COMP327 Mobile Computing

Session: 2010-2011

Tutorial 4-5 - Objective-C and the Foundation Framework

In these Tutorial Slides...

- These slides introduce you to Objective-C, with a focus on the object-oriented components
  - History of the Language
  - Methods/Classes/Objects
  - Message Passing
  - Polymorphism, Dynamic Binding, and Reflection
- Foundation Classes
  - Memory Management
  - Container classes

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
    - ObjC++
  - Originally used within NeXT’s NeXTSTEP OS
    - Precursor to Mac OS X

History

- Originally developed by Brad Cox and Tome Love in the early 1980
  - Inspired by Smalltalk, a C-compiler was modified to add some of its OOP elements
  - Language was then licensed by NeXT in 1988, and used to develop NeXTstep UI
    - NeXT was created by Steve Jobs after having been thrown out of Apple,
  - NeXT was acquired by Apple in 1996 on his return, resulting in the development of OSX, and the adoption of Objective-C for Cocoa
- Objective-C 2.0 was released by Apple in 2006
  - Improved garbage collection and provided syntax enhancements

What is Objective-C

- A simple language, but introduces some new syntax
- Single inheritance
  - Inherits from only one superclass
- Protocols
  - Introduces the notion of multiple inheritance
  - A pattern that defines a list of methods that should be defined when implementing a class
    - Pre Objective-C 2.0, all protocol methods were mandatory
    - Different to the notion of protocols in Java
      - e.g. if you want to define a table, you would need a delegate to enter data into the table
- Dynamic Runtime
  - Everything is looked up and dispatched at runtime (not compile time)
- (Optionally) Loosely Typed

Object Oriented Recap

- Object Oriented Vocabulary
  - Elements
    - **Class**: defines the grouping of data and code, the "type" of an object
    - **Instance**: a specific allocation of a class
    - **Method**: a "function" that an object knows how to perform
    - **Instance Variable (or "ivar")**: a specific piece of data belonging to an object
  - Principles
    - **Encapsulation**: keep implementation private and separate from interface
    - **Polymorphism**: different objects, same interface
    - **Inheritance**: hierarchical organisation, share code, customise or extend behaviours

Syntax Additions

- ANSI C extended with some syntax extensions
- New Types
  - Anonymous object (`id` type)
    - Used for loosely typing
  - Language Level Constructs
    - Class
    - Selectors (for sending messages to objects)
- Syntax for...
  - ...defining classes
  - ...defining message expressions

Dynamic Runtime

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

Objective-C Example

```
#import <stdio.h>
#import <objc/Object.h>

// ---- @interface section ----
@interface Fraction: Object {
    int numerator;
    int denominator;
}

- (void) print;
- (void) setNumerator: (int) n;
- (void) setDenominator: (int) d;
@end

// ---- @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
```

Define the class and its methods in the interface section. Note that these are instance classes, by the prefix '-'.

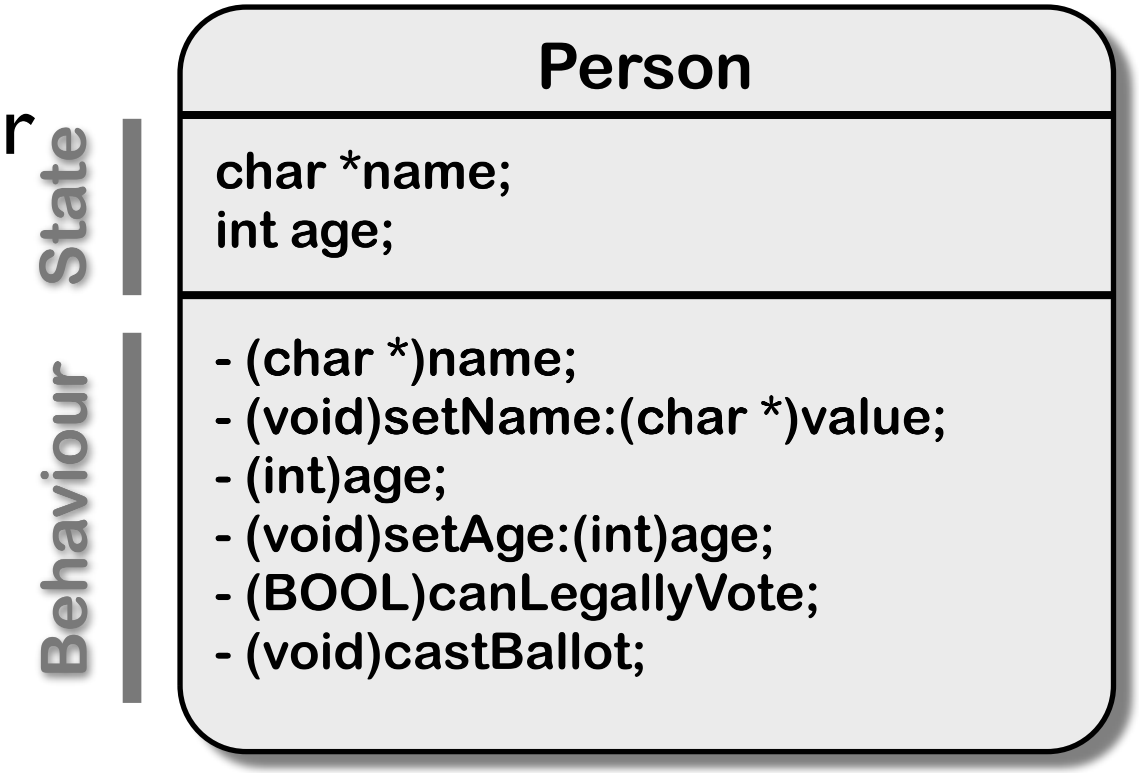
Create the methods within the implementation section

The main code goes in the program section, where a new instance is allocated, used, and then freed.

```
// ---- program section ----
int main(int argc, char *argv[]) {
    Fraction *myFraction;

    // Create an instance of fraction
    myFraction = [Fraction alloc];
    myFraction = [myFraction init];

    [myFraction setNumerator: 1];
    [myFraction setDenominator: 3];

    printf("The value of myFraction is:");
    [myFraction print];
    [myFraction free];

    return 0;
}
```

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



```
char *name;
int age;

- (char *)name;
- (void)setName:(char *)value;
- (int)age;
- (void)setAge:(int)age;
- (BOOL)canLegallyVote;
- (void)castBallot;
```

Messaging Syntax

- Classes and Instance Methods
  - Instances respond to instance methods (“-”)

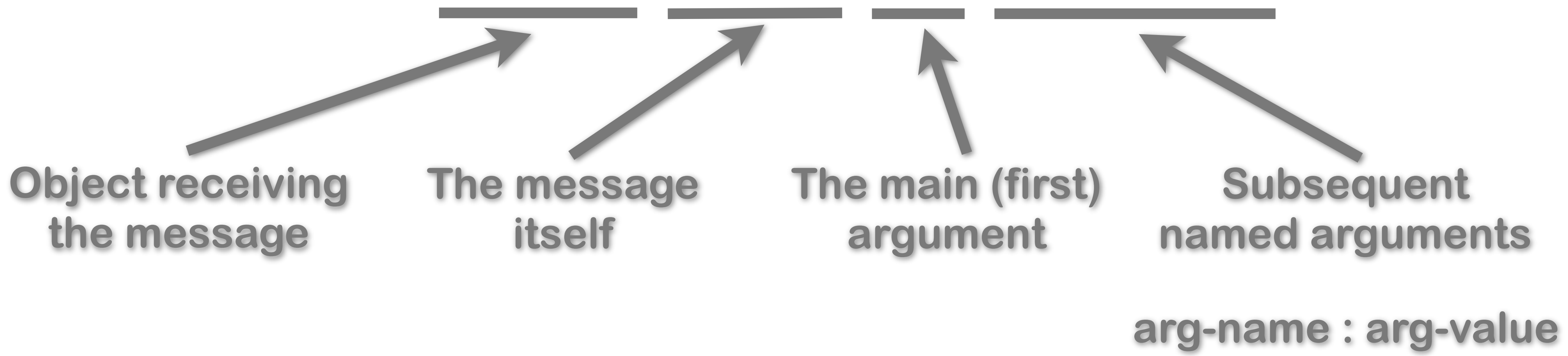
    ```
    - (id)init;
    - (float)height;
    - (void)walk;
    ```
  - Methods called on instances of a class
  - Classes respond to class methods (“+”)

