

Разработка iOS приложений

Лекция 3.

Objective-C продолжение

Селекторы

Что же такое селекторы?

- Аналог указателя на функцию
- `SEL s = @selector(<method name>);`
- `[obj performSelector: s];`
- Главный вопрос: «Как реализовать callback-функции»



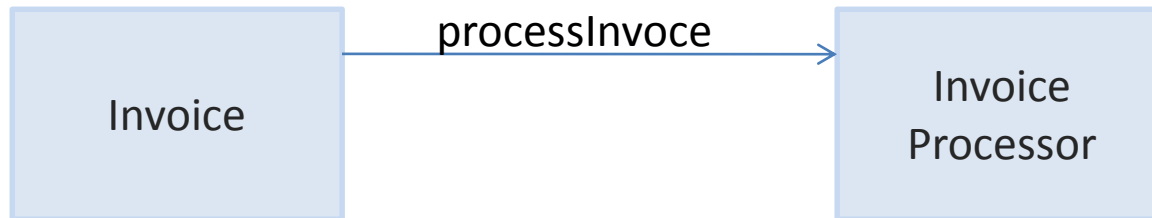
Invoice

Invoice
Processor

Селекторы

Что же такое селекторы?

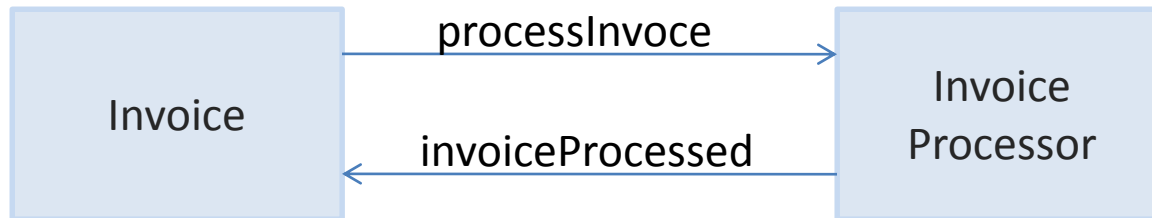
- Аналог указателя на функцию
- `SEL s = @selector(<method name>);`
- `[obj performSelector: s];`
- Главный вопрос: «Как реализовать callback-функции»



Селекторы

Что же такое селекторы?

- Аналог указателя на функцию
- `SEL s = @selector(<method name>);`
- `[obj performSelector: s];`
- Главный вопрос: «Как реализовать callback-функции»



Селекторы

Тут и помогут нам селекторы

```
@interface InvoiceProcessor : NSObject {
    id target;
    SEL callback;
}


-( void ) processInvoice: ( Invoice* )invoice
                    target: ( id )target
                    selector: ( SEL )selector {
    self.target = target;
    self.callback = selector;
    /* do invoice processing */
}

@end
```

Селекторы

Тут и помогут нам селекторы

Вызываемый класс



```
@interface InvoiceProcessor : NSObject {
    id target;
    SEL callback;
}

-( void ) processInvoice: ( Invoice* )invoice
                        target: ( id )target
                        selector: ( SEL )selector {
    self.target = target;
    self.callback = selector;
    /* do invoice processing */
}

@end
```

Селекторы

Тут и помогут нам селекторы

Вызываемый класс

```
@interface InvoiceProcessor : NSObject {  
    id target;  
    SEL callback;  
}
```

Объявление типа селектора

```
-( void ) processInvoice: ( Invoice* )invoice  
                        target: ( id )target  
                        selector: ( SEL )selector {  
    self.target = target;  
    self.callback = selector;  
    /* do invoice processing */  
}
```

```
@end
```

Селекторы

Тут и помогут нам селекторы

Вызываемый класс

```
@interface InvoiceProcessor : NSObject {  
    id target;  
    SEL callback;  
}
```

Объявление типа селектора

```
-( void ) processInvoice: ( Invoice* )invoice  
                        target: ( id )target  
                        selector: ( SEL )selector {  
    self.target = target;  
    self.callback = selector;  
    /* do invoice processing */  
}
```

Передаем ссылку на
объект ...

