Adopting Modern Objective-C

Important: This document is no longer being updated. For the latest information about Apple SDKs, visit the documentation website.

Over the years, the Objective-C language has grown and evolved. Although the core concepts and practices remain the same, parts of the language have been through significant changes and improvements. These modernizations improve type safety, memory management, performance, and other aspects of Objective-C, making it easier for you to write correct code. It's important to adopt these changes in your existing and future code to help it become more consistent, readable, and resilient.

Xcode provides a tool to help make some of these structural changes for you. But before you use this tool, you want to understand what changes it will offer to make to your code, and why. This document highlights some of the most significant and useful modernizations to adopt in your codebase.

instancetype

Use the `instancetype` keyword as the return type of methods that return an instance of the class they are called on (or a subclass of that class). These methods include `alloc`, `init`, and class factory methods.

Using `instancetype` instead of `id` in appropriate places improves type safety in your Objective-C code. For example, consider the following code:

```
@interface MyObject : NSObject
+ (instancetype)factoryMethodA;
+ (id)factoryMethodB;
@end

@implementation MyObject
+ (instancetype)factoryMethodA { return [[[self class] alloc] init]; }
+ (id)factoryMethodB { return [[[self class] alloc] init]; }
@end

void doSomething() {
    NSUInteger x, y;

    x = [[MyObject factoryMethodA] count]; // Return type of +factoryMethodA is taken to be "MyObject *"
    y = [[MyObject factoryMethodB] count]; // Return type of +factoryMethodB is "id"
}
```

Because of the `instancetype` return type of `+factoryMethodA`, the type of that message expression is `MyObject *`. Since `MyObject` doesn't have a `-count` method, the compiler gives a warning about the `x` line:

```
main.m: 'MyObject' may not respond to 'count'
```

However, because of the `id` return type in `+factoryMethodB`, the compiler can give no warning about the `y` line. Since an object of type `id` can be any class, and since a method called `-count` exists somewhere on some class, to the compiler it's possible that the return value of `+factoryMethodB` implements the method.

To make sure `instancetype` factory methods have the right subclassing behavior, be sure to use `[self class]` when allocating the class rather than referring directly to the class name. Following this convention ensures that the compiler will infer subclass types correctly. For example, consider trying to do this with a subclass of `MyObject` from the previous example:

```
@interface MyObjectSubclass : MyObject
@end


void doSomethingElse() {
        NSString *aString = [MyObjectSubclass factoryMethodA];
}
```

The compiler gives the following warning about this code:

```
main.m: Incompatible pointer types initializing 'NSString *' with an expression of type
'MyObjectSubclass *'
```

In the example, the `+factoryMethodA` message send returns an object of type `MyObjectSubclass`, which is the type of the receiver. The compiler appropriately determines that the return type of `+factoryMethodA` should be of the subclass `MyObjectSubclass`, not of the superclass in which the factory method was declared.

How to Adopt

In your code, replace occurrences of `id` as a return value with `instancetype` where appropriate. This is typically the case for `init` methods and class factory methods. Even though the compiler automatically converts methods that begin with "alloc," "init," or "new" and have a return type of `id` to return `instancetype`, it doesn't convert other methods. Objective-C convention is to write `instancetype` explicitly for all methods.

Note that you should replace `id` with `instancetype` for return values only, not elsewhere in your code. Unlike `id`, the `instancetype` keyword can be used only as the result type in a method declaration.

For example:

```
@interface MyObject
- (id)myFactoryMethod;
@end
```

should become:

```
@interface MyObject
- (instancetype)myFactoryMethod;
@end
```

Alternatively, you can use the modern Objective-C converter in Xcode to make this change to your code automatically. For more information, see Refactoring Your Code Using Xcode.

Properties

An Objective-C *property* is a public or private method declared with the `@property` syntax.

```
@property (readonly, getter=isBlue) BOOL blue;
```

Properties capture the state of an object. They reflect the object's intrinsic attributes and relationships to other objects. Properties provide a safe, convenient way to interact with these attributes without having to write a set of custom accessor methods (although properties do allow custom getters and setters, if desired).

Using properties instead of instance variables in as many places as possible provides many benefits:

- **Autosynthesized getters and setters.** When you declare a property, by default getter and setter methods are created for you.
- **Better declaration of intent of a set of methods.** Because of accessor method naming conventions, it's clear exactly what the getter and setter are doing.
- **Property keywords that express additional information about behavior.** Properties provide the potential for declaration of attributes like `assign` (vs `copy`), `weak`, `atomic` (vs `nonatomic`), and so on.

Property methods follow a simple naming convention. The *getter* is the name of the property (for example, `date`), and the *setter* is the name of the property with the *set* prefix, written in camel case (for example, `setDate`). The naming convention for Boolean properties is to declare them with a named getter starting with the word "is":

```
@property (readonly, getter=isBlue) BOOL blue;
```

As a result, all of the following work:

```
if (color.blue) { }
if (color.isBlue) { }
if ([color isBlue]) { }
```

When deciding what can be a property, keep in mind that the following are not properties:

- `init` method
- `copy` method, `mutableCopy` method
- A class factory method
- A method that initiates an action and returns a `BOOL` result
- A method that explicitly changes internal state as a side effect of a getter

Additionally, consider the following set of rules when identifying potential properties in your code:

- A read/write property has two accessor methods. The setter takes one argument and returns nothing, and the getter takes no arguments and returns one value. If you convert this set of methods into a property, tag it with the `readwrite` keyword.
- A read-only property has a single accessor method, the getter, which takes no arguments and returns one value. If you convert this method into a property, tag it with the `readonly` keyword.
- The getter should be *idempotent* (if a getter is called twice, the second call results in the same result as the first). However, it is acceptable for a getter to compute the result each time it's called.

How to Adopt

Identify a set of methods that qualify to be converted into a property, such as these:

```
- (NSColor *)backgroundColor;
- (void)setBackgroundColor:(NSColor *)color;
```

and declare them using the `@property` syntax with the appropriate keyword(s):

```
@property (copy) NSColor *backgroundColor;
```

For information about property keywords and other considerations, see Encapsulating Data.

Alternatively, you can use the modern Objective-C converter in Xcode to make this change to your code automatically. For more information, see Refactoring Your Code Using Xcode.

Enumeration Macros

The `NS_ENUM` and `NS_OPTIONS` macros provide a concise, simple way of defining enumerations and options in C-based languages. These macros improve code completion in Xcode and explicitly specify the type and size of your enumerations and options. Additionally, this syntax declares enums in a way that is evaluated correctly by older compilers, and by newer ones that can interpret the underlying type information.

Use the `NS_ENUM` macro to define *enumerations*, a set of values that are mutually exclusive:

```
typedef NS_ENUM(NSInteger, UITableViewCellStyle) {
        UITableViewCellStyleDefault,
        UITableViewCellStyleValue1,
        UITableViewCellStyleValue2,
        UITableViewCellStyleSubtitle
};
```

The `NS_ENUM` macro helps define both the name and type of the enumeration, in this case named `UITableViewCellStyle` of type `NSInteger`. The type for enumerations should be `NSInteger`.

Use the `NS_OPTIONS` macro to define *options*, a set of bitmasked values that may be combined together:

```
typedef NS_OPTIONS(NSUInteger, UIViewAutoresizing) {
        UIViewAutoresizingNone                 = 0,
        UIViewAutoresizingFlexibleLeftMargin   = 1 << 0,
        UIViewAutoresizingFlexibleWidth        = 1 << 1,
        UIViewAutoresizingFlexibleRightMargin  = 1 << 2,
        UIViewAutoresizingFlexibleTopMargin    = 1 << 3,
        UIViewAutoresizingFlexibleHeight       = 1 << 4,
        UIViewAutoresizingFlexibleBottomMargin = 1 << 5
};
```

Like enumerations, the `NS_OPTIONS` macro defines both a name and a type. However, the type for options should usually be `NSUInteger`.

How to Adopt

Replace your `enum` declarations, such as this one:

```
enum {
        UITableViewCellStyleDefault,
        UITableViewCellStyleValue1,
        UITableViewCellStyleValue2,
        UITableViewCellStyleSubtitle
};
typedef NSInteger UITableViewCellStyle;
```

with the `NS_ENUM` syntax:

```
typedef NS_ENUM(NSInteger, UITableViewCellStyle) {
        UITableViewCellStyleDefault,
        UITableViewCellStyleValue1,
        UITableViewCellStyleValue2,
        UITableViewCellStyleSubtitle
};
```

