GeekBrains

Лекция 2. Объекты. Наследование, инкапсуляция и полиморфизм в Objective-C

Знакомство с работой объектов в Objective-C. Принципы объектно – ориентированного программирования на Objective-C.

Оглавление

Введение в классы

Objective-C — объектно – ориентированный язык, что означает, что он использует объекты везде и всегда, когда это возможно. В контексте его долгой истории этот подход имел смысл. ООП был новым и лучшим подходом в разработке, когда создавался язык Objective-C, и, честно говоря, этот подход хорошо работает уже много десятков лет.

Несмотря на свою центральную важность для языка, объектная ориентация в Objective-C остается на удивление неряшливой, как вы скоро увидите. Отчасти это связано с тем, что эта часть языка значительно изменилась с момента его выпуска, особенно за последнее десятилетие или около того, поэтому вы найдете множество стилей кода.

Никто не знает с чем вы столкнетесь в коде Objective-C, который вы найдете в дикой природе легаси кода, поэтому в данной лекции вы научитесь различным подходам, но перед погружением в объекты и ООП повторим основные термины.

Терминология

Класс – это структура, которая формирует тип и задает действия объекта. Объекты получают от классов сведения о возможном поведении, которое полностью реализуют в себе. В крупных программах счет может идти на десятки классов. Согласно стилю программирования на Objective-C названия классов начинаются с прописной буквы.

Сообщение(аналог вызова метода в Swift) – это действие, которое способен выполнить объект. В коде его реализация выглядит так:

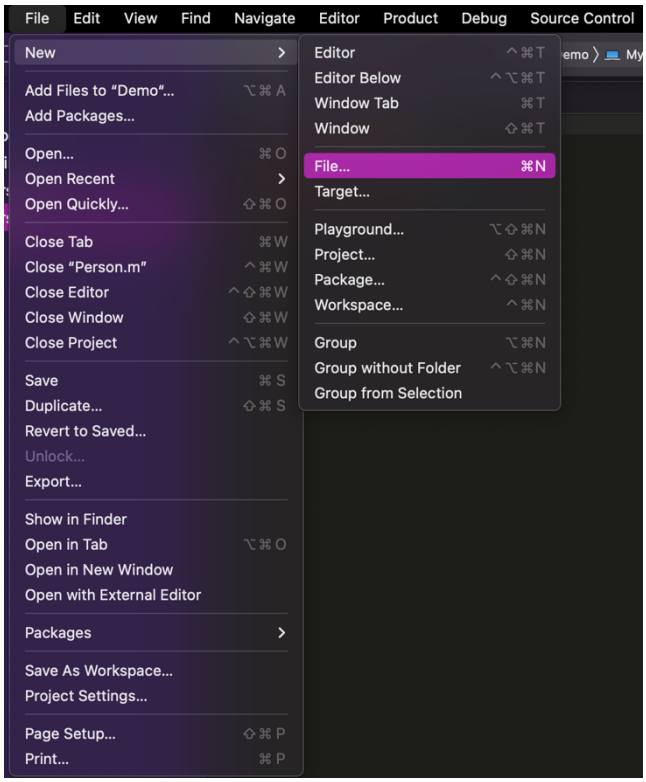
```
[project method];
```

Объекту **project** посылается сообщение **method**. При получении он обращается к исходному классу, чтобы найти код, который необходимо выполнить.

Метод – это код, который выполняется в ответ на сообщение.
Интерфейс – это описание возможностей класса (**@interface**).
Реализация – это код, который реализует описанные в интерфейсе возможности.

