USING CORE DATA IN SWIFT

JESSE SQUIRES

JESSESQUIRES.COM • @JESSE_SQUIRES • GITHUB/JESSESQUIRES

WHAT IS CORE DATA?

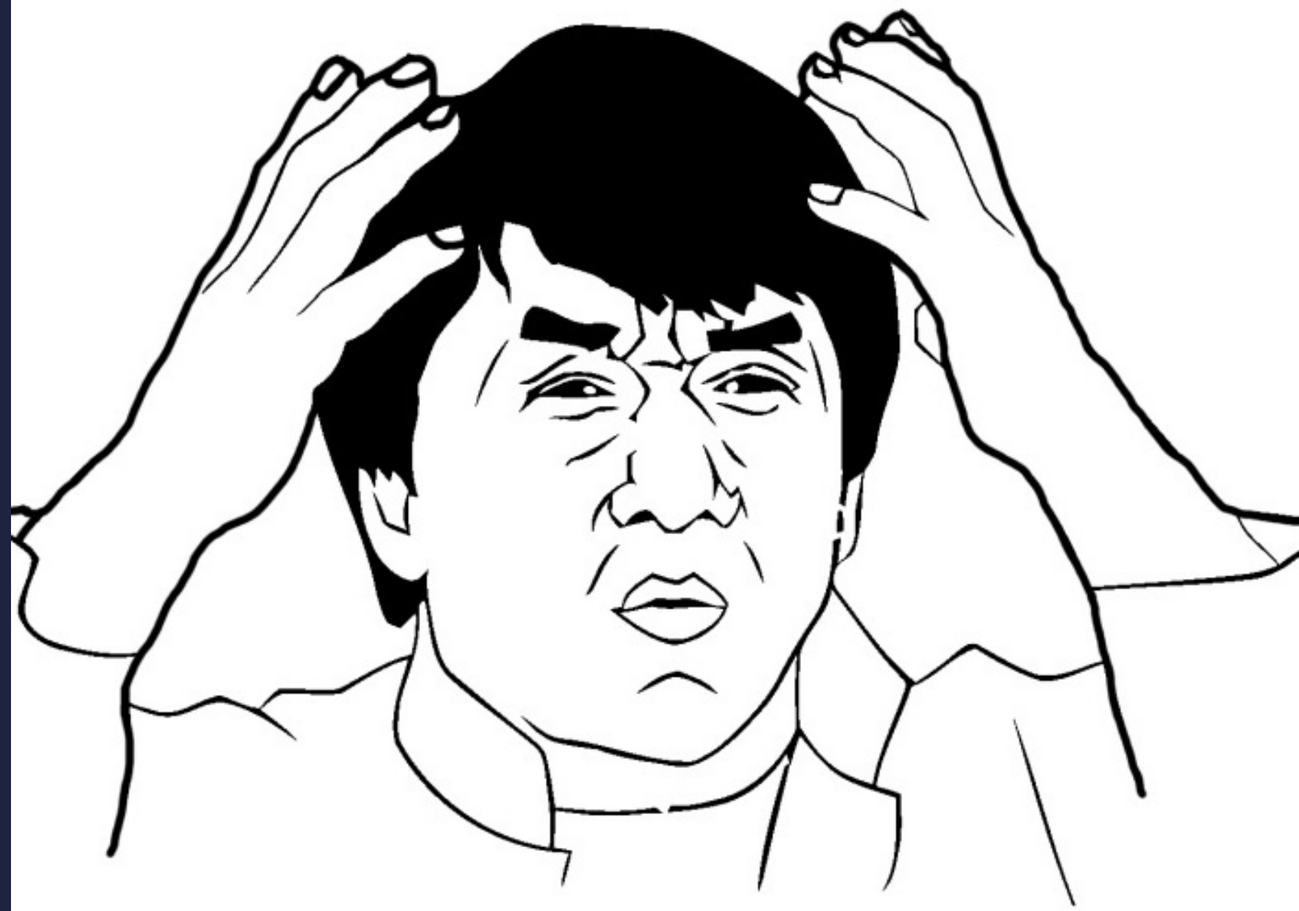
The Core Data framework provides generalized and automated solutions to common tasks associated with object life-cycle and object graph management, including persistence.

– Core Data Programming Guide

Core Data is backed by a SQLite database.

However, it is not a relational database or RDBMS.

BACKED BY SQLITE.
NOT A DATABASE?



