

Kotlin

Author: Daria Shutina

Kotlin

Organization stuff

22-08-09

print vs println

variables

functions

when-expression

&& vs AND /TODO

loops in lists

while-cycle

labels

ranges

null safety

elvis-operator (?:)

safe calls

unsafe calls

TODO

lambda-выражения

22-09-16

ООП

Наследование

Конструкторы

Operator overload

ComponentN operator

Infix functions

22-09-23

gradle

22-09-30

Generics

Movable

<:

Implicit subtyping

Type projections

Interface? Who?

For subclasses

For methods

Type erasure

Nullability для T

Inline functions

Пример использования

`inline`

`noinline`

`crossinline`

`reified`

22-10-07

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 ↔ 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 scheduling

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** отношение?

data-race

Atomics

Atomic Field Updater

Клэвая библйотечка для котлина

22-11-25

Предисловие

`Promise<T>`

suspend

Coroutine scope

Способ раз, немодный

Пример

Способ два, модный

Тип Job

Job state

22-12-02

Диспатчеры

Контексты диспатчера

Диспатчер под капотом

Переключение контекста

Сравнение корутины и thread'a

Исключения

Конвенция

Доп ссылки

Organization stuff

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

22-08-09

print vs println

```
1 fun main(args: Array<String>) {
2     print("Hello")
3     println(", world!")
4 }
```

→ `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.

variables

```

1 | val a: Int = 1    // immediate assignment
2 | var b = 2        // `Int` type is inferred
3 | b = a            // Reassigning to `var` is okay
4 | val c: Int       // Type required when no initializer is provided
5 | c = 3            // Deferred assignment
6 | a = 4            // Error: Val cannot be reassigned
7 | 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

```

1 | fun sum(a: Int, b: Int): Int {
2 |     return a + b;
3 | }
4 |
5 | fun mul(a: Int = 1, b: Int) = a * b;

```

Значения по умолчанию можно писать в любом порядке.

```

1 | fun max_val1(a: Int, b: Int) {
2 |     if (a > b) {
3 |         return a
4 |     } else {
5 |         return b
6 |     }
7 | }
8 |
9 |
10 | fun max_val2(a: Int, b: Int) =
11 | return {
12 |     if (a > b) {
13 |         a
14 |     } else {
15 |         b
16 |     }
17 | }
18 |
19 | 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.

```
1  when (x) {  
2      1 -> print("x = 1")  
3      2 -> print("x = 2")  
4      else -> {  
5          print("aboba")  
6      }  
7  }  
8  
9  when (val input = parseInput()) {  
10     is String -> print("A string value was passed")  
11     is Int, is Double -> print("An number value was passed")  
12 }
```

&& vs AND /TODO

loops in lists

```
1  val items = listOf("a", "b", "c") // это 0, а не ноль :)  
2  
3  for (item in items) {  
4      println(item)  
5  }  
6  
7  for (index in items.indices) {  
8      println("item at $index is ${items[index]}")  
9  }  
10  
11  for ((index, item) in items.withIndex()) {  
12      println("item at $index is $item")  
13  }
```

while-cycle

```

1  val items = listOf("apple", "banana", "kiwifruit")
2
3  var index = 0
4  while (index < items.size) {
5      println("item at $index is ${items[index]}")
6      index++
7  }
8
9  var isComplete: Boolean
10 do {
11     --.
12     toComplete = --.
13 } while(toComplete)

```

labels

Даём название циклу, чтобы можно было к нему обратиться

```

1  myLabel@ for (item in items) {
2      for (item2 in otheritems) {
3          if (...) break@myLabel
4          else continue@myLabel
5          ...
6      }
7  }

```

ranges

```

1  val x = 10;
2  if (x in 1..10) {
3      println("x is between 1 and 10")
4  }
5
6  for (x in 1..5) {...}
7
8  for (x in 9 downTo 0 step 3) {...}

```

null safety

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

```
1 val CanBeNull: String?
2 val CantBeNull: String
```

elvis-operator (?:)

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

```
1 fun check(id: String): String? {
2     val item = findItem(id) ?: return null
3     return id
4 }
```

safe calls

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

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

`it` -- зарезервированная переменная в котлине

unsafe calls

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

```
1 for (employee: Employee) {
2     println(employee.department!!.head!!.name!!)
3 }
```

TODO

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

```
1 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 ответствен 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

```
1 open class Point(val x: Int, val y: Int, private z: String) {
2     constructor(other: Point) : this(other.x, other.y) {...}
3 }
```

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

Operator overload

```
1 class Ex {
2     operator fun plus(other Ex) {...}
3     operator fun dec() {...}
4 }
```

ComponentN operator

```
1 |
```

Infix functions

```
1 data class Person(val name: String)
2
3 infix fun String.with(other: String) = Person(this, other)
4
5 fun main() {
6     val readHero = "Rayan" with "Gosling"
7     val (real, bean) = readHero
8 }
```

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.

```
1 class Holder<T> (val value: T) { ... }
2
3 val intHolder = Holder<Int>(23)
4 val intHolderImpl = Holder(24)
```

Movable

```
1 class Pilot< T : Movable>(val vehicle: T) {
2     fun go() {
3         vehicle.move()
4     }
5 }
```

`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

```

1 open class A
2 open class B : A()
3 class C : B()
4
5 // <=> Nothing <: C <: B <: A <: Any

```

Implicit subtyping

```

1 val c: C = C()
2 val b: B = c    // C <: B, OK
3
4 val holderB: Holder<B> = Holder(C()) // OK, because of casting
5
6 val holderC: Holder<C> = Holder(C())
7 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 экземпляров, отнаследованных от интерфейса, появляется метод, но со своей реализацией.

For subclasses

```

1 interface Holder<T> {
2     fun push(newValue: T) // consume an element
3     fun pop(): T // produce an element
4     fun size(): Int // does not interact with T
5 }
6
7 G<T> // invariant, can consume and produce elements
8
9 G<in T> // contravariant, can only consume elements
10     // funtions that return T are not allowed
11
12 G<out T> // covariant, can only produce elements
13     // functions that get T as an arg are not allowed
14
15 G<*> // star-projection, does not interact with T

```

For methods

```

1 class Holder<T> (var value: T?) {
2     ...
3     fun gift(other: Holder<in T>) { other.push(pop()) }
4 }
5
6 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)

```

1 fun quickSort(collection: Collection<Int>) { ... }
2 fun quickSort(collection: Collection<Double>) { ... }
3
4 // Both become quickSort(collection: Collection<*>) => Error :(
5
6 @JvmName("quickSortInt")
7 fun quickSort(collection: Collection<Int>) { ... }
8 fun quickSort(collection: Collection<Double>) { ... }
9

```

Nullability для T

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

```

1 class Holder<T> (val value: T) { ... }
2 val holderA: Holder<A?> = Holder(null) // T = A?
3                                     // OK
4
5 class Holder<T : Any> (val value: T) { ... }
6 val holderA: Holder<A?> = Holder(null) // ERROR: Type argument is not within
7                                     // Expected: Any. Found: A?.
8
9 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.

