ETSIT UPM

dit UPM

Desarrollo de Apps para iOS

Persistencia - Core Data

IWEB,LSWC 2013-2014
Santiago Pavón

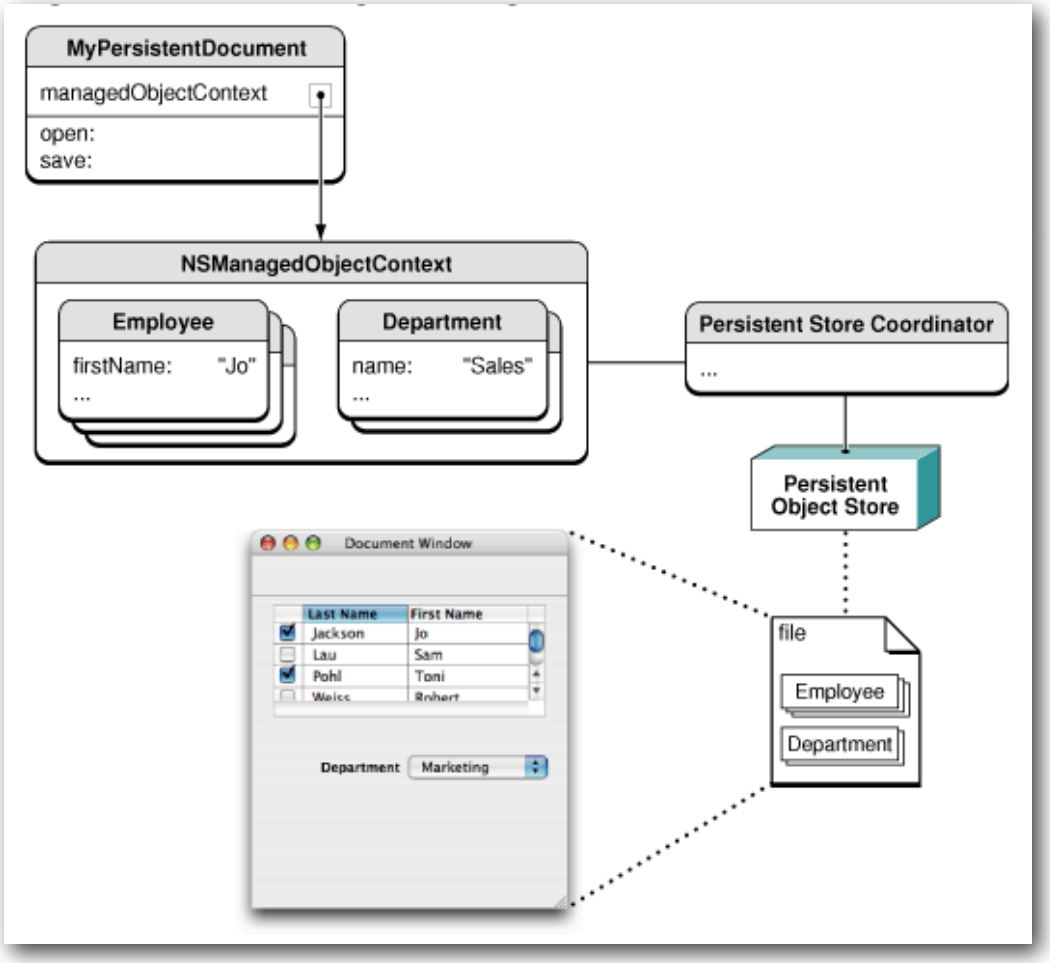
ver: 2012.09.19 p1

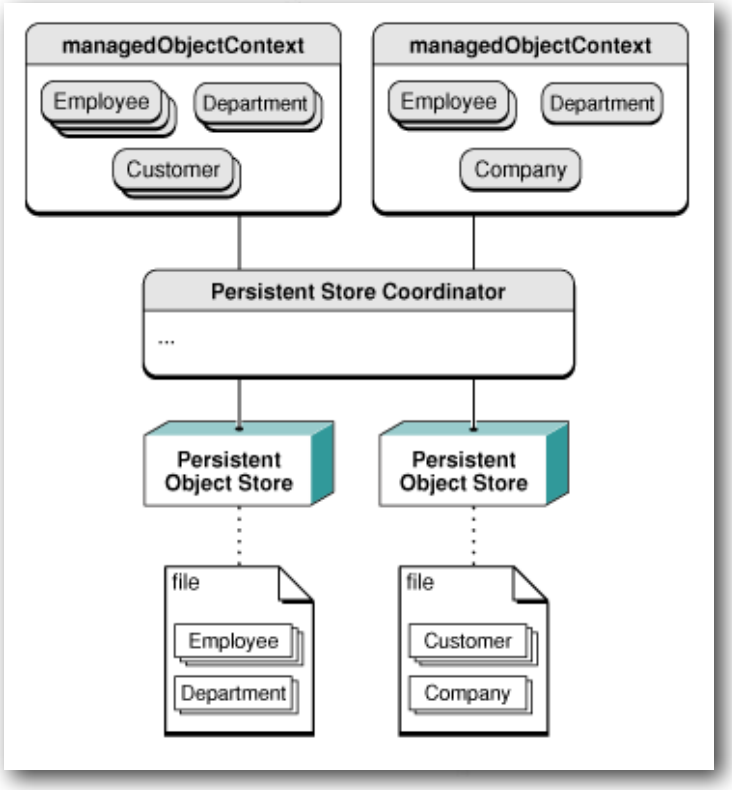
¿Qué es Core Data?

- Framework que proporciona persistencia de datos.
  - Para salvar datos de app reales (no sólo datos triviales).
  - Optimiza el uso de la memoria.
  - Mejorar el tiempo de respuesta de las apps.
  - Evita que escribamos código repetitivo.
  - Diseño visual de los modelos de datos.
  - Soporte para migrar datos a nuevas versiones.
  - Los datos se almacenan normalmente usando una base de datos SQLite.
    - Alternativas: ficheros binarios, memoria.
    - Se manejan usando un contexto que oculta los detalles.
- Es un framework muy grande y complicado de aprender.

Arquitectura

- Consultar las guías de Core Data.





- Managed Object Model:
  - Contiene la definición de los objetos que se guardan en la base de datos: sus atributos y relaciones.
  - Los objetos guardados en la base de datos se llaman entidades.
- Managed Object Context:
  - El objeto usado para crear, borrar, obtener, modificar o salvar los objetos de la aplicación y que tienen su almacenamiento en el store.
  - Accede al store para realizar las operaciones necesarias en la base de datos y nos proporciona objetos que reflejan los datos almacenados.
- Persistent Store Coordinator:
  - Coordina el acceso al store(s) desde el context(s).
- Persistent Object Store:
  - Accede sistema donde se salvan los datos (ej: una base de datos).

Crear una app que use Core Data

- Se necesita importar el Framework CoreData, codificar las sentencias de manejo de objetos para gestionar el contexto, coordinar el almacenamiento, etc.
- Podemos simplificar estas tareas:
  - Crear un proyecto y marcar "Use Core Data".
    - Usando la plantilla Master-Detail Application.
    - En AppDelegate se ha credo la propiedad `managedObjectContext`.
  - Crear en la aplicación un `UIManagedDocument`.
    - Acceder a su propiedad `managedObjectContext`.
    - Interesante para soportar iCloud.

Partiendo de la Plantilla Master-Detail

- Crear la nueva aplicación partiendo de la plantilla Master-Detail Application.
  - Marcar la casilla: Use Core Data.
  - Marcar también las casillas para usar Storyboard y ARC.
- Si nuestra aplicación no va a seguir la plantilla Master-Detail, borrar (a la papelera) los siguientes ficheros generados:
    - `MasterViewController` y `DetailViewController` (`.h` y `.m`)
  - Crearemos unos ficheros `MasterViewController.h` y `.m` nuevos.
    - Añadir al .h la siguiente propiedad (y sintetizarla en el .m).

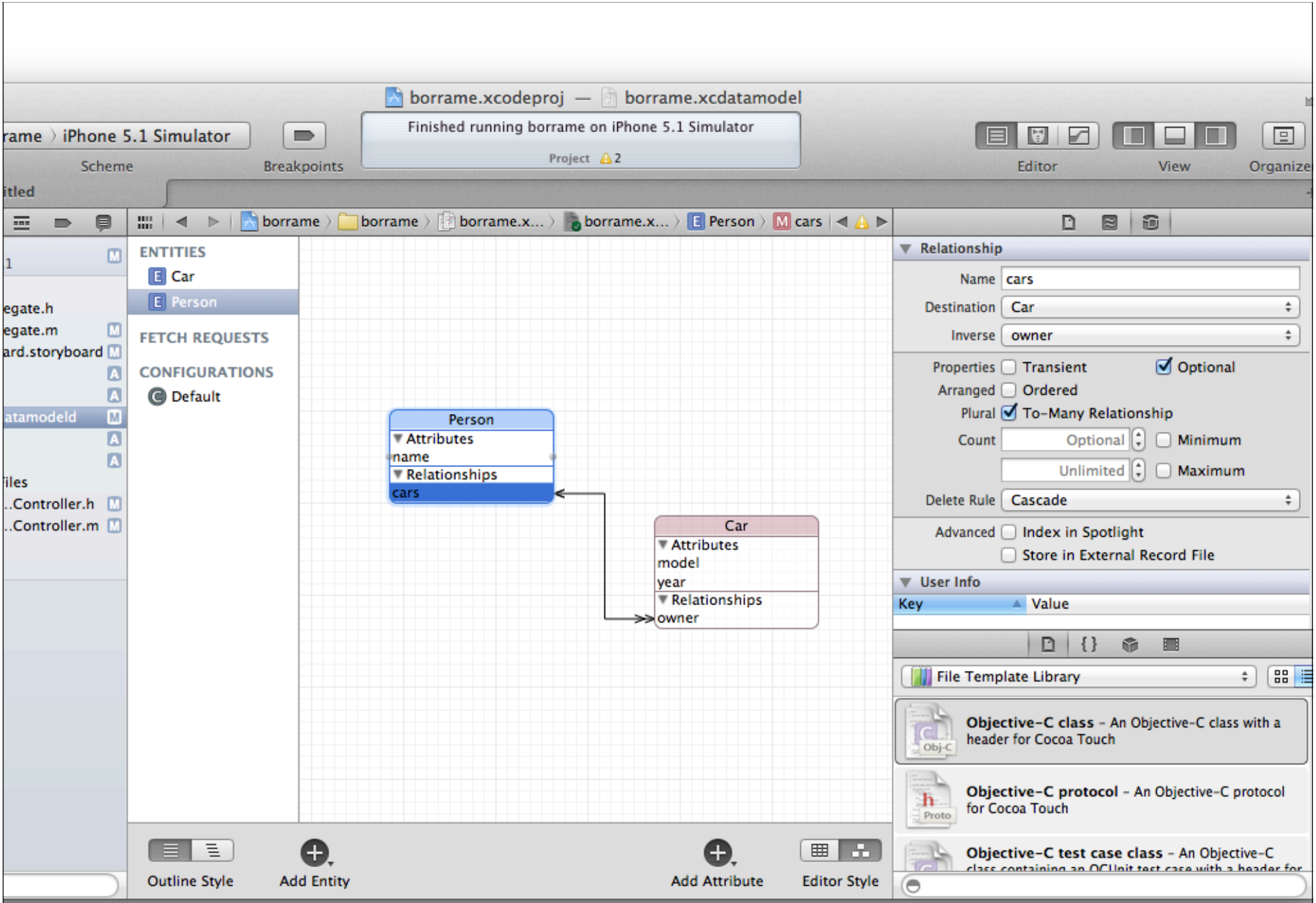
```
@property (nonatomic,strong)
        NSManagedObjectContext* managedObjectContext;
```

    - El valor de esta propiedad se asigna desde el delegado de la aplicación.
      - Se necesita en todos los métodos relacionados con Core Data.
      - Deberemos propagarla por todos los rincones de la aplicación donde se use Core Data.

