COMP327 Mobile Computing

Session: 2012-2013

Lecture Set 1a - Objective-C Refresher and the Foundation Framework

In these Slides...

- **We will cover...**
  - Review of The Objective-C language
    - Methods/Classes/Objects
    - Message Passing
    - Object Creation
  - What's new
    - Properties and Accessor Method Synthesis
    - ARC and Memory Management
  - Recap Foundation Framework
    - An introduction to the framework
    - Mutable Classes
    - Value Objects and Container Classes
- **Related Lab:**
  - Creating a fraction class

Objective-C and Foundation

These slides provide an introduction to Objective-C, with a focus on the object-oriented features.

Later slides will introduce more features such as protocols, design patterns etc.

UNIVERSITY OF LIVERPOOL

Objective-C Refresher

Objective-C and the Foundation Framework

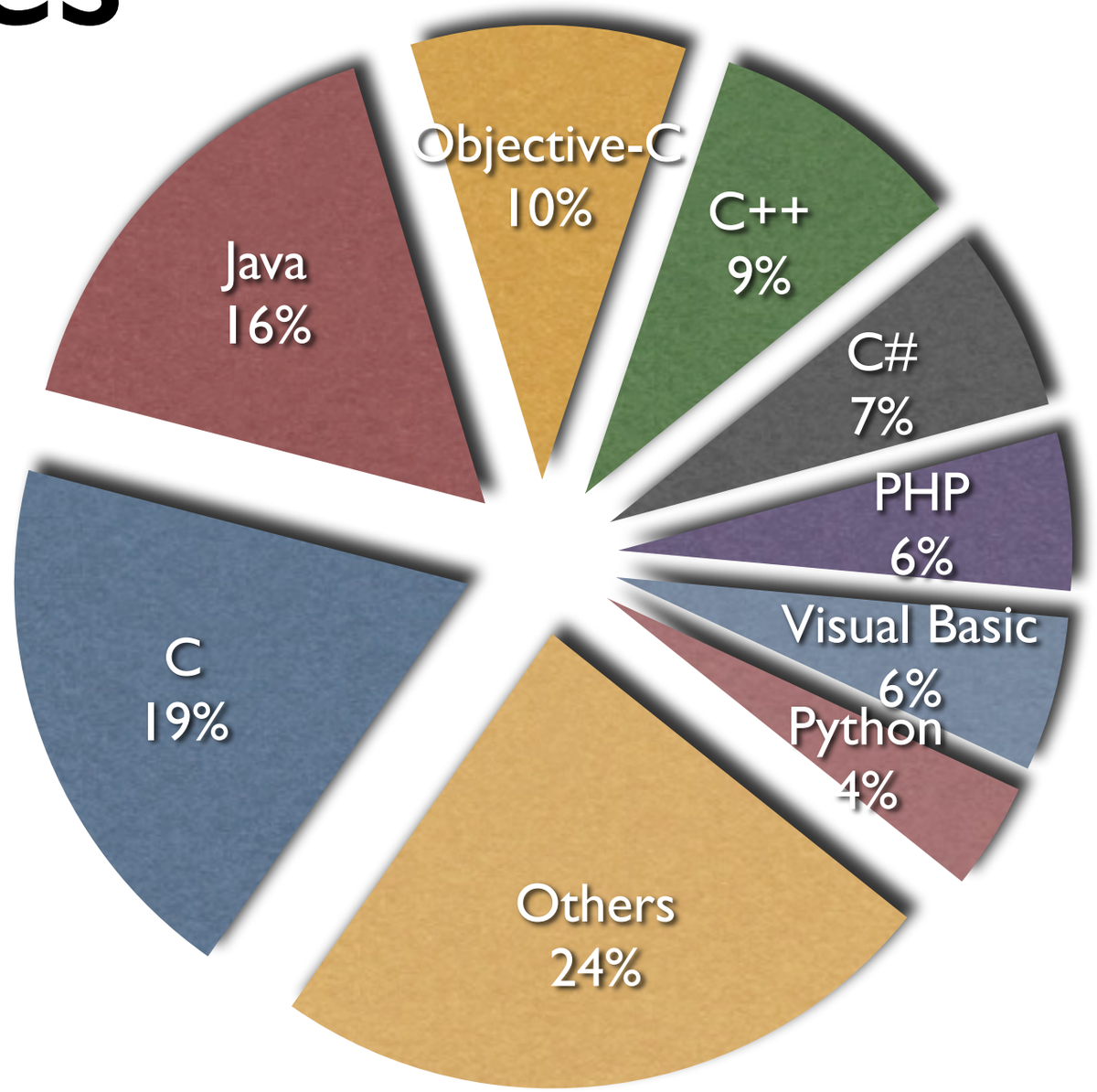
What is Objective-C

- An object-oriented language
  - *"..focused on simplicity and the elegance of object oriented design..."*
  - A strict superset of ANSI C
    - Very Different (and somewhat simpler) to C++
    - Exploits a number of object oriented principles
      - Inherits many principles from Smalltalk
    - Used to develop the Cocoa API Framework
- A variant for C++ also exists
- Originally used within NeXT's NeXTSTEP OS
  - Precursor to Mac OS X

Popularity compared to other languages

- Objective-C is one of the fastest growing languages (in terms of uptake) in the last few years
  - Ranked 43rd in 2007
  - Ranked 3rd in 2012 after C and Java

- TIOBE index (opposite) based on estimates of the *most lines of code written* per year



TIOBE Programming Community Index for September 2012

http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Dynamic Runtime

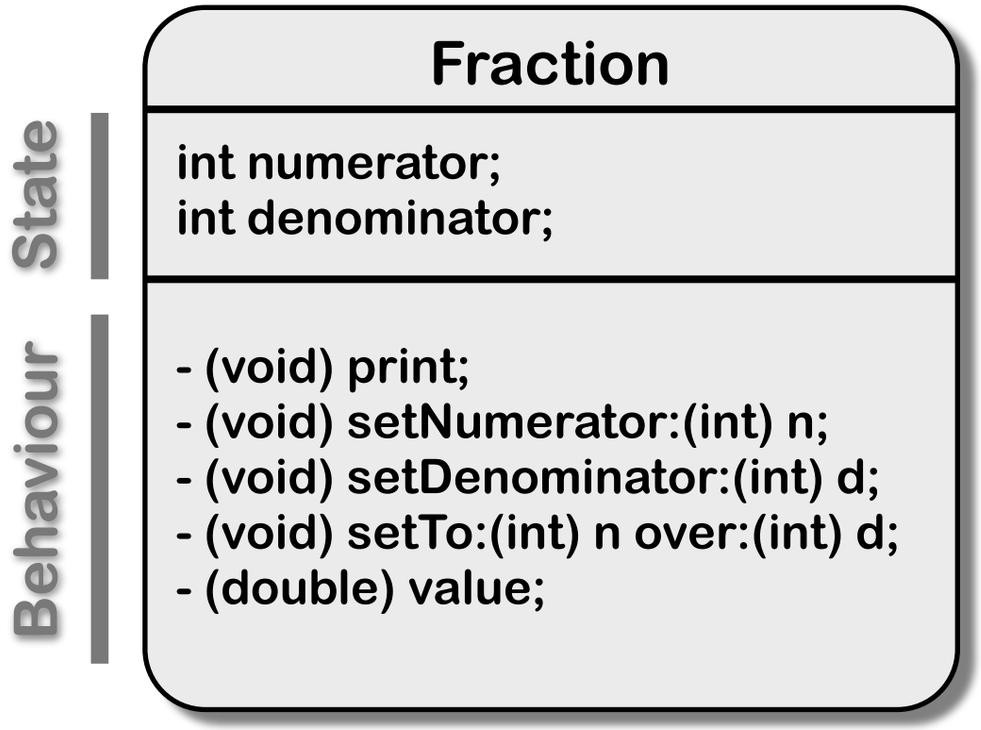
- Object Creation
  - Everything is allocated from the **heap**
  - No **stack** based objects!!

