A or 1B?
- Categories
- Protocols
- Cocoa Memory Management
- Object life cycle
- Demo of Favorite Things I

Objective-C Files

```objc
#import <Cocoa/Cocoa.h>

@interface Person:NSObject
{
    NSString *name;
    int age;
    float weight;
}

- (void)printName;
- (float)ageToWeightRatio;

@end
```

```objc
#import "Person.h"

@implementation Person

- (void)printName {
 NSLog(name);
}

- (float)ageToWeightRatio {
 return age/weight;
}

@end
```

Objective-C Categories

Objective-C Categories

- Allows additional methods to be added to an existing class
- Common alternative to subclassing for small functional additions
- Can be used to break a class up into multiple source files
- Can’t add instance variables, just methods

Objective-C Categories

- Category name in parenthesis

```
/* Category on NSString */
@interface NSString (MailAddressUtilities)
- (NSString *)emailAddress;
- (NSString *)fullName;
@end
```

- At runtime, the new methods are part of the class

```
NSString *string = @“Derek Clegg <dclegg@stanford.edu>”;

[addr emailAddress];   // returns “dclegg@stanford.edu”
[addr fullName];       // returns “Derek Clegg”
```

Objective-C Categories

```
@implementation NSString (MailAddressUtilities)

- (NSString *)emailAddress {
    // Extract and return the email address
}

- (NSString *)fullName {
    // Extract and return the full name
}

@end
```

Protocols

Protocols

- Objective-C supports single inheritance
- Sometimes desired functionality cuts across class boundaries
- Protocols define only an interface, no implementation
- A class conforms to a protocol by implementing all of its methods
- Almost identical to Java interfaces

Protocols define interface across classes

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

@protocol Drawing

// Only method declarations - no implementation
- (void)draw;
- (NSSize)maxSize;
- (NSSize)minSize;

@end
```

Classes declare conformance

```
@interface Shape : NSObject <Drawing>
{
   NSRect shapeRect;
}

// implement all the methods of the Drawing protocol
- (void)draw;
- (NSSize)maxSize;
- (NSSize)minSize;

@end
```

Working with protocols

- Use angle brackets to declare conformance of a class

```
@interface Shape : NSObject <Drawing>
```

- List multiple protocols separated by commas

```
@interface NSColor : NSObject <NSCoding, NSCopying>
```

- Using protocols in variable and method declarations

```
id <NSCopying> anObject;
MyObject <NSCopying> *obj;

- (void)saveCopyOfObject:(id <NSCopying>)obj;
```

Informal Protocols

- Sometimes a more informal arrangement is desired
- A collection of methods in a category

```
@interface NSObject (NSDraggingSource)
- (NSDragOperation)draggingSourceOperationMaskForLocal:
      (BOOL)flag;
- (BOOL)ignoreModifierKeysWhileDragging;
/* more... */
@end
```

- No requirement that all (or any!) be implemented
- No compile-time type checking

Memory Management

Memory Management

- In C you’ve got malloc/free
  - Ownership is explicit
- In Java you’ve got garbage collection
  - Objects simply go away when nobody is using them any more
- In Cocoa you’ve got reference counting
  - It’s somewhere in the middle

malloc / free

Just Plain C

```
void *someMem = malloc(128);


free(someMem);
```

+alloc / -init

Allocate a new object with class method +alloc

Different -init methods for same class

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



[person release];
```

-copy

Copy an existing object

```
NSString *string; // assume this exists
NSString *string2 = [string copy];



[string2 release];
```

Accessing objects

- Allocating a new object: you need to release it

```
MyObject *obj = [[MyObj alloc] init];
```

- Copying an existing object: you need to release it

```
NSString *string
NSString *aCopy = [string copy];
```

- Everything else: not your responsibility by default

```
NSSet *set = [NSSet setWithObjects:obj1, obj2, nil];
```

Everything else

How to hold on to an object?

```
NSString *title = [window title];
```

-retain

Hold onto an existing object

```
NSString *title = [window title];
[title retain];




[title release];
```

Reference Counting

- All objects have a “retain count”
  If retain count > 0 object is alive
- When retain count goes to 0, object is deallocted
- alloc/copy/retain increments retain count
- release decrements retain count
- Balance every alloc/copy/retain with a release

Ref Count Example

```
Dog *myDog = [[Dog alloc] init];
               [myDog retain];
                 [myDog retain];
                   [myDog retain];
                     [myDog shake];
                     [myDog rollOver];
                     [myDog giveTreat];
                   [myDog release];
                 [myDog release];
               [myDog release];
[myDog release];
```

Ref Count Example

```
Dog *myDog = [[Dog alloc] init];
Dog *yourDog = [myDog copy];

[myDog playNiceWith:yourDog];

[yourDog release];
[myDog release];
```

Returned Objects

- What about objects returned by non-alloc and non-copy methods?
- Don’t release them, unless you retain them
- These returned objects are automatically released for you
  - Sounds like memory management magic

Returned Objects

```
- (void)playFetch {
  Dog *myDog = [self favoriteDog];

  [self throwTennisBall];
  [myDog fetchTennisBall];
  [myDog dropTennisBall];
  [myDog dropTennisBall];
  [myDog dropTennisBall];
  [self pryTennisBallFromDogsMouth];

  [myDog release];
}
```

Example method

```
+ (NSString *)stringWithString:(NSString *)str {


}
```

A problematic implementation

```
+ (NSString *)stringWithString:(NSString *)str {
    return [[NSString alloc] initWithString:str];
}
```

Caller of the method would be responsible for releasing the object!

