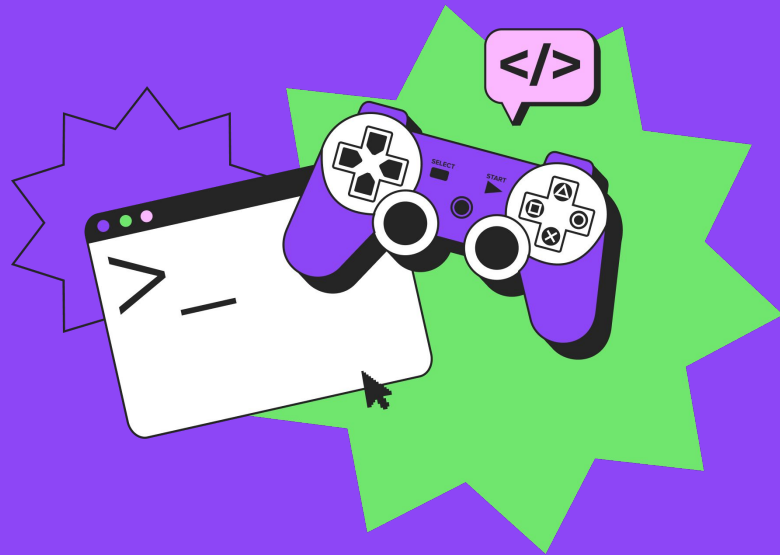


Объекты. Наследование, инкапсуляция и полиморфизм в Objective – C

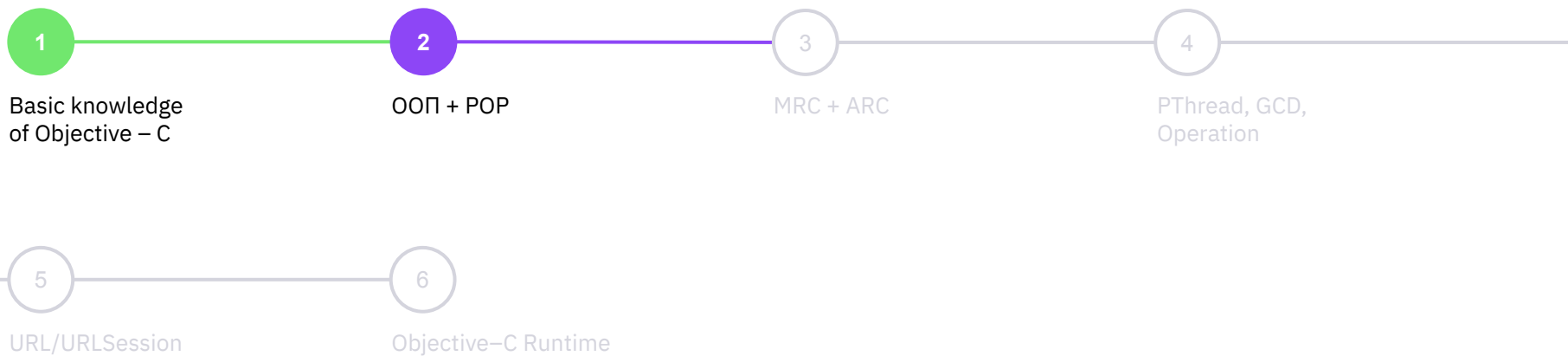
Урок 2

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













План курса





Что будет на уроке сегодня

-  Терминология
-  Введения в классы
-  Создание и использование классов
-  Методы
-  Свойства
-  Инициализатор
-  Деинициализатор
-  ООП



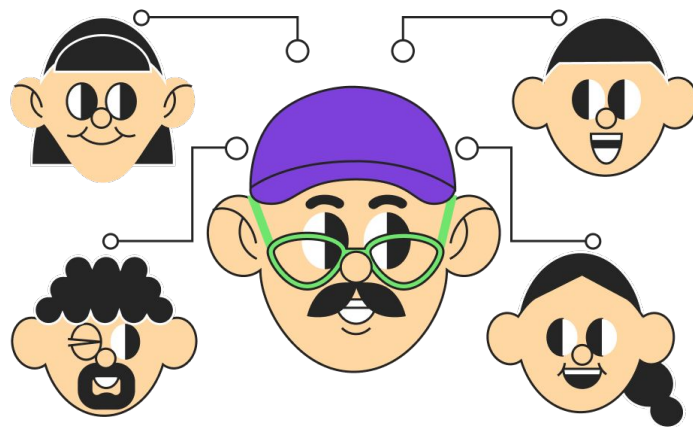


Терминалогия



Класс

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.

