Object Oriented Programming in Objective-C

2501ICT/7421ICT Nathan

René Hexel

School of Information and Communication Technology
Griffith University

Semester 1, 2012

Outline

1. Subclasses, Access Control, and Class Methods
   - Subclasses and Access Control
   - Class Methods

2. Advanced Topics
   - Memory Management
   - Strings

Objective-C Subclasses

Objective-C Subclasses

Subclasses in Objective-C

- Classes can extend other classes
  - `@interface` *`AClass`*`: NSObject`
  - every class should extend at least `NSObject`, the root class
  - to subclass a different class, replace `NSObject` with the class you want to extend
- `self`
  - references the current object
- `super`
  - references the parent class for method invocations

Creating Subclasses: `Point3D`

Parent Class: `Point.h`

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

@interface Point:  NSObject
{
  int x;          // member variables
  int y;          // protected by default
}
- init;           // constructor

- (int) x;        // access methods

- (void) setX: (int) newx
            y:  (int) newy;
@end
```

Child Class: `Point3D.h`

```
#import "Point.h"

@interface Point3D: Point
{
  int z;            // add z dimension
}
- init;             // constructor

- (void) setZ: (int) newz;

- (void) setX: (int) newx
            y:  (int) newy
            z:  (int) newz;
@end
```

Subclass Implementation: Point3D

Parent Class: Point.m

```
#import "Point.h"

@implementation Point

- init                  // initialiser
{
        x = 0;
        y = 0;

        return self;
}

- (int) x               // get method
{
        return x;
}

- (void) setX: (int) nx  y:  (int) ny
{
        x = nx;     y = ny;
}

@end
```

Child Class: Point3D.m

```
#import "Point3D.h"

@implementation Point3D

- init                  // initialiser
{       if ([super init])
                z = 0;
        return self;
}

- (void) setZ: (int) newz
{
        z = newz;
}

- (void) setX: (int) nx
            y:  (int) ny
            z:  (int) nz
{
        [super setX: nx y:  ny];
        [self  setZ: nz];
}
@end
```

Access Control in Objective-C

Access Control in Objective-C

Access Control

- `@public`:
  - everyone has access
  - violates the principle of information hiding for member variables ⇒ not usually a good idea!
- `@private`:
  - nobody has access, except the defining class
  - useful for variables that should not be accessed by subclasses
- `@protected`:
  - mix between `@public` and `@private`
  - only the defining class and subclasses have access
  - useful for most member variables
  - default for Objective-C classes
- In Objective-C, `@public`, `@private`, and `@protected` applies to member variables only
  - methods are always public

Access Control Example

Example

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

@interface MyClass:  MySuperClass
{
  @public       // public vars
        int a;
        int b;

  @private      // private vars
        int c;
        int d;

  @protected    // protected vars
        int e;
        int f;
}

- init;         // constructor

// ...  other class methods

@end
```

Which `printf` is wrong?

Example (Which line(s) will cause a compiler error?)

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

@interface ClassX: NSObject
{
  @public       int a;
  @private      int b;
  @protected    int c;
}
@end


@interface ClassY: ClassX
- (void) print;                         // a print method
@end


@implementation ClassY

- (void) print
{
        printf("a = %d\n", a);          // print a
        printf("b = %d\n", b);          // print b
        printf("c = %d\n", c);          // print c
}

@end
```

Class Methods in Objective-C

Class Methods in Objective-C

Class Methods

- So far we only had Instance Methods
  - refer to objects (instances) of a class
- Class Methods
  - sometimes it's good to have a method that can be invoked without an instance
    - e.g. `alloc` which is needed to *create an instance of a class* by allocating memory
  - in Java, these methods were called `static`
  - in C, `static` means valid for a particular scope across invocations
- Objective-C Class Methods are simply denoted by a + instead of a −
  - e.g. `+ alloc`

Class Method Example

Example

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

@interface Point:  NSObject
{ int x, y; }                   // member variables

+ (int) numberOfInstances;      // a class method
- init;                         // an instance method (e.g.  the constructor)
@end
@implementation Point

static int instanceCount = 0;   // number of instances of the Point class

+ (int) numberOfInstances       // count the number of instances
{
        return instanceCount;   // return the current instance count
}

- init                          // constructor implementation
{
        if (!(self = [super init])) return nil;

        instanceCount++;        // we created a new instance

        return self;
}
@end
```