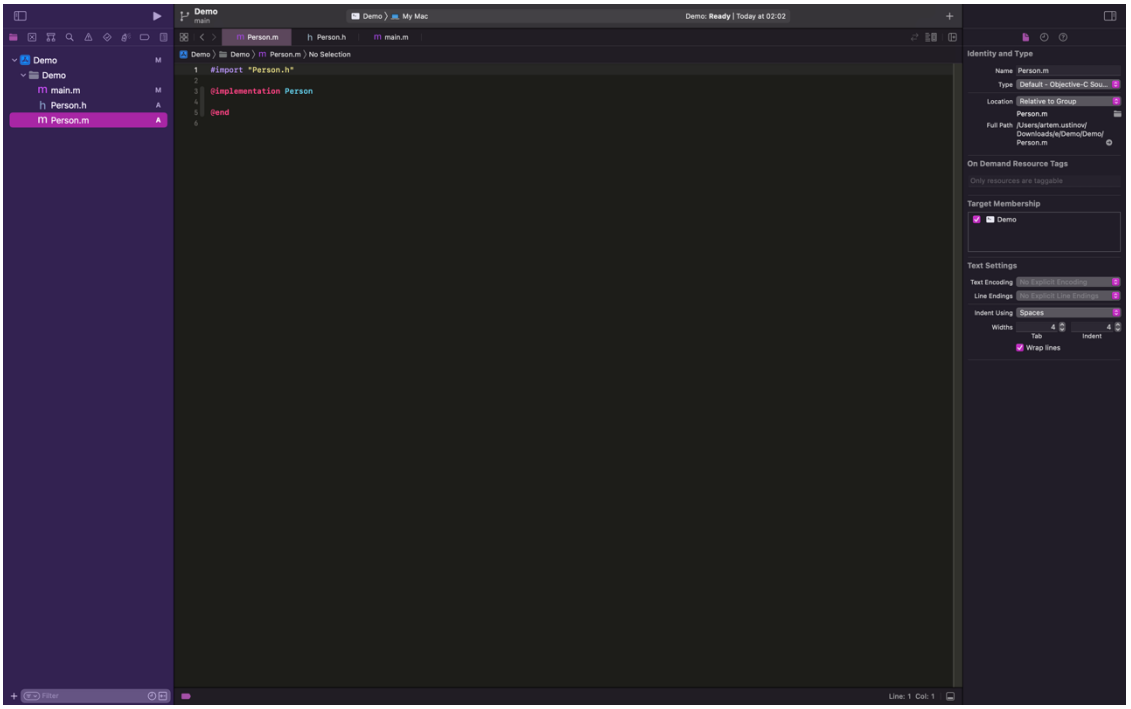
Создание класса

Если вы используете проект командной строки Xcode, перейдите в меню «Файл» и выберите «Создать» > «Файл», затем выберите > «Cocoa Class». Дайте ему имя Person, пусть он будет подклассом NSObject, затем нажмите «Далее», затем «Готово».



Это создаст два файла, Person.h и Person.m.

Вы увидите @interface Person:NSObject в Person.h, где определяется внешний интерфейс для класса, которые должны использовать другие части вашего кода. Файл Person.m содержит @implementation Person, куда вы поместите реализацию своего интерфейса, то есть исходный код для выполнения создаваемых вами методов.



Итак, теоретически вы определяете свои методы и свойства в заголовочном файле, а затем реализуете методы в файле реализации. На практике все немного запутаннее, как вы скоро увидите.

Методы

Методы — это одно из мест, где Objective-C больше всего похож на Swift, и в некотором смысле обучение чтению и написанию методов Objective-C помогает вам раньше увидеть, как появился синтаксис Swift.

Начнем с простого: создадим метод printGreeting. Это не будет принимать никаких параметров и ничего не возвращать, поэтому оно написано так:

Поместите этот код в Person.h между строками @interface Person: NSObject и @end и реализацию непосредственно в Person.m файл. Тут есть несколько важных вещей, на которые стоит обратить внимание:

Person.h

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

NS_ASSUME_NONNULL_BEGIN

@interface Person : NSObject

- (void)printGreeting;

@end

NS_ASSUME_NONNULL_END

```

Person.m

```objc
#import "Person.h"

@implementation Person

- (void)printGreeting {
    NSLog(@"Hello!");
}
@end

```

1. “–“ отмечает начало обычного метода. Если бы вместо этого мы использовали “+”, это был бы статический метод, то есть тот, который принадлежит классу, а не экземплярам.

2. Тип возвращаемого значения помещается в круглые скобки перед именем метода. void означает «ничего не возвращается».

3. Обычно пробел ставится после “-“, но не перед именем метода. Поэтому не ставьте пробел после закрывающей скобки слова (void).

Если мы хотим использовать этот метод в main.m, нам нужно сначала сделать пару вещей. Откройте main.m сейчас и замените существующую функцию main() на эту:

```objc
#import <Foundation/Foundation.h>
#import "Person.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Person *person = [Person new];
        [person printGreeting];
    }
    return 0;
}


```

Это код создает новый объект Person, а затем вызывает для него printGreeting. Но, не забудьте импортировать сущность #import “Person.h”, иначе код не скомпилируется: вы получите ошибку «Use of undeclared identifier ‘person’»

В Swift любой файл, который вы создаете в своем проекте, автоматически встраивается в пространство имен вашего приложения, что означает, что класс, который вы объявляете в файле A, автоматически доступен в файле B. В Objective-C

- не так. Нам нужно прописывать вручную #import файл заголовка для Person.h, чтобы иметь возможность использовать его в main.m. Итак, проверьте, добавлен ли этот импорт в main.m: #import "Person.h".

🔥 **Внимание!** Есть еще одна загвоздка, но на этот раз более серьезная: в Objective-C нет никакого понимания приватных методов. Конечно, мы можем избежать объявления метода в заголовочном файле, но любой может вызвать его, если знает, как это сделать.

Например, этот код работает после удаления printGreeting из Person.h:

```
Person *person = [Person new];
[person performSelector:@selector(printGreeting)];
```

Теперь, этот код выдаст ошибку. В большинстве случаев разница между селектором и методом на самом деле не имеет значения, но здесь она имеет значение: метод — это фактическая реализация некоторого кода в классе, а селектор — это просто имя метода.

Это различие не только академическое, но вам, возможно, придется прочитать следующее несколько раз, чтобы оно имело смысл.

Представьте, что у нас есть еще один класс, Dog, в котором тоже есть метод printGreeting, который выглядит точно так же, как метод printGreeting из Person: без параметров или возвращаемого значения. Несмотря на то, что сам метод очень разный — один в классе Dog, а другой в классе Person, и, предположительно, один печатает «Гав!» – селектор идентичен. Прямо сейчас ваш код PerformSelector выдает ошибку «Необъявленный селектор printGreeting», потому что вы удалили объявление в Person.h. Но если бы вы добавили #import для нашего гипотетического класса Dog, эта ошибка исчезла бы: Xcode не волнует, что printGreeting не объявлен в Person.h, пока он где-то объявлен.

Это означает, что вы можете отправлять сообщения объектам, даже если они не указаны в их интерфейсе, и Xcode не будет предупреждать вас, если тот же самый селектор объявлен в другом заголовочном файле. Хуже того, объект, которому вы его отправляете, может даже не поддерживать этот селектор, и в этом случае ваше приложение упадет.

Свойства

У Objective-C непростые отношения со свойствами, в основном потому, что они были введены как концепция только после того, как язык уже существовал около 20 лет. Что еще более сбивает с толку, свойства сами по себе развивались с тех пор, как они были представлены, так что, откровенно говоря, это удача, с которой вы столкнетесь в дикой природе.

🔥 **Предупреждение:** свойства — одна из самых сложных концепций в Objective-C, и эта глава длинная по одной причине: нужно многому научиться, и независимо от того, с каким кодом вы в итоге будете работать. Работа со свойствами – это базовая концепция для написания кода.

Instance variables (Переменные экземляра)

Переменные экземпляра (обычно называемые ivars) выглядят как свойства Swift, но в Objective-C они намного сложнее и больше не используются сами по себе. Мы собираемся изменить класс Person, чтобы он имел переменную экземпляра имени.

Вот новый файл Person.h:

