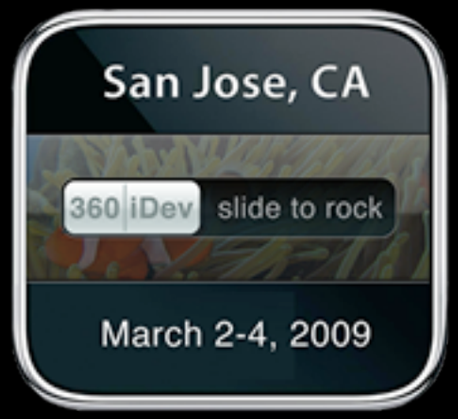
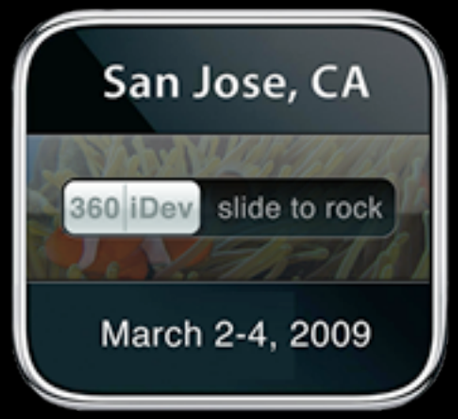
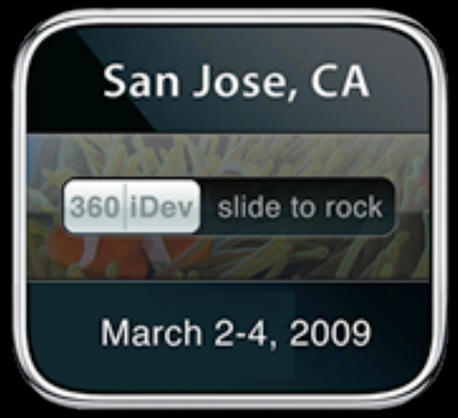


SQLitePersistentObjects



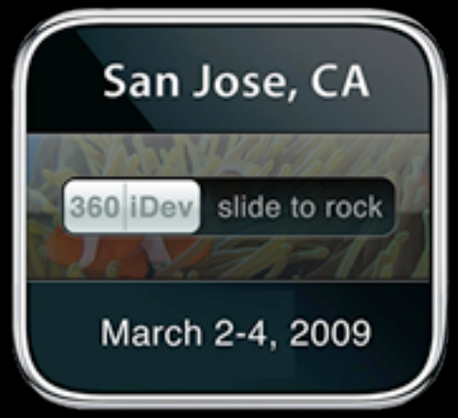
SQLite Without the SQL

Jeff LaMarche



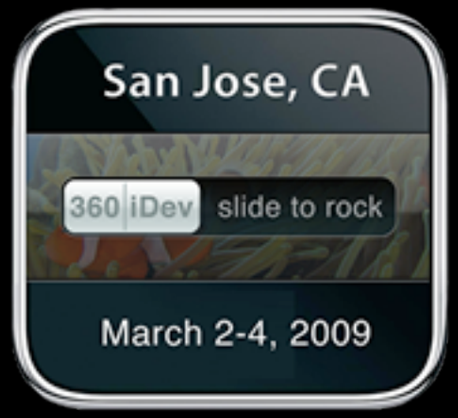
Contacting Me

- jeff_lamarche@mac.com
- http://iphonedevelopment.blogspot.com
- Twitter: jeff_lamarche



About Me

- Writer
  - Beginning iPhone Development (Apress)
  - Several Articles for ADC & MacTech Mag
  - Beginning Cocoa (Apress, in progress)
  - Regular blog postings



More About Me

- Programmer
  - Primarily contract work
  - Using Cocoa since about 2000
  - OO Programming since early 1990s
  - Programming since 1980
  - Full-time with Cocoa for only a year

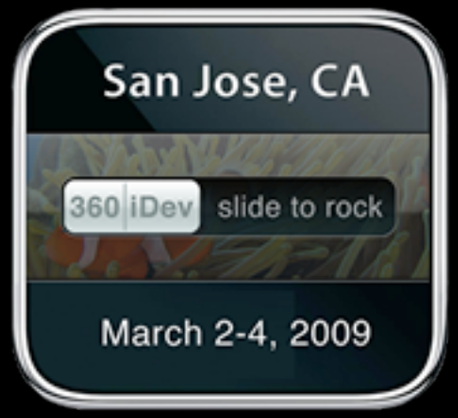
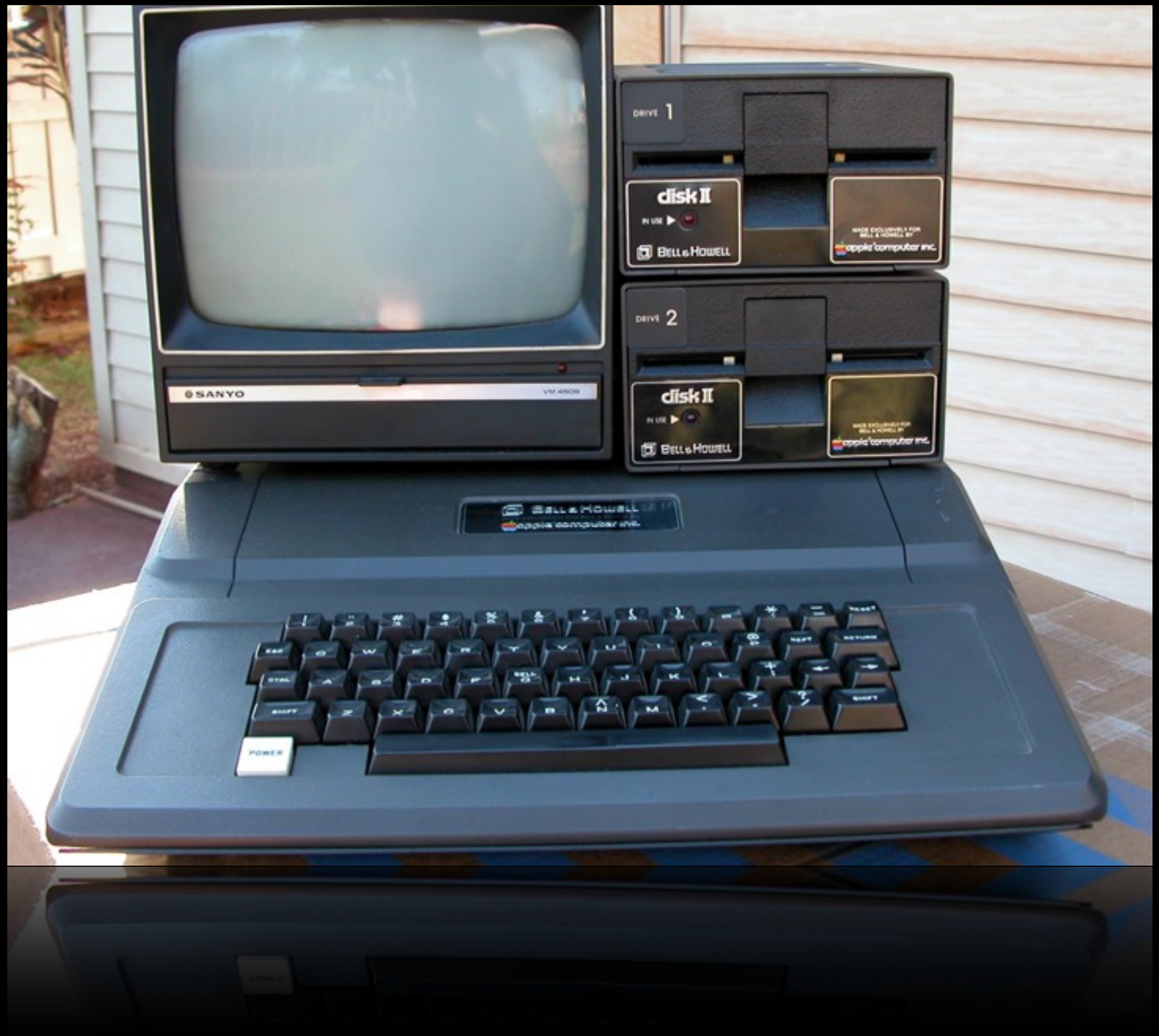
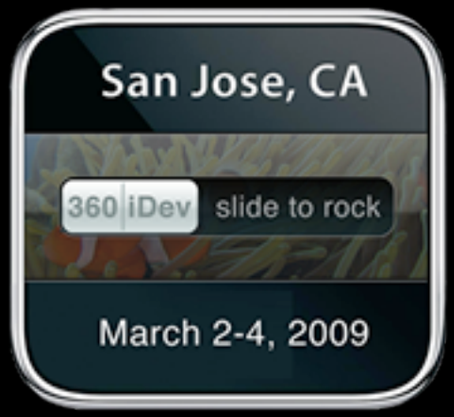
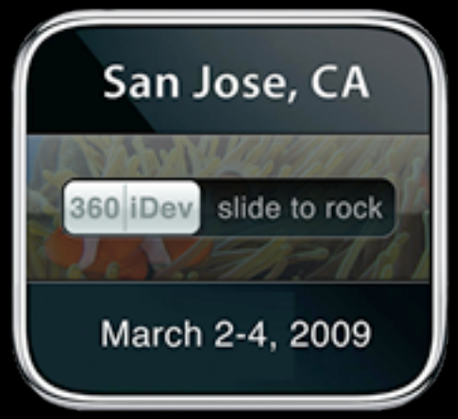