- Si hemos borrado/sustituido los ficheros MasterViewController y DetailViewController,
  - debemos actualizar el fichero Storyboard borrando las escenas viejas y añadiendo las nuevas.
  - debemos editar el fichero .xcdatamodel (que contiene los modelos).
    - Borrar las entidades definidas.
      - Crear nuevas entidades.

Crear Modelos

- El fichero .xcdatamodel contiene la definición de las entidades (los modelos de nuestros datos).
  - Los editaremos añadiendo sus atributos y relaciones.
- Para crear nuevas entidades, atributos y relaciones:
  - pulsar los botones/menús Add Entity, Add Attribute, Add Relationship, y editar sus características usando el inspector.
- Cada vez que se modifique algo en los modelos, la aplicación no podrá leer los datos ya almacenados según el modelo anterior.
  - Debe gestionarse el cambio de versión de modelo.
  - o puede desinstalarse la aplicación vieja del terminal (o simulador) e instalar la nueva versión.



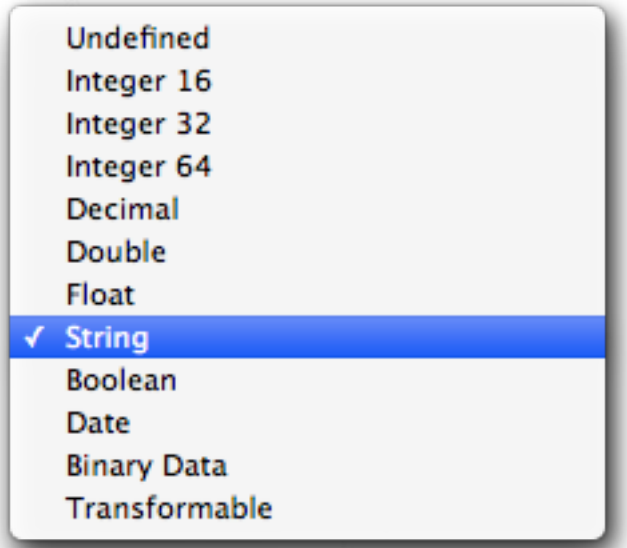
Crear Modelos - Entidades

- Los objetos creados según los modelos definidos son las entidades.
- Las entidades (por defecto) son de la clase **`NSManagedObject`**.
  - Accederemos y modificaremos sus propiedades usando KVC.

```
// Crear entidad para una nueva persona:
NSManagedObject *person = [NSEntityDescription
      insertNewObjectForEntityForName:@"Person"
      inManagedObjectContext:context];

// Modificar el nombre de la persona:
[person setValue:@"Pedro" forKey:@"name"];

// Acceder al nombre de la persona:
NSString * name = [person valueForKey:@"name"];
```

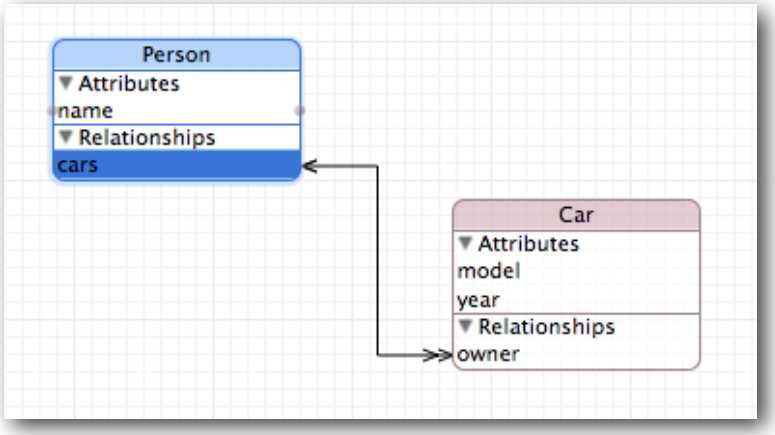
Crear Modelos - Atributos

- Los atributos de las entidades pueden ser de varios tipos.
- Asignaremos el tipo de cada atributo desde el inspector.
- Estos valores indican como se almacena en la base de datos el valor de los atributos, pero nosotros usaremos siempre clases en el acceso a estos valores usando KVC.



Crear Modelos - Relaciones

- Una relación es una conexión entre dos entidades.
- Tipos de relaciones:
  - one-to-one
  - one-to-many
  - many-to-many



  - Ejemplo one-to-many:
    - Una persona tiene muchos coches.
    - Un coche pertenece a una persona.
- El tipo de las relaciones se llama cardinalidad y puede ser to-one o to-many
  - En el editor de modelos, las relaciones se representan uniendo las entidades con flechas de punta sencilla o doble.

- Al definir una relación hay que especificar:
  - **Name**: El nombre de la relación.
  - **Destination**: La entidad destino de la relación.
  - **Plural**: La cardinalidad de la relación (to-one, to-many).
  - **Inverse**: La relación inversa.
    - La relación definida en la entidad destino que apunta a la entidad origen.
    - Es muy conveniente crear siempre la relación inversa ya que Code Data la usa para comprobaciones de consistencia.
  - **Delete Rule**: Que hacer cuando se borra la entidad origen de una relación.
    - **No action**: No hacer nada.
    - **Nullify**: Poner a null el valor de la relación inversa de la entidad destino.
    - **Cascade**: Borrar las entidades destino.
    - **Deny**: No puede borrarse mientras existan entidades destino con relaciones inversas apuntado a la entidad origen.

Uso

- Todos los sitios del programa que deseen usar los datos gestionados por Core Data deben usar el mismo objeto de contexto NSManagedObjectContext.
  - En nuestro ejemplo (hemos partido de la plantilla Master-Detail), el objeto con el contexto se ha creado en el delegado de la aplicación y se lo ha guardado en una propiedad llamada managedObjectContext.
  - Tenemos que propagar este único contexto por todos los objetos (view controllers, modelos, ...) que lo necesiten.

- Para crear un nuevo objeto, llamaremos al método:

```
NSManagedObject * obj =
  [NSEntityDescription
     insertNewObjectForEntityForName:entityName
     inManagedObjectContext:context];
```

  - nos devuelve una instancia de NSManagedObject.

- Para acceder a los campos de los objetos NSManagedObject usaremos los métodos:

  –valueForKey:
  –setValue:forKey:
  –mutableSetValueForKey: (para las relaciones to-many)

- Para borrar un objeto llamaremos al método **deleteObject:** del objeto contexto.

- Para almacenar en la base de datos los objetos que tenemos en memoria llamaremos al método **save** del objeto contexto.

- Para recuperar/buscar objetos en la base de datos construiremos un objeto NSFetchRequest.

```
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
```

  - Luego configuraremos este objeto:
    - indicar el tipo de entidad que estamos buscando.

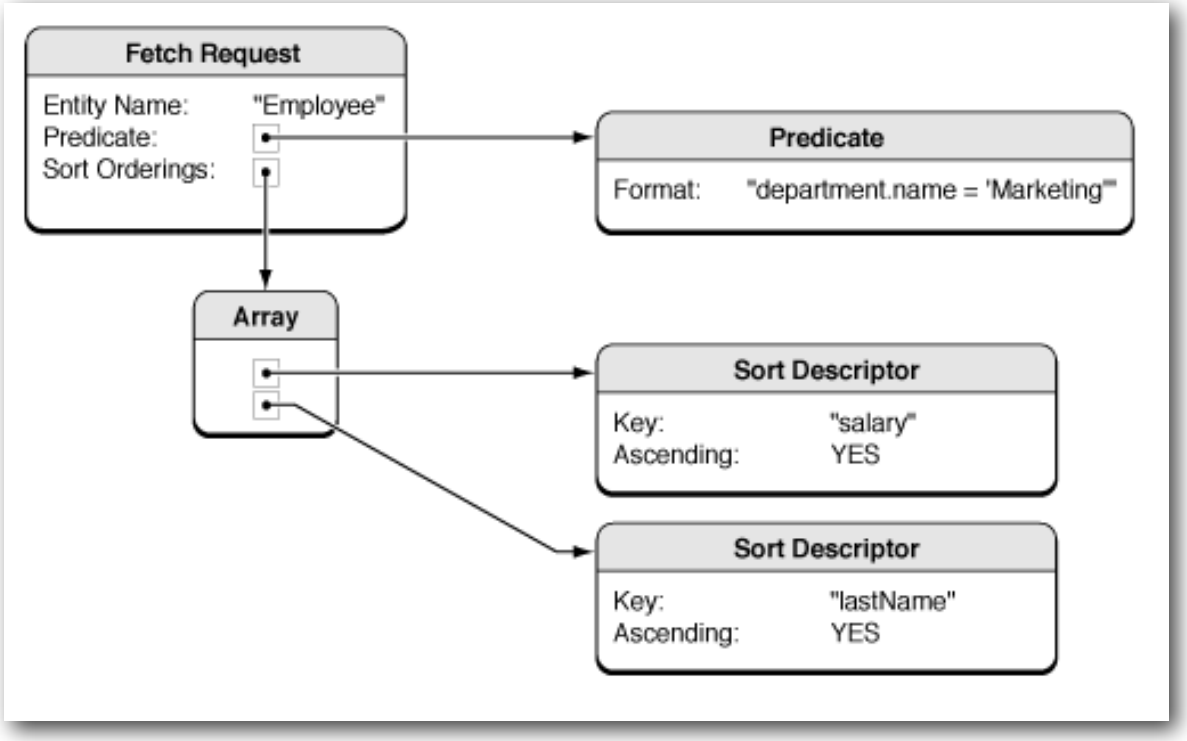
```
NSEntityDescription *entity =
   [NSEntityDescription entityForName:@"Person"
                inManagedObjectContext:context];

[fetchRequest setEntity:entity];
```

    - proporcionar predicados que indiquen las condiciones que deben cumplir los objetos que nos interesan.
      - Si no se especifican predicados se devuelven todos los objetos existentes.
    - limitar el número de objetos devueltos, indicar un offset.
    - indicar como ordenar los resultados.