Пример использования

Если лямбда передается в функцию `f` последним аргументом, её тело можно написать в вызове `f` в фигурных скобках.

```

1 fun func(val1: Int,
2         val2: Int,
3         myLambda: (Int, Int) -> Int) = myLambda(val1, val2)
4
5 fun main() {
6     print(func(10, 20) {
7         val1, val2 -> val1 + val2
8     })
9 }

```

inline

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

```

1 inline fun foo(str: String, call: (String) -> Unit) {
2     call(str)
3 }
4 fun main() {
5     foo("Top level function with lambda example", ::print)
6 }

```

noinline

позволяет в inline-функции не инлайнить какие-то аргументы

```
1 inline fun foo(str: String, call1: (String) -> Unit, noinline call2: (String)
  -> Unit) {
2     call1(str) // Will be inlined
3     call2(str) // Will not be inlined
4 }
```

crossinline

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

```
1 inline fun foo(crossinline call1: () -> Unit, call2: () -> Unit) {
2     call1()
3     call2()
4 }
5
6 fun main() {
7     println("Step#1")
8     foo({ println("Step#2")
9         return }, // ERROR: 'return' is not allowed here
10        { println("Step#3") })
11 }
12 println("Step#4")
13 }
```

reified

По дефолту, тип доступен только во время компиляции и стирается в сигнатуре функции.

`reified` оставляет доступ к типу во время рантайма.

```
1 inline fun <reified T: Animal> foo() {
2     println(T::class) // OK
3 }
```

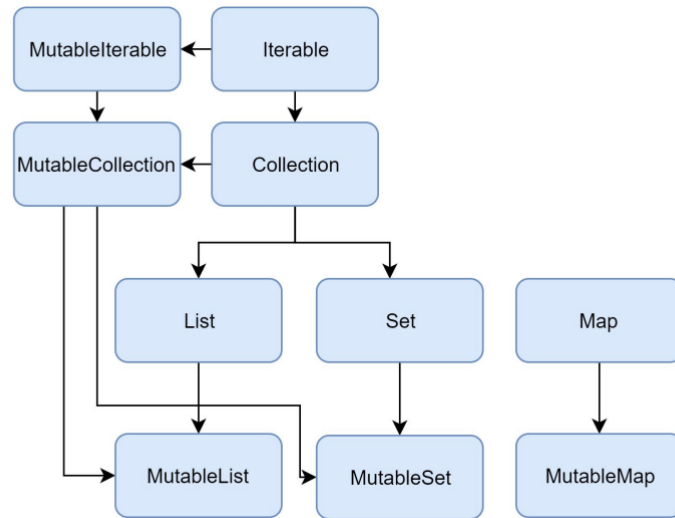
22-10-07

Containers

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

`MutableCollection` ⇒ можно менять элементы.

Красивая схемка (taxonomy of collections)



interface Iterable

```

1 public interface Iterable<out T> {
2     // Returns an iterator over the elements of this object.
3     public operator fun iterator(): Iterator<T>
4 }
  
```

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

```

1 val iterator = myIterableCollection.iterator()
2 while (iterator.hasNext()) {
3     iterator.next()
4 }
  
```

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

Mutable Collection != Mutable Variable

```

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

```

Set

```

1  class A(val primary: Int, val secondary: Int)
2
3  class B(val primary: Int, val secondary: Int) {
4      override fun hashCode(): Int = primary
5
6      override fun equals(other: Any?) = primary == (other as? B)?.primary
7  }
8
9  fun main() {
10     val a1 = A(1,1)
11     val a2 = A(1,2)
12     val s1 = setOf(a1, a2) // two elements
13
14     val b1 = B(1, 1)
15     val b2 = B(1, 2)
16     val s2 = setOf(b1, b2) // one element: b1.equals(b2) = true
17
18     val s3 = buildSet { // constructs read-only Set<Int>
19         add(5)
20         addAll(listOf(1, 2, 3))
21     }
22 }

```

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

Map


```

1  val map1 = emptyMap<Int, String>()           // Builds the internal object
    EmptyMap
2  val map2 = mapOf<Int, String>()              // Calls emptyMap()
3  val map3 = mapOf(1 to "one", 2 to "two")    // The type can be inferred
4
5  val map4 = buildMap {                        // constructs read-only Map<Int, String>
6      put(1, "one")
7      putAll(mutableMapOf(2 to "two"))
8  }

```

Array

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

Sequence

```

1  val sequence1 = emptySequence<Int>()        // Builds the internal object
    EmptySequence
2  val sequence2 = sequenceOf<Int>()           // Calls emptySequence()
3  val sequence3 = sequenceOf(1, 2, 3)         // The type can be inferred
4
5  val sequence4 = sequence {                  // constructs Sequence<Int>
6      yield(1)
7      yieldAll(listOf(2, 3))
8  }
9
10 val sequence5 = generateSequence(1) { it + 1 } // an infinite sequence
    consisting of `1`, evaluated lazily
11 println(sequence5.take(5).toList()) >/ [1, 2, 3, 4, 5]

```

22-10-14

Functional programming

map

```

1 fun main() {
2     val l = listOf(1, 2, 3)
3     val m = l.map {it * it }
4     print(m)    // [1, 4, 9]
5     val n = l.map {it + 1}.map {it * it}
6     print(n)    // [4, 9, 16]
7 }

```

filter

Возвращает новую коллекцию, удовлетворяющую какому-то предикату

```

1 fun main() {
2     val l = listOf(1, 2, 3)
3     val n = l.filter { it % 2 == 0 }
4     print(n)
5 }

```

fold

Левосторонняя свёртка.

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

```

1 fun main() {
2     val l = listOf("ab", "ob", "a!")
3     val n = l.fold("") { acc, x -> "$acc$x" }
4     print(n)    // "aboba!"
5 }

```

```

1 fun main()
2 {
3     val string = """
4     One-one was a race horse.
5     Two-two was one too.
6     One-one won one race.
7     Two-two won one too.
8     """.trimIndent()
9

```

```

10     val result = string
11         .split(" ", "-", ".", System.lineSeparator())
12         .filter { it.isNotEmpty() }
13         .map { it.lowercase() }
14         .groupingBy { it } // сгруппировать слова
15         .eachCount()       // по повторениям
16         .toList()
17         .sortedBy { (_, count) -> count } // сортируем по второму аргументу
18         .reversed()
19
20     print(result)
21     // [(one, 7), (two, 4), (won, 2), (too, 2), (race, 2), (was, 2), (horse,
22     1), (a, 1)]
23 }

```

foldRight

Правосторонняя свертка.

```

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

```

lambda

```

1 fun isEven(x: Int) = x % 2 == 0
2
3 fun main()
4 {
5     val isEvenLambda = { x: Int -> x % 2 == 0 }
6
7     val l = listOf(1, 2, 3)
8
9     val res1 = l.partition { it % 2 == 0 }
10    val res2 = l.partition { it.isEven } // function reference
11    val res3 = l.partition { it.isEvenLambda } // pass lambda by name
12
13    // разные способы сделать одно и то же
14    print(res1 == res2 && res2 == res3) // true
15
16    print(res1) // ([2], [1, 3])
17 }

```

Lazy vs Deferred computations

```

1 fun <F> withFunction(number: Int, even: F, odd: F): F
2     = when (number % 2) {
3         0 -> even
4         else -> odd
5     }
6
7 fun main() {
8     withFunction(4, println("even"), println("odd"))
9 }

```

Функции (второй и третий аргументы) передаются как функции, не лямбды или еще что-то.

