Kotlin

Advanced

Александр Попов

Generic-типы

- позволяют использовать единую логику для разных типов объектов
- могут определяться для всего класса, для конкретной функции или поля
- стирание типов (type erasure) – информация о generic-типе не хранится в скомпилированном коде: данные хранятся в виде Any (Object), в местах использования – приведение к нужному типу

  исключение: inline-функции с reified-параметром
- <*> - может быть использован любой тип (тип неизвестен или неважен)

```
class Cage<out T : Animal>(val animal: T)
```

```
public inline fun <T> Iterable<T>.filter(predicate: (T) → Boolean): List<T> {
    return filterTo(ArrayList<T>(), predicate)
}
```

Вариантность

Через вариантность мы выставляем отношения между разными типа дженериков имеющие иерархию наследования

- **Инвариантность** - типы не связаны, по умолчанию дженерики инварианты даже если у них есть четкая иерархия

- **Ковариантность (out)** - класс производит экземпляры типа T, но не потребляет их. *Например, собака это животное Animal -> Dog*

- **Контрвариантность (in)** - может только потребляться, но не может производиться *Пример. Сыр это еда, но сыроеду нельзя давать другую еду Food <- Cheese*

```
class Cage<out T : Animal>(val animal: T)


class Eater<in T : Food>() {
    fun consume(food: T) {
    }
}
```

Лямбда-выражения

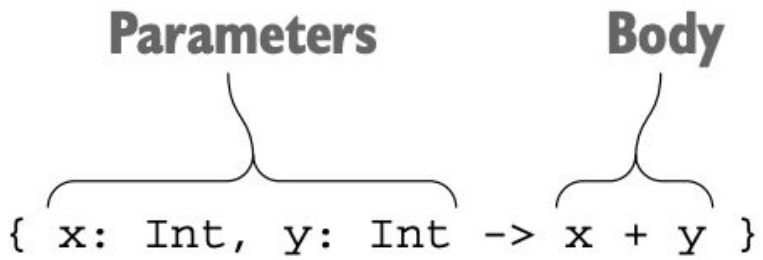
1. Отложенное исполнение операций

```
button.setOnClickListener {
    /* do on click */
}
```

2. Переиспользование операций

```
val largestNumber = listOf(10, 40, 5).maxBy { it }
val lastWord = listOf("Computer", "Code").maxBy { it }
```

Синтаксис лямбда-выражений




```
val oldest = people.maxBy { human: Human ->
    human.age
}
```

```
val oldest = people.maxBy { human ->
    human.age
}
```

```
val oldest = people.maxBy { it.age }
```

Функции высшего порядка

- Функция высшего порядка - это функция, которая принимает функции как параметры, или возвращает функцию в качестве результата.

```
public inline fun <T> Iterable<T>.filter(predicate: (T) → Boolean): List<T> {
    return filterTo(ArrayList<T>(), predicate)
}
```

inline функции

- Функции помеченный ключевым словом **inline** на этапе компиляции встраивается в место вызова. Это позволяет экономить ресурсы если в функции есть лямбды, так как не тратится память на их создание
- **noinline** - помечаем те те функции-параметры, которые встроены не будут
- **crossinline** - запрет использования return в лямбде. Так как код встраивается в место вызова return из лямбды просто завершит функцию
- **reified** - можно смотреть тип дженериков, так как код встраивается в место вызова и мы точно знаем тип

```
public inline fun <T> Iterable<T>.filter(predicate: (T) → Boolean): List<T> {
    return filterTo(ArrayList<T>(), predicate)
}
```

Функции infix

Такие функция могут вызываться с использованием инфиксной записи (без точки и скобок для вызова)

Инфиксные функции должны соответствовать следующим требованиям:

- Они должны являться членом другой функции или расширения;
- В них должен использоваться только один параметр;
- Параметр не должен принимать переменное количество аргументов и не должен иметь значения по умолчанию.

```
fun main() {
    val acc = Account( sum: 1000)
    acc put 150
    // равноценно вызову
    acc.put( amount: 150)
    acc.printSum()  // 1300
}
new *
class Account(private var sum: Int) {
    new *
    infix fun put(amount: Int){
        sum += amount
    }
    new *
    fun printSum() = println(sum)
}
```

