Digging into Objective-C

Kevin Cathey
Digging into Objective-C

acm.uiuc.edu/macwarriors/devphone
Digging into Objective-C

Memory management

acm.uiuc.edu/macwarriors/devphone
Digging into Objective-C

Memory management

Uses of categories
Digging into Objective-C

- Memory management
- Uses of categories
- More on properties

acm.uiuc.edu/macwarriors/devphone
Digging into Objective-C

- Memory management
- Uses of categories
- More on properties
- Common Objective-C compiler directives
Digging into Objective-C

- Memory management
- Uses of categories
- More on properties
- Common Objective-C compiler directives
- Exceptions

acm.uiuc.edu/macwarriors/devphone
<table>
<thead>
<tr>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory management</td>
</tr>
<tr>
<td>Uses of categories</td>
</tr>
<tr>
<td>More on properties</td>
</tr>
<tr>
<td>Common Objective-C compiler directives</td>
</tr>
<tr>
<td>Exceptions</td>
</tr>
<tr>
<td>Delegation</td>
</tr>
</tbody>
</table>
Digging into Objective-C

- Memory management
- Uses of categories
- More on properties
- Common Objective-C compiler directives
- Exceptions
- Delegation

acm.uiuc.edu/macwarriors/devphone
Reference counting
Reference counting

- Reference count is how many people claim ownership of an object.
Reference counting

• Reference count is how many people claim ownership of an object.
• If you want an object you didn’t create to stay around, you retain (increases reference count by 1).
Reference counting

• Reference count is how many people claim ownership of an object.
• If you want an object you didn’t create to stay around, you retain (increases reference count by 1).
• When done with an object you’ve created or retained, you release (decrease reference count by 1).
Reference counting

- Reference count is how many people claim ownership of an object.
- If you want an object you didn’t create to stay around, you retain (increases reference count by 1).
- When done with an object you’ve created or retained, you release (decrease reference count by 1).
- When reference count hits 0, object is deallocated.
An example
Keeping track of an object

Memory
An example
Keeping track of an object
An example
Keeping track of an object

In Foo.m

```c
NSArray *someArray = [[NSArray alloc] init];
```
An example
Keeping track of an object

In Bar.m

[someArray retain];
An example
Keeping track of an object

In Bar.m
[someArray retain];
An example
Keeping track of an object

In Foo.m
[someArray release];
An example
Keeping track of an object

In Foo.m

[someArray release];
An example
Keeping track of an object

NSArray

Bar Class

Memory
An example
Keeping track of an object

In Bar.m

[someArray release];
An example
Keeping track of an object

In Bar.m
[someArray release];
An example
Keeping track of an object

NSMutableArray 0

Memory
An example

Keeping track of an object

Memory
To review

• For objects you don’t create (e.g. get from methods)
  ▪ **Retain** only when saving to instance variable (or static variable).
  ▪ **Release** only when explicitly told so, or if you retained it by saving it (as in above case).
To review

• For objects you don’t create (e.g. get from methods)
  • **Retain** only when saving to instance variable (or static variable).
  • **Release** only when explicitly told so, or if you retained it by saving it (as in above case).

---

**Instance when you should retain:**

```objective-c
- (void)someMethod {
    id myObject = [someArray objectAtIndex:...];
    myInstanceVariable = [myObject retain];
    ...
}

- (void)dealloc {
    [myInstanceVariable release];
    [super dealloc];
}
```
To review

- For objects you create with `[[SomeClass alloc] init]` or `[myInstance copy]` (without autoreleasing)
  - **Retain** should not need to be called.
  - **Release** when you are done using it.
To review

• For objects you create with `[[SomeClass alloc] init]` or `myInstance copy` (without autoreleasing)
  ▪ **Retain** should not need to be called.
  ▪ **Release** when you are done using it.

Instance where you don’t need to retain

```objective-c
- (void)someMethod {
    NSArray *someArray = [[NSArray alloc] init];
    myInstanceVariable = someArray;
    ...
}

- (void)dealloc {
    [myInstanceVariable release];
    [super dealloc];
}
```
To review

• Match every **retain** with a **release**: `init` and `copy` both count as implicit retains.

• If you don’t know whether you should retain or not, don’t because over-releasing is easier to debug than over-retaining.
Autoreleasening

- What if you create an object and you are returning it from a method, how would you be able to release it?