Image Source: http://www.macgeek.org/museum/bhapple2plus/page02.html



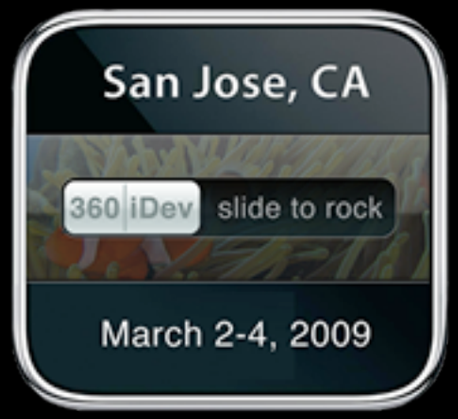



```
HELLO, WORLD!

]LIST

10   HOME
20   INVERSE
30   PRINT "HELLO, WORLD!"
40   NORMAL
50   PRINT  CHR$ (7)

]
```



Enough About Me

(yeah, seriously)



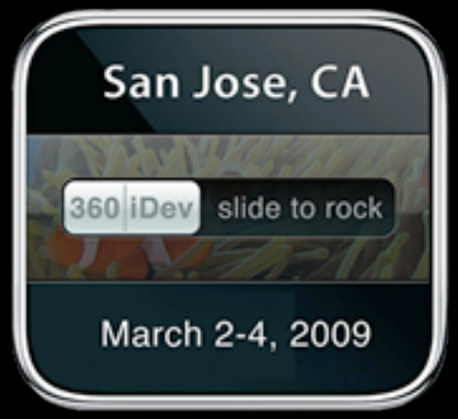
SQLPO

SQLitePersistentObjects

- Object-Relational Mapping (ORM) Tool

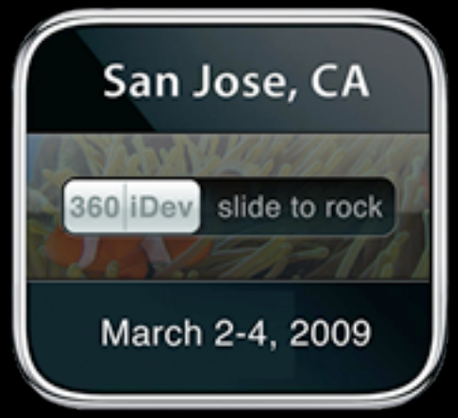
  or for the more technically-minded:

  code that takes data from your program and sticks it into or pulls it out of a relational database



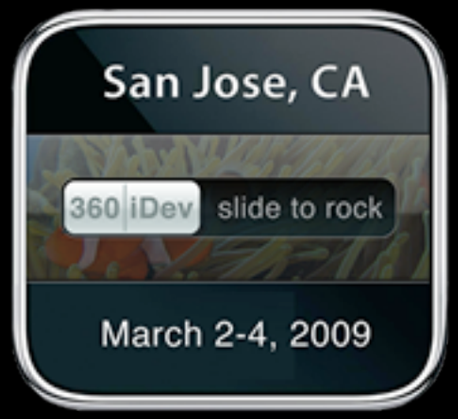
In the Beginning...

- Early OO programs that needed to use data from a database embedded SQL statements in code
  - Difficult to maintain
  - Strings are a “black box” to the compiler



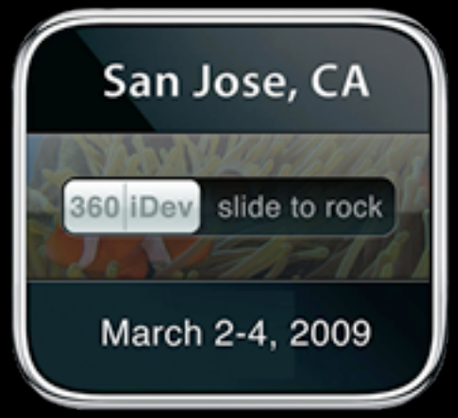
ORM History

- 1996 Enterprise Object Frameworks (EOF)
  - Part of NeXT's Cocoa-Based WebObjects
  - Used visual tool to map data objects to database table
  - Once model created, you interacted with database tables as if they were objects
  - Didn't even have to create specific classes if you didn't want to



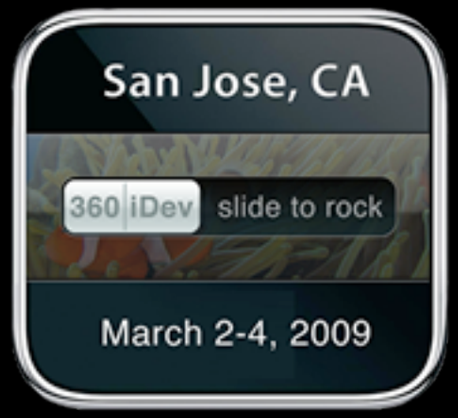
ORM History

- Basic idea was borrowed by other platforms, changed to work with other languages, sometimes extended to be better
  - Hibernate, Cayenne, ADO .net, Outlet, Django, many others



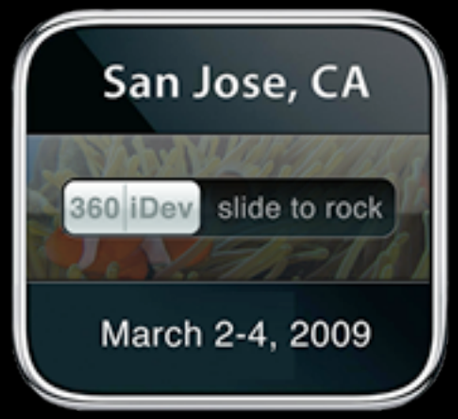
ORM History

- One language evolved the concept
  - Ruby on Rails' ActiveRecord
    - No longer needed mapping document
    - Object structure dynamically created based on table structure in database



ORM History

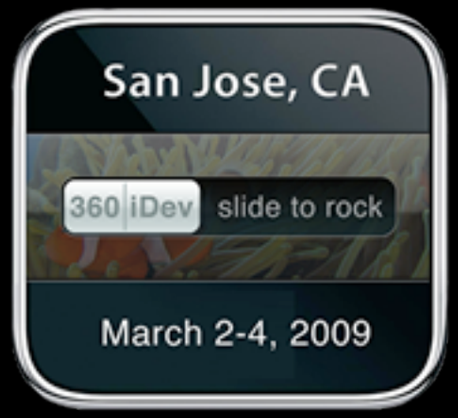
- Core Data
  - Core Data on the Mac is EOF's stepchild
  - Core Data is NOT EOF, but shares some DNA



ORM History

Apple has not ported Core Data to the iPhone (yet).

That fact is what triggered the birth of SQLitePersistentObjects



SQLPO

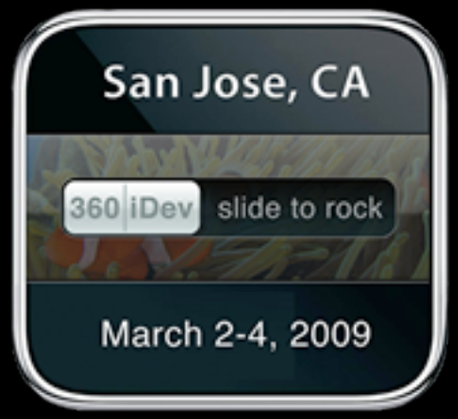
SQLitePersistentObjects

- You can always find the latest version here:

  http://code.google.com/p/sqlitepersistentobjects/

- Add the code from the /src directory to your Xcode project

- Make sure to link to the SQLite3 dynamic library at /usr/lib/libsqlite3.dylib



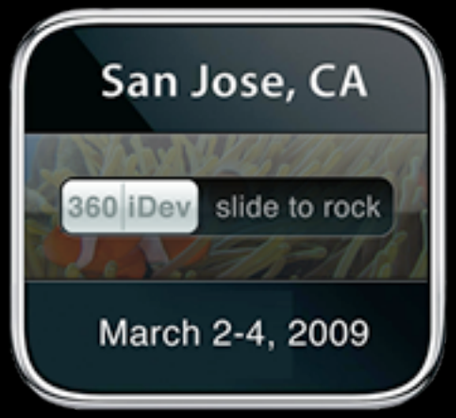
SQLPO

SQLitePersistentObjects

- We have a support mailing list:

  http://groups.google.com/group/sqlitepersistentobjects-user

- Licensed under the New BSD License
  - Free, open source.
  - No attribution required, non-viral, can be used in commercial projects, no need to publish your code
  - If you can't use BSD licensed code for some reason, contact me - we're flexible on the license.

