Announcement

- **Start Final Project Pitches on Wednesday**
  - Presentation slides dues by Tuesday at 11:59 PM
  - Email slides to cse438ta@gmail.com

Final Project Proposal Presentations and Updates

- **We will start the final project presentations on Wednesday November 8th in class and continue them on Monday**
- **Each group will have 5 minutes to present their app**
- **Due to the large size of this course, some groups may not be able to present**
- **If someone from your group is not here to present your project you will receive a 0 that portion of the project**

Today's Topic - 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

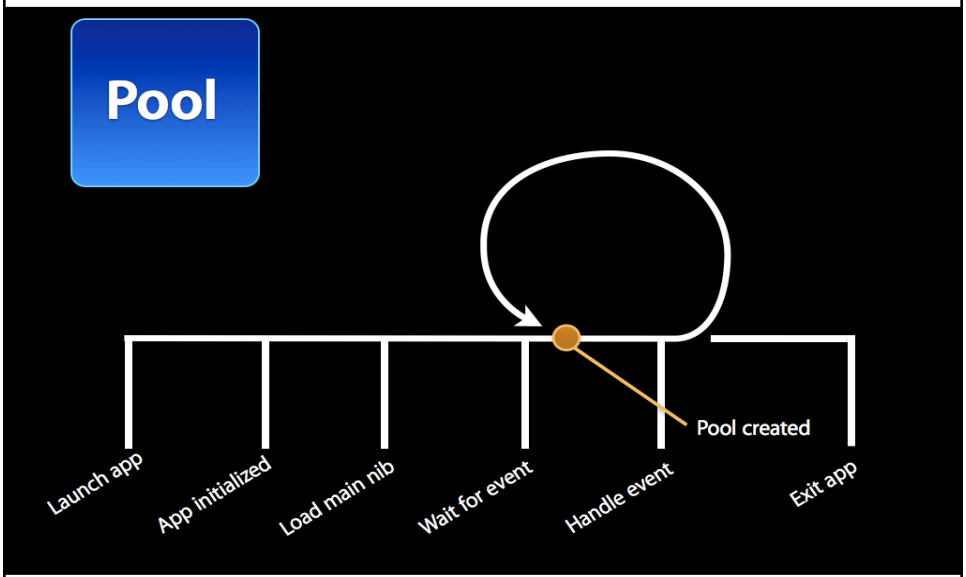
```
NSMutableString *string = [NSMutableString string];

// 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

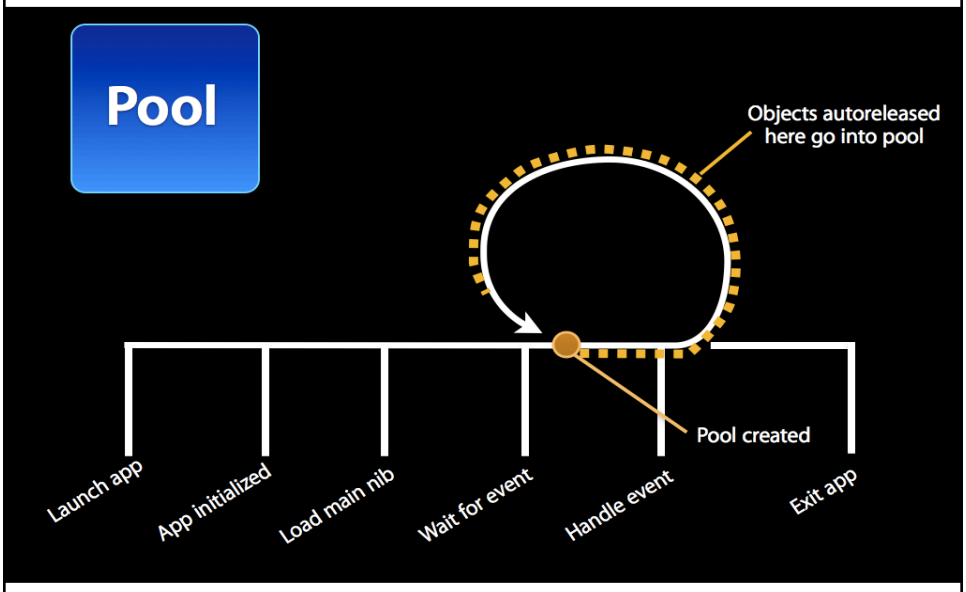
How does -autorelease work?

