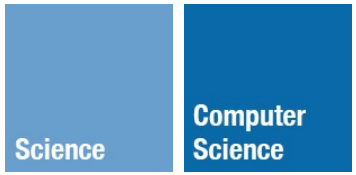
Persistence



CS 442: Mobile App Development
Michael Saelee <lee@iit.edu>

Things to persist

- Application settings
- Application state
- Model data
- Model relationships

Persistence options

- User defaults
- Property lists serialization
- Object archives
- SQLite database
- Core Data
- Apple frameworks: Address Book, Photos, iCloud, etc.
- Roll-your-own

§ User Defaults

`NSUserDefaults`

- encapsulates access to global/app-specific user “defaults”
  - i.e., system/application preferences
- glorified NSDictionary

Different “domains” for preferences:

- application domain (persistent)
- global domain (persistent)
- “registration” (volatile)

when multiple settings for the same key, search in this order by default

Persistent domains save/restore settings across app-launches

Getting defaults object:

```
NSUserDefaults.standardUserDefaults()
```

by default, set up with default domain search order — will persist to app domain

```
@interface NSUserDefaults : NSObject {
+ (NSUserDefaults *)standardUserDefaults;

- (void)registerDefaults:(NSDictionary *)registrationDictionary;

- (id)objectForKey:(NSString *)defaultName;
- (NSString *)stringForKey:(NSString *)defaultName;
- (NSInteger)integerForKey:(NSString *)defaultName;
- (BOOL)boolForKey:(NSString *)defaultName;
- (NSURL *)URLForKey:(NSString *)defaultName;

- (NSDictionary *)dictionaryRepresentation;

- (void)setObject:(id)value forKey:(NSString *)defaultName;
- (void)setInteger:(NSInteger)value forKey:(NSString *)defaultName;
- (void)setBool:(BOOL)value forKey:(NSString *)defaultName;
- (void)setURL:(NSURL *)url forKey:(NSString *)defaultName;

- (BOOL)synchronize;

...
@end
```

User Defaults API

Typical workflow:

- register default values in registration domain on app launch
- retrieve/set user customized settings in app domain during use

```swift
func application(application: UIApplication, didFinishLaunchingWithOptions
    launchOptions: [NSObject: AnyObject]?) -> Bool {
  // default "registered" settings (not persisted)
  NSUserDefaults.standardUserDefaults().registerDefaults([
    "setting1": true,
    "setting2": "val2",
    "setting3": 100
    ])
  return true
}
```

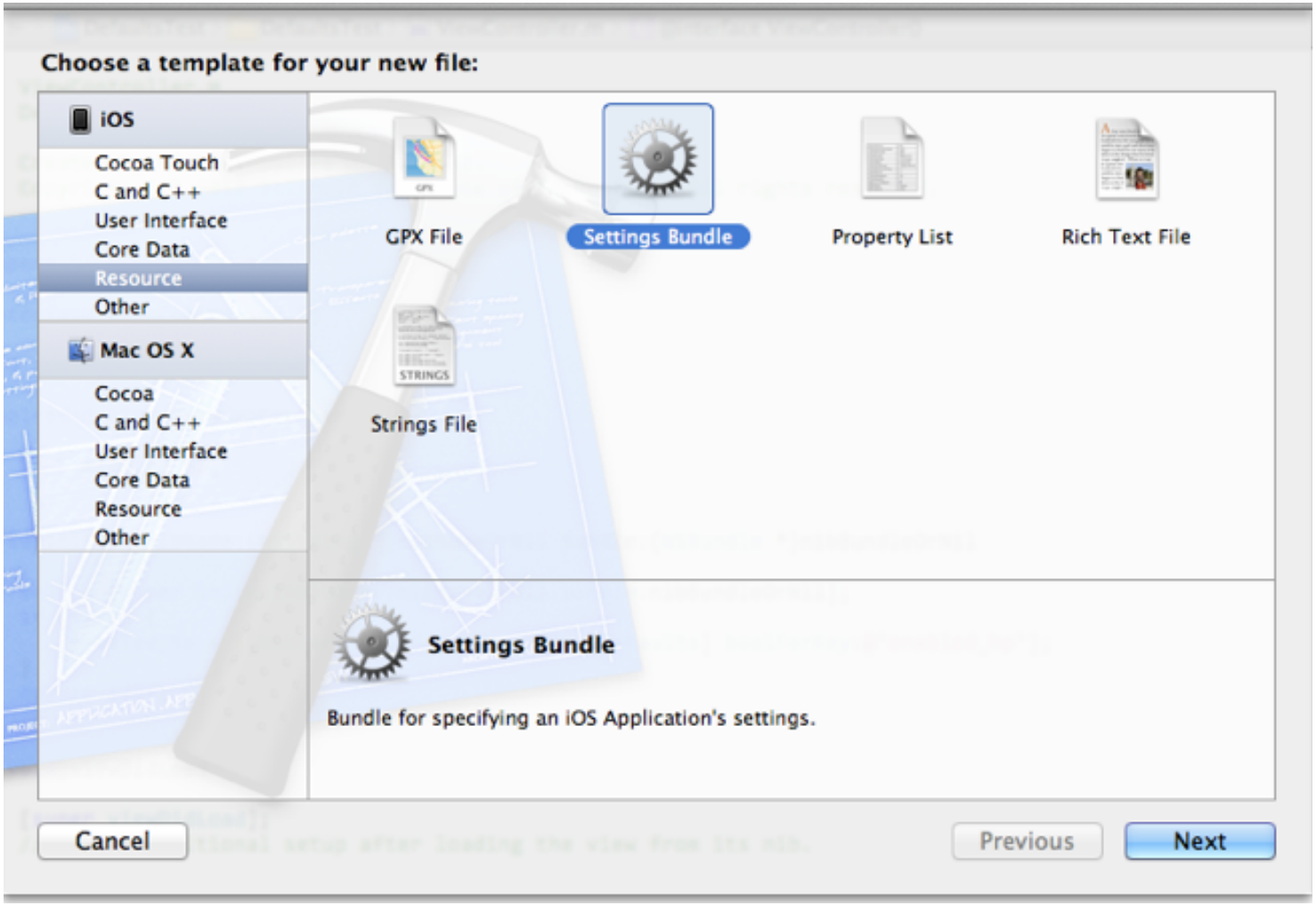
registering defaults

```
class ViewController: UIViewController {
  var some_setting: Bool

  required init(coder aDecoder: NSCoder) {
    some_setting = NSUserDefaults.standardUserDefaults().boolForKey("setting1")
    super.init(coder: aDecoder)
  }

  @IBAction func toggleSetting(sender: AnyObject) {
    some_setting = !some_setting
    NSUserDefaults.standardUserDefaults().setBool(true, forKey: "setting1")
    NSUserDefaults.standardUserDefaults().synchronize() // not strictly needed
  }
}
```