Extension-функции

- Позволяют добавлять функциональность в классы и интерфейсы, не модифицируя их
- Компилируются в обычные функции с добавленным параметром – объектом расширяемого класса

```
public interface CoroutineScope {
    The context of this scope. Context is encapsulated by the scope and used for implementation of
    coroutine builders that are extensions on the scope. Accessing this property in general code is not
    recommended for any purposes except accessing the Job instance for advanced usages.

    By convention, should contain an instance of a job to enforce structured concurrency.

    public val coroutineContext: CoroutineContext
}
```

```
public fun CoroutineScope.ensureActive(): Unit = coroutineContext.ensureActive()
```

```
@Suppress( ...names: "FunctionName")
public fun CoroutineScope(context: CoroutineContext): CoroutineScope =
    ContextScope(if (context[Job] != null) context else context + Job())
```

```
public val CoroutineScope.isActive: Boolean
    get() = coroutineContext[Job]?.isActive ?: true
```

Делегаты

- Делегаты классов – позволяют делегировать часть реализации другим объектам
- Делегаты свойств – позволяют делегировать логику записи и/или чтения свойства другому классу

```
class Delegated(b: Base) : Base by b

        val lazyValue: String by lazy {
            greetMark()
        }
```

Делегаты свойств

- **lazy** – создает объект, который запускает указанную лямбду при первом вызове get() и затем возвращает сохраненное значение при последующих вызовах.
- **observable** – принимает исходное значение свойства и функцию, которая выполняется при изменении значения.
- **vetoable** – похоже на observable. С помощью vetoable можно узнать, когда значение меняется и тем самым, например добавить проверку перед присваиванием значения
- **notNull** – Похоже на lateinit. Оно позволяет объявить свойство без начального значения. Если попытаться прочитать значение до того, как присвоили значение – будет брошено исключение

```
val lazyValue: String by lazy {
    greetMark()
}
```

```
var observable by Delegates.observable( initialValue: 1) { prop, oldVal, newVal →
    println("Observable property changed from $oldVal to $newVal")
}
```

```
var vetoable by Delegates.vetoable( initialValue: 1) { prop, oldVal, newVal →
    return@vetoable newVal ⩽ 10
}
```

```
var notNull: Int by Delegates.notNull()
```

Рефлексия

- Позволяет динамически во время исполнения получить доступ к свойствам, методам и т.д.

```kotlin
class ClassToReflect {
    val timeCreated = System.currentTimeMillis()

    fun callMe(): String {
        return "Thank you for calling"
    }
}


fun main() {
    val obj = ClassToReflect()
    val kClass = obj.javaClass.kotlin
    println(kClass.members.joinToString { it.name })

    val prop = kClass.memberProperties.find { it.name == "timeCreated" }
    val method = kClass.memberFunctions.find { it.name == "callMe" }
    println("Reflected: timeCreated=${prop?.get(obj)} callMe=${method?.call(obj)}")
}
```

Scope функции

- **let** - часто используется для безопасного выполнения блока кода с null-выражениями
- **also** - используется для выполнения каких-либо дополнительных действий
- **apply** - настроить объект и не надо возвращать результат
- **run** - используется для настройки объекта и вычисления результата
- **with** - используется для объединения вызовов функций объекта

Scope функции

	Функция будет принимать this	Функция будет принимать it
Будет возвращен объект на котором вызывается функция (self)	**apply**	**also**
Будет возвращен результат функции (result)	**run, with**	**let**

Разница между this и it: this может быть опущено, а it в явном виде заменено на другое имя переменной

Виды классов: data class

- используется для классов, основная функция которых – «обёртка» над данными
- обязательно должен содержать поля, определённые в конструкторе
- при компиляции генерируется несколько методов, среди которых equals, hashCode, toString, copy
- нельзя наследоваться от data class

```
data class Person(val name: String, val age: Int)
```

Виды классов: object

- singleton из коробки
- к «обычному» классу можно добавить companion object – сымитировать (*Java*-) статические члены
- в объектах поля могут быть помечены как const