    ```
    + (id)alloc;
    + (id)person;
    + (Person *)sharedPerson;
    ```
  - Methods called on the class itself
    - e.g. modifying class-wide static variables

Message Syntax

- A square brace syntax is used

```
[receiver message]
[receiver message:argument]
[receiver message:arg1 andArg:arg2]
```


Object receiving the message
The message itself
The main (first) argument
Subsequent named arguments
arg-name : arg-value

Message Examples

```
Person *voter; //assume this exists

[voter castVote];

int theAge = [voter age];

[voter setAge:21];

if ([voter canLegallyVote]) {

    // do something voter-y

}

[voter registerForElection:@"Wirral" party:@"Labor"];

char *name = [[voter spouse] name];
```

Terminology

- Message expression

```
[receiver method: argument]
```

- Message

```
[receiver method: argument]
```

- Selector

```
[receiver method: argument]
```

- Method
  - The code selected by a message

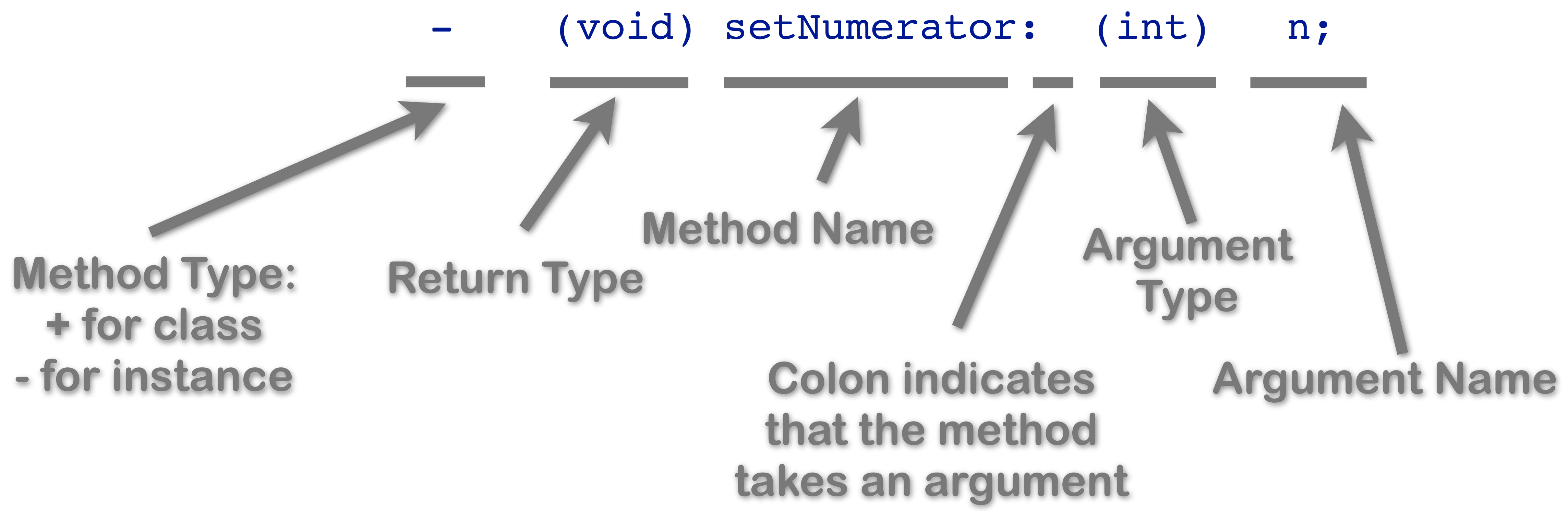
@interface Section

```
@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, but using the Objective-C syntax for methods, and not the C syntax for functions
  - 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


- (void) setNumerator: (int) n;
Method Type: + for class - for instance
Return Type
Method Name
Colon indicates that the method takes an argument
Argument Type
Argument Name

More on Arguments

- Methods can take **more than one** argument
  - Each argument consists of:
    - a name
    - a type in parentheses
    - a variable

```
- (void) setTo: (int) n over: (int) d;

// Called by sending the message

[myFraction setTo: 3 over: 4]
```

  - The name is used to refer to the argument to make messages readable
    - The first argument does not have an explicit name, as its name is the method name

- Methods **without** argument names
  - Argument names are actually optional
  - This would result in the method set::
    - This can make code unreadable

```
- (void) set: (int) n: (int) d;

// Called by sending the message

[myFraction set: 3 : 4]
```

iVar scope

```
@interface NewClass: Parent {
@private
    memberDeclarations;
@protected
    memberDeclarations;
@public
    memberDeclarations;
}
...
@end
```

- By default, instance variables have a scope that is limited to the methods of that class
    - Variables are also inherited through subclassing
- Three directives can be used to modify the scope of instance variables
    - **`@protected`**
        - This is the default case! Instance variables can be directly accessed by methods of that class and **any** subclass.
    - **`@private`**
        - The instance variables can only be accessed by those methods in the **same** class, but not by subclasses
    - **`@public`**
        - The instance variables can be accessed by other methods as well, by using the '**`->`**' operator; for example using **`fraction->numerator`**

@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
    - Categories can be defined in other files to allow groups of methods to be defined together
      - They also allow classes to be extended, even when the original class source is not available
- 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

- Many of the accessor methods are defined here
    - A convention is often used, where setters are prefixed with the string “set”
    - Objective-C 2.0 introduces a new syntax for properties to simplify the definition of accessors
- No non-class related code should appear in this section

```
// ---- @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
```

Program section

- The program section includes functions that look like regular C functions
  - There should be at least one main function, which is called when the application executes.
- The code can be split across multiple files
- However, this section should not contain additional class methods

```
// ---- program section ----
int main(int argc, char *argv[]) {
    Fraction *myFraction;

    // Create an instance of fraction
    myFraction = [Fraction alloc];
    myFraction = [myFraction init];

    [myFraction setNumerator: 1];
    [myFraction setDenominator: 3];

    printf("The value of myFraction is:");
    [myFraction print];
    [myFraction free];

    return 0;
}
```

Selectors identify methods by name

- A selector has type SEL

```
SEL action = [button action];
[button setAction:@selector(start:)];
```

- Conceptually similar to function pointer

- Selectors include the name and all colons, for example:

```
-(void)setName:(NSString *)name age:(int)age;
```

  - would have a selector:

```
SEL sel = @selector(setName:age:);
```

Working with selectors

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

```
id obj;
SEL sel = @selector(start:);
if ([obj respondsToSelector:sel]) {
    // it responds to this selector, so call it
    [obj performSelector:sel withObject:self]
}
```

- This sort of introspection and dynamic messaging underlies many Cocoa design patterns
  - You will see this in use within Interface Builder!!!

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

Reflection

- Instances can ask questions about themselves
  - Am I a certain class?
  - Do I support a certain method?
- Various methods exist (defined within the Object class) to answer these questions

Method	Question or Action
`-(BOOL) isKindOf:` ***`class-object`***	Is the object a member of *class-object* or a descendent?
`-(BOOL) isMemberOf:` ***`class-object`***	Is the object a member of *class-object*?
`-(BOOL) respondsTo:` ***`selector`***	Can the object respond to the method specified by *selector*?
`+(BOOL) instancesRespondTo:` ***`selector`***	Can instances of the specified class respond to this particular message?
`-(BOOL) perform:` ***`selector`***	Apply the method specified by *selector*.

Class Introspection

- You can ask an object about its class

```
Class myClass = [myObject class];
NSLog(@"My class is %@", [myObject className]);
```

- Testing for general class membership (subclasses included):

```
if ([myObject isKindOfClass:[UIControl class]]) {
    // something
}
```

- Testing for specific class membership (subclasses excluded):

```
if ([myObject isMemberOfClass:[NSString class]]) {
    // something string specific
}
```

Protocols

- A protocol is a list of methods that is shared among classes
  - There is no existing implementation of these methods; rather it is up to the developer to implement them for that class
- A class can choose to *conform to*, or *adopt* a protocol
  - Conforming to more than one protocol is possible
    - Informally similar to multiple inheritance
- Two types of protocol are possible
  - Informal Protocols
    - Specified only in the documentation of the class, but is not stated within source code. A class can then determine if the method exists through reflection, and invoke it if it exists.
  - Formal Protocols
    - Similar to an interface in Java
    - Defined, using the @protocol directive ...

Formal Protocols

