CS193P - Lecture 3

iPhone Application Development

Custom Classes
Object Lifecycle
Autorelease
Properties

Announcements

- Assignments 1A and 1B **due Wednesday 1/13 at 11:59 PM**
  - Enrolled Stanford students can email cs193p@cs.stanford.edu with any questions
  - Submit early! Instructions on the website...
    - **Delete the "build" directory manually, Xcode won't do it**

Announcements

- Assignments 2A and 2B **due Wednesday 1/20 at 11:59 PM**
  - 2A: Continuation of Foundation tool
    - Add custom class
    - Basic memory management
  - 2B: Beginning of first iPhone application
    - Topics to be covered on Thursday, 1/14
    - Assignment contains extensive walkthrough

Enrolled students & iTunes U

- Lectures have begun showing up on iTunes U
- Lead time is longer than last year
- **Come to class!!**
  - Lectures may not post in time for assignments

Office Hours

- Paul's office hours: Thursday 2-4, Gates B26B
- David's office hours: Mondays 4-6pm: Gates 360

Today's Topics

- Questions from Assignment 1A or 1B?
- Creating Custom Classes
- Object Lifecycle
- Autorelease
- Objective-C Properties

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

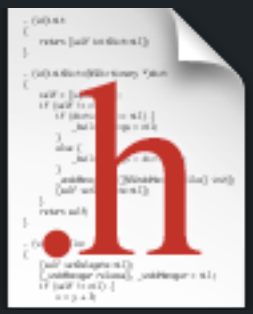
A public header and a private implementation

Header File

Implementation File

Defining a class

A public header and a private implementation



Header File

Implementation File

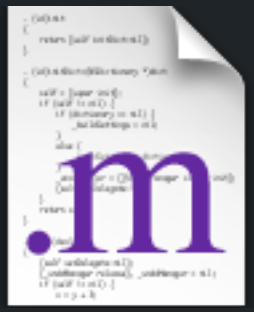
Class interface declared in header file

```
#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 {

}

- (void)castBallot {




}

@end
```

Calling your own methods

```
#import "Person.h"

@implementation Person

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

- (void)castBallot {




}

@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;
  // ...
}
```

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 *person = nil;

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

Implementing your own -init method

```
#import "Person.h"

@implementation Person

- (id)init {
  // allow superclass to initialize its state first
  if (self = [super init]) {
       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

```
- (id)init {
    return [self initWithName:@“No Name”];
}

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

Finishing Up With an Object

```
Person *person = nil;

person = [[Person alloc] init];

[person setName:@“Jimmy Jones”];
[person setAge:32];

[person castBallot];
[person doSomethingElse];
```

Finishing Up With an Object

```
Person *person = nil;

person = [[Person alloc] init];

[person setName:@“Jimmy Jones”];
[person setAge:32];

[person castBallot];
[person doSomethingElse];

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

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...

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:@“Jimmy Jones”];
[person setAge:32];

[person castBallot];
[person doSomethingElse];

// 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
```

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
```

Messaging deallocated objects

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

person = nil;
```

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









@end
```

Object Ownership

```
#import "Person.h"

@implementation Person

- (NSString *)name {
    return name;
}
- (void)setName:(NSString *)newName {




}

@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 {




}

@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 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**!
Method returns bogus value

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, copy, or new** 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

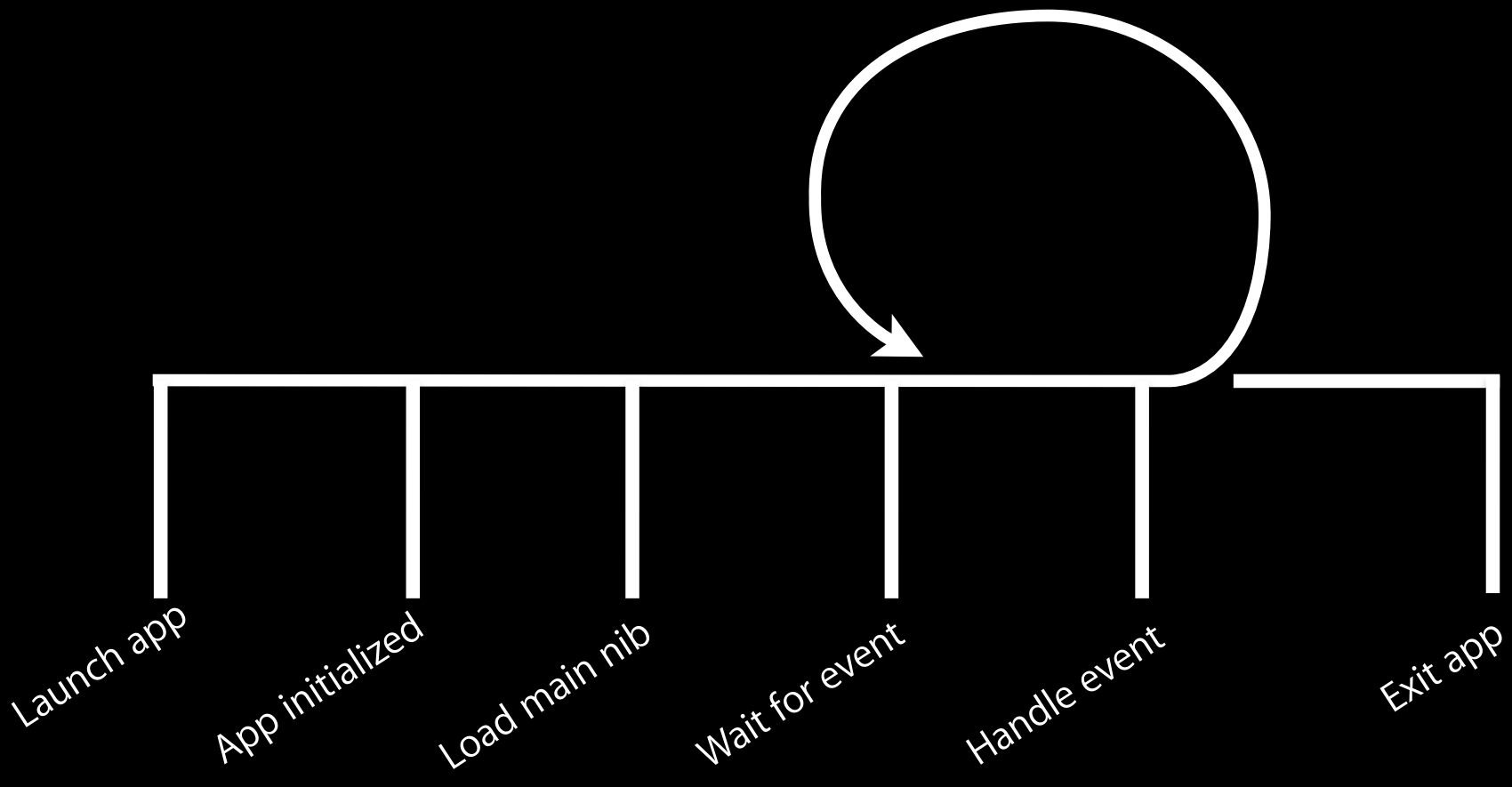
```
NSMutableString *string = [NSMutableString string];
// The method name doesn't indicate that we need to release it
// So don't- we're cool!
```