`withFunction` будет посчитана до самого вызова

```

1 even
2 odd

```

```

1 fun <F> withLambda(number: Int, even: () -> F, odd: () -> F): F
2     = when (number % 2) {
3         0 -> even()
4         else -> odd()
5     }
6
7 fun main() {
8     withLambda(4, { println("even") }, { println("odd") })
9 }

```

Теперь выведется только `even`.

sealed class

```

1 sealed class NewColor(val name: String)
2 class WhiteColor(name: String): NewColor(name)
3 class AzureColor(name: String): NewColor(name)
4 class HoneydewColor(name: String): NewColor(name)

```

`sealed` \iff все наследники `NewColor` известны на этапе компиляции.

enum class

```

1  enum class COLOR {
2      WHITE,
3      AZURE,
4      HONEYDEW
5  }
6
7  fun COLOR.getRGB() = when (this) {
8      COLOR.WHITE -> "#FFFFFF"
9      COLOR.AZURE  -> "#F0FFFF"
10     COLOR.HONEYDEW -> "F0FFF0"
11  }

```

enum

Инстансы `enum` класса можно сразу проинициализировать:

```

1  enum class Color(val rgb: Int) {
2      RED(100),
3      GREEN(101),
4      BLUE(110)
5  }
6
7  fun main() {
8      println(Color.RED)    // RED
9      println(Color.RED.rgb) // 100
10 }

```

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 ассемблерный 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

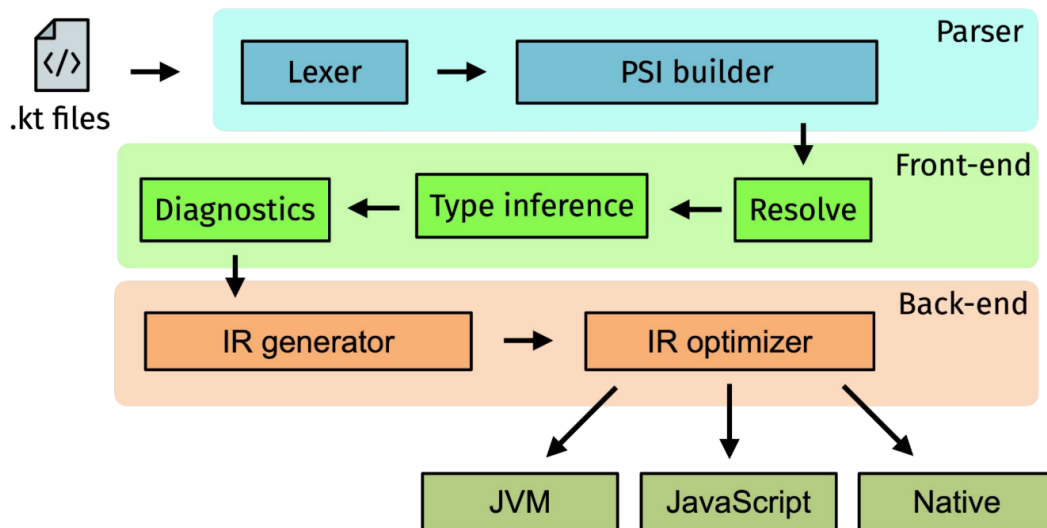
А теперь про котлин

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



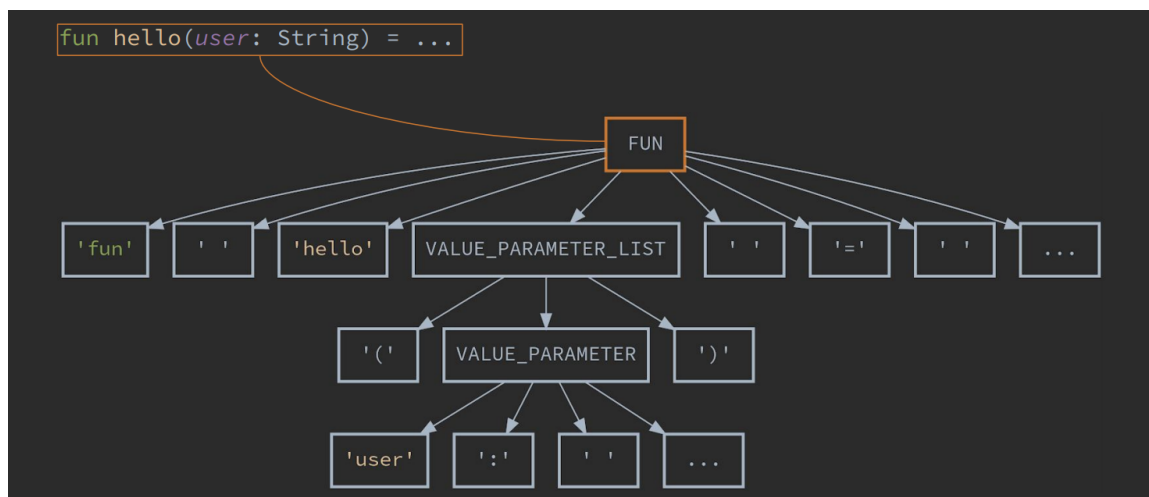


1. Parser

- 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:

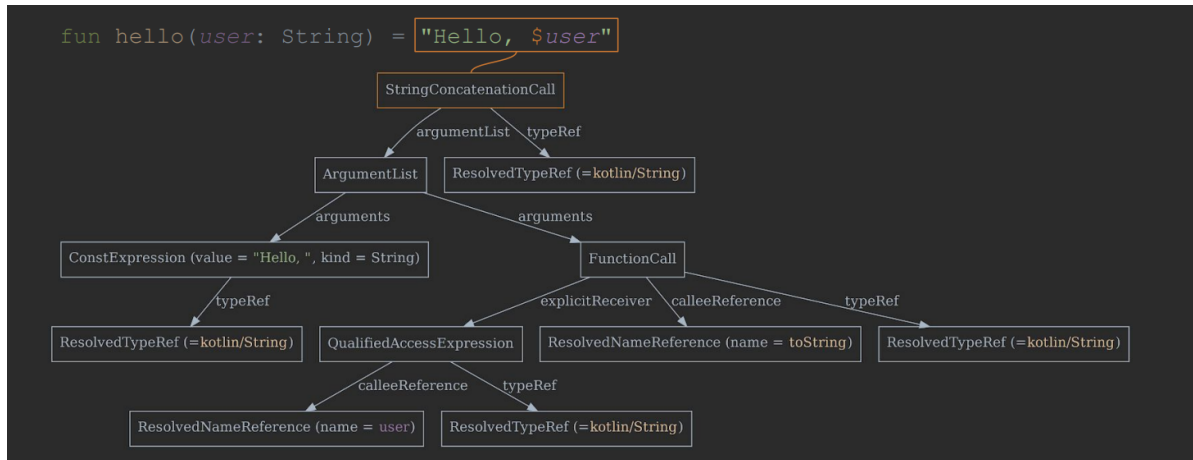


2. Front-end

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

Строится 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

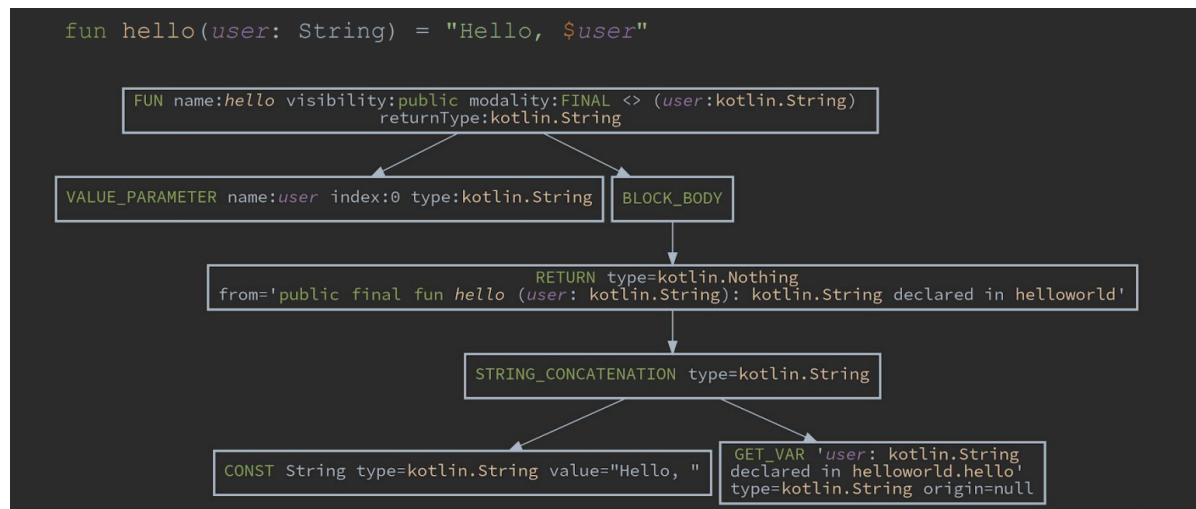
Всевозможные проверки, которые можно увидеть в идее. Диктат, детект -- это всё сюда.

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 конкретно, 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`.



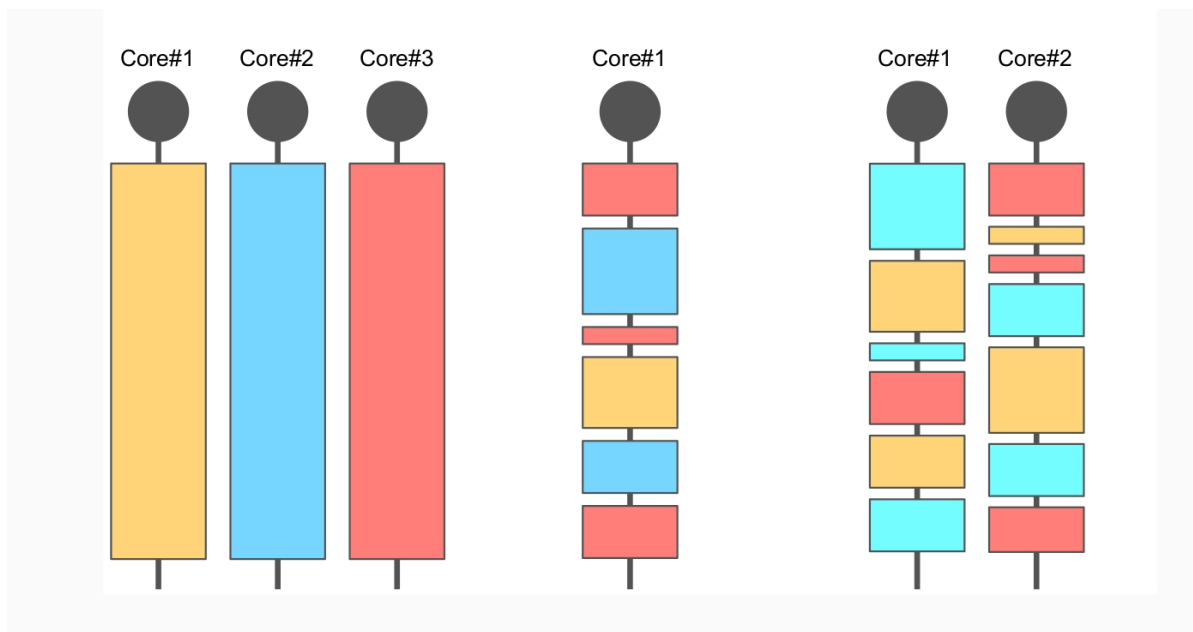
Плагины \todo

22-11-11

Parallelism & Concurrency

Parallel -- задачи на разных потоках могут выполняться одновременно

Concurrent -- есть прогресс на нескольких потоках



Preemptive & Cooperative scheduling

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 конкретного 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`:

```

1  @FunctionalInterface
2      public interface Runnable {
3      public abstract void run();
4  }
5
6  class RunnableWrapper(val runnable: Runnable)
7  val myWrapperObject = RunnableWrapper(object : Runnable {
8      override fun run() { println("I run") }
9  })
10 val myWrapperLambda = RunnableWrapper { println("yo") } // вместо
    аргумента передается лямбда

```

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

```

1  class MyThread : Thread() {
2      override fun run() {
3          println("${currentThread()} is running")
4      }
5  }
6
7  fun main() {
8      val myThread1 = MyThread()
9      myThread1.start()
10     val myThread2 = MyThread()
11     myThread2.run() // maybe be got blocked
12 }

```

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.

```
1 fun main() {
2     val myRunnable = Runnable { println("Sorry, gotta run!") }
3     val thread1 = Thread(myRunnable)
4     thread1.start()
5     val thread2 = Thread(myRunnable)
6     thread2.start()
7 }
```

