Review - Message Syntax

Method definition:

- (int) numberOfWords { ... }

Method call:

int words = [object numberOfWords];
Review - Message Syntax

Method definition:

- (int) rowsInSection: (int) section
  

Method call:

int rows = [table rowsInSection: 3];
Review - Message Syntax

Method definition:

- (void) moveX: (int) x andY: (int) y
{ ... }

Method call:

[turtle moveX: 5 andY: -4];
Selectors
Selectors

• "Objects" that identify method names
  • NOT method implementations
  • NOT function pointers

• Of type \texttt{SEL}

• Essential for polymorphism/dynamic binding
Creating Selectors

- `@selector()` compiler directive
- The SEL is assigned by the system at compile time

Example:

```objective-c
SEL setWidthHeight;
setWidthHeight = @selector(setWidth:height:);
```
Creating Selectors

The selector

@selector(setWidth:height:)

identifies all of these methods:

+ (void) setWidth: (int) w height: (int) h;
- (int) setWidth: (id) w height: (int) h;
- (void) setWidth: (float) w height: (float) h;

and many more...
Creating Selectors

• Selectors can also be created dynamically from **NSStrings** (though slightly less efficient)

• Use the **C FUNCTION**
  NSSelectorFromString()

• Example:

  ```
  SEL mySelector = NSSelectorFromString(someString);
  ```
Using Selectors

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

is equivalent to

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

is equivalent to

[fido fetch:theBall];
Why is this useful?

• Polymorphism!
• Allows very dynamic method dispatch
• Example (from Apple's documentation):

```objective-c
id helper = getTheReceiver();
SEL request = getTheSelector();
[helper performSelector: request];
```
Target-Action Pattern
Target-Action Pattern

• Control objects (elements in the graphical user interface of an application) give instructions to the application to do something. But how?
Target-Action Pattern

- Control objects (elements in the graphical user interface of an application) give instructions to the application to do something. But how?

- They send a message!
Target-Action Pattern

- Control objects (elements in the graphical user interface of an application) give instructions to the application to do something. But how?

- They send a message!

- An action is specified (a SEL) - the method to call when some event happens to the control
Target-Action Pattern

- Control objects (elements in the graphical user interface of an application) give instructions to the application to do something. But how?

- They send a message!

- An action is specified (a SEL) - the method to call when some event happens to the control

- A target is specified - the object that is to receive the message
Target-Action Example

/* Assume this is in some class */

NSButton *myButton;
[myButton setTarget: self];
[myButton.setAction: @selector(buttonClicked:)];

These slides are licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/ or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.
Safe Messaging

• The compiler will issue a warning for attempts to invoke a method that an object might not respond to

• It won't, however, catch `performSelector` with selectors that don't work

• Attempts to `performSelector` with a selector the object doesn't respond to will result in errors
Duck Typing

```swift
SEL quackSel = @selector(quack);
if ([obj respondsToSelector: quackSel]) {
    [obj quack];
}
```
Properties
Properties
Properties

@interface Fraction: NSObject
{
}
@end
Properties

@interface Fraction : NSObject
{

}

@end

@implementation Fraction

@end
Properties

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

@implementation Fraction
Properties

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

@implementation Fraction
- (void)setNumerator: (int)n;
- (void)setDenominator: (int)d;
@end
Properties

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

@implementation Fraction

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

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

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

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

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

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

@interface Fraction : NSObject
{
    int numerator;
    int denominator;
}
@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 since 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.
Properties
Properties

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

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

@implementation Fraction
@end
Properties

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

@property int numerator;
@property int denominator;
@end

@end
Properties

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

@property int numerator;
@property int denominator;

@end

@implementation Fraction

@synthesize numerator;
@synthesize denominator;

@end
Properties

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

@property int numerator;
@property int denominator;
@end

@implementation Fraction

@synthesize numerator;
@synthesize denominator;

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

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

@property int numerator;
@property 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

\[ a\text{Fraction}.\text{numerator} ; \iff [a\text{Fraction} \ \text{numerator}] ; \]

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

\[ \text{owner.dog.tail} \iff [[[\text{owner} \ \text{dog}] \ \text{tail}]] \]
Property Attributes

