LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

mobile and distributed systems group

Praktikum iOS-Entwicklung

Sommersemester 2016

Prof. Dr. Linnhoff-Popien

Florian Dorfmeister, Marco Maier, Mirco Schönfeld

Agenda Heute

- Benachrichtigung über (Zustands-)Änderungen
  - Notifications & KVO
- Einschub: Zustände von iOS Apps
- Speichern von Zuständen und Daten
  - Archiving mittels NSCoding
  - Binäres Speichern
  - NSUserDefaults

Gemeinsames Themen-Brainstorming am 01.06.!

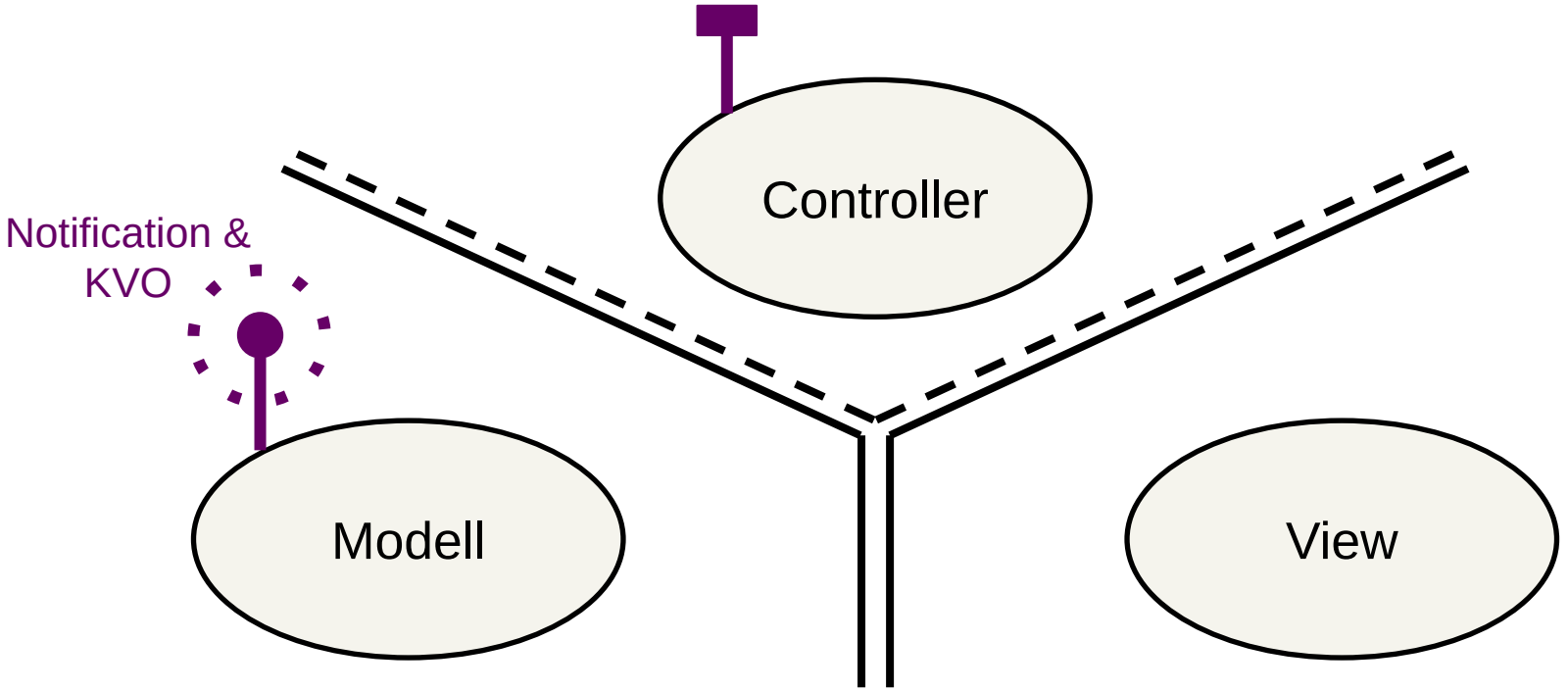
Wir suchen Ideen für die Praxisphase!

Das heißt:

- Eure Ideen sind gefragt!
- Vorstellen der Ideen in 5-minütigen Präsentationen
- Vergabe der Themen mit Beginn der Programmierphase

NOTIFICATIONS & KVO

Zur Erinnerung: MVC bzgl. Notifications



Wie werden Modifikationen an den Daten des Modells dem Controller mitgeteilt bzw. in der View aktualisiert?

- Verwendung eines Broadcast Mechanismus (**Notifications und Key-Value-Observing (KVO)**)
- Controller „lauschen“ nach interessanten Nachrichten bzw. Veränderungen

Notifications vs. Message Passing

Normalerweise werden Informationen zwischen Objekten über Message Passing ausgetauscht. Dies erfordert jedoch, dass sich die Objekte „kennen“. Oft will man diese enge Kopplung vermeiden => Notification Mechanismus

- Objekte „broadcasten“ Informationen
- Andere Objekte beobachten Broadcasts und können darauf reagieren (oder auch nicht)

Notifications...

- können von beliebigen Objekten verarbeitet werden
- können von beliebig vielen Objekten verarbeitet werden (auch keinem)
- liefern dem Sender keinen Rückgabewert

Notifications: NSNotificationCenter

Beispiele für typische Notifications:

- UIDeviceOrientationDidChangeNotification
- UIDeviceBatteryStateDidChangeNotification
- UIApplicationDidReceiveMemoryWarningNotification
- …

Wir nutzen das NSNotificationCenter

- realisiert Observable-Pattern
- verwaltet Observer
- Verschickt Nachrichten über Änderungen

Objekte müssen sich als Observer bei NSNotificationCenter anmelden.

