TODAY'S AGENDA

1. CUSTOM CLASS FILES
    A. PROPERTIES
    B. METHODS
2. SINGLETONS
3. INHERITANCE
4. CATEGORIES
5. MVC

FILES & SLIDES

Week 3 Folder

bit.ly/18e5tW9

01 CUSTOM CLASSES

Private. Public. Methods & Properties

PROPERTIES

atomic (default) - It will lock the code which will set or get your attribute and make sure that process is completed before it allow any other code to execute. (DON'T USE)

nonatomic - this property getter and setter are not thread safe
We do not need locking. (USE IT)

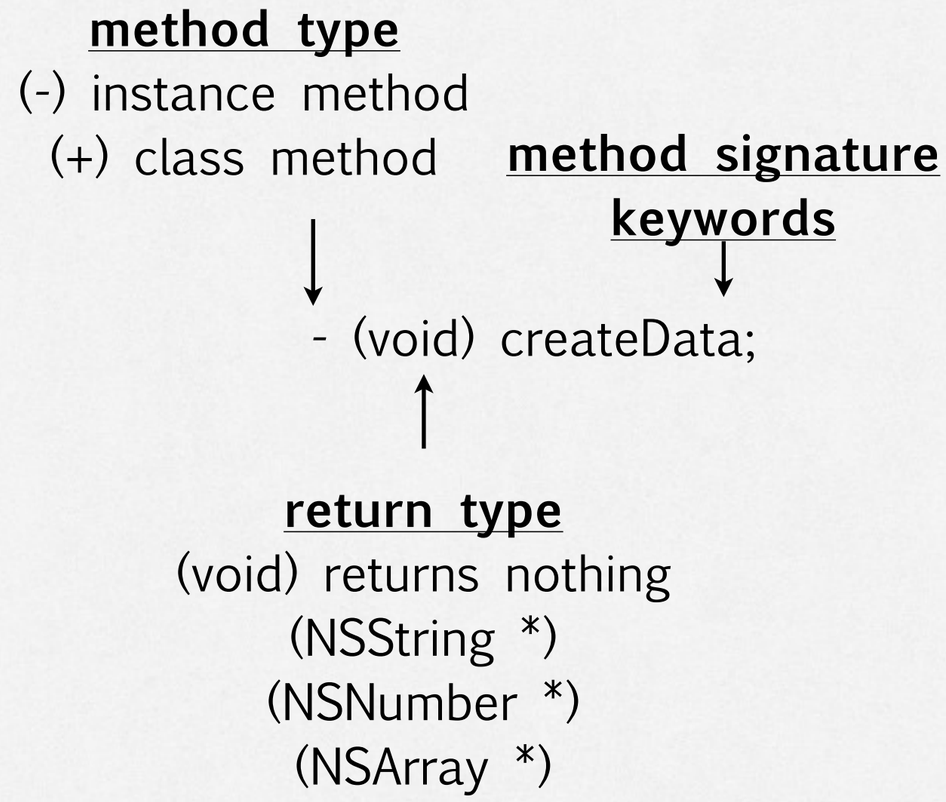
strong - when using ARC strong (helps the compiler) says I want this object to stay in the heap as long as I point to it
example setting an object to 0 or nil ARC will remove it.

weak - only keep this in the heap as long as someone else is pointing to it strong (only interested in pointing this as long as someone else in interested in pointing to it)

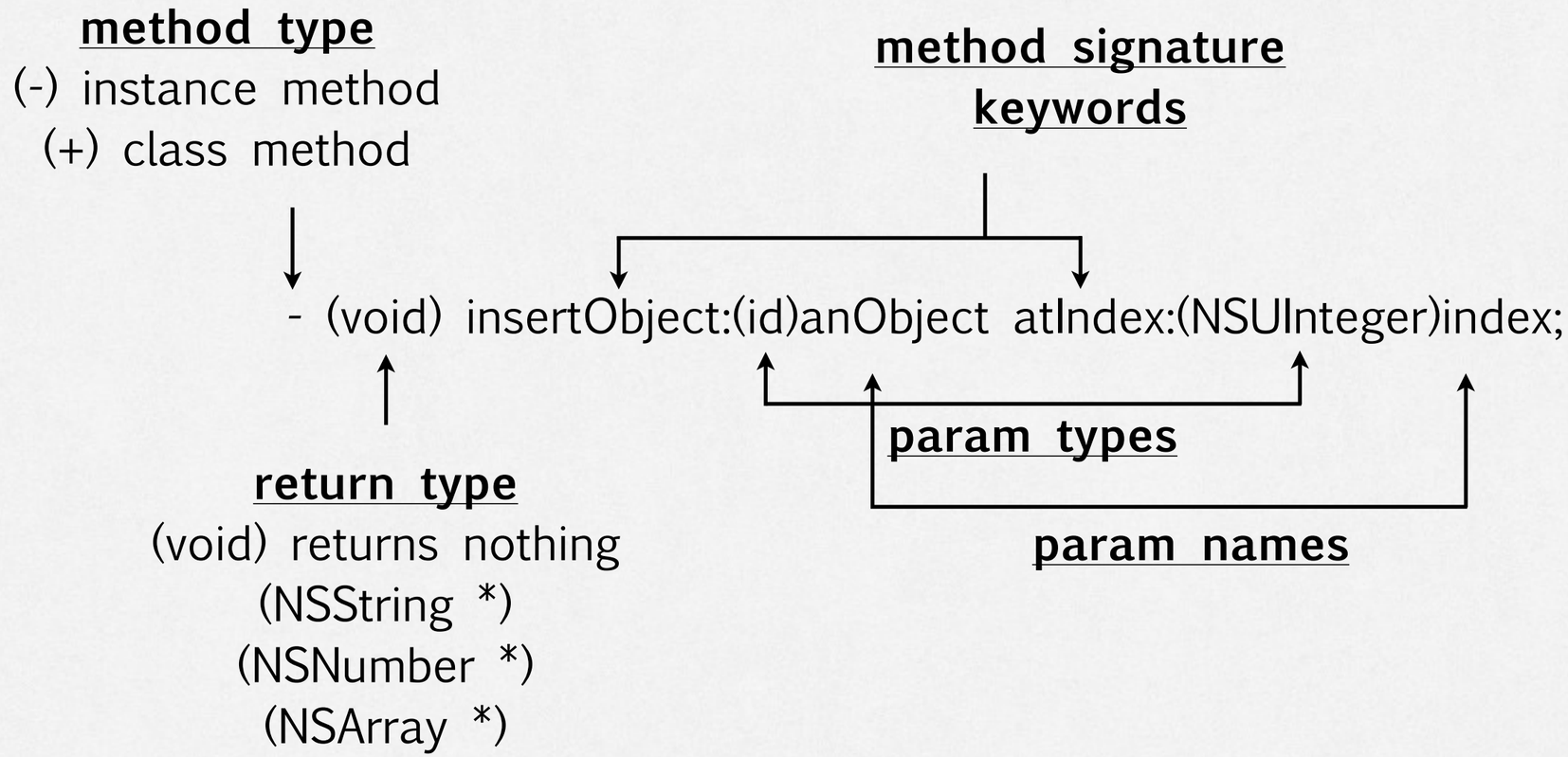
PROPERTIES

```
@property (nonatomic, strong) NSString* title;
@property (nonatomic, isGetter=isCompleted) BOOL completed;
@property (nonatomic, strong) NSNumber* age;
@property (nonatomic) int weight;
@property (nonatomic) float winningPct;
```

