Object Orientated Programming

Week 1 Day 3

What are objects?

So far we've seen that objects are things that have properties where we can store data and methods that can act on that data. We also saw that classes are "blueprints" for creating objects.

That's just the start...

OOP - Inheritance

Classes don’t have to be defined from scratch. A class can inherit from another class. This means that the ‘subclass’ gets all the same variables and methods as the class it inherits from (the ‘superclass’), and we can add more variables and methods. We can create a tree of inheritance to build up complex objects a bit at a time.

Examples of inheritance

We've already used subclasses. NSString and NSArray both have subclasses we've used...

Also, **every** class we've used has been a subclass of the base class: `NSObject`

The *self* and *super* keywords

We can use `self` in an object to refer to the methods in its *most* inherited class. `super` refers to the class one step up the class hierarchy. We can use `super` in overridden methods to do the same function as in the superclass while adding some of our own. We can extend the behaviour, not just replace it.

Code example: overriding methods

Here’s a code example where we’re overriding methods in order to add new functionality on top of the existing functionality.

OOP - Initialization

When we create objects, we need to give them a chance to set up their default state. This is the purpose of the `init` method. `init` is defined by `NSObject`, and often overridden by subclasses.

Custom initializers

Init methods are just that: methods. We can write custom initializers that take parameters, to signal that these parameters are required by this subclass to function correctly.

e.g.

```
-(instancetype)initWithString:(NSString *)str;
```

Custom initializers - IMPORTANT!

When you write a custom initializer, or override an existing initializer, you have to make sure your parent class' initializer is called, or your objects may not be set up correctly.

e.g. `self = [super init];`

Nil: more than nothing

Represents the non-existence of an object.

- nil is not the same as NULL.
- Messages can be sent to nil. Nil returns 0 or a zero-like value for all message sent to it.

Using nil

As we've mentioned already, nil can be used for references to objects in order to note "no object is actually here". Then we can check for nil before we access an object reference, if we need to. But sometimes, we don't even need to do that.

Using nil, part 2

Messages sent to nil are valid, but ignored. This is very useful.

There’s a class called NSNull that you can use to represent nothing where nil is not accepted: e.g NSArray, NSDictionary, etc.

