Blocks, Memory Management, Delegation

lecturer: Davidov Dmitriy

Memory Management

Memory Management Policy

Ownership
To make sure it is clear when you own an object and when you do not, Cocoa sets the following policy:

You own any object you create
You create an object using a method whose name begins with "**alloc**", "**new**", "**copy**", or "**mutableCopy**" (for example, alloc, newObject, or mutableCopy).

You can take ownership of an object using retain
A received object is normally guaranteed to remain valid within the method it was received in, and that method may also safely return the object to its invoker. You use retain in two situations: (1) In the implementation of an accessor method or an init method, to take ownership of an object you want to store as a property value; and (2) To prevent an object from being invalidated as a side-effect of some other operation (as explained in Avoid Causing Deallocation of Objects You're Using).

When you no longer need it, you must relinquish ownership of an object you own
You relinquish ownership of an object by sending it a **release** message or an **autorelease** message. In Cocoa terminology, relinquishing ownership of an object is therefore typically referred to as "**releasing**" an object.

You must not relinquish ownership of an object you do not own
This is just corollary of the previous policy rules, stated explicitly.

Ownership Policy Is Implemented Using Retain Counts

The ownership policy is implemented through reference counting—typically called "retain count" after the retain method. Each object has a retain count.

When you create an object, it has a retain count of 1.

When you send an object a retain message, its retain count is incremented by 1.

When you send an object a release message, its retain count is decremented by 1.

When you send an object a autorelease message, its retain count is decremented by 1 at the end of the current autorelease pool block.

If an object's retain count is reduced to zero, it is deallocated.

Automatic Reference Counting (ARC)

In Objective-C and Swift programming, Automatic Reference Counting (ARC) is a memory management enhancement where the burden of keeping track of an object's reference count
is lifted from the programmer to the compiler.

Property Declarations

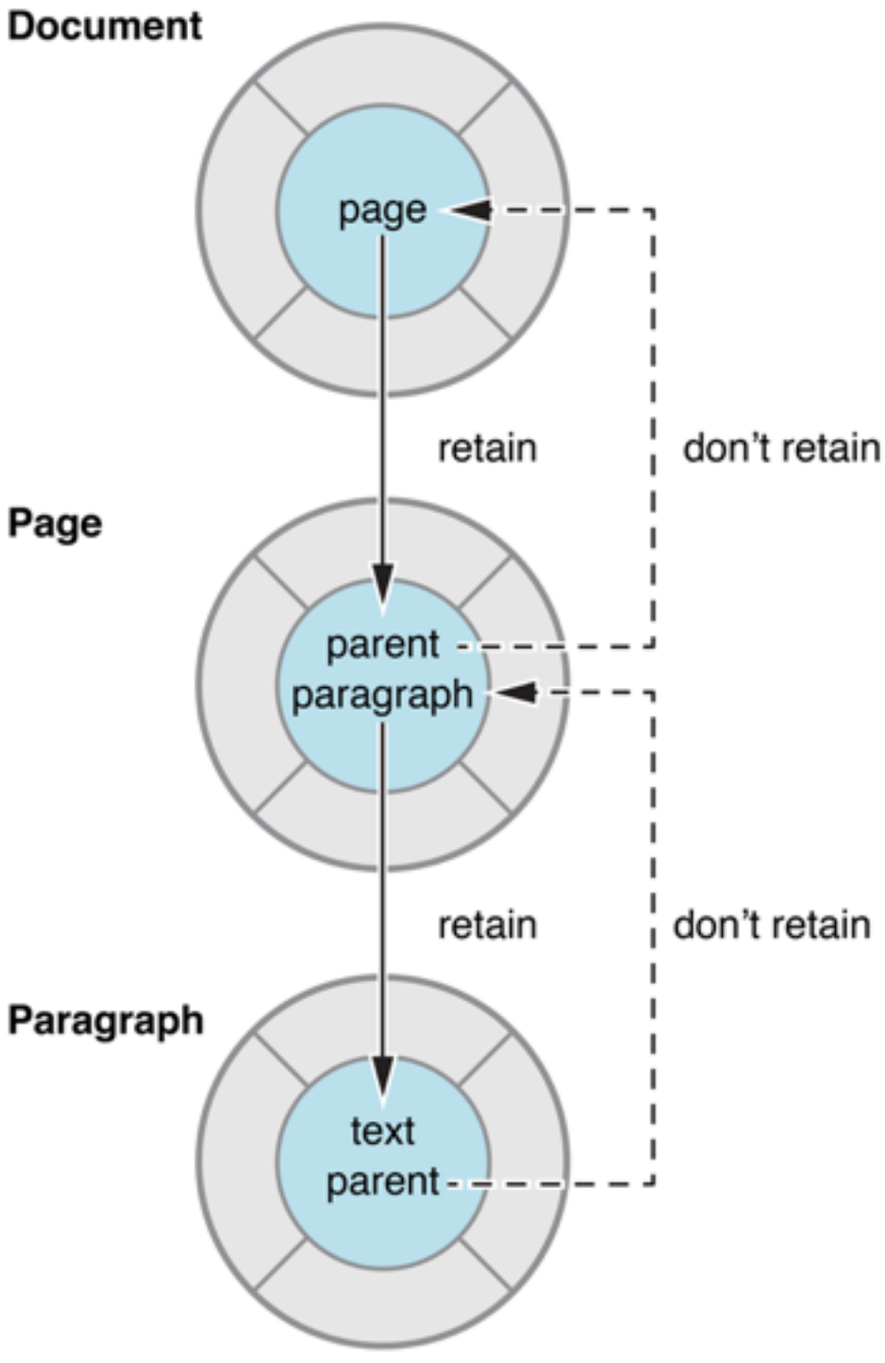
ARC introduces some new property declaration attributes, some of them replacing the old attributes.