CORE DATA MANAGES:

LIFE-CYCLE
GRAPH
PERSISTENCE
SEARCHING
OF OBJECTS

CORE DATA STACK:

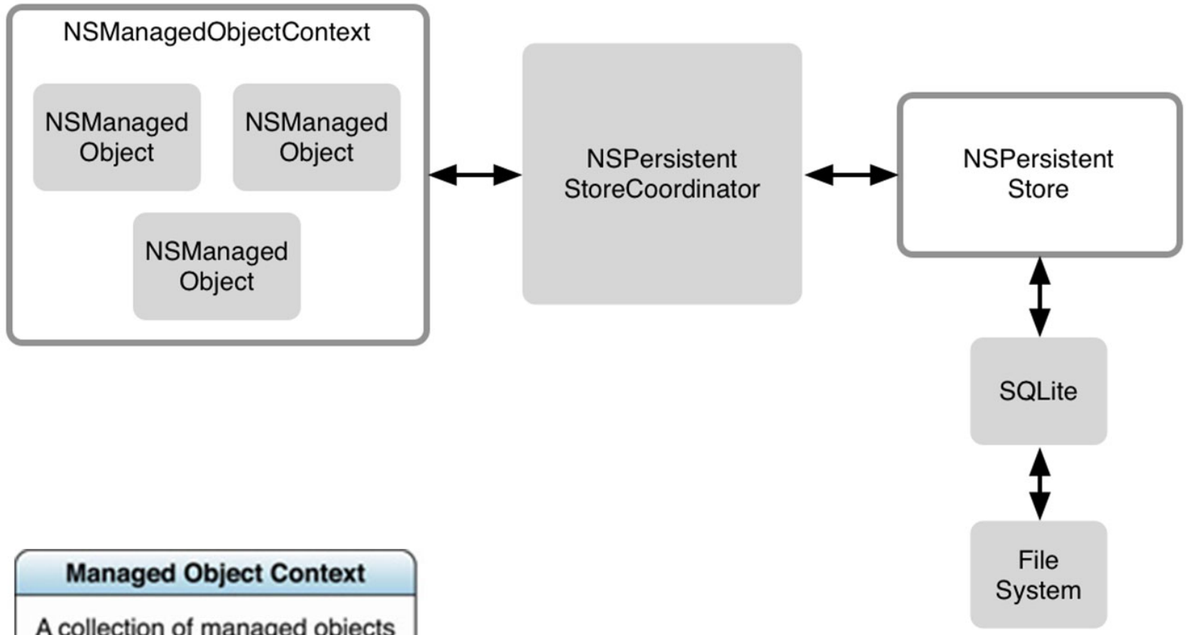
Managed objects
`NSMangedObject`

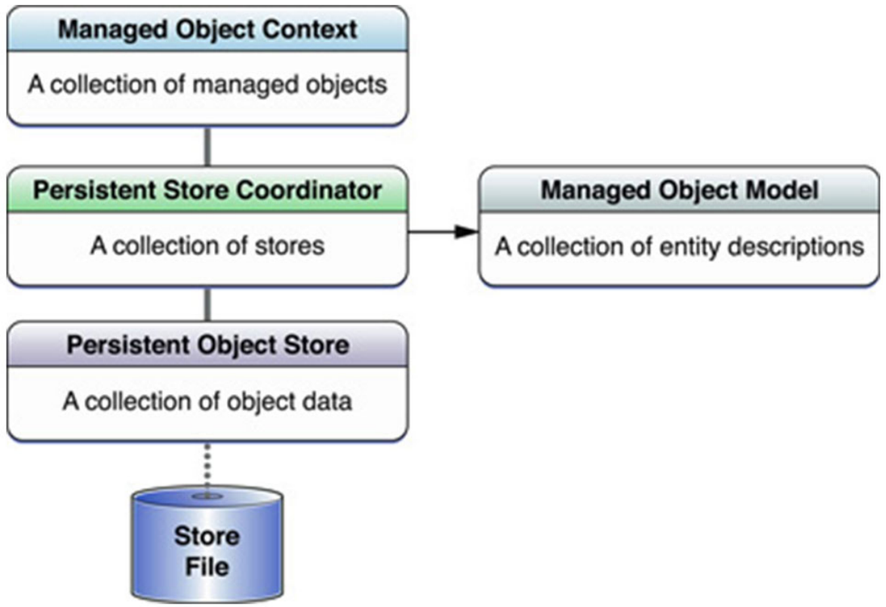
Managed object context
`NSManagedObjectContext`

Persistent Store Coordinator
`NSPersistentStoreCoordinator`

Persistent Store
`NSPersistentStore`

SQLite





WHY USE CORE DATA?

- Provides featues you need
- "Mature, unit tested, optimized"
- Part of iOS and OS X toolchain
- Apple continues to invest heavily in it
- Popular, tons of resources online

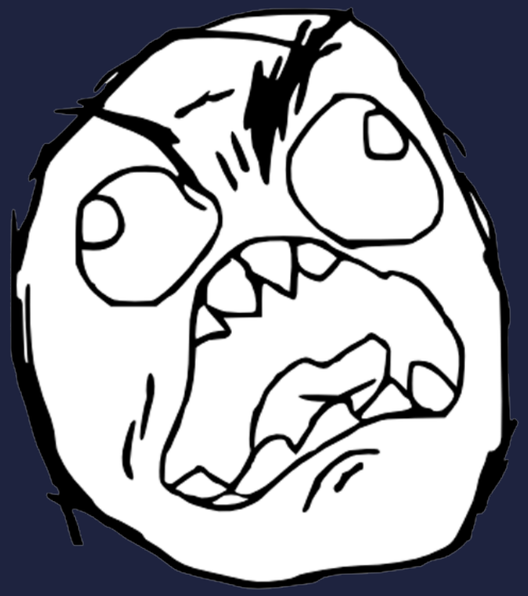
WHY USE SWIFT?

