Developing Applications for iOS

Lecture 9:
Persistence and Blocks

Radu Ionescu
raducu.ionescu@gmail.com
Faculty of Mathematics and Computer Science
University of Bucharest

Content

- Property Lists
- Archiving Objects
- Filesystem Storing
- SQLite
- Blocks
- Grand Central Dispatch

Property Lists

Persistence

- How to make things stick around between launchings of your app (besides `NSUserDefaults`)

Property Lists

- A Property List is any graph of objects containing only the following classes: `NSArray`, `NSDictionary`, `NSNumber`, `NSString`, `NSDate`, `NSData`.
- Use `writeToURL:atomically:` and `initWithContentsOfURL:` in `NSArray` or `NSDictionary`.
- Or `NSUserDefaults` if appropriate.
- Also `NSPropertyListSerialization` converts Property Lists to/from `NSData`.

Archiving

There is a mechanism for making **any** object graph persistent

- Not just graphs with `NSArray` or `NSDictionary` (or other Foundation classes) in them.
- For example, the view hierarchies you build in Interface Builder.

  Those are obviously graphs of very complicated objects.
- Requires all objects in the graph to implement `NSCoding` protocol:

  ```
  - (void)encodeWithCoder:(NSCoder *)coder;
  ```

  ```
  - initWithCoder:(NSCoder *)coder;
  ```
- It is extremely unlikely you will use this in this course. Certainly not during the labs.
- There are other, simpler, (or more appropriate), persistence mechanisms that we are about to discuss.

Archiving

- Object graph is saved by sending all objects `encodeWithCoder:`.

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

  Absolutely, must call `super`’s version or your superclass’s data won’t get written out!

- Object graph is read back in with `alloc`/`initWithCoder:`.

```
- initWithCoder:(NSCoder *)coder
{
   self = [super initWithCoder:coder];
   scale = [coder decodeFloatForKey:@”scale”];
   expression = [coder decodeObjectForKey:@”expression”];
   origin = [coder decodeCGPointForKey:@”origin”];
   // notice that the order does not matter
}
```

Archiving

NSKeyed{Un}Archiver classes are used to store/retrieve graph

- Storage and retrieval is done to NSData objects.
- NSKeyedArchiver stores an object graph to an NSData:

```
+ (NSData *)archivedDataWithRootObject:
                                    (id <NSCoder>)rootObject;
```

- NSKeyedUnarchiver retrieves an object graph from an NSData:

```
+ (id <NSCoder>)unarchiveObjectWithData:(NSData *)data;
```

What do you think this code does?

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

- It makes a “deep copy” of object. But beware, you may get more or less than you expect. Object graphs like “view hierarchies” can be very complicated.

App Sandbox

- Your application can see the iOS file system like a normal Unix file system.
- It starts with the root directory: /.
- There are file protections, of course, like normal Unix, so you can't see everything.

You can only write inside your "Sandbox". Why?

- **Security** - so no one else can damage your application.
- **Privacy** - so no other applications can view your application's data.
- **Cleanup** - when you delete an application, everything its ever written goes with it.