- (NSArray *)objects {
  NSArray *myArray = [[NSArray alloc] initWithObject:myObject];
  return myArray;
}

- (NSArray *)objects {
  NSArray *myArray = [[NSArray alloc] initWithObject:myObject];
  return [myArray autorelease];
}
Autoreleasing

- What if you create an object and you are returning it from a method, how would you be able to release it?

```objc
- (NSArray *)objects {
    NSArray *myArray = [[NSArray alloc] initWithObject:myObject];
    return myArray;
}
```

```objc
- (NSArray *)objects {
    NSArray *myArray = [[NSArray alloc] initWithObject:myObject];
    return [myArray autorelease];
}
```
Autoreleaseng

• What if you create an object and you are returning it from a method, how would you be able to release it?

```objective-c
- (NSArray *)objects {
    NSArray *myArray = [[NSArray alloc] initWithObject:myObject];
    return myArray;
}
```

```objective-c
Leaks!
- (NSArray *)objects {
    NSArray *myArray = [[NSArray alloc] initWithObject:myObject];
    return [myArray autorelease];
}
```
Autoreleasing

- What if you create an object and you are returning it from a method, how would you be able to release it?

```
- (NSArray *)objects {
    NSArray *myArray = [[NSArray alloc] initWithObject:myObject];
    return myArray;
}
```

Leaks!

```
- (NSArray *)objects {
    NSArray *myArray = [[NSArray alloc] initWithObject:myObject];
    return [myArray autorelease];
}
```

Right
Autoreleasing

- Instead of explicitly releasing something, you mark it for a later release.
- An object called a release pool manages a set of objects to release when the pool itself is released.
- Add an object to the release pool by calling `autorelease`.
- In Cocoa, always guaranteed to have an autorelease pool.
Returning to our earlier example

Keeping track of an object

Memory
Returning to our earlier example

Keeping track of an object
Returning to our earlier example

Keeping track of an object
Returning to our earlier example

Keeping track of an object
Returning to our earlier example

Keeping track of an object

NSMutableArray

Memory
Returning to our earlier example
Keeping track of an object
Methods and autorelease

• Objects returned from methods (class or instance) are understood to be autoreleased.
• An exception: init and variants.
• Examples:
Methods and autorelease

- Objects returned from methods (class or instance) are understood to be autoreleased.
- An exception: `init` and variants.
- Examples:
Methods and autorelease

- Objects returned from methods (class or instance) are understood to be autoreleased.
- An exception: `init` and variants.
- Examples:

<table>
<thead>
<tr>
<th>Method</th>
<th>Is the object returned from this method autoreleased?</th>
</tr>
</thead>
<tbody>
<tr>
<td>- (NSArray *) allValues</td>
<td>✔</td>
</tr>
<tr>
<td>- (id) init</td>
<td>✘</td>
</tr>
<tr>
<td>+ (NSArray *) arrayWithObject:(id)obj</td>
<td>✔</td>
</tr>
<tr>
<td>+ (id) alloc</td>
<td>✘</td>
</tr>
</tbody>
</table>
### Methods and autorelease

- Objects returned from methods (class or instance) are understood to be autoreleased.
- An exception: `init` and variants.
- Examples:

<table>
<thead>
<tr>
<th>Method</th>
<th>Autoreleased?</th>
</tr>
</thead>
<tbody>
<tr>
<td>-(NSArray *)allValues;</td>
<td>✔</td>
</tr>
<tr>
<td>-(id)init;</td>
<td>✘</td>
</tr>
<tr>
<td>+ (NSArray *)arrayWithObject:(id)obj;</td>
<td>✔</td>
</tr>
<tr>
<td>+ (id)alloc;</td>
<td>✘</td>
</tr>
</tbody>
</table>
Methods and autorelease

- Objects returned from methods (class or instance) are understood to be autoreleased.
- An exception: `init` and variants.
- Examples:

<table>
<thead>
<tr>
<th>Method</th>
<th>Autoreleased?</th>
</tr>
</thead>
<tbody>
<tr>
<td>- (NSArray *) allValues;</td>
<td>✔</td>
</tr>
<tr>
<td>- (id) init;</td>
<td>✘</td>
</tr>
<tr>
<td>+ (NSArray *) arrayWithObject:(id)obj;</td>
<td>✔</td>
</tr>
<tr>
<td>- (id) alloc;</td>
<td>✘</td>
</tr>
</tbody>
</table>
Methods and autorelease

