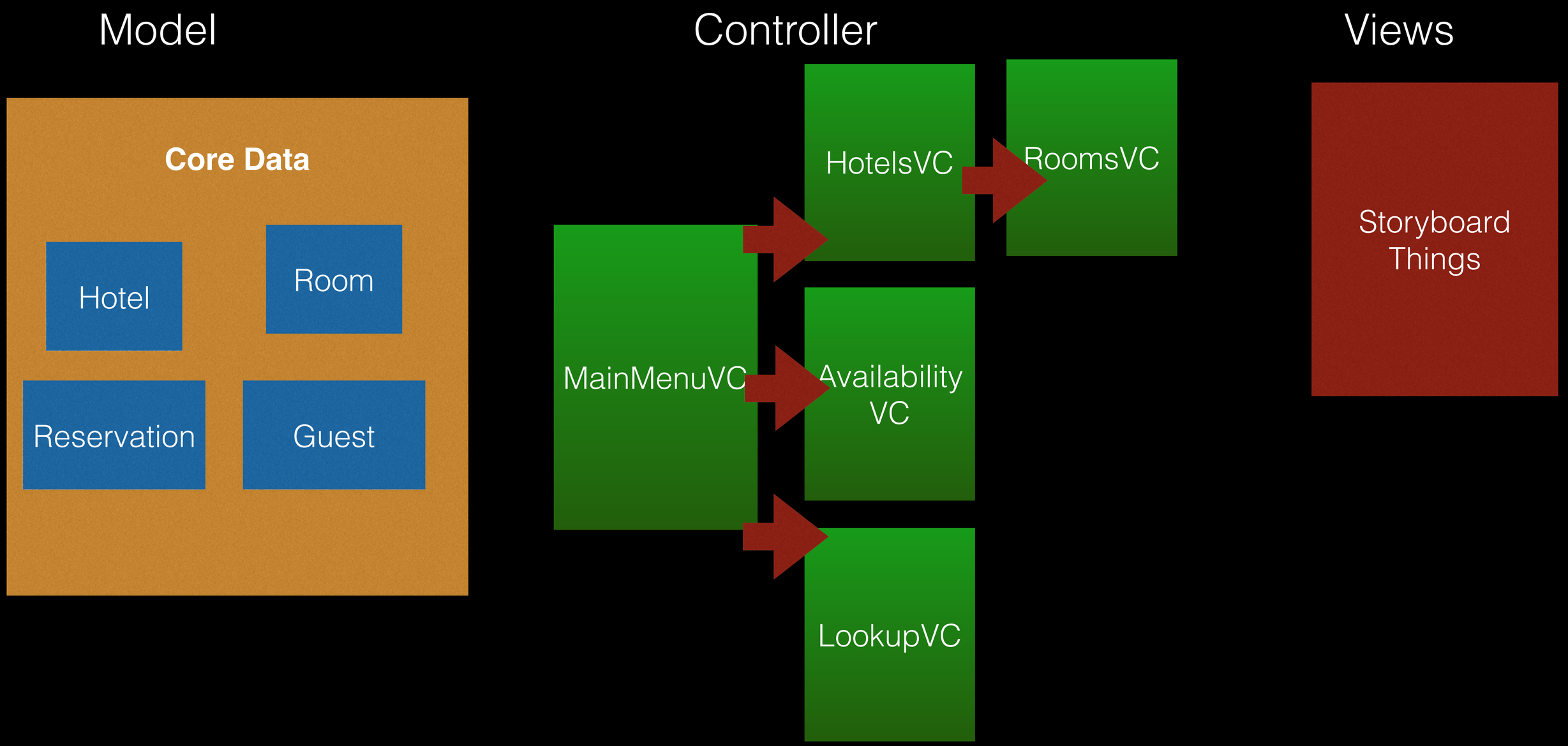
iOS Dev Accelerator
Week 6 Day 1

- Relational Databases
- Intro to Core Data:
  - Core Data Stack
  - MOM
  - NSManagedObjects
  - The Context

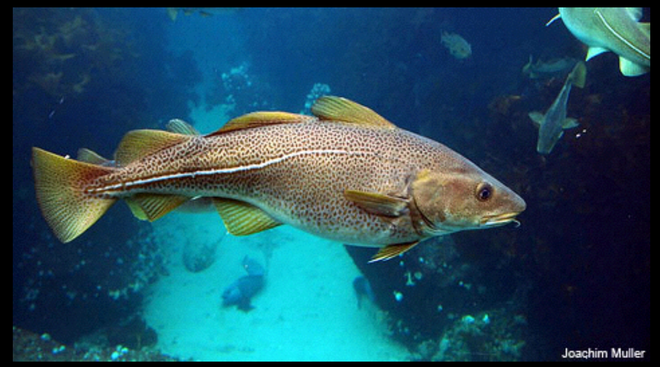
Week 6 App


Model
Controller
Views
Core Data
Hotel
Room
Reservation
Guest
HotelsVC
RoomsVC
MainMenuVC
Availability VC
LookupVC
Storyboard Things

Relational Databases

Database History

- “Databases are important”
- They have been a staple of computing since the dawn of the digital age
- Relational database were invented in 1970 by E.F. Codd



E.F. Codd

Databases

- Originally, all databases were flat. Information was stored in one long text file.
- Each entry was separated by some special character, like a | pipe.
- Heres an example:

Lname, FName, Age, Salary|Smith, John, 35, $280|Doe, Jane, 28, $325|Brown, Scott, 41, $265|Howard, Shemp, 48, $359|Taylor, Tom, 22, $250

- This works, but it made searching through your databases very slow
- Relational Databases to the rescue!