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

`thread` помимо лямбды принимает аргументы.

```
1 fun thread(
2     start: Boolean = true,
3     isDaemon: Boolean = false,
4     contextClassLoader: ClassLoader? = null,
5     name: String? = null,
6     priority: Int = -1,
7     block: () -> Unit
8 ): Thread
```

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

Пример:

```
1 import kotlin.concurrent.thread
2
3 fun main() {
4     val thread1 = thread {
5         println("I start instantly")
6     }
7     val thread2 = thread(false) {
8         println("I don't start instantly")
9     }
10    thread2.start()
11    thread1.join() // явно выражена точка, в которой мы ждем `thread1`
12 }
```

Подробнее про `kotlin.concurrent.thread`

Поля `Thread()`:

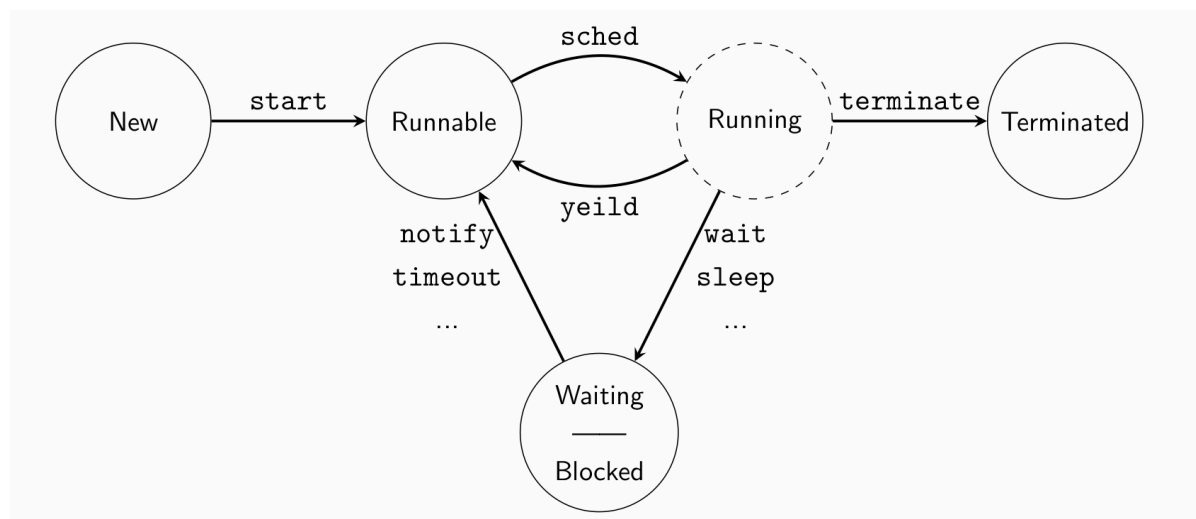
- `id: Long`
- `name: String?`
- `priority: Int` $\in [1, 10]$; чем больше, тем выше приоритет
- `daemon: Boolean`
- `state: Thread.state`
- `isAlive: Boolean`

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

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

Пример:

```
1 class LockedCounter {
2     private var c = 0
3     private val lock = ReentrantLock()
4
5     fun increment() {
6         lock.withLock {
7             c++
8         }
9     }
10
11     // same for other methods
12 }
```

Condition variables

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

`condition` говорит потоку, что ему нужно делать

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

```

3     private val lock = ReentrantLock()
4     private val condition = lock.newCondition()
5
6     fun increment() =
7         lock.withLock {
8             c++
9             condition.signal()    // wakes the thread up
10        }
11
12    fun decrement() =
13        lock.withLock {
14            if (c == 0)
15                condition.await() // `c` must be always positive
16            c--
17        }
18
19    fun value() = return lock.withLock { c }
20 }

```

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`

```

1    class Counter {
2        private var c = 0
3
4        fun increment() {
5            synchronized(this) { // так
6                c++
7            }
8        }
9
10       @Synchronized           // или так
11       fun decrement() {
12           c--
13       }
14   }

```

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

ReadWriteLock

Несколько потоков-читателей, но только один поток-модификатор

```
1 rwLock.readLock()    // возвращает интерфейс lock который захватывает readlock
2 rwLock.writeLock()   // возвращает интерфейс lock, захватывающий writelock
3
4 rwLock.read { ... }  // лямбда под readlock
5 rwLock.write { ... } // лямбда под writelock
```

Пример:

```
1 class RWLockedCounter {
2     private var c = 0
3     private val rwLock = ReadWriteReentrantLock()
4
5     fun increment() = rwLock.write { c++ }
6     fun value() = return rwLock.read { c }
7 }
```

JMM

Weak behaviors

Нет гарантированной последовательности внутри одного потока. Оптимизации 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 записать в регистр.

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

Пример раз:

```
1 class OrderingTest {
2     @Volatile var x = 0
3     @Volatile var y = 0
```

```

4      fun test() {
5          thread {
6              x = 1    // если переставить строки 6, 7, можно в строке 12
получить вывод '1, 0'
7              y = 1    // но строки не переставляются компилятором из-за
volatile
8          }
9          thread {
10             val a = y
11             val b = x
12             println("$a, $b")
13         }
14     }
15 }

```

Пример два:

```

1  class ProgressTest {
2      @Volatile
3      var flag = false
4      fun test() {
5          thread {
6              while (!flag) {}    // если flag не volatile, компилятор заменит
условие внутри while на true
7              println("I am free!")
8          }
9          thread {
10             flag = true
11         }
12     }
13 }

```

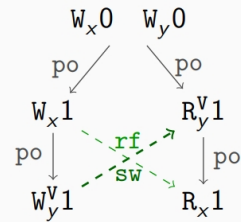
Happens-before relation

Порядок операций между потоками может быть разным. 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 $= a \rightarrow b$ --- 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, $hb = W_x1 \rightarrow R_x1$, потому что сначала идем по левой ветке, потом через **sw** попадаем в правую ветку; порядок определен.

Почему в данном случае не может быть $rf = W_x0 \Rightarrow R_x1$? Тогда это бы означало, что чтение из $y(R_y1)$ уже произошло. Значит, мы должны были пройти по левой ветке. Но мы ее пропустили, противоречие.

Когда возникает **sw** отношение?

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

```

1  class Counter {
2      @Volatile
3      private var c = 0
4      companion object {
5          private val updater =
6              AtomicIntegerFieldUpdater.newUpdater(Counter::class.java, "c")
7      }
8      fun increment() {
9          updater.incrementAndGet(this)
10     }
11     fun decrement() {
12         updater.decrementAndGet(this)
13     }
14     fun value(): Int = updater.get(this)
15 }
```

`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](https://github.com/kotlin-atomicfu/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`.

22-11-25

Мы сделали джобу, в которой очень мало работы. Но она запустила тысячу других джоб.