App Sandbox

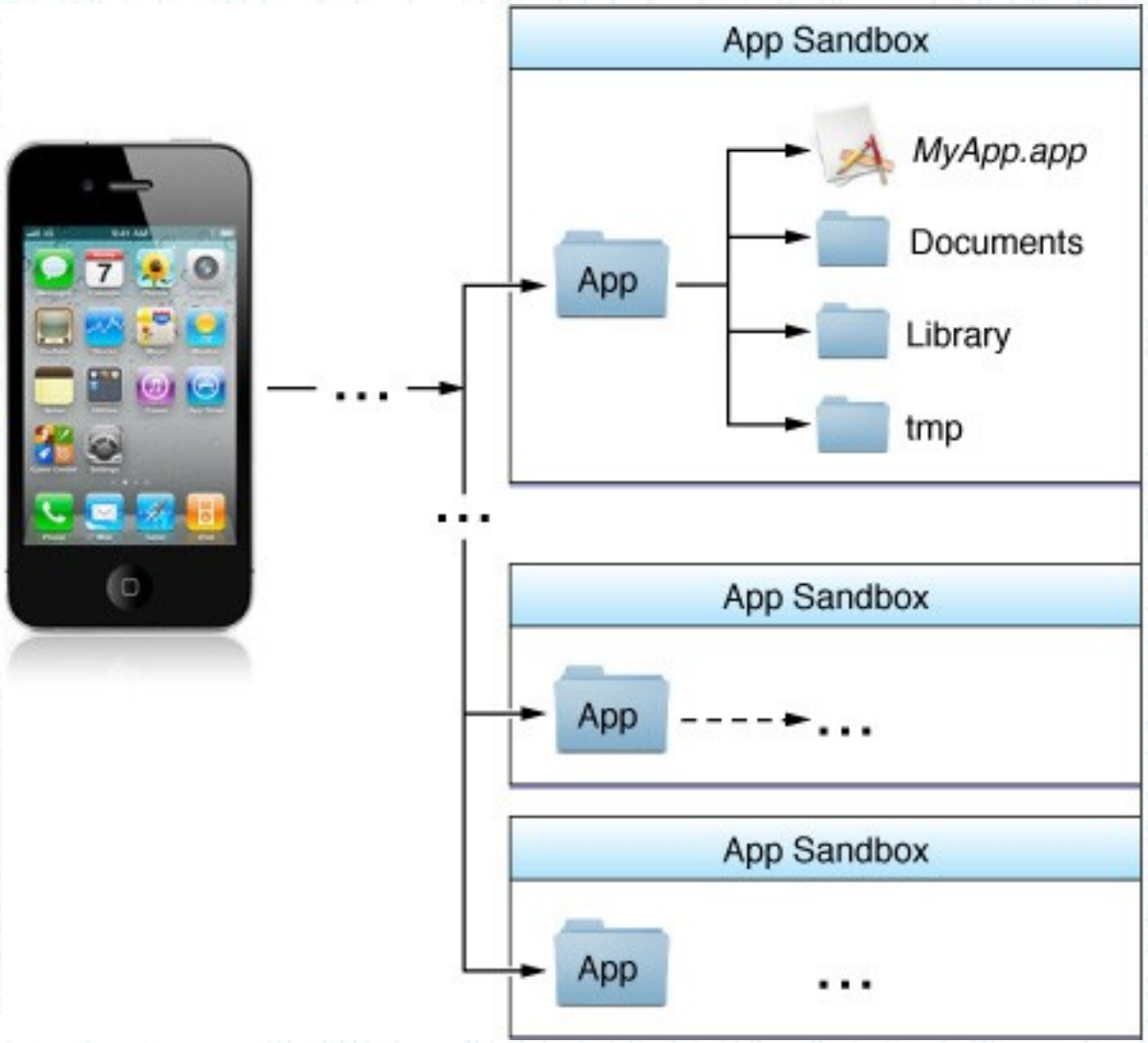
The App “Sandbox”

- As part of the sandboxing process, the system installs your app in its own sandbox directory. It acts as the home for the app and its data.

What’s in this “Sandbox”

- Application bundle directory (binary, .storyboards, .PNGs, etc.). This directory is not writeable.
- Documents directory. This is where you store permanent data created by the user.
- Caches directory. Store temporary files here (this is not backed up by iTunes).
- Other directories (check out `NSSearchPathDirectory` in the documentation).

App Sandbox


App Sandbox
MyApp.app
App
Documents
Library
tmp
App Sandbox
App
App Sandbox
App

File System

What if you want to write to a file you ship with your app?

- Copy it out of your application bundle into the documents (or other) directory to make it writeable.

How do you get the paths to these special sandbox directories?

