

Разработка мобильных приложений

Лекция 4.2

Kotlin

Ключев А.О. к.т.н., доцент ФПИиКТ Университета ИТМО

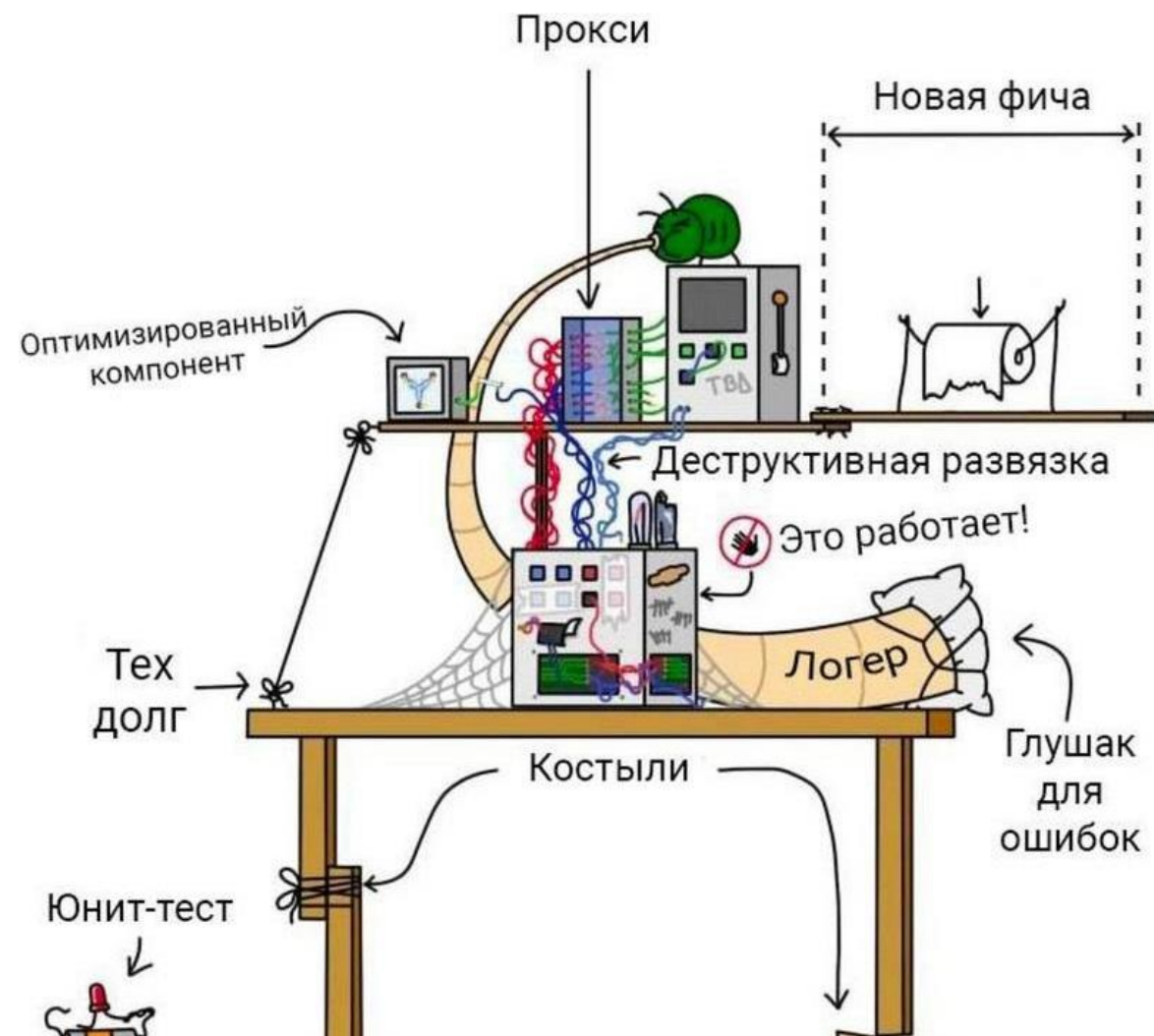
Санкт-Петербург

2025

Литература и видео

- <https://kotlinlang.org>
- Примеры исходных текстов и ссылки на видео по Kotlin
<https://github.com/kluchev/rmp-kotlin-lecture>
- Кауфман В.Ш . Языки программирования. Концепции и принципы
- Ссылки на дополнительную литературу и видео можно найти в слайдах

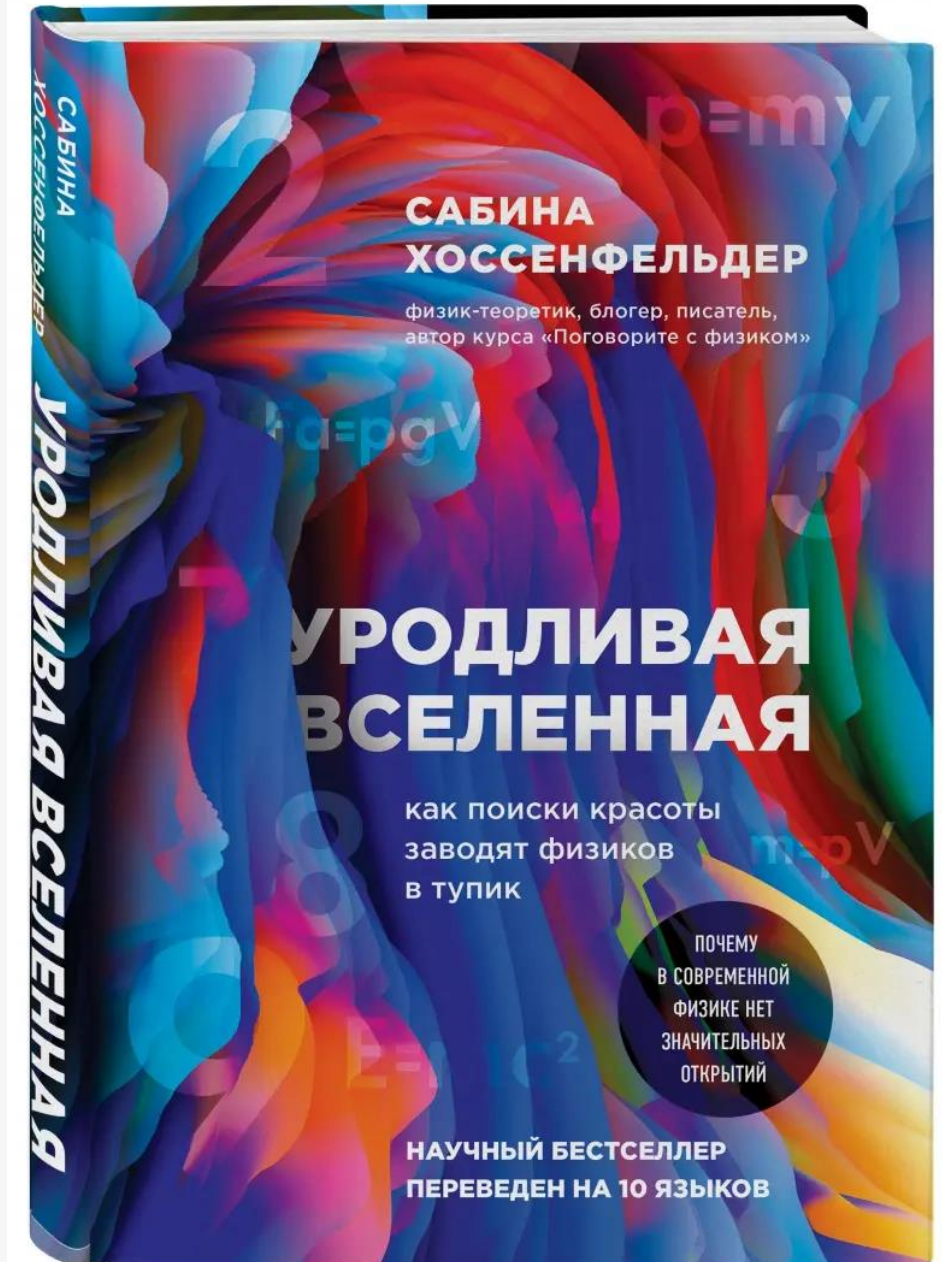
Рабочий проект обычно выглядит так...



Что такое программирование?

- Программирование – процесс превращения идеи из головы разработчика в формальное описание плана работ исполнителя
- Исполнителем может быть кто или что угодно, например, процессор или человек
- Формализация – сведение содержания к форме (например, идеи к программе или схеме электрической принципиальной), то есть это отображение результатов мышления в точных понятиях и утверждениях.

