Daria Shutina, CS, 2^{nd} year

Kotlin

- Containers
  - Красивая схемка (taxonomy of collections)
  - interface Iterable
  - Mutable Collection != Mutable Variable
- Set
- Map
- Array
- Sequence

22-10-14

- Functional programming
  - map
  - filter
  - fold
  - foldRight
  - lambda
  - Lazy vs Deferred computations
  - sealed class
  - enum class

22-10-14

- Компиляция
- Компоненты JVM
  - 1. Classloading services
  - 2. Memory management
  - 3. Interpreter $\leftrightarrow$ JIT-compiler
- Можно ли оценить время компиляции кода?

22-10-21

- А теперь про котлин
  - Компилятор котлина
    - 1. Parser
    - 2. Front-end
      - Resolve
      - Type inference
      - Diagnostics
    - 3. Back-end
- Плагины \todo

22-11-11

- Parallelism & Concurrency
- Preemptive & Cooperative sheduling
- JVM потоки
- Java packages
- Как создать поток и как им управлять
- Подробнее про kotlin.concurrent.thread
  - Thread.state & isAlive

22-11-18

- Lock
- Condition variables
- Synchronized Statement
- ReadWriteLock
- JMM
  - Weak behaviors
  - Volatile fields
  - Happens-before relation
    - Когда возникает `sw` отноошение?
- data-race
- Atomics

Atomic Field Updater
Клёвая библиотечка для котлина

Organization stuff

Help!!!! https://kotlinlang.org/docs/getting-started.html

22-08-09

print vs println

```
fun main(args: Array<String>) {
    print("Hello")
    println(", world!")
}
```

➔ `main` function accepts a variable number of String arguments that can be omitted.

➔ print prints its argument to the standard output

➔ println prints its arguments and adds a line break.

vairables

```
val a: Int = 1   // immediate assignment
var b = 2        // `Int` type is inferred
b = a            // Reassigning to `var` is okay
val c: Int       // Type required when no initializer is provided
c = 3            // Deferred assignment
a = 4            // Error: Val cannot be reassigned
const val NAME_CNST = "Kotlin"  // const-time compile
```

➔ **var** - mutable

➔ **val** - immutable

➔ **const val** - compile-time const value

➔ Type can be inferred in most cases

➔ for `const val` use uppercase for naming

functions

```
fun sum(a: Int, b: Int): Int {
    return a + b;
}

fum mul(a: Int = 1, b: Int) = a * b;
```

Значения по умолчанию можно писать в любом порядке.

```
fun max_val1(a: Int, b: Int) {
    if (a > b) {
        return a
    } else {
        return b
    }
}


fun max_val2(a: Int, b: Int) =
return {
    if (a > b) {
        a
    } else {
        b
    }
}

fun max_val3 (a:Int, b:Int) = if (a > b) a b
```

Разная реализация одного и того же. Возвращаемые значения должны быть одинаковые.

when-expression

Аналог `case` в плюсах.

else -- как `default` в плюсах.

```
when (x) {
    1 -> print("x = 1")
    2 -> print("x = 2")
    else -> {
        print("aboba")
    }
}

when (val input = parseInput()) {
 is String -> print("A string value was passed")
 is Int, is Double -> print("An number value was passed")
}
```

&& vs AND /TODO

loops in lists

```
val items = listOf("a", "b", "c")  // это О, а не ноль :)

for (item in items) {
    println(item)
}

for (index in items.indices) {
    println("item at $index is ${items[index]}")
}

for ((index, item) in items.withIndex()) {
 println("item at $index is $item")
}
```

while-cycle

```
val items = listOf("apple", "banana", "kiwifruit")

var index = 0
while (index < items.size) {
 println("item at $index is ${items[index]}")
 index-+
}

var isComplete: Boolean
do {
 --.
 toComplete = --.
} while(toComplete)
```

labels

Даем название циклу, чтобы можно было к нему обратиться

```
myLabel@ for (intem in items) {
    for (item2 in otheritems) {
        if (...) break@myLabel
        else continue@myLabel
        ...
    }
}
```

ranges

```
val x = 10;
if (x in 1..10) {
    println("x is between 1 and 10")
}

for (x in 1..5) {...}

for (x in 9 downTo 0 step 3) {...}
```

null safety

Нельзя давать значение `null`, если явно не указано, что можно хранить `null`

```
val CanBeNull: String?
val CantBeNull: String
```

elvis-operator (?:)

Проверяет, вдруг значение `null`, и тогда что-нибудь делает

```
fun check(id: String): String? {
    val item = findItem(id) ?: reutrn null
    return id
}
```

safe calls

something?.otherthing не кидает исключений, если `something` окажется `null`

```
employee.department?.head?.name?.let{println(it)} ?: println("smth is null")
```

`it` -- зарезервированнаяя переменная в котлине

unsafe calls

something!!.otherthing кидает исключение, если `something` окажется `null`

```
for (employee: Employee) {
    println(employee.department!!.head!!.name!!)
}
```

TODO

Если есть какая-то функция, которая не до конца реализована и пока не используется, то можно написать `TODO`, и программа скомпилируется

```
fun findItem(id: String): Item? = TODO("Find item $id")
```

lambda-выражения

```
1 val sum: (Int, Int) -> Int = {x, y -> return x + y}
```

22-09-16

ООП

Объектно-ориентированное программирование -- об объектах и их взаимодействиях друг с другом.