METHODS NO PARAMETERS



METHOD WITH PARAMETERS



CUSTOM CLASS FILES

Person.h

```objc
#import <Foundation/Foundation.h>

@interface Person : NSObject
// declaration of public methods

@property (nonatomic, strong) NSString* firstName;
@property (nonatomic, strong) NSString* lastName;
@property (nonatomic, strong) NSNumber* age;



- (NSString *) fullName:(NSString *)firstName
withLastName:(NSString *)lastName;



@end
```

Person.m

```objc
#import "Person.h"

@interface Person()
// declaration of private methods (as needed)

@end

@implementation Person
// implementation of public and private methods
- (NSString *) fullName:(NSString *)firstName
withLastName:(NSString *)lastName
{
   return @"";
}



@end
```

02 SINGLETON

Created only once

SINGLETONS

One of my most used design patterns when developing for iOS is the singleton pattern. It's an extremely powerful way to share data between different parts of code without having to pass the data around manually. Singleton classes are an important concept to understand because they exhibit an extremely useful design pattern.

Basically once a singleton is created it can't be created again which is where it gets the name. All Models are created with singletons because you really only one your data in one place. Singletons are used for other things but commonly used for Models.

SINGLETON

MyManager.h

```
#import <Foundation/Foundation.h>

@interface MyManager : NSObject
// declaration of public methods

@property (nonatomic, strong) NSString*
someProperty;



+ (id) sharedManager;









@end
```

MyManager.m

```
#import "MyManager.h"

@implementation MyManager

+ (id)sharedManager {
    static MyManager *sharedMyManager = nil;
    @synchronized(self) {
        if (sharedMyManager == nil)
            sharedMyManager = [[self alloc] init];
    }
    return sharedMyManager;
}

- (id)init
{
  if (self = [super init]) {
      someProperty = [[NSString alloc]
initWithString:@"Default Property Value"];
  }
  return self;
}
```

SINGLETON

MyManager.h

```
#import <Foundation/Foundation.h>

@interface MyManager : NSObject
// declaration of public methods

@property (nonatomic, strong) NSString*
someProperty;

+ (id) sharedManager;




@end
```

MyManager.m

```
#import "MyManager.h"

@implementation MyManager

+ (id)sharedManager {
    static MyManager *sharedMyManager = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedMyManager = [[self alloc] init];
    });

    return sharedMyManager;
}

- (id)init
{
  if (self = [super init]) {
      someProperty = [[NSString alloc]
initWithString:@"Default Property Value"];
  }
  return self;
}
```

SINGLETONS

You would use the singleton like this:
MyManager *sharedManager = [MyManager sharedManager];

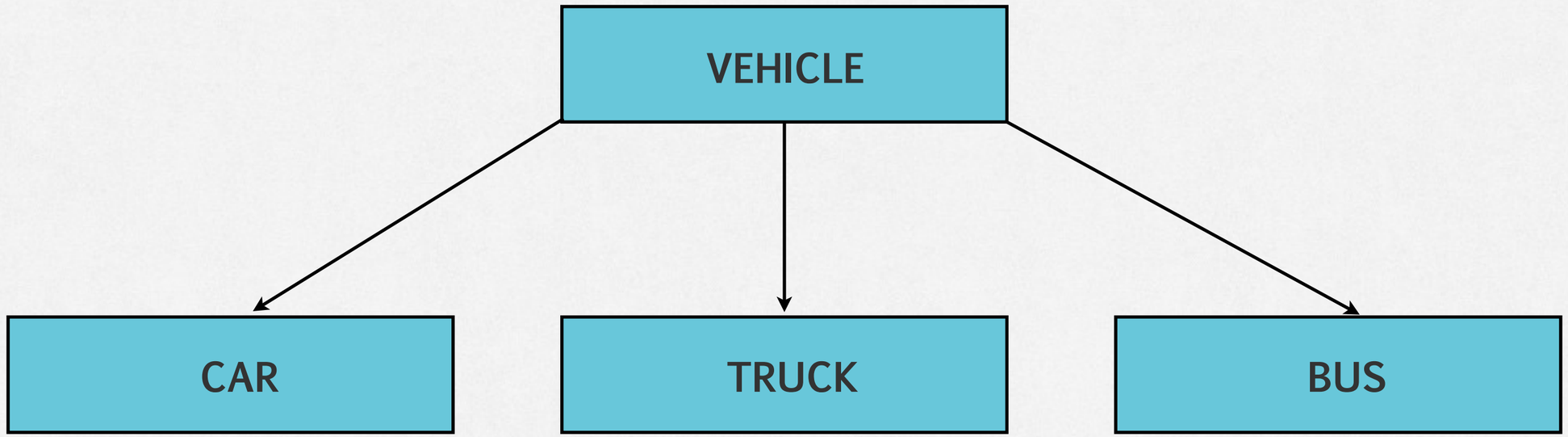
03 INHERITANCE

Why did I get my dad's hairline?

