CS193P - Lecture 10

iPhone Application Development

Performance

Announcements

- Presence 2 is due tomorrow (May 5) at 11:59pm
- Presence 3 assignment will be released tomorrow
- Final project proposals due on Monday (May 11)
  - See class website for more details

Today's Topics

- Memory Usage
  - Leaks
  - Autorelease
  - System warnings
- Concurrency
  - Threads
  - Operations and queues
- Additional Tips & Tricks

iPhone Performance Overview

- iPhone applications must work with...
  - Limited memory
  - Slow or unavailable network resources
  - Less powerful hardware
- Write your code with these constraints in mind
- **Use performance tools** to figure out where to invest

Memory Usage

Memory on the iPhone

- Starting points for performance
  - Load lazily
  - Don’t leak
  - Watch your autorelease footprint
  - Reuse memory
- System memory warnings are a last resort
  - Respond to warnings or be terminated

Loading Lazily

- Pervasive in Cocoa frameworks
- Do only as much work as is required
  - Application launch time!
- Think about where your code **really** belongs
- Use multiple NIBs for your user interface

Loading a Resource Too Early

- What if it's not needed until much later? Or not at all?

```
- (id)init
{
    self = [super init];
    if (self) {
        // Too early...
        myImage = [self readSomeHugeImageFromDisk];
    }
    return self;
}
```

Loading a Resource Lazily

- Wait until someone actually requests it, then create it

```
- (UIImage *)myImage
{
   if (myImage == nil) {
      myImage = [self readSomeHugeImageFromDisk];
   }
}
```

- Ends up benefiting both memory and launch time
- Not always the right move, consider your specific situation
- Notice that above implementation is **not thread-safe!**

Plugging Leaks

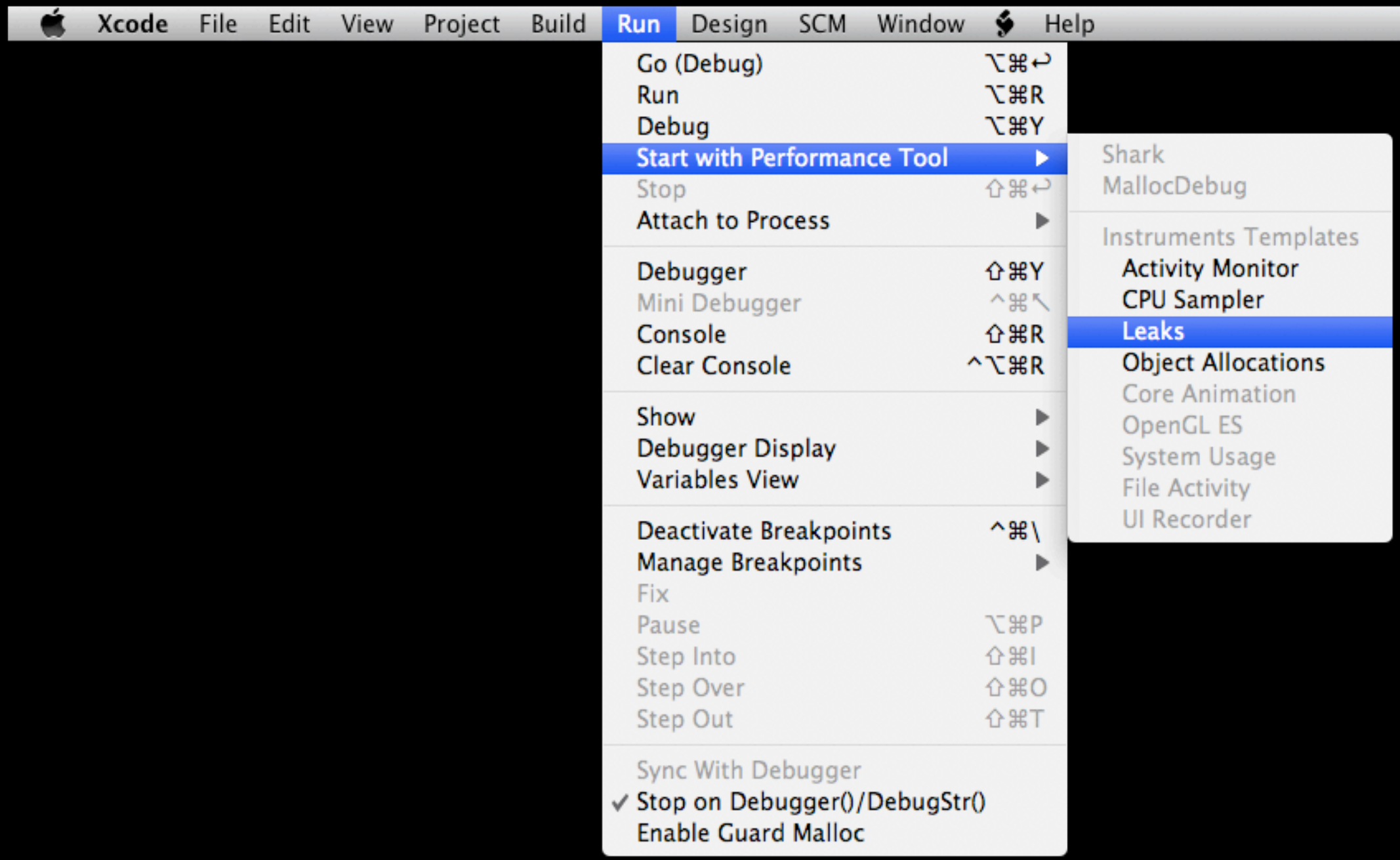
- Memory leaks are very bad
  - Especially in code that runs often
- Luckily, leaks are **easy to find** with the right tools