- Clarity
- Type-safety
- Swift-only features
- Functional paradigms

SWIFT + CORE DATA

WARNING: TOOLS ARE IMMATURE

```
SourceKitService
Terminated
Editor functionality
temporarily limited.
```



STANDING UP THE CORE DATA STACK

Same boilerplate code as Objective-C

Better in Swift

CORE DATA STACK

```
struct CoreDataModel {

    let name: String

    let bundle: NSBundle

    init(name: String, bundle: NSBundle)

    // other properties & methods
}
```

CORE DATA STACK

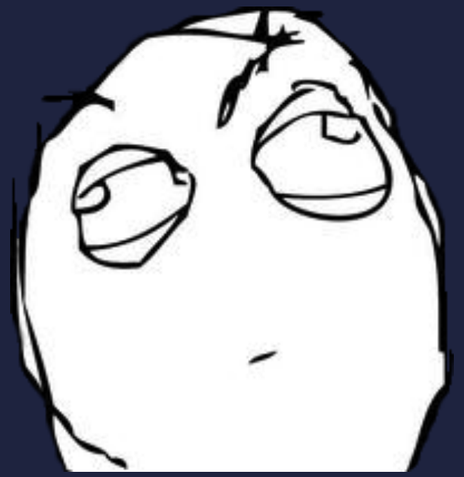
```
class CoreDataStack {

    let model: CoreDataModel

    let managedObjectContext: NSManagedObjectContext

    let persistentStoreCoordinator: NSPersistentStoreCoordinator

    init(model: CoreDataModel,
         storeType: String,
         concurrencyType: NSManagedObjectContextConcurrencyType)

    // other properties and methods
}
```

CORE DATA STACK

```
let model = CoreDataModel(name: "MyModel", bundle: myBundle)

let stack = CoreDataStack(model: model,
                          storeType: NSSQLiteStoreType,
                          concurrencyType: .MainQueueConcurrencyType)

// Use context
stack.managedObjectContext
```

