Coding Conventions

Objective-C coding conventions and styling

22-06-2012

Jens Willy Johannsen – jens@greenerpastures.dk

Change log

Revision 3 (JWJ, 2012-06-22)

- Properties: only use @property, @synthesize – don't declare instance variables.
- Updated organization/resources: storyboard files should be located in Resources group at root level.
- Added deployment target section to Implementation section.
- Projects should now use Storyboards instead of XIB files.
- Private methods are now *not* declared (either in .h or .m files).
- Fixed bug in DEBUG_LOG definition.
- Changed section on table cells to use prototype cells defined in storyboard files.
- Added "no warnings tolerated"

Revision 2

- ARC and related issues.
- A few more naming guidelines.

Comments

Use lots of comments! There is no such thing as “but the code is self-documenting” or “it is just as easy to read the code” or “the method name says it all”.

Uncommented code can not be maintained.

Use lots of comments to explain what the code does and write them while you code (or before) – not afterwards. Remember: other people have to review and maintain this code.

All methods should have a comment immediately above the method explaining what it does and, if necessary, what the parameters are (ok, you don’t need a comment explaining what dealloc or viewDidLoad or similar methods do).

Use comments above or on the same line as conditionals to explain when is being checked.

Add an empty line above the comment (unless the comment is the first line in a block) so it is easy to see what the comment is attached to.

For brief comments, it is also acceptable to put the comment at the end of the line with a tab or two in front.

Check your spelling in comments as well. And use capital letters at the start of comments.

Incorrect example:

```
NSArray* dataArray = [NSArray arrayWithArray:[filterSet allObjects]];
//sort array alphabeticaly
dataArray = [dataArray sortedDataArray];
```

Correct example:

```
NSArray* dataArray = [NSArray arrayWithArray:[filterSet allObjects]];

// Sort array alphabetically
dataArray = [dataArray sortedDataArray];
```

Example:

```
// Removes the current brochure and any page view controllers
- (void)removeCurrentBrochure
{
    DEBUG_POSITION;

    // Clear brochure
    self.brochure = nil;

    // Remove all page views
    for( PageViewController *pvc in pages )
            [pvc.view removeFromSuperview];

    // Reset content view size and current page number
    currentPage = 0;
    outerScroller.contentSize = CGSizeZero;
}
```

Special comments

Two special comment notations are used: temporary code and to-do's.

Temporary code is for debugging purposes and should be removed from the final production code. Temporary code should be marked like this:

```
/// TEMP: ensure that both arrays are of identical size
NSAssert( [visiblePins count] == [visiblePinInfos count], @"Arrays not of the same
size!" );
```

Note that there are *three* slashes. This is to make it easy to do a project wide search for temporary code.

To-do's look like this:

```
/// TODO: change to production App ID
#define FACEBOOK_APPID @"142505742450282"
```

Again, note the three slashes instead of two.

Syntax

Tabbing

Always use tabs, not spaces. Tab length is 4.

Curly braces

Curly braces must always be on a line by themselves:

```
// Correct:
- (void)dealloc
{
    // ...
}
```

Note that this is different from Apple's standard which is placing the opening brace at the end of the line:

```
// Wrong:
- (void)dealloc {
    // ...
}
```

Exceptions

There are three exceptions:

1. when there is nothing between the opening and closing braces
   ```
   - (void)handleError
   {}
   ```

2. when defining an enum or struct

```
typedef enum
{
    // Enums...
} EnumType;
```

3. blocks (see below)

One-liners vs. curly braces

For simple statements, one-line statements without curly braces should be used:

```
// OK:
if( [controller allIsGood] )
  return YES;
else
  return NO;
```

Blocks

When using blocks, the curly braces should *not* be on a line by themselves:

```
[UIView animateWithDuration:kSplashAnimationDuration delay:kSplashDelayTime
options:0 animations:^{
     splashImageView.frame = frame; splashImageView.alpha=0;
} completion:^(BOOL finished){
     [splashImageView removeFromSuperview];
}];
```

Also note, that there are no spaces before the opening curly brace.