... и селектор

```
@end
```


Селекторы

А теперь клиент

```
@implementation Client
```

```
-( void ) run {  
    [ invoiceProcessor processInvoice: invoice target: self  
selector: @selector( invoiceProcessed: ) ];  
}  
  
-( void ) invoiceProcessed: ( Invoice* )invoice {  
    /* do something */  
}
```

Селекторы

А теперь клиент

Вызывающий класс

`@implementation Client`



```
-( void ) run {  
    [ invoiceProcessor processInvoice: invoice target: self  
selector: @selector( invoiceProcessed: ) ];  
}  
  
-( void ) invoiceProcessed: ( Invoice* )invoice {  
    /* do something */  
}
```

Селекторы

А теперь клиент

Вызывающий класс

`@implementation Client`

Передаёт себя в качестве аргумента

```
-( void ) run {
    [ invoiceProcessor processInvoice: invoice target: self
selector: @selector( invoiceProcessed: ) ];
}

-( void ) invoiceProcessed: ( Invoice* )invoice {
    /* do something */
}
```

Селекторы

А теперь клиент

Вызывающий класс

`@implementation Client`

Передаёт себя в качестве аргумента

```
-( void ) run {  
    [ invoiceProcessor processInvoice: invoice target: self  
selector: @selector( invoiceProcessed: ) ];  
}
```

И селектор на метод

```
-( void ) invoiceProcessed: ( Invoice* )invoice {  
    /* do something */  
}
```

Селекторы

А теперь клиент

Вызывающий класс

Передаёт себя в качестве аргумента

```
@implementation Client
```

```
-( void ) run {  
    [ invoiceProcessor processInvoice: invoice target: self  
selector: @selector( invoiceProcessed: ) ];  
}
```


И селектор на метод (сложные имена!)

```
-( void ) invoiceProcessed: ( Invoice* )invoice {  
    /* do something */  
}
```

Селекторы

Тут и помогут нам селекторы

```
@interface InvoiceProcessor : NSObject {  
    id target;  
    SEL callback;  
}  
  
-( void ) processInvoice: ( Invoice* )invoice  
                target: ( id )target  
                selector: ( SEL )selector {  
    self.target = target;  
    self.callback = selector;  
    /* do invoice processing */  
    [ target performSelector: selector ];  
}  
  
@end
```



После того, как требуемая работа выполнена
сообщаем клиенту

Протоколы

Вопрос: А как быть, если требуется вызвать разные функции в зависимости от условий?

Протоколы

Вопрос: А как быть, если требуется вызвать разные функции в зависимости от условий?

Простой ответ: передать несколько селекторов

Протоколы

Вопрос: А как быть, если требуется вызвать разные функции в зависимости от условий?

Простой ответ: передать несколько селекторов

Однако есть идея лучше – протоколы...

Протоколы

- Некоторый аналог интерфейсов в других языках
- Можно объявить только методы, данные – нельзя!
- Для определения протокола используется директива `@protocol`
- `ClassName : NSObject< ProtocolName >` - указываем о поддержке протокола («класс реализует интерфейс»)
- `id< ProtocolName >` - указатель на любой объект поддерживающий протокол

Протоколы

```
@protocol DoSomethingProtocol
```

```
-( void ) doSomething;
```

```
-( void ) doAnotherThingWithData: ( NSData* )data;
```

```
@optional
```

```
-( BOOL ) doOptionalThing;
```

```
@end
```

Протоколы

Объявляем протокол



```
@protocol DoSomethingProtocol
```

```
- ( void ) doSomething;
```

```
- ( void ) doAnotherThingWithData: ( NSData* )data;
```

```
@optional
```

```
- ( BOOL ) doOptionalThing;
```

```
@end
```

Протоколы

Объявляем протокол

`@protocol`

`DoSomethingProtocol`

Только методы, полей нет

`-(void) doSomething;`

`-(void) doAnotherThingWithData: (NSData*)data;`

`@optional`

`-(BOOL) doOptionalThing;`

`@end`

Протоколы

Объявляем протокол

`@protocol` DoSomethingProtocol

Только методы, полей нет

```
-( void ) doSomething;  
-( void ) doAnotherThingWithData: ( NSData* )data;
```

`@optional`

Методы могут быть опциональными (не обязательными к реализации)

```
-( BOOL ) doOptionalThing;
```

`@end`

Протоколы

```
@interface Worker : NSObject< DoSomethingProtocol > {  
}
```

```
@end
```