INHERITANCE

As the name inheritance suggests an object is able to inherit characteristics from another object. In more concrete terms, an object is able to pass on its state and behaviors to its children. For inheritance to work the objects need to have characteristics in common with each other.

INHERITANCE



INHERITANCE

Employee.h

```
#import “Person.h”

@interface Employee : Person
// declaration of public methods


@property (nonatomic, strong) NSDate* startDate;
@property (nonatomic)         BOOL* isManager;
@property (nonatomic, strong) NSInteger* daysEmployed;









@end
```

Employee.m

```
#import "Employee.h"

@interface Employee()
// declaration of private methods (as needed)
- (int) daysEmployed:(NSDate *)date;

@end



@implementation Person
// implementation of public and private methods

- (NSInteger *) daysEmployed:(NSDate *)date
{

   return daysEmployed;
}

@end
```

04 CATEGORIES

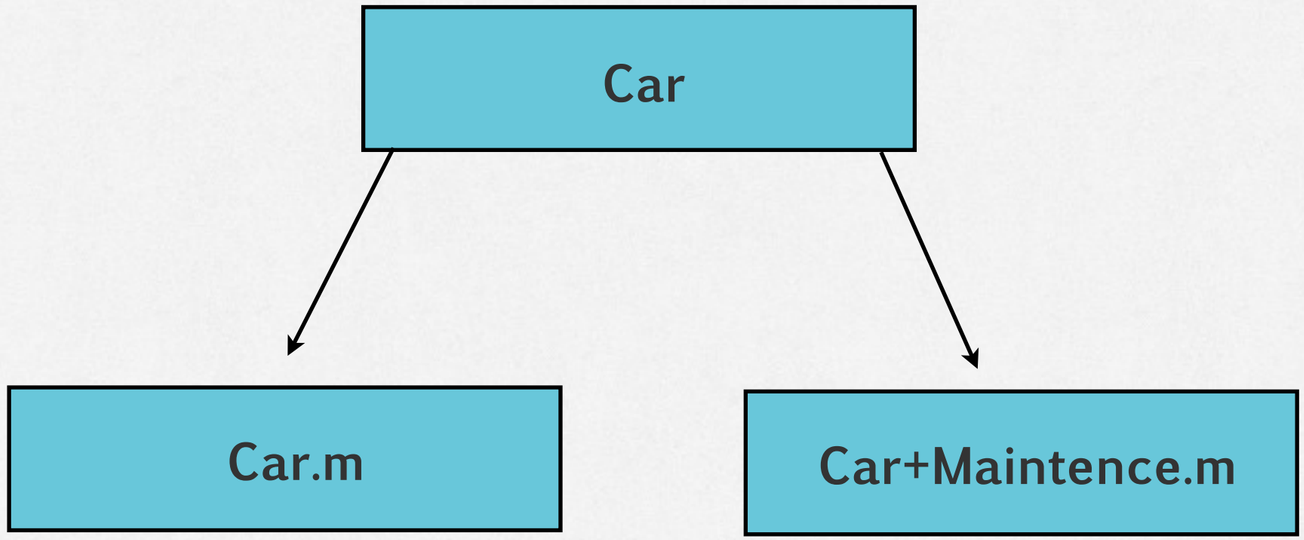
Custom Features

CATEGORIES

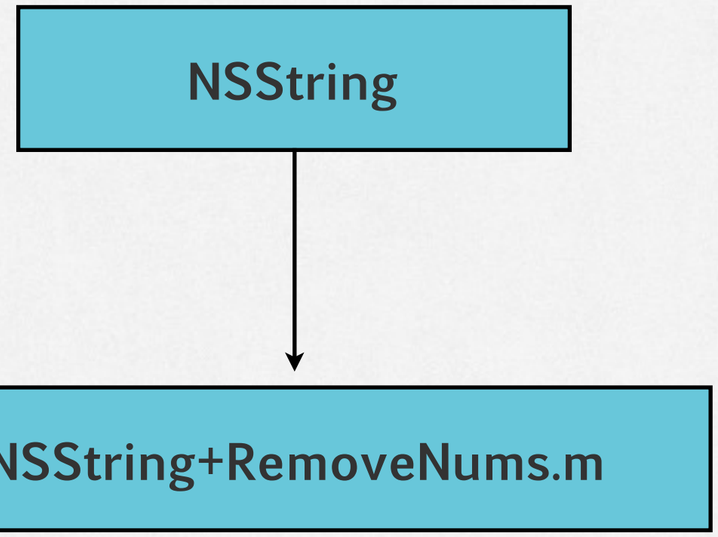
Categories provide the ability to add functionality to an object without subclassing or changing the actual object. A handy tool, they are often used to add methods to existing classes, such as NSString or your own custom objects.

Their goal is to ease the burden of maintaining large code bases by modularizing a class. This prevents your source code from becoming monolithic 10000+ line files that are impossible to navigate and makes it easy to assign specific, well-defined portions of a class to individual developers.

CATEGORY



CATEGORY



CATEGORY

NSString+RemoveNums.h

```
#import <Foundation/Foundation.h>

@interface NSString (RemoveNums)
// declaration of public methods

- (NSString *) removeNumbersFromString:(NSString
*)string;




@end
```

NSString+RemoveNums.m

```
#import "Person.h"

@implementation NSString (RemoveNums)
// implementation of public and private methods

- (NSString *) removeNumbersFromString:(NSString
*)string
{
    NSString *trimmedString = nil;
    NSCharacterSet *numbersSet = [NSCharacterSet
characterSetWithCharactersInString:@"0123456789"];
    trimmedString = [string
stringByTrimmingCharactersInSet:numbersSet];

    return trimmedString;
}




@end
```

CATEGORIES

Subclassing is one way to add functionality to an object, but avoiding unnecessary subclassing by using a category will help reduce the amount of code and keep your projects more organized. There are a number of scenarios where using a category is beneficial.