```
@protocol Locking
- (void)lock;
- (void)unlock;
@end
```

The protocol is defined somewhere, using the @protocol directive, and may group together a set of related method that should be implemented if the class adopts to the protocol.

A class is then defined to adopt the protocol using the angled bracket notation.

Several protocols could be adopted, by separating the protocols with commas

```
@interface SomeClass : SomeSuperClass <Locking>
@end

@interface AnotherClass : AnotherSuperClass <Locking, Archiving>
// This class adopts two protocols!!!
@end
```

The methods can then be defined in the implementation part of the class

```
id currentObject;
...
if ([currentObject conformsTo: @protocol (Locking)] == YES) {
    // Call lock
    [currentObject lock];
```

An object can later check to see if it conforms to a protocol, using the **`conformsTo`** method

Memory Management

- Objective-C 1.0 defines new functions to allocate memory and deallocate heap memory
- Once a class has been defined, it needs instantiating
  - 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!!!

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

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

Implementing your own -init method

In this case the new init method is defined. We start by calling the parent class's init method before we do any of our own initialisation.

By convention, the return type is id. This is because the method itself could be called by a child class, and setting the return value to a specific class here could cause type checking problems.

NOTE that when calling the parent's init method, we set self to be the returned value, as init may change the address of the instance!

Once the init method has been defined, return self.

```
#import "Person.h"

@implementation Person

- (id)init {
    // allow superclass to initialise its state first
    if (self = [super init]) {
        age = 0;
        status = NEWBORN;

        // do other initialisation...
    }
    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 ones with default values

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

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

Note we have NSString - which is defined in the Foundation Framework, discussed in the next Tutorial Slides

Memory Management

- Once the instance is no longer needed, the memory can be deallocated using the **`free`** method call
  - Once an instance is freed, it can no longer be used.

```
...
[myFraction free];
...
```

- Sometimes, a class instance may also allocate memory
  - This can be done by overriding the free method for that class.
  - However, remember to call the parent's free method once the instance's own variables have been freed

```
-(id) free {
    [myElem1 free];
    [myElem2 free];
    // Now all the sub-elements have been freed
    // the parent method can free the instance
    // ensure you return whatever the parent class method returns
    return [super free];
}
```

Memory Management in Objective C

- Calls must be balanced
  - Otherwise your program may leak or crash
- Objective-C 2.0 and the Foundation Classes introduce reference counting within memory management
  - Each alloc call should be balanced with a release call
- You should never call dealloc directly
  - With only one exception...

	Allocation	Destruction
C	`malloc`	`free`
Objective-C	`alloc`	`free`
Objective-C 2.0	`alloc`	`release / dealloc`

Reference Counting

- Every object has a retain count
  - Defined on NSObject (in the Foundation Framework)
  - 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

Reference Counting in Action

```
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

- Program defensively when managing memory

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

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

- Typically, it is good practice to set pointers to nil when they are not in use

  ```
  person = nil;
  ```

  - Easy to check
  - Avoids memory corruption if contents at the pointer address is mistakenly updated
- Objective-C silently ignores messages to nil

Object Lifecycle Recap

- Objects begin with a retain count of 1
    - Increase with -retain
    - Decrease with -release
- When the retain count reaches 0, the object is deallocated automatically
- You **never** need to call dealloc explicitly in your code
    - The only exception is when redefining the dealloc method to do additional housekeeping
        - In this case, send a message to the superclass via [super dealloc]
- You only deal with `alloc`, `copy`, `retain`, and `release`

Object Ownership

- It is important to know who owns an object
  - Hence who is responsible for its memory!
- Can be mediated through -retain and -release
  - Sometimes, ownership can be ambiguous
    - For this, we have -autorelease
- Also, mutability is an important consideration
  - Are objects retained or copied?
    - Can depend on the programmer’s paranoia...!
    - Typically depends on whether the object is mutable

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
- may be changed by the setter accessor.

Header File

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
```

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).

.m
Implementation File

Object Ownership

```
#import "Person.h"

@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...

Implementation File

Releasing Instance Variables

```
#import "Person.h"

@implementation Person

- (NSString *)name {
    return name;
}

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

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

A newly defined dealloc may need to clean up (i.e. release) any objects before it calls its superclass dealloc method.