Может быть такое, что у кого-то из детей джобы вылетел `Exception` или кто-то из детей сказал ей: "Отменись. Всё. Я самый грустный ребёнок. Ты как родитель провалился, поэтому я тебя канселлю".

И тогда джоба, несмотря на то, что свою работу она закончила, и, вроде как, всё в порядке, из-за ошибки у кого-то из детей переходит в канселлинг. Иерархия позволяет джобе сказать: "Я не нормально завершаюсь. Я отменяюсь".

Предисловие

В синхронизированном программировании есть проблема: поток отправляет какой-то запрос, потом ждёт ответ от пользователя. Можно создавать дополнительные потоки, которые будут ждать пользователя вместо основного. Но не понятно, сколько таких потоков понадобится. Для решения проблемы используется асинхронное программирование.

Суть асинхронного программирования: в функцию передается инструкция, что делать после получения токена. При этом основной поток идёт дальше.

```
1 fun preparePostAsync(callback: (Token) -> Unit) { // тут `callback` --
    функция
2     // make request and return immediately
3     // invoke callback later
```

```

4  }
5
6  fun postItem(item: Item) {
7      preparePostAsync { token ->
8          submitPostAsync(token, item) { post ->
9              processPostAsync(post) {
10                 ...
11             }
12         }
13     }
14 }

```

Что происходит в `postItem`:

Вызывается функция `preparePostAsync`, которой передается лямбда. Внутри лямбды токен передается в функцию `submitPostAsync`, внутри которой тоже вызывается лямбда. И т.д.

Лесенка из скобочек на строках 11-14 называется **Callback Hell**.

Promise<T>

Сам по себе `callback` не работает в фоне. Нужно сообщать основному потоку, что данная функция будет работать на новом потоке, а основной поток может идти дальше.

Можно использовать интерфейс `Promise`. Тогда не будет необходимости, передавать инструкцию, что нужно делать с токеном. Операции промиса будут отложенными.

```

1  fun preparePostAsync(): Promise<Token> {
2      // make request and return a promise that is completed later
3      return promise;
4  }
5
6  fun postItem(item: Item) {
7      preparePostAsync()
8          .thenCompose { token -> submitPostAsync(token, item) }
9          .thenAccept { post -> processPostAsync(post) }
10 }

```

Что происходит в `postItem`:

`preparePostAsync()` возвращает результат, завернутый в класс `Promise`. Методы этого класса вызываются внутри `postItem`.

`.thenCompose` вызывает функцию `submitPostAsync`, возвращающую `post`.

`.thenAccept` принимает значение `post` и вызывает функцию `processPostAsync`.

Методы промисов создают новые промисы. Тратятся время и память.

suspend

Suspend у функции означает, что она может в какой-то момент приостановить исполнение. Другими словами, функция, которая может заблокировать поток, помечается suspend.

А ещё suspend функция может быть вызвана либо внутри корутинов, либо внутри других suspend функций.

```
1 suspend fun postItem(item: Item) {
2     val token = preparePost()
3     val post = submitPost(token, item)
4     processPost(post)
5 }
```

Под капотом превращается в

```
1 Object postItem(Item item, Continuation<Post> cont) { ... }
```

где `Continuation<in T>` -- интерфейс, работающий как callback функция.

Всё, что в теле `postItem`, передаётся в `cont` и работает параллельно с основным потоком. При этом функции внутри `postItem` должны быть помечены suspend, иначе компилятор не узнает, что во время исполнения этой функций можно уйти.

Вызовы внутри `cont` пронумерованы. Поле `cont.label` показывает, какая функция сейчас должна вызываться:

```
1 switch(cont.label) {
2     case 0: // suspend call 0
3         cont.label = 1;
4         preparePost(cont);
5         break;
6     case 1: // suspend call 1
7         Token token = (Token) prevResult;
8         cont.label = 2;
9         submitPost(token, item, cont);
10        break;
11    case 2: // suspend call 2
12        Post post = (Post) prevResult;
13        processPost(post);
14        break;
15 }
```

Это не large switch, а state machine. Запоминает, на каком вызове остановилась и какой результат был последний. После последнего кейса идёт спать.

Coroutine scoupe

Способ раз, немодный

Используем `runBlocking`, создающий coroutine scope. Означает, что то, что внутри `runBlocking`, работает, как callback функция.

```
1 fun main() = runBlocking { // this: CoroutineScope
2     launch { // launch a new coroutine and continue
3         delay(1000L) // non-blocking delay for 1 second
4         println("World!")
5     }
6     println("Hello") // main coroutine continues while the previous one is
    delayed
7 }
```

`launch` -- это Coroutine Builder. `lauch` работает, как suspend функция: параллельно запускает то, что внутри, а основной поток идёт дальше. По дефолту, второй поток запускается сразу.

!!! Если внутри `runBlocking` не написать `launch`, поток запустится как синхронный и не будет работать в фоне.

`launch` и `delay` не могут быть вызваны внутри мейна, потому что мейн не помечен suspend.

Пример

Тип `Job` взят из стандартной библиотеки `kotlin.coroutines`.

```
1 val jobs: List<Job> = List(1_000_000) {
2     launch(Dispatchers.Default
3         + CoroutineName("#$it")
4         + CoroutineExceptionHandler { context, error ->
5             println("${context[CoroutineName]?.name}: $error")
6         },
7         CoroutineStart.LAZY
8     ) {
9         delay(Random.nextLong(1000))
10        if (it % 10 == 0) { throw Exception("No comments") }
11        println("Hello from coroutine $it!")
12    }
13 }
14
15 jobs.forEach { it.start() }
```

Считаем, что вызов `jobs` происходит в coroutine scope.

Что делает `launch`: сначала поток спит, потом либо печатает своё имя, либо кидает исключение. Исключение обрабатывается штучкой `CoroutineExceptionHandler`.

`CoroutineStart.LAZY` -- запуск потока откладывается, пока не будет вызван метод `start()` в строке 15.

Способ два, модный

Используем интерфейс `CoroutineScope`. У него есть `CoroutineContext`, отвечающий за элементы внутри скоупа. Элементом может быть что угодно: функция, объект, имя объекта, и т.д.

У одного `CoroutineScope` может быть несколько `CoroutineContext`.

```

1  public interface CoroutineScope {
2      public val coroutineContext: CoroutineContext
3  }
4
5  public interface CoroutineContext {
6      public operator fun <E : Element> get(key: Key<E>): E?
7      ...
8      public interface Element : CoroutineContext {
9          public val key: Key<*>
10         ...
11     }
12 }
```

Можно сказать, что контекст -- это мапа `<Key<Element>, Element>`, в которой хранится всё, что нужно для управления корутиной.

Тип Job

Наследуется от `Element`, чтобы быть частью `CoroutineContext`.