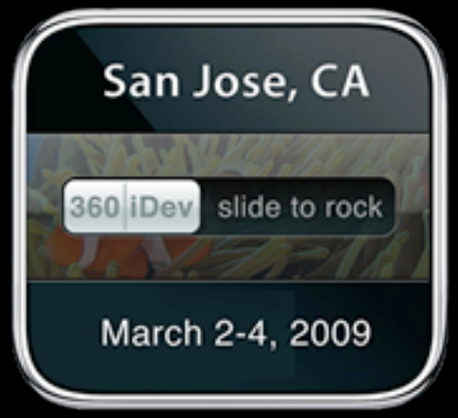


The Basics

SQLitePersistentObjects

- To create a persistent object, you simply subclass an existing class.

```
#import <Foundation/Foundation.h>
#import "SQLitePersistentObject.h"

@interface Person : SQLitePersistentObject
{
   NSString     *firstName;
   NSString     *lastName;
   NSDate       *birthdate;
   int          numberOfChildren;
   float        contribution;
   UIImage      *photo;
}
@property (nonatomic, retain) NSString *firstName;
@property (nonatomic, retain) NSString *lastName;
@property (nonatomic, retain) NSDate *birthdate;
@property int numberOfChildren;
@property float contribution;
@property (nonatomic, retain) UIImage *photo;
@end
```

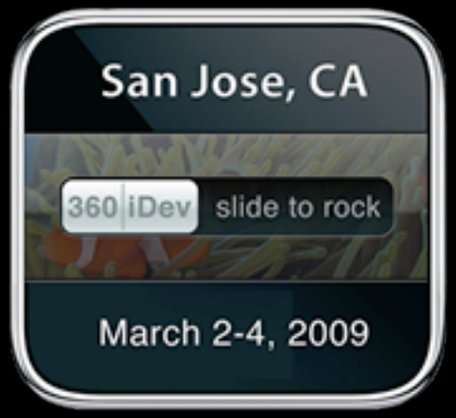


The Basics

SQLitePersistentObjects

- To create a persistent object, you simply subclass an existing class.

```
#import "SQLitePersistentObject.h"

@interface Person  SQLitePersistentObject
{
   NSString     *firstName;
   NSString     *lastName;
   NSDate       *birthdate;
   int          numberOfChildren;
   float        contribution;
   UIImage      *photo;
}
@property (nonatomic, retain) NSString *firstName;
@property (nonatomic, retain) NSString *lastName;
@property (nonatomic, retain) NSDate *birthdate;
@property int numberOfChildren;
@property float contribution;
@property (nonatomic, retain) UIImage *photo;
@end
```

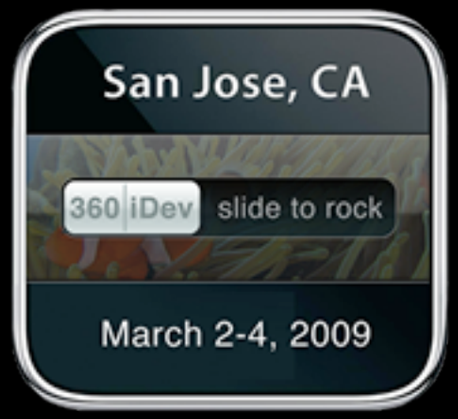


The Basics

SQLitePersistentObjects

- To create a persistent object, you simply subclass an existing class.

```
#import "Person.h"

@implementation Person
@synthesize firstName;
@synthesize lastName;
@synthesize birthdate;
@synthesize numberOfChildren;
@synthesize contribution;
@synthesize photo;
- (void)dealloc
{
   [firstName release];
   [lastName release];
   [birthdate release];
   [photo release];
   [super dealloc];
}
@end
```

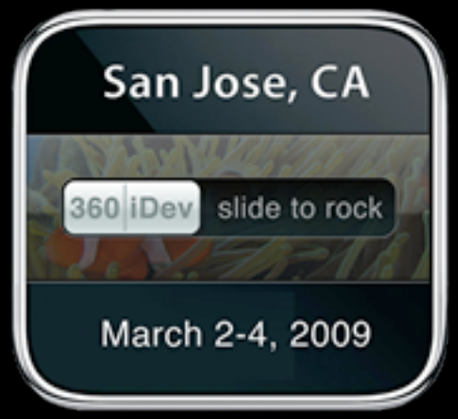


The Basics

SQLitePersistentObjects

- Once you've defined a persistent object, creating a new object and saving it to the database couldn't be easier:

```
Person *newPerson = [[Person alloc] init];
newPerson.firstName = @"Martha";
newPerson.lastName = @"Washington";
newPerson.birthdate = [NSDate date];
newPerson.numberOfChildren = 5;
newPerson.contribution = 27.32;
newPerson.photo = [UIImage imageNamed:@"MarthaWashington.png"];
[newPerson save];
[newPerson release];
```

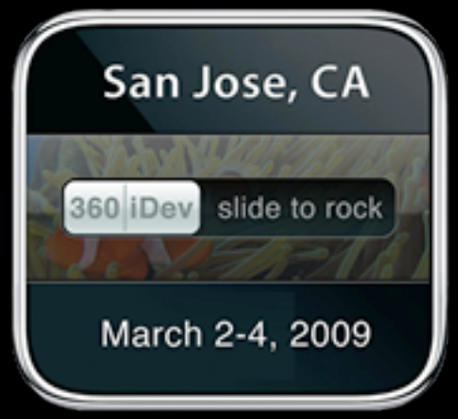


The Basics

SQLitePersistentObjects

- Once you've defined a persistent object, creating a new object and saving it to the database couldn't be easier:

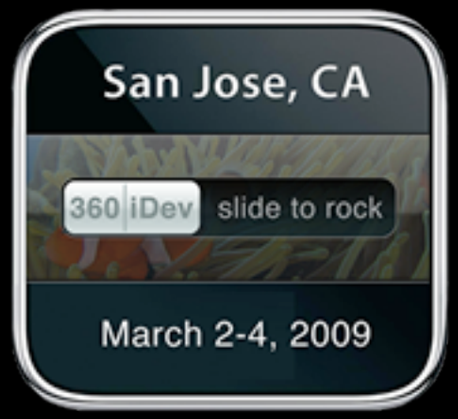
```
Person *newPerson = [[Person alloc] init];
newPerson.firstName = @"Martha";
newPerson.lastName = @"Washington";
newPerson.birthdate = [NSDate date];
newPerson.numberOfChildren = 5;
newPerson.contribution = 27.32;
newPerson.photo = [UIImage imageNamed:@"MarthaWashington.png"];
[newPerson save];
[newPerson release];
```



The Basics

SQLitePersistentObjects

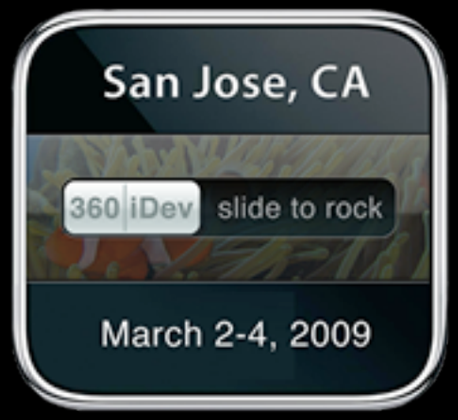
- How does it save the data?
  - Raw datatypes are mapped to appropriate columns
    - e.g. `int` goes into INTEGER field, `float` goes into REAL field
  - Other persistent objects are stored as references to the row and table where that object is stored