Spaces

Use spaces *inside* parentheses for function calls and control structures only; *not* between method name/keyword and opening parenthesis:

```
// Correct:
CGSize newSize = CGSizeMake( 12, 20 );
if( error )
    return;

// Wrong:
result = CFFunction (myObj, 2);
if (result != 0)
    return;
```

Arrays

Put spaces before and after square brackets for arrays:

```
dataArray[ counter++ ] = obj;
```

Operators

And around operators:

```
if( [myString length] <= 5 || counter > 3 )
  // Do stuff...
```

Unary operators (++, --, ^ , ! etc) do not need spaces:

```
if( !success )
    return nil;

dataArray[ counter++ ] = obj;
```

Method arguments

Do *not* put space before method arguments or before and after square brackets for method invocations:

```
// Correct:
[self showActivityView:loadActivityView animated:NO];
```

Inheritance declaration

Use one space for padding around inheritance colon:

```
@interface HTTPDataLoader : NSObject
```

Method declarations

Methods are declared like this:

```
- (UITableViewCell*)tableView:(UITableView*)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
```

- Space after the initial - (or +)
- *no* space between object type and *
- *no* space between return type and method name
- *no* space before * in parameters and *no* space before parameter name

Variable and property declarations

Object variables are declared with a space *before* the asterisk:

```
NSArray *dataArray;

for( EventModel *model in inputArray )
```

Properties are declared like this (note: no spaces in parentheses):

```
@property (nonatomic, strong) NSDate *currDate;
```

Multiple parentheses

Whenever there are multiple parentheses in succession, there should *not* be spaces between them – put a space after the first parenthesis only:

```
// Wrong:
if( ( ( 1 << BITMASK_LOAD) & flags ) || isLoaded )


// Correct:
if( ((1 << BITMASK_LOAD) & flags) || isLoaded )
```

Line breaks

Use line breaks to divide the code into blocks of functionality to improve readability.

Never use more than one empty line!

Use empty lines *after* code but not before – this is correct:

```
// Correct:
- (void)removeCurrentBrochure
{
    // Clear brochure ivar
    [brochure release], brochure = nil;

    // Remove all page views
    for( PageViewController *pvc in pages )
            [pvc.view removeFromSuperview];
```

and this is wrong:

```
// Wrong:
- (void)removeCurrentBrochure
{

    // Clear brochure ivar
    [brochure release], brochure = nil;


    // Remove all page views
    for( PageViewController *pvc in pages )

            [pvc.view removeFromSuperview];
```

Switch statements

In switch statements, make en empty line after each break statement (except for the final break).

Multiple cases are not separated by empty lines:

```
switch (transaction.transactionState)
{
     case SKPaymentTransactionStatePurchasing:
            DEBUG_LOG( @"Purhcasing…" );
            break;

     case SKPaymentTransactionStateUnknown:
     case SKPaymentTransactionStateRestored:
            [self restoreTransaction:transaction];
            break;
}
```

Avoid using curly braces within a case:

```
// Wrong:
switch( segmentIndex )
{
     case 0:
            // Sort by name
            [self resortByName];
            break;

     case 1:
     {
            // Show details
            DetailViewController *detailViewController =
[[DetailViewController alloc] initWithNibName:@"DetailViewController"
bundle:nil];
            textMessageInputViewController.currentTitle =
currentPerson.title;
            [self.navigationController
pushViewController:detailViewController animated:YES];

     }
            break;

}
```

Instead, declare the variable above the switch statement and instantiate and use it within the case block.

