Programming in Objective-C (Part 2)

MSDOS-X Lecture 2
Administrivia

- Enrollment will probably decline pretty quickly
- If not, we will open up another section
- If you’re on the wait list for 198, watch for spots to open
- Don’t panic! We won’t turn away people who really want the class.
Methods/Messages

• Declaring and invoking methods

• Many people have a hard time connecting the two because the syntax looks different

• But if you think about it in message-passing terms, they really are not that different
Methods without args

Declaring:
- (int) getLength;

Invoking:
[someObject getLength]
Methods without args

Declaring:
- (int) getLength;

Class (+) or instance (-) method

Invoking:
[someObject getLength]

If getLength were a class method, we would replace someObject with the class name.
Methods without args

Declaring:
- \((\text{int}) \text{getLength}\);

The return type.
The whole expression below
is of that type.

Invoking:
[\text{someObject getLength}]

That is, [\text{someObject getLength}] \Leftrightarrow \text{an integer}
Methods without args

Declaring:
- `(int) getLength;`

Invoking:
`[someObject getLength]`

The message (or method name)
Methods with args

Declaring:

+ (int)inchesForFeet: (int)feet
    andInches: (int)inches;

Invoking:

[TheClassName inchesForFeet: 2
    andInches: 5]
Methods with args

Declaring:

\[ + \text{(int)inchesForFeet: (int)feet andInches: (int)inches; } \]

Class (+) or instance (-) method

Invoking:

[TheClassName inchesForFeet: 2 andInches: 5]

Since inchesForFeet:andInches: is a class method, we send a message to the class, rather than an instance.
Methods with args

Declaring:
+ (int)inchesForFeet: (int)feet
   andInches: (int)inches;

Invoking:
[TheClassName
   inchesForFeet: 2
   andInches: 5]
Methods with args

Declaring:

\[ + \ (\text{int})\text{inchesForFeet}: \ (\text{int})\text{feet} \]
\[ \text{andInches}: \ (\text{int})\text{inches}; \]

The parameters with their types

Invoking:

\[ \text{[TheClassName inchesForFeet}: \ 2 \]
\[ \text{andInches}: \ 5] \]

Actual arguments
Class Methods

+ (BOOL)isSubclassOfClass:(Class)aClass;

Class methods are called on the class itself

[NSButton isSubclassOfClass:[NSControl class]]
nil

- **nil** is the “nothing object”

- If `person == nil` then:
  
  ```
  [person age] == 0
  [person name] == nil
  [person eat:taco] // fails silently
  ```

- Some methods take **nil** as a “don’t care” parameter
<table>
<thead>
<tr>
<th>ObjC</th>
<th>YES</th>
<th>NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ / C99</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>actual value</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
Selectors

SEL quackSelector = @selector(quack);
[donaldDuck performSelector:quackSelector];

is equivalent to

[donaldDuck quack];
Selectors

SEL fetchSelector = @selector(fetch:);
[fido performSelector:fetchSelector withObject:theBall];

is equivalent to

[fido fetch:theBall];
Selectors in Cocoa

Delegates / “duck typing”

```objective-c
if ([obj respondsToSelector:@selector(quack)]) {
    [obj quack];
}
```

Target / action pattern

```objective-c
[button setTarget:self];
[button.setAction:@selector(buttonPressed:)];
```
Naming Conventions

• Variables start with a lowercase letter and use capitalization to distinguish words: e.g. thisIsAGoodName, but not a_bad_name

• Class names start with a prefix indicating the project, company, or author to avoid name collisions: e.g. SRSscroll or ProjMainView, but not mutableString or TEXT_FIELD
Properties

@interface Fraction : NSObject {
    int numerator;
    int denominator;
}
@end

@end

@implementation Fraction

-(void)setNumerator:(int)n {
    numerator = n;
}

-(void)setDenominator:(int)d {
    if (d != 0)
        denominator = d;
}

-(int)numerator {
    return numerator;
}

-(int)denominator {
    return denominator;
}
@end

You write this sort of thing all the time... so shouldn’t there be an easier way to do it?
Properties

• These kinds of “accessor” methods are so common, that as of Objective-C 2.0, you don’t have to write them!

• You simply declare the desired instance variables as properties, and the compiler writes the setter/getter methods for you.
@interface Fraction: NSObject {
    int numerator;
    int denominator;
}

@property (assign) int numerator;
@property (assign) int denominator;

@end

@implementation Fraction

@synthesize numerator;
@synthesize denominator;

- (void) setDenominator: (int) d {
    if (d != 0)
        denominator = d;
}

@end

But what about the denominator? We don’t want it getting set to 0!
Properties

```objc
@interface Fraction : NSObject {
    int numerator;
    int denominator;
}

@property (assign) int numerator;
@property (assign) int denominator;

@end

@implementation Fraction

@synthesize numerator;

- (void) setDenominator: (int) d {
    if (d != 0)
        denominator = d;
}

- (int) denominator {
    return denominator;
}

@end

In fact, we don’t need to use `@synthesize` at all!

```
Properties

- But there must be something else neat about properties...
Properties

\[\text{aFraction.numerator; } \iff \text{[aFraction numerator];}\]

\[\text{aFraction.denominator = 5; } \iff \text{[aFraction setDenominator: 5];}\]

\[\text{owner.dog.tail } \iff \text{[[owner dog] tail]}\]
Protocols

• A protocol is simply a list of methods
• Shared among classes
• No corresponding implementations
Protocols

@protocol Drawing
- (void) paint;
- (void) erase;

@optional
- (void) outline;
@end

#import "Drawing.h"

@interface Rectangle : NSObject <Drawing> {
  // Insert instance variables
}

// Other methods here
@end

@implementation Rectangle

- (void) paint { ... }
- (void) erase { ... }

// Other methods here
@end
Protocols

Classes that implement all of the required methods of a protocol are said to “conform to” or “adopt” the protocol.

```swift
if ([obj conformsToProtocol: @protocol(Drawing)])
    [obj paint];
```