Основы ООП:

- инварианты
    - не должны быть публичными
    - объект ответственнен за соблюдение инвариантов
    - если поле не задается в инварианте, то, возможно, оно не нужно и интерфейс программы плохо реализован
- Абстракция ⇔ виртуальное наследование
    - можно наследоваться только от одного абстрактного класса
    - можно наследоваться от нескольких интерфейсов
- Инкапсуляция -- про область видимости переменных и объектов
    - private -- видны только внутри класса (class)
    - protected -- видны в том числе наследникам
    - public -- видны всем
    - internal -- видны только внутри модуля
- Наследование
    - Если в классе метод объявлен как `private`, наследник не сможет этот метод изменить
- Полиморфизм -- несколько наследников у класса `Based`, и функции будет передаваться просто `Based` как аргумент

Наследование

Конструкторы

```
open class Point(val x: Int, val y: Int, private z: String) {
    constructor(other: Point) : this(other.x, other.y) {...}
}
```

Чтобы сделать класс возможным для наследования, нужно написать `open`. Абстрактные классы по умолчанию `open`, остальные по умолчанию `final`

Operator overload

```
class Ex {
    operator fun plus(other Ex) {...}
    operator fun dec() {...}
}
```

ComponentN operator

```

```

Infix functions

```
data class Person(val name: String)

infix fun String.with(other: String) = Person(this, other)

fun main() {
    val readHero = "Rayan" with "Gosling"
    val (real, bean) = readHero
}
```

22-09-23

- ошибки времени разработки (генерируются IDE)

- ошибки компиляции (генерируются компилятором)
- ошибки во время выполнения программы ("ну взяли в коде поделили на ноль")

gradle

)

22-09-30

Generics

Аналог полиморфизма в плюсах.

```
class Holder<T> (val value: T) { ... }

val intHolder = Holder<Int>(23)
val intHolderImpl = Holder(24)
```

Movable

```
class Pilot< T : Movable>(val vehicle: T) {
    fun go() {
        vehicle.move()
    }
}
```

`T` -- тип, который наследуется от `Movable` $\Rightarrow$ у `vehicle` точно будет метод `move()`.

Бтв, если в данном примере написать `Pilot<*>`, тип выберется неявно и получится `Pilot<Movable>`.

<:

== что-то является суперклассом чего-то. Просто обозначение

```
open class A
open class B : A()
class C : B()

//  <=>  Nothing <: C <: B <: A <: Any
```

Implicit subtyping

```
val c: C = C()
val b: B = c    // C <: B, OK

val holderB: Holder<B> = Holder(C())  // OK, because of casting

val holderC: Holder<C> = Holder(C())
val holderB: Holder<B> = holder<C>     // ERROR: Type mismatch. Required:
Holder<B>. Found: Holder<C>.
```

Type projections

Interface? Who?

Интерфейс -- это как абстрактный класс в плюсах: внутри есть объявление методов, иногда реализация. У инстансов, отнаследованных от интерфейса, появляется метод, но со своей реализацией.

For subclasses

```
interface Holder<T> {
    fun push(newValue: T) // consume an element
    fun pop(): T // produce an element
    fun size(): Int // does not interact with T
}

G<T> // invariant, can consume and produce elements

G<in T> // contravariant, can only consume elements
        // funtions that return T are not allowed

G<out T> // covariant, can only produce elements
         // functions that get T as an ang are not allowed

G<*> // star-projection, does not interact with T
```

For methods

```
class Holder<T> (var value: T?) {
   ...
   fun gift(other: Holder<in T>) { other.push(pop()) }
}

holderB.gift(holderA)
```

`T` может быть `null`. Но есть гарантия, что в `gift` передаётся ненулевой `T`.

Type erasure

Инстансы классов не хранят тип in the runtime

- ➔ `MutableMap<K, V>` становится `MutableMap<*, *>`
- ➔ `Pilot<T : Movable>` becomes `Pilot<Movable>`

Если у функции есть две реализации, отличающиеся только типом, можно одной из них дать имя (расширение для JVM)

```
fun quickSort(collection: Collection<Int>) { ... }
fun quickSort(collection: Collection<Double>) { ... }

// Both become quickSort(collection: Collection<*>) => Error :(

@JvmName("quickSortInt")
fun quickSort(collection: Collection<Int>) { ... }
fun quickSort(collection: Collection<Double>) { ... }

```

Nullability для T

Чтобы запретить появление типа `T?`, можно наследовать `T` от `Any` или пересечь `T` и `Any`

```
class Holder<T>(val value: T) { ... }
val holderA: Holder<A?> = Holder(null) // T = A?
                                       // OK

class Holder<T : Any>(val value: T) { ... }
val holderA: Holder<A?> = Holder(null) // ERROR: Type argument is not within its bounds.
                                       // Expected: Any. Found: A?.

fun <T> elvisLike(x: T, y: T & Any): T & Any = x ?: y
```

Inline functions

Лямбды в котлине хранятся как объекты -- на каждый вызов выделяется память и сохраняются данные.

`inline`

говорит "вместо создания нового объекта скопируй функцию: просто подставь её реализацию".

```
inline fun foo(str: String, call: (String) -> Unit) {
    call(str)
}
fun main() {
    foo("Top level function with lambda example", ::print)
}
```

`noinline`

позволяет в inline-функции не инлайнить какие-то аргументы

```
inline fun foo(str: String, call1: (String) -> Unit, noinline call2: (String) -> Unit) {
    call1(str) // Will be inlined
    call2(str) // Will not be inlined
}
```

`crossinline`

запрещает использование `return` в лямбда-функции на уровне компиляции

```
inline fun foo(crossinline call1: () -> Unit, call2: () -> Unit) {
    call1()
    call2()
}

fun main() {
    println("Step#1")
    foo({ println("Step#2")
          return }, // ERROR: 'return' is not allowed here
        { println("Step#3") }
    )
    println("Step#4")
}
```

`reified`

По дефолту, тип доступен только во время компиляции и стирается в сигнатуре функции. `reified` оставляет доступ к типу во время рантайма.

```
inline fun <reified T: Animal> foo() {
    println(T::class)  // OK
}
```