  - Y finalmente le decimos al objeto contexto que realice la búsqueda y recuperación de los objetos.

```
NSError *error;
NSArray *fetchedObjects = [context executeFetchRequest:fetchRequest
                                                 error:&error];
```

    - Inicialmente puede que no se recuperen los datos reales, sino fault objects. Si es el caso, al acceder a los atributos se recuperan automáticamente los datos reales de la base de datos.



Copiado de la documentación de Apple

Ejemplo

- Añadir el siguiente código en el método didFinishLaunchingWithOptions: de AppDelegate.h.
  - Se crea una persona llamada Pedro.
  - Se crean dos coches.
  - Se indica que los coches son de Pedro.
  - Se salva todo en la base de datos.
  - Se recuperan todas las personas de la bases de datos.
    - Se sacan logs de la información recuperada.
- Nota: cada vez que se ejecute la aplicación se crea otro Pedro y otros dos coches nuevos.

```objc
NSManagedObjectContext *context = self.managedObjectContext;


// Crear una Persona llamada Pedro

NSManagedObject *person = [NSEntityDescription
                                    insertNewObjectForEntityForName:@"Person"
                                            inManagedObjectContext:context];

[person setValue:@"Pedro" forKey:@"name"];


// Crear dos coches: un Seat Leon del año 2005 y un Fiat Punto del 2000:

NSManagedObject *car1 = [NSEntityDescription
                                    insertNewObjectForEntityForName:@"Car"
                                            inManagedObjectContext:context];
[car1 setValue:@"Seat Leon" forKey:@"model"];
[car1 setValue:[NSNumber numberWithInt:2005] forKey:@"year"];

NSManagedObject *car2 = [NSEntityDescription
                                    insertNewObjectForEntityForName:@"Car"
                                            inManagedObjectContext:context];
[car2 setValue:@"Fiat Punto" forKey:@"model"];
[car2 setValue:[NSNumber numberWithInt:2000] forKey:@"year"];
```

```
   // Pedro es el dueño de los dos coches

[car1 setValue:person forKey:@"owner"];
[car2 setValue:person forKey:@"owner"];

/* No es necesario porque ya se ha hecho con las dos sentencias anteriores
  NSMutableSet * cars = [person mutableSetValueForKey:@"cars"];
   [cars addObject:car1];
   [cars addObject:car2];
*/



// Salvar datos de la memoria en la base de datos

NSError *error;
if (![context save:&error]) {
    NSLog(@"Whoops, couldn't save: %@", [error localizedDescription]);
    abort();
}
```

```
   // Recuperar todas las personas de la bases de datos:

NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];


NSEntityDescription *entity = [NSEntityDescription entityForName:@"Person"
                                          inManagedObjectContext:context];
[fetchRequest setEntity:entity];


NSArray *fetchedObjects = [context executeFetchRequest:fetchRequest
                                                 error:&error];


// Sacar unas trazas por la consola:

for (NSManagedObject * person in fetchedObjects) {
    NSLog(@"Coches de %@:", [person valueForKey:@"name"]);

    NSMutableSet * cars = [person mutableSetValueForKey:@"cars"];
    for (NSManagedObject * car in cars) {
       NSLog(@"   %@ del %@", [car valueForKey:@"model"],
                              [car valueForKey:@"year"]);
    }
}
```

Subclases de Managed Objects

- Las entidades por defecto son de la clase **`NSManagedObject`**.
- Conviene crear subclases de `NSManagedObject` para poder definir propiedades para cada uno de los atributos y evitar el uso de KVC.
  - El código queda más limpio.

```
person.name = @"Pedro";
```

  - Más fácil detectar errores al usar propiedades en lugar de NSStrings.
    - Xcode nos marcará los errores en el editor.

- Xcode puede generar automáticamente los ficheros con los modelos, es decir, los ficheros con las subclases de NSManagedObject.
  - Seleccionar las entidades de las que queremos generar ficheros con las subclases.
  - Invocar:
    - **`File > New > File > Core Data > NSManagedObject subclass`**
    - Podemos indicar si queremos generar valores primitivos para las propiedades que representan los atributos, o usar siempre objectos.
      - Ejemplo: una propiedad numerica puede ser de tipo int16_t o NSNumber.
- Nota: Si generamos modelos que usan modelos que aun no se han generado, pueden aparecer referencias a NSManagedObject en vez de referencias a las nuevas subclases.
  - Se soluciona regenerando otra vez los mismos ficheros.

Crear Person.h.m y Car.h.m

- Seleccionar en el editor de modelos las entidades People y Car.
- Ejecutar:
  - **`File > New > File > Core Data > NSManagedObject subclass`**
- Hacerlo dos veces para eliminar referencias a NSManagedObject debido a que cuando se crearon aun no se habian generado todos los nuevos ficheros con las nuevas clases.
- Se crearán los ficheros Person.h, Peson.m, Car.h y Car.m.

Person.h

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

@class Car;

@interface Person : NSManagedObject

@property (nonatomic, retain) NSString * name;
@property (nonatomic, retain) NSSet *cars;
@end

@interface Person (CoreDataGeneratedAccessors)

- (void)addCarsObject:(Car *)value;
- (void)removeCarsObject:(Car *)value;
- (void)addCars:(NSSet *)values;
- (void)removeCars:(NSSet *)values;

@end
```

Person.m

```
  #import "Person.h"
#import "Car.h"


@implementation Person

@dynamic name;
@dynamic cars;

@end
```

@dynamic indica que no se generará la implementación de los métodos de acceso.

Se interceptan las llamadas a los métodos sin implementar y se resuelven usando los métodos KVO.

Car.h

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

@class Person;

@interface Car : NSManagedObject

@property (nonatomic, retain) NSString * model;
@property (nonatomic, retain) NSNumber * year;
@property (nonatomic, retain) Person *owner;

@end
```

Car.m

```
  #import "Car.h"
#import "Person.h"


@implementation Car

@dynamic model;
@dynamic year;
@dynamic owner;

@end
```

Rehacemos el Ejemplo

- Añadir el siguiente código en el método didFinishLaunchingWithOptions: de AppDelegate.h.
  - Se crea una persona llamada Pedro.
  - Se crean dos coches.
  - Se indica que los coches son de Pedro.
  - Se salva todo en la base de datos.
  - Se recuperan todas las personas de la bases de datos.
    - Se sacan logs de la información recuperada.
- Nota: cada vez que se ejecute la aplicación se crea otro Pedro y otros dos coches nuevos.

```objc
NSManagedObjectContext *context = self.managedObjectContext;


// Crear una Persona llamada Pedro

Person *person = [NSEntityDescription insertNewObjectForEntityForName:@"Person"
                                                inManagedObjectContext:context];

person.name = @"Pedro";


// Crear dos coches: un Seat Leon del año 2005 y un Fiat Punto del 2000:

Car *car1 = [NSEntityDescription insertNewObjectForEntityForName:@"Car"
                                                inManagedObjectContext:context];
car1.model = @"Seat Leon";
car1.year  = [NSNumber numberWithInt:2005];


Car *car2 = [NSEntityDescription insertNewObjectForEntityForName:@"Car"
                                                inManagedObjectContext:context];
car2.model = @"Fiat Punto";
car2.year  = [NSNumber numberWithInt:2000];
```

```objc
// Pedro es el dueño de los dos coches

car1.owner = person;
// car2.owner = person;

// [person addCarsObject:car1];
[person addCarsObject:car2];




// Salvar datos de la memoria en la base de datos

NSError *error;
if (![context save:&error]) {
    NSLog(@"Whoops, couldn't save: %@", [error localizedDescription]);
    abort();
}
```

```objc
// Recuperar todas las personas de la bases de datos:

NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];


NSEntityDescription *entity = [NSEntityDescription entityForName:@"Person"
                                          inManagedObjectContext:context];
[fetchRequest setEntity:entity];


NSArray *fetchedObjects = [context executeFetchRequest:fetchRequest
                                                 error:&error];


// Sacar unas trazas por la consola:

for (Person * person in fetchedObjects) {
    NSLog(@"Coches de %@:", person.name);

        for (Car * car in person.cars) {
            NSLog(@"   %@ del %@", car.model, car.year);
            NSLog(@"   Verificar dueño: %@", car.owner.name);
        }
    }
}
```

car.owner.name

Usar Categorias

- Conviene no modificar manualmente los ficheros autogenerados para no perder los cambios introducidos en caso de que necesitemos volver a crearlos.
  - Podemos usar categorías para modificar las subclases generadas
  - Las categorias nos permiten añadir más métodos a las clases generadas sin tener que tocar los ficheros creados automáticamente.

