Digging into Objective-C

Kevin Cathey

Roadmap

How do you dig into Objective-C?

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

Memory Management

The Thanksgiving Turkey

Reference counting

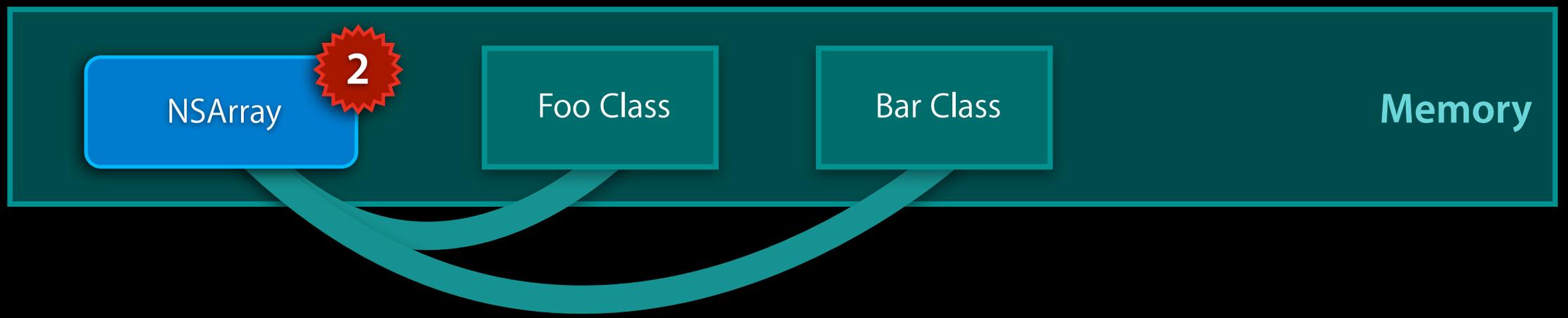
- Reference count keeps track of how many people claim ownership of that object.
- If you want an object you didn't create to stay around, you say so by retaining (increases reference count by 1).
- When you are done with an object you've created or retained, you remove responsibility (decrease reference count by 1).
- When an object reaches a reference count of 0, it's finally actually deallocated.

An example

Keeping track of an object

An example

Keeping track of an object

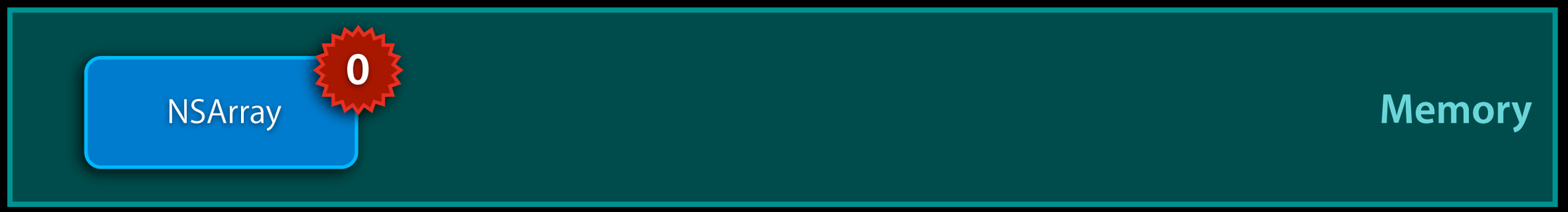


An example

Keeping track of an object

An example

Keeping track of an object



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).
- 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.
- 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.

