Review (Basic Objective-C)

- Classes

  Header .h (public) versus Implementation .m (private)

  `@interface MyClass : MySuperclass … @end` (only in header file)

  `@interface MyClass() … @end` (only in implementation file)

  `@implementation … @end` (only in implementation file)

  `#import`

- Properties

  `@property (nonatomic) <type> <property name>` (always `nonatomic` in this course)

  It's just setter and getter methods. Default ones automatically generated for you by compiler.

  Better than instance variables alone (lazy instantiation, consistency checking, UI updating, etc.).

  `@property (strong or weak) <type which is a pointer to an object> <property name>`

  `@property (getter=<getter name>) …`

  `@property (readonly) …` & `@property (readwrite) …`

  Invoking setter and getter using dot notation, e.g., `self.cards = …` or `if (rank > self.rank) …`

  `@synthesize <prop name> = _<prop name>` (only if you implement both setter and getter)

Review (Basic Objective-C)

- ## Types and Memory

  Types: `MyClass *`, `BOOL` (`YES` or `NO`), `int`, `NSUInteger`, etc. (`id` not fully explained yet.)

  All objects live in the heap (i.e. we only have pointers to them).

  Object storage in the heap is managed automatically (guided by `strong` and `weak` declarations).

  Lazy instantiation (using a `@property`'s getter to `alloc`ate and `init`ialize the object that the `@property` points to in an "on demand" fashion). Not everything is lazily instantiated, btw. :)

  If a pointer has the value `nil` (i.e. `0`), it means the pointer does not point to anything.

- ## Methods

  Declaring and defining instance methods, e.g., `- (int)match:(NSArray *)otherCards`

  Declaring and defining class methods, e.g., `+ (NSArray *)validSuits`

  Invoking instance methods, e.g., `[myArray addObject:anObject]`

  Invoking class methods, e.g., `unsigned int rank = [PlayingCard maxRank]`

  Method's name and its parameters are interspersed, e.g., `[deck addCard:aCard atTop:YES]`

Review (Basic Objective-C)

- NSString

  Immutable and usually created by manipulating other strings or @“” notation or class methods.

  e.g. `NSString *myString = @“hello”`

  e.g. `NSString *myString = [otherString stringByAppendingString:yetAnotherString]`

  e.g. `NSString *myString = [NSString stringWithFormat:@“%d%@”, myInt, myObj]`

  There is an NSMutableString subclass but we almost never use it.

  Instead, we create new strings by asking existing ones to create a modified version of themselves.

Review (Basic Objective-C)

- NSArray

  Immutable and usually created by manipulating other arrays (not seen yet) or with `@[]` notation.

  `@[@“a”,@“b”]` is the same as `[[NSArray alloc] initWithObjects:@“a”,@“b”,nil]`.

  Access the array using `[]` notation (like a normal C array), e.g., `myArray[index]`.

  `myArray[index]` works the same as `[myArray objectAtIndex:index]`.

  The method `count` (which returns `NSUInteger`) will tell you how many items in the array.

  (We accidentally used dot notation to call this method in Lecture 2!)

  Be careful not to access array index out of bounds (crashes). Only `last/firstObject` immune.

  Can contain any mix of objects of any class) No syntax to say which it contains.

  Use `NSMutableArray` subclass if mutability is needed. Then you get …

  ```
  - (void)addObject:(id)anObject;
  - (void)insertObject:(id)anObject atIndex:(int)index;
  - (void)removeObjectAtIndex:(int)index;
  ```

  Usually created with `[[NSMutableArray alloc] init]`

Review (Basic Objective-C)

Creating Objects in the Heap

Allocation (`NSObject`'s `alloc`) and initialization (with an `init`... method) always happen together!

e.g. `[[NSMutableArray alloc] init]`

e.g. `[[CardMatchingGame alloc] initWithCardCount:c usingDeck:d]`

Writing initializers for your own classes ...

Two kinds of initializers: designated (one per class) and convenience (zero or more per class).

Only denoted by comments (not enforced by the syntax of the language in any way).

Must call your `super`'s designated initializer (from your designated initializer)
or your own designated initializer (from your own convenience initializers).

This whole concept takes some getting used to.

Luckily, because of lazy instantiation, et. al., we don't need initializers that much in Objective-C.

And calling initializers is easy (it's just `alloc` plus whatever `init`ializer you can find that you like).