Announcements

–Final Project descriptions are due tonight

–Final Project Proposal slides are due tomorrow night

Final Project Point Distribution and Due Dates

- **The final project accounts for 30% of your final grade**
  - Final project score is out of 100 points
- **Final Project Group Description – 5 points**
  - Due on Monday November 5th by 11:59 PM
- **Project Proposal Presentation – 15 points**
  - Submit as PPT, Keynote, or Google slides
  - Due on Tuesday November 6th by 11:59 PM
- **Project Update – 10 points**
  - Provide a brief description of what has been accomplished in the email
  - Submit code to demonstrate the accomplishments
  - Submit either a 30 second recording or multiple screenshots (5 max) of your app
  - Due on Sunday November 25th by 11:59 PM
- **Final Project Code - 70 points**
  - Due on Monday December 3rd by 11:40 AM
- **Submit all portions of the final project to cse438ta@gmail.com**
- **Late submissions will result in a 0 for that portion of the final project**
- **Final Project groups consist of 4 - 5 people**

Today's Topics

- **Final Project App Idea**
- **Objective-C Language**

Guest Presentation

- **Amy Mueller**
  - Nurse Practitioner with the Adult Congenital Cardiology Team at Children's Hospital
  - amy.mueller@bjc.org

Collections

- **Array - ordered collection of objects**
- **Dictionary - collection of key-value pairs**
- **Set - unordered collection of unique objects**

- **Common enumeration mechanism**
- **Immutable and mutable versions**
- **Immutable collections can be shared without side effect**
  - Prevents unexpected changes
  - Mutable objects typically carry a performance overhead

NSArray

- **Common NSArray methods**

```
+ arrayWithObjects:(id)firstObj, ...; // nil terminated!!!
  -(unsigned)count;
 -(id)objectAtIndex:(unsigned)index;
 -(unsigned)indexOfObject:(id)object;
```

- **NSNotFound returned for index if not found**

```
NSArray *array = [NSArray arrayWithObjects:@"Red", @"Blue",
    @"Green",nil];

if ([array indexOfObject:@"Purple"] == NSNotFound) {
    NSLog (@"No color purple");
}
```

NSMutableArray

- **NSMutableArray subclasses NSArray**
  - So, everything in NSArray
- **Common NSMutableArray Methods**

```
+ (NSMutableArray *)array;
- (void)addObject:(id)object;
- (void)removeObject:(id)object;
- (void)removeAllObjects;
- (void)insertObject:(id)object atIndex:(unsigned)index;
```

```
NSMutableArray *array = [NSMutableArray array];
[array addObject:@"Red"];
[array addObject:@"Green"];
[array addObject:@"Blue"];
[array removeObjectAtIndex:1];
```


NSDictionary

- **Common NSDictionary methods**

```
+ dictionaryWithObjectsAndKeys:(id)firstObject, ...;
-(unsigned)count;
-(id)objectForKey:(id)key;
```

- **nil returned if no object found for given key**

```
NSDictionary *colors =
 [NSDictionary dictionaryWithObjectsAndKeys:@"Red", @"Color 1",
   @"Green", @"Color 2", @"Blue", @"Color 3", nil];

NSString *firstColor = [colors objectForKey:@"Color 1"];

if ([colors objectForKey:@"Color 8"]) {
  // won't make it here
}
```

NSMutableDictionary

- NSMutableDictionary subclasses NSDictionary
- Common NSMutableDictionary methods

```
+ (NSMutableDictionary *)dictionary;
- (void)setObject:(id)object forKey:(id) key;
- (void)removeObjectForKey:(id)key;
- (void) removeAllObjects;
```

```
NSMutableDictionary *colors = [NSMutableDictionary dictionary];
[colors setObject:@"Orange" forKey:@"HighlightColor"];
```


NSSet

- Unordered collection of distinct objects
- Common NSSet methods

```
+ setWithObjects:(id)firstObj, ...; // nil terminated
- (unsigned)count;
- (BOOL)containsObject:(id)object;
```

NSMutableSet

- NSMutableSet subclasses NSSet
- Common NSMutableSet methods

```
+ (NSMutableSet *)set;
- (void)addObject:(id)object;
- (void)removeObject:(id)object;
- (void)removeAllObjects;
- (void)intersectSet:(NSSet *)otherSet;
- (void)minusSet:(NSSet *)otherSet;
```


Enumeration

- Consistent way of enumerating over objects in collections
- Use with NSArray, NSDictionary, NSSet, etc.

```
NSArray *array = ... ; // assume an array of People objects

// old school
 Person *person;
int count = [array count];
for (i = 0; i < count; i++) {
  person = [array objectAtIndex:i];
  NSLog([person description]);
}

// new school
for (Person *person in array) {
  NSLog([person description]);
}
```

Other Classes

- **NSData / NSMutableData**
  - Arbitrary sets of bytes
- **NSDate / NSCalendarDate**
  - Times and dates