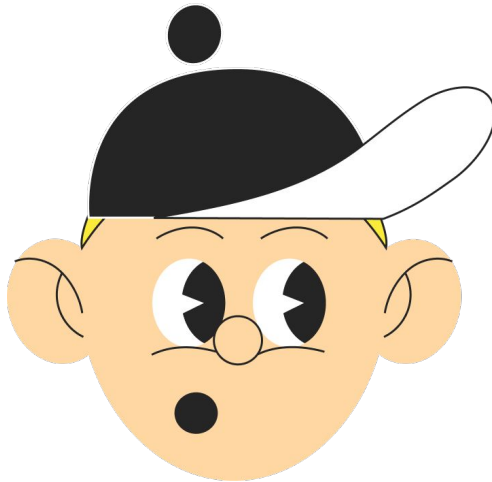




Сообщение

Сообщение – это действие, которое способен выполнить объект. В коде его реализация выглядит так:

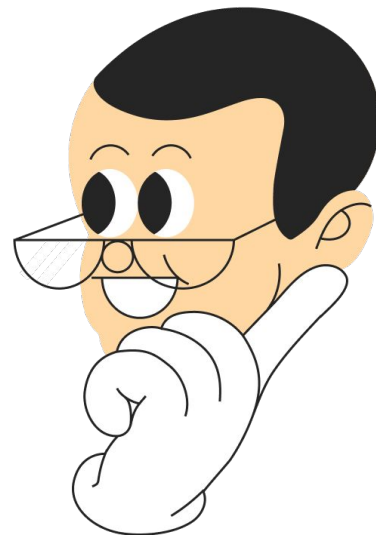
```
[project method];
```





Метод

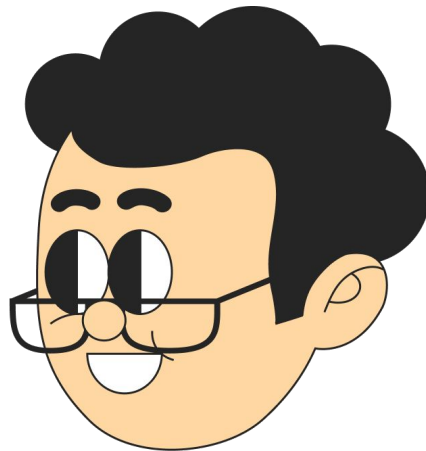
Метод – это код, который выполняется в ответ на сообщение.





Интерфейс

Интерфейс – это описание возможностей класса (@interface).