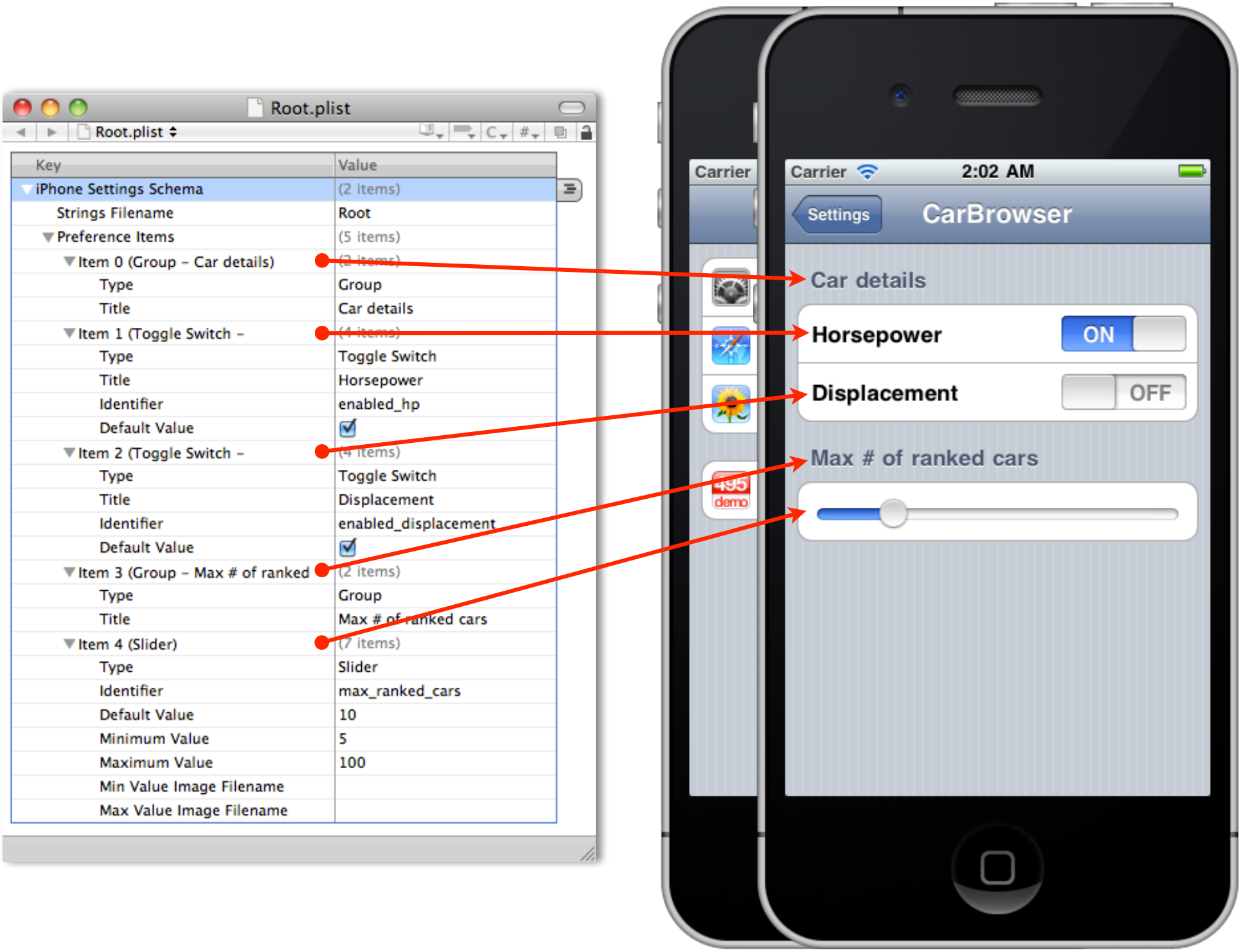
reading/setting defaults

for rarely changed top-level settings, may want to expose them in “Settings” app


Carrier
9:14 AM
Photos
Settings
Game Ce
Contacts
Safari
Carrier
9:14 AM
Settings
General
Safari
Photos
Carrier
9:15 AM
Settings
Safari
General
Search Engine
Google
AutoFill
Off
Security
Fraud Warning
ON
Warn when visiting fraudulent websites.
JavaScript
ON
Block Pop-ups
ON
Accept Cookies
From visited