Methods and Selectors

Terminology

- **Message expression**

  [receiver method: argument]

- **Message**

  [receiver method: argument]

- **Selector**

  [receiver method: argument]

- **Method**

  The code selected by a message

Methods, Messages, Selectors

- **Method**
  - Behavior associated with an object

```
-(NSString *)name
{
    // Implementation
}
-(void)setName:(NSString *)name
{
   //Implementation
}
```

Methods, Selectors, Messages

- **Selector**
  - Name for referring to a method
  - Includes colons to indicate arguments
  - Doesn't actually include arguments or indicate types

```
SEL mySelector = @selector(name);

SEL anotherSelector =@selector(setName:);

SEL lastSelector = @selector(doStuff:withThing:andThing:);
```


Methods, Messages, Selectors

- **Message**
  - The act of performing a selector on an object
  - With arguments, if necessary

```
NSString *name = [myPerson name];

[myPerson setName:@"New Name"];
```

Selectors identify methods by name

- **A selector has type SEL**
  SEL action = [button action];
  [button setAction:@selector(start:)];
- **Conceptually similar to function pointer**
- **Selectors include the name and all colons, for example:**
  (void)setName:(NSString *)name age:(int)age;
- **Would have a selector:**
  SEL sel = @selector(setName:age:);

Working with selectors

•You can determine if an object responds to a given selector

```
id obj;

SEL sel = @selector(start:);

if ([obj respondsToSelector:sel]) {
        [obj performSelector:sel withObject:self];
  //equivalent to [obj start:self];
  //For multiple arguments use … withObject:  withObject:
}
```

•This sort of introspection and dynamic messaging underlies many Cocoa design patterns

```
-(void)setTarget:(id)target;
-(void)setAction:(SEL)action;
```

More Info on Selectors

- **Selectors are unique identifiers that replace the name of methods when compiled**
- **Compiler writes each method name into a table and associates it with this unique id (the selector)**
- **The compiler assigns all method names a unique selector or SEL (the selector type)**
  - Every "method name" whether it is a part of your class or another class has an entry in that table with a unique selector value
- **More information at:**

  https://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjectiveC/Articles/ocSelectors.html

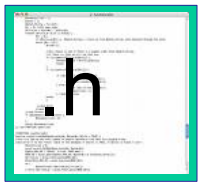
Custom Classes

Design Phase

- **Create a class**
  - Person
- **Determine the superclass**
  - NSObject (in this case)
- **What properties should it have?**
  - Name, age, whether they can vote
- **What actions can it perform?**
  - Cast a ballot

Defining a class

A public header and a private implementation



Header file

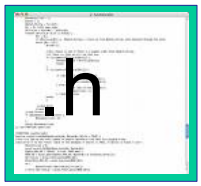
Implementation file

Class interface declared in header file

```
#import <Foundation/Foundation.h>
@interface Person : NSObject
{
    // instance variables
    NSString *name;
    int age;
}

// method declarations
- (NSString *)name;
- (void)setName:(NSString *)value;

- (int)age;
- (void)setAge:(int)age;
- (BOOL)canLegallyVote;
- (void)castBallot;

@end
```


Defining a class

A public header and a private implementation



Header file

Implementation file

Implementing custom class

- Implement setter/getter methods
- Implement action methods

Class Implementation

```
#import "Person.h“

@implementation Person

-(int)age {
  return age;
}

-(void)setAge:(int)value {
   age = value;
}
//... and other methods

@end
```

Calling your own methods

```
#import "Person.h“

@implementation Person

-(BOOL)canLegallyVote {
   return ([self age] >= 18);
}

-(void)castBallot {
    if ([self canLegallyVote]) {
       // do voting stuff
    } else {
       NSLog (@“I’m not allowed to vote!”);
    }
} @end
```

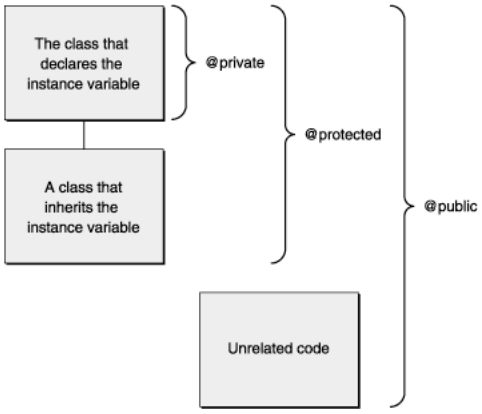

Superclass methods

- As we just saw, objects have an implicit variable named “self”
  - Like “this” in Java and C++
- Can also invoke superclass methods using “super”

```
-(void)doSomething {
   // Call superclass implementation first
   [super doSomething];

  // Then do our custom behavior
  int foo = bar;
  // …
}
```

Public and Private Instance Variables

- By default all variables are protected



Instance Variable Protection

```
@interface Worker : NSObject
{
 char *name;  //actually protected

 @private
   int age;
   char *evaluation;

 @protected
   id job;
   float wage;

 @public
  id boss;

}
```