```
@interface Person : NSObject {
    @public
    NSString *name;
}
- (void)printGreeting;
@end
```

Немного изменим наш метод printGreeting и выведем еще свойство name

```
- (void)printGreeting {
    NSLog(@"Hello! %@", name);
}
```

Посмотрим на синтаксис более внимательно. Чтобы создать ivars внутри класса, вам нужно открыть фигурную скобку сразу после строки @interface. Затем вы перечисляете каждый нужный вам ivar с его типом данных и именем, что немного похоже на свойства Swift. Однако обратите внимание, что мне пришлось поместить @public перед ivar: без этого значение недоступно вне класса.
При настройке ivar в main.m вам нужно использовать оператор непрямого доступа к членам ->, например:

```objc
#import <Foundation/Foundation.h>
#import "Person.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Person *person = [Person new];
        person->name = @"Str";
        [person printGreeting];
    }
    return 0;
}
```

_instanceVariable для компилятора аналогично self->instanceVariable.

Instance variable напрямую использовать не рекомендуется, это описано в документе "Adopting Modern Objective-C" на сайте Apple Developer. Вместо этого предлагается использовать **property** с нужными атрибутами, для которых компилятор сам создаст соответствующие **ivar'ы** и сгенерирует геттеры и сеттеры.

Следующие случаи, когда следует обращаться напрямую к ivar'ам вместо property:

1. В инициализаторах, поскольку геттеры и сеттеры могут иметь побочные эффекты, зависящие от ещё не проинициализированных свойств
2. В реализации протокола NSCopying при прямом переносе значений ivar'ов между исходным объектом и его копией (причина такая же, что и прошлом пункте)
3. В реализации собственных геттеров и сеттеров.

Решение с iVar применяется редко из-за своего недостатка – оно требует определить строение объекта на стадии компиляции. При каждом обращении к переменной компилятор задает фиксированное смещение в памяти. Такой подход работает корректно до тех пор, пока в объект не будет добавлена другая переменная.
Например, если добавить перед name еще одну переменную, то смещение, которое прежде указывало на name, теперь будет указывать на нее. Код, где применяется фиксированное смещение, прочитает неверное значение.

- **@property (свойства)**

В Objective-C свойство — это метод, который получает и устанавливает значение ivar. В более старых версиях компилятора вам нужно было создать ivar, объявить свойство, а затем указать компилятору соединить их. Начиная с Xcode 4.4 это было упрощено, и в результате вам больше не нужно беспокоиться об иварах: свойства могут сделать все за вас.

Как вы скоро увидите, существует множество способов записи свойств. Тем не

менее, проще всего просто написать @property перед тем, что раньше было вашим иваром, хотя вам нужно поместить это вне фигурных скобок, где у вас были ваши ивары ранее. Поскольку для чистых свойств вообще не нужны ивары, фигурные скобки обычно полностью удаляются. Например, Person.h будет выглядеть так:

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

NS_ASSUME_NONNULL_BEGIN

@interface Person : NSObject

@property (nonatomic, strong) NSString *name;

- (void)printGreeting;

@end

NS_ASSUME_NONNULL_END

```

Изначально устанавливается ключевое имя **@property**, которое дает компилятору знать, что будет объявлено свойство. Затем устанавливаются атрибуты (о них пойдет речь в следующем уроке). После этого указывается тип свойства и его имя.
При объявлении свойств компилятор автоматически синтезирует его в реализации. В результате создаются переменные, которые имеют структуру имени «подчеркивание + имя свойства». Именем переменной можно управлять при помощи **@synthesize** в реализации класса:

```
@implementation Object
@synthesize name = objectName;

@end

```

В результате вместо переменной **_name** будет синтезирована переменная с именем **objectName**. Рекомендуется применять имена по умолчанию: соблюдение единых соглашений об именах упрощает чтение кода.
При инициализации объекта всем его свойствам проставляются стандартные значения, как и у обычных переменных. Чтобы присвоить иное значение, необходимо использовать следующую конструкцию:

```
@implementation Object
@synthesize name = objectName;

- (instancetype)init
{
    self = [super init];
    if (self) {
        objectName = @"Name";    // Присвоение значения переменной свойства
        self.name = @"Name";     // Присвоение значения свойству
        [self setName:@"Name"];  // Аналогичное присвоение значения свойству
    }
    return self;
}

@end

```

При инициализации свойство **name** будет иметь значение “**Name**”.

Использовать свойство можно двумя способами:

- Как члена класса;
- Как свойства класса.

```
// Обращение к члену класса
_name = @"Name";
// Обращение к свойству класса
self.name = @"Name";

```

В чем разница вызова члена и свойства класса? Чтобы ответить на этот вопрос, определим термины **getter** и **setter**. Они генерируются автоматически после объявления свойства, и их можно переопределить необходимым образом.