Notifications: Anmelden und behandeln


```
//MeinObjekt als Observer beim NSNotificationCenter anmelden

// Default Center holen
NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];

[nc addObserver:self  //das aktuelle Objekt wird Observer
       selector:@selector(retrieveMessage:) //retrieveMessage: für Empfang
           name:@"HundEntlaufen"  //wenn "HundEntlaufen" gepostet wurde
            object:nil ]; //von jedem beliebigen Objekt
```

nil gilt als wildcard: Wenn *name* und *object* nil sind, bekommen wir jede Notification

```
//Methode implementieren, die beim Aufruf der Notification ausgeführt wird

- (void)retrieveMessage:(NSNotification *)note //NSNotification als Argument
{
   id absender = [note object];
   NSString *name = [note name];
   NSDictionary *extraInformation = [note userInfo];
   NSLog(@"%@", [name description]);
}
```

Notifications: Posten und abmelden


```
// Notifications verschicken

NSNotification *note = [NSNotification notificationWithName:@"HundEntlaufen"
                                                     object:self
                                                   userInfo:nil];

[[NSNotificationCenter defaultCenter] postNotification:note];
```

```
// MeinObjekt: Abmelden nicht vergessen!

- (void)dealloc
{
   [[NSNotificationCenter defaultCenter] removeObserver:self];
   [super dealloc];
}
```

Notifications: Notification Queues

- Aufruf von `postNotification:` des NSNotificationCenters ist synchron, d.h. blockiert, bis alle Observer-Methoden abgearbeitet sind
- NSNotificationQueues bieten diesbzgl. bessere Einflussmöglichkeiten:
  - Zeitpunkt bestimmen => `enqueueNotification:postingStyle:`
    - NSPostASAP: Am Ende des aktuellen Run-Loops
    - NSPostWhenIdle: Wenn der Run-Loop im Wartezustand ist
    - NSPostNow: Sofort
  - Gleiche Notifications zusammenfassen => `enqueueNotification:postingStyle:coalesceMask:forModes:`
    - NSNotificationNoCoalescing: Kein Zusammenfassen
    - NSNotificationCoalescingOnName: Notifications mit gleichem Namen zusammenfassen
    - NSNotificationCoalescingOnSender: Notifications vom selben Objekt zusammenfassen

Alternative: KVO

Key-Value-Observing (KVO) erlaubt es, über Änderungen an Instanzvariablen informiert zu werden.

NSObject implementiert KVO

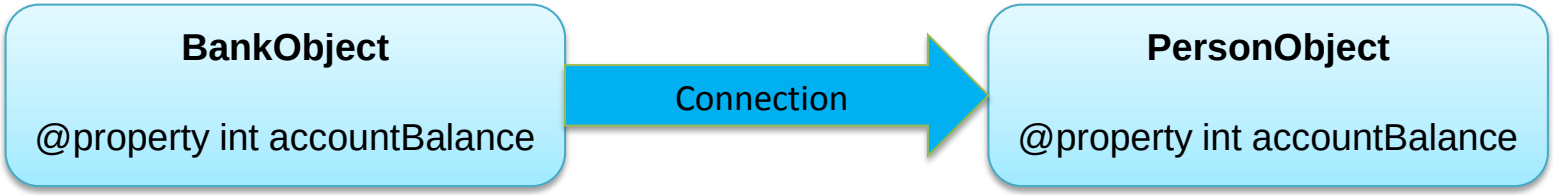
Kann automatisch verwendet werden, solange Namenskonventionen befolgt werden!

KVO – Ein Beispiel

1. Schritt: Hinzufügen eines Beobachters, der über Änderungen einer entsprechenden Instanzvariablen informiert wird


```
[bankInstance addObserver:personInstance
    forKeyPath@“accountBalance“
    options:NSKeyValueObservingOptionNew
    context:NULL];
```