Математика позволяет ошибиться, но не позволяет солгать (С) Сабина Хоссенфельдер, Уродливая вселенная



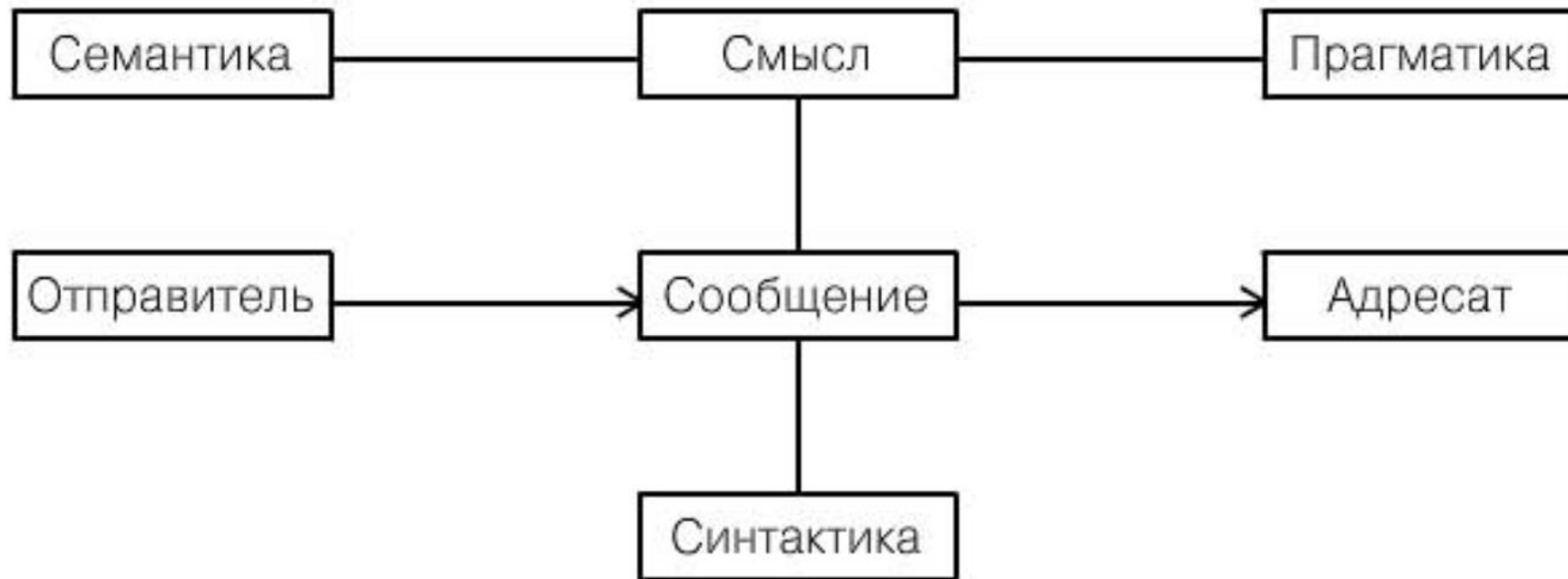
Что такое язык программирования?

- Язык программирования это инструмент для планирования поведения процессора
- Поведение процессора на самом низком уровне зависит от используемой в нем модели вычислений. За счет метасистемных переходов (см. [В.Ф. Турчин, «Феномен науки»](#)) возможно создание новых уровней абстракции с новыми моделями вычислений, например с сетями процессов или конечными автоматами.
- Процесс происходящий в процессоре – вычислительный процесс



Модель передачи сообщений

(семиотика – теория знаков и знаковых систем)



См. Кауфман В.Ш . Языки программирования. Концепции и принципы

Семантика, прагматика, синтактика

- Семантика изучает смысловое значение языковых конструкций. Семантика статична и не зависит от контекста.
- Прагматика показывает отношение между знаковыми системами и теми, кто их использует (показывает условия использования языковых знаков). Прагматика зависит от контекста и участников.
 - Фраза «Иди нафиг!» сказанная своему другу и начальнику на работе может иметь совершенно разные последствия. Также результат может меняться в зависимости от условий получения этой фразы одним и тем же адресатом.
- Синтактика (синтаксис) – совокупность правил, определяющих структуру предложений на языке.

Как происходит процесс программирования?

- **Декомпозиция** – разделение на элементы: уровни, модули, классы, функции, процессы, состояния;
- Обратное действие декомпозиции – **интеграция**.
- **Абстракция** – выделение главного.
- **Моделирование** – создание упрощенного представления реального мира.
- **Оптимизация** – улучшение эффективности решения
- **Итерация** – постепенное улучшение/уточнение решения
- **Обобщение** – создание универсальных решений
- **Интерпретация** – перевод неформальных требований в формальное описание задачи
- **Управление сложностью** – борьба с хаосом
- **Предвидение** – учет будущих изменений и возможных проблем

Программирование и архитектура

- Когда вы программируете вы представляете ЧТО вы хотите сделать и КАК это сделать.
- Если задача сложная, то вы можете попытаться сохранить часть идей в виде архитектурного описания на бумаге, чтобы думать о задаче по частям.
- Если у вас есть большой опыт и вы достаточно долго работаете над сложными проектами, вы можете держать архитектуру в голове и сразу писать код на каком-то языке программирования.

Языки и уровни абстракции

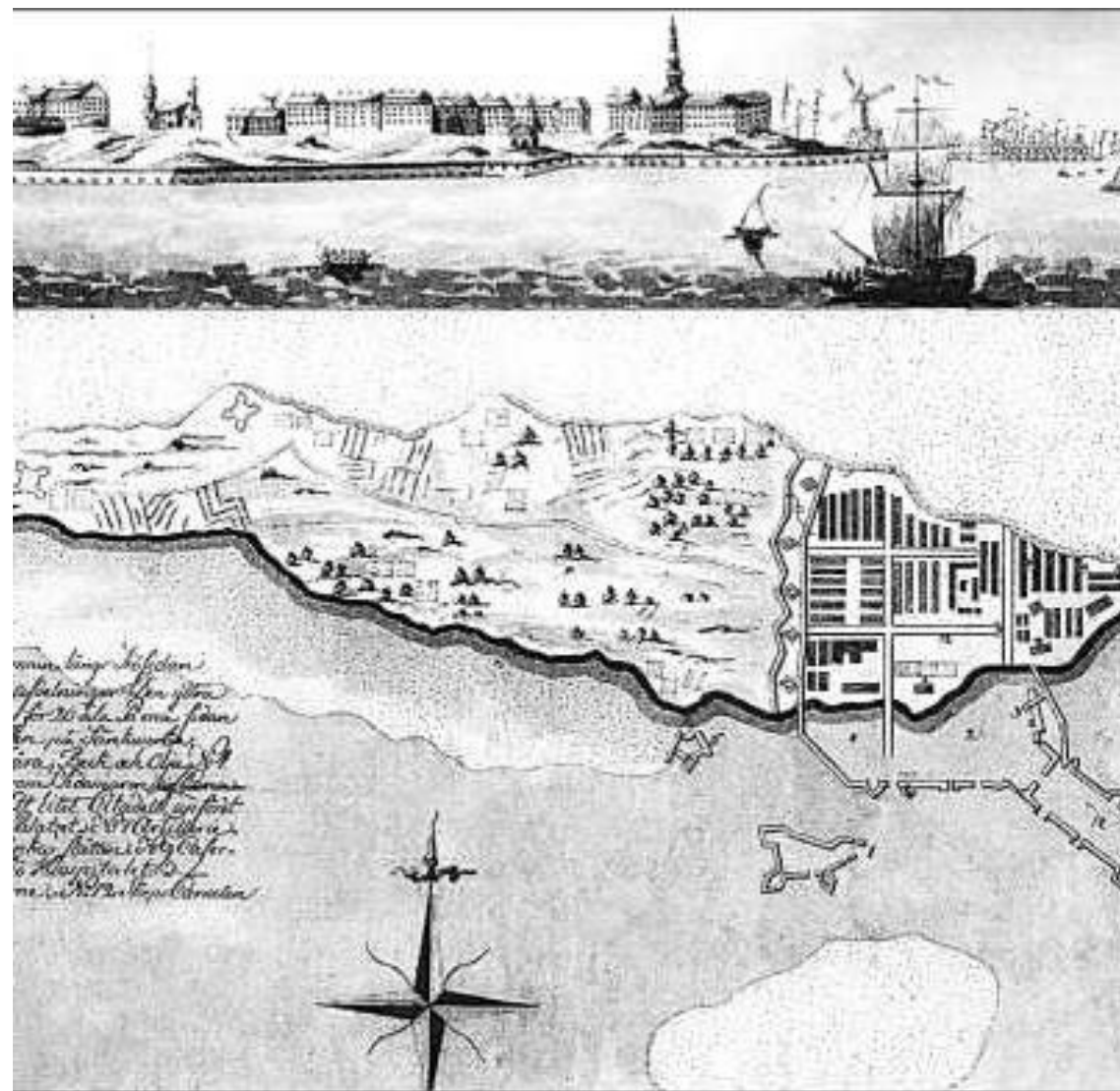
- Чем выше уровень абстракции языка, тем больше архитектурного проектирования переносится на процесс программирования.
- Пример языка низкого уровня – ассемблер, в нем уровень языка равен уровню команд процессора.
- Пример языка высокого уровня – Kotlin, который мы сегодня рассмотрим.

О языке Kotlin

- Начало разработки - 2010 г.
- В 2012 г. открыты исходники (лицензия Apache)
- В 2016 г. - релиз 1.0. Очень долгий выход к релизу объясняется тем, что язык обкатывался, из-за того, что изменения в релизную версию языка вносить почти невозможно. Пример быстрого выхода в релиз - JavaScript.
- 2017 год - язык Kotlin и IntelliJIDEA включены в Android Studio, релиз версии 1.2
- 2025 - Latest stable version: 2.1.20 (20.03.2025)
- <https://github.com/JetBrains/kotlin>

Откуда взялось название Котлин?

Котлин - остров в Финском заливе Балтийского моря, в 30 километрах западнее центра Санкт-Петербурга. В восточной части острова располагается город Кронштадт.



Применение Kotlin

- Kotlin – прагматичный (удобный на практике), безопасный язык с сильной статической типизацией для мультиплатформенных приложений.
 - Программирование серверных платформ (Backend)
 - Spring, Ktor (JVM, нативный код, JavaScript)
 - Кроссплатформенное программирование мобильных систем (Android, iOS)
 - Программирование Android (ART)
 - Программирование Web-приложений (JavaScript)

Базовые концепции языка программирования Kotlin

- Машина Фон-Неймана
 - Принцип программного управления, принцип адресности, принцип однородности памяти
 - Последовательное выполнение операторов, условные ветвления
- ООП
 - Классификация
 - Классы, объекты, интерфейсы, инкапсуляция, перегрузка методов
- Сети процессов
 - Потoki, корутины, семафоры, критические секции, очереди
- Функциональное программирование
 - Лямбда выражения (анонимные функции), элвис операторы, фильтры, рекурсия,...
- Автоматное программирование
 - When

Kotlin vs Java

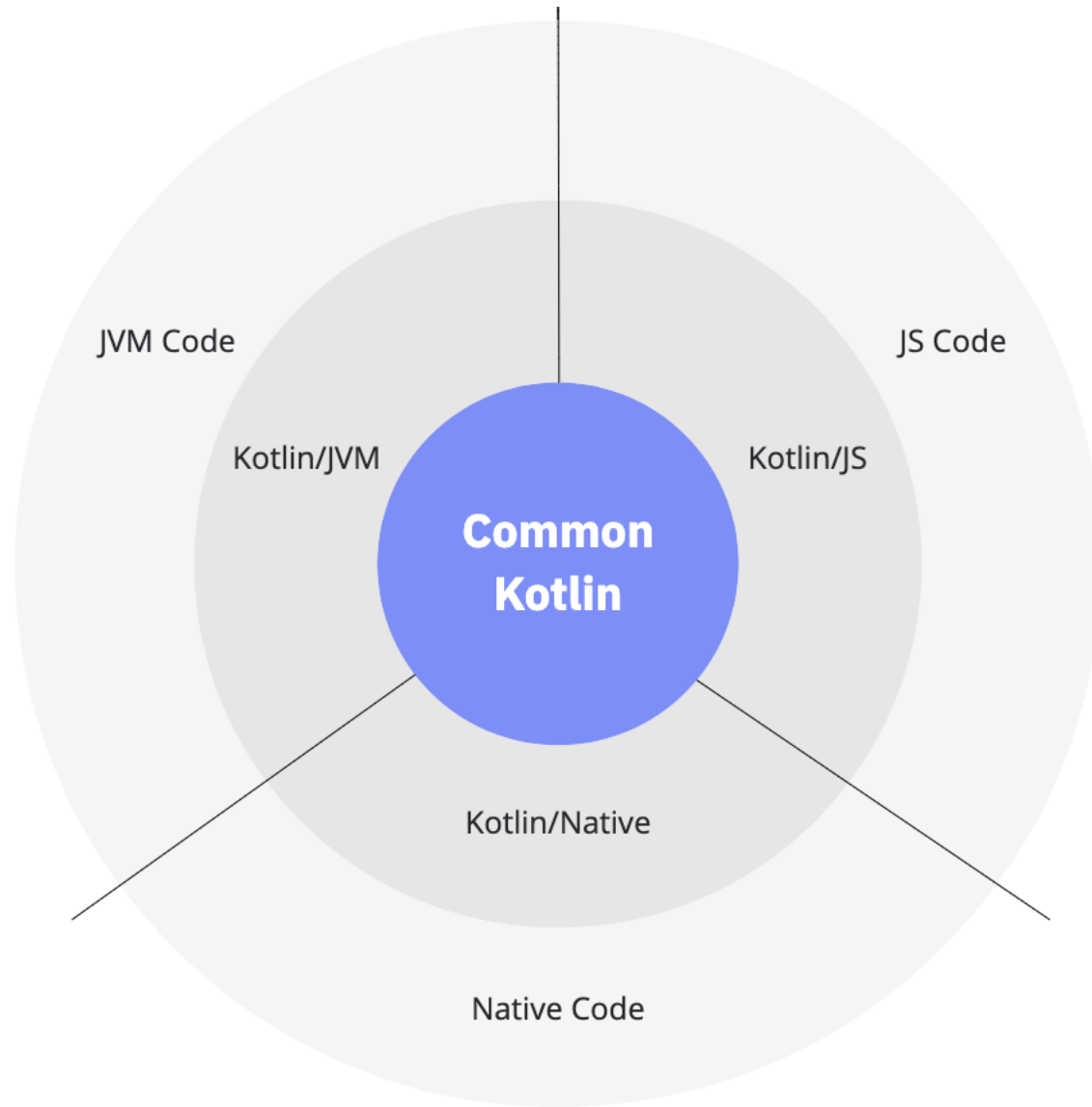
- Kotlin более лаконичен чем Java (примерно на 40%), более читаем (меньше кода и много синтаксического сахара) и более выразителен.
- Kotlin является типобезопасным (null-безопасность)
- Умное приведение типов (smart casts)
- Функции высшего порядка (лямбда-выражения)
- Лямбда-выражения с получателями
- Функции-расширения (для добавления новых методов в имеющиеся классы)
- Поддержка создания DSL (пример – Gradle)
- см. [Илья Матвеев. Kotlin вообще и Native в частности](#)

Kotlin vs Go (серверная разработка)

- Преимущества Kotlin
 - Красивый. Kotlin – «идеальная» Java, которая как известно – «написал один раз, запускай везде».
 - Выше уровень абстракции
 - Доступны все библиотеки Java (для JVM)
 - Кроссплатформенный (серверные платформы, мобильные системы, Web, embedded (ограниченно))
- Преимущества Go
 - Быстрый
 - Лаконичный
 - Низкоуровневый (корни Си сказываются), следовательно большие программы на нем писать труднее, чем на Kotlin, C# или Java.
 - Язык можно любить только за одно то, что в проекте участвовал Кен Томпсон, один из авторов языка Си и ОС Unix.
- Я планирую использовать Kotlin на начальных этапах разработки, постепенно переписывая на Go куски бекэнда, которые являются бутылочным горлышком. Как известно, сперва нужно сделать систему, а уже потом заниматься оптимизацией.
- **Личное мнение.** *Kotlin довольно комфортный язык, но Kotlin поверх JVM медленно работает. Нативный Kotlin не позволяет простым способом подключать библиотеки, что ограничивает его использование. IntelliJ IDEA – довольно сложная IDE, в которой регулярно возникают разные проблемы. Ну и не забываем про невозможность использования платных версий IntelliJ IDEA и обновления плагинов без танцев с бубном.*

Цели разработки Kotlin

- Увеличение продуктивности программиста
 - Лаконичность кода приводит к меньшему количеству ошибок
 - Выразительность языка повышает качество жизни программиста, ему приятнее и интереснее заниматься процессом программирования
 - Демократичность, возможность применения широкими кругами программистов (не хуже Java, проще Scala и удобнее их обоих). По мнению программистов Scala слишком заумный язык и на нем опытные программисты скорее выпячивают свое мастерство, чем решают практические задачи.
 - Хорошие и удобные инструментальные средства (IDE, отладка, компиляция, сборка)
- Расширение интероперабельности (JVM, C/C++, JS,...)
- Повышение безопасности кода



Kotlin Multiplatform



Kotlin Multiplatform

- Общий код разделяется между платформами (Android, iOS, Linux, MacOS X, Windows,...)
- В платформенном коде реализуются специфические действия для работы с API конкретной платформы
- Все это происходит в рамках единого инструментария
- Поддержка CPU:
 - ARM32
 - ARM64
 - X86
 - X86 X64
 - MIPS32
 - MIPSEL32
 - WASM32

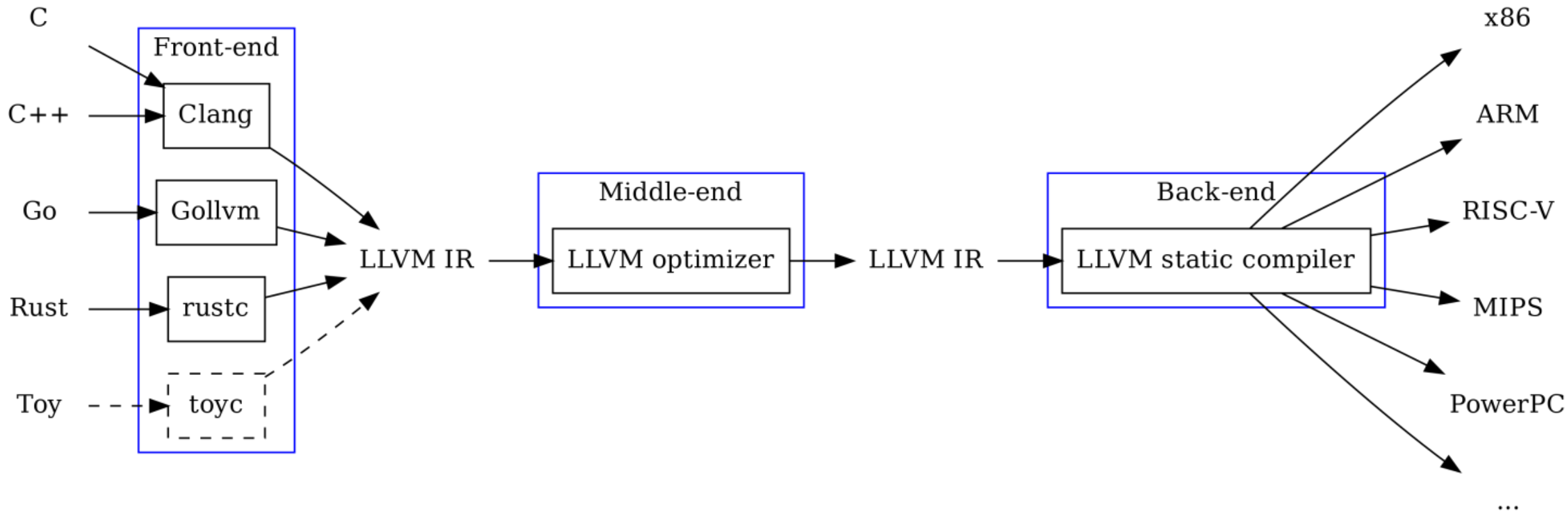
Kotlin Native

- Компилятор Kotlin компилирует в LLVM (см. llvm.org).
 - Проект LLVM представляет собой набор модульных и повторно используемых технологий компиляции и инструментов.
 - LLVM Core - оптимизатор, CLang – C/C++/Objective C компилятор, LLDB – отладчик, libc ++- библиотеки и т.д.
- **Нет интеропа с Kotlin/JVM**
- <https://github.com/JetBrains/kotlin/tree/master/kotlin-native>

Устройство компилятора (кратко)

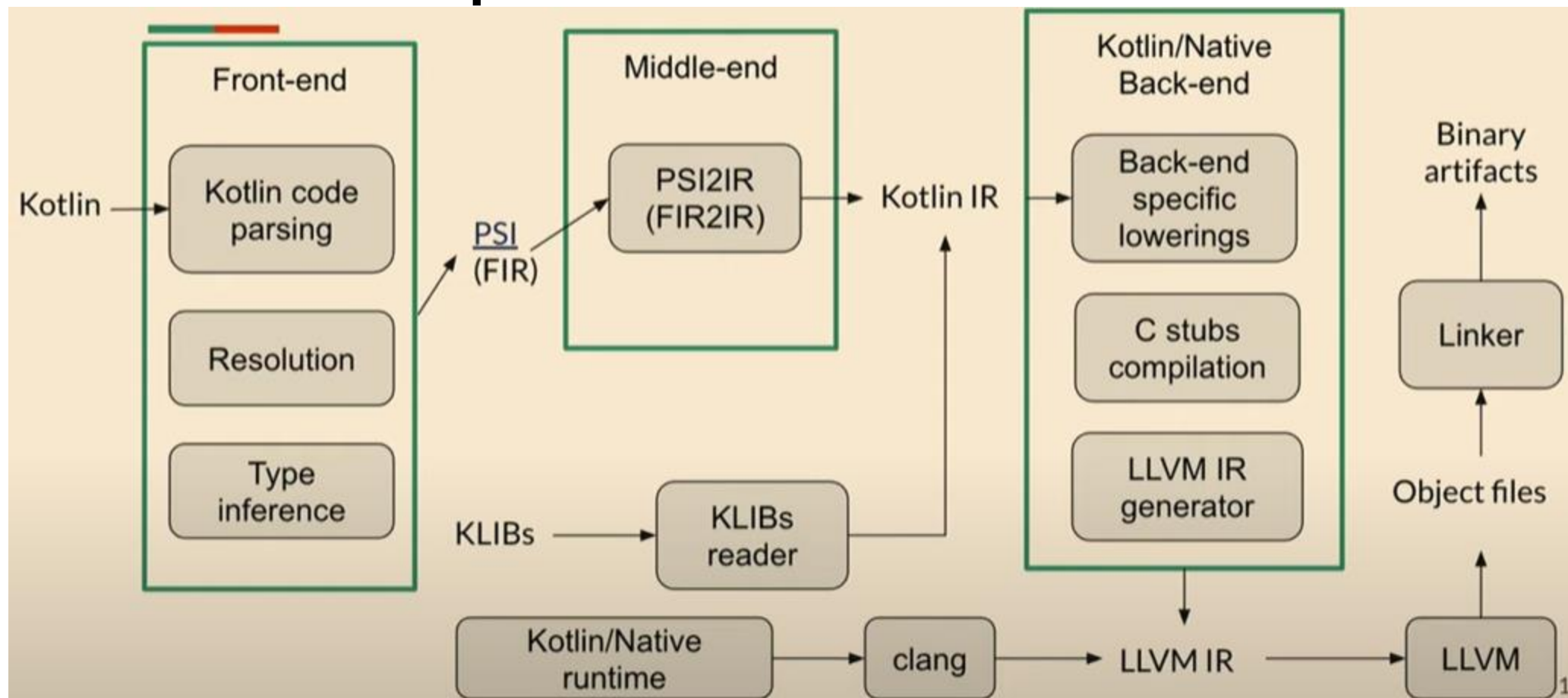
- Фронтэнд
 - Лексический анализатор
 - Синтаксический анализатор
 - Семантический анализатор и контроль типов
 - На выходе – промежуточный код в какой-либо форме
- Бэкэнд
 - Оптимизатор
 - Генератор кода

LLVM – просто название, не аббревиатура (нет там никаких VM)



LLVM IR — низкоуровневое промежуточное представление(
intermediate representation)

Компилятор Kotlin-native



См. [Елена Лепилкина - Kotlin/Native: между LLVM и VM](#)

Интероперабельность

- Интероперабельность – способность взаимодействия с другими продуктами, например, вызывать библиотеки, написанные на C++ из программы на Kotlin.
- Kotlin-native – позволяет использовать обычные и динамические **нативные** библиотеки
 - C/C++
 - Objective C
 - [Interoperability with C](#)
 - См. [Елена Лепилкина - Kotlin/Native: между LLVM и VM](#)
 - [Use C Libraries in Kotlin Native Code \(C Interop\) for C Noobs](#)
- Если вы хотите ускорить код и организовать интероперабельность с JVM используйте Kotlin + GraalVM

Зачем все это?

- Работа со специфическими платформами (мобильные системы бывают не только на базе iOS и Android)
 - Программирование iOS + Android + Linux + Windows + ...
 - Встроенные системы – очень популярная тема, особенно Linux для встроенных (embedded) применений (посмотрите вакансии на hh.ru).
 - Web (WebAssembly, WASM – см. <https://webassembly.org/>) – виртуальная машина для работы внутри браузеров, а также внутри RTOS (например, [Zephyr](#)).
- Реализация алгоритмов на высокоуровневом языке (Kotlin) при возможности работы с нативными системными вызовами и библиотеками.
- Оптимизация производительности, использования памяти, времени запуска
- Избавление от JVM:
 - во встроенных приложениях на базе Linux, с высокими требованиями к производительности и небольшими ресурсами, например STM32MP1
 - В критически важных приложениях, с целью уменьшения прослойки различных виртуальных машин и библиотек.

Специфика Kotlin-native

- Собственный рантайм с автоматическим сборщиком мусора (это расширяет футпринт и вносит изменение времени исполнения программы)
- Поддержка основных концепций Kotlin
- В библиотеку Kotlin-native включены платформенные библиотеки:
 - POSIX
 - Linux
 - Windows
 - ...
- Ограниченная поддержка рефлексии*
- Нет оптимизации кода на этапе исполнения
- *Рефлексия — набор возможностей языка и библиотек, позволяющий интроспектировать программу (обращаться к её структуре) во время её исполнения.*

Сборка мусора в Kotlin-native

- Реализована в рантайме (на LLVM IR) для того, чтобы обеспечить совместимость с AppStore Apple, в котором запрещается выкладывать код сделанный не с помощью LLVM (форк от Apple) в целях обеспечения безопасности.
- Сборка мусора реализована на двух алгоритмах:
 - Подсчет ссылок (reference counting)
 - Сборка циклического мусора (trial deletion algorithm)
- [kotlin-stdlib / kotlin.native.runtime / GC](#)
- См. [Roman Elizarov. Kotlin/Native Memory Management Update](#)

Производительность (сериализация объекта в JSON)

- Kotlin - 16 mks
- Kotlin-native - 2 mks, – примерно в 8 раз быстрее Kotlin
- Go Lang – 0,25 mks, примерно в 64 раза быстрее Kotlin

```
class SerializationTest {  
    fun toJson() {  
        val taskResult = TaskResult(  
            task: "testTask", errorCode: 12345,  
            errorMessage: "test1",  
            buf: "test2"  
        )  
        var dt : Long = 0  
        var json = ""  
        dt = measureNanoTime {  
            json = Json.encodeToString(taskResult)  
        }  
        println("dt = ${dt/1000} mks, json: $json")  
    }  
}
```

```
func main2() {  
    r := Result2{"testTask", 12345, "test1", "test2"}  
  
    for i := 0; i < 10; i++ {  
        start := time.Now()  
  
        b, _ := json.Marshal(r)  
  
        duration := time.Since(start)  
        fmt.Println(duration)  
        fmt.Println(string(b))  
    }  
}
```

Обычный Kotlin. Эффект прогрева

```
> Task :test
dt = 17517 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
dt = 2362 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
dt = 23 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
dt = 17 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
dt = 17 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
dt = 16 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
dt = 15 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
dt = 18 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
dt = 15 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
dt = 16 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
BUILD SUCCESSFUL in 9s
5 actionable tasks: 5 executed
11:55:16: Execution finished ':test --tests "TestSerialization.test1"'.

```


Kotlin-native

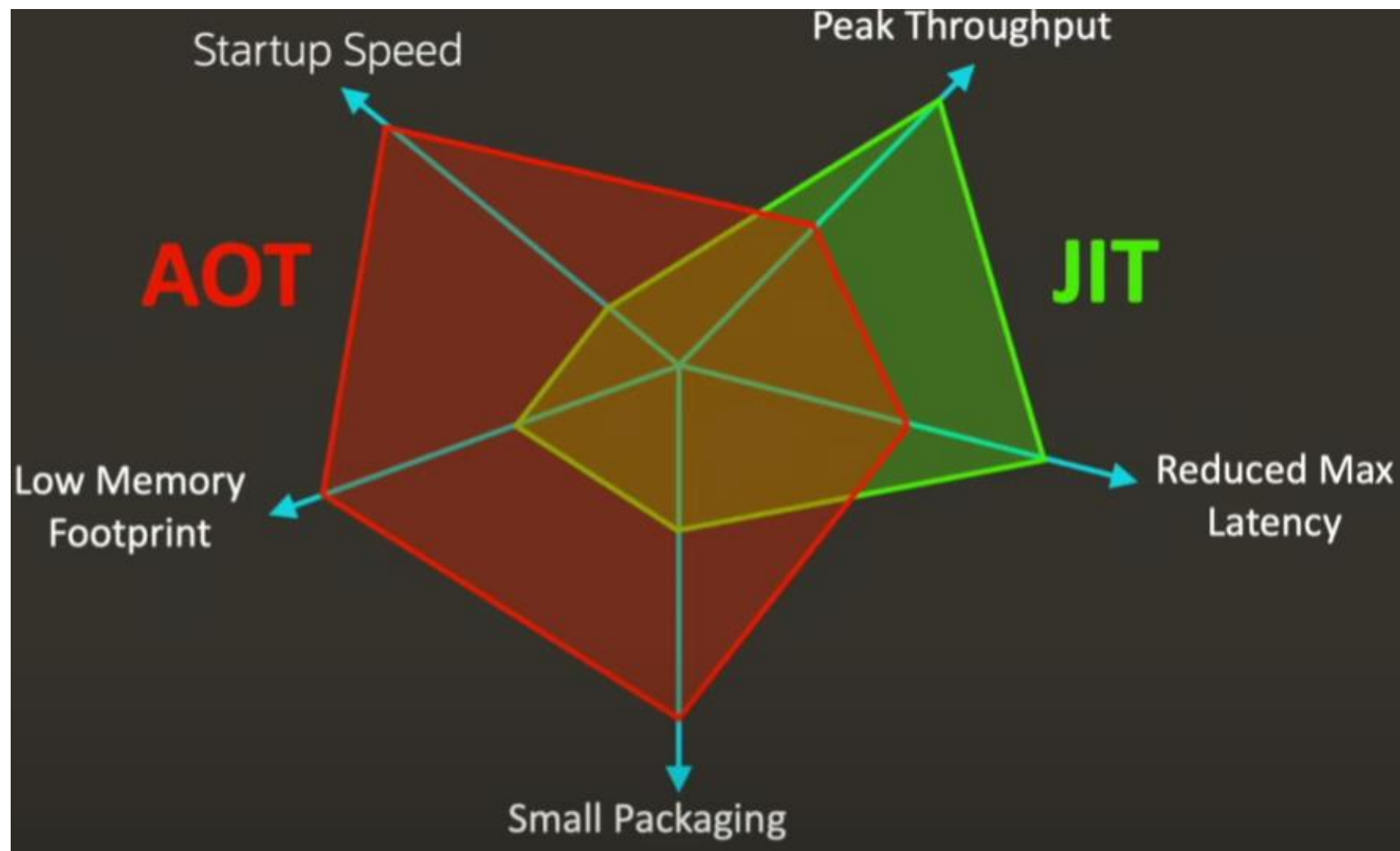
```
dt = 156 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
dt = 3 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
dt = 2 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
dt = 2 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
dt = 2 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
dt = 2 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
dt = 2 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
dt = 3 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
dt = 2 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
dt = 2 mks, json: {"task":"testTask","errorCode":12345,"errorMessage":"test1","buf":"test2"}
```

Go

```
101.583µs
{"Task": "testTask", "ErrorCode": 12345, "ErrorMessage": "test1", "Buf": "test2"}
584ns
{"Task": "testTask", "ErrorCode": 12345, "ErrorMessage": "test1", "Buf": "test2"}
250ns
{"Task": "testTask", "ErrorCode": 12345, "ErrorMessage": "test1", "Buf": "test2"}
250ns
{"Task": "testTask", "ErrorCode": 12345, "ErrorMessage": "test1", "Buf": "test2"}
208ns
{"Task": "testTask", "ErrorCode": 12345, "ErrorMessage": "test1", "Buf": "test2"}
208ns
{"Task": "testTask", "ErrorCode": 12345, "ErrorMessage": "test1", "Buf": "test2"}
209ns
{"Task": "testTask", "ErrorCode": 12345, "ErrorMessage": "test1", "Buf": "test2"}
208ns
{"Task": "testTask", "ErrorCode": 12345, "ErrorMessage": "test1", "Buf": "test2"}
250ns
{"Task": "testTask", "ErrorCode": 12345, "ErrorMessage": "test1", "Buf": "test2"}
250ns
{"Task": "testTask", "ErrorCode": 12345, "ErrorMessage": "test1", "Buf": "test2"}
11.1µs
```

GraalVM + Kotlin

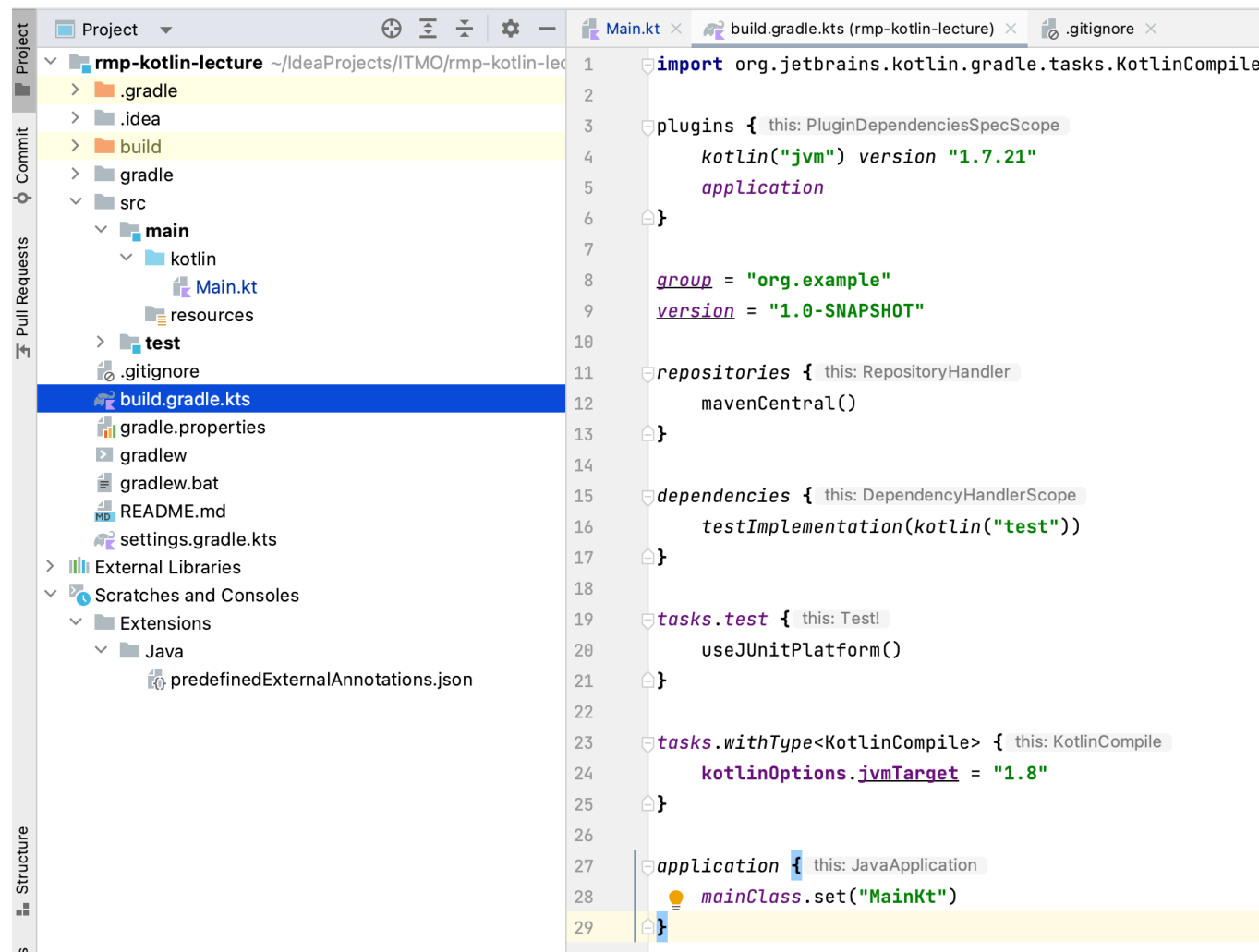
[KotlinConf 2019: The best runtime for Kotlin is obviously GraalVM, isn't it? by Oleg Šelajev](#)



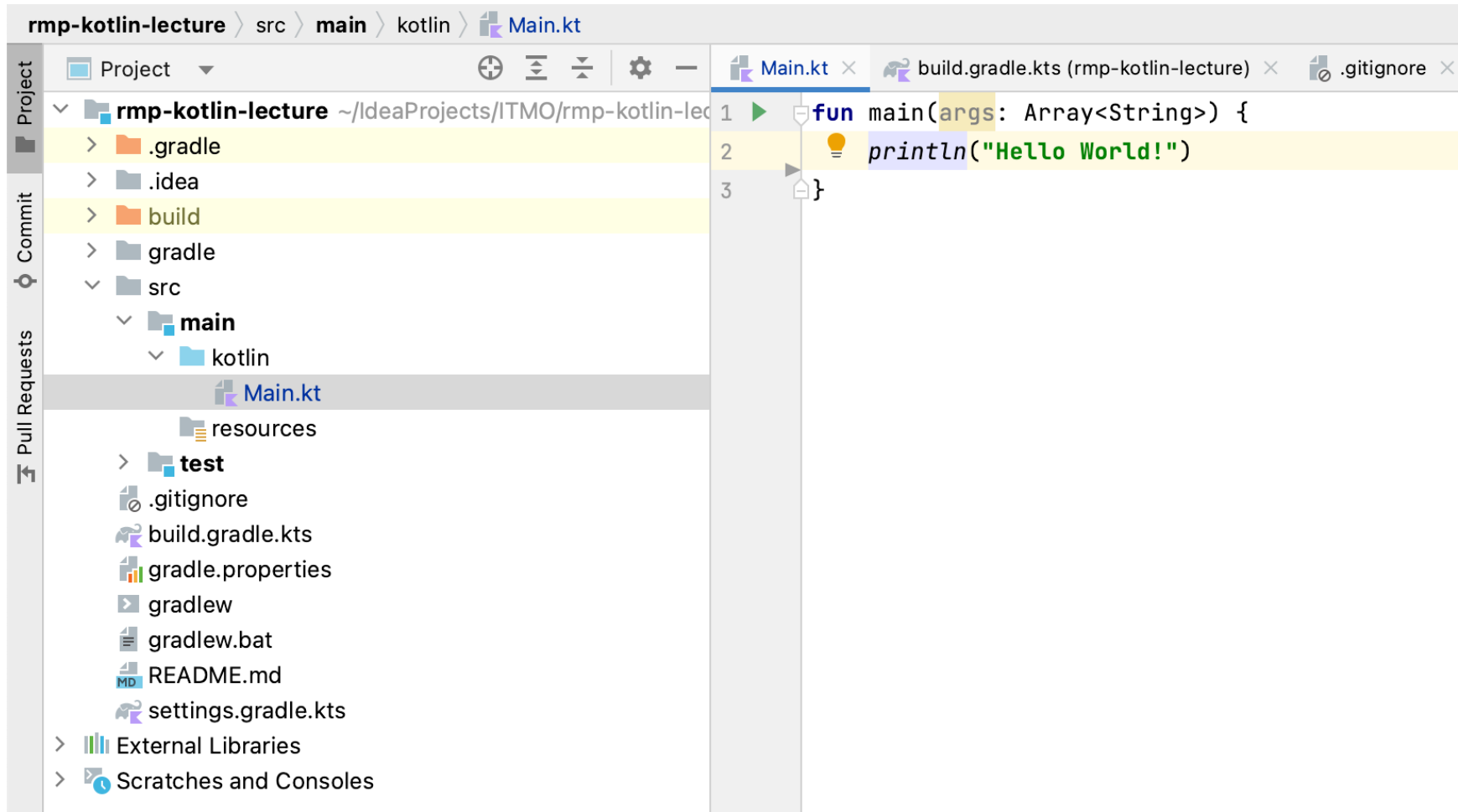
Программирование на Kotlin

- Как изучать язык?
 - Смотрим официальную документацию и обучающие видео
 - Ищем примеры везде, где только возможно
 - **Смотрим видео и читаем статьи авторов языка**, изучаем тонкие места, которые обычно обходят стороной в документации. Выступления авторов языка на профильных конференциях – бесценный источник информации.
 - Пишем код и пробуем. Чем больше пишем и чем больше пробуем, тем быстрее вникаем в особенности языка.
- Все языки программирования (в рамках схожих моделей вычислений (парадигм программирования) довольно похожи и имеют примерно одинаковый набор средств.
- Отличия заключаются в степени удобства выражения ваших мыслей (понятность, лаконичность и т.п.) см. книгу «Р.У. Себеста - Основные концепции языков программирования»

Минимальный проект Gradle



Минимальная программа



Hello world на Kotlin



The image shows a code editor window titled 'Main.kt'. The code is written in Kotlin and consists of a single function `main()` that prints 'Hello World!'. The code is as follows:

```
1  fun main() {  
2      print("Hello World!")  
3  }  
4  |
```


Основной синтаксис, var, val,...

```
fun fun1(a : Int, b : Int) : Int {  
    return (a + b) * 2  
}
```

Hello World!

result = 5

10

```
fun main() {  
    val const1 = 1  
    var result: Int  
    print(«Hello «)  
    println(«World!»)  
    result = const1 + 3  
    result += 1  
    println( «result = $result»)  
    println( fun1( 2,3) )  
}
```

Класс Any

- Вершина иерархии классов Kotlin
- В объект типа Any можно положить значение любого типа:
 - `val foo: Any = "Смысл жизни и всего такого"`
 - `val bar: Any = 42`
- Any? Может хранить null
 - `val baz: Any? = null`
- `val size : Int = (foo as String).length`
- `if(foo is String) { println("foo - String") }`
- См.: [Kotlin: взгляд изнутри — преимущества, недостатки и особенности](#)

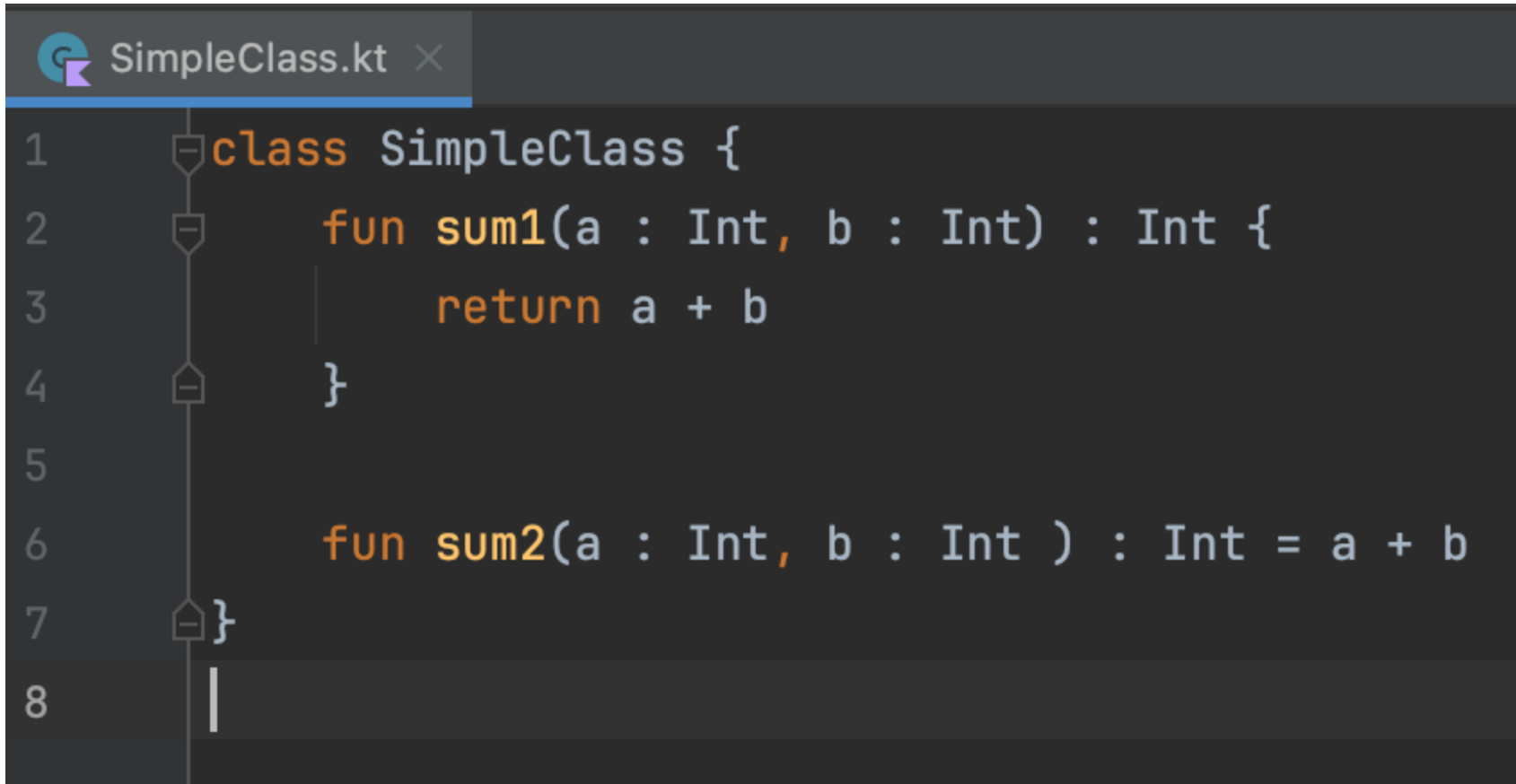
Класс Unit

- Аналог void в Java
- Этот класс возвращает функция, которой нечего вернуть
- Unit – object, то есть это синглтон (объект существующий в единственном экземпляре). От Unit нельзя плодить экземпляры.

Nothing

- Этот тип (тоже объект, синглтон) используют, чтобы показать, что функция никогда не возвращает значения или вызывает исключение (возврат значения недостижим в принципе).
- `throw` является выражением и возвращает тип `Nothing`:
 - `val s = person.name ? : throw IllegalArgumentException("Name required")`

Описание класса



```
1  class SimpleClass {  
2      fun sum1(a : Int, b : Int) : Int {  
3          return a + b  
4      }  
5  
6      fun sum2(a : Int, b : Int ) : Int = a + b  
7  }  
8  |
```

Виды классов

- Open – открытый класс, от него можно наследовать
- Sealed – запечатанный или изолированный класс, от этого класса можно наследовать только в родном пакете (package). Этот класс является абстрактным. Придуман для управлением дерева наследования.
- Inner – вложенный класс (класс внутри класса с доступом к ко всем переменным и функциям)
- Enum – перечисляемые классы (каждый элемент класса – объект)
- Data – класс данных, не имеет методов, от него нельзя наследовать (final)

Функции области видимости (Scope): let

```
fun letTest() {  
    val str = "Ответ на главный вопрос жизни и вселенной:"  
    val result = str.let { it: String  
        println(it)  
        it.length ^let  
    }  
    println("str $result")  
}
```

Функции области видимости (Scope): run

```
fun runTest() {  
    val result = "Ответ на главный вопрос жизни и вселенной:".run { this: String  
        print(this)  
        length ^run  
    }  
    println(" $result")  
}
```


Функции области видимости (Scope): apply

```
fun applyTest() {  
    class Bar( var x : Int = 2, var y : Int = 3) {  
        fun print() {  
            println("${x}x${y}")  
        }  
    }  
  
    val bar = Bar().apply { this: Bar  
        x = 10  
        y = 12  
    }  
    bar.print()  
}
```

Наследование

```
InheritanceExample.kt x
1  open class A() {
2      fun a() {
3          println("a")
4      }
5  }
6
7  open class B() : A() {
8      fun b() {
9          println("b")
10     }
11 }
12
13 class InheritanceExample : B() {
14     fun c() {
15         a()
16         b()
17         println("c")
18     }
19 }
```

Интерфейсы

```
InterfaceExample.kt x
1 interface ICommand {
2     fun execute(x : Int, y : Int) : Int
3 }
4
5 class Add() : ICommand {
6     override fun execute(x: Int, y: Int) : Int = x + y
7 }
8
9 class Sub() : ICommand {
10    override fun execute(x: Int, y: Int) : Int = x - y
11 }
12
13 class InterfaceExample1 {
14     fun calc(f : ICommand, x : Int, y : Int) : Int = f.execute(x, y)
15 }
16
17 class InterfaceExample2 {
18     fun calc(f : ICommand, x : Int, y : Int) : Int =
19         when(f) {
20             is Add -> f.execute(x, y)
21             is Sub -> f.execute(x, y)
22             else -> throw Exception("??")
23         }
24 }
```

Циклы

```
ForExample.kt x
1  class ForExample {
2
3      fun testFor() {
4          val array : ArrayList<String> = arrayListOf("foo", "bar", "baz")
5
6          for(item in array) {
7              println(item)
8          }
9      }
10
11     fun testRepeatIt() {
12         repeat( times: 3) { it: Int
13             println(it)
14         }
15     }
16
17     fun testRepeatI() {
18         repeat( times: 3) { i ->
19             println(i)
20         }
21     }
22 }
```

When (аналог Case)

```
class WhenExample {  
  
    fun testWhen1(value: Int): String {  
        var result = ""  
        when (value) {  
            1 -> result = "foo"  
            2 -> result = "bar"  
            3 -> result = "baz"  
            else -> result = "?"  
        }  
        return result  
    }  
}
```

```
fun testWhen2(value: Int): String {  
    var result = ""  
    when (value) {  
        1, 2 -> result = "foobar"  
        3 -> result = "baz"  
        else -> result = "?"  
    }  
    return result  
}
```

```
fun testWhen3(value: Int): String {  
    var result = ""  
    when (value) {  
        in 1 ≤ .. ≤ 3 -> result = "foo"  
        in 4 ≤ .. ≤ 6 -> result = "bar"  
        in 7 ≤ .. ≤ 9 -> result = "baz"  
        else -> result = "?"  
    }  
    return result  
}
```

```
fun testWhen4(value: Int): String {  
    val result = when (value) {  
        1 -> "foo"  
        2 -> "bar"  
        3 -> "baz"  
        else -> "?"  
    }  
    return result  
}
```

```
fun testWhen5(value: Int): String =  
    when (value) {  
        1 -> "foo"  
        2 -> "bar"  
        3 -> "baz"  
        else -> "?"  
    }
```

```
fun testWhen6(value: Int): String = when (value) {  
    1 -> "foo"; 2 -> "bar"; 3 -> "baz"; else -> "?"  
}
```

```
fun testWhen7(msg : Msg) : String =  
    when ( msg ) {  
        is MsgA -> msg.mA()  
        is MsgB -> msg.mB()  
        else -> "?"  
    }
```

ПОТОКИ

```
class ThreadClass1 : Thread() {  
    override fun run() {  
        repeat( times: 5 ) { it: Int  
            println("${Thread.currentThread()} run #$it")  
            Thread.sleep( millis: 500)  
        }  
    }  
}
```

```
class ThreadClass3 {  
    fun start() {  
        Thread {  
            repeat( times: 5 ) { it: Int  
                println("${Thread.currentThread()} run #$it")  
                Thread.sleep( millis: 500)  
            }  
        }.start()  
    }  
}
```

```
class ThreadClass2 : Runnable {  
    override fun run() {  
        repeat( times: 5 ) { it: Int  
            println("${Thread.currentThread()} run #$it")  
            Thread.sleep( millis: 500)  
        }  
    }  
}
```

```
class ThreadClass4 {  
    fun start() {  
        val thread = Thread {  
            repeat( times: 5 ) { it: Int  
                println("${Thread.currentThread()} run #$it")  
                Thread.sleep( millis: 500)  
            }  
        }  
        thread.start()  
    }  
}
```

Корутины

```
import kotlinx.coroutines.*

class CoroutineExample {

    fun foo() {
        val context = newSingleThreadContext( name: "test")

        CoroutineScope(context).launch { this: CoroutineScope
            repeat( times: 10 ) { it: Int
                println("${Thread.currentThread()} run #${it}")
                delay( timeMillis: 50)
            }
        }
    }

    fun bar() {
        val context = newFixedThreadPoolContext( nThreads: 2, name: "test2")

        CoroutineScope(context).launch { this: CoroutineScope
            repeat( times: 10 ) { it: Int
                println("${Thread.currentThread()} run #${it}")
                delay( timeMillis: 50)
            }
        }
    }
}
```

Исключения

```
class ExceptionExample {  
  
    fun foo(a : Int) {  
        if( a > 20 ) {  
            throw Exception("Big bada boom!")  
        }  
    }  
  
    fun bar() {  
        try {  
            foo( a: 21 )  
        } catch (e : Exception) {  
            println("ExceptionClass.bar(): exception -> ${e.message}")  
            e.printStackTrace()  
        }  
    }  
  
    fun baz() {  
  
        runCatching { this: ExceptionExample  
            foo( a: 21)  
        }.onFailure { it: Throwable  
            println("ExceptionClass.bar(): exception -> ${it.message}")  
        }.onSuccess {  
            println("Ok")  
        }  
    }  
}
```


Сериализация в JSON

```
@Serializable
data class TaskResult(
    var task : String = "",
    var errorCode : Int = -1,
    var errorMessage : String = "",
    var buf : String = "",
```

```
class SerializationTest {
    fun toJson() {
        val taskResult = TaskResult(
            task: "testTask", errorCode: 12345,
            errorMessage: "test1",
            buf: "test2"
        )
        var dt : Long = 0
        var json = ""
        dt = measureNanoTime {
            json = Json.encodeToString(taskResult)
        }
        println("dt = ${dt/1000} mks, json: $json")
    }
}
```

Время и дата

```
import java.time.*
import java.time.format.DateTimeFormatter

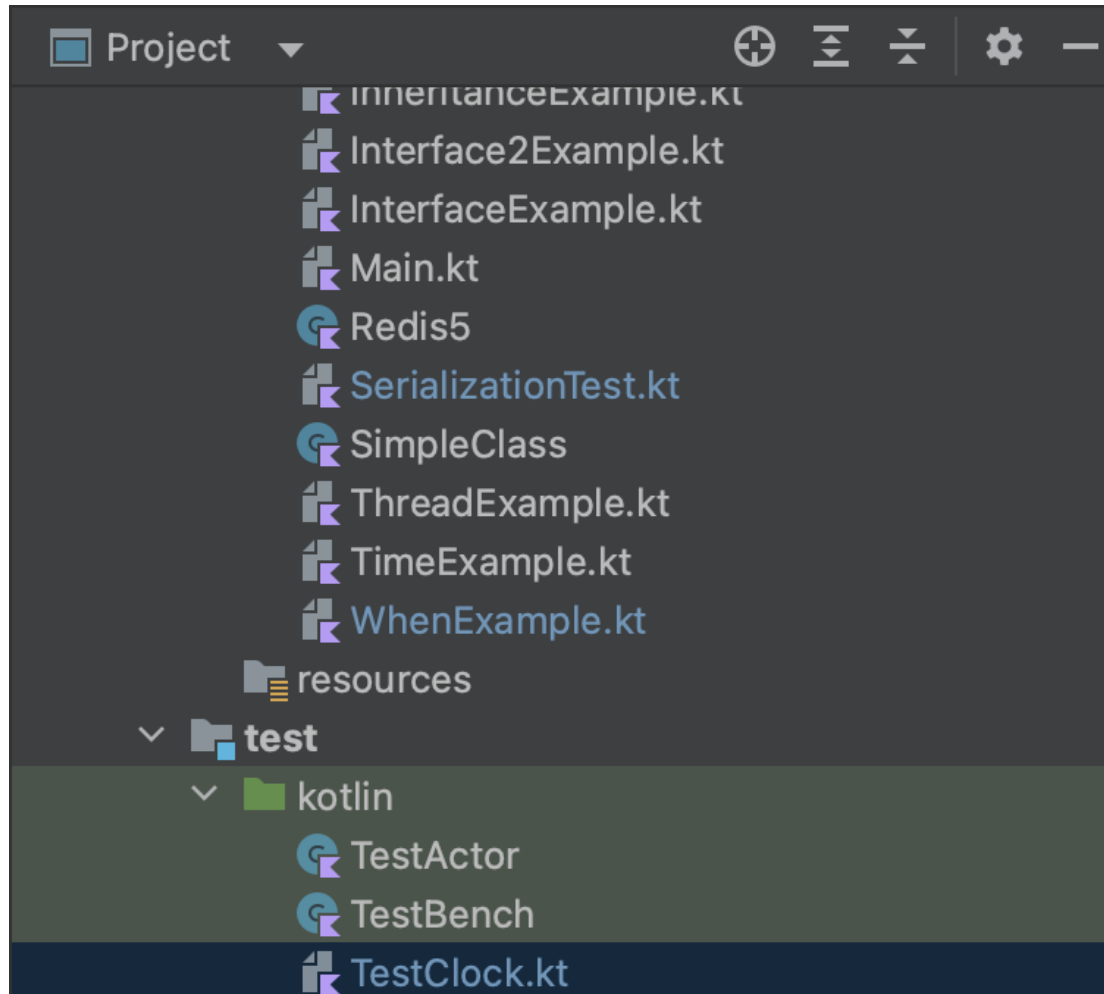
class ClockExample {
    private val patternTimestamp = "dd.MM.yyyy HH:mm:ss"

    private fun getInstantNow() : Instant {
        return Clock.systemDefaultZone().instant()
    }

    private fun getLocalFormattedTs(instant : Instant) : String {
        val ldt = LocalDateTime.ofInstant(instant, ZoneId.systemDefault())
        return DateTimeFormatter.ofPattern(patternTimestamp).format(ldt)
    }

    fun getDateTime() : String = getLocalFormattedTs( getInstantNow() )
}
```

Unit тесты



```
import kotlinx.coroutines.delay
import kotlinx.coroutines.runBlocking
import org.junit.Test

class ClockTest {
    @Test
    fun test1() {
        runBlocking { this: CoroutineScope
            println( ClockExample().getDateTime() )
            delay( timeMillis: 2000)
            println( ClockExample().getDateTime() )
        }
    }
}
```