



Objective-C for Java Developers

Bob McCune
<http://bobmccune.com>

Objective-C Overview

Objective-C

The old new hotness

- Strict superset of ANSI C
 - Object-oriented extensions
 - Additional syntax and types
- Native Mac & iOS development language
- Flexible typing
- Simple, expressive syntax
- Dynamic runtime

Why Use Objective-C?

No Java love from Apple?

- Key to developing for Mac & iOS platforms
- Performance
 - Continually optimized runtime environment
 - Can optimize down to C as needed
 - Memory Management
- Dynamic languages provide greater flexibility
 - Cocoa APIs rely heavily on these features

Java Developer's Concerns

Ugh, didn't Java solve all of this stuff

- Pointers
- Memory Management
- Preprocessing & Linking
- No Namespaces
 - Prefixes used to avoid collisions
 - Common Prefixes: NS, UI, CA, MK, etc.

Creating Classes

Classes

- Classes define the blueprint for objects.
- Objective-C class definitions are separated into an *interface* and an *implementation*.
 - Usually defined in separate .h and .m files



- Defines the programming interface
- Defines the object's instance variable



- Defines the actual implementation code
- Defines one or more initializers to properly initialize and object instance

Defining the class *interface*

```
@interface BankAccount : NSObject {  
    float accountBalance;  
    NSString *accountNumber;  
}  
  
- (float)withDraw:(float)amount;  
- (void)deposit:(float)amount;  
  
@end
```

Defining the class *interface*

```
@interface BankAccount : NSObject {  
    float accountBalance;  
    NSString *accountNumber;  
}
```

- (float)withDraw:(float)amount;
- (void)deposit:(float)amount;

```
@end
```

Defining the class *interface*

```
@interface BankAccount : NSObject {  
    float accountBalance;  
    NSString *accountNumber;  
}
```

- (float)withDraw:(float)amount;
- (void)deposit:(float)amount;

```
@end
```

Defining the class *interface*

```
@interface BankAccount : NSObject {  
    float accountBalance;  
    NSString *accountNumber;  
}  
  
- (float)withDraw:(float)amount;  
- (void)deposit:(float)amount;  
  
@end
```

Defining the class *interface*

```
@interface BankAccount : NSObject {  
    float accountBalance;  
    NSString *accountNumber;  
}
```

- (float)withDraw:(float)amount;
- (void)deposit:(float)amount;

```
@end
```

Defining the class *interface*

```
@interface BankAccount : NSObject {  
    float accountBalance;  
    NSString *accountNumber;  
}  
  
- (float)withDraw:(float)amount;  
- (void)deposit:(float)amount;  
  
@end
```

Defining the class *implementation*

```
#import "BankAccount.h"
```

```
@implementation BankAccount
```

```
- (id)init {  
    self = [super init];  
    return self;  
}  
  
- (float)withdraw:(float)amount {  
    // calculate valid withdrawal  
    return amount;  
}  
  
- (void)deposit:(float)amount {  
    // record transaction  
}  
@end
```

Defining the class *implementation*

```
#import "BankAccount.h"
```

```
@implementation BankAccount
```

```
- (id)init {  
    self = [super init];  
    return self;  
}  
  
- (float)withdraw:(float)amount {  
    // calculate valid withdrawal  
    return amount;  
}  
  
- (void)deposit:(float)amount {  
    // record transaction  
}  
@end
```

Defining the class *implementation*

```
#import "BankAccount.h"
```

```
@implementation BankAccount
```

```
- (id)init {  
    self = [super init];  
    return self;  
}  
  
- (float)withdraw:(float)amount {  
    // calculate valid withdrawal  
    return amount;  
}  
  
- (void)deposit:(float)amount {  
    // record transaction  
}  
@end
```

Defining the class *implementation*

```
#import "BankAccount.h"
```

```
@implementation BankAccount
```

```
- (id)init {  
    self = [super init];  
    return self;  
}  
  
- (float)withdraw:(float)amount {  
    // calculate valid withdrawal  
    return amount;  
}  
  
- (void)deposit:(float)amount {  
    // record transaction  
}  
@end
```