05 MVC

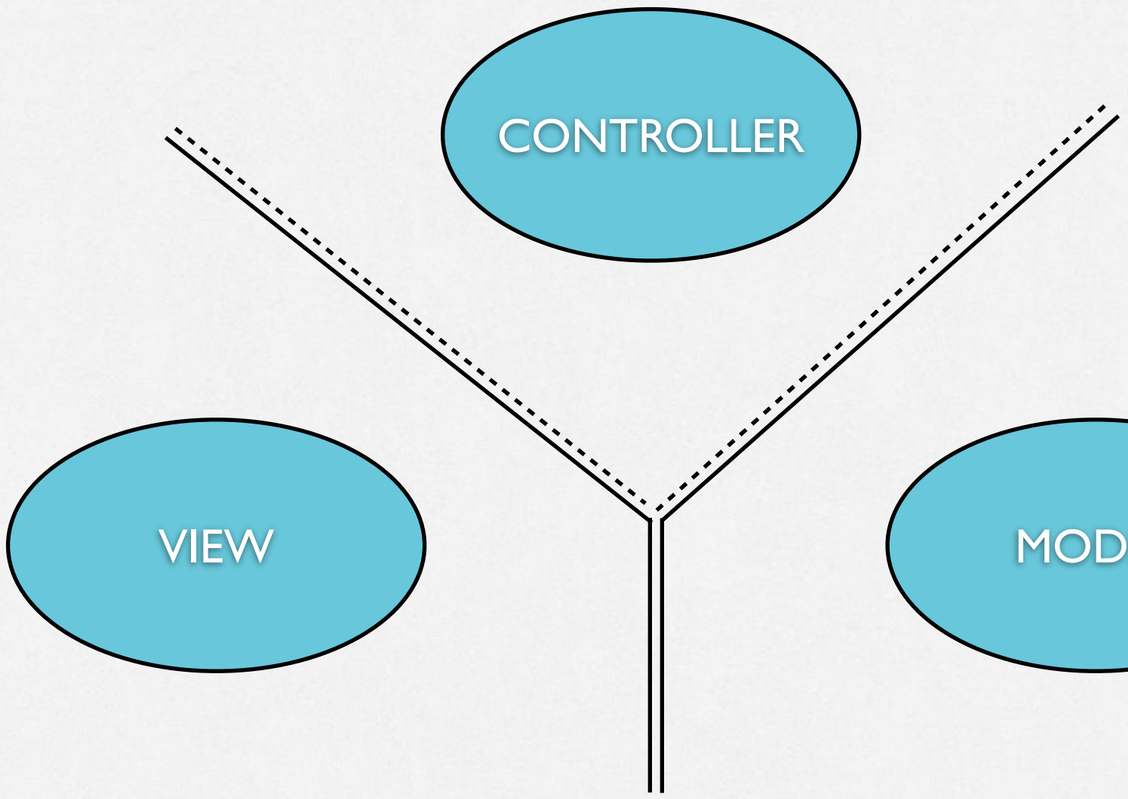
Model, View, & Controller

MODEL VIEW CONTROLLER

MVC stands for Model View Controller, and it is a pattern that allows developers to differentiate code depending on three different roles. It is extremely popular due to its simplicity and it is implemented in different ways in almost every single technology out there -in different ways though. MVC helps you making your code lot more reusable, maintainable and easier to extend.

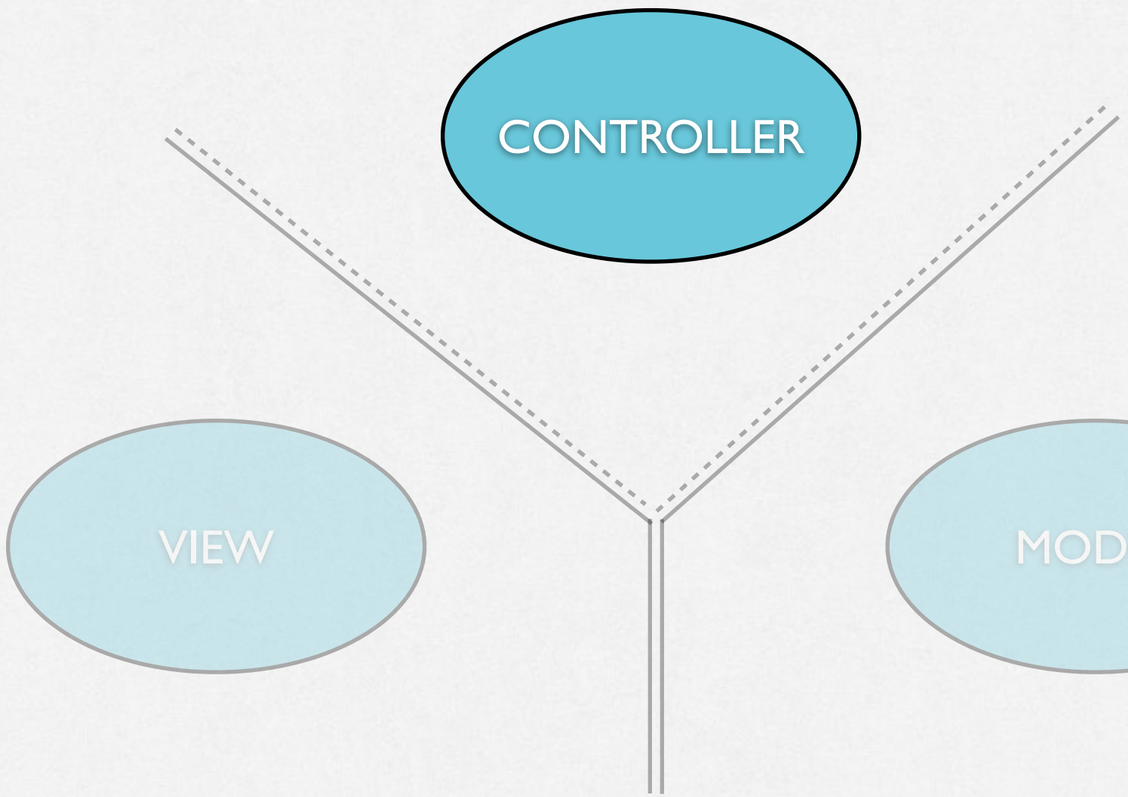
MVC

Divide objects in your program into 3 “camps.”