Method Naming and Object Ownership

- If a method's name contains **alloc**, **copy** or **new**, then it **returns a retained object**
- Balance calls to alloc, copy, new or retain with calls to release or autorelease
  - **Early returns** can make this very difficult to do!

Finding Leaks

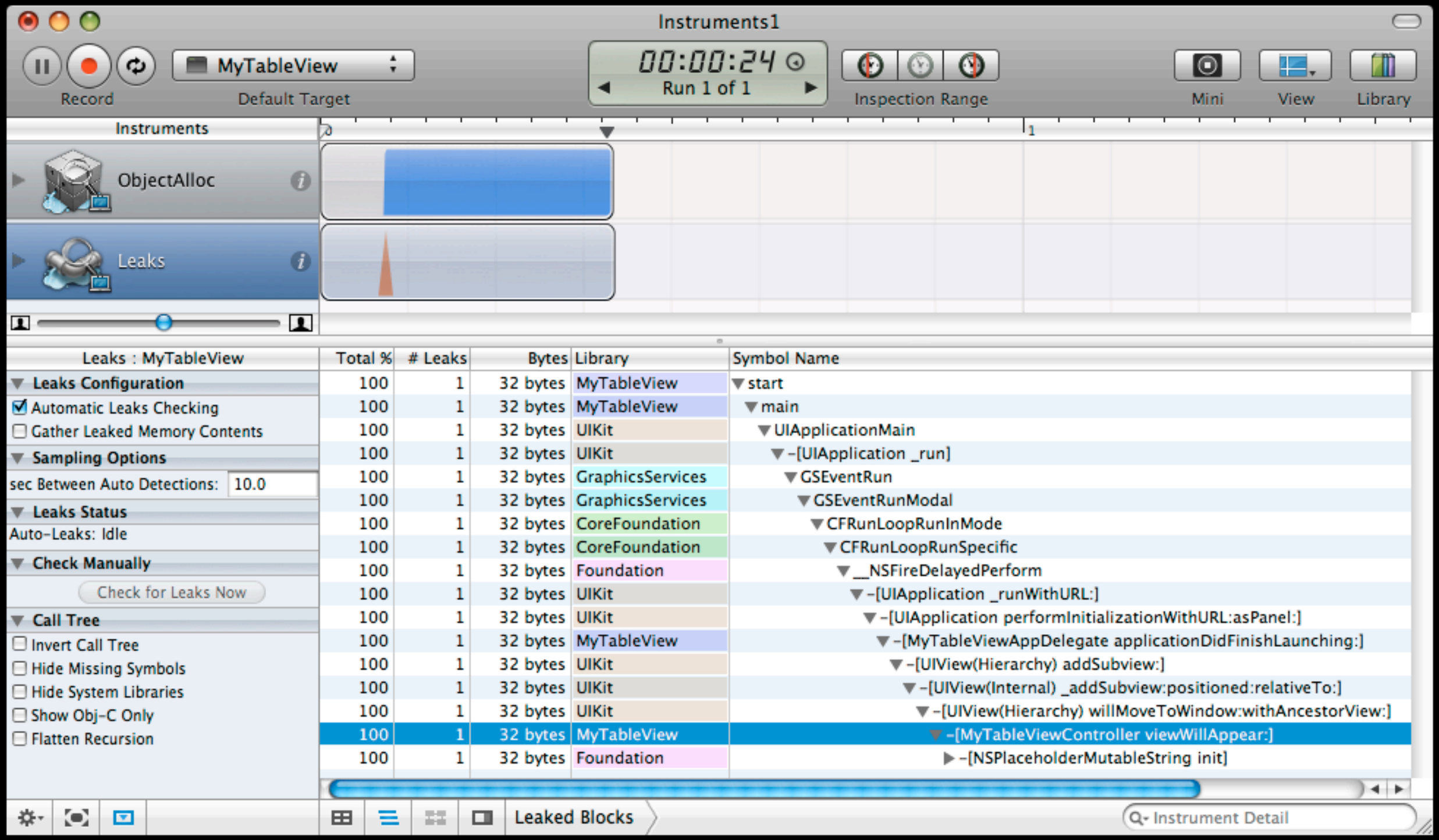
- Use **Instruments** with the **Leaks** recorder



