Objective-C:
An Introduction for Java Programmers

Praxis der Softwareentwicklung 12

Jörg Henß | December 24, 2009

CHAIR FOR SOFTWARE DESIGN AND QUALITY

Outline

Objective-C

- Developed in the 80s by Stepstone
- Strict superset of C
  - Any C program compilable as Objective-C
- Object syntax derived from Smalltalk
  - Message based
  - Dynamic typing
  - Reflective
- Only usable with runtime system
  - GNUStep
  - Cocoa
  - Cocoa Touch

Objective-C 2.0

- Introduced with Mac OS 10.5
- Supports garbage collectors
- Properties
- Dot notations
- Fast enumerations
- Also used on the iPhone

C - Data types

- Basic types
  - `char`
  - `int`, `short`, `long`
  - `float`, `double`
  - `void`
- Specialised
  - `signed`, `unsigned`
  - `int64_t`, `uint32_t`, `uint_fast32_t`, `int_least16_t`
  - `long long int`, `long double`
- Arrays
  - `int scores[100];`
  - `int chessBoard[8][8];`
  - Attention: no bound checking!

C - Basic types

Overview

Name	Size	Minimum Range
char, signed char	8 bit	-128...+127
unsigned char	8 bit	0...255
short, signed short	16 bit	-32768...+32767
unsigned short	16 bit	0...65535
int, signed int	32 bit	-2147483648...+2147483647
unsigned int	32 bit	0...4294967295
long, signed long	32 bit	-2147483648...+2147483647
unsigned long	32 bit	0...4294967295
long long, long long int	64 bit	$2^{63}...2^{63}-1$
unsigned long long int	64 bit	$0...2^{64}$
float	32 bit	$3.4*10^{-38}...3.4*10^{38}$
double	64 bit	$1.7*10^{-308}...1.7*10^{308}$
long double	80 bit	$3.4*10^{-4932}...3.4*10^{4932}$

C - Complex data types

- Structures `struct`
- Unions `union`

Example: `struct`

```
struct date
 {
   int day;
   char month[10];
   int year;
 } birthday;

struct date christmas;
struct date birthday = {7, "Mai", 2005};
birthday.year = 1988;
```

C - Complex data types (2)

- Enumerations `enum`
- Type definitions `typedef`

Example: `enum` & `typedef`

```
enum Colors { blue, yellow, orange, brown, black };
enum Prime { two = 2, three, five = 5, seven = 7 };
enum Prime myPrime = five;
typedef char name[100];
typedef struct {
  name street;
  unsigned int number;
} adress;
adress myAdress;
myadress.number = 5;
```

C - Pointers

- Pointers point to memory locations
  - Variable value
  - Function
  - Pointer
  - Random garbage if not initialised
- Notation:
  - "*" anywhere between type and name
  - `int *intPointer;`
  - `void* voidPointer;`
- Similar to Java references
  - More powerful
  - More dangerous

"With great power there must also come – Great responsibility!"

Amazing Fantasy #15 (August 1962)

C - Pointers (2)

- Pointer Dereference
  - "`*`" used to access value
  - `*intPointer = 15;`
- Pointer Assignment
  - "`&`" retrieves adress
  - `intPointer = &intValue;`
- Null Pointer
  - Pointer is number
  - Initialise pointzer with "`NULL`" or "0"
- Sharing vs. Deep Copying

C - Pointers (3)

- Memory has to be allocated
  - `malloc` allocates heap space
  - `sizeof` retrieves size of object in bytes
  - `int *array = malloc(sizeof(int)*6)`
- Memory should be freed after usage
  - `free(array);`
- Dereference struct pointers
  - `(*structPointer).day = 6;`
  - `structPointer->day = 6;`
- Return parameters
  - Pointer in method signature
  - Used as return object
  - `void toBuffer(char* string, void *buf)`