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:myOb
    return myArray;
}
```

Leaks!

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

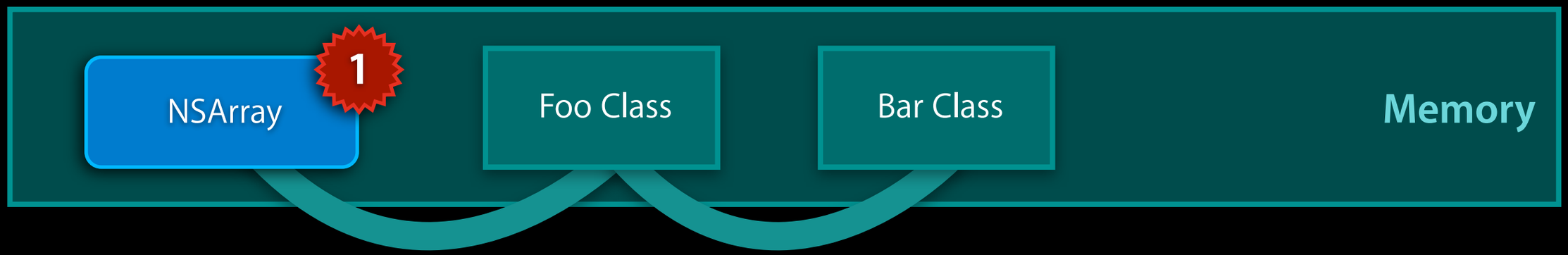
Right

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.

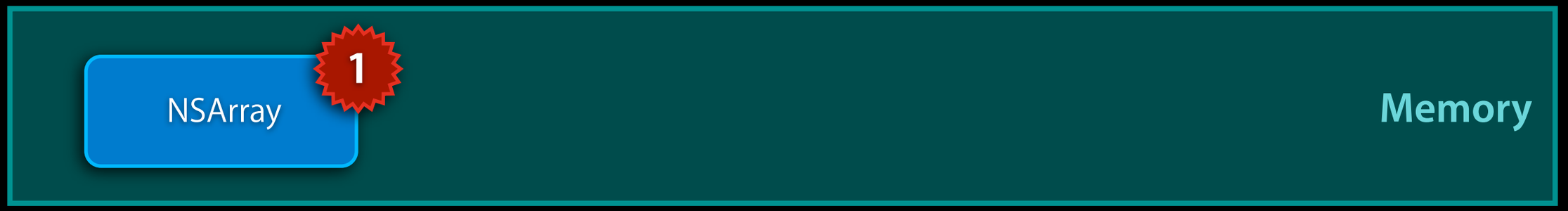
Returning to our earlier example

Keeping track of an object



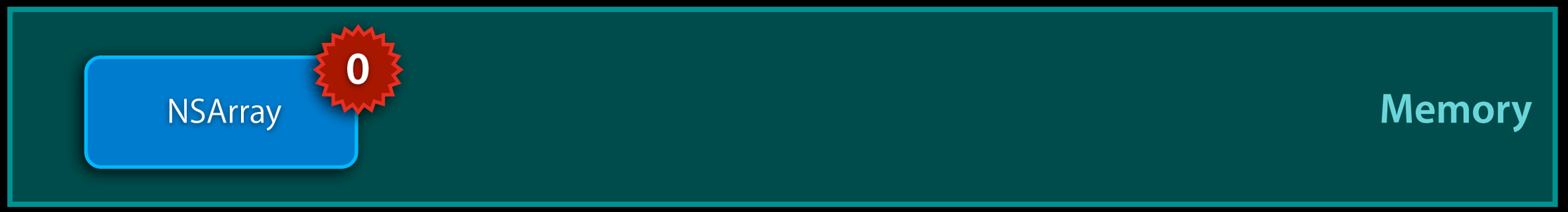
Returning to our earlier example

Keeping track of an object



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:

Is the object returned from this method autoreleased?	
`- (NSArray *)allValues;`	✔
`- (id)init;`	✘
`+ (NSArray *)arrayWithObject:(id)obj;`	✔
`+ (id)alloc;`	✘

Autorelease semantics

- When you have an autoreleased object and you want to save it to an instance variable, retain it.
- When you create an autoreleased object and add it to a collection, it is retained.

```
// 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

- One per thread.
  - If you spawn your own thread (POSIX, Mach, or Cocoa), you'll have to create your own NSAutoreleasePool.
- Stack based.

```
// 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

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

```
@implementation MyClass

- (id)init {
    if ((self = [super init])) {
        myIVar = @”Hello”;
    }
    return self;
}
@end
```

Writing your own classes: init & dealloc

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

```
@implementation MyClass
- (void)saveThis:(id)object {
    if (myIvar != object ) {
        [myIvar release];
        myIvar = [object retain];
    }
}

- (void)dealloc {
    [object release];
    [super dealloc];
}
@end
```

Everything Else

The gravy, stuffing, and mashed potatoes

Why use categories?

- Add private methods to a class.
  - Put category method in implementation file.
- Extend a class's functionality
  - `firstObject` on NSArray, for example.
- Delegation
  - Add methods to NSObject to call on your delegate.

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.
- `@selector`
  - Creates a selector.
  - @selector(doSomething:withObject:)
- `@“”`
  - Constant string creator.
  - Created at compile time.
  - Release, retain, autorelease do nothing.

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.

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 asks another for information or tells of what it is doing, and the asking class does not need to know anything about the delegate.
- This is how classes like `UITableView` work to get information.