Objective C methods are all public (a way around this…)
http://iphonedevelopertips.com/objective-c/private-methods.html

Private Methods

- Private methods in Objective-C are declared in your .m file

```
@interface Worker ()

  - (void) myPrivateMethod1;
  - (int) anotherPrivateMethod:  (NSString *) myStringArgument

@end

@implementation Worker

  -(void) myPrivateMethod1 {
    //Do something here
   }

   -(int) anotherPrivateMethod: (NSString *) myStringArgument {
     return 0;
   }

   - (void)  somePublicMethod {

   }

@end
```


Object Lifecycle

Object Lifecycle

- Creating objects
- Memory management
- Destroying objects

Object Creation

- Two step process
  - allocate memory to store the object
  - initialize object state

+alloc

- Class method that knows how much memory is needed

-init

- Instance method to set initial values, perform other setup

Create = Allocate + Initialize

```
Person *student = nil;

student = [[Person alloc] init];
```

Or

```
Person *student = nil;
student = [Person alloc];
student = [student init];
```


Implementing your own -init method

```
#import "Person.h“

@implementation Person

-(id)init {
  // allow superclass to initialize its state first
   self = [super init];
  if (self != nil) {
            age = 0;
            name = @“Bob”;

     // do other initialization...
  }
  return self;
}

@end
```

Multiple init methods

- Classes may define multiple init methods

```
- (id)init;
- (id)initWithName:(NSString *)name;
- (id)initWithName:(NSString *)name age:(int)age;
```

- Less specific ones typically call more specific with default values
  - Designated Initializers

```
- (id)init { return [self initWithName:@"Bob"];
 }
- (id)initWithName:(NSString *)name {
  return [self initWithName:name age:0];
 }
```


Finishing Up With an Object

```
Person *person = nil;

person = [[Person alloc] init];

[person setName:@"Alan Cannistraro"];
[person setAge:29];
[person setWishfulThinking:YES];

[person castBallot];

// What do we do with person when we're done?
```

Two flavors of Memory Management

- **Automatic Reference Counting (ARC)**
  - Full support starting in iOS 5
- **Manual Reference Counting**
  - Original Objective C design
- **Choose one or the other**
  - Do not attempt to use both in the same .m file

Why learn both methods?

- **Many of the tutorials and examples on the web were created pre-ARC**
- **A solid understanding of manual reference counting makes ARC easier to understand**
- **Xcode can run into problems with migrating existing code to use ARC**

(Manual) Memory Management

	Allocation	Destruction
C	malloc	free
Objective-C	alloc	dealloc

- **Calls must be balanced**
  - Otherwise your program may leak or crash
- **However, you'll never call -dealloc directly**
  - One exception, we'll see in a bit...

(Manual) Reference Counting

- **Every object has a retain count**
  - Defined on NSObject
  - As long as retain count is > 0, object is alive and valid

- **+alloc and -copy create objects with retain count == 1**
- **-retain increments retain count**
- **-release decrements retain count**

- **When retain count reaches 0, object is destroyed**
- **-dealloc method invoked automatically**
  - One-way street, once you're in -dealloc there's no turning back

Balanced Calls

```
Person *person = nil;
person = [[Person alloc] init];

[person setName:@“John Smith”];
[person setAge:29];
[person setWishfulThinking:YES];

[person castBallot];

// When we’re done with person, release it
[person release];  // person will be destroyed here
```


Reference counting in action

```
Person *person = [[Person alloc] init];
```

- Retain count begins at 1 with +alloc

```
[person retain];
```

- Retain count increases to 2 with –retain

```
[person release];
```

- Retain count decreases to 1 with –release

```
[person release];
```

- Retain count decreases to 0, -dealloc automatically called

Messaging deallocated objects

```
Person *person = [[Person alloc] init];
// ...
[person release]; // Object is deallocated

[person doSomething]; // Crash!
```


Messaging deallocated objects

```
Person *person = [[Person alloc] init];
// ...
[person release]; // Object is deallocated

person=nil;

[person doSomething]; // No effect
```

Implementing a -dealloc method

```
#import "Person.h“

@implementation Person

- (void)dealloc {
  // Do any cleanup that’s necessary
  // ...

  // when we’re done, call super to clean us up
  [super dealloc];
}
@end
```


Object Lifecycle Recap

- Objects begin with a retain count of 1
- Increase and decrease with -retain and –release
- When retain count reaches 0, object deallocated automatically
- You *never* call dealloc explicitly in your code
  - Exception is calling -[super dealloc]
  - You only deal with alloc, copy, retain, release

Object Ownership

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

@Interface Person : NSObject
{
  // instance variables
  NSString *name;    //Person class "owns" the name
  int age;
}

// method declarations
-(NSString *)name;
-(void)setName:(NSString *)value;

