NIKME
Quality Inside

Nikmesoft Ltd

Lecture 7

Data Storage

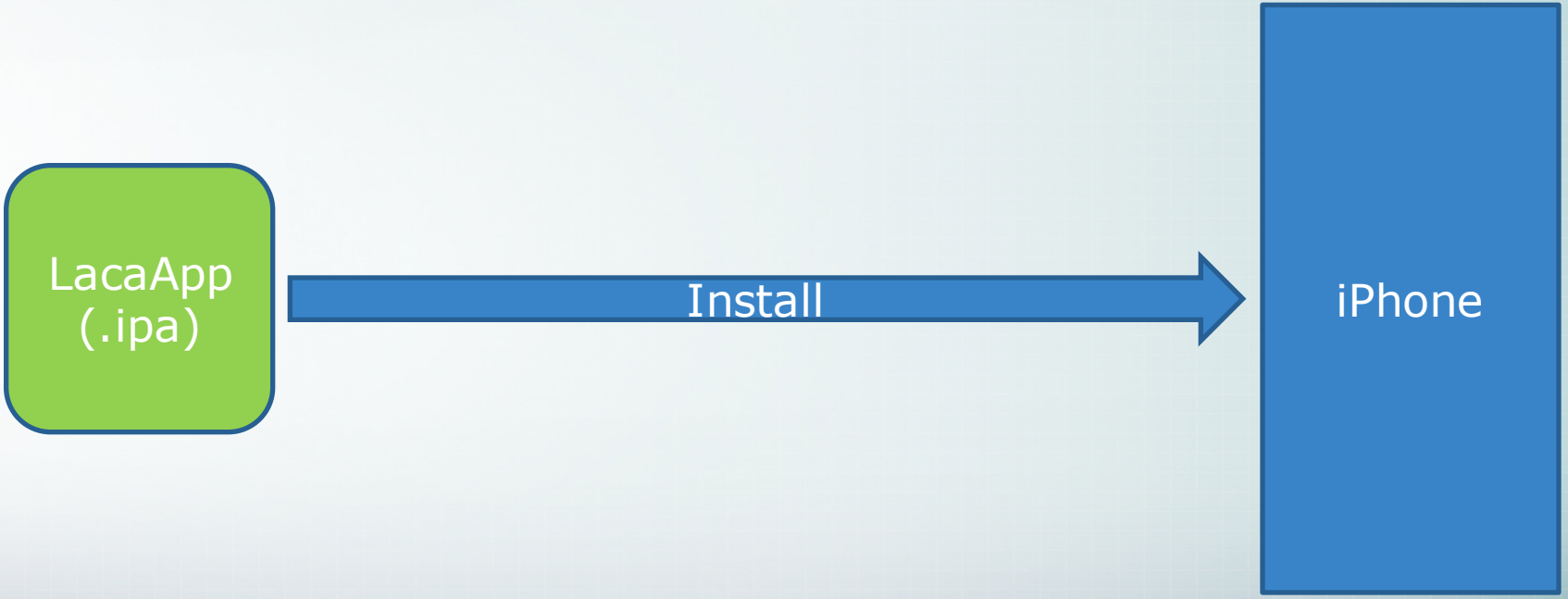
Linh NGUYEN

Contents

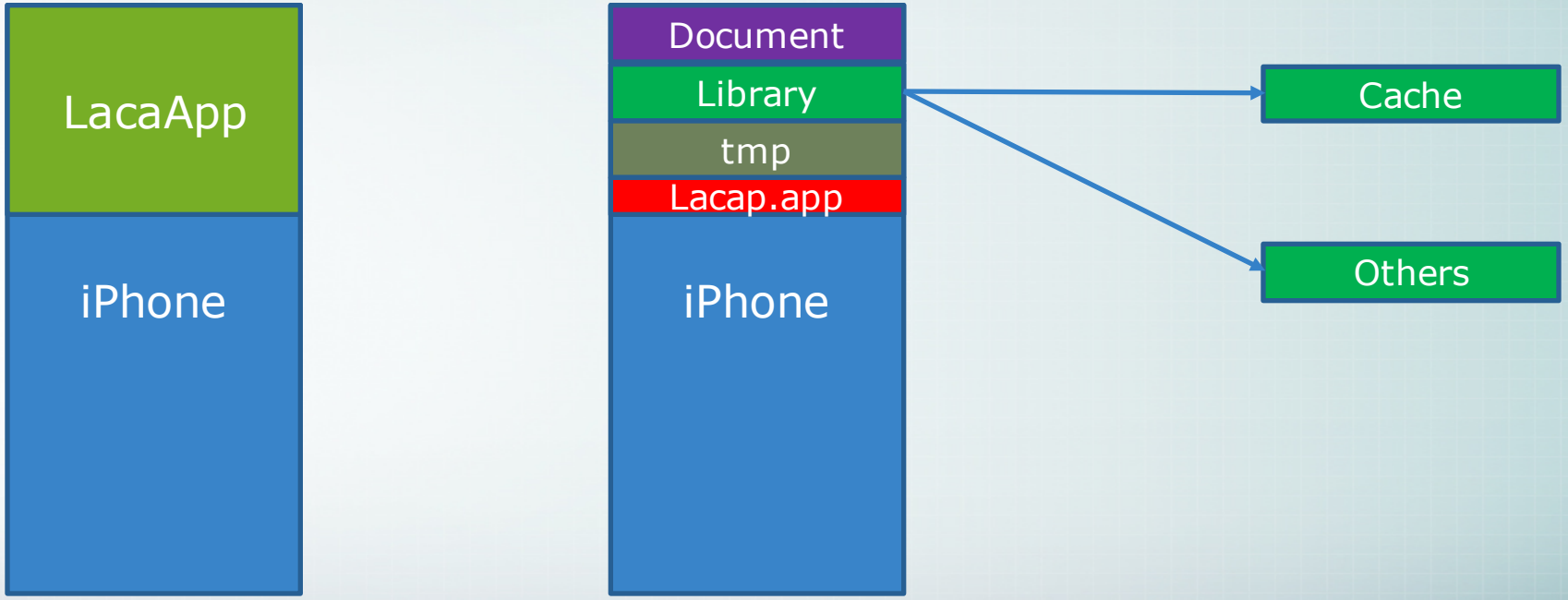
Data Storage

Overview

Overview



Overview


LacaApp
iPhone
Document
Library
tmp
Lacap.app
iPhone
Cache
Others

Overview

- **Document**: will be synced with iCloud
- **Library**: will not be synced
- **Lacap.app**: read only

Data Storage

Working with Files

Working with Files

- The Foundation Framework provides three classes that are indispensable when it comes to working with files and directories:
  - **NSFileManager**: The NSFileManager class can be used to perform basic file and directory operations such as creating, moving, reading and writing files and reading and setting file attributes

Working with Files

- The Foundation Framework provides three classes that are indispensable when it comes to working with files and directories:
  - **NSFileHandle**: The NSFileHandle class is provided for performing lower level operations on files, such as seeking to a specific position in a file and reading and writing a file's contents by a specified number of byte chunks and appending data to an existing file.
  - **NSData**: The NSData class provides a useful storage buffer into which the contents of a file may be read, or from which data may be written to a file.

Working with Files

- Playing with Directory

```
// Document
[[NSFileManager defaultManager]
URLsForDirectory:NSDocumentDirectory
inDomains:NSUserDomainMask] lastObject];

// Library
[[NSFileManager defaultManager]
URLsForDirectory:NSLibraryDirectory
inDomains:NSUserDomainMask] lastObject];

// Cache
[[NSFileManager defaultManager]
URLsForDirectory:NSCacheDirectory
inDomains:NSUserDomainMask] lastObject];
```

Working with Files

```
// Document
[[NSFileManager defaultManager]
URLsForDirectory:NSDocumentDirectory
