Persistance

- Property Lists
- Archivierung
- Filesystem
- SQLite
- CoreData

Property Lists

- für ‚kleine' Objektgefüge, die nur aus `NSArray`, `NSDictionary`, `NSString`, `NSData`, `NSDate`, `NSNumber` bestehen (preferences, settings - NOT real application data)
- Spezialfall `NSUserDefaults` (s. o.)
- Generell serialisierbar (in/aus `NSData`): (NSPropertyListSerialization)

```
+ (NSData *)dataWithPropertyList:(id)plist
  format:(NSPropertyListFormat)format // XML or Binary
 options:(NSPropertyListWriteOptions)options // unused, set to 0
   error:(NSError **)error

+ (id)propertyListWithData:(NSData *)plist
 options:(NSPropertyListReadOptions)options // see below
  format:(NSPropertyListFormat *)format // returns XML or Binary
   error:(NSError **)error;
```

Property Lists

- NSPropertyListReadOptions

```
enum {
   NSPropertyListImmutable = ...,
   NSPropertyListMutableContainers = ...,
   NSPropertyListMutableContainersAndLeaves = ...
};
typedef NSUInteger NSPropertyListMutabilityOptions;
```

- NSPropertyListFormat

```
enum {
   NSPropertyListOpenStepFormat = ...// nur Lesen old-style PL
   NSPropertyListXMLFormat_v1_0 = ...,
   NSPropertyListBinaryFormat_v1_0 = ...
};
typedef NSUInteger NSPropertyListFormat;
```

Property Lists

- dann NSData in File schreiben ...
  - NSData Methode

    ```
    + (BOOL)writeToURL:(NSURL *)fileURL atomically:(BOOL)atomically;
    ```

    return ‚erfolgreich‘

    nur File URLs (“file://”) erlaubt.

- oder NSData von einer URL lesen
  - NSData Methode

    ```
    + initWithContentsOfURL:(NSURL *)aURL;
    ```

    lesen von non-file URL (z.B. web server) ok

Archivierung

- beliebige Objektgefüge permanent machen:
  (z.B. die View-Hierarchien aus dem IB)
- dazu müssen die beteiligten Objekte das NSCoding Protokoll implementieren
  - `- (void)encodeWithCoder:(NSCoder *)coder;`
  - `- initWithCoder:(NSCoder *)coder;`

Archivierung

- der Benutzer legt die Kodierung fest und muss sich insb. auch um Zyklen kümmern :-(
- Beispiel: eine Klasse mit 3 zu archivierenden Eigenschaften

```
- (void)encodeWithCoder:(NSCoder *)coder {
    [super encodeWithCoder:coder]; // !!!
    [coder encodeFloat:scale forKey:@“scale”];
    [coder encodeCGPoint:origin forKey:@“origin”];
    [coder encodeObject:expression forKey:@“expression”];
}
```

- Objekterzeugung aus Archiv mit alloc/initWithCoder:

```
- initWithCoder:(NSCoder *)coder {
    self = [super initWithCoder:coder]; // !!!
    scale = [coder decodeFloatForKey:@“scale”];
    expression =
      [[coder decodeObjectForKey:@“expression”] retain];
    origin = [coder decodeCGPointForKey:@“origin”]; // Reihgenfolge beliebig
}
```

Archivierung

- in Gang kommt das Ganze dann durch die Methoden:

```
// NSKeyedArchiver:
+ (NSData *)archivedDataWithRootObject:(id <NSCoder>)rootObject;
// NSKeyedUnArchiver:
+ (id <NSCoder>)unarchiveObjectWithData:(NSData *)data;
```

- was macht ?

```
id <NSCoder> object = ...;
NSData *data = [NSKeyedArchiver archivedDataWithRootObject:object];
id <NSCoder> dup = [NSKeyedUnArchiver unarchiveObjectWithData:data];
```

- Archivierung: ein einfacher Mechanismus, der aber schwer (korrekt) umzusetzen ist :-(

Filesystem

- Aus Sicht der Applikation ein ‚normales Unix Filesystem`
  - beginnt bei /
  - Zugriffsrechte wie üblich, aber
  - Schreiben nur in der (appl.-spezifischen) Sandbox erlaubt:

- Sandbox enthält:
  - das ‚Application bundle directory` (binary, .xibs, .jpgs, ...) NICHT schreibbar!
  - das ‚Documents directory`: hier kann man permantente Daten ablegen/lesen
  - ‚Caches directory`: für temporäre Files (kein Backup durch iTunes)
  - weitere Verzeichnisse (s. NSSearchPathDirectory)

Filesystem

- eine App will in eine Datei (aus dem bundle) schreiben?
  - aus dem bundle ins documents (oder anderes) directory kopieren
- Zugang zu den directories?

```
NSArray *NSSearchPathForDirectoriesInDomains(
    NSSearchPathDirectory directory, // see below
    NSSearchPathDomainMask domainMask, // NSUserDomainMask
    BOOL expandTilde // YES
);
```

- Array, aber meist nur ein Eintrag (lastObject)
- NSSearchPathDirectory z.B.