Relational Databases

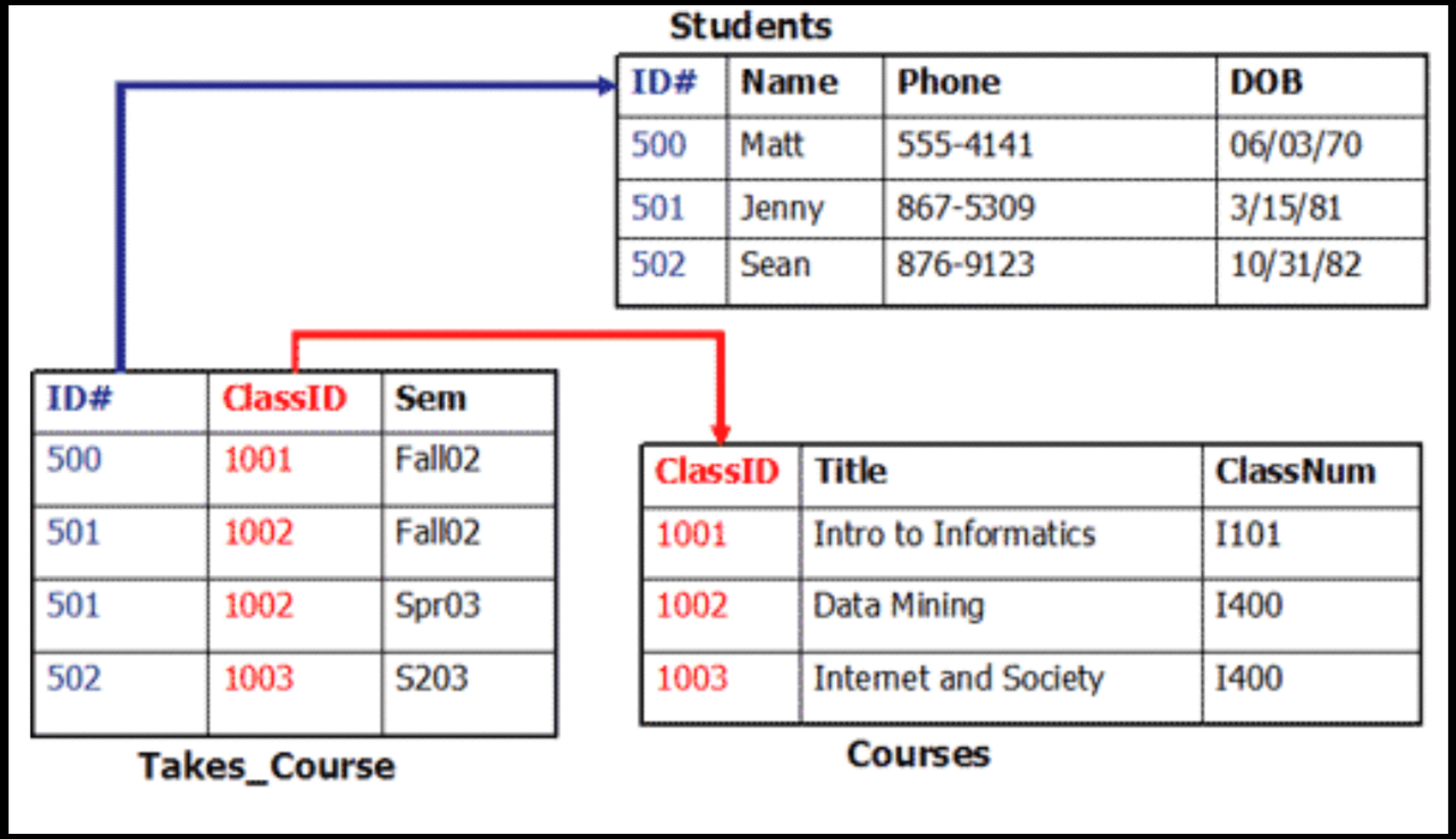
- Relational Databases make finding specific information way faster and easier
- Relational Databases store information in tables
- Each column represents a field (think attribute or property)
- Each row represents a record (an actual instance of data)
- Table Example:

Students

ID#	Name	Phone	DOB
500	Matt	555-4141	06/03/70
501	Jenny	867-5309	3/15/81
502	Sean	876-9123	10/31/82

Relational Databases

- So here is the power of relational databases:



Students

ID#	Name	Phone	DOB
500	Matt	555-4141	06/03/70
501	Jenny	867-5309	3/15/81
502	Sean	876-9123	10/31/82

Takes_Course

ID#	ClassID	Sem
500	1001	Fall02
501	1002	Fall02
501	1002	Spr03
502	1003	S203

Courses

ClassID	Title	ClassNum
1001	Intro to Informatics	I101
1002	Data Mining	I400
1003	Internet and Society	I400

CoreData

CoreData

- Core Data is a framework designed to generalize and automate common tasks associated with object life-cycle and persistence.
- Technically, Core Data is not a database, its an object graph management and persistence framework that allows you to easily work with a database.
- Core Data isn't just about loading your data from a database, its also about working with that data in memory.
- **Core Data will manage all or most of the model layer of your MVC components.**

Why use CoreData?

Saving to disk using NSKeyedArchiver

```
- (NSString *) pathForDataFile
{
  NSFileManager *fileManager = [NSFileManager defaultManager];

  NSString *folder = @"~/Library/Application Support/MailDemo/";
  folder = [folder stringByExpandingTildeInPath];

  if ([fileManager fileExistsAtPath: folder] == NO)
  {
    [fileManager createDirectoryAtPath: folder attributes: nil];
  }

  NSString *fileName = @"MailDemo.cdcmaildemo";
  return [folder stringByAppendingPathComponent: fileName];
}
```

and

```
- (void) saveDataToDisk
{
  NSString * path = [self pathForDataFile];

  NSMutableDictionary * rootObject;
  rootObject = [NSMutableDictionary dictionary];

  [rootObject setValue: [self mailboxes] forKey:@"mailboxes"];
  [NSKeyedArchiver archiveRootObject: rootObject toFile: path];
}
```

Saving to disk using CoreData

```
NSError *saveError;
[_managedObjectContext save:&saveError];
```

and

Why use CoreData?

Loading from disk using NSKeyedArchiver

```
- (NSString *) pathForDataFile
{
  NSFileManager *fileManager = [NSFileManager defaultManager];

  NSString *folder = @"~/Library/Application Support/MailDemo/";
  folder = [folder stringByExpandingTildeInPath];

  if ([fileManager fileExistsAtPath: folder] == NO)
  {
    [fileManager createDirectoryAtPath: folder attributes: nil];
  }

  NSString *fileName = @"MailDemo.cdcmaildemo";
  return [folder stringByAppendingPathComponent: fileName];
}
```

and

```
- (void) loadDataFromDisk
{
  NSString     * path        = [self pathForDataFile];
  NSDictionary * rootObject;

  rootObject = [NSKeyedUnarchiver unarchiveObjectWithFile:path];
  [self setMailboxes: [rootObject valueForKey:@"mailboxes"]];
}
```

Loading from disk using CoreData

```
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc]
  initWithEntityName:@"Hotel"];
self.results = [self.context executeFetchRequest:
  fetchRequest error:nil];
```