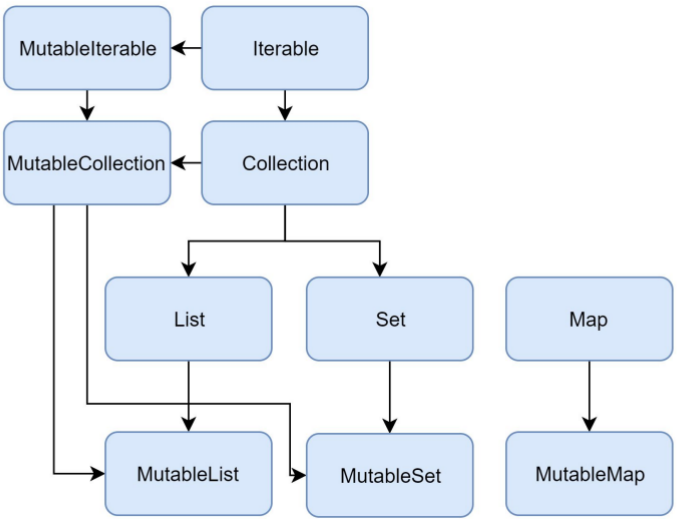
22-10-07

Containers

`Collection` -- штука, содержащая элементы определенного типа (i.e. `List`, `Map`, `Set`).

`MutableCollection` $\Rightarrow$ можно менять элементы.

Красивая схемка (taxonomy of collections)



interface Iterable

```
public interface Iterable<out T> {
    // Returns an iterator over the elements of this object.
    public operator fun iterator(): Iterator<T>
}
```

Класс наследуется от `Iterable` ⇒ можно итерироваться по элементам.

```
val iterator = myIterableCollection.iterator()
while (iterator.hasNext()) {
    iterator.next()
}
```

Если `MutableIterable`, появляется еще метод `remove()` у `iterator`.

Mutable Collection != Mutable Variable

```
val mutCol1 = mutableListOf(1, 2, 3)
mutCol1.add(4)                    // OK: `mutCol2` is mutable
mutCol1 = mutableListOf(4, 5, 6) // ERROR: Val cannot be reassigned

var mutCol2 = mutableListOf(1, 2, 3)
mutCol2.add(4)                    // OK: `mutCol2` is mutable
mutCol2 = mutableListOf(4, 5, 6) // OK: `mutCol2` is var
```

Set

```
class A(val primary: Int, val secondary: Int)

class B(val primary: Int, val secondary: Int) {
    override fun hashCode(): Int = primary

    override fun equals(other: Any?) = primary == (other as? B)?.primary
}

fun main() {
    val a1 = A(1,1)
    val a2 = A(1,2)
    val s1 = setOf(a1, a2)  // two elements

    val b1 = B(1, 1)
    val b2 = B(1, 2)
    val s2 = setOf(b1, b2)  // one element: b1.equals(b2) = true

    val s3 = buildSet {  // constructs read-only Set<Int>
        add(5)
        addAll(listOf(1, 2, 3))
    }
}
```

`as?` проверяет, что `other` не нулл, и кастит к классу `B`.

Map

```
val map1 = emptyMap<Int, String>()        // Builds the internal object
EmptyMap
val map2 = mapOf<Int, String>()           // Calls emptyMap()
val map3 = mapOf(1 to "one", 2 to "two")  // The type can be inferred

val map4 = buildMap {   // constructs read-only Map<Int, String>
    put(1, "one")
    putAll(mutableMapOf(2 to "two"))
}
```

Array

- не коллекция, но есть метод `iterator`
- фиксированный размер
- элементы можно изменять

Sequence

```
val sequence1 = emptySequence<Int>()  // Builds the internal object EmptySequence
val sequence2 = sequenceOf<Int>()     // Calls emptySequence()
val sequence3 = sequenceOf(1, 2, 3)   // The type can be inferred

val sequence4 = sequence {   // constructs Sequence<Int>
    yield(1)
    yieldAll(listOf(2, 3))
}

val sequence5 = generateSequence(1) { it + 1 } // an infinite sequence consisting of `1`, evaluated lazily
println(sequence5.take(5).toList()) >/ [1, 2, 3, 4, 5]
```

22-10-14

Functional programming

map

```
fun main() {
    val l = listOf(1, 2, 3)
    val m = l.map {it * it }
    print(m)   // [1, 4, 9]
    val n = l.map {it + 1}.map {it * it}
    print(n)   // [4, 9, 16]
}
```

filter

Возвращает новую коллекцию, удовлетворяющую какому-то предикату

```
fun main() {
    val l = listOf(1, 2, 3)
    val n = l.filter { it % 2 == 0 }
    print(n)
}
```

fold

Левосторонняя свёртка.

Создает аккумулятор, который обновляется на каждой итерации `for` по элементам, и возвращает его значение

```
fun main() {
    val l = listOf("ab", "ob", "a!")
    val n = l.fold("") { acc, x -> "$acc$x" }
    print(n)  // "aboba!"
}
```

```
fun main()
{
    val string = """
    One-one was a race horse.
    Two-two was one too.
    One-one won one race.
    Two-two won one too.
""".trimIndent()

    val result = string
        .split(" ", "-", ".", System.lineSeparator())
        .filter { it.isNotEmpty() }
        .map { it.lowercase() }
        .groupingBy { it }  // сгруппировать слова
        .eachCount()        // по повторениям
        .toList()
        .sortedBy { (_, count) -> count } // сортируем по второму аргументу
        .reversed()

    print(result)
    // [(one, 7), (two, 4), (won, 2), (too, 2), (race, 2), (was, 2), (horse, 1), (a, 1)]
}
```

foldRight

Правосторонняя свертка.

```
val list = listOf(1, 2, 3)
list.fold(0) { acc, x -> acc - x } // (((0 - 1) - 2) - 3) = -6
list.foldRight(0) { x, acc -> acc - x } // (-1 + (-2 + (0 - 3))) = -6
list.foldRight(0) { acc, x -> acc - x } // (1 - (2 - (3 - 0))) = 2
```

lambda

```
fun isEven(x: Int) = x % 2 == 0

fun main()
{
    val isEvenLambda = { x: Int -> x % 2 == 0 }

    val l = listOf(1, 2, 3)

    val res1 = l.partition{ it % 2 == 0 }
    val res2 = l.partition(::isEven) // function reference
    val res3 = l.partition(isEvenLambda) // pass lambda by name

    // разные способы сделать одно и то же
    print(res1 == res2 && res2 == res3) // true

    print(res1) // ([2], [1, 3])
}
```