C - Pointer Arithmetic

- Pointer arithmetic
  - `newPointer = intPointer + 1;`
  - `*(intPointer + 1) = 5;`
- Array is special pointer
  - ```
    int array[5];
    int *ptr = array;
    ptr[0] = 1;
    *(array + 1) = 2;
    *(1 + array) = 3;
    2[array] = 4;
    ```
  - Can not be reassigned
- Further hints on C:
  - `http://users.ece.utexas.edu/~adnan/c-refcard.pdf`
  - `http://cslibrary.stanford.edu/101/`
    ```

Objective-C - Classes

- Class
  - Methods
  - Instance variables
- Inheritance
  - Classes can have super class
  - NSObject common root class
  - No single root class like Java
- Definition splitted up:
  - Interface: .h file
    - Also used for documenting methods
  - Implementation: .m file
  - Please note: Interface $\neq$ Interface in Java

Example

- Interface: MyObject.h

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

@interface MyObject : NSObject {

}

@end
```

- Implementation: MyObject.m

```
#import "MyObject.h"

@implementation MyObject

@end
```

Types

- Builtin object types
  - Wildcard type: `id`
  - Null object: `nil`
  - Identifier: `self`, `super`
    - Cf. Java this and super
- C types
  - All types can be used
- Mixing types possible

Methods - Messaging

- Methods on object called via messaging
- Notation: `[receiver message]`
  - Sends message with method name (selector) to receiver
  - Message can have return value
  - Nested messages `[[receiver message1] message2]`
- Dispatched at runtime
  - Uses caching
- Polymorphism
  - Method calls not dependend on type
  - Called methods can be not implemented
- Sending to `nil` possible
  - returns `0` or `nil`

Methods - Parameters

- Parameters are part of message
  - `[myRectangle setWidth:20.0];`
  - A colon (:) splits name and parameter
    - Part of method name
  - Parameter can have default value
- Multiple parameters
  - Parameters also splitted by colon
  - Good style: Use pseudo naming
  - `[myRectangle setOriginX: 30 y:  50];`
- Variable parameter count
  - `[myRectangle addStyles:styleOne, styleTwo];`
- Dot syntax
  - Used for accessor methods
  - `myRectangle.width = 20.0;`

Methods - Declaration

- Simple methods
  - `(float)radius`
  - `(void)display`
- With parameters
  - `(void)setRadius:(float)aRadius`
  - `(void)setOriginX:(int)x y:(int)y`
- Variable parameter count
  - `(void)addStyles:...`
- Pseudo naming
  - `(void)setAge:(int)age andHeight:(int)height`

Example - Methods

- Interface: MyObject.h

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

@interface MyObject : NSObject {
}
-(void)printName;
-(void)setAge:(int)age andHeight:(int)height;
@end
```

- Implementation: MyObject.m

```
#import "MyObject.h"

@implementation MyObject
-(void)printName
{}
-(void)setAge:(int)age andHeight:(int)height
{}
@end
```

Access modifiers

- Instance variables
  - @private
  - @protected (default)
  - @public
  - @package (public in image)
  - Variables declared in interface
- Class variables
  - Declared in implementation
  - `static` has to be used for objects
  - Objects should be initialized using `+(void)initialize`
- Methods
  - ”+” precedes class method declaration (cf. Java static)
  - ”-” precedes instance method declaration
  - No private or protected

Example - Instance Variables

- Interface: Worker.h

```
@interface Worker : NSObject
{
    char *name;
@private
    int age;
    char *evaluation;
@protected
    id job;
    float wage;
@public
    id boss;
}

...
```

Example - Static Variables

- Interface: Worker.m

```
#import "Worker.h"

int Workcount;
static Worker *MyWorker;

