iOS Dev Accelerator
Week2 Day2

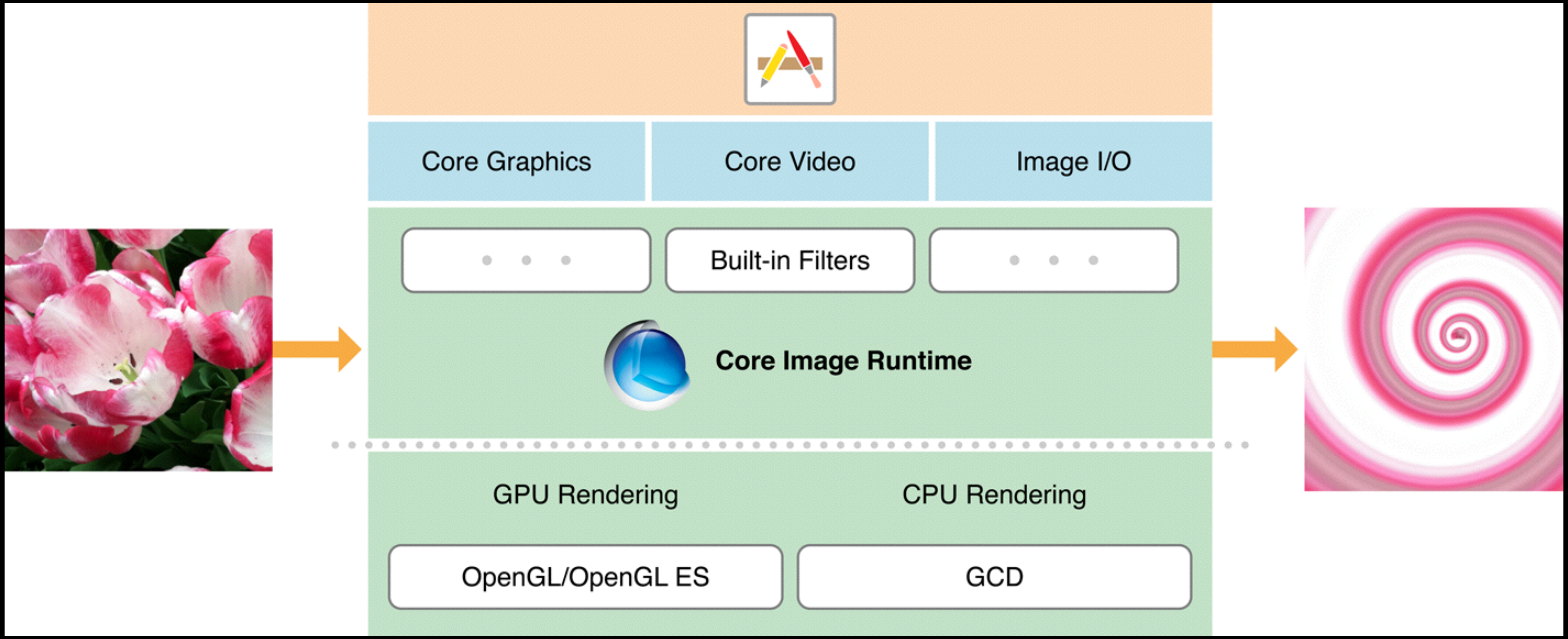
- CoreImage
- CoreData
- Git if we have time today

CoreImage

CoreImage

- “Core Image is an image processing and analysis technology designed to provide near real-time processing for still and video images”
- Can use either the GPU or CPU
- “Core Image hides the details of low-level graphic processing….You don’t need to know the details of OpenGL/ES to leverage the power of the GPU”

CoreImage

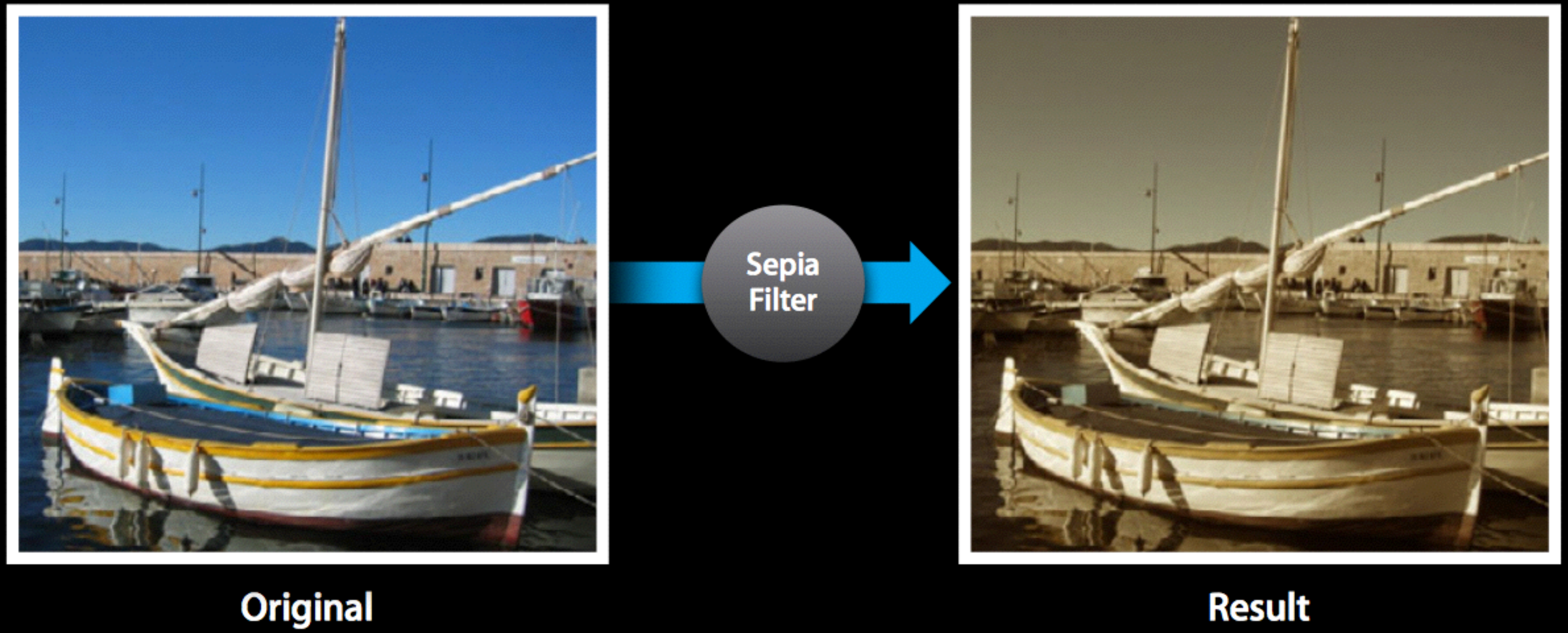

Core Graphics
Core Video
Image I/O
• • •
Built-in Filters
• • •
Core Image Runtime
GPU Rendering
CPU Rendering
OpenGL/OpenGL ES
GCD