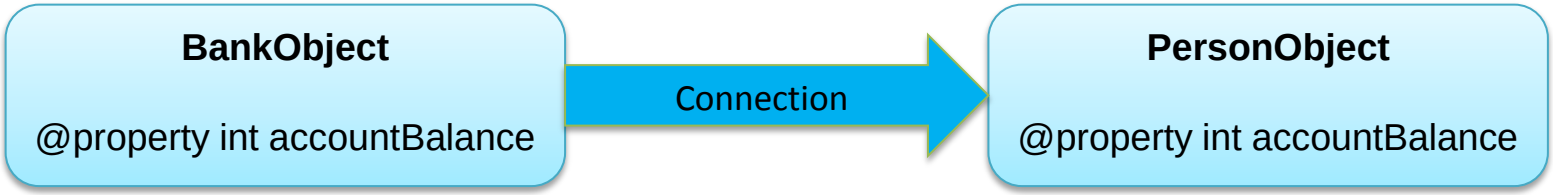
KVO – Ein Beispiel



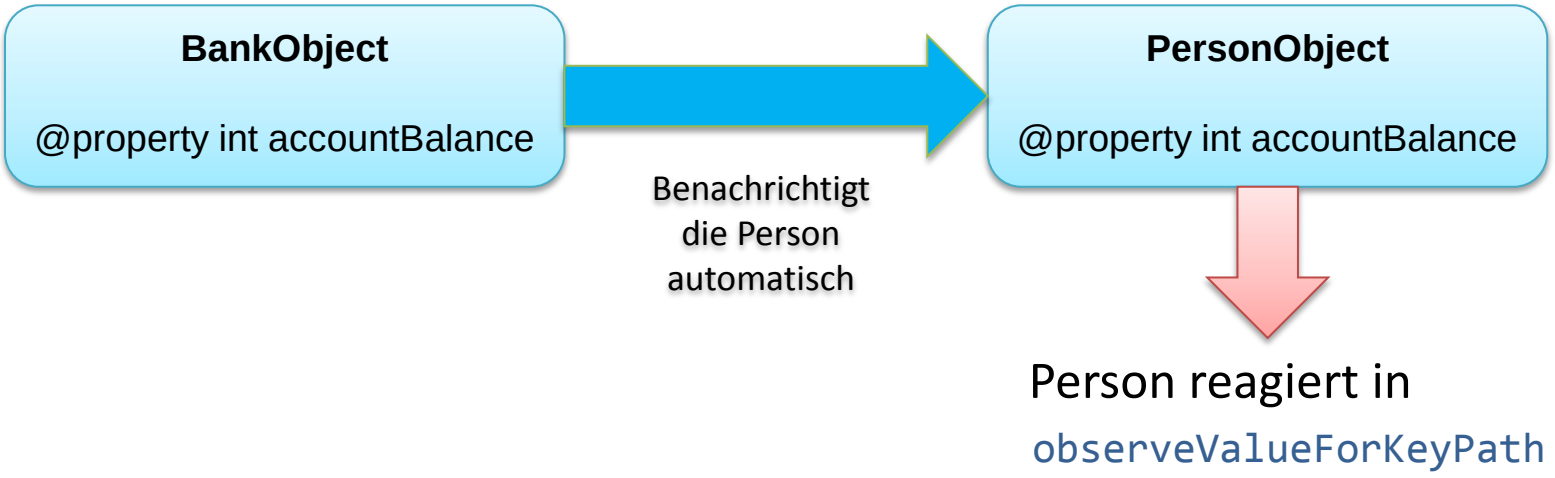
2. Schritt: Der Observer muss das NSKeyValueObserving Protocol implementieren!

```
// In PersonObject:

-(void) observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
    change:(NSDictionary *)change context:(void *)context
{
    // Custom Implementation
    if ([keyPath isEqualToString:@“accountBalance"] ) {
        // do something… e.g. update accountBalance Information
    }
}
```

KVO – Ein Beispiel

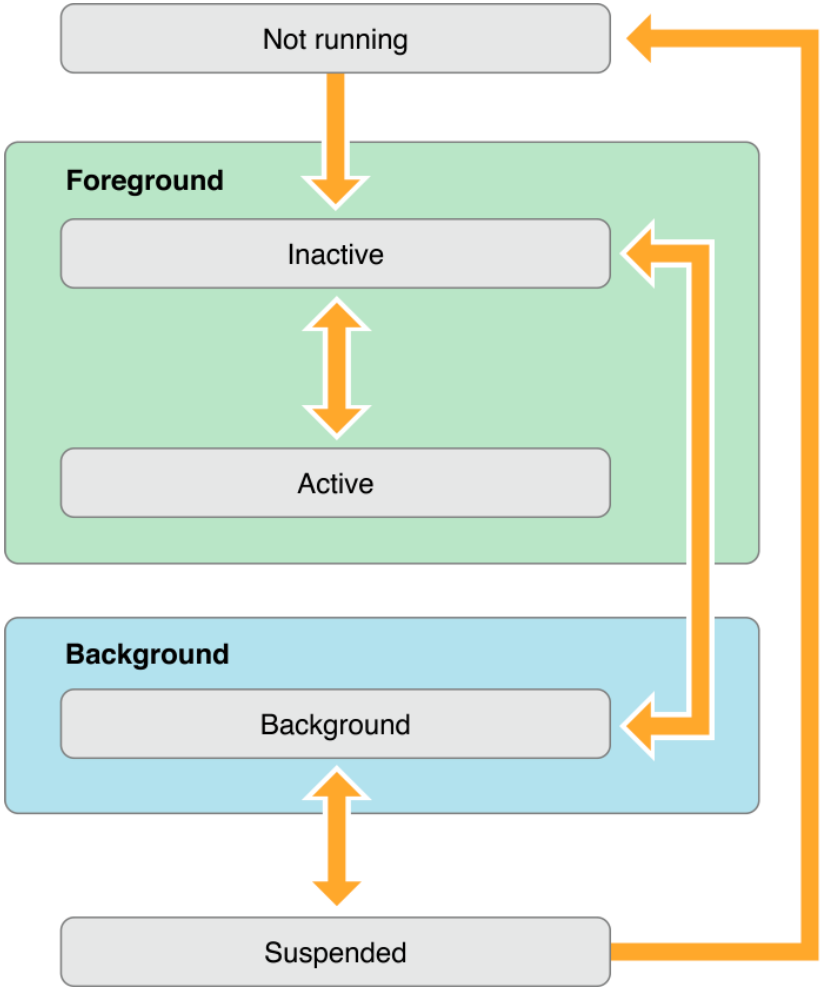



```
//Änderung der Instanzvariablen von BankObject

[bankInstance setAccountBalance:50];
// oder self.accountBalance = 50;
```



Einschub: Zustände von iOS Apps

Zustände von iOS Anwendungen



Quelle: developer.apple.com

Zustände von iOS Anwendungen

Zustand	Sichtbar	Empfängt Events	Führt Code aus
Not Running	Nein	Nein	Nein
Active	Ja	Ja	Ja
Inactive	Meistens	Nein	Ja
Background	Nein	Nein	Ja
Suspended	Nein	Nein	Nein

Anwendungen können in den inactive-Zustand gebracht werden, indem der Bildschirm gesperrt wird (Lock-Button).

Quelle: iOS Programming, The Big Nerd Ranch Guide

Archiving

Das *NSCoding* Protokoll

Motivation

Objektinstanzen sollen in bestimmten Fällen persistent auf dem Dateisystem abgespeichert werden können.

Dies trifft meistens auf Instanzen eines Modell Objects zu, da diese sich um die Datenhaltung kümmern.

Durch Archiving können nun ganze Instanzen und damit komplette Zustände einer App gespeichert werden. Bsp.:

- Spielstände
- Einstellungen
- Eingaben
- ...

