Custom Classes

iPhone and iPod touch Development
Fall 2009 — Lecture 3

Questions?

Announcements

- Project #1 out this evening
  - Due Tuesday September 15th by 11:59pm

Today's Topics

- Custom Classes
- Object Lifecycle
- Memory Management
- Properties

Custom Classes

Designing a Class

- Create a class
  - Person
- Determine the superclass
  - NSObject
- What properties should it have?
  - Name, age
- What actions can it perform
  - Register for class

Class Source File Organization

- Though not required by the compiler, ObjC classes are typically split into 2 files:
  - The interface for a class is declared in a header file and it usually given the extension ".h"
  - The implementation contains the actual ObjC source code and is given the extension ".m"

Naming Conventions

- Class names are in UpperCamelCase
  - File names are typically the same as the class name
- Instance variables are in lowerCamelCase
- Setter methods are named just like Java — setMemberName
- Getter methods are named a bit differently
  - Instead of getMemberName it is simply memberName
  - The “get” is omitted
  - Booleans tend to be a common exception and are typically prefixed with “is” instead of “get” — isBoolMemberName

Header File

- Declares the class
- Specifies the superclass
- Lists instance variables
- Declares public methods

Anatomy of an Interface ".h" File

```
@interface ClassName : ParentClassName {
    /* member declarations */
}
/* method declarations */
@end
```

Anatomy of an Interface ".h" File

```
@interface ClassName : ParentClassName {
    /* member declarations */
}
/* method declarations */
@end
```

ObjC classes are declared within @interface and @end

Anatomy of an Interface ".h" File

```
@interface ClassName : ParentClassName {
    /* member declarations */
}
/* method declarations */
@end
```

The ObjC class name appears immediately after @interface

Anatomy of an Interface ".h" File

```
@interface ClassName : ParentClassName {
    /* member declarations */
}
/* method declarations */
@end
```

The superclass is specified by a colon and the name of the parent class

Anatomy of an Interface ".h" File

```
@interface ClassName : ParentClassName {

    /* member declarations */

}

/* method declarations */

@end
```

Curly braces delimit the instance variable section

Anatomy of an Interface ".h" File

```
@interface ClassName : ParentClassName {
    /* member declarations */
}
/* method declarations */
@end
```

Instance variables (ivars) appear within the curly braces

Anatomy of an Interface ".h" File

```
@interface ClassName : ParentClassName {
    /* member declarations */
}
/* method declarations */
@end
```

Methods are declared between @interface and @end (and outside of the curly braces)

Person.h

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

@interface Person : NSObject {
  NSString * _name;
  int _age;
}

- (int)age;
- (void)setAge:(int)age;
- (NSString *)name;
- (void)setName:(NSString *)name;
- (BOOL)isAdult;

@end
```

Anatomy of an Implementation ".m" File

```
#import "ClassName.h"

@implementation ClassName

    /* method definitions */

@end
```

Anatomy of an Implementation ".m" File

```
#import "ClassName.h"

@implementation ClassName

    /* method definitions */

@end
```

The class interface is imported within the implementation file

Anatomy of an Implementation ".m" File

```
#import "ClassName.h"

@implementation ClassName

    /* method definitions */

@end
```

ObjC classes are defined within @implementation and @end

Anatomy of an Implementation ".m" File

```
#import "ClassName.h"

@implementation ClassName

    /* method definitions */

@end
```

The ObjC class name appears immediately after @implementation

Anatomy of an Implementation ".m" File

```
#import "ClassName.h"

@implementation ClassName

    /* method definitions */

@end
```

Methods are defined between @implementation and @end

Person.m

```
#import "Person.h"

@implementation Person

- (int)age {
  return _age;
}
- (void)setAge:(int)age {
  _age = age;
}
- (NSString *)name {
  return _name;
}
- (void)setName:(NSString *)name {
  _name = name;
}
- (BOOL)isAdult {
  return _age >= 18;
}

@end
```

Driver

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

int main (int argc, const char * argv[]) {
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

  // instantiate a Person instance
  Person *dwight = [[Person alloc] init];

  // set ivars
  [dwight setName:@"Dwight Schrute"];
  [dwight setAge:38];

  // dump dwight
  NSLog(@"%@ (%d)", [dwight name], [dwight age]);
  NSLog(@"is a %@", [dwight isAdult] ? @"Adult" : @"Child");

  [pool drain];
  return 0;
}
```

Calling Your Own Methods

- In languages like Java and C++ something like the following is completely normal...

```
void vote() {
  if(getAge() >= 18) {  /* vote */ }
}
```

- However, in ObjC we send messages, and thus we are going to need a receiver for the message
  - Here we use the ObjC keyword "self"
  - Similar to "this" in Java or C++

Sending Messages to self

```
#import "Person.h"

@implementation Person

/* ... */

- (BOOL)isAdult {
  // could simply use _age
  return [self age] >= 18;
}

- (void)vote {
  if ([self isAdult]) {
    NSLog(@"Going to vote");
  }
}

@end
```

Calling Super Class Methods

- If we want to invoke a superclass method, we use the keyword “super”

```
- (void)doSomething {

  // Call superclass implementation first
  [super doSomething];

  // Then do our custom behavior
  // ...
}
```

Object Life Cycle

Object Life Cycle

- Creating objects
- Memory management
- Destroying objects

Object Creation

- Object creation is a two-step process
  - First, we allocate memory to store the object
  - Next, we initialize the object's state
- The +alloc class method knows how much memory is needed and reserves it

```
+ (id)alloc;
```

- The -init instance method initializes values, optionally performing additional setup

```
- (id)init;
```

Create = Allocate and Initialize

- The +alloc/-init combo is so common that we typically chain them together

```
Person * 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]) {

      // do our initialization...
      _name = @"John Doe";
      _age = 0;
  }
  return self;

}

/* ... */

@end
```

Convenience Initialization Methods

- Classes may define multiple -init methods

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

- Frequently less specific initializers call more specific initializers with some default values

```
- (id)init {
  return [self initWithName:@"John Doe" age:30];
}
- (id) initWithName:(NSString *)name {
  return [self initWithName: name age:30];
}
- (id) initWithName:(NSString *)name age:(int)age {
  /* actual implementation */
}
```

Finishing Up With an Object

```
Person * person = [[Person alloc] init];
[person setName:@"Dwight Schrute"];
[person setAge:38];
/* do a bunch of stuff with person... */
/* done with object - now what? */
```

Memory Management

	Allocation	Deallocation
C	`malloc`	`free`
C++	`new`	`delete`
Objective-C	`+alloc`	`-dealloc`

- Calls must be balanced
  - Otherwise your program may leak memory or crash
- However, you'll never call -dealloc directly
  - There is one exception we'll see shortly

Reference Counting

- Reference counting is the process of storing the number of references to an Object
- The general premise is to stake a claim to an object if we want to use it
- When we no longer need an object, we'll release our claim to that object

Reference Counting in Objective-C

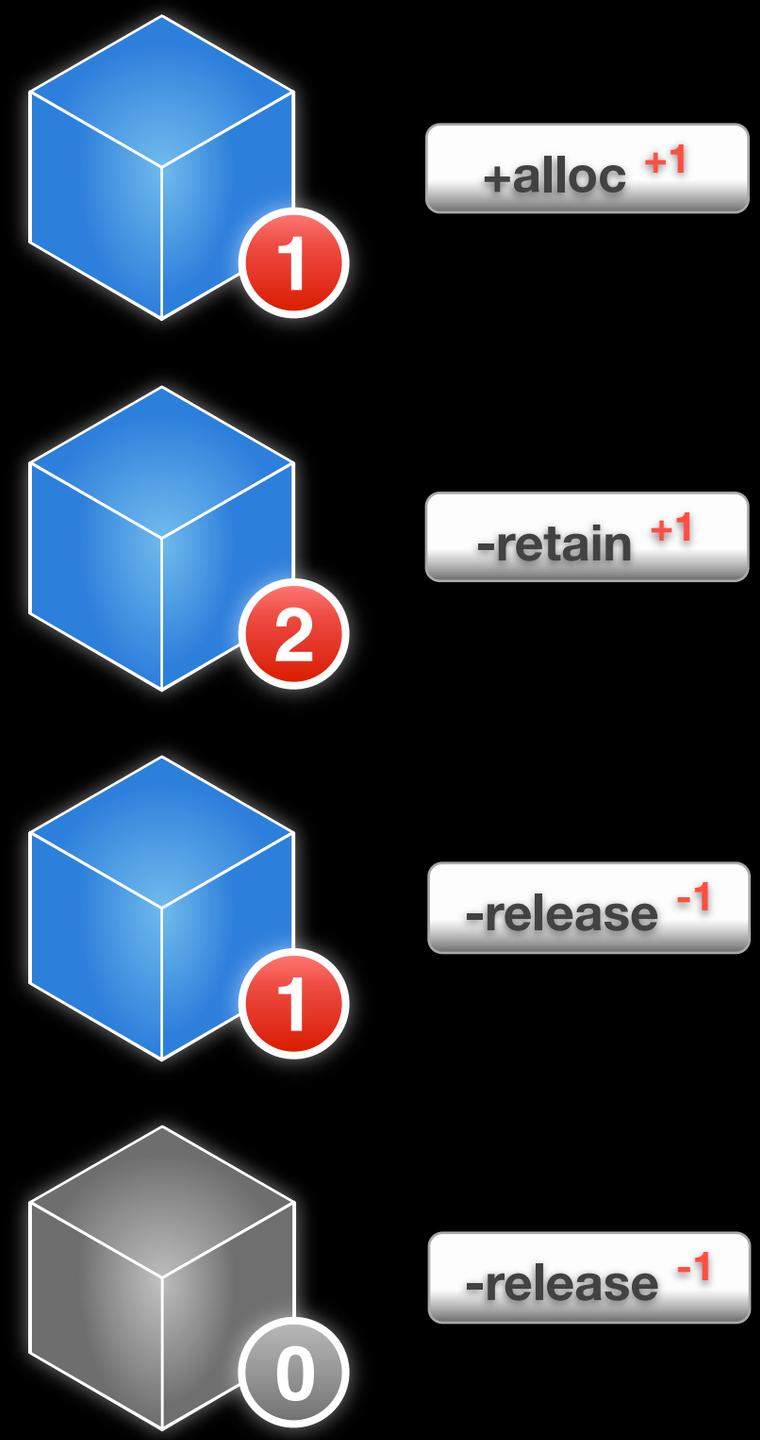
- Every object has a retain count
  - Defined as part of NSObject
  - As long as retain count is greater than zero, object is alive and valid
- +alloc, -copy and +new methods create objects with a retain count of one
- -retain increments retain count
- -release decrements retain count
- When retain count reaches zero the object is destroyed
  - -dealloc method is invoked automatically
  - Once -dealloc is called, there's no turning back

Balanced Calls

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

[person setName:@"Dwight Schrute"];

[person setAge:38];

/* do a bunch of stuff with person... */

/* done with object - give back memory */
[person release];
```

Reference Counting in Action

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

/* ... */

Person *brother = [person retain];

/* ... */

[person release];

/* ... */

[brother release];
```



3 Rules of Object Ownership

1. When you create an object via +alloc, -copy or +new the object has a retain count of one
   - You are responsible for releasing that object
2. If you retain an object, you need to eventually release it
   - These retains and releases must be balanced

Messaging Deallocated Objects

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

[person release];

[person somethingOrAnother];
```

This doesn't end well
will crash your app

Implementing a -dealloc Method

- You never call -dealloc explicitly from your code
- The only exception is [super dealloc] in -dealloc method

```
- (void)dealloc {

  // Do any cleanup specific to this class here

  // once we're done, call super's -dealloc to finish clean up
  [super dealloc];

}
```

So, We're Done Right?

- We need to make sure these rules are applied consistently within our class — are they?
- What about our getters/setters?