- Ejemplo: Crear una categoría de Car para definir un método de clase que cree instancias de Car:
  - Car+Create.h

```
#import "Car.h"
@interface Car (CarCreate)
+ (Car *)    carWithModel:(NSString*)model
                 andYear:(int)year
   inManagedObjectContext:(NSManagedObjectContext *)context;
@end
```

  - Car+Create.m

```
#import "Car+CarCreate.h
@implementation Car (CarCreate)
+ (Car *)    carWithModel:(NSString*)model
                 andYear:(int)year
   inManagedObjectContext:(NSManagedObjectContext *)context {
    Car * car = [NSEntityDescription insertNewObjectForEntityForName:@"Car"
                                       inManagedObjectContext:context];
    car.model = model;
    car.year  = [NSNumber numberWithInt:year];
    return car;
}

@end
```

  - Uso:

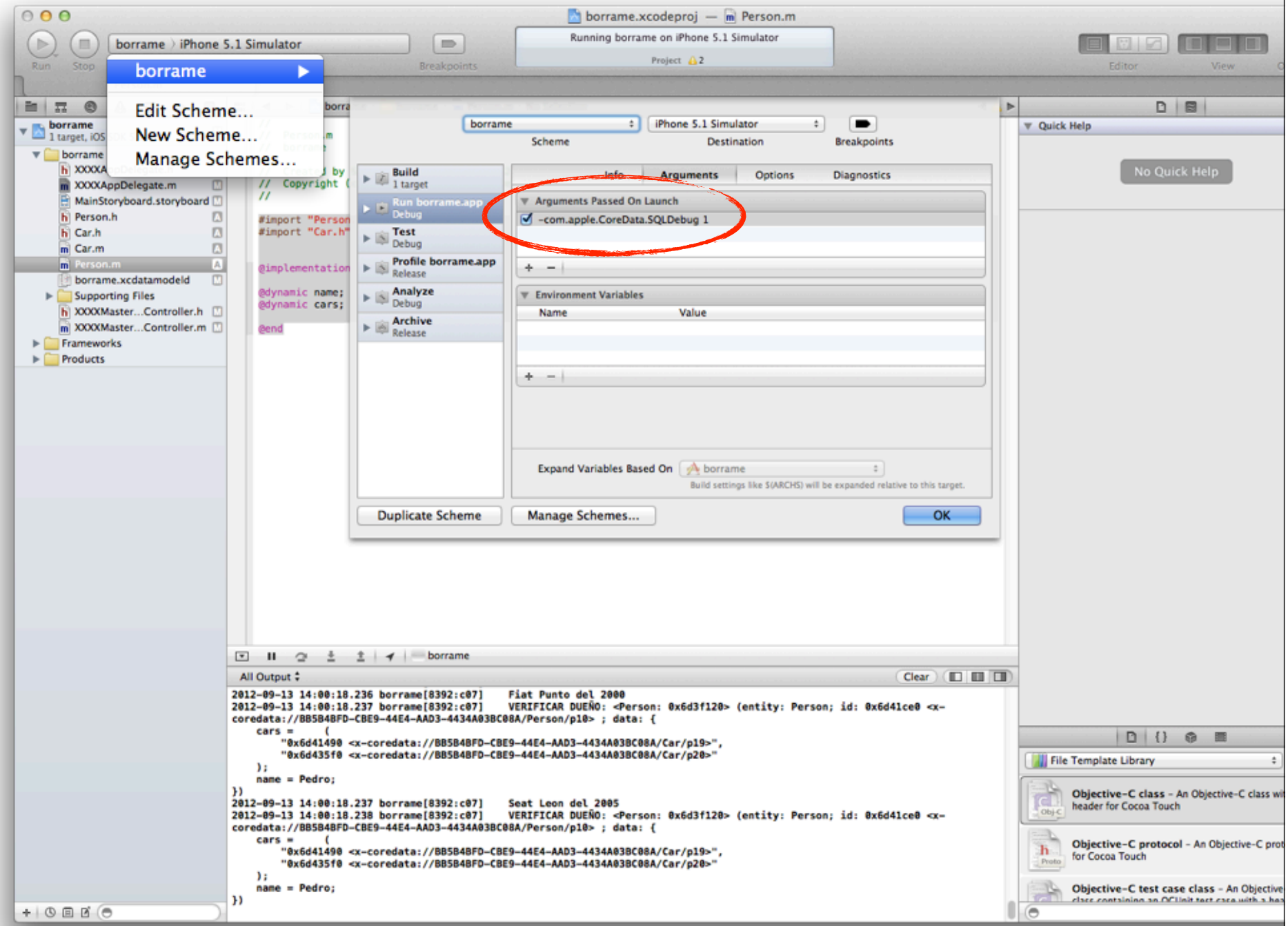
```
 #import "Car+CarCreate.h
Car *car3 = [Car carWithModel:@"Opel" andYear:1999 inManagedObjectContext:context];
```

Truco: Ver Sentencias SQL

- Para depurar, puede ser interesante ver que sentencias SQL se están ejecutando cuando corremos la aplicación.
  - Editar el scheme Run, añadir el siguiente argumento:

```
-com.apple.CoreData.SQLDebug 1
```

- Ahora se generarán logs en la consola de depuacion



Crear una Table View

- Opción para mostrar los datos de la base de datos en una Table View:
  - Recuperar (NSFetchRequest) todos los datos de la base de datos y meterlos en un array.
  - Implementar los métodos del data source de la Table View (lo típico que se hace siempre) para que acceda al array.
  - Esto es fácil de hacer, pero:
    - gasta mucha memoria recuperando todos los datos.
      - podrían ser muchos.
    - se recupera informaciín que puede que nunca veamos
      - si no hago scroll en la tabla.
    - no es eficiente.

- Mejora:
  - No recuperar todos los datos de golpe.
    - Usar las opciones de límite y offset de NSFetchRequest para recuperar sólo los datos que vamos a motrar en cada momento.

- Si modificamos, añadimos o borrarmos datos:
  - Hay que gestionar el refresco de la Tabla View para mostrar siempre los datos actualizados.
  - Ejecutaremos nuevos Fetch Requests.

- Otra opción mejor:
  - Usar **`NSFetchedResultsController`**.

El Controlador `NSFetchedResultsController`

- Nos simplifica la tarea de mostrar en una Table View los datos obtenidos con Fetch Request.
  - Es eficiente.
  - Optimiza el uso de memoria.
  - Puede usar una cache para evitar repetir calculos.
  - Gestiona las secciones de la tabla.
  - Puede monitorizar el cambio en los datos para realizar tareas de refresco.
  - ...
- La plantilla Master-Detail genera código que usa este controlador.
  - Pueden copiarse los métodos de esta plantilla para usar en nuestras app.
  - En la documentación también se proporciona código que podemos copiar directamente.

Examinando la Plantilla Master-Detail.

- La Clase MasterViewController es una UITableViewController
  - Muestra las entidades definidas en el modelo en la table view.
    - Las entidades son eventos con una fecha.
- Para manejar el acceso de las entidades existentes se usa un objeto NSFetchedResultsController.
  - Se define como una propiedad en la interface.h

```
@property (strong,nonatomic)
NSFetchedResultsController *fetchedResultsController;
```

  - El método getter se ha reescrito para crear el objeto `NSFetchedResultsController` de forma perezosa.
    - Si no existe lo crea, y si ya existe lo devuelve.

```
    @property (strong, nonatomic) NSFetchedResultsController *fetchedResultsController;


- (NSFetchedResultsController *)fetchedResultsController
{
    if (_fetchedResultsController != nil) {
        return _fetchedResultsController;
    }

    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    // Edit the entity name as appropriate.
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Event"
                                               inManagedObjectContext:self.managedObjectContext];
    [fetchRequest setEntity:entity];

    // Set the batch size to a suitable number.
    [fetchRequest setFetchBatchSize:20];

    // Edit the sort key as appropriate.
    NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:@"timeStamp" ascending:NO];
    NSArray *sortDescriptors = @[sortDescriptor];
    [fetchRequest setSortDescriptors:sortDescriptors];

    // Edit the section name key path and cache name if appropriate.
    // nil for section name key path means "no sections".
    NSFetchedResultsController *aFetchedResultsController = [[NSFetchedResultsController alloc]
           initWithFetchRequest:fetchRequest
           managedObjectContext:self.managedObjectContext
             sectionNameKeyPath:nil
                      cacheName:@"Master"];
    aFetchedResultsController.delegate = self;
    self.fetchedResultsController = aFetchedResultsController;

   NSError *error = nil;
    if (![self.fetchedResultsController performFetch:&error]) {
         // Replace this implementation with code to handle the error appropriately.
         // abort() causes the application to generate a crash log and terminate. You should not use
         // this function in a shipping application, although it may be useful during development.
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        abort();
    }

    return _fetchedResultsController;
}
```

- El objeto `NSFetchedResultsController` se crea pasando en el método de inicialización los siguientes datos:
  - Un objeto `FetchRequest` a usar para obtener datos.
    - Indicando la entidad, la forma de ordenar los datos, la cantidad de datos a recuperar en cada petición.
  - El Managed Object Context que nos ha pasado el delegado de la aplicación.
  - KeyPath que devuelve el nombre de la sección de la table view a la que pertenece cada managed object recuperado.
    - Un KeyPath es una cadena de nombres de atributos separados por puntos
    - Usar `nil` si no dividimos en secciones.
    - El valor usado de KeyPath debe encajar con el criterio de ordenación usado en el objeto FetchRequest (sortDescriptors).
      - Los resultados recuperados deben venir ordenados por sección. Primero las datos de la primera sección, luego los de la segunda, etc.
  - Nombre de la cache (o nil si no la usamos) donde se almacenan resultados de tareas repetitivas para no realizarlas más de una vez.
    - Cuidado: no puede usarse el mismo nombre de cache en varios controladores.

```
    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];


// Edit the entity name as appropriate.
NSEntityDescription *entity =
         [NSEntityDescription entityForName:@"Event"
                     inManagedObjectContext:self.managedObjectContext];
[fetchRequest setEntity:entity];


// Set the batch size to a suitable number.
[fetchRequest setFetchBatchSize:20];


// Edit the sort key as appropriate.
NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:@"timeStamp"
                                                               ascending:NO];
NSArray *sortDescriptors = @[sortDescriptor];
[fetchRequest setSortDescriptors:sortDescriptors];



// Edit the section name key path and cache name if appropriate.
// nil for section name key path means "no sections".
NSFetchedResultsController *aFetchedResultsController =
     [[NSFetchedResultsController alloc] initWithFetchRequest:fetchRequest
                                         managedObjectContext:self.managedObjectContext
                                           sectionNameKeyPath:nil
                                                    cacheName:@"Master"];
```

- El objeto NSFetchedResultsController puede usar un objeto delegado.
  - Al delegado se le informa de las modificaciones realizadas en los datos para que refresque lo que sea necesario en la Table View.

```
aFetchedResultsController.delegate = self;
```

    - es decir, usa al propio MasterViewController como delegado.