- Message Dispatch (Dynamic Binding)
  - Everything is looked up and dispatched at runtime (not compile time)
    - If you send a message to an object, the existence of that object is checked at runtime
    - Differs from C++ and Java, which bind methods at compile-time

- Introspection
  - A "thing" (class, instance, etc) can be asked at runtime what type it is
    - Can pass anonymous objects to a method, and get it to determine what to do depending on the object's actual type

Classes and Objects

- Classes and instances are **both** objects
- Classes declare state and behaviour
  - State (data) is maintained using instance variables
  - Behaviour is implemented using methods
- Instance variables typically hidden
  - Accessible only using *accessor* (i.e. getter and setter) methods



Method Syntax

- Methods are defined as Instance or Class methods
  - Methods called on a **Class** prefixed with ("+")
    - Affect the class object, not its instances
    - Often used to create instances, such as calling alloc, or using factory classes
- Methods called on **Instances** prefixed with ("-")
  - Operate on the instance receiving the message

Fraction Methods

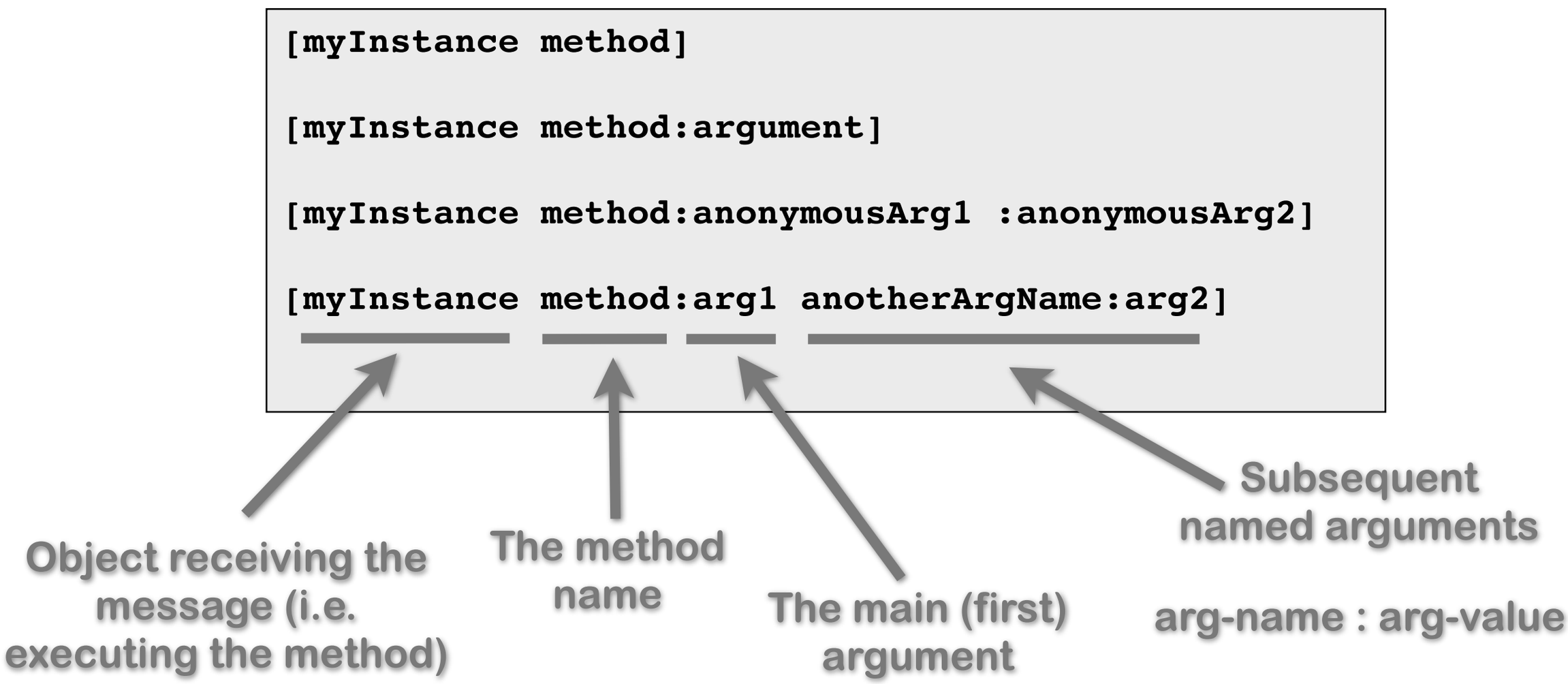
```
+ (id) alloc;
+ (int) fractionInstanceCount;
+ (id) newFraction:(double) value;
```

```
- (void) print;
- (void) setNumerator:(int) n;
- (void) setDenominator:(int) d;
- (void) setTo:(int) n over:(int) d;
- (double) value;
```

Message Syntax

- A square brace syntax is used

```
[myInstance method]

[myInstance method:argument]

[myInstance method:anonymousArg1 :anonymousArg2]

[myInstance method:arg1 anotherArgName:arg2]
```



Examples

```
// Assume that we have a Person Class defined

Person *voter = [[Person alloc] init];

[voter castVote];                   // Do something
int theAge = [voter age];           // Get something (getter)
[voter setAge:16];                  // Define something
(setter)


if ([voter canLegallyVote]) {
        // allow the user to submit a ballot
}


// Send several arguments to a method
[voter registerForElection:@"Wirral" party:@"Labour"];


// Embed the returning value of one message in another
char *name = [[voter spouse] name];
```

Unicode vs ASCII

C traditionally denotes strings as an array of ASCII characters, represented in double quotes; i.e.

"Hello ASCII folk!"

Unicode characters are bigger than ascii characters, and support different alphabets. Unicode strings are represented by double quotes with a preceding "@" character; i.e.

@"Hello Unicode folk!"

@interface Section

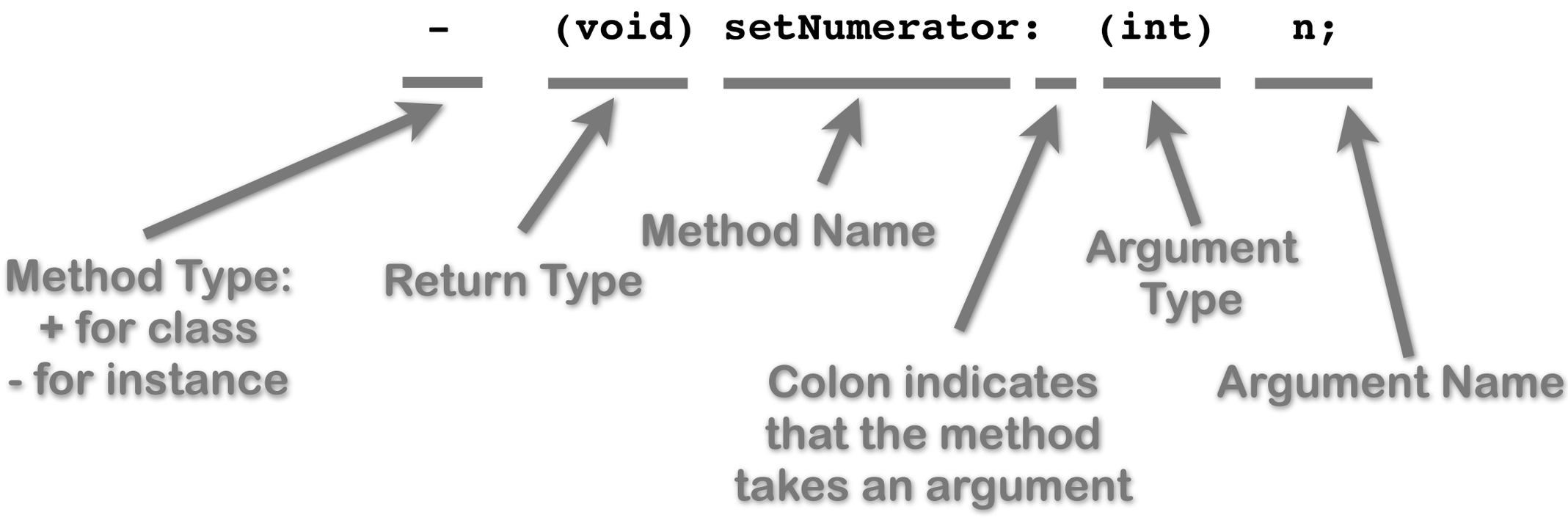
```
@interface NewClassName: ParentClassName {
    memberDeclarations;
}

methodDeclarations;
@end
```