- **Getter**

Геттер (Getter) – это метод для получения значения свойства класса. Выглядит он следующим образом:

```
- (NSString *)name {
    return @"Name";
}

```

В результате вызова свойства класса будет всегда возвращаться значение **Name**, так как метод геттера был переопределен для свойства **name**. В этом и заключается разница в вызове свойства и члена класса. При вызове свойства класса вызывается геттер этого свойства, а при обращении к члену класса возвращается значение, которое хранится в нем (последнее установленное значение).

- **Setter**

Сеттер (Setter) – это метод для установления значения свойству класса. Имеет следующую конструкцию:

```
- (void)setName:(NSString *)name {
    _name = name;
}

```

В результате обращения к сеттеру устанавливается значение свойству. Если не указать в теле сеттера установление значения члену класса, то он не будет хранить значение, которое установлено. При присваивании значения свойству вызывается его сеттер и обрабатывается вся необходимая логика, которая в нем указана.

Пример установления значения, при котором вызывается сеттер:

```
self.prop = @"Value";          // Вызов из класса
[self setProp:@"Value"];       // Аналогичный вызов из класса

object.prop = @"Value";        // Вызов извне
[object setProp:@"Value"];     // Аналогичный вызов извне

```

Теперь можем сделать вывод об отличии свойства класса от члена класса. При обращении к свойству класса и установлении значения вызывается его сеттер и геттер. При обращении к члену класса они не вызываются.

Член класса можно представить в виде обычной переменной, которая хранит значение. Более того, к члену класса нельзя обратиться извне: его можно применять только внутри данного класса.

Свойства всегда должны использоваться для внешних обращений к внутренним переменным экземпляров объекта, но существует несколько способов внутренних обращений к переменным. Внутри самого объекта тоже можно вызвать геттер и сеттер, используя ключевое слово **self**. Другой вариант – изменить саму переменную напрямую. Последний способ предпочтительнее, так как доступ будет быстрее, если не задействовать дополнительные методы.

💡 **Замечание:** как мы уже поняли, если объявлять переменные или методы в .h файлы (header file), то переменные будут доступны везде, так, а как сделать приватные переменные?

Для этого необходимо объявить такую конструкцию в .m файле.

```
@interface Person ()
@property NSString *secondName;
@end
```

Данная переменная secondName будет доступна только в данном файле.

Атрибуты свойства

Когда свойства начинают усложняться, это значит к ним присоединяются атрибуты. Это инструкции компилятора, которые влияют на автоматически сгенерированные методы доступа и бывают разных форм. Мы не используем никаких явных атрибутов прямо сейчас, но мы используем неявные атрибуты: нашему свойству имени присваиваются значения по умолчанию, которые влияют на то, как оно

работает. Со временем эти значения по умолчанию немного изменились, но, по моему опыту, большинство разработчиков просто пишут атрибуты для всего, даже если они совпадают со значениями по умолчанию.

Для начала поделим все атрибуты, которые есть у свойства, на группы:

1. атрибуты доступности (readonly/readwrite),
2. атрибуты владения (retain/strong/copy/assign/unsafe_unretained/weak),
3. атрибут атомарности (atomic/nonatomic).
4. nullability атрибут (null_unspecified/null_resettable/nullable/nonnull) — появился в xcode 6.3

Атрибуты, позволяющие задать имя сеттера и геттера, рассматривать не будем — для них как таковых правил нет, за исключением тех, что предусматривает используемый вами Code Style. Явно или неявно, но атрибуты всех типов указываются у каждого свойства.

Атрибуты доступности

- **readwrite** — указывает, что свойство доступно и на чтение, и на запись, то есть будут сгенерированы и сеттер, и геттер. Это значение задается всем свойствам по умолчанию, если не задано другое значение.
- **readonly** — указывает, что свойство доступно только для чтения. Это значение стоит применять в случаях, когда изменение свойства «снаружи» во время выполнения объектом своей задачи нежелательно, либо когда значение свойства не хранится ни в какой переменной, а генерируется исходя из значений других свойств. Например, есть у объекта User свойства firstName и lastName, и для удобства заданное readonly свойство fullName, значение которого генерируется исходя из значений первых двух свойств.

В случае, когда нежелательно изменение свойства «снаружи», оно, как правило, объявляется в интерфейсе класса как readonly, а потом переопределяется как readwrite в расширении класса (class extension), чтобы внутри класса также изменять значение не напрямую в переменной, а через сеттер.

Атрибуты владения

Это самый обширный тип атрибутов, во многом из-за сосуществования ручного и автоматического управления памятью. При включенном ARC у переменных, как и у свойств, есть атрибут владения, только в этом случае набор значений меньше, чем у свойств: __strong/__weak/__unsafe_unretained, и касается это только тех типов

данных, которые подпадают под действие ARC. Поэтому при описании значений этого атрибута для свойств будем еще указывать, какое значение атрибута владения должно быть у соответствующей переменной экземпляра при включенном ARC (если переменная создается автоматически — она сразу создается с нужным значением этого атрибута. Если же вы определяете переменную сами — нужно вручную задать ей правильное значение атрибута владения).