```
- (void)setName: (NSString *)name {
  _name = name;
}
```

What happens if we set the name after it has already been set?

Non-Leaking Person

- Our setters must release ownership of the previous object before assigning and accepting ownership of the new object

```
- (void)setName: (NSString *)name {
  if (_name != name) {
    [_name release];
    _name = [name retain];
  }
}
```

- We must also release ownership in our -dealloc method

```
- (void)dealloc {
  [_name release];
  [super dealloc];
}
```

Copying Strings

- Instead of calling -retain on a string, we usually instead choose to -copy the string
  - This gives us a new string with a retain count of one
  - We do this in case someone passes in an NSMutableString
  - Choosing to copy creates a local copy — if the string passed in is changed in another scope we are not affected

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

Okay, So Now We're Done?

- Well, let's say we implement that -description method for our class

```
- (NSString *)description {
  NSString * result = [[NSString alloc] initWithFormat:@"%@ (%d)",
                                _name, _name];
  return result;
}
```

- Are there any issues here?

We've called +alloc inside of this method.
How does the user know they need to release it?
This object will likely be leaked by the user.

Release Before Return?

- What happens if we try to release before returning...

```
- (NSString *)description {
  NSString * result = [[NSString alloc] initWithFormat:@"%@ (%d)",
                       _name, _name];
  [result release];
  return result;
}
```

- Are there any issues here?

The app will crash as soon as the user tries to invoke a method (on the now deallocated) object

Release After Return?

- What happens if we try to release after returning?

```
- (NSString *)description {
  NSString * result = [[NSString alloc] initWithFormat:@"%@ (%d)",
                       _name, _name];
  return result;
  [result release];
}
```

Wait? What?
Compiler Error!

- Are there any issues here?

Autorelease to the Rescue!

- We need a mechanism to automatically call -release at a later point in time
- This still provides valid object to the consumer
  - Gives them an opportunity to -retain and claim ownership

```
- (NSString *)description {
  NSString * result = [[NSString alloc] initWithFormat:@"%@ (%d)",
                       _name, _name];
  [result autorelease];
  return result;
}
```

Autoreleasing Objects

- Calling -autorelease flags an object to be sent release at some point in the future
- Lets 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 and Autorelease

- Methods whose names include +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];
// We're cool, no -release or -autorelease required
```

- This is a convention — follow it!

3 Rules of Object Ownership

1. When you create an object via +alloc, -copy or +new the object has a retain count of one
   - You are responsible for releasing that object

-autorelease also counts

2. If you retain an object, you need to eventually release it
   - These retains and releases must be balanced
3. When you get an object by any other mechanism, assume it has a retain count of one and has been autoreleased
   - You do not need to do any additional work to make sure it gets cleaned up
   - If you're going to hold onto it be sure to retain and release

More on Autorelease

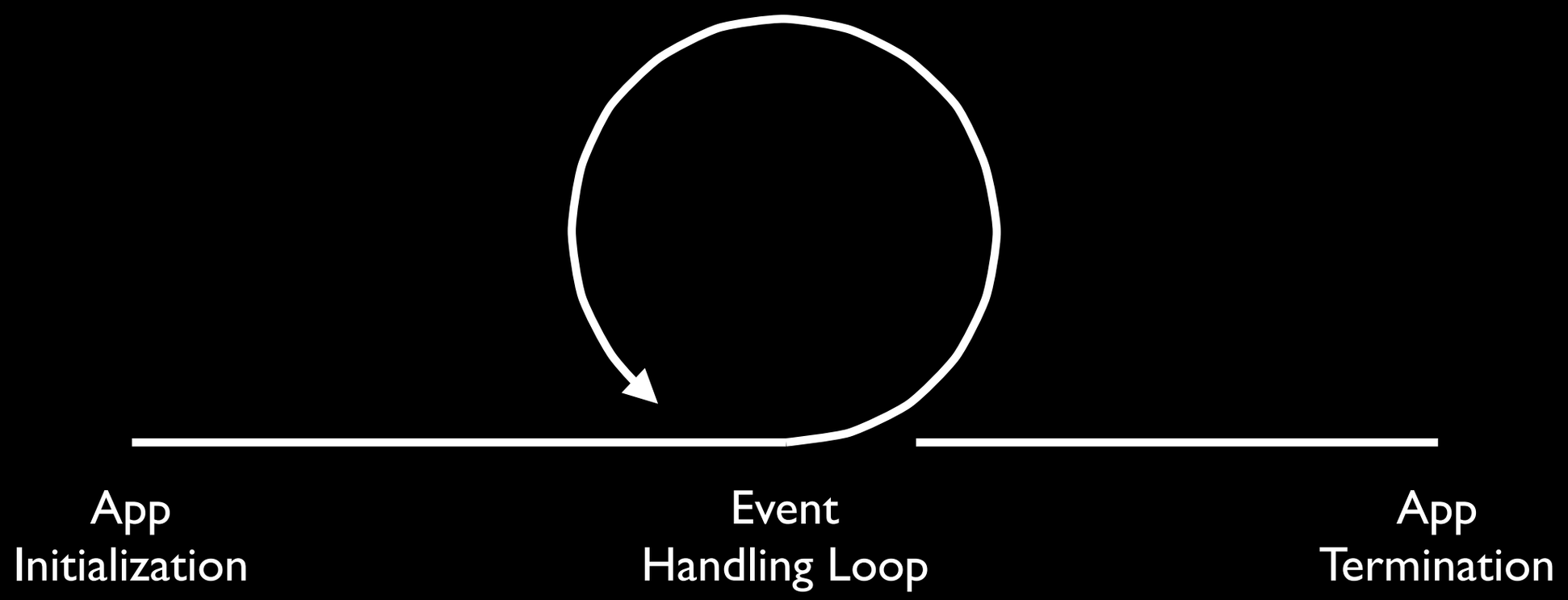
How Does -autorelease Work?

- Object is added to the current NSAutoreleasePool
- Autorelease pools track objects scheduled to be released
  - When the pool itself is released, it sends -release to all of the objects in the pool
- Autorelease pools are arranged in a stack
  - When a new autorelease pool is allocated it is added to the top of the stack
  - When an object is autoreleased it is added to the pool on the top of the stack
  - UIKit automatically wraps an autorelease pool around every event dispatch

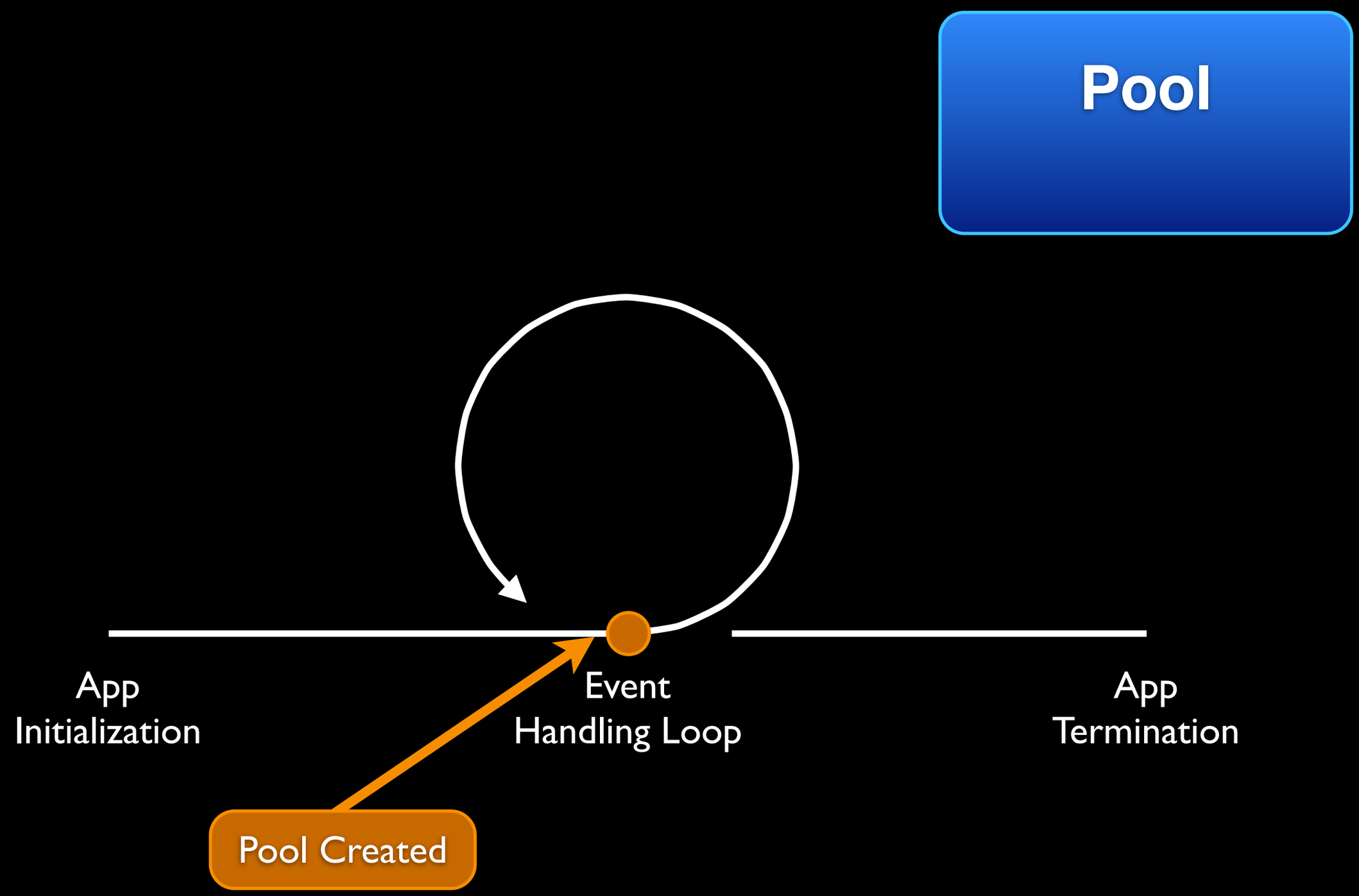
Autorelease Pools in Code

```
int main (int argc, const char * argv[]) {
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
  /* ... */
  [pool drain];
  return 0;
}
```