About 0, FALSE, NULL, and nil

- In Java, null denoted an empty reference
  - null does not exist in C, Objective-C, C++
- 0 in C denotes a number of things
  - integer or floating point values of 0 (or 0.0)
  - a false result of a boolean expression
  - a null pointer or object reference
    - ⇒ can be confusing what the actual meaning is
    - ⇒ better use FALSE, NULL, nil, etc. to express meaning
- EXIT_SUCCESS – successful program completion
- FALSE – a false boolean expression
- NULL – a null pointer
- nil – an empty object reference in Objective-C
  - e.g. nil does not exist in C/C++ (there, you should use NULL instead

Objective-C Memory Management

Objective-C Memory Management

Memory Management

- Memory needs to be handled explicitly in C, Objective-C, and C++
  - How is memory allocated, how is it released?
  - When should I release memory?
- Java Memory Management reviewed
  - `new` operator allocates memory
  - object references are automatically counted and tracked
  - a a Garbage Collector periodically releases unused objects

    ⇒ convenient, but no direct control by the programmer
- C provides `malloc()` and `free()` functions

  ⇒ completely manual memory management
- C++ has `new` and `delete` operators

  ⇒ completely manual memory management
- Objective-C has `+alloc` and `-dealloc` methods
  - Objective-C uses reference counting
    - allows to keep track of how often objects are referenced

  ⇒ semi-automatic memory management

Objective-C Memory Management

- `+ alloc`
  - allocates memory for an object, sets reference count to 1
  - `init` needs to be called then for initialisation
- `- release`
  - releases an object
    - → decrements reference count, if 0 then calls `dealloc`
- `- dealloc`
  - deallocates memory for an object
  - → never call `dealloc` directly (`release` calls `dealloc` when needed
- `- retain`
  - increments reference counter
    - → call whenever you need the same object in multiple places
- `- copy`
  - creates a new object by copying everything
    - copy has retain count of 1 (needs to be `release`d later on)
    - → expensive (but needed if objects will be modified)

Person Record Interface Example

Example (Interface)

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

@interface Person:  NSObject              // an object referencing a person
{
  int            yearOfBirth;            // the year the person was born
  NSString      *name;                   // the name of the person
  Person        *mother, *father;        // the parents of the person
}
                                        // access methods:
- (void) setYearOfBirth:  (int) born;
- (void) setName:  (NSString *) newName;
- (void) setMother:  (Person *) theMother
            father:  (Person *) theFather;
- (int) yearOfBirth;                    // no ’get’ needed in Objective-C
- (NSString *) name;
- (Person *) mother;
- (Person *) father;

- (void) dealloc;                       // needed for memory management!

@end
```

Person Record Implementation, Part 1

Example (Implementation part 1)

```
#import "Person.h"

@implementation Person

- (int) yearOfBirth                     // yearOfBirth getter method
{       return yearOfBirth;     }       // return yearOfBirth member variable

- (NSString *) name                     // name getter method
{
        return name;                    // return name member variable
}

- (Person *) mother                     // mother getter method
{
        return mother;                  // return mother member variable
}

- (Person *) father                     // father getter method
{       return father;          }       // return father member variable

- (void) setYearOfBirth:  (int) born     // a simple setter method
{
        yearOfBirth = born;             // just assign the ’int’
}
```

Person Record Implementation (continued)

Example (Implementation part 2)

```
- (void) setName:  (NSString *) newName
{
  [name release];                              // release the old name
  name = [newName copy];                       // copy the new name
}

- (void) setMother:  (Person *) theMother
            father:  (Person *) theFather
{
  [theMother retain]; [theFather retain];      // retain references
  [mother release];                            // release the old mother and
  [father release];                            // father references (if any)
  mother = theMother; father = theFather;      // store references
}

/*
* every class that retains other objects needs a dealloc method!
*/
- (void) dealloc
{
  [name release];                              // release all objects held!
  [mother release];
  [father release];

  [super dealloc];                             // call super class dealloc last
}
```

Autorelease Pools

- Object Ownership Reviewed
  - → any entity that uses an object needs to `retain` it
  - → `release` can become difficult with collections
    - e.g., an object that gets removed from a List but used elsewhere:
    - list would need to `release` object before it gets `retain`ed again
    - ⇒ danger of using an expired pointer!
- `- autorelease`
  - marks an object for later release
    - → puts the object on an autorelease pool
- Autorelease Pools
  - are just lists of objects to be `release`d
  - → objects actually get `released` when the pool gets `dealloc`ated

Autorelease Pool Example

Example (What does this program print?)

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

int main(int argc, char *argv[])
{
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

  NSString *str1 = [[NSString alloc] initWithUTF8String:  "self managed string"];
  NSString *str2 =  [NSString stringWithUTF8String:  "autorelease managed string"];

  printf("str1 is a %s\n", [str1 UTF8String]);
  printf("str2 is a %s\n", [str2 UTF8String]);

  [str1 release];       // release the self-managed string
  [pool release];       // release the pool (also releases the autoreleased str2)

  return EXIT_SUCCESS;
}
```

Answer

```
str1 is a self managed string
str2 is a autorelease managed string
```

Autorelease Pool Example 2

Example (What does this program print?)

