iOS Dev accelerator Week 5 Day 2

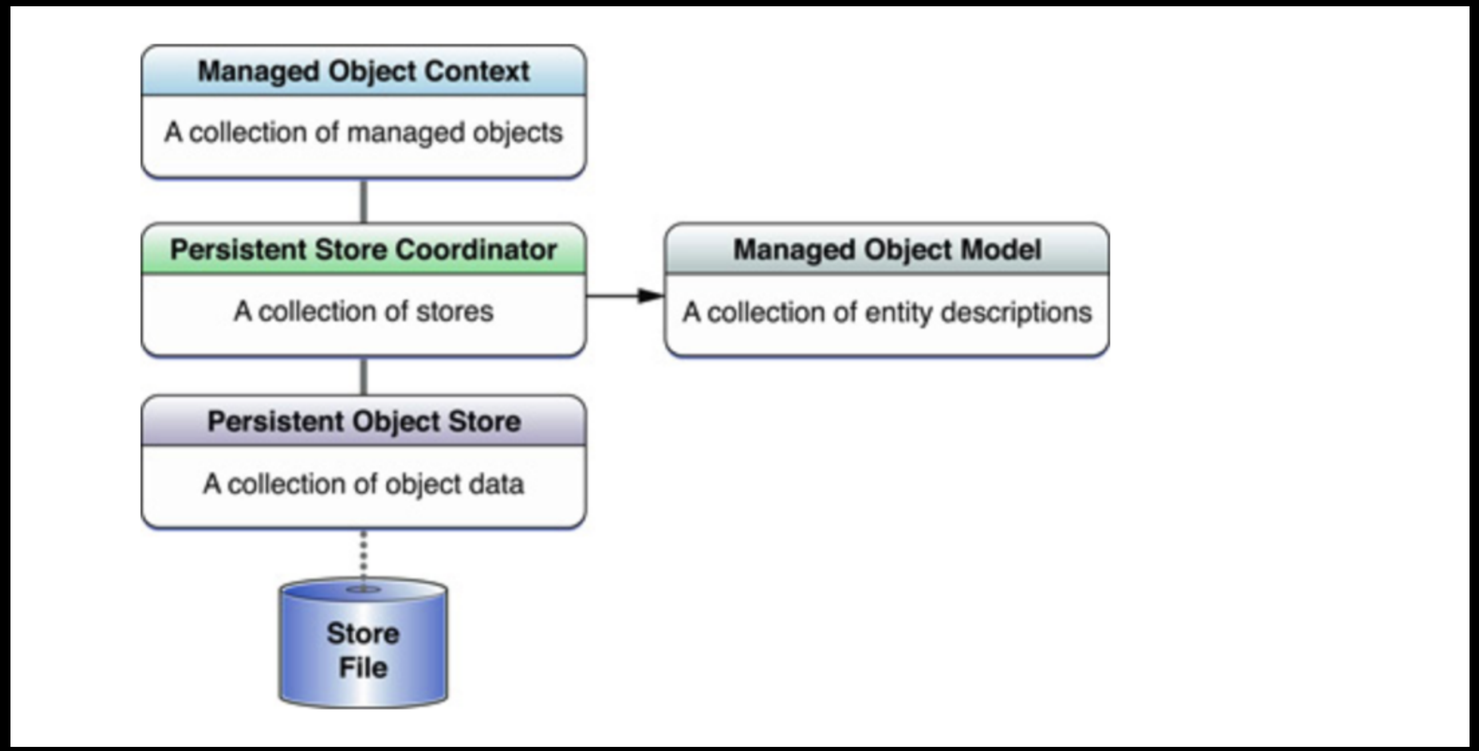
- Core Data

CoreData

- Core Data isn't just about loading your data from a database, its also about working with that data in memory.
- Basically, it manages the Model in MVC.
- Why use Core Data? Apple claims app your model layer will have 50% to 70% less code when using core data, sometimes.
- Core Data itself is not a database, its a way to easily allow your application to harness the power of a database. You can use CoreData without persisting to a DB if you want.