Протоколы

```
@interface Worker : NSObject<DoSomethingProtocol> {  
}
```

Объявляем о поддержке протокола

```
@end
```


Протоколы

```
@interface Worker : NSObject < DoSomethingProtocol > {  
}
```

Объявляем о поддержке протокола

@end

Повторно объявлять методы не нужно
(но нужно реализовать)

Протоколы

```
id< DoSomethingProtocol > worker = [ [ Worker alloc ] init ];
```

```
[ worker doSomething ];
```

Протоколы

```
id< DoSomethingProtocol > worker = [ [ Worker alloc ] init ];
```

```
[ worker doSomething ];
```

Указатель на объект реализующий
протокол

Протоколы

```
id< DoSomethingProtocol > worker = [ [ Worker alloc ] init ];
```

Звездочка не ставится

```
[ worker doSomething ];
```

Указатель на объект реализующий
протокол

Паттерн «Делегат»

- Стандартный паттерн, на котором строится почти весь UIKit
- Для его реализации лучше всего использовать протоколы (или блоки...)
- С его помощью задаем перечень callback методов (больше чем один)

Паттерн «Делегат»

Определяет протокол

InvoiceViewDelegate

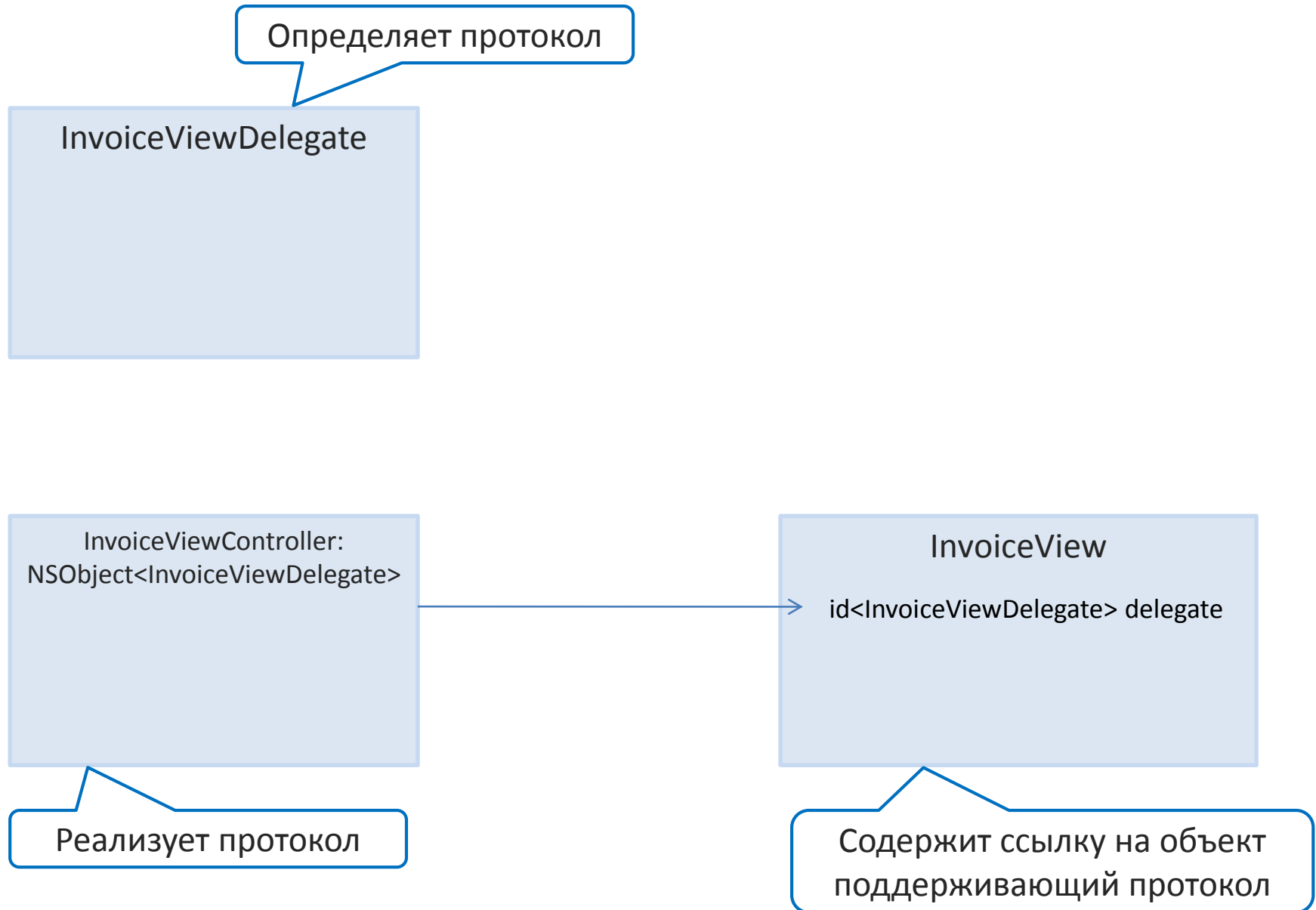
InvoiceViewController:
NSObject<InvoiceViewDelegate>