CORE DATA STACK

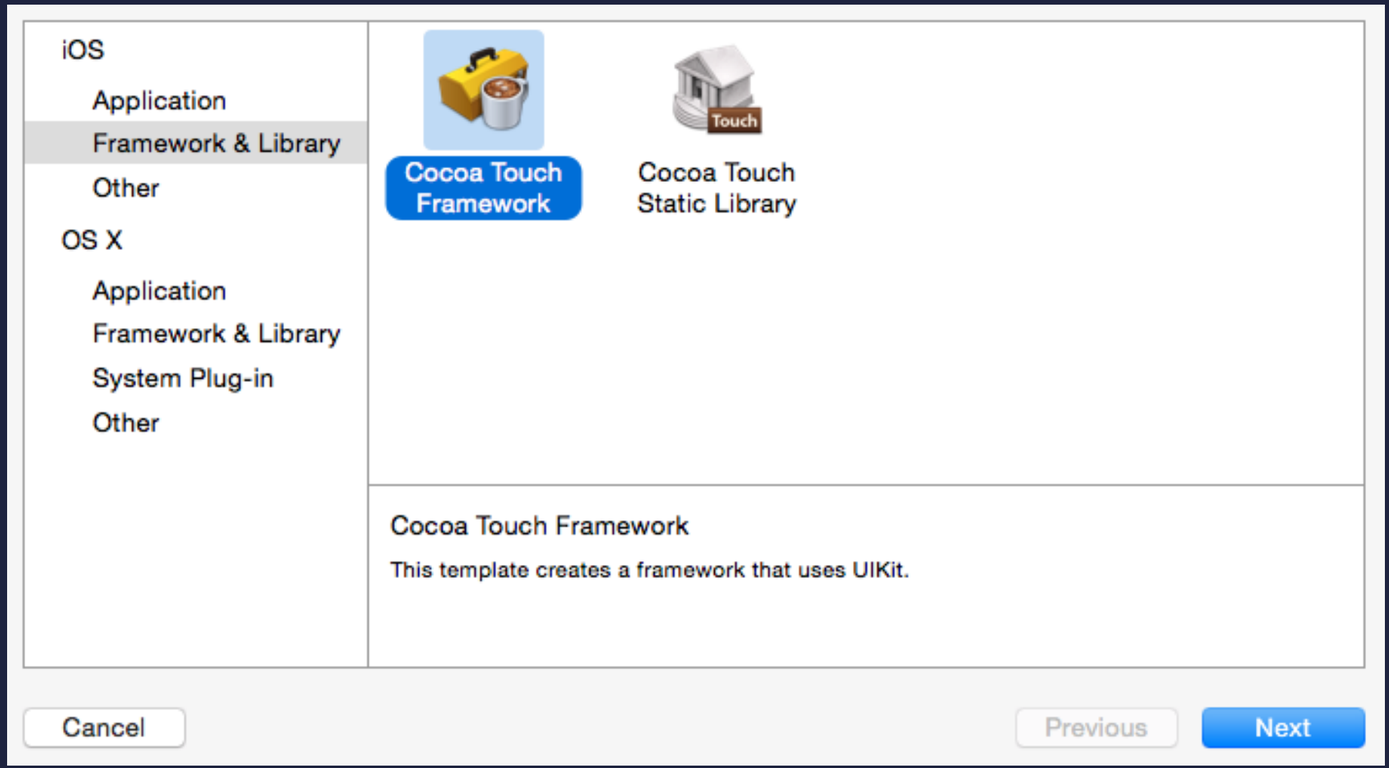
AppDelegate.m



DO NOT

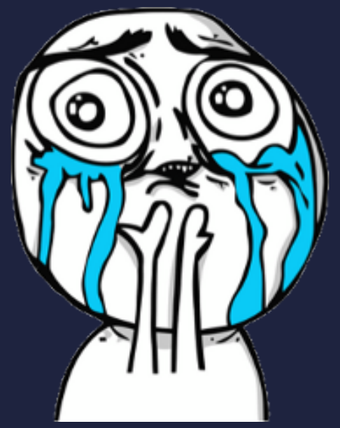
USE FRAMEWORKS

Clear model namespace, Modular, Reusable, Unit Testing



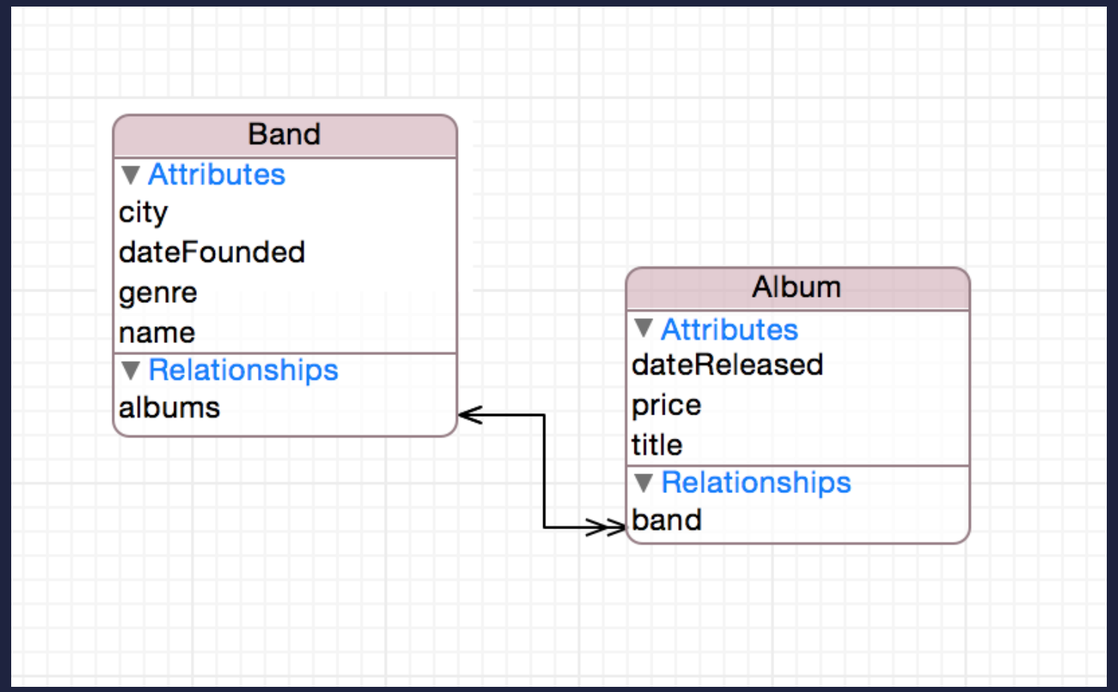
CREATING MANAGED OBJECTS

- Xcode generated classes are terrible
- mogenerator Swift support still experimental
  (Last release Sept 2014)