inDomains:NSUserDomainMask] lastObject];

// Library
[[NSFileManager defaultManager]
URLsForDirectory:NSLibraryDirectory
inDomains:NSUserDomainMask] lastObject];

// Cache
[[NSFileManager defaultManager]
URLsForDirectory:NSCacheDirectory
inDomains:NSUserDomainMask] lastObject];
```

Working with Files

```
// Create path
NSURL *cacheURL = [[NSFileManager defaultManager]
URLsForDirectory:NSDocumentDirectory
inDomains:NSUserDomainMask] lastObject];
// file
NSURL *fileURL = [cacheURL URLByAppendingPathComponent:
[NSString stringWithFormat:@"%@",@"test.png"]];

// check file is exist
BOOL isDirectory = FALSE;
BOOL isExist = [fileManager fileExistsAtPath:fileURL.path
isDirectory:&isDirectory];
```

Working with Files

```
// Create a directory
[fileManager createDirectoryAtURL:fileURL withIntermediateDirectories:YES
attributes:nil error:&error];
if(!error)
{
    NSLog(@"Created cache directory");
} else {
    NSLog(@"%@",error.debugDesciption);
}

// remove
[fileManager removeItemAtPath:fileURL.path error:&error];
```

Working with Files

```
// Create a directory
[fileManager createDirectoryAtURL:fileURL withIntermediateDirectories:YES
attributes:nil error:&error];
if(!error)
{
    NSLog(@"Created cache directory");
} else {
    NSLog(@"%@",error.debugDesciption);
}

// remove
[fileManager removeItemAtPath:fileURL.path error:&error];

// get All files
NSArray *directoryContents = [fileManager contentsOfDirectoryAtPath:
literatureURL.path error:&error];
```

Data Storage

Store User's Settings

Store User's Settings

- Saving and retrieving different types of data using the **NSUserDefaults** object. Saving this way is great for when you want to save small amounts of data such as High Scores, Login Information, and program state.

- Saving to the NSUserDefaults is great because it does not require any special database knowledge.