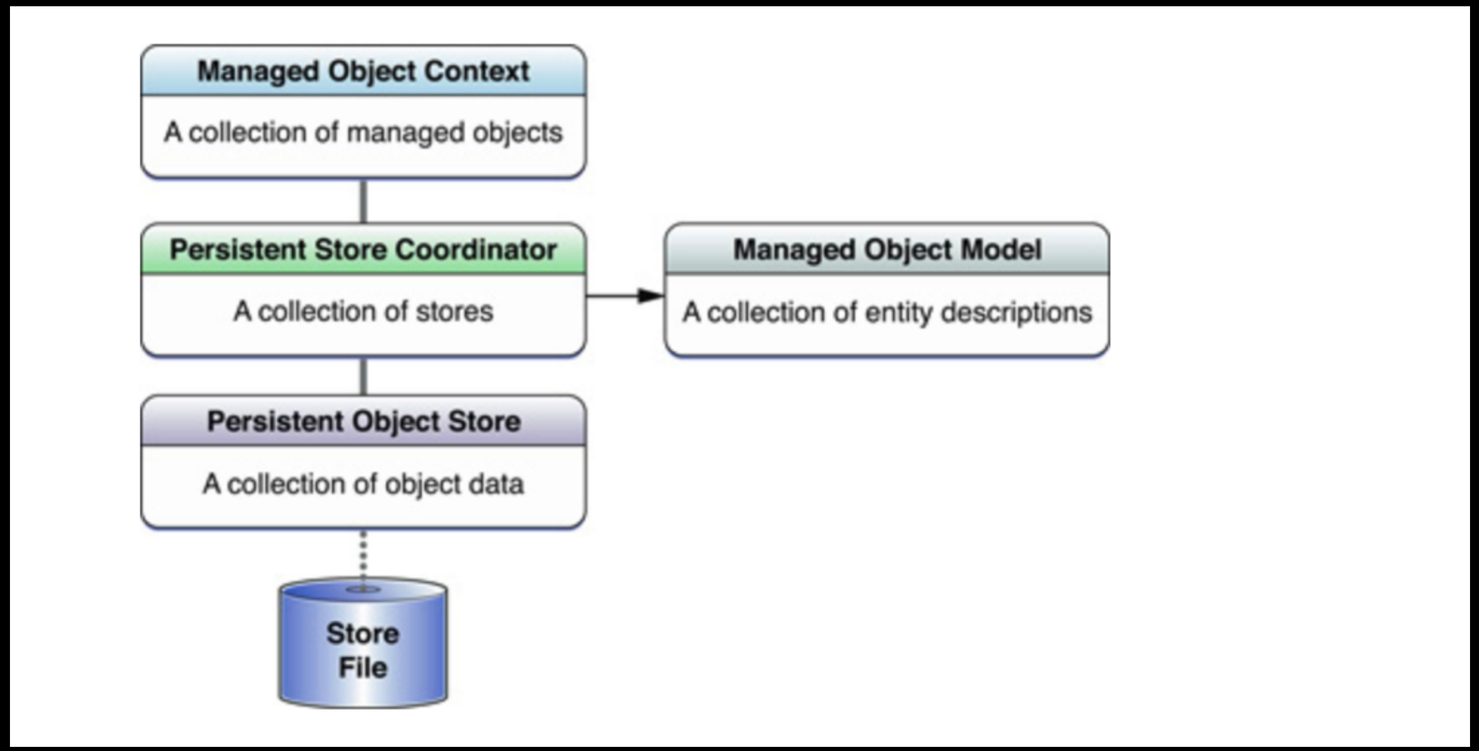
We can also add predicates to our fetchRequests, which allows us to only fetch certain entities that meet the criteria we specify. We cannot do this with NSKeyedArchiver, which will always unarchive everything in the file. What if we had thousands of data points? Core Data allows us to easily ask for only what we need

CoreData and You

- Apple claims the amount of code you write in your model layer is reduced by 50-70% with Core Data
- This is simply because the features Core Data provides are features you don’t have to implement yourself
- Most people resist Core Data at first, including myself
- Now whenever I start an app I intend to release on the app store, I will always start with Core Data managing my model layer.
- I would wager 80% of people are scared off by Core Data because of the topic we are going to discuss next..

CoreData Stack

- A Core Data stack contains everything you need to fetch, create, and manipulate managed objects:

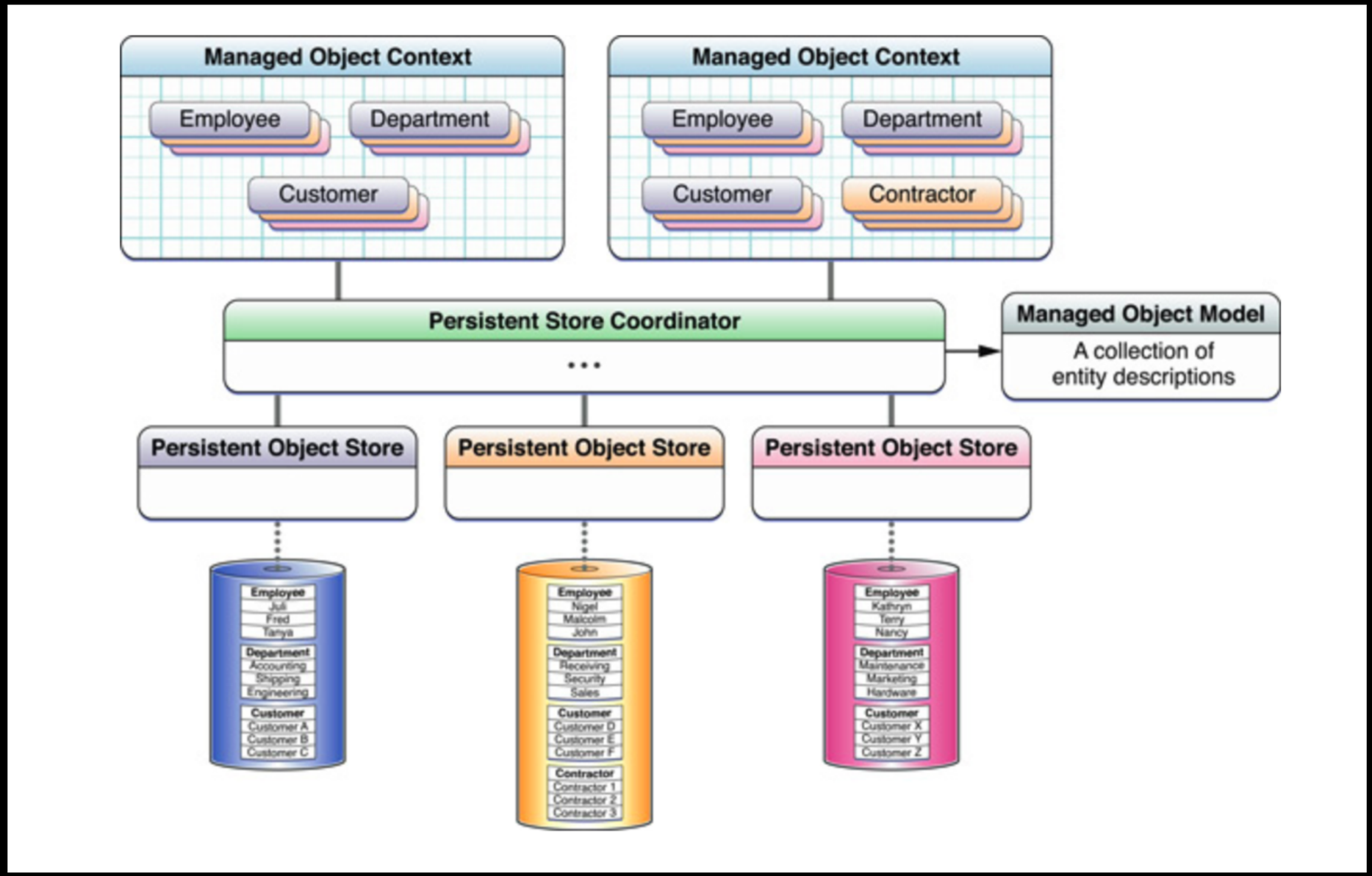


It will seem complicated at first!

NSPersistentStoreCoordinator

- Persists objects to disk, reads objects from disk
- The PSC associates persistent store objects and a managed object model, and presents a facade to managed object contexts (more on these things later)
- It lumps all the store objects together, so to the developer it appears as a single store
- Most apps only need one, but a super complex app may have several
- Sits between `managedObjectContext` & persistent store (file on disk)
- Has a reference to the managedObjectModel (aka MOM)
- Can automatically migrate your existing database to a new schema (Sweet!)