Implementation File

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;
}
```

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

Autorelease

- 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

Naming conventions and predicting ownership

- 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

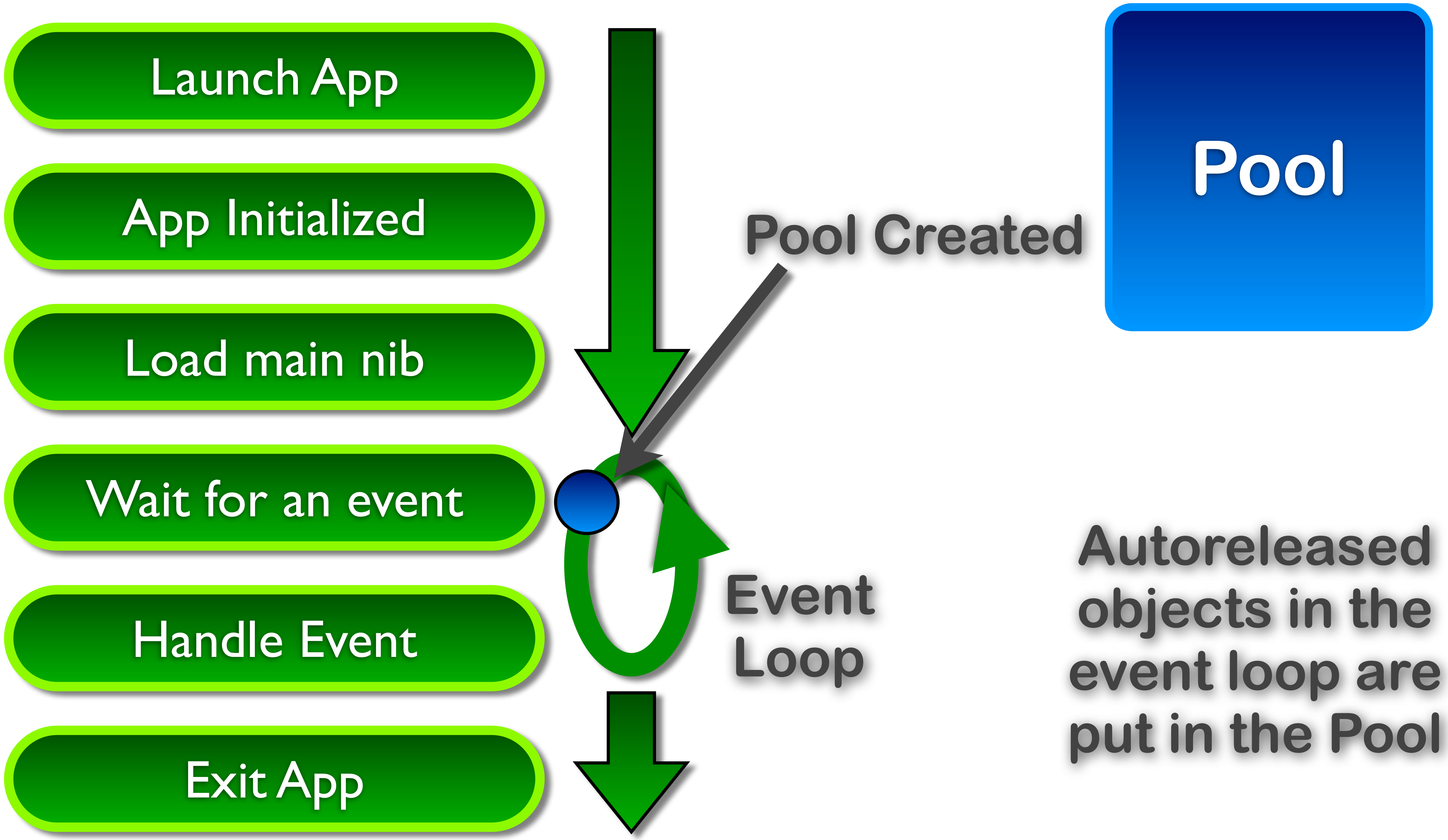
```
NSMutableString *string = [NSMutableString string];
// The method name doesn’t indicate that we
// need to release it ... so don’t!!!
```

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

How does -autorelease work???

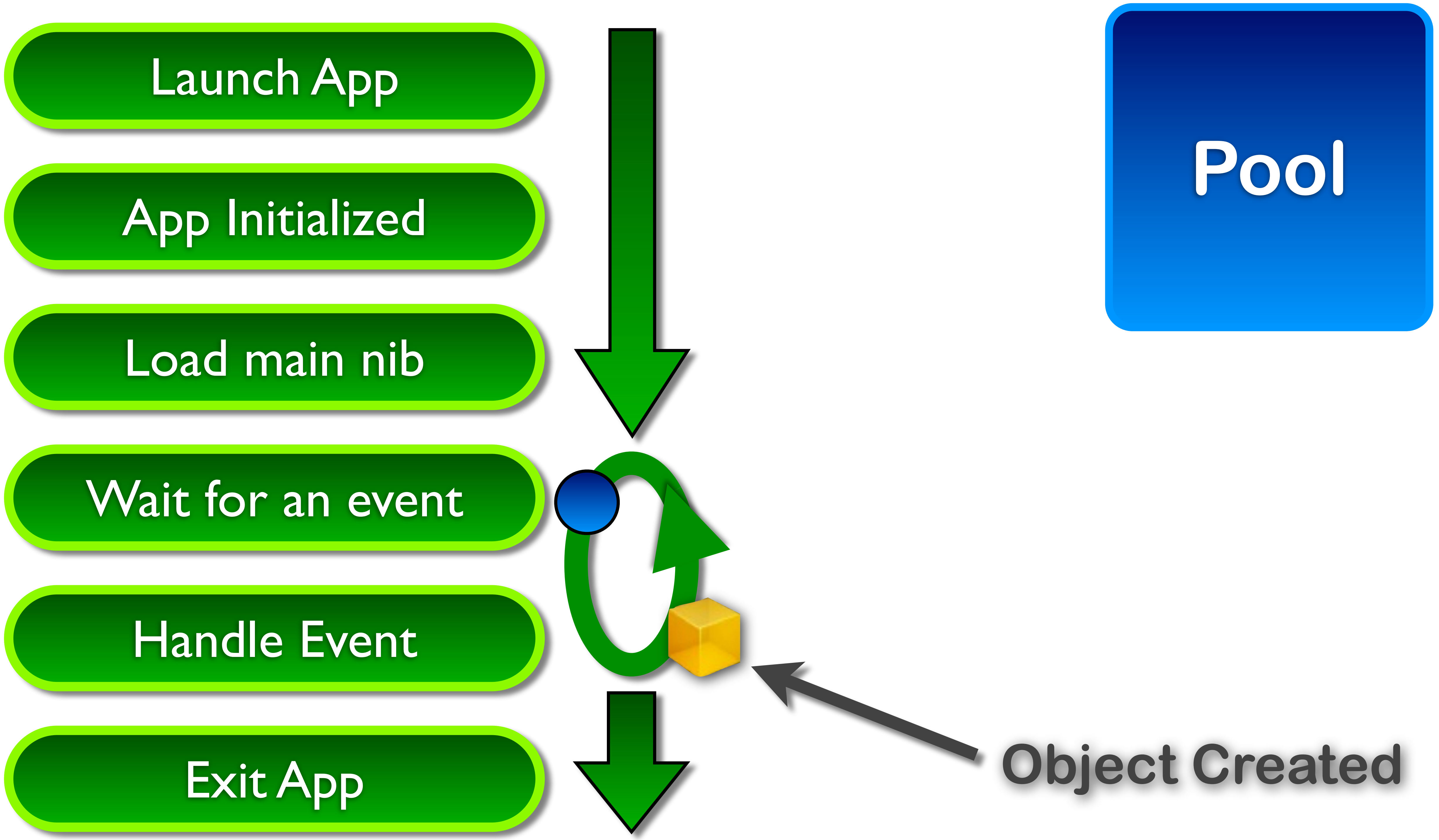
- Object is added to current autorelease pool