- **retain** (соответствующая переменная должна быть с атрибутом __strong) — это значение показывает, что в сгенерированном сеттере счетчик ссылок на присваиваемый объект будет увеличен, а у объекта, на который свойство ссылалось до этого, — счетчик ссылок будет уменьшен. Это значение применимо при выключенном ARC для всех Objective-C классов во всех случаях, когда никакие другие значения не подходят. Это значение сохранилось со времен, когда ARC еще не было и, хотя ARC и не запрещает его использование, при включенном автоматическом подсчете ссылок лучше вместо него использовать strong.

```
//примерно вот такой сеттер будет сгенерирован для свойства
//с таким значением атрибута владения с отключенным ARC
-(void)setFoo(Foo *)foo {
  if (_foo ≠ foo) {
    Foo *oldValue = _foo;

    //увеличивается счетчик ссылок на новый объект и укзатель на него сохраняется в ivar
    _foo = [foo retain];

    //уменьшается счетчик ссылок на объект, на который раньше указывало свойство
    [oldValue release];
  }
}

```

- **strong** (соответствующая переменная должна быть с атрибутом __strong) — это значение аналогично retain, но применяется только при включенном автоматическом подсчете ссылок. При использовании ARC это значение используется по умолчанию. Используйте strong во всех случаях, не подходящих для weak и copy, и все будет хорошо.
- **copy** (соответствующая переменная должна быть с атрибутом __strong) — при таком значении атрибута владения в сгенерированном сеттере соответствующей переменной экземпляра присваивается значение, возвращаемое сообщением copy, отправленным присваиваемому объекту.

Использование этого значения атрибута владения накладывает некоторые ограничения на класс объекта:

1. Класс должен поддерживать протокол NSCopying,
2. Класс не должен быть изменяемым (mutable). У некоторых классов есть mutable-подкласс, например, NSString-NSMutableString. Если ваше свойство — экземпляр «мутабельного» класса, использование copy приведет к

нежелательным последствиям, так как метод copy вернет экземпляр его «немутабельного» сородича. Например, вызов copy у экземпляра NSMutableString вернет экземпляр NSString.

```
@property (copy, nonatomic) NSMutableString *foo;
    ...
    //в сгенерированном сеттере будет примерно следующее
- (void)setFoo:(NSMutableString)foo {
    _foo = [foo copy]; //метод copy класса NSMutableString вернет объект типа NSString, так что после присвоения значения этому свойству, оно будет указывать на неизменяемую строку, и вызов методов, изменяющих строку, у этого объекта приведет к крэшу
}
```

- **weak** (соответствующая переменная должна быть с атрибутом __weak) — это значение аналогично assign и unsafe_unretained. Разница в том, что особая уличная магия позволяет переменным с таким значением атрибута владения менять свое значение на nil, когда объект, на который указывала переменная, уничтожается, что очень хорошо сказывается на устойчивости работы приложения (ибо, как известно, nil отвечает на любые сообщения, а значит никаких вам EXC_BAD_ACCESS при обращении к уже удаленному объекту). Это значение атрибута владения стоит использовать при включенном ARC для исключения retain cycle'ов для свойств, в которых хранится указатель на делегат объекта и им подобных. Это единственное значение атрибута владения, которое не поддерживается при выключенном ARC (как и при включенном ARC на iOS до версии 5.0).
- **unsafe_unretained** (соответствующая переменная должна быть с атрибутом __unsafe_unretained) — свойство с таким типом владения просто сохраняет адрес присвоенного ему объекта. Методы доступа к такому свойству никак не влияют на счетчик ссылок объекта. Он может удалиться, и тогда обращение к такому свойству приведет к крэшу (потому и unsafe). Это значение использовалось вместо weak, когда уже появился ARC, но нужно было еще поддерживать iOS 4.3. Сейчас его использование можно оправдать разве что скоростью работы (есть сведения, что магия weak свойств требует значительного времени, хотя, конечно, невооруженным глазом при нынешней производительности этого не заметишь), поэтому, особенно на первых порах, использовать его не стоит.
- **assign** (соответствующая переменная должна быть с атрибутом __unsafe_unretained, но так как атрибуты владения есть только у типов попадающих под ARC, с которыми лучше использовать strong или weak, это значение вам вряд ли понадобится) — просто присвоение адреса. Без ARC является дефолтным значением атрибута владения. Его стоит применять к свойствам типов, не попадающих под действие ARC (к ним относятся примитивные типы и так называемые необъектные типы (non-object types) вроде CFTypeRef). Без ARC он также используется вместо weak для

исключения retain cycle’ов для свойств, в которых хранится указатель на делегат объекта и им подобных.

Атрибут атомарности

- **atomic** — это дефолтное значение для данного атрибута. Оно означает, что акцессор и мутатор будут сгенерированы таким образом, что при обращении к ним одновременно из разных потоков, они не будут выполняться одновременно (то есть все равно сперва один поток сделает свое дело — задаст или получит значение, и только после этого другой поток сможет заняться тем же). Из-за особенностей реализации у свойств с таким значением атрибута атомарности нельзя переопределять только один из методов доступа (уж если переопределяете, то переопределяйте оба, и сами заморочьтесь с защитой от одновременного выполнения в разных потоках). Не стоит путать атомарность свойства с потокобезопасностью объекта. К примеру, если вы в одном потоке получаете значение имени и фамилии из свойств объекта, а в другом — изменяете эти же значения, вполне может получиться так, что значение имени вы получите старое, а фамилии — уже измененное. Соответственно, применять стоит для свойств объектов, доступ к которым может осуществляться из многих потоков одновременно, и нужно спастись от получения каких-нибудь невалидных значений, но при необходимости обеспечить полноценную потокобезопасность, одного этого может оказаться недостаточно. Кроме прочего стоит помнить, что методы доступа таких свойств работают медленнее, чем nonatomic, что, конечно, мелочи в масштабах вселенной, но все же копейка рубль бережет, поэтому там, где нет необходимости, лучше использовать nonatomic.
- **nonatomic** — значение противоположное atomic — у свойств с таким значением атрибута атомарности методы доступа не обременены защитой от одновременного выполнения в разных потоках, поэтому выполняются быстрее. Это значение пригодно для большинства свойств, так как большинство объектов все-таки используются только в одном потоке, и нет смысла «обвешивать» их лишними фичами. В общем, для всех свойств, для которых не сможете объяснить, почему оно должно быть atomic, используйте nonatomic, и будет вам fast and easy and smart и просто праздник какой-то.