@implementation Worker

+(void)initialize
{
    MyWorker = [[Worker alloc] init];
    Workcount = 0;
}

+(Worker*)getWorker
{
    return MyWorker;
}
```

Creating Class Instances

- Allocating and initializing objects
  - `id anObject = [[Rectangle alloc] init];`
- `alloc` allocates instance of class
  - allocates memory for instance variables
- `init` initializes object
  - Initializer method
  - Can return other object
- Combine alloc and init messages
  - Safe when other object returned
  - Similar to Java `new Object();`

Initializers

- Parameter usage changes name of method
  - Name should start with `init...`
  - `- (id)initWithName:(NSString *)anName`
  - `- (id)initWithImage:(NSImage *)anImage`
- Return type should be `id`
- A designated Initializer is required
  - Invokes designated initializer of super class
  - Set `self` to return value of super initializer
- Set instance variables direct
- Return `self` at end of initializer

Example - Initializer

```
...

- (id)initWithImage:(NSImage *)anImage
{
  // Find the size for the new instance from the image
  NSSize size = anImage.size;
  NSRect frame = NSMakeRect(0.0, 0.0, size.width, size.height);
  // Assign self to value returned by super's initializer
  // Designated initializer for NSView is initWithFrame:
  if (self = [super initWithFrame:frame])
  {
     image = [anImage retain];
  }

  return self;
}

...
```

Memory Management

- Objects are “created” by methods that
  - start with `alloc` or `new`
  - contain `copy`
- Created objects are owned objects
  - Owned objects have to be released
  - Send `release` message to these objects after usage
- Delayed release
  - `autorelease` releases an object when scope ends
- Taking ownership of objects
  - Send `retain` message
- Release instance objects on dealloc
  - `- (void)dealloc`
  - Also call dealloc on super

Protocols

- Protocols are similar to Java interfaces
  - Declare methods that are expected to be implemented
  - Capture similarities among classes
  - Hide information about concrete class of an object
- List of methods independent from a class
- Class can *conform* to a protocol
- Methods can be optional
- Properties can be part of a protocol

Example - Protocol

```
@protocol MyProtocol

  - (void)requiredMethod;

@optional
  - (void)anOptionalMethod;
  - (void)anotherOptionalMethod;

@required
  - (void)anotherRequiredMethod;

@end
```

Using Protocols

- `@protocol(protocolName)` refers to a protocol object
- Adopting a protocol:
  - `@interface ClassN : SuperC < protocols >`
  - Protocols are separated by a comma
- Check conformance:
  - `[receiver conformsToProtocol:myProtocol)]`
- Used for typing:
  - `id <MyProtocol> anObject;`
- One protocol can incorporate other protocols
  - `@protocol ProtocolName < protocols >`

Categories

- Categories used to add behaviour to classes
  - Define new methods
  - Can not define new instance variables
- Have a name or are anonymous
- Notation:
  - `@interface ClassName ( CategoryName )`
  - `@implementation ClassName ( CategoryName )`
- Categories extending NSObject called informal protocols
- Anonymous protocols used for “`private`” methods
  - Declared in an implementation file

Example

- Interface: MyCategory.h

```
#import "MyObject.h"

@interface MyObject (MyCategory) {
  //method declaration
}

@end
```

- Implementation: MyCategory.m

```
#import "MyCategory.h"