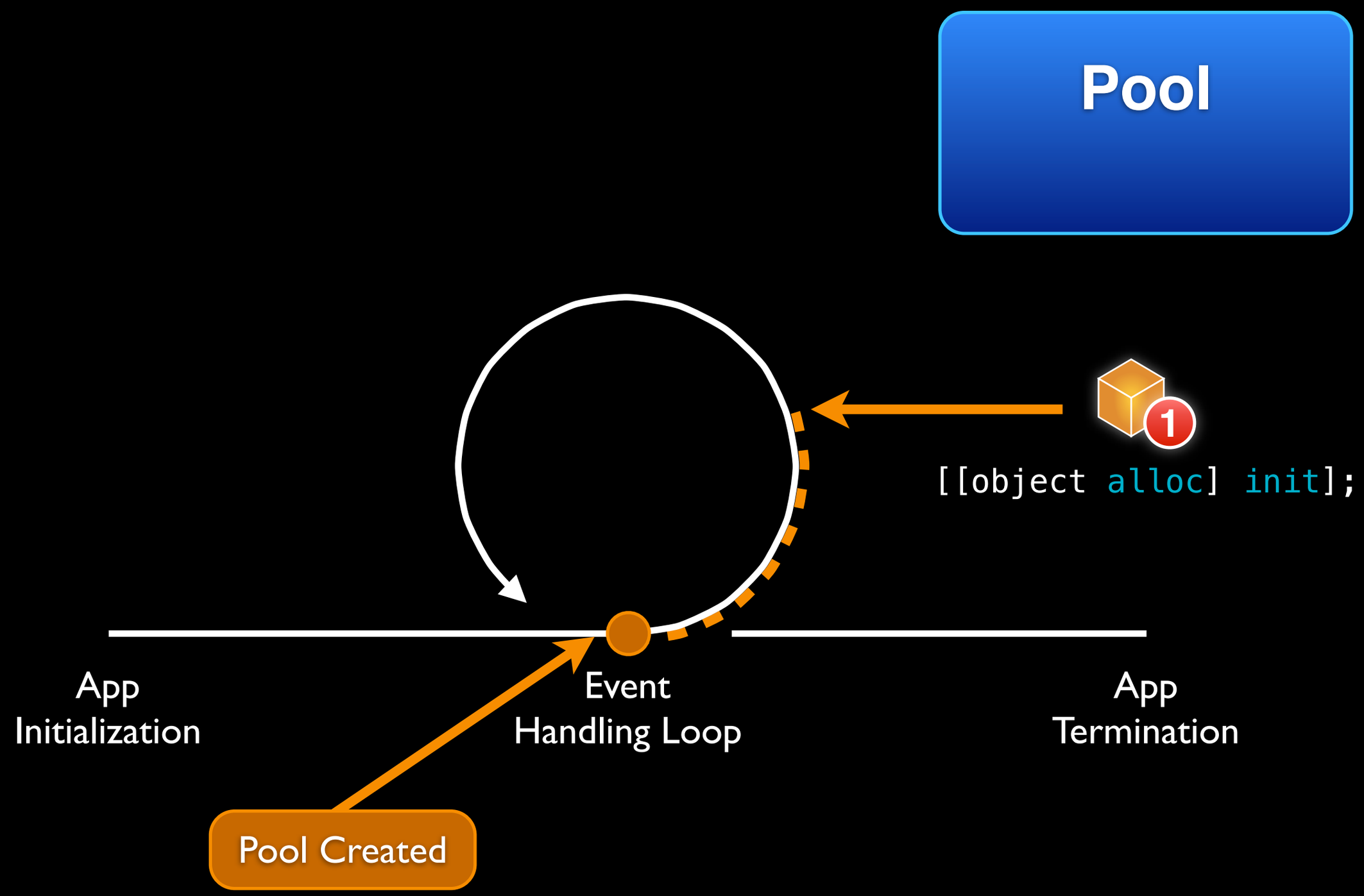
Autorelease Pool Example



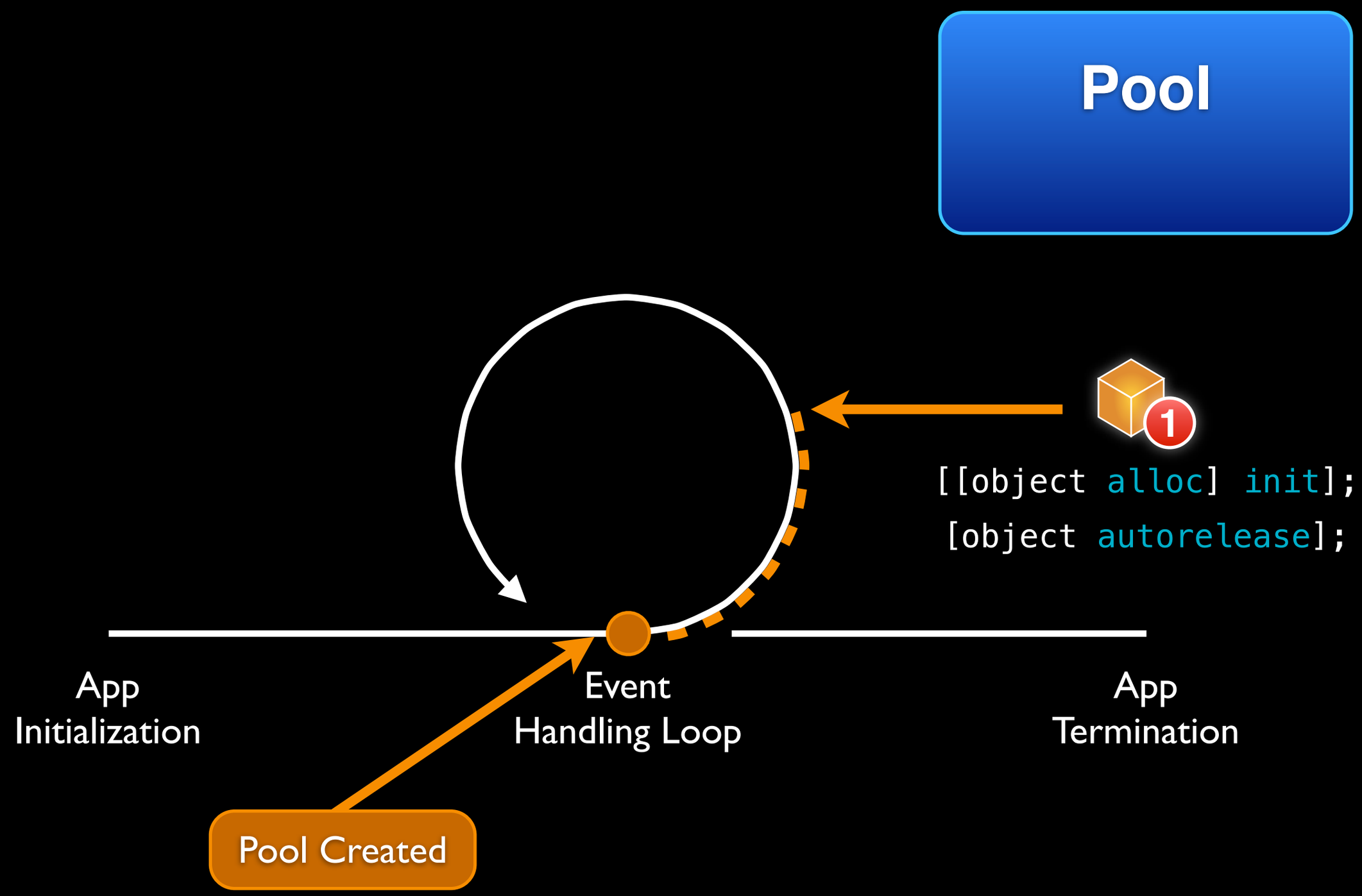
Autorelease Pool Example



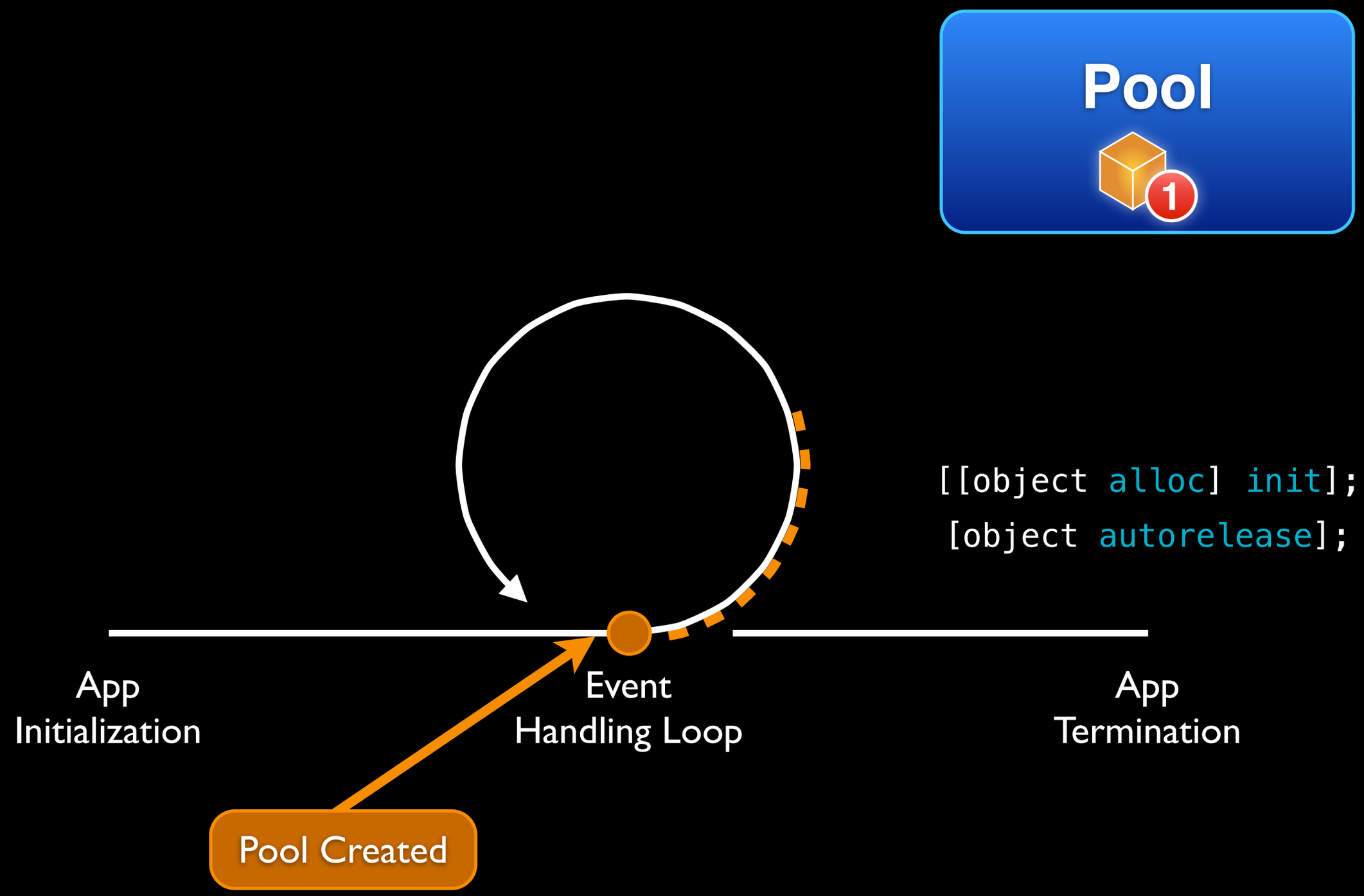
Autorelease Pool Example