strong = retain
assign = unsafe_unretained
weak
copy is same

Automatic Reference Counting (ARC)

Use Weak References to Avoid Retain Cycles

Retaining an object creates a strong reference to that object. An object cannot be deallocated until all of its strong references are released. A problem, known as a retain cycle, can therefore arise if two objects may have cyclical references—that is, they have a strong reference to each other (either directly, or through a chain of other objects each with a strong reference to the next leading back to the first).



Automatic Reference Counting (ARC)

To use non-ARC files in ARC environment

Click the Project -> Build Phases Tab -> Compile Sources Section -> Double Click on the file name

Then add **-fno-objc-arc** to the popup window.

Likewise, if you want to include a file in ARC, you can use the **-fobjc-arc** flag.

Blocks

What is the "Block"?

Blocks are a **language-level feature** added to C, Objective-C and C++, which allow you to create distinct segments of code that can be passed around to methods or functions as if they were values. **Blocks are Objective-C objects**, which means they can be added to collections like NSArray or NSDictionary. They also have the ability to capture values from the enclosing scope, making them similar to closures or lambdas in other programming languages.

Block definition

As a local variable:
returnType (^blockName)(parameterTypes) = ^returnType(parameters) {...};

As a property:
@property (nonatomic, copy) returnType (^blockName)(parameterTypes);

As a method parameter:
- (void)someMethodThatTakesABlock:(returnType (^)(parameterTypes))blockName;

As an argument to a method call:
[someObject someMethodThatTakesABlock:^returnType (parameters) {...}];

As a typedef:
typedef returnType (^TypeName)(parameterTypes);
TypeName blockName = ^returnType(parameters) {...};

Block definition

```
- (void)testMethod {
    int anInteger = 42;
    void (^testBlock)(void) = ^{
        NSLog(@"Integer is: %i", anInteger);
    };
    testBlock();
}
```

> Integer is: 42

```
- (void)testMethod {
double (^multiplyTwoValues)(double, double) =
                          ^(double firstValue, double secondValue) {
                             return firstValue * secondValue;
                          };

double result = multiplyTwoValues(2,4);
NSLog(@"The result is %f", result);
}
```

> Integer is: 8

Block argument

A Block Should Always Be the Last Argument to a Method

```
- (void)beginTaskWithName:(NSString *)name
                  completion:(void(^)(void))callback;

[self beginTaskWithName:@"MyTask" completion:^{
        NSLog(@"The task is complete");
    }];
```

Complex block

```
void (^(^complexBlock)(void (^)(void)))(void) = ^ (void (^aBlock)(void)) {
    ...
    return ^{
        ...
    };
};
```

Blocks as Objects

Objects Use Properties to Keep Track of Blocks
The syntax to define a property to keep track of a block is similar to a block variable:

```
@interface XYZObject : NSObject
@property (copy) void (^blockProperty)(void);
@end
```

Note: You should specify copy as the property attribute, because a block needs to be copied to keep track of its captured state outside of the original scope. This isn't something you need to worry about when using Automatic Reference Counting, as it will happen automatically, but it's best practice for the property attribute to show the resultant behavior.

Blocks and Memory Management

Avoid Strong Reference Cycles when Capturing self

```
@interface XYZBlockKeeper : NSObject
@property (copy) void (^block)(void);
@end

- (void)configureBlock {
    self.block = ^{
        [self doSomething];    // capturing a strong reference to self creates a strong
reference cycle
    }; }
```

To avoid this problem, it's best practice to capture a weak reference to self, like this:

```
- (void)configureBlock {
    XYZBlockKeeper * __weak weakSelf = self;
    self.block = ^{
        [weakSelf doSomething];   // capture the weak reference to avoid the reference
cycle
    }; }
```

By capturing the weak pointer to self, the block won't maintain a strong relationship back to the XYZBlockKeeper object. If that object is deallocated before the block is called, the weakSelf pointer will simply be set to nil.