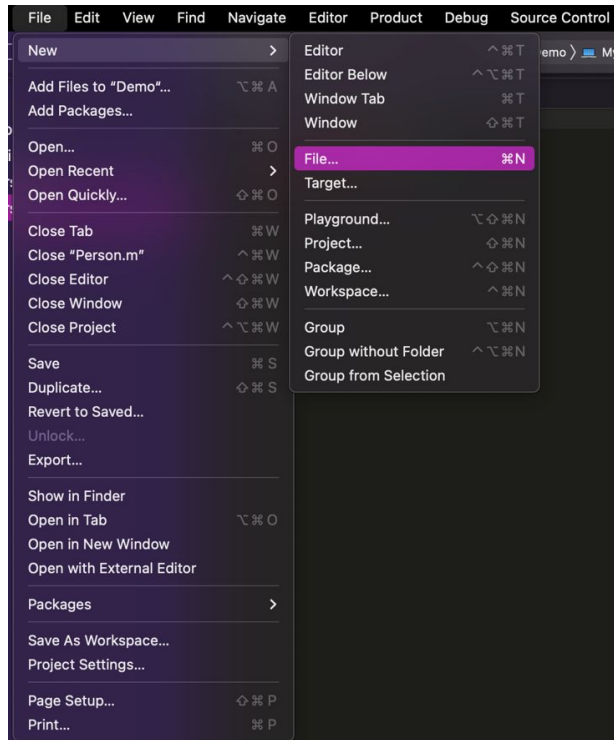


Введение в классы



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».





```
@interface Human: NSObject Class name
{
    int age;
    id data;
    NSString *_name;
}

- (void)sayHello;
+ (Human *)createHumanWithName:(NSString *)name;

@end
```



```
@interface Human: NSObject
{
    int age;
    id data;
    NSString *_name;
}

- (void)sayHello;
+ (Human *)createHumanWithName:(NSString *)name;

@end
```

Superclass name



```
@interface Human: NSObject
{
    int age;
    id data;
    NSString *_name;
}

- (void)sayHello;
+ (Human *)createHumanWithName:(NSString *)name;

@end
```

Instance variables



```
@interface Human: NSObject
{
    int age;
    id data;
    NSString *_name;
}
```

```
- (void) sayHello;
+ (Human *) createHumanWithName : (NSString *) name;
```

```
@end
```

Methods



```
@implementation Human
```

```
- (void) sayHello
```

```
{
```

```
    puts("Hello!");
```

```
}
```

```
+ (Human *) createHumanWithName:(NSString *) name
```

```
{
```

```
    Human *human = [[Human alloc] init];
```

```
    human.name = name;
```

```
    return human;
```

```
}
```

```
@end
```

Class
implementation



Методы



Методы

Методы — это функции, связанные с конкретным типом. Классы, структуры и перечисления могут определять методы экземпляра, которые инкапсулируют конкретные задачи и функциональные возможности для работы с экземпляром данного типа.





```
@interface SomeClass : NSObject
```

```
- (void)foo; Instance method
```

```
+ (int)foo:(int)bar;
```

```
- (NSInteger)foo:(int)bar1 oof:(int)bar2;
```

```
@end
```



```
@interface SomeClass : NSObject

- (void)foo;
+ (int)foo:(int)bar;
- (NSInteger)foo:(int)bar1 oof:(int)bar2;

@end
```

Class method



```
@interface SomeClass : NSObject
```

```
- (void)foo;
+ (int)foo:(int)bar;
- (NSInteger)foo:(int)bar1 oof:(int)bar2;
```

```
@end
```

Returning type



```
@interface SomeClass : NSObject
```

```
- (void)foo;
+ (int)foo:(int)bar;
- (NSInteger)foo:(int)bar1 oof:(int)bar2;
```

```
@end
```

Method name



```
@interface SomeClass : NSObject  
  
- (void)foo;  
+ (int)foo:(int)bar; Parameter type  
  
- (NSInteger)foo:(int)bar1 oof:(int)bar2;  
  
@end
```



```
@interface SomeClass : NSObject

- (void)foo;
+ (int)foo:(int)bar;
- (NSInteger)foo:(int)bar1 oof:(int)bar2;

@end
```

Parameter name



```
@interface SomeClass : NSObject
```

```
- (void)foo;
```

```
+ (int)foo:(int)bar;
```

```
- (NSInteger)foo:(int)bar1
```

```
oof
```

```
:(int)bar2;
```

Method name
continuation

```
@end
```




```
@interface SomeClass : NSObject
```

```
- (void)foo;
```

```
+ (int)foo:(int)bar;
```

```
- (NSInteger)foo:(int)bar1 oof:(int)bar2;
```

Another
parameter type

```
@end
```



```
@interface SomeClass : NSObject
```

```
- (void)foo;
```

```
+ (int)foo:(int)bar;
```

```
- (NSInteger)foo:(int)bar1 oof:(int)bar2;
```

Another
parameter
name

```
@end
```



```
@implementation SomeClass
```

```
- (void)foo {  
}
```

```
+ (int)foo:(int)bar {  
    return 123;  
}
```

```
- (NSInteger)foo:(int)bar1 oof:(int)bar2 {  
    return 123;  
}
```

```
@end
```

Methods
implementation



Свойства



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.