Lazy vs Deferred computations

```
fun <F> withFunction(number: Int, even: F, odd: F): F
    = when (number % 2) {
        0 -> even
        else -> odd
    }

fun main() {
    withFunction(4, println("even"), println("odd"))
}
```

Функции (второй и третий аргументы) передаются как функции, не лямбды или еще что-то. `withFunction` будет посчитана до самого вызова

```
even
odd
```

```
fun <F> withLambda(number: Int, even: () -> F, odd: () -> F): F
    = when (number % 2) {
        0 -> even()
        else -> odd()
    }

fun main() {
    withLambda(4, { println("even") }, { println("odd") })
}
```

Теперь выведется только `even`.

sealed class

```
sealed class NewColor(val name: String)
class WhiteColor(name: String): NewColor(name)
class AzureColor(name: String): NewColor(name)
class HoneydewColor(name: String): NewColor(name)
```

`sealed` $\Longleftrightarrow$ все наследники `newColor` известны на этапе компиляции.

enum class

```
enum class COLOR {
    WHITE,
    AZURE,
    HONEYDEW
}

fun COLOR.getRGB() = when (this) {
    COLOR.WHITE -> "#FFFFFF"
    COLOR.AZURE -> "#F0FFFF"
    COLOR.HONEYDEW -> "F0FFF0"
}
```

`enum`

Инстансы `enum` класса можно сразу проинициализировать:

```
enum class Color(val rgb: Int) {
    RED(100),
    GREEN(101),
    BLUE(110)
}

fun main() {
    println(Color.RED)  // RED
    println(Color.RED.rgb)  // 100
}
```

22-10-14

Компиляция

Сишный компилятор превращает исходный код сразу в машинный код, программа может работать некорректно на нескольких платформах.

Скомпилированный код на джаве/котлине превращается в набор инструкций -- `bytecode`. Потом используется `JVM` , которая интерпретирует бинарник, превращая его в машинный код. Программа становится кросс-платформенной.

Компоненты JVM

1. Classloading services

подгружают нужные библиотеки (*aka* классы). После завершения загрузки все важные классы помещаются в кэш JVM, чтобы был быстрый доступ во время исполнения. Остальные классы подгружаются по запросу.

Первый запрос медленнее остальных из-за загрузки классов. Этот процесс называется прогревом JVM.

2. Memory management

- heap memory manager
- garbage collection

3. Interpreter ↔ JIT-compiler

Интерпретатор и JIT-компилятор работают параллельно.

Интерпретатор превращает байт-код в ассемблерный без оптимизаций. Как он это делает: у каждого фрагмента (i.e., функции, класса) есть указатель либо на байт-код, либо на ассемблерный код (таковой появляется из-за JIT-компилятора). Интерпретатор гуляет по этим указателям и таким образом выстраивает финальную версию ассемблерного кода.

JIT == Just-In-Time compilation. *JIT-компилятор* находит какие-то кусочки уже скомпилированного кода, долгие по времени, и меняет их на оптимизированный ассемблерный код (отсюда и возникает случай, когда указатель указывает на память, в которой живет ассмеблерный код).

Рефлексия: если кусок кода вдруг оказался невалидным (i.e., сломались инварианты), JIT-компилятор возвращает этот кусок интерпретатору. Всё происходит быстро: указатель меняется с ассемблерного кода на байт-код.

Первый запуск этого дуэта долгий из-за оптимизации, зато следующие исполнения быстрые, потому что есть кэширование.

Можно ли оценить время компиляции кода?

Тьюринг сказал, что нет. Но мы можем предположить: если что-то отработало долго, то велика вероятность, что в следующий раз оно снова отработает долго.

Еще время зависит от того, как устроен JIT-компилятор и сама JVM. Об этом в следующем полугодии.

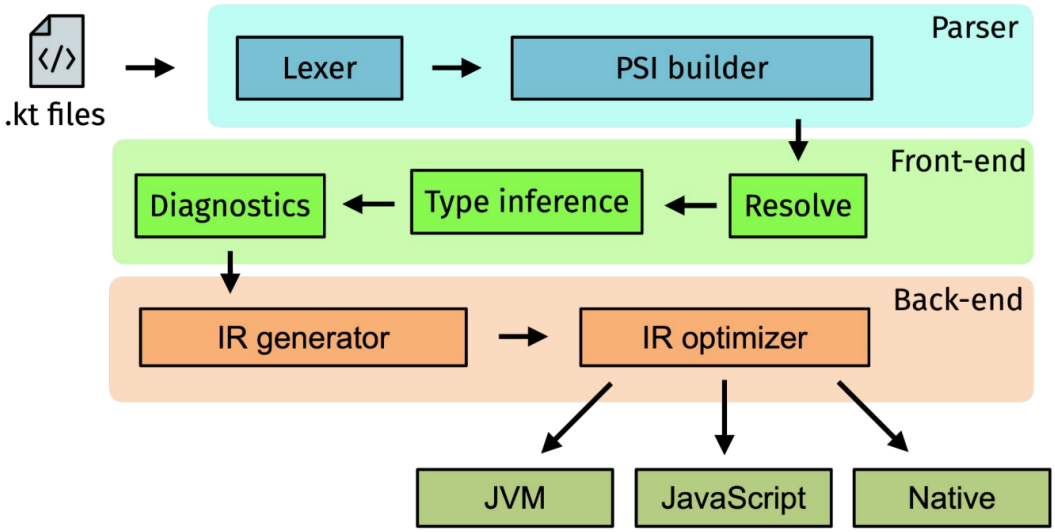
22-10-21

А теперь про котлин

Байт-коды котлина и джавы устроены одинаково, JVM может распарсить и то, и то. Поэтому это совместимые языки.