Casing

Pascal casing (ThisIsPascalCasing):

- classes
- protocols
- categories
- enums
- structs

Upper-case with underscore (THIS_IS_UPPERCASE):

- #defines

Camel casing (thisIsCamelCasing):

- methods
- instance variables
- ... and pretty much everything else

Example of correct casing:

```
#define MAX_CONNECTIONS 10

typedef enum
{
    ErrorTypeNotFound,
    ErrorTypeRejected,
    ErrorTypeUnauthorized
} ErrorType;

@protocol HTTPDataLoaderDelegate
- (void)dataFinishedLoading:(NSData*)data forContext:(id)context;
- (void)dataFailedToLoadWithError:(ErrorType)error forContext:(id)context;
@end

@interface HTTPDataLoader : NSObject
{
    id<HTTPDataLoaderDelegate>  delegate;
    NSURLConnection             *connection;
    NSMutableData               *data;
}

@property (nonatomic, retain) id<HTTPDataLoaderDelegate> delegate;

+ (BOOL)isConnectedToNetwork;
- (void)loadDataFromURL:(NSURL*)url forContext:(id)context;
- (NSData*)loadDataSynchronouslyFromURL:(NSURL*)url forContext:(id)context;

@end
```

Long lines

Break up long lines into several if necessary:

```
UIAlertView *alert = [[UIAlertView alloc] initWithTitle:alertTitle
                                                message:alertMessage
                                               delegate:nil
                                      cancelButtonTitle:nil
                                      otherButtonTitles:NSLocalizedString( @"OK", @"OK button
for cannot send mail alert" ), nil];
```

Pragma marks

Use #pragma marks to separate the methods in the .m files into groups.

Create a group for:

4. Delegate methods. Use one group for each delegate. So is a class is both a UIAlertViewDelegate and a UIActionSheetDelegate, use two groups.
5. Private vs. public methods

Groups are created like this (remember the - between #praga mark and the name to insert a separator in the drop-down list):

```
// Other code...

#pragma mark - UIAlertViewDelegate methods

- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:
(NSInteger)buttonIndex
{
    // Alert dismissed - hide self
    [self.navigationController popViewControllerAnimated:YES];
}
```

Naming

Use Apple's Objective-C naming guildelines except for differences specified here.

Property names

Always make sure property names are meaningful.

Never just name a property "model". Instead use "current" plus the type of object. For example, "currentNewsItem" or "currentPerson".

One exception is for tables where it is acceptable to have an instance variable (and property) named "dataArray" or "dataDictionary" (this is simply to allow easier copy/paste of table related code).

Class prefixes

Class names are usually *not* prefixed with anything. So a class would be named e.g. "Person" and not "GPPerson" or something like that.

However, if it is a generic class that is used in many projects and – especially – if it is code that is distributed outside the company, "GP" may be used as prefix (e.g. "GPFileCache").

Class postfixes

Class names should be postfixed to indicate what type of class they are or from what class they derive:

- PageViewController
- AccountModel
- ArrowView

- CustomNavigationBar
- TitleLabel
- SignUpButton

Delegate protocols

Delegate protocols should be named <ClassName>Delegate (see example above)

Enums

Define enums by using a typedef'ed anonymous enum. All enum values should be prefixed by the enum type name:

```
typedef enum
{
    ErrorTypeNotFound,
    ErrorTypeRejected,
    ErrorTypeUnauthorized
} ErrorType;
```

Booleans

Use YES and NO for boolean values (*don't* use 1 and 0 or true and false).

Instance variables and properties

Do *not* declare the instance variable specifically. Use only @property and @synthesize keywords.

It is optional whether to use

```
@synthesize accountNumber;
```

or

```
@synthesize accountNumber = _accountNumber;
```

Categories

Category files should be named <Base class>+<category name>.h/.m.