Store User's Settings

```
NSUserDefaults *prefs = [NSUserDefaults standardUserDefaults];
// saving an NSString
[prefs setObject:@"TextToSave" forKey:@"keyToLookupString"];
// saving an NSInteger
[prefs setInteger:42 forKey:@"integerKey"];
// saving a Double
[prefs setDouble:3.1415 forKey:@"doubleKey"];
// saving a Float
[prefs setFloat:1.2345678 forKey:@"floatKey"];
// sync
[prefs synchronize];
```

Store User's Settings

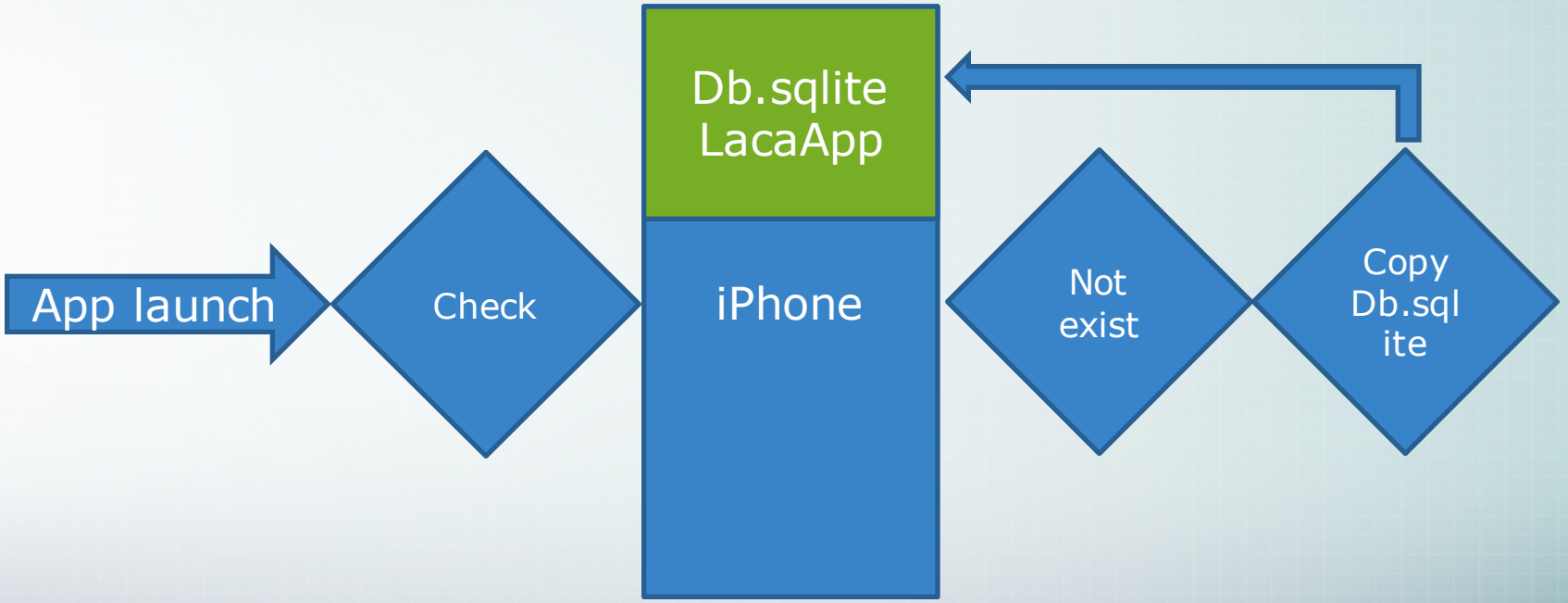
```
NSUserDefaults *prefs = [NSUserDefaults standardUserDefaults];

// getting an NSString
NSString *myString = [prefs stringForKey:@"keyToLookupString"];

// getting an NSInteger
NSInteger myInt = [prefs integerForKey:@"integerKey"];

// getting an Float
float myFloat = [prefs floatForKey:@"floatKey"];
```

Data Storage

Core Data

Core Data

- SQLite



Core Data

- Why Should You Use Core Data?
  - One of the simplest metrics is that, with Core Data, the amount of code you write to support the model layer of your application is typically 50% to 70% smaller as measured by lines of code.
  - Core Data has a mature code base whose quality is maintained through unit tests, and is used daily by millions of customers in a wide variety of applications. The framework has been highly optimized over several releases.

Core Data

- Why Should You Use Core Data?
  - You don’t have to implement yourself
  - You don’t have to test yourself
  - You don’t have to optimize yourself.

Core Data

- What Core Data Is Not
  - Core Data is not a relational database or a relational database management system (RDBMS).
  - Core Data is not a silver bullet. Core Data does not remove the need to write code. Although it is possible to create a sophisticated application solely using the Xcode data modeling tool and Interface Builder, for more real-world applications you will still have to write code.