Identifying Leaks in Instruments

- Each leak comes with a backtrace
- Leaks in system code do exist, but they're rare
  - If you find one, tell us at http://bugreport.apple.com
- Consider your own application code first

Caught in the Act



Demo:
Finding Leaks with Instruments

Autorelease and You

- Autorelease **simplifies your code**
  - Worry less about the scope and lifetime of objects
- When an autorelease pool pops, it calls -release on each object
- An autorelease pool is created automatically for each iteration of your application's run loop

So What's the Catch?

- What if many objects are autoreleased before the pool pops?
- Consider the **maximum memory footprint** of your application

A Crowded Pool...

Reducing Your High-Water Mark

- When many objects will be autoreleased, **create and release your own pool**
  - Usually not necessary, don't do this without thinking!
  - Tools can help identify cases where it's needed
  - **Loops** are the classic case

Autorelease in a Loop

- Remember that many methods return autoreleased objects

```
for (int i = 0; i < someLargeNumber; i++) {
  NSString *string = ...;
  string = [string lowercaseString];
  string = [string stringByAppendingString:...];
  NSLog(@“%@”, string);
}
```

Creating an Autorelease Pool

- One option is to create and release for each iteration

```
for (int i = 0; i < someLargeNumber; i++) {
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

  NSString *string = ...;
  string = [string lowercaseString];
  string = [string stringByAppendingString:...];
  NSLog(@“%@”, string);

  [pool release];
}
```

Outliving the Autorelease Pool

- What if some object is needed outside the scope of the pool?

```
NSString *stringToReturn = nil;

for (int i = 0; i < someLargeNumber; i++) {
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

  NSString *string = ...;
  string = [string stringByAppendingString:...];

  if ([string someCondition]) {
    stringToReturn = [string retain];
  }

  [pool release];
  if (stringToReturn) break;
}

return [stringToReturn autorelease];
```

Reducing Use of Autorelease

- Another option is to cut down on use of autoreleased objects
  - Not always possible if you're callling into someone else's code
- When it makes sense, switch to alloc/init/release
- In previous example, perhaps use a single NSMutableString?

Demo:
Measuring Your High-Water Mark

Object Creation Overhead

- Most of the time, creating and deallocating objects is **not** a insignificant hit to application performance
- In a tight loop, though, it can become a problem...

```
for (int i = 0; i < someLargeNumber; i++) {
    MyObject *object = [[MyObject alloc] initWithValue:...];
    [object doSomething];
    [object release];
}
```

Reusing Objects

- Update existing objects rather than creating new ones
- Combine **intuition** and **evidence** to decide if it's necessary

```
MyObject *myObject = [[MyObject alloc] init];

for (int i = 0; i < someLargeNumber; i++) {
  myObject.value = ...;
  [myObject doSomething];
}

[myObject release];
```

- Remember -[UITableView dequeueReusableCellWithIdentifier]

Memory Warnings

- Coexist with system applications
- Memory warnings issued when memory runs out
- Respond to memory warnings or **face dire consequences!**

Responding to Memory Warnings

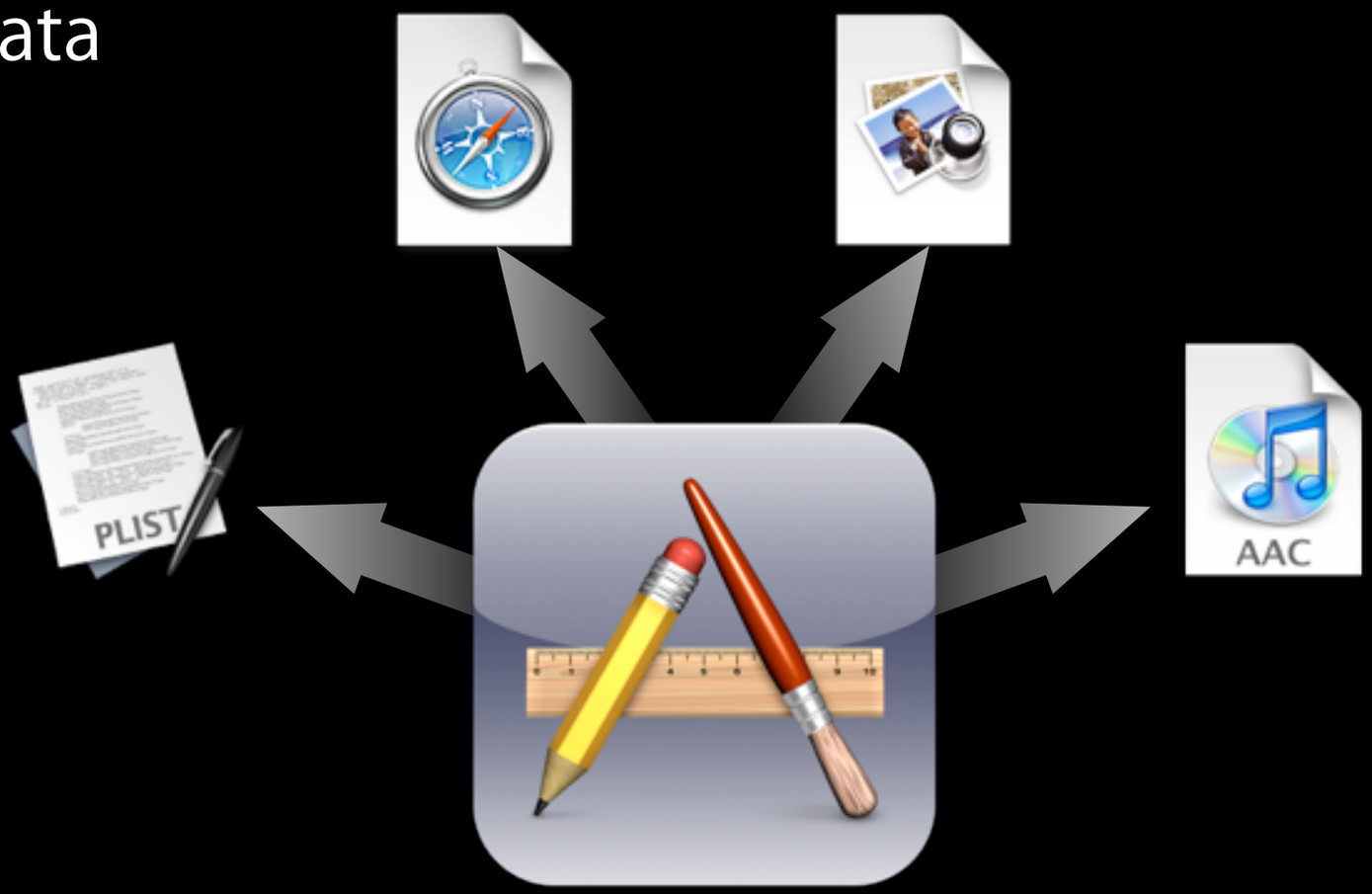
- Every view controller gets -didReceiveMemoryWarning
  - By default, releases the view if it's not visible
  - Release other expensive resources in your subclass

```
- (void)didReceiveMemoryWarning
{
    // Always call super
    [super didReceiveMemoryWarning];

    // Release expensive resources
    [expensiveResource release];
    expensiveResource = nil;
}
```

What Other Resources Do I Release?

- Images
- Sounds
- Cached data



Use SQLite for Large Data Sets

- Many data formats keep everything in memory
- SQLite can work with your data in chunks

More on Memory Performance

- “Memory Usage Performance Guidelines”
  https://developer.apple.com/iphone/library/documentation/Performance/Conceptual/ManagingMemory/

Concurrency

Why Concurrency?

- With a single thread, long-running operations may interfere with user interaction
- Multiple threads allow you to load resources or perform computations without locking up your entire application

Threads on the iPhone

- Based on the POSIX threading API
  - /usr/include/pthread.h
- Higher-level wrappers in the Foundation framework

NSThread Basics

- Run loop automatically instantiated for each thread
- Each NSThread needs to create its own autorelease pool
- Convenience methods for messaging between threads

Typical NSThread Use Case

```
- (void)someAction:(id)sender
{
  // Fire up a new thread
  [NSThread detachNewThreadSelector:@selector(doWork:)
                           withTarget:self object:someData];
}

- (void)doWork:(id)someData
{
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

  [someData doLotsOfWork];

  // Message back to the main thread
  [self performSelectorOnMainThread:@selector(allDone:)
         withObject:[someData result] waitUntilDone:NO];

  [pool release];
}
```

UIKit and Threads

- Unless otherwise noted, UIKit classes are **not threadsafe**
  - Objects must be created and messaged from the main thread

Demo:
Threads and Xcode

Locks

- Protect critical sections of code, mediate access to shared data
- NSLock and subclasses

```
- (void)someMethod
{
   [myLock lock];
   // We only want one thread executing this code at once
   [myLock unlock]
}
```

Conditions

- NSCondition is useful for producer/consumer model

```
// On the producer thread
- (void)produceData
{
   [condition lock];

   // Produce new data
   newDataExists = YES;

   [condition signal];
   [condition unlock];
}
```

```
// On the consumer thread
- (void)consumeData
{
   [condition lock];
   while(!newDataExists) {
      [condition wait];
   }

   // Consume the new data
   newDataExists = NO;

   [condition unlock];
}
```

- **Wait** is equivalent to: **unlock**, **sleep** until signalled, **lock**

The Danger of Locks

- **Very difficult** to get locking right!
- All it takes is one client poorly behaved client
  - Accessing shared data outside of a lock
  - Deadlocks
  - Priority inversion

Threading Pitfalls

- Subtle, **nondeterministic bugs** may be introduced
- Code may become **more difficult to maintain**
- In the worst case, more threads can mean **slower code**

Alternatives to Threading

- Asynchronous (nonblocking) functions
  - Specify target/action or delegate for callback
  - **NSURLConnection** has synchronous and asynchronous variants
- Timers
  - One-shot or recurring
  - Specify a callback method
  - Managed by the run loop
- Higher level constructs like **operations**

NSOperation

- Abstract superclass
- Manages thread creation and lifecycle
- Encapsulate a **unit of work** in an object
- Specify priorities and dependencies

Creating an NSOperation Subclass

- Define a **custom init method**

```
- (id)initWithSomeObject:(id)someObject
{
    self = [super init];
    if (self) {
        self.someObject = someObject;
    }
    return self;
}
```

- **Override -main method** to do work

```
- (void)main
{
    [someObject doLotsOfTimeConsumingWork];
}
```

Using an NSInvocationOperation

- Concrete subclass of NSOperation
- For lightweight tasks where creating a subclass is overkill

```
- (void)someAction:(id)sender
{
    NSInvocationOperation *operation =
        [[NSInvocationOperation alloc] initWithTarget:self
                                             selector:@selector(doWork:)
                                               object:someObject];

    [queue addOperation:operation];

    [operation release];
}
```

NSOperationQueue

- Operations are typically scheduled by **adding to a queue**
- Choose a maximum number of concurrent operations
- Queue runs operations based on priority and dependencies

Demo:
Threaded Flickr Loading

More on Concurrent Programming

- “Threading Programming Guide”
  https://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/Multithreading

Additional Tips & Tricks

Drawing Performance

- Avoid transparency when possible
  - Opaque views are much faster to draw than transparent views
  - Especially important when scrolling

- Don’t call -drawRect: yourself
- Use -setNeedsDisplayInRect: instead of -setNeedsDisplay

Reuse Table View Cells

- UITableView provides mechanism for reusing table view cells

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Ask the table view if it has a cell we can reuse
    UITableViewCell *cell =
    [tableView dequeueReusableCellWithIdentifier:MyIdentifier];

    if (!cell) { // If not, create one with our identifier
        cell = [[UITableViewCell alloc] initWithFrame:CGRectZero
                                           identifier:MyIdentifier];
        [cell autorelease];
    }

    return cell;
}
```

Get notified

- Don’t continously poll!
  - Unless you must, which is rare
- Hurts both responsiveness and battery life
- Look in the documentation for a notification, delegate callback or other asynchronous API

Recap

- Performance is an art and a science
  - Combine tools & concrete data with intuition & best practices
- Don’t waste memory
- Concurrency is tricky, abstract it if possible
- Drawing is expensive, avoid unnecessary work

Questions?