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

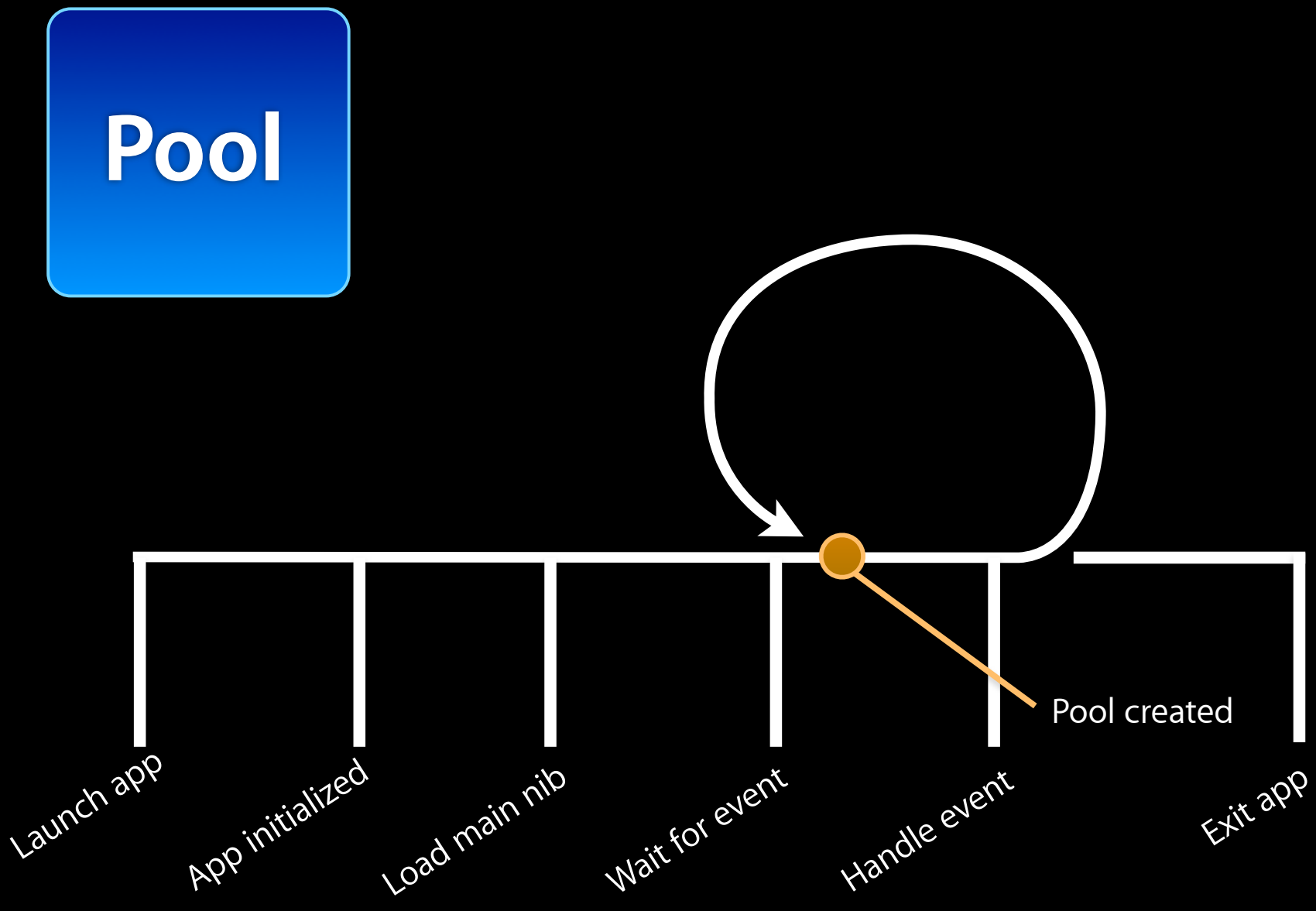
How does -autorelease work?

