iOS Coding Guidelines

Introduction

The following list outline key elements of coding convention used in Here iOS app. It will present all examples in objective-C syntax as it is the main application language.

If you find yourself in a situation where our Objective-C Coding Guideline can't help you, please take a look on other Coding Guidelines linked at the end of this page. If you find an answer on those pages, discuss it with the team over Slack (#code-support channel) and update this page.

File header

Use following file header in your .h/.m files:

```
/*******************************************************************
 * Copyright © 2011-Present HERE Global B.V. All rights reserved. *
 *******************************************************************/
```

Naming

Should follow Apple naming convention as match as possible (camel-casing). Long, descriptive names are **obligatory**.

All classes, categories and typedefs name should be prefixed (right now it is 'HSI'), constants name also had common class prefix with context name.

Good:

```
@interface HSIBaseViewController
@interface HSIBaseViewController (RotationSupport)
const NSTimeInterval HSIBaseViewControllerAnimationTime = 0.5;
```

Bad:

```
@interface BaseViewController
@interface BaseViewController (RS)
const NSTimeInterval BaseAnimationTime = 0.5;
```

The same rule apply to protocols name, but additionally it should point its purpose, like data source or generic interface.

Example:

```
@protocol HSIPlaceDetailsDataSource;
```

Method, properties and ivar names should start with lowercase and follow camel-casing, it is good to mention again that names should be **as self-explanatory as possible**.

When ivars are used, they should start with an underscore.

Good:

```
- (NSString *)destinationName;
NSString *_destinationName;
@property (copy, nonatomic) NSString *destinationName;
```

Bad:

```
- (NSString *)destName;
NSString *destinationN;
@property (copy, nonatomic) NSString *DestinationName;
```

There is one exception to method names, in case of non-informal category method its names should be prefixed with category name and underscore.

Example:

```
- (void)rotationSupport_lockOrientation;
```

Spacing

Indent using 4 spaces, it should be mapped under Xcode IDE under the tab as a default behavior. There should be max 160 columns in a row.

Separate operation/s from different logics/contexts/abstractions with whitespace.

All kind of braces open and close on the next line, the only exception are blocks, which should start on the same line as the statement.

Examples:

```
// instance ivars
@implementation HSIPerson
{
 NSString *_name;
 NSString *_adress;
}

// method
- (NSString *)description
{
  return @"Person";
}

// if statement
if (!name)
{
 // Do something
}
else
{
 // Do something else
}

// for statement
for (NSInteger i = 0; i < 10; i++)
{
 // Do something
}

// pure enum
enum HSIMapState
{
 MapStateIdle = 1,
 MapStateDraw
}
```

Exceptions:

```
// block
{
 point = ...
 duration = ...

 [self.animator translateToPoint:point duration:duration completion:^{
  // Do something
 }];
}
```

Method declaration spacing should follow Apple style: starts with instance/class level operator, then space, return type, name of method with parameters separated by space.

Good:

```
- (void)translateToPoint:(CGPoint)point duration:(NSTimeInterval)duration;
+ (BOOL)isSubclassOfClass:(Class)aClass;
```

Bad:

```
-(void)translateToPoint:(CGPoint)point duration:(NSTimeInterval)duration;
+ (BOOL)isSubclassOfClass:(Class) aClass;
```

Rest of style(spacing) rules should be follow by custom Uncrustify config, which should be executed via git pre-commit hook.

To setup automatic code style validation, please follow bellow instruction:

1. In project root execute '*sudo rake install_uncrustify*' command, it will download a proper exec and move it to system executables
2. It could be done in automatic or manual way:
    - (automatic) In project root execute *'rake configure_git*', it will create hooks folder for git and move there all hooks that we already have. Be careful, it is a bit outdated file so it could generate some git config problems, but this is preferable why of installing hooks as this will also add hooks for commit message formatting
    - (manual) Move pre-commit.0001.here from scripts/git/ to .git/hooks/ and rename it to just pre-commit
3. Modify some *.[h/m] file (by adding more lines between statements) and add it to stage, then commit it, it will pass the commit but will prompt also warning that file violates code style rules

For ease of use you should get plugins manager for Xcode (Alcatraz) and install BBUncrustifyPlugin-Xcode plugin from packages browser.

Comments

When they are needed, comments should be used to explain **why** a particular piece of code does something, they should be only an extension of well-named code. Any comments that are used must be kept up-to-date or deleted.

Block comments should generally be avoided, as code should be as self-documenting as possible, with only the need for intermittent, few-line explanations. This does not apply to those comments used to generate documentation.

Properties

Prefer usage of properties instead of instance variables, it provides consistency in code and helps to debug the code if needed. There are some exception to previous rule, like not using property in initializer/dealloc method.

Accessing and mutating properties should be used via dot-notation, where brackets are reserved for methods.

Good:

```
self.place = @"Berlin";
NSString *newPlace = self.place;
```

Bad:

```
[self setPlace:@"Berlin"];
NSString *newPlace = [self place];
```

The order to declare the properties should be as follows:

- Memory management, when required (the implicit model should be favored, clang considers pointer variables strong by default and primitives as assign by default)
- Atomicity (default atomic should be omitted)

- Access model (when required)
- Nullability

Good:

```
@property (copy, nonatomic, readonly) NSArray *details;
@property (nonatomic) NSTimeInterval time;
@property (copy) NSString *name;
@property (nonatomic) NSDate *arriveDate;
```

Bad:

```
@property (nonatomic, copy, readonly) NSArray *details; //wrong order of
specifiers
@property (assign, nonatomic) NSTimeInterval time; //default specifiers not
required
@property (copy, atomic, nullable) NSString *name; //default specifiers not
required
@property (nonatomic, readwrite) NSDate *arriveDate; //default specifiers
not required
```

Methods

Designated Initializer

Use the NS_DESIGNATED_INITIALIZER to mark the class' designated initializer on the header

Good:

```
@interface TestClass: NSObject
- (instancetype)initWithText:(NSString *) font:(UIFont *)font
NS_DESIGNATED_INITIALIZER;
- (instancetype)initWithText:(NSString *);
@end
```

Nullability

Use Objective-C nullability annotations to make explicit if you expect nil arguments. The usage of NS_ASSUME_NONNULL_BEGIN is encouraged.

```
NS_ASSUME_NONNULL_BEGIN
@interface TestClass: NSObject

@property (nonatomic, nullable) UIColor *color;
@property (nonatomic) BOOL aborted;

- (nullable NSString *)randomThingWithSeed(nullable NSString *)seed;
@end
NS_ASSUME_NONNULL_END
```

Literals

Sugar syntax should be used for all immutable classes, like NSDictionary, NSArray, NSString or NSNumber. Note the round bracket around number literals, those are strongly suggested.

Good:

```
NSDictionary *countriesOfPlaces = @{@"Berlin":@"Germany",
@"Szczecin":@"Poland", @"Tallinn":@"Estonia", @"Vancouver":@"Canada"};
NSArray *places = @[@"Berlin", @"Szczecin", @"Tallinn", @"Vancouver"];
NSString *place = @"Berlin";
NSNumber *age = @(18);
```

Bad:

```
NSDictionary *countriesOfPlaces = [NSDictionary
dictionaryWithObjectsAndKeys:@"Germany", @"Berlin", @"Poland", @"Szczecin",
@"Estonia", @"Tallinn", @"Canada", @"Vancouver", nil];
NSArray *places = [NSArray arrayWithObjects:@"Berlin", @"Szczecin",
@"Tallinn", @"Vancouver"];
NSNumber *age = [NSNumber numberWithInt:18];
```

It is important especially for NSArray and NSDictionary, where inserting nil won't raise as an exception and could lead to hard to find bugs.

Braces

Make sure to use curly braces to indicate block of code, even if the block is one line

Good:

```
for (NSInteger i = 0; i < 10; i++)
{
    i++;
}
j++;
```

Bad:

```
for (NSInteger i = 0; i < 10; i++)
    i++;
    j++;
```

Use round braces to indicate order of operations:

Good:

```
if ((value & mask) == kMaskValueOne)
```

Bad:

```
if (value & mask == kMaskValueOne)
```

Constants

Constants should be used instead of in-line string literals or numbers, it is easier to maintain and less error-prone code.

They should be decelerated with const and static keywords, not #define unless explicitly being used as a macro.

Const names should start with HSI and be camel cased.

For non-pointer types const qualifier should be placed before type name.

Good:

```
static NSString *const HSIRouteOptionsKey = @"routeOptions";
static const CGFloat HSIDefaultAnimationTime = 0.5f;
static const CGFloat HSIHorizontalPadding = 10;

CGRect frame = CGRectMake(CGRectGetMinX(self.frame),
CGRectGetMinY(self.frame), CGRectGetWidth(self.frame) +
HSIHorizontalPadding, CGRectGetHeight(self.frame));
```

Bad:

```
#define HSIROUTEOPTIONS @"routeOptions"
#define HSIDEFAULTANIMATIONTIME 0.5f

CGRect frame = CGRectMake(0, 0, CGRectGetWidth(self.frame) + 22,
CGRectGetHeight(self.frame));

CGFloat const HSIHorizontalPadding = 10; //should be 'const CGFloat'
```

Enums

In case of enumerated types defintion it should be wrapped in NS_ENUM macro for types and NS_OPTIONS macro for bitmasks, they have stronger type checking and code completion.

Examples:

```
// normal enum
typedef NS_ENUM(NSInteger, HSIMapState)
{
 HSIMapStateIdle, // 0 is default value for first item
 HSIMapStateDraw
};

// bitmask enum
typedef NS_OPTIONS(NSUInteger, HSIPlaceCategory)
{
 HSIPlaceCategoryCafe  = 1 << 0,
 HSIPlaceCategoryRestaurant  = 1 << 1,
 HSIPlaceCategoryMuseum  = 1 << 2
}
```

Blocks

Blocks used as property/ivar should be defined using typedef keyword.

Examples:

```
typedef void(^HSIMapUpdateBlock)();
typedef void(^HSIMapPlaceBlock)(NSString *name, NMAGeoCoordinates
*geoCoordinates);
typedef NSString *(^HSIMapForceNameBlock)();

// Property
@property (copy, nonatomic) HSIMapUpdateBlock mapUpdate;

...
{
 self.mapUpdate = ^{
  // Do something
 };
}
...

// Instance variable
HSIMapPlaceBlock mapPlace;

...
{
 mapPlace = ^(NSString *name, NMAGeoCoordinates *geoCoordinates){
  // Do something
 };
}
...
```

Object types and Protocols

Don't put a space between an object type and the protocol it conforms to.

Good:

```
@property (attributes) id<Protocol> object;
@property (nonatomic) NSObject<Protocol> *object;
- (void)setDelegate:(id<MyFancyDelegate>)delegate;
```

Bad:

```
@property (attributes) id <Protocol> object; //extra space between id and
<Protocol>
@property (nonatomic) NSObject <Protocol> *object; //extra space between
NSObject and <Protocol>
- (void)setDelegate:(id <MyFancyDelegate>)delegate; //extra space between
id and <MyFancyDelegate>
```

Reference pages

- New York Times style guide (https://github.com/NYTimes/objective-c-style-guide#dot-notation-syntax)
- GitHub style guide (https://github.com/github/objective-c-style-guide)
- Google style guide (https://google.github.io/styleguide/objcguide.xml)
- Futurice good practices (https://github.com/futurice/ios-good-practices/)
- Alcatraz plugins manager (http://alcatraz.io/)