- **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**

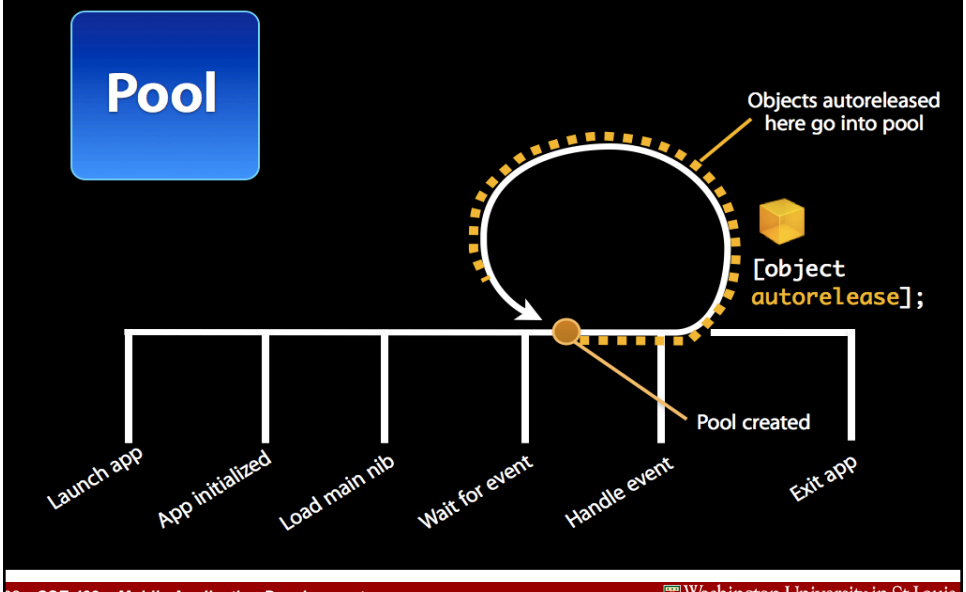
Autorelease Pools (from cs193p slides)



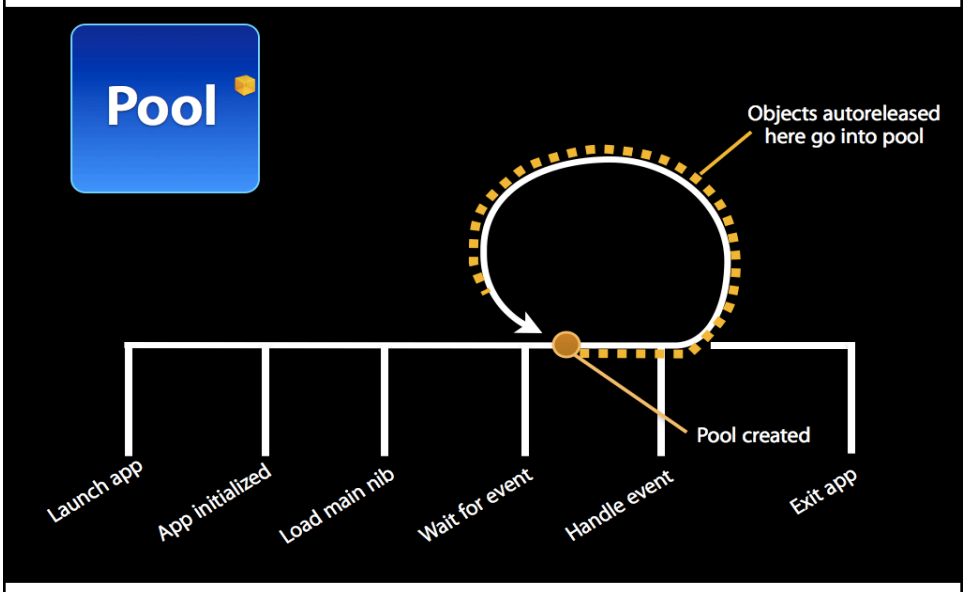
Autorelease Pools (from cs193p slides)



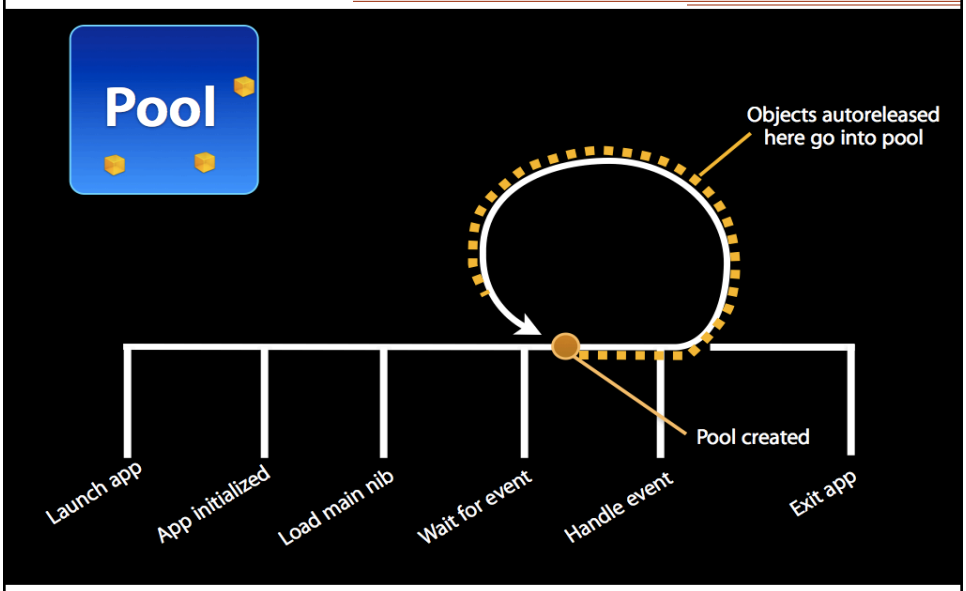