Working with Objects

Creating Objects

From blueprint to reality

- No concept of constructors in Objective-C
- Regular methods create new instances
 - Allocation and initialization are performed separately
 - Provides flexibility in *where* an object is allocated
 - Provides flexibility in how an object is initialized

Java

```
BankAccount account = new BankAccount();
```

Objective C

```
BankAccount *account = [[BankAccount alloc] init];
```

Creating Objects

- NSObject defines class method called `alloc`

- Dynamically allocates memory for object
- Returns new instance of receiving class

```
BankAccount *account = [BankAccount alloc];
```

- NSObject defines instance method `init`

- Implemented by subclasses to initialize instance after memory for it has been allocated
- Subclasses commonly define several initializers

```
account = [account init];
```

- `alloc` and `init` calls are almost always nested into single line

```
BankAccount *account = [[BankAccount alloc] init];
```

Methods

- Classes can define both class and instance methods
- Methods are always public.
 - “Private” methods defined in implementation
- Class methods (prefixed with +):
 - Define behaviors associated with class, not particular instance
 - No access to instance variables
- Instance methods (prefixed with -)
 - Define behaviors specific to particular instance
 - Manage object state
- Methods invoked by passing *messages*...

Messages

Speaking to objects

- Methods are invoked by passing *messages*
 - Done indirectly. We never directly invoke methods
 - Messages dynamically bound to method implementations at runtime
 - Dispatching handled by runtime environment
- Simple messages take the form:
`[object message];`
- Can pass one or more arguments:
`[object messageWithArg1:arg1 arg2:arg2];`

self and super

- Methods have implicit reference to owning object called **self** (similar to Java's **this**)
- Additionally have access to superclass methods using **super**

```
-(id)init {  
    if (self = [super init]) {  
        // do initialization  
    }  
    return self;  
}
```

Invoking methods in Java

```
Map person = new HashMap();
```

```
person.put("name", "Joe Smith");
```

```
String address = "123 Street";
```

```
String house = address.substring(0, 3);
```

```
person.put("houseNumber", house);
```

```
List children = Arrays.asList("Sue", "Tom");
```

```
person.put("children", children);
```

Invoking methods in Objective-C

```
NSMutableDictionary *person =  
    [NSMutableDictionary dictionary];  
  
[person setObject:@"Joe Smith" forKey:@"name"];  
  
NSString *address = @"123 Street";  
NSString *house =  
    [address substringWithRange:NSMakeRange(0, 3)];  
[person setObject:house forKey:@"houseNumber"];  
  
NSArray *children =  
    [NSArray arrayWithObjects:@"Sue", @"Tom", nil];  
  
[person setObject:children forKey:@"children"];
```

Memory Management

Memory Management

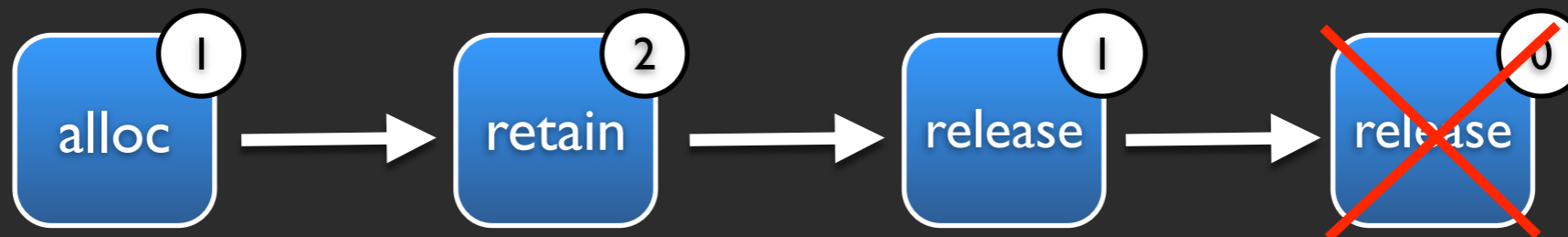
Dude, where's my Garbage Collection?