-(int)age;
-(void)setAge:(int)age;

-(BOOL)canLegallyVote;
-(void)castBallot;

@end
```


Object Ownership

```
#import "Person.h"
@implementation Person

- (NSString *)name {
  return name;
}

- (void)setName:(NSString *)newName {
   if (name != newName) {
      [name release];
      name = [newName retain];
      // name's retain count has been bumped up by 1
   }
}

@end
```

Object Ownership

```
#import "Person.h“
@implementation Person

- (NSString *)name {
  return name;
}

- (void)setName:(NSString *)newName {
   if (name != newName) {
      [name release];
      name = [newName copy];
      // name has a retain count of 1, we own it
   }
}

@end
```


Releasing Instance Variables

```
#import "Person.h“
@implementation Person

- (void)dealloc{
    //Do any cleanup that’s necessary
    [name release];

    // when we’re done, call super to clean us up
    [super dealloc];
}

@end
```

Autorelease

Returning a newly created object

```
-(NSString *)fullName {
  NSString *result;

  result = [[NSString alloc] initWithFormat:@"%@ %@",
    firstName, lastName];

  return result;
}
```

- Wrong: result is leaked!

Returning a newly created object

```
-(NSString *)fullName {
  NSString *result;

  result = [[NSString alloc] initWithFormat:@"%@ %@",
    firstName, lastName];

  [result release];
  return result;
}
```

- Wrong: result is released too early!
- Uncertain what method returns

Returning a newly created object

```
-(NSString *)fullName {
  NSString *result;

  result = [[NSString alloc] initWithFormat:@"%@ %@",
    firstName, lastName];

  [result autorelease];
  return result;
}
```

- Just right: result is released, but not right away!
- Caller gets valid object and could retain if needed

Autoreleasing Objects

- Calling -autorelease flags an object to be sent release at some point in the future
- Let's you fulfill your retain/release obligations while allowing an object some additional time to live
- Makes it much more convenient to manage memory
- Very useful in methods which return a newly created object

Method Names & Autorelease

- Methods whose names includes **alloc** or **copy** return a retained object that the caller needs to release

```
NSMutableString *string = [[NSMutableString alloc] init];

// We are responsible for calling -release or –autorelease
[string autorelease];
```

- All other methods return autoreleased objects

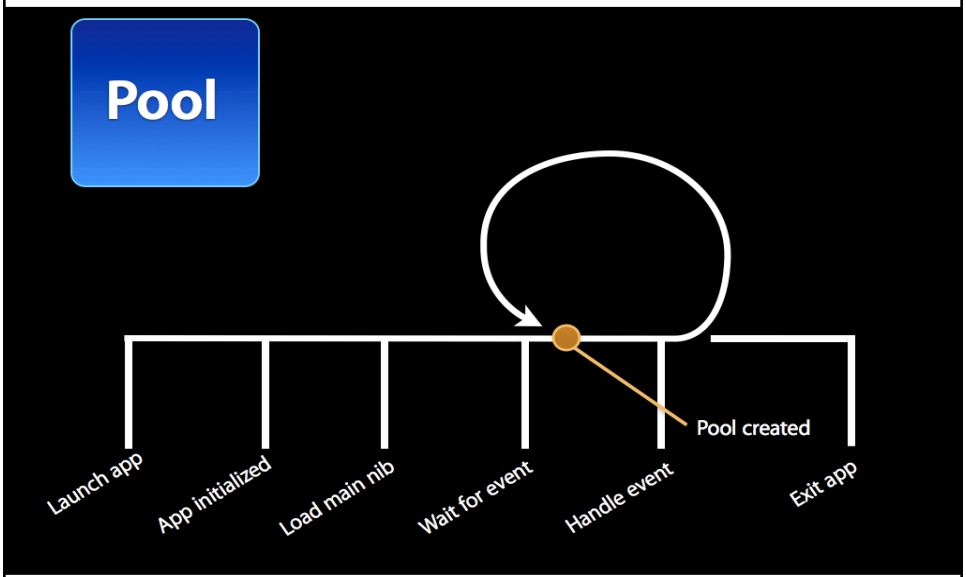
```
NSMutableString *string = [NSMutableString string];