- This is where the class is defined
  - Often defined in a header file (suffix ".h", as in C)
  - Encapsulated within the @interface and @end
    - The instance variables (ivars) are given
      - These appear within the curly braces, and can vary in scope
    - The method declarations are then listed
      - The arguments and return types are specified
  - No code is defined, just the name, return type and arguments of each method

- By convention, class names start with upper case letters

Class and Instance Method Syntax

- Each method declaration consists of:
  - a name
  - a return type
  - an optional list of arguments (and their data or object types)
  - an indicator to determine if the method is a class or instance method
- The syntax is



@implementation Section

```
@implementation NewClassName;
methodDefinitions;
@end
```

- This is where the methods for the class are declared
  - Code is in a source file (suffix ".m")
  - Encapsulated within the @implementation and @end
    - Normally, all the methods defined within the class interface are implemented here

- The class can reference itself or its parent class
  - **self** - this is the instance itself
  - **super** - this is the parent class
    - Useful to call class methods of its parent

```
[self setNumerator 5];
...
[super dealloc];
```

@implementation section

- Accessor methods are defined here
  - A convention should be used, where setters are prefixed with the string "set"
- No non-class related code should appear in this section

Creating Getters and Setters

Accessor methods can be constructed automatically using the @property and @synthesise statements. We'll cover these later.

```
// ---- @implementation section ----
@implementation Fraction;
- (void) print {
    printf(" %i/%i\n", numerator, denominator);
}
- (void) setNumerator: (int) n {
    numerator = n;
}
- (void) setDenominator: (int) d {
    denominator = d;
}
@end
```

Dot Syntax

- Objective-C 2.0 introduced dot syntax
  - Convenient shorthand for invoking accessor methods

```
float height = [person height];
float height = person.height;

[person setHeight:newHeight];
person.height = newHeight;
```

- Follows the dots...

```
[[person child] setHeight:newHeight];

// exactly the same as
person.child.height = newHeight;
```

- Essentially provides a shorthand
  - Assumes a setter naming convention (**setX**) etc
  - Dot syntax is automatically converted into the brace syntax

UNIVERSITY OF LIVERPOOL

Memory Management within Objective-C

Objective-C and the Foundation Framework

Managing Memory

- There are two approaches to managing memory
  - **ARC: Automatic Reference Counting**
    - Introduced in iOS5
    - Compiler evaluates the requirements of your objects, and automatically inserts memory management calls at compile time.
  - **MRR: Manual Retain-Release**
    - Used prior to iOS5, but still available
    - Developer explicitly inserts memory management calls (*retain* and *release*) into the code
- Both approaches assume the developer will "allocate" new memory for new objects
  - Both approaches use retain counts for each object
  - However, ARC takes responsibility for retaining or releasing objects at compile time.

Creating objects

- Objective-C 2.0 defines new functions to allocate memory and deallocate heap memory
- Once a class has been defined, it needs instantiating

```
...
// Create an instance
// of the class Fraction
myFraction = [Fraction alloc];
myFraction = [myFraction init];

// Alternative syntax
myF2 = [[Fraction alloc] init];
...
```

- This involves the creation of a new instance and the allocation of heap memory for that object
- The "`+ alloc`" method (inherited from the Object class) is a class method that allocates the necessary memory and returns a new instance
- All classes should also implement an "`- init`" method (also defined in the Object class)
  - This is responsible for performing any initialisation within the new instance
  - Note that the init method could itself change the memory location of the instance, and hence you should set your variable to its return value!!!

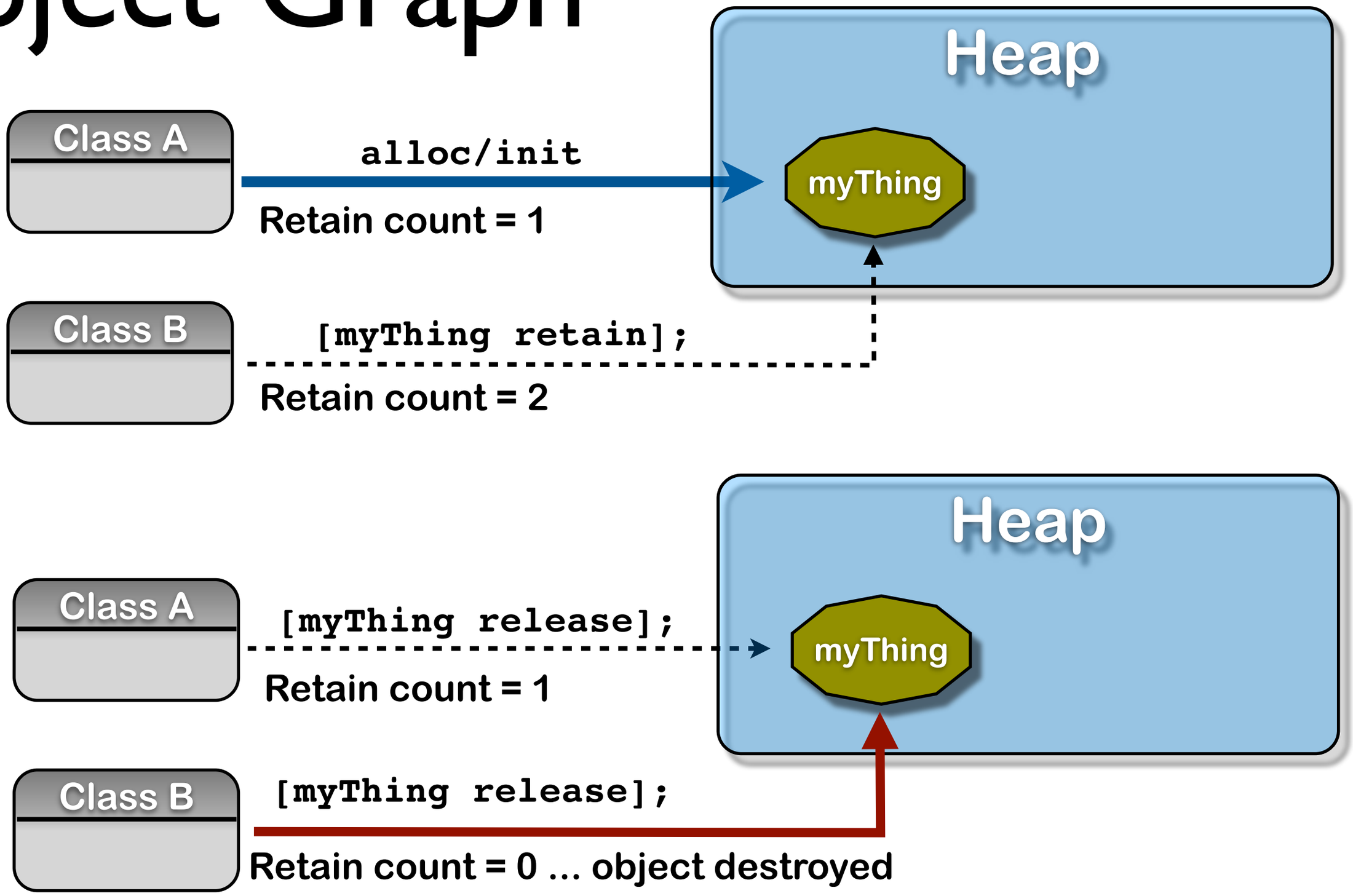
Reference Counting in Memory Management

- Every allocated object has a retain count
  - Defined on **NSObject** (in the Foundation Framework)
  - As long as retain count is > 0, object is alive and valid

- When objects are shared (i.e. owned by more than one variable), then the retain count should track this
  - **`+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

Object Graph



MRR vs ARC

- When using MRR, if the instance is shared, then it should be explicitly retained using the **`retain`** method call
  - This is important when objects are shared, or when object ownership passes from one parent to another.
  - When the object is no longer needed, the memory can be freed using the **`release`** method call
  - Ownership may need managing using **`autorelease`**

- When using ARC, you don't need to worry about this!