The Basics

SQLitePersistentObjects

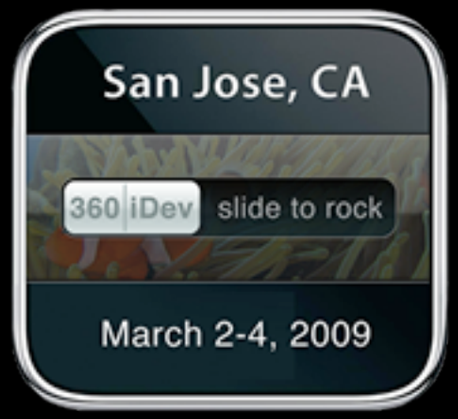
- How does it save the data? (cont)
  - Objects that aren't subclasses of SQLitePersistentObject get stored IF they conform to a protocol called `SQLitePersistence`
  - We have provided categories for most common objects to conform them to `SQLitePersistence`
    - `NSString`, `NSNumber`, `UIImage`, `UIColor`, `NSData`, `NSDate`, `NSMutableData`



The Basics

SQLitePersistentObjects

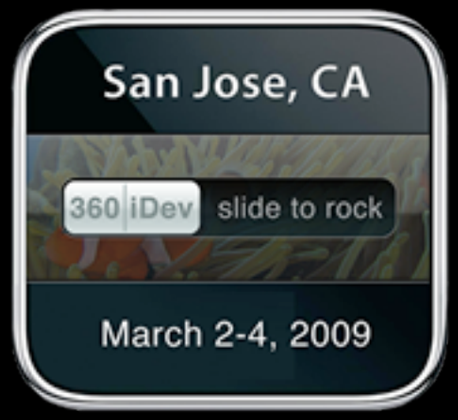
- How does it save the data? (cont)
  - We also provide category on `NSObject` as a fallback. Any object that doesn't conform to `SQLitePersistence` but that does conform to `NSCoding` can be stored in the database, but gets archived as a BLOB, which can't be searched or used in criteria-based queries.



The Basics

SQLitePersistentObjects

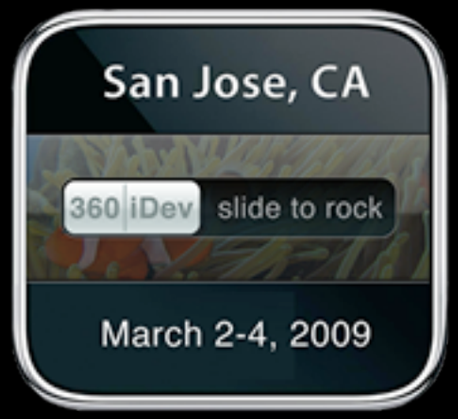
- How does it save the data? (cont)
  - Dictionaries, Arrays, and Sets do not get stored in a column in the object's table, but rather get stored in a child table.
    - Each item in collection gets one row in the child table
    - Raw Datatypes and non-persistent objects get stored right in child table
    - Persistent objects get stored as references



The Basics

SQLitePersistentObjects

- How does it save the data? (cont)
  - Every objects gets assigned a primary key value, which is an integer that acts as a unique identifier for that object
  - Stored in a private instance variable called pk
  - Used in some cases load objects from database

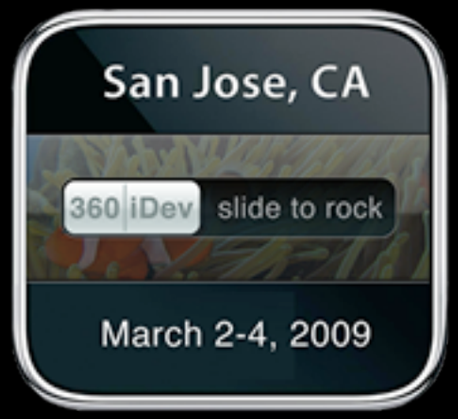


The Basics

SQLitePersistentObjects

- Loading an object from the database is accomplished through class methods. This is the simplest one:

```
Person *martha = [Person findByPK:1];
```

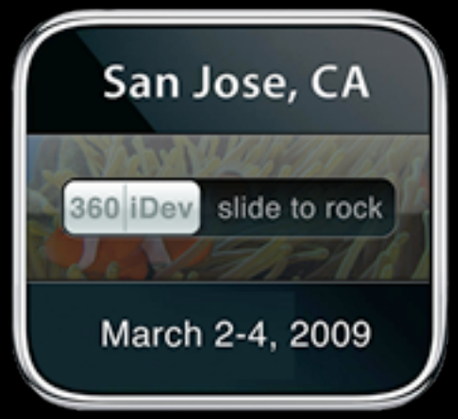


The Basics

SQLitePersistentObjects

- You can load all objects into an array. Be careful doing this, however, as it is not a very efficient use of memory in most cases.

```
NSArray *people = [Person allObjects];
```



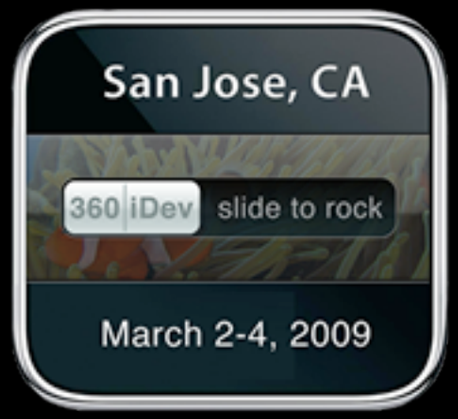
The Basics

SQLitePersistentObjects

- SQLPO also creates dynamic find methods based on your property names:

```
NSArray *people = [Person allObjects];
```

```
NSArray *people = [Person findByLastName:@"Washington"];
```



The Basics

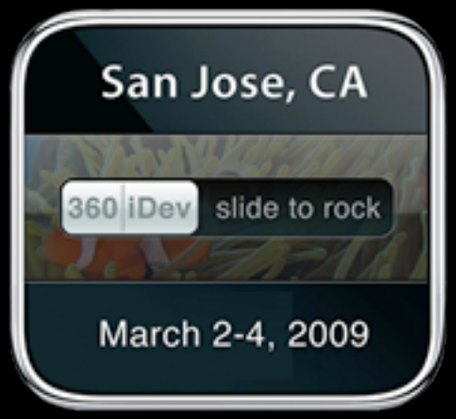
SQLitePersistentObjects

- Dynamic find by methods support SQL wildcards

```
NSArray *people = [Person allObjects];
```

```
NSArray *people = [Person findByLastName:@"Washington"];
```

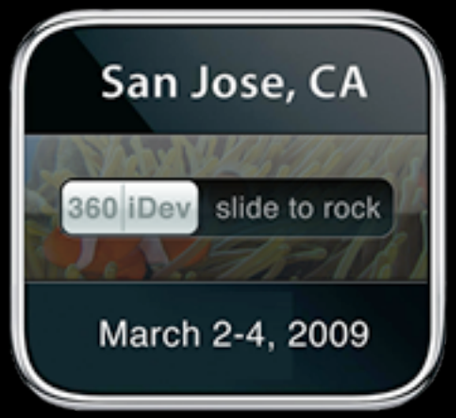
```
NSArray *people = [Person findByLastName:@"Wash%"];
```



The Basics

SQLitePersistentObjects

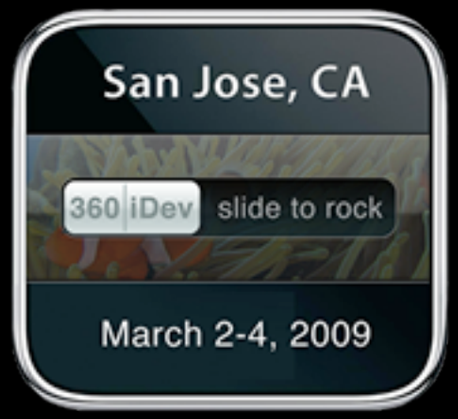
```
#import <Foundation/Foundation.h>
#import "SQLitePersistentObject.h"

@interface Person : SQLitePersistentObject
{
   NSString      *firstName;
   NSString      *lastName;
   NSDate        *birthdate;
   int           numberOfChildren;
   float         contribution;
   UIImage       *photo;
}
@property (nonatomic, retain) NSString *firstName;
@property (nonatomic, retain) NSString *lastName;
@property (nonatomic, retain) NSDate *birthdate;
@property int numberOfChildren;
@property float contribution;
@property (nonatomic, retain) UIImage *photo;
@end

@interface Person (squelch)
+ (id)findByName:(NSString *)theName;
@end
```



The Basics

SQLitePersistentObjects

```
#import <Foundation/Foundation.h>
#import "SQLitePersistentObject.h"

@interface Person : SQLitePersistentObject
{
   NSString      *firstName;
   NSString      *lastName;
   NSDate        *birthdate;
   int           numberOfChildren;
   float         contribution;
   UIImage       *photo;
}
@property (nonatomic, retain) NSString *firstName;
@property (nonatomic, retain) NSString *lastName;
@property (nonatomic, retain) NSDate *birthdate;
@property int numberOfChildren;
@property float contribution;
@property (nonatomic, retain) UIImage *photo;
@end

@interface Person (squelch)
+ (id)findByName:(NSString *)theName;
@end
```