@implementation MyObject (MyCategory)
  //method definition
@end
```

Declared Properties

- Properties provide accessors for instance variables
- Notation:
  - `@property type name;`
- Optional parameters
  - `@property(attribute, ...) type name;`
  - Accessor method names
    - `getter=getterName`, `setter=setterName`
  - Writability
    - `readwrite`, `readonly`
  - Setter semantics
    - `assign`, `retain`, `copy`
- Property implementation directives
  - `@synthesize`, `@dynamic`
  - `@synthesize firstName, age = yearsOld;`

Exception Handling

- Exceptions are handled very similar to Java
  - `@throw myException`
  - `@try{`
    `}`
    `@catch (NSException *exception){`
    `NSLog(@"Caught %@: %@", [exception name],`
    `[exception reason]);`
    `}`
  - `@finally{}`

NSString

- NSString is a string object
  - Array of Unicode characters
- `@"TestString"` can be used in code
- NSString has several useful initializers
- NSString is not editable
  - Use NSMutableString for editing

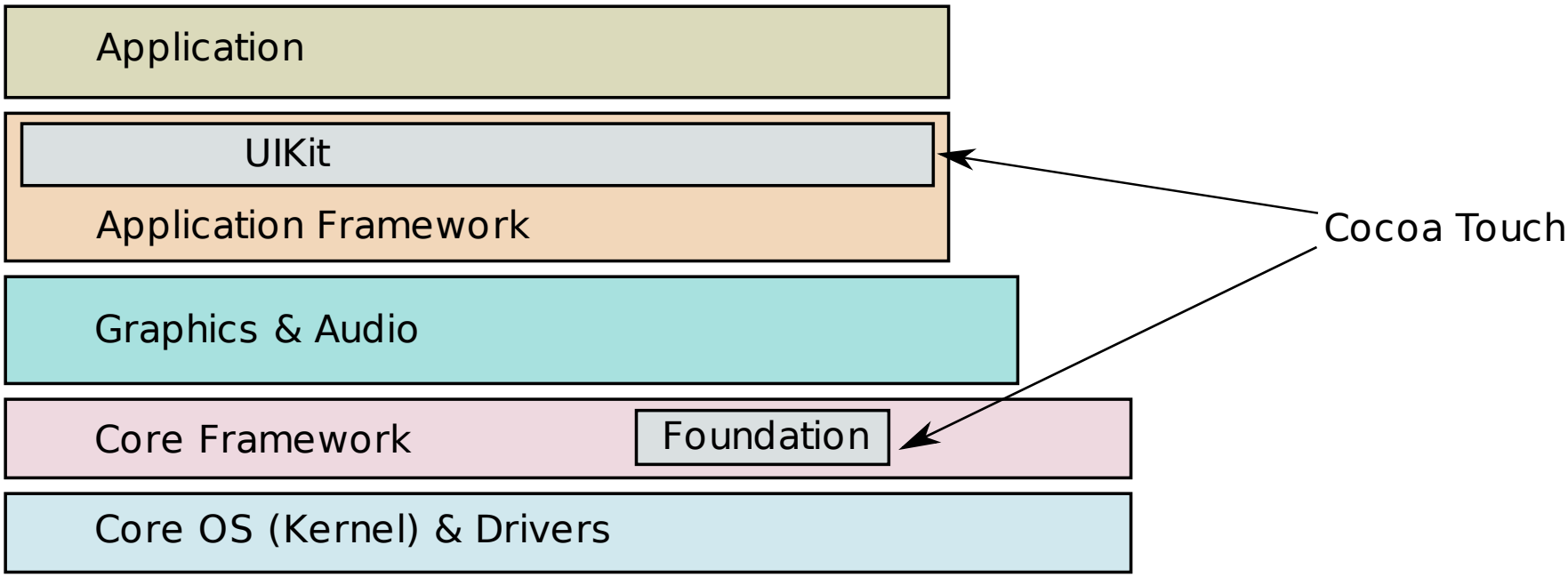
Logging

- `NSLog(NSString *format, ...);`
- Format string contains parameters
  - %@ Object
  - %d, %i signed int
  - %u unsigned int
  - %f float/double
  - and much more
- `NSLog(@"my ns string = %@", myNSString);`

Misc Hints

- Java `instanceof`
  - `[anObject isKindOfClass: AClass]`
- Java `synchronized`
  - `@synchronized(anObject)`
- Has class a method query
  - `[anObject respondsToSelecto: @selector(methodWithArg:)]`
- `#import "MyView.h"` makes the things defined in MyView.h accessible

iPhone OS - Overview



- Reference library can be found here:
  `http://developer.apple.com/iphone/library/`

Cocoa Touch

- An abstraction layer

UIKit

- UI Elements
- Application Runtime
- Event Handling
- Hardware APIs

Foundation

- Utilities
- Collections
- Wrappers for system services
- Basic object behaviour

iPhone Development Tasks

- Create project
- Design the user interface
- Write Code
- Build and run application
- Measure and tune application performance

Xcode

- Xcode is the IDE used for iPhone development
  - Multi windows paradigm
  - Wizards for creating iPhone projects
  - Organizer shows
    - Connected devices
    - Crash logs
    - Screenshots
    - Provisioning profiles
