Лекция 11. Классы, объекты и интерфейсы

Лекция 11. Классы, объекты и интерфейсы

1. Чистим код: локальные функции и расширения
2. Создание иерархий классов
3. Объявление классов с нетривиальными конструкторами или свойствами
4. Методы, сгенерированные компилятором: классы данных и делегирование

```
class User(val id: Int, val name: String, val address: String)

fun saveUser(user: User) {
  if (user.name.isEmpty()) {
    throw IllegalArgumentException(
      "Can't save user ${user.id}: empty Name")
  }

  if (user.address.isEmpty()) {
    throw IllegalArgumentException(
      "Can't save user ${user.id}: empty Address")
  }

  // Сохранение информации о пользователе в базе данных
}

>>> saveUser(User(1, "", ""))
java.lang.IllegalArgumentException: Can't save user 1: empty Name
```

```
class User(val id: Int, val name: String, val address: String)

fun saveUser(user: User) {
  fun validate(user: User,
               value: String,
               fieldName: String) {
    if (value.isEmpty()) {
      throw IllegalArgumentException(
        "Can't save user ${user.id}: empty $fieldName")
    }
  }

  validate(user, user.name, "Name")
  validate(user, user.address, "Address")

  // Сохранение информации о пользователе в базе данных
}
```

Объявление локальной функции для проверки произвольного поля

Вызов функции для проверки конкретных полей

```
fun saveUser(user: User) {
  fun validate(value: String, fieldName: String) {
    if (value.isEmpty()) {
      throw IllegalArgumentException(
        "Can't save user ${user.id}: " +
        "empty $fieldName")
    }
  }

  validate(user.name, "Name")
  validate(user.address, "Address")

  // Сохранение информации о пользователе в базе данных
}
```

Теперь не нужно дублировать параметра user в функции saveUser

Можно напрямую обращаться к параметрам внешней функции

```
class User(val id: Int, val name: String, val address: String)

fun User.validateBeforeSave() {
  fun validate(value: String, fieldName: String) {
    if (value.isEmpty()) {
      throw IllegalArgumentException(
        "Can't save user $id: empty $fieldName")
    }
  }
  validate(name, "Name")
  validate(address, "Address")
}

fun saveUser(user: User) {
  user.validateBeforeSave()

  // Сохранение пользователя в базу данных
}
```

К свойствам класса User можно обращаться напрямую

Вызов функции-расширения

```
interface Clickable {
    fun click()
}
class Button : Clickable {
    override fun click() = println("I was clicked")
}

>>> Button().click()
I was clicked
```

```
interface Clickable {
    fun click()
}

class Button : Clickable {
    override fun click() = println("I was clicked")
}

>>> Button().click()
I was clicked
```

```
interface Clickable {
    fun click()
    fun showOff() = println("I'm clickable!")
}
```

Обычное объявление метода

Метод с реализацией по умолчанию

```
interface Focusable {
    fun setFocus(b: Boolean) =
      println("I ${if (b) "got" else "lost"} focus.")

    fun showOff() = println("I'm focusable!")
}

class Button : Clickable, Focusable {
    override fun click() = println("I was clicked")

    override fun showOff() {
      super<Clickable>.showOff()
      super<Focusable>.showOff()
    }
}
fun main(args: Array<String>) {
    val button = Button()
    button.showOff()
    button.setFocus(true)
    button.click()
}
```

Вы должны явно реализовать метод, если наследуется несколько его реализаций

Ключевое слово «super» с именем супертипа в угловых скобках определяет родителя, чей метод будет вызван

```
I'm clickable!
I'm focusable!
I got focus.
I was clicked.
```

```
open class RichButton : Clickable {

    fun disable() {}

    open fun animate() {}

    override fun click() {}
}
```

Это открытый класс: другие могут наследовать его

Это закрытая функция: ее невозможно переопределить в подклассе

Это открытая функция: ее можно переопределить в подклассе

Переопределение открытой функции также является открытым

```
open class RichButton : Clickable {
    final override fun click() {}
}
```

Ключевое слово «final» здесь не лишнее, потому что модификатор «override» без «final» означает, что метод останется открытым

```
abstract class Animated {

    abstract fun animate()

    open fun stopAnimating() {
    }

    fun animateTwice() {
    }
}
```

Это абстрактный класс: нельзя создать его экземпляр

Это абстрактная функция: она не имеет реализации и должна быть переопределена в подклассах

Конкретные функции в абстрактных классах по умолчанию закрыты, но их можно сделать открытыми

Модификатор	Соответствующий член	Комментарии
`final`	Не может быть переопределен	Применяется к членам класса по умолчанию
`open`	Может быть переопределен	Должен указываться явно
`abstract`	Должен быть переопределен	Используется только в абстрактных классах; абстрактные методы не могут иметь реализацию
`override`	Переопределяет метод суперкласса или интерфейса	По умолчанию переопределяющий метод открыт, если только не объявлен как `final`