- Use this `NSFileManager` method:

```
- (NSArray *)URLsForDirectory:(NSSearchPathDirectory)dir
                    inDomains:(NSSearchPathDomainMask)domainMask;
// domainMask is usually NSUserDomainMask
```

- Notice that it returns an `NSArray` of paths (not a single path).

  Since the file system is limited in scope, there is usually only one path in the array in iOS. No user home directory, no shared system directories (for the most part), etc. Thus you will almost always just use `lastObject` (for simplicity).

- Examples of `NSSearchPathDirectory` values: `NSDocumentsDirectory`, `NSCachesDirectory`, `NSAutosavedInformationDirectory`, etc.

NSFileManager

- Provides utility operations (reading and writing is done via `NSData`).
- Check to see if files exist; create and enumerate directories; move, copy, delete, replace files.
- Thread safe (as long as a given instance is only ever used in one thread).
- Just `alloc/init` an instance and start performing operations. If you don't use the `delegate` you can use the `defaultManager`:

```
NSFileManager *manager = [NSFileManager defaultManager];

- (BOOL)createDirectoryAtPath:(NSString *)path
  withIntermediateDirectories:(BOOL)createIntermediates
                   attributes:(NSDictionary*)attributes
                        error:(NSError **)error;

- (BOOL)isReadableFileAtPath:(NSString *)path;

- (NSArray *)contentsOfDirectoryAtPath:(NSString *)path
                                 error:(NSError**)error;
```

- Has a delegate with lots of “should” methods (to do an operation or proceed after an error). Check out the documentation.

File System

NSString

- Path construction methods and reading/writing strings to files:

```
- (NSString *)stringByAppendingPathComponent:
                                    (NSString *)component;

- (NSString *)stringByDeletingLastPathComponent;

- (BOOL)writeToFile:(NSString *)path
         atomically:(BOOL)flag
           encoding:(NSStringEncoding)encoding
              error:(NSError **)error;

// encoding can be ASCII, UTF-8, etc.

- (NSString *)stringWithContentsOfFile:(NSString *)path
              usedEncoding:(NSStringEncoding *)encoding
                     error:(NSError **)error;
```

- And plenty more. Check out the documentation.

SQLite

SQL in a single file

- Fast, low memory, reliable.
- Open Source, comes bundled in iOS.
- Not good for everything (e.g. not video or even serious sounds/images).
- Not a server-based technology (not great at concurrency, but usually not a big deal on a phone).
- Used in countless applications across many platforms, SQLite is considered a standard for lightweight embedded SQL database programming.

SQLite

SQLite API

- Get a database from a file:

```
int sqlite3_open(const char *filename, sqlite3 **db);
```

- Execute SQL statements:

```
int sqlite3_exec(sqlite3 *db, // an opened database
        const char *sql, // the SQL to be executed
        int (*callback)(void *, int, char **, char **),
        void *context,
        char **error);
```

- The callback function gets called for each returned row:

```
int myCallback(void *context, int count,
        char **values, char **cols);
```

- Close the database:

```
int sqlite3_close(sqlite3 *db);
```

Blocks

What is a block?

- Blocks represent typically small, self-contained pieces of code.
- A block of code is a sequence of statements inside `{}`.
- Usually included in-line with the calling of method that is going to use the block of code.
- Very smart about local variables, referenced objects, etc. The block is able to make use of variables from the same scope in which it was defined.
- Using blocks in your iOS (and Mac) applications allows you to attach arbitrary code to Apple-provided methods.
- Similar in concept to delegation, but passing short in-line blocks of code to methods is often more convenient and elegant.
- Although blocks are available to pure C and C++, a block is also always an Objective-C object.

Blocks

What does it look like?

- You use the caret `^` operator to declare a block variable and to indicate the beginning of a block literal. Then it has (optional) arguments in `( )`. The body of the block itself is contained within `{}` (and `;` indicates the end of the statement).
- Here is an example of calling a method that takes a block as an argument:

```
[dict enumerateKeysAndObjectsUsingBlock:
 ^(id key, id value, BOOL *stop)
 {
    NSLog(@"Drinking %@ is %@.", value, key);
    if ([@"too much" isEqualToString:key])
    {
       *stop = YES;
    }
 }];
```

- This `NSLog()`s every `key` and `value` in `dict` (but stops if the key is `@"too much"`).