Archiving: Einführung

Archiving ist eine weit verbreitete Möglichkeit, Objektinstanzen auf dem Dateisystem zu speichern

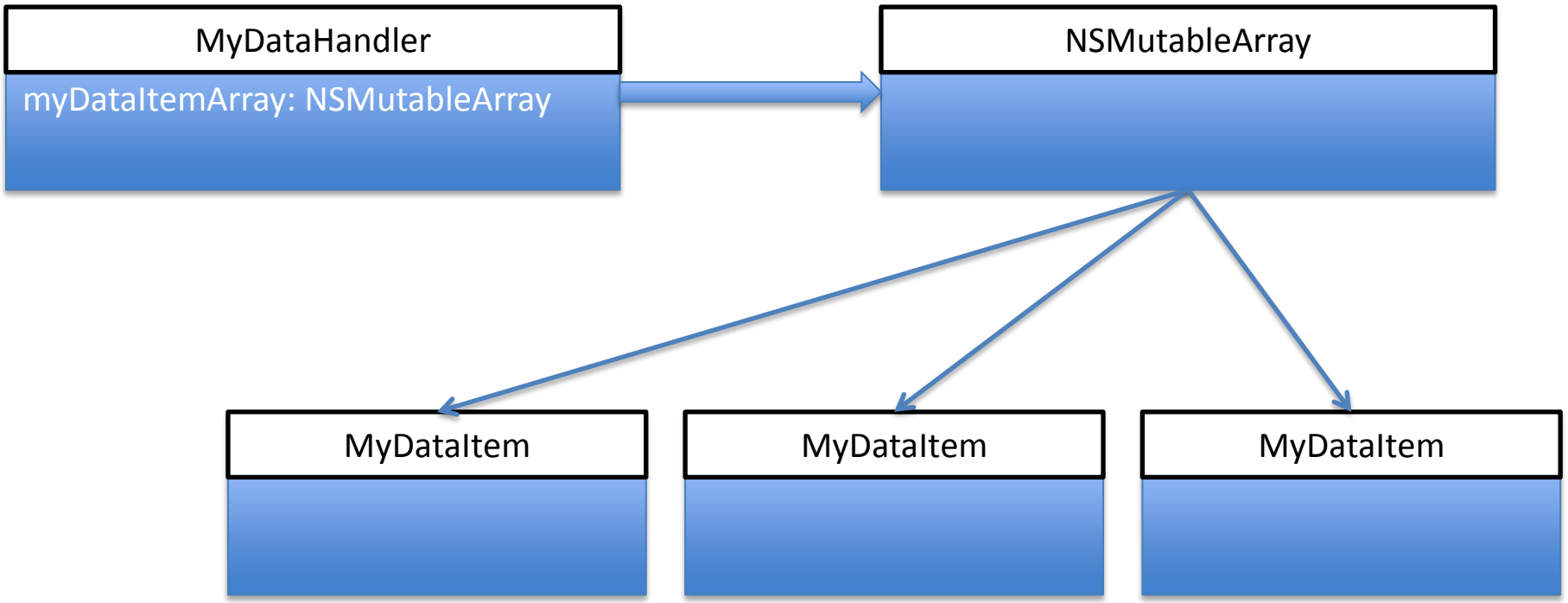
Archiving Obj bedeutet, sämtliche Instanzvariablen eines Objekts zu sammeln und auf dem Dateisystem abzulegen

Unarchiving Obj lädt die Daten aus dem Dateisystem und erzeugt daraus ein Objekt

Klassen, deren Instanzen archiviert bzw. geladen werden sollen, müssen dem *NSCoding* Protokoll entsprechen und die beiden Methoden **encodeWithCoder:** und **initWithCoder:** implementieren

Beispiel: MyDataItemArray

Der Zustand des gesamten myDataItemArrays soll archiviert werden



NSCoding Protokoll


```
// NSCoding Protocol

@protocol NSCoding
-(void)encodeWithCoder:(NSCoder *) aCoder;
-(void)initWithCoder:(NSscoder *) aDecoder;
@end


// MyDataItem.h

@interface MyDataItem: NSObject <NSCoding>
[…]
@end
```

Archiving mit encodeWithCoder:

MyDataItem.m muss nun `encodeWithCoder:` und `initWithCoder:` implementieren

Wenn MyDataItem die Nachricht `encodeWithCoder` erhält, werden die Instanzvariablen in den mitgelieferten `NSCoder` codiert.

`NSCoder` organisiert die Daten als key-value Paar

```
// MyDataItem.m

@implementation MyDataItem
-(void)encodeWithCoder:(NSCoder *) aCoder {
    [aCoder encodeInt:someValue forKey:@"someValue"];
    [aCoder encodeObject:someObj forKey:@"someObj"];
    // Auch andere Datentypen können de-/kodiert werden
    //(siehe NSCoder Dokumentation)
}
@end
```

Archiving mit encodeWithCoder:

Die Kodierung von Objekten mit `encodeObject: forKey:` stößt einen kaskadierenden Prozess an.

Die zu kodierenden Objekte müssen ebenfalls *NSCoding*-konform sein

Die Methoden `encodeWithCoder:` und `initWithCoder:` müssen also wiederum überschrieben werden

Es kann z.B. (wie in unserem Fall) einfach eine Variable vom Typ `NSMutableArray` archiviert werden, dessen zugehöriges Array nur *NSCoding*-konforme Elemente enthält. Bsp.:

- myDataItemArray enthält nur myDataItem Elemente (NSCoding-konform)

Anmerkung: XIB-Files werden über Archiving gewonnen bzw. gespeichert.