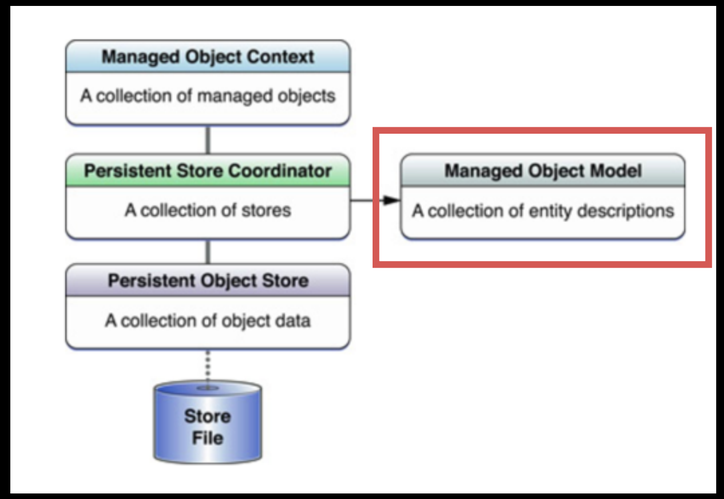
NSPersistentStoreCoordinator



Demo

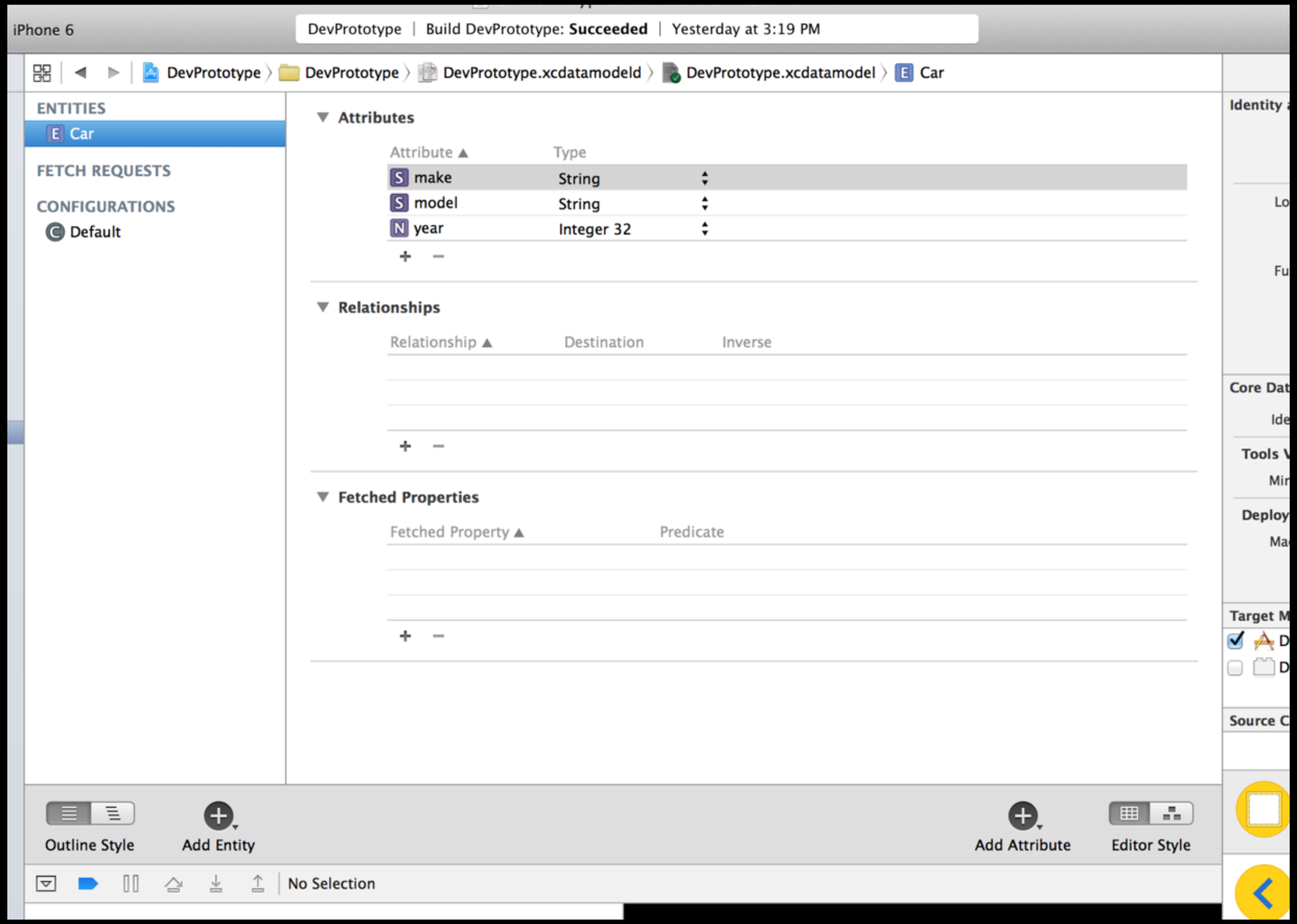
Managed Object Model

- A managed object model is a set of objects that together form a blueprint describing the managed objects you use in your application.
- A managed object model, or MOM, allows core data to map records from a persistent store to managed objects that you use in your app.
- Describes a Core Data database schema:
  - Entities (objects)
  - Attributes (object properties)
  - Relationships (has_many, belongs_to, etc.)
  - Validation (e.g. regex for email address)
  - Storage rules (e.g. separate file for binary data)



- It is a collection of entity description objects. Think of entities as a table in a relational database.
- You should take special considerations when updating an app's schema
- Xcode provides an awesome GUI to create this without writing any code! Hooray

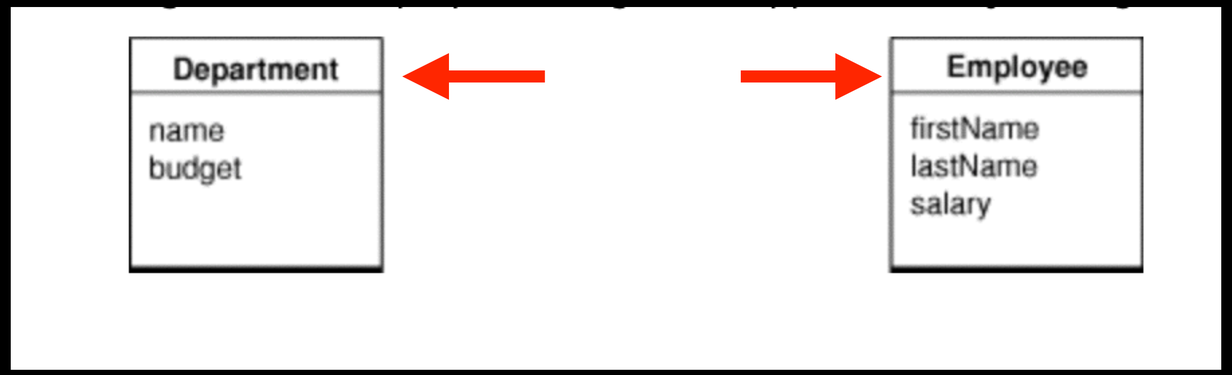
Managed Object Model



Demo

Entities

- “The Entity-relationship modeling is a way of representing objects typically used to describe a data source’s data structures in a way that allows those data structures to be mapped to objects in an object-oriented system” Not unique to Cocoa.
- The objects that hold data are called entities.
- Entities can use inheritance just like regular classes.
- The components of an entity are called attributes (basically properties), and references to other entity objects are called relationships.