Xcode settings bundle


Root.plist
Root.plist
Key
Value
iPhone Settings Schema
(2 items)
Strings Filename
Root
Preference Items
(5 items)
Item 0 (Group – Car details)
Type
Group
Title
Car details
Item 1 (Toggle Switch –
Type
Toggle Switch
Title
Horsepower
Identifier
enabled_hp
Default Value
Item 2 (Toggle Switch –
Type
Toggle Switch
Title
Displacement
Identifier
enabled_displacement
Default Value
Item 3 (Group – Max # of ranked
(2 items)
Type
Group
Title
Max # of ranked cars
Item 4 (Slider)
(7 items)
Type
Slider
Identifier
max_ranked_cars
Default Value
10
Minimum Value
5
Maximum Value
100
Min Value Image Filename
Max Value Image Filename
Carrier
Carrier
2:02 AM
Settings
CarBrowser
Car details
Horsepower
ON
Displacement
OFF
Max # of ranked cars

concurrent modification?

— one solution is to call `synchronize` defensively

… messy, and not-robust

```
required init(coder aDecoder: NSCoder) {
  NSNotificationCenter.defaultCenter().addObserver(self,
    selector: "defaultsChanged:",
    name: NSUserDefaultsDidChangeNotification,
    object: nil)

  super.init(coder: aDecoder)
}


func defaultsChanged(notification: NSNotification) {
  let defaults = notification.object as NSUserDefaults
  let newDefault = defaults.boolForKey("setting1")
}
```

NSNotificationCenter

§ Property Lists
(a.k.a. plists)

built-in serialization of supported objects

“property list types”

array, dict, string, date, number, boolean

can read/write “root” array or dict as plist

```
var data = NSArray(contentsOfFile: "pathToFile").mutableCopy()

// modify array

data.writeToFile("pathToUpdatedFile", atomically: true)
```

more fine-grained control:

NSPropertyListSerialization

(serialize plist object to byte stream)

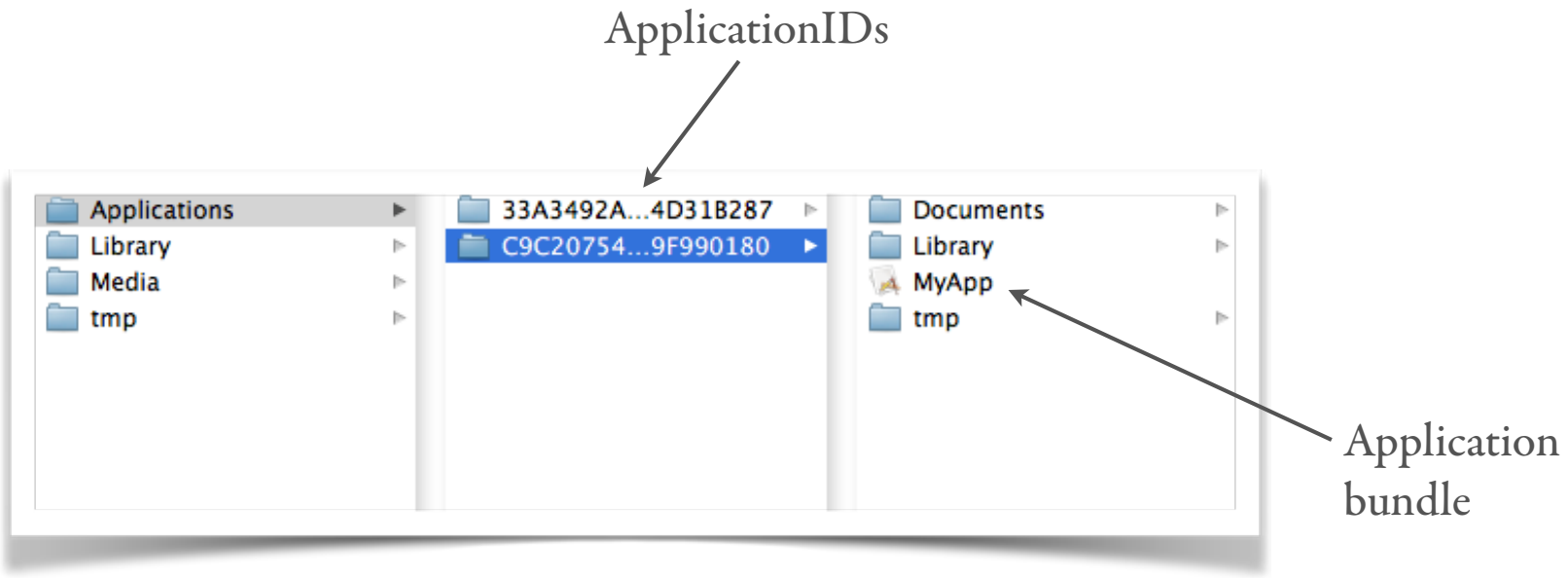
```
+ (NSData *)dataWithPropertyList:(id)plistObj
                          format:(NSPropertyListFormat)format
                         options:(NSPropertyListWriteOptions)opt
                           error:(NSError **)error
```