Модификатор	Член класса	Объявление верхнего уровня
`public` (по умолчанию)	Доступен повсюду	Доступно повсюду
`internal`	Доступен только в модуле	Доступно в модуле
`protected`	Доступен в подклассах	–
`private`	Доступен в классе	Доступно в файле

```
internal open class TalkativeButton : Focusable {
    private fun yell() = println("Hey!")
    protected fun whisper() = println("Let's talk!")
}
fun TalkativeButton.giveSpeech() {
    yell()
    whisper()
}
```

Ошибка: «публичный» член класса раскрывает «внутренний» тип-приемник «TalkativeButton»

Ошибка: функция «yell» недоступна; в классе «TalkativeButton» она объявлена с модификатором «private»

Ошибка: функция «whisper» недоступна; в классе «TalkativeButton» она объявлена с модификатором «protected»

```
interface State: Serializable

interface View {
    fun getCurrentState(): State
    fun restoreState(state: State) {}
}

  class Button : View {
      override fun getCurrentState(): State = ButtonState()

      override fun restoreState(state: State) { /*...*/ }

      class ButtonState : State { /*...*/ }
  }
```

Это аналог статического вложенного класса в Java

```
sealed class Expr {
  class Num(val value: Int) : Expr()
  class Sum(val left: Expr, val right: Expr) : Expr()
}

fun eval(e: Expr): Int =
    when (e) {
      is Expr.Num -> e.value
      is Expr.Sum -> eval(e.right) + eval(e.left)
    }
```

Представление выражений запечатанными классами...

...и перечислить все возможные подклассы в виде вложенных классов.

Выражение «when» охватывает все возможные варианты, поэтому ветка «else» не нужна.

При обработке всех подклассов запечатанного класса в выражении when нет необходимости в ветке по умолчанию. Обратите внимание: модификатор sealed означает, что класс по умолчанию открыт, добавлять модификатор open не требуется. Поведение запечатанных классов показано на рис. 4.2.

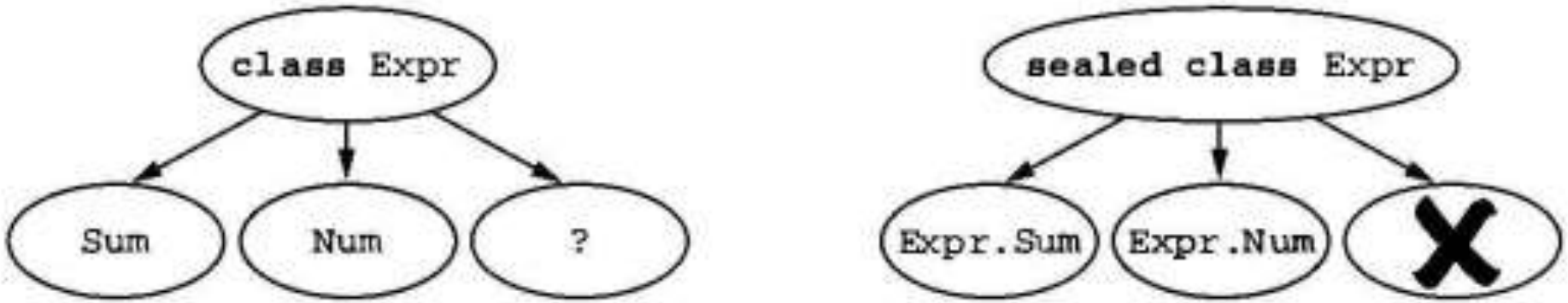


Рис. 4.2. Запечатанный класс не может иметь наследников, объявленных вне класса

Когда выражение when используется с запечатанными классами, при добавлении нового подкласса выражение when, возвращающее значение, не скомпилируется, а сообщение об ошибке укажет, какой код нужно изменить.

```
class User constructor(_nickname: String) {
  val nickname: String

  init {
    nickname = _nickname
  }
}
```

Основной конструктор с одним параметром

Блок инициализации

```
class User(_nickname: String) {
    val nickname = _nickname
}
```

Основной конструктор с одним параметром

Свойство инициализируется значением параметра

```
class User(val nickname: String)
```

«val» означает, что для параметра должно быть создано соответствующее свойство

```
class User(val nickname: String,
           val isSubscribed: Boolean = true)
```

Значение по умолчанию для параметра конструктора

Если класс имеет суперкласс, основной конструктор также должен инициализировать свойства, унаследованные от суперкласса. Сделать это можно, перечислив параметры конструктора суперкласса после имени его типа в списке базовых классов:

```
open class User(val nickname: String) { ... }

class TwitterUser(nickname: String) : User(nickname) { ... }
```