• Objects returned from methods (class or instance) are understood to be autoreleased.
• An exception: `init` and variants.
• Examples:

<table>
<thead>
<tr>
<th>Is the object returned from this method autoreleased?</th>
</tr>
</thead>
<tbody>
<tr>
<td>- <code>(NSArray *)allValues;</code></td>
</tr>
<tr>
<td>- <code>(id)init;</code></td>
</tr>
</tbody>
</table>

- (id)alloc; ❌

- (id)init; ❌
Methods and autorelease

- Objects returned from methods (class or instance) are understood to be autoreleased.
- An exception: `init` and variants.
- Examples:

<table>
<thead>
<tr>
<th>Method</th>
<th>Autoreleased</th>
</tr>
</thead>
<tbody>
<tr>
<td>(NSArray *)allValues;</td>
<td>✔</td>
</tr>
<tr>
<td>(id)init;</td>
<td>✘</td>
</tr>
<tr>
<td>+ (NSArray *)arrayWithObject:(id)obj;</td>
<td>✔</td>
</tr>
</tbody>
</table>
Methods and autorelease

- Objects returned from methods (class or instance) are understood to be autoreleased.
- An exception: `init` and variants.
- Examples:

<table>
<thead>
<tr>
<th>Method</th>
<th>Autoreleased</th>
</tr>
</thead>
<tbody>
<tr>
<td>-(NSArray *)allValues;</td>
<td>✔</td>
</tr>
<tr>
<td>-(id)init;</td>
<td>✘</td>
</tr>
<tr>
<td>+(NSArray *)arrayWithObject:(id)obj;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Methods and autorelease

- Objects returned from methods (class or instance) are understood to be autoreleased.
- An exception: `init` and variants.
- Examples:

<table>
<thead>
<tr>
<th>Method</th>
<th>Autoreleased</th>
</tr>
</thead>
<tbody>
<tr>
<td>- (NSArray *)allValues;</td>
<td>✔</td>
</tr>
<tr>
<td>- (id)init;</td>
<td>✗</td>
</tr>
<tr>
<td>+ (NSArray *)arrayWithObject:(id)obj;</td>
<td>✔</td>
</tr>
</tbody>
</table>
Methods and autorelease

• Objects returned from methods (class or instance) are understood to be autoreleased.
• An exception: `init` and variants.
• Examples:

<table>
<thead>
<tr>
<th>Method</th>
<th>Autoreleased</th>
</tr>
</thead>
<tbody>
<tr>
<td>-(NSArray *)allValues;</td>
<td>✔</td>
</tr>
<tr>
<td>-(id)init;</td>
<td>❌</td>
</tr>
<tr>
<td>+(NSArray *)arrayWithObject:(id)obj;</td>
<td>✔</td>
</tr>
<tr>
<td>+(id)alloc;</td>
<td>❌</td>
</tr>
</tbody>
</table>
Methods and autorelease

- Objects returned from methods (class or instance) are understood to be autoreleased.
- An exception: `init` and variants.
- Examples:

<table>
<thead>
<tr>
<th>Is the object returned from this method autoreleased?</th>
</tr>
</thead>
<tbody>
<tr>
<td>- (NSArray *)allValues;</td>
</tr>
<tr>
<td>- (id)init;</td>
</tr>
<tr>
<td>+ (NSArray *)arrayWithObject:(id)obj;</td>
</tr>
<tr>
<td>+ (id)alloc;</td>
</tr>
</tbody>
</table>
Autorelease semantics

- When saving autoreleased object to instance variable, retain.
- Adding object to collection automatically retains it.
Autorelease semantics

- When saving autoreleased object to instance variable, retain.
- Adding object to collection automatically retains it.

```plaintext
// Save autoreleased object to ivar.
NSDictionary *dict = [NSDictionary dictionary];
myIvar = [dict retain];

// Above same as:
myIvar = [[NSDictionary alloc] init];

// Adding autoreleased object to collection.
// ‘today’ not going away b/c dict retains.
NSDate *today = [NSDate date];
[dict setObject:today forKey:@“dateToday”];
```
Autorelease semantics

- When saving autoreleased object to instance variable, retain.
- Adding object to collection automatically retains it.

```
// Save autoreleased object to ivar.
NSDictionary *dict = [NSDictionary dictionary];
myIvar = [dict retain];

// Above same as:
myIvar = [[NSDictionary alloc] init];

// Adding autoreleased object to collection.
// ‘today’ not going away b/c dict retains.
NSDate *today = [NSDate date];
[dict setObject:today forKey:@“dateToday”];
```
Autorelease semantics

• When saving autoreleased object to instance variable, retain.
• Adding object to collection automatically retains it.