- Finalmente se invoca **`performFetch:`** para ejecutar el FetchRequest con el que se creó el objeto NSFetchedResultsController.
  - Tras invocar este método ,ya se puede acceder a los objetos recuperados de la base de datos.

```
NSError *error = nil;
if (![self.fetchedResultsController performFetch:&error]) {
    NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
    abort();
}
```

- NSFetchedResultsController vigila de los cambios que ocurren en los datos gestionados por Core Data, y si esto ocurre informa a su delegado.
  - Cuando al delegado se le informa de que se han añadido o borrado secciones:
    - El delegado debe añadir o borrar las secciones de la table view.
  - Cuando al delegado se le informa sobre objetos añadidos, borrados, actualizados o movidos:
    - El delegado debe añadir, borrar, reconfigurar o mover las filas de la table view.

```
- (void)controllerWillChangeContent:(NSFetchedResultsController *)controller {
    [self.tableView beginUpdates];
}

- (void)controller:(NSFetchedResultsController *)controller didChangeSection:(id <NSFetchedResultsSectionInfo>)sectionInfo
           atIndex:(NSUInteger)sectionIndex forChangeType:(NSFetchedResultsChangeType)type {
    switch(type) {
        case NSFetchedResultsChangeInsert:
            [self.tableView insertSections:[NSIndexSet indexSetWithIndex:sectionIndex]
                          withRowAnimation:UITableViewRowAnimationFade];
            break;

        case NSFetchedResultsChangeDelete:
            [self.tableView deleteSections:[NSIndexSet indexSetWithIndex:sectionIndex]
                          withRowAnimation:UITableViewRowAnimationFade];
            break;
    }
}

- (void)controller:(NSFetchedResultsController *)controller didChangeObject:(id)anObject
       atIndexPath:(NSIndexPath *)indexPath forChangeType:(NSFetchedResultsChangeType)type
      newIndexPath:(NSIndexPath *)newIndexPath {
    UITableView *tableView = self.tableView;

    switch(type) {
        case NSFetchedResultsChangeInsert:
            [tableView insertRowsAtIndexPaths:@[newIndexPath] withRowAnimation:UITableViewRowAnimationFade];
            break;

        case NSFetchedResultsChangeDelete:
            [tableView deleteRowsAtIndexPaths:@[indexPath] withRowAnimation:UITableViewRowAnimationFade];
            break;

        case NSFetchedResultsChangeUpdate:
            [self configureCell:[tableView cellForRowAtIndexPath:indexPath] atIndexPath:indexPath];
            break;

        case NSFetchedResultsChangeMove:
            [tableView deleteRowsAtIndexPaths:@[indexPath] withRowAnimation:UITableViewRowAnimationFade];
            [tableView insertRowsAtIndexPaths:@[newIndexPath] withRowAnimation:UITableViewRowAnimationFade];
            break;
    }
}

- (void)controllerDidChangeContent:(NSFetchedResultsController *)controller {
    [self.tableView endUpdates];
}
```

- Se accede al objeto FetchedResultsController desde los métodos del DataSource de la Table View para saber:
  - el número de secciones y filas de la table view
  - el objeto a mostrar en un indexPath de la table view.

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return [[self.fetchedResultsController sections] count];
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section {
    id <NSFetchedResultsSectionInfo> sectionInfo =
          [[self.fetchedResultsController sections] objectAtIndex:section];
    return [sectionInfo numberOfObjects];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"Cell"];
    [self configureCell:cell atIndexPath:indexPath];
    return cell;
}

- (void)configureCell:(UITableViewCell *)cell atIndexPath:(NSIndexPath *)indexPath {
    NSManagedObject *object = [self.fetchedResultsController
                                              objectAtIndexPath:indexPath];
    cell.textLabel.text = [[object valueForKey:@"timeStamp"] description];
}
```

Predicados

- A los objetos NSFetchRequest puede asignárseles un predicado para filtrar los objetos que deben devolver.

```
NSFetchRequest *fetchRequest = ...;
NSPredicate *pred = [NSPredicate
   predicateWithFormat:@"name == %@",unNombre];
[fetchRequest setPredicate:pred];
```

- Con un predicado pueden expresarse muchos tipos de condiciones.
  - Consultar detalles en la guía *Predicate Format String Format*.
  - Ejemplos:

```
@"name == %@"

@"name BEGINSWITH %@"

@"name CONTAINS[c] %@"

@"(age > %d) AND (name like %@)"
```

Precarga Inicial de Datos

- La primera vez que ejecutemos la aplicación el fichero de la base de datos SQLite no existirá.
  - Core Data lo creará vacío.
- Si queremos que la base de datos contenga ciertos datos iniciales por defecto, este es el momento de proporcionar estos datos iniciales.
- Opciones:
  - Obtener los datos iniciales de algun sitio:
    - de unos ficheros de datos en el main bundle.
    - descargando los datos de internet.
    - y meterlos en la nueva base de datos.
  - Tener ya en el main bundle un fichero SQLite con los datos iniciales.
    - y copiar ese fichero en el directorio de documentos para que lo use CoreData.
      - *Para crear un fichero SQLite con los datos inciales que sea compatible con nuestra app iOS, podemos crear una aplicacion OS X usando los mismos ficheros de modelo (.xcdatamodel, subclases de NSManagedObject) que usamos para la app iOS. La única utilidad de esta aplicación OS X es rellenar el fichero SQLite con los datos iniciales.*

Usar un Fichero SQLite Preinicializado

- Tarea:
  - Tenemos en el main bundle un fichero SQLite precargado con los datos iniciales por defecto que usará nuestra aplicación iOS, y lo queremos copiar en el directorio de documentos para que lo use Core Data como base de datos inicial.
- ¿Como se copia ese fichero?
  - Editamos en `AppDelegate.m` el método `persistentStoreCoordinator`.
  - En este método, la variable `storeURL` contienee el path al fichero SQLite que usará CoreData como almacenamiento.
  - Hay que comprobar si existe el fichero SQLite referenciado por `storeURL`.
  - Si no existe, copiaremos el fichero SQLite que tenemos en el bundle en el sitio indicado por `storeURL`.

```
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator
{
    if (_persistentStoreCoordinator != nil) {
        return _persistentStoreCoordinator;
    }

    NSURL *storeURL = [[self applicationDocumentsDirectory] URLByAppendingPathComponent:@"Prueba.sqlite"];

    if (![[NSFileManager defaultManager] fileExistsAtPath:[storeURL path]]) {
        NSURL *preloadURL = [NSURL fileURLWithPath:[[NSBundle mainBundle]
                                                    pathForResource:@"DatosPrecargados"
                                                    ofType:@"sqlite"]];
        NSError* err = nil;
        if (![[NSFileManager defaultManager] copyItemAtURL:preloadURL
                                                    toURL:storeURL
                                                    error:&err]) {
            NSLog(@"No puedo copiar la base de datos inicial: %@",
                  [err localizedDescription]);
        }
    }

    NSError *error = nil;
    _persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
                                   initWithManagedObjectModel:[self managedObjectModel]];
    if (![_persistentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType configuration:nil
                                  URL:storeURL options:nil error:&error]) {
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        abort();
    }