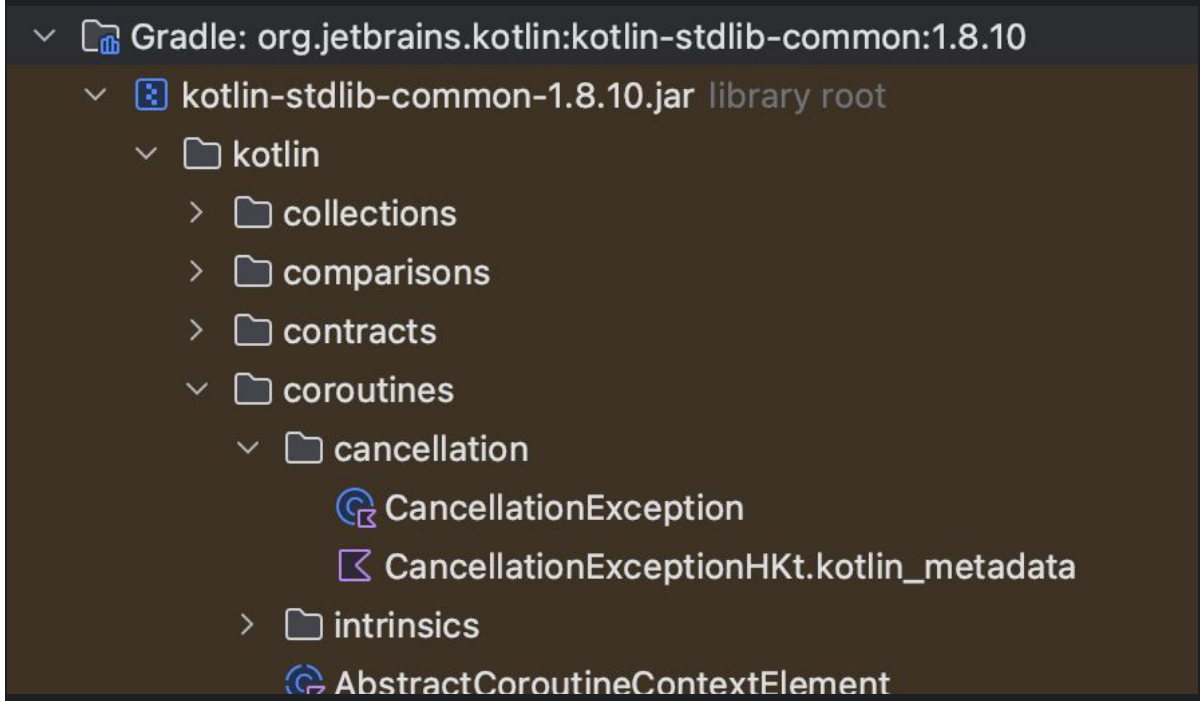
Пакеты

- пакеты (packages) позволяют группировать файлы исходного кода
- пакет файла может не совпадать с путём к файлу
- можно определять функции и поля на уровне пакета
- с помощью import kotlin.coroutines.cancellation.* можно импортировать все классы пакета

```
package kotlin.coroutines.cancellation

Thrown by cancellable suspending functions if the coroutine is cancelled while it is suspended.
It indicates normal cancellation of a coroutine.

@SinceKotlin( version: "1.4")
public expect open class CancellationException : IllegalStateException {
    public constructor()
    public constructor(message: String?)
}
```



Аннотации

- Добавляют «мета»-информацию к коду
- Информация может использоваться на уровне исходного кода, при компиляции или во время исполнения

```
@Deprecated( message: "Will be removed soon")
class Rectangle {
    var width: Int = 0
    var height: Int = 0
```

```
@RunWith(AndroidJUnit4::class)
class ExampleInstrumentedTest {
    @Test
    fun useAppContext() {
        // Context of the app under test.
        val appContext = InstrumentationRegistry.getInstrumentation().targetContext
        assertEquals( expected: "com.pa_vel.kotlin", appContext.packageName)
    }
}
```

```
@Suppress( ...names: "EXTENSION_SHADOWED_BY_MEMBER")
public val CoroutineScope.isActive: Boolean
    get() = coroutineContext[Job]?.isActive ?: true
```

```
@Composable
fun Greeting(name: String) {
    Text("Hello $name")
}
```

Спасибо
за внимание!