Department and Employee are Entities

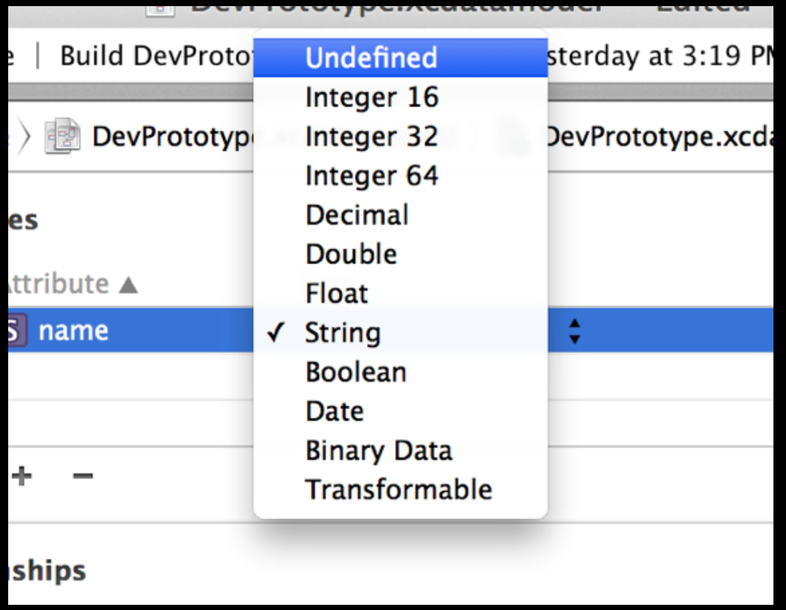
Attributes

- Attributes represent the containment of data.
- An attribute can be a simple value, like a scalar (int, float ,double)
- Or a C struct (array of chars, or an NSPoint)
- Or an instance of a primitive class (NSNumber or NSData)
- Core data is specific about what types of data it supports, but there are techniques for storing non standard values as well.

Think of attributes as properties for your entities

Attributes cont.

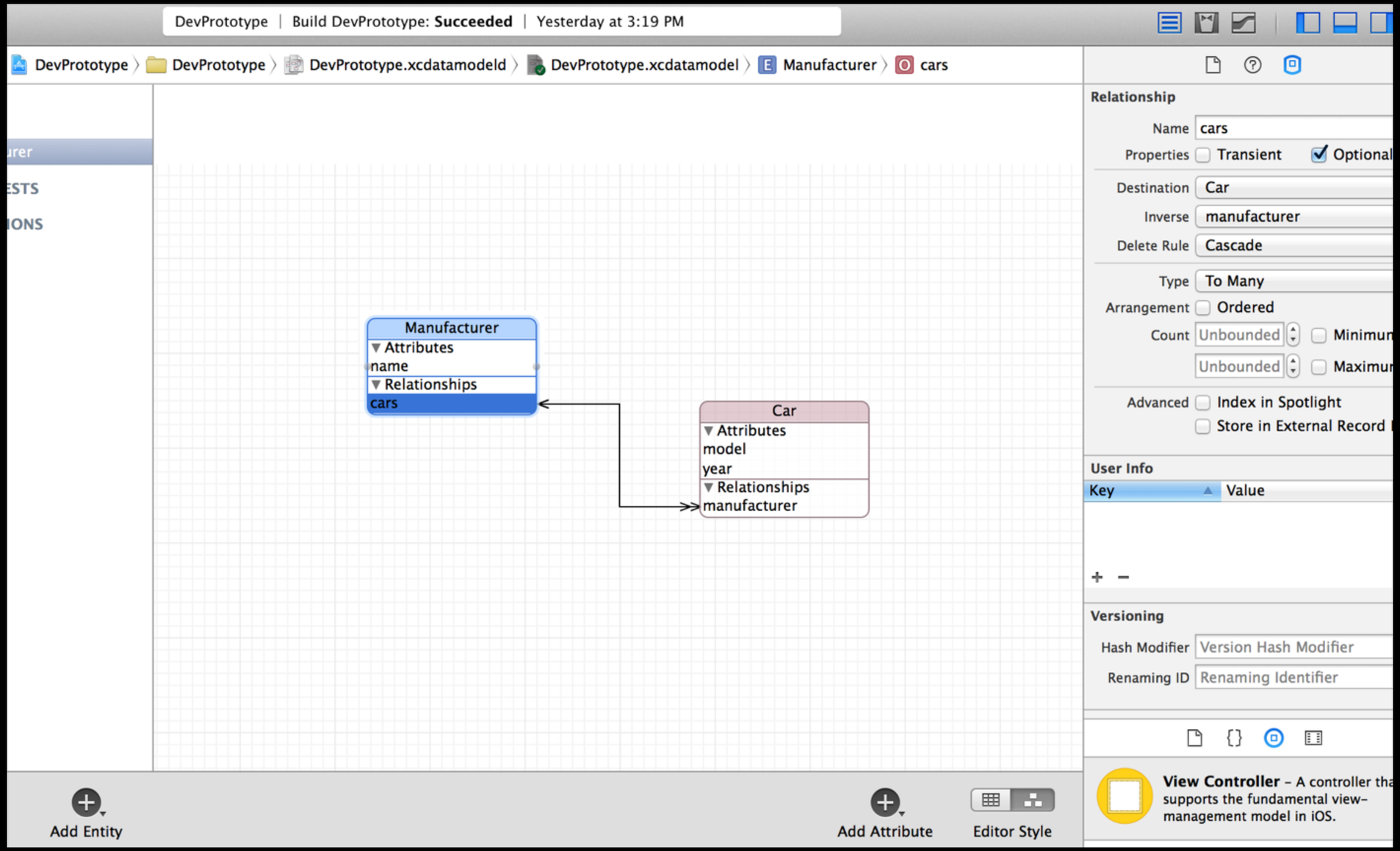
- For integers, you will see 3 types based on how many bits it uses:
  - int16 : -32768 to 32767
  - int32 : -2147483648 to 2147483648
  - int64: -9223372036854775808 to 9223372036854775808
- Double and Float for numbers with decimals
- Decimal for currency
- Binary Data for for saving instances of NSData directly
- Transformable for saving instances of classes that can be converted to and from NSData. Anything NSCoding compliant.



Relationships

- Not all properties of an entity are attributes, some are relationships to other entities.
- These relationships are inherently bidirectional, but you can set them to be navigable in only one direction, with no inverse.
- The cardinality of relationship tells you how many objects can potentially resolve the relationship. If the destination object is a single entity, its considered a to-one relationship.
- If there may be more than one object, then its a called a to-many relationship.
- Relationships can be optional or mandatory.
- The values of a to one relationship is just the related object, the value of a to-many in CoreData is an NSSet collection of all related objects.

