

# АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 6

# ПЛАН

- Интерфейсы, наследование
- Модификаторы доступа
- Sealed-классы
- Исключения
- lambda с получателем

# РЕАЛИЗАЦИЯ ПО УМОЛЧАНИЮ

- Примерно как в Java
- В контексте синтаксиса Kotlin - как метод

# РЕАЛИЗАЦИЯ ПО УМОЛЧАНИЮ

```
1 interface URLFactory {  
2     fun newURL(): URL = URL("http://www.ya.ru")  
3 }
```

# СТАТИКА И ИНТЕФЕЙСЫ

- В том же духе, как и в классах
- Статических методов в прямом смысле нет
- Но можно завести компаньона
- Или именованный объект

# ПРИМЕР

```
1 interface Iface {  
2     companion object {  
3         fun f() = 5  
4         const val FLAG = 0x123  
5     }  
6 }  
7 object Use {  
8     val v1 = Iface.FLAG  
9     val v2 = Iface.f()  
10 }
```

# ВАРИАНТ ИСПОЛЬЗОВАНИЯ

- Интерфейс описывает контракт
- Есть несколько реализаций
- В компаньон помещаем factory-метод
- Который по параметрам определяет подходящую реализацию

# СВОЙСТВА В ИНТЕРФЕЙСЕ

- В интерфейсе могут встречаться свойства
- Но их нельзя материализовывать
- Но можно определить get-метод
- Будет как метод по умолчанию



# ПРИМЕР

```
1 interface Base {  
2     val url: String  
3 }  
4  
5 class C: Base {  
6     override val url: String = "https://www.yandex.ru"  
7 }
```

# ПЛОХОЙ ПРИМЕР

```
1 interface Base {  
2     val url: String = "https://www.yandex.ru"  
3     // так нельзя  
4 }  
5  
6 interface Base {  
7     val url: String  
8     get() = field.substring(1) // и так тоже  
9 }
```

# ХОРОШИЙ ПРИМЕР

```
1 interface Base {  
2     val url: String  
3     get() = "https://www.yandex.ru"  
4     // А так - МОЖНО  
5 }  
6  
7  
8 class C1: Base {  
9     override val url = ""  
10 }  
11  
12 class C2: Base {  
13     init {  
14         println(url)  
15     }
```

# НЮАНСЫ

- В определении метода по умолчанию могут быть параметры со значением по умолчанию
- Более того - у метода без тела тоже может быть параметр по умолчанию
- В месте определения параметр надо указать
- Но свое значение по умолчанию указать нельзя

# ПРИМЕР

```
1 interface Base {  
2     fun createUrl(host: String="www.yandex.ru"): String =  
3         "https://${host}"  
4 }  
5  
6 object 0 : Base  
7  
8 fun main() {  
9     println(0.createUrl())  
10 }
```

# ПРИМЕР

```
1 interface Base {  
2     fun createUrl(host: String="www.yandex.ru"): String =  
3         "https://${host}"  
4 }  
5 object 0 : Base {  
6     override fun createUrl(host: String) = "ftp://${host}"  
7 }  
8 fun main() {  
9     println(0.createUrl())  
10 }
```

# РЕАЛИЗАЦИЯ

- Форме без параметра соответствует синтетический статический интерфейсный метод
- Ему передается параметром объект
- Он вызывает над объектом его реализацию метода
- И передает значение по умолчанию

# ДЕТАЛИЗАЦИЯ

- Можно обратиться к конкретному методу по умолчанию
- Указываем `super` и тип в угловых скобках
- Помогает в разрешении неоднозначностей
- Или в вызове "скрытого" метода



# ПРИМЕР

```
1 interface I1 {  
2     fun m() = println("I - 1")  
3 }  
4 interface I2 {  
5     fun m() = println("I - 2")  
6 }  
7 class C: I1, I2 {  
8     override fun m() {  
9         super<I1>.m()  
10        super<I2>.m()  
11    }  
12 }
```

# НА ПОНИМАНИЕ

```
1 interface I1 {  
2     fun m(v: Int = 10) = println("I - 1: $v")  
3 }  
4 interface I2 : I1 {  
5     override fun m(v: Int) = println("I - 2: $v")  
6 }  
7 class C: I1, I2 {  
8     init {  
9         super<I1>.m(1)  
10        super<I2>.m(2)  
11        m()  
12        //super<I1>.m() - так нельзя  
13    }  
14 }
```