Компилятор котлина

- parser-- парсит
- front-end-- генерирует сообщения об ошибках, диагностиках, проверяет ассерты, диктат, детект, все дела
- back-end -- генерирует ассемблерный код

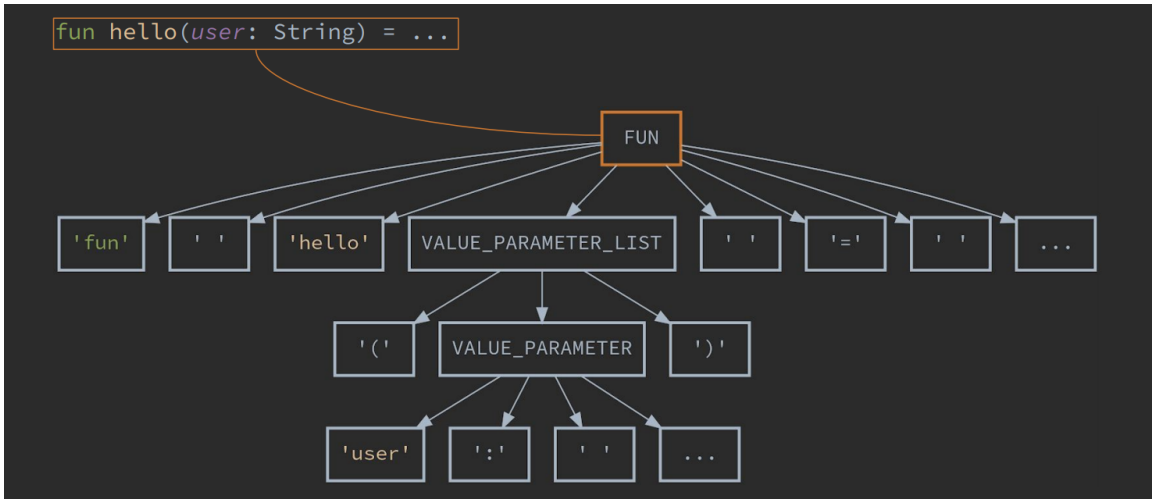


1. Parser

- Lexer -- сплитит программу на токены.
- PSI builder -- все токены хранятся в виде дерева -- расширенного AST. Дерево неизменяемое.

  Нет информации о типах, возвращаемых значениях и т.д.

  Пример PSI:

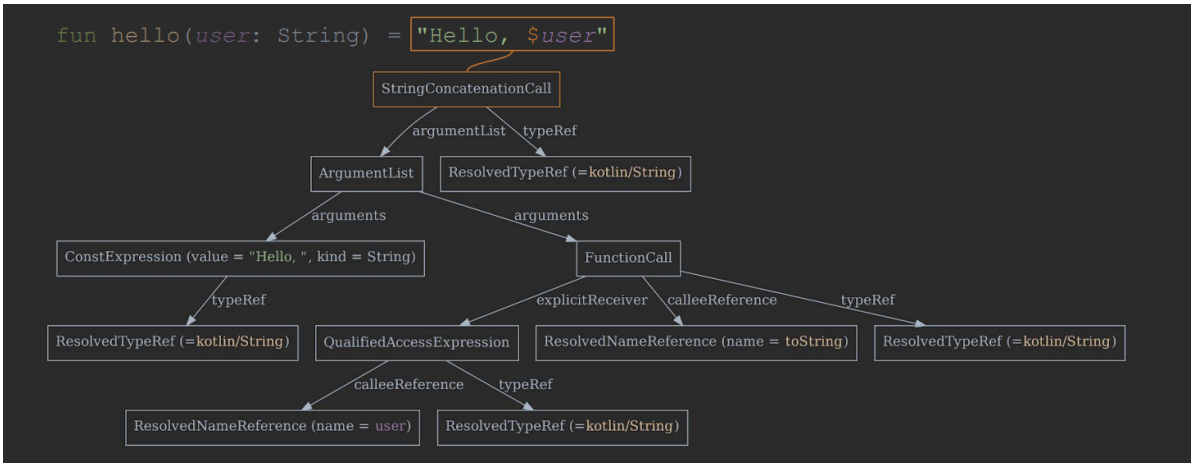


2. Front-end

Используется *FIR* -- Frontend Intermediate Representation -- дерево, которое можно изменять. FIR от слова ёлка!

Строется на основе PSI. Внутри хранятся названия и типы аргументов/переменных, возвращаемые значения, сообщения об ошибках.

Пример FIR:



Используется дешугаринг (i.e., `if` → `when`, `for` → `while`), чтобы уменьшить количество различных видов структур ⇒ уменьшить количество вершинок в FIR (в PSI такого нет).

Resolve

Разруливание ошибок компиляции, связанных с именами.

Например, если есть две функции с одинаковыми именами, но в разных библиотеках, они сохранены в PSI с одинаковыми именами. Но в FIR для них прописан полный путь.

Type inference

Некоторая информация про типы может быть выкинута, компилятор сможет сам вывести тип.

Smart casts: в условиях `if`, `when` компилятор сам догадается, какой тип у переменной, и предложит соответствующие методы.

