Объектно-ориентированное программирование

Допустим, мы хотим уметь хранить вместе связанные данные. Например, точку — две координаты и имя. Ну, *как-то* это сделать мы умеем:

```
listOf(1.0, 2.0, "A")
```

```
=> : List<Comparable<*>> = [1.0, 2.0, A]
```

И теперь при чтении элементов, если мы абсолютно уверены, какого они типа, используем знакомый нам оператор `as`:

```
val arr = listOf(1.0, 2.0, "A")
arr[2] as String
```

```
=> : String = "A"
```

Во-первых, компилятор нам надоест предупреждениями об unchecked cast. Во-вторых, оператор `as` на самом деле занимает довольно много времени, так как неявно ещё и производит дорогую проверку типов. В-третьих, мы сами рискуем быстро забыть, кто на ком стоял, в каком элементе хранится что и какого типа.

Можно решить проблему с типами с помощью специальных конструкций языка: `Pair` и `Triple`.

```
Triple(1.0, 2.0, "A")
```

```
=> : Triple<Double, Double, String> = (1.0, 2.0, A)
```

Теперь мы можем обращаться к элементам по именам first, second, third:

```
Triple(1.0, 2.0, "A").third
```

```
=> : String = "A"
```

Для создания пары есть инфиксная функция to:

```
1 to "PTHS"
```

```
=> : Pair<Int, String> = (1, PTHS)
```

Элементы, соответственно, first и second. Вроде всё замечательно?

> Предположим, вы ведёте курс в ФТШ и хотите, чтобы код в ваших конспектах был корректно подсвечен. Вы пишете программу, разбирающую код и решающую, что и как подсветить.
>
> Из чего состоит, скажем, объявление функции?
> - Перечень модификаторов
> - Имя
> - Список параметров
> - Возможно, явное объявление возвращаемого типа
> - Тело функции
>
> Это пять элементов. Мы так плохо умеем, но ведь пары и тройки можно вкладывать друг в друга.
>
> `Triple<modifiers, name, Triple<arguments, return type?, function body>>`

> Модификаторы — это строки, имя — строка. Возвращаемый тип — для простоты допустим, тоже, тело функции — перечень выражений. Аргументы — список пар (имя, тип)
>
> `Triple<List<String>, String, Triple<List<Pair<String, String>>, String?, List<statement>>`
>
> И это ещё не всё, statement-то это тоже нечто непростое. А лично я уже запутался.

Давайте объявим новый тип данных.

```
data class Point(val x: Double, val y: Double, val name: String)
```

`data class` — собственно, объявление (чем `data class` отличается от просто `class` — в конце лекции). Дальше идёт объявление *полей* — постоянная (`val`) или переменная (`var`), затем имя и тип.

```
Point(1.0, 2.0, "A")
```

```
=> : Point =  Point(x=1.0, y=2.0, name=A)
```

Строковое представление тут тоже немного хромает, но читаемо. Да и не это обычно важно, и это мы ещё научимся менять. Что важно, теперь у нас есть именованные и типизированные элементы, а ещё лаконичеое именование типа.

```
val point = Point(1.0, 2.0, "A")
point.y
```

```
=> : Double = 2.0
```

Давайте попробуем написать трёхмерный вектор.

```
data class Vector(val x: Double, val y: Double, val z: Double)
```

Отлично. Теперь добавим пару функций для работы с ним.

```
fun add(a: Vector, b: Vector) = Vector(a.x + b.x, a.y + b.y, a.z + b.z)

fun sub(a: Vector, b: Vector) = Vector(a.x - b.x, a.y - b.y, a.z - b.z)

fun multiply(a: Vector, k: Double) = Vector(a.x * k, a.y * k, a.z * k)

fun scalar(a: Vector, b: Vector) = a.x * b.x + a.y * b.y + a.z * b.z

fun magnitude(a: Vector) = sqrt(scalar(a, a))
```

