

Programming in Kotlin

Syllabus

Prerequisites

Basic programming knowledge

Duration

1 semester

Recommendations for preparation

Students are expected to have a working programming environment. This may include any IDEs that support Kotlin, such as IntelliJ IDEA. Free educational licenses are available for download [here](#).

Description

In this course students will learn about the Kotlin programming language – a modern, powerful, and expressive language that is used for various purposes, from Android development all the way through to web development and data science. Students will learn how to apply Kotlin to solve practical software development problems and will learn about data types, variables and control flow, functions, object-oriented programming, exception handling, collections and generics, lambdas, and higher-order functions. They will also learn about various key features of Kotlin such as null safety, extension functions, and coroutines. At the end of the course students will study build systems, using Gradle as an example, as well as explore compilation techniques and how the Kotlin K2 compiler works.

Contents

- Introduction to Kotlin
- Object-oriented programming
- Build systems and testing
- Generics
- Containers
- Functional programming
- JVM + the Kotlin compiler
- Parallel and concurrent programming
- Asynchronous programming
- Exceptions
- Testing

Assessment resources for educators:

- Quizzes
- Homework assignments
- Tests

Goals

- Provide students with a solid foundation in the Kotlin programming language.
- Teach students how to apply Kotlin to solve practical problems in software development.
- Enable students to write efficient, readable, and maintainable Kotlin code.
- Familiarize students with key features of Kotlin, such as null safety, extension functions, and coroutines.
- Give students a deeper understanding of the fundamental concepts of computer science, such as concurrent computations and how they can be applied to software development in Kotlin.

Intended learning outcomes

Upon completion of this module, students will be able to:

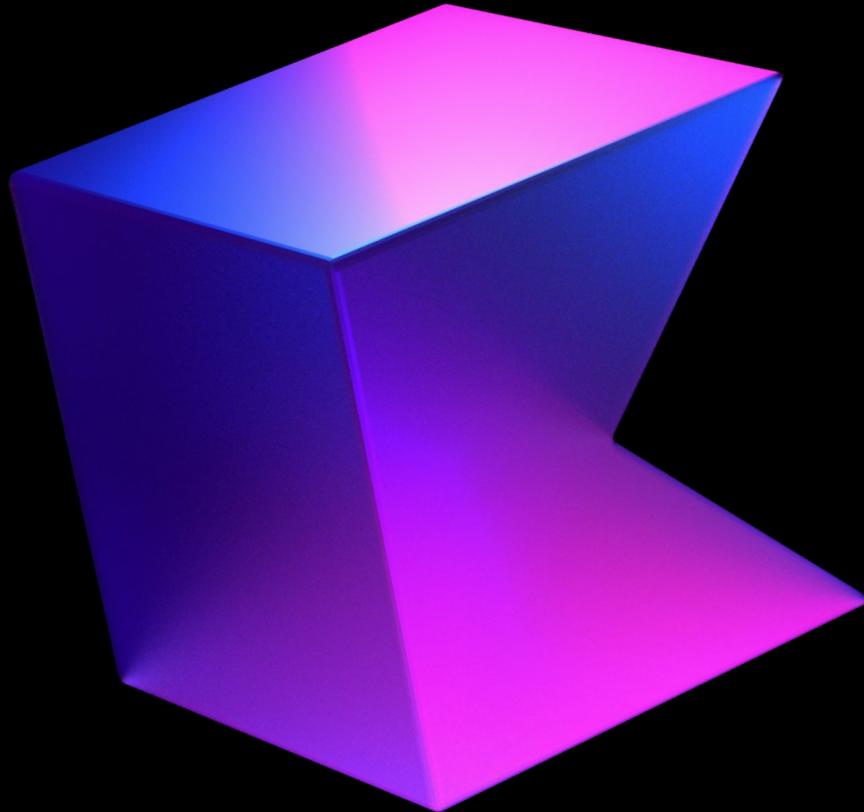
- Use the unique features of Kotlin to write readable, maintainable, and expressive code.
- Use Kotlin to solve practical problems in software development.
- Write efficient and optimized Kotlin code.
- Use the Gradle build system.
- Understand how the Kotlin compiler works.