Реализует протокол

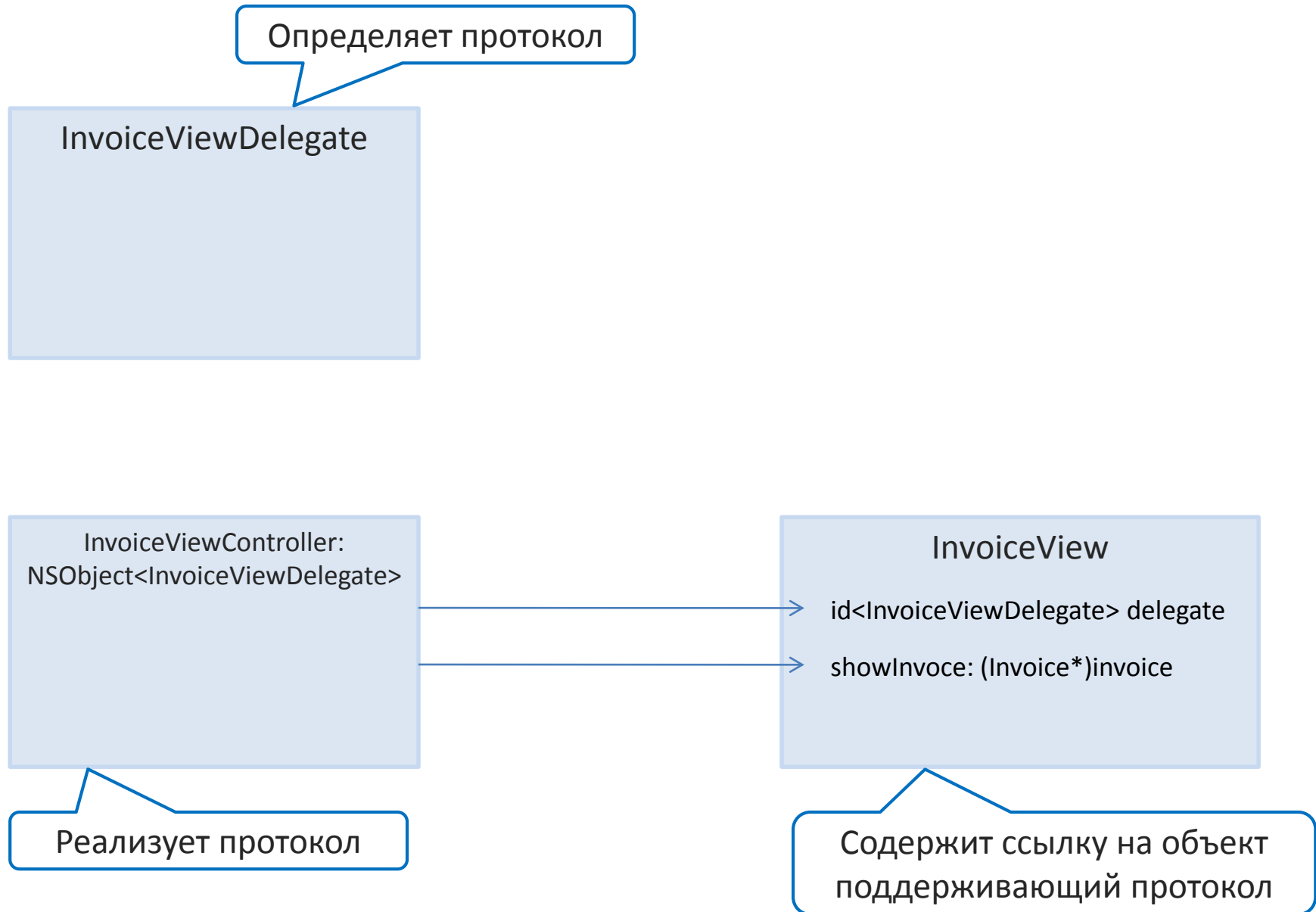
InvoiceView

Содержит ссылку на объект
поддерживающий протокол

Паттерн «Делегат»