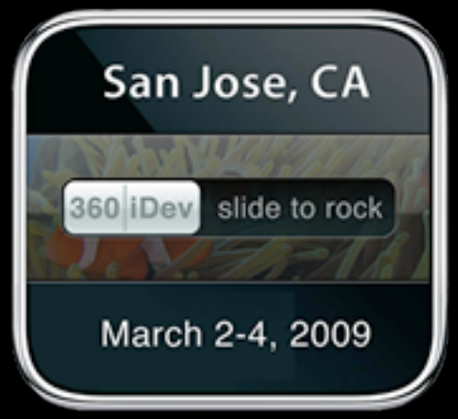


The Basics

SQLitePersistentObjects

- If you need more flexibility, you can always specify the exact criteria by supplying a SQL where clause with `findByCriteria:`

```
NSArray *people = [Person findByCriteria:@"WHERE first_name = 'John' and last_name like 'S%' and date(birthdate) <= date('now')"];
```



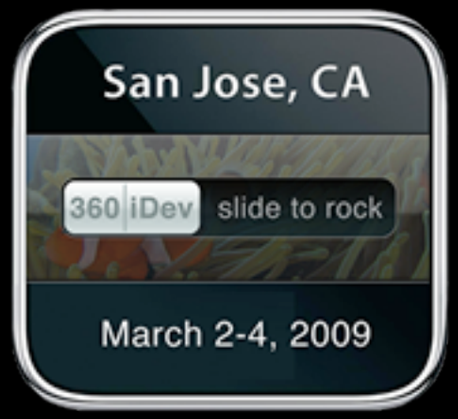
The Basics

SQLitePersistentObjects

- If you need more flexibility, you can always specify the exact criteria by supplying a SQL where clause with `findByCriteria:`

```
NSArray *people = [Person findByCriteria:@"WHERE first_name = 'John' and last_name like 'S%' and date(birthdate) <= date('now')"];
```

Note: Property `firstName` becomes column `first_name`

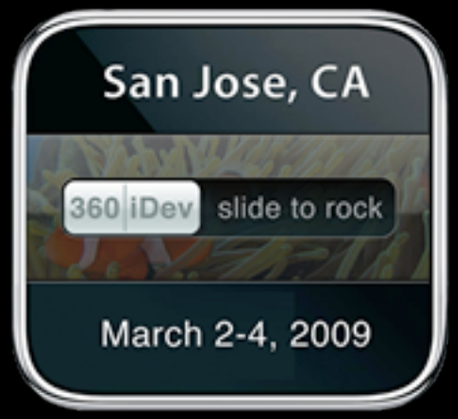


The Basics

SQLitePersistentObjects

- You can find out the column name for a property name like so:

```
#import "NSString-SQLiteColumnName.h"
…
NSString *columnName = [@"firstName" stringAsSQLColumnName];
```

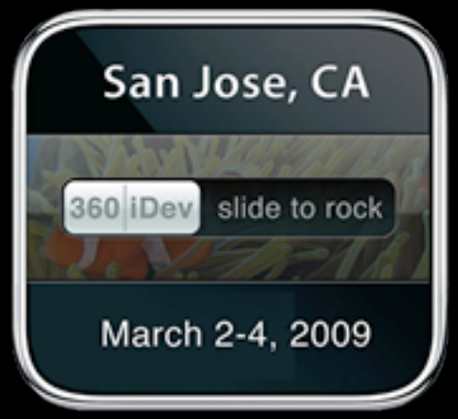


The Basics

SQLitePersistentObjects

- If you only want the first object that meets your criteria, you can do that also:

```
    Person *firstPerson = [Person findFirstByCriteria:@"WHERE first_name =
'John' and last_name like 'S%' and date(birthdate) <= date('now')"];
```

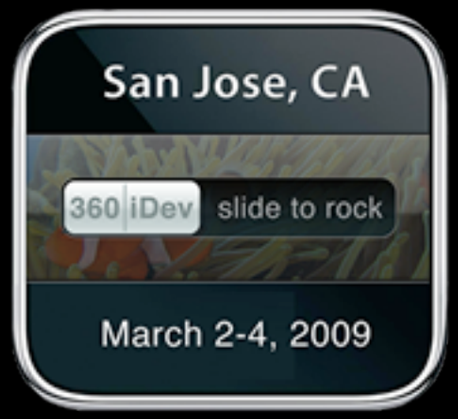


The Basics

SQLitePersistentObjects

- Deleting an object should be done using the class method `deleteObject:cascade:`, which takes the primary key of the object to be deleted.

```
[Person deleteObject:5 cascade:YES];
```

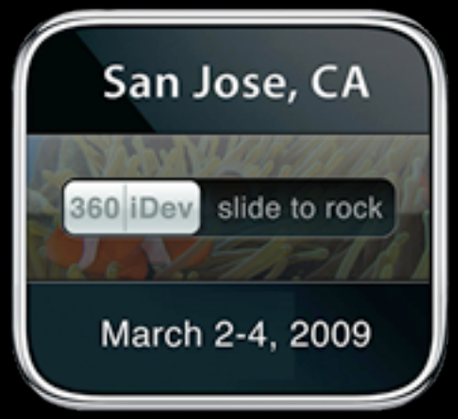


The Basics

SQLitePersistentObjects

- Database tables can benefit from adding indices to them. SQLitePersistentObjects has a mechanism for adding indices without writing SQL. Override this method in your class:

```
+(NSArray *)indices
{
    return nil;
}
```

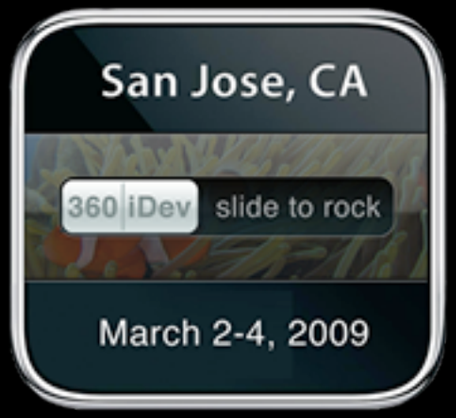


The Basics

SQLitePersistentObjects

- Method should return an array of arrays. Each contained array represents one index and should have the properties to be indexed in the order they should be in the index.

```
+(NSArray *)indices
{
    NSArray *firstIndex = [NSArray arrayWithObjects:@"lastName",
@"firstName", @"pk", nil];
    NSArray *secondIndex = [NSArray arrayWithObjects:@"birthdate", @"pk",
nil];
    return [NSArray arrayWithObjects:firstIndex, secondIndex, nil];
}
```

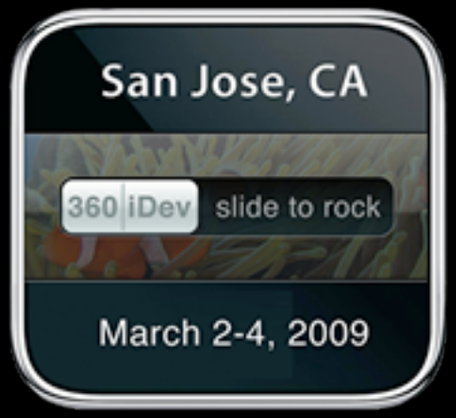


The Basics

SQLitePersistentObjects

- Let's look at our class declaration again.

```
#import <Foundation/Foundation.h>
#import "SQLitePersistentObject.h"

@interface Person : SQLitePersistentObject
{
   NSString     *firstName;
   NSString     *lastName;
   NSDate       *birthdate;
   int          numberOfChildren;
   float        contribution;
   UIImage      *photo;
}
@property (nonatomic, retain) NSString *firstName;
@property (nonatomic, retain) NSString *lastName;
@property (nonatomic, retain) NSDate *birthdate;
@property int numberOfChildren;
@property float contribution;
@property (nonatomic, retain) UIImage *photo;
@end
```

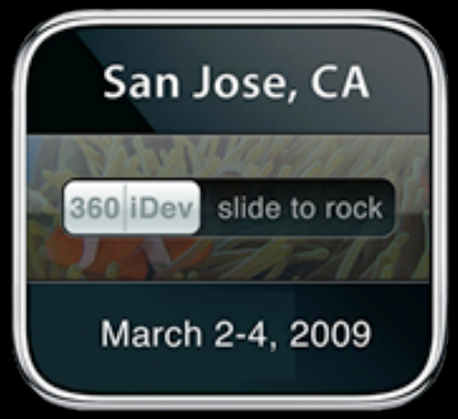


The Basics

SQLitePersistentObjects

- Let's look at our class declaration again.