  - Integrated debugger (gdb)
    - Debug app on iPhone

Important iPhone Bundle Files

- `MyApp.app` - executable file
- `Settings.bundle` - settings dialog folder
- `Icon.png` - icon file
- `MainWindow.xib` - default interface object
- `Default.png` - background while loading
- `Info.plist` - application settings
- `language.lproj` - localized ressources directory

iPhone User Interface

- iPhone uses the Model-View-Controller design pattern
  - `UIView` objects arranged in an `UIWindow`
  - An `UIWindow` has several views
  - An `UIView` can have subviews
    - Buttons, textfields etc are all views
- Create views using Interface Builder
  - Can also be created programatically
- Consider the “iPhone Human Interface Guidelines”

Interface Builder

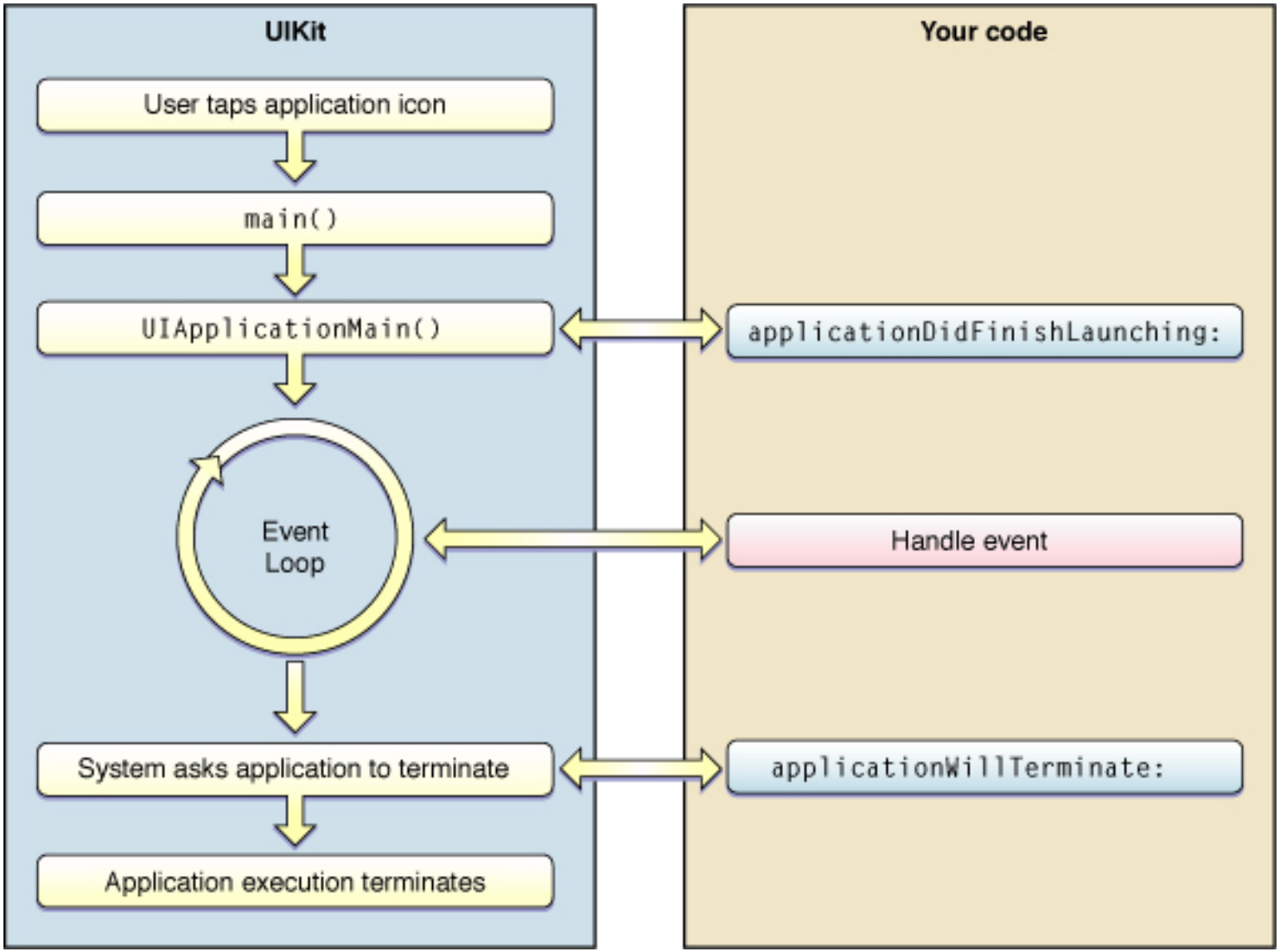
- Interface Builder can be used to arrange views and objects
- Interface definition is stored in `xib`/`nib` file
  - Loaded at initialization
- Interface Builder lets you make connections between views and controllers
  - `IBOutlet` used to connect to properties
  - `IBAction` used to connect to methods
    - Define actions for events, e.g. touches
  - Define delegate relations

Switch to code...

Write Code

- The AppDelegate class is system entry point
  - Conforms to the UIApplicationDelegate protocol
    - `applicationDidFinishLaunching` called on start
    - `applicationWillTerminate` called on end
    - `...`
  - `window` is the UIWindow object of the app
- Code is strongly related to user interface
  - Most code will be view controllers

Application life cycle



Source: Apple iPhone Application Programming Guide

View Controller

- View needs controller for handling events
  - Clicks, Shake, Rotation
- Subclassed from `UIViewController`
  - Custom view controllers
  - Container view controllers
  - Modal view controllers
- Controller contains:
  - Variables for the content to display
  - Properties/outlets pointing to view objects
    - `IBOutlet UIButton *myButton;`
  - Action methods
    - `(IBAction)respondToClick;`
    - `(IBAction)respondToClick:(id)sender;`
    - `(IBAction)respondToClick:(id)sender forEvent:(UIEvent*)event;`

Details: “View Controller Programming Guide for iPhone OS”

Example

- Interface: FirstViewController.h