Autorelease Pool Example

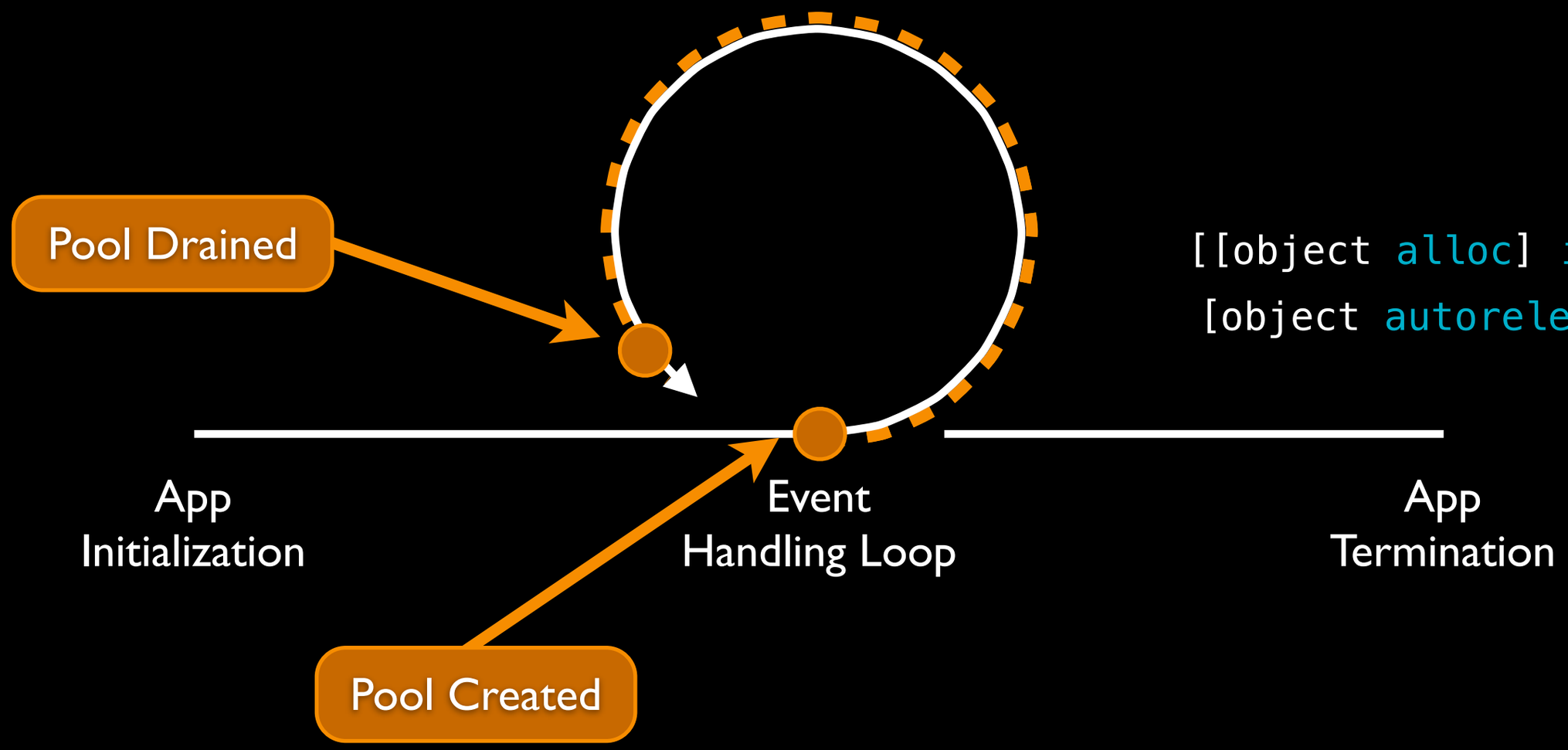


Autorelease Pool Example



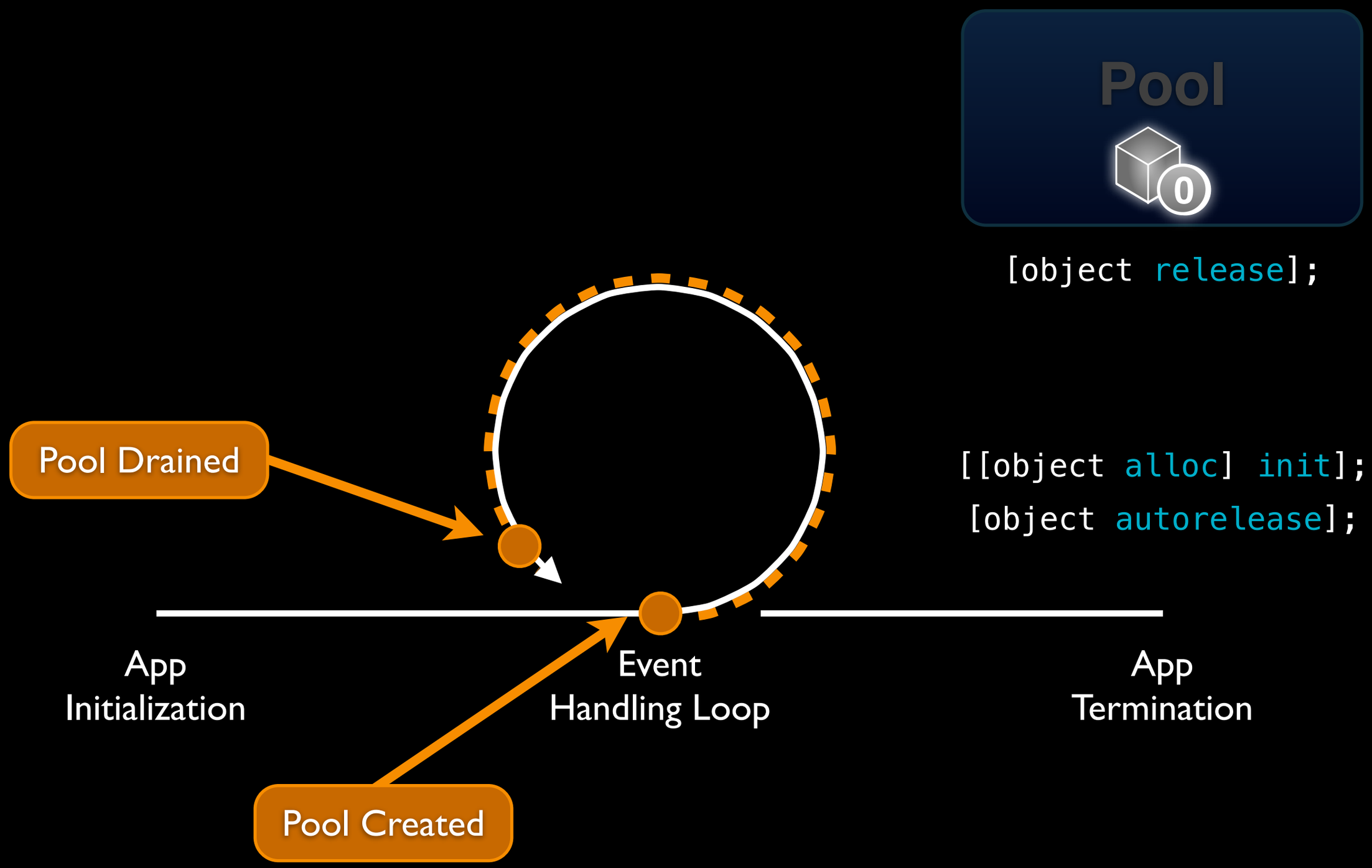

```
[[object alloc] init];
[object autorelease];
```