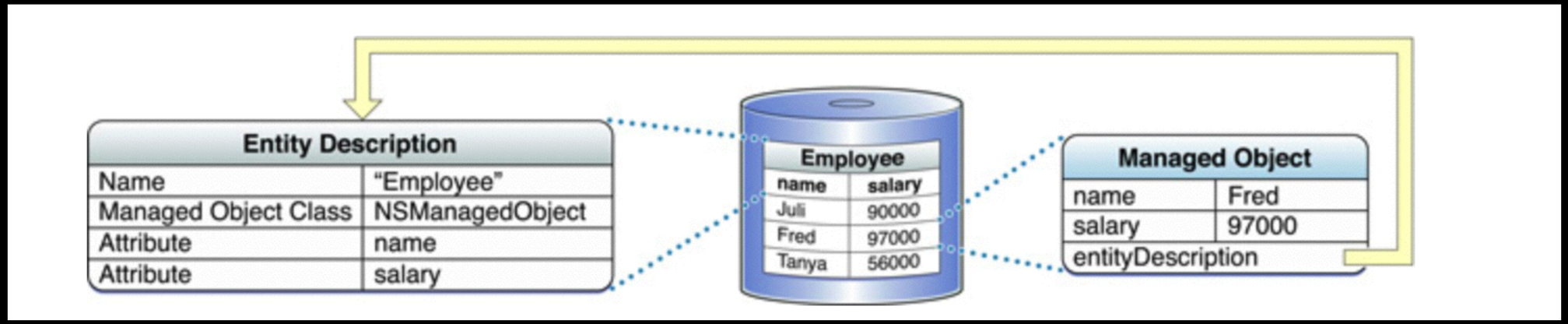
Relationships



Demo

Managed Object

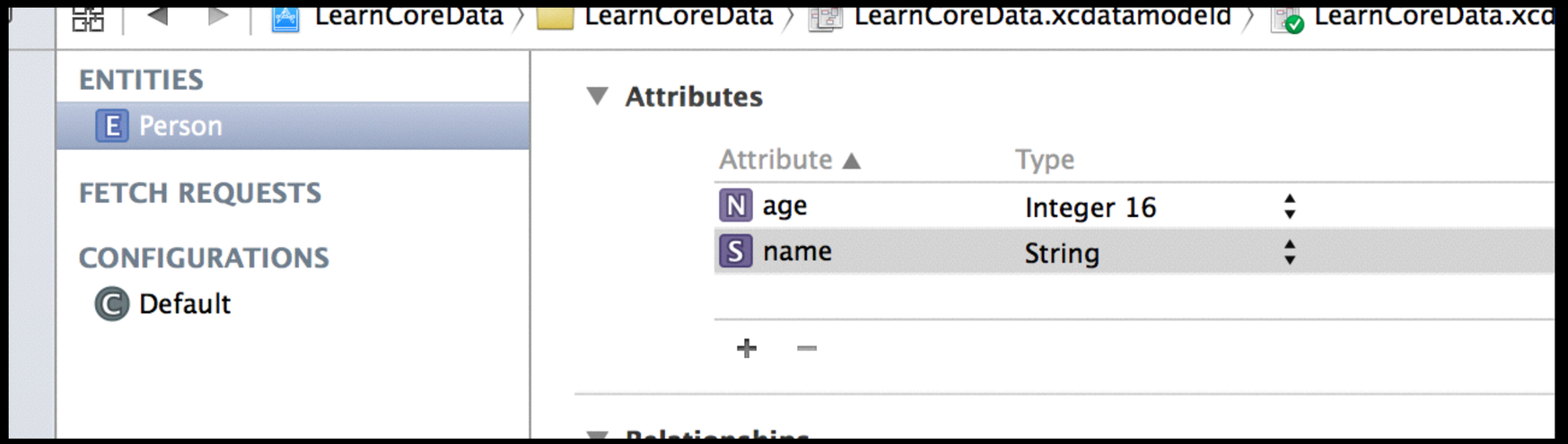
- In Core Data, a managed object is an instance of an entity. It represents a record from a persistent store.
- It takes the place of your regular model objects.
- It is an instance of NSManagedObject or a subclass of it.
- Every managed object is registered with a context (more on the context later).
- In any given context, there is at most one instance of a managed object that corresponds to a given record.
- A managed object has a reference to an entity description object that tells it what entity it represents from the MOM.



NSManagedObject

- 2 ways to create a managed object:

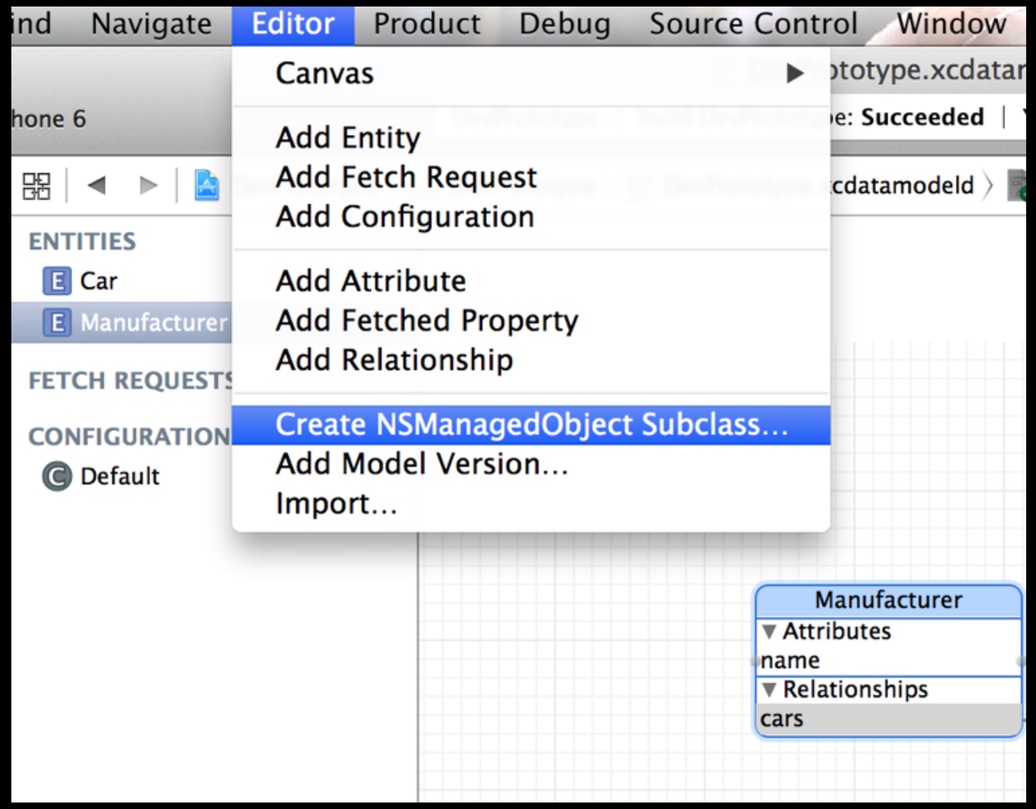
```
//shortway
NSManagedObject *person = [NSEntityDescription insertNewObjectForEntityForName:@"Person" inManagedObjectContext:self.context];
[person setValue:@"Brad" forKey:@"name"];
[person setValue:@26 forKey:@"age"];

//longway
NSEntityDescription *entity = [NSEntityDescription entityForName:@"Person" inManagedObjectContext:self.context];

NSManagedObject *anotherPerson = [[NSManagedObject alloc] initWithEntity:entity insertIntoManagedObjectContext:self.context];
[anotherPerson setValue:@"Russell" forKey:@"name"];
[anotherPerson setValue:@26 forKey:@"age"];
```



Demo

Custom Managed Object Classes

- Xcode can generate custom subclasses NSManagedObject that are tailored to all of your entities.
- NSManagedObject provides a rich set of default behaviors.
- This is the preferred way to create and interact with instances of your entities.