По конвенции, во всех контекстах должен быть `Job`.

```

1  public interface Job : CoroutineContext.Element {
2      public companion object Key : CoroutineContext.Key<Job>
3      public fun start(): Boolean
4      public fun cancel(cause: CancellationException? = null)
5      public val children: Sequence<Job>
6      ...
7  }
```

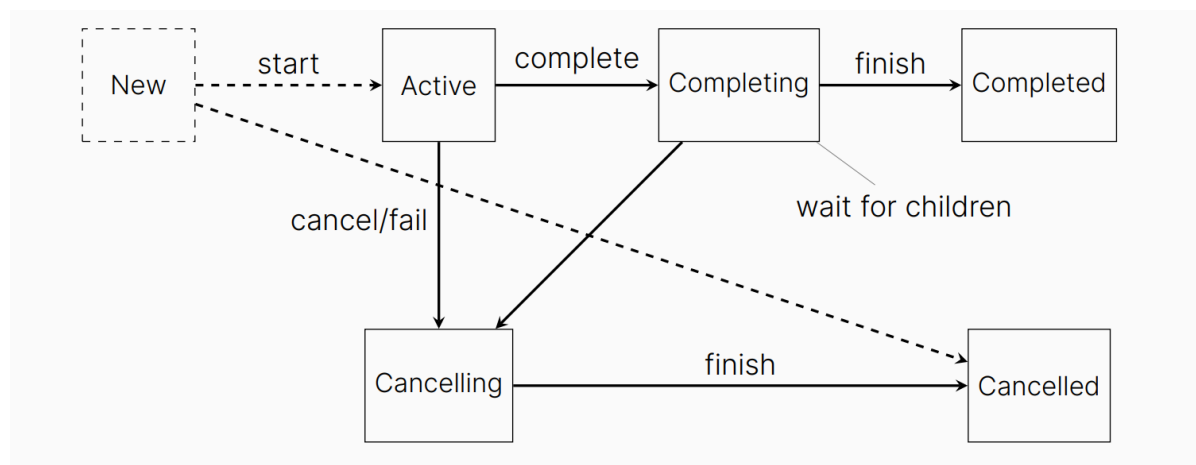
Генерируется ключ `Key`, который будет общим для всех экземпляров `Job`.

`start()`, чтобы запустить корутину, если она была `CoroutineStart.LAZY`. Возвращает `false`, если корутина уже запущена или уже отработала.

Между интерфейсами `Job` есть иерархия. Джоб-родитель ждёт своих джоб-детей, прежде чем завершить работу.

Пример: вот есть `runBlocking{}`. У него свой `Job`. Мы внутри запускаем `launch` и создаём корутину. У корутины будет уже свой `Job`, являющийся ребёнком первого `Job`.

Job state



По умолчанию, корутина запускается сразу после создания. Чтобы отложить запуск, используется флаг `CoroutineStart.LAZY`.

Active: корутина еще не дошла до последнего кейса в state machine.

Completing: корутина закончила работу, но её дети нет.

Cancelling: у какого-то ребенка произошла ошибка или вылетел exception, и корутина отменяет остальных детей и отменяется сама.

22-12-02

Диспатчеры

Часть контекста. Есть функция `dispatch`, принимает контекст и блок.

Диспатчер запускает поток, когда его `cont.label = 0`, и не прерывает его.

```

1 public abstract class CoroutineDispatcher : ... {
2     ...
3     public abstract fun dispatch(context: CoroutineContext, block: Runnable)
4 }
```

Контексты диспатчера

`Dispatchers.Default` -- кол-во потоков = кол-во ядер процессора (при этом минимум всегда 2)

`Dispatchers.IO` -- минимум 64 потока, большинство потоков находятся в состоянии `blocked`.

`Dispatchers.Main` -- поток, использующийся для UI. Есть объявление в библиотеке `kotlinx.coroutines`, но без реализации. Реализация появится с подключением другой либы. Ипользуется, чтобы приложение не замирало: нажимались кнопки и т.д. События от пользователя перекидываются на `Dispatchers.Default` или `Dispatchers.IO`.

`Dispatchers.Unconfined` -- обычно не используется. Если встречается в коде, что-то пошло не так.

Диспатчер под капотом

Есть две глобальные lock-free очереди, в которых хранятся задачи.

У шедьюлера есть атомик массив, состоящий из штук типа `worker`. Воркер -- это, по сути, поток. У него есть свой собственный массив задач, а также метод `findTask`. Отличие воркера от обычного потока в том, что шедьюсер не говорит воркеру, что делать. Воркер сам идет к глобальной очереди и ищет задачу.

Если воркер не нашел задачу в глобальной очереди, он начинает выполнять заблокированные процессы

Переключение контекста

```

1 suspend fun preparePost(): Token = withContext(Dispatchers.IO) { ... }
2
3 suspend fun processPost(post: Post) = withContext(Dispatchers.Default) { ...
4 }
5 suspend fun postItem(item: Item) {
6     val token = preparePost()
7     val post = submitPost(token, item)
8     processPost(post)
9 }
10
11 viewScope.launch { postItem(someItem) }

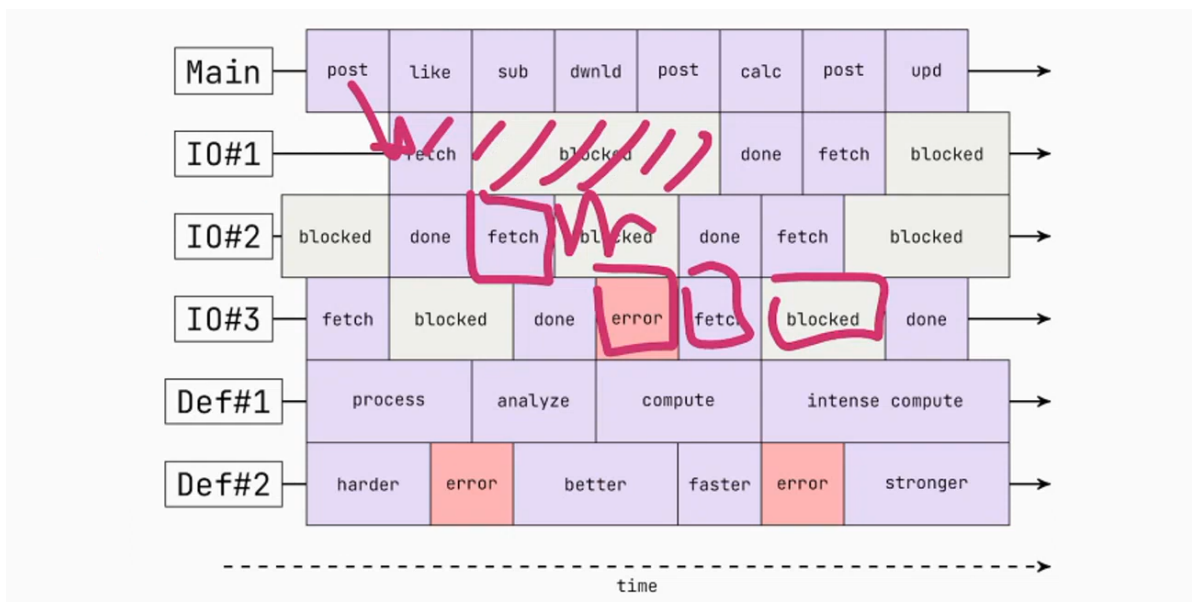
```

Как происходит переключение контекста?