CoreImage Offerings

guy using CoreImage

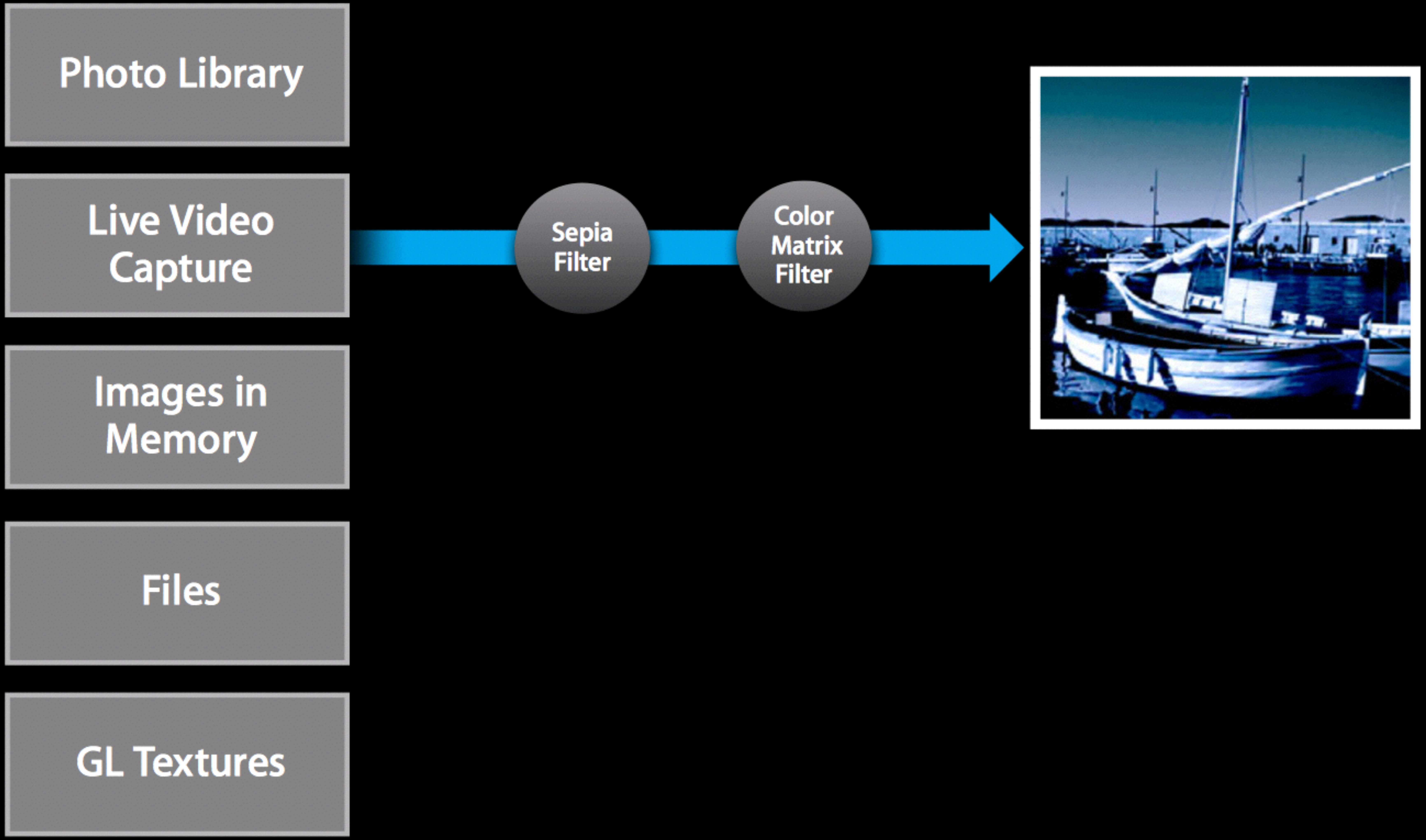
- Built-in image processing filters (90+ on iOS)
- Feature detection capability
- Support for automatic image enhancement
- Ability to chain multiple filters together to create custom effects