Nullability атрибут

Этот атрибут никак не влияет на генерируемые методы доступа. Он предназначен для того, чтобы обозначить, может ли данное свойство принимать значение nil или NULL. Xcode использует эту информацию при взаимодействии Swift-кода с вашим классом. Кроме того, это значение используется для вывода предупреждений в случае, если ваш код делает что-то, противоречащее заявленному поведению.

Например, если вы пытаетесь задать значение nil свойству, которое объявлено как nonnull. А самое главное, эта информация будет полезна тому, кто будет читать ваш код. На использование этого атрибута есть пара ограничений:

1. Его нельзя использовать для примитивных типов (им и не нужно, так как они в любом случае не принимают значений nil и NULL)
2. его нельзя использовать для многоуровневых указателей (например, id* или NSError**)

- **null_unspecified** — используется по умолчанию и ничего не говорит о том, может ли свойство принимать значение nil/NULL или нет. До появления этого атрибута именно так мы и воспринимали абсолютно все свойства. Это плохо в плане содержательности заголовков, поэтому использование этого значения не рекомендуется.
- **null_resettable** — это значение свидетельствует о том, что геттер такого свойства никогда не вернет nil/NULL в связи с тем, что при задании такого значения, на самом деле свойству будет присвоено некое дефолтное. А так как генерируемые методы доступа от значения этого атрибута не зависят, вы сами должны будете либо переопределить сеттер так, чтобы при получении на вход nil/NULL он сохранял в ivar значение по умолчанию, либо переопределить геттер так, чтобы он возвращал дефолтное значение в случае, если соответствующая ivar == nil/NULL. Соответственно, если у вашего свойства есть дефолтное значение — объявляйте его как null_resettable.
- **nonnull** — это значение свидетельствует о том, что свойство, помеченное таким атрибутом, не будет принимать значение nil/NULL. На самом деле, вы все еще можете получить nil, если, к примеру, попробуете получить значение этого свойства у nil'а и просто потому, что Xcode не очень жестко следит за этим. Но это уже будет ваша ошибка и Xcode будет по мере сил указывать вам на нее в своих предупреждениях. Использовать стоит, если вы уверены, что значение данного свойства никогда не будет nil/NULL и хотите, чтобы IDE помогала вам следить за этим.
- **nullable** — это значение свидетельствует о том, что свойство может иметь значение nil/NULL. Оно так же, как и null_unspecified, ни к чему не обязывает, но все же ввиду его бОльшей определенности, среди этих двух правильнее использовать именно nullable. Таким образом, если вам не подходит ни null_resettable, ни nonnull — используйте nullable.

Для большего удобства вы можете изменить значение по умолчанию с null_unspecified на nonnull для определенного блока кода. Для этого нужно

поставить NS_ASSUME_NONNULL_BEGIN перед таким блоком и NS_ASSUME_NONNULL_END — после него.

```
//например вот такое объявление
@property (copy, nonatomic, nonnull) NSString *foo;
@property (copy, nonatomic, nonnull) NSString *str;
@property (strong, nonatomic, nullable) NSNumber *bar;
@property (copy, nonatomic, null_resettable) NSString *baz;

//аналогично следующему блоку
NS_ASSUME_NONNULL_BEGIN
@property (copy, nonatomic) NSString *foo;
@property (copy, nonatomic) NSString *str;
@property (strong, nonatomic, nullable) NSNumber *bar;
@property (copy, nonatomic, null_resettable) NSString *baz;
NS_ASSUME_NONNULL_END
```

Инициализация

Инициализация – это создание объекта.

Пример:

```
Object *object = [[Object alloc] init];
```

Изначально вызывается метод **alloc**, который отвечает за выделение памяти для объекта, а после – конструктор класса (метод инициализации). Для множества компонентов в Objective-C уже существуют конструкторы, но для своих классов можно создавать собственные. Метод инициализации в самом классе выглядит так:

```
- (instancetype)init
{
    self = [super init];
    if (self) {
        // Необходимая логика
    }
    return self;
}

```

Чтобы создать собственный конструктор, добавим необходимые параметры, создадим нужную нам логику и объявим этот инициализатор в интерфейсе класса:

```
- (instancetype)initWithName:(NSString *)name
{
    self = [super init];
    if (self) {
        self.name = name;
    }
    return self;
}

```

Обновим интерфейс класса:

```objc
@interface Object: NSObject

- (instancetype)initWithName:(NSString *)name;

@property (nonatomic, strong) NSString *name;

@end

```

Использование собственного конструктора:

```objc
 Object *object = [[Object alloc] initWithName:@"Name"];
```

При вызове созданного конструктора будет инициализирован объект и установлено значение для свойства **name**.

