Introduction to Objective-C

Kyrill Schmid,Lenz Belzner

Mobile und Verteilte Systeme

03.05.2017

Classes: Interface

Classes describe the properties and behavior (blueprints) for particular objects. Class definition comprises two parts:

- Header files (.h) to define the public interface of a class
- Implementation files (.m) to implement (private) behavior

```
@interface Person : NSObject // erbt von NSObject
// Deklariere zwei Properties: firstName und lastName
@property NSString *firstName;
@property NSString *lastName;
// Deklariere dedizierte init-Methode und zwei Instanzmethoden
-(id)initWithFirstName:(NSString *)aFirstName lastName:(NSString *)aLastName;
-(void)setFirstName:(NSString *)aFirstName andLastName:(NSString *)aLastName;
-(void)sayHallo;
@end
```

Classes: Implementation

After we have defined the interface in our header file we add another file called an implementation file (.m) where we add the following:

```
-(id)initWithFirstName:(NSString *)aFirstName lastName:(NSString *)aLastName{
    self = [super init];
    if (self)  {
        _firstName = aFirstName;
        _lastName  = aLastName;
    }
    return self;
}
-(void)setFirstName:(NSString *)aFirstName andLastName:(NSString *)aLastName;{
    _firstName = aFirstName;
    _lastName = aLastName;
}
-(void)sayHallo{
    NSLog(@"Hello World! My name is: %@ %@", [self firstName], [self lastName])
}
```

Using Objects

To use objects we do the following:

```
#import <Foundation/Foundation.h>
#import "Person.h"
int main(int argc, const char * argv[]) {
    // insert code here...
    Person *p1 = [[Person alloc] initWithFirstName:@"Hans" lastName:@"Peter"];
    [p1 sayHallo];
    return 0;
}
```

Properties

Objective-C properties offer a way to encapsulate data

- Property declarations are included in the interface (.h):

```
@property NSString *firstName;
@property NSString *lastName;
```

- Attributes tell compiler which accessor methods should be synthesized (default readwrite):

```
@property (nonatomic, readwrite) NSString *firstName;
@property (nonatomic, readwrite) NSString *lastName;
```

- Implicit declaration and implementation of getter and setters
- Implicit declaration of instance variables:

```
_fullname;
```

Accessor Methods

An object's properties are accessed or set via accessor methods. These accessor methods are automatically synthesized from the compiler. Accessor methods have the following naming convention:

- getter method has same name as the property:

```
-(NSString *)firstName;
-(NSString *)lastName;
```

- setter method starts with set followed by capitalized property name:

```
-(NSString *)setFirstName;
-(NSString *)setLastName;
```

Instance Variables

Properties are backed by an instance variable

- An instance variable is a variable that exists and holds it's value for the life of the object
- Instance variables can be accessed directly from any instance method of the object

```
@property (readonly) NSSString *firstName; // creates also variable: _firstName
- (void)someMethod {
    NSString *someString = @"Lduwig";
    self.firstName = someString;
    // or
    [self setFirstName:someString];
    // or
    _firstName = someString;
}
```

Custom names

You can specify a custom instance variable name by telling the compiler to synthesize:

```
@implementation Person
@synthesize firstName = myFirstName;
...
@end
```

Initialization

Within initialization methods one should always access instance variables directly:

```
- (id)initWithFirstName:(NSString *)aFirstName
                          lastName:(NSString *)aLastName {
    self = [super init];

    if (self) {
        _firstName = aFirstName;
        _lastName = aLastName;
    }

    return self;
}
```

Message Passing

In Objective-C a method call is a message towards an object. The most common way to send messages between objects is using square brackets syntax:

```
Person *p1 = [[Person alloc] init];
[p1 sayHello];
```