- UIView ist NSCoding-konform und wird beim App-Launch durch unarchiving der XIB-File hergestellt

Unarchiving mit initWithCoder:

Gespeicherte Objekte können durch `initWithCoder:`-Nachricht wieder aus dem Speicher geladen werden

Die Methode soll alle in `encodeWithCoder:` kodierte Objekte sammeln, dekodieren und den entsprechenden Instanzvariablen zuweisen

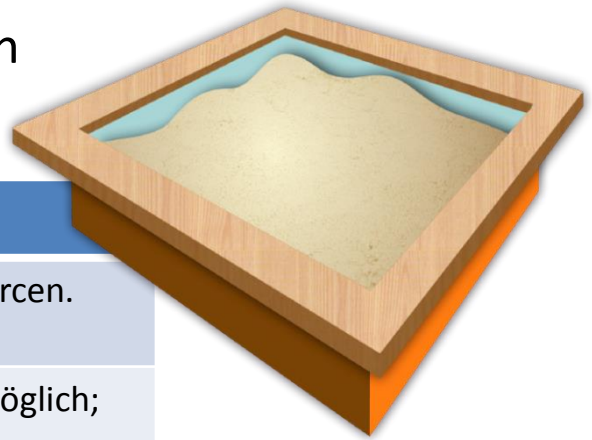
```
// MyDataItem.m
[...]
-(id)initWithCoder:(NSCoder *)aDecoder {
    self = [super init];
    if(self){
        [self setSomeValue:[aDecoder decodeIntForKey:@"someValue"]];
        [self setSomeObj:[aDecoder decodeObjectForKey:@"someObj"]];
    }
    return self;
}
@end
```

Archiving – Wo werden Daten abgelegt?

Eine Sandbox ist ein abgeschotteter Teil des Dateisystems, der nur für die dazugehörige App zugänglich ist.

Apps können Daten nur in ihrer eigenen Sandbox ablegen

Andere Apps erhalten keinen Zugang



Pfad	Bedeutung
AppName.app	Das App-Bundle. Enthält die App und alle Resourcen. (read-only)
Documents/	Persistente Nutzerdaten; Zugriff durch Nutzer möglich; wird mit iTunes synchronisiert
Documents/Inbox	Dateien, die andere Apps der App zum Öffnen zur Verfügung gestellt haben; Dateien können nur gelesen und gelöscht werden, nicht geschrieben; wird mit iTunes synchronisiert
Library/	Persistente Daten, die für den Nutzer nicht zugänglich sein sollen; wird mit iTunes synchronisiert (außer „Caches“-Unterordner)
tmp/	Temporäre Dateien gehören hierhin. Dateien sollten wieder gelöscht werden, da sie auch von iOS gelöscht werden können, wenn die App nicht aktiv ist.

Pfade in einer Sandbox

`NSSearchPathForDirectoriesInDomains:` erzeugt einen Pfad innerhalb einer Sandbox:

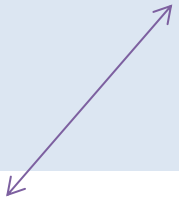
Folgende Funktion erzeugt den Pfad *Documents/dataItems.archive*

```
-(NSString *)pathForDataItems {
    NSArray *docDirs =
        NSSearchPathForDirectoriesInDomains(
            NSDocumentDirectory, NSUserDomainMask, YES);
                    // Funktion ursprgl. von OS X. Array wegen OS X. Beiden
                    // letzten Argumente immer gleich in iOS

        NSString *docDir = [docDirs objectAtIndex:0];
            // Bei iOS immer nur 1       Pfad

        return [docDir stringByAppendingPathComponent:@"dataItems.archive"];
}
```

Daten speichern mit NSKeyedArchiver

- Um MyDataItems zu speichern nutzen wir `NSKeyedArchiver`

```
// MyDataHandler.m

- (BOOL) saveChanges
{
  // returns success or failure
  NSString *path = [self pathForDataItems]; //Pfad bestimmen

  // sendet archiveRootObject: toFile: an die NSKeyedArchiver Klasse
  return [NSKeyedArchiver archiveRootObject:myDataItemArray toFile:path];
}
```



Arrayinstanz mit myDataItem Elementen wird übergeben - solange alle im Array enthaltenen Objekte *NSCoding*-konform sind, werden sie "automatischt" serialisiert.

Was macht `archiveRootObject:` ?

Erzeugt neue Instanz von `NSKeyedArchiver`

Sendet die `encodeWithCoder:` Nachricht an das RootObject (in unserem Fall `myDataItemArray`)

`NSKeyedArchiver` ist eine Unterklasse von `NSCoder`, also kann die neue Instanz als Argument von `encodeWithCoder:` mitgegeben werden

`myDataItemArray` schickt die Nachricht `encodeWithCoder:` an alle im Array enthaltenen Objekte, die dann ihre Instanzvariablen im selben `NSKeyedArchiver` kodieren

`NSKeyedArchiver` hinterlegt die gesammelten Daten auf dem angegebenen Pfad

Fall-Beispiel: Speichern mit Archiving

Fall-Beispiel:

- Speichern soll erzwungen werden, wenn der Nutzer den Home-Button drückt

Lösung:

- saveChanges Nachricht an myDataHandler schicken in der MyAppDelegate.m `applicationDidEnterBackground:`

```
// MyAppDelegate.m
#import MyDataHandler.h

@implementation MyAppDelegate

- (void)applicationDidEnterBackground:(UIApplication *)app {
    BOOL success = [[MyDataHandler instance] saveChanges];
    if(!success) {
        NSLog(@"Could not save MyDataHandler files");
    }
}
```

Daten laden mit `NSKeyedUnarchiver` - MyDataHandler.m

Laden der Daten analog mit `unarchiveObjectWithFile:`-Nachricht auf `NSKeyedUnarchiver`

```
// MyDataHandler.m
- (id) init
{
  self = [super init];
  if(self) {
    NSString *path = [self pathForDataItems];
    myDataItemArray = [NSKeyedUnarchiver unarchiveObjectWithFile:path];

    if (!myDataItemArray) {
      //wichtig, falls noch keine Datei exisitiert
      myDataItemArray = [[NSMutableArray alloc] init];
    }
  }
  return self;
}
```

Was macht `unarchiveObjectWithFile:` ?