Custom Managed Object Classes

- The main advantage you get with your custom sub classes are you don't have to call setValue:forKey: anymore.
- Instead you get access to the attributes listed in the MOM file as properties on the objects.
- This is way better, since it lets us avoid “stringly typed” attribute setting with setValue:ForKey: where mistyping the name of an attribute will lead to an exception.

Custom Managed Object Classes

```objc
//crap way
NSEntityDescription *entity = [NSEntityDescription entityForName:@"Person" inManagedObjectContext:self.context];

NSManagedObject *anotherPerson = [[NSManagedObject alloc] initWithEntity:entity insertIntoManagedObjectContext:
    self.context];
[anotherPerson setValue:@"Russell" forKey:@"name"];
[anotherPerson setValue:@26 forKey:@"age"];

//way better
Person *myPerson = [NSEntityDescription insertNewObjectForEntityForName:@"Person" inManagedObjectContext:self.
    context];
myPerson.name = @"Pete";
myPerson.age = @65;
```

Ah Crap

Yay

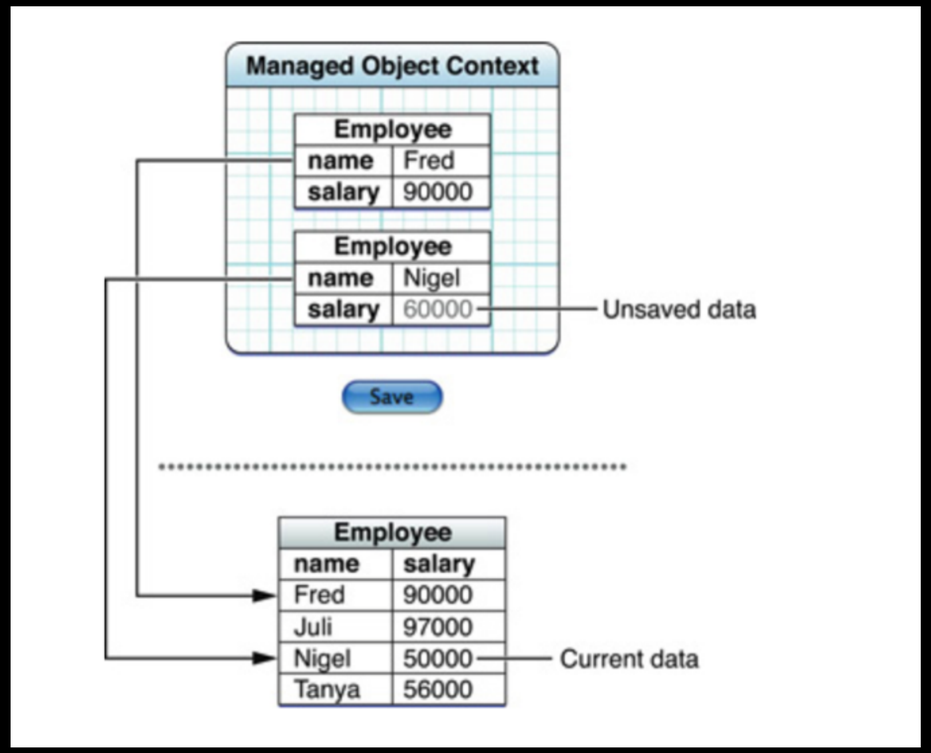
Custom Managed Object Classes Life-cycle

- Core Data “owns” the life-cycle of managed objects. objects can be created, destroyed, and resurrected by the framework at any time.
- There different methods you can override to customize initialization of your managedObjects:
  - awakeFromInsert: - invoked only once in the lifetime of an object, when it is first created.
  - initWithEntity:insertIntoManagedObjectContext: - generally discouraged as state changes made in this method may not properly integrate with undo and redo
  - awakeFromFetch: - is invoked when an object is reinitialized from a persistent store during a fetch.