// Save autoreleased object to ivar.
NSDictionary *dict = [NSDictionary dictionary];
myIvar = [dict retain];

// Above same as:
myIvar = [[NSDictionary alloc] init];

// Adding autoreleased object to collection. 
// ‘today’ not going away b/c dict retains.
NSDate *today = [NSDate date];
[dict setObject:today forKey:@“dateToday”];
More on autorelease pools
More on autorelease pools

• One per thread.
  - If you spawn your own thread (POSIX, Mach, or Cocoa), you’ll have to create your own NSAutoreleasePool.
More on autorelease pools

• One per thread.
  • If you spawn your own thread (POSIX, Mach, or Cocoa), you’ll have to create your own NSAutoreleasePool.

• Stack based.

```objective-c
// Outer autorelease pool.
NSAutoreleasePool *outer = [[NSAutoreleasePool alloc] init];

// ... do something here

for (id obj in myCollection) {
    NSAutoreleasePool *inner = [[NSAutoreleasePool alloc] init];
    [obj doSomethingHuge];
    [inner drain];
}

[outer drain];
```
Writing your own classes: init & dealloc
Writing your own classes: init & dealloc

• **init**
  - Needs to call super. Period.
  - Setup instance variables.
  - Returns self.
Writing your own classes: init & dealloc

- **init**
  - Needs to call super. Period.
  - Setup instance variables.
  - Returns self.

```objective-c
@implementation MyClass

- (id)init {
    if ((self = [super init])) {
        myIVar = @"Hello";
    }
    return self;
}
@end
```
Writing your own classes: init & dealloc
Writing your own classes: init & dealloc

- **dealloc**
  - Never call explicitly.
  - Release all retained (or copied) ivars
  - Calls [super dealloc]
Writing your own classes: init & dealloc

- dealloc
  - Never call explicitly.
  - Release all retained (or copied) ivars
  - Calls [super dealloc]

@interface MyClass
  -(void)saveThis:(id)object {
    if (myIvar != object ) {
        [myIvar release];
        myIvar = [object retain];
    }
  }
@end
Digging into Objective-C

- Memory management ✓
- Uses of categories
- More on properties
- Common Objective-C compiler directives
- Exceptions
- Delegation

acm.uiuc.edu/macwarriors/devphone
Digging into Objective-C

- Memory management ✓
- Uses of categories
- More on properties
- Common Objective-C compiler directives
- Exceptions
- Delegation

acm.uiuc.edu/macwarriors/devphone
Why use categories?

• Add private methods to a class.
  ▪ Put category method in implementation file.
Why use categories?

• Add private methods to a class.
  • Put category method in implementation file.

In Foo.m

@interface Foo ()
- (void)myPrivateMethod;
@end

@implementation
- (void)myPrivateMethod {
 ...
}
@end
Why use categories?

• Extend a class’s functionality
  • `firstObject` on `NSArray`, for example.
Why use categories?

- Extend a class’s functionality
  - `firstObject` on NSArray, for example.

In `MyArrayAdditions.h`

```objc
@interface NSArray (MyArrayAdditions)
- (id)firstObject;
@end
```
Why use categories?

• Extend a class’s functionality
  • `firstObject` on NSArray, for example.

In `MyArrayAdditions.h`

```objc
@interface NSArray (MyArrayAdditions)
- (id)firstObject;
@end
```

In `MyArrayAdditions.m`

```objc
@implementation NSArray (MyArrayAdditions)
- (id)firstObject {
    return [self objectAtIndex:0];
}
@end
```
Why use categories?

- Delegation
  - Add methods to NSObject to call on your delegate.
Why use categories?

• Delegation
  ▪ Add methods to NSObject to call on your delegate.

In Foo.h

```objc
@interface Foo : NSObject {
}
...
@end

@interface NSObject (FooDelegation)
- (void)foo:(Foo *)foo didFindBar:(Bar *)bar;
@end
```
Why use categories?