Instance variables (Переменные экземпляра)

```
1 @interface Person : NSObject {  
2     @public  
3     NSString *name;  
4 }  
5 - (void)printGreeting;  
6 @end
```

```
1 - (void)printGreeting {  
2     NSLog(@"Hello! %@", name);  
3 }
```



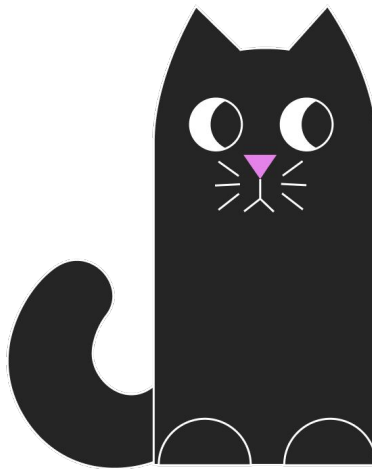
Instance variables (Переменные экземпляра)

```
1 #import <Foundation/Foundation.h>
2 #import "Person.h"
3
4 int main(int argc, const char * argv[]) {
5     @autoreleasepool {
6         Person *person = [Person new];
7         person->name = @"Str";
8         [person printGreeting];
9     }
10    return 0;
11 }
12
13
```



@Property

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.





@Property

```
1 #import <Foundation/Foundation.h>
2
3 NS_ASSUME_NONNULL_BEGIN
4
5 @interface Person : NSObject
6
7 @property (nonatomic, strong) NSString *name;
8
9 - (void)printGreeting;
10
11 @end
12
13 NS_ASSUME_NONNULL_END
14
```

```
1 @implementation Object
2 @synthesize name = objectName;
3
4 @end
5
```



@Property

```
1 @implementation Object
2 @synthesize name = objectName;
3
4 - (instancetype)init
5 {
6     self = [super init];
7     if (self) {
8         objectName = @"Name";    // Присвоение значения переменной свойства
9         self.name = @"Name";    // Присвоение значения свойству
10        [self setName:@"Name"];  // Аналогичное присвоение значения свойству
11    }
12    return self;
13 }
14
15 @end
16
```

```
1 // Обращение к члену класса
2 _name = @"Name";
3 // Обращение к свойству класса
4 self.name = @"Name";
5
```



Setter

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

```
1 - (void)setName:(NSString *)name {  
2     _name = name;  
3 }  
4
```

```
1 self.prop = @"Value";           // Вызов из класса  
2 [self setProp:@"Value"];        // Аналогичный вызов из класса  
3  
4 object.prop = @"Value";         // Вызов извне  
5 [object setProp:@"Value"];      // Аналогичный вызов извне  
6
```



Getter

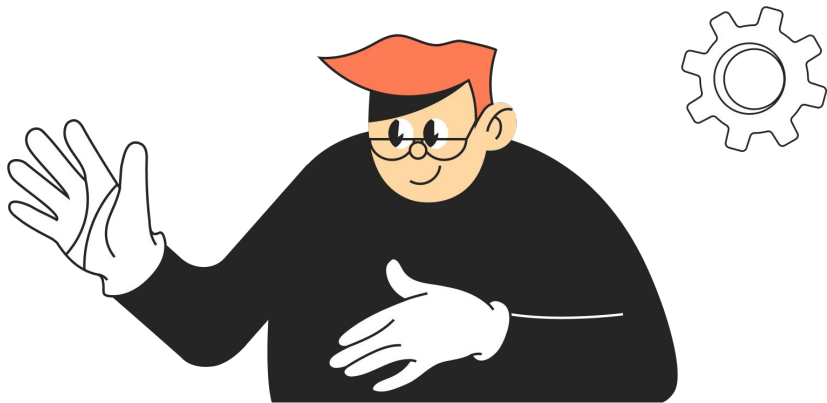
Геттер (Getter) – это метод для получения значения свойства класса. Выглядит он следующим образом:

```
1 - (NSString *)name {  
2     return @"Name";  
3 }  
4
```