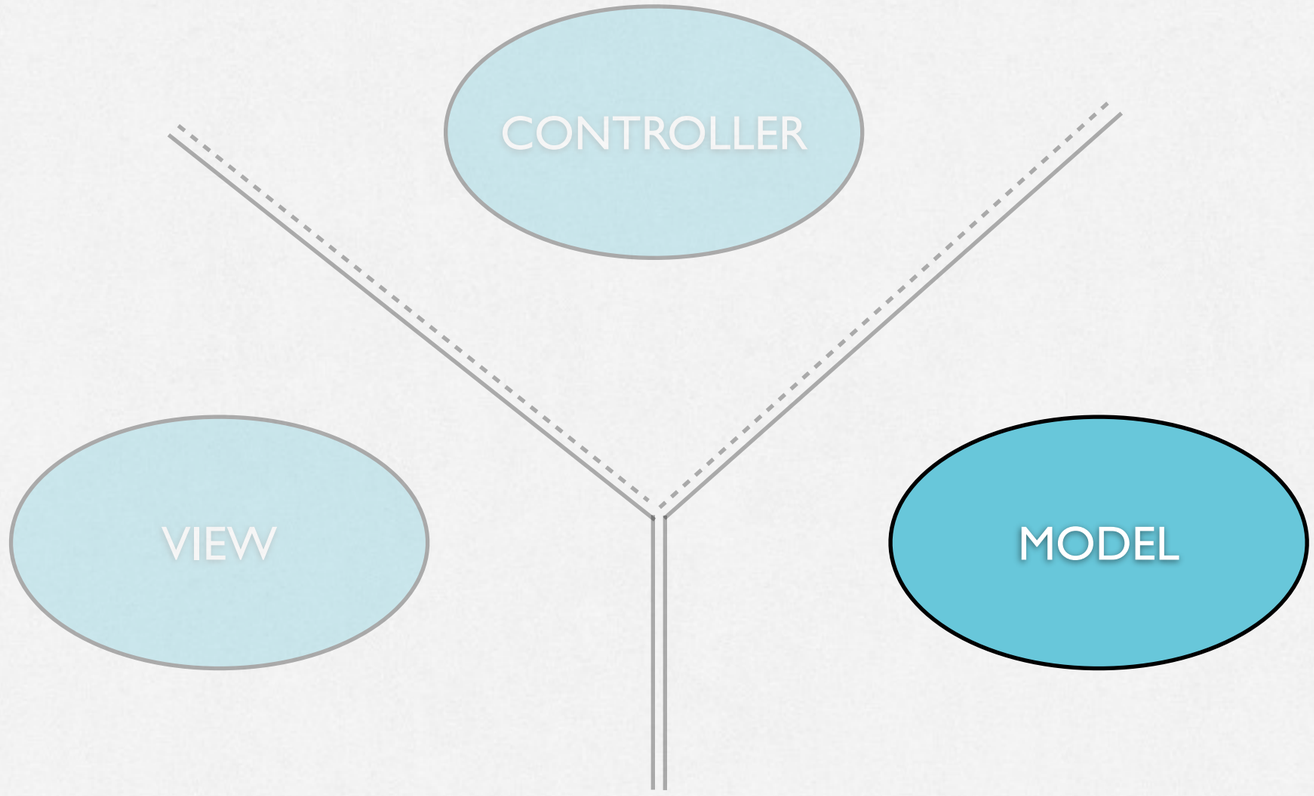
CONTROLLER

How your Model is presented to the user
(UI logic)