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

int main(int argc, char *argv[])
{
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  NSString *string = [[NSString alloc] initWithCString:  "self managed string"];

  printf("string retain count is %d\n", [string retainCount]);

  [string autorelease];         // put the string on the autorelease pool

  printf("string retain count now is %d\n", [string retainCount]);

  [pool release];       // release the pool (also releases string)

  return EXIT_SUCCESS;
}
```

Answer

```
string retain count is 1
string retain count now is 1
```

When to use Autorelease Pools

- In Convenience Methods
  - – `stringWithCString` allocates an `NSString`, then `autorelease`s it
  - → any method thad does `alloc`, then `init...`, then `autorelease`
- Any collection method that removes then returns an object
  - `return [object autorelease];`
- Temporary Variables
  - variables that you only use briefly and would `release` almost straight away
- Don’t use Autorelease Pools as “poor man’s garbage collector”!
  - no replacement for proper memory management!
  - → where should Pools be created?

Where to create Autorelease Pools

- Always create a pool first thing after `main()`
  → release that pool at the very end of your program (right before `return EXIT_SUCCESS;`)
- Around areas that use or create temporary objects
  - within long loops
  - around short loops
  - within methods

Example

```
for (int i = 0; i < 100; i++)
{
  NSAutoreleasePool *innerpool = [[NSAutoreleasePool alloc] init];

  NSString *string =  [NSString stringWithInt:  i];       // temporary string

  // do something useful with ’string’
  printf("string is %s\n", [string UTF8String]);

  [innerpool release];  // release the pool (releases all autoreleased strings)
}
```

Object Lifecycle

Task	Objective-C	Java	C++ Heap	C++ Stack
allocate	`+ alloc`	`new`	`new`	entry
initialise	`- init`	constr.	constr.	constr.
hold object	`- retain`	automatic	-	-
let go	`- release`	automatic	-	-
destroy	final `- release`	G.C.	`delete`	fn exit
clean up	`- dealloc`	`finalise()`	destr.	destr.
deallocate	`[super dealloc]`	G.C.	`delete`	return

Strings

String Objects in Objective-C

Objective-C Strings

- `NSString`
  - basic string class
  - class cluster with concrete classes optimized for different string sources
  - much nicer than having to use `char *`
- `NSMutableString`
  - subclass of `NSString` for strings that can be modified
- String Constants
  - embedded in `@""`
    - e.g. `@"Hello, Objective-C Strings"`

  → don’t mix up with C Strings embedded in `""`

Objective-C String Examples

Example (Some NSString methods)

```
NSString *s1 = [NSString new];                                  // empty string
NSString *s2 = [NSString stringWithString:  @"Hello, void"];     // from ObjC or
NSString *s3 = [NSString stringWithUTF8String:  "Hello, void"];  // C string
NSString *s4 = [NSString stringWithFormat:                      // printf-style
                                @"Hi, it's %d degrees", 28];    // format
NSString *s5 = [s4 stringByAppendingString:  @" celsius"];       // appending
NSString *s6 = @"12345";                                // a string constant
int len4 = [s4 length];                                 // get length of s4
int val6 = [s6 intValue];                               // convert s6 to int

if ([s1 isEqualToString:  s2])                           // same content?
        printf("s1 is equal to s2 -- how come?\n");
else if ([s1 compare:  s2] == NSOrderedAscending)        // which one comes first?
        printf("s1 comes before s2\n");
else
        printf("s2 comes before s1\n");

printf("s2 is:  %s\n", [s2 UTF8String]); // convert s2 to a C string for printf

NSLog(@"s3 is:  %@\n", s3);              // NSLog() prints formatted NSStrings
                                        // %@ = place holder for ObjC objects

[s1 release];                           // don't forget proper memory management!
```

Other Useful Methods

+ `stringWithContentsOfFile:`
    - convenience method
    - reads the whole content of a file into a string
    - most efficient way of reading files

- `rangeOfString:`
    - searches for a string within another String

- `substringWithRange:`
    - returns a substring within a given range

- `mutableCopy`
    - returns a mutable copy of a string

→ See `NSString` and `NSMutableString` in the Foundation API