

Тема 1. Знакомство с объектно-ориентированными возможностями языка Kotlin.

Романов Владимир Юрьевич
МГУ им. М.В.Ломоносова, ф-т ВМК
vladimir.romanov@gmail.com

1.5. ДЕЛЕГИРОВАНИЕ В ЯЗЫКЕ KOTLIN

1.5.1. Делегирование класса

- *Шаблон делегирования* - альтернатива наследованию реализации
- Класс **Derived** может реализовать интерфейс **Base**, делегируя все свои публичные члены указанному в параметре объекту
- Предложение **by** в списке супертипов для **Derived** указывает, что **b** будет храниться внутри в объектах **Derived**,
- Компилятор сгенерирует все методы **Base**, которые направляют на **b**.

```
interface Base {  
    fun print()  
}  
class BaseImpl(val x: Int) : Base {  
    override fun print() { print(x) }  
}  
class Derived(b: Base) : Base by b  
fun main(args: Array<String>) {  
    val b = BaseImpl(10)  
    Derived(b).print() // prints 10  
}
```

Переопределение члена интерфейса, реализованного делегированием

```
interface Base {  
    fun printMessage()  
    fun printMessageLine()  
}  
class BaseImpl(val x: Int) : Base {  
    override fun printMessage() { print(x) }  
    override fun printMessageLine() { println(x) }  
}  
class Derived(b: Base) : Base by b {  
    override fun printMessage() { print("abc") }  
}  
fun main() {  
    val b = BaseImpl(10)  
    Derived(b).printMessageLine() // 10  
    Derived(b).printMessage() // abc  
}
```

Доступ к реализациям членов интерфейса

```
interface Base {  
    val message: String  
    fun print()  
}  
class BaseImpl(val x: Int) : Base {  
    override val message = "BaseImpl: x = $x"  
    override fun print() { println(message) }  
}  
class Derived(b: Base) : Base by b {  
    // Это свойство недоступно из реализации b `print`  
    override val message = "Message of Derived"  
}  
fun main() {  
    val b = BaseImpl(10)  
    val derived = Derived(b)  
    derived.print() // BaseImpl: x = 10  
    println(derived.message) // Message of Derived  
}
```

1.5.2 Делегирование свойств

```
class Example {  
    var p: String by Delegate1()  
}
```

```
fun main() {  
    var e = Example()  
    e.p = "Hello"  
    println(e.p);  
}
```

Синтаксис делегируемых свойств:

`val/var <имя свойства>: <Тип> by <выражение>`

- Выражение после **by** — делегат:
- обращения **get()**, **set()** к свойству будут обрабатываться этим выражением.
- Делегат не обязан реализовывать какой-то интерфейс. Достаточно, чтобы у него были методы **getValue()** и **setValue()** с определённой сигнатурой.

Сигнатура делегата

```
class Delegate1 {  
  operator  
  fun getValue(thisRef: Any?,  
               property: KProperty<*>): String  
  {  
    return "$thisRef, спасибо за делегирование мне"  
      + " '${property.name}'!"  
  }  
  operator  
  fun setValue(thisRef: Any?,  
              property: KProperty<*>,  
              value: String)  
  {  
    println("$value было присвоено значению " +  
      "'${property.name}' в $thisRef." )  
  }  
}
```

Действия делегата

```
class Delegate1 {  
operator fun getValue(thisRef: Any?,  
    property: KProperty<*>): String {  
    return "$thisRef, спасибо за делегирование мне"+  
        " '${property.name}'!"  
}  
operator fun setValue(thisRef: Any?,  
    property: KProperty<*>,  
    value: String) {  
    println("$value было присвоено значению " +  
        "'${property.name}' в $thisRef." )  
}  
}
```

*// Hello было присвоено значению 'p' в
delegates_prop.Example@433c675d.'*

*// delegates_prop.Example@433c675d, спасибо за делегирование
мне 'p'!*

Ленивые свойства (lazy properties)

- **lazy** функция, которая принимает лямбду и возвращает экземпляр класса **Lazy<T>**
- экземпляр - делегат для реализации ленивого свойства
- первый вызов **get()** запускает лямбда-выражение, переданное **lazy()** в качестве аргумента, и запоминает полученное значение
- последующие вызовы просто возвращают вычисленное значение

```
val lazyValue: String by lazy {  
    println("computed!")  
    "Hello"  
}  
fun main(args: Array<String>) {  
    println(lazyValue)  
    println(lazyValue)  
}  
//computed!  
//Hello  
//Hello
```

Ленивые свойства. Функция lazy

```
public actual  
fun <T> lazy(initializer: () -> T): Lazy<T>  
    = SynchronizedLazyImpl(initializer)
```

```
public interface Lazy<out T> {  
    public val value: T  
  
    public fun isInitialized(): Boolean  
}
```

Обозреваемые свойства. Observable (1)

- Функция **Delegates.observable()** принимает два аргумента:
- начальное значение свойства
- обработчик (лямбда), который вызывается *после* изменения свойства.
- У обработчика три параметра: описание свойства, которое изменяется, старое значение и новое значение.

```
class User {  
  var name: String  
    by Delegates.observable("<no name>") {  
      prop, old, new -> println("$prop: $old -> $new")  
    }  
}  
fun main(args: Array<String>) {  
  val user = User()  
  user.name = "first"  
  user.name = "second"  
}
```

Обозреваемые свойства. Observable (2)

```
class User {  
    var name: String  
        by Delegates.observable("<no name>") {  
            prop, old, new -> println("$prop: $old -> $new")  
        }  
}  
  
fun main(args: Array<String>) {  
    val user = User()  
    user.name = "first"  
    user.name = "second"  
}  
  
// var delegates_prop.observable.User.name: kotlin.String: <no  
name> -> first  
// var delegates_prop.observable.User.name: kotlin.String: first ->  
second
```

Хранение свойств в ассоциативном списке

- Хранение свойств в ассоциативном списке.
- В этом примере конструктор принимает ассоциативный список
- Делегированные свойства берут значения из этого ассоциативного списка (по строковым ключам)

```
class User(val map: Map<String, Any?>) {  
    val name: String by map  
    val age: Int    by map  
}  
val user = User(mapOf(  
    "name" to "John Doe",  
    "age"  to 25  
))  
println(user.name) // Prints "John Doe"  
println(user.age)  // Prints 25
```