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)castVote;
@end
```

The object name requires memory management. It is owned by Person.

- may be returned by the getter accessor
  - *Do you return a copy of the name, or a reference to the stored name?*

- may be changed by the setter accessor
  - *Do you store a copy of the name, or retain a reference to the name owned by the calling method?*

Shared Object Ownership

```
#import "Person.h"
// Using the MMR memory model

@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
```

If a new name string is sent to setName:

- the previous string is released (i.e. it is not longer needed)
- the new string retained (i.e. this string will also be owned by Person).

Copied Object Ownership

```
#import "Person.h"
// Using the MMR memory model

@implementation Person

- (NSString *)name {
    return name;
}

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

        name = [newName copy];
        // name now has a retain count of 1, we own it
    }
}

@end
```

This version makes a copy of the string in the setter, rather than keeping the argument string.

- The ownership of the argument string is not changed. It remains the responsibility of its previous owner...
- The setter now has its own copy of the string and is responsible for it

Returning a newly created object

- In some cases, objects may be passed with no clear or obvious ownership
  - Hence no responsibility to clean up

```
- (NSString *)fullName {
    NSString *result;
    result = [[NSString alloc] initWithFormat:@“%@ %@”,
                                   firstName, lastName];
    return result;

    // result was allocated from the heap...
    // ... but now it has been passed to the calling
    // method, it is too late for fullName to manage it!
}
```

- In this case, result is leaked...!
  - result is passed as an allocated object with no owner

Returning a newly created object

- Can't release result before it is returned
  - Yet, after **`return`**, the method looses access to the object

```
- (NSString *)fullName {
    NSString *result;
    result = [[NSString alloc] initWithFormat:@"%@ %@",
                                    firstName, lastName];
    [result autorelease];
    return result;