Filtering

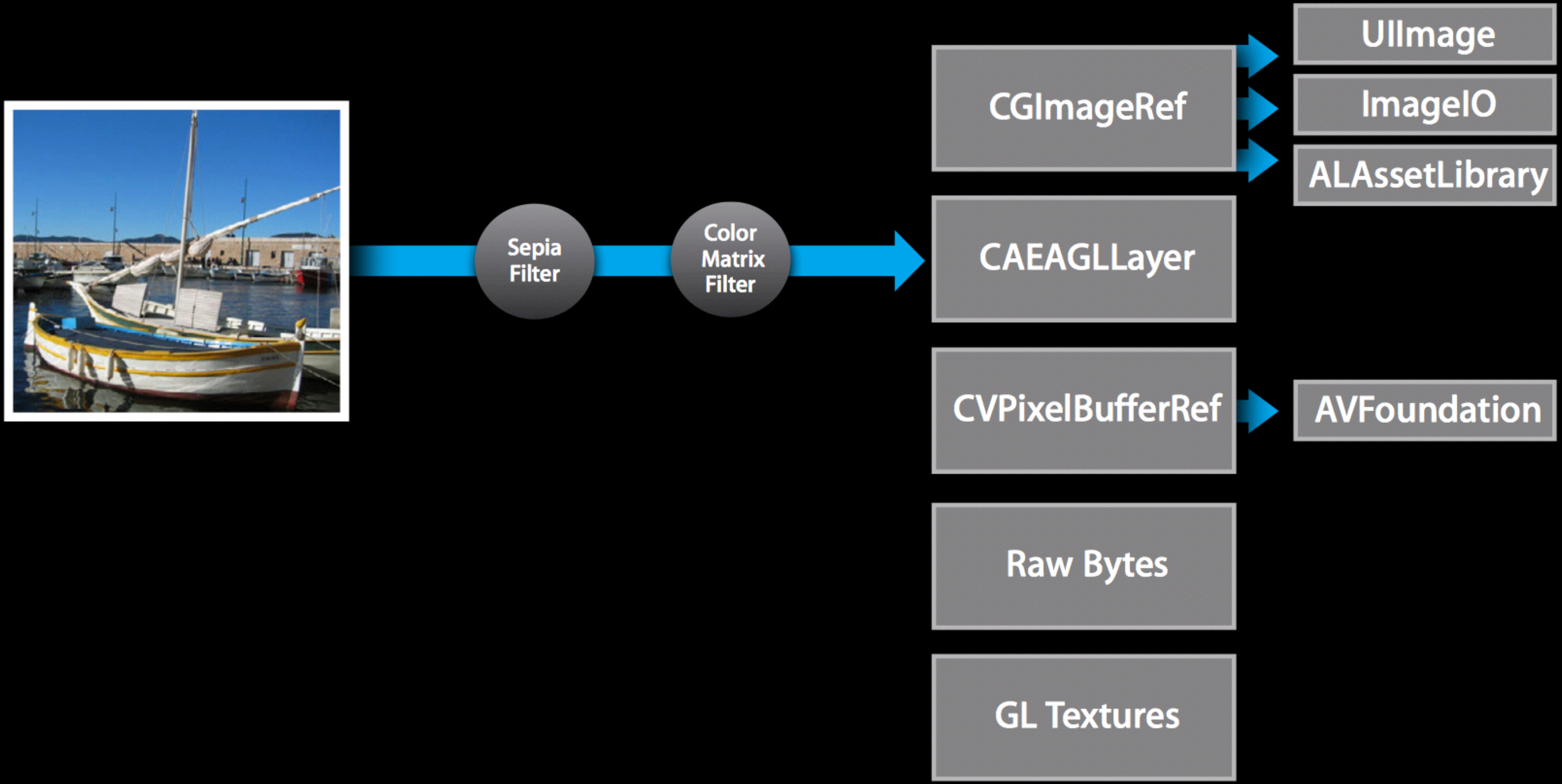


- Filters applied on a per pixel basis
- Can be chained together

Filtering Inputs are Flexible



As are the Outputs



CIAdditionCompositing
CIAffineClamp
CIAffineTile
CIAffineTransform
CIBarsSwipeTransition
CIBlendWithMask
CIBloom
CICheckerboardGenerator
CICircleSplashDistortion
CICircularScreen
CIColorBlendMode
CIColorBurnBlendMode
CIColorControls
CIColorCube
CIColorDodgeBlendMode
CIColorInvert
CIColorMap
CIColorMatrix
CIColorMonochrome

CIColorPosterize
CIConstantColorGenerator
CICopyMachineTransition
CICrop
CIDarkenBlendMode
CIDifferenceBlendMode
CIDisintegrateWithMask
CIDissolveTransition
CIDotScreen
CIEightfoldReflectedTile
CIExclusionBlendMode
CIExposureAdjust
CIFalseColor
CIFlashTransition
CIFourfoldReflectedTile
CIFourfoldRotatedTile
CIFourfoldTranslatedTile
CIGammaAdjust
CIGaussianBlur

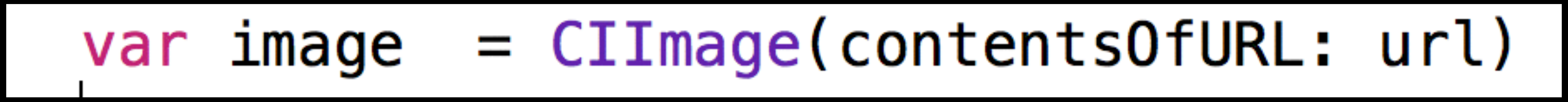
CIGaussianGradient
CIGlideReflectedTile
CIGloom
CIHardLightBlendMode
CIHatchedScreen
CIHighlightShadowAdjust
CIHoleDistortion
CIHueAdjust
CIHueBlendMode
CILanczosScaleTransform
CILightenBlendMode
CILightTunnel
CILinearGradient
CILineScreen
CILuminosityBlendMode
CIMaskToAlpha
CIMaximumComponent
CIMaximumCompositing
CIMinimumComponent

CIMinimumCompositing
CIModTransition
CIMultiplyBlendMode
CIMultiplyCompositing
CIOverlayBlendMode
CIPerspectiveTile
CIPerspectiveTransform
CIPinchDistortion
CIPixellate
CIRadialGradient
CIRandomGenerator
CISaturationBlendMode
CIScreenBlendMode
CISepiaTone
CISharpenLuminance
CISixfoldReflectedTile
CISixfoldRotatedTile
CISoftLightBlendMode
CISourceAtopCompositing

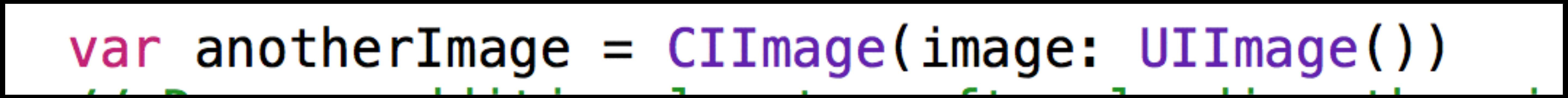
CISourceInCompositing
CISourceOutCompositing
CISourceOverCompositing
CIStarShineGenerator
CIStraightenFilter
CIStripesGenerator
CISwipeTransition
CITemperatureAndTint
CIToneCurve
CITriangleKaleidoscope
CITwelvefoldReflectedTile
CITwirlDistortion
CIUnsharpMask
CIVibrance
CIVignette
CIVortexDistortion
CIWhitePointAdjust