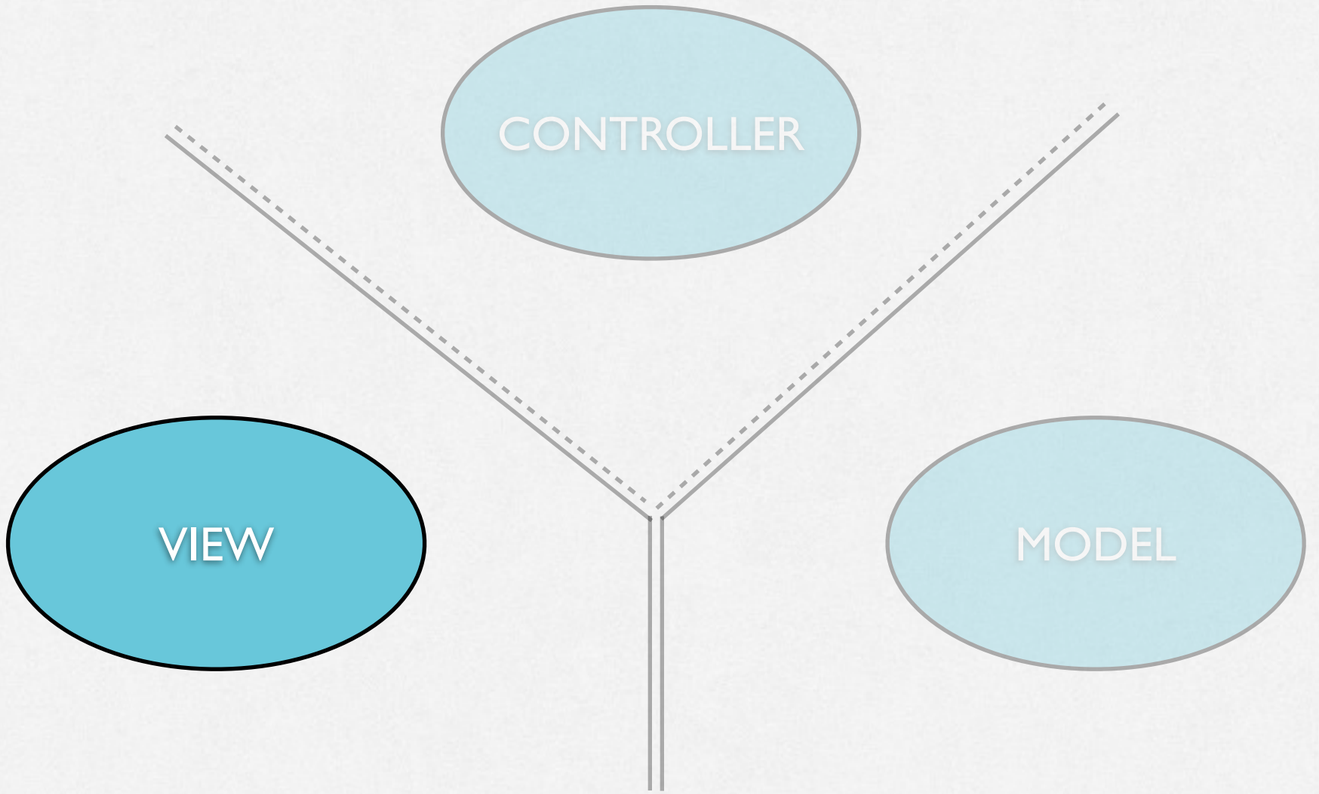
MODEL

What your application is. Data side and it should always be a Singleton.
(but not how it is displayed)



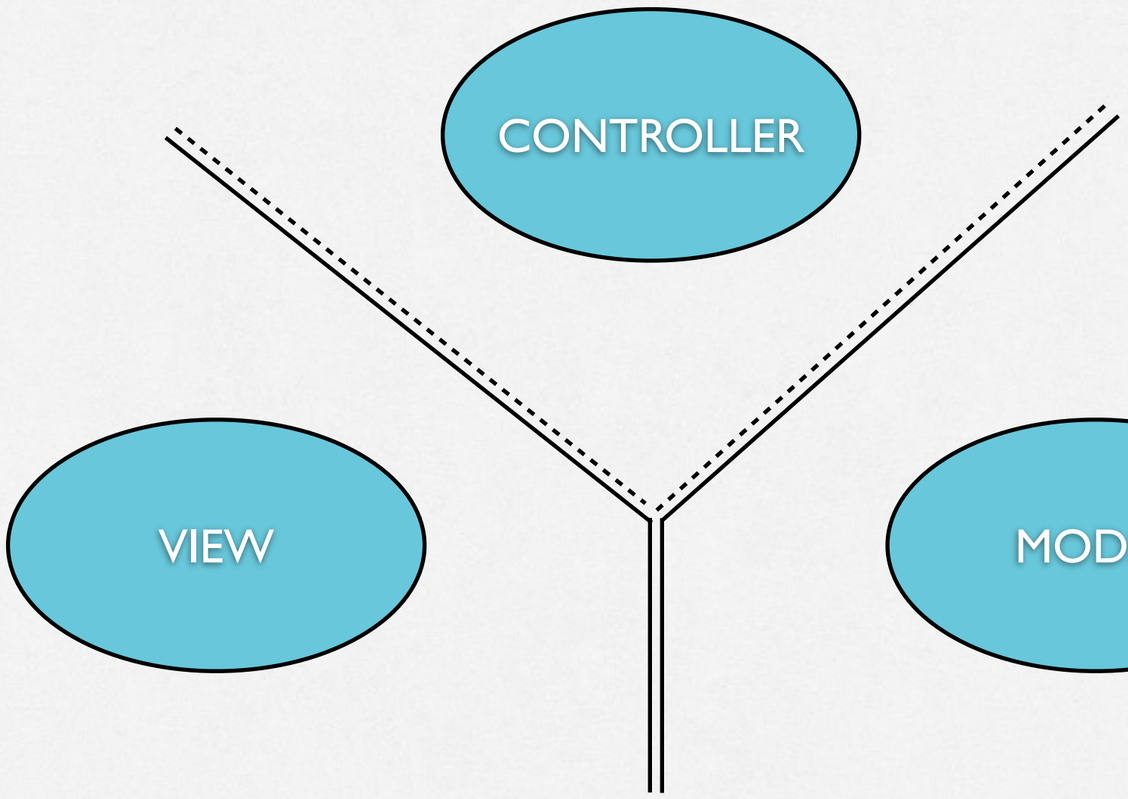
VIEW

Your Controller's minions. What people see and interact with.