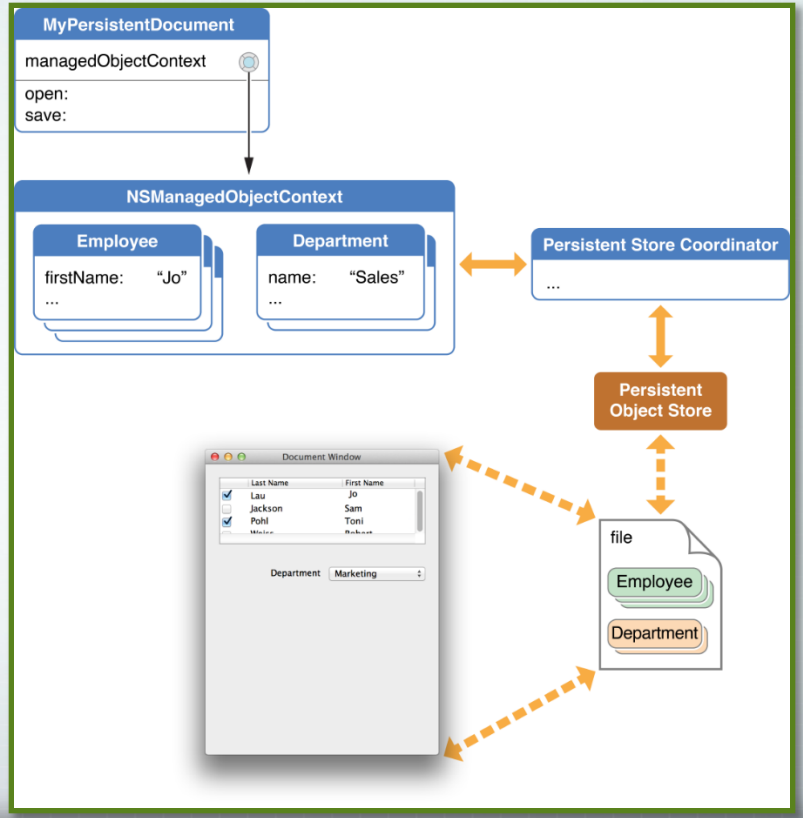
Core Data

- What Core Data Is Not
  - Core Data is not a relational database or a relational database management system (RDBMS).
  - Core Data is not a silver bullet. Core Data does not remove the need to write code. Although it is possible to create a sophisticated application solely using the Xcode data modeling tool and Interface Builder, for more real-world applications you will still have to write code.

Core Data

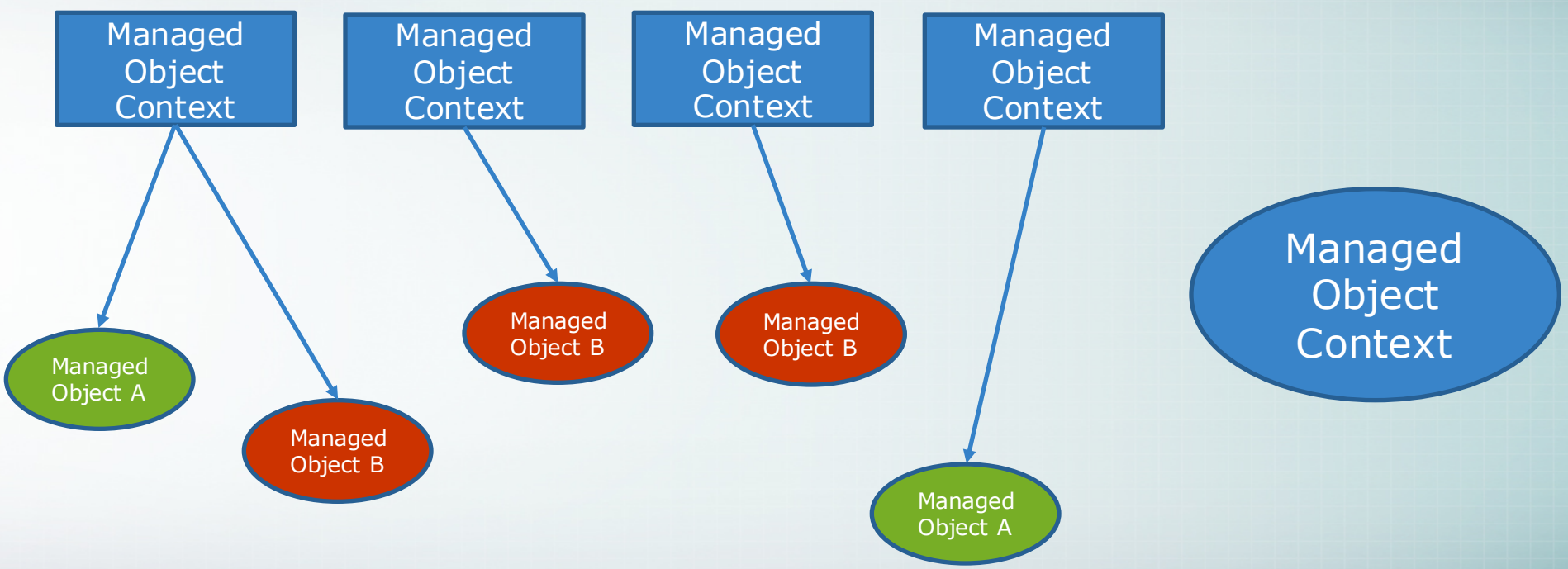
- What Core Data Is Not
  - Core Data is not a relational database or a relational database management system (RDBMS).
  - Core Data is not a silver bullet. Core Data does not remove the need to write code. Although it is possible to create a sophisticated application solely using the Xcode data modeling tool and Interface Builder, for more real-world applications you will still have to write code.