После создания (инициализации) объекта можно хранить его, обращаться к свойствам, использовать необходимые методы. Но существует понятие жизненного цикла объекта – это время от его создания до удаления из памяти. Вывод: после применения объект необходимо уничтожить.

Деинициализация

Деинициализация – это уничтожение объекта и освобождение памяти, которая использовалась при его жизненном цикле. Чтобы уничтожить объект, достаточно установить ему значение **nil**. Будет вызван метод класса **dealloc**, обращение к которому свидетельствует о деинициализации объекта.

Пример:

```objc
Object *object = [[Object alloc] initWithName:@"Name"];
NSLog(@"Name - %@", object.name);
object = nil;
NSLog(@"Name - %@", object.name);

```

Метод **dealloc** у класса **Object**:

```objc
- (void)dealloc {
    NSLog(@"Dealloc object");
}

```

В результате выполнения этой программы будет выведено первоначальное имя объекта, установленное при инициализации. Затем ему будет присвоено значение **nil**, и он будет уничтожен. Для подтверждения удаления в консоль выводится значение имени: оно уже соответствует значению **null**.

Принципы объектно-ориентированного программирования

ООП

Объектно-ориентированное программирование – это принцип, при котором основными компонентами программирования являются объекты. В той или иной степени он присутствует во всех языках программирования. Этот подход имеет 4 основных принципа: абстракция, инкапсуляция, наследование и полиморфизм.

На мой взгляд, если функциональное или структурное программирование — принципы, которые больше относятся к написанию именно кода, то ООП это уже не про код, а моделирование сложных систем. Наш мозг воспринимает мир как набор объектов, которые взаимодействуют друг с другом. Если функция — просто действие без контекста, то метод класса это уже действие в определенном контексте, это действие, которое относится к определенному объекту. И система приобретает вид взаимодействия различных объектов между собой. Таким образом, ООП позволяет сделать описание системы более понятным для восприятия.

🔥 Также заранее хочу добавить, это описание идеального сферического программирования в вакууме и в реальности множество вещей нарушаются в угоду практичности. Но стремление к идеалу только улучшит качество кода. И не стоит забывать, что гонка за крайностью – тоже плохо.

Абстракция

Абстракция – это подход, при котором вместо непосредственного использования значения в коде применяется указатель на это значение. Любая переменная в программировании является абстракцией, так как она скрывает за собой значение. Абстракция позволяет работать с объектом, не задумываясь о его содержимом.

На этом подходе построено множество программ: благодаря понятным определениям он позволяет разработчику однозначно устанавливать смысл переменной.

*Например, человеку проще понять, что в переменной с названием **width** хранится ширина, чем если бы в ней просто стояло значение 113.0.*

Наследование

Наследование – это еще один из принципов объектно-ориентированного программирования, при котором класс-наследник перенимает от класса-родителя некоторые свойства, методы и поведение.
Рассмотрим пример. Создадим интерфейс для класса-родителя:

```
@interface Parent : NSObject

@property (nonatomic, strong) NSString *value;

- (void)print;

@end

```

После чего реализуем объявленный метод:

```
@implementation Parent

- (void)print {
    NSLog(@"Current value = %@", self.value);
}

@end

```

При вызове метода **print** в консоль будет выводиться сообщение с текущим значением свойства **value**.
После необходимо объявить и наследовать класс наследника:

```
#import "Parent.h"

@interface Child : Parent

@end

```

Для этого добавим файл объявления, используя **#import**. Затем при объявлении класса после названия через двоеточие укажем класс родителя.

```
Child *child1 = [[Child alloc] init];
child1.value = @"CHILD - 1";
[child1 print];

Child *child2 = [[Child alloc] init];
child2.value = @"CHILD - 2";
[child2 print];

```

Теперь объекты класса **Child** будут наследниками класса **Parent**. Рассмотрим реализацию:

В результате выполнения программы будет выведено текущее значение для первого и второго наследника. Так наследники использовали метод и свойство родительского класса.
Родительские методы можно переопределять: изменять необходимую логику. Для этого вызывается метод родительского класса в реализации и создается новая логика:

```
@implementation Child

- (void)print {
    NSLog(@"Child print");
}

@end

```

Теперь при обращении к методу **print** у наследника будет выводиться в консоль значение «Child print».
Наследование помогает однозначно разделять классы и передавать другим классам свои возможности. Программы становятся более понятными и удобными для дальнейшей поддержки.

Инкапсуляция

Инкапсуляция – это еще один принцип объектно-ориентированного программирования, который основан на сокрытии или открытии определенных методов и свойств. Инкапсуляция помогает однозначно определить, какие методы и свойства необходимо скрыть от использования извне, а какие можно применять.

Реализация инкапсуляции в Objective-C выглядит достаточно просто. Чтобы предотвратить использование метода класса извне, можно просто не объявлять его в файле объявления. Так метод будет виден внутри собственной реализации, но скрыт от других классов.

В файле реализации можно расширить объявленные методы и свойства. Отсюда – второй способ реализации инкапсуляции: объявление методов и свойств прямо в в файле реализации. Для этого в файле реализации используется такая конструкция:

```objc
@interface Object ()

- (void)privateMethod;

@property (nonatomic, strong) NSString *privateName;

@end

@implementation Object

- (void)privateMethod {
    NSLog(@"Name - %@", self.privateName);
}

@end

```

В другие классы импортируется только файл объявления, так что методы и свойства, объявленные в файле реализации, будут недоступны из других классов. Данный принцип позволит более правильно организовать работу с классом, у которого есть важные методы и свойства, которые требуется защитить от изменений извне. Так гарантируется, что класс сможет выполнять необходимую логику.