- Delegation
  - Add methods to NSObject to call on your delegate.

In Foo.h

```objc
@interface Foo : NSObject {
}
...
@end

@interface NSObject (FooDelegation)
- (void)foo:(Foo *)foo didFindBar:(Bar *)bar;
@end
```

Tells compiler that there is someone who implements these methods. Allows you to call these are direct methods.
Why use categories?

- Delegation
  - Add methods to NSObject to call on your delegate.

In `Foo.m`

```objc
@implementation
- (void)someMethod {
    Bar *someBar = ...;
    [[self delegate] foo:self didFindBar:someBar];
}
@end
```
Properties

@property (nonatomic, readwrite, retain) MyClass *someProp;
Properties

@property (nonatomic, readwrite, retain) MyClass *someProp;

• Keywords
Properties

@property (nonatomic, readwrite, retain) MyClass *someProp;

• Keywords
  • nonatomic or atomic
Properties

@property (nonatomic, readwrite, retain) MyClass *someProp;

• Keywords
  ▪ nonatomic or atomic
    ▪ Only use nonatomic
Properties

@property (nonatomic, readwrite, retain) MyClass *someProp;

• Keywords
  • nonatomic or atomic
    • Only use nonatomic
    • Default is atomic, so change it
Properties

@property (nonatomic, readwrite, retain) MyClass *someProp;

• Keywords
  • nonatomic or atomic
    • Only use nonatomic
    • Default is atomic, so change it
  • readonly or readwrite
Properties

@property (nonatomic, readwrite, retain) MyClass *someProp;

• Keywords
  ▪ nonatomic or atomic
    ▪ Only use nonatomic
    ▪ Default is atomic, so change it
  ▪ readonly or readwrite
    ▪ Default is readwrite
Properties

@property (nonatomic, readwrite, retain) MyClass *someProp;

• Keywords
  ▪ nonatomic or atomic
    ▪ Only use nonatomic
    ▪ Default is atomic, so change it
  ▪ readonly or readwrite
    ▪ Default is readwrite
  ▪ assign, retain, copy
Properties

@property (nonatomic, readwrite, retain) MyClass *someProp;

- Keywords
  - nonatomic or atomic
    - Only use nonatomic
    - Default is atomic, so change it
  - readonly or readwrite
    - Default is readwrite
  - assign, retain, copy
    - Assign is default. Use for integers, floats, constants.
Properties

@property (nonatomic, readwrite, retain) MyClass *someProp;

• Keywords
  ▪ nonatomic or atomic
    ▪ Only use nonatomic
    ▪ Default is atomic, so change it
  ▪ readonly or readwrite
    ▪ Default is readwrite
  ▪ assign, retain, copy
    ▪ Assign is default. Use for integers, floats, constants.
    ▪ Retain, well, retains it. Use for all objects (except delegates)
Properties

@property (nonatomic, readwrite, retain) MyClass *someProp;

• Keywords
  ▪ nonatomic or atomic
    ▪ Only use nonatomic
    ▪ Default is atomic, so change it
  ▪ readonly or readwrite
    ▪ Default is readwrite
  ▪ assign, retain, copy
    ▪ Assign is default. Use for integers, floats, constants.
    ▪ Retain, well, retains it. Use for all objects (except delegates)
• Copy calls copy on the object. Use for strings.
Properties

@property (nonatomic, readwrite, retain) MyClass *someProp;

• Keywords
  ▪ nonatomic or atomic
    ▪ Only use nonatomic
    ▪ Default is atomic, so change it
  ▪ readonly or readwrite
    ▪ Default is readwrite
  ▪ assign, retain, copy
    ▪ Assign is default. Use for integers, floats, constants.
    ▪ Retain, well, retains it. Use for all objects (except delegates)
• Copy calls copy on the object. Use for strings.
• Don’t need if using readonly.
Common Compiler Directives

- `@class`
  - Forward class declaration.
  - Don’t have to import header, just say that class exists, I promise.
  - If you are using a class for a variable or method return/parameter type, forward declare it.
Common Compiler Directives

- `@class`
  - Forward class declaration.
  - Don’t have to import header, just say that class exists, I promise.
  - If you are using a class for a variable or method return/parameter type, forward declare it.