- 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

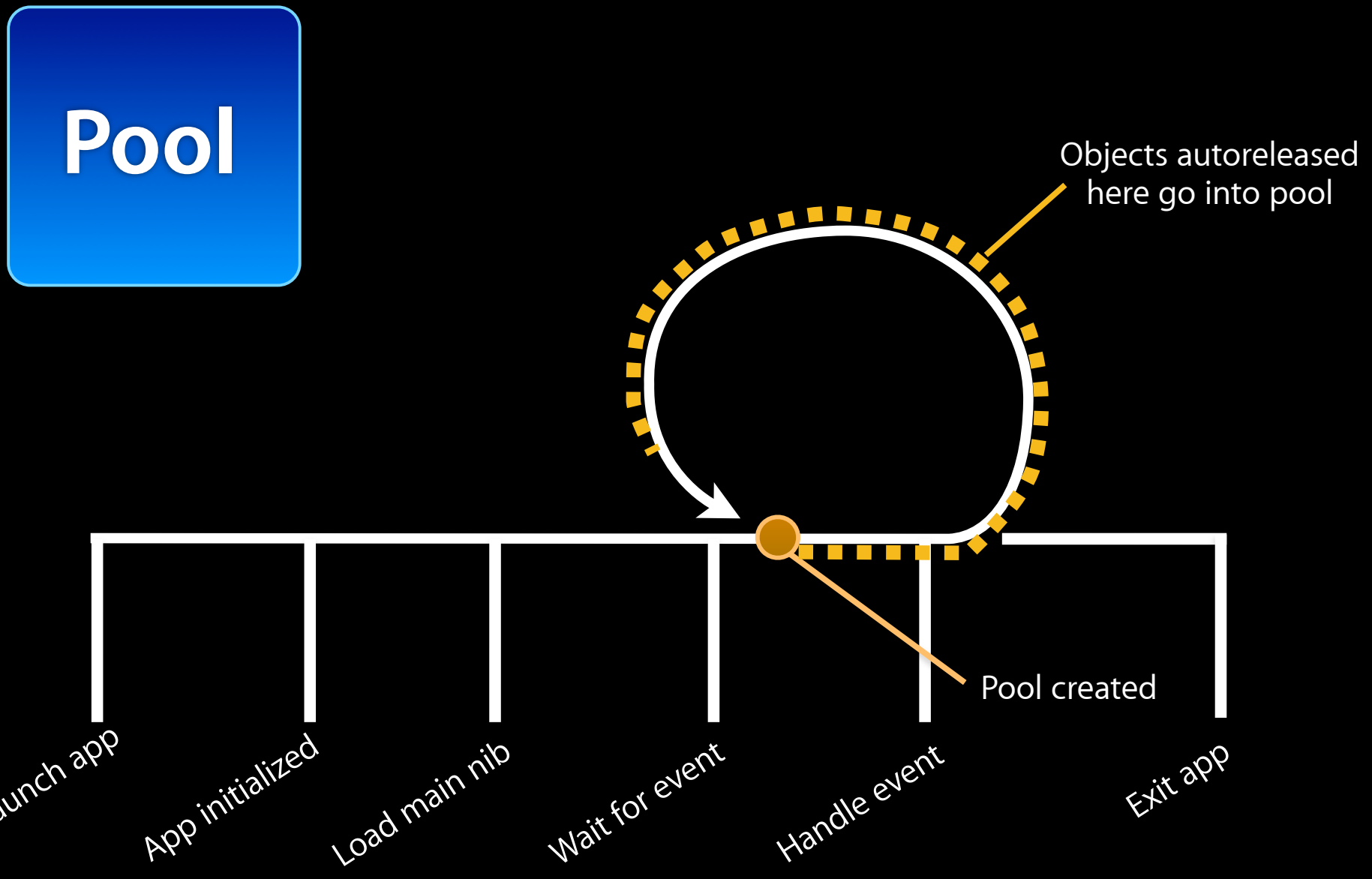
Autorelease Pools (in pictures)