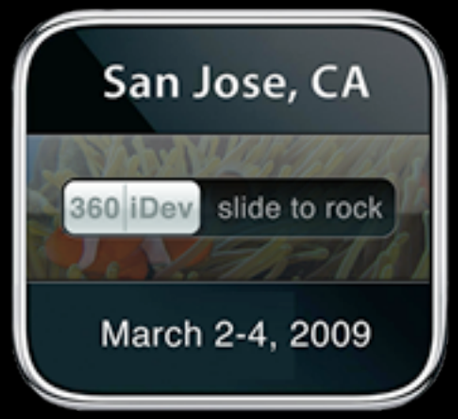
```
#import <Foundation/Foundation.h>
#import "SQLitePersistentObject.h"

@interface Person : SQLitePersistentObject
{
   NSString     *firstName;
   NSString     *lastName;
   NSDate       *birthdate;
   int          numberOfChildren;
   float        contribution;
   UIImage      *photo;
}
@property (nonatomic, retain) NSString *firstName;
@property (nonatomic, retain) NSString *lastName;
@property (nonatomic, retain) NSDate *birthdate;
@property int numberOfChildren;
@property float contribution;
@property (nonatomic, retain) UIImage *photo;
@end
```



The Basics

SQLitePersistentObjects

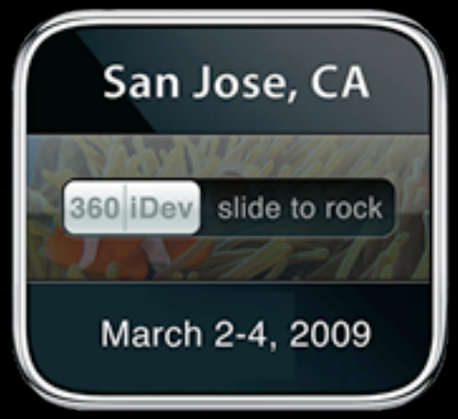
- Let's look at our class declaration again.



The Basics

SQLitePersistentObjects

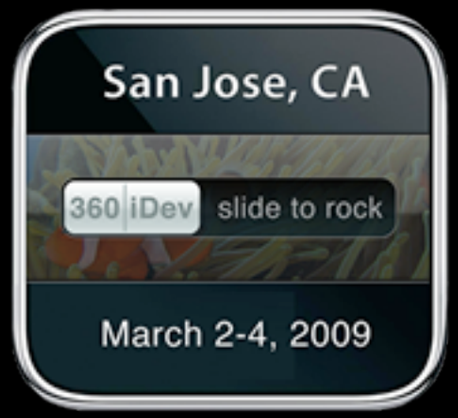
- Let's talk about Paired Arrays.
  - Simple Concept - Multiple Arrays
  - Every array has same number of rows
  - Object at same index in each array corresponds to information about the same object
    - e.g. fifth object in one array might hold Joe's age, and the fifth object in the other array might hold Joe's last name.



The Basics

SQLitePersistentObjects

- Let's talk about Paired Arrays (cont)
- Can have as many arrays as necessary.
- SQLPO has built-in method to return specified paired arrays.
- It packs all the paired arrays together inside another array
- First array in the array always contains a list of the primary keys.

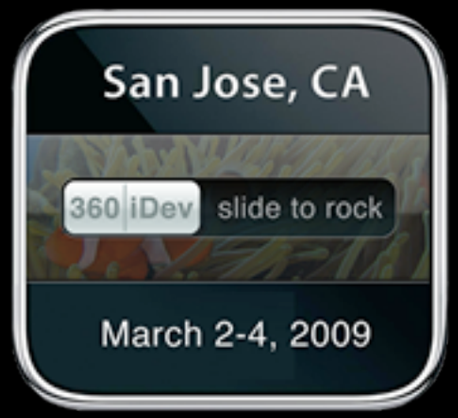


The Basics

SQLitePersistentObjects

Example: Three NSArrays

pks	firstNames	lastNames
1	Martha	Washington
2	Joe	Smith
3	Sally	Ride
4	George	Washington
5	Buster	Keaton



The Basics

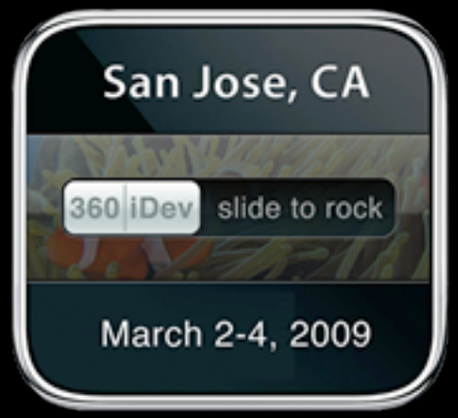
SQLitePersistentObjects

Example: Three NSArrays

pks
1
2
3
4
5

firstNames
Martha
Joe
Sally
George
Buster

lastNames
Washington
Smith
Ride
Washington
Keaton

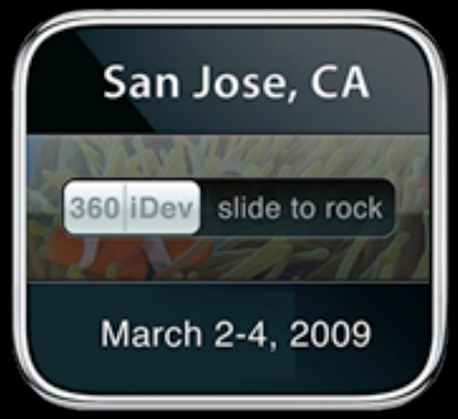


The Basics

SQLitePersistentObjects

Example: Three NSArrays

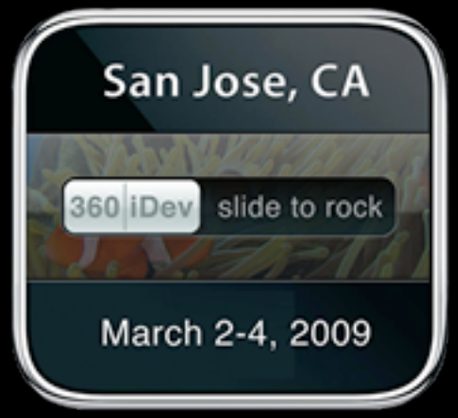
pks	firstNames	lastNames
1	Martha	Washington
2	Joe	Smith
3	Sally	Ride
4	George	Washington
5	Buster	Keaton



The Basics

SQLitePersistentObjects

- This means we can load in only the information we need to display in the table, along with the information we need to load the full object if the user selects it.



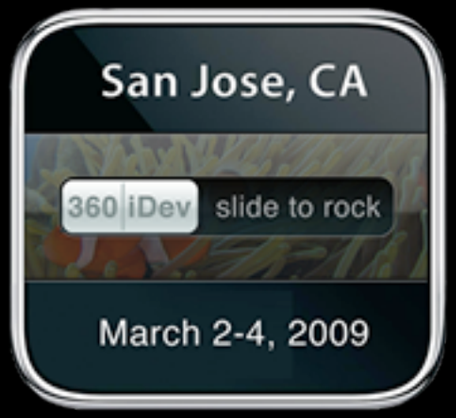
The Basics

SQLitePersistentObjects

- In our controller class, we'll need mutable arrays to hold the data.