Diagnostics

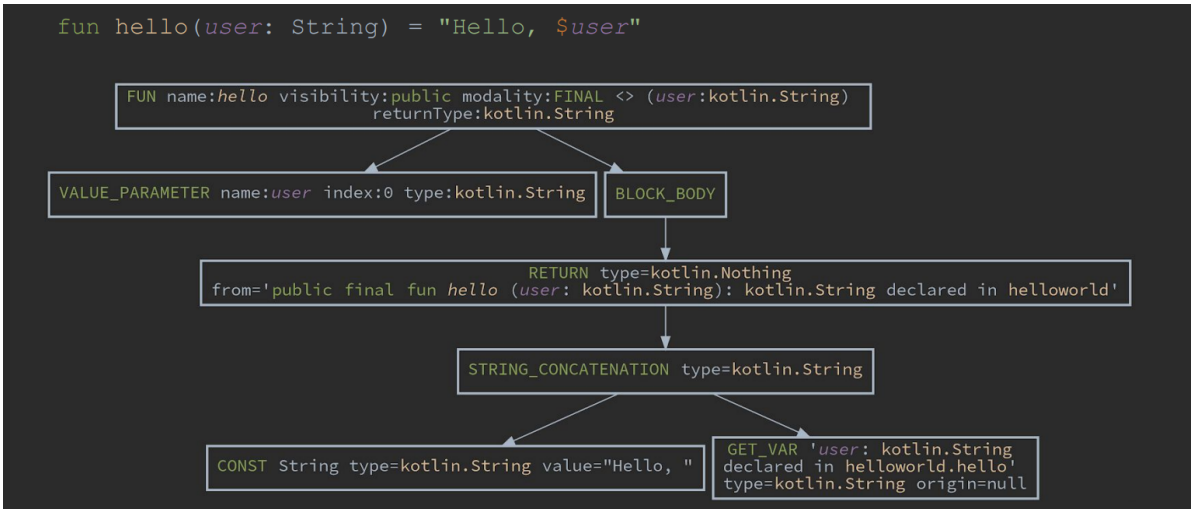
Всевозможные проверки, которые можно увидеть в идее. Диктат, детект -- это всё сюда.

А ещё этой части компилятора может и не быть.

3. Back-end

Генерация ассемблерного кода, специфичного для платформы, на основе FIR. Полученный код отправляется, например, в JVM или JavaScript.

- Используется IR -- Intermediate Representation -- еще одно дерево. Благодаря третьему дереву обеспечивается кросс-платформенность котлина: для каждой машины своё IR.
- Вершинки IR подписаны более конткретно, имена функций/переменных используются без ошибок. Никакого resolve в этой части компилятора.
- Есть некоторые простые оптимизации по типу "подставь константы"
- Происходит обратный процесс дешугаринга: по указателям на память восстанавливаются `if`, `when`, `for`, `while`.



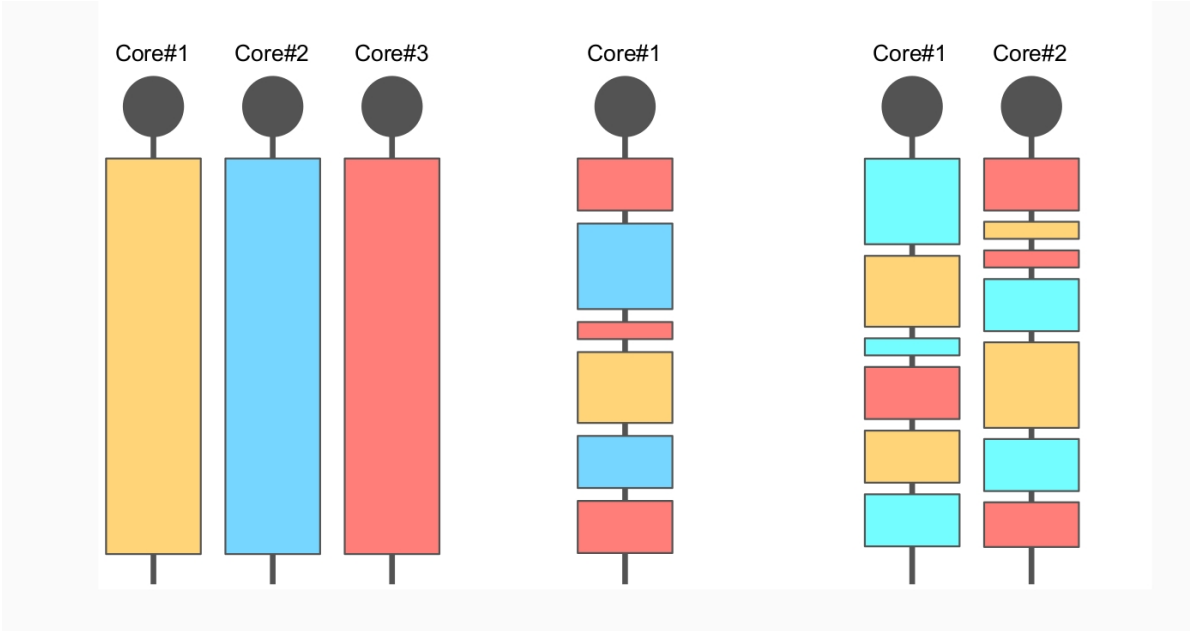
Плагины \todo

22-11-11

Parallelism & Concurrency

Parallel -- задачи на разных потоках могут выполняться одновременно

Concurrent -- есть прогресс на нескольких потоках



Preemptive & Cooperative sheduling

Вытесняющая и кооперативная многозадачность.

В первом варианте потоки могут быть прерваны в какой-то рандомный момент, во втором -- в коде явно выделены точки, в которых происходит переключение.

JVM потоки

JVM потоки != потоки операционки. У них может быть разный шедулинг.

Типы JVM потоков:

- user thread -- потоки, запущенные пользователем.
- daemon потоки -- что-то, что было запущено и работает в фоне. Периодически проверяем, что такие потоки живы. Если не живы, перезапускаем.

  Daemon поток останавливается после того, как все user потоки закончили работу.

  JVM завершает работу и не ждет, когда daemon поток закончит работу.

Замечание: накладывать на daemon поток обязанность записывать какую-то инфу -- плохая идея. JVM может завершить работу, пользователь решит завершить процесс записи, и произойдет потеря данных. Для записи лучше использовать user потоки, которые точно завершатся корректно.

Java packages

Есть несколько основных пакетов, которые предоставляют примитивы для конктретного параллельного программирования.

`java.lang` -- основа основ: `Runnable`, `Thread`, etc.