Проверим теперь, что работает:

```
val a = Vector(5.0, 6.0, 6.0)
val b = Vector(30.0, 30.0, 30.0)
val c = Vector(2.0, 3.0, 9.0)
add(a, b)
```

```
=> : Vector =  Vector(x=35.0, y=36.0, z=36.0)
```

```
sub(a, c)
```

```
=> : Vector =  Vector(x=3.0, y=3.0, z=-3.0)
```

```
multiply(a, 3.0)
```

```
=> : Vector =  Vector(x=15.0, y=18.0, z=18.0)
```

Вроде всё хорошо. Однако теперь стоит заметить, что в глобальном поле может оказаться очень много функций с одинаковыми названиями. Например, мы захотим складывать вектора, матрицы, комплексные числа и прочие штуки. Так-то это не проблема, если типы аргументов известны в момент компиляции, то выберется правильная версия:

```
fun process(x: Int) {
    println("Int version for " + x)
}
fun process(x: Double) {
    println("Double version for " + x)
}

process(1)
process(2.0)
```

```
=> : Unit = Unit
```

```
Int version for 1
Double version for 2.0
```

Тип аргументов, как и название функции, входит в *сигнатуру функции*, то есть, в некотором роде, её уникальное описание. Возвращаемый тип — не входит, хотя есть определённые способы обойти это ограничение, можно посмотреть лекцию про generics, раздел reified.

Но вот загрязнение глобального пространства имён — не дело. Давайте поместим эти функции в отдельный namespace.

Для этого делаем следующее:

```
data class Vector(val x: Double, val y: Double, val z: Double) {
    companion object {
        fun add(a: Vector, b: Vector) = Vector(a.x + b.x, a.y + b.y, a.z + b.z)

        fun sub(a: Vector, b: Vector) = Vector(a.x - b.x, a.y - b.y, a.z - b.z)

        fun multiply(a: Vector, k: Double) = Vector(a.x * k, a.y * k, a.z * k)

        fun scalar(a: Vector, b: Vector) = a.x * b.x + a.y * b.y + a.z * b.z

        fun magnitude(a: Vector) = sqrt(scalar(a, a))
    }
}
```

Теперь, чтобы обратиться к этим функциям, нужно их явно квалифицировать:

```
Vector.add(Vector(5.0, 6.0, 6.0), Vector(2.0, 3.0, 9.0))
```

```
=> : Vector =  Vector(x=7.0, y=9.0, z=15.0)
```

Технически, теперь `Vector.add` — это и есть полное имя функции.

Да, давайте добавим ещё пару интересных функций...

```
data class Vector(val x: Double, val y: Double, val z: Double) {
    companion object {
        fun add(a: Vector, b: Vector) = Vector(a.x + b.x, a.y + b.y, a.z + b.z)

        fun sub(a: Vector, b: Vector) = Vector(a.x - b.x, a.y - b.y, a.z - b.z)

        fun multiply(a: Vector, k: Double) = Vector(a.x * k, a.y * k, a.z * k)

        fun scalar(a: Vector, b: Vector) = a.x * b.x + a.y * b.y + a.z * b.z

        fun magnitude(a: Vector) = sqrt(scalar(a, a))

        fun ofList(list: List<Double>) = Vector(list[0], list[1], list[2])

        fun ofNumbers(x: Number, y: Number, z: Number) : Vector {
            return Vector(x.toDouble(), y.toDouble(), z.toDouble())
        }
    }
}
```

Честно говоря, можно заметить, что часто функции, которые хотелось бы положить в `companion object` класса T, либо принимают T одним из аргументов, либо возвращают T. Более того, на практике часто приходится писать очень много вложенных функций, как то, например:

```
joinToString(
    takeWhile(
        map(
            filter(
                0 until 100,
                { it % 6 == 0}
            ),
            { it * it }
        ),
        { it <= 1000}
    )
)
```

(Примечание: код выше является иллюстрацией, в текущих версиях Котлина таких функций в стандартной библиотеке нет).

Что делает этот код?

- Берёт список от нуля до ста
- Фильтрует его, оставляя только числа, кратные шести
- Возводит каждое число в квадрат
- Оставляет только те, которые меньше 1000
- Соединяет всё в строку

```
=> : String = "0, 36, 144, 324, 576, 900"
```

Всё бы ничего, но выглядит этот код... плохо. Читать снизу вверх, справа налево. Поэтому существует возможность объявить функцию как условно-постфиксную (помните, в предыдущей лекции об этом было?).