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

@interface PeopleListViewController : UITableViewController
{
   NSMutableArray    *pks;
   NSMutableArray    *firstNames;
   NSMutableArray    *lastNames;
}
@property (nonatomic, retain) NSMutableArray *pks;
@property (nonatomic, retain) NSMutableArray *firstNames;
@property (nonatomic, retain) NSMutableArray *lastNames;
- (void)refreshData;
@end
```

- We also declare a method for loading the arrays.



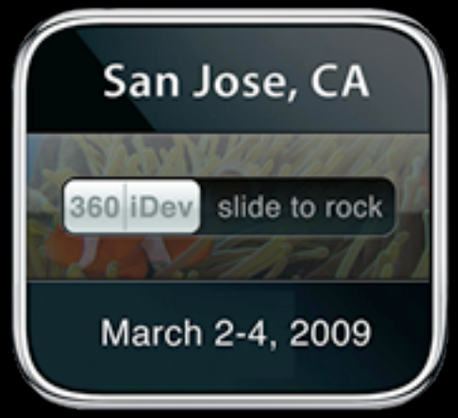
The Basics

SQLitePersistentObjects

- Getting the paired arrays is simple enough:

```
- (void)refreshData
{
   NSArray *array = [Person pairedArraysForProperties:[NSArray
arrayWithObjects:@"firstName", @"lastName", nil] withCriteria:@"where birthdate is not
null"];
   self.pks = [array objectAtIndex:0];
   self.firstNames = [array objectAtIndex:1];
   self.lastNames = [array objectAtIndex:2];
}
```

- Just tell it which properties you want, and it will give you all those plus the primary keys.

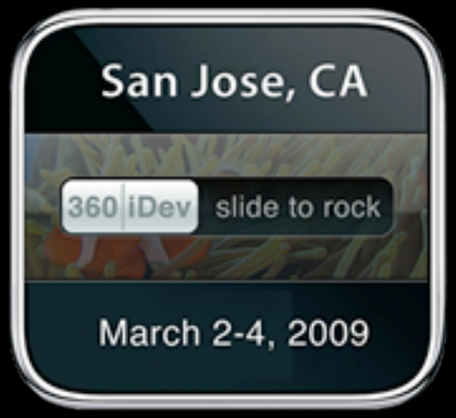


The Basics

SQLitePersistentObjects

- In our Table View Data Source, we just get a count of one of the arrays so we know the number of rows we need in our table:

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:
(NSInteger)section
{
    return [pks count];
}
```

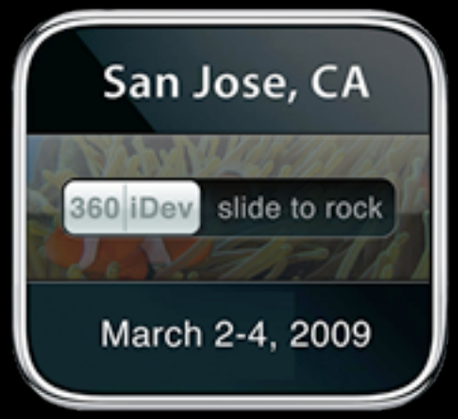


The Basics

SQLitePersistentObjects

- We can then use the information from the arrays to populate our table:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil)
    {
        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
                                        reuseIdentifier:CellIdentifier] autorelease];
    }
    NSUInteger row = [indexPath row];
    cell.text = [NSString stringWithFormat:@"%@, %@",
                 [lastNames objectAtIndex:row],
                 [firstNames objectAtIndex:row]];

    return cell;
}
```

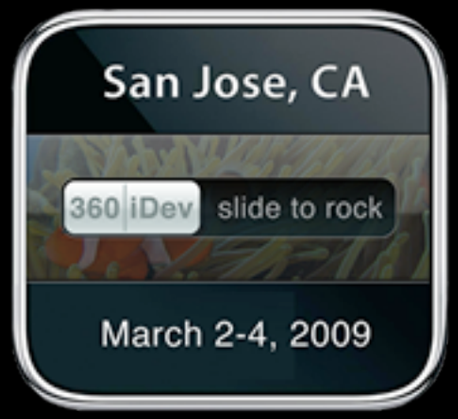


The Basics

SQLitePersistentObjects

- When the user selects a row, we grab the primary key for the selected row, and use that to load the Person:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
   int  thePk = [[pks objectAtIndex:[indexPath row]] intValue];
   Person *thePerson = (Person *)[Person findByPK:thePk];
   // Do something with the person here...
}
```

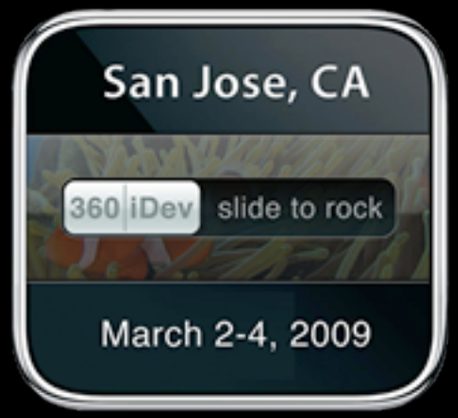


The Basics

SQLitePersistentObjects

Easy Enough, but there's a problem.

What if you want to sort the arrays?



The Basics

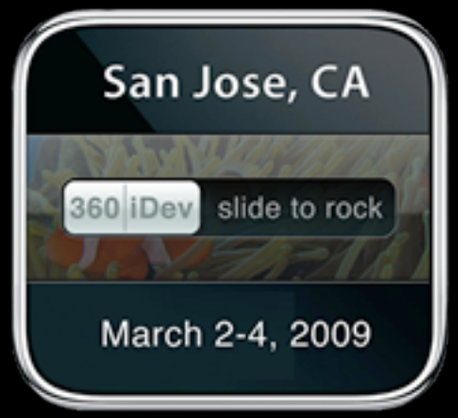
SQLitePersistentObjects

Example: Three NSArrays

pks
1
2
3
4
5

firstNames
Martha
Joe
Sally
George
Buster

lastNames
Washington
Smith
Ride
Washington
Keaton

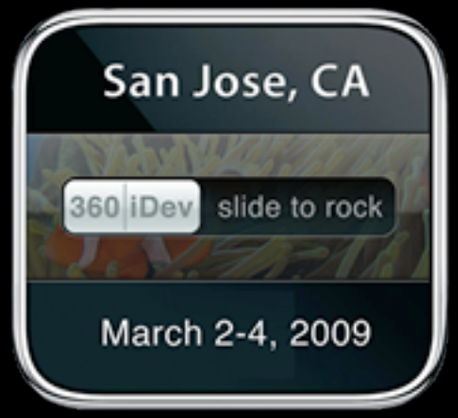


The Basics

SQLitePersistentObjects

Example: Three NSArrays

pks	firstNames	lastNames
1	Buster	Keaton
2	George	Ride
3	Joe	Smith
4	Martha	Washington
5	Sally	Washington



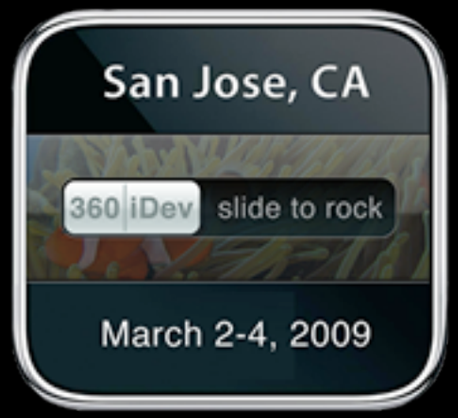
The Basics

SQLitePersistentObjects

- Categories to the Rescue!
- Just call `sortArrayUsingSelector:withPairedMutableArrays:`

```
[lastNames sortArrayUsingSelector:@selector(compare:)
           withPairedMutableArrays:firstNames, pks, nil];
```

- Now all three arrays are sorted based on the last name of the people represented in the arrays.



The Basics

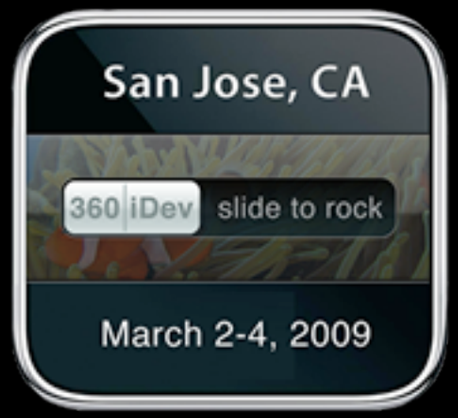
SQLitePersistentObjects

Example: Three NSArrays

pks
1
2
3
4
5

firstNames
Martha
Joe
Sally
George
Buster

lastNames
Washington
Smith
Ride
Washington
Keaton

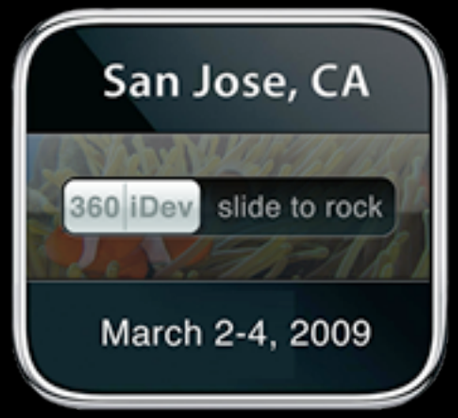


The Basics

SQLitePersistentObjects

Example: Three NSArrays

pks	firstNames	lastNames
5	Buster	Keaton
4	George	Washington
2	Joe	Smith
1	Martha	Washington
3	Sally	Ride