Like this:

NSData+Base64.h and NSData+Base64.m

And the declaration would look like this:

```
@interface NSData (Base64)
```

Nesting

Avoid unnecessarily long nested method invocations:

```
// Not so good:
```

```
NSURLRequest *request = [NSURLRequest requestWithURL:[NSURL URLWithString:[NSString
stringWithFormat:@"http://%@%@", SITE_PREFIX,[dataObject valueForKey:[keysArray
objectAtIndex:0]]]]];

// Much better:
NSString *urlFilePath = [dataObject valueForKey:[keysArray objectAtIndex:0]];
NSString *urlString = [NSString stringWithFormat:@"http://%@%@", SITE_PREFIX,
urlFilePath];
NSURLRequest *request = [NSURLRequest requestWithURL:[NSURL
URLWithString:urlString];
```

Organization

Grouping in Xcode

Use the following minimum group structure in Xcode:

- View Controllers
  contains .h and .m files for UIViewController descendants
  Note: no XIB files here
- Views
  contains .h and .m files for UIView and UITableViewCell descendants
  Note: no XIB files here
- Models
  contains .h, .m and .xcdatamodel files
- [name of the project]
  contains and AppDelegate files and .h and .m files for other classes and also contains the Supporting Files group with .plist, .pch and main.m files.
  This group can contain subgroups if necessary.
- Resources
  contains storyboard and audio/video, .plist files and other resource files
  - Resources/images subgroup
    contains image files
  - Resources/XIB subgroup
    contains all XIB files. Be sure to move the XIB file from the Classes group if it is created there by Xcode (but you don't have to move the file on disk).
- Frameworks
  contains all linked frameworks
- Products
  all product files

It is a good idea to specify the groups' location as the corresponding folder on disk. That way, files created in a group are automatically placed in the correct folder on the disk.

If the project includes other Xcode projects, place the individual project groups at the top level at the top of the tree (immediately above the View Controllers group).

Directory structure

The project's folder should contain:

1. .xcodeproj file
2. A folder with the same name as the project containing:
    2.1. All .h and .m files (be sure that no .h or .m files are in the project root folder)
    2.2. Folders containing .h and .m files for "modules" (e.g. json parser)
    2.3. XIB files (even though they are placed in the Resources/XIB group in Xcode; this is just because Xcode places the files in the same location as the .m and .h files when creating them)
3. "Resources" folder containing:
    3.1. Image files
    3.2. Audio files
    3.3. Video files
    3.4. Data plist files
    3.5. Any other resources
    3.6. (XIB files may also be placed here)
    3.7. If deemed necessary, subfolders for better organization although this is not necessary; the Resouces group in Xcode should contain subgroups to structure the files but it is not necessary with lots of subfolders in the Resources folder on the disk (since files will be deleted or moved from within Xcode).
4. Folders for included projects (e.g. RestKit)

Header file structure

In the .h file the structure should be as follows:

1. #import/#includes
2. #defines
3. enum and struct declarations
4. Protocol declarations
5. @interface
6. Group all IBOutlet instance variables
7. Other instance variables (use blank lines to separate into groups if necessary)
8. Group all properties
9. Group method declarations: class methods first, then instance methods

Example:

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

#define MAX_CONNECTIONS 10

typedef enum
{
     ErrorTypeNotFound,
     ErrorTypeRejected,
     ErrorTypeUnauthorized
} ErrorType;

@protocol HTTPDataLoaderDelegate
- (void)dataFinishedLoading:(NSData*)data forContext:(id)context;
- (void)dataFailedToLoadWithError:(ErrorType)error forContext:(id)context;
@end

@interface HTTPDataLoader : NSObject
{
     IBOutlet MKMapView *map;
     IBOutlet UIBarButtonItem *userLocButton;
     IBOutlet UISegmentedControl *mapTypeControl;

     id<HTTPDataLoaderDelegate> delegate;
     NSURLConnection *connection;
     NSMutableData *data;
}

@property (nonatomic, retain) id<HTTPDataLoaderDelegate> delegate;

+ (BOOL)isConnectedToNetwork;