Если вообще не объявить никакого конструктора, компилятор добавит конструктор по умолчанию, который ничего не делает:

```
open class Button
```

Будет сгенерирован конструктор по умолчанию без аргументов

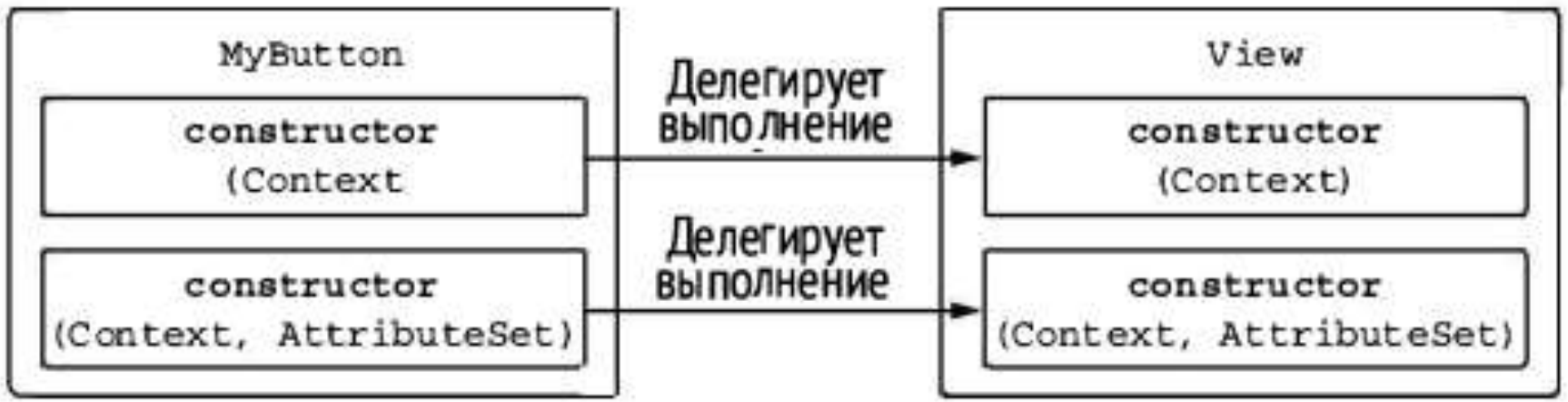
Если вы захотите унаследовать класс `Button` в другом классе, не объявляя своих конструкторов, вы должны будете явно вызвать конструктор суперкласса, даже если тот не имеет параметров:

```
class RadioButton: Button()
```

```
open class View {
    constructor(ctx: Context) {
      // некоторый код
    }

    constructor(ctx: Context, attr: AttributeSet) {
      // некоторый код
    }
}
```

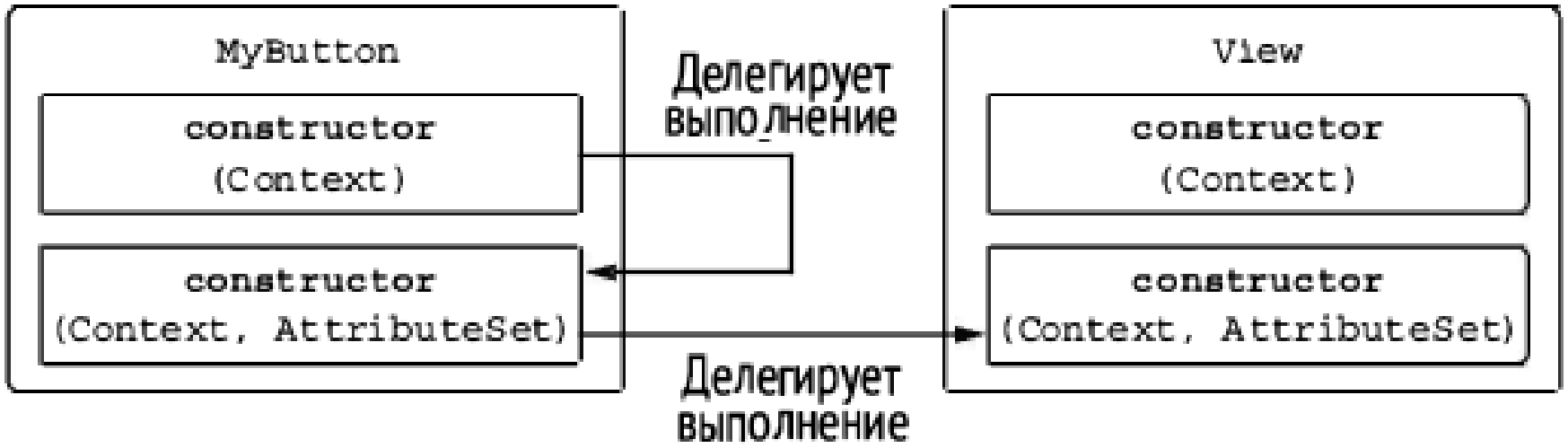
```
class MyButton : View {
    constructor(ctx: Context)
      : super(ctx) {
      // ...
    }

    constructor(ctx: Context, attr: AttributeSet)
      : super(ctx, attr) {
      // ...
    }
}
```