- The reference on the left is called the receiver
- The message on the right is the name of the method to call on that receiver
- Receiver type will be determined at runtime
- It is unsure whtether objects listen to message

alloc: allocate memory for an object

Memory for an objective-object is allocated dynamically for its properties and inherited properties. This process is handled through the call of alloc:

```
+ (id) alloc;
```

The return type of this method is: id. This special keyword used in ObjC means: some kind of object (special pointer without an asterisk). Calling alloc also clears out the memory allocated for the object by setting them to zero.

init: initialize an object

Calling alloc needs to be combined with a call to init which is another NSObject method:

```
- (id) init;
```

The init method is used by a class to make sure the properties have suitable inital values. If a method returns a pointer it is possible to nest method calls:

```
NSObject *newObject = [[NSObject alloc] init];
```

Dont

Dont do this:

```
NSObject *newObject = [NSObject alloc];
[newObject init];
```

Because if init returns a different pointer we'll be left with a pointer that was originally allocated but never initialized.

id

In order to keep track of an object in memory we need to use a pointer. Because of ObjC's dynamic nature the class type of that pointer doesn't matter as the correct method will be called on the relevant object when you send it a message. The id type specifies a generic object pointer.

```
id someObject = @"Hello World!";
[someObject removeAllObjects];
```

This compiles because the compiler has no further information about the object although we know it's a string. However, a NSString object can't respond to removeAllObjects so we'll get an exception at runtime. Whereas with the following code the compiler will generate an error because removeAllObjects is not declared in any public interface of NSString;

```
NSString *someObject = @"Hello World!";
[someObject removeAllObjects];
```

nil

If we declare an object pointer without initializing it the variable will automatically point to nil.

```
Person *somePerson; // somePerson is automatically set to nil
```

This is considered best practice if no better initialization value is available because it is perfectly acceptable in ObjC to send a message to nil as nothing will happen. However, if you expect a return value from a message send to nil the return value will be nil for object return types, 0 for numeric types and NO for BOOL. To check whether an object is nil we can use standard C inequality operator

```
if (somePerson != nil){
        // somePerson points to an object
}
```

nil

or simply supply the variable:

```
if(someObject){
        // somePerson points to an object
}
```

because if it is nil its logical value is 0.

Memory Management

Memory management model is based on object ownership:

- Any object may have one or many owners
- As long as it has at least one owner it exists (otherwise it will be destroyed)

Object ownership rules:

- You own an object you create (alloc, new, copy) or retain explicitly
- Relinquish ownership with release or autorelease

With automatic reference counting we don't have to retain or release manually.

Automatic Reference Counting

Automatic reference counting can be understood as some preprocessing step that virtually inserts release and autorelease statements into code. In general we don't have to worry about memory management. However, we can introduce memory leaks through cirular references.

Circular References

Assume we have two objects - A and B. A has a strong reference to B and B has a strong reference to A. In this case ARC will not be able to release any of these objects:

```
@interface A : NSObject
@property (nonatomic, strong) NSObject *b;
@end

@implementation A
@end

@interface B : NSObject
@property (nonatomic, strong) NSObject *a;
@end

@implementation B
@end
```

Circular References

If we create two instances like:

```
int main(int argc, const char *argv[]){
        A *a = [[A alloc] init];
        B *b = [[B alloc] init];
        // now a and b have reference count == 1
        a.b = b
        b.a = a
        // now a and b have reference count == 2
}
// a and b have now reference count == 1
```

This code introduces a memory leak as ref count for a and b will now be 1 at most and they will never be released.

Circular References

The solution is to make one of the references to (weak):

```
@interface B : NSObject
@property (nonatomic, weak) NSObject *a;
@end
```

A weak reference does not add a new reference count.

Circular References

```
int main(int argc, const char *argv[]){
        A *a = [[A alloc] init];
        B *b = [[B alloc] init];
        // now a and b have reference count == 1
        a.b = b
        b.a = a
        // a now has ref count == 1 and b has reference count == 2
}
// a and b have now reference count == 0 and are destroyed
```