`java.util.concurrent` -- синхронизация, структуры данных для concurrency

`kotlin.concurrent` -- обёртки и расширения для Java классов

Как создать поток и как им управлять

Первый вариант -- сновной интерфейс `Runnable`:

```
@FunctionalInterface
    public interface Runnable {
    public abstract void run();
}

class RunnableWrapper(val runnable: Runnable)
val myWrapperObject = RunnableWrapper(object : Runnable {
    override fun run() { println("I run") }
})
val myWrapperLambda = RunnableWrapper { println("yo") }   // вместо
аргумента передается лямбда
```

Второй вариант -- отнаследоваться от интерфейса `Thread()`:

```
class MyThread : Thread() {
    override fun run() {
        println("${currentThread()} is running")
    }
}

fun main() {
    val myThread1 = MyThread()
    myThread1.start()
    val myThread2 = MyThread()
    myThread2.run()   // maybe be got blocked
}
```

Метод `start()` создает новый поток, который будет работать конкуррентно. Метод `run()` запускает поток внутри потока.

Третий вариант -- передать потоку объект, который реализует интерфейс `Runnable`. Один и тот же объект можно передать нескольким потокам.

```
fun main() {
    val myRunnable = Runnable { println("Sorry, gotta run!") }
    val thread1 = Thread(myRunnable)
    thread1.start()
    val thread2 = Thread(myRunnable)
    thread2.start()
}
```

Четвертый вариант -- лучший -- котлине можно использовать обёртки, живущие в библиотеке `kotlin.concurrent.thread`.

`thread` помимо лямбды принимает аргументы.

```
fun thread(
    start: Boolean = true,
    isDaemon: Boolean = false,
    contextClassLoader: ClassLoader? = null,
    name: String? = null,
    priority: Int = -1,
    block: () -> Unit
): Thread
```

По умолчанию, новый поток создаётся и сразу запускается конкуррентно. `start = false` $\Rightarrow$ новый поток не будет запущен сразу.

Пример:

```
import kotlin.concurrent.thread

fun main() {
    val thread1 = thread {
    println("I start instantly")
    }
    val thread2 = thread(false) {
    println("I don't start instantly")
    }
    thread2.start()
    thread1.join()   // явно выражена точка, в которой мы ждем `thread1`
}
```

Подробнее про kotlin.concurrent.thread

Поля `Thread()`:

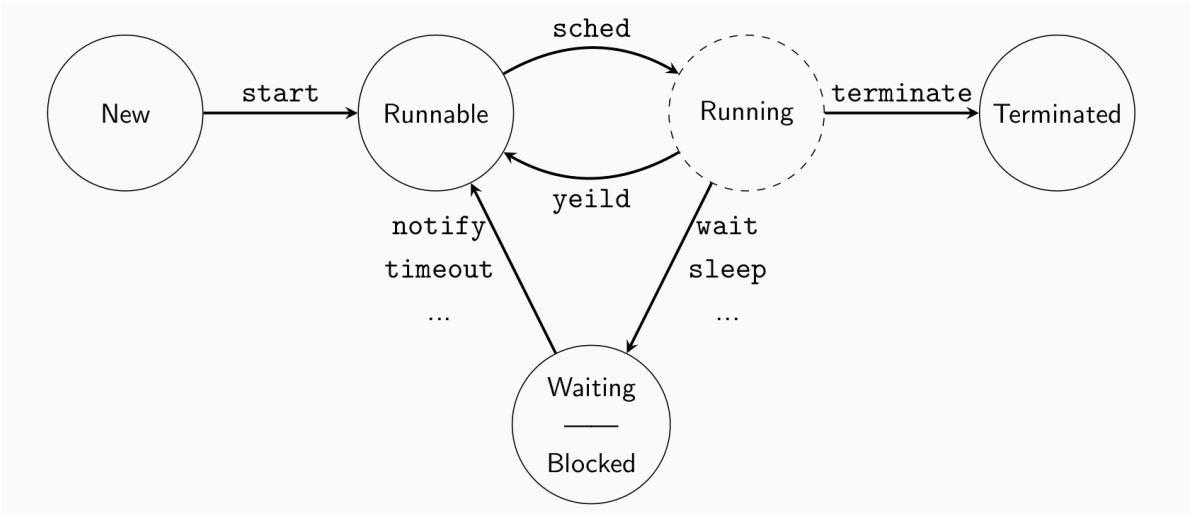
- `id: Long`
- `name: String?`
- `priority: Int` $\in [1, 10]$; чем больше, тем выше приоритет
- `daemon: Boolean`
- `state: Thread.state`
- `isAlive: Boolean`

Поля не могут быть изменены после того, как поток был запущен.

Thread.state & isAlive

state	isAlive
NEW	false
RUNNABLE	true
BLOCKED	true
WAITING	true
TIMED_WAITING	true
TERMINATED	false

тоже waiting, но установлен таймер



`yeild` означает "останови поток, дай ему отдохнуть".

22-11-18

Lock

Интерфейс, что-то вроде мьютексов

```
lock = ReentrantLock()
lock.lock()
lock.unlock()
lock.tryLock()  // возвращает `true` если получилось взять залочить
lock.withLock { ... }  // выполняется лямбда под локом
lock.newCondition() // создание condition variable
```

Пример:

```
class LockedCounter {
    private var c = 0
    private val lock = ReentrantLock()

    fun increment() {
        lock.withLock {
            c++
        }
    }

    // same for other methods
}
```

Condition variables

```
private val lock = ReentrantLock()
private val condition = lock.newCondition()
```

`condition` говорит потоку, что ему нужно делать

```
class PositiveLockedCounter {
    private var c = 0
```

```
    private val lock = ReentrantLock()
    private val condition = lock.newCondition()

    fun increment() =
        lock.withLock {
            c++
            condition.signal()   // wakes the thread up
        }

    fun decrement() =
        lock.withLock {
            if (c == 0)
                condition.await()  // `c` must be always positive
            c--
        }

    fun value() = return lock.withLock { c }
}
```

Synchronized Statement

Только для JVM !!!!!