Blocks

- Can use local variables declared before the block inside the block:

```
NSString *maxQuantity = @"four";

[dict enumerateKeysAndObjectsUsingBlock:
 ^(id key, id value, BOOL *stop)
 {
    NSLog(@"Drinking %@ is %@.", value, key);
    if ([@"too much" isEqualToString:key]
        || [value hasPrefix:maxQuantity])
    {
       *stop = YES;
    }
 }];
```

Blocks

- But they are read only!

```
NSString *maxQuantity = @"four";

[dict enumerateKeysAndObjectsUsingBlock:
 ^(id key, id value, BOOL *stop)
 {
    // This is ILLEGAL:
    maxQuantity = @"Doesn't matter, get drunk!";

    NSLog(@"Drinking %@ is %@.", value, key);
    if ([@"too much" isEqualToString:key]
        || [value hasPrefix:maxQuantity])
    {
       *stop = YES;
    }
 }];
```

Blocks

- Unless you mark the local variable as `__block`:

```
__block NSString *maxQuantity = @"four";

[dict enumerateKeysAndObjectsUsingBlock:
 ^(id key, id value, BOOL *stop)
 {
    // This is ok now:
    maxQuantity = @"Doesn't matter, get drunk!";

    NSLog(@"Drinking %@ is %@.", value, key);
    if ([@"too much" isEqualToString:key]
        || [value hasPrefix:maxQuantity])
    {
       *stop = YES;
    }
 }];
```

- Or if the “variable” is an instance variable.

  But we only access instance variables (e.g. `_display`) in setters and getters. So this is of minimal value to us.

Blocks

- So what about objects which are messaged inside the block?

```
__block NSString *maxQuantity = @"four";

[dict enumerateKeysAndObjectsUsingBlock:
 ^(id key, id value, BOOL *stop)
 {
    // This is ok now:
    maxQuantity = @"Doesn't matter, get drunk!";

    NSLog(@"Drinking %@ is %@.", value, key);
    if ([@"too much" isEqualToString:key]
        || [maxQuantity hasPrefix:value])
    {
       *stop = YES;
    }
 }];
```

- `maxQuantity` will essentially have a `strong` pointer to it until the block goes out of scope or the block itself leaves the heap (i.e. no one points `strongly` to the block anymore).

Blocks

- Imagine we added the following method to `CalculatorBrain`:

```
- (void)addUnaryOperation:(NSString *)operation
        whichExecutesBlock:...;
```

- This method adds another operation to the brain (like sqrt) which you get to specify the code for.

  For now, we'll not worry about the syntax for passing the block (but the mechanism for that is the same as for defining `enumerateKeysAndObjectsUsingBlock:`).

- That block we pass in will not be executed until much later. It will be executed only when that "operation" is pressed in some UI somewhere.

Blocks

- Example call of `addUnaryOperation:whichExecutesBlock:`.

```
NSNumber *secret = [NSNumber numberWithDouble:7.0];
[brain addUnaryOperation:@"luckyMultiply"
      whichExecutesBlock: ^(double operand) {
    return operand * [secret doubleValue];
}];
```

- Imagine if `secret` were not automatically kept in the heap here.
- What would happen later when this block executed (when the `@"luckyMultiply"` operation was pressed)?

  Bad things! Luckily, `secret` is automatically kept in the heap until block can't be run anymore. This means blocks capture their surrounding state.
- Blocks are also called closures because they close around variables that are in scope at the time the block is declared.

Blocks

Creating a type for a variable that can hold a block

- Blocks are kind of like “objects” with an unusual syntax for declaring variables that hold them.
- Usually if we are going to store a block in a variable, we `typedef` a type for that variable:

```
typedef double (^unary_operation_t)(double op);
```

Returns a `double`.