Кол-во IO потоков около 64. Кол-во Default потоков = кол-ву ядер.

IO поток начинает работу и блокируется. Тогда следующая задача отдается другому свободному IO потоку.

Если в потоке выкидывается ошибка, поток не умирает. Эту ошибку обрабатывает `CoroutineExceptionHandler`, а поток идет дальше.



Сравнение корутины и thread'a

Запускаем `launch`, создаётся объект `Job`, работающий в фоне.

С помощью `delay` корутина говорит "Разбудите меня через столько-то времени".

```
1 fun main(): Unit = runBlocking {
2     repeat(1_000_000) { // it: Int
3         launch { // new asynchronous activity
4             delay(1000L)
5             println("Hello from coroutine $it!")
6         }
7     }
8 }
```

Запускаем `thread`, получаем объект `Thread`

```
1 fun main(): Unit {
2     repeat(1_000_000) { // it: Int
3         thread { // new thread
4             sleep(1000L)
5             println("Hello from thread $it!")
6         }
7     }
8 }
```

Получаем `Exception: unable to create native thread: possibly out of memory or process/resource limits reached`, потому что 1'000'000 потоков это слишком много.

Исключения

В этот момент корутина понимает, что всё, она сломалась, она продолжать свою работу нормально не может. Произошла исключительная ситуация. Она `terminated`, `cancelled`, `smert'`.

После этого она сообщает своему родителю: "У меня произошло исключение, мне очень плохо, можешь с этим что-нибудь сделать?" В этот момент `SupervisorJob`, поскольку он супер-крутой, говорит: "Я ничего не буду делать, давай сам". Вызовется exception-handler у корутины, обработается ошибка. Дальше корутина мертва, `supervisorJob` сидит довольный и говорит: "Меня проблемы моих детей вообще не интересуют".

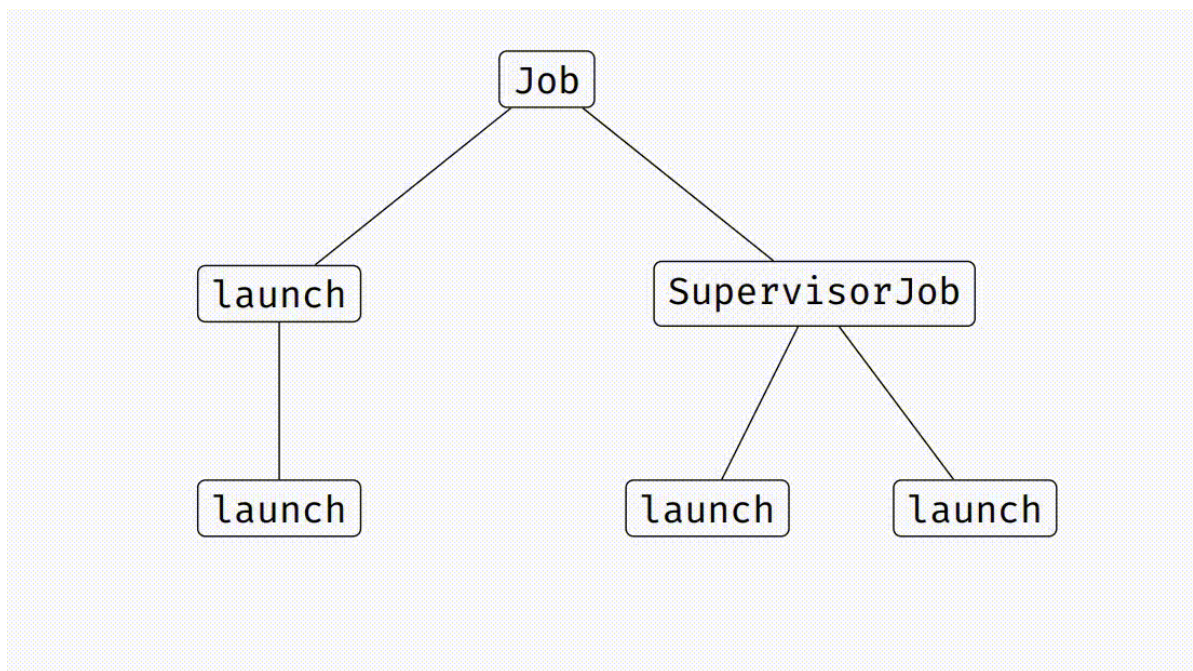
Теперь представим, что у нас есть джоба, у нее есть `launch`, у которого есть exception-handler. У этого `launch` есть еще один `launch`. У нижнего `launch` выкидывается ошибка. Корутина умирает и говорит своему родителю: "Сделай с этим что-нибудь, пожалуйста". Ваше ожидание, естественное, человеческое, абсолютно понятное и интуитивно правильное -- это что `launch`-родитель скажет: "Вот у меня есть handler, щас разберемся". А `launch` делает не так. Он такой: "Я тоже пожалуй умру. Очень грустно, что умер мой

ребенок. Это абсолютно devastating новость для меня, поэтому я попрошу своего родителя что-нибудь с этим сделать".

В этот момент вы такие: "А handler у него же есть". А вот так вот это работает. Ну есть handler и есть, какая разница. Дальше джоба получает это исключение от launch'a и сообщается другим детям: "Слушайте, у вас брат умер (или сестра)". `supervisorJob` тоже от этого немного в шоке и говорит своим детям: "У вас дедушка умер (или бабушка). Или дядя (тетя). Не знаю, в общем, какой-то родственник, далекий очень, но это вас убивает".

Последний launch умирает, `supervisorJob` умирает, и после этого все мертвы.

Корутины работают так. `supervisorJob` не даст никому умереть. А если обычная джоба и в каком-то launch вылетело исключение, наступит тотальная и бесповоротная смерть всего.



Конвенция

Если есть функция, внутри которой вызывается `launch`, она должна быть как-то помечена.

```

1 // работает долго и может быть прервана
2 suspend fun work(...) { ... }
3
4
5 // используется расширение CoroutineScope:
6 // способ 1:
7 fun CoroutineScope.backgroundWork(...) {
8     launch { ... }
9 }
10 // способ 2:
11 fun CoroutineScope.moreWork(...): Job = launch { ... }
  
```

Доп ссылки

[код с презентаций](#)

[либа kotlinx.coroutines](#)