# УМОЛЧАНИЯ

- По умолчанию классы финальны
- Методы классов - тоже
- Методы интерфейсов - нет (было бы странно)
- Ключевое слово `open` отменяет финальность

# УМОЛЧАНИЯ

- `override` подразумевает `open`
- Но иногда хочется положить этому конец
- Можно перед `override` указать `final`

# МОДИФИКАТОРЫ ДОСТУПА

- public доступность везде
- По умолчанию все - public
- protected - видимость в наследниках
- Только в наследниках

# МОДИФИКАТОРЫ ДОСТУПА

- Вне класса `protected` нет
- `private`-элемент класса виден только в классе
- `private`-элемент файла - только в файле
- Внешний класс не видит `private`-элементы своих внутренних классов

# JAVA

```
1 class C1 {  
2     class C2 {  
3         private int f;  
4     }  
5  
6     void f(C2 v) {  
7         System.out.println(v.f);  
8     }  
9 }
```

# KOTLIN

```
1 class C {  
2     protected class C1 {  
3         private val q = 5  
4     }  
5  
6     fun f(v: Int) {  
7         v.q  
8     }  
9 }
```



# РЕГУЛЯРНОСТЬ

- Kotlin-модификаторы более регулярны
- Например, можно объявить `private class`
- Который не будет внутренним
- Но это чисто компиляторная история

# INTERNAL

- Обозначает видимость в рамках единицы сборки
- Например, собираем библиотеку
- Класс нужен много где в библиотеке
- Но он служебный

# ВНУТРЕННИЕ КЛАССЫ

- Примерно как в Java
- По умолчанию - статические
- Умолчание меняется ключевым словом `inner`

# KOTLIN

```
1 class Outer {  
2     class Static  
3  
4     inner class Inner {  
5         init {  
6             println(this)  
7             println(this@Inner)  
8             println(this@Outer)  
9         }  
10  
11 // .....
```

# KOTLIN

```
1 // .....
2
3     inner class AnotherInner {
4         init {
5             println(this)
6             println(this@Inner)
7             println(this@Outer)
8             println(this@AnotherInner)
9         }
10    }
11 }
12 // .....
```

# KOTLIN

```
1 // .....
2     init {
3         println(this)
4         println(this@Outer)
5     }
6 }
7
8 fun main() {
9     Outer()
10    println("-----")
11    Outer().Inner()
12    println("+++++")
13    Outer().Inner().AnotherInner()
14 }
```

# SEALED-КЛАССЫ

- Начнем с мотивирующего примера
- Вспомним задание с семинара про вычисление выражений
- Которые в виде дерева
- Допустим, решаем ее через when (на то могут быть причины)

# KOTLIN

```
1 abstract class Expr
2
3 data class Value(val v: Int) : Expr()
4 data class Sum(val e1: Expr, val e2: Expr): Expr()
5
6 fun eval(e: Expr): Int = when (e) {
7     is Value -> e.v
8     is Sum -> eval(e.e1) + eval(e.e2)
9     else -> throw IllegalArgumentException("why ?")
10 }
```



# SEALED-КЛАССЫ

- Хотим убедить компилятор в том, что других подклассов нет
- Это делается с помощью ключевого слова `sealed`
- Все подклассы должны быть в том же файле
- Нужно, чтобы был не анонимный `package`

# УБЕДИЛИ

```
1 sealed class Expr {  
2     class Value(val v: Int) : Expr()  
3     class Sum(val e1: Expr, val e2: Expr) : Expr()  
4 }  
5  
6 fun eval(e: Expr): Int = when (e) {  
7     is Expr.Value -> e.v  
8     is Expr.Sum -> eval(e.e1) + eval(e.e2)  
9 }
```

# КОНСТРУКТОР СУПЕРКЛАССА

- Связь задается при объявлении класса
- Не в описании конструктора
- Страдает single-responsibility
- Меньше вариантов стыковки
- Вторичные связываются с суперклассом через первичного

# ИСКЛЮЧЕНИЯ

- Бросить - throw
- Поймать - try/catch/finally
- Нет понятия 'checked exception'
- try - выражение

# CHECKED EXCEPTION

```
1 class C {
2     public static void m(String s) throws IOException {
3         FileInputStream fis = new FileInputStream(s);
4         ///
5     }
6 }
7
8 class C2 {
9     public static void m2(List<String> lines) {
10         lines.forEach(C::m); // так нельзя
11     }
12 }
```