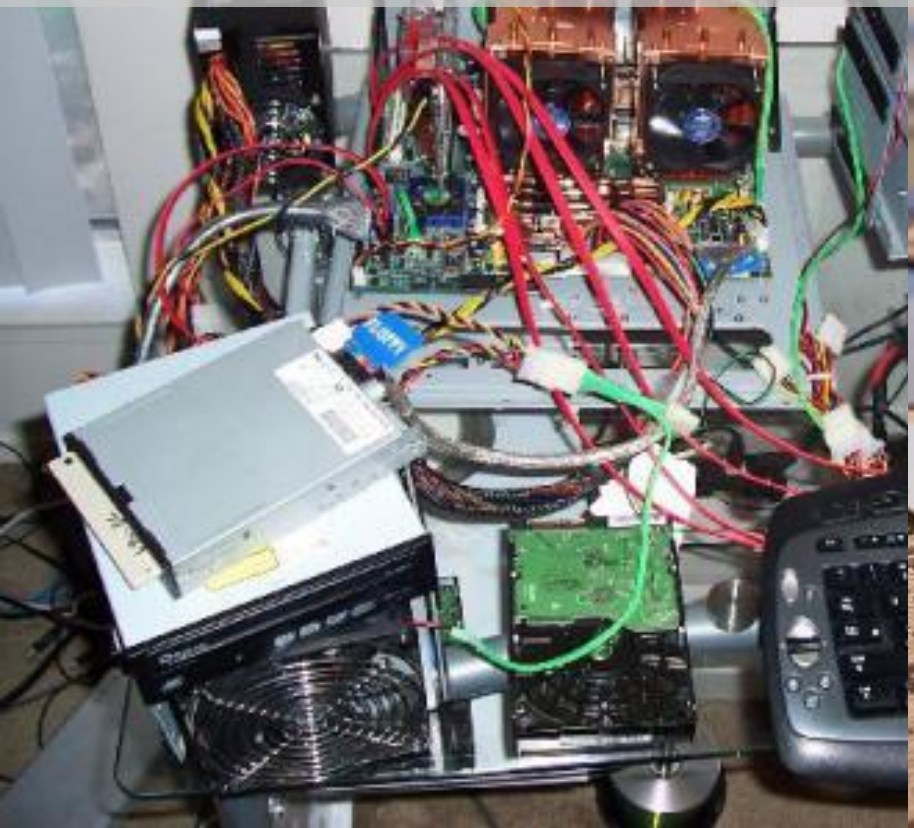
Code example: using nil

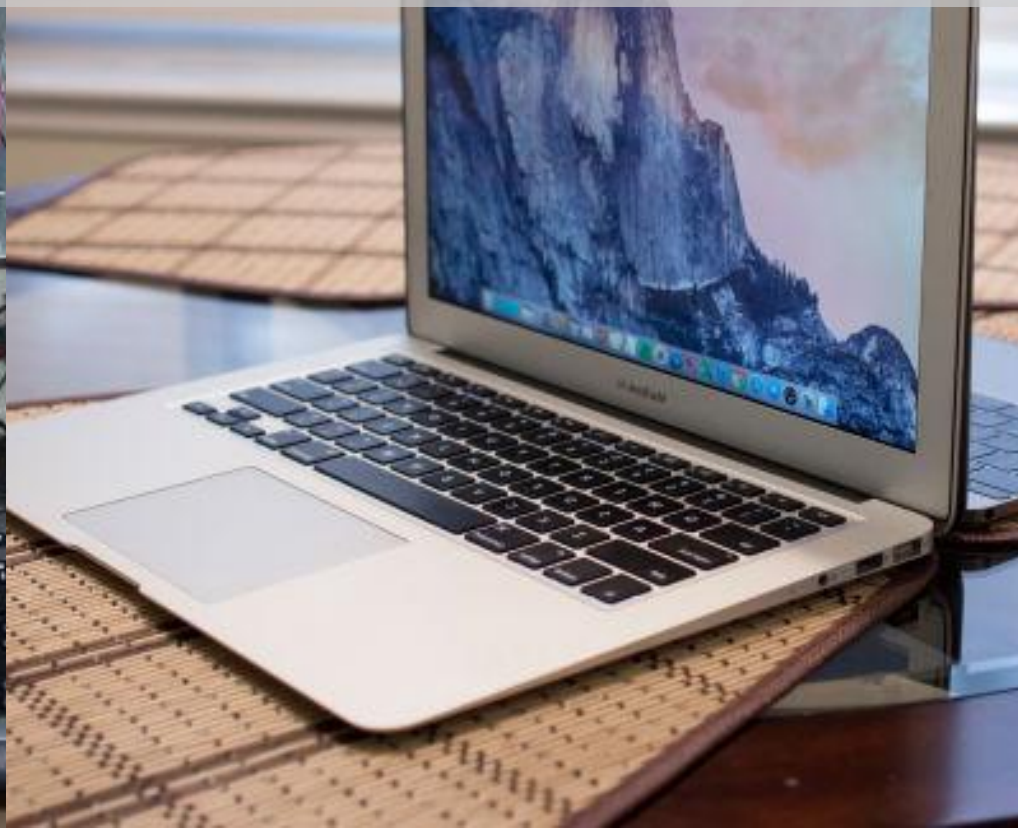
Here’s an example of how to use nil and when to check if an object is actually nil before using it.

OOP - Encapsulation

The data and methods used by an object do not have to be publicly accessible. We can set boundaries on how other objects interact with an object.

PUBLIC / PRIVATE





Interface vs implementation

In Objective-C, the part of the object accessible to the outside world is defined in its interface (.h). The part that's hidden, the encapsulated part, is defined in the implementation (.m). We can put properties and methods inside of the implementation file so that the object can use them internally.

Modularity

Because we have a defined interface for an object, we can switch out different objects for that object if they have the same methods. This allows us to change functionality behind the scenes while not needing to change other parts of the program.

The big question: why?

We use object-oriented design because it is a way of structuring a program that encourages us to think abstractly and understand complex systems more easily than procedurally-oriented languages like C.

One more thing...

One more thing...

NSSet & NSMutableSet

NSSet

Like arrays, sets are also a collection of objects. Unlike arrays, they don’t care about the order, and they don’t allow duplicates. These properties make them better suited than arrays in some situations.

NSSet

```
NSSet *foods = [[NSSet alloc]
initWithArray:@[@”tacos”, @”burgers”,
@”tacos”]];

// foods.count == 2
```

NSSet

- There is no literal syntax for sets or mutable sets
- Sets can be filtered, like arrays
- You can get the intersection of two sets
- You can ask if one set is a subset of another
- You can ask if an object is a member of a set

NSMutableSet

- addObject:
- containsObject:
- unionSet:
- minusSet:
- removeObject:
- removeAllObjects