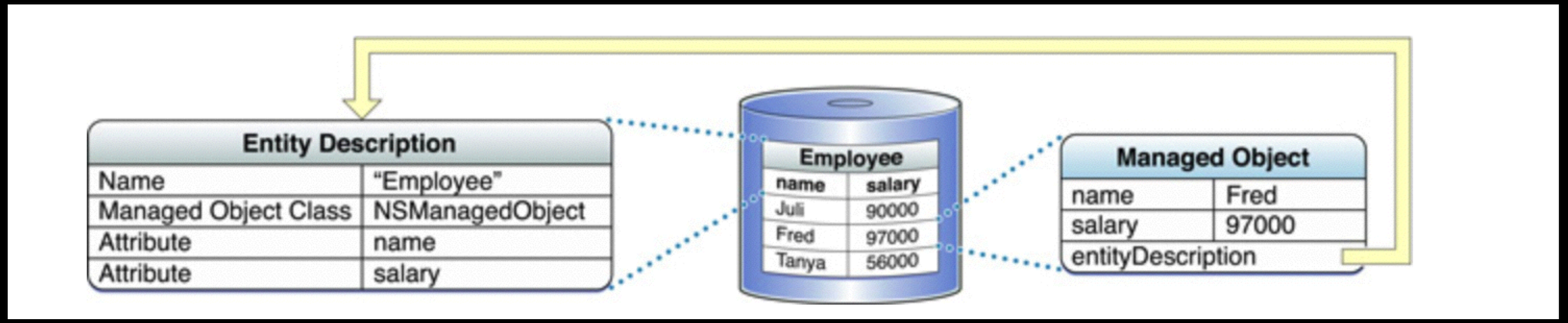
CoreData Stack

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



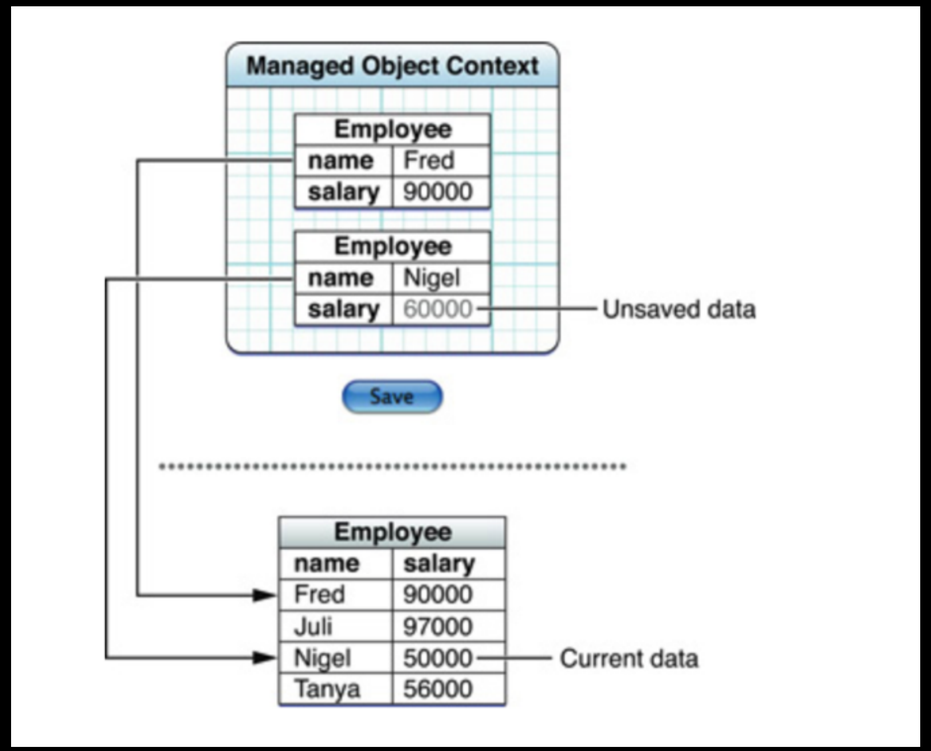
Managed Object

- A managed object is a model object in the MVC pattern. It represents a record from a persistent store.
- It is an instance of NSManagedObject or a subclass of it.
- Every managed object is registered with a context.
- 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.



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 stores.
- 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.