Blocks and Variables

Use __block Variables to Share Storage

```
__block int anInteger = 42;
void (^testBlock)(void) = ^{
    NSLog(@"Integer is: %i", anInteger);
};
anInteger = 84;
testBlock();
```

Because anInteger is declared as a __block variable, its storage is shared with the block declaration. This means that the log output would now show:

Integer is: 84

```
__block int anInteger = 42;
void (^testBlock)(void) = ^{
    NSLog(@"Integer is: %i", anInteger);
    anInteger = 100;
};
testBlock();
NSLog(@"Value of original variable is now: %i", anInteger);
```

This time, the output would show:
Integer is: 42
Value of original variable is now: 100

Blocks and Objective-C Objects

When a block is copied, it creates **strong references** to object variables used within the block. If you use a block within the implementation of a method:

If you access an **instance variable by reference**, a strong reference is made to **self**;
If you access an **instance variable by value**, a strong reference is made to the **variable**.

```
dispatch_async(queue, ^{
    // instanceVariable is used by reference, a strong reference is made to self
    doSomethingWithObject(instanceVariable);
});

id localVariable = instanceVariable;
dispatch_async(queue, ^{
    /*
      localVariable is used by value, a strong reference is made to localVariable
      (and not to self).
    */
    doSomethingWithObject(localVariable);
});
```

NSDictionary

The NSDictionary class declares the programmatic interface to objects that manage immutable associations of keys and values.

A key-value pair within a dictionary is called an entry. Each entry consists of one object that represents the key and a second object that is that key's value. Within a dictionary, the keys are unique. That is, no two keys in a single dictionary are equal (as determined by isEqual:).

- valueForKey:
- objectForKey:
allKeys
- enumerateKeysAndObjectsUsingBlock:

Modern code:
@{@"key" : @"value"};

KVC

Key-Value Coding

A **key is a string** that identifies a specific property of an object. Typically, a key corresponds to the name of an accessor method or instance variable in the receiving object.
Keys must use ASCII encoding, begin with a lowercase letter, and may not contain whitespace.
A key path is a string of dot separated keys that is used to specify a sequence of object properties to traverse. The property of the first key in the sequence is relative to the receiver, and each subsequent key is evaluated relative to the value of the previous property.

For example, the **key path address.street would get the value of the address** property from the receiving object, and then determine the street property relative to the address object.

Getting Attribute Values Using Key-Value Coding

The method **valueForKey**: returns the value for the specified key, relative to the receiver. If there is no accessor or instance variable for the specified key, then the receiver sends itself a valueForUndefinedKey: message. The default implementation of valueForUndefinedKey: raises an NSUndefinedKeyException; subclasses can override this behavior.

Similarly, **valueForKeyPath**: returns the value for the specified key path, relative to the receiver. Any object in the key path sequence that is not key-value coding compliant for the appropriate key receives a valueForUndefinedKey: message.

The method **dictionaryWithValuesForKeys**: retrieves the values for an array of keys relative to the receiver. The returned NSDictionary contains values for all the keys in the array.

Key-Value Coding

Note: Collection objects, such as NSArray, NSSet, and NSDictionary, can't contain nil as a value. Instead, you represent nil values using a special object, NSNull. NSNull provides a single instance that represents the nil value for object properties.

To illustrate the difference between the properties dot syntax and KVC key paths, consider the following.

```
MyClass *anotherInstance = [[MyClass alloc] init];
myInstance.linkedInstance = anotherInstance;
myInstance.linkedInstance.integerProperty = 2;

This has the same result as:
MyClass *anotherInstance = [[MyClass alloc] init];
myInstance.linkedInstance = anotherInstance;
[myInstance setValue:@2 forKeyPath:@"linkedInstance.integerProperty"];
```

Delegation

Delegation

Delegation is a simple and powerful pattern in which **one object in a program acts on behalf of, or in coordination with, another object**. The delegating object keeps a reference to the other object—the delegate—and at the appropriate time **sends a message** to it. The message informs the delegate of an event that the delegating object is about to handle or has just handled. **The delegate may respond to the message** by updating the appearance or state of itself or other objects in the application, and in some cases it can return a value that affects how an impending event is handled. **The main value of delegation is that it allows you to easily customize the behavior of several objects in one central object.**

Delegation

```
if (delegate && [delegate respondsToSelector:@selector(someMethod:)]) {
    [delegate performSelector:@selector(someMethod:) withObject:self];
}
```

Apple Programming code style

https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.html

https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Conventions/Conventions.html

Home work

Create three instances of custom classes (ABCBoy, ABCGirl, ABCDj)
ABCDj should have protocol (ex ABCDjDelegate) with method musicStateChanged: with value YES / NO
ABCBoy should implement ABCDj protocol and pass self as a delegates
ABCBoy should should call their empty methods(startDancing and stopDancing) when musicStateChanged:
Add to ABCGirl static variable danceBlock
Implement same functionality using blocks with ABCGirl class

Use code style guide in your work!!!!