И код внезапно обретёт смысл не только на бумаге, но и визуально:

```
(0 until 100)
    .filter({ it % 6 == 0 })
    .map({ it * it })
    .takeWhile({ it <= 1000 })
    .joinToString()
```

```
=> : String = "0, 36, 144, 324, 576, 900"
```

И как это сделать? Объявить функцию в теле класса, но не в `companion object`. Тогда это будет *метод*, который можно *вызвать на* объекте этого класса. Объект, на котором вызывается метод (слева от точки) передаётся функции в неявный аргумент с именем `this`.

```
data class Vector(val x: Double, val y: Double, val z: Double) {
    companion object {
        fun ofList(list: List<Double>) = Vector(list[0], list[1], list[2])

        fun ofNumbers(x: Number, y: Number, z: Number) : Vector {
            return Vector(x.toDouble(), y.toDouble(), z.toDouble())
        }
    }

    fun add(that: Vector) = Vector(this.x + that.x, this.y + that.y, this.z + that.z)

    fun sub(that: Vector) = Vector(this.x - that.x, this.y - that.y, this.z - that.z)

    fun multiply(k: Double) = Vector(this.x * k, this.y * k, this.z * k)

    fun scalar(that: Vector) = this.x * that.x + this.y * that.y + this.z * that.z

    fun magnitude() = sqrt(this.scalar(this))
}
```

Если есть второй (на самом деле первый) аргумент того же типа, то его часто называют `that` или `another`. Проверим, что работает:

```
val a = Vector.ofNumbers(5, 6, 6)
val b = Vector.ofNumbers(2, 3, 9)
a.add(b)
```

```
=> : Vector =  Vector(x=7.0, y=9.0, z=15.0)
```

```
a.magnitude()
```

```
=> : Double = 9.848857801796104
```

Замечание первое: обращаться к `this` явным образом не всегда обязательно. Если просто написать идентификатор, то в первую очередь компилятор поищет локальную переменную с таким именем, потом аргумент функции, и наконец, поле `this`. Если вы не занимаетесь странным неймингом, называя одним и тем же именем всё, что попало, то всё у вас будет хорошо. Замечание второе: чтобы объявить функцию инфиксной, достаточно дописать к ней модификатор `infix` (работает только с методами, имеющими ровно один явный аргумент). Для того, чтобы код лучше читался, поменяем имена немного (add, sub, multiply на plus, minus, times) Замечание третье: чтобы обеспечить, наконец, нормальное строковое представление нашему классу, определим в нём специальный метод toString.

Итого получаем:

```
data class Vector(val x: Double, val y: Double, val z: Double) {
    companion object {
        fun ofList(list: List<Double>) = Vector(list[0], list[1], list[2])

        fun ofNumbers(x: Number, y: Number, z: Number): Vector {
            return Vector(x.toDouble(), y.toDouble(), z.toDouble())
        }
    }

    infix fun plus(that: Vector) = Vector(x + that.x, y + that.y, z + that.z)

    infix fun minus(that: Vector) = Vector(x - that.x, y - that.y, z - that.z)

    infix fun times(k: Double) = Vector(x * k, y * k, z * k)

    infix fun dot(that: Vector) = x * that.x + y * that.y + z * that.z

    fun magnitude() = sqrt(this dot this)

    override fun toString(): String = "($x, $y, $z)"
}
```

Два вопроса. Что такое `override` и что за доллары в строке? Первое — узнаете в следующей лекции. Пока так надо, поверьте. Второе — способ автоматически встроить строковое представление переменной в строку. Ну, то же самое выглядело бы в какой-нибудь Java как `"(" + x + ", " + y + ", " + z + ")"`. *Не очень* лаконично.