But when you use `enum` to define a bitmask, such as here:

```
enum {
        UIViewAutoresizingNone                 = 0,
        UIViewAutoresizingFlexibleLeftMargin   = 1 << 0,
        UIViewAutoresizingFlexibleWidth        = 1 << 1,
        UIViewAutoresizingFlexibleRightMargin  = 1 << 2,
        UIViewAutoresizingFlexibleTopMargin    = 1 << 3,
        UIViewAutoresizingFlexibleHeight       = 1 << 4,
        UIViewAutoresizingFlexibleBottomMargin = 1 << 5
};
typedef NSUInteger UIViewAutoresizing;
```

use the `NS_OPTIONS` macro.

```
typedef NS_OPTIONS(NSUInteger, UIViewAutoresizing) {
        UIViewAutoresizingNone                 = 0,
        UIViewAutoresizingFlexibleLeftMargin   = 1 << 0,
        UIViewAutoresizingFlexibleWidth        = 1 << 1,
        UIViewAutoresizingFlexibleRightMargin  = 1 << 2,
        UIViewAutoresizingFlexibleTopMargin    = 1 << 3,
        UIViewAutoresizingFlexibleHeight       = 1 << 4,
        UIViewAutoresizingFlexibleBottomMargin = 1 << 5
};
```

Alternatively, you can use the modern Objective-C converter in Xcode to make this change to your code automatically. For more information, see Refactoring Your Code Using Xcode.

Object Initialization

In Objective-C, object initialization is based on the notion of a *designated initializer*, an initializer method that is responsible for calling one of its superclass's initializers and then initializing its own instance variables. Initializers that are not designated initializers are known as *convenience initializers*. Convenience initializers typically delegate to another initializer—eventually terminating the chain at a designated initializer—rather than performing initialization themselves.

The designated initializer pattern helps ensure that inherited initializers properly initialize all instance variables. A subclass that needs to perform nontrivial initialization should override all of its superclass's designated initializers, but it does not need to override the convenience initializers. For more information about initializers, see Object Initialization.

To clarify the distinction between designated and designated initializers clear, you can add the `NS_DESIGNATED_INITIALIZER` macro to any method in the `init` family, denoting it a designated initializer. Using this macro introduces a few restrictions:

- The implementation of a designated initializer must chain to a superclass `init` method (with `[super init...]`) that is a designated initializer for the superclass.
- The implementation of a convenience initializer (an initializer not marked as a designated initializer within a class that has at least one initializer marked as a designated initializer) must delegate to another initializer (with `[self init...]`).
- If a class provides one or more designated initializers, it must implement all of the designated initializers of its superclass.

If any of these restrictions are violated, you receive warnings from the compiler.

If you use the `NS_DESIGNATED_INITIALIZER` macro in your class, you need to mark all of your designated initializers with this macro. All other initializers will be considered to be convenience initializers.

How to Adopt

Identify designated initializers in your classes, and tag them with the `NS_DESIGNATED_INITIALIZER` macro. For example:

```
- (instancetype)init NS_DESIGNATED_INITIALIZER;
```

Automatic Reference Counting (ARC)

Automatic Reference Counting (ARC) is a compiler feature that provides automatic memory management of Objective-C objects. Instead of your having to remember when to use `retain`, `release`, and `autorelease`, ARC evaluates the lifetime requirements of your objects and automatically inserts appropriate memory management calls for you at compile time. The compiler also generates appropriate `dealloc` methods for you.

How to Adopt

Xcode provides a tool that automates the mechanical parts of the ARC conversion (such as removing `retain` and `release` calls) and helps you to fix issues that the migrator can't handle automatically. To

use the ARC migration tool, choose Edit > Refactor > Convert to Objective-C ARC. The migration tool converts all files in a project to use ARC.

For more information, see *Transitioning to ARC Release Notes*.

Refactoring Your Code Using Xcode

Xcode provides a modern Objective-C converter that can assist you during the modernization process. Although the converter helps with the mechanics of identifying and applying potential modernizations, it doesn't interpret the semantics of your code. For example, it won't detect that your `-toggle` method is an action that affects your object's state, and it will erroneously offer to modernize this action to be a property. Make sure to manually review and confirm any changes the converter offers to make to your code.

Of the previously described modernizations, the converter offers to:

- Change `id` to `instancetype` in appropriate places
- Change `enum` to `NS_ENUM` or `NS_OPTIONS`
- Update to the `@property` syntax

Besides these modernizations, this converter recommends additional changes to your code, including:

- Converting to literals, so a statement like `[NSNumber numberWithInt:3]` becomes `@3`.
- Using subscripting, so a statement like `[dictionary setObject:@3 forKey:key]` becomes `dictionary[key] = @3`.

To use the modern Objective-C converter, choose Edit > Refactor > Convert to Modern Objective-C Syntax.