- Autorelease pools track objects scheduled to be released
  - When the pool itself is released, it in turn sends the -release message to all its objects
- UIKit automatically wraps a pool around every event dispatch
  - Important for event driven GUI programming

App Lifecycle

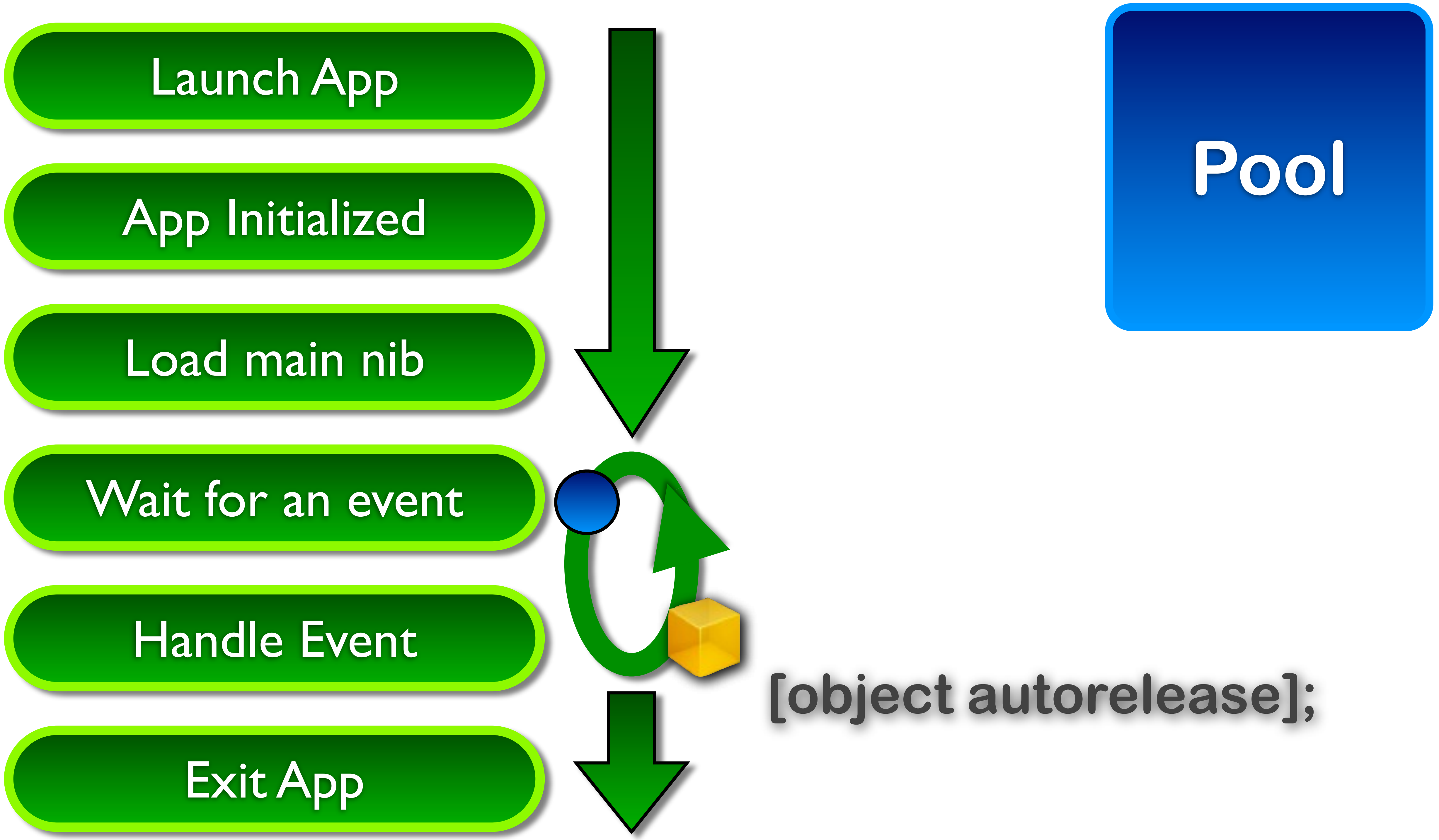


Autoreleased objects in the event loop are put in the Pool

App Lifecycle

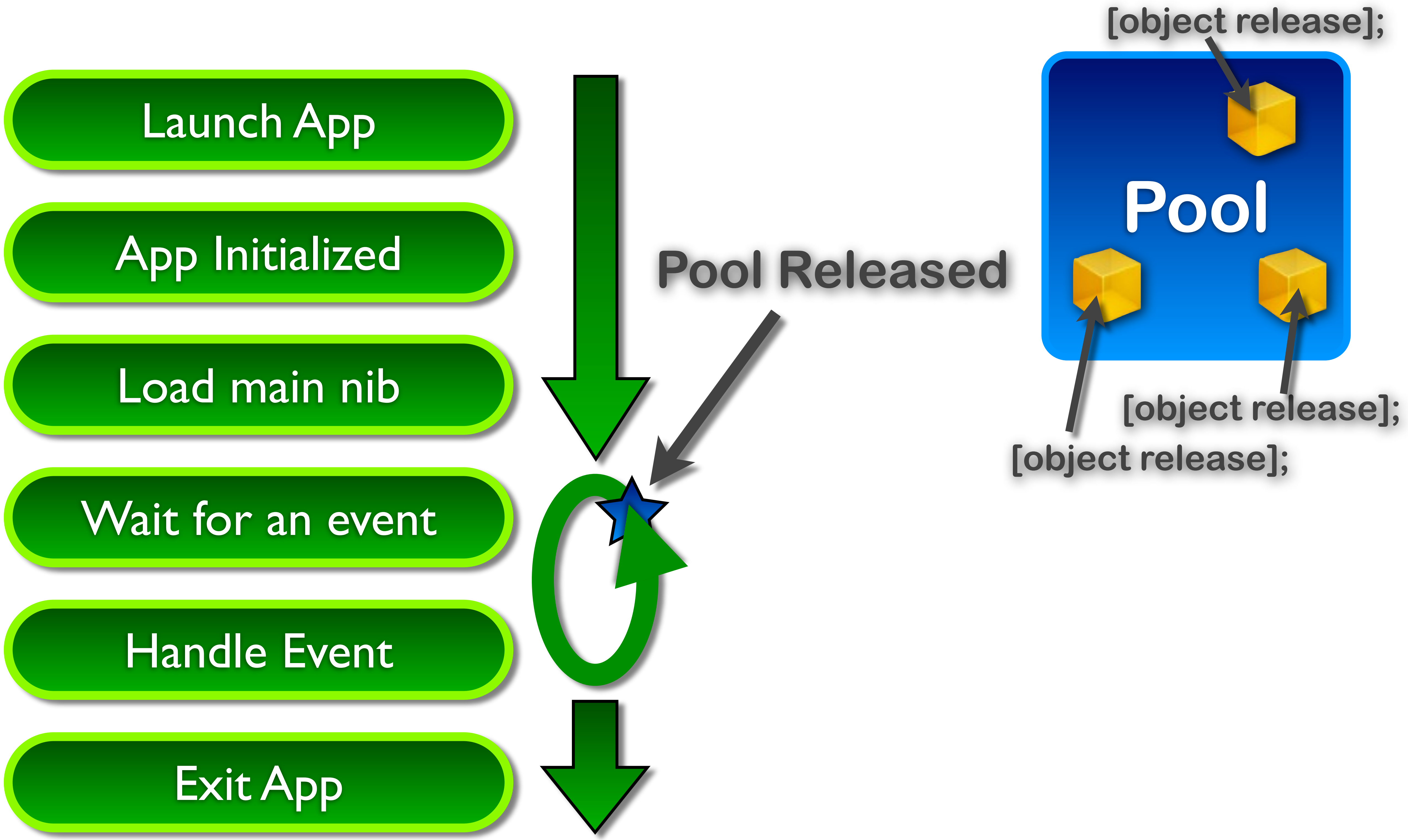


App Lifecycle

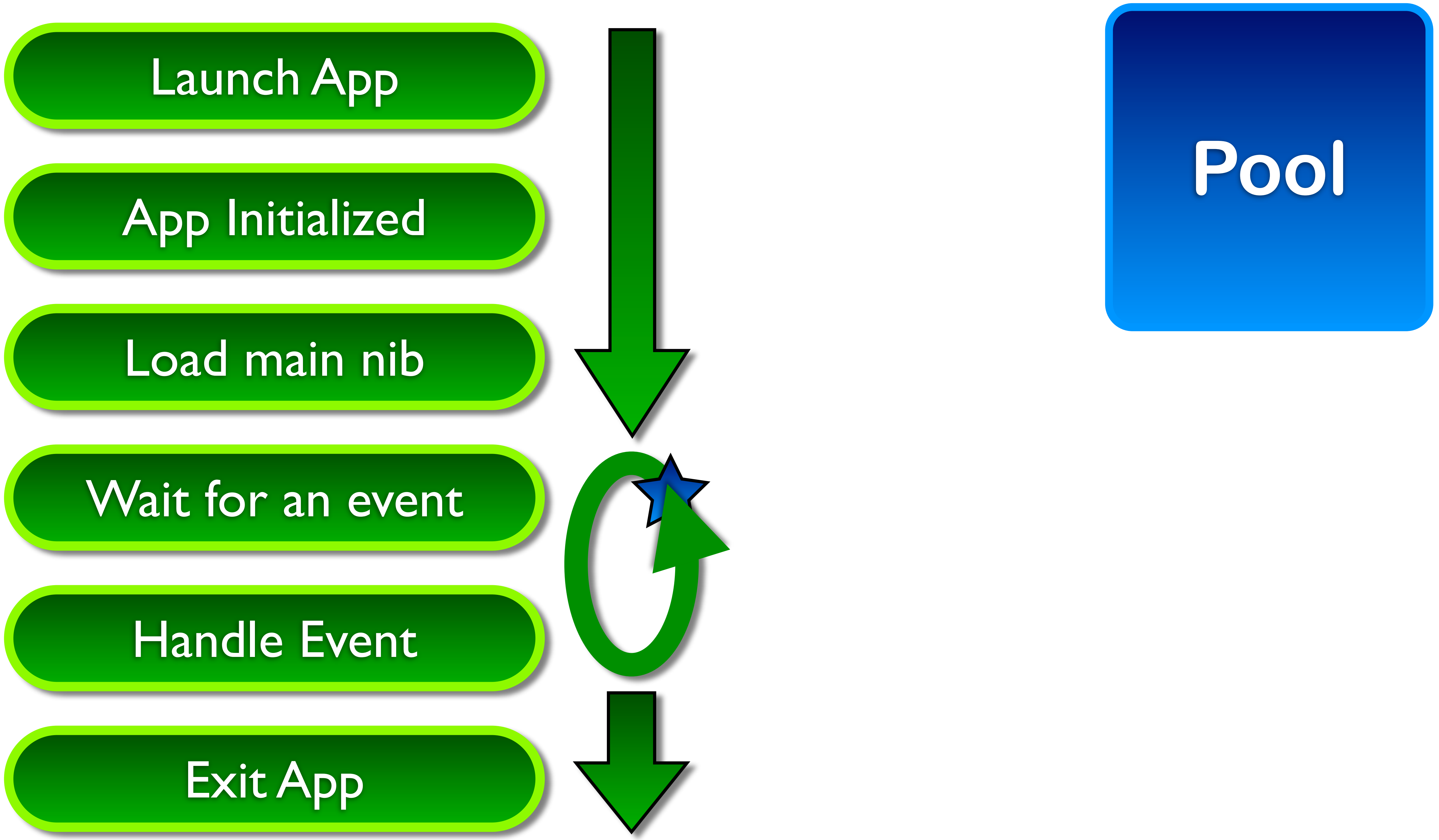


[object autorelease];

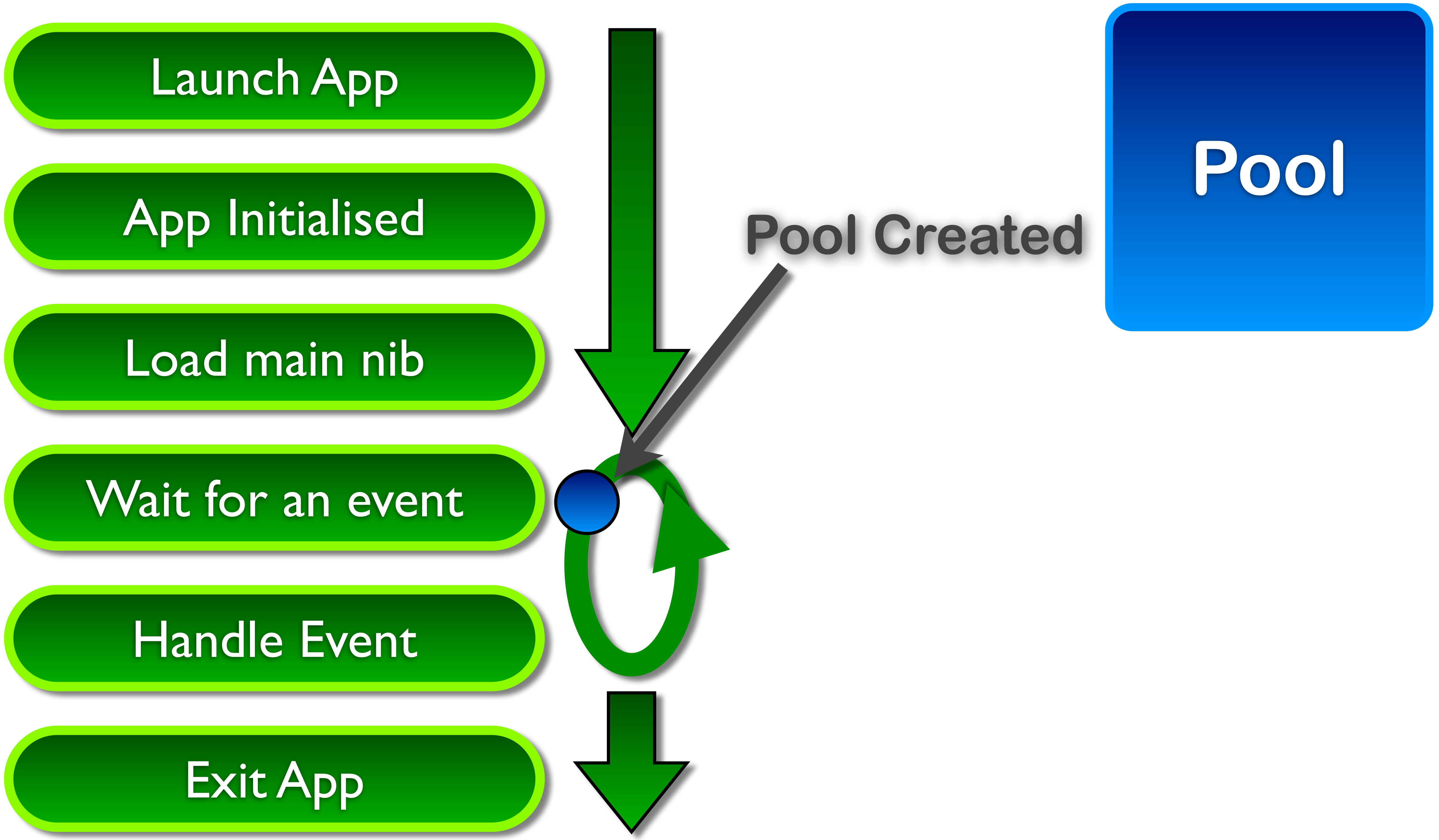
App Lifecycle



App Lifecycle



App Lifecycle



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];
```