Отлично, давайте попробуем:

```
val a = Vector.ofNumbers(5, 6, 6)
val b = Vector.ofNumbers(2, 3, 9)
a plus b
```

```
=> : Vector = (7.0, 9.0, 15.0)
```

А что нам выдаст следующий код?

```
a plus b times 3.0
```

На самом деле, очевидно, что компилятор не может догадываться о том, какие приоритеты мы закладываем в инфиксные методы, так что он прочитает это слева направо:

```
(a plus b) times 3.0
```

```
=> : Vector = (21.0, 27.0, 45.0)
```

... Хочется перегрузки операторов. И да, я не просто так назвал методы `plus`, `minus`, `times`. Это методы, которые отвечают за перегрузку соответствующих операторов — сложения, вычитания, умножения. Надо лишь добавить им модификатор `operator`.

```
data class Vector(val x: Double, val y: Double, val z: Double) {
      companion object {
          fun ofList(list: List<Double>) = Vector(list[0], list[1], list[2])

          fun ofNumbers(x: Number, y: Number, z: Number): Vector {
              return Vector(x.toDouble(), y.toDouble(), z.toDouble())
          }
      }

      operator fun plus(that: Vector) = Vector(x + that.x, y + that.y, z + that.z)

      operator fun minus(that: Vector) = Vector(x - that.x, y - that.y, z - that.z)

      operator fun times(k: Double) = Vector(x * k, y * k, z * k)

      infix fun dot(that: Vector) = x * that.x + y * that.y + z * that.z

      fun magnitude() = sqrt(this dot this)

      override fun toString(): String = "($x, $y, $z)"
  }
```

Никто, конечно, не запрещает оставить `infix` на операторном методе. Но зачем?

Снова проверяем:

```
val a = Vector.ofNumbers(5, 6, 6)
val b = Vector.ofNumbers(2, 3, 9)
a + b
```

```
=> : Vector = (7.0, 9.0, 15.0)
```

```
a + b * 3.0
```

```
=> : Vector = (11.0, 15.0, 33.0)
```

Вот тут уже результат другой, потому что у операторов `+` и `*` вполне понятные приоритеты. Полный перечень операторов и методов, им соответствующих, можно найти в документации. Говоря коротко, есть

- `plus`, `minus`, `times`, `div`, `rem`, делающих понятно что,
- `invoke` — оператор вызова, `()`,
- `get` и `set` — доступа по индексу, `[]`,
- `compareTo` — сравнение, `<`, `>`, `>=`, `<=`,
- `equals` — сравнение на равенство, `==` и `!=` (чаще всего вы *очень* хотите, чтобы он был согласован с `compareTo`)
- `contains` — проверка на включение, `in` и `!in`
- `rangeTo` и `rangeUntil` — промежутки, `..` и `..<` соответственно.
- Есть ещё операторы для `+=`, `-=` и проч., а также для `++` и `--`, но их надо реализовывать с большой осторожностью — читайте документацию.
- И операторы `componentN`, о которых поговорим позже.

А можно ли добавить метод к классу, который определяем не мы? Например, к чему-то из стандартной библиотеки?

Очень хочется, особенно операторы. А то что за дела, `"1" + 2` работает, а `1 + "2"` — нет?

Можно. Для этого нужно всего лишь написать так называемый receiver type — тип, для которого этот метод объявляется. Ну, скажем,

```
operator fun Int.plus(str: String) = this.toString() + str
```

Вот то, что слева от точки — это и есть receiver type. Он, кстати, вполне может быть nullable — тогда при вызове этого метода необязательно писать safe call. Но вам придётся разбираться с тем, что `this` может оказаться `null`. Например, есть встроенная перегрузка toString для nullable типов:

```
inline fun Any?.toString() = this?.toString() ?: "null"
```

У нас в `companion object` всё ещё остались методы. Это, конечно, не плохо, но... помните, я говорил?..

...часто функции, которые хотелось бы положить в `companion object` класса T, либо принимают T одним из аргументов, либо возвращают T...