Паттерн «Делегат»



Паттерн «Делегат»

Определяет протокол

InvoiceViewDelegate

-(void)invoiceWillBeShown
-(void)invoiceDidShown

InvoiceViewController:
NSObject<InvoiceViewDelegate>

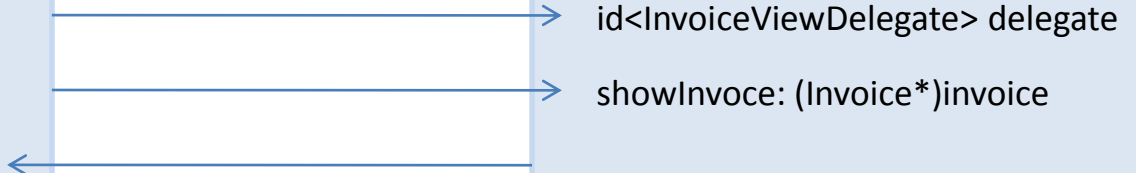
-(void)invoiceWillBeShown
-(void)invoiceDidShown

Реализует протокол

InvoiceView

id<InvoiceViewDelegate> delegate
showInvoice: (Invoice*)invoice

Содержит ссылку на объект
поддерживающий протокол



Паттерн «Делегат»

InvoiceView.h

```
@protocol InvoiceViewDelegate;
```

```
@interface InvoiceView : NSObject
```

```
@property ( ... ) id< InvoiceViewDelegate > delegate;
```

```
@end
```

```
@protocol InvoiceViewDelegate
```

```
-( void ) invoiceWillBeShown;
```

```
-( void ) invoiceDidShown;
```

```
@end
```

Паттерн «Делегат»

InvoiceView.h

Предварительное объявление

```
@protocol InvoiceViewDelegate;
```

```
@interface InvoiceView : NSObject
```

Указатель на
делегат

```
@property ( ... ) id< InvoiceViewDelegate > delegate;
```

```
@end
```

```
@protocol InvoiceViewDelegate
```

```
-( void ) invoiceWillBeShown;
```

```
-( void ) invoiceDidShown;
```

```
@end
```

Объявление протокола

Паттерн «Делегат»

InvoiceViewController.h

```
@interface InvoiceViewController : NSObject< InvoiceViewDelegate >
{
}

@end
```

Паттерн «Делегат»

InvoiceViewController.h

```
@interface InvoiceViewController : NSObject<InvoiceViewDelegate>
{
}
```

```
@end
```

Протокол делегата

Паттерн «Делегат»

InvoiceViewController.m

```
@implementation InvoiceViewController
```

```
-( id ) init {
```

```
...
```

```
    InvoiceView* invoiceView = [ InvoiceView view ];
```

```
    invoiceView.delegate = self;
```

```
}
```

```
#pragma mark - InvoiceViewDelegate
```

```
-( void ) invoiceWillBeShown {
```

```
}
```

```
-( void ) invoiceDidShown {
```

```
}
```


```
@end
```

Паттерн «Делегат»

InvoiceViewController.m

@implementation InvoiceViewController Создаем InvoiceView

-(**id**) init {



```
InvoiceView* invoiceView = [ InvoiceView view ];  
invoiceView.delegate = self;
```

}

#pragma mark - InvoiceViewDelegate

-(**void**) invoiceWillBeShown {
}

-(**void**) invoiceDidShown {
}

@end

Паттерн «Делегат»

InvoiceViewController.m

@implementation InvoiceViewController Создаем InvoiceView

-(**id**) init {

InvoiceView* invoiceView = [**InvoiceView** view];

invoiceView.delegate = **self**;

}

Устанавливаем себя в качестве делегата

#pragma mark - InvoiceViewDelegate

-(**void**) invoiceWillBeShown {
}

-(**void**) invoiceDidShown {
}

@end

Паттерн «Делегат»