Core Data

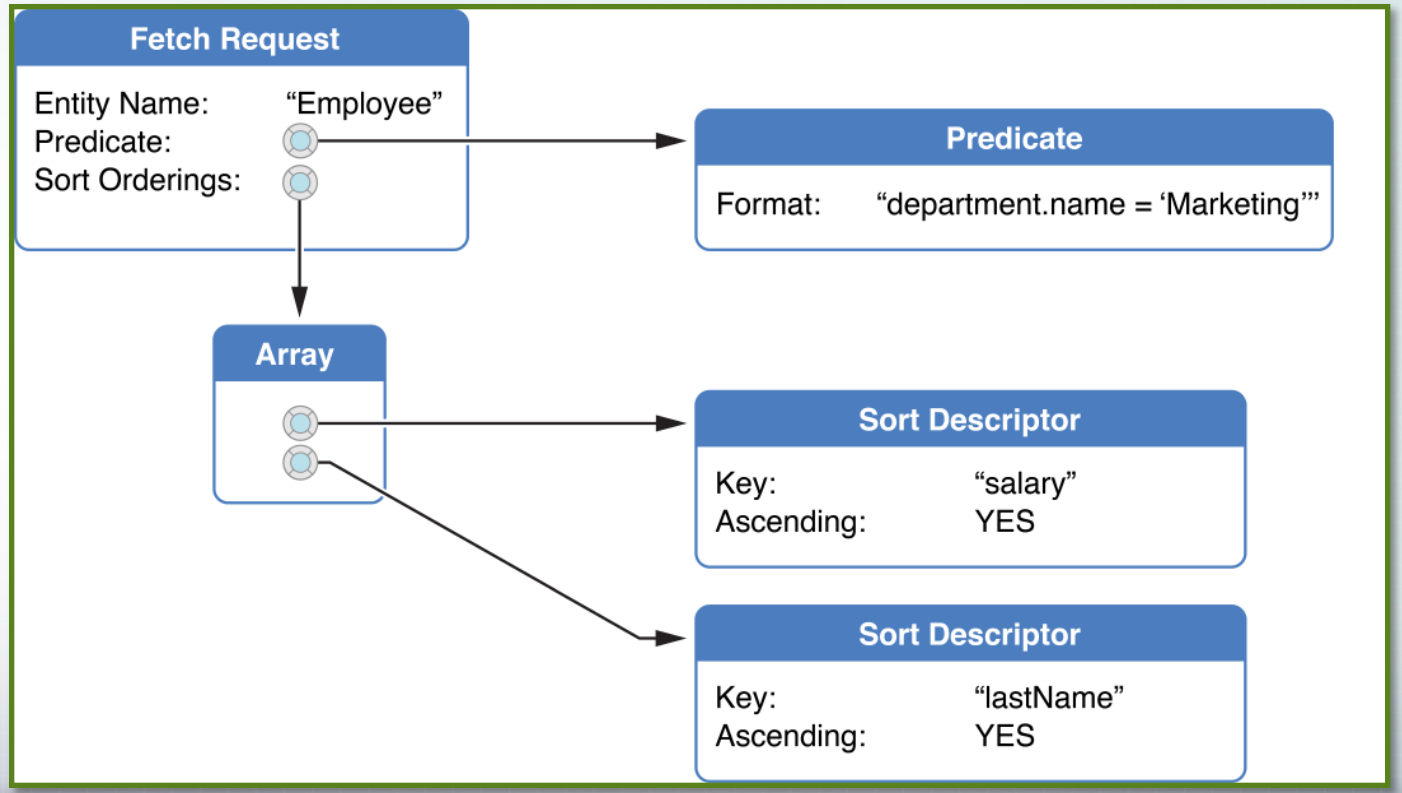


Overview

Core Data

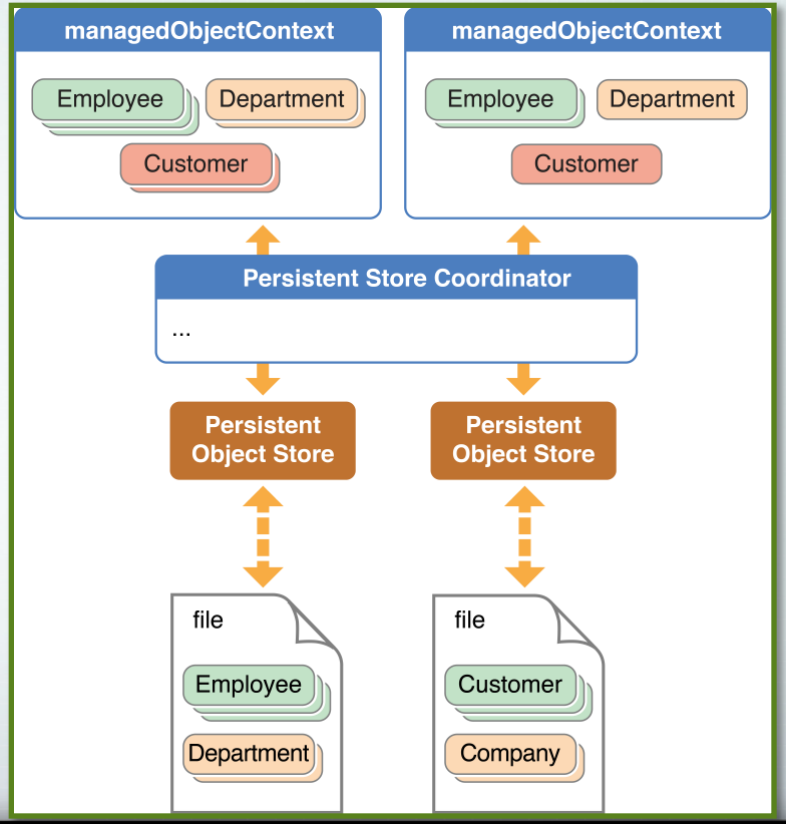

Managed Object Context
Managed Object Context
Managed Object Context
Managed Object Context
Managed Object A
Managed Object B
Managed Object B
Managed Object B
Managed Object A
Managed Object Context