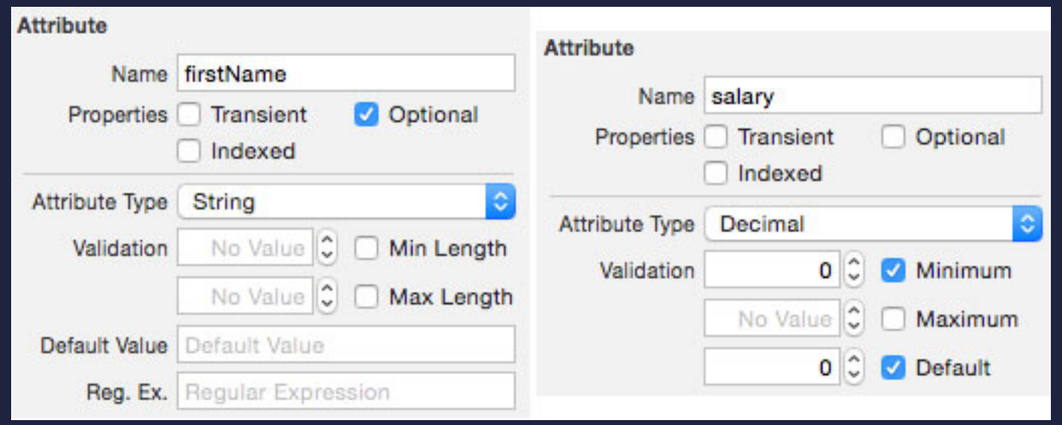
CREATING MANAGED OBJECTS

VISUAL MODEL EDITOR



CREATING MANAGED OBJECTS

ATTRIBUTE VALIDATION



OBJECTIVE-C

```
@interface Employee : NSManagedObject

@property (nonatomic, retain) NSString * address;
@property (nonatomic, retain) NSDate * dateOfBirth;
@property (nonatomic, retain) NSString * email;
@property (nonatomic, retain) NSString * name;
@property (nonatomic, retain) NSDecimalNumber * salary;
@property (nonatomic, retain) NSNumber * status;

@end
```

SWIFT

```
class Employee: NSManagedObject {

    @NSManaged var address: String?
    @NSManaged var dateOfBirth: NSDate
    @NSManaged var email: String?
    @NSManaged var name: String
    @NSManaged var salary: NSDecimalNumber
    @NSManaged var status: Int32

}
```

OPTIONALS

Xcode will generate String instead of String?

```
@property (nonatomic, retain) NSString * address;
```

```
@NSManaged var address: String?
```

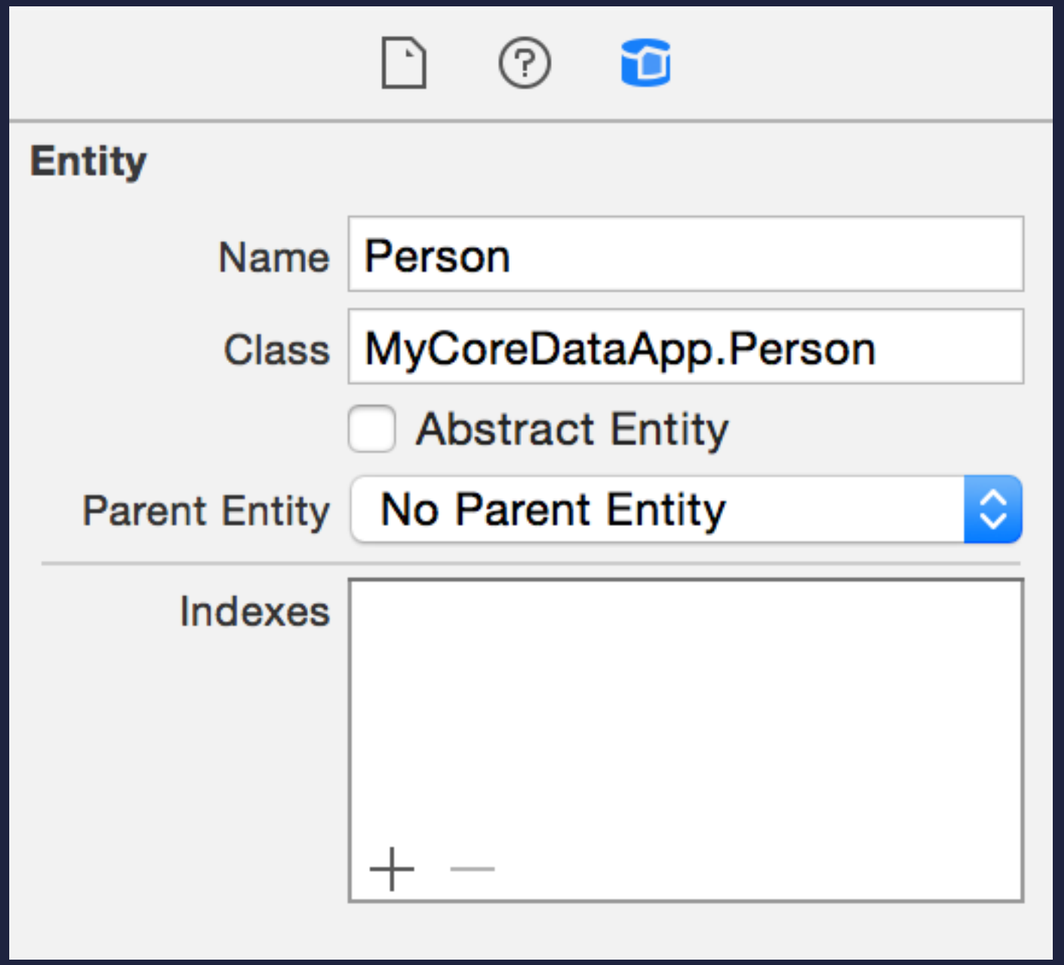
CREATING MANAGED OBJECTS

PREFIXED SUBCLASSES

NSManagedObject

<ModuleName>.<ClassName>

Swift namespaces



Xcode does not prefix classes automatically

Must add prefix manually after generating classes

No prefix means runtime crash, obscure errors

INSTANTIATING MANAGED OBJECTS

Reduce boilerplate, generalize

`NSEntityDescription`

THE OBJECTIVE-C WAY

```
// "Person"
NSString *name = [Person entityName];


@implementation NSManagedObject (Helpers)

+ (NSString *)entityName
{
   return NSStringFromClass([self class]);
}

@end
```