Autorelease Pools (from cs193p slides)



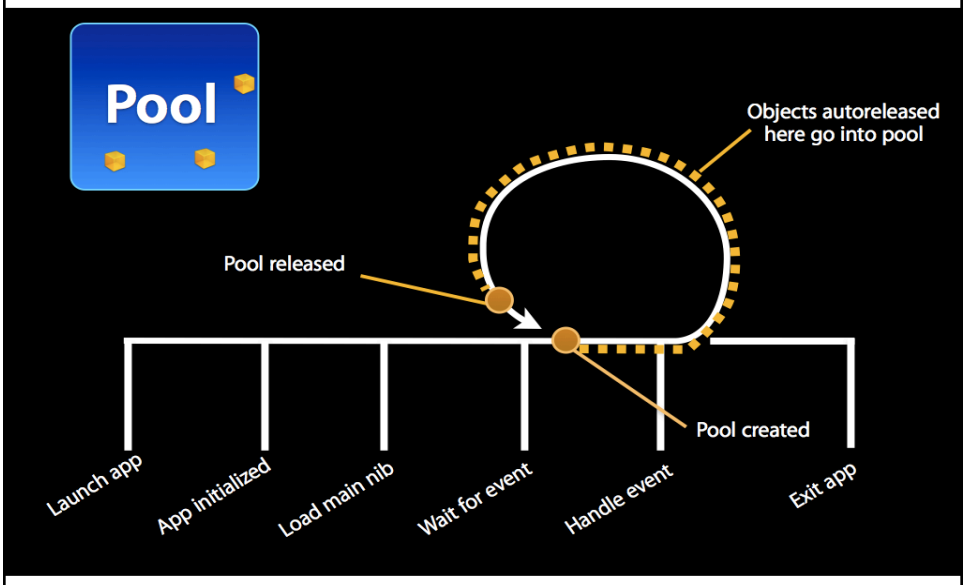
Autorelease Pools (from cs193p slides)



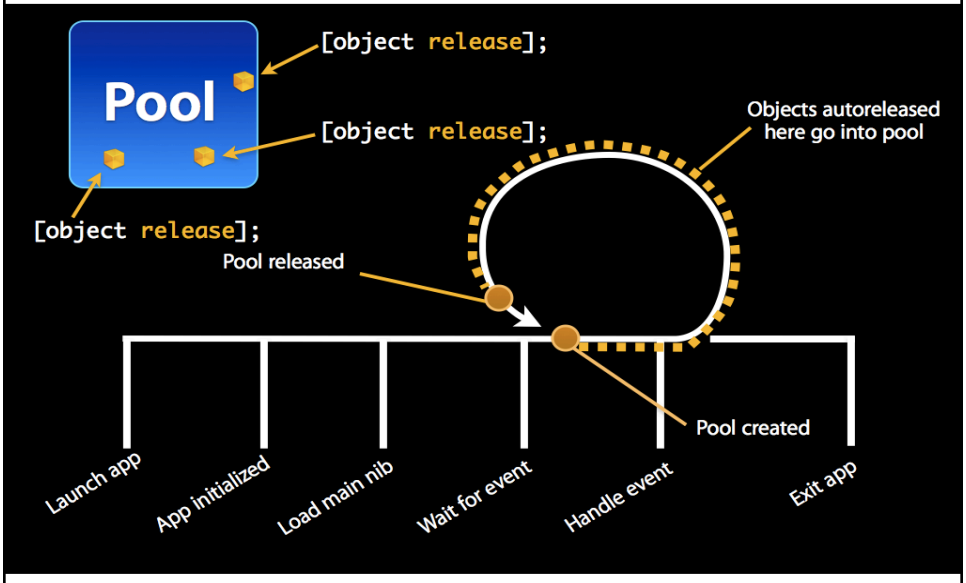
Autorelease Pools (from cs193p slides)



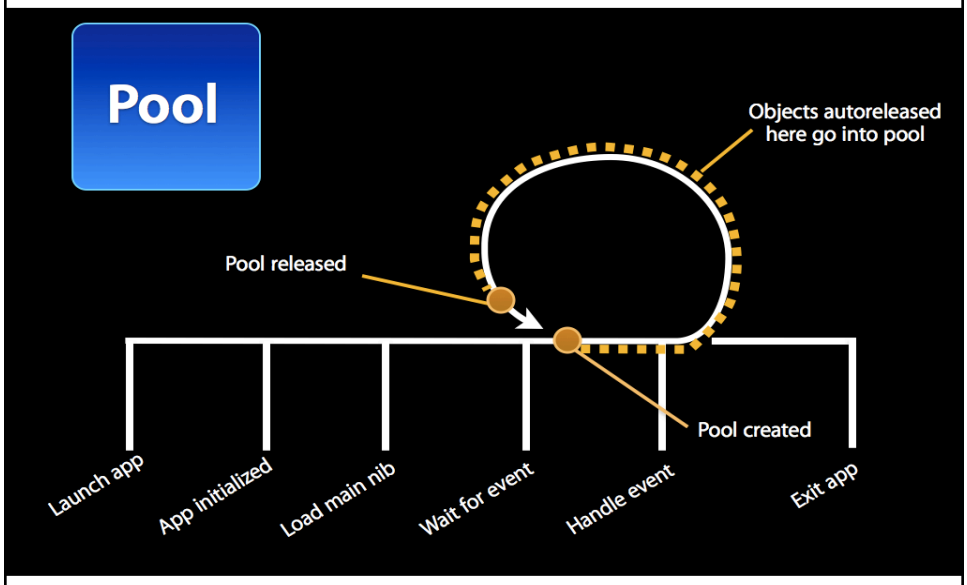