- Properties are even more powerful than just free setters and getters
- Property declarations can be decorated with various attributes that will alter the behavior of the accessor methods that the compiler writes (or specify an API for ones you write)
Accessor Method Names
Accessor Method Names

- Change the getter's name
  
  \texttt{getter=\textit{getterName}}

- Default getter name: \textit{propertyName}
Accessor Method Names

- Change the getter's name
  `getter = getterName`

- Default getter name: `propertyName`

- Change the setter's name
  `setter = setterName`

- Default setter name: `setPropertyPropertyName`
Writability
Writability

- Read and write (default)
  readwrite
Writability

• Read and write (default)
  readwrite

• Read-only - no setter is generated
  readonly
Setter Semantics
Setter Semantics

• Specifies the behavior of the setter
Setter Semantics

• Specifies the behavior of the setter
• assign - simple assignment (default)
Setter Semantics

• Specifies the behavior of the setter

• assign - simple assignment (default)

• retain - invoke retain on the object upon assignment (more on this later)
Setter Semantics

• Specifies the behavior of the setter

• assign - simple assignment (default)

• retain - invoke retain on the object upon assignment (more on this later)

• copy - assign a copy of the object
Atomicity

- By default, the accessors are atomic
  - One thread at a time
- The attribute `nonatomic` can be used to make the accessor methods not atomic
  - Faster because the runtime system does not need to use locks/semaphores to make the accessors atomic
Examples

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

@property (nonatomic, assign) int numerator;
@property (nonatomic, assign) int denominator;
@end
Examples

@interface CheckersGame : NSObject
{
    NSMutableArray *board;
}
@property (retain) NSMutableArray *board;
@end
Examples

@interface Person : NSObject
{
    int age;
}

@property (copy) NSString *name;
@end
Examples

@interface Lightbulb : NSObject
{

@end

@property (getter=isLit, setter=setLight:) BOOL lit;
@end
Protocols
Protocols

- A protocol is simply a list of methods
- Shared among classes
- No corresponding implementations
- Exact same idea as Java interfaces
(Formal) Protocol Syntax

@protocol ProtocolName

/* Method declarations */

@end
Optional and Required

@interface MyProtocol

- (void) requiredMethod;

@optional
- (void) optionalMethod;
- (void) anotherOptionalMethod;

@required
- (void) anotherRequiredMethod;

@end
Example
Example

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

@optional
- (void) outline;
@end
Example

@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 {
    /* Other methods here */
}
-
  (void) erase {
    /* Other methods here */
}
@end
Protocols

- Protocol objects can be created using the `@protocol()` directive

- Example:
  ```
  Protocol *myXMLSupportProtocol = @protocol(MyXMLSupport);
  ```
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];
```
Protocols

• Classes can implement many protocols

• Simply comma-separate the protocol names in the angle-brackets, e.g.

```objc
@interface MyClass : NSObject <Proto1, Proto2, ...>
{
    ...
}
...
@end
```
Protocol Type Declarations

- Object type declarations can be extended to include formal protocols.

- Examples:
  
  ```
  id<MyProtocol> anObject;
  MyObject<MyProtocol> *anObject;
  ```
Categories
Categories

- Two common uses:
  - Class extension
  - "Informal" protocols
Categories

ClassName+CategoryName.h

#import "ClassName.h"

@interface ClassName (CategoryName)
/* method declarations */
@end

ClassName+CategoryName.m

#import "ClassName+CategoryName.h"

@implementation ClassName (CategoryName)
/* method definitions */
@end
"Informal" Protocol

@interface NSObject (MyXMLSupport)

- (id) initFromXMLRepresentation:
  (NSXMLElement *) XMLElement;

- (NSXMLElement *) XMLRepresentation;

@end
Exceptions
Exception Handling

- Exceptions are thrown using the @throw directive, followed by an exception - basically an Objective-C object (usually an NSError)

- Exceptions are handled using the @try, @catch, and @finally compiler directives
Exception Handling

@try
{
    /* Code that may throw an exception */
}
@catch (SomeException *exception)
{
    /* Handle the exception */
}
@finally
{
    /* Do stuff after (exception or no) */
}