Core Data



Fetch Request

Core Data



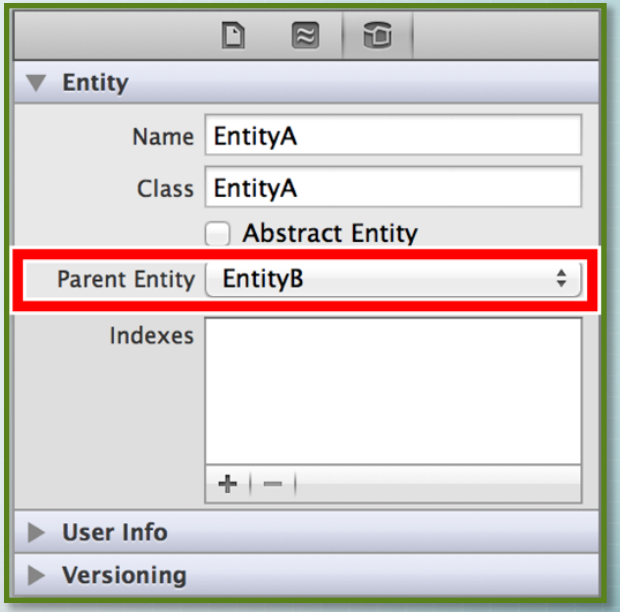
Persistent Store Coordinator

Core Data

- Managed Object Models
  - Much of Core Data's functionality depends on the schema you create to describe your application's entities, their properties, and the relationships between them. The schema is represented by a managed object model—an instance of NSManagedObjectModel
  - Features of a Managed Object Model

Core Data

- Managed Object Models
  - Features of a Managed Object Model
    - Entities
    - Entity Inheritance
    - Abstract Entities
    - Properties
    - An entity's properties are its attributes and relationships, including its fetched properties (if it has any). Amongst other features, each property has a name and a type. Attributes may also have a default value.



Core Data

- Managed Object Models
  - Features of a Managed Object Model
    - **Attributes**: Core Data natively supports a variety of attribute types, such as string, date, and integer (represented as instances of NSString, NSDate, and NSNumber respectively). If you want to use an attribute type that is not natively supported, you can use one of the techniques described in "**Non-Standard Persistent Attributes.**"
    - You can specify that an attribute is optional—that is, it is not required to have a value. In general, however, you are discouraged from doing so—especially for numeric values (typically you can get better results using a mandatory attribute with a default value—in the model—of 0).

Core Data

- Managed Object Models
  - Features of a Managed Object Model
    - **Relationships:** Core Data supports to-one and to-many relationships, and **fetched properties**.
    - **Inverse Relationships**
    - **Relationship Delete Rules**

Core Data

- Deny
  - If there is at least one object at the relationship destination, then the source object cannot be deleted.
  - For example, if you want to remove a department, you must ensure that all the employees in that department are first transferred elsewhere (or fired!) otherwise the department cannot be deleted.

Core Data

- Nullify
  - Set the inverse relationship for objects at the destination to null.
  - For example, if you delete a department, set the department for all the current members to null. This only makes sense if the department relationship for an employee is optional, or if you ensure that you set a new department for each of the employees before the next save operation.

Core Data

- Cascade
  - Delete the objects at the destination of the relationship.
  - For example, if you delete a department, fire all the employees in that department at the same time.