Полиморфизм

Полиморфизм – это принцип объектно-ориентированного программирования, при котором множество однообразных классов перенимают логику и основываются на одном классе-родителе.

Рассмотрим пример, где нам необходимо реализовать новостное приложение, которое сможет работать сразу с несколькими видами публикаций: новостями, объявлениями, статьями. Для каждого из перечисленных видов необходимо создать класс, но все они содержат определенную единую логику: название, текст, дату публикации. Прибегаем к помощи полиморфизма. Реализуем класс публикации, который будет содержать единые методы и свойства для всех, а после – конкретные виды публикаций на основе этого родительского класса.

Интерфейс класса публикации, который содержит все необходимые методы и свойства:

```
enum PublicationType {
    PublicationTypeNews,
    PublicationTypeAnnouncement,
    PublicationTypeArticle
};

typedef enum PublicationType PublicationType;

@interface Publication : NSObject

@property (nonatomic, strong) NSString *title;
@property (nonatomic, strong) NSString *text;
@property (nonatomic, strong) NSDate *date;
@property (nonatomic) PublicationType type;

- (void)print;

@end

```

Реализация класса публикации:

```
@implementation Publication

- (void)print {
    NSLog(@"Title = %@ \nText = %@ \nDate = %@ \nType = %@", self.title, self.text, self.date, [self typeNameFrom:self.type]);
}

- (NSString *)typeNameFrom:(PublicationType)type {
    switch (type) {
        case PublicationTypeNews:
            return @"Новость";
            break;
        case PublicationTypeAnnouncement:
            return @"Объявление";
            break;
        case PublicationTypeArticle:
            return @"Статья";
            break;
    }
}

@end

```

Объявление конкретных публикаций (новость, статья, объявление):

```
@interface News: Publication
```

Создадим в файле **main** две функции – для печати и для создания публикаций всех видов:

```objc
void printPublication(Publication *publication) {
    [publication print];
}

void createPublications() {

    News *news = [[News alloc] init];
    news.title = @"Title news";
    news.text = @"Text news";
    news.date = [NSDate date];
    news.type = PublicationTypeNews;
    printPublication(news);

    Announcement *announcement = [[Announcement alloc] init];
    announcement.title = @"Title announcement";
    announcement.text = @"Text announcement";
    announcement.date = [NSDate date];
    announcement.type = PublicationTypeAnnouncement;
    printPublication(announcement);

    Article *article = [[Article alloc] init];
    article.title = @"Title article";
    article.text = @"Text article";
    article.date = [NSDate date];
    article.type = PublicationTypeArticle;
    printPublication(article);

}

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        createPublications();
    }
    return 0;
}

```

Так как каждый вид публикаций является наследником класса **Publication**, то их можно передавать в качестве параметра в функцию **printPublication()**. Все конкретные классы создаются в функции **createPublications** и отправляются на печать.

В итоге вся повторяющаяся логика для видов публикаций содержится в их родительском классе. Так полиморфизм позволяет избежать множественного копирования одинаковых свойств и методов, помогает создавать более абстрактный и читабельный код.

Категории

С помощью категорий в Objective-C можно добавить функциональность любому классу, в том числе и стандартному. Это сравнимо с расширениями в других языках программирования(Swift).

При долгом поддержании проектов классы могут превращаться в огромный файл

реализации с множеством методов. Решить эту проблему могут категории, которые позволяют выносить часть функционала из основного класса.

Реализуются они также просто, как и класс:

```
@interface NSNumber (toString)

- (NSString *)toString;

@end

@implementation NSNumber (toString)

- (NSString *)toString {
    return [NSString stringWithFormat:@"%@", self];
}

@end

```

Но есть и различия. Вместо имени интерфейса и реализации вводится имя класса, функционал которого необходимо расширить, а в скобках указывается произвольное имя. Разумеется, в объявлении указываются методы, которые планируется добавить и реализовать. Для доступа к текущему значению применяется ключевое слово **self**.

Чтобы использовать созданный метод расширения, необходимо сделать следующее:

```
NSNumber *number = @1;

NSString *string = [number toString];

NSLog(@"Result %@", string);

```

В результате число будет преобразовано в строку.
При объявлении категории в скобках указывается ее имя. Что будет, если не указать имя? Или указать имя для существующей категории?

Во время выполнения исполнительная среда определяет методы, которые были реализованы категорией, и добавляет их к основному классу. Если методы со схожими названиями уже были добавлены, то может произойти коллизия имен. При реализации категории без имени программа определит ее и внесет ее методы вместо предыдущей реализации. Поэтому следует внимательно относиться к именованию категорий.

Если не указать имя категории, то она будет продолжением класса. Это позволяет добавлять интерфейс в файл реализации. В нем можно также объявлять свойства и методы для применения в классе, но они не будут доступны из других классов.
Кроме методов, в категории можно добавлять и свойства. При этом надо иметь в виду, что категории «не умеют» синтезировать методы доступа (геттер и сеттер), так

что необходимо реализовать их самостоятельно.

Дополнительные материалы

1. https://habrahabr.ru/post/147927/;
2. https://habr.com/ru/post/479640/;
3. https://habr.com/ru/post/87205/
4. Стивен Кочан. «Программирование на Objective-C»;
5. Скотт Кнастер, Вакар Малик, Марк Далримпл.«Objective-C. Программирование для Mac OS X и iOS».