# CHECKED EXCEPTION

```
1 public static void m2(List<String> lines) {  
2     lines.forEach(e -> {  
3         try { // можно, но хочется покомпактнее  
4             C.m(e);  
5         } catch (IOException exc) {  
6             throw new UncheckedIOException(exc);  
7         }  
8     });  
9 }
```

# CHECKED EXCEPTION

```
1 // Так можно, но толку мало
2 public static Consumer<String> wrap(Consumer<String> f) {
3     return e -> {
4         try {
5             f.accept(e);
6         } catch (Exception exc) {
7             throw new RuntimeException(exc);
8         }
9     };
10 }
```

# CHECKED EXCEPTION

```
1 // Потому что так - все равно нельзя
2
3 public static void m3(List<String> lines) {
4     lines.forEach(wrap(C1::m1));
5 }
```



# USE

- `try-with-resources` в явном виде нет
- Потому что это умножение сущностей
- Есть метод `use`
- Как расширение `Closeable`

# РЕАЛИЗАЦИЯ

```
1 public inline fun <T : Closeable?, R> T.use(  
2     block: (T) -> R  
3 ): R {  
4     contract {  
5         callsInPlace(block, InvocationKind.EXACTLY_ONCE)  
6     }  
7     var exception: Throwable? = null  
8     try {  
9         return block(this)  
10    } catch (e: Throwable) {  
11        exception = e  
12        throw e  
13    } finally {  
14 // .....  
15 }
```

# РЕАЛИЗАЦИЯ

```
1 // .....
2     when {
3         apiVersionIsAtLeast(1, 1, 0) ->
4             this.closeFinally(exception)
5             this == null -> {}
6             exception == null -> close()
7             else -> try {
8                 close()
9             } catch (exc: Throwable) {
10                // cause.addSuppressed(exc) - to keep legacy behaviour
11            }
12     }
13 }
14 }
```

# WITH/APPLY

- Хотим получить строку через StringBuilder
- Можно в лоб завести переменную-билдер
- Что-то с ней поделать
- Вернуть результат
- Kotlin позволяет сделать красивее

# BASELINE

```
1 fun alphabet(): String {  
2     val result = StringBuilder()  
3     for (letter in 'A'..'Z') {  
4         result.append(letter)  
5     }  
6     result.append("\nNow I know the alphabet!")  
7     return result.toString()  
8 }
```

# KOTLIN-STYLE

```
1 fun alphabet() = with(StringBuilder()) {  
2     for (letter in 'A'..'Z') {  
3         append(letter)  
4     }  
5     append("\nNow I know the alphabet!")  
6     this.toString()  
7 }
```

# WITH ВНУТРИ

```
1 public inline fun <T, R> with(  
2     receiver: T, block: T.() -> R  
3 ): R {  
4     contract {  
5         callsInPlace(block, InvocationKind.EXACTLY_ONCE)  
6     }  
7     return receiver.block()  
8 }
```

# РАЗБЕРЕМ

- $T.() \rightarrow R$  - "lambda with receiver"
- Формально - анонимная функция, в которой можно использовать `this`
- Явно или неявно
- И которая в конечном итоге вызывается как расширение



# ПРИМЕР

```
1 fun main() {  
2     val f: String.(Int) -> Int = {  
3         it + length  
4     }  
5  
6     println("hello".f(1))  
7 }
```

# ЭКВАВАЛЕНТЫ ПО РЕАЛИЗАЦИИ

```
1 fun main() {  
2     val f1: (Pair<String, Int>) -> Int = {  
3         it.second + it.first.length  
4     }  
5  
6     println(f1(Pair("hello", 1)))  
7 }
```

# ЭКВАВАЛЕНТЫ ПО РЕАЛИЗАЦИИ

```
1 fun main() {  
2     val f1: (String, Int) -> Int = { s, n ->  
3         n + s.length  
4     }  
5  
6     println(f1("hello", 1))  
7 }
```

# В ЧЕМ ПОЛЬЗА

- Помимо сахара
- Разделяем монолит на переменную и действия
- Можем комбинировать и выбирать независимо
- Можно преобразовывать действия
- Способствует DSL-стилю

# APPLY

- Близкий аналог with
- Метод, а не функция
- Возвращает this
- Вариант использования: быстрая отладка