Another problematic implementation

```
+ (NSString *)stringWithString:(NSString *)str {
  NSString *newString =
            [[NSString alloc] initWithString:str];

  [newString release];

  return newString;
}
```

We are returning an object that we’ve already released!

We'd like something like this:

```
+ (NSString *)stringWithString:(NSString *)str {
  NSString *newString =
          [[NSString alloc] initWithString:str];

  [newString
  tossInABucketOfThingsThatWillGetSentReleaseLater];

  return newString;
}
```

This would let us fulfill our obligation to release the object, but give the caller a chance to use and retain the returned object.

Autorelease

```
+ (NSString *)stringWithString:(NSString *)str {
  NSString *newString =
            [[NSString alloc] initWithString:str];

  [newString autorelease];

  return newString;
}
```

An autorelease pool is the ‘bucket’

Autorelease

```
+ (NSString *)stringWithString:(NSString *)str {
  return [[[NSString alloc] initWithString:str] autorelease];
}
```

Often used nested with other messages

Autoreleasing Objects

- “autorelease” means “release later”
- Allows you to return an object without making the caller worry about ownership
- Main event loop has an “autorelease pool” around it
- Additional pools can be used for fine grained memory management, e.g. in tight loops

Autorelease Example

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

int main (int argc, const char * argv[]) {

    NSAutoreleasePool * pool =
                [[NSAutoreleasePool alloc] init];

     NSString *string =
          [NSString stringWithFormat:
               @"The date is %@", [NSDate date]];

    [pool release];
    return 0;
}
```

Event loop pseudocode

```
NSEvent *event;

while (event = [NSApp nextEvent] ) {

     // AppKit provides a pool for every event
     NSAutoreleasePool * pool =
               [[NSAutoreleasePool alloc] init];

     [NSApp handleEvent: event];

    [pool release];
    return 0;
}
```

Collections

- Collections retain objects when inserted and release them when removed
- Dictionaries copy keys, but retain values
- All collections release values when they are deallocated

```
NSMutableArray *myArray // assume this exists


Thing myObj = [[Thing alloc] init];
[myArray addObject: myObj];
[myObj release];
```

Class implementations

Class Implementations

- -init method(s)
  Often set up ivars (`copy`/`retain`)
- -dealloc method
  Used to clean up ivars (`release`)
- “setter” accessor method
  `release` the old, `retain`/`copy` the new
- “getter” accessor methods
  Often will `retain` then `autorelease`

Writing an init method

- *Override* -[NSObject init]

```
- (id)init {
    if (self = [super init]) {
        // do your initialization here
    }
    return self;
}
```

- *Some classes have various flavors of init*

```
- (id)initWithName:(NSString *)value {
 if (self = [super init]) {
        name = [value copy];
  }
  return self;
}
```

Writing an init method

- All variables are initialized to be zero
- Some classes have various flavors of init
  - Check docs for "designated initializer"
  - Subclasses should invoke superclasses designated initializer

Object Deallocation

- `dealloc` method called when object freed
- If your class retains objects as instance variables, override `dealloc` to release them

```
- (void)dealloc {
    // do your cleanup here
    [name release];

    [super dealloc];
}
```

- Compiler will warn if you forget to call `[super dealloc]` in your override

Object Deallocation

- You never call dealloc directly!!!
- Deallocation is a one-way street, there's no turning back once dealloc is called
- Messages sent to dealloc'ed objects crash

Accessor Methods

Scalar values

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

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

Accessor Methods

Object values

```
- (NSString *)name {
 return [[name retain] autorelease];
}

- (void)setName:(NSString *)value {
  if (value != name) {
    [name release];
    name = [value copy]; // or retain
  }
}
```

Conventional Wisdom

- `alloc`/`copy` return retained objects
- All other methods don’t need special handling
- Any `alloc`/`copy`/`retain` calls must be balanced with a `release` or `autorelease` call

Questions?

Favorite Things

Favorite Things Assignment

- Two assignments over two weeks
- Favorite Things 1 due Wed, January 23rd, 11:59 PM
- Small, single window application
- Explore on a small scale the big design patterns of Cocoa

Favorite Things Demo

Questions?