Core Data

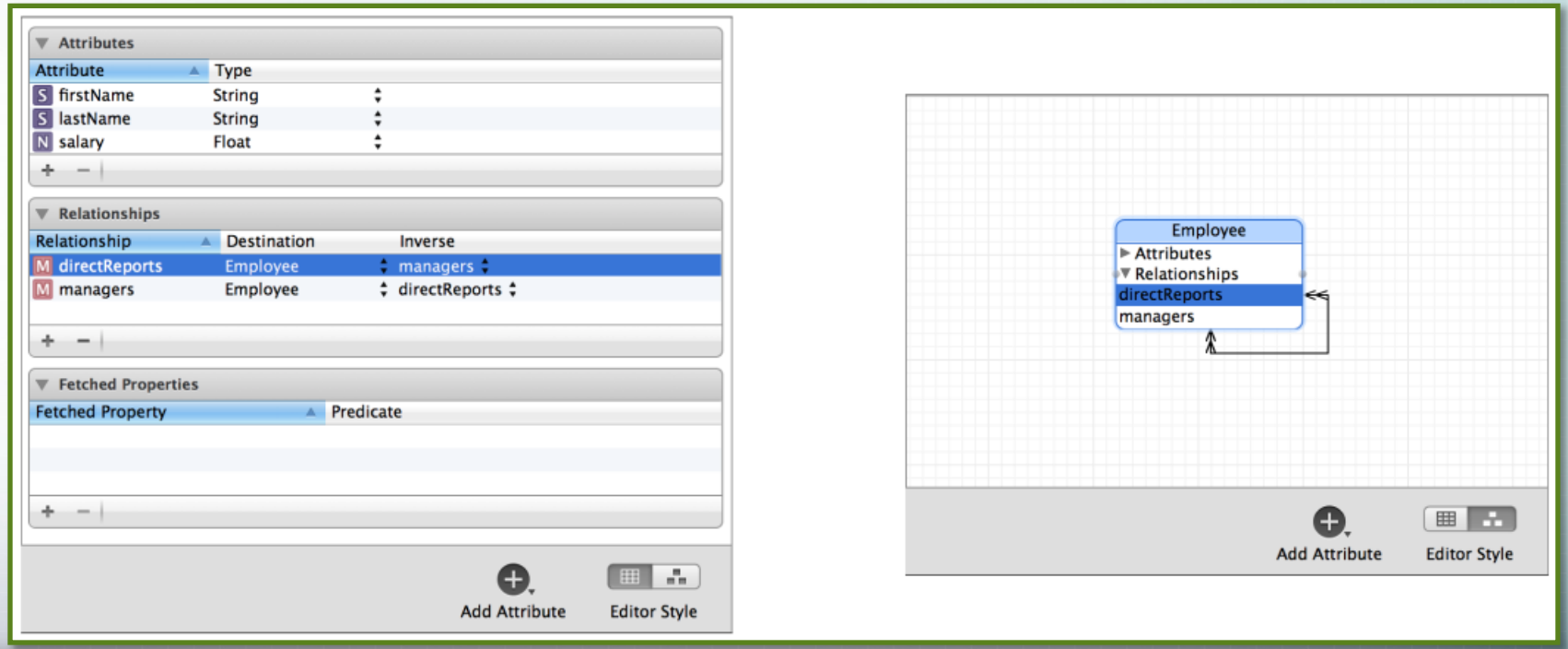
- No Action
  - Do nothing to the object at the destination of the relationship.
  - For example, if you delete a department, leave all the employees as they are, even if they still believe they belong to that department.

Core Data

- Many-to-Many Relationships
  - You define a many-to-many relationship using two to-many relationships. The first to-many relationship goes from the first entity to the second entity. The second to-many relationship goes from the second entity to the first entity. You then set each to be the inverse of the other.
  - A relationship can also be the inverse of itself. For example, a Person entity may have a cousins relationship that is the inverse of itself.

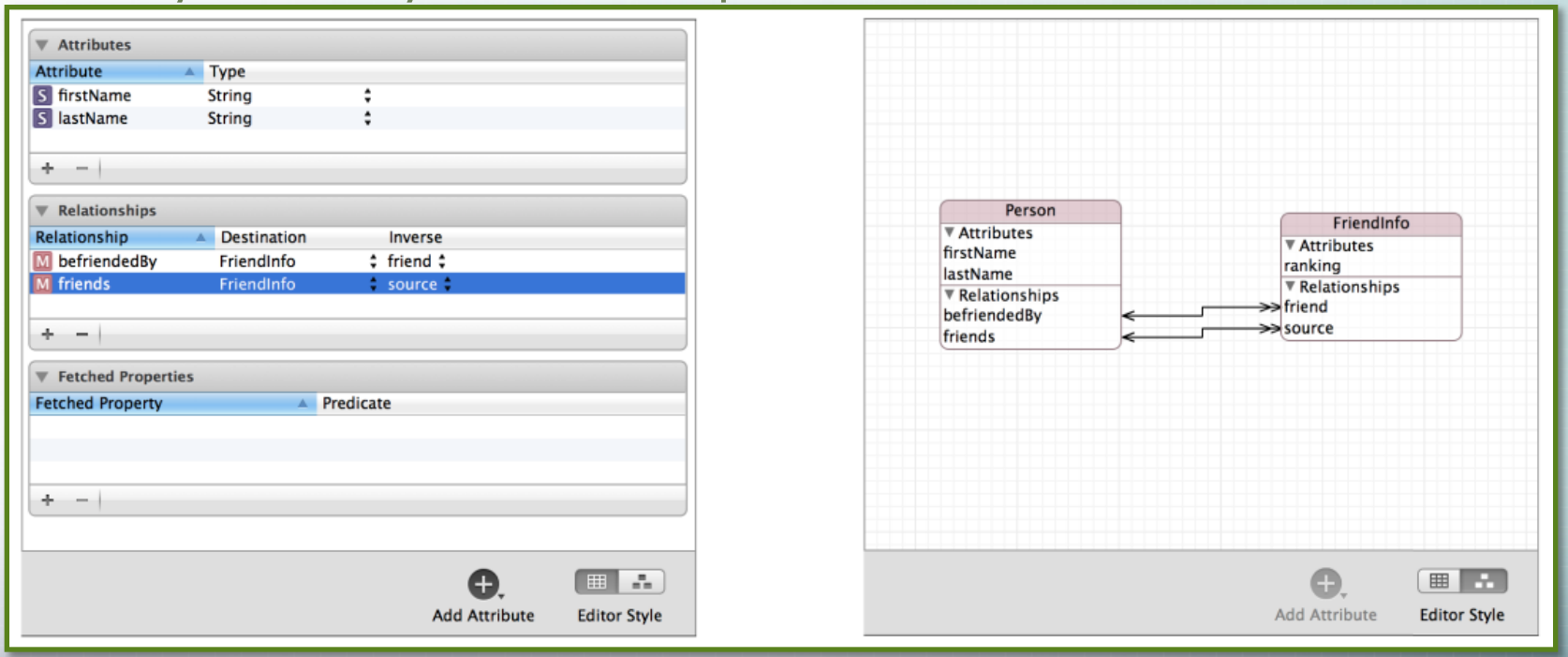
Core Data

- Many-to-Many Relationships



Core Data

- Many-to-Many Relationships



Core Data

- Cross-Store Relationships
  - You must be careful not to create relationships from instances in one persistent store to instances in another persistent store, as this is not supported by Core Data. If you need to create a relationship between entities in different stores, you typically use fetched properties