The `double` argument is named `op`.

- This declares a type called `unary_operation_t` for variables which can store a block. Specifically, a block which takes a `double` as its only argument and returns a `double`.

Blocks

Creating a type for a variable that can hold a block

- Then we could declare a variable of this type and give it a value:

```
unary_operation_t square;
square = ^(double operand) {
    // the value of the square variable is a block
    return operand * operand;
};
```

- And then use the variable `square` like this:

```
double squareOfFive = square(5.0);
// squareOfFive has the value 25.0 after this
```

- You don't have to `typedef`, for example, the following is also a legal way to create `square`:

```
double (^square)(double op) = ^(double op) {
    return op * op;
};
```

Blocks

We could then use the `unary_operation_t` to define a method

- For example, `addUnaryOperation:whichExecutesBlock:`.
- We would have to add this property to our `CalculatorBrain`:

```
@property (nonatomic, strong)
                NSMutableDictionary *unaryOperations;
```

- Then implement the method like this:

```
typedef double (^unary_operation_t)(double op);

- (void)addUnaryOperation:(NSString *)op
       whichExecutesBlock:(unary_operation_t)opBlock
{
    [self.unaryOperations setObject:opBlock
                             forKey:op];
}
```

- Note that the block can be treated somewhat like an object (e.g., adding it to a dictionary).

Blocks

- Later in our `CalculatorBrain` we could use an operation added with the method above like this:

```
- (double)performOperation:(NSString *)operation
{
    unary_operation_t unaryOp =
        [self.unaryOperations objectForKey:operation];

    if (unaryOp)
    {
        self.operand = unaryOp(self.operand);
    }

    ...
}
```

Blocks

We don't always `typedef`

- When a block is an argument to a method and is used immediately, often there is no `typedef`.
- Here is the declaration of the `NSDictionary` enumerating method we showed earlier:

```
- (void)enumerateKeysAndObjectsUsingBlock:
            (void (^)(id key, id obj, BOOL *stop))block;
```

No name for the type appears here.

This is the local variable name for the argument inside the method implementation.

- The syntax is exactly the same as the `typedef` except that the type name is not there.
- For reference, a `typedef` for this argument would look like this:

```
typedef
  void (^enumeratingBlock)(id key, id obj, BOOL *stop);
```

Blocks

Some shorthand allowed when defining a block

- "Defining" means you are writing the code between the {}.
- 1. You do not have to declare the return type if it can be inferred from your code in the block. If the return type is inferred and multiple return statements are present, they must exactly match (using casting if necessary).
- 2. If there are no arguments to the block, you do not need to have any parentheses.
- Recall this code:

```
NSNumber *secret = [NSNumber numberWithDouble:7.0];
[brain addUnaryOperation:@"luckyMultiply"
      whichExecutesBlock: ^(double operand) {
    return operand * [secret doubleValue];
}];
```

No return type. Inferred from the `return` inside.

Blocks

Some shorthand allowed when defining a block

- “Defining” means you are writing the code between the {}.
- 1. You do not have to declare the return type if it can be inferred from your code in the block. If the return type is inferred and multiple return statements are present, they must exactly match (using casting if necessary).
- 2. If there are no arguments to the block, you do not need to have any parentheses.
- Another example:

```
[UIView animateWithDuration:5.0 animations:^ {
    view.opacity = 0.5;
}];
```

No arguments to this block. There is no need to say ^( ) {...}.

Memory Cycles (a bad thing)

- What if you had the following property in a class?

```
@property (nonatomic, strong) NSArray *myBlocks;
```

- And then tried to do the following in one of that class's methods?

```
[self.myBlocks addObject:^() {
    [self doSomething];
}];
```

- We said that all objects referenced inside a block will stay in the heap as long as the block does. In other words, blocks keep a `strong` pointer to all objects referenced inside of them.
- In this case, `self` is an object reference in this block. Thus the block will have a `strong` pointer to `self`.
- But notice that `self` also has a strong pointer to the block (through its `myBlocks` property)!
- **This is a serious problem.** Neither `self` nor the block can ever escape the heap now.
- That's because there will always be a `strong` pointer to both of them (each other's pointer). This is called a memory "cycle".