```
@interface FirstViewController : UIViewController {
IBOutlet UIButton *myButton;
}
@property(retain) UIButton *myButton;

- (IBAction)doYellow;
@end
```

- Implementation: FirstViewController.m

```
@implementation FirstViewController
@synthesize myButton;

// The designated initializer.
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil {
 NSLog(@"Before Init");
    if (self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil]) {
        NSLog(@"Init");
    }
    return self;
}
- (IBAction)doYellow{
myButton.backgroundColor = [UIColor yellowColor];
}
- (void)viewDidLoad {
NSLog(@"test");
    [super viewDidLoad];
}
```

Frameworks

- Large collection of frameworks available
  - Core Data
  - Core Foundation
  - Core Location
  - Foundation
  - Map Kit
  - Core Audio
  - Quartz Core
  - ...

Instruments and Shark

- Instruments
  - Collects data about performance and behaviour
  - Find memory leaks
  - Can be launched from Xcode
- Shark
  - Low level profiler
  - Takes snapshots of the stack

References

- Apple Developer Connection:
  - “iPhone Application Programming Guide”
  - “The Objective-C Programming Language”
  - “iPhone Development Guide”
  - “iPhone Human Interface Guidelines”
- Objective C reference card
  `http://www.mecodegoodsomeday.com/ObjectiveCRef.pdf`