    // result was allocated from the heap...
    // ... and is now managed by autorelease :)
}
```

- result will be released some time in the future (not now)
  - caller can choose to **`retain`** it to keep it around!

Autorelease (MRR)

- Calling **`-autorelease`** flags an object to be sent release at some point in the future
  - Let's you fulfil 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

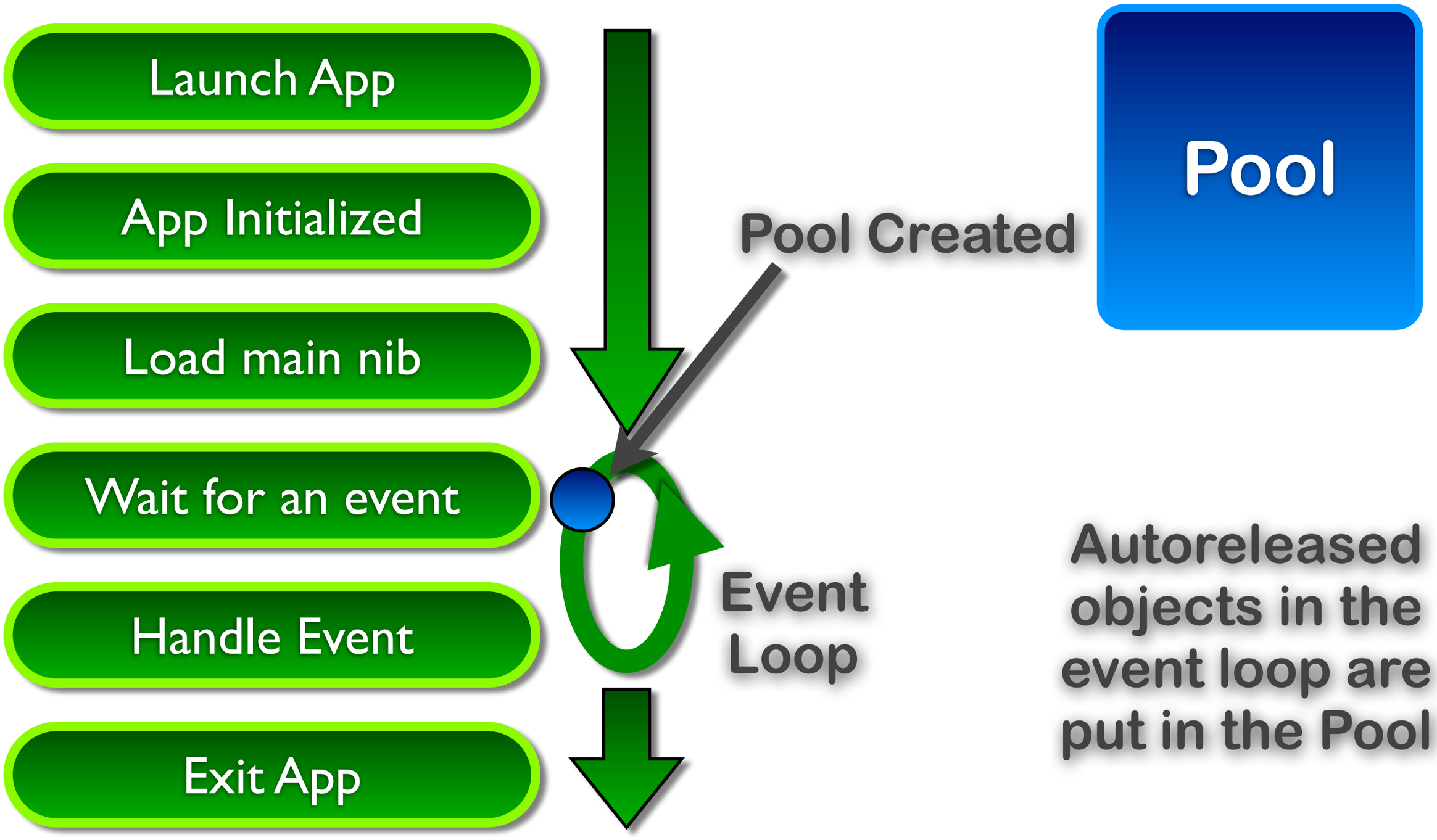
How does -autorelease work???

1. Object is added to current autorelease pool

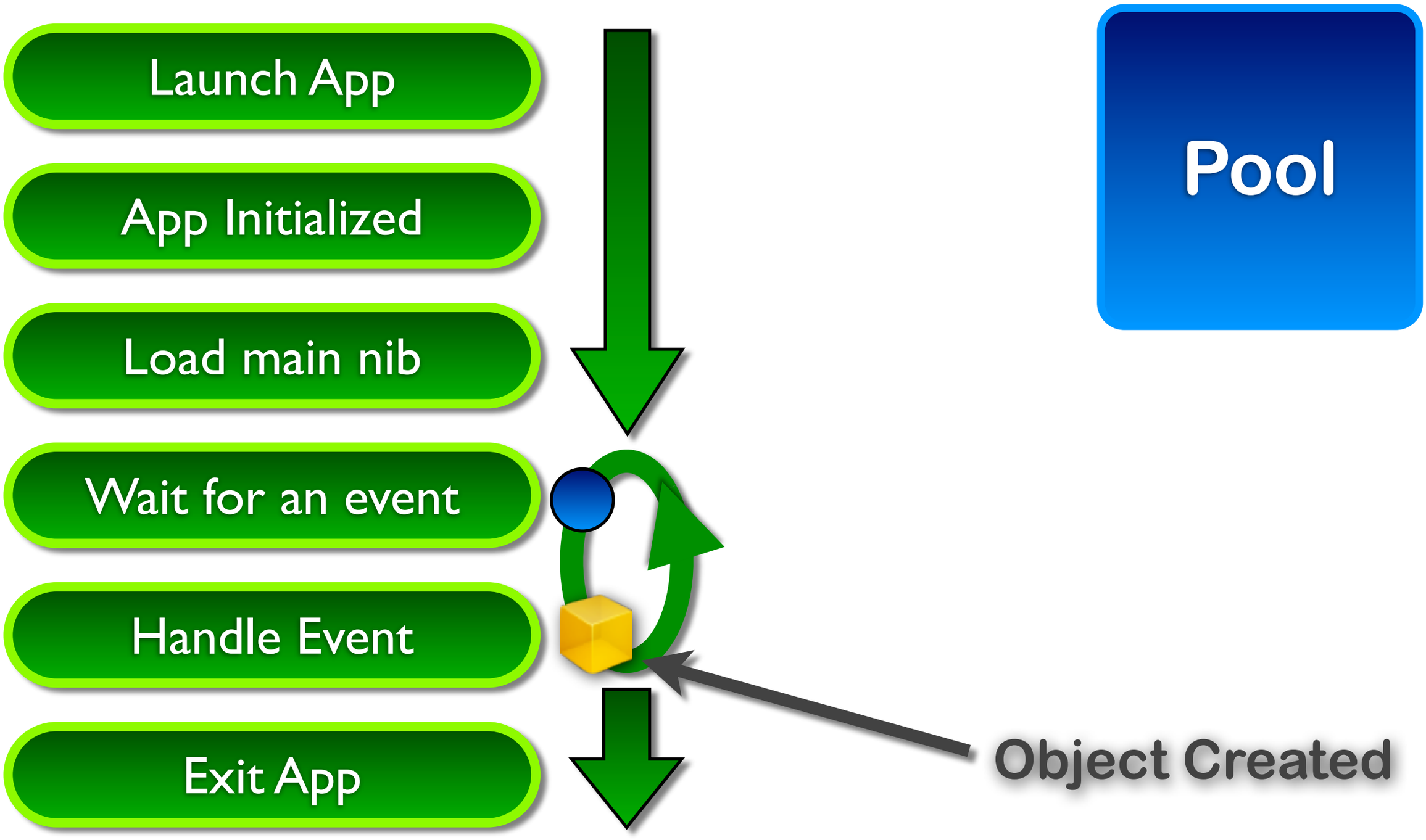
2. Autorelease pools track objects scheduled to be released
   - When the pool itself is released, it in turn sends the -release message to all its objects

3. UIKit automatically wraps a pool around every event dispatch
   - Important for event driven GUI programming
   - However, new threads will need their own pool creating

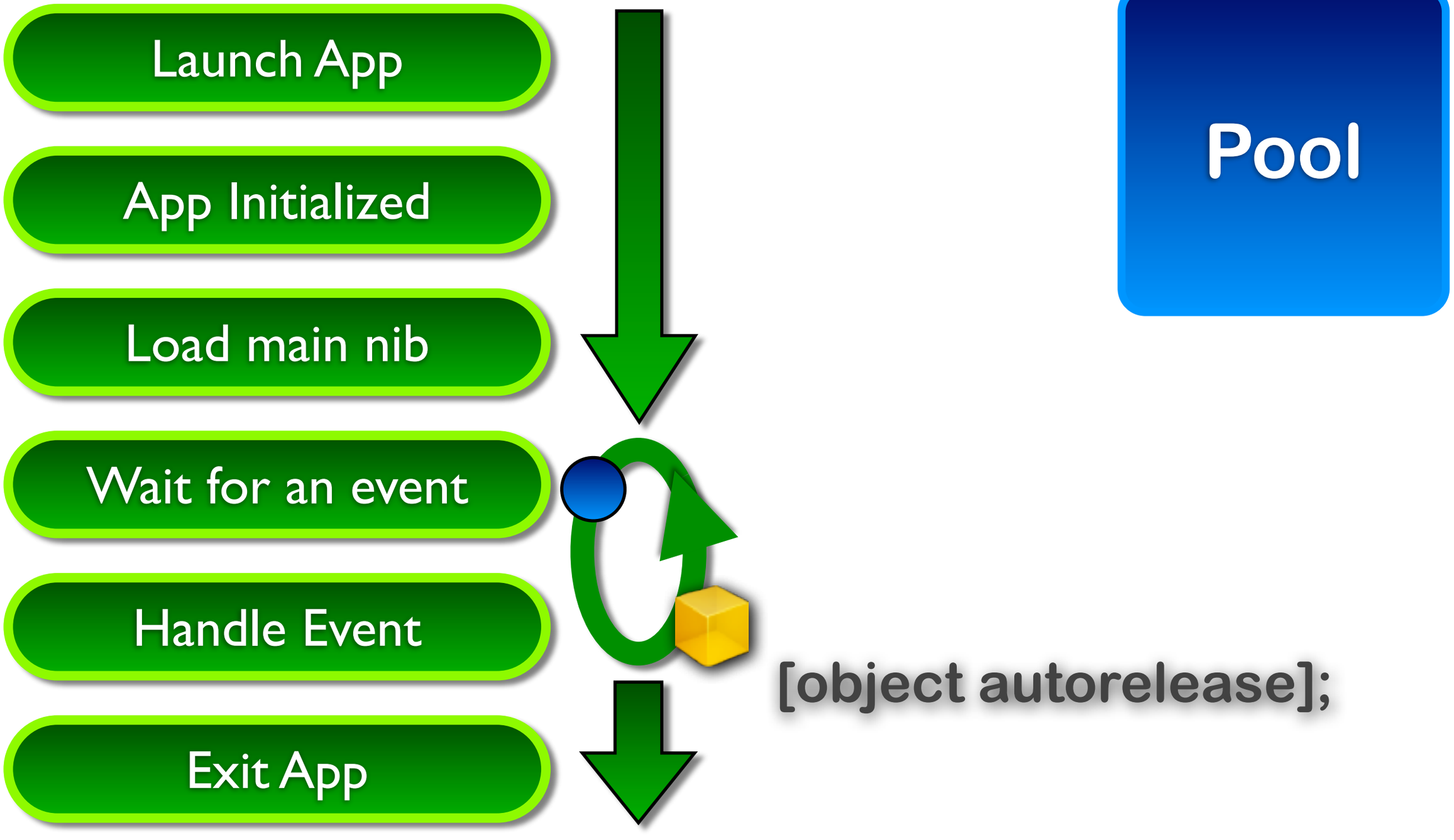
App Lifecycle



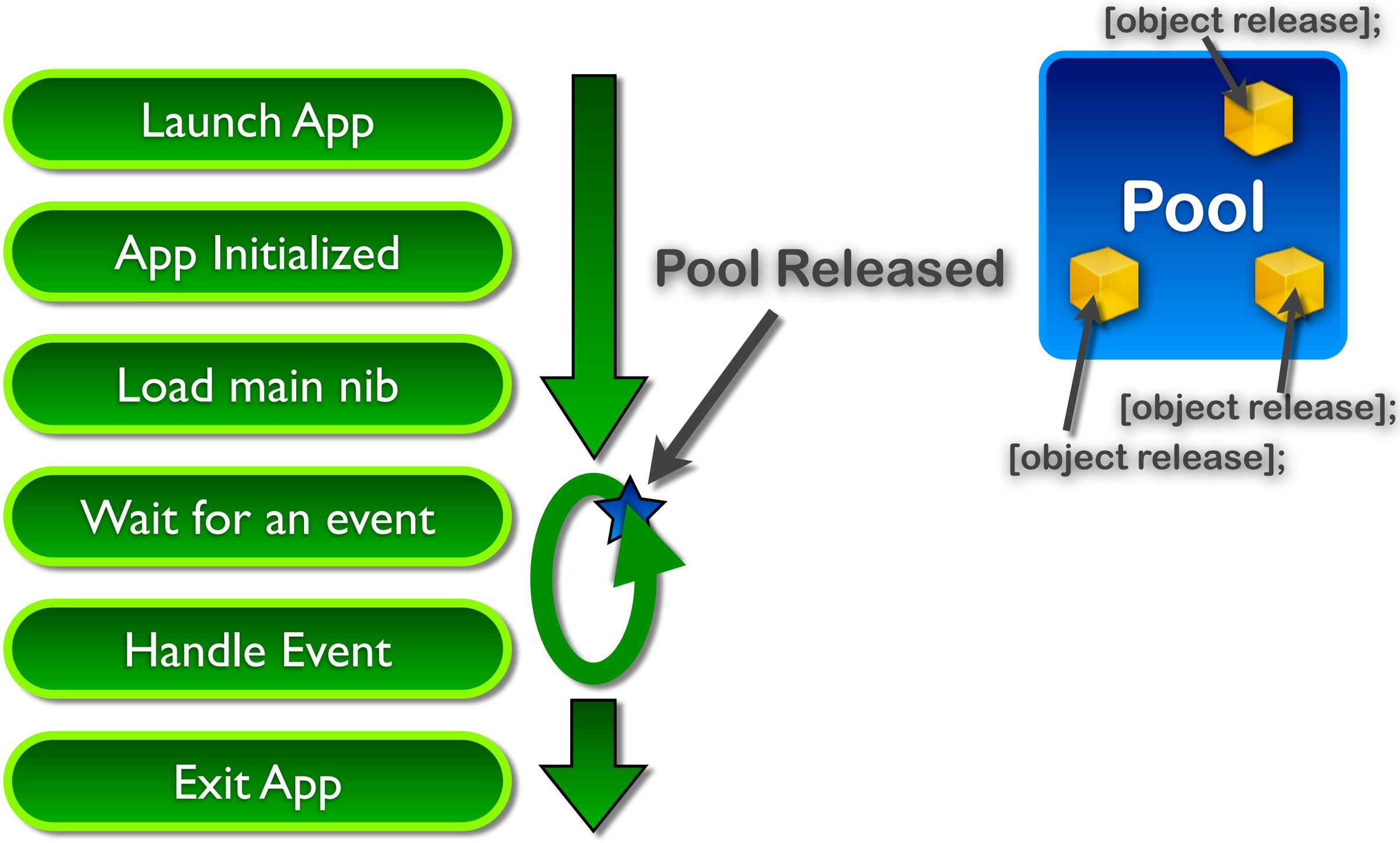
App Lifecycle



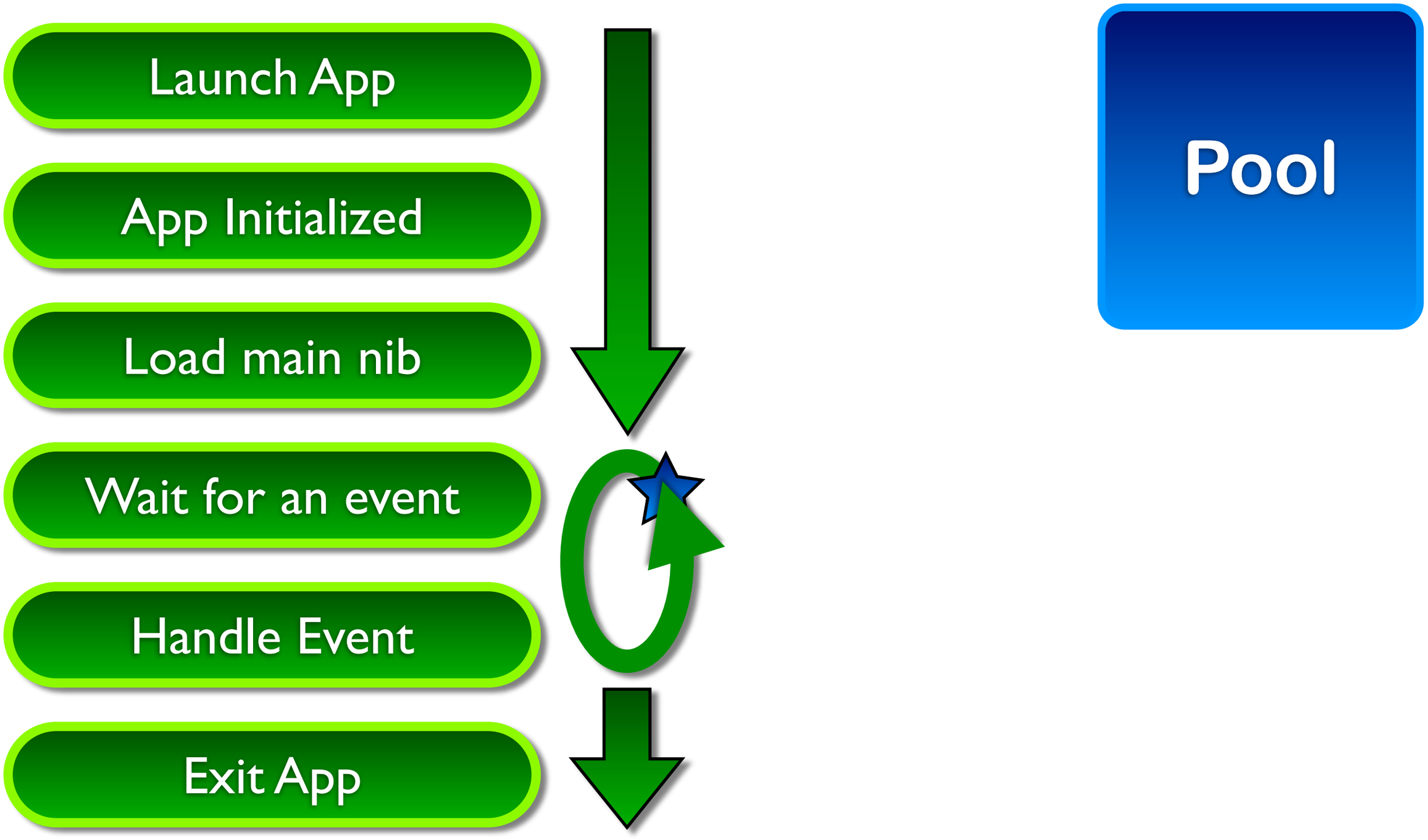
App Lifecycle



App Lifecycle



App Lifecycle



Creating an Autorelease Pool

- An autorelease pool is automatically created in main, and released by the event loop