  NSDocumentsDirectory, NSCachesDirectory,
  NSAutosavedInformationDirectory, ...

Filesystem

- alle Lese-/Schreiboperationen arbeiten auf NSData (NSFileHandle: writeData/readDataOfLength/readDataToEndOfFile)
- alle anderen Fileoperationen via (thread safe) NSFileManager

```
NSFileManager *manager = [[NSFileManager alloc] init];

// NSFileManager-Methoden:
- (BOOL)createDirectoryAtPath:(NSString *)path
  withIntermediateDirectories:(BOOL)createIntermediates
                   attributes:(NSDictionary *)attributes // permissions, etc.
                        error:(NSError **)error;
- (BOOL)isReadableFileAtPath:(NSString *)path;
- (NSArray *)contentsOfDirectoryAtPath:(NSString *)path error:(NSError **)error;
```

- befragt bei allen Operationen potentiell einen (beliebigen NSObject) delegate:

  `fileManager:should“IdoThis“`

- diverse Pfadoperationen in `@interface NSString (NSStringPathExtensions)`

SQLite

- SQL dateibasiert
  - schnell, speichereffizient, zuverlässig
  - Open Source, in iOS integriert
  - nicht für alles gut (z.B. video oder große sounds/images).
  - nicht Server-basiert („not great at concurrency“).
- (C-)API :-(

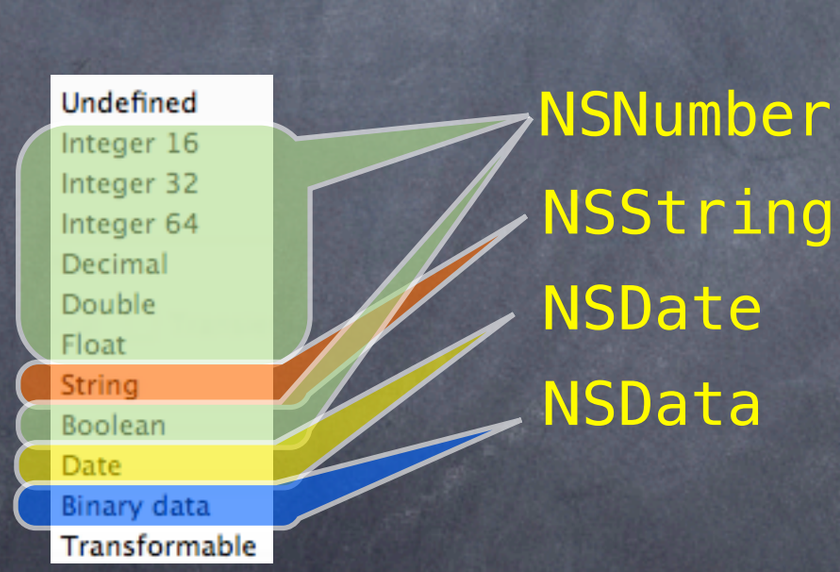
```
int sqlite3_open(const char *filename, sqlite3 **db); // get a database into db
int sqlite3_exec(sqlite3 *db, // execute SQL statements
                 const char *sql,
                 int (*callback)(void *, int, char **, char **),
                 void *context,
                 char **error);
int mycallback(void *context, int count, char **values, char **cols)
                 // data returned
int sqlite3_close(sqlite3 *db); // close the database
```

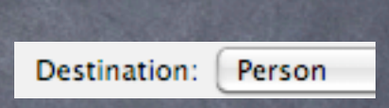
Core Data

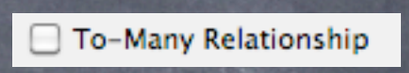
- ‚OO above SQLite‘
- direkte Abbildung von Objekten auf DB-Einträge:
  - Objekte <---> Entitäten
  - Properties <---> Attribute
  - Objektverknüpfungen <---> Relationen
- Erzeugung des Datenmodells mit xcode
- Objekterzeugung und -anfrage mit einen OO-API