Вызов конструкторов суперкласса



```
class MyButton : View {
    constructor(ctx: Context): this(ctx, MY_STYLE) {
      // ...
    }

    constructor(ctx: Context, attr: AttributeSet): super(ctx, attr) {
      // ...
    }
}
```

Делегирует выполнение другому конструктору класса



```
interface User {
    val nickname: String
}

class PrivateUser(override val nickname: String) : User

class SubscribingUser(val email: String) : User {
    override val nickname: String
      get() = email.substringBefore('@') )
    }

class FacebookUser(val accountId: Int) : User {
    override val nickname = getFacebookName(accountId)
}

>>> println(PrivateUser("test@kotlinlang.org").nickname)
test@kotlinlang.org
>>> println(SubscribingUser("test@kotlinlang.org").nickname)
test
```

Свойство основного конструктора

Собственный метод чтения

Инициализация свойства

```
interface User {
    val email: String
    val nickname: String
        get() = email.substringBefore('@')
}
```

Свойство не имеет поля для хранения значения: результат вычисляется при каждой попытке доступа

Этот интерфейс определяет абстрактное свойство `email`, а также свойство `nickname` с методом доступа. Первое свойство должно быть переопределено в подклассах, а второе может быть унаследовано.

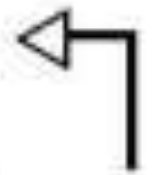
В отличие от свойств, реализованных в интерфейсах, свойства, реализованные в классах, имеют полный доступ к полям, хранящим их значения. Давайте посмотрим, как обращаться к ним из методов доступа.

```
class User(val name: String) {
    var address: String = "unspecified"
      set(value: String) {
        println("""
          Address was changed for $name:
          "$field" -> "$value".""".trimIndent())
        field = value
      }
}

>>> val user = User("Alice")
>>> user.address = "Elsenheimerstrasse 47, 80687 Muenchen"
```

Чтение значения из поля

Изменение значения поля

```
class LengthCounter {
    var counter: Int = 0
      private set

    fun addWord(word: String) {
      counter += word.length
    }
}
```



Значение этого свойства нельзя изменить вне класса

```
>>> val lengthCounter = LengthCounter()
>>> lengthCounter.addWord("Hi!")
>>> println(lengthCounter.counter)
3
```

class Client(val name : String , val postatCode : Int)

equals, hashCode и
toString

```
class Client(val name: String, val postalCode: Int) {
    override fun equals(other: Any?): Boolean {
      if (other == null || other !is Client)
        return false
      return name == other.name &&
        postalCode == other.postalCode
    }
    override fun toString() = "Client(name=$name, postalCode=$postalCode)"
}
```

«Any» – это аналог java.lang.Object: суперкласс всех классов в Kotlin. Знак вопроса в «Any?» означает, что аргумент «other» может иметь значение null

Убедиться, что «other» имеет тип Client

Вернуть результат сравнения свойств

```
class DelegatingCollection<T> : Collection<T> {
    private val innerList = arrayListOf<T>()

    override val size: Int get() = innerList.size
    override fun isEmpty(): Boolean = innerList.isEmpty()
    override fun contains(element: T): Boolean = innerList.contains(element)
    override fun iterator(): Iterator<T> = innerList.iterator()
    override fun containsAll(elements: Collection<T>): Boolean =
        innerList.containsAll(elements)
}
```

```
class DelegatingCollection<T>(
    innerList: Collection<T> = ArrayList<T>()
) : Collection<T> by innerList {}
```

```
class CountingSet<T>(
    val innerSet: MutableCollection<T> = HashSet<T>()
) : MutableCollection<T> by innerSet {

    var objectsAdded = 0

    override fun add(element: T): Boolean {
      objectsAdded++
      return innerSet.add(element)
    }

    override fun addAll(c: Collection<T>): Boolean {
      objectsAdded += c.size
      return innerSet.addAll(c)
    }
}
```

Делегирование реализации MutableCollection объекту в поле innerSet

Собственная реализация вместо делегирования

```
>>> val cset = CountingSet<Int>()
>>> cset.addAll(listOf(1, 1, 2))
>>> println("${cset.objectsAdded} objects were added, ${cset.size} remain")
3 objects were added, 2 remain
```