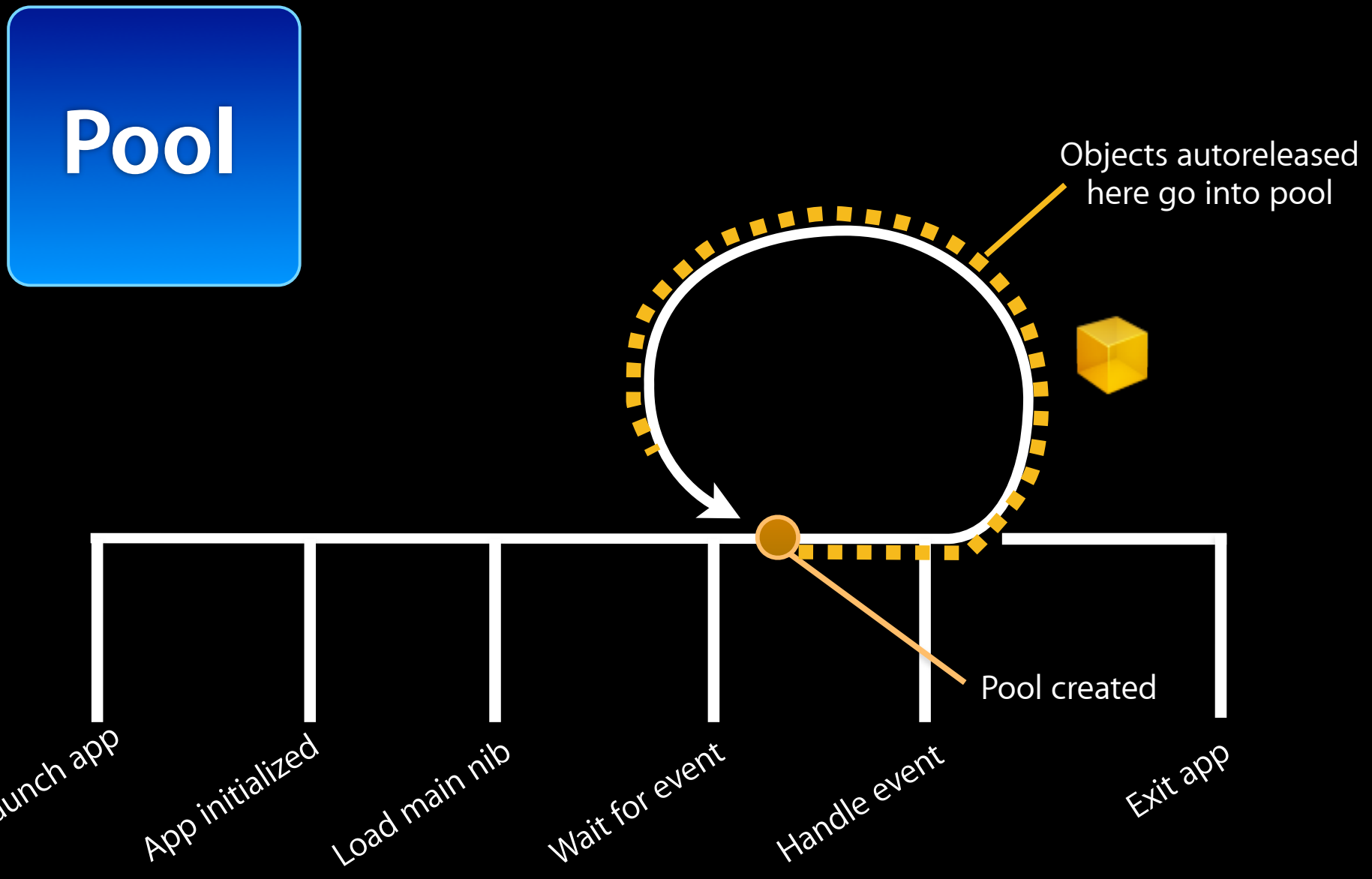
Autorelease Pools (in pictures)



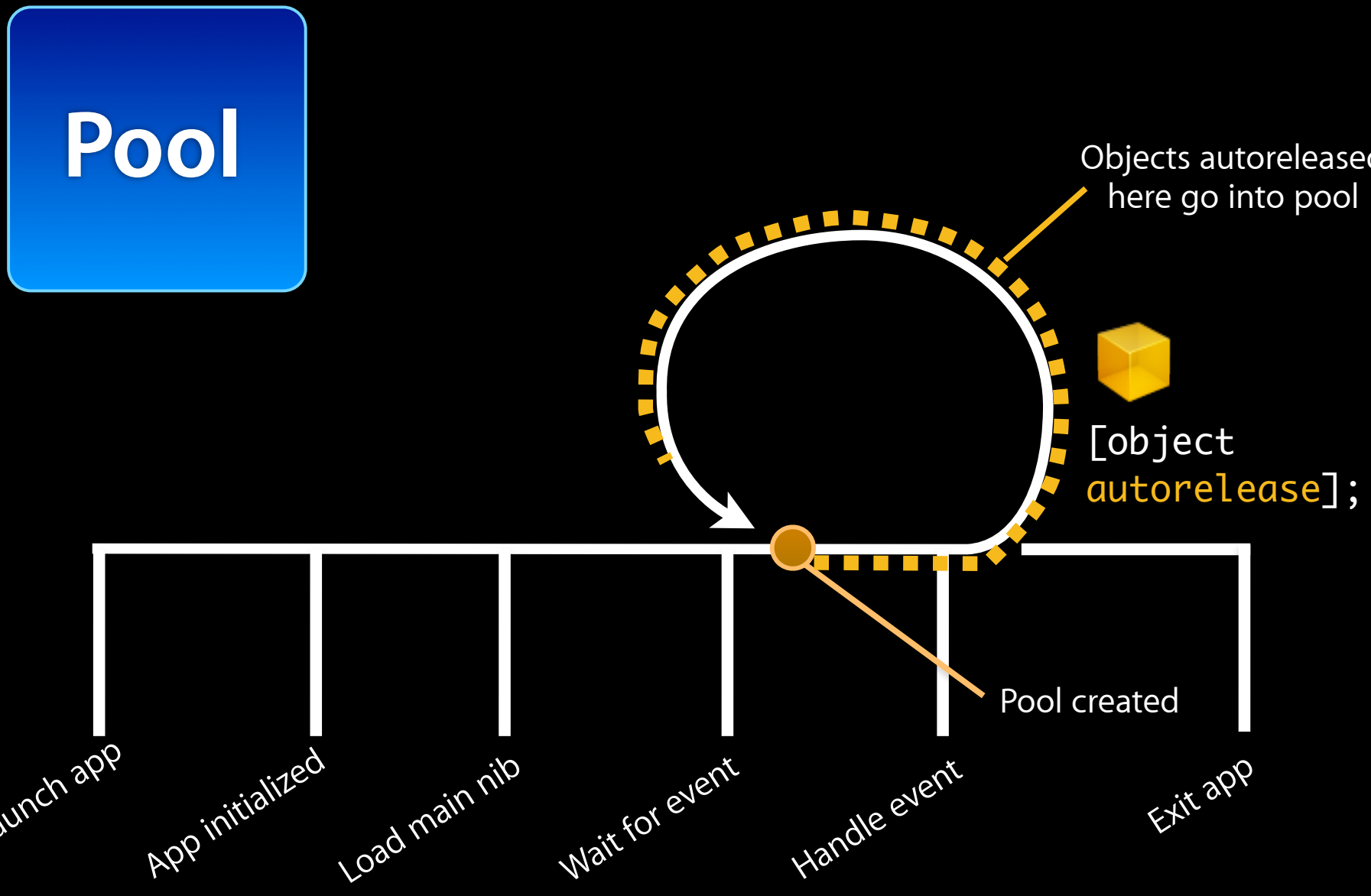
Autorelease Pools (in pictures)