Dynamic and Static Typing

- Dynamically-typed (loosely typed) object

```
id anObject
```

  - Just id
    - Not id * (unless you really, really mean it...)
  - Dynamic typic can be very powerful
  - No type-checking, so can be very dangerous!!!

- Statically-typed object

```
Person *anObject
```

  - Compile-time (not runtime) type checking
- Objective-C always uses dynamic binding

Foundation Framework

- Different to Cocoa
  - Supports all the underlying (common) classes
  - Common to all Apple platforms (OS X, iPhone etc)
- Defines classes to support the following:
  - Value and collection classes
  - User defaults
  - Archiving
  - Notifications
  - Undo manager
  - Tasks, timers, threads
  - File system, pipes, I/O, bundles

Two useful foundation classes

- **NSObject**
    - Root class
    - Implements many basics
        - Memory management
        - Introspection
        - Object equality
- **NSString**
    - General-purpose Unicode string support
        - Unicode is a coding system which represents all of the world's languages
    - Consistently used throughout Cocoa Touch instead of "char *"
    - Without doubt the most commonly used class
        - Easy to support any (spoken) language in the world with Cocoa

String Constants

- In C and Java, constant strings are 8 bit char arrays

```
“simple”
```

- In Objective C 2.0, constant strings are represented using unicode characters

```
@“just as simple”
```

- Constant strings are NSString instances
  - The “@” converts the string into an NSString

```
NSString *aString = @”Hello World!”;
```

Format Strings

- Similar to printf, but with %@ added for objects

```
NSString *aString = @”Johnny”;

NSString *log = [NSString stringWithFormat: @”It’s ‘%@’”, aString];
```

  - log would be set to

```
It’s Johnny
```

- Also used for logging

```
NSLog(@”I am a %@, I have %d items”, [array className], [array count]);
```

  - would log something like:

```
I am a NSArray, I have 5 items
```

More on NSString

- Often ask an existing string for a new string with modifications

```
- (NSString *)stringByAppendingString:(NSString *)string;
- (NSString *)stringByAppendingFormat:(NSString *)string;
- (NSString *)stringByDeletingPathComponent;
```

- Example:

```
NSString *myString = @”Hello”;
NSString *fullString;
fullString = [myString stringByAppendingString:@” world!”];
```

- fullString would be set to

```
Hello world!
```

More on NSString

- Common NSString methods

```
- (BOOL)isEqualToString:(NSString *)string;
- (BOOL)hasPrefix:(NSString *)string;
- (int)intValue;
- (double)doubleValue;
```

- Example:

```
NSString *myString = @”Hello”;
NSString *otherString = @”449”;
if ([myString hasPrefix:@”He”]) {
    // will make it here
}
if ([otherString intValue] > 500) {
    // won’t make it here
}
```

NSMutableString

- NSMutableString subclasses NSString
- Allows a string to be modified
- Common NSMutableString methods

```
+ (id)string;

- (void)appendString:(NSString *)string;

- (void)appendFormat:(NSString *)format, ...;


NSMutableString *newString = [NSMutableString string];

[newString appendString:@”Hi”];

[newString appendFormat:@”, my favourite number is: %d”,
                                [self favouriteNumber]];
```

NSNumber

- In Objective-C, you typically use standard C number types
  - NSNumber is used to wrap C number types as objects
  - Subclass of NSValue
  - 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;
```

Collections

- **Array** - ordered collection of objects
- **Dictionary** - collection of key-value pairs
- **Set** - unordered collection of unique objects
- Common enumeration mechanism
  - *Immutable* and *mutable* versions
    - Immutable collections can be shared without side effect
    - Prevents unexpected changes
    - Mutable objects typically carry a performance overhead

BOOL typedef

- When ObjC was developed, C had no boolean type
- ObjC uses a typedef to define BOOL as a type

```
BOOL flag = NO;
```

- Macros included for initialisation and comparison
  - YES and NO

```
if (flag == YES)
if (flag)
if (!flag)
if (flag != YES)
flag = YES;
flag = 1;
```

Identity vs Equality

- Identity—testing equality of the pointer values