THE OBJECTIVE-C WAY

```
// Create new person
Person *person = [Person insertNewObjectInContext:context];


@implementation NSManagedObject (Helpers)

+ (instancetype)insertNewObjectInContext:(NSManagedObjectContext *)context
{
   return [NSEntityDescription insertNewObjectForEntityForName:[self entityName]
                                          inManagedObjectContext:context];
}

@end
```

THE SWIFT WAY?

```
// "MyApp.Person"
let fullName = NSStringFromClass(object_getClass(self))

extension NSManagedObject {

    class func entityName() -> String {
        let fullClassName = NSStringFromClass(object_getClass(self))
        let nameComponents = split(fullClassName) { $0 == "." }
        return last(nameComponents)!
    }
}

// "Person"
let entityName = Person.entityName()
```

THE SWIFT WAY?

```
// Create new person
let person = Person(context: context)

extension NSManagedObject {

    convenience init(context: NSManagedObjectContext) {
        let name = self.dynamicType.entityName()

        let entity = NSEntityDescription.entityForName(name,
                                inManagedObjectContext: context)!

        self.init(entity: entity,
                  insertIntoManagedObjectContext: context)
    }
}
```

THE SWIFT WAY?

```
class Employee: NSManagedObject {

    init(context: NSManagedObjectContext) {

        let entity = NSEntityDescription.entityForName("Employee",
                                        inManagedObjectContext: context)!

        super.init(entity: entity,
                   insertIntoManagedObjectContext: context)
    }
}
```

NOT VERY SWIFT

"OBJECTIVE-C WITH A NEW SYNTAX"

THE OBJECTIVE-C WAY
IS NOT ALWAYS
THE SWIFT WAY

EMBRACE
THE SWIFTNESS

SWIFT DESIGNATED INITIALIZERS

1. Stored properties must be assigned initial value
2. Designated `init` fully initializes all properties
3. Convenience `init` are secondary
4. Convenience `init` must call designated `init`
5. Superclass initializers not inherited in subclasses by default

DESIGNATED INITIALIZERS?

```
// designated init
init(entity:insertIntoManagedObjectContext:)

// our convenience init
convenience init(context:)
```

CORE DATA BYPASSES INITIALIZATION RULES

@NSManaged

```
class Employee: NSManagedObject {

    init(context: NSManagedObjectContext,
         name: String,
         dateOfBirth: NSDate,
         salary: NSDecimalNumber,
         employeeId: String = NSUUID().UUIDString,
         email: String? = nil,
         address: String? = nil) {

          // init
      }
}
```

typealias

TYPEALIAS

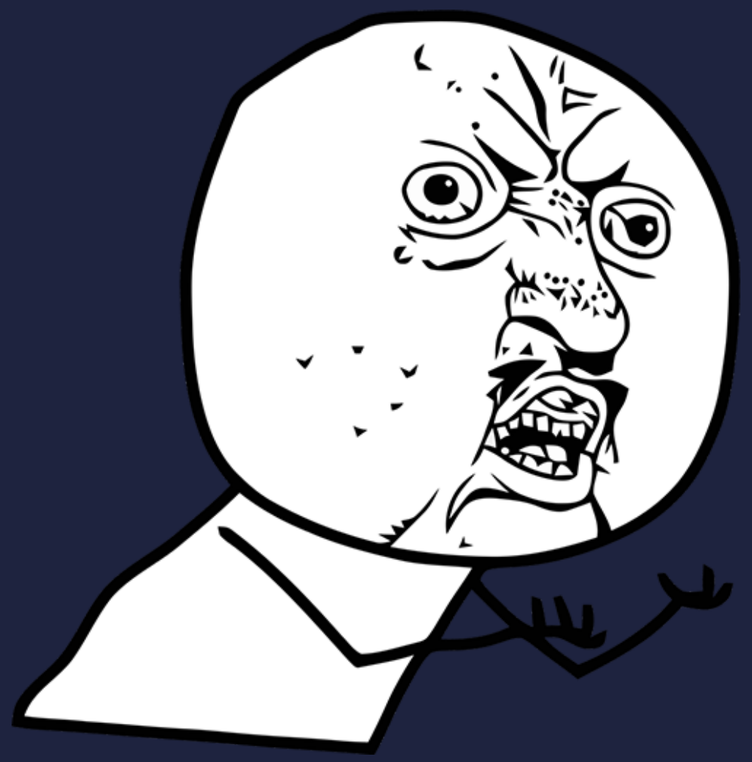
```
typealias EmployeeId = String

class Employee: NSManagedObject {

    @NSManaged var employeeId: EmployeeId
}

// Example
let id: EmployeeId = "12345"
```