Атрибуты свойств

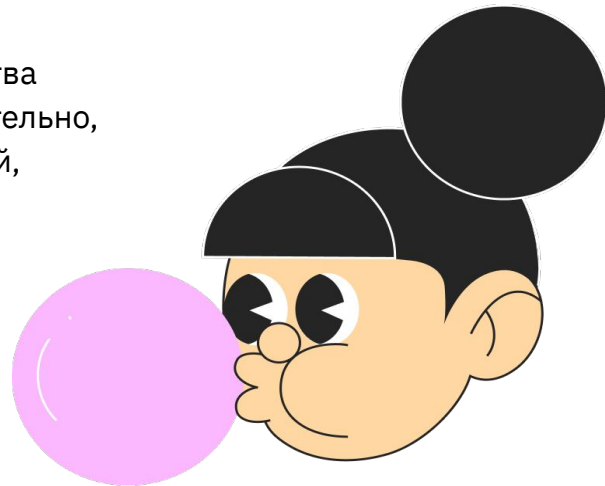
1. Атрибуты доступности (readonly/readwrite),
2. Атрибуты владения (retain/strong/copy/assign/unsafe_unretained/weak),
3. Атрибут атомарности (atomic/nonatomic).
4. Nullability атрибут (null_unspecified/null_resettable/nullable/nonnull) — появился в xcode 6.3





Атрибуты 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.





Атрибуты владения

1. **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
2. **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 сору, и все будет хорошо.
3. **copy** (соответствующая переменная должна быть с атрибутом **__strong**) — при таком значении атрибута владения в сгенерированном сеттере соответствующей переменной экземпляра присваивается значение, возвращаемое сообщением сору, отправленным присваиваемому объекту.

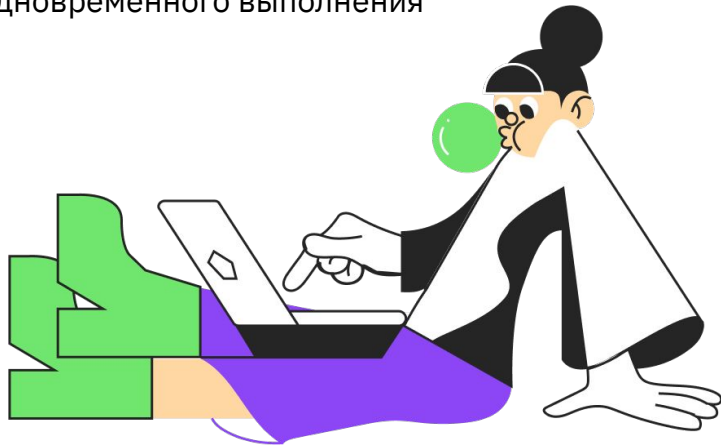


Атрибуты владения

4. **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.
5. **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 крашу (потому и `unsafe`).
6. **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.

Атрибуты 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).
- **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.





Атрибуты `nullability`



Этот атрибут никак не влияет на генерируемые методы доступа. Он предназначен для того, чтобы обозначить, может ли данное свойство принимать значение `nil` или `NULL`.

1. **`null_unspecified`** — используется по умолчанию и ничего не говорит о том, может ли свойство принимать значение `nil/NULL` или нет.
2. **`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.
3. **`nonnull`** — это значение свидетельствует о том, что свойство, помеченное таким атрибутом, не будет принимать значение `nil/NULL`.
4. **`nullable`** — это значение свидетельствует о том, что свойство может иметь значение `nil/NULL`.



Инициализаторы



Инициализаторы

Инициализация – это создание объекта.

```
1 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:

```
1 - (instancetype)init
2 {
3     self = [super init];
4     if (self) {
5         // Необходимая логика
6     }
7     return self;
8 }
9
```



Инициализаторы

```
1 - (instancetype)initWithName:(NSString *)name
2 {
3     self = [super init];
4     if (self) {
5         self.name = name;
6     }
7     return self;
8 }
9
```

```
1 @interface Object: NSObject
2
3 - (instancetype)initWithName:(NSString *)name;
4
5 @property (nonatomic, strong) NSString *name;
6
7 @end
8
```



Деинициализаторы



Деинициализаторы

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.