(deserialize plist object from byte stream)

```
+ (id)propertyListFromData:(NSData *)data
          mutabilityOption:(NSPropertyListMutabilityOptions)opt
                    format:(NSPropertyListFormat *)format
          errorDescription:(NSString **)errorString
```

read/write paths?

recall: “sandboxed” filesystem




```
// get path to home directory
homePath = NSHomeDirectory();

// get path to tmp directory
tmpPath = NSTemporaryDirectory();

// get path to Documents directory
paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);
documentsDirectory = paths[0];
```

`AppBundle/`	application bundle built by Xcode; signed and not writeable
`Library/`	used for auto-managed settings (defaults) and system caches
`Documents/`	writeable by running application; backed up by iTunes
`tmp/`	writeable by running application; **not** backed up by iTunes

typical workflow:

- open initial plist from app bundle
- save modified plist to Documents/
- on next launch, use modified version, if available

good for basic settings, string/numeric data;
inefficient for binary data

for complex data, or >1MB, don't use plists!

data is either *all-in* or *all-out*

§ Object Archives

archive **object graphs**

(à la nibfiles)

archivable class must adopt the `NSCoding` protocol

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

```objc
@interface Sprocket : NSObject <NSCoding> {
    NSInteger sprockId;
}
```

```objc
@implementation Sprocket

- (void)encodeWithCoder:(NSCoder *)aCoder {
    [aCoder encodeInteger:self.sprockId  forKey:@"sID"];
}

- (NSString *)description {
    return [NSString stringWithFormat:@"Sprocket[%d]", self.sprockId];
}

@end
```

```objc
@interface Widget : NSObject <NSCoding> {
@property (assign) NSInteger widgetId;
@property (strong) NSString *widgetName;
@property (assign) BOOL tested;
@property (strong) NSArray *sprockets;
@end
```

```objc
@implementation Widget
- (id)initWithCoder:(NSCoder *)aDecoder {
    if (self = [self init]) {
        self.widgetId   = [aDecoder decodeIntegerForKey:@"wID"];
        self.widgetName = [aDecoder decodeObjectForKey:@"wName"];
        self.tested     = [aDecoder decodeBoolForKey:@"wTested"];
        self.sprockets  = [aDecoder decodeObjectForKey:@"wSprockArray"];
    }
    return self;
}

- (void)encodeWithCoder:(NSCoder *)aCoder {
    [aCoder encodeInteger:self.widgetId  forKey:@"wID"];
    [aCoder encodeObject:self.widgetName forKey:@"wName"];
    [aCoder encodeBool:self.tested       forKey:@"wTested"];
    [aCoder encodeObject:self.sprockets  forKey:@"wSprockArray"];
}
@end
```

```
Sprocket *sprock1 = [Sprocket sprocketWithId:10],
         *sprock2 = [Sprocket sprocketWithId:101],
         *sprock3 = [Sprocket sprocketWithId:202],
         *sprock4 = [Sprocket sprocketWithId:333];

NSArray *sprockArr1 = @[sprock1],
        *sprockArr2 = @[sprock2, sprock3],
        *sprockArr3 = @[sprock3, sprock4];

NSArray *widgetArray = @[[Widget widgetWithId:11 name:@"Foo" tested:YES sprockets:sprockArr1],
                         [Widget widgetWithId:22 name:@"Bar" tested:YES sprockets:sprockArr2],
                         [Widget widgetWithId:33 name:@"Baz" tested:YES sprockets:sprockArr3]];


// archive object graph to file
[NSKeyedArchiver archiveRootObject:widgetArray toFile:@"widgets.archive"];


// unarchive object graph (thaw)
NSArray *unarchivedRoot = [NSKeyedUnarchiver unarchiveObjectWithFile:archiveFile];


Sprocket *sprockA = [[[unarchivedRoot objectAtIndex:1] sprockets] objectAtIndex:1],
         *sprockB = [[[unarchivedRoot objectAtIndex:2] sprockets] objectAtIndex:0];

// test object identity (evaluates to YES)
sprockA == sprockB;

// test object identity (evaluates to NO)
sprockA == sprock3;
```

keyed unarchiving allows support across different class implementations

supports multiple references to one object (true object graphs)

big problem: once again, all-or-nothing
(no swapping on iOS)

not practical for **large datasets**

§ SQLite (v3)

"an in-process library that implements a *self-contained*, *serverless*, *zero-configuration*, *transactional* SQL database engine."

¶ RDBMS crash course

relational database management systems

“relations” = tables of data

“attributes” = table columns

“records” = table rows

ID (key)	Name	Extension	Room #
A1010101	Michael Lee	x5709	SB 226A
A2020202	Cynthia Hood	x3918	SB 237E
A3030303	Bogdan Korel	x5145	SB 236B
A4040404	Matthew Bauer	x5148	SB 237B