Autorelease Pool Example




```
[[object alloc] init];
[object autorelease];
```

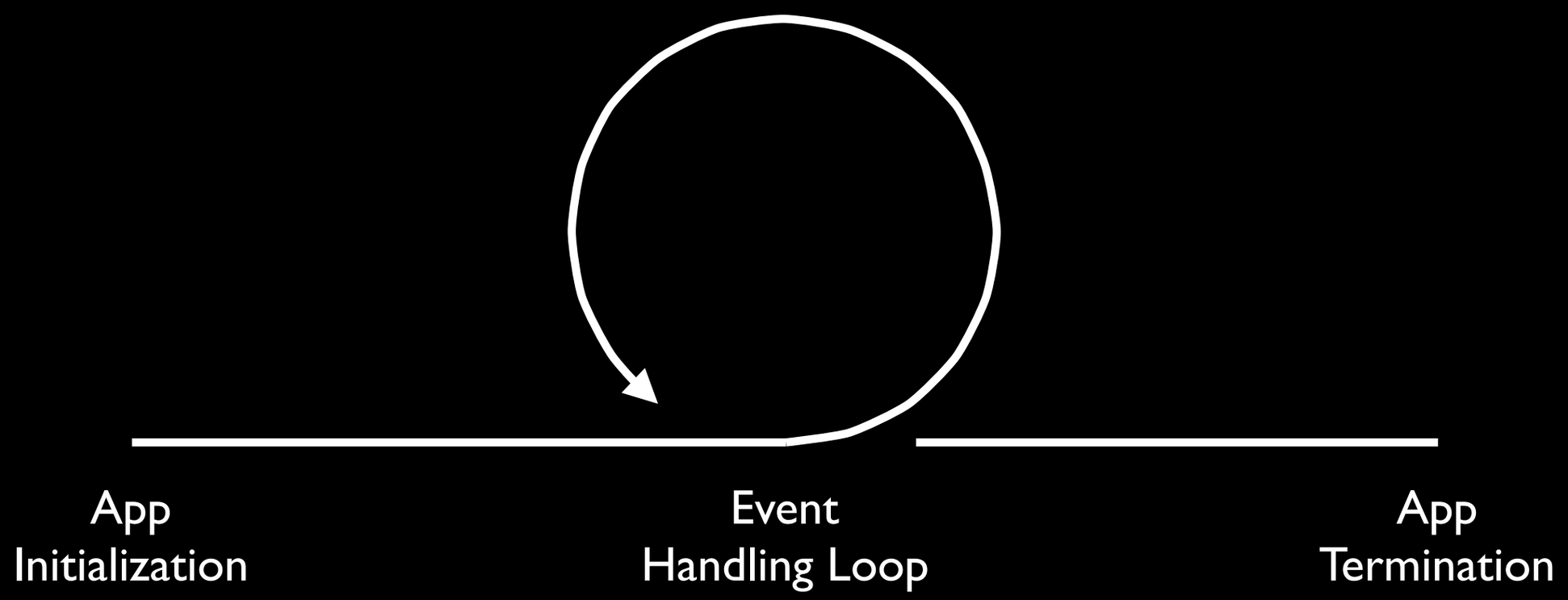
Autorelease Pool Example



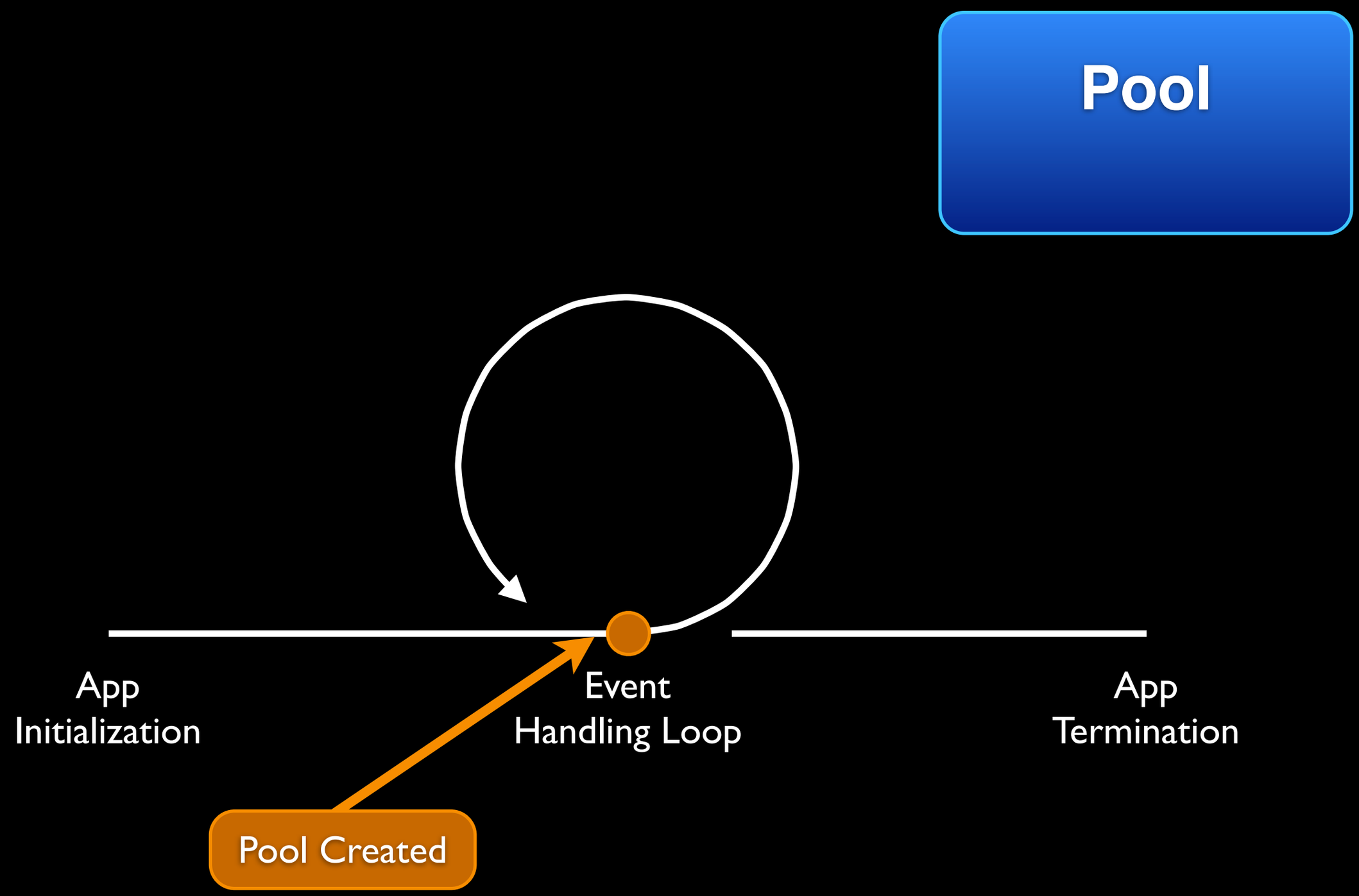
Hanging Onto an Autoreleased Object

- Many methods return autoreleased objects
  - Remember the naming conventions
  - They're hanging out in the pool and will be released when the pool is drained
- If you need to hold onto these objects, simply retain them
  - This will increment the retain count before the object is released
  - You'll need to remember to release at a later point