InvoiceViewController.m

@implementation InvoiceViewController Создаем InvoiceView

-(**id**) init {

InvoiceView* invoiceView = [**InvoiceView** view];
invoiceView.delegate = **self**;

Устанавливаем себя в качестве
делегата

#pragma mark - InvoiceViewDelegate

-(**void**) invoiceWillBeShown {
}

-(**void**) invoiceDidShown {
}

Делаем то, что нужно при
Выполнении действия

@end

Паттерн «Делегат»

InvoiceView.m

@implementation InvoiceView

```
-( void ) showInvoice {  
    [ delegate invoiceWillBeShown ];  
    /* show invoice */  
    [ delegate invoiceDidShown ];  
}
```



Вызов методов делегата

@end

Категории

Иногда, нужно в ЧУЖОЙ класс добавить СВОЁ сообщение, например в NSString

Категории

Иногда, нужно в ЧУЖОЙ класс добавить СВОЁ сообщение, например в NSString

В этом нам помогут «категории»


Объявление

```
@interface NSString ( OurCategory )
```

```
-( void ) doSomething;
```

```
@end
```

В скобочках пишем имя категории



Категории

Иногда, нужно в ЧУЖОЙ класс добавить СВОЁ сообщение, например в NSString

В этом нам помогут «категории»

Объявление

```
@interface NSString ( OurCategory )
```

```
-( void ) doSomething;
```

```
@end
```



Наследование
указывать не надо

Категории

Иногда, нужно в ЧУЖОЙ класс добавить СВОЁ сообщение, например в NSString

В этом нам помогут «категории»

Объявление

```
@interface NSString ( OurCategory )
```

```
-( void ) doSomething;
```

```
@end
```

Объявление
добавляемой функции

Категории

Иногда, нужно в ЧУЖОЙ класс добавить СВОЁ сообщение, например в NSString


В этом нам помогут «категории»

Определение (реализация)

```
@implementation NSString ( OurCategory )
```

```
-( void ) doSomething  
{  
    /* do something */  
}
```

```
@end
```



Аналогично определению
обычного класса, добавляем
скобки

Категории

Иногда, нужно в ЧУЖОЙ класс добавить СВОЁ сообщение, например в NSString

В этом нам помогут «категории»

Определение (реализация)

```
@implementation NSString ( OurCategory )
```

```
-( void ) doSomething  
{  
    /* do something */  
}
```

```
@end
```

Реализуем метод обычным способом

Неявные Категории

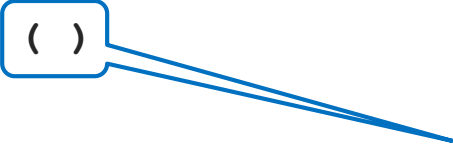
Иногда нет смысла задавать имя категории, тогда можно оставить скобки пустыми.

ВАЖНО! В одной единице трансляции (.m, .mm, ...) может быть только одна неявная категория

```
@implementation NSString ( )
```

```
-( void ) doSomething  
{  
    /* do something */  
}
```

```
@end
```



Скобки пусты, но быть должны

Инкапсуляция на стероидах

OurClass.h

```
@interface OurClass : NSObject
```

```
-( void ) publicMessage;
```

```
@end
```

OurClass.m

```
@interface OurClass ( )
```

```
-( void ) secretMessage;
```

```
@end
```

```
@implementation NSString
```

```
{  
    NSString* _field;  
}
```

```
-( void ) publicMessage { /* do something */ }
```

```
-( void ) doSomething { /* do something */ }
```

```
@end
```

Практическое задание

Сегодня требуется проделать три вещи (по окончании каждой – я смотрю):

1. Потренироваться с target/selector, для этого
 1. Реализовать у класса Weather метод loadFromFile:target:selector: (пусть пока ничего не грузится, а возвращается hard coded)
 2. Написать клиента, который будет грузить Weather из файла и в callback писать в лог, что файл загружен – NSLog(@"file: %@ loaded", fileName)
2. Улучшить предыдущую реализацию с помощью протокола
 1. Объявить протокол WeatherDelegate с двумя методами didWeatherLoadSucceeded и didWeatherLoadFailed: (NSError*)error
 2. Модифицировать клиента чтобы он писал в лог соответствующее функции сообщение
3. Сделать собственно загрузку Weather из xml файла (forecast.xml)

См. Apple Guides

- Event-Driven XML Programming Guide, “Handling XML Elements and Attributes”