- Garbage collection available for Mac apps (10.5+)
 - Use it if targeting 10.5+!
 - Use explicit memory management if targeting ≤ 10.4
- No garbage collection on iOS due to performance concerns.
- Memory managed using simple *reference counting* mechanism:
 - Higher level abstraction than `malloc / free`
 - Straightforward approach, but must adhere to conventions and rules

Memory Management

Reference Counting

- Objective-C objects are reference counted
 - Objects start with reference count of 1
 - Increased with **retain**
 - Decreased with **release**, **autorelease**
 - When count equals 0, runtime invokes **dealloc**



Memory Management

Understanding the rules

- Implicitly **retain** any object you create using:
Methods starting with "alloc", "new", or contains "copy"
e.g `alloc`, `allocWithZone`, `newObject`, `mutableCopy`, etc.
- If you've retained it, you must **release** it
- Many objects provide convenience methods that return an *autoreleased* reference.
- Holding object reference from these methods does not make you the retainer
 - However, if you **retain** it you need an offsetting **release** or **autorelease** to free it

retain / release / autorelease

```
@implementation BankAccount
```

```
- (NSString *)accountNumber {  
    return [[accountNumber retain] autorelease];  
}
```

```
- (void)setAccountNumber:(NSString *)newNumber {  
    if (accountNumber != newNumber) {  
        [accountNumber release];  
        accountNumber = [newNumber retain];  
    }  
}
```

```
- (void)dealloc {  
    [self setAccountNumber:nil];  
    [super dealloc];  
}
```

```
@end
```

Properties

Properties

Simplifying Accessors

- Object-C 2.0 introduced new syntax for defining accessor code
 - Much less verbose, less error prone
 - Highly configurable
 - Automatically generates accessor code
- Compliments existing conventions and technologies
 - Key-Value Coding (KVC)
 - Key-Value Observing (KVO)
 - Cocoa Bindings
 - Core Data

Properties: Interface

```
@property(attributes) type variable;
```

Attribute	Impacts
readonly/readwrite	Mutability
assign/retain/copy	Storage
nonatomic	Concurrency
setter/getter	API

Properties: Interface

```
@property(attributes) type variable;
```

Attribute	Impacts
readonly/readwrite	Mutability
assign/retain/copy	Storage
nonatomic	Concurrency
setter/getter	API

```
@property(readonly) NSString *acctNumber;
```

Properties: Interface

```
@property(attributes) type variable;
```

Attribute	Impacts
readonly/readwrite	Mutability
assign/retain/copy	Storage
nonatomic	Concurrency
setter/getter	API

```
@property(readwrite, copy) NSString *acctNumber;
```

Properties: Interface

```
@property(attributes) type variable;
```

Attribute	Impacts
readonly/readwrite	Mutability
assign/retain/copy	Storage
nonatomic	Concurrency
setter/getter	API

```
@property(nonatomic, retain) NSDate *activeOn;
```

Properties: Interface

```
@property(attributes) type variable;
```

Attribute	Impacts
readonly/readwrite	Mutability
assign/retain/copy	Storage
nonatomic	Concurrency
setter/getter	API

```
@property(readonly, getter=username) NSString *name;
```

Properties: Interface

```
@interface BankAccount : NSObject {  
    NSString *accountNumber;  
    NSDecimalNumber *balance;  
    NSDecimalNumber *fees;  
    BOOL accountCurrent;  
}
```

```
@property(readwrite, copy) NSString *accountNumber;  
@property(readwrite, retain) NSDecimalNumber *balance;  
@property(readonly) NSDecimalNumber *fees;  
@property(getter=isCurrent) BOOL accountCurrent;
```

```
@end
```

Properties: Interface

New in 10.6
and iOS 4

```
@interface BankAccount : NSObject {  
    // Look ma, no more ivars  
}
```

```
@property(readwrite, copy) NSString *accountNumber;  
@property(readwrite, retain) NSNumber *balance;  
@property(readonly) NSNumber *fees;  
@property(getter=isCurrent) BOOL accountCurrent;
```

```
@end
```

Properties: Implementation

```
@implementation BankAccount
```

```
@synthesize accountNumber;
```

```
@synthesize balance;
```

```
@synthesize fees;
```

```
@synthesize accountCurrent;
```

```
...
```

```
@end
```

Properties: Implementation

```
@implementation BankAccount
```

New in 10.6
and iOS 4

```
// no more @synthesize statements
```

```
...
```

```
@end
```

Accessing Properties

- Generated properties are standard methods
- Accessed through normal messaging syntax

```
[object property];
```

```
[object setProperty:(id)newValue];
```

- Objective-C 2.0 property access via dot syntax

```
object.property;
```

```
object.property = newValue;
```

Dot notation is just syntactic sugar. Still uses accessor methods. Doesn't get/set values directly.

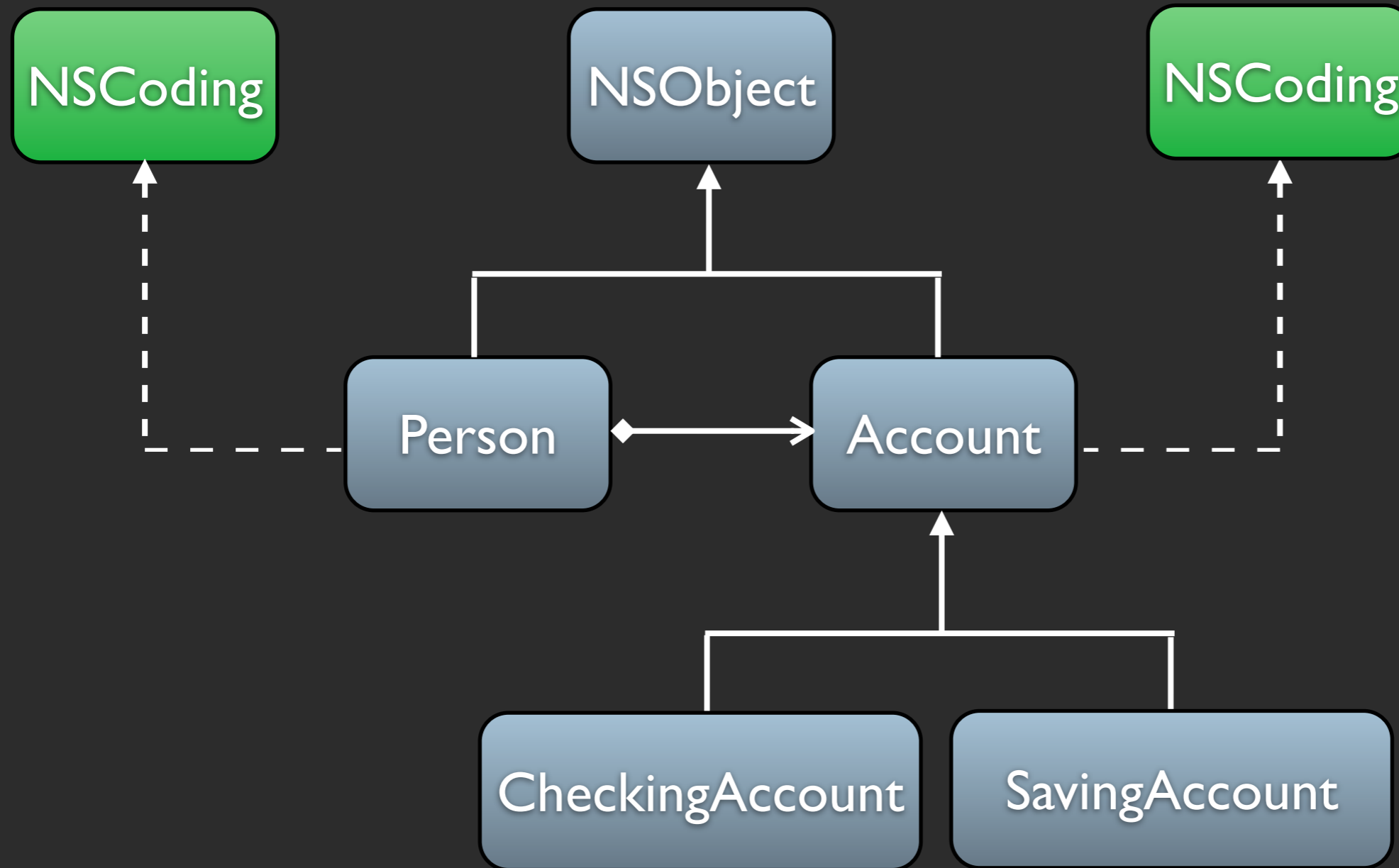
Protocols

Protocols

Java's Interface done Objective-C style

- List of method declarations
 - Not associated with a particular class
 - Conformance, not class, is important
- Useful in defining:
 - Methods that others are expected to implement
 - Declaring an interface while hiding its particular class
 - Capturing similarities among classes that aren't hierarchically related

Protocols



Defining a Protocol

*NSCoding from Foundation Framework
Objective-C equivalent to java.io.Externalizable*

@protocol NSCoding

- (void)encodeWithCoder:(NSCoder *)aCoder;
- (id)initWithCoder:(NSCoder *)aDecoder;

@end

Adopting a Protocol

```
@interface Person : NSObject <NSCoding> {  
    NSString *name;  
    NSString *street;  
    NSString *city, *state, *zip;  
}  
  
// method declarations  
  
@end
```

Conforming to a Protocol

Partial implementation of conforming Person class

```
@implementation Person
```

```
-(id)initWithCoder:(NSCoder *)coder {  
    if (self = [super init]) {  
        name = [coder decodeObjectForKey:@"name"];  
        [name retain];  
    }  
    return self;  
}
```

```
-(void)encodeWithCoder:(NSCoder *)coder {  
    [coder encodeObject:name forKey:@"name"];  
}
```

```
@end
```

@required and @optional

- Protocols methods are required by default
- Can be relaxed with `@optional` directive

```
@protocol SomeProtocol
```

```
- (void)requiredMethod;
```

```
@optional
```

```
- (void)anOptionalMethod;
```

```
- (void)anotherOptionalMethod;
```

```
@required
```

```
- (void)anotherRequiredMethod;
```

```
@end
```

Categories & Extensions

Categories

Extending Object Features

- Add new methods to existing classes
 - Alternative to subclassing
 - Defines new methods and can override existing
 - Does not define new instance variables
 - Becomes part of the class definition
 - Inherited by subclasses
- Can be used as organizational tool
- Often used in defining "private" methods

Using Categories

Interface

```
@interface NSString (Extensions)
-(NSString *)trim;
@end
```

Implementation

```
@implementation NSString (Extensions)
-(NSString *)trim {
    NSMutableCharacterSet *charSet =
        [NSMutableCharacterSet whitespaceAndNewlineCharacterSet];
    return [self stringByTrimmingCharactersInSet:charSet];
}
@end
```

Usage

```
NSString *string = @"  A string to be trimmed  ";
NSLog(@"Trimmed string: '%@'", [string trim]);
```

Class Extensions

Unnamed Categories

- Objective-C 2.0 adds ability to define "anonymous" categories
 - Category is unnamed
 - Treated as class interface continuations
- Useful for implementing required "private" API
- Compiler enforces methods are implemented

Class Extensions

Example

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

Class Extensions

Example

```
@interface Person ()  
-(void)setName:(NSString *)newName;
```

```
@end
```

```
@implementation Person
```

```
- (NSString *)name {  
    return name;  
}
```

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

```
@end
```

Summary

Objective-C

Summary

- Fully C, Fully Object-Oriented
- Powerful dynamic runtime
- Objective-C 2.0 added many useful new features
 - Garbage Collection for Mac OS X apps
 - Properties, Improved Categories & Protocols
- Objective-C 2.1 continues its evolution:
 - Blocks (Closures)
 - Synthesize by default for properties

Questions?