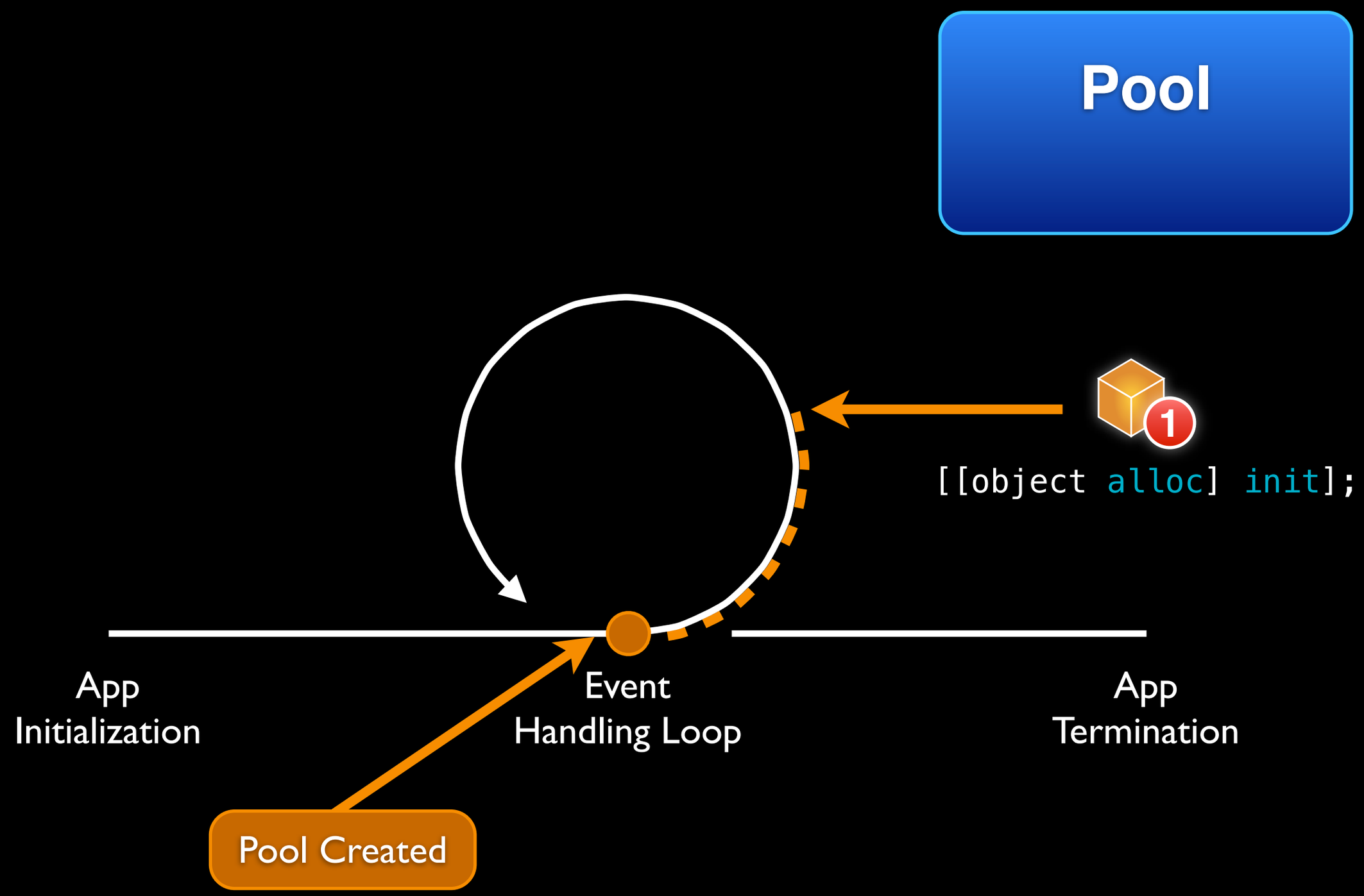
Autorelease Pool Example



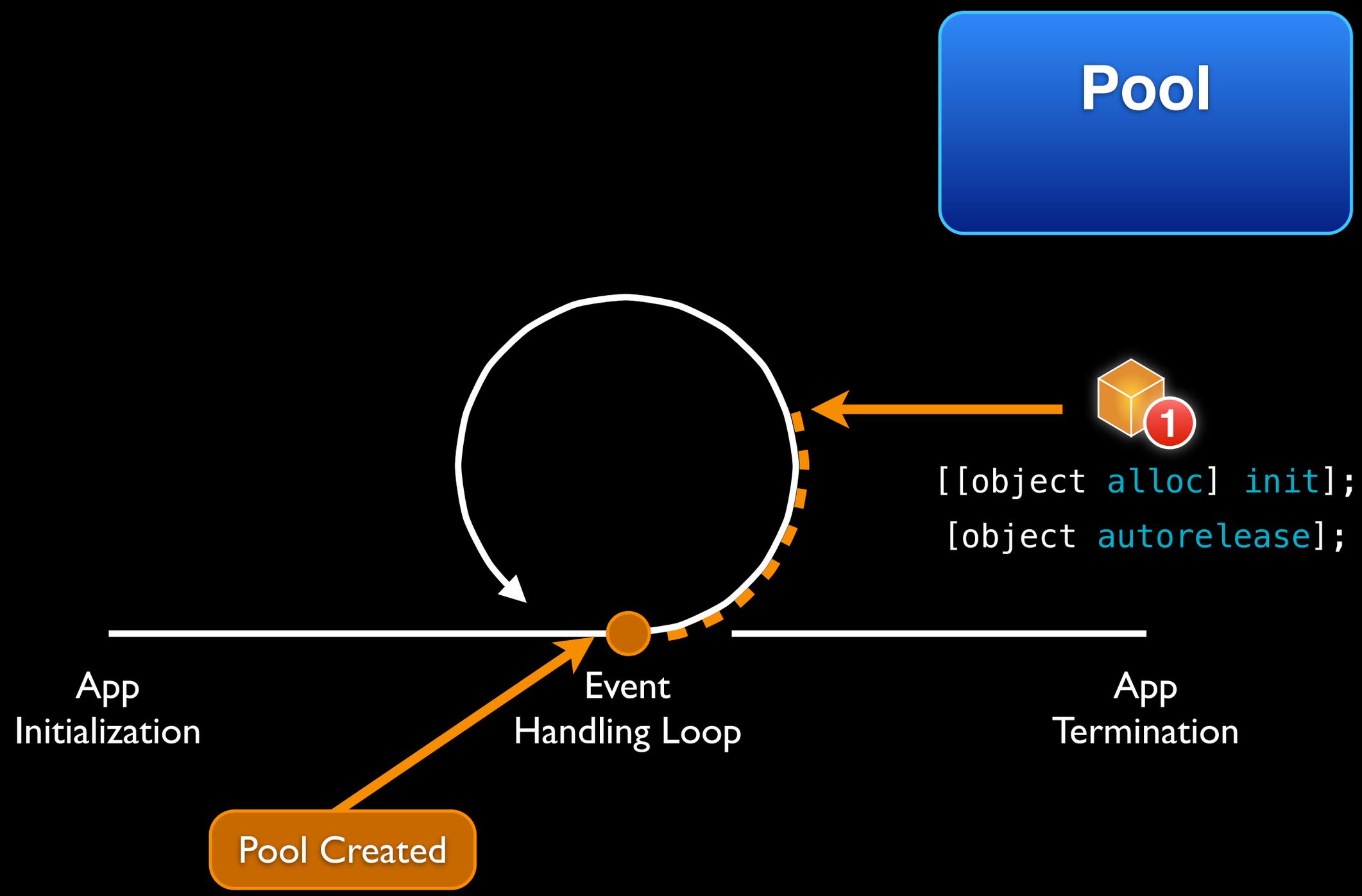
Autorelease Pool Example



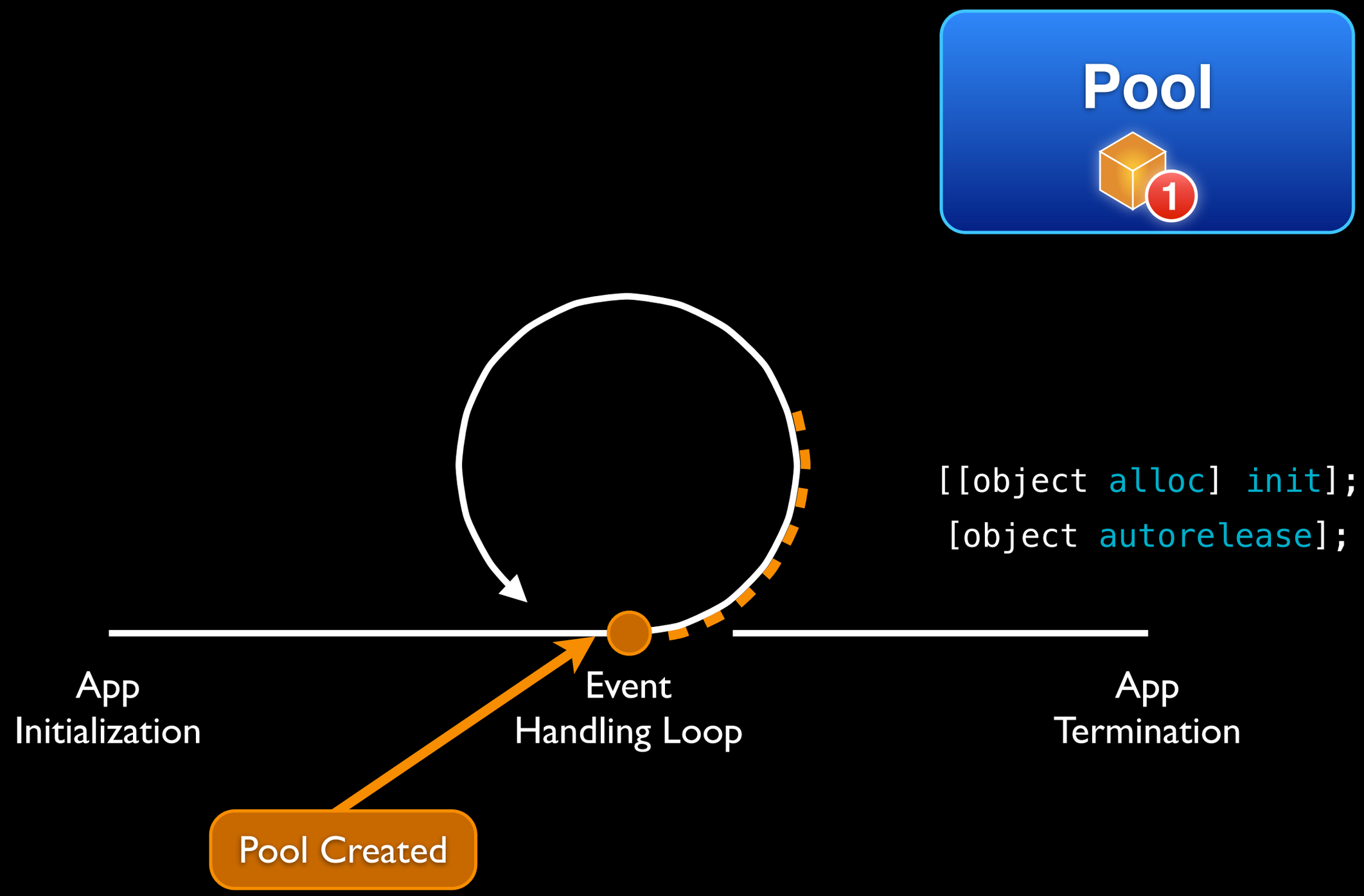
Autorelease Pool Example



Autorelease Pool Example



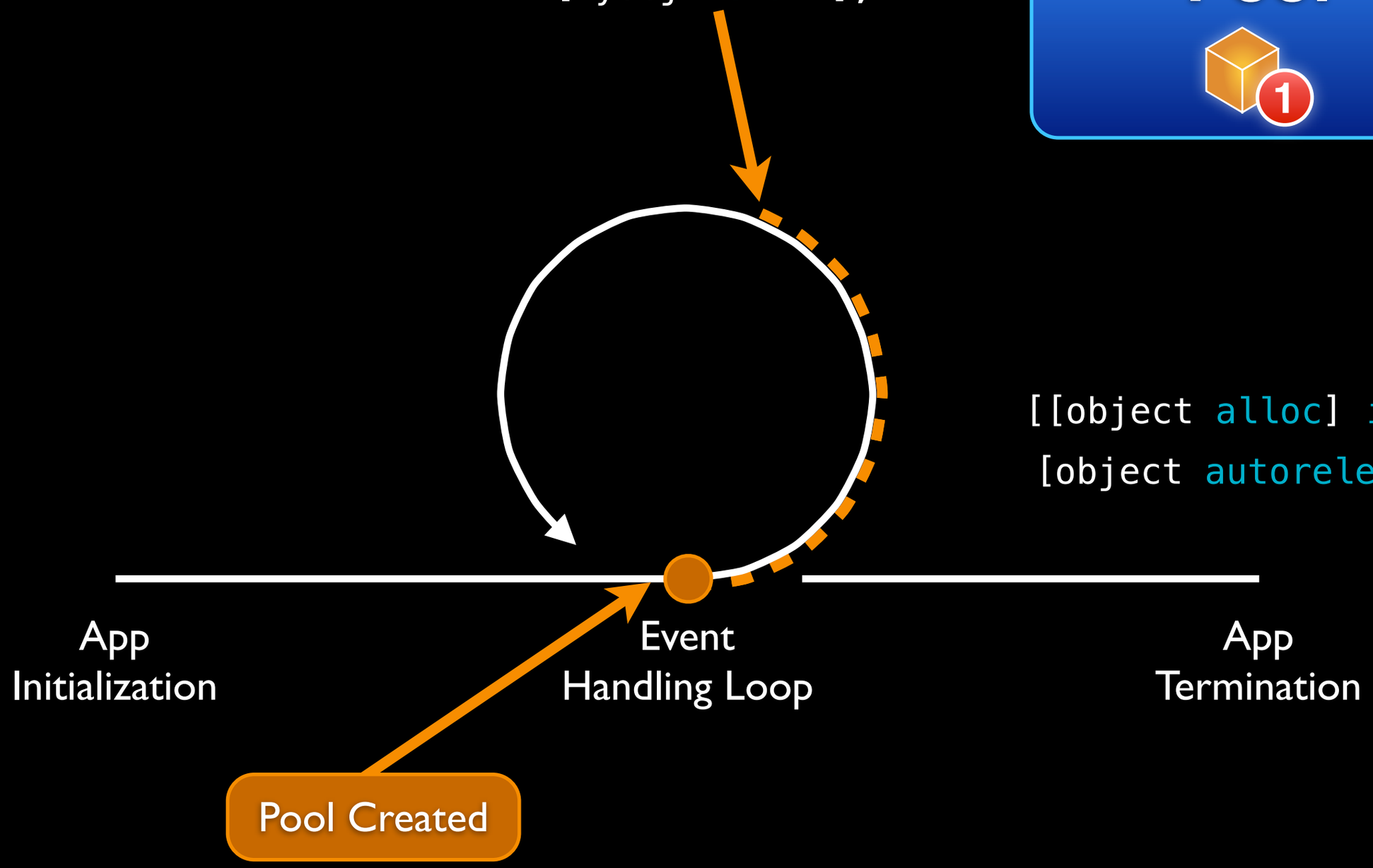
Autorelease Pool Example




```
[[object alloc] init];
[object autorelease];
```

Autorelease Pool Example

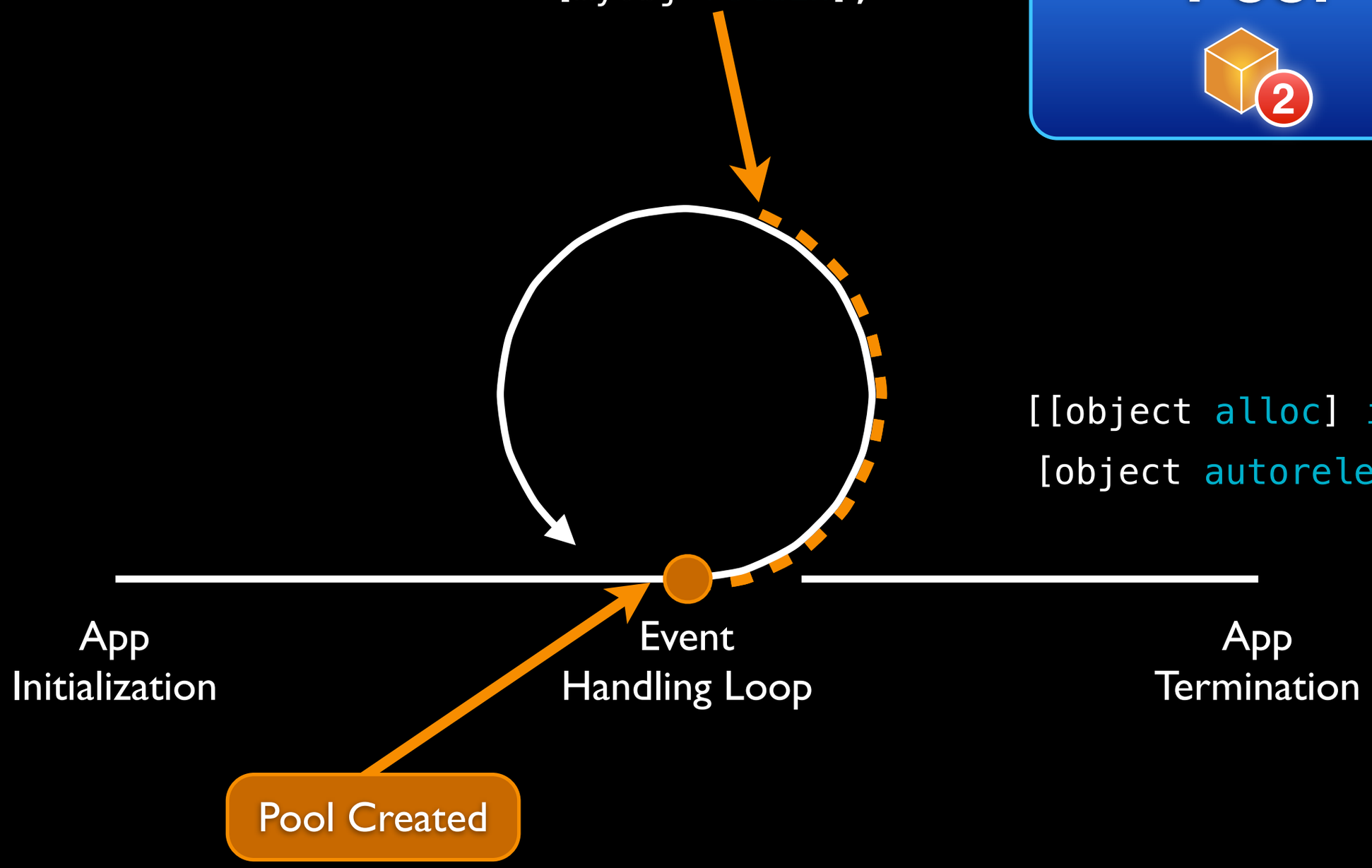
```
id myObj = object;
[myObj retain];
```




```