Structured **Q**uery **L**anguage

- querying & data manipulation
- transaction management
- data definition language

inconsistent / incompatible syntax and extensions across databases

if you want to use SQLite effectively, you need to become a (SQL)ite expert

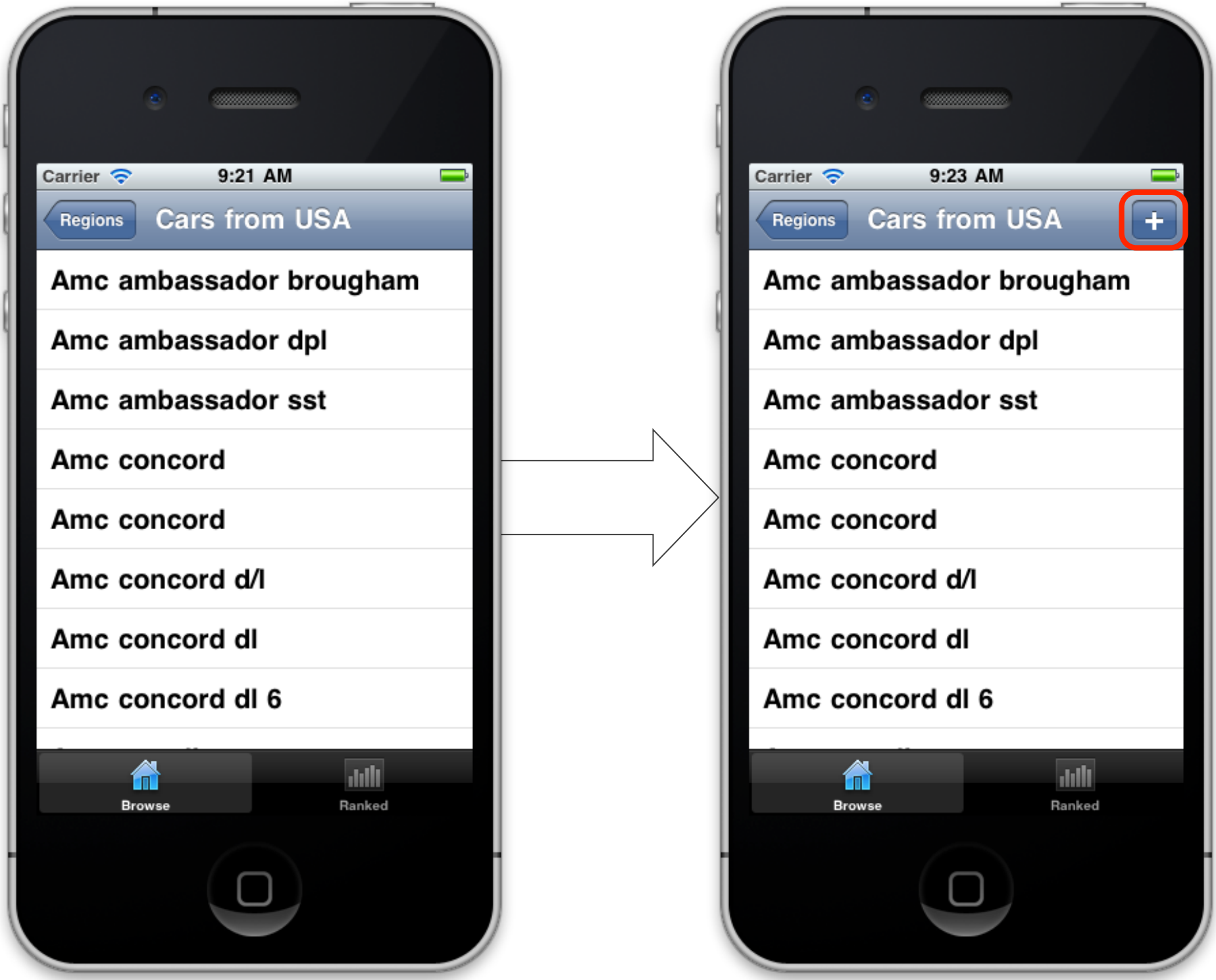
http://www.sqlite.org/docs.html

Notable SQLite features:

1. portable, single file source
2. C implementation & API
3. Most of SQL-92 standard
4. “Dynamic” SQL types

Notable missing features:

- db/table access control
- altering columns/constraints
- writing to views


Carrier
9:21 AM
Regions
Cars from USA
Amc ambassador brougham
Amc ambassador dpl
Amc ambassador sst
Amc concord
Amc concord
Amc concord d/l
Amc concord dl
Amc concord dl 6
Browse
Ranked
Carrier
9:23 AM
Regions
Cars from USA
+
Amc ambassador brougham
Amc ambassador dpl
Amc ambassador sst
Amc concord
Amc concord
Amc concord d/l
Amc concord dl
Amc concord dl 6
Browse
Ranked

```
$ sqlite3 cars.sqlite
SQLite version 3.6.12
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> CREATE TABLE cars      (id integer PRIMARY KEY,
   ...>                         name text);
sqlite> CREATE TABLE regions   (id integer PRIMARY KEY,
   ...>                         name text);
sqlite> CREATE TABLE car_stats (car_id integer,
   ...>                         mpg real,
   ...>                         cylinders integer,
   ...>                         displacement real,
   ...>                         horsepower real,
   ...>                         weight real,
   ...>                         acceleration real,
   ...>                         year integer,
   ...>                         region_id integer);
```

```
sqlite> INSERT INTO cars (name) VALUES ("Volkswagen Rabbit");
sqlite> INSERT INTO regions (name) VALUES ("Europe");
sqlite> INSERT INTO car_stats (car_id, region_id, mpg,
   ...>   cylinders, displacement, horsepower, weight,
   ...>   acceleration, year)
   ...> VALUES (1, 1, 21, 4, 2500, 170, 3072, 12, 2009);
```