CIImage

- An Immutable object that represents the recipe for an Image
- Can represent a file from disk or the output of a CIFilter
- Multiple ways to create one:



```
var image  = CIImage(contentsOfURL: url)
```

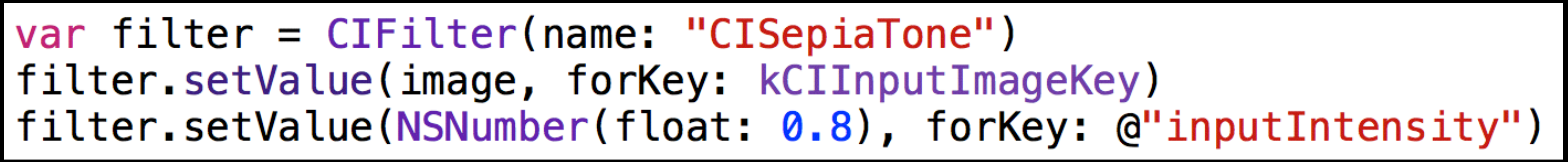


```
var anotherImage = CIImage(image: UIImage())
```

Also has inits from Raw bytes,
NSData,CGImage,PixelBuffers,etc

CIFilter

- Mutable object that represents a filter (not thread safe since its mutable!)
- Produces an output image based on the input.
- Each filter has a different set of inputKey's you can modify to alter the effect of the filter:



```
var filter = CIFilter(name: "CISepiaTone")
filter.setValue(image, forKey: kCIInputImageKey)
filter.setValue(NSNumber(float: 0.8), forKey: @"inputIntensity")
```

CIContext

- An object through which Core Image draws results
- Can be based on CPU or GPU
- Always use GPU because the CPU performance sucks

CPU context
GPU context

```
self.context = CIContext(options: nil)

var options = [kCIContextWorkingColorSpace : NSNull()]
var myEAGLContext = EAGLContext(API: EAGLRenderingAPI.OpenGLES2)
self.gpuContext = CIContext(EAGLContext: myEAGLContext, options: options)
```

Demo

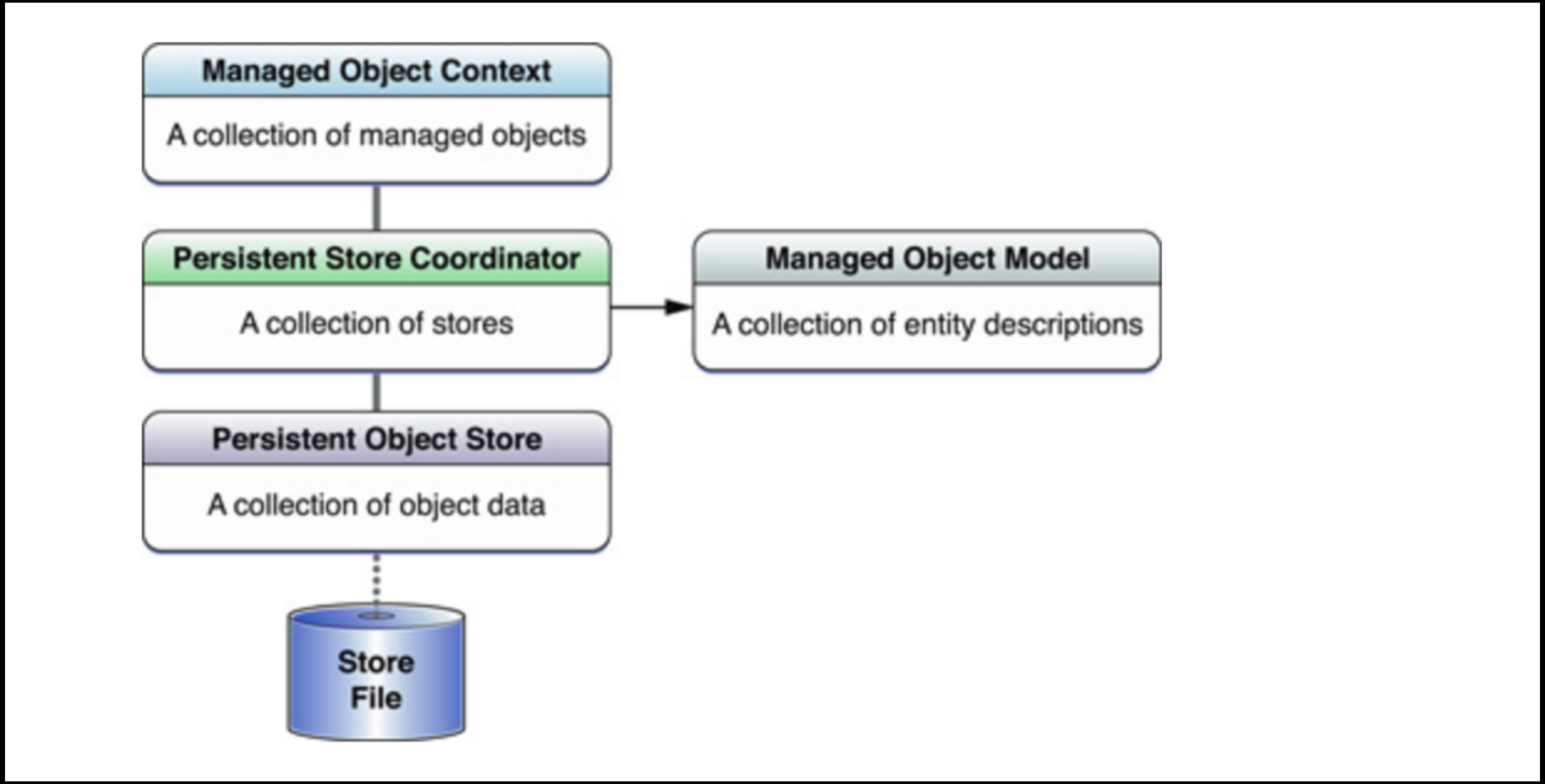
CoreData

CoreData

- Core Data is a framework designed to generalize and automate common tasks associated with object life-cycle and persistence.
- 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.