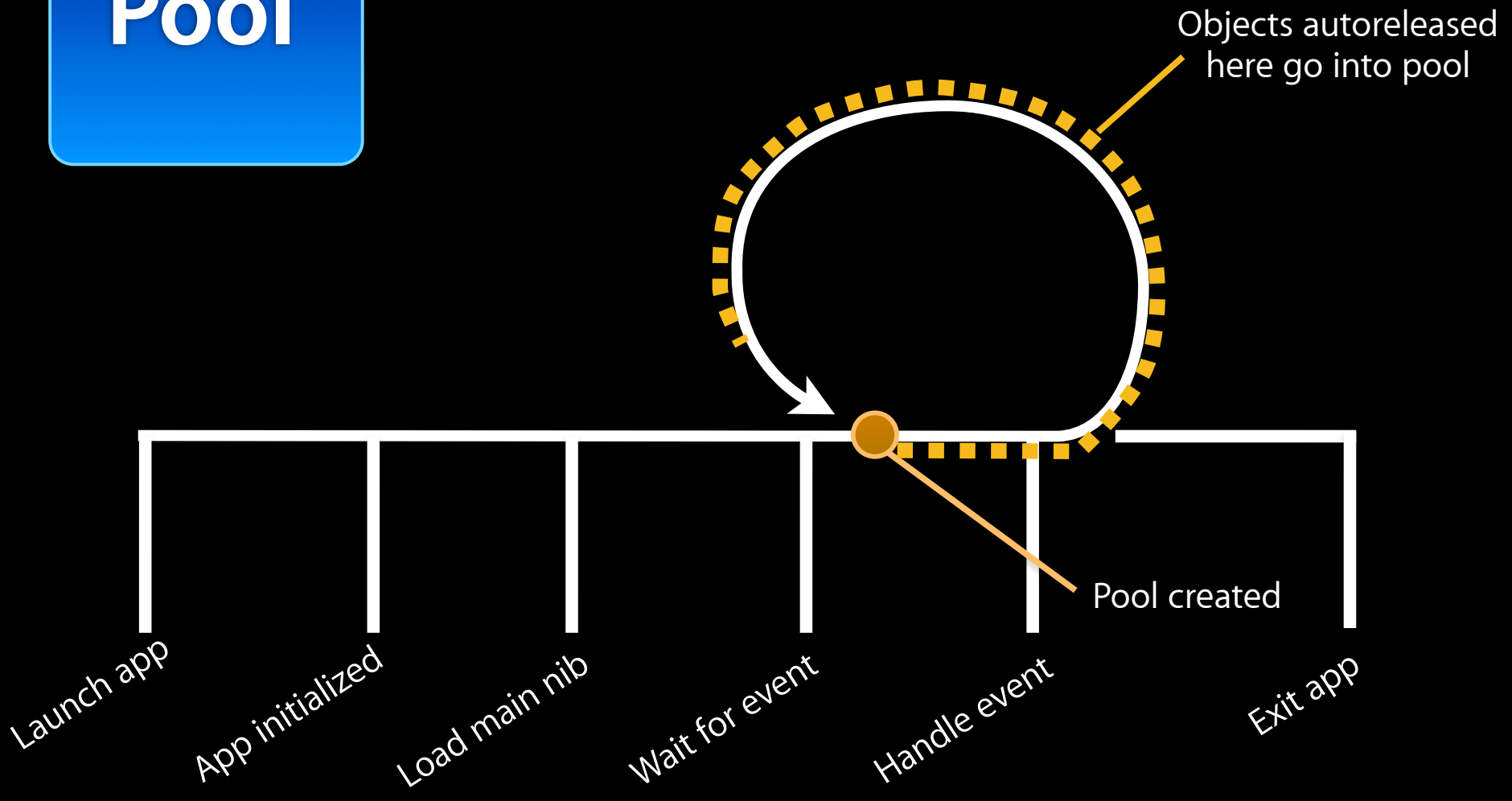
Autorelease Pools (in pictures)