Memory Cycles Solution

- You'll recall that local variables are always `strong`.
- That's okay because when they go out of scope, they disappear, so the `strong` pointer goes away.
- But there's a way to declare that a local variable is `weak`:

```
__weak MyClass *weakSelf = self;

[self.myBlocks addObject:^() {
    [weakSelf doSomething];
}];
```

- This solves the problem because now the block only has a `weak` pointer to `self`.
- Note that `self` still has a `strong` pointer to the block, but that's okay.
- As long as someone in the universe has a `strong` pointer to this `self`, the block's pointer is good. And since the block will not exist if `self` does not exist (since `myBlocks` won't exist), all is well!

Recap

A block is an anonymous inline collection of code that:

- Has a typed argument list just like a function.
- Has an inferred or declared return type.
- Can capture state from the lexical scope within which it is defined.
- Can optionally modify the state of the scope.
- Can share the potential for modification with other blocks defined within the same lexical scope.

Blocks

When do we use blocks in iOS?

- Enumeration
- View Animations (this is really nice and easy)
- Sorting (sort this thing using a block as the comparison method)
- Notification (when something happens, execute this block)
- Error handlers (if an error happens while doing this, execute this block)
- Completion handlers (when you are done doing this, execute this block)
- And a super-important use: Multithreading

  Using the Grand Central Dispatch (GCD) API. This is also a C API by the way.

Grand Central Dispatch

The basic idea is that you have queues of operations

- The operations are specified using blocks.
- Most queues run their operations serially (a true “queue”). We’re only going to talk about serial queues.

The system runs operations from queues in separate threads

- Though there is no guarantee about how/when this will happen.
- All you know is that your queue’s operations will get run (in order) at some point. The good thing about this is that if your operation blocks, only that queue will block.
- Other queues (like the main queue, where UI is happening) will continue to run.

So how can we use this to our advantage?

- Get blocking activity (e.g. network) out of our user-interface (main) thread. Do time-consuming activity concurrently in another thread.

Grand Central Dispatch

Important functions in this C API

- Creating and releasing serial queues:

```
dispatch_queue_t dispatch_queue_create(const char *label,
                                       NULL);

void dispatch_release(dispatch_queue_t);
```

- Putting blocks in the queue:

```
typedef void (^dispatch_block_t)(void);

void dispatch_async(dispatch_queue_t queue,
                    dispatch_block_t block);
```

- Getting the current or main queue:

```
dispatch_queue_t dispatch_get_current_queue();

void dispatch_queue_retain(dispatch_queue_t);
// keep it in the heap until dispatch_release

dispatch_queue_t dispatch_get_main_queue();
```

Grand Central Dispatch

What does it look like to call these?

- Example: fetching an image from the network (this would be slow).

```
- (void)viewWillAppear:(BOOL)animated
{
    NSData *data = [NSData dataWithContentsOfURL:imageURL];
    UIImage *image = [UIImage imageWithData:data];
    self.imageView.image = image;
    self.scrollView.contentSize = image.size;
}
```

Grand Central Dispatch

What does it look like to call these?

- Example: fetching an image from the network (this would be slow).

```
- (void)viewWillAppear:(BOOL)animated
{



        NSData *data = [NSData dataWithContentsOfURL:imageURL];
        UIImage *image = [UIImage imageWithData:data];
        self.imageView.image = image;
        self.scrollView.contentSize = image.size;


}
```

Grand Central Dispatch

What does it look like to call these?

- Example: fetching an image from the network (this would be slow).

```
- (void)viewWillAppear:(BOOL)animated
{
    dispatch_queue_t downloadQueue =
            dispatch_queue_create("image downloader", NULL);

        NSData *data = [NSData dataWithContentsOfURL:imageURL];
        UIImage *image = [UIImage imageWithData:data];
        self.imageView.image = image;
        self.scrollView.contentSize = image.size;

}
```

Grand Central Dispatch

What does it look like to call these?

