Collections & Memory Management

Lecture 3
Collections
NSArray

a list of objects in order

[array objectAtIndex:0]  [array objectAtIndex:3]

Counting starts at zero, not one
NSSet

a bunch of objects with no order

[set containsObject:purpleSquare] == YES
NSCountedSet

a bunch of objects with no order, duplicates allowed

[countedSet countForObject:blueSquare] == 2
an association between keys and values

greenSquare == [dictionary objectForKey:blueSquare]
cocoa  |ˈkōkō|
noun
1 a chocolate powder made from roasted and ground cacao seeds.
   • a hot drink made from such a powder mixed with sugar and milk or water.
2 variant spelling of coco, usu. regarded as a misspelling.
ORIGIN early 18th cent. (denoting cacao seed): alteration of cacao.
Mutability

- For efficiency reasons, collections are by default immutable: unchangeable after their creation
- To create a mutable collection, use NSMuttableWhatever
- To change immutable to mutable, use mutableCopy
Mutability

Added methods for mutable versions

NSMutableArray:

[array addObject: someObj]

NSMutableDictionary:

[dict setObject: obj forKey: key]

For more info, see the documentation
Enumeration

accessing each element in a collection

```
NSErrornerator *enumerator = [collection objectEnumerator];
MyObject *object;
while (object = [enumerator nextObject])
{
    /* do something with object */
}
```

The old, verbose, slow way
Fast Enumeration

accessing each element in a collection, fast

for (MyObject *object in collection) {
    /* do something with object */
}

The new, concise, fast way (10.5+)
Don’t Subclass Collections

MyArray : NSArray
{
    ... 
}

MyArray : NSObject
{
    NSArray *ivar;
}

Don’t Subclass Collections

MyArray : NSArray
{
    ...  
}

MyArray : NSObject
{
    NSArray *ivar;
}

Wrong:

MyArray : NSArray
{
    ...  
}

Wrong:

MyArray : NSObject
{
    NSArray *ivar;
}
Don’t Subclass Collections

MyArray : NSArray
{
    ...
}

MyArray : NSObject
{
   NSArray *ivar;
}✓

"There is typically little reason to subclass NSArray. The class does well what it is designed to do—maintain an ordered collection of objects."
– Apple Documentation
A Few Handy Foundation Classes
NSString

Useful methods:

+ (id) stringWithFormat: (NSString *) format, ...;
- (id) initWithFormat: (NSString *) format, ...;
- (NSString *) lowercaseString;
- (NSUInteger) length;
- (int) intValue; // floatValue, doubleValue, boolValue
- (BOOL) isEqualToString: (NSString *) aString;
- (NSString *) substringToIndex: (NSUInteger) index;
- (NSRange) rangeOfString: (NSString *) aString;
- (BOOL) hasPrefix: (NSString *) aString;
NSNumber

Useful for putting numbers in collections (since collections must contain NSObjects)

+ (id) numberWithInt: (int) i;
+ (id) numberWithDouble: (double) d;
- (id) initWithFloat: (float) f;
- (id) initWithBool: (BOOL) b;
- (char) charValue;
- (long) longValue;
NSDate

Used to represent "a single point in time"

+ (id) date;
+ (id) dateWithTimeIntervalSinceNow: (NSTimeInterval) seconds;
+ (id) distantFuture;
+ (id) distantPast;
- (id) dateByAddingTimeInterval: (NSTimeInterval) seconds;
- (NSDate *) earlierDate: (NSDate *) otherDate;
- (NSDate *) laterDate: (NSDate *) otherDate;
Memory Management
Memory Management
Memory Management

- C — malloc/free
Memory Management

- C – malloc/free
- Java – Garbage collection
Memory Management

• C — malloc/free
• Java — Garbage collection
• Obj-C — "Reference counting"
Reference Counting

• Every object has a reference count

• `+alloc` and `-copy` set the reference count to 1

• Use `retain` and `release` to add and subtract from the count

• Cocoa calls `dealloc` when the reference count reaches 0
The Rules
The Rules

[Class alloc]
The Rules

[Class alloc]
[object copy]
The Rules

[Class alloc]
[object copy]
[object retain]
The Rules

[Class alloc]
[object copy] → [object release]
[object retain]
MyObject *object = [[[MyObject alloc] init];