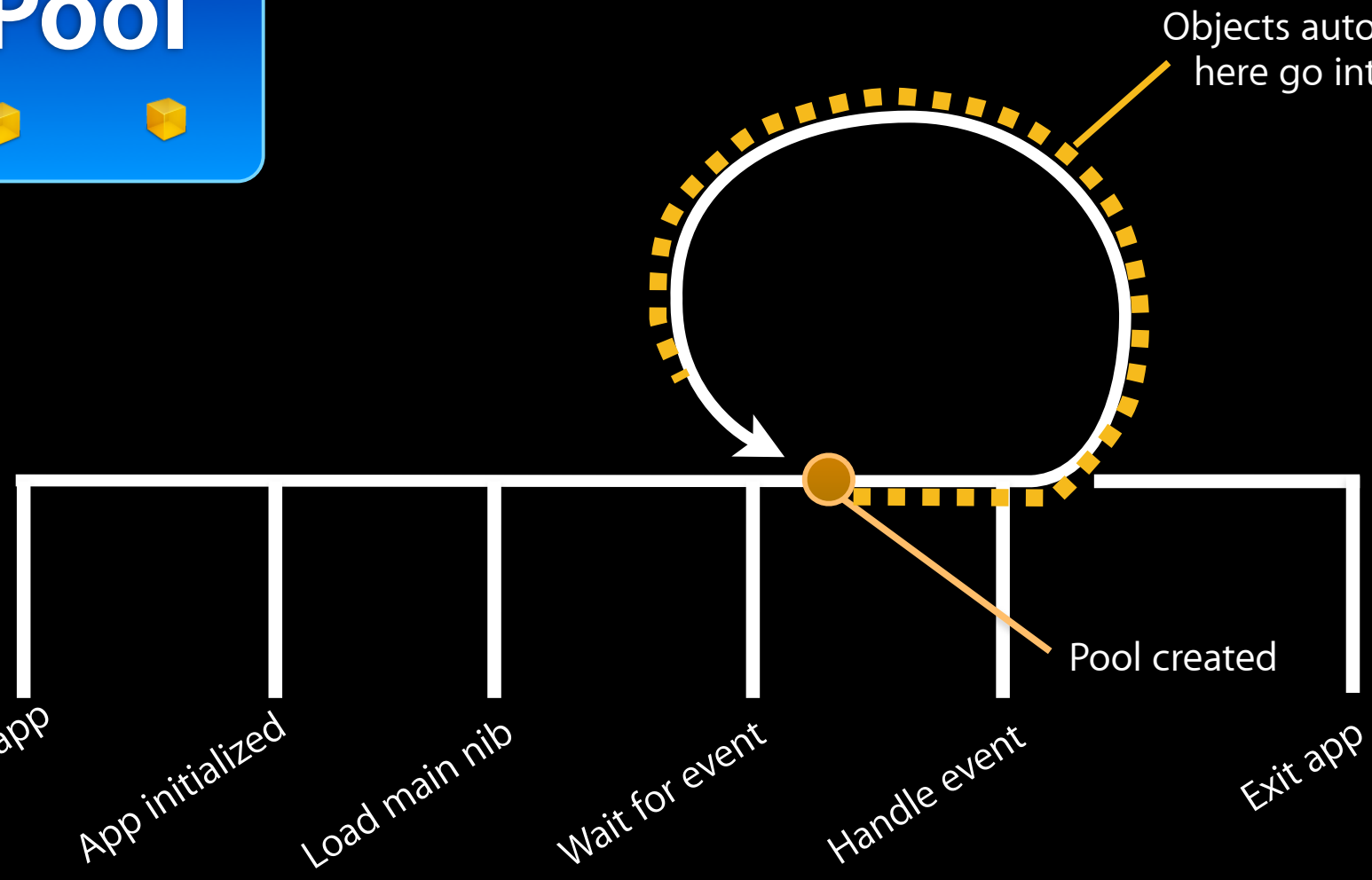
Autorelease Pools (in pictures)



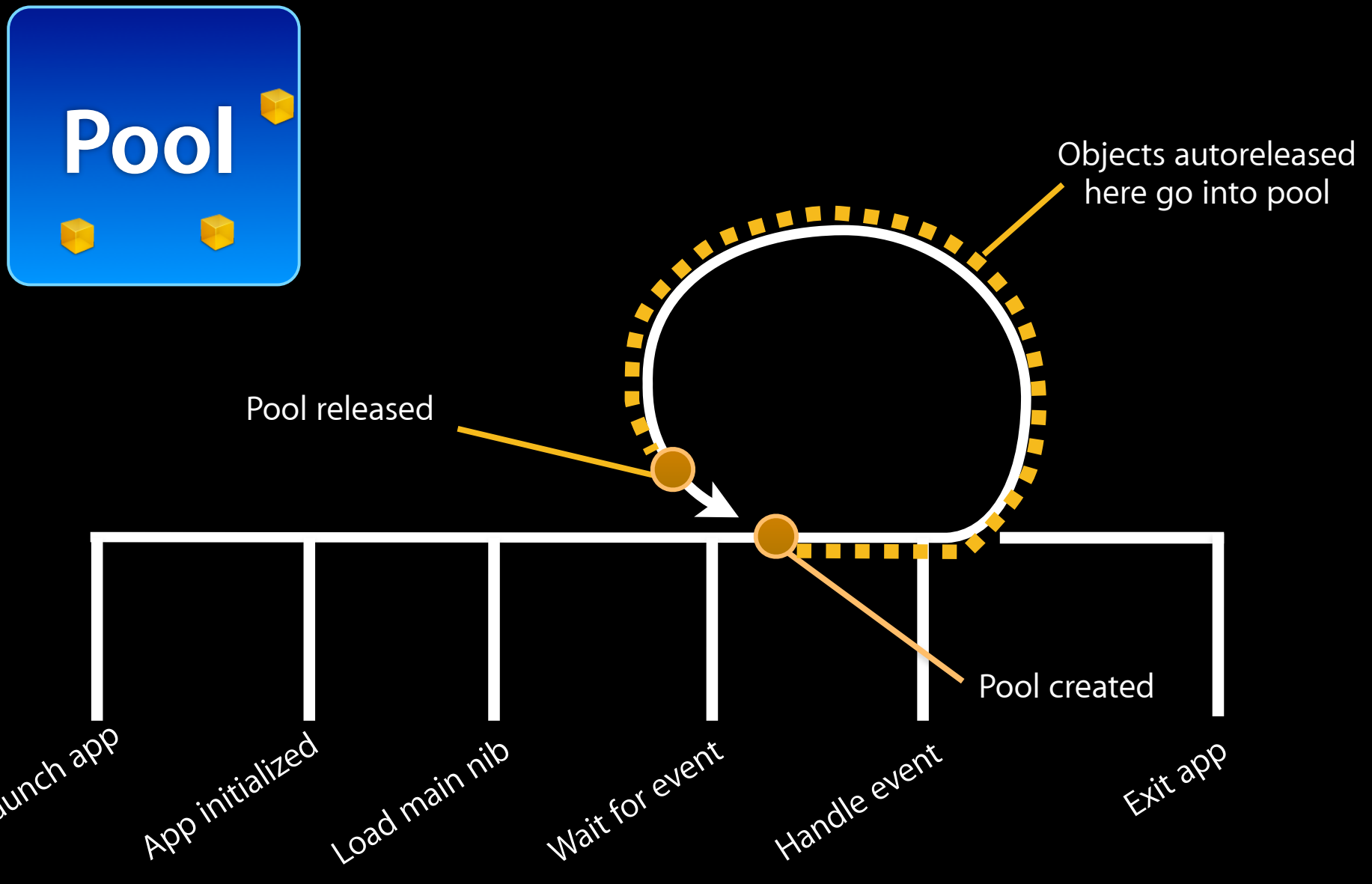
Autorelease Pools (in pictures)