```
int main(int argc, char *argv[])
{
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil,
                NSStringFromClass([TestSplitViewAppDelegate class]));
    }
}
```

- However, it can be desirable (or necessary) to create your own:
  - When creating new threads/NSOperations etc.
    - Autorelease pool is thread specific
    - Always create one for a new thread
  - When using large (nested) loops
    - Might be desirable to explicitly maintain and release a pool, to avoid using up memory in a single event loop

Autorelease Example

```
// Creating over 5000 strings... in 50 string chunks!
for (int i=0; i<100; i++) {
    @autoreleasepool {
        NSString *str = [NSString stringWithFormat:@"%i:", i];
        for (int j=0; j<50; j++) {
            str = [str stringByAppendingString:@"."];
        }
        NSLog(@"Created String %@", str);
        // Have created 50 strings in the pool
    }
    // pool is released, releasing the strings as well!
}
```

```
// Main method in a new thread
- (void)main {
     @autoreleasepool {

         if (self.isCancelled)
             return;
         ...
    }
}
```

NSAutorelease vs autorelease blocks

Pre-ARC, a pool would be allocated, and explicitly released.

```
NSAutoreleasePool *pool =
    [[NSAutoreleasePool alloc] init];

// Code benefitting from a
// local autorelease pool.

[pool release];
```

However, this cannot be used in ARC, so autorelease blocks are used instead.

```
@autoreleasepool {

    // Code benefitting from a
    // local autorelease pool.

}
```

Blocks are also more efficient than using an instance of a pool

Object Ownership and Factory Classes

- Ownership of objects (and hence release responsibility) is one of the challenges in memory managed code
  - If some class returns an object, whose responsibility is it?