RELATIONSHIPS

NSSet

Set<T>

Set<T>



Nope

enum

ENUM

```
enum Genre: String {
    case BlackMetal = "Black Metal"
    case DeathMetal = "Death Metal"
    case DoomMetal = "Doom Metal"
    case FolkMetal = "Folk Metal"
    case Grindcore = "Grindcore"
    case Hardcore = "Hardcore"
    case CrustPunk = "Crust Punk"
    case StreetPunk = "Street Punk"
    case Thrash = "Thrash"
}
```

ENUM

```
public class Band: NSManagedObject {

    @NSManaged private var genreValue: String

    public var genre: Genre {
        get {
            return Genre(rawValue: self.genreValue)!
        }
        set {
            self.genreValue = newValue.rawValue
        }
    }
}
```

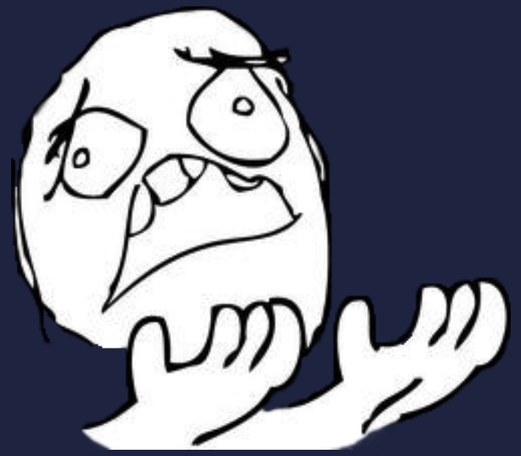
ENUM

```
// old
band.genre = "Black Metal"

// new
band.genre = .BlackMetal
```

ENUM

Unfortunately, must use private property for fetch requests

```
let fetch = NSFetchRequest(entityName: "Band")
fetch.predicate = NSPredicate(format: "genreValue == %@", genre)
```



FUNCTIONAL PARADIGMS

WITH MICRO-LIBRARIES

SAVING

```
var error: NSError?

let success: Bool = managedObjectContext.save(&error)

// handle success or error
```

SAVING

```
func saveContext(context:) -> (success: Bool, error: NSError?)

// Example
let result = saveContext(context)

if !result.success {
    println("Error: \(result.error)")
}
```

FETCH REQUESTS

```
var error: NSError?
var results = context.executeFetchRequest(request, error: &error)

// [AnyObject]?
if results == nil {
    println("Error = \(error)")
}
```

FETCH REQUESTS

```
// T is a phantom type
class FetchRequest <T: NSManagedObject>: NSFetchRequest {

    init(entity: NSEntityDescription) {
        super.init()
        self.entity = entity
    }
}
```

FETCH REQUESTS

```
typealias FetchResult = (success: Bool, objects: [T], error: NSError?)

func fetch <T> (request: FetchRequest<T>,
                context: NSManagedObjectContext) -> FetchResult
```

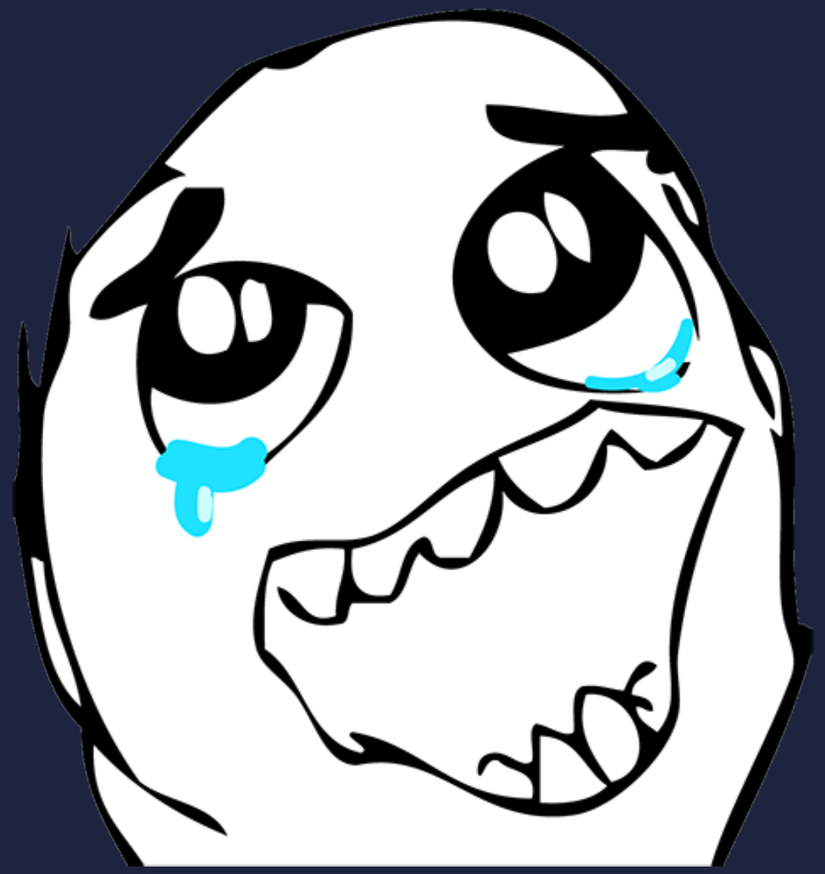
FETCH REQUESTS

```
typealias FetchResult = (success: Bool, objects: [T], error: NSError?)

func fetch <T>(request: FetchRequest<T>,
               context: NSManagedObjectContext) -> FetchResult {

    var error: NSError?

    if let results = context.executeFetchRequest(request, error: &error) {
        return (true, results as! [T], error)
    }

    return (false, [], error)
}
```