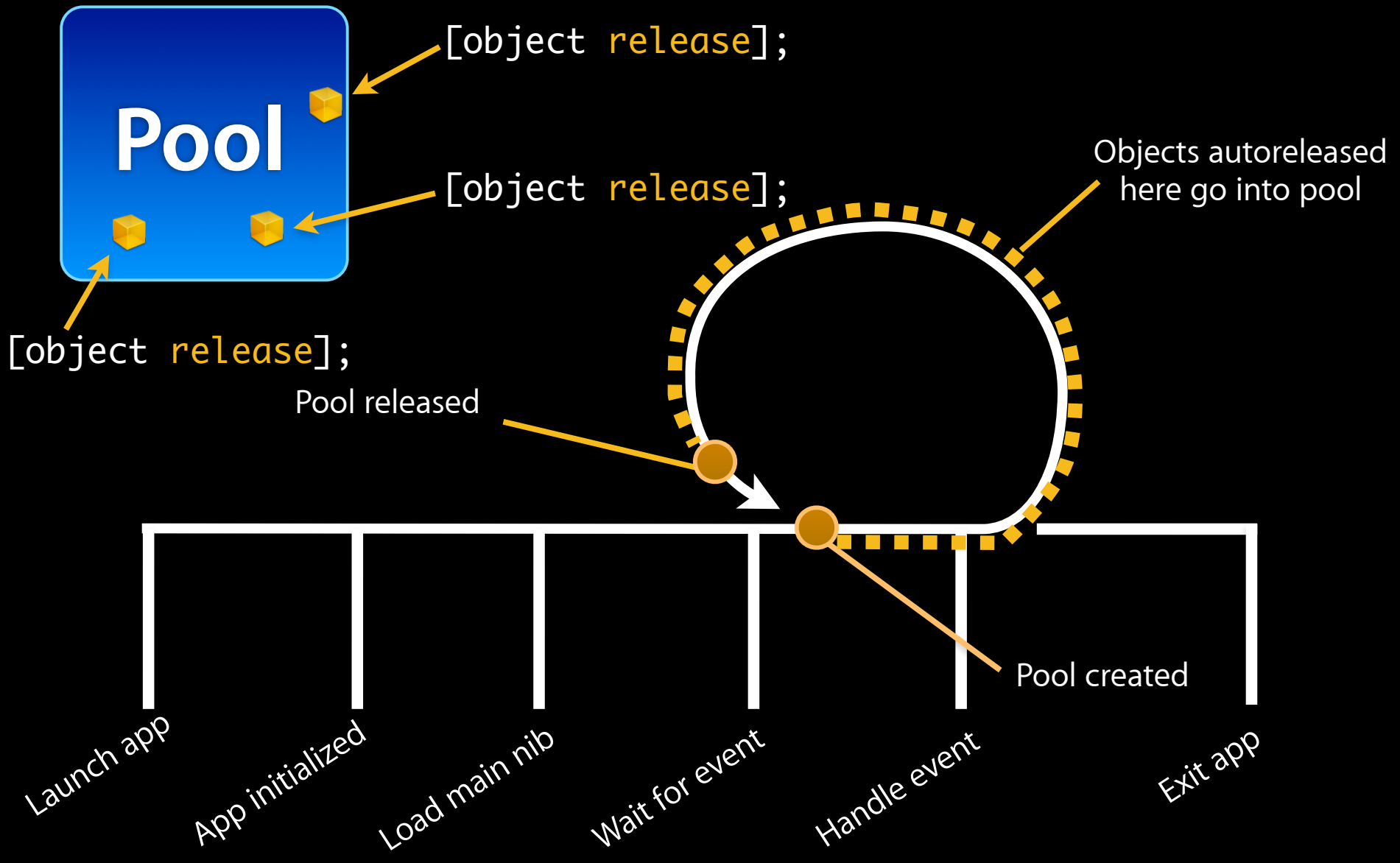
Autorelease Pools (in pictures)