- Example: fetching an image from the network (this would be slow).

```
- (void)viewWillAppear:(BOOL)animated
{
    dispatch_queue_t downloadQueue =
             dispatch_queue_create("image downloader", NULL);
    dispatch_async(downloadQueue, ^{
        NSData *data = [NSData dataWithContentsOfURL:imageURL];
        UIImage *image = [UIImage imageWithData:data];
        self.imageView.image = image;
        self.scrollView.contentSize = image.size;
    });
}
```

Grand Central Dispatch

What does it look like to call these?

- Example: fetching an image from the network (this would be slow).

```
- (void)viewWillAppear:(BOOL)animated
{
    dispatch_queue_t downloadQueue =
             dispatch_queue_create(“image downloader”, NULL);
    dispatch_async(downloadQueue, ^{
        NSData *data = [NSData dataWithContentsOfURL:imageURL];
        UIImage *image = [UIImage imageWithData:data];
        self.imageView.image = image;
        self.scrollView.contentSize = image.size;
    });
}
```

- **Problem:** UIKit calls can only happen in the main thread!

Grand Central Dispatch

What does it look like to call these?

- Example: fetching an image from the network (this would be slow).

```
- (void)viewWillAppear:(BOOL)animated
{
    dispatch_queue_t downloadQueue =
            dispatch_queue_create("image downloader", NULL);
    dispatch_async(downloadQueue, ^{
        NSData *data = [NSData dataWithContentsOfURL:imageURL];

            UIImage *image = [UIImage imageWithData:data];
            self.imageView.image = image;
            self.scrollView.contentSize = image.size;

    });
}
```

Grand Central Dispatch

What does it look like to call these?

- Example: fetching an image from the network (this would be slow).

```
- (void)viewWillAppear:(BOOL)animated
{
    dispatch_queue_t downloadQueue =
            dispatch_queue_create("image downloader", NULL);
    dispatch_async(downloadQueue, ^{
        NSData *data = [NSData dataWithContentsOfURL:imageURL];
        dispatch_async(dispatch_get_main_queue(), ^{
            UIImage *image = [UIImage imageWithData:data];
            self.imageView.image = image;
            self.scrollView.contentSize = image.size;
        });
    });
}
```

Grand Central Dispatch

What does it look like to call these?

- Example: fetching an image from the network (this would be slow).

```
- (void)viewWillAppear:(BOOL)animated
{
    dispatch_queue_t downloadQueue =
            dispatch_queue_create("image downloader", NULL);
    dispatch_async(downloadQueue, ^{
        NSData *data = [NSData dataWithContentsOfURL:imageURL];
        dispatch_async(dispatch_get_main_queue(), ^{
            UIImage *image = [UIImage imageWithData:data];
            self.imageView.image = image;
            self.scrollView.contentSize = image.size;
        });
    });
}
```

- **Problem:** This "leaks" the `downloadQueue` in the heap. We have to `dispatch_release` it.

Grand Central Dispatch

What does it look like to call these?

- Example: fetching an image from the network (this would be slow).

```
- (void)viewWillAppear:(BOOL)animated
{
    dispatch_queue_t downloadQueue =
             dispatch_queue_create("image downloader", NULL);
    dispatch_async(downloadQueue, ^{
        NSData *data = [NSData dataWithContentsOfURL:imageURL];
        dispatch_async(dispatch_get_main_queue(), ^{
            UIImage *image = [UIImage imageWithData:data];
            self.imageView.image = image;
            self.scrollView.contentSize = image.size;
        });
    });
    dispatch_release(downloadQueue);
}
```

- Don't worry, it won't remove the queue from the heap until all blocks have been processed.

Next Time

Core Data and Categories:

- Core Data and Documents
- `NSNotificationCenter`
- Objective-C Categories