FETCH REQUESTS

```
// Example
let request = FetchRequest<Band>(entity: entityDescription)

let results = fetch(request: request, inContext: context)

if !results.success {
   println("Error = \(results.error)")
}

results.objects // [Band]
```

SLIGHTLY LESS TERRIBLE?



CLARITY

OPTIONALS, ENUMS, TYPEALIAS, DESIGNATED INIT

SAFETY

TYPES, GENERICS, DESIGNATED INIT

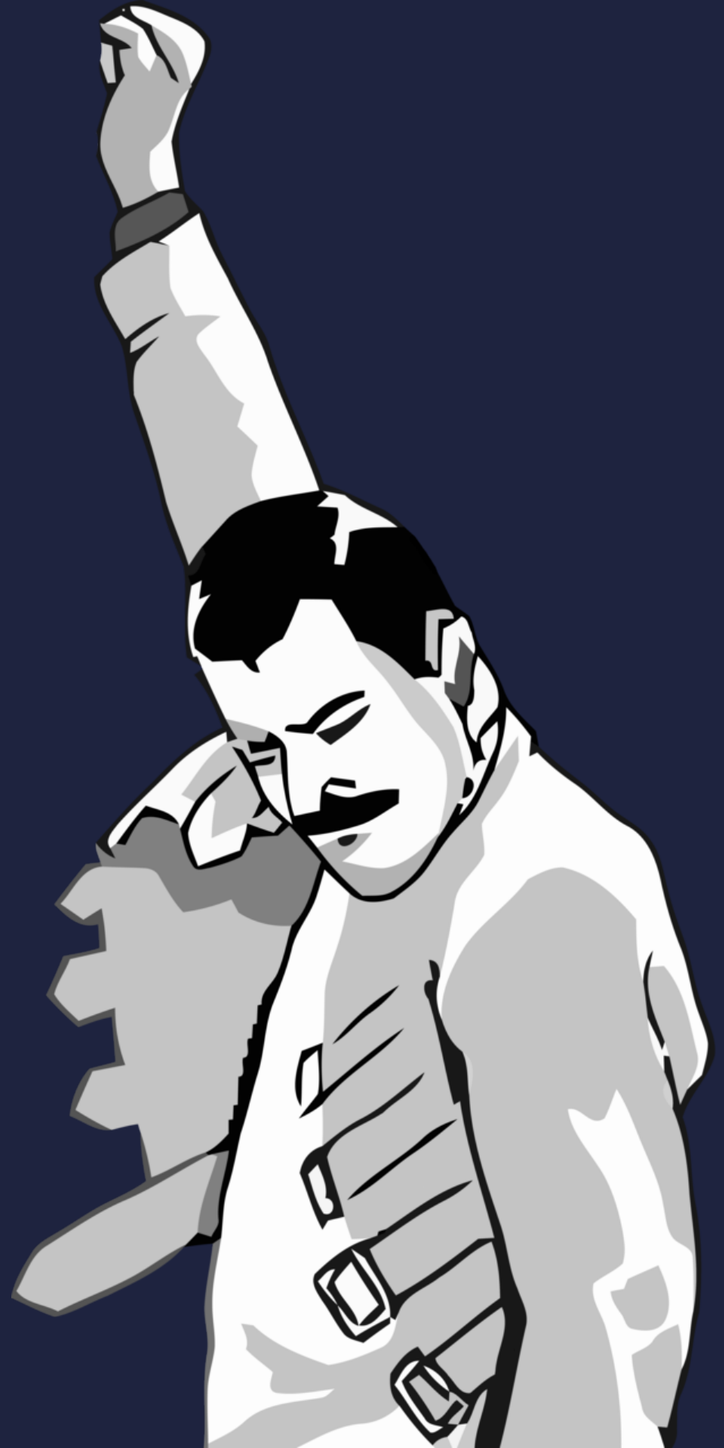
SWIFTNESS

ENUMS, TYPEALIAS, OPTIONALS

FUNCTIONAL

SAVING, FETCHING, MORE

github.com/jessesquires/JSQCoreDataKit



Thank you!

QUESTIONS?

JESSESQUIRES.COM • @JESSE_SQUIRES • GITHUB/JESSESQUIRES