Core Data

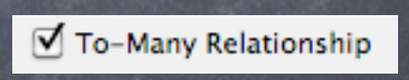
- Demo: data model
- Objective-C Abbildung der Attribute/Relationen:








```
NSManagedObject*
```




```
NSSet*
(of NSManagedObject*)
```

Core Data

- Wie ist der Zugang aus der Applikation heraus?
- man braucht einen NSManagedObjectContext
  - der Dreh- und Angelpunkt für alle CoreData Operationen
  - verwaltet das in-memory Abbild der Datenbank
  - nicht thread-safe aber separate Kontexte in separaten Threads können koexistieren
- Woher bekommt man einen solchen?
  - Klick auf "Use Core Data for storage" bei der Projekterzeugung
  - die Applikation wird mit Core Data-Support ausgestattet:
  - insb. application erhält eine

```
@property (nonatomic, retain, readonly) NSManagedObjectContext *managedObjectContext;
```

Core Data

- Objekterzeugung (CoreData-synchron):

```
NSManagedObject *person =
[NSEntityDescription insertNewObjectForEntityForName:@“Person”
                                  inManagedObjectContext:context];
```

  - gibt immer ein `NSManagedObject` zurück !

- NSManagedObjectContext wird „überall“ gebraucht
  - übliche Verfahrensweise: App.delegate reicht seinen an erzeugt Controller etc. weiter:
  - Core Data-anzeigende ViewController haben Initializers wie:

```
- initInManagedContext:(NSManagedObjectContext *)context;
```

  - oder im app delegate:

```
vc.managedObjectContext = self.managedObjectContext;
```

Core Data

- Attribute am Objekt lesen/setzen?
  - KVC key value coding:
    - `- (id)valueForKey:(NSString *)key; //read`
    - `- (void)setValue:(id)value forKey:(NSString *)key; // write`
  - `key` ist der Attributname aus dem Datenmodell z.B. `@“name”` (der Person)
  - `value` wird im Objekt hinterlegt
  - `nil` solange nicht gesetzt (bzw. DefaultValue aus Modell)
  - alle Werte sind Objektzeiger (ggf. `NSNumber` Objekte)
- Alles passiert automatisch:
  - passende SQL statements werden erzeugt
  - ‚lazy‘ Zugriffe auf Objekte erst dann, wenn nötig

Core Data

- aber alles geschieht zunächst nur im Speicher, bis man save am NSManagedObjectContext ruft
  - nicht nach jeder Änderung, sondern nach einer Menge ‚zusammengehöriger Änderungen am Kontext:

```
- (BOOL)save:(NSError **)errors; // NSError* kommt raus
// [context save:NULL]; Fehler ignoriert!

// u.U. hat sich gar nichts geändert?
- (BOOL)hasChanges;
// if ([context hasChanges]) ...
```

Core Data

- Beispiel:

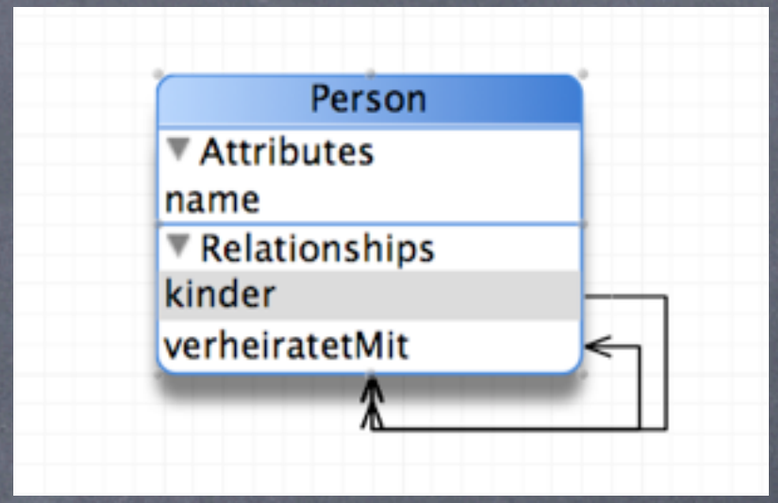
```
- (void)saveChangesToObjectsInMyMOC:(NSManagedObjectContext *)context
{
    NSError *error = nil;
    if ([context hasChanges] && ![context save:&error]) {
        NSLog(@“Error! %@, %@”, error, [error userInfo]);
        abort(); // generates a crash log for debugging purposes
    }
}
```