- (void)loadDataFromURL:(NSURL*)url forContext:(id)context;
- (NSData*)loadDataSynchronouslyFromURL:(NSURL*)url forContext:(id)context;

@end
```

Implementation file structure

The .m file should be structured as follows:

1. Standard file comment with creation data, copyright etc.
2. #imports and #includes
3. @implementation
4. @synthesize
5. dealloc and init methods
6. Overridden methods (e.g. standard view controller methods)
7. Delegate methods
8. Public methods and IBActions
9. Private methods

Example:

```
//
//  BrochureViewController.m
```

```
//
//  Created by Jens Willy Johannsen on 05-10-10.
//  Copyright 2010 Greener Pastures. All rights reserved.
//

#import "BrochureViewController.h"
#import "UIBarButtonItem+CustomTextButton.h"

//  Constants
static const CGFloat kBrochureViewControllerDoubleTapZoomInLevel = 4.0;

@implementation BrochureViewController
@synthesize outerScroller, navItem, navBar;

// Clean-up
- (void)dealloc
{
    // ...
}

// The brochure view supports only portrait orientation
- (BOOL)shouldAutorotateToInterfaceOrientation:
(UIInterfaceOrientation)toInterfaceOrientation
{
    return UIInterfaceOrientationIsPortrait( toInterfaceOrientation );
}

// Setup view for initial display
- (void)viewDidLoad
{
    // ...
}

// Release everything here that has been retained in viewDidLoad
- (void)viewDidUnload
{
    // ...
    [super viewDidUnload];
}

#pragma mark - UIScrollViewDelegate methods

// Zoom ended. Enable or disable paging and possible snap to whole page.
- (void)scrollViewDidEndZooming:(UIScrollView *)scrollView withView:(UIView
*)view atScale:(float)scale
{
    // ...
}
```

```
// A "coasting" pan ended. Update current page number.
- (void)scrollViewDidEndDecelerating:(UIScrollView *)scrollView
{
    // ...
}

#pragma mark - UIAlertViewDelegate methods

// Callback method from the "New brochure available. Download?" alert
- (void)alertView:(UIAlertView *)alertView didDismissWithButtonIndex:
(NSInteger)buttonIndex
{
    // ...
}


#pragma mark - Public methods

// Sets the brochure to display
- (void)loadBrochure: (BrochureModel*)newBrochure
{
    // ...
}

#pragma mark - Private methods

// Calls loadBrochureInfo if enough time has passed since last time we did
it or forceCheck is set to YES
- (void)checkForNewBrochureIfNecessary: (BOOL)forceCheck
{
    // ...
}

// Updates the current page number based on the contentOffset of the
scroller's content view.
- (void)calculateCurrentPage
{
    // ...
}

@end
```

Implementation

Deployment target

Default deployment target is iOS 5.0.

Warnings

There may be **no warnings** in the project. Make sure you fix all warnings immediately (and not, "I'll fix the warnings tomorrow").

The only exception is warnings in 3rd party included libraries etc.

Interface Builder and Storyboard files

Use the Interface Builder and Storyboard files as much as possible instead of creating view hierarchies programatically (except for custom table cells – see later section).

Individual XIB files should be used only if there is a good reason for not putting it in the main Storyboard file.

This doesn't mean that each and every view should be defined in Interface Builder. It is perfectly fine to programatically create a view, add buttons and labels and slide it onto the screen, for example. But the basic view controller's hierarchies should be created in IB.

Localization

If a project is localized (i.e. will support multiple languages either now or in the future), observe the following guidelines:

All UI texts specified in code must be written using NSLocalizedString() to facilitate localization.

All Storyboard and XIB files must be localized.

All data files (e.g. plist or xml) files containing texts shown to the user must be localized.

All images containing text must be localized (and, in general, avoid having text on images – use background images and labels whenever possible).

Be sure to set the correct "Localization native development region" in the Info.plist file (do not just use the default English localization for Danish texts) and change all data files and Storyboard/XIB files to the correct localized version (da for Danish).

Ususued language versions of files must be deleted. For example, when creating an app for Danish language only, a default English version is often also created. This should be deleted. (Check to see if there are any files in the English.lproj folder.)

Constants vs. inline values

In general, use constants instead of hardcoded, inline values for things like URLs, coordinates, dictionary keys and so on.

Use static and const whenever appropriate.

Prefix constants with a “k” and give them meaningful names. Often a postfix is a good thing. And remember to use comments to explain what the constants are used for.

Example:

```
static const NSTimeInterval kSMCalendarDataSourceUpdateInterval = 600; // Seconds between
getting items (10 minutes)
static const NSTimeInterval kSMCalendarDataSourceCacheTTL = 3600; // Time to elapse before we
will get new calendar items (1 hour)
static const NSInteger kSMCalendarDataSourceItemsToPersist = 60; // How many items to store in
cache
static NSString* const kSMCalendarDataSourceEarlierKey = @"SMCalendarDataSourceEarlierKey";
static NSString* const kSMCalendarDataSourceTodayKey = @"SMCalendarDataSourceTodayKey";
static NSString* const kSMCalendarDataSourceTomorrowKey = @"SMCalendarDataSourceTomorrowKey";
```

Place the constants in the .m file immediately below the #imports.

When possible, make constants static and const. String constants that are used in other files are only const and are declared like this in the .m file:

```
NSString* const kSMCalendarDataSourceTomorrowKey = @"SMCalendarDataSourceTomorrowKey";
```

And like this in the .h file:

```
extern NSString* const kSMCalendarDataSourceTomorrowKey;
```

Memory management

Automatic Reference Counting

All new projects use ARC!

Object ownership and autorelease

Follow Apple’s guidelines for object ownership and method naming.

For non-ARC projects, if the method name begins with ”alloc” or ”new” or contains ”copy”, you take ownership of the object and need to explicitly call ”release” or ”autorelease”; all other methods return an autoreleased object.

Property assignment – non-ARC only

Use autorelease, then retain for assigning properties:

```
- (void)setPerson:(Person*)newPerson
{
     [person autorelease];
     person = [newPerson retain];
```

```
}
```

String property setters – non-ARC only

For string property setters, use copy:

```
- (void)setName:(NSString*)newName
{
      [name autorelease];
      name = [newName copy];
}
```

viewDidUnload

Nil any properties/instance variables that are set in viewDidLoad in viewDidUnload.

The basic rule is: "If it is created in viewDidLoad, it should be removed in viewDidUnload".

Releasing objects and nilling

When using ARC, objects need not be released in dealloc. However, delegates and so on should be set to nil so the current objects can be released.

Use properties to relinquish ownership of objects by setting them to nil.

Private vs. public methods

Only methods are declared in the .h file.

Do not declare private methods. (Xcode takes care of this automatically.)

```
// Public Methods (.h file)
- (void)loadBrochure: (BrochureModel*)newBrochure;
```

And use #pragma mark to create public/private method sections in the .m file:

```
#pragma mark - Public methods

// Sets the brochure to display
- (void)loadBrochure: (BrochureModel*)newBrochure
{
    // Release old and set new brochure
    [brochure release], brochure = newBrochure;
}

#pragma mark - Private methods

// Calls loadBrochureInfo if enough time has passed since last time we did
it or forceCheck is set to YES
- (void)checkForNewBrochureIfNecessary: (BOOL)forceCheck
{
    // ...
}
```

NSLog()

Never use NSLog() in production code unless it is information about a critical error!

Instead do the following:

- Add the DEBUG_LOG macros to the .pch file
- Use DEBUG_LOG(@"simple text") or DEBUG_LOG(@"%d results", [results count]) or DEBUG_POSITION in the code

This will allow you to keep all the debug messages since they are ignored when compiling for Release configuration.

Macros:

```
#if !defined( DEBUG_LOG ) && defined( DEBUG )
#define DEBUG_LOG( msg, ... ) NSLog( msg,  ##__VA_ARGS__ )
#define DEBUG_POSITION NSLog( @"-> %@ - %@", NSStringFromClass( [self class] ),
NSStringFromSelector(_cmd) )
#elif !defined( DEBUG_LOG ) && !defined( DEBUG )
#define DEBUG_LOG( msg, ... )
#define DEBUG_POSITION
#endif
```

Usage:

```
DEBUG_LOG( @"simple text without parameters" );
```
Writes a simple text to the console and log.

```
DEBUG_LOG( @"found %d results, time: %.2f, name: %@", [results count], theTime,
name );
```
Writes a formatted string to the console and log.

```
DEBUG_POSITION;
```
Writes the current position in the form "-> <class name> - <method signature>" (e.g. "-> Q8AppDelegate - applicationDidFinishLaunching:"). This is good for tracking flow.

Custom table cells

The preferred way of working with custom table cells is by using *prototype cells* in combination with storyboards.

Prototype cells

Custom cells should be made by creating a custom subclass of UITableViewCell, creating IBOutlets (manually adding @property in .h and @synthesize in .m) and wiring IBOutlets and interface elements in IB.

Only interface elements that should change from cell to cell should have IBOutlets, of course.

Note that using -[UITableView dequeueReusableCellWithIdentifier:] automatically creates a new cell if a reusable cell does not exist.

Sub-classing UITableViewCell

When sub-classing UITbaleViewCell there is some rules to adhere to:

1. Set fonts, text colors, static images and so on in Interface Builder. Set as many properties as possible here.
2. Set properties for remaining GUI objects (if necessary) in the **awakeFromNib** method (e.g. selected background color etc.)
3. Override the layoutSubview method to adjust size and position of labels. Be sure to call [super layoutSubviews] as the first thing in that method.
   Use sizeToFit and sizeWithFont:constrainedToSize:lineBreakMode: and similar methods to adjust text labels' sizes to fit contents.
   Make sure to check if the cell looks good in both portrait and landscape modes if required.
4. Implement setSelected:animated: *and* setHighlighted:animated: if a cell requires custom selected status (for example if a static image needs to be changed when the cell is selected). Be sure to call super's setSelected:animated:/setHighlighted:animated:
5. If all cells have the same height, use the **rowHeight** property on the table to specify height.
6. If cells have varying height depending on the cell's contents, create a method named **cellHeight** that returns the height of the cell.
7. Instead of setting each UI element individually in the table view delegate's cellForRowAtIndexPath method a custom method can be created that sets all the elements (e.g. setNewsItem:(NewsItem*)newsItem or setCustomer:(Person*)customer).
8. Any additional UI elements must be added to the cell's contentView – and never directly to the cell.
9. When positioning UI elements use the contentView's bounds rather that hardcoded values or the cell's bounds whenever possible. This will help making sure the cell behaves correctly when in editing mode and in both portrait and landscape orientations and on iPad.
10. Document any UI contants used. It is ok to enter the values directly and not create constants for everything but be sure to write a comment so it is easy to modify later on. (But, of course, constants with comments at the top of the file makes it even easier to read and modify the code.)

Other approaches

There are several other ways of creating custom table cells – for example:

- using XIB files and instantiating the cells using
  ```
  NSArray *nibItems = [[NSBundle mainBundle]
  loadNibNamed:@"JobItemTableCell" owner:self options:nil];
  cell = (JobItemTableCell*)[nibItems objectAtIndex:0];
  ```
- using XIB files and instantiating cells by using a UINib proxy object (same as above but the XIB file is only loaded into memory once)
- sub-classing UITableViewCell and adding or modifying sub-views

All three approaches are acceptable in principle but only under the right circumstances. Please do use prototype cells and storyboards if at all possible!