[[object alloc] init];
[object autorelease];
```

Autorelease Pool Example

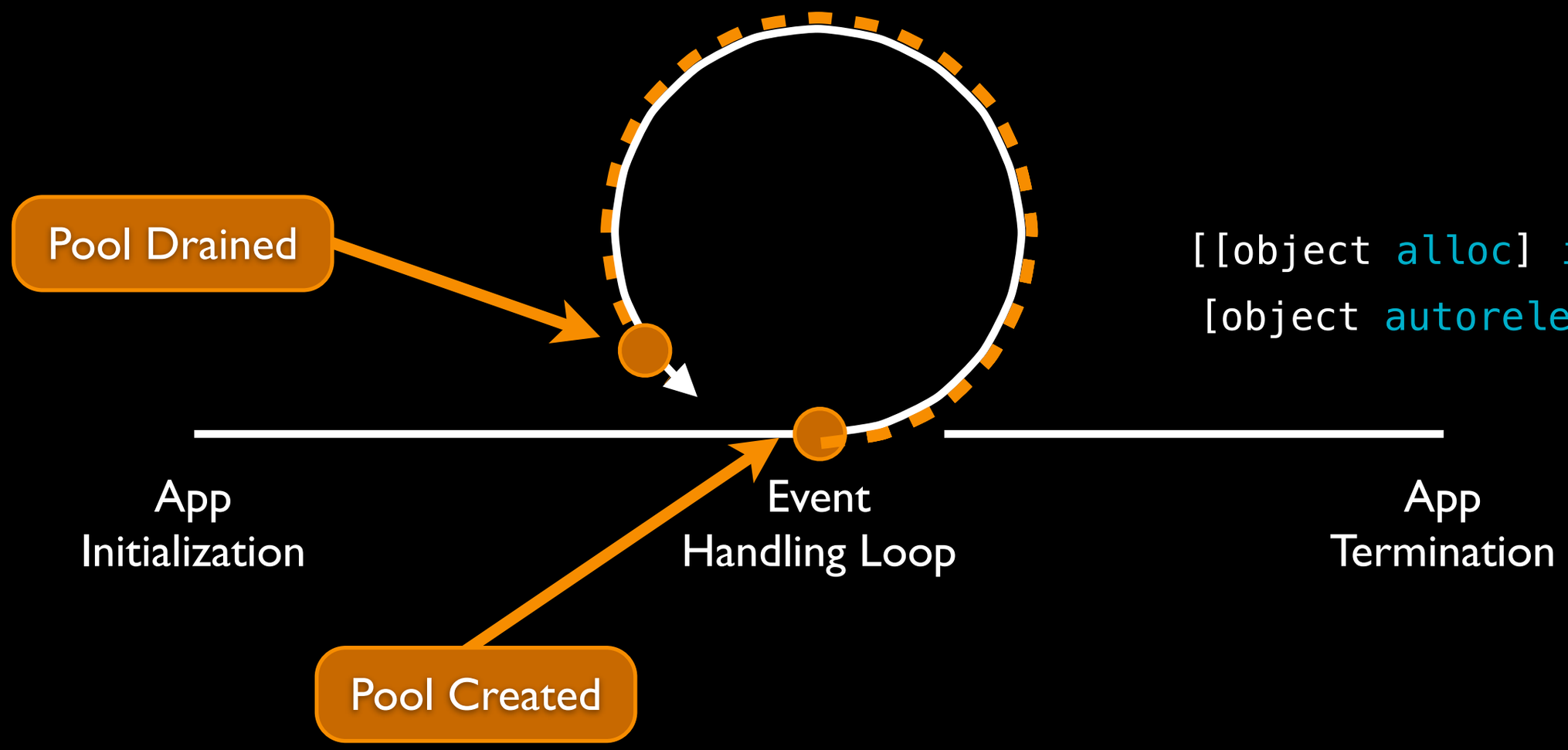
```
id myObj = object;
[myObj retain];
```




```
[[object alloc] init];
[object autorelease];
```

Autorelease Pool Example

```
id myObj = object;
[myObj retain];
```




```
[[object alloc] init];
[object autorelease];
```

Autorelease Pool Example

```
id myObj = object;
[myObj retain];
```


```
[object release];
```

```
[[object alloc] init];
[object autorelease];
```

