Урок 4

Объекты. Наследование, инкапсуляция и полиморфизм в Objective-C

Обзор объектов. Принципы объектно-ориентированного программирования на Objective-C.

Терминология

Перед погружением в объекты и объектно-ориентированное программирование повторим основные термины этой области:

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 – 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.

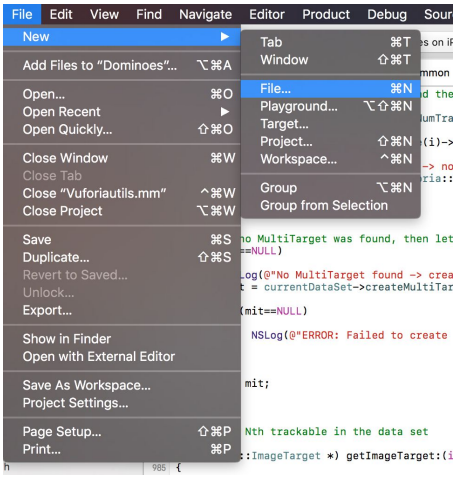
Объекты

Объект – это экземпляр класса, который содержит все его значения и поведение. В одном проекте может быть множество объектов, так как с ними приходится работать постоянно. Имена переменных, которые содержат объекты, пишутся с прописной буквы.

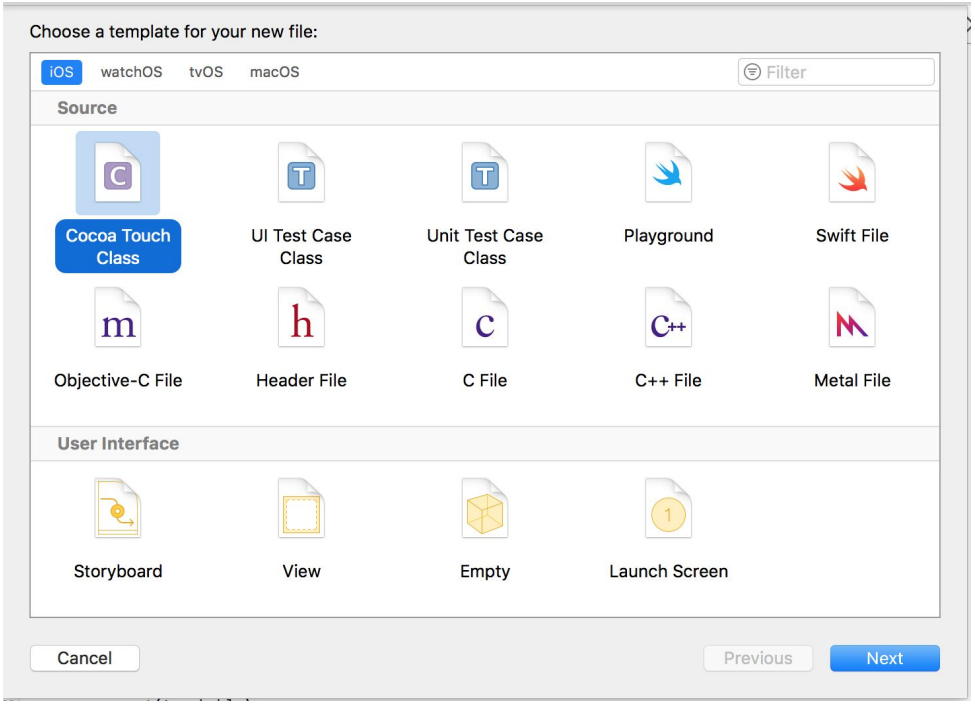
Так как объект создается на основе класса, то сначала научимся создавать класс.