    return _persistentStoreCoordinator;
}
```

Threads

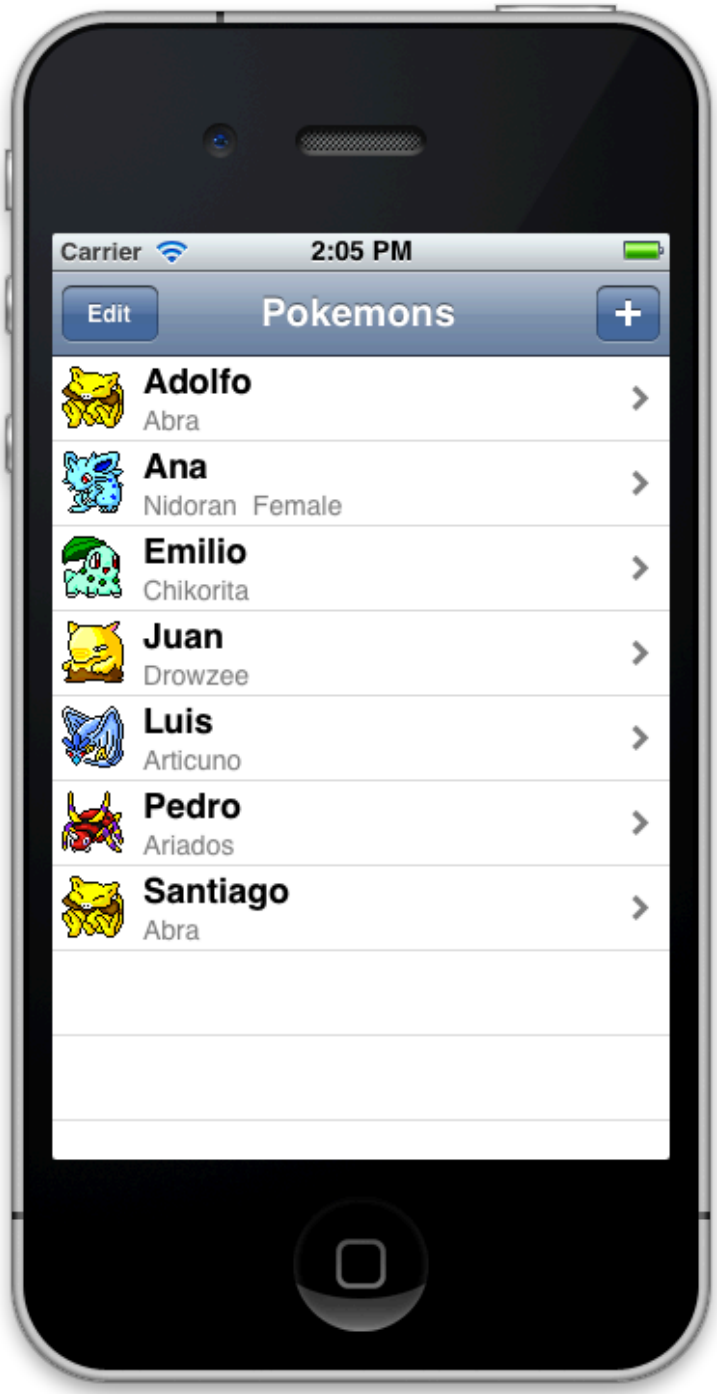
- `NSManagedObjectContext` no es Thread-Safe.
  - Sólo puede usarse desde el thread que lo creó.
- Si se necesita ejecutar realizar alguna operacion sobre el contexto desde otro thread puede usarse:

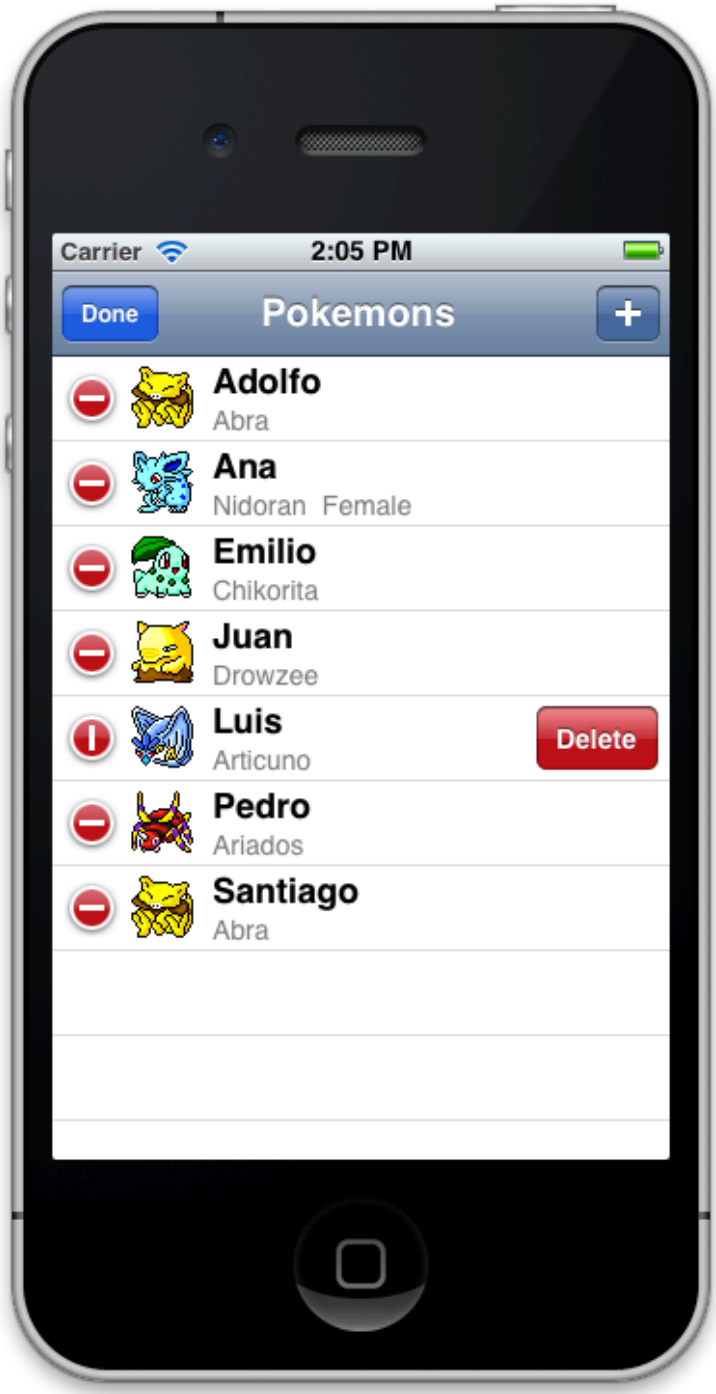
```
[context performBlock:^{ ... }];
```

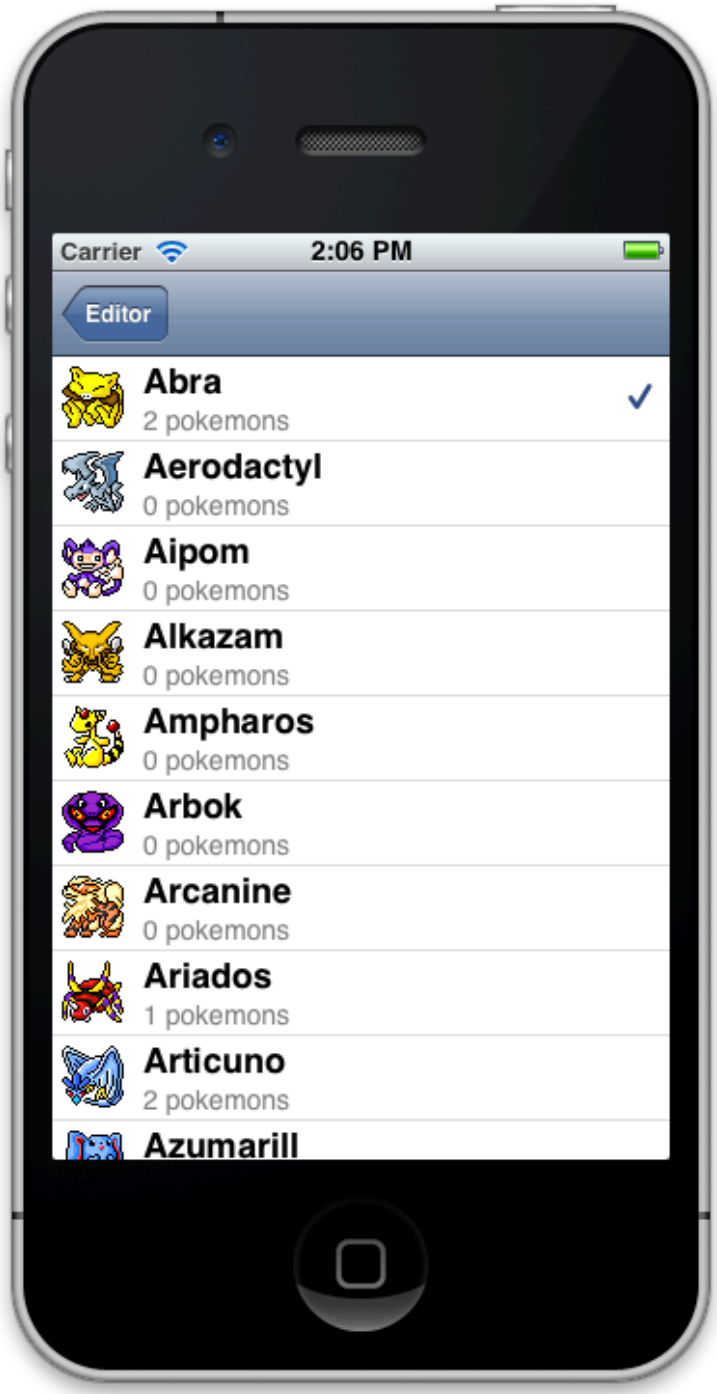
```
[context performBlockAndWait:^{ ... }];
```

  - El bloque se ejecuta en el thread donde se creó el contexto.

Demo
Pokedesk



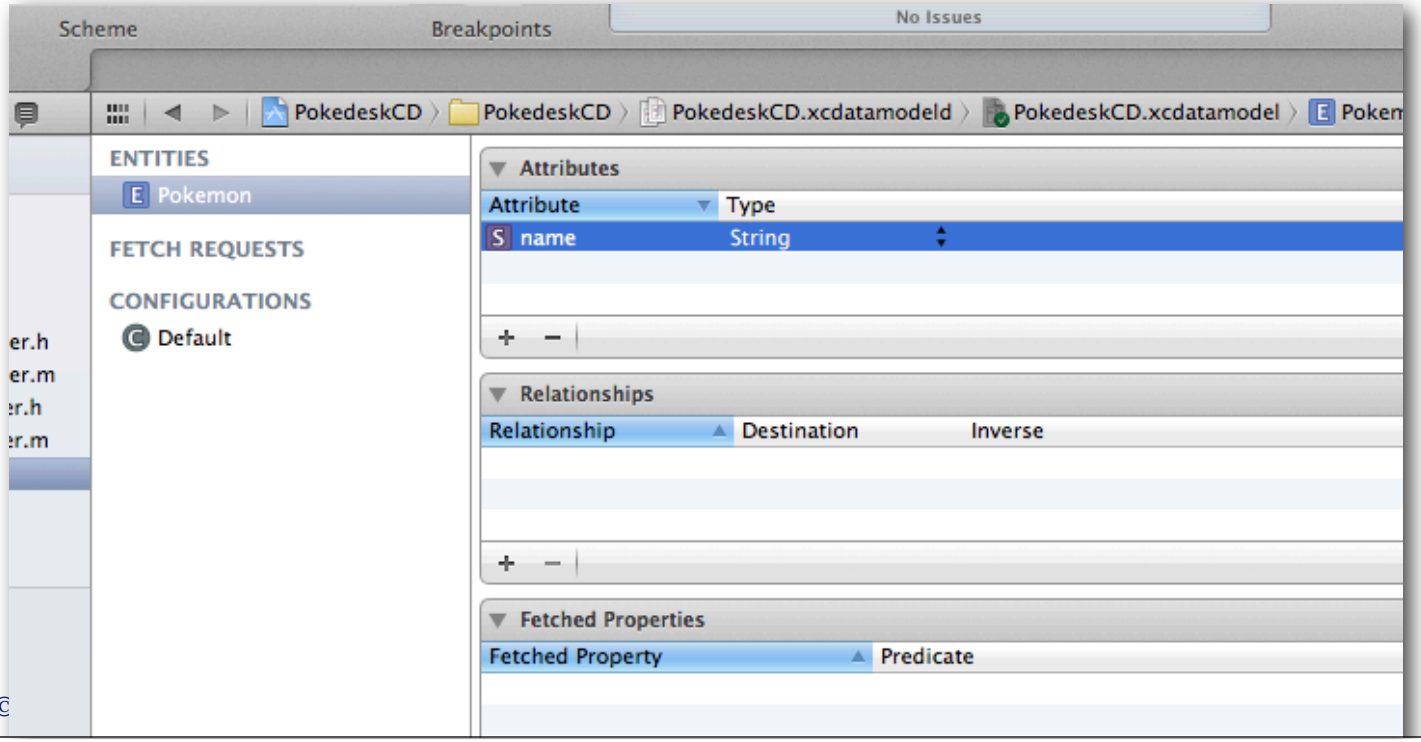


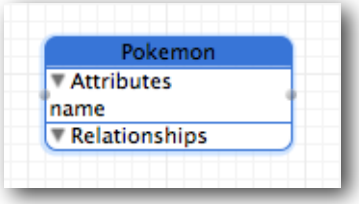


- Crear un nuevo proyecto usando la plantilla Master-Detail Application.
  - Nombre del producto: PokedeskCD
  - Organización: UPM
  - Identificador de la Compañía: es.upm.dit.swcm
  - Prefijo para nombre de las clases: vacio
  - Dispositivo: iPhone
  - Seleccionar:
    - Usar Storyboard
    - Usar CoreData
    - Usar ARC
- En el MasterViewController mostraré los individuos pokemon que tengo en mi pokedesk, y cuando seleccione un individuo lo mostraré en la vista DetailViewController para editarlo.
  - Renombrar las clases:
    - MasterViewController a PokemonsTableViewController
    - DetailViewController a PokemonEditorViewController
      - Seleccionar nombre de la clase y ejecutar Refactor+Rename

- Editar el fichero de modelo de datos PokedeskCD.xcdatamodeld para borrar la entidad Event que viene creada por defecto y crear una nueva entidad llamada Pokemon.
  - Con un atributo: name (de tipo String).

- Reemplazar en las clases PokemonsTableVC y PokemonEditorVC todas las referencias y usos de la antigua entidad Event y de su atributo timeStamp, por la nueva entidad Pokemon y su atributo name.





- Editar PokemonsTableVC.m
  - Cambiar el nombre de la cache usado al crear el objeto NSFetchedResultsController a "Pokemon".
  - Cambiar en tableView:dellForRowAtIndexPath: el identificador de la TableViewCell de "Cell" a "Pokemon Cell".
    - Cambiarlo también en el Storyboard.
  - Para que los pokemons salgan ordenados en la table view de forma ascendente y sin importar mayúsculas/minúsculas, hay que cambiar el objeto NSSortDescriptor que se crea en el método fetchedResultsController: por lo siguiente:

```
NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
            initWithKey:@"name"
              ascending:YES
               selector:@selector(localizedCaseInsensitiveCompare:)];
```

- Cambiar en la clase PokemonEditorVC
  - El nombre de la propiedad "detailItem" por "pokemon".
    - Refactorizar + Rename
  - Cambiar tambien su tipo de id a NSManagedObject*.

  - Hay que hacer unos retoques a mano:
    - Hay que editar en PokemonEditorVC el método setPokemon para eliminar algunos restos llamados detailItem y cambiarlos por pokemon. Y editar también el cambio del tipo id a NSManagedObject*.
    - En PokemonsTableVC hay que retocar el metodo prepareForSegue:sender: para cambiar las referencias de detailItem a pokemon.

- Editar el storyboard.
  - Cambiar el título de la barra de navegación de la escena PokemonsTableVC de Master a Pokemons.
  - Cambiar el título de la barra de navegación de la escena PokemonEditorVC de Detail a Editor.
- Ya podemos ejecutar la aplicación.
  - Crear varios pokemons.
  - Borrarlos
  - Seleccionarlos para verlos en la vista PokemonEditorVC.

- Quiero poder editar el nombre del pokemon en la vista PokemonEditorVC.
  - Editar PokemonEditorVC.h y .m y su escena en el Storyboard.
    - Sustituir la UILabel donde se muestra el nombre del pokemon por un UITextField.
      - Poner en el atributo Placeholder el valor Name.
    - Eliminar el outlet "detailDescriptionLabel", y crear un nuevo outlet llamado "pokemonNameTextField".
      - No olvidar hacer la conexión del outlet.

```
@property (weak, nonatomic)
          IBOutlet UITextField *pokemonNameTextField;
```

    - Sustituir en los sitios donde se use, el viejo outlet por el nuevo outlet.

- Para actualizar la base de datos con el nuevo nombre del pokemon que he editado en PokemonEditorVC:
  - Editar en el storyboard la view TextField:
    - Indicar que su delegado es el File Owner (PokemonEditorVC), y añadir en PokemonEditorVC.m el siguiente método:

```
- (void)textFieldDidEndEditing:(UITextField *)textField {
    [self.pokemon setValue:self.pokemonNameTextField.text forKey:@"name"];

NSError *error;
    if ([self.pokemon.managedObjectContext hasChanges] &&
        ![self.pokemon.managedObjectContext save:&error]) {
       NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
       abort();
    }
}
```

    - Conectar el evento "Did End On Exit" con una nueva IBAction para ocultar el teclado.
      - Crear la IBAction y conectarla a ese evento.

```
- (IBAction)hideKeyboard:(id)sender {  [sender resignFirstResponder];}
```

    - Cambiar en el teclado el valor de Return Key a Done.

- Ejecutar a aplicación.
  - Comprobar que funciona la actualización de los nombres de los pokemons.

- Quiero hacer una subclase de NSManagedObject para la entidad Pokemon, y así poder usar propiedades en vez de KVC.
  - En el editor de modelos, seleccionar la entidad Pokemon.
    - Ejecutar una de estas dos opciones (hacen lo mismo):

      File > New > File > Core Data + NSManagedObject Subclass

      Editor > Create NSManagedObject Subclass

      - Se crea la clase Pokemon (Pokemon.h .m)
- En PokemonEditorVC.h:
  - Cambiar el tipo de la propiedad pokemon a Pokemon*.
  - Importar Pokemon.h.
- En PokemonEditorVC.m:
  - En todos los sitios donde se usa id o NSManagedObject* como tipo de la propiedad pokemon, cambiar el tipo a Pokemon *.
  - Cambiar en todos los sitios las llamadas KVC (setValue:forKey: y valueForKey:) por el uso de la propiedad name.

- Ampliar la base de datos usando nuevas entidades:
  - Razas de pokemons y tipos de razas.
    - Son los mismos datos de los ejemplos usados en otros temas.

- Editar la definición de los modelos:
  - Añadir la entidad Race.
    - Añadir los atributos name (String), imageFile (String) y code (integer16).

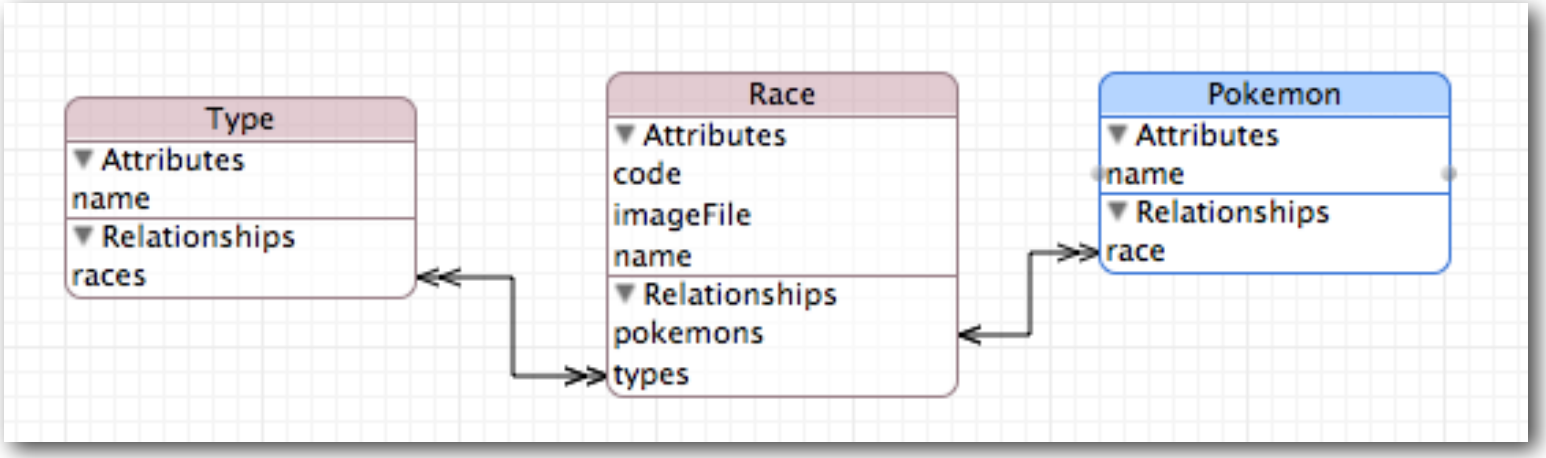
- Añadir la entidad Type.
  - Añadir el atributo name (String)

- Añadir una relación muchos-a-muchos entre Race y Type:

  - Añadir a la entidad Race la relación types.
    - Destination: Type
    - Inverse: races (no se puede poner hasta no crear la relacion en Type)
    - Plural: marcado
    - Delete Rule: Nullify

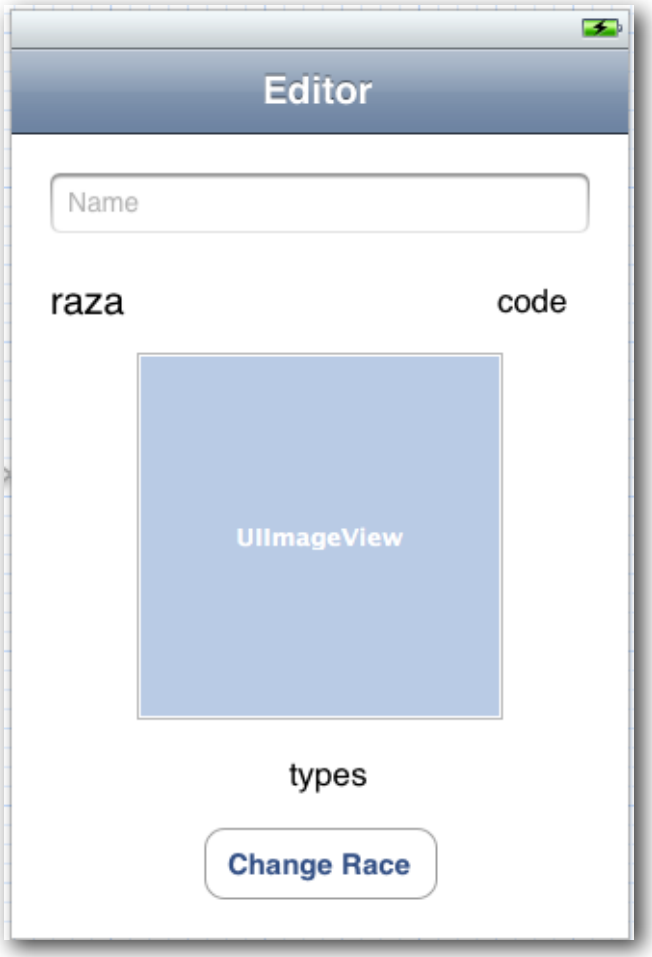
  - Añadir a la entidad Type la relación races.
    - Destination: Race
    - Inverse: types
    - Plural: marcado
    - Delete Rule: Nullify

- Los pokemons son de una raza, por lo que añadimos las siguientes relaciones:
  - En la entidad Race se añade la relación:
    - pokemons
      - (destino=Race, inverse=race , plural=marcado, y Nullify)
  - En la entidad Pokemon se añade la relación:
    - race
      - (destino=Race, inverse=pokemons, plural=desmarcado, Nullify)



- Crear subclases de NSManagedObject de todas las entidades.
  - En el editor de modelos, seleccionar todas las entidades.
    - Ejecutar dos veces:
      - Editor > Create NSManagedObject Subclass
    - Se crean los ficheros Pokemon.h.m, Race.h.m y Type.h.m.

- Mostrar todos los detalles de un pokemon en la pantalla de edición.
  - el nombre de la raza, el codigo de la raza, la foto y los nombres de los tipos.
  - Editar el storyboard y añadir a la escena PokemonEditorVC labels e image views para todos estos datos.
  - Añadir un botón "Change Race" para cambiar la raza del Pokemon.
  - Crear IBOutlets a las labels y a la image view.