Recommended materials

- Roman Elizarov, Svetlana Isakova, Sebastian Aigner, and Dmitry Jemerov: [Kotlin in Action](#), Second Edition, Manning Publications, 2022.
- [Kotlin Documentation](#)
- [Kotlin Onboarding: Introduction](#)
- [Kotlin Onboarding: Object-Oriented programming](#)



Introduction to Kotlin

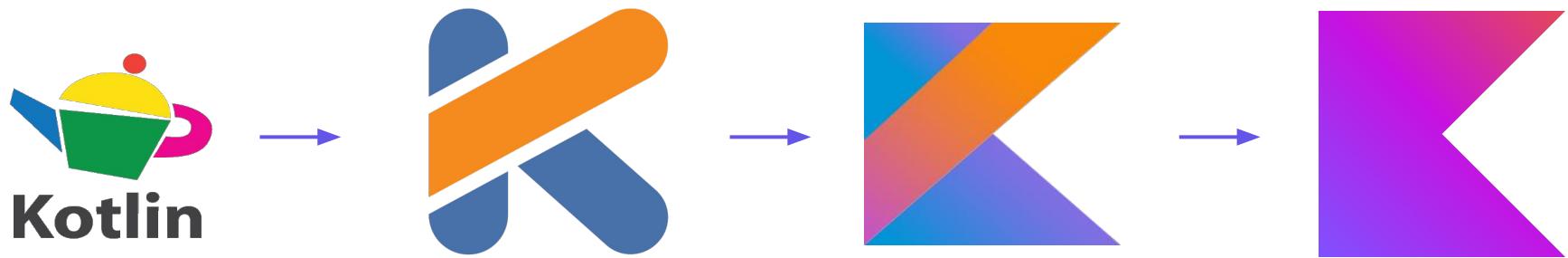


@kotlin

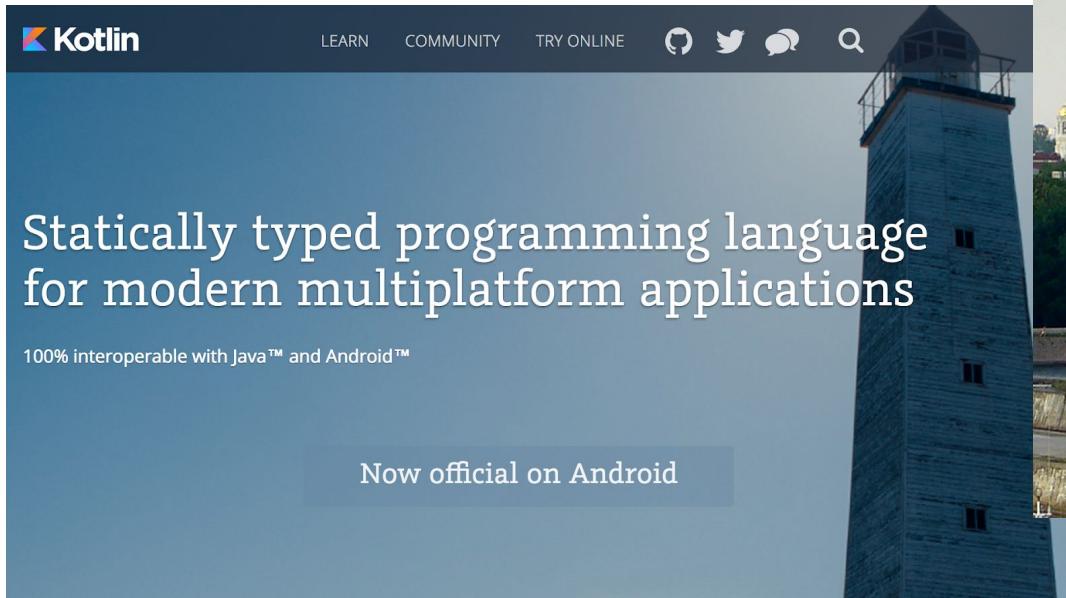
Why Kotlin?