Pool Drained

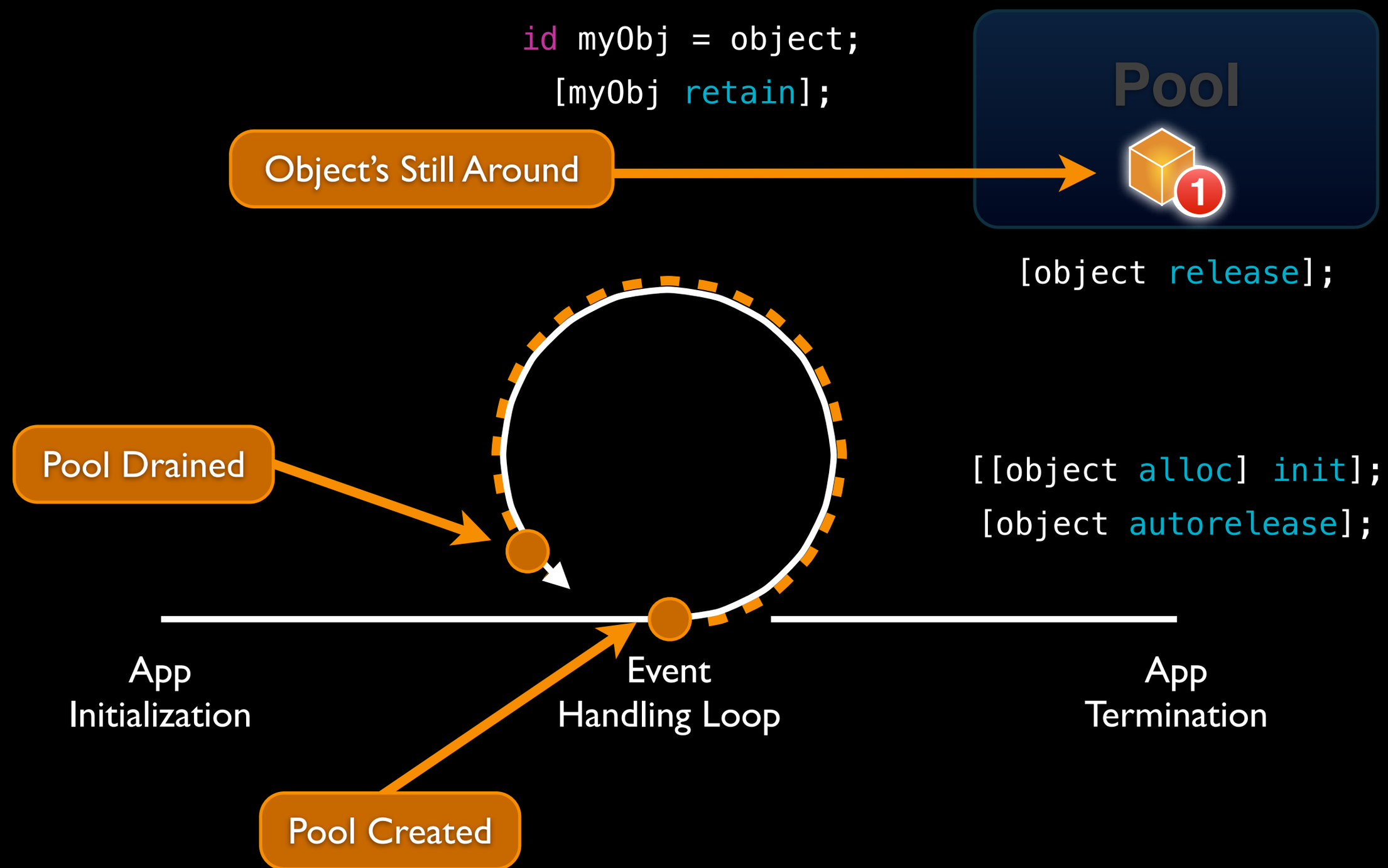
Pool Created

App Initialization

Event Handling Loop

App Termination

Autorelease Pool Example



Autorelease vs. Garbage Collection

- Autorelease is not garbage collection
  - It's a mechanism to aid in allocation and deallocation
  - But, the user is still responsible for managing objects
- Objective-C 2.0 added garbage collection to the language
  - However, (presumably) for performance reasons the iPhone does not provide a garbage collection facility
  - May it be added one day? who knows?

Properties

Properties

- Objective-C 2.0 added properties to the language
- Provide access to object attributes
- Shortcut to implementing getter/setter methods
- They also allow you to specify behaviors such as...
  - Read-only (just getter) vs. read-write (getter and setter)
  - Memory management policy

Person.h

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

@interface Person : NSObject {
   NSString * _name;
   int _age;
}

- (int)age;
- (void)setAge:(int)age;
- (NSString *)name;
- (void)setName:(NSString *)name;
- (BOOL)isAdult;

@end
```

Person.h

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

@interface Person : NSObject {
   NSString * _name;
   int _age;
}

- (int)age;
- (void)setAge:(int)age;
- (NSString *)name;
- (void)setName:(NSString *)name;
- (BOOL)isAdult;

@end
```

Lots of boilerplate here

Person.h

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

@interface Person : NSObject {
   NSString * _name;
   int _age;
}

- (int)age;
- (void)setAge:(int)age;
- (NSString *)name;
- (void)setName:(NSString *)name;
- (BOOL)isAdult;

@end
```

Lots of boilerplate here

@property

- The @property directive is specified in the interface
- Identifies the properties for the class
  - Usually they are instance variables
  - Though it need not be restricted to just ivars
- For Person, we're going to specify the following behaviors...
  - For name, we'll choose to copy the argument passed in (in case someone passes in an NSMutableString)
  - For isAdult, we'll specify that it is readonly — a value that can only be read and not set

Person.h with Properties

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

@interface Person : NSObject {
   NSString * _name;
   int _age;
}

  @property int age;
  @property(copy) NSString *name;
  @property(readonly) BOOL isAdult;

@end
```

Person.m

```
#import "Person.h"

@implementation Person

- (int)age {
  return _age;
}
- (void)setAge:(int)age {
  _age = age;
}
- (NSString *)name {
  return _name;
}
- (void)setName:(NSString *)name {
  if (_name != name) {
    [_name release];
    _name = [name retain];
  }
}
- (BOOL)isAdult {
  return _age >= 18;
}

@end
```

Person.m

```
#import "Person.h"

@implementation Person

- (int)age {
  return _age;
}
- (void)setAge:(int)age {
  _age = age;
}
- (NSString *)name {
  return _name;
}
- (void)setName:(NSString *)name {
  if (_name != name) {
    [_name release];
    _name = [name retain];
  }
}
- (BOOL)isAdult {
  return _age >= 18;
}

@end
```

Even more boilerplate here

Person.m

```
#import "Person.h"

@implementation Person

- (int)age {
  return _age;
}
- (void)setAge:(int)age {
  _age = age;
}
- (NSString *)name {
  return _name;
}
- (void)setName:(NSString *)name {
  if (_name != name) {
    [_name release];
    _name = [name retain];
  }
}
- (BOOL)isAdult {
  return _age >= 18;
}

@end
```

Even more boilerplate here

@synthesize

- Instead of providing our own method implementations, we'll let the compiler synthesize them for us
- The getter/setter methods are then built out based on the @property and @synthesize directives
- If no @synthesize directive appears for a @property, implementation(s) must be supplied
- If a given getter/setter is implemented, then it overrides anything provided by @synthesize

Person.m with Synthesized Properties

```
#import "Person.h"

@implementation Person

@synthesize age = _age;
@synthesize name = _name;

- (BOOL)isAdult {
  return _age >= 18;
}

@end
```

All gets boiled down to these two synthesize statements

We still need to provide this implementation, as it is not backed by an ivar

Sample Property Usage (Traditional Notation)

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

int main (int argc, const char * argv[]) {
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

  // instantiate a Person instance
  Person *dwight = [[Person alloc] init];

  // set ivars
  [dwight setName:@"Dwight Schrute"];
  [dwight setAge:38];

  // dump dwight
  NSLog(@"%@ (%d)", [dwight name], [dwight age]);
  NSLog(@"is a %@", [dwight isAdult] ? @"Adult" : @"Child");

  [pool drain];
  return 0;
}
```

Sample Property Usage (Dot Notation)

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

int main (int argc, const char * argv[]) {
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

  // instantiate a Person instance
  Person *dwight = [[Person alloc] init];

  // set ivars
  dwight.name = @"Dwight Schrute";
  dwight.age = 38;

  // dump dwight
  NSLog(@"%@ (%d)", dwight.name, dwight.age);
  NSLog(@"is a %@", dwight.isAdult ? @"Adult" : @"Child");

  [pool drain];
  return 0;
}
```

To Property or Not to Property

- Properties seem to be one of the more controversial additions to ObjC 2.0
  - Some people are full-adopters of properties
  - Some partially use properties
    - Use @property & @synthesize to create methods
    - But avoid using dot notation
  - Some avoid it altogether

Properties in Practice

- Many newer APIs use @property older APIs use getter and 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 be non-obvious

Naming of Instance Variables

- You'll notice that I've been naming ivars using preceding underscores
- There tend to be different opinions on ivar naming
  - Prefix with a leading underscore — _memberName
  - Omit all underscores — memberName
  - Add a trailing underscore — memberName_

Synthesize with Matching Instance Variables

- Up to this point, I've shown synthesizing a property to an underscored ivar, like so...

```
@synthesize name = _name;
@synthesize age = _age;
```

- However, if the ivar is the same name as the property, you may omit the "= ivar" notation, like so...

```
@synthesize name;
@synthesize age;
```

A Property Example

- When using properties, it is important to understand what is happening behind the scenes
- For example...

```
- (void)setName:(NSString *)newName {
  if (self.name != newName) {
    [name release];
    self.name = newName;
  }
}
```

- Is actually...

```
- (void)setName:(NSString *)newName {
  if ([self name] != newName) {
    [name release];
    [self setName: newName];
  }
}
```

Infinite Recursion!

Additional Resources

- Chapter 5 of "The Objective-C 2.0 Programming Language" is a valuable resource for understanding properties
  - The section "Property Declaration Attributes" details the different attributes/behaviors which can be specified
  - http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf

For Next Class

- Read chapters 4, 6, 9 & 10 of "The Objective-C 2.0 Programming Language"
  - http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf