Introduction to Objective-C

Kevin Cathey
## Introduction to Objective-C

- What are object-oriented systems?
- What is the Objective-C language?
- What are objects?
- How do you create classes in Objective-C?
Introduction to Objective-C

What are object-oriented systems?

What is the Objective-C language?

What are objects?

How do you create classes in Objective-C?
What is an object-oriented system?

• Has four components:
What is an object-oriented system?

- Has four components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Example in iPhone Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>A programming language</td>
<td>Objective-C</td>
</tr>
<tr>
<td>Library of classes and functions for common tasks</td>
<td>CoreFoundation, Foundation, UIKit, CoreGraphics, CoreLocation, …</td>
</tr>
<tr>
<td>Suite of developer tools</td>
<td>Xcode, Interface Builder, Instruments</td>
</tr>
<tr>
<td>Runtime system that “runs” code at runtime</td>
<td>Objective-C Runtime</td>
</tr>
</tbody>
</table>
What is an object-oriented system?

- Has four components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Example in iPhone Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>A programming language</td>
<td>Objective-C</td>
</tr>
<tr>
<td>Library of classes and functions for common tasks</td>
<td>CoreFoundation, Foundation, UIKit, CoreGraphics, CoreLocation, …</td>
</tr>
<tr>
<td>Suite of developer tools</td>
<td>Xcode, Interface Builder, Instruments</td>
</tr>
<tr>
<td>Runtime system that “runs” code at runtime</td>
<td>Objective-C Runtime</td>
</tr>
</tbody>
</table>
What is an object-oriented system?

- Has four components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Example in iPhone Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>A programming language</td>
<td>Objective-C</td>
</tr>
<tr>
<td>Library of classes and functions for common tasks</td>
<td>CoreFoundation, Foundation, UIKit, CoreGraphics, CoreLocation, ...</td>
</tr>
</tbody>
</table>
What is an object-oriented system?

- Has four components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Example in iPhone Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>A programming language</td>
<td>Objective-C</td>
</tr>
<tr>
<td>Library of classes and functions for common tasks</td>
<td>CoreFoundation, Foundation, UIKit, CoreGraphics, CoreLocation, …</td>
</tr>
<tr>
<td>Suite of developer tools</td>
<td>Xcode, Interface Builder, Instruments</td>
</tr>
</tbody>
</table>
What is an object-oriented system?

- Has four components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Example in iPhone Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>A programming language</td>
<td>Objective-C</td>
</tr>
<tr>
<td>Library of classes and functions for common tasks</td>
<td>CoreFoundation, Foundation, UIKit, CoreGraphics, CoreLocation, …</td>
</tr>
<tr>
<td>Suite of developer tools</td>
<td>Xcode, Interface Builder, Instruments</td>
</tr>
<tr>
<td>Runtime system that “runs” code at runtime</td>
<td>Objective-C Runtime</td>
</tr>
</tbody>
</table>
Introduction to Objective-C

What are object-oriented systems?

What is the Objective-C language?

What are objects?

How do you create classes in Objective-C?

acm.uiuc.edu/macwarriors/devphone
Objective-C Programming Language
Objective-C Programming Language

• Similarities with C
Objective-C Programming Language

• Similarities with C
  ■ Strict-superset of C — **everything** that works in C works in Objective-C (unlike C++).
Objective-C Programming Language

• Similarities with C
  ▪ Strict-superset of C — **everything** that works in C works in Objective-C (unlike C++).
  ▪ Super-set because it adds object-oriented constructs.
Objective-C Programming Language

• Similarities with C
  ▪ Strict-superset of C — *everything* that works in C works in Objective-C (unlike C++).
  ▪ Super-set because it adds object-oriented constructs.
  ▪ When compiling Objective-C, largely get turned into C.
Objective-C Programming Language

• Similarities with C
  ▪ Strict-superset of C — **everything** that works in C works in Objective-C (unlike C++).
  ▪ Super-set because it adds object-oriented constructs.
  ▪ When compiling Objective-C, largely get turned into C.
• Derives a lot of design and functionality from Smalltalk.
Objective-C Programming Language

• Similarities with C
  ■ Strict-superset of C — **everything** that works in C works in Objective-C (unlike C++).
  ■ Super-set because it adds object-oriented constructs.
  ■ When compiling Objective-C, largely get turned into C.
• Derives a lot of design and functionality from Smalltalk.
• Typing
Objective-C Programming Language

• Similarities with C
  ▪ Strict-superset of C — **everything** that works in C works in Objective-C (unlike C++).
  ▪ Super-set because it adds object-oriented constructs.
  ▪ When compiling Objective-C, largely get turned into C.
• Derives a lot of design and functionality from Smalltalk.
• Typing
  ▪ Dynamic typing for Objective-C objects (classes).
Objective-C Programming Language

• Similarities with C
  ▪ Strict-superset of C — **everything** that works in C works in Objective-C (unlike C++).
  ▪ Super-set because it adds object-oriented constructs.
  ▪ When compiling Objective-C, largely get turned into C.

• Derives a lot of design and functionality from Smalltalk.

• Typing
  ▪ Dynamic typing for Objective-C objects (classes).
  ▪ Static typing for C scalar and pointer types.
Objective-C Programming Language

• Another improvement over C
  ▪ #import
    ▪ Smart #include
    ▪ No more ifndef and define __MYHEADER_H__
Introduction to Objective-C

- What are object-oriented systems?
- What is the Objective-C language?
- What are objects?
- How do you create classes in Objective-C?
What are (Objective-C) objects?
What are (Objective-C) objects?

• In C, we have scalar types (int, long, struct) and pointers to those (including void *).
What are (Objective-C) objects?

• In C, we have scalar types (int, long, struct) and pointers to those (including void *).
• In Objective-C, we have C types, but also a core type called an object (or instance).
Objects are an instance of some type (or class) that contain nouns (instance variables) and act with verbs (methods).
Objects
Objects

• Instance variables — the nouns.
  ▪ Data.
  ▪ Store some state or information pertaining to the specific instance they reside in.
Objects

• Instance variables — the nouns.
  ▪ Data.
  ▪ Store some state or information pertaining to the specific instance they reside in.

• Instance methods — the verbs.
  ▪ Act on one instance’s instance variables.
Objects

Instance variables & methods example
Objects

Instance variables & methods example

Jim
instance of Person
Objects

Instance variables & methods example

Instance variables:
- name = "Jim"
- age = 25

Jim
instance of Person
Objects

Instance variables & methods example

Jim
instance of Person

**Instance variables:**
- name = “Jim”
- age = 25

**Instance methods:**
- age()
- isOlderThan(Person p)
Objects
Instance variables & methods example

Jim
instance of Person

Pam
instance of Person

**Instance variables:**
- name = “Jim”
- age = 25

**Instance methods:**
- age()
- isOlderThan(Person p)

**Instance variables:**
- name = “Pam”
- age = 24

**Instance methods:**
- age()
- isOlderThan(Person p)
Objects
Instance variables & methods example

Jim
instance of Person

Instance variables:
• name = “Jim”
• age = 25

Instance methods:
• age()
• isOlderThan(Person p)

Jim.isOlderThan(Pam) returns true

Pam
instance of Person

Instance variables:
• name = “Pam”
• age = 24

Instance methods:
• age()
• isOlderThan(Person p)
Objects

Instance variables & methods example

Jim
instance of Person

Pam
instance of Person

**Instance variables:**
- name = “Jim”
- age = 25

**Instance methods:**
- age()
- isOlderThan(Person p)

Jim.isOlderThan(Pam) returns true

Pam.isOlderThan(Jim) returns false

**Instance variables:**
- name = “Pam”
- age = 24

**Instance methods:**
- age()
- isOlderThan(Person p)
Special type — id
Special type — **id**

- Every true object (not void *) is of implicit type **id** (pronounced “id” as in “did”, not “ID”).
Special type — id

• Every true object (not void *) is of implicit type id (pronounced “id” as in “did”, not “ID”).

• id is not a class, it’s a type.
Special type — id

• Every true object (not void *) is of implicit type id (pronounced “id” as in “did”, not “ID”).

• id is not a class, it’s a type.

• Typing object as id tells compiler: “This will eventually be some Objective-C object.”
Special type — id

• Every true object (not void *) is of implicit type id (pronounced “id” as in “did”, not “ID”).
• id is not a class, it’s a type.
• Typing object as id tells compiler: “This will eventually be some Objective-C object”.
• Powerful because you can send any message to an id object and nothing happens till runtime.
Objects and messages
Objects and messages

- When calling instance method (or class method), not actually calling a physical address or function pointer.
Objects and messages

• When calling instance method (or class method), not actually calling a physical address or function pointer.
• Instead, sending object a command to execute a method by some name (called a selector).
Objects and messages

- When calling instance method (or class method), not actually calling a physical address or function pointer.
- Instead, sending object a command to execute a method by some name (called a selector).
- Basic result: dynamic binding.
Messaging

An example

Programmer
subclass of Person

Elephant
subclass of Animal
Messaging

An example

Programmer
subclass of Person

Elephant
subclass of Animal
Messaging
An example

Programmer subclass of Person

Elephant subclass of Animal
Messaging

An example

Need to feed entire list, but don’t know if entry is Programmer or Elephant.

So just send the feed message (call feed instance method) on each. Runtime then figures out who it is.
Messaging
An example

Need to feed entire list, but don’t know if entry is Programmer or Elephant.
So just send the feed message (call feed instance method) on each. Runtime then figures out who it is.
Messages

• Are powerful. Make the following very easy:
  ▪ Dynamic code loading (like plugin architectures).
  ▪ Categories (I’ll explain later)
Messages

Basic syntax

Java/C++

myObject.doSomething();

Objective-C

[myObject doSomething]
Messages

Nesting

Java/C++

myObject.doSomething().doSomethingElse(someArgument);

Objective-C

[[myObject doSomething] doSomethingElse:someArgument];
Messages

Multiple Parameters

Java/C++

```java
myObject.doThisWithThatAndThat(argument1, argument2);
```

Objective-C

```objective-c
[myObject doThisWithThat:argument1 andThat:argument2];
```
Messages

Multiple Parameters

Java/C++

myObject.doThisWithThatAndThat(argument1, argument2);

Objective-C

[myObject doThisWithThat:argument1 andThat:argument2];

Named Parameters!
Messages

Sending to nil

Java

```java
MyClass instance = null;
instance.doSomething(); // throws NullPointerException
```

Objective-C

```objective-c
MyClass *instance = nil; // use nil instead of NULL
[instance doSomething]; // just does nothing
```

Objective-C — nil and return values

```objective-c
Person *person = nil;
[person age]; // returns 0
[person name]; // returns nil
```
Selectors

- A **selector** describes a message, based upon its name.
- Many arguments in frameworks like Foundation and UIKit take selectors.

**To create a selector directly:**

```c
SEL mySelector = @selector(myMethod:);
```

**To create a selector from a string:**

```c
SEL mySelector = NSSelectorFromString(@"myMethod:");
```
Selectors

- A **selector** describes a message, based upon its name.
- Many arguments in frameworks like Foundation and UIKit take selectors.

To create a selector directly:

```
SEL mySelector = @selector(myMethod:);
```

To create a selector from a string:

```
SEL mySelector = NSSelectorFromString(@"myMethod:";
```
Introduction to Objective-C

What are object-oriented systems? ✓
What is the Objective-C language? ✓
What are objects? ✓

How do you create classes in Objective-C?

acm.uiuc.edu/macwarriors/devphone
Interface versus Implementation
Interface versus Implementation

• Two parts to creating a class (unlike Java).
Interface versus Implementation

• Two parts to creating a class (unlike Java).

• Interface
  ▪ Tell what capabilities a class has.
  ▪ Declare the class, it’s superclass, instance variables, and methods.
  ▪ Lives (usually) in header file (.h).
Interface versus Implementation

• Two parts to creating a class (unlike Java).

• **Interface**
  ▪ Tell what capabilities a class has.
  ▪ Declare the class, it’s superclass, instance variables, and methods.
  ▪ Lives (usually) in header file (.h).

• **Implementation**
  ▪ The code behind the methods.
  ▪ Lives (always) in implementation files (.m).
Interface
Interface

• What goes in the interface
Interface

• What goes in the interface
  • Superclass
Interface

• What goes in the interface
  ▪ Superclass
  ▪ Instance variables
Interface

• What goes in the interface
  ▪ Superclass
  ▪ Instance variables
  ▪ Instance methods
Interface

• What goes in the interface
  ▪ Superclass
  ▪ Instance variables
  ▪ Instance methods
  ▪ Properties (we’ll get to these)
Interface

• What goes in the interface
  ▪ Superclass
  ▪ Instance variables
  ▪ Instance methods
  ▪ Properties (we’ll get to these)
  ▪ Protocols (we’ll get to these)
Essential types
Essential types

- **NSUInteger & NSInteger**
  - Use instead of `int, long, unsigned,` etc.
  - For architecture independence.
Essential types

• **NSUInteger & NSInteger**
  • Use instead of `int`, `long`, `unsigned`, etc.
  • For architecture independence.

• **NSString**
  • Wrapper for C-string.
  • Easiest way to create them is `@"myString"`. 
Essential types

- **NSUInteger & NSInteger**
  - Use instead of `int`, `long`, `unsigned`, etc.
  - For architecture independence.

- **NSString**
  - Wrapper for C-string.
  - Easiest way to create them is `@"myString"`.

- **BOOL**
  - Objective-C boolean (implemented as 8-bit unsigned char).
  - Valid values are `YES` and `NO`. 
@interface Person : NSObject {
    NSString * name;
    NSUInteger age;
}

- (id)initWithName:(NSString *)name andAge:(NSUInteger)age;
- (BOOL)isPersonOlder:(Person *)otherPerson;

@end
@interface Person : NSObject {
    NSString * name;
    NSUInteger age;
}

- (id)initWithName:(NSString *)name andAge:(NSUInteger)age;
- (BOOL)isPersonOlder:(Person *)otherPerson;
@end
@interface Person : NSObject {
    NSString * name;
    NSUInteger age;
}

- (id)initWithName:(NSString *)name andAge:(NSUInteger)age;
- (BOOL)isPersonOlder:(Person *)otherPerson;
@end
In Person.h:

```objective-c
@interface Person : NSObject {
    NSString *name;
    NSUInteger age;
}
@end
```

**Instance methods**

- (id)initWithName:(NSString *)name andAge:(NSUInteger)age;
- (BOOL)isPersonOlder:(Person *)otherPerson;

@end
In Person.h:

```objective-c
@interface Person : NSObject {
    NSString * name;
    NSUInteger age;
}
@end

- (id)initWithName:(NSString *)name andAge:(NSUInteger)age;
- (BOOL)isPersonOlder:(Person *)otherPerson;
```

“-“ specifies instance method
“+” specifies class method
@interface Person : NSObject {
    NSString * name;
    NSUInteger age;
}

-(id)initWithName:(NSString *)name andAge:(NSUInteger)age;
-(BOOL)isPersonOlder:(Person *)otherPerson;

@end
Implementation

Example

In Person.m:

```c
@implementation

-(id)initWithName:(NSString *)name andAge:(unsigned)age {
    return ...;
}

-(BOOL)isPerson0lder:(Person *)otherPerson {
    return ...;
}
@end
```
Instantiating a class

• First, call `alloc` class method to allocate actual memory.
• Next, call an initializer (`init`, `initWith...`).
Instantiating a class

• First, call `alloc` class method to allocate actual memory.
• Next, call an initializer (`init`, `initWith...`).

Allocating an instance of the Person class:

```objective-c
Person *person = [[Person alloc] init];
```
Instantiating a class

• First, call `alloc` class method to allocate actual memory.
• Next, call an initializer (`init`, `initWith...`).

Allocating an instance of the Person class:

```objective-c
Person *person = [[Person alloc] init];
```

Using different initializers:

```objective-c
NSString *person = [[Person alloc] init];
NSArray *array = [[NSArray alloc] initWithObject:person];
```
Properties

• You should never allow clients to access instance variables directly.
Properties

• You should never allow clients to access instance variables directly.

In Person.h:

```objc
@interface Person : NSObject {
    NSString * name;
    NSUInteger age;
}
@end
```
Properties

• You should never allow clients to access instance variables directly.

In Person.h:

```swift
@interface Person : NSObject {
    NSString * name;
    NSUInteger age;
}
@end
```

In a client using the Person class:

```swift
Person *jim = ...; // created Person
NSLog(@"%@ is %u years old.", jim->name, jim->age);
jim->age = 30;
```
Properties

• You should never allow clients to access instance variables directly.

In Person.h:

```objc
@interface Person : NSObject {  
    NSString * name;  
    NSUInteger age;  
}
@end
```

In a client using the Person class:

```c
Person *jim = ...; // created Person  
NSLog(@"%@ is %u years old.", jim->name, jim->age);  
jim->age = 30;
```
Properties

• Create getters and setters instead.
Properties

• Create getters and setters instead.

In `Person.h`:

```c
@interface Person : NSObject {
    NSString * name;
    NSUInteger age;
}
@end
```
Properties

• Create getters and setters instead.

In Person.h:

```c
@interface Person : NSObject {
    NSString * name;
    NSUInteger age;
}
@end
```

In a client using the Person class:

```c
Person *jim = ...; // created Person
NSLog("%@ is %u years old.", [jim name], [jim age]);
[jim setAge:30];
```
Properties

• Create getters and setters instead.

In Person.h:

```
@interface Person : NSObject {
    NSString * name;
    NSUInteger age;
}

-(NSString *)name;
-(void)setName:(NSString *)newName;
-(unsigned)age;
-(void)setAge:(NSUInteger)age;
@end
```

In a client using the Person class:

```
Person *jim = ...; // created Person
NSLog(@"%@ is %u years old.\n", [jim name], [jim age]);
[jim setAge:30];
```
Properties

• If you have lots of instance variables to give access to, declaring \textit{and} implementing all the methods is a pain.

• Objective-C properties “write” the methods for you.
Properties

• If you have lots of instance variables to give access to, declaring \textit{and} implementing all the methods is a pain.
• Objective-C properties “write” the methods for you.
• Parts to properties
Properties

• If you have lots of instance variables to give access to, declaring and implementing all the methods is a pain.
• Objective-C properties “write” the methods for you.
• Parts to properties
  ▪ Property declaration with name, type, and modifiers.
Properties

• If you have lots of instance variables to give access to, declaring *and* implementing all the methods is a pain.

• Objective-C properties “write” the methods for you.

• Parts to properties
  ▪ Property declaration with name, type, and modifiers.
  ▪ Instance variables with same name.
Properties

• If you have lots of instance variables to give access to, declaring *and* implementing all the methods is a pain.

• Objective-C properties “write” the methods for you.

• Parts to properties
  ▪ Property declaration with name, type, and modifiers.
  ▪ Instance variables with same name.
  ▪ Property synthesization in implementation.
Properties
Properties

In Person.h:
@interface Person : NSObject {
    NSString * name;
    NSUInteger age;
}
@property(nonatomic, copy) NSString *name;
@property(nonatomic, assign) NSUInteger age;
@end
Properties

In Person.h:

```c
@interface Person : NSObject {
    NSString * name;
    NSUInteger age;
}
@property(nonatomic, copy) NSString *name;
@property(nonatomic, assign) NSUInteger age;
@end
```

In a client using the Person class:

```c
Person *jim = ...; // created Person
NSLog("@"%@ is %u years old.", [jim name], [jim age]);
[jim setAge:30];
```
Properties

In Person.h:
@interface Person : NSObject {
    NSString * name;
    NSUInteger age;
}
@property(nonatomic, copy) NSString *name;
@property(nonatomic, assign) NSUInteger age;
@end

In a client using the Person class:
Person *jim = ...; // created Person
NSLog(@"%@ is %u years old.", [jim name], [jim age]);
[jim setAge:30];
Properties

In Person.h:

@interface Person : NSObject {
    NSString * name;
    NSUInteger age;
}
@property(nonatomic, copy) NSString *name;
@property(nonatomic, assign) NSUInteger age;
@end

In Person.m

@implementation
@synthesize name, age;
@end // we are leaking name/age, but ignore that for now.

In a client using the Person class:

Person *jim = ...; // created Person
NSLog(@"%@ is %u years old.", [jim name], [jim age]);
[jim setAge:30];
Properties

• Unwritten mandatory rule:

Never use “get” some property (e.g. getName)!
Properties

• Unwritten mandatory rule:

Never use “get” some property (e.g. getName)!

@interface Person : NSObject {
    NSString * name;
    NSUInteger age;
}
- (NSString *)getName;
@end

@interface Person : NSObject {
    NSString * name;
    NSUInteger age;
}
- (NSString *)name;
@end
Properties

• Unwritten mandatory rule:

Never use “get” some property (e.g. getName)!

@interface Person : NSObject {
    NSString * name;
    NSUInteger age;
}
- (NSString *)getName;
@end

@interface Person : NSObject {
    NSString * name;
    NSUInteger age;
}
- (NSString *)name;
@end
Properties

- More in Advanced Objective-C talk.
Categories

• Allow you to extend classes — even if they are precompiled.
• This is extremely powerful!
In some header or implementation file:

```objc
@interface Person (MyCategory)
- (void)someExtraMethod;
@end
```
Categories

Example

In some header or implementation file:

```objc
@interface Person (MyCategory)
- (void)someExtraMethod;
@end
```

Name of category
Categories

Example

In some header or implementation file:

```objective-c
@interface Person (MyCategory)
- (void)someExtraMethod;
@end
```

Instance (or class) methods
Categories

Example

In some header or implementation file:

```cpp
@interface Person (MyCategory)
- (void)someExtraMethod;
@end
```

In some implementation file:

```cpp
@implementation Person (MyCategory)
- (void)someExtraMethod {
    ...; // do something
}
@end
```
Categories

• But why?
  ▪ This will be covered in the Advanced Objective-C talk.
Protocols

• A set of methods a class is promised to implement.
• Similar to interfaces in Java.
Protocols

• A set of methods a class is promised to implement.
• Similar to interfaces in Java.
• Real-life example:
  ▪ Instead of C++ copy constructor, there is NSCopying protocol.
  ▪ Classes adopting NSCopying protocol implement copyWithZone method.
  ▪ When object is copied, if class adopts NSCopying, then send object copyWithZone message.
Protocols

NSCopying example

In Person.h:

```swift
@interface Person : NSObject <NSCopying> {
    NSString * name;
    NSUInteger age;
}
@end
```

- (id)initWithName:(NSString *)name andAge:(NSUInteger)age;
- (BOOL)isPersonOlder:(Person *)otherPerson;

Put all protocols
Protocols
NSCopying example

In Person.m:

@implementation

- (id)copyWithZone:(NSZone *)zone {
    return ...;
}

@end
Protocols

Forcing compiler to check protocol for an object:

- (void)setCopyingCompliantObject:(id<NSCopying>)anObject;
Demo
Introduction to Objective-C

What are object-oriented systems? ✓

What is the Objective-C language? ✓

What are objects? ✓

How do you create classes in Objective-C? ✓

acm.uiuc.edu/macwarriors/devphone
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What are object-oriented systems?</td>
<td>✓</td>
</tr>
<tr>
<td>What is the Objective-C language?</td>
<td>✓</td>
</tr>
<tr>
<td>What are objects?</td>
<td>✓</td>
</tr>
<tr>
<td>How do you create classes in Objective-C?</td>
<td>✓</td>
</tr>
</tbody>
</table>
About These Slides

If you need to reference the content of this presentation, these slides will be posted on our website, listed below:

acm.uiuc.edu/macwarriors/devphone
macwarriors