In `Foo.h`

```objc
@class Bar;
@interface Foo : NSObject {
  Bar *myBar;
}
@end
```

No need for `#import` since you are not calling any methods on Bar. You are just saying there is a class that exists somewhere.
Common Compiler Directives

- `@selector`
  - Creates a selector.
  - `@selector(doSomething:withObject:)`
Common Compiler Directives

- @""
  - Constant string creator.
  - Created at compile time.
  - Release, retain, autorelease do nothing.
Common Compiler Directives

- `@""`
  - Constant string creator.
  - Created at compile time.
  - Release, retain, autorelease do nothing.

Do this...

```c
NSString *myString = @"Some string";
```

Instead of...

```c
NSString *myString = [NSString stringWithCString...];
```
Architecture Independence

- For constant-type variables, use 64-bit types.
  - For integers: `NSUInteger` and `NSInteger`
  - For reals: `CGFloat`
- These are simply typedefs (not classes), and are changed depending on whether you are building 64-bit or not.
- Avoid `float`, `int`, `unsigned` if you can (if you have to, use specific size types: `uint8_t`, `int32_t`, etc).
Exceptions
Exceptions

• When an exception is thrown, it is a cause of programmer error, not user error. This is vastly different from Java.
Exceptions

- When an exception is thrown, it is a cause of **programmer error**, not user error. This is vastly different from Java.
- Very few uses for `@try` `@catch` blocks.
Exceptions

• When an exception is thrown, it is a cause of **programmer error**, not user error. This is vastly different from Java.
• Very few uses for `@try @catch` blocks.
• If you get an exception, it’s your fault.
Delegation

• One of the greatest features of Objective-C.
• One object sends another messages without knowing anything else about that object.
• This is how classes like UITableView work to get information.
Delegation

Example
Delegation
Example

MyObject

<table>
<thead>
<tr>
<th>NSArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jack</td>
</tr>
<tr>
<td>Kate</td>
</tr>
<tr>
<td>Richard</td>
</tr>
<tr>
<td>Desmond</td>
</tr>
<tr>
<td>Hurley</td>
</tr>
</tbody>
</table>

PeopleTableView
delegate

People Table
Delegation
Example

MyObject

NSArray
Jack
Kate
Richard
Desmond
Hurley

PeopleTable

delegate
MyObject

People Table
Delegation Example

MyObject

NSArray
- Jack
- Kate
- Richard
- Desmond
- Hurley

PeopleTableViewCell

delegate:
- MyObject

tableViewNumberOfNames:

People Table
Delegation Example

**MyObject**
- NSArray
  - Jack
  - Kate
  - Richard
  - Desmond
  - Hurley

**PeopleTableView**
- delegate: MyObject

**Methods**
- tableViewNumberOfNames:
- tableView:nameAtIndex:
Delegation
Example

MyObject

- NSArray
  - Jack
  - Kate
  - Richard
  - Desmond
  - Hurley

PeopleTableView

- delegate
  - MyObject

- tableView:numberOfNames:
- tableView:nameAtIndex:
- tableView:didSelectName:
Demo
## Digging into Objective-C

<table>
<thead>
<tr>
<th>Topic</th>
<th>Complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory management</td>
<td>✔️</td>
</tr>
<tr>
<td>Uses of categories</td>
<td>✔️</td>
</tr>
<tr>
<td>More on properties</td>
<td>✔️</td>
</tr>
<tr>
<td>Common Objective-C compiler directives</td>
<td>✔️</td>
</tr>
<tr>
<td>Exceptions</td>
<td>✔️</td>
</tr>
<tr>
<td>Delegation</td>
<td>✔️</td>
</tr>
</tbody>
</table>

[acm.uiuc.edu/macwarriors/devphone](acm.uiuc.edu/macwarriors/devphone)
# Digging into Objective-C

<table>
<thead>
<tr>
<th>Topic</th>
<th>Complete?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory management</td>
<td>✓</td>
</tr>
<tr>
<td>Uses of categories</td>
<td>✓</td>
</tr>
<tr>
<td>More on properties</td>
<td>✓</td>
</tr>
<tr>
<td>Common Objective-C compiler directives</td>
<td>✓</td>
</tr>
<tr>
<td>Exceptions</td>
<td>✓</td>
</tr>
<tr>
<td>Delegation</td>
<td>✓</td>
</tr>
</tbody>
</table>

[acm.uiuc.edu/macwarriors/devphone](acm.uiuc.edu/macwarriors/devphone)
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
macwarriors
macwarriors