```
if (object1 == object2) {
    NSLog(@"Same exact object instance");
}
```

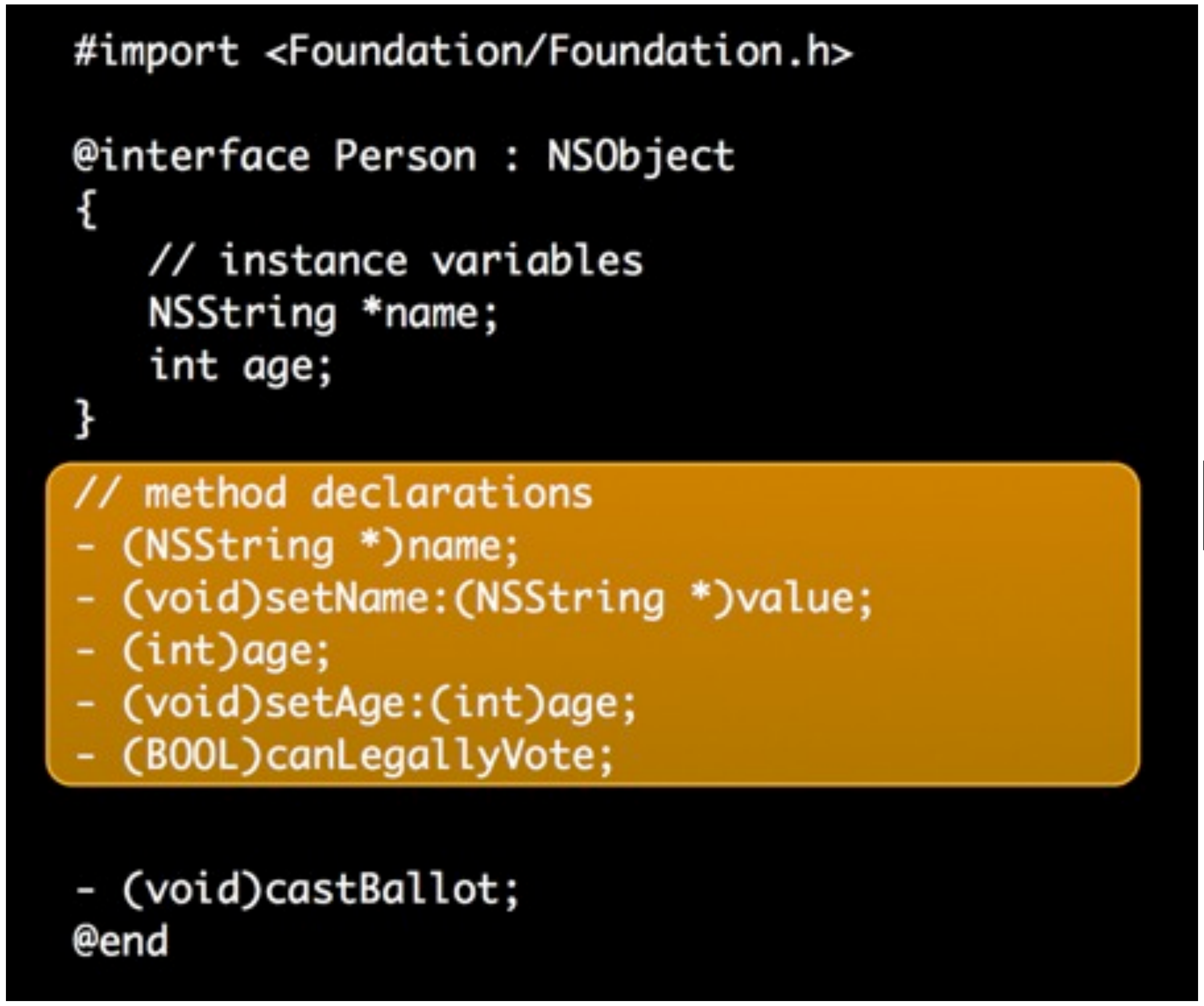
- Equality—testing object attributes

```
if ([object1 isEqual: object2]) {
    NSLog(@"Logically equivalent, but may be
different object instances");
}
```

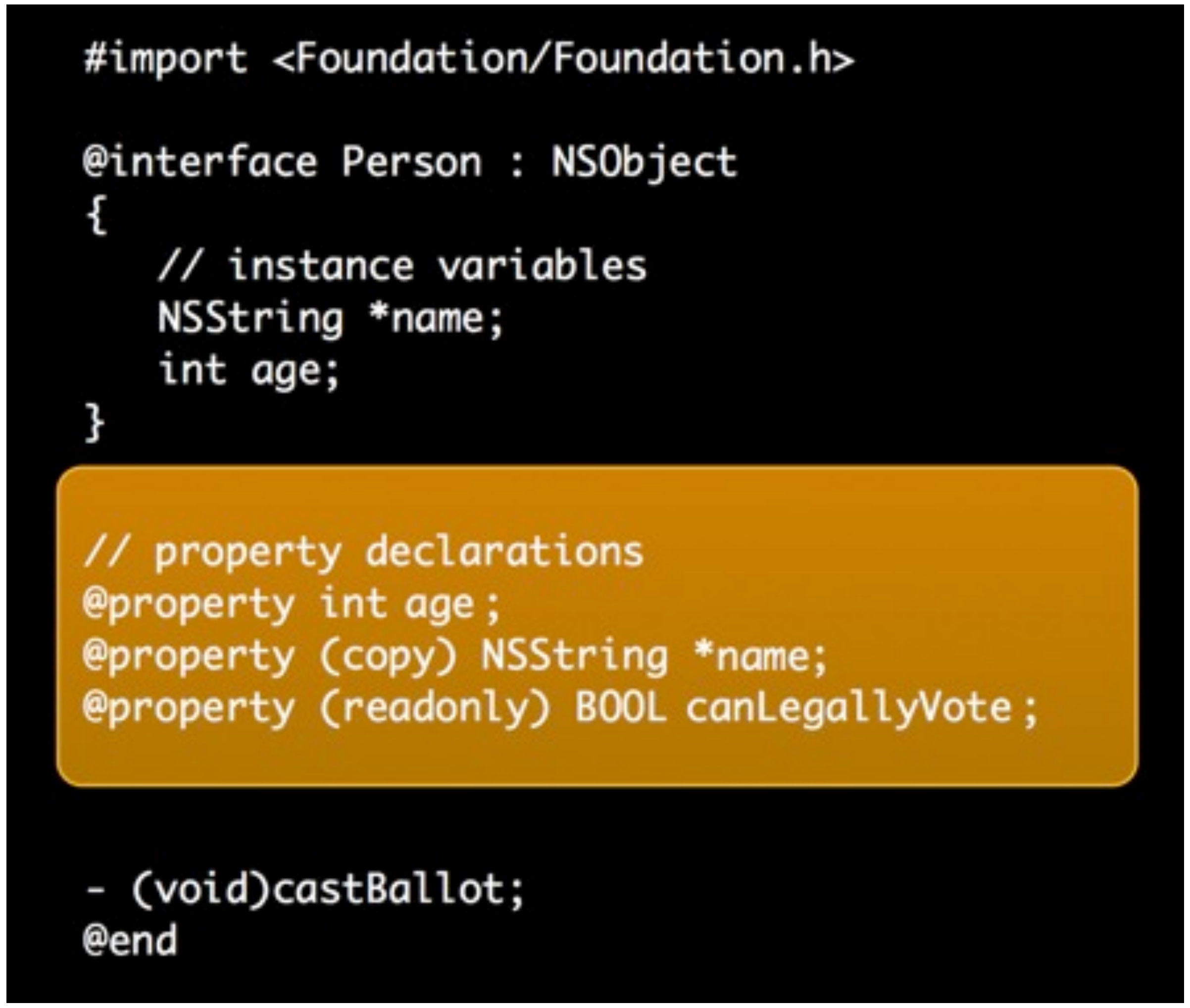
Properties

- 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

Defining Properties

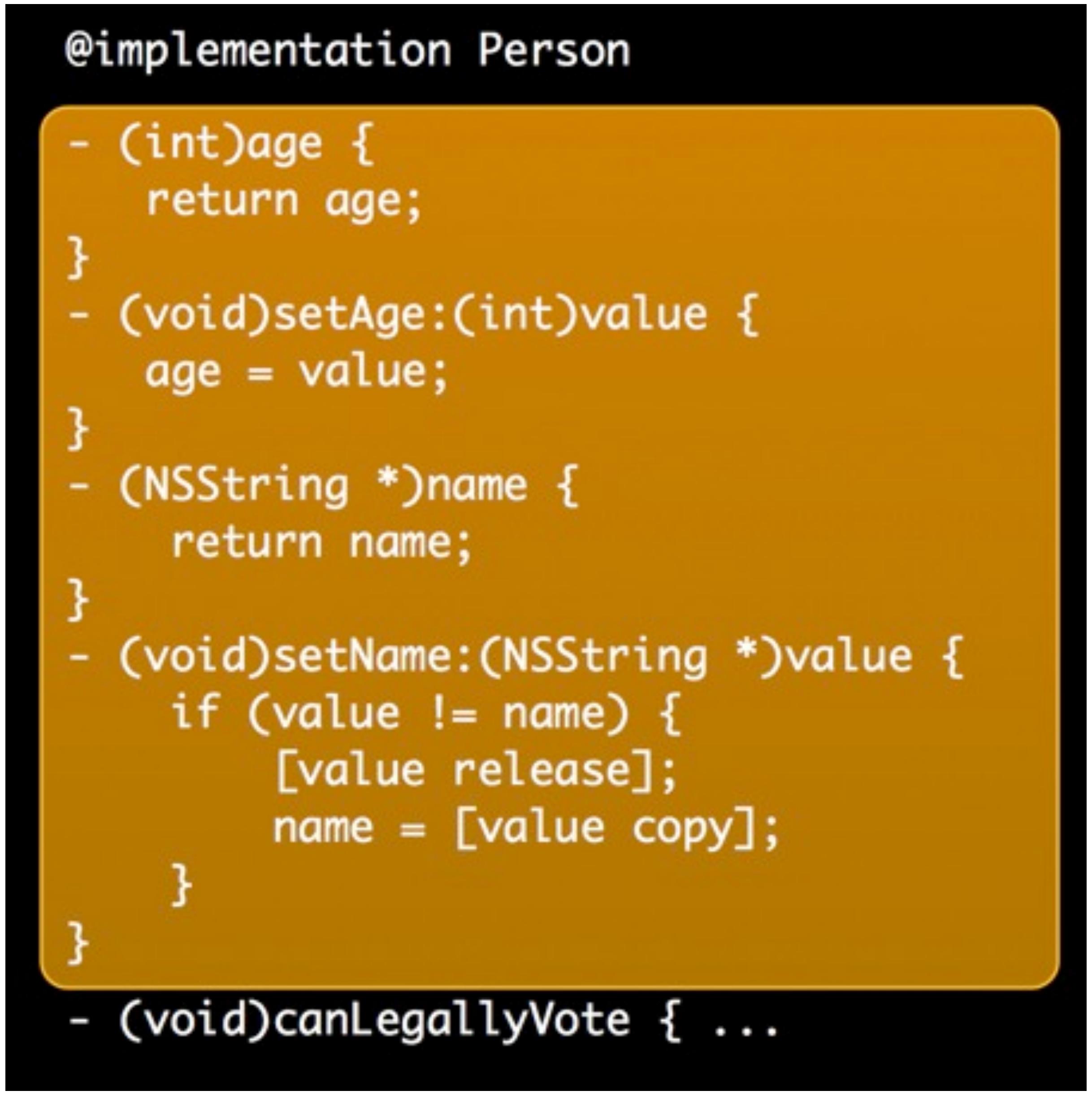


