SCIENCES SORBONNE UNIVERSITÉ

Objective-C, memory management and automatic generation of setters and getters

Fabrice.Kordon@lip6.fr

LIP6

As an introduction…

Remember «properties»

- And how they are qualified with regards to memory management

```
#import <Foundation/Foundation.h>

@interface Student : NSObject

@property (readwrite, nonatomic, assign, getter=myNumber) int number;
@property (readwrite, nonatomic, copy, getter=myName) NSString *name;

// Initialization
- (id) initWithNumber:(int)aNumber;
- (id) initWithNumber:(int)aNumber andName:(NSString*)aName;

// methods
- (NSString*) identity;

@end
```

Properties and memory management

@property directive

- Impact on the automatic generation of accessors

Reminder on memory management

- assign (default) = assignment
  - **For «simple» types (int, BOOL, etc.)**
  - **Possibly for objects `self` is not a (co)owner**
- retain = incrementing the reference counter
  - `self` **becomes (co)owner of the referenced object**
- copy = creation of a new object
  - **Corresponding class must support copy**
  - `self` **becomes the only owner of the copy**

Setter for an «assign» property

This

```
@property (readwrite,nonatomic, assign) Student *myStudent;
```

Leads to

```
- (void) setMyStudent:(Student*)value {
    _myStudent = value;
}
```

- Inappropriate for object references of course

Setter for an «retain» property

This

```
@property (readwrite,nonatomic, retain) Student *myStudent;
```

Leads to

```
- (void) setMyStudent:(Student*)value {
    if (_myStudent != value) {
        [_myStudent release];
        _myStudent = [value retain];
    }
}
```

Setter for an «copy» property

This

```
@property (readwrite,nonatomic, copy) Student *myStudent;
```

Leads to

```
- (void) setMyStudent:(Student*)value {
    [_myStudent release];
    _myStudent = [value copy];
}
```

Remember…

We can «talk» to ghosts

What about getters

Simpler

- assign
  - just return the value
- retain / copy
  - copy + autorelease pool

ARC and its new classification

- **weak and strong are variants of retain**
  - Main impact on the way dealloc is handled
- **Weak**
  - when deallocated, the object remains
- **Strong**
  - when deallocated, the object is deallocated too
    - **Prior to the deallocation of self**
- **But…**
  - Mostly used when ARC is activated
  - We will have a closer look when dealing with Swift

As a conclusion…

That's all for now

- We will (almost) not discuss anymore about memory

But please play and experiment