Erzeugt neue Instanz von `NSKeyedUnarchiver`

Lädt das Archiv, das am angegebenen Pfad liegt

`NSKeyedUnarchiver` prüft den Typ des RootObjekts und erzeugt eine Instanz davon (In unserem Fall `NSMutableArray`)

Die erzeugte Instanz von `NSMutableArray` erhält Nachricht von `initWithCoder:` mit dem Argument von `NSKeyedUnarchiver`

Das Array beginnt mit dem Dekodieren seines Inhalts, indem allen Objekten die Nachricht `initWithCoder:` geschickt wird

Binäres Speichern

NSData

C-Funktionen

Zum binären Speichern (z.B. von Bildern, Dateien usw.) stehen unter anderem übliche C-Funktionen zur Verfügung:

```
FILE *inFile = fopen("myfile", "r");
char *buffer = malloc(gimmeSome);
fread(buffer, byteCount, 1, inFile);

FILE *outFile = fopen("myfile", "w");
fwrite(buffer, byteCount, 1, outFile);
```

writeToFile:atomically: auf NSData

Beispiel: Mit NSData Bilder speichern.

```
- (void) saveImage:(UIImage *)i forKey:(NSString *)s {
    // Create full path for image
    NSString *imagePath = [self imagePathForKey:s];
    // Turn image into JPG Data
    NSData *d = UIImageJPEGRepresentation(i, 0.5);
    // Write it to full path
    [d writeToFile:imagePath atomically:YES];
}
```

Analog zu pathForDataItems: im Archiving-Kapitel

Speichern mit `writeToFile:atomically:`

Mit `writeToFile:atomically:` können Objekte gespeichert werden.

`atomatically`: Datei wird temporär gespeichert und anschließend umbenannt => Schutz vor Datenverlust

`writeToFile:atomically`: steht in den folgenden Klassen zur Verfügung:

- `NSData`
- `NSString`
- `NSDate`
- `NSNumber`
- `NSDictionary`
- `NSArray`

Bilder löschen

Beispiel: Mit `NSData` gespeicherte Bilder löschen.

```
- (void) removeImage:(UIImage *)i forKey:(NSString *)s {
    NSString *imagePath = [self imagePathForKey:s];
    [[NSFileManager defaultManager] removeItemAtPath:imagePath
                                               error:NULL];
}
```

Beispiel: Mit `NSString` Text speichern.

```
NSError *err;
NSString *str = @"Text to save";
BOOL success = [str writeToFile:@"/path/to/file“
                     atomically: YES
                       encoding:NSUTF8StringEncoding
                          error:&err];
if(!success) {
    NSLog(@"Error: %@", [err localizedDescription];
}
```

writeToFile:atomically: auf NSString

Beispiel: Mit `NSString` gespeicherten Text laden.

```
NSError *err;
NSString *str =
    [[NSString alloc] initWithContentsOfFile:@"/path/to/file„
                                    encoding:NSUTF8StringEncoding
                                       error:&err];
if(!str) {
    NSLog(@"Error: %@", [err localizedDescription];
}
```

Speichern von `NSDictionarys` (oder `NSArrays`) erzeugt eine XML property list:

```
NSMutableDictionary *d = [NSMutableDictionary dictionary];
[d setObject:@"My String data" forKey:@"StrKey"];
[d writeToFile:@"/path/to/file" atomically:YES];
```

```
xml …
<plist version="1.0">
<dict>
  <key>StrKey</key>
  <string>My String data</string>
</dict>
</plist>
```

Zum Weiterlesen

Mehr zu Property Lists im "Property List Programming Guide":

http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/PropertyLists/Introduction/Introduction.html

Archiving vs. Serialisation

Archiving vs. Serialization

Archiving:

- Ein Archiv bietet die Möglichkeit, Objekte und Werte in einen architekturunabhängigen Byte-Strom zu konvertieren
- Die Identität und die Beziehungen zwischen den Objekten und Werten **bleibt dabei erhalten**!
- Gesamter Objektgraph kann direkt aus dem entsprechenden Byte-Strom wieder hergestellt werden

- Vorgehen:
  - Implementierung des `NSCoding`-Protokolls (`encodeWithCoder:`, `initWithCoder:`) für die Klasse der zu archivierenden Objekte
  - Rekursion erfolgt über Verwendung der Methoden (z.B. `encodeObject:forKey:`) des verwendet Coders (Unterklasse von `NSCoding`)
  - Anwendung von Methoden (z.B. `archiveRootObject:toFile:`) der Klasse `NSKeyedArchiver`) zur Erstellung und Speicherung eines Archivs

Archiving vs. Serialization

Serialization:

- Bei der Serialisierung werden Objective-C Typen in einen / von einem architekturunabhängigen Byte-Strom konvertiert / wieder hergestellt.
- Typen und Beziehungen zwischen den Objekten **werden nicht gespeichert**!
- Es ist die Aufgabe des Programmierers Daten bei der Deserialisierung wieder korrekt herzustellen

- Vorgehen
  - Verwendung von Methoden (z.B. `dataWithPropertyList:format:options:error:` bzw. `propertyListWithData:format:options:error:`) der Klasse `NSPropertyListSerialization` zur De-/Serialisierung der entsprechenden Datenstruktur
  - Das Speicherformat (binär oder XML) kann über den Parameter `format` spezifiziert werden.

Archiving vs. Serialization

Unterschiede:

- Archiving:
  - Archive erhalten die Identität und alle Beziehungen zwischen den Objekten eines Objektgraphen
  - Anstatt ein Objekt für jede auf dieses Objekt verweisende Referenz mehrfach zu kodieren, werden die Referenzen auf das Objekt archiviert

- Serialization:
  - Es werden nur die Werte der Objekte und ihre Position in der Hierarchie gespeichert
  - Es wird nicht gespeichert, ob es sich bei Container-Objekten (z.B. `NSArray` oder `NSDictionary`) um mutuable oder immutable Versionen handelt
  - Es wird nicht gespeichert, ob eine Objekt im Objektgraphen mehrfach referenziert wird. Bei der Deserialisierung werden solche Objekte mehrfach erzeugt.