// The method name doesn't indicate that we need to release it
// So don't- we're cool!
```

- This is a convention
  - follow it in methods you define

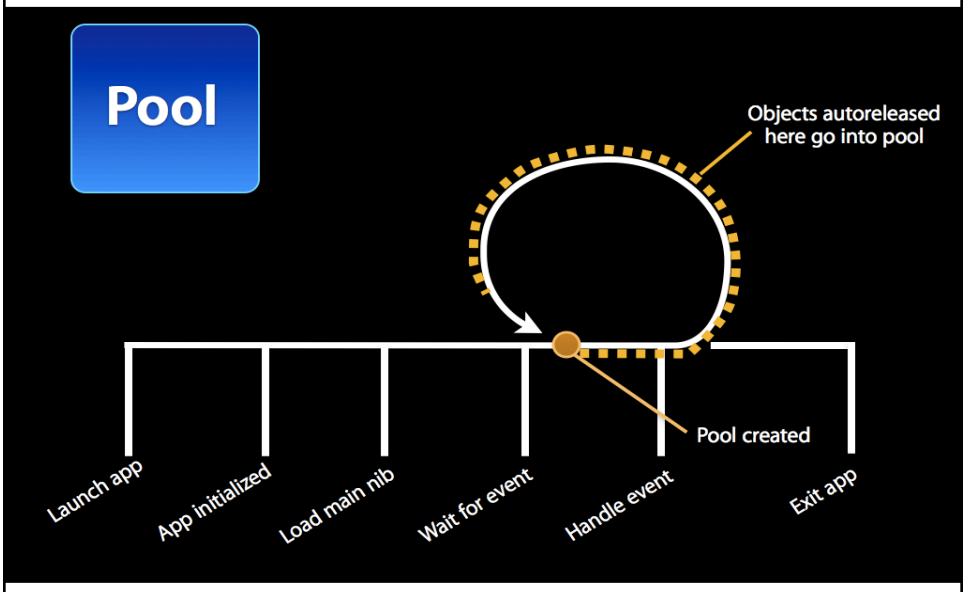
How does -autorelease work?

- **Object is added to current autorelease pool**
- **Autorelease pools track objects scheduled to be released**
  - When the pool itself is released, it sends -release to all its objects
- **UIKit automatically wraps a pool around every event dispatch**

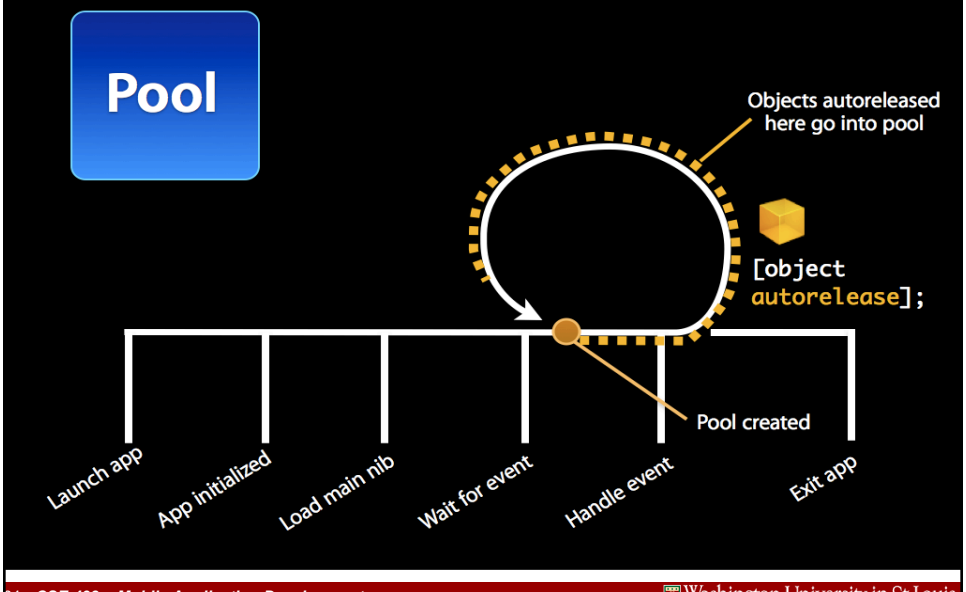
Autorelease Pools (from cs193p slides)



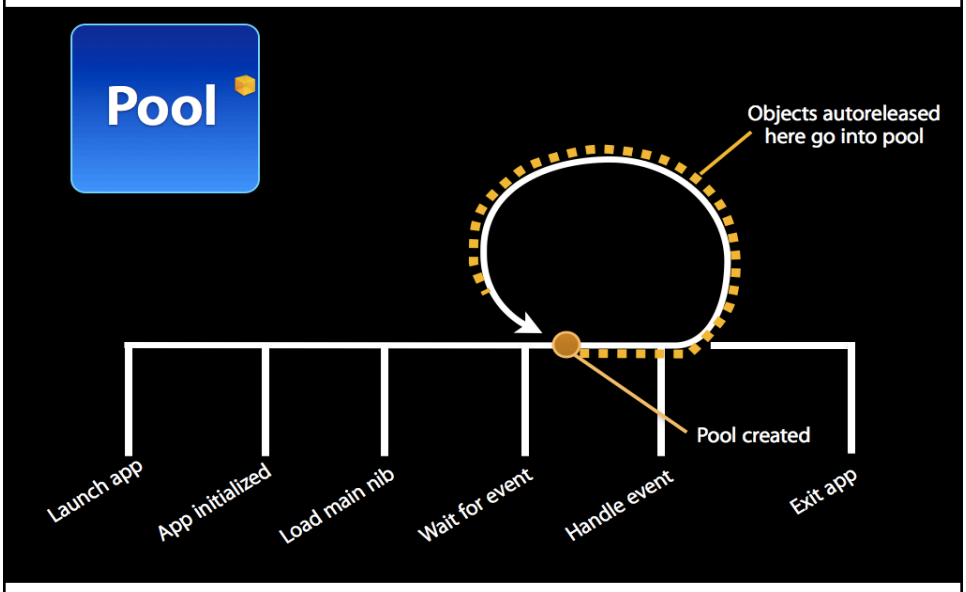
Autorelease Pools (from cs193p slides)



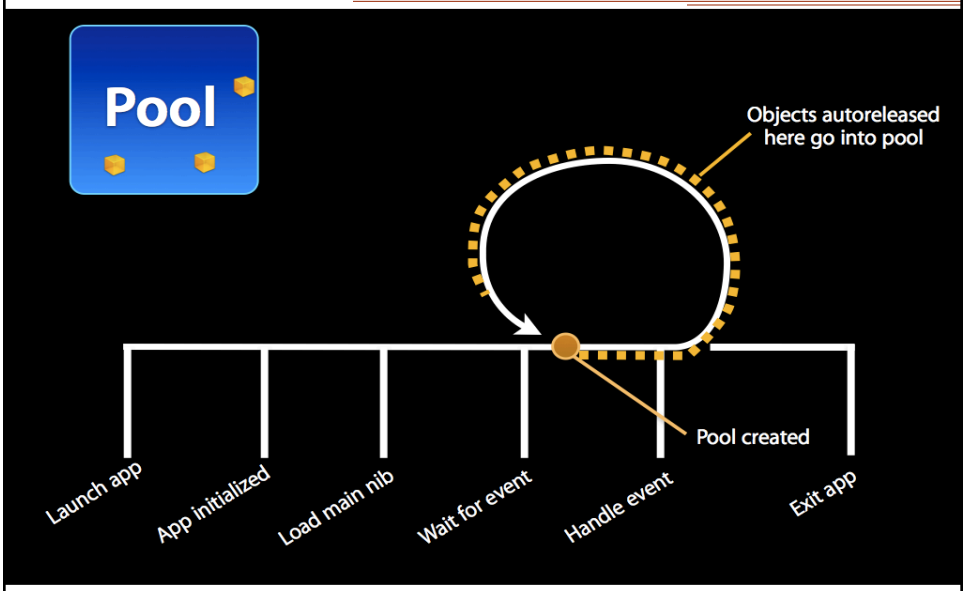