```
1 Object *object = [[Object alloc] initWithName:@"Name"];
2 NSLog(@"Name - %@", object.name);
3 object = nil;
4 NSLog(@"Name - %@", object.name);
5
```

```
1 - (void)dealloc {
2     NSLog(@"Dealloc object");
3 }
4
```



Object-Oriented Programming



Объектно-ориентированное программирование

1. Класс представляет собой шаблон для создания объектов, описывает методы и свойства, доступные для работы с объектом.
2. Объект — это конкретный экземпляр класса.



Класс:
программист

Объект:
разработчик Иван

Атрибуты:
зарплата, обязанности

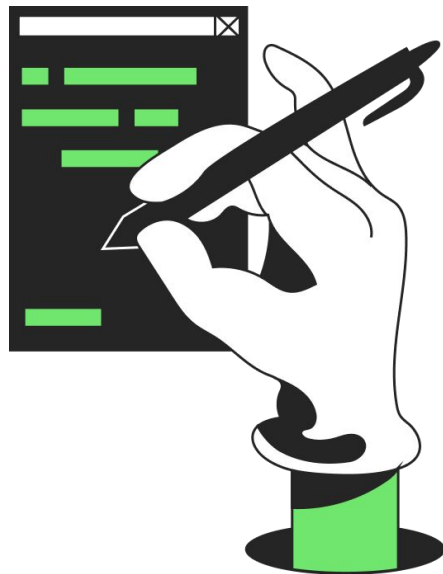
Методы:
написание кода



Концепции объектно-ориентированного программирования

Концепции

1. Абстракция
2. Наследование
3. Полиморфизм
4. Инкапсуляция





Абстракция

Концепция – это выделение наиболее значимых характеристик объекта или системы.

Как это выглядит: когда мы описываем что-то, мы упоминаем только о тех вещах, которые важны в нашем повествовании. Например, когда парень рассказывает другу о том, как в салоне видел крутую машину, он говорит о важных для них вещах: мощность двигателя, систему тормозов, диаметр колес. Хотя особенностей автомобиля безграничное множество.



Абстракция гласит — останавливаем внимание на важных и необходимых аспектах объекта и игнорируем ненужные для нас.



Наследование

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

```
1 @interface Parent : NSObject
2
3 @property (nonatomic, strong) NSString *value;
4
5 - (void)print;
6
7 @end
8
```



Полиморфизм

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











Инкапсуляция

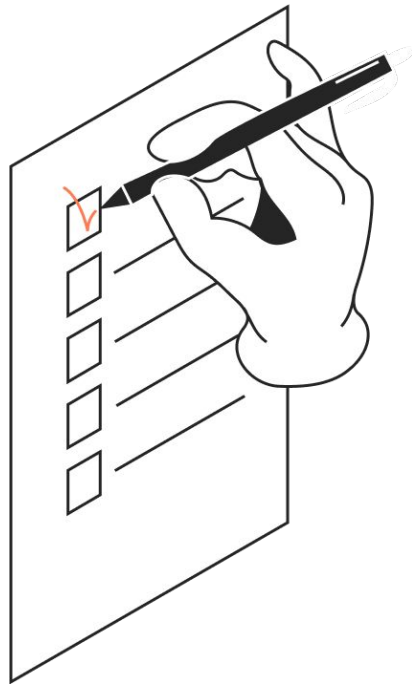
Инкапсуляция – это еще один принцип объектно-ориентированного программирования, который основан на сокрытии или открытии определенных методов и свойств.

Инкапсуляция помогает однозначно определить, какие методы и свойства необходимо скрыть от использования извне, а какие можно применять.



Что почитать?

-  Стивен Кочан. «Программирование на Objective-C».
-  Скотт Кнастер, Вакар Малик, Марк Далримпл. «Objective-C. Программирование для Mac OS X и iOS».
-  <https://www.objc.io/>
-  <https://habr.com/ru/post/147927/>
-  <https://habr.com/ru/post/87205/>
-  <https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/>





Вопросы?

Вопросы?



Вопросы?