Archiving vs. Serialization

Implementierungsdetails:

- Jedes Objekt, das das `NSCoding`-Protokoll implementiert, kann archiviert werden
- Serialisiert werden können hingegen nur Instanzen der Klassen `NSArray`, `NSDictionary`, `NSString`, `NSDate`, `NSNumber` und `NSData` (und einige ihrer Unterklassen)

Wann sollte man was verwenden?

- Serialization:
  - Wenn es sich um Property Lists handelt, die primär Strings und Zahlen beinhalten (z.B. `NSUserDefaults` oder pList)
  - Ineffizient für große Blöcke binärer Daten
- Archiving:
  - Immer wenn (Teil-)Objektgraph gespeichert werden soll

NSUserDefaults

Zum Speichern von Einstellungen

NSUserDefaults

`NSUserDefaults` speichert **Einstellungen** über das Programmende hinweg.

`NSUserDefaults` speichert Daten in key-value-Paaren ähnlich einer HashMap in Java.

Die Schlüssel (keys) sind immer vom Typ `NSString` und sollten als Variable mit `const` deklariert werden.

```
NSString * const MyAppViewTypePrefKey = @"MyAppViewTypePrefKey"
```

NSUserDefaults

Jede App hat Zugriff auf eine `NSUserDefaults`-Instanz.

`[NSUserDefaults standardUserDefaults]` liefert diese Instanz.

Einstellungen werden unter *Library/Preferences/* gespeichert.

Alle Präferenzen, die der Nutzer setzt werden in der *application domain* gespeichert.

Default Values werden in der *registration domain* gespeichert.

Die gespeicherten Daten werden beim ersten Aufruf von `standardUserDefaults` automatisch geladen.

NSUserDefaults

Speichern von Werten und/oder ganzen Objekten mit

- `(void) setBool:(BOOL) forKey:(NSString *)`
- `(void) setInteger:(NSInteger) forKey:(NSString *)`
- `(void) setObject:(id) forKey:(NSString *)`

Beispiel:

```
[[NSUserDefaults standardUserDefaults] setBool:YES
    forKey:MyAppUserLeftWithoutLoggingOutPrefKey]
```

NSUserDefaults

Laden von Werten und/oder ganzen Objekten mit

- `(BOOL) boolForKey: (NSString *)`
- `(NSInteger) integerForKey: (NSString *)`
- `(id) objectForKey: (NSString *)`

Beispiel:

```
Bool userLeftWithoutLoggingOut =
    [[NSUserDefaults standardUserDefaults]
        boolForKey:MyAppUserLeftWithoutLoggingOutPrefKey];
```

`NSUserDefaults` und Standardeinstellungen

`registerDefaults`: belegt Settings mit initialen Werten in *registration domain*.

Falls *application domain* leer ist, wird in der *registration domain* nach Default Werten gesucht

Typischerweise wird `registerDefaults`: in der `initialize`-Methode einer Klasse gesendet (Klassenmethode!), so wird verhindert, dass Objekte vorher auf eine Instanz von `NSUserDefaults` zugreifen

```
+ (void) initialize {
    NSDictaionary *defaults =
        [NSDictionary dictionaryWithObject:[NSNumber numberWithBool:NO]
        forKey:MyAppUserLeftWithoutLoggingOutPrefKey];

    [[NSUserDefaults standardUserDefaults] registerDefaults:defaults];
}
```

NSUserDefaults und Settings

Jede iOS App besitzt eine Settings Application

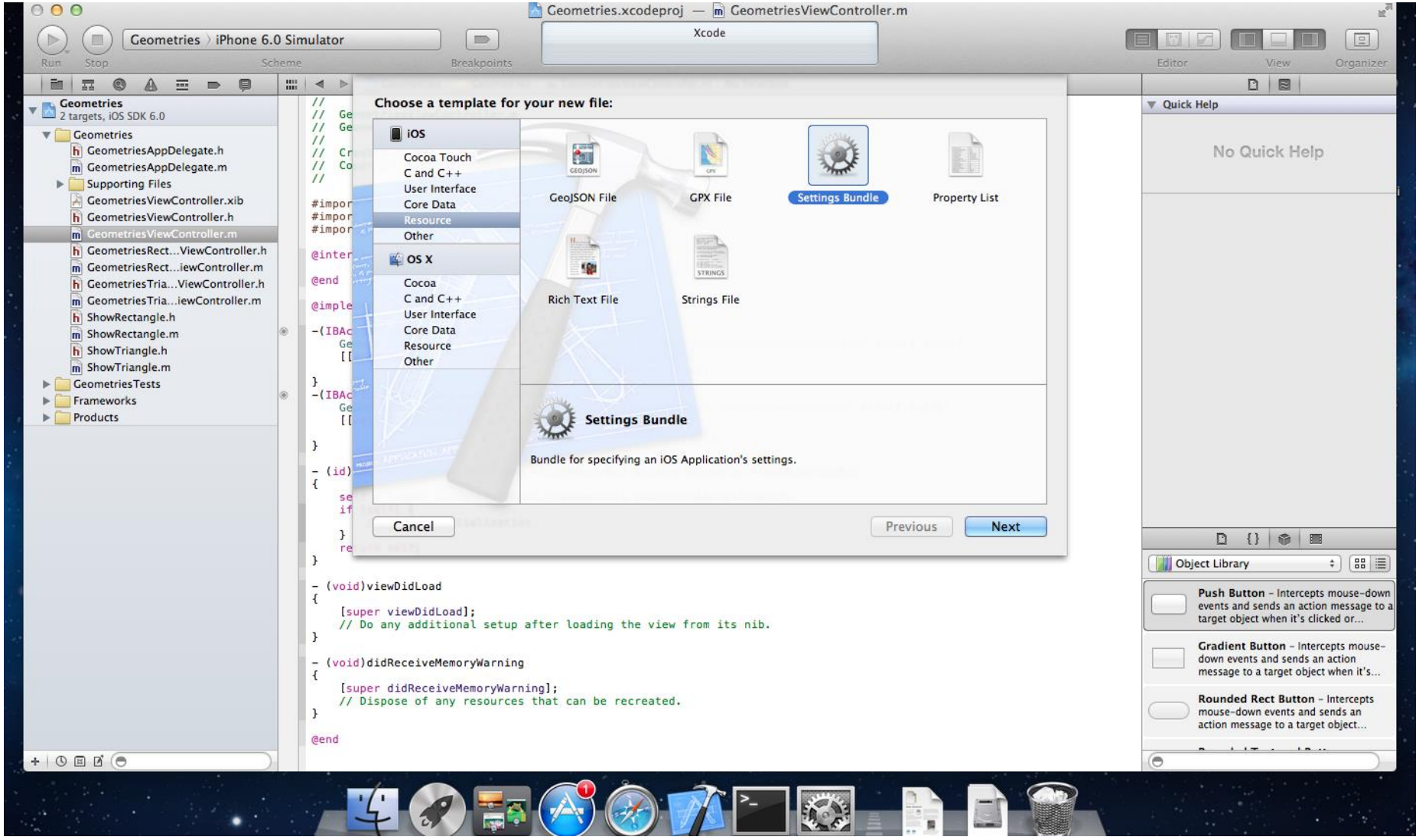
Settings, die mit `NSUserDefaults` gespeichert werden, können für die Settings-App freigegeben werden.