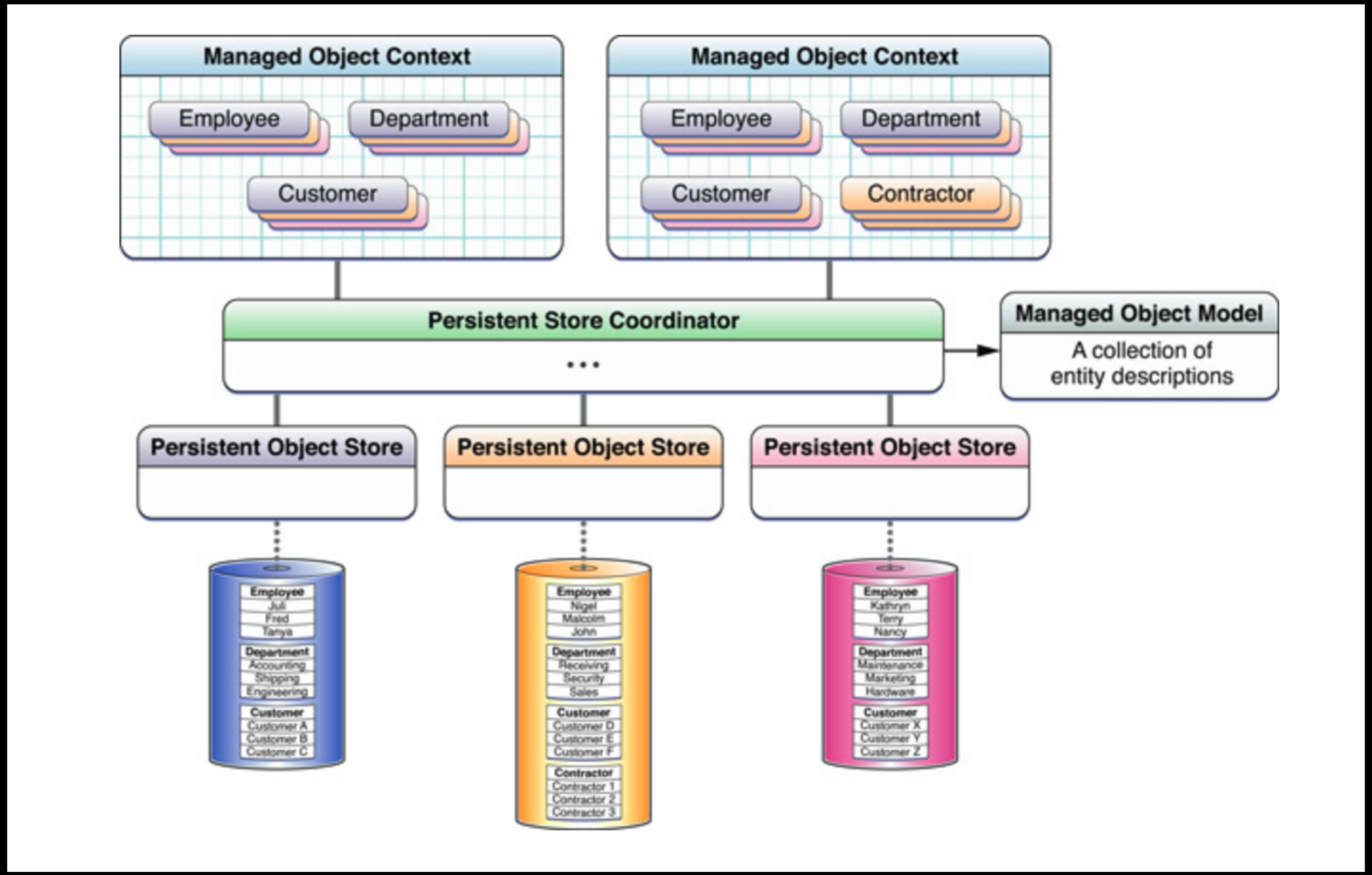


NSPersistentStoreCoordinator

- The PSC associates persistent store objects and a managed object model, and presents a facade to managed object contexts
- It lumps all the store objects together, so to the developer it appears as a single store
- Most apps only need one, but complex apps may need 2 or more
- Sits between `managedObjectContext` & persistent store (on disk)
- Persists objects to disk, reads objects from disk
- Has a reference to the `managedObjectModel`
- Can automatically migrate your existing database to a new schema*