/* Insert lots of code here... */

[object release];
The Rules

MyObject *object = [otherObject copy];

/* Code */

[object release];
The Rules

MyObject *object = [controller myObject];

/* More code */

/* no release */
The Rules

MyObject *object = [controller myObject];
[object retain]; /* gain ownership */

/* Do work here */

[object release];
The Rules

MyObject *object = [controller myObject];
object = [object copy]; /* gain ownership */

/* Lots of code */

[object release];
Reference Counting
Reference Counting

```
Dog *fido = [[Dog alloc] init];
```
Reference Counting

Dog *fido = [[Dog alloc] init];

[fido retain];
Dog *fido = [[[Dog alloc] init];
[fido release];
[fido retain];
Dog *fido = [[Dog alloc] init];
[fido release];

[fido retain];
[fido release];
Returning Objects

• Callers of your method should not need to release the returned object

• You can return an object you own (retain’d, copy’d, or alloc’d), as long as you release it later
Returning Objects

return instanceVariable;

return [instanceVariable copy];
Returning Objects

return instanceVariable;  ✓

return [instanceVariable copy];  ✗
Autorelease

+ (NSString *)stringWithString:(NSString *)aString
{
}

Goal: return a copy of aString
Autorelease

+ (NSString *)stringWithString:(NSString *)aString
{
    return [[[NSString alloc] initWithString:aString];
}

Goal: return a copy of aString
+ (NSString *)stringWithString:(NSString *)aString {
    return [[[NSString alloc] initWithString:aString] autorelease];
}
Autorelease

+ (NSString *)stringWithString:(NSString *)aString
{
    return [[[NSString alloc] initWithString:aString] autorelease];
}

alloc leaves caller with responsibility to release the object

Goal: return a copy of aString
Autorelease

+ (NSString *)stringWithString:(NSString *)aString
{
    NSString *newString =
        [[[NSString alloc] initWithString:aString];
    [newString release];
    return newString;
}

Goal: return a copy of aString
Autorelease

+ (NSString *)stringWithString:(NSString *)aString
{
    NSString *newString =
        [[[NSString alloc] initWithString:aString];
          [newString release];   X
    return newString;
}

Goal: return a copy of aString
Autorelease

+ (NSString *)stringWithString:(NSString *)aString
{
    NSString *newString =
        [[NSString alloc] initWithString:aString];
    [newString release];
    return newString;
}

Goal: return a copy of aString

✗ after calling release, newString gets deallocated
Autorelease

+ (NSString *)stringWithString:(NSString *)aString
{
    NSString *newString =
        [[[NSString alloc] initWithString:aString];

    [newString autorelease];
    return newString;
}
Autorelease

+ (NSString *)stringWithString:(NSString *)aString {
    NSString *newString =
        [[[NSString alloc] initWithString:aString] autorelease];
    [newString autorelease]; ✓
    return newString;
}
Autorelease

+ (NSString *)stringWithString:(NSString *)aString
{
    NSString *newString = 
        [[[NSString alloc] initWithString:aString] autorelease];
    [newString autorelease];
    return newString;
}

autorelease releases it sometime later so
you can return newString without fear of deallocation, but
without the responsibility of releasing it later
Autorelease — How does it work?
Autorelease — How does it work?

Magic!
Autorelease — How does it work?

- Every time Cocoa calls your code, there is an NSAutoreleasePool allocated.
- autorelease throws your object into the most recently allocated pool.
- When an NSAutoreleasePool is deallocated, it releases all of its objects.
Collections and Memory Management

- Cocoa collections retain the objects they contain
- When the collection is deallocated, the objects are release’d with it
dealloc

• Method defined on NSObject
• Called *automatically* when an object's retain count reaches zero
  • You should NEVER invoke dealloc directly
  • E.g.: [object dealloc];
 dealloc

• Method defined on NSObject
• Called automatically when an object's retain count reaches zero
• You should NEVER invoke dealloc directly
• E.g.: [object dealloc];
Overriding dealloc

• If instances of a class will retain objects until the instance is deallocated, those retained objects cannot be safely released until deallocation.

• Typical case - objects as instance variables

• Use dealloc to release objects that are retained upon object creation (and not released earlier)
Overriding dealloc

- (void) dealloc
{
    [object release];
    [otherObject release];
    [ivar release];
    /* ... */
    [super dealloc];
}