Autorelease Pools (from cs193p slides)



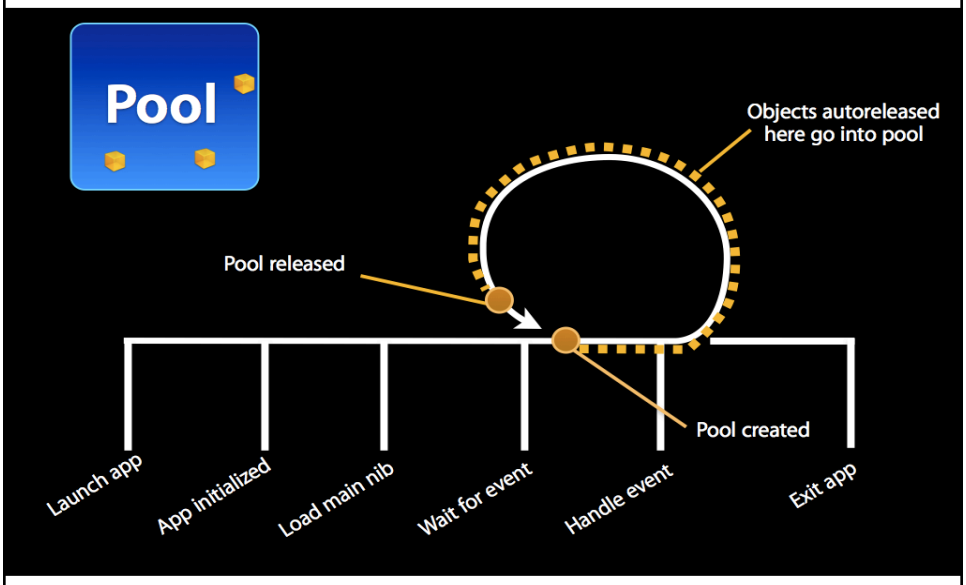
Autorelease Pools (from cs193p slides)



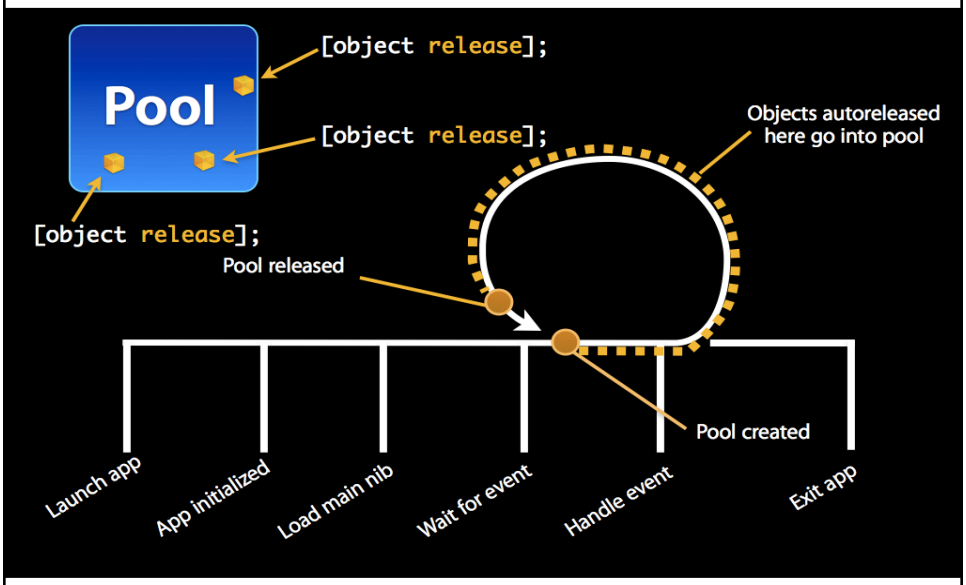
Autorelease Pools (from cs193p slides)



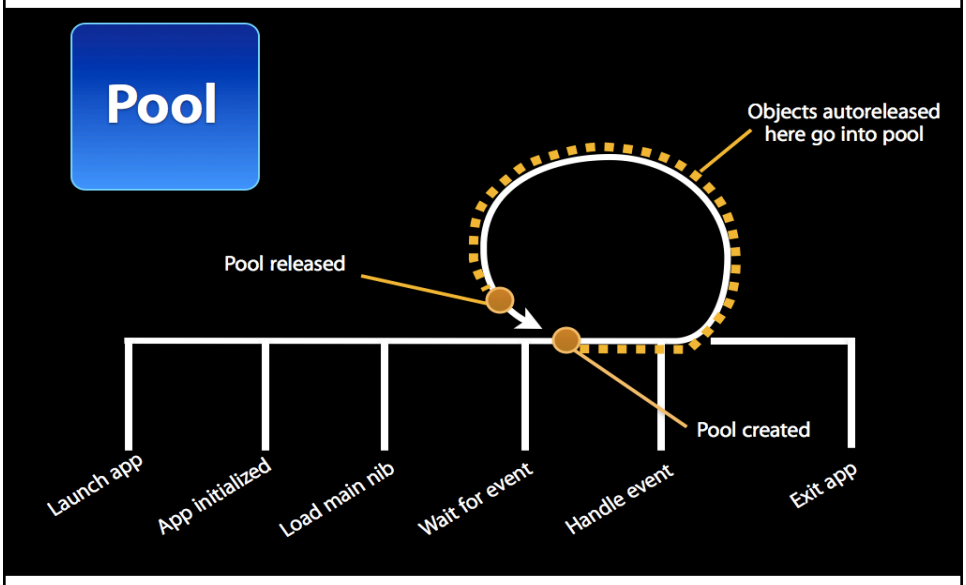
Autorelease Pools (from cs193p slides)



Autorelease Pools (from cs193p slides)



Autorelease Pools (from cs193p slides)



Hanging Onto an Autoreleased Object

- **Many methods return autoreleased objects**
  - Remember the naming conventions...
  - They're hanging out in the pool and will get released later
- **If you need to hold onto those objects you need to retain them**
  - Bumps up the retain count before the release happens