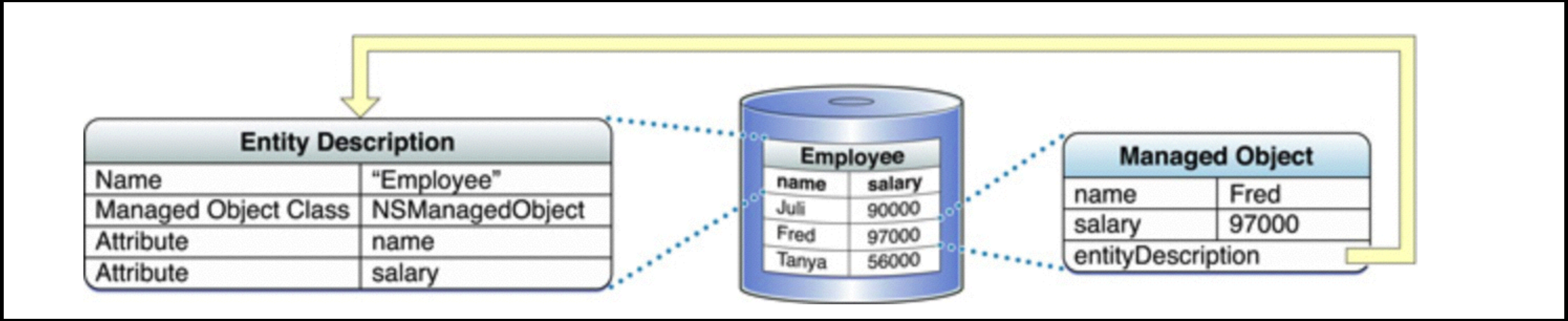
CoreData Stack

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


Managed Object Context
A collection of managed objects
Persistent Store Coordinator
A collection of stores
Managed Object Model
A collection of entity descriptions
Persistent Object Store
A collection of object data
Store File

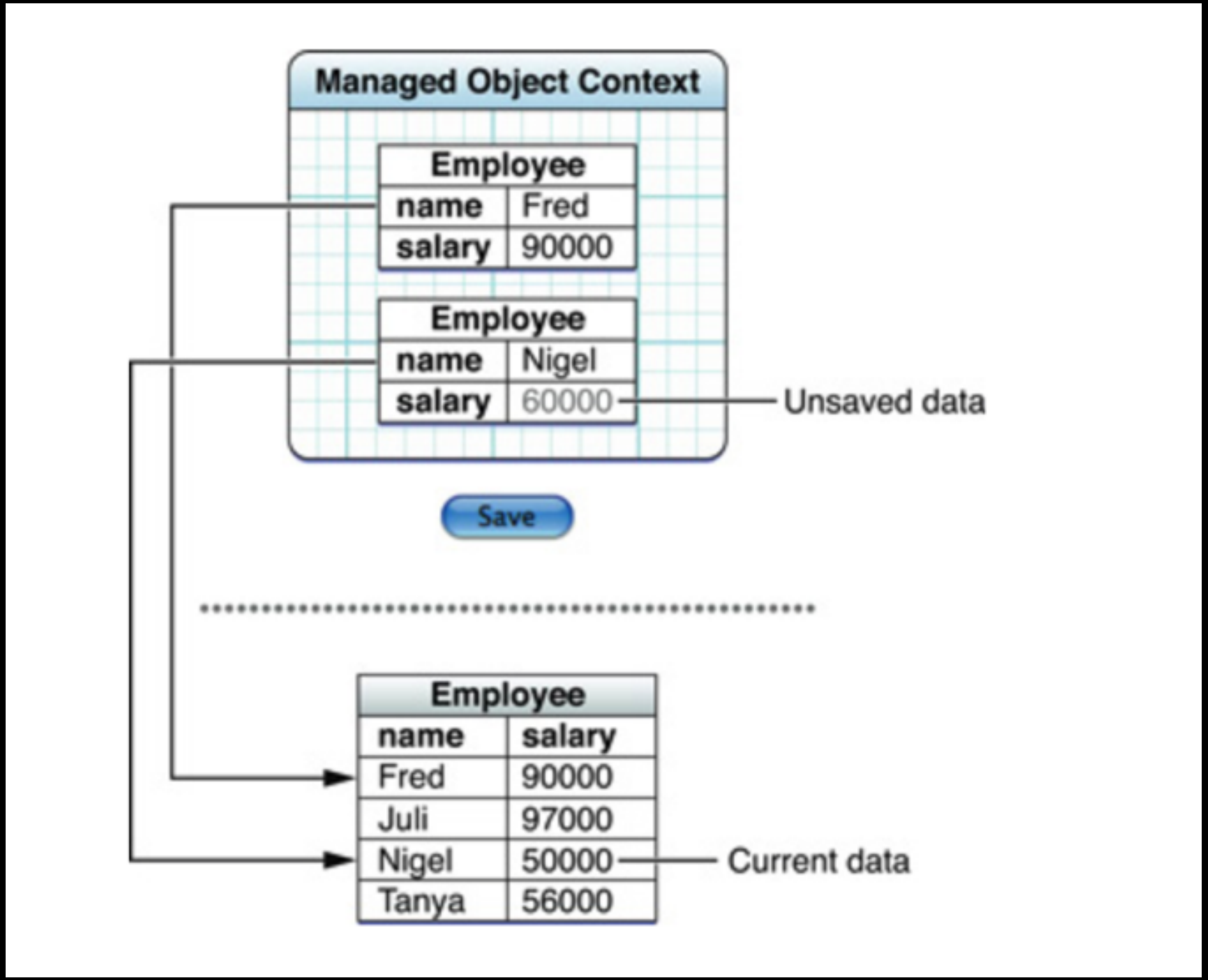
Managed Object

- A managed object is a model object in the MVC pattern. It represents a record from a persistent store.
- Instance of NSManagedObject or a subclass.
- Every managed object is registered with one 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, a “scratch pad”
- Manages 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 a persistent store coordinator (PSC)
- Every managed object knows which context it belongs to, and every context knows which objects it’s managing.