cars.csv

```
1,chevrolet chevelle malibu
2,buick skylark 320
3,plymouth satellite
4,amc rebel sst
5,ford torino
6,ford galaxie 500
7,chevrolet impala
8,plymouth fury iii
9,pontiac catalina
10,amc ambassador dpl
...
```

stats.csv

```
1,18.0,8,307.0,130.0,3504,12.0,70,1
2,15.0,8,350.0,165.0,3693,11.5,70,1
3,18.0,8,318.0,150.0,3436,11.0,70,1
4,16.0,8,304.0,150.0,3433,12.0,70,1
5,17.0,8,302.0,140.0,3449,10.5,70,1
6,15.0,8,429.0,198.0,4341,10.0,70,1
7,14.0,8,454.0,220.0,4354,9.0,70,1
8,14.0,8,440.0,215.0,4312,8.5,70,1
9,14.0,8,455.0,225.0,4425,10.0,70,1
10,15.0,8,390.0,190.0,3850,8.5,70,1
...
```

regions.csv

```
1,USA
2,Europe
3,Japan
```

```
sqlite> .separator ','
sqlite> .import ./cars.csv     cars
sqlite> .import ./regions.csv  regions
sqlite> .import ./data.csv     car_stats
```

cars:

id	name
1	chevrolet chevelle malibu
2	buick skylark 320
3	plymouth satellite
4	amc rebel sst
5	ford torino
6	ford galaxie 500
7	chevrolet impala
8	plymouth fury iii
9	pontiac catalina
10	amc ambassador dpl
...	

regions:

id	name
1	USA
2	Europe
3	Japan

car_stats:

car_id	mpg	cylinders	displacement	horsepower	weight	acceleration	year	region_id
1	18.0	8	307.0	130.0	3504.0	12.0	70	1
2	15.0	8	350.0	165.0	3693.0	11.5	70	1
3	18.0	8	318.0	150.0	3436.0	11.0	70	1
4	16.0	8	304.0	150.0	3433.0	12.0	70	1
5	17.0	8	302.0	140.0	3449.0	10.5	70	1
6	15.0	8	429.0	198.0	4341.0	10.0	70	1
7	14.0	8	454.0	220.0	4354.0	9.0	70	1
8	14.0	8	440.0	215.0	4312.0	8.5	70	1
9	14.0	8	455.0	225.0	4425.0	10.0	70	1
10	15.0	8	390.0	190.0	3850.0	8.5	70	1
...								

```
sqlite> .mode column
sqlite> .width 30 10 10
sqlite> SELECT c.name AS name, r.name AS origin, s.horsepower
   ...> FROM cars c, regions r, car_stats s
   ...> WHERE c.id = s.car_id AND s.region_id = r.id
   ...> ORDER BY horsepower DESC
   ...> LIMIT 10;

name                           origin      horsepower
-----------------------------  ----------  ----------
pontiac grand prix             USA         230.0
pontiac catalina               USA         225.0
buick estate wagon (sw)        USA         225.0
buick electra 225 custom       USA         225.0
chevrolet impala               USA         220.0
plymouth fury iii              USA         215.0
ford f250                      USA         215.0
chrysler new yorker brougham   USA         215.0
dodge d200                     USA         210.0
mercury marquis                USA         208.0
```

```
sqlite> .width 10 10 10
sqlite> SELECT r.name AS origin,
   ...>        COUNT(*) AS num_cars,
   ...>        AVG(s.horsepower) AS avg_hp
   ...> FROM regions r, car_stats s
   ...> WHERE r.id = s.region_id
   ...> GROUP BY origin;

origin      num_cars    avg_hp
----------  ----------  ----------
Europe      73          78.7534246
Japan       79          79.8354430
USA         254         117.996062
```

recall: initialize database from app bundle; save changes to “Documents”

```
#define DB_NAME "cars.sqlite"

- (void)checkAndCreateDatabase {
    NSString *docDirPath = [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                                                NSUserDomainMask, YES)
                            objectAtIndex:0];
    self.databasePath = [docDirPath stringByAppendingPathComponent:DB_NAME];

   if ([[NSFileManager defaultManager] fileExistsAtPath:databasePath]) {
        return;
    }

   NSString *databasePathFromApp = [[[NSBundle mainBundle] resourcePath]
                                    stringByAppendingPathComponent:DB_NAME];
   [[NSFileManager defaultManager] copyItemAtPath:databasePathFromApp
                                           toPath:databasePath error:nil];
}
```

¶ C API Overview

Essential objects:

1. db connection:
   `sqlite3`
2. prepared statement:
   `sqlite3_stmt`

Essential functions:

1. `sqlite3_open()`
2. `sqlite3_prepare()`
3. `sqlite3_step()`
4. `sqlite3_column_...()`
5. `sqlite3_finalize()`
6. `sqlite3_close()`

```
- (void)logNames {
    sqlite3 *db;
    sqlite3_stmt *stmt;
    sqlite3_open([databasePath UTF8String], &db);
    sqlite3_prepare_v2(db, "SELECT name FROM cars", -1, &stmt, NULL);
    while (sqlite3_step(stmt) == SQLITE_ROW) {
        NSString *aName = [NSString stringWithUTF8String:(char *)sqlite3_column_text(stmt, 0)];
        NSLog(@"Name: %@", aName);
    }
    sqlite3_finalize(stmt);
    sqlite3_close(db);
}
```

```
- (void)insertName {
    sqlite3 *db;
    sqlite3_stmt *stmt;
    sqlite3_open([databasePath UTF8String], &db);
    sqlite3_prepare_v2(db, "INSERT INTO cars (name) VALUES ('VW Rabbit')", -1, &stmt, NULL);
    sqlite3_step(stmt);
    sqlite3_finalize(stmt);
    sqlite3_close(db);
}
```

opening a new connection
to the database is **time-consuming**!