```
#import <Foundation/Foundation.h>

@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
```

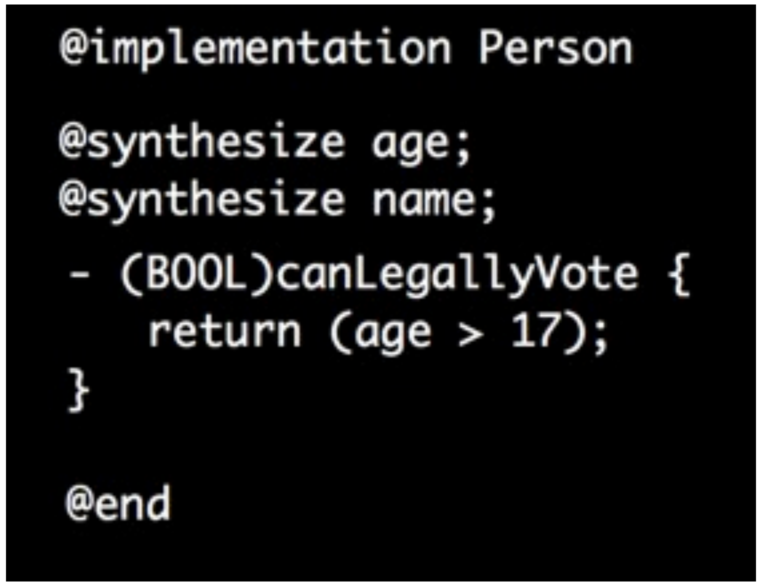


```
#import <Foundation/Foundation.h>

@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
```

Synthesising 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];
     }
}
- (void)canLegallyVote { ...
```



```
@implementation Person

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

@end
```

Property Attributes

- Read-only versus read-write

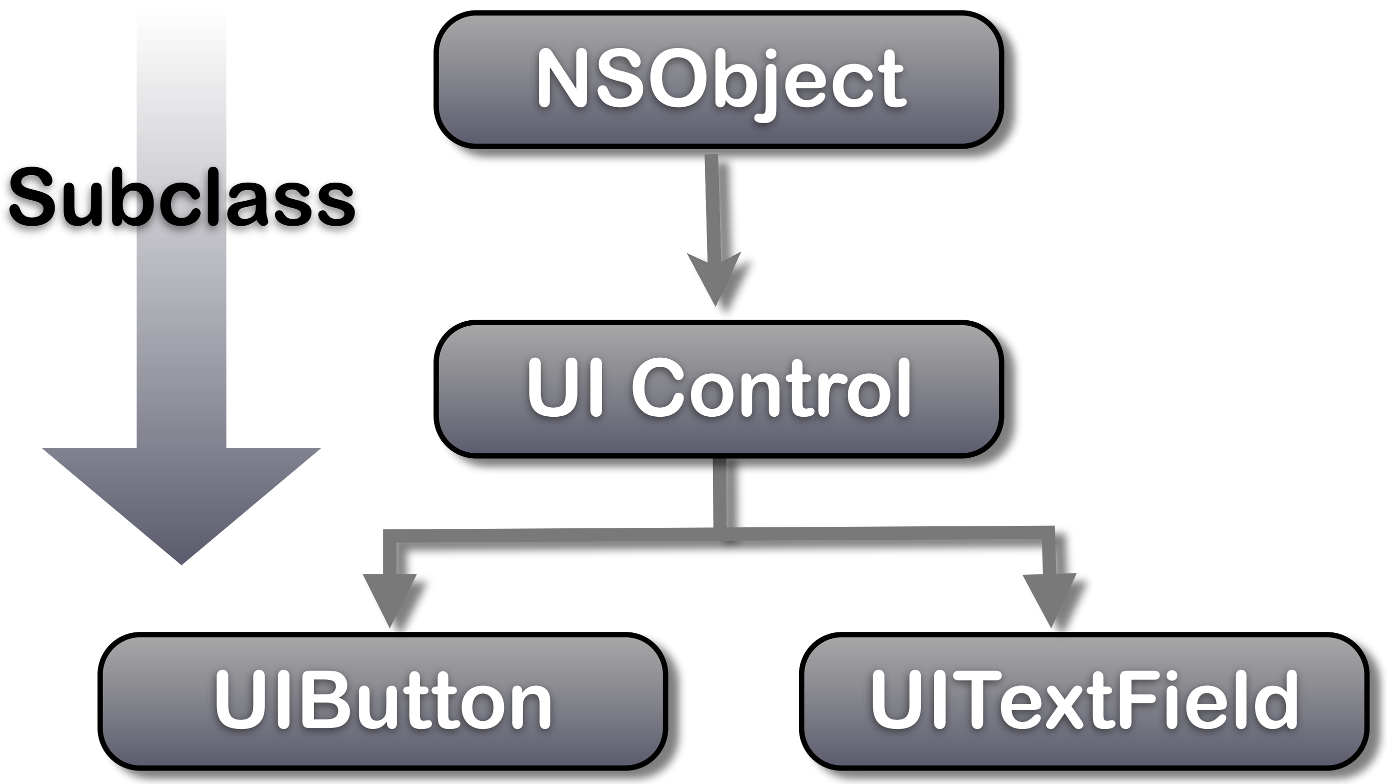
```
@property int age; // read-write by default
@property (readonly) BOOL canLegallyVote;
```

- 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
```

Inheritance

- Hierarchical relationship between classes
  - Subclasses inherit from superclasses
  - Everything **is** an object!!!
- Top Level class is NSObject
  - Takes care of memory management, introspection etc...
  - Typically not directly used, but often used as a superclass for other classes



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

To Summarise

- In this Tutorial Set we covered
  - A crash course in Objective-C !!!
  - Recap on Object Oriented Programming
    - Classes and Objects
  - Messaging Syntax
  - Foundation Framework
    - NSObject, NSString and mutable objects
  - C Pointers
  - Memory Allocation in Objective-C
    - Retain, Release and Autorelease