- Expressiveness/Conciseness
- Safety
- Portability/Compatibility
- Convenience
- High Quality IDE Support
- Community
- Android 
- More than a gazillion devices run ~~Java~~ Kotlin
- Lactose free
- ~~Sugar free~~
- Gluten free

Logo



Name



The image shows the official Kotlin website homepage. The header features the Kotlin logo and navigation links for LEARN, COMMUNITY, TRY ONLINE, and social media icons for GitHub, Twitter, and LinkedIn. Below the header is a large, semi-transparent white text area containing the slogan: "Statically typed programming language for modern multiplatform applications". At the bottom of this area is the text "100% interoperable with Java™ and Android™". A blue button at the bottom right says "Now official on Android". The background of the page is a photograph of a tall, weathered wooden lighthouse.



Kotlin is named after an island in the Gulf of Finland.

Hello, world!

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

Where is ";" ???

The basics

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

- An entry point of a Kotlin application is the `main` **top-level** function.
- It 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

```
val/var myValue: Type = someValue
```

- `var` - mutable
- `val` - immutable
- Type can be inferred in most cases
- Assignment can be deferred

```
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
```

Variables

```
const val/val myValue: Type = someValue
```

- `const val` - compile-time const value
- `val` - immutable value
- for `const val` use uppercase for naming

```
const val NAME = "Kotlin" // can be calculated at compile-time
```

```
val nameLowered = NAME.lowercase() // cannot be calculated at compile-time
```

Functions

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

Single expression function.

```
fun mul(a: Int, b: Int) = a * b
```

`Unit` means that the function does not return anything meaningful.

```
fun printMul(a: Int, b: Int): Unit {  
    println(mul(a, b))  
}
```

It can be omitted.

```
fun printMul1(a: Int = 1, b: Int) {  
    println(mul(a, b))  
}
```

Arguments can have **default** values.

```
fun printMul2(a: Int, b: Int = 1) = println(mul(a, b))
```

If expression

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

is the same as



```
fun maxOf(a: Int, b: Int) =  
    if (a > b) {  
        a  
    } else {  
        b  
    }
```

if can be an expression (it can return).

Can be a one-liner:

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

When expression

```
when (x) {  
    1 → print("x = 1")  
    2 → print("x = 2")  
    else → {  
        print("x is neither 1 nor 2")  
    }  
}
```

`when` returns, the same way that `if` does.

```
when {  
    x < 0 → print("x < 0")  
    x > 0 → print("x > 0")  
    else → {  
        print("x = 0")  
    }  
}
```

The condition can be inside of the branches.

When statement

```
fun serveTeaTo(customer: Customer) {  
    val teaSack = takeRandomTeaSack()  
  
    when (teaSack) {  
        is OolongSack → error("We don't serve Chinese tea like $teaSack!")  
        in trialTeaSacks, teaSackBoughtLastNight →  
            error("Are you insane?! We cannot serve uncertified tea!")  
    }  
  
    teaPackage.brew().serveTo(customer)  
}
```

`when` can accept several options in one branch. `else` branch can be omitted if `when` block is used as a *statement*.

&& vs and

```
if (a && b) { ... }      VS      if (a and b) { ... }
```

Unlike the `&&` operator, this function does not perform short-circuit evaluation.

The same behavior with OR:

```
if (a || b) { ... }      VS      if (a or b) { ... }
```

Loops

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

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")
}
```

Loops

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

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

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

The condition variable can be initialized inside to the `do...while` loop.

Loops

There are `break` and `continue` labels for loops:

```
myLabel@ for (item in items) {  
    for (anotherItem in otherItems) {  
        if (...) break@myLabel  
        else continue@myLabel  
    }  
}
```

Ranges

```
val x = 10
if (x in 1..10) {
    println("fits in range")
}

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

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

downTo and step are extension functions, not keywords.