Core Data

- Using Persistent Stores
  - Access to stores is mediated by an instance of **NSPersistentStoreCoordinator**. You should not need to directly access a file containing a store. From a persistent store coordinator, you can retrieve an object that represents a particular store on disk. Core Data provides an **NSPersistentStore** class to represent persistent stores.

Core Data

- Using Persistent Stores
  - To create a store, you use a persistent store coordinator. You must specify the type of the store to be created, optionally a configuration of managed object model associated with the coordinator, and its location if it is not an in-memory store.

Core Data

- Using Persistent Stores

```
NSManagedObjectContext *moc = <#Get a context#>;
NSPersistentStoreCoordinator *psc = [moc persistentStoreCoordinator];
NSError *error = nil;
NSDictionary *options =
    [NSDictionary dictionaryWithObject:[NSNumber numberWithBool:1]
                                forKey:NSReadOnlyPersistentStoreOption];

NSPersistentStore *roStore =
    [psc addPersistentStoreWithType:NSXMLStoreType
                                configuration:nil URL:url
                                options:options error:&error];
```

Core Data

- Using Persistent Stores

```
//To retrieve a store object from a coordinator,
//you use the method persistentStoreForURL:

NSPersistentStoreCoordinator *psc = <#Get a coordinator#>;
NSURL *myURL = <#A URL identifying a store#>;
NSPersistentStore *myStore = [psc persistentStoreForURL:myURL];
NSFetchRequest *request = [[NSFetchRequest alloc] init];
[request setAffectedStores:[NSArray arrayWithObject:myStore]];
```

Core Data

- Using Fetched Request

```
// create the fetch request to get all Employees matching the IDs
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
[fetchRequest setEntity:
        [NSEntityDescription entityForName:@"Employee"
        inManagedObjectContext:context]];
[fetchRequest setPredicate: [NSPredicate predicateWithFormat:
@"(employeeID IN %@)", employeeIDs]];
// Make sure the results are sorted as well.
[fetchRequest setSortDescriptors:
    @[ [[NSSortDescriptor alloc] initWithKey: @"employeeID"
    ascending:YES] ]];
// Execute the fetch.
NSError *error;
NSArray *employeesMatchingNames = [aMOC executeFetchRequest:
fetchRequest error:&error];
```

Core Data

- New/Delete a NSManagedObject

```
// insert object
[NSEntityDescription insertNewObjectForEntityForName:
@"Employee" inManagedObjectContext:self];

// save context
[context save:&error];

// delete object
[context deleteObject:employee];
```

Core Data

- Example 7.1

Core Data

- NSFetchedResultsControllers

```
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
NSEntityDescription *entity = [NSEntityDescription
    entityForName:@"Enployee" inManagedObjectContext:managedObjectContext];
[fetchRequest setEntity:entity];
NSSortDescriptor *sort = [[NSSortDescriptor alloc]
    initWithKey:@"employeeID" ascending:YES];
[fetchRequest setSortDescriptors:[NSArray arrayWithObject:sort]];
[fetchRequest setFetchBatchSize:20];
NSFetchedResultsController *theFetchedResultsController =
    [[NSFetchedResultsController alloc] initWithFetchRequest:fetchRequest
        managedObjectContext:managedObjectContext sectionNameKeyPath:nil
        cacheName:@"Root"];
self.fetchedResultsController = theFetchedResultsController;
_fetchedResultsController.delegate = self;
```

Core Data

- NSFetchedResultsControllers

```
- (void)controllerWillChangeContent:
(NSFetchedResultsController *)controller {
    // The fetch controller is about to
    //start sending change notifications,
    //so prepare the table view for updates.
    [self.tableView beginUpdates];
}

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

Core Data

- NSFetchedResultsControllers

```
- (void)controller:(NSFetchedResultsController *)controller
didChangeObject:(id)anObject atIndexPath:(NSIndexPath *)indexPath
forChangeType:(NSFetchedResultsChangeType)type newIndexPath:
(NSIndexPath *)newIndexPath {
    switch(type) {
        case NSFetchedResultsChangeInsert:
            break;
        case NSFetchedResultsChangeDelete:
            break;
        case NSFetchedResultsChangeUpdate:
            break;
        case NSFetchedResultsChangeMove:
           break;
    }
}
```

Core Data

- NSFetchedResultsControllers

```
- (void)controller:(NSFetchedResultsController *)
controller didChangeSection:(id )sectionInfo
atIndex:(NSUInteger)sectionIndex
forChangeType:(NSFetchedResultsChangeType)type {

    switch(type) {
        case NSFetchedResultsChangeInsert:
            break;
        case NSFetchedResultsChangeDelete:
            break;
    }
}
```

Core Data

- Example 7.2

NIKME
Quality Inside

Thank You !