generating a prepared statement
incurs **overhead**

1. *reuse* single db connection
2. *reuse* prepared statements with `sql3_bind_...()`

```
static const char * const INSERT_NAME_SQL = "INSERT INTO cars (name) VALUES (?)";

static sqlite3_stmt *insert_stmt          = NULL;


- (BOOL)openDB {
    if (sqlite3_open([databasePath UTF8String], &db) != SQLITE_OK) {
        return NO;
    }
    if (sqlite3_prepare_v2(db, INSERT_NAME_SQL, -1, &insert_stmt, NULL) != SQLITE_OK) {
        sqlite3_finalize(insert_stmt);
        sqlite3_close(db);
        return NO;
    }
}

- (void)writeNameToDB:(NSString *)name {
    sqlite3_reset(insert_stmt);
    sqlite3_bind_text(insert_stmt, 1, [name UTF8String], -1, SQLITE_TRANSIENT);
    if (sqlite3_step(insert_stmt) == SQLITE_DONE) {
        NSLog(@"Insert successful!");
    }
}
```

lot of potential for **bad design**

MVC-ish?

nope.

want to avoid mixing *db management*
with *controller* logic

model should handle its **own persistence**

(debatable)

also: all data currently in memory

— how to unload/reload on low memory notifications?

design pattern: **active record**

— model objects act as in-memory, dynamic records of the database

with relational databases, active record is an *object-relational mapping* (ORM) technique

model objects only need to know their DB *primary keys* to access the rest of their data

key behavior: **hydrate/dehydrate**

hydrate = fetch model data from database

dehydrate = flush data to database (persist)
& release data in core

important goal: on-demand / lazy loading

e.g., top-level models: Cars & Regions

```
@interface Car : NSObject {
    sqlite3 *database;
    NSInteger primaryKey;

    NSString *name;
    double mpg;
    /* etc. */

    BOOL hydrated;
    BOOL dirty;
}

- (id)initWithPrimaryKey:(NSInteger)pk database:(sqlite3 *)db;
- (BOOL)insertIntoRegion:(Region *)region database:(sqlite3 *)db;

- (void)hydrate;
- (void)dehydrate;
@end
```

```
@interface Region : NSObject {
    sqlite3 *database;
    NSInteger primaryKey;
    NSString *name;
    NSMutableArray *carsFromRegion;
}

- (id)initWithPrimaryKey:(NSInteger)pk database:(sqlite3 *)db;

- (void)hydrate;
- (void)dehydrate;
@end
```

(Car.m)

```
static const char * const INIT_SQL      = "SELECT name FROM cars WHERE id=?";
```

```
- (id)initWithPrimaryKey:(NSInteger)pk database:(sqlite3 *)db {
    if (self = [super init]) {
        primaryKey = pk;
        database = db;
        if (init_stmt == NULL) {
            sqlite3_prepare_v2(database, INIT_SQL, -1, &init_stmt, NULL);
        }

        sqlite3_bind_int(init_stmt, 1, primaryKey);
        if (sqlite3_step(init_stmt) == SQLITE_ROW) {
            self.name = [NSString stringWithUTF8String:
                                  (char *)sqlite3_column_text(init_stmt, 0)];
        }
        sqlite3_reset(init_stmt);

        dirty = NO;
        hydrated = NO;
    }
    return self;
}
```

(Car.m)

```
static const char * const HYDRATE_SQL    = "SELECT name,mpg,cylinders,displacement,"
                                            "       horsepower,weight,acceleration,year "
                                            "FROM cars c, car_stats s "
                                            "WHERE c.id=? AND s.car_id=?";
```

```
- (void)hydrate {
    if (!hydrated) {
        if (hydrate_stmt == NULL)
            sqlite3_prepare_v2(database, HYDRATE_SQL, -1, &hydrate_stmt, NULL);

        sqlite3_bind_int(hydrate_stmt, 1, primaryKey);
        sqlite3_bind_int(hydrate_stmt, 2, primaryKey);
        if (sqlite3_step(hydrate_stmt) == SQLITE_ROW) {
            self.name = [NSString stringWithUTF8String:
                                    (char *)sqlite3_column_text(hydrate_stmt, 0)];
            mpg = sqlite3_column_double(hydrate_stmt, 1);
            cylinders = sqlite3_column_int(hydrate_stmt, 2);
            /* etc. */
        }
        sqlite3_reset(hydrate_stmt);

        hydrated = YES;
        dirty = NO;
    }
}
```