```
NSString myString [[NSString alloc] initWithString:@"I own this"];

// Need to clean up myString using release with MMR
// No need to do anything with ARC
```

- Various conventions exist for determining ownership
  - e.g. when using **factory classes**

```
NSString theirString [NSString stringWithString:@"I don't own this"];

// This factory-class generated string theirString will
// be managed later using the autorelease pool
```

Object Lifecycle Recap

- Objects are created using **`alloc`**, **`copy`** or using a factory method
  - Objects begin with a retain count of 1
- When the retain count reaches 0, the object is freed automatically

- **When using MRR**
  - Call **`retain`** to increase the retain count (e.g. when sharing an object)
  - Call **`release`** when the object is no longer needed by a variable

- **When using ARC**
  - Things will happen automatically

UNIVERSITY OF LIVERPOOL

Properties and accessor method synthesis

Objective-C and the Foundation Framework

Properties

- Provide access to object attributes
  - Shortcut to implementing getter/setter methods
  - Instead of declaring "boilerplate" code, have it generated automatically
    - Specify @properties in the header (*.h) file
    - Create the accessor methods by @synthesizing the properties in the implementation (*.m) file
- Also allow you to specify:
  - read-only versus read-write access
  - memory management policy

Properties

- Defined within the header file, and determines:
  - The iVar that the accessor manages
  - The name of the accessor
    - by default, the same as the ivar, but can be specified as a different name
  - The way in which data is owned

```
// Fraction.h
@interface Fraction : NSObject {
    int numerator;
    int denominator;
}

- (void) setNumerator: (int) n;
- (void) setDenominator: (int) d;
- (int) numerator;
- (int) denominator;
- (double) convertToNum;
@end
```

```
// Fraction.h (using properties)
@interface Fraction : NSObject {
    int numerator;    // optional
    int denominator;  // optional
}

@property int numerator;
@property int denominator;
@property (readonly) double convertToNum;
@end
```

Synthesising Properties

```
// Fraction.m
@implementation Fraction

- (void) setNumerator: (int) n {
    numerator = n;
}
- (void) setDenominator: (int) d {
    denominator = d;
}
- (int) numerator {
    return numerator;
}
- (int) denominator {
    return denominator;
}

- (double) convertToNum {
    if (denominator != 0)
        return (double)
          numerator / denominator;
    else
        return 1.0;
}
@end
```

- Performed in implementation file, and generates the accessor methods
- If the synthesize operation isn't used, then the corresponding method should be defined by hand

```
// Fraction.m (using properties)
@implementation Fraction

@synthesize numerator;
@synthesize denominator;

- (double) convertToNum {
    if (denominator != 0)
        return (double)
          numerator / denominator;
    else
        return 1.0;
}
@end
```

Property Attributes

- Properties Read-only versus read-write

```
@property (readwrite) int numerator; // rarely seen
@property int numerator; // read-write by default, same as above

@property (readonly) double convertToNum; (getter=gettername)
```

- Property attributes with different names to ivars
  - Change the getter name **`(getter=gettername)`** ...
  - ... or the setter name **`(setter=settername)`** or both!!!

Property Attributes

- MMR Policies (only for object properties)

```
@property (assign) NSString *name; // value assignment (e.g. for int float etc)
@property (retain) NSString *name; // retain called
@property (copy) NSString *name;  // copy called
```

- Automatic Reference Count Policies

```
@property (assign) NSString *name; // value assignment (e.g. for int float etc)
@property (strong) NSString *name; // reference counting should be managed
@property (weak) NSString *name; // reference will not keep the object alive
```

http://developer.apple.com/library/ios/#releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html

Property Attributes

- Atomicity - for threaded accessors
  - Properties are atomic by default, to provide robust access in a multithreaded environment
  - For retained or copied properties that are atomic, then locking is used
  - For nonatomic properties, the value is simply returned directly

```
@property (nonatomic, strong) UIWindow *window;
@property (nonatomic, strong) FractionPickerViewController *viewController;
```

UNIVERSITY OF LIVERPOOL

Recent Updates to Objective-C

Objective-C and the Foundation Framework

Objective-C Updates

- **Explicit definition of instance variables**
  - Previously, these would be declared in the header file
  - Properties would then be defined for them
    - This resulted in an iVar and a getter method with the same name

```
@interface ViewController : UIViewController {
    NSArray *myStorage;
}

@property (nonatomic, strong) NSArray *myStorage;

@end
```

  - However, this is not necessary, as @synthesise creates this iVar automatically

```
@interface ViewController : UIViewController;

@property (nonatomic, strong) NSArray *myStorage;

@end
```

Objective-C Updates

- **Definition of private instance variables in the implementation file**
  - Previously, these would be exposed in the header file
    - Viewable by other implementation files that include the header
    - Could be protected by using @private

```
@interface ViewController : UIViewController {
    @private
    NSDictionary *namesDictionary;
}

// No property declared here for namesDictionary
@end
```

  - These can now be included after the @implementation element in the implementation file:

```
// ViewController.h

@interface ViewController : UIViewController

@end
```

```
// ViewController.m

#include “ViewController.h”
@implementation ViewController {
    NSDictionary *namesDictionary;
}
```

Objective-C Updates

- **iVar vs Properties**
  - Properties have often been declared as the same as the getters
    - Results in the creation of a method and a variable with the same name
    - If iVars are created automatically, this still occurs when using synthesis
    - Dot notation could be ambiguous

```
self.myString = @"Hello";
// Are we changing the value of iVar myString? (no)
// Or are we calling the method myString: on self? (yes)
```

  - Different names can be generated using @synthesis foo = _foo;
    - Avoids the ambiguity
    - The "underscore" variable is only modified when absolutely necessary
    - Good practice dictates that the getter and setter are always used instead

```
// ViewController.h
@interface ViewController : UIViewController;

@property (nonatomic, strong) NSString *myString;

@end
```

```
// ViewController.m
@implementation ViewController

@synthesise myString = _myString;

@end
```

Objective-C Updates

- **iVar vs Properties (cont)**
  - An important advantage of using getter and setter methods is with memory management
    - When using iVars directly, then management of the previous object pointed to by the iVar needs to be done (see slides on ownership)
    - If property setter is used, then this is handled automatically

```
// ViewController.m
@implementation ViewController

...
    _myString = @"Hello";
    // This can be problematic, as the previous value of _myString could be lost

    [self myString:@"Hello"];
    // This ensures that any releasing (if necessary) is performed

    [self myString:nil];
    // very useful, as it ensures memory is released, and no invalid memory address is kept

....
@end
```

Objective-C Updates

- **Private Categories and class extensions**
  - Like private iVars, these would be exposed in the header file
    - Previously necessary to define forward declarations

```
@interface ViewController (Private)
- (void)sortTable;
- (void)updateValues;
@end
```

- These can now be added to the implementation file, before the @implementation element:

```
// ViewController.m

#include “ViewController.h”

@interface ViewController ()
- (void)sortTable;
- (void)updateValues;
@end

@implementation ViewController

...
@end
```

Objective-C Updates

- **Synthesis ... or not**
  - Prior to Xcode 4.5, properties needed to be synthesised
    - This also allowed the developer to use the "underscore" syntax
  - Post Xcode 4.5 synthesis is performed automatically
    - No need to synthesise properties in the implementation file
    - iVars using the "underscore" syntax are automatically created
- **Forward Declarations**
  - Traditionally, methods had to be declared before they were used
    - Either they appeared before they were called in the code
    - Or a forward declaration would appear in either the header or implementation
  - This is no longer necessary for Objective-C methods in Xcode 4.3 onwards
    - Note - C functions still need forward declarations!!!

UNIVERSITY OF LIVERPOOL

The Foundation Framework (Recap)

Objective-C and the Foundation Framework

Foundation Framework

- Defines a base layer of classes for use in creating applications
  - Supports all the underlying (common) classes
  - Common to all Apple platforms (OS X, iPhone etc)
  - Provides object wrappers for primitive types and collections
  - Provides support for internationalisation and localisation with the use of dynamically loaded bundles
  - Facilitated access to various lower level services of the operating system, including threads, file system, IPC etc

- Practically all applications include this framework

Foundation Framework

- Introduces several paradigms and policies to Cocoa Programming, to ensure consistent and predictable behaviour
  - Object retention and disposal (i.e. memory management)
    - Includes object ownership policies
  - Mutable Class variants
    - Many container or value classes define an immutable class and a mutable subclass
  - Class clusters
    - Abstraction of a class that hides several private subclasses, to handle different objects optimised for different kinds of storage/memory requirements
  - Notifications
    - Allows classes to be kept informed of changes to other classes

Value Objects

- Value Objects encapsulate values of various primitive types:
  - strings
  - numbers (integers and floating point values)
  - dates
  - structures and pointers
- Allows types to be wrapped and used as attributes of other objects, or stored within other objects (e.g. container classes)
  - Supports object behaviour, including introspection, serialisation

Value Objects vs Primitive Data Types

It is often desirable to use C types, structs and pointers for some code fragments, especially when the fragment is algorithmic and doesn't involve objects.

However, when objects are involved, it is better to use Value Objects, or create these objects when necessary from primitive types.

Useful Value Object subclasses

- **NSNumber**: Instantiates objects containing numeric values
  - Such as integers, floating point values and doubles
- **NSDate**: Define objects that represent times, dates, calendar and locales.
  - Supporting classes handle formatting, time zones etc
- **NSString**: Provide object-oriented storage for sequences of Unicode characters
  - Unicode is a coding system which represents all of the world's languages
  - C strings (char *) rarely ever used
  - Support for reading and writing of files, search, string comparison, parsing, etc
  - Without doubt the most commonly used class
- **NSData**: Provides object-oriented storage for streams of bytes
  - Support for reading and writing to files

NSNumber

- In Objective-C, you typically use standard C number types
  - NSNumber is used to wrap C number types as objects
  - No mutable equivalent!
  - Typically better to use floats, ints etc, and only convert when necessary
    - Avoids having to unpack an NSNumber, modify it, and repackage it!
- Common NSNumber methods:

