Stanford CS193p
Developing Applications for iPhone 4, iPod Touch, & iPad
Spring 2011
Stanford Logistics

Please get your Axess situation right
If you are not in Axess, please add yourself.
If you are in Axess, be sure your grading option is correct.

Homework
If you have not downloaded the SDK by now, it should be your top priority.
Don’t wait until Wednesday night to try your first submission.
Only your last submission will be graded.
Use corn.stanford.edu (not myth machines). Make sure your README is truly “plain” text.

Friday Section
Xcode 4
Running your homework on your device
Office Hours

🔗 Jeff
Sunday, 1pm to 3pm, Meyer Library, 2nd Floor, Area 220
Wednesday, 1pm to 3pm, Meyer Library, 2nd Floor, Area 220

🔗 Salik
Sunday, 8pm to 10pm, Meyer Library, 2nd Floor
TBD

🔗 Beyang
Tuesday, 7pm to 9pm, Meyer Library, 2nd Floor
Friday, 12:15-2:15pm, Meyer Library, 2nd Floor

🔗 Paul
Tuesday & Thursday 3:30pm, 420-041
Today

Objective-C
Methods (Class and Instance)
Instance Variables
Properties
Dynamic Binding
Introspection
nil and BOOL
Method Syntax

- (NSArray *)shipsAtPoint:(CGPoint)bombLocation withDamage:(BOOL)damaged;

Dash for instance method.
Plus for class method.
Will explain difference in a moment.
Method Syntax

- (NSArray *)shipsAtPoint:(CGPoint)bombLocation withDamage:(BOOL)damaged;

Return type in parentheses
Method Syntax

- (NSArray *)shipsAtPoint:(CGPoint)bombLocation withDamage:(BOOL)damaged;

First part of method name.
Full name is shipsAtPoint:withDamage:
Method Syntax

- (NSArray *)shipsAtPoint:(CGPoint)bombLocation withDamage:(BOOL)damaged;

First part of method name.
Full name is shipsAtPoint:withDamage:

Second part of method name.
Method Syntax

- (NSArray *)shipsAtPoint:(CGPoint)bombLocation withDamage:(BOOL)damaged;

Type of first argument in parentheses. This one happens to be a C struct.
Method Syntax

- (NSArray *) shipsAtPoint:(CGPoint) bombLocation withDamage:(BOOL) damaged;

Type of first argument in parentheses. This one happens to be a C struct.

Type of second argument in parentheses. This one is a BOOL (boolean value).
Method Syntax

- (NSArray *)shipsAtPoint:(CGPoint)bombLocation withDamage:(BOOL)damaged;

Name of first argument.
Use it like a local variable inside method definition.
Method Syntax

- (NSArray *)shipsAtPoint:(CGPoint)bombLocation withDamage:(BOOL)damaged;

Name of first argument.
Use it like a local variable inside method definition.

Name of second argument.
Method Syntax

- (NSArray *)shipsAtPoint:(CGPoint)bombLocation withDamage:(BOOL)damaged;

Line up colons when there are lots of arguments (or argument names are long).

- (void)splitViewController:(UISplitViewController*)svc
  willHideViewController:(UIViewController *)aViewController
  withBarButtonItem:(UIBarButtonItem *)barButtonItem
  forPopoverController:(UIPopoverController *)popoverController;

Use IBAction (same as void) to alert Interface Builder of an action.

- (IBAction)digitPressed:(UIButton *)sender;
- (IBAction)digitPressed:(id)sender;
- (IBAction)digitPressed:sender;  // same as (id)sender version
- (IBAction)digitPressed;
Instance Methods

- (IBAction)digitPressed:(UIButton *)sender;

“Normal” methods you are used to

- Can access instance variables inside as if they were locals
- Can send messages to self and super inside
  - Both dispatch the message to the calling object, but use different implementations
  - If a superclass of yours calls a method on self, it will use your implementation (if one exists)

Example calling syntax:
  BOOL destroyed = [ship dropBomb:bombType at:dropPoint from:height];
Class Methods

- Starts with a plus. Used for allocation, singletons, utilities
  - + (id)alloc; // makes space for an object of the receiver's class (always pair w/init)
  - + (id)motherShip; // returns the one and only, shared (singleton) mother ship instance
  - + (int)turretsOnShipOfSize:(int)shipSize; // informational utility method

- Can not access instance variables inside

- Messages to self and super mean something a little different
  Both invoke only other class methods. Inheritance does work.

- Example calling syntax (a little different from instance methods)
  CalculatorBrain *brain = [[CalculatorBrain alloc] init];
  Ship *theMotherShip = [Ship motherShip];
  Ship *newShip = [Ship shipWithTurrentCount:5];
  int turretsOnMediumSizedShip = [Ship turretsOnShipOfSize:4];
Instance Variables

Scope
By default, instance variables are `@protected` (only the class and subclasses can access). Can be marked `@private` (only the class can access) or `@public` (anyone can access).

Scope syntax
```objective-c
@interface MyObject : NSObject
{
    int foo;
    @private
    int eye;
    @protected
    int bar;
    @public
    int forum;
    int apology;
    @private
    int jet;
}
```

Protected: foo & bar
Private: eye & jet
Public: forum & apology
Properties

Instance variables almost always want to be private
With the “modern compiler” this means putting them in the implementation file.
But there’s a nice way to let public API users set and get properties of your object ...

Use methods to set/get an instance variable’s value

@interface MyObject : NSObject
- (int)eye;
- (void)setEye:(int)anInt;
@end

If you do this, public API users can access your instance variable using “dot notation”

someObject.eye = newEyeValue; // set the instance variable
int eyeValue = someObject.eye; // get the instance variable’s current value
Properties

@property
You can declare these set/get methods more cleanly with an @property directive
@interface MyObject : NSObject
-
(int)eye;
-
(void)setEye:(int)anInt;
@end
Properties

@property
You can declare these set/get methods more cleanly with an @property directive.  
@interface MyObject : NSObject
@property int eye;
- (int)eye;
- (void)setEye:(int)anInt;
@end
Properties

@property
You can declare these set/get methods more cleanly with an @property directive
@interface MyObject : NSObject
@property int eye;
@end
Properties

@property

You can declare these set/get methods more cleanly with an @property directive
@interface MyObject : NSObject
@property int eye;
@end

If you use the readonly keyword, only the getter will be declared
@property (readonly) int eye;  // does not declare a setEye: method
Properties

But however you declare, you **must then implement** ...

For example, consider the following header (.h) file:

```objective-c
@interface MyObject : NSObject
@property int eye;
@end
```

The corresponding implementation (.m) file might look like this:

```objective-c
@implementation MyObject
{
    int eye;
}
- (int)eye
{
    return eye;
}
- (void)setEye:(int)anInt
{
    eye = anInt;
}
@end
```
Properties

There's no need for the instance variable to have the same name. For example, consider the following header (.h) file:

```c
@interface MyObject : NSObject
@property int eye;
@end
```

The corresponding implementation (.m) file could also look like this:

```c
@implementation MyObject
{
    int private_eye;
}
-(int)eye
{
    return private_eye;
}
-(void)setEye:(int)anInt
{
    private_eye = anInt;
}
@end
```
Properties

Or or even for a corresponding instance variable to exist!

Header (.h) file:

@interface MyObject : NSObject
@property (readonly) int eye;
@end

Implementation (.m) file:

@implementation MyObject
{
}
-(int)eye
{
    return <some calculated value for eye>;
}
@end
Properties

Compiler can generate **definitions** with `@synthesize`!

Header (.h) file:

```objc
@interface MyObject : NSObject
@property int eye;
@end
```

Implementation (.m) file:

```objc
@implementation MyObject
{
    int eye;
}

-(int)eye {
    return eye;
}
-(void)setEye:(int)anInt {
    eye = anInt;
}
@end
```
Properties

Compiler can generate definitions with @synthesize!

Header (.h) file:
@interface MyObject : NSObject
@property int eye;
@end

Implementation (.m) file:
@implementation MyObject
{
    int eye;
}
@synthesize eye;
- (int)eye {
    return eye;
}
- (void)setEye:(int)anInt {
    eye = anInt;
}
@end
Properties

Compiler can generate **definitions** with **@synthesize**!

Header (.h) file:

```c
@interface MyObject : NSObject
@property int eye;
@end
```

Implementation (.m) file:

```c
@implementation MyObject
{  
  int eye;
}
@end
```

```c
@synthesize eye;
@end
```
Properties

Again, the instance variable name can be different

Header (.h) file:
@interface MyObject : NSObject
@property int eye;
@end

Implementation (.m) file:
@implementation MyObject
{
    int private_eye;
}

- (int)eye {
    return eye;
}
- (void)setEye:(int)anInt {
    eye = anInt;
}
@end
Properties

Again, the instance variable name can be different

Header (.h) file:
@interface MyObject : NSObject
@property int eye;
@end

Implementation (.m) file:
@implementation MyObject
{
    int private_eye;
}
@synthesize eye = private_eye;
-(int)eye {
    return eye;
}
-(void)setEye:(int)anInt {
    eye = anInt;
}
@end
Properties

Again, the instance variable name can be different

Header (.h) file:

```objective-c
@interface MyObject : NSObject
@property int eye;
@end
```

Implementation (.m) file:

```objective-c
@implementation MyObject
{
    int private_eye;
}
@end
```

```objective-c
@synthesize eye = private_eye;
@end
```
@synthesize will even create the instance variable for you!

Header (.h) file:
@interface MyObject : NSObject
@property int eye;
@end

Implementation (.m) file:
@implementation MyObject
@end

Notice no public instance variables ...

... and no private instance variables.
**Properties**

@synthesize will even **create the instance variable** for you!

Header (.h) file:
@interface MyObject : NSObject
@property int eye;
@end

Implementation (.m) file:
@implementation MyObject
@synthesize eye;
@end

Notice no public instance variables ... And no private instance variables.
Properties

@synthesize will even create the instance variable for you!

Header (.h) file:
@interface MyObject : NSObject
@property int eye;
@end

Implementation (.m) file:
@implementation MyObject
@synthesize eye;
-(void)setEyeToTheMeaningOfLife // methods can access created ivar directly or thru property
{
    eye = 42; // or self.eye = 42, but those would be different! (more on this in a moment)
}
@end

Notice no public instance variables ...

... and no private instance variables.
Properties

@synthesize will even create the instance variable for you!

Header (.h) file:
@interface MyObject : NSObject
@property int eye;
@end

Implementation (.m) file:
@implementation MyObject
@synthesize eye;
-(void)setEyeToTheMeaningOfLife  // methods can access created ivar directly or thru property
{
    eye = 42;  // or self. eye = 42, but those would be different! (more on this in a moment)
}
@end

MyObject *someObject = ...;
someObject. eye = newEyeValue;  // this will still work (i.e. public use of the property)
int eyeValue = someObject. eye;  // and so will this
@synthesize will even create the instance variable for you!

Header (.h) file:
@interface MyObject : NSObject
@property int eye;
@end

Implementation (.m) file:
@implementation MyObject
@synthesize eye = private_eye;  // instance variable can even have a different name than property
-(void)setEyeToTheMeaningOfLife  // methods can access created ivar directly or thru property
{
  private_eye = 42;  // or self.eye = 42, but those would be different! (more on this in a moment)
}
@end

MyObject *someObject = ...;
someObject.eye = newEyeValue;  // this will still work (i.e. public use of the property)
int eyeValue = someObject.eye;  // and so will this

Notice no public instance variables ...
... and no private instance variables.
Properties

Even if you use `@synthesize`, you can still implement the setter or the getter (or both) and yours will win

```objc
@implementation MyObject
{
    int eye;
}
@synthesize eye;
-(void)setEye:(int)anInt {
    if (anInt > 0) eye = anInt;
}
@end
```

The method `- (int)eye` will still be implemented for you by `@synthesize`

Your implementation of `setEye:` is the one that will count (`@synthesize` will not generate one).

If you implemented both the setter and the getter, the `@synthesize` would not generate them,
but it still would create the instance variable for you if you didn’t do it yourself.
Properties

It’s common to use dot notation to access ivars inside your class
But it’s not the same as referencing the instance variable directly.
For example, if `eye` is an instance variable ...
```java
int x = eye;
```
... inside a method is not the same as ...
```java
int x = self.eye;
```
The latter calls the getter method (which is usually what you want for subclassability).

But occasionally things can go terribly wrong!
What’s wrong with the following code?
```java
-(void)setEye:(int)anInt
{
    self.eye = anInt;
}
```
Infinite loop. Can happen with the getter too ...
```java
-(int)eye {
    if (self.eye > 0) {
        return eye;
    } else {
        return -1;
    }
}
```
Properties

Why properties?

Most importantly, it provides safety and subclassability for instance variables. But the syntax also makes code look more consistent with C structs.

```c
typedef struct {
    float x;
    float y;
} Point;

@interface Bomb
@property Point position;
@end

@interface Ship : Vehicle
@property float width;
@property float height;
@property Point center;
-(BOOL)getHitByBomb:(Bomb *)bomb;
@end
```

Notice that we capitalize **Point** (just like a class name). It makes our C struct seem just like an object.
Why properties?
Most importantly, it provides safety and subclassability for instance variables.
But the syntax also makes code look more consistent with C structs.

typedef struct {
    float x;
    float y;
} Point;

@interface Bomb
@property Point position;
@end

@interface Ship : Vehicle
@property float width;
@property float height;
@property Point center;
-(BOOL)getsHitByBomb:(Bomb *)bomb;
@end

Instance variables may or may not exist here. Remember that @property is just declaring the property. Bomb would still have to implement setter and getter (could use @synthesize, of course).

Stanford CS193p
Spring 2011
Properties

Why properties?
Most importantly, it provides safety and subclassability for instance variables.
But the syntax also makes code look more consistent with C structs.

```c
typedef struct {
    float x;
    float y;
} Point;

@interface Bomb
@property Point position;
@end

@interface Ship : Vehicle
@property float width;
@property float height;
@property Point center;
-(BOOL)getsHitByBomb:(Bomb *)bomb;
@end
```

Returns whether the passed bomb would hit the receiving Ship.
Properties

Why properties?
Most importantly, it provides safety and subclassability for instance variables. But the syntax also makes code look more consistent with C structs.

```c
typedef struct {
    float x;
    float y;
} Point;
@interface Bomb
@property Point position;
@end
@interface Ship : Vehicle
@property float width;
@property float height;
@property Point center;
- (BOOL)getsHitByBomb:(Bomb *)bomb;
@end
@end
```

```c
@implementation Ship
@synthesize width, height, center;
- (BOOL)getsHitByBomb:(Bomb *)bomb
{
    float leftEdge = self.center.x - self.width/2;
    float rightEdge = ...;
    return ((bomb.position.x >= leftEdge) &&
            (bomb.position.x <= rightEdge) &&
            (bomb.position.y >= topEdge) &&
            (bomb.position.y <= bottomEdge));
}
@end
```

Notice access to instance variable using property of self.
Properties

Why properties?
Most importantly, it provides safety and subclassability for instance variables.
But the syntax also makes code look more consistent with C structs.

```c
typedef struct {
    float x;
    float y;
} Point;
@interface Bomb
@property Point position;
@end
@interface Ship : Vehicle
@property float width;
@property float height;
@property Point center;
- (BOOL)getsHitByBomb:(Bomb *)bomb;
@end
```

```objc
@implementation Ship
@synthesize width, height, center;
- (BOOL)getsHitByBomb:(Bomb *)bomb {
    float leftEdge = self.center.x - self.width/2;
    float rightEdge = ...
    return ((bomb.position.x >= leftEdge) &&
            (bomb.position.x <= rightEdge) &&
            (bomb.position.y >= topEdge) &&
            (bomb.position.y <= bottomEdge));
}
@end
```

Dot notation to reference an object’s property.
Properties

Why properties?
Most importantly, it provides safety and subclassability for instance variables. But the syntax also makes code look more consistent with C structs.

typedef struct {
    float x;
    float y;
} Point;

@interface Bomb
@property Point position;
@end
@interface Ship : Vehicle
@property float width;
@property float height;
@property Point center;
- (BOOL)getsHitByBomb:(Bomb *)bomb;
@end

@implementation Ship
@synthesize width, height, center;
- (BOOL)getsHitByBomb:(Bomb *)bomb
{
    float leftEdge = self.center.x - self.width/2;
    float rightEdge = ...
    return ((bomb.position.x >= leftEdge) &&
            (bomb.position.x <= rightEdge) &&
            (bomb.position.y >= topEdge) &&
            (bomb.position.y <= bottomEdge));
}
@end

Dot notation to reference an object’s property.
Do all `@property` s have to be public?

No. It is possible to declare a “private interface” to your class inside your implementation file.

Example (this is all in `MyObject’s .m` file):

```objc
@interface MyObject
@property double myEyesOnly;
@end

@implementation MyObject
@synthesize eye, myEyesOnly;
@end
```

`myEyesOnly` can only be set/get thru setter/getter via `self.myEyesOnly` since it is private.

(Actually, there’s really no such thing as a private method or property in Objective-C. But since we’re not declaring it in our header, no one will know about it, so it’s effectively private.)
Properties

There's more to think about when a `@property` is an object.
We'll postpone that discussion to later on when we talk about memory management.
Properties

Best Practices

When you are providing public access to the state of your object, use `@property`.
“Public access” might also mean “to subclassers” (no “protected” properties in Objective-C (yet)).
Infinitely preferable to accessing instance variables directly.
Probably best to always use `@synthesize` for all properties, even if you write both setter/getter.
Probably best to let `@synthesize` create the instance variable too (but be consistent either way).
Probably best to have the instance variable’s name match the property’s name (be consistent).
Internally to your object, if there is a property for an instance variable, use it.
Whether you create/use `private` properties is a code-readability choice (the “art” of programming).
Dynamic Binding

All objects are allocated in the heap, so you always use a pointer

Examples ...

```c
NSString *s = ...;  // “static” typed
id obj = s;
```

Never use “id *” (that would mean “a pointer to a pointer to an object”).

Decision about code to run on message send happens at runtime

Not at compile time. None of the decision is made at compile time.

Static typing (e.g. `NSString *` vs. `id`) is purely an aid to the compiler to help you find bugs.

If neither the class of the receiving object nor its superclasses implements that method: crash!

It is legal (and sometimes even good code) to “cast” a pointer

But we usually do it only after we’ve used “introspection” to find out more about the object.

More on introspection in a minute.

```c
id obj = ...;
NSString *s = (NSString *)obj;  // dangerous ... best know what you are doing
```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;

No compiler warning. Perfectly legal since s “isa” Vehicle.
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

Compiler warning!
Would not crash at runtime though.
But only because we know v is a Ship.
Compiler only knows v is a Vehicle.
Object Typing

@interface Vehicle
- (void)move;
@end

@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];

The compiler knows that the method shoot exists, so it's not impossible that obj might respond to it. But we have not typed obj enough for the compiler to be sure it's wrong. So no warning.

Might crash at runtime if obj is not a Ship (or an object of some other class that implements a shoot method).
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end
Ship *s = [[Ship alloc] init];
[s shoot];
[s move];
Vehicle *v = s;
[v shoot];
id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];

Compiler warning!
Compiler has never heard of this method. Therefore it's pretty sure obj will not respond to it.
@interface Vehicle
- (void)move;
@end

@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];

NSString *hello = @"hello";
[hello shoot];
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];

NSString *hello = @"hello";
[hello shoot];
Ship *helloShip = (Ship *)hello;

*No compiler warning.*
We are “casting” here.
The compiler thinks we know what we’re doing.
Object Typing

@interface Vehicle
- (void)move;
@end

@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];

NSString *hello = @"hello";
[hello shoot];
Ship *helloShip = (Ship *)hello;
[helloShip shoot];

No compiler warning!
We’ve forced the compiler to think that the NSString is a Ship.
“All’s well,” the compiler thinks.
Object Typing

@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];

NSString *hello = @"hello";
[hello shoot];
Ship *helloShip = (Ship *)hello;
[helloShip shoot];
[(id)hello shoot];
Introspection

So when do we use id? Isn’t it always bad?
No, we might have a collection (e.g. an array) of objects of different classes.
But we’d have to be sure we know which was which before we sent messages to them.
How do we do that? Introspection.

All objects that inherit from NSObject know these methods
isKindOfClass: returns whether an object is that kind of class (inheritance included)
memberOfClass: returns whether an object is that kind of class (no inheritance)
respondsToSelector: returns whether an object responds to a given method

Arguments to these methods are a little tricky
Class testing methods take a Class
You get a Class by sending the class method class to a class :)
if ([obj isKindOfClass:[NSString class]]) {
    NSString *s = [(NSString *)obj stringByAppendingString:@"xyzzy"];
}
Introspection

Method testing methods take a selector (SEL)
Special @selector() directive turns the name of a method into a selector
if ([obj respondsToSelector:@selector(shoot)]) {
    [obj shoot];
}

SEL is the Objective-C “type” for a selector
SEL shootSelector = @selector(shoot);
SEL moveToSelector = @selector(moveTo:);
Target/action uses this, e.g. [button addTarget:self action:@selector(digitPressed:)]

If you have a SEL, you can ask an object to perform it
Using the performSelector: or performSelector:withObject: methods in NSObject
[obj performSelector:shootSelector];
[obj performSelector:moveToSelector withObject:coordinate];
The value of an object pointer that does not point to anything

```objective-c
id obj = nil;
NSString *hello = nil;
```

Like “zero” for a primitive type (`int`, `double`, etc.)

Actually, it’s not “like” zero: it is zero.

`NSObject` sets all its instance variables to zero

Thus, instance variables that are pointers to objects start out with the value of `nil`.

Can be implicitly tested in an `if` statement

```objective-c
if (obj) { } // curly braces will execute if `obj` points to an object
```

Sending messages to `nil` is (mostly) okay. No code gets executed.

If the method returns a value, it will return zero.

```objective-c
int i = [obj methodWhichReturnsAnInt]; // `i` will be zero if `obj` is `nil`
```

Be careful if the method returns a C struct. Return value is undefined.

```objective-c
CGPoint p = [obj getLocation]; // `p` will have an undefined value if `obj` is `nil`
```
Objective-C’s boolean “type” (actually just a typedef)

Can be tested implicitly

```c
if (flag) { }
if (!flag) { }

YES means “true,” NO means “false”

NO == 0, YES is anything else

if (flag == YES) { }
if (flag == NO) { }
if (flag != NO) { }
```