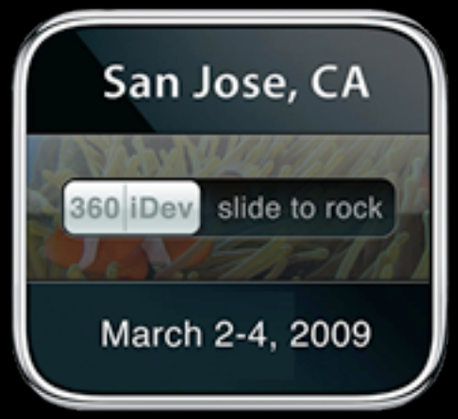


The Not-So-Basics

SQLitePersistentObjects

- SQLite makes aggregations easy (if you know SQL)

```
double averageAge = [Person performSQLAggregation:
@"select avg(date('now') - date(birthdate)) from people where birthdate is not null"];
```

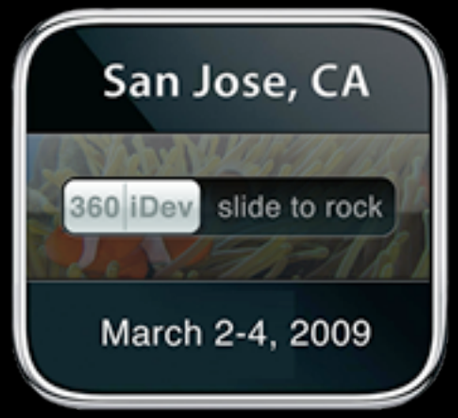


The Not-So-Basics

SQLitePersistentObjects

- But, if you don't know SQL… you're not totally out of luck. SQLPO creates dynamic methods for common aggregations based on your objects' properties:

```
NSNumber *average = [Person averageOfContribution];
NSNumber *washAverage = [Person averageOfContributionWithCriteria:@"where name = 'Washington'"];
NSNumber *sum = [Person sumOfContribution];
NSNumber *washSum =  [Person sumOfContributionWithCriteria:@"where name = 'Washington'"];
NSNumber *count = [Person countOfPk];
NSNumber *min = [Person minOfContribution];
NSNumber *washMin = [Person minOfContributionWithCriteria:@"where name = 'Washington'"];
NSNumber *max = [Person maxOfContribution];
NSNumber *washMax = [Person maxOfContributionWithCriteria:@"where name = 'Washington'"];
```

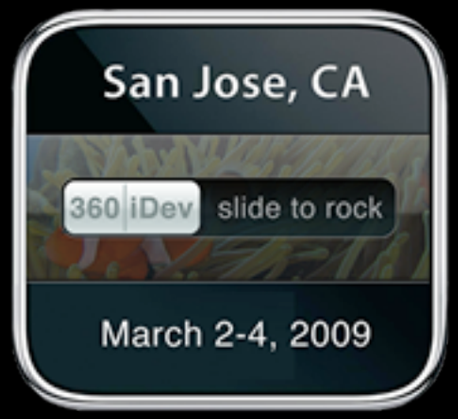


The Not-So-Basics

SQLitePersistentObjects

- Defining how non-standard, non-persistent objects get stored.

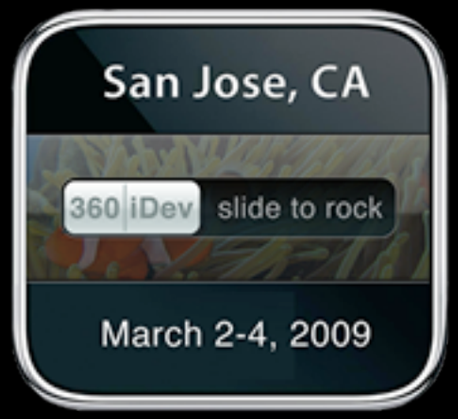
```
@protocol SQLitePersistence
@required
+ (BOOL)canBeStoredInSQLite;
+ (NSString *)columnTypeForObjectStorage;
+ (BOOL)shouldBeStoredInBlob;
@optional
+ (id)objectWithSqlColumnRepresentation:(NSString *)columnData;
- (NSData *)sqlBlobRepresentationOfSelf;
+ (id)objectWithSQLBlobRepresentation:(NSData *)data;
- (NSString *)sqlColumnRepresentationOfSelf;
@end
```



The Not-So-Basics

SQLitePersistentObjects

- The Instance Manager
- SQLPO has a singleton class that manages the database instance.
- Mostly, this class is used behind-the-scenes without any need for you to interact with it.
- But... it's there if you need it.

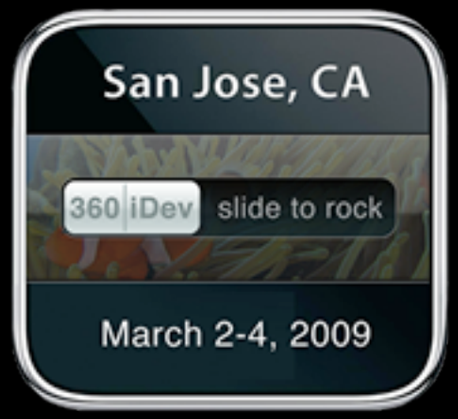


The Not-So-Basics

SQLitePersistentObjects

- Getting the shared instance:

```
SQLiteInstanceManager *manager = [SQLiteInstanceManager sharedManager];
```



The Not-So-Basics

SQLitePersistentObjects

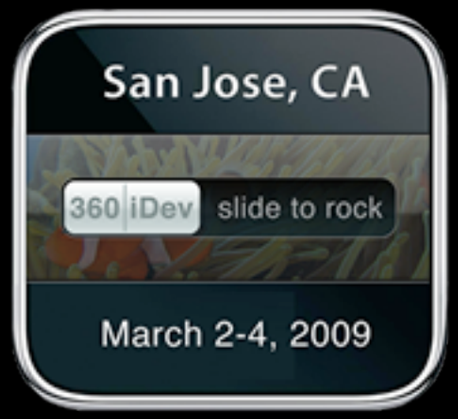
- Once you have it, what can you do with it?

- Get a reference to the database

```
sqlite3 *db = [manager database];
```

- Execute arbitrary SQL Updates:

```
[manager executeUpdateSQL:@"update foo set bar = 1"];
```



The Not-So-Basics

SQLitePersistentObjects

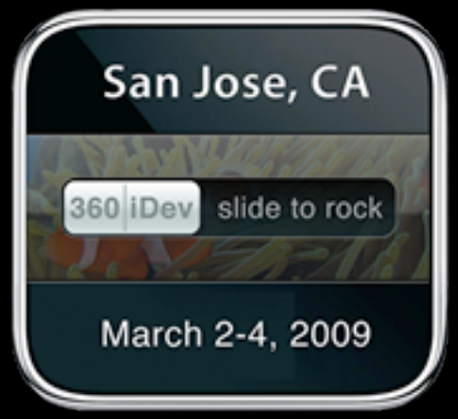
- Once you have it, what can you do with it? (cont)

- Find out if a table exists in the database.

```
BOOL exists = [manager tableExists:@"superheroes"];
```

- Change database configuration and do maintenance

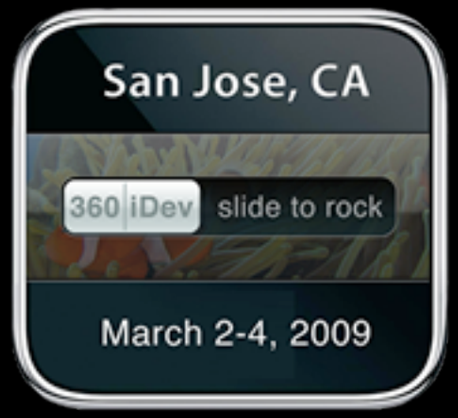
```
[manager vacuum];
[manager setCacheSize:100]
```



Performance

SQLitePersistentObjects

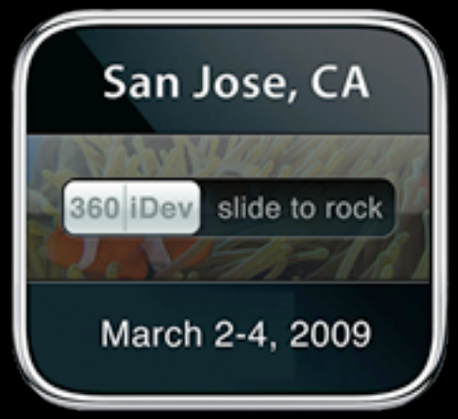
- What Lies Ahead
  - validation architecture & willSave/didSave/okayToSave:
  - transient properties
  - performance optimizations
  - rollback
  - refactoring to generalize code where makes sense
  - ??? (ideas welcome)



Performance

SQLitePersistentObjects

- How well does it perform?
  - Umm... good question.
  - Can we get back to you on that?



SQLitePersistentObjects

Questions?