```
+ (NSNumber *)numberWithInt:(int)value;

+ (NSNumber *)numberWithDouble:(double)value;

- (int)intValue;

- (double)doubleValue;
```

Note that these methods are class factories, and thus you don't have to worry about releasing them. More on these later...

Mutable and Immutable classes

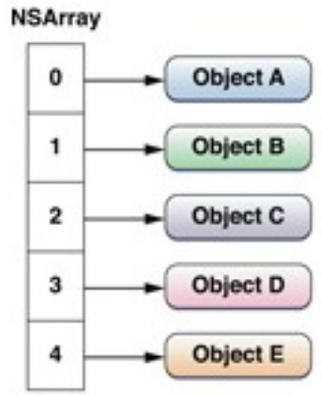
- Many classes have mutable variants
  - **Mutable Object**: an object who's value may be changed at any time
  - **Immutable Object**: an object who's value will remain the same throughout the cannot be changed
- Why mutable and immutable object variants?
  - Mutable objects shared between classes may change unexpectedly
    - When passing a mutable object to another object, it is often desirable to make a copy to avoid it being changed
    - If the object is immutable, then it can be retained without fear of mutation
  - Mutable variants involve additional overhead in process and storage requirements, compared to immutable classes
- A mutable class has the string "Mutable" in it!
  - NSArray vs NS*Mutable*Array, NSString vs NS*Mutable*String, etc
  - ***NSNumbers are not mutable!!!***

Collections

- Collections are objects that store other objects
  - **NSArray** - ordered collection of objects
  - **NSDictionary** - collection of key-value pairs
  - **NSSet** - unordered collection of unique objects
- Each has a mutable variant
  - NSMutableArray, NSMutableDictionary and NSMutableSet
- Collection classes can contain any object
  - Any object stored in a collection is automatically retained
    - If an object is created and then added to the collection class, it can then be released, thus passing ownership (and hence responsibility) to that container
- Common enumeration mechanism

NSArray

- NSArrays manage ordered collections of objects called arrays
  - NSMutable arrays are dynamic - i.e. they can change their size or values, etc.
  - Typically, values at some index are accessed

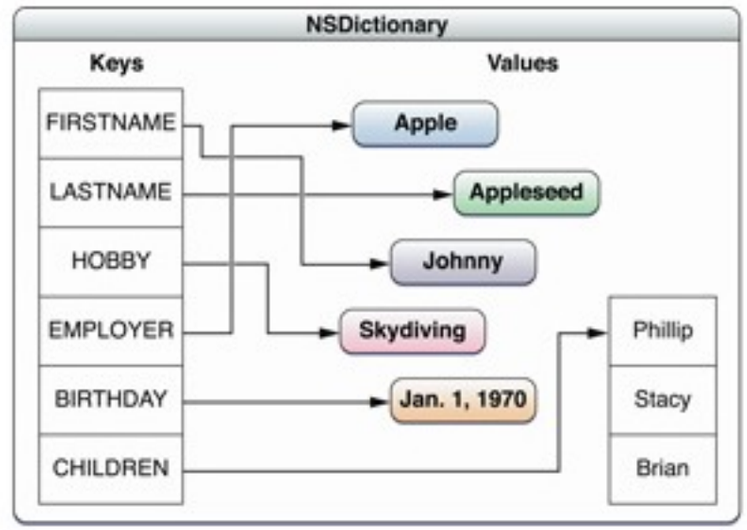



```
NSArray *myArray;
NSDate *aDate = [NSDate distantFuture];
NSValue *aValue = [NSNumber numberWithInt:5];
NSString *aString = @"a string";

myArray = [NSArray arrayWithObjects:aDate, aValue, aString, nil];

id myVal = [myArray objectAtIndex:2];
// myVal is of type NSString with the value “a string”
```

Note that nil marks the end of the list

NSDictionary

- NSDictionaries manage collections of key-value pairs (known as "entries")
  - Each entry consists of:
    - A **value**, which can be any object
    - A **key**, which is unique. This key can be any object, but is most commonly an NSString
  - Objects must never be nil




```
NSDictionary *myDict = [[NSDictionary alloc]
    initWithObjectsAndKeys:
    @"value1", @"key1", @"value2", @"key2", nil];

id myVal = [myDict objectForKey:@"key2"];
// myVal is of type NSString
// with the value "value2"
```

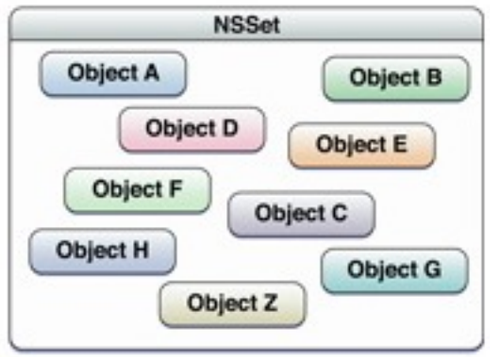
Empty values in Dictionaries

If you need to represent a NULL value in a dictionary, use NSNull.

This is a singleton object used to represent NULL in collections

NSSet

- NSSets manage unordered objects
  - Can be used as an alternative to arrays when the order is not important
    - In this case, membership testing is faster with sets than arrays
    - Good for maintaining collections of items, when membership, or intersection with other set collections is required
  - NSCountedSet maintains a count for each object in the set




```
NSSet *mySet;
NSData *someData = [NSData dataWithContentsOfFile:aPath];
NSValue *aValue = [NSNumber numberWithInteger:5];
NSString *aString = @"a string";

mySet = [NSSet setWithObjects:someData, aValue, aString, nil];

id myVal = [mySet anyObject];
// myVal will correspond to one of the three objects in the set
```

Fast Enumeration

- Cocoa supports three forms of enumeration through a collection
  - **Fast enumeration**
  - Block-based enumeration
  - Using the NSEnumerator class
- Fast enumeration is preferred as it has the following benefits:
  - The enumeration is more efficient than using NSEnumerator directly.
  - The syntax is concise.
  - The enumerator raises an exception if you modify the collection while enumerating.
  - You can perform multiple enumerations concurrently.

Fast Enumeration Example

```
NSString *element ;
for (element in someArray) {
    NSLog(@"element: %@", element);
}

NSString *key;
for(key in someDictionary){
    NSLog(@"Key: %@, Value %@", key,
            [someDictionary objectForKey:key]);
}

id myVal;
for (myVal in mySet) {
   NSLog(@”Element %@ in the set”, myVal);
}
```

UNIVERSITY OF LIVERPOOL

Questions?

Objective-C and the Foundation Framework