```
name = [NSMutableString string];

// We want to name to remain valid!
[name retain];
// ...
// Eventually, we'll release it (maybe in our -dealloc?)
[name release];
```

Side Note: Garbage Collection

- Autorelease is not garbage collection
- Objective-C on iPhone OS (iOS) does not have garbage collection

Automatic Reference Counting (ARC)

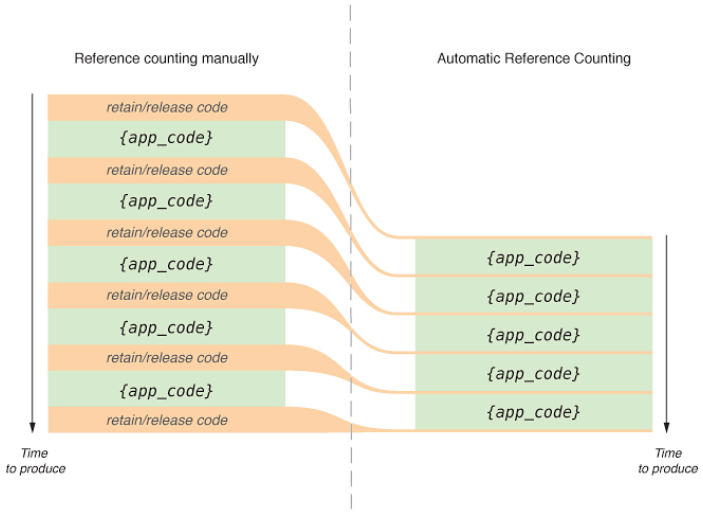
Automatic Reference Counting (ARC)

- **The new and "improved" way to manage memory**
  - All objects are either strong or weak
- **Strong**
  - Keep me around until I no longer need this memory
- **Weak**
  - Keep me around as long as some other object needs this memory

Automatic Reference Counting

- **By default all objects allocated when using ARC are strong**
  - NSNumber *myNumber = [NSNumber alloc] init];
- **Weak references are often used when pointing to objects on a storyboard**
  - UIButton, UILabel, UIImage
    - These objects are already instantiated when the storyboard loads
    - We just want a pointer to them while they are alive

Automatic Retain/Release



Properties

- **Provide access to object attributes**
- **Shortcut to implementing getter/setter methods**
- **Also allow you to specify:**
  - read-only versus read-write access
  - memory management policy

Defining Properties