(Car.m)

```
static const char * const DEHYDRATE_CARNAME_SQL = "UPDATE cars SET name=?"
                                                  "WHERE id=?";

static const char * const DEHYDRATE_CARSTATS_SQL = "UPDATE car_stats "
                                                   "SET mpg=?,cylinders=?,displacement=?,"
                                                   "    horsepower=?,weight=?,"
                                                   "    acceleration=?,year=? "
- (void)dehydrate {                                "WHERE car_id=?";
    BOOL success = YES;

    if (!dirty || !hydrated)
        return;

    if (dehydrate_carname_stmt == nil) {
        sqlite3_prepare_v2(database, DEHYDRATE_CARNAME_SQL, -1,
                           &dehydrate_carname_stmt, NULL);
        sqlite3_prepare_v2(database, DEHYDRATE_CARSTATS_SQL, -1,
                           &dehydrate_carstats_stmt, NULL);
    }

    sqlite3_step(begin_transaction_stmt);

    sqlite3_bind_text(dehydrate_carname_stmt, 1, [name UTF8String], -1, SQLITE_TRANSIENT);
    sqlite3_bind_int(dehydrate_carname_stmt, 2, primaryKey);
    success = sqlite3_step(dehydrate_carname_stmt) == SQLITE_DONE;
    sqlite3_reset(dehydrate_carname_stmt);
```

(Car.m)

(-dehydrate continued)

```
    sqlite3_bind_double(dehydrate_carstats_stmt, 1, mpg);
    sqlite3_bind_int(   dehydrate_carstats_stmt, 2, cylinders);
    /* etc. */

    success = success && sqlite3_step(dehydrate_carstats_stmt) == SQLITE_DONE;
    sqlite3_reset(dehydrate_carstats_stmt);

    if (success) {
        self.name = nil;
        mpg = displacement = horsepower = weight = acceleration = 0.0;
        cylinders = year = 0;
        hydrated = NO;
        dirty = NO;
        sqlite3_step(commit_transaction_stmt);
    } else {
        sqlite3_step(rollback_transaction_stmt);
    }
}
```

(Car.m)

```
- (void)prepareTransactionalStatements {
    if (begin_transaction_stmt != NULL)
        return;
    sqlite3_prepare_v2(database, "BEGIN EXCLUSIVE TRANSACTION", -1,
                       &begin_transaction_stmt, NULL);
    sqlite3_prepare_v2(database, "COMMIT TRANSACTION", -1,
                       &commit_transaction_stmt, NULL);
    sqlite3_prepare_v2(database, "ROLLBACK TRANSACTION", -1,
                       &rollback_transaction_stmt, NULL);
}
```

(Car.m)

```
- (void)setName:(NSString *)aName {
    dirty = YES;
    name = aName;
}

- (void)setMpg:(double)theMpg {
    dirty = YES;
    mpg = theMpg;
}

/* etc. */
```

(Car.m)

```
static const char * const INSERT_CARNAME_SQL   = "INSERT INTO cars (name) VALUES (?)";

static const char * const INSERT_CARSTATS_SQL = "INSERT INTO car_stats "
                                                "(car_id,mpg,cylinders,displacement,horsepower,"
                                                " weight,acceleration,year,region_id) "
                                                "VALUES (?,?,?,?,?,?,?,?,?)";
```

```
- (BOOL)insertIntoRegion:(Region *)region database:(sqlite3 *)db {
    ...
    sqlite3_bind_text(insert_carname_stmt, 1, [name UTF8String], -1, SQLITE_TRANSIENT);
    if (sqlite3_step(insert_carname_stmt) == SQLITE_DONE) {
        primaryKey = sqlite3_last_insert_rowid(database);
    }

    sqlite3_bind_int(   insert_carstats_stmt, 1, primaryKey);
    sqlite3_bind_double(insert_carstats_stmt, 2, mpg);
    sqlite3_bind_int(   insert_carstats_stmt, 3, cylinders);
    /* etc. */
    sqlite3_step(insert_carstats_stmt);

    hydrated = YES;
    dirty = NO;
    ...
}
```

(Region.m)

```
static const char * const ALL_CARS_SQL  = "SELECT c.id FROM cars c, car_stats s "
                                          "WHERE c.id=s.car_id AND s.region_id=?"
                                          "ORDER BY c.name";
```

```
- (id)initWithPrimaryKey:(NSInteger)pk database:(sqlite3 *)db {
    ...
    carsFromRegion = [[NSMutableArray alloc] init];
    sqlite3_bind_int(all_cars_statement, 1, primaryKey);
    while (sqlite3_step(all_cars_statement) == SQLITE_ROW) {
        Car *car = [[Car alloc] initWithPrimaryKey:sqlite3_column_int(
                                                       all_cars_statement, 0)
                                          database:database];
        [carsFromRegion addObject:car];
    }
    ...
}

- (void)hydrate {
    [carsFromRegion makeObjectsPerformSelector:@selector(hydrate)];
}

- (void)dehydrate {
    [carsFromRegion makeObjectsPerformSelector:@selector(dehydrate)];
}
```

when to hydrate/dehydrate?

(CarListViewController.m)

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    CarViewController *viewController = [[CarViewController alloc
                                          initWithStyle:UITableViewStyleGrouped];
    Car *car = [region.carsFromRegion objectAtIndex:indexPath.row];
    [car hydrate];
    viewController.car = car;
    [self.navigationController pushViewController:viewController animated:YES];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    [region dehydrate];
}
```

(CarsAppDelegate.m)

```
- (void)applicationWillTerminate:(UIApplication *)application {
    [allRegions makeObjectsPerformSelector:@selector(dehydrate)];
    [Region finalizeStatements];
    sqlite3_close(database);
}
```

phew!

not a bad solution, but it still needs
a lot of work ...

some issues:

- how to cleanly create a new top-level object without knowing about primary keys?
- what if one controller adds objects to the DB that another controller cares about?
- how to ensure object uniqueness? (i.e., only one object for a given primary key)

§ Core Data, next!