С первыми мы разобрались, теперь со вторыми. По сути это — создание нового объекта, такое же, как `Vector(1.0, 2.0, 3.0)`. Давайте сделаем, чтобы можно было так же. Для этого есть ключевое слово `constructor` и абьюз ключевого слова `this`.

```
data class Vector(val x: Double, val y: Double, val z: Double) {
    constructor(list: List<Double>) : this(list[0], list[1], list[2])

    constructor(x: Number, y: Number, z: Number)
            : this(x.toDouble(), y.toDouble(), z.toDouble())

    constructor() : this(0.0, 0.0, 0.0)

    operator fun plus(that: Vector) = Vector(x + that.x, y + that.y, z + that.z)
    operator fun minus(that: Vector) = Vector(x - that.x, y - that.y, z - that.z)
    operator fun times(k: Double) = Vector(x * k, y * k, z * k)
    infix fun dot(that: Vector) = x * that.x + y * that.y + z * that.z
    fun magnitude() = sqrt(this dot this)
    override fun toString(): String = "($x, $y, $z)"
}
```

Заодно создали конструктор без аргументов, создающий нулевой вектор.

Стоит заметить, что такой трюк удастся только если все необходимые параметры можно вычислить одним выражением. В противном случае придётся всё-таки писать отдельную функцию.

```
Vector()
```

```
=> : Vector = (0.0, 0.0, 0.0)
```

```
Vector(566L, 0xEF.toByte(), 3.0)
```

```
=> : Vector = (566.0, 239.0, 3.0)
```

И последний вопрос. Что же значит `data` в объявлении класса? На самом деле, это означает, что за нас сгенерируют кучу методов. Давайте попробуем поделать очевидные штуки:

```
Vector(566, 566, 566) === Vector(566, 566, 566)
```

```
=> : Boolean = true
```

Очевидно? *Нет.*

Давайте попробуем без слова `data`.

```
class Vector(val x: Double, val y: Double, val z: Double)

Vector(566.0, 566.0, 566.0) === Vector(566.0, 566.0, 566.0)
=> : Boolean = false

Vector(1, 2, 3)
=> : Vector = Vector@7a81197d
```

Почему так? Чтобы понять это, придётся пройти несколько этапов. Написать `data class`, скомпилировать его, декомпилировать в Java и сконвертировать в Kotlin.

А теперь

```
data class Vector(val x: Double, val y: Double, val z: Double)
```

Посмотрим на результат компиляции... 261 строка байткода. Против 84, если бы мы не объявляли его `data`. Я не хочу это читать, ладно? Давайте сразу декомпилируем в Java (92 строки, уже лучше) и сконвертируем в Kotlin.

```
class Vector(val x: Double, val y: Double, val z: Double) {
    operator fun component1(): Double = x

    operator fun component2(): Double = y

    operator fun component3(): Double = z

    fun copy(x: Double, y: Double, z: Double): Vector {
        return Vector(x, y, z)
    }

    override fun toString(): String = "Vector(x=$x, y=$y, z=$z)"

    override fun hashCode(): Int {
        return (java.lang.Double.hashCode(x) * 31 + java.lang.Double.hashCode(y)) *
31 + java.lang.Double.hashCode(z)
    }

    override fun equals(var1: Any?): Boolean {
        return if (this !== var1) {
            if (var1 is Vector) {
                val var2 = var1
                if (java.lang.Double.compare(x, var2.x) == 0 &&
                        java.lang.Double.compare(y, var2.y) == 0 &&
                        java.lang.Double.compare(z, var2.z) == 0
                ) {
                    return true
                }
            }
            false
        } else {
            true
        }
    }
}
```

Ох. С чего бы начать.

В общем, за нас сгенерировали equals, hashCode, toString, copy и ещё на закуску — `copy$default`, который не помещается на полях этого конспекта. Всего этого бойлерплейта можно избежать одним ключевым словом — разве не здорово?

А есть ещё методы `component1`, `component2` и `component3` (нумерация здесь, что забавно, с единицы). Это методы, отвечающие за *деструктивное присваивание*. Что означает, что вы можете делать так:

```
val v = Vector(1.0, 2.0, 3.0)
val (x, y, z) = v
y
```

```
=> : Double = 2.0
```

И это то же самое, что написать:

```
val v = Vector(1.0, 2.0, 3.0)
val x = v.component1(),
val y = v.component2()
val z = v.component3()
```