```
#import<Foundation/Foundation.h>
@interface Person : NSObject
{
// instance variables
   NSString *name;
   int age;
}

// method declarations
-(NSString *)name;
-(void)setName:(NSString *)value;
-(int)age;
-(void)setAge:(int)age;
-(BOOL)canLegallyVote;

-(void)castBallot;
@end
```


Defining Properties

```
#import<Foundation/Foundation.h>
@interface Person : NSObject
{
// instance variables
   NSString *name;
   int age;
}

// method declarations
-(NSString *)name;
-(void)setName:(NSString *)value;
-(int)age;
-(void)setAge:(int)age;
-(BOOL)canLegallyVote;

-(void)castBallot;
@end
```

Defining Properties

```
#import<Foundation/Foundation.h>
@interface Person : NSObject
{
// instance variables
   NSString *name;
   int age;
}

// method declarations
-(NSString *)name;
-(void)setName:(NSString *)value;
-(int)age;
-(void)setAge:(int)age;
-(BOOL)canLegallyVote;

-(void)castBallot;
@end
```


Defining Properties

```
#import<Foundation/Foundation.h>
@interface Person : NSObject
{
// instance variables
   NSString *name;
   int age;
}

// property declarations
@property int age;
@property (copy) NSString  *name;
@property (readonly) BOOL canLegallyVote;

-(void)castBallot;
@end
```

Synthesizing Properties

```
@implementation Person

-(int)age {
 return age;
}

-(void)setAge:(int)value {
  age = value;
}

-(NSString *)name {
  return name;
}

-(void)setName:(NSString *)value {
   if (value != name) {
      [value release];
      name = [value copy];
   }

- (BOOL)canLegallyVote { …
```


Synthesizing Properties

```
@implementation Person

-(int)age {
 return age;
}

-(void)setAge:(int)value {
  age = value;
}

-(NSString *)name {
  return name;
}

-(void)setName:(NSString *)value {
   if (value != name) {
      [value release];
      name = [value copy];
   }

- (BOOL)canLegallyVote { …
```

Synthesizing Properties

```
@implementation Person

-(int)age {
  return age;
 }

-(void)setAge:(int)value {
   age = value;
 }

-(NSString *)name {
   return name;
 }

-(void)setName:(NSString *)value {
    if (value != name) {
        [value release];
        name = [value copy];
    }

- (BOOL)canLegallyVote { ...
```


Synthesizing Properties

```
@implementation Person

@synthesize age;
@synthesize name;
-    (BOOL)canLegallyVote {
   return (age > 17);
}

@end
```

iOS Property Attributes

- Use strong and weak instead of retain and assign

```
@property (retain) NSString *name; // retain called
@property (strong) NSString *name;// new way


@property (assign) NSString *name; // pointer assignment
@property (weak) NSString *name; //new way
```


Property Names vs. Instance Variables

- Property name can be different than instance variable

```
@interface Person : NSObject {
  int numberOfYearsOld;
}

@property int age;

@end


@implementation Person

@synthesize age = numberOfYearsOld;

@end
```

Properties

- Mix and match synthesized and implemented properties

```
@implementation Person

@synthesize age;
@synthesize name;

(void)setAge:(int)value {
    age = value;
}
@end
```

- Setter method explicitly implemented
- Getter method still synthesized

Properties In Practice

- Newer APIs use @property
- Older APIs use getter/setter methods
- Properties used heavily throughout UIKit APIs
  - Not so much with Foundation APIs
- You can use either approach
  - Properties mean writing less code, but "magic" can sometimes be non-obvious

Further Reading

- **Objective-C 2.0 Programming Language**
  - "Defining a Class"
  - "Declared Properties"
- **Memory Management Programming Guide for Cocoa**