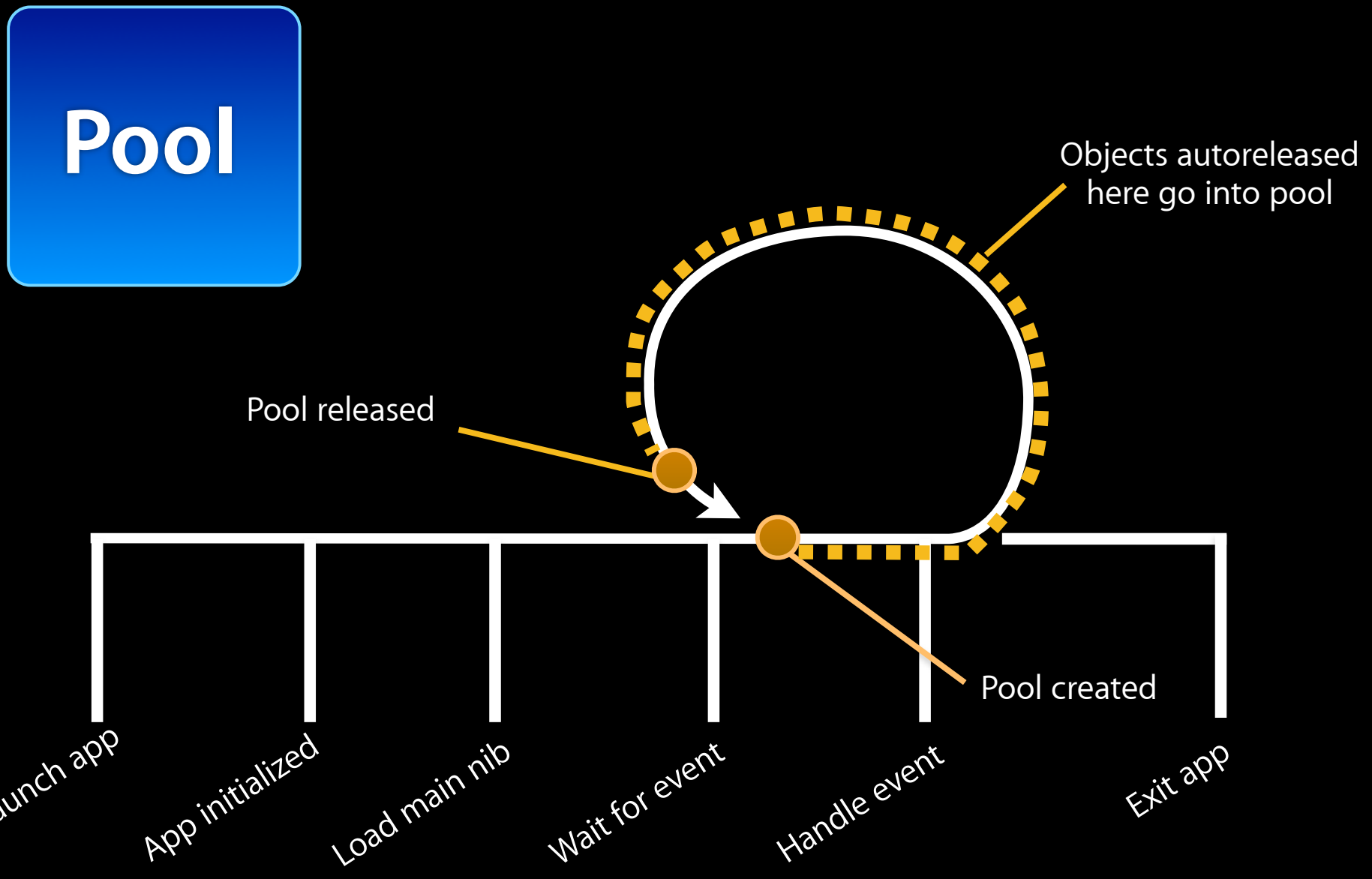
Autorelease Pools (in pictures)



Autorelease Pools (in pictures)



Autorelease Pools (in pictures)



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 does not have garbage collection

Objective-C Properties

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
```

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) {
          [name release];
          name = [value copy];
     }
}
- (void)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) {
        [name release];
        name = [value copy];
    }
}
- (void)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) {
        [name release];
        name = [value copy];
    }
}

- (void)canLegallyVote { ...
```

Synthesizing Properties

```
@implementation Person

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

@end
```

Property Attributes

- Read-only versus read-write

```
@property int age;  // read-write by default
@property (readonly) BOOL canLegallyVote;
```

- Memory management policies (only for object properties)

```
@property (assign) NSString *name; // pointer assignment
@property (retain) NSString *name; // retain called
@property (copy) NSString *name;   // copy called
```

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;

    // now do something with the new 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

Dot Syntax and self

- When used in custom methods, be careful with dot syntax for properties defined in your class
- References to properties and ivars behave very differently

```
@interface Person : NSObject
{
    NSString *name;
}
@property (copy) NSString *name;
@end

@implementation Person
- (void)doSomething {
    name = @“Fred”;          // accesses ivar directly!
    self.name = @“Fred”;     // calls accessor method
}
```

Common Pitfall with Dot Syntax

What will happen when this code executes?

```
@implementation Person
- (void)setAge:(int)newAge {
    self.age = newAge;
}
@end
```

This is equivalent to:

```
@implementation Person
- (void)setAge:(int)newAge {
    [self setAge:newAge]; // Infinite loop!
}
@end
```

Further Reading

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

Questions?