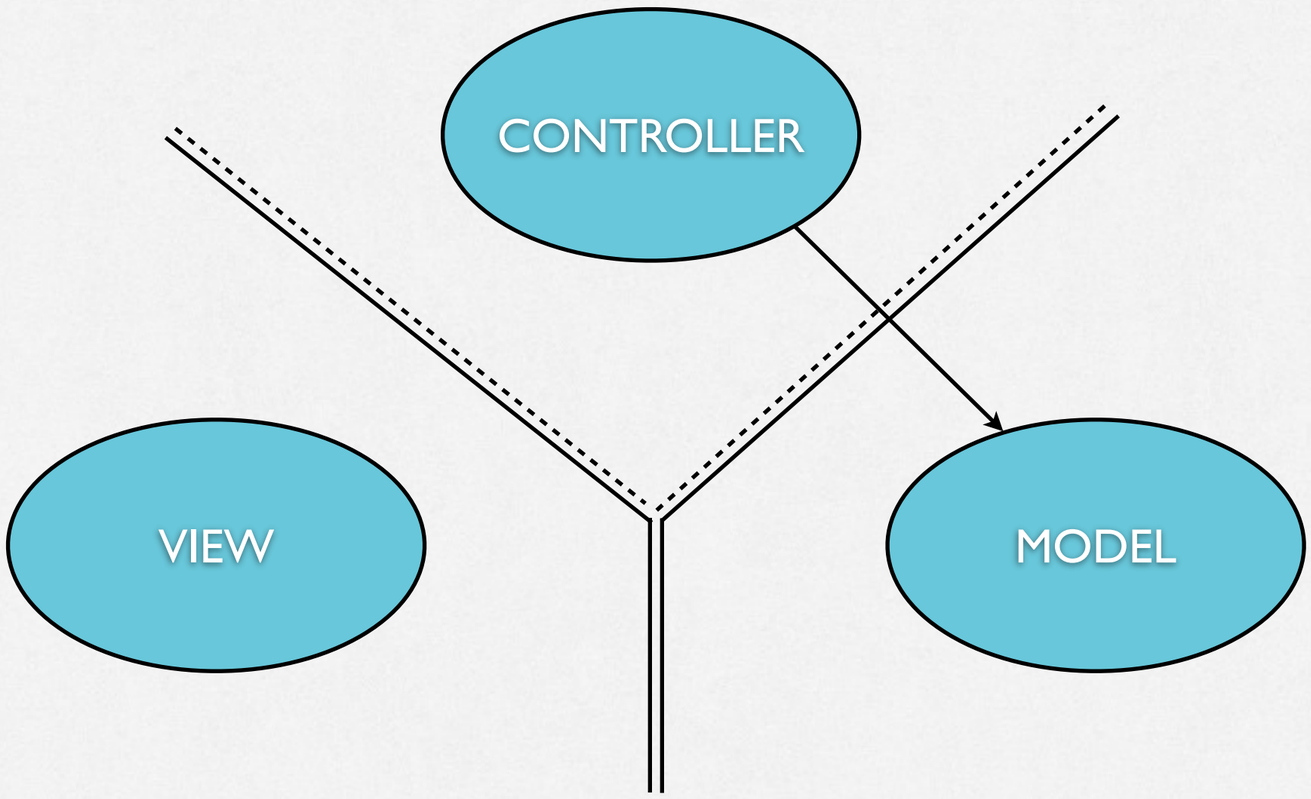
MVC

It's all about managing communication between camps



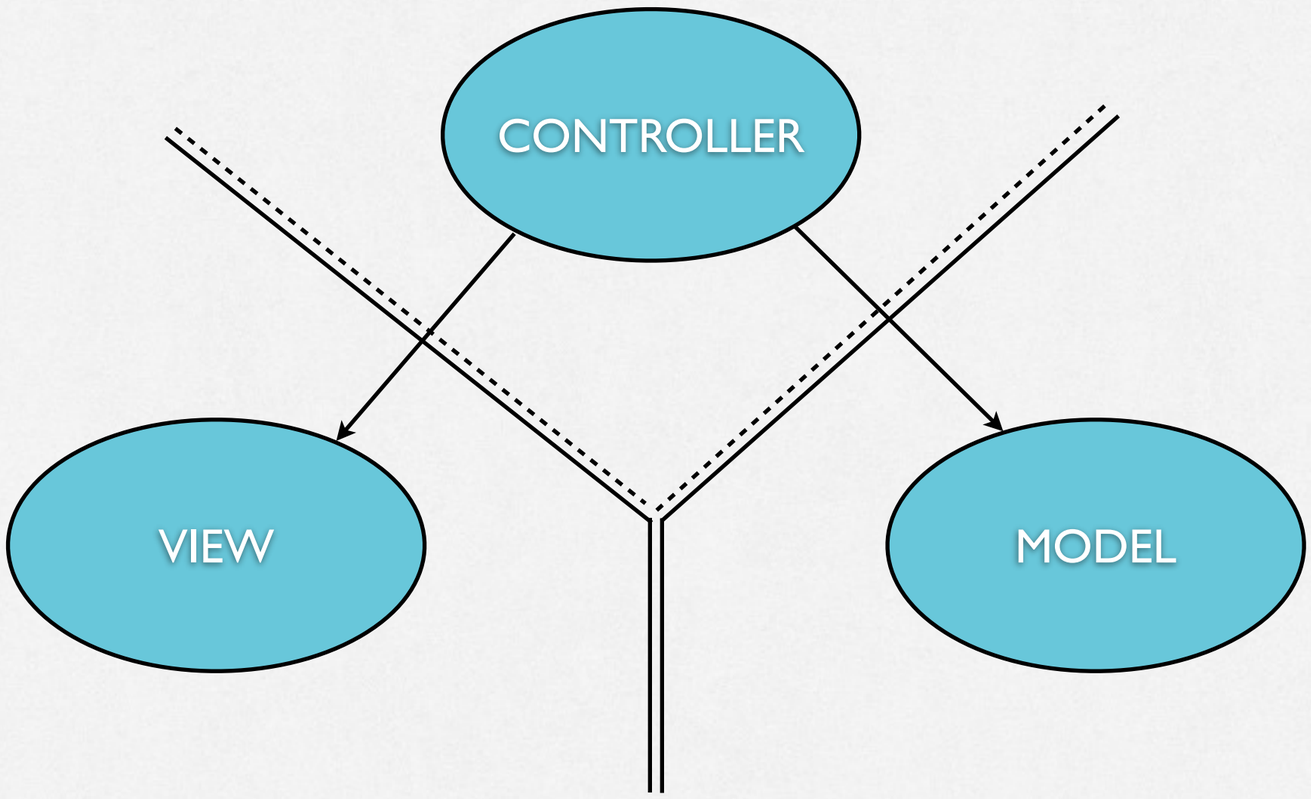
MVC

Controllers can always talk directly to their Model.