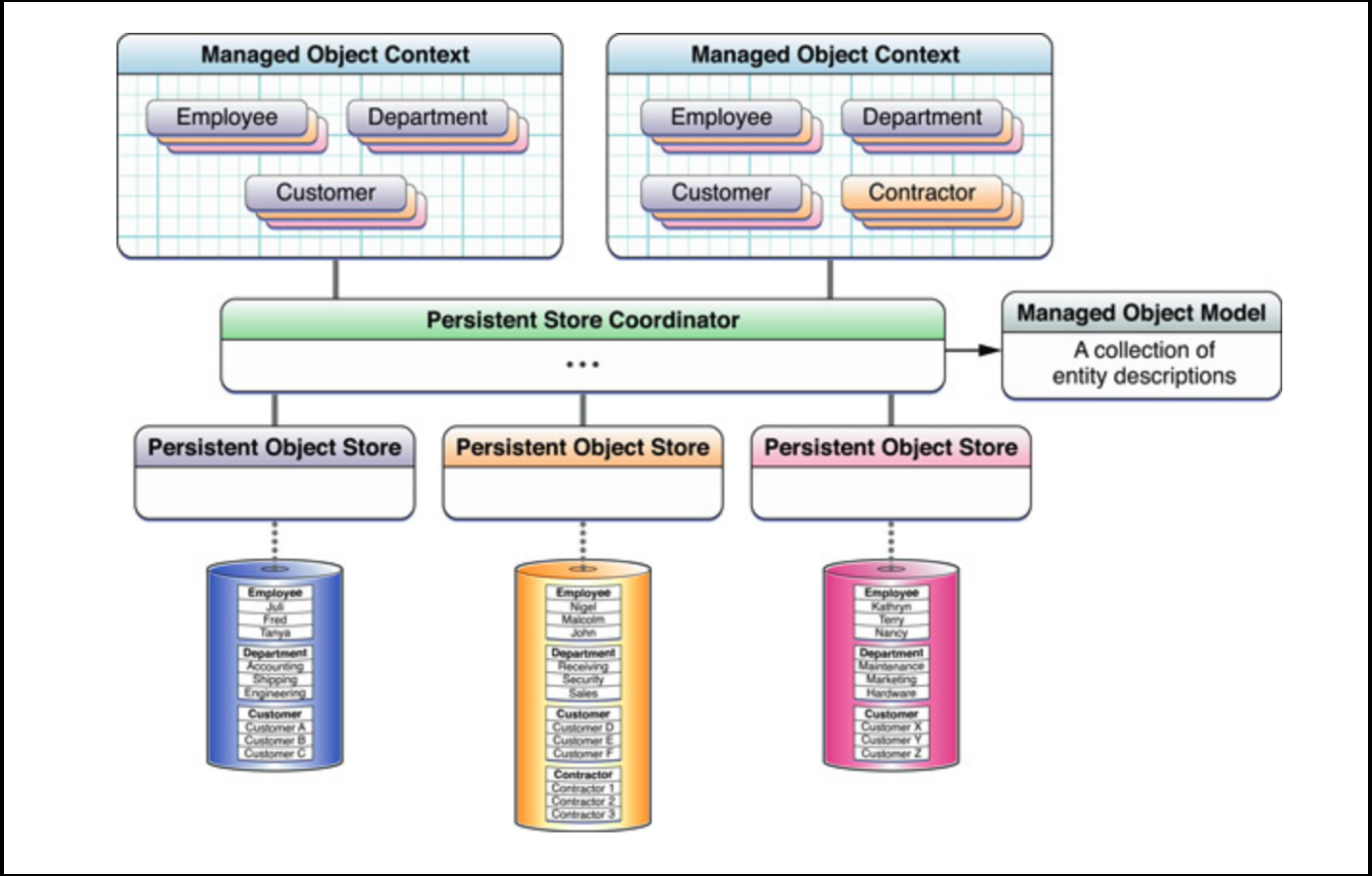


NSPersistentStoreCoordinator

- 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 super complex app may have several
- Exists 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*

