Лекция #9.9 Делегирование

Делегирование представляет паттерн объектно-ориентированного программирования, который позволяет одному объекту делегировать/перенаправить все запросы другому объекту. В определенной степени делегирование может выступать альтернативой наследованию. И преимуществом Kotlin в данном случае состоит в том, что Kotlin нативно поддерживает данный паттерн, предоставляя необходимый инструментарий.

Формальный синтаксис:

```
interface Base {
    fun someFun()
}

class BaseImpl() : Base {
    override fun someFun() { }
}

class Derived(someBase: Base) : Base by someBase
```

Есть некоторый интерфейс - Base, который определяет некоторый функционал. Есть его реализация в виде класса BaseImpl.

И есть еще один класс - Derived, который также применяет интерфейс Base. Причем после указания применяемого интерфейса идет ключевое слово **by**, а после него - объект, которому будут делегироваться вызовы.

```
class Derived(someBase: Base) : Base by someBase
```

То есть в данной схеме класс Derived будет делегировать вызовы объекту someBase, который представляет интерфейс Base и передается через первичный конструктор. При этом Derived может не реализовать интерфейс Base или реализовать неполностью - какие-то отдельные свойства и функции.

Например, рассмотрим следующие классы:

```
interface Messenger{
    fun send(message: String)
}
class InstantMessenger(val programName: String) : Messenger{

    override fun send(message: String){
        println("Message `$message` has been sent")
    }
}
class SmartPhone(val name: String, m: Messenger): Messenger by  m
```

Здесь определен интерфейс `Messenger`, который представляет условно программу для отправки сообщений. Для условной отправки сообщений определена функция `send()`.

Также есть класс `InstantMessenger` - программа мгновенных сообщений или проще говоря мессенджер, который применяет интерфейс Messenger, реализуя его функцию `send()`.

Далее определен класс `SmartPhone`, который представляет смартфон и также применяет интерфейс Messenger, но не реализует его. Вместо этого он принимает через первичный конструктор объект Messenger и делегирует ему обращение к функции `send()`.

Применим классы:

```
fun main() {
    val telegram = InstantMessenger("Telegram")
    val pixel = SmartPhone("Pixel 5", telegram)
    pixel.send("Hello Kotlin")
    pixel.send("Learn Kotlin on Metanit.com")
}
```

Здесь создан объект pixel, который представляет класс SmartPhone. Поскольку SmartPhone применяет интерфейс Messenger, то мы можем вызвать у объекта pixel функцию `send()` для отправки условного сообщения. Однако сам класс SmartPhone НЕ реализует функцию send - само выполнение этой функции делегируется объекту `telegram`, который в реальности выполняет отправку сообщения. Соответственно при выполнении программы мы увидим следующий консольный вывод:

```
Message `Hello Kotlin` has been sent
Message `Learn Kotlin on Metanit.com` has been sent
```

Множественное делегирование

Подобным образом один объект может делегировать выполнение различных функций разным объектам. Например:

```
fun main() {
    val telegram = InstantMessenger("Telegram")
    val photoCamera = PhotoCamera()
    val pixel = SmartPhone("Pixel 5", telegram, photoCamera)
    pixel.send("Hello Kotlin")
    pixel.takePhoto()
}

interface Messenger{
    fun send(message: String)
}
class InstantMessenger(val programName: String) : Messenger{
    override fun send(message: String) = println("Send message: `$message`")
}
interface PhotoDevice{
    fun takePhoto()
}
class PhotoCamera: PhotoDevice{
    override fun takePhoto() = println("Take a photo")
}
class SmartPhone(val name: String, m: Messenger, p: PhotoDevice)
    : Messenger by  m, PhotoDevice by p
```

Здесь класс SmartPhone также реализует интерфейс PhotoDevice, который предоставляет функцию `takePhoto()` для съемки фото. Но выполнение этой функции он делегирует параметру p, который представляет интерфейс PhotoDevice и в роли которого выступает объект PhotoCamera.

Переопределение функций

Класс может переопределять часть функций интерфейса, в этом случае выполнение этих функций не делегируется. Например:

```
fun main() {
    val telegram = InstantMessenger("Telegram")
    val pixel = SmartPhone("Pixel 5", telegram)
    pixel.sendTextMessage()
    pixel.sendVideoMessage()
}

interface Messenger{
    fun sendTextMessage()
    fun sendVideoMessage()
}
class InstantMessenger(val programName: String) : Messenger{
    override fun sendTextMessage() = println("Send text message")
    override fun sendVideoMessage() = println("Send video message")
}
class SmartPhone(val name: String, m: Messenger) : Messenger by  m{
    override fun sendTextMessage() = println("Send sms")
}
```

В данном случае класс SmartPhone реализует функцию `sendTextMessage()`, поэтому ее выполнение не делегируется. Консольный вывод программы:

```
Send sms
Send video message
```

Делегирование свойств

По аналогии с функциями объект может делегировать обращение к свойствам:

```
fun main() {
    val telegram = InstantMessenger("Telegram")
    val pixel = SmartPhone("Pixel 5", telegram)
    println(pixel.programName)  // Telegram
}
interface Messenger{
    val programName: String
}
class InstantMessenger(override val programName: String) : Messenger
class SmartPhone(val name: String, m: Messenger) : Messenger by  m
```

Здесь интерфейс Messenger определяет свойство `programName` - название программы отправки. Класс SmartPhone не реализует это свойство, поэтому обращение к этому свойству делегируется объекту `m`.

Если бы класс SmartPhone сам реализовал это свойство, то делегирования бы не было:

```kotlin
fun main() {
    val telegram = InstantMessenger("Telegram")
    val pixel = SmartPhone("Pixel 5", telegram)
    println(pixel.programName)  // Default Messenger
}
interface Messenger{
    val programName: String
}
class InstantMessenger(override val programName: String) : Messenger
class SmartPhone(val name: String, m: Messenger) : Messenger by  m{
    override val programName = "Default Messenger"
}
```