Core Data

- `valueForKey:/setValueForKey:` erfolgt ohne Typprüfung und ist nicht sonderlich elegant :-(
- was man wirklich will sind setter/getter auf `@property`s!
- xcode kann Ableitungen von `NSManagedObject` mit entspr. Properties automatisch erzeugen
  - sollten wie die Entities im Datenmodell heißen
  - zwei Varianten:
    1. Klassen neu erzeugen: File: New File: Managed Object Class
    2. existierenden Klassen erweitern: Design: Data Model: ‚special Copy'

Core Data

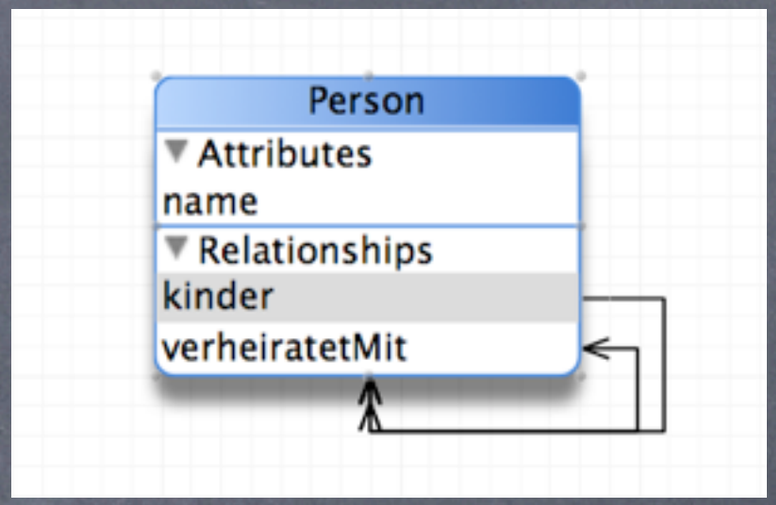
- generiert werden Klassen a la




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


@interface Person :  NSManagedObject  {}

@property (nonatomic, retain) NSString * name;
@property (nonatomic, retain) NSManagedObject * verheiratetMit;
@property (nonatomic, retain) NSSet* kinder;

@end


@interface Person (CoreDataGeneratedAccessors)
- (void)addKinderObject:(NSManagedObject *)value;
- (void)removeKinderObject:(NSManagedObject *)value;
- (void)addKinder:(NSSet *)value;
- (void)removeKinder:(NSSet *)value;

@end
```

NSSet is immutable

Core Data



- + Implementation

```
#import "Person.h"

@implementation Person

@dynamic name;
@dynamic verheiratetMit;
@dynamic kinder;

@end
```

statt @synthesize: teilt dem Compiler mit, dass die Methoden rufbar sind (durch spezielle NSManagedObject-Magie bereitgestellt werden)

Core Data

- am Ende benutzbar, als wären es synthetisierte Properties:

```
Person *egon = [NSEntityDescription
insertNewObjectForEntityForName:@“Person” inManage...];

egon.name = @“Egon Olsen“;

if (egon.verheiratetMit) ... // nil: nobody
```

Core Data

- das wichtigste fehlt noch: Queries
  - Anfragen nach Objekten in der Datenbank (mit gewissen Eigenschaften)
- dazu muss man ein `NSFetchRequest`-Objekt konstruieren (s.u.) und an einem `NSManagedObjectContext` ausführen:

```
NSFetchRequest *request = ...;
NSError **error = nil;
NSArray *results = [mangedObjectContext
                executeFetchRequest: request error:&error];
```

  - `nil` bei Fehlern (Details im `NSError`)
  - ein leeres Array (nicht `nil`) wenn keine passenden Objekte in der Datenbank gefunden wurden
  - ein Array von `NSManagedObjects` (bzw. Ableitungen) wenn passende Objekte gefunden wurden

Core Data

- `NSFetchRequest` erzeugen:

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

- `NSFetchRequest`-Zutaten (als Properties zu setzen)

1. eine `NSEntityDescription`: welcher Entitytyp wird gesucht (`SELECT`) (muss angegeben werden) `request.entity`
2. ein `NSPredicate` : welche Eigenschaft wird gesucht (`WHERE`) (optional, default: alle) `request.predicate`
3. `NSSortDescriptors` steuern die Reihenfolge der Objekte im Ergebnisfeld (optional, default zufällig) `request.sortDescriptors`
4. Anzahl der Objekte im Ergebnis bei einer Ausführung bzw. insgesamt (optional, default: alle) `request.fetchBatchSize`, `request.fetchLimit`