В JVM гарантируется, что у каждого объекта есть свой `lock`. Вместо того, чтобы явно использовать методы лока, можно использовать ключевое слово `synchronized`

```
class Counter {
    private var c = 0

    fun increment() {
        synchronized(this) {  // так
            c++
        }
    }

    @Synchronized             // или так
    fun decrement() {
        c--
    }
}
```

Synchronized может быть только у объектов. Примитивные типы -- не объекты, они могут быть только volatile.

ReadWriteLock

Несколько потоков-читателей, но только один поток-модификатор

```
rwLock.readLock()   // возращает интерфейс lock который захватывает readlock
rwLock.writeLock()  // возвращает интерфейс lock, захватывающий writelock

rwLock.read { ... }   // лямбда под readlock
rwLock.write { ... }  // лямбда под writelock
```

Пример:

```
class RWLockedCounter {
    private var c = 0
    private val rwLock = ReadWriteReentrantLock()

    fun imcrement() = rwLock.write { c++ }
    fun value() = return rwLock.read { c }
}
```

JMM

Weak behaviors

Нет гарантированной последовательности внутри одного потока. Опитимизации JVM могут переставить строки.

При использовании нескольких потоков можно получить совершенно разные результаты.

Volatile fields

Слово `volatile` сообщает компилятору, что нельзя предсказать, как будет меняться значение переменной. Потоки должны читать значение переменной из общей памяти, потому что его нельзя кэшировать или запихнуть в регистр.

Компилятор не делает агрессивные оптимизации (не меняет строки местами, не заменяет условие на `true` внутри `while`)

Пример раз:

```
class OrderingTest {
    @Volatile var x = 0
    @Volatile var y = 0
```

```
    fun test() {
        thread {
            x = 1    // если переставить строки 6, 7, можно в строке 12 получить вывод '1, 0'
            y = 1    // но строки не переставляются компилятором из-за volatile
        }
        thread {
            val a = y
            val b = x
            println("$a, $b")
        }
    }
}
```

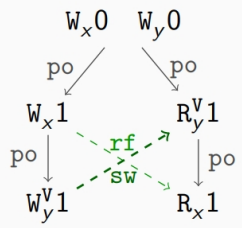
Пример два:

```
class ProgressTest {
    @Volatile
    var flag = false
    fun test() {
        thread {
            while (!flag) {}    // если flag не volatile, компилятор заменит условие внутри while на true
            println("I am free!")
        }
        thread {
            flag = true
        }
    }
}
```

Happens-before relation

Порядок операций между потоками может быть разный. Для каждого такого порядка построим граф

```
class OrderingTest {
  var x = 0
  @Volatile var y = 0
  fun test() {
    thread {
      x = 1
      y = 1
    }
    thread {
      val a = y
      val b = x
      println("$a, $b")
    }
  }
```



$\xrightarrow{po}$ *program-order*
$\xrightarrow{rf}$ *reads-from*
$\xrightarrow{sw}$ *synchronizes-with*
-*e.g.* reads-from on `Volatile` field
$\xrightarrow{hb}$ *happens-before*

po -- порядок, в котором происходят события в одном потоке

rf $= a \to b$ --- из a записали значение в b.

sw появляется в нескольких случаях, один из них -- когда происходит `reads-from` из volatile переменной. **sw** появляется на месте **rf**.

hb --- возникает, если есть единственный путь между вершинками. В данном случае, $\mathtt{hb} = \mathtt{W_x}1 \to \mathtt{R_x}1$, потому что сначала идем по левой ветке, потом через **sw** попадаем в правую ветку; порялок определен.

Почему в данном случае не может быть $\mathtt{rf} = \mathtt{W_x}0 \Rightarrow \mathtt{R_x}1$? Тогда это бы означало, что чтение из $\mathtt{y}(\mathtt{R_y}1)$ уже произошло. Значит, мы должны были пройти по левой ветке. Но мы ее скипнули, противоречие.

Когда возникает sw отноошение?

1. Если переменная volatile
2. lock/unlock: лок отпускается, и только после этого берется новый лок
3. thread run/start: внутри `thread1` запускаем `thread2` (с помощью start), тогда `thread2` начинает выполняться (функция run)
4. thread finish/join: если внутри `thread1` запущен `thread2` и `thread1` хочет подождать `thread2`, то завершение `thread2` и `join` синхронизированы.

data-race

Возникает, если:

- потоки обращаются к одной и той же памяти и они не атомарные
- хотя бы один из потоков пишет
- между ними нет отношения happens-before

Программа data-race-free, если мы построили всевозможные графы и ни в одном из них нет гонок.

Atomics

Если atomic, то подразумевается, что ещё и volatile.

Пример: `AtomicInteger` -- атомарный инт.

Все функции ниже атомарные.

`get()` -- получить значение

`set(v)` -- изменить значение

`getAnsSet(v)` -- атомарно заменить значение на `v`

`compareAnsSet(e, v)` -- если старое значение = `e`, оно зменяется на `v`. Сама функция возвращает `true`, если замена произошла, иначе `false`.

`compareAndExchange(e, v)` -- делает то же самое, но возвращает прочитанное значение (`e` в случае равенства, иначе другое).

`getAndIncrement(), addAndGet()`, etc -- атомарные арифметические операции

Atomic Field Updater

```
class Counter {
    @Volatile
    private var c = 0
    companion object {
        private val updater =
        AtomicIntegerFieldUpdater.newUpdater(Counter::class.java, "c")
    }
    fun increment() {
        updater.incrementAndGet(this)
    }
    fun decrement() {
        updater.decrementAndGet(this)
    }
    fun value(): Int = updater.get(this)
}
```

`updater` -- такая штука, которая привязывается к определённому полю класса. Поле должно быть volatile.

Когда вызывается какой-то метод, `updater` сам находит нужное поле в классе и атомарно обновляет/достаёт значение.

Клёвая библиотечка для котлина

kotlin.atomicfu для использования атомиков в котлине

Создана по образу и подобию джавы, но внутри используется компиляторный плагин, который модифицирует код, заменяя volatile на Atomic<...>FieldUpdater.