Settings.bundle ist eine property list, die alle freigegebenen Settings enthält.

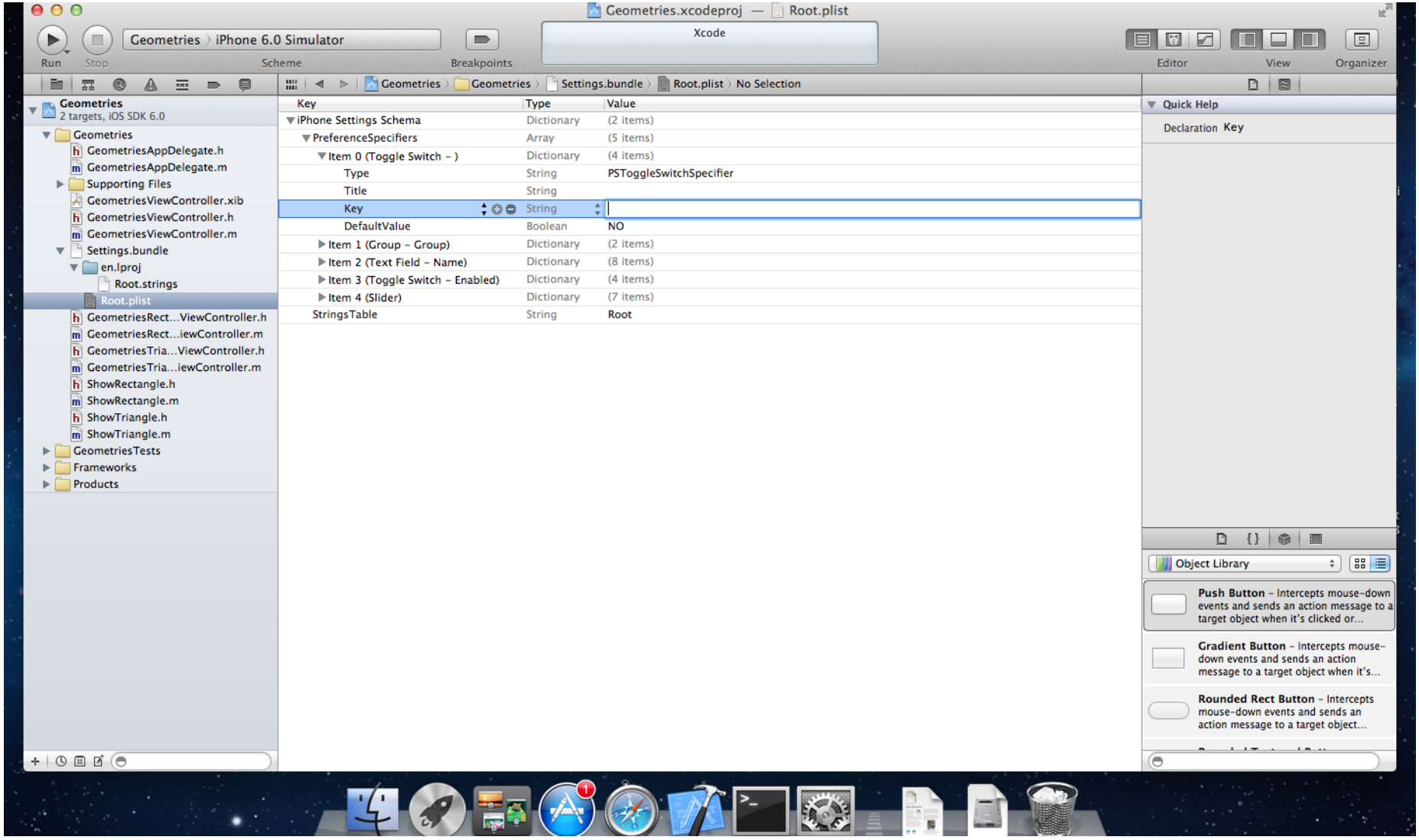
LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

NSUserDefaults und Settings

mobile and distributed systems group



NSUserDefaults und Settings



NSUserDefaults und Settings

Wichtig: Zustand der eigenen App, wenn Einstellungen über Settings-App geändert werden!

Zwei Fälle:

1. Anwendung wurde vorher beendet (terminated):
   Änderungen über Settings-App stehen beim nächsten Start der App automatisch zur Verfügung

2. Anwendung wurde in den Hintergrund verschoben (suspended):
   Anwendung wird über eine `NSUserDefaultsDidChangeNotification` benachrichtigt, dass sich Settings geändert haben