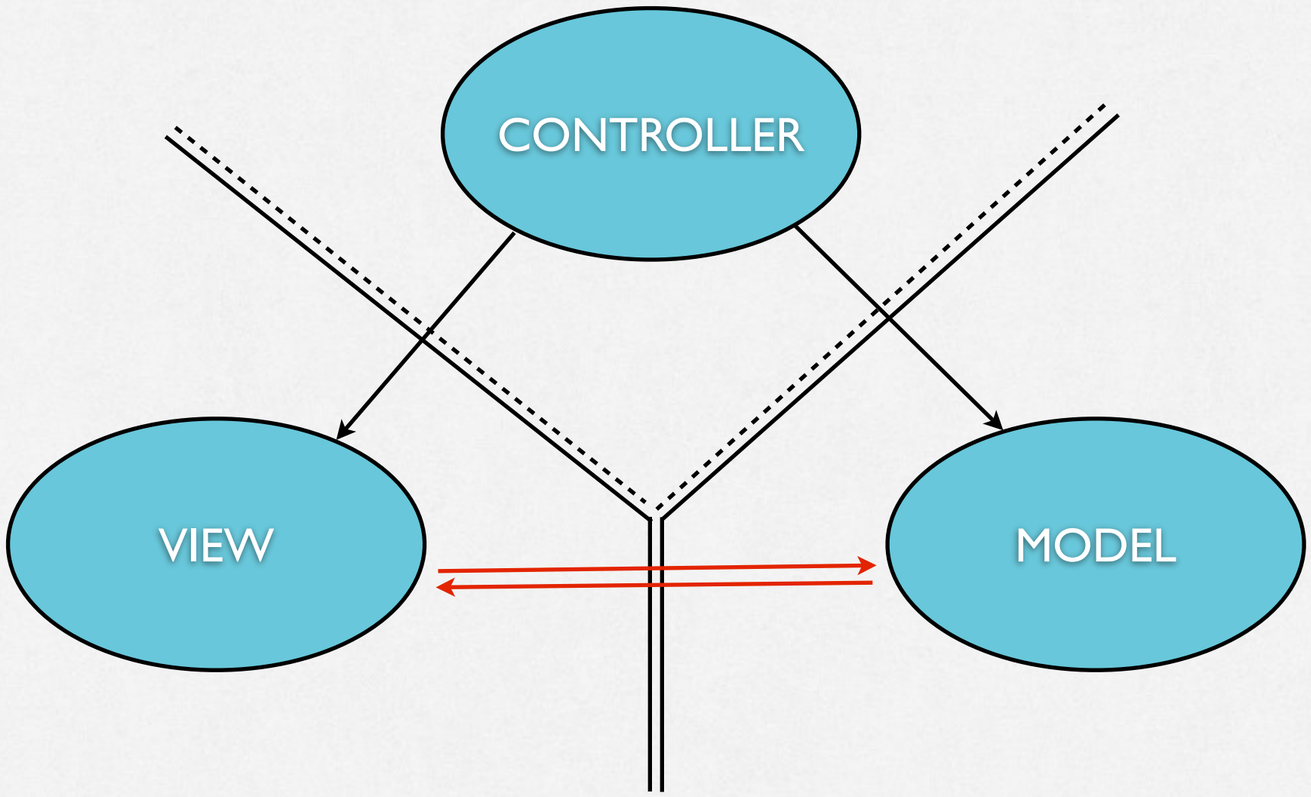
MVC

Controllers can also talk directly to their View.



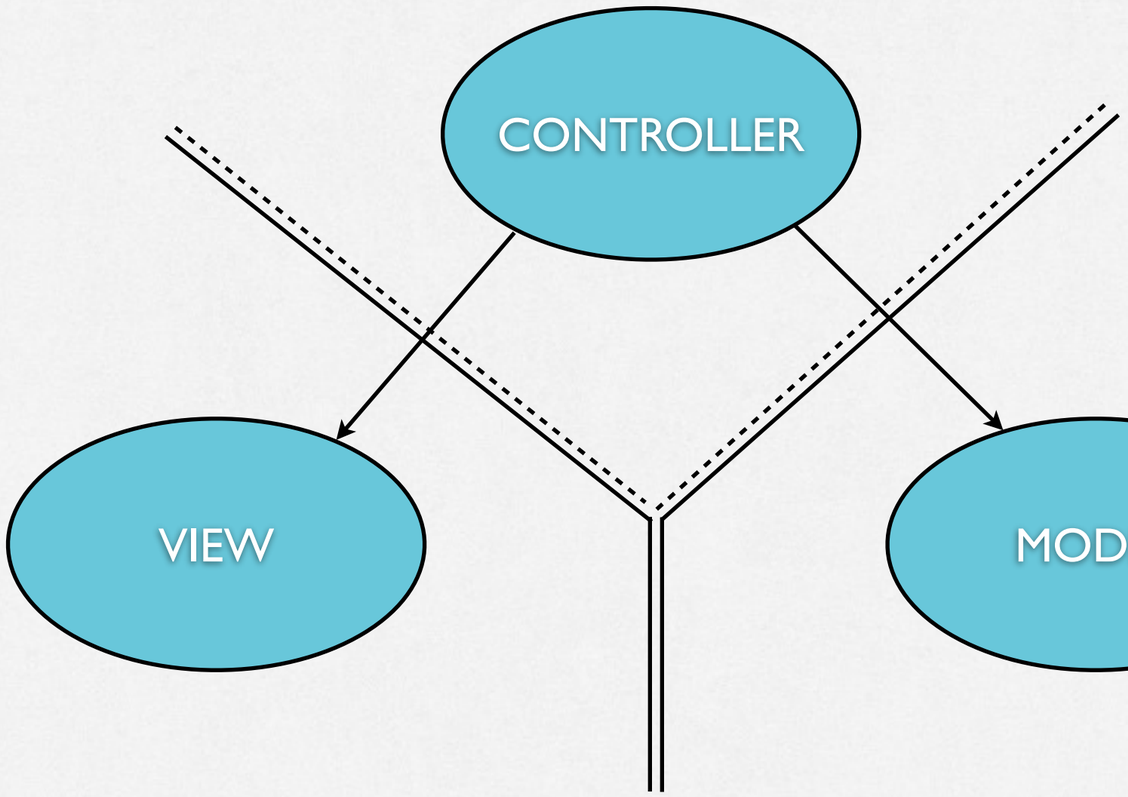
MVC

The Model and View should never speak to each other.



MVC

Controllers interpret/format **Model** information for the **View**.



NEXT WEEK'S AGENDA

1. TAB BAR CONTROLLER
2. SEGMENTED CONTROLLER
3. WEB VIEW
4. PICKERS
5. BUTTONS
6. SWITCH