```
@property (weak, nonatomic) IBOutlet UILabel *raceLabel;
@property (weak, nonatomic) IBOutlet UILabel *codeLabel;
@property (weak, nonatomic) IBOutlet UIImageView *photoImageView;
@property (weak, nonatomic) IBOutlet UILabel *typesLabel;
```

- Editar PokemonEditorVC.m para los valores a mostrar a los outlets creados.
  - Importar Race.h y Type.h.
  - Cambiar el contenido de configureView:

```
- (void)configureView {
    if (self.pokemon) {
        self.pokemonNameTextField.text = self.pokemon.name;

        self.raceLabel.text = self.pokemon.race.name;
        self.codeLabel.text = [NSString stringWithFormat:@"%d", [self.pokemon.race.code intValue]];;

        UIImage * img = [UIImage imageNamed:self.pokemon.race.imageFile];
        self.photoImageView.image = img;

        NSSet *types = self.pokemon.race.types;
        NSMutableArray *typeNamesArray = [[NSMutableArray alloc] initWithCapacity:types.count];
        for (Type *type in types) {
            [typeNamesArray addObject:type.name];
        }
        self.typesLabel.text = [typeNamesArray componentsJoinedByString:@","];
    }
}
```

- Este método se llamará desde viewWillAppear en vez desde viewDidLoad.
  - En un futuro se podrá editar la raza del pokemon y será necesario refrescar esta pantalla cada vez que se haga visible.

- Crear un VC nuevo, llamado RacesTableViewController, para mostrar las razas de pokemons existentes.
  - Este VC será el que se muestre cuando en PokemonEditorVC se pulse el botón "Change Race" para indicar que quiero cambiar la raza del pokemon.
    - En este VC se seleccionará la nueva raza, y se mostrará la raza actual del pokemon
  - Indicar que la clase RacesTableViewController deriva de UITableViewController.
    - Se crearán los ficheros RacesTableViewController.h y .m
- En el storyboard crear una nueva escena para la clase RacesTableViewController.
  - Usar como identificador del prototipo de celda "Race Cell".
  - Usar para el prototipo de celda el estilo Subtitle.
  - Crear un segue desde el boton "Change Race" hasta esta escena.
    - El segue será de tipo Push.
    - Usar "Race Selector" como identificador del segue.
- Añadir a RacesTableVC.h una propiedad para indicar sobre que pokemon estamos trabajando.

```
@property (nonatomic, strong) Pokemon *pokemon;
```

  - Es necesario importar Pokemon.h

- Para gestionar la table view de la clase RacesTableVC usaré un objeto NSFetchedResultController.
  - Crearlo como una propiedad privada (en la categoría de RacesTableVC.m).

```
@property (strong, nonatomic)
     NSFetchedResultsController *fetchedResultsController;
```

  - Crear el método getter típico para la propiedad fetchedResultsController.
    - Copiarlo de PokemonsTableVC.m y adaptarlo.
    - Notar que:
      - La entidad usada es Race.
      - No hay secciones.
      - El nombre de la cache es Races.
      - El delegado es nil porque no cambian nunca los datos de razas o tipos.

```
- (NSFetchedResultsController *)fetchedResultsController
{
    if (_fetchedResultsController != nil) {
        return _fetchedResultsController;
    }

    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    // Edit the entity name as appropriate.
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Race"
                                              inManagedObjectContext:self.pokemon.managedObjectContext];
    [fetchRequest setEntity:entity];

    // Set the batch size to a suitable number.
    [fetchRequest setFetchBatchSize:20];

    // Edit the sort key as appropriate.
    NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:@"name"
                                                                   ascending:YES
                                                                    selector:@selector(localizedCaseInsensitiveCompare:)];
    NSArray *sortDescriptors = @[sortDescriptor];
    [fetchRequest setSortDescriptors:sortDescriptors];

    // Edit the section name key path and cache name if appropriate.
    // nil for section name key path means "no sections".
    NSFetchedResultsController *aFetchedResultsController =
    [[NSFetchedResultsController alloc] initWithFetchRequest:fetchRequest
                                        managedObjectContext:self.pokemon.managedObjectContext
                                          sectionNameKeyPath:nil
                                                   cacheName:@"Races"];
    aFetchedResultsController.delegate = nil;
    self.fetchedResultsController = aFetchedResultsController;

   NSError *error = nil;
   if (![self.fetchedResultsController performFetch:&error]) {
       NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
       abort();
   }

    return _fetchedResultsController;
}
```

- En RacesTableVC hay que adaptar los métodos del protocolo Data Source.

```
#import "Race.h"
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    // Return the number of sections.
    return [[self.fetchedResultsController sections] count];
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
    // Return the number of rows in the section.
    id <NSFetchedResultsSectionInfo> sectionInfo =
                [[self.fetchedResultsController sections] objectAtIndex:section];
    return [sectionInfo numberOfObjects];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath
*)indexPath {
    static NSString *CellIdentifier = @"Race Cell";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

    // Configure the cell...
    Race *race = [self.fetchedResultsController objectAtIndexPath:indexPath];
    cell.textLabel.text = race.name;
    cell.detailTextLabel.text = [NSString stringWithFormat:@"%d pokemons",[race.pokemons count]];
    cell.imageView.image = [UIImage imageNamed:race.imageFile];
    if (self.pokemon.race == race) {
        cell.accessoryType = UITableViewCellAccessoryCheckmark;
    } else {
        cell.accessoryType = UITableViewCellAccessoryNone;
    }
    return cell;
}
```

- Preparar el segue que se dispara con el botón Change Race de PokemonEditorVC

```
#import "RacesTableViewController.h"

- (void) prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    if ([segue.identifier isEqualToString:@"Race Selector"]) {

        RacesTableViewController *rtvc = segue.destinationViewController;

        rtvc.pokemon = self.pokemon;
    }
}
```

- PENDIENTE DE MIRAR:

  - Cuando vuelvo desde la pantalla PokemonEditorVC a la pantalla PokemonsTableVC puede que se hayan actualizado los datos que gestiona Core Data. Si esto ha ocurrido, se habrán llamado a los métodos del delegado de FetchedResultsController informando de los cambios, y estos métodos habrán refrescado las celdas oportunas de la table view. Sin embargo, en el simulador me ocurre que algunas veces la celda que he editado no se muestra actualizada, pero he comprobado que se llama a configureCell con los parámetros correctos. No entiendo que está pasando. Seguramente se está cacheando la imagen en pantalla y no se repinta al volver. Para evitar este problema, de momento he añadido a PokemonsTableVC este método:

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    [self.tableView reloadData];
}
```

- Precargar datos iniciales.
  - La primera vez que se ejecuta la aplicación en la base de datos no habrá ningún dato sobre razas o tipos.
  - En viewDidLoad de RacesTableVC comprobaré si existen estos datos, y si no es así los crearé.
  - Copiar en el proyecto:
    - el fichero pokemons.plist que usamos en el tema de las tablas.
    - el directorio de los iconos.
  - En viewDidLoad se llamará a un método (que crearemos) para precargar los datos.

```
- (void) preloadRacesAndTypes {

    if (self.fetchedResultsController.fetchedObjects.count > 0) {return;}

    NSString * path = [[NSBundle mainBundle] pathForResource:@"pokemons"
                                                      ofType:@"plist"];
    NSDictionary * data = [NSDictionary dictionaryWithContentsOfFile:path];

    NSMutableDictionary * races = [NSMutableDictionary dictionary];

    NSDictionary * raceNames = [data valueForKey:@"nombres"];
    NSDictionary * raceIcons = [data valueForKey:@"iconos"];

    [raceNames enumerateKeysAndObjectsUsingBlock:^(id key, id obj, BOOL* stop) {
        Race *race = [NSEntityDescription insertNewObjectForEntityForName:@"Race"
                                                    inManagedObjectContext:self.pokemon.managedObjectContext];
        race.name = obj;
        race.code = [NSNumber numberWithInt:[key intValue]];
        race.imageFile = [raceIcons valueForKey:key];
        [races setObject:race forKey:key];
    }];

    NSDictionary * typeNames = [data valueForKey:@"tipos"];
    [typeNames  enumerateKeysAndObjectsUsingBlock:^(id key, id obj, BOOL* stop) {
        Type *type = [NSEntityDescription insertNewObjectForEntityForName:@"Type"
                                                    inManagedObjectContext:self.pokemon.managedObjectContext];
        type.name = key;
        [(NSArray *)obj enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {
            Race * race = [races objectForKey:[NSString stringWithFormat:@"%@",obj]];
            [race addTypesObject:type];
        }];
    }];

    NSError *error = nil;
    if (![self.pokemon.managedObjectContext save:&error]) {
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        abort();
    }

   if (![self.fetchedResultsController performFetch:&error]) {
       NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
       abort();
   }
}
```

- Al seleccionar una celda en RacesTableVC debe cambiarse la raza del pokemon, y volver a la pantalla de edición.
  - Editar el método didSelectRowAtIndexPath de RacesTableVC.

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    Race * race =  [self.fetchedResultsController objectAtIndexPath:indexPath];

    self.pokemon.race = race;

    // Save the context.
    NSError *error = nil;
    if (![self.pokemon.managedObjectContext save:&error]) {
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        abort();
    }

    [self.navigationController popViewControllerAnimated:YES];
}
```

- Ver en la pantalla inicial la foto del Pokemon y el nombre de su raza.
  - Editar en el Storyboard el prototipo de celda Pokemon Cell para que sea su estilo sea Subtitle.
  - En PokemonsTableVC.m editar el método configureCell:objectAtIndexPath para mostrar en la celda la imagen y la raza del pokemon.
    - Y de paso lo reescribimos para no usar KVC.

```
#import "Race.h"

- (void)configureCell:(UITableViewCell *)cell
        atIndexPath:(NSIndexPath *)indexPath
{
    Pokemon *pokemon = [self.fetchedResultsController objectAtIndexPath:indexPath];

    cell.textLabel.text = pokemon.name;
    cell.detailTextLabel.text = pokemon.race.name;
    cell.imageView.image = [UIImage imageNamed:pokemon.race.imageFile];
}
```

- Mejorar pendientes:
  - Hacer scroll en las tablas para que se vea el pokemon que acabo de añadir.
  - Al pulsar el botón + para añadir un pokemon, que se muestre directamente la ventana de edición.
  - Gestionar geometrías al rotar para que todo se vea bien.
  - Crear un VC llamado TypesTableViewController situado entre PokemonEditorVC y RacesTableVC.
    - Primero se selecciona un tipo y luego una raza dentro de ese tipo.
- . . .