Создание классов

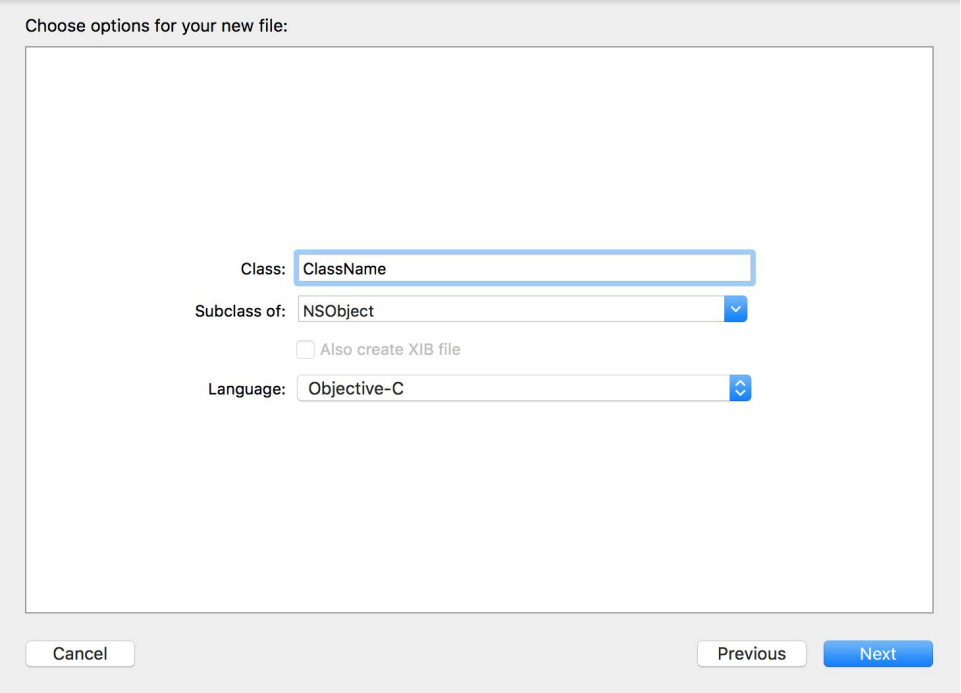
Чтобы создать класс в Xcode, необходимо перейти на вкладку **File -> New -> File**:



Затем необходимо выбрать один из видов файлов. Для создания класса подойдет файл **«Cocoa Touch Class»**. Нажимаем на кнопку **Next**:



Далее вводим название файла и класса, указываем родительский класс (по умолчанию – **NSObject**) и выбираем язык **Objective-C**. Нажимаем **Next.** Выбираем место, где будет храниться файл, и нажимаем **Create**:



Класс создан: можно переходить к реализации необходимого функционала.

Свойства

Свойство – это параметр, который описывает состояние объекта. С помощью свойств объект может хранить определенные состояния или значения, необходимые для его функционирования.

Рассмотрим, как объявляются переменные класса:

```
@interface Student : NSObject {
@public
    NSString *name;
    NSString *surname;
@private
    NSDate *birthDate;
}

@end
```

Это решение 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.*

Свойства объявляются в интерфейсе класса с помощью такой конструкции:

```
@interface Object: NSObject

    @property (nonatomic, strong) NSString *name;

@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.

Методы

Методы – это действия объекта, которые можно реализовать только в его теле. Структура метода выглядит так:

```
- (/* Возвращаемый тип */) /* Название метода */: (/* Тип первого параметра
*/)/* Имя первого параметра */ /* Название второго параметра в функции */:(/*
Тип второго параметра */) /* Имя второго параметра */ {

    return /* Возвращаемое значение */  // Если необходимо
}
```

Пример реализации метода:

```
-(void)print:(NSString *)message {
    NSLog(@"%@", message);
}
```

Метод ничего не возвращает, а на вход принимает сообщение в формате **NSString** (строки) и выводит его в консоль.

Чтобы вызвать созданный метод, в квадратных скобках указываем объект, которому принадлежит этот метод. Если этот метод относится к текущему объекту, то указывается ключевое слово **self**. Далее через пробел пишем название метода и значение параметров, если они необходимы.

Выглядит это так:

```
[self print:@"Message"];      // Метод вызывается в самом объекте

[object print:@"Message"];    // Метод вызван из другого объекта
```

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:

```
@interface Object: NSObject

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

@property (nonatomic, strong) NSString *name;

@end
```

Использование собственного конструктора:

```
 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:

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

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

```
- (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.

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.

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

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

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

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:

```
@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:

```
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.

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 уже 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.

Практическое задание

1. Создать программу, которая будет выводить список студентов. Для этого необходимо создать класс **Студент**, а значения свойств устанавливать, используя собственный конструктор.
2. У студента должно быть свойство age (возраст), оно должно быть только для чтения
3. У студента должны быть свойства name, surname и fullName. Первые два должны быть обычными свойствами, а fullName должно возвращать строку состоящую из склеенных name и surname.
4. Переопределите метод description чтобы при выводе объекта в NSLog выводилась информация по всем его свойствам.
5. Добавьте метод который будет увеличивать его возраст. (Да свойство age только для чтения).

Дополнительные материалы

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

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Стивен Кочан. «Программирование на Objective-C»;
2. Скотт Кнастер, Вакар Малик, Марк Далримпл. «Objective-C. Программирование для Mac OS X и iOS».