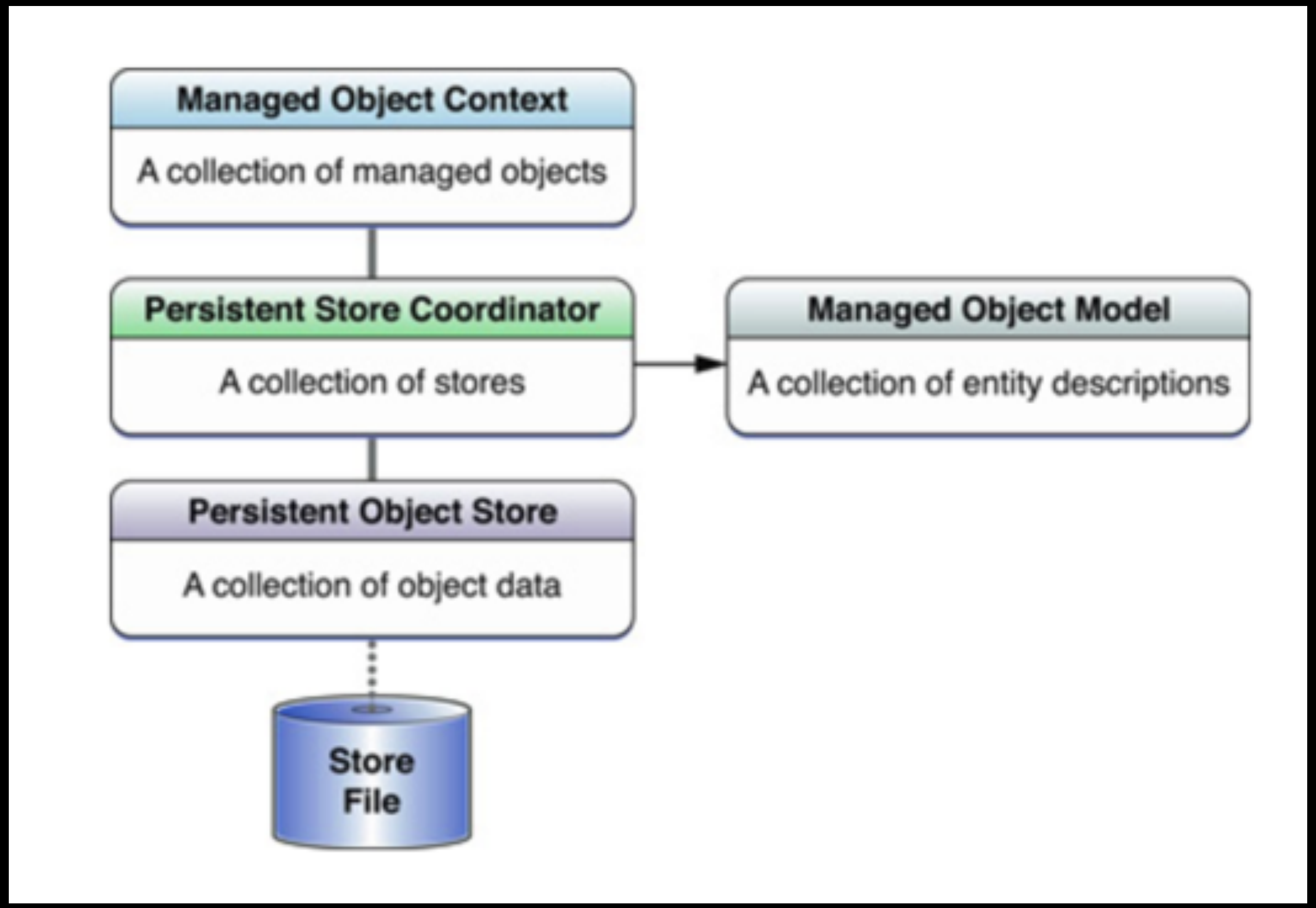
*sometimes

NSPersistentStoreCoordinator



Managed Object Model

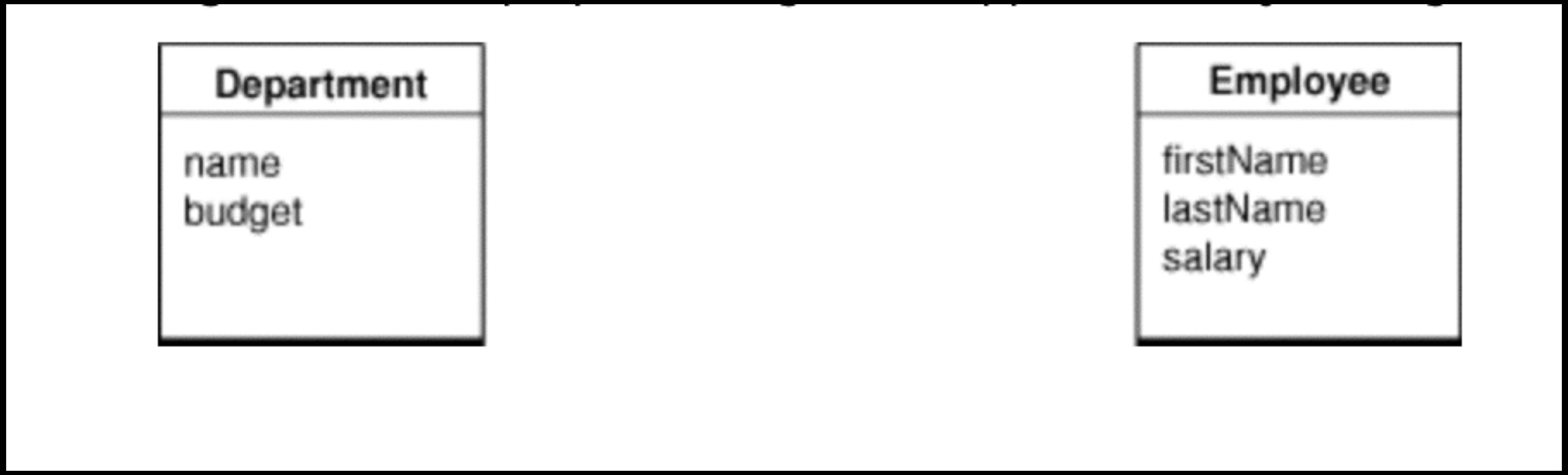
- A managed object model is a set of objects that together form a blueprint describing the managed objects you use in your application.
- A managed object model, or MOM, allows core data to map records from a persistent store to managed objects that you use in your app.
- Describes a Core Data database 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)


Managed Object Context
A collection of managed objects
Persistent Store Coordinator
A collection of stores
Managed Object Model
A collection of entity descriptions
Persistent Object Store
A collection of object data
Store File

- 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

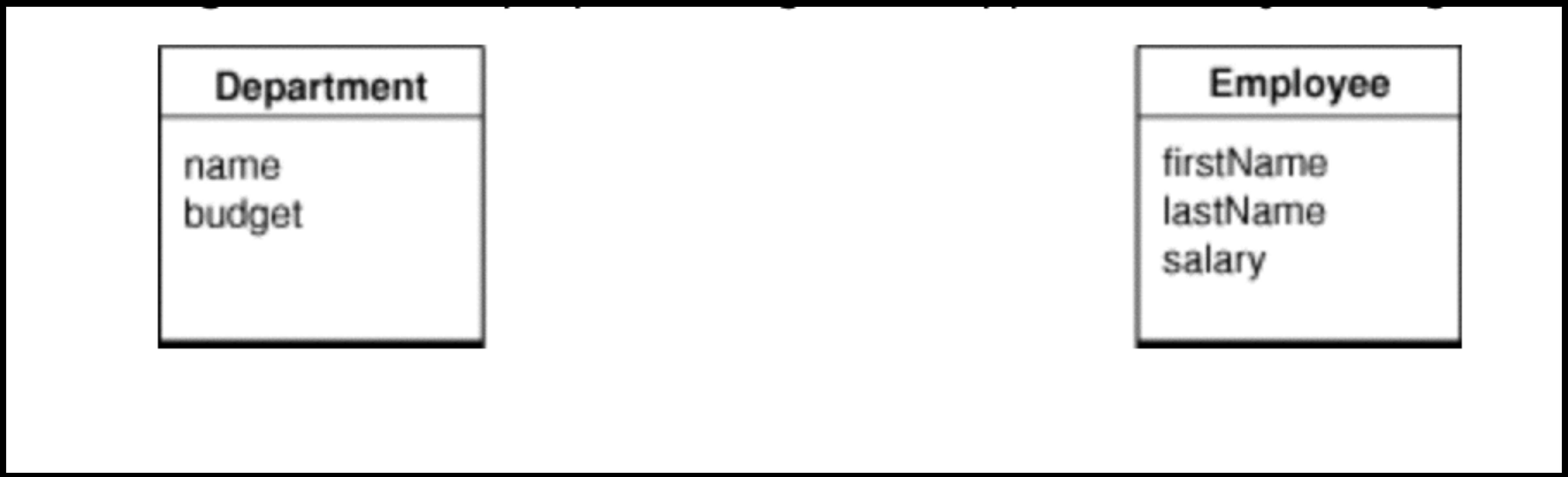
Entities

- “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.