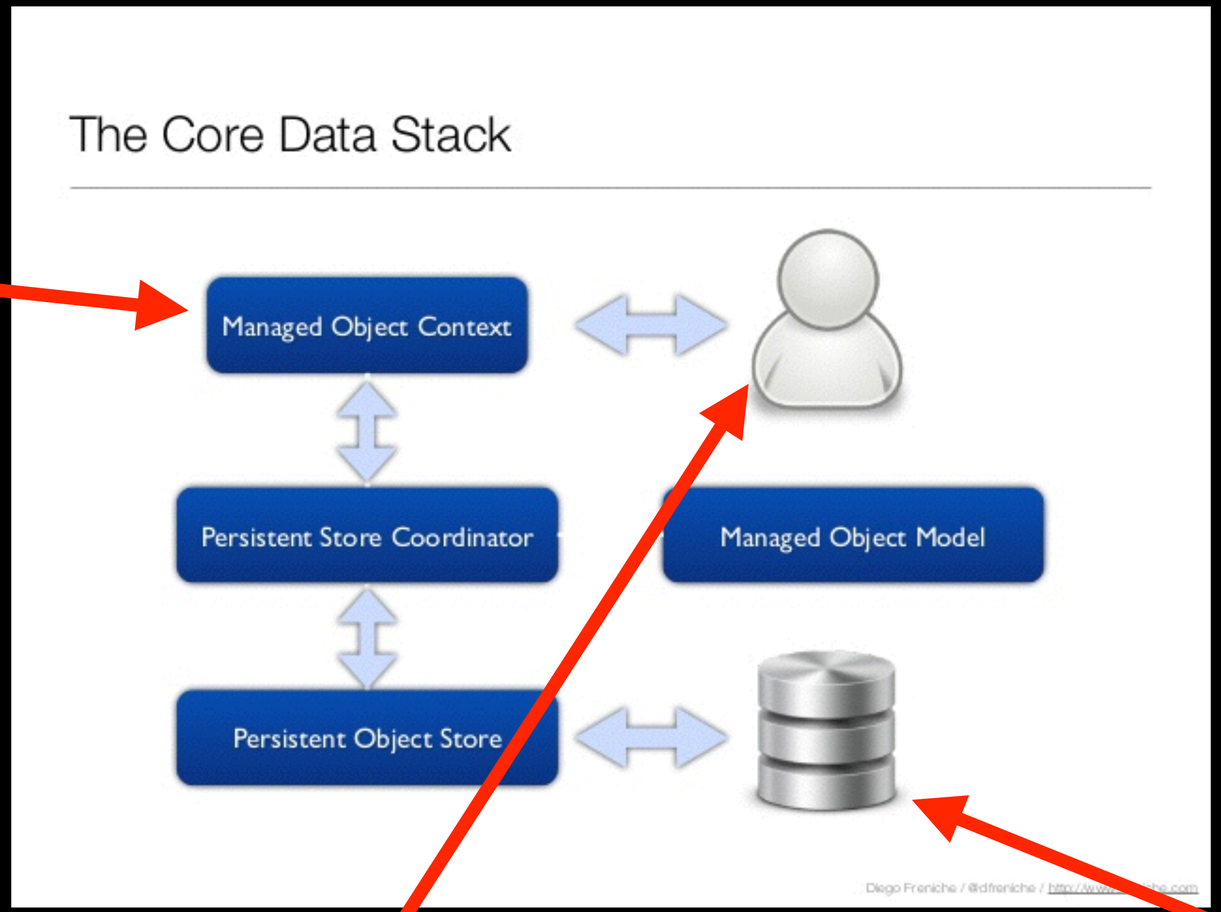
Demo

NSManagedObjectContext

- **The link between your code and the database**
- Represents a single object space, or “scratch pad”, in a Core Data application.
- It’s primary responsibility is to manage a collection of managed objects.
- These objects represent an ‘internally consistent’ view of the persistent store(s).
- **To the developer, the context is the central object in the Core Data stack.**
- It is connected to the persistent store coordinator
- Every managed object knows which context it belongs to, and every context knows which objects its managing.

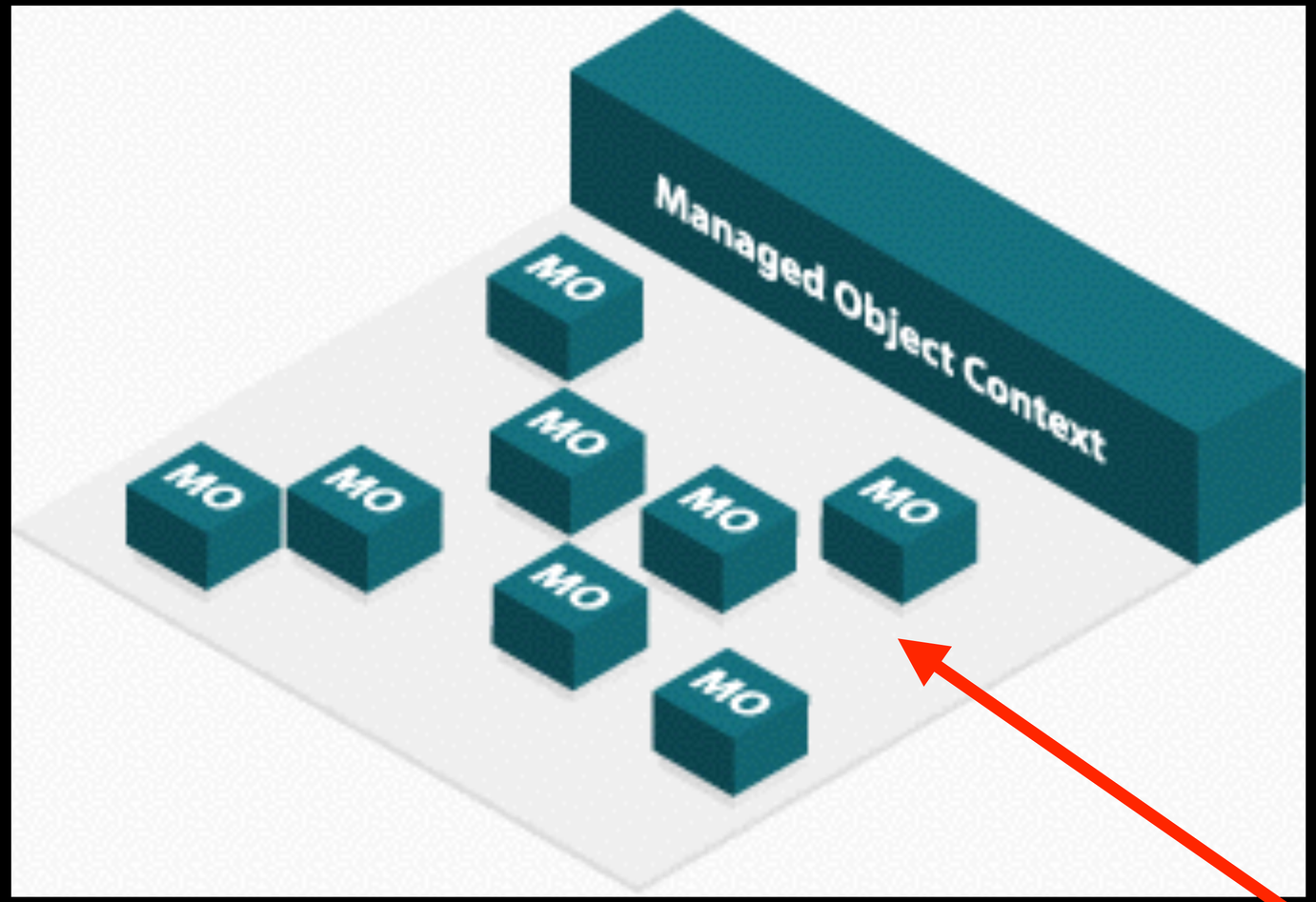


NSManagedObjectContext pictures to help you understand



It is how you, the developer, interfaces with the actual data on disk!

NSManagedObjectContext pictures to help you understand



It is like a scratch pad that holds all your managed objects you have created in memory

NSManagedObjectContext pictures to help you understand

It’s super important!

Saving with CoreData workflow

1. Get a reference to your context
2. Insert a new entity into the context
3. use the returned managed object from the method call in step 2, and set its attributes
4. Repeat step 2 and 3 as many times as you need to create and configure as many objects as you need
5. Call save on the context, passing in an Error pointer
6. Check if the Error pointer isn't nil, and if its not inspect the error description

Demo

Fetching

- So we know how to save to core data, but how do we retrieve those items we have saved? How do we perform a Query?
- We can do this with fetching
- Fetching allows us to query our data store for all objects of a specific entity, or a subset which meets certain criteria.
- We perform fetches with the class NSFetchRequest
- We will learn more about customizing our fetch requests later this week

Fetching

- 4 primary things you set on your fetches:
  - Tell it which entity you are fetching on (can only set one, and this is required)
  - An optional fetch count (default set to fetch all objects)
  - An optional sort descriptor to tell core data how to order the results
  - An optional NSPredicate to specify exactly which objects you want from the entity

Simple Fetch

Swift

```swift
func fetchFilters() {
    var fetch = NSFetchRequest(entityName: "Filter")
    let fetchResults = self.managedObjectContext?.executeFetchRequest(fetch, error: nil)
    if let filters = fetchResults as [Filter]? {
       self.filters = filters
    }
}
```

Objective-C

```objc
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc]
  initWithEntityName:@"Hotel"];
self.results = [self.context executeFetchRequest:
  fetchRequest error:nil];
```

Demo

Faulting

- “Faulting is a mechanism CoreData employs to reduce your applications memory usage”
- A fault is a placeholder object that represents a managed object that has not yet been fully realized, or a collection object that represents a relationship.
  - A managed object fault is an instance of the appropriate class, but its persistent variables are not yet initialized.
  - A relationship fault is a subclass of the collection class that represents the relationship.
- Fault handling is transparent, the fault is realized only when that variable or relationship is accessed.
- You can turn realized objects back into faults by calling refreshObjects:mergeChanges: method on the context.