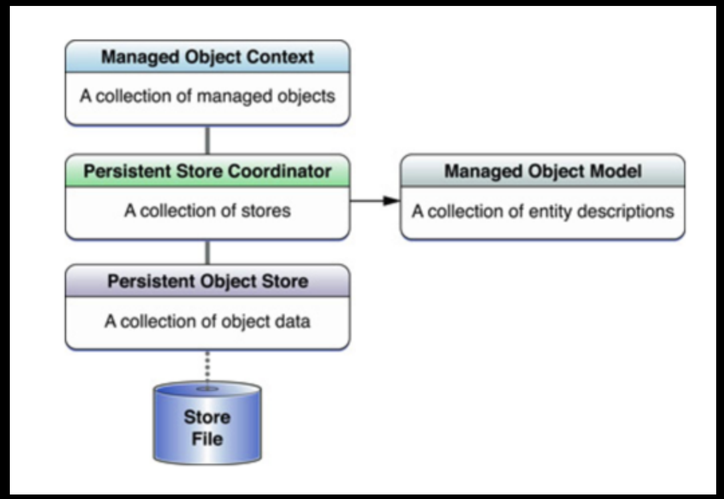
*sometimes

NSPersistentStoreCoordinator



Managed Object Model

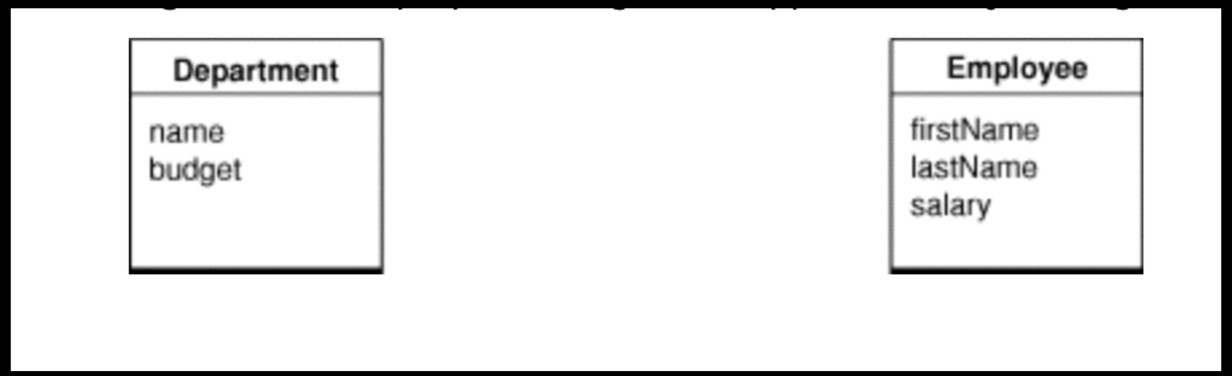
- A managed object model is a set of objects that together form a blueprint describing the managed objects you use in your application.
- A managed object model, or mom, allows core data to map records from a persistent store to managed objects that you use in your app.
- Describes a Core Data database 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 database.
- Special considerations when updating an app's schema

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, and references to other data baring objects are called relationships.



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 storing non standard values as well.

Relationships

- Not all properties of a model are attributes, some are relationships to other model objects.
- 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.
- Xcode can generate custom sublcasses NSManagedObject that are tailored to all of your entities.
- NSManagedObject provides a rich set of default behaviors.

Custom Managed Object Classes

Core data relies on NSManagedObject's implementation of the following methods, so don't override them:

- primitiveValueForKey:
- setPrimitiveValue:forKey:
- isEqual:
- hash
- superclass
- class
- self
- zone
- isProxy
- isKindOfClass:
- isMemberOfClass:
- conformsToProtocol:
- respondsToSelector:
- managedObjectContext
- entity
- objectID
- isInserted
- isUpdated
- isDeleted
- description
- isFault

Custom Managed Object Classes

- 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.

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.