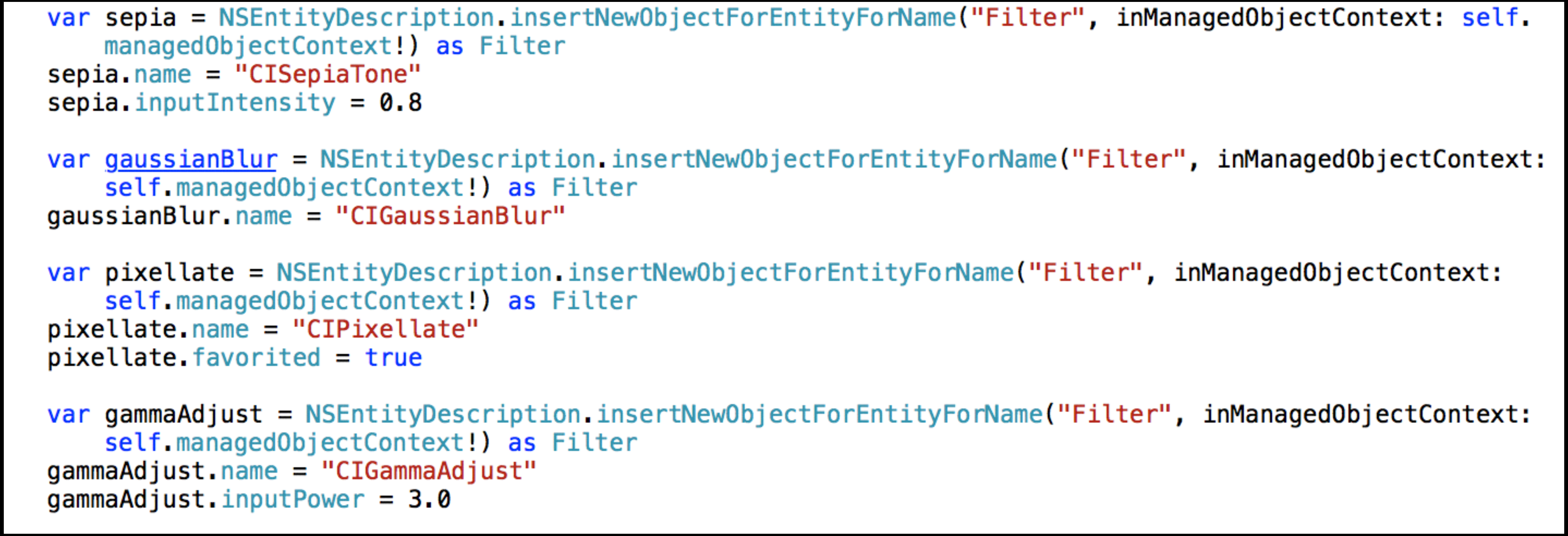
Custom Managed Object Classes

- Xcode can generate custom sublcasses NSManagedObject that are tailored to all of your entities.
- NSManagedObject provides a rich set of default behaviors.
- Core data relies on NSManagedObject's implementation of the following methods: primitiveValueForKey:, setPrimitiveValue:forKey:, isEqual:, hash, superclass, class, self, zone, isProxy, isKindOfClass:, isMemberOfClass:, conformsToProtocol:, respondsToSelector:, managedObjectContext, entity, objectID, isInserted, isUpdated, isDeleted, description, and isFault. So don't override them.
- 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.

Inserting into your data store



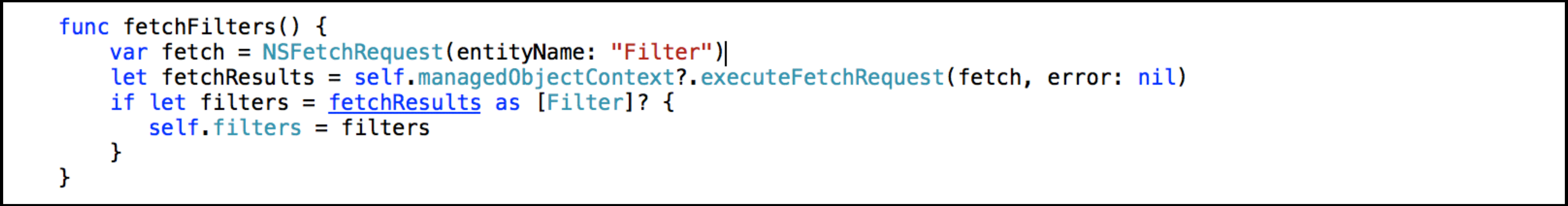
```swift
var sepia = NSEntityDescription.insertNewObjectForEntityForName("Filter", inManagedObjectContext: self.
    managedObjectContext!) as Filter
sepia.name = "CISepiaTone"
sepia.inputIntensity = 0.8

var gaussianBlur = NSEntityDescription.insertNewObjectForEntityForName("Filter", inManagedObjectContext:
    self.managedObjectContext!) as Filter
gaussianBlur.name = "CIGaussianBlur"

var pixellate = NSEntityDescription.insertNewObjectForEntityForName("Filter", inManagedObjectContext:
    self.managedObjectContext!) as Filter
pixellate.name = "CIPixellate"
pixellate.favorited = true

var gammaAdjust = NSEntityDescription.insertNewObjectForEntityForName("Filter", inManagedObjectContext:
    self.managedObjectContext!) as Filter
gammaAdjust.name = "CIGammaAdjust"
gammaAdjust.inputPower = 3.0
```

Simple Fetch



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

Demo