Autorelease Pools (from cs193p slides)



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)

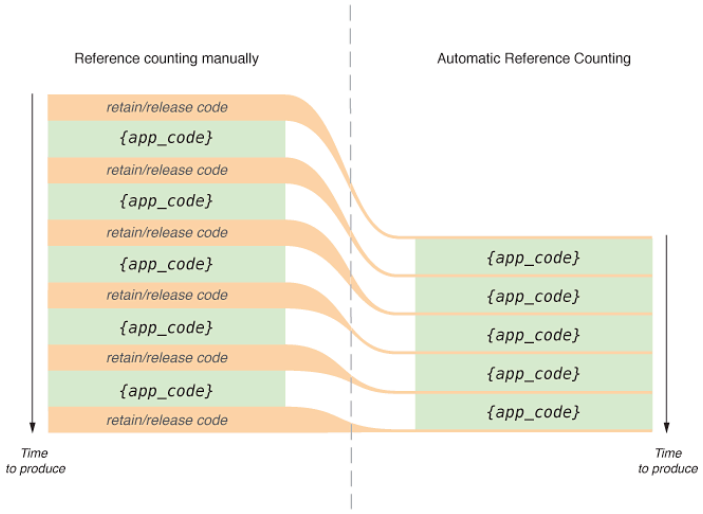
Automatic Reference Counting (ARC)

- **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**

Objective C and Swift