' .. ' is actually T.rangeTo(that: T)

Null safety

```
val notNullText: String = "Definitely not null"
val nullableText1: String? = "Might be null"
val nullableText2: String? = null

fun funny(text: String?) {
    if (text != null)
        println(text)
    else
        println("Nothing to print :(")
}

fun funnier(text: String?) {
    val toPrint = text ?: "Nothing to print :("
    println(toPrint)
}
```

Elvis operator ?:

If the expression to the left of ?: is not `null`, the Elvis operator returns it; otherwise, it returns the expression to the right.

Note that the expression on the right-hand side is evaluated only if the left-hand side is `null`.

```
fun loadInfoById(id: String): String? {  
    val item = findItem(id) ?: return null  
    return item.loadInfo() ?: throw Exception("...")  
}
```



Safe Calls

someThing ?. otherThing does not throw an NPE if someThing is `null`.

Safe calls are useful in chains. For example, an employee may be assigned to a department (or not). That department may in turn have another employee as a department head, who may or may not have a name, which we want to print:

```
fun printDepartmentHead(employee: Employee) {  
    println(employee.department?.head?.name)  
}
```

To print only for non-null values, you can use the safe call operator together with `let`:

```
employee.department?.head?.name?.let { println(it) }
```

Unsafe Calls

The not-null assertion operator (!!) converts any value to a non-null type and throws an **NPE** exception if the value is null.

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

Please, avoid using unsafe calls!

TODO

Always throws a `NotImplementedError` at **run-time** if called, stating that operation is not implemented.

```
// Throws an error at run-time if calls this function, but compiles
fun findItemOrNull(id: String): Item? = TODO("Find item $id")
```

```
// Does not compile at all
fun findItemOrNull(id: String): Item? = { }
```

String templates and the string builder

```
val i = 10
val s = "Kotlin"

println("i = $i")
println("Length of $s is ${s.length}")

val sb = StringBuilder()
sb.append("Hello")
sb.append(", world!")
println(sb.toString())
```

Lambda expressions

```
val sum: (Int, Int) → Int = { x: Int, y: Int → x + y }
val mul = { x: Int, y: Int → x * y }
```

According to Kotlin convention, if the last parameter of a function is a function, then a lambda expression passed as the corresponding argument can be placed outside the parentheses:

```
val badProduct = items.fold(1, { acc, e → acc * e })
```

```
val goodProduct = items.fold(1) { acc, e → acc * e }
```

If the lambda is the only argument, the parentheses can be omitted entirely (the documentation calls this feature "trailing lambda as a parameter"):

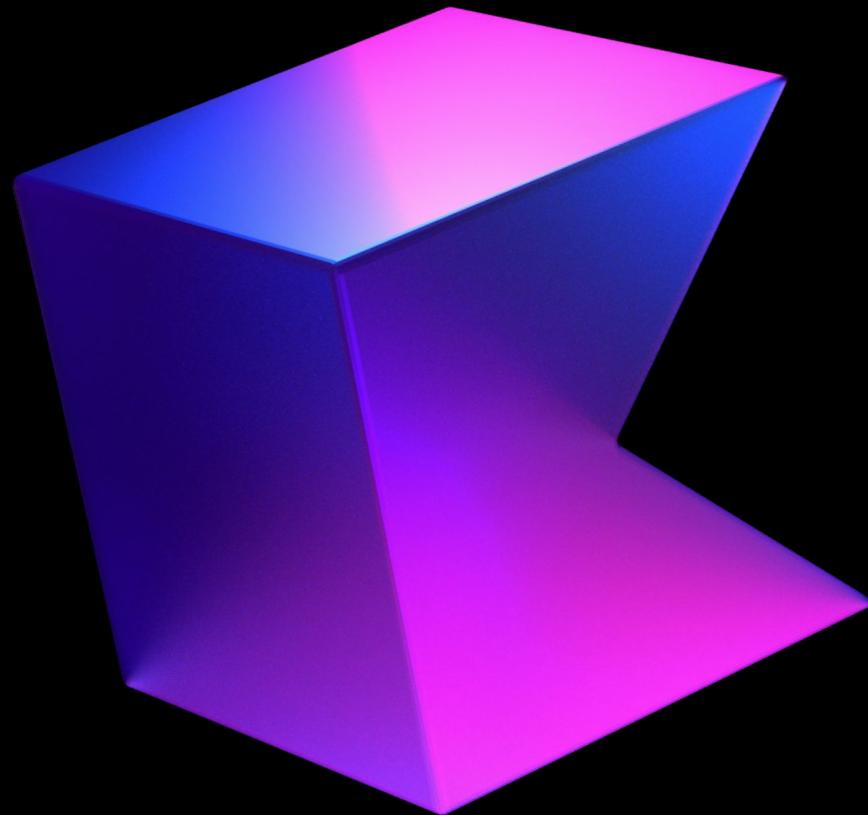
```
run{ println("Not Cool") }
run { println("Very Cool") }
```

When in doubt

Go to:

- kotlinlang.org
- kotlinlang.org/docs
- play.kotlinlang.org/byExample

Thanks!



@kotlin



Object-Oriented Programming

Introduction and Basic Principles



Object-Oriented Programming

Object-oriented programming (OOP) – A programming paradigm based on the representation of a program as a set of objects and interactions between them

Class and Object

Class – A set of attributes (fields, properties, data) and related methods (functions, procedures) that together represent some abstract entity.

Attributes store state, while procedures express behavior.

Classes are sometimes called prototypes.

Object – An instance of a class, which has its own specific state.

```
class Person:
```

- String attribute name
- Boolean attribute married
- Method greet

```
Person x:
```

```
    name = "Olek",  
    married = false
```

```
x.greet()
```

Object (class/type) invariant

Invariants place constraints on the state of an object, maintained by its methods right from construction.

It is the object's own responsibility to ensure that the invariant is being maintained.

Corollaries:

- Public fields are nasty.
- If a field does not participate in the object's invariant, then it is not clear how it belongs to this object at all, which is evidence of poor design choices.

Abstraction

Objects are data abstractions with internal representations, along with methods to interact with those internal representations. There is no need to expose internal implementation details, so those may stay “inside” and be hidden.

Encapsulation

Encapsulation – The option to bundle data with methods operating on said data, which also allows you to hide the implementation details from the user.

- An object is a black box. It accepts messages and replies in some way.
- Encapsulation and the interface of a class are intertwined: Anything that is not part of the interface is encapsulated.
- OOP encapsulation differs from encapsulation in abstract data types.

Abstraction vs Encapsulation

Abstraction is about what others see and how they interact with an object.

Encapsulation is about how an object operates internally and how it responds to messages.

Encapsulation

Most programming languages provide special keywords for modifying the accessibility or visibility of attributes and methods.

In Kotlin:

- **public** – Accessible to anyone
- **private** – Accessible only inside the class
- **protected** – Accessible inside the class and its inheritors
- **internal** – Accessible in the module

Inheritance

Inheritance – The possibility to define a new class based on an already existing one, keeping all or some of the base class functionality (state/behavior).

- The class that is being inherited from is called a base or parent class
- The new class is called a derived class, a child, or an inheritor
- The derived class fully satisfies the specification of the base class, but it may have some extended features (state/behavior)

Inheritance

- "General concept – specific concept".
 - "Is-a" relationship.
- Motivation
 - Keep shared code separate – in the base class – and reuse it.
 - Type hierarchy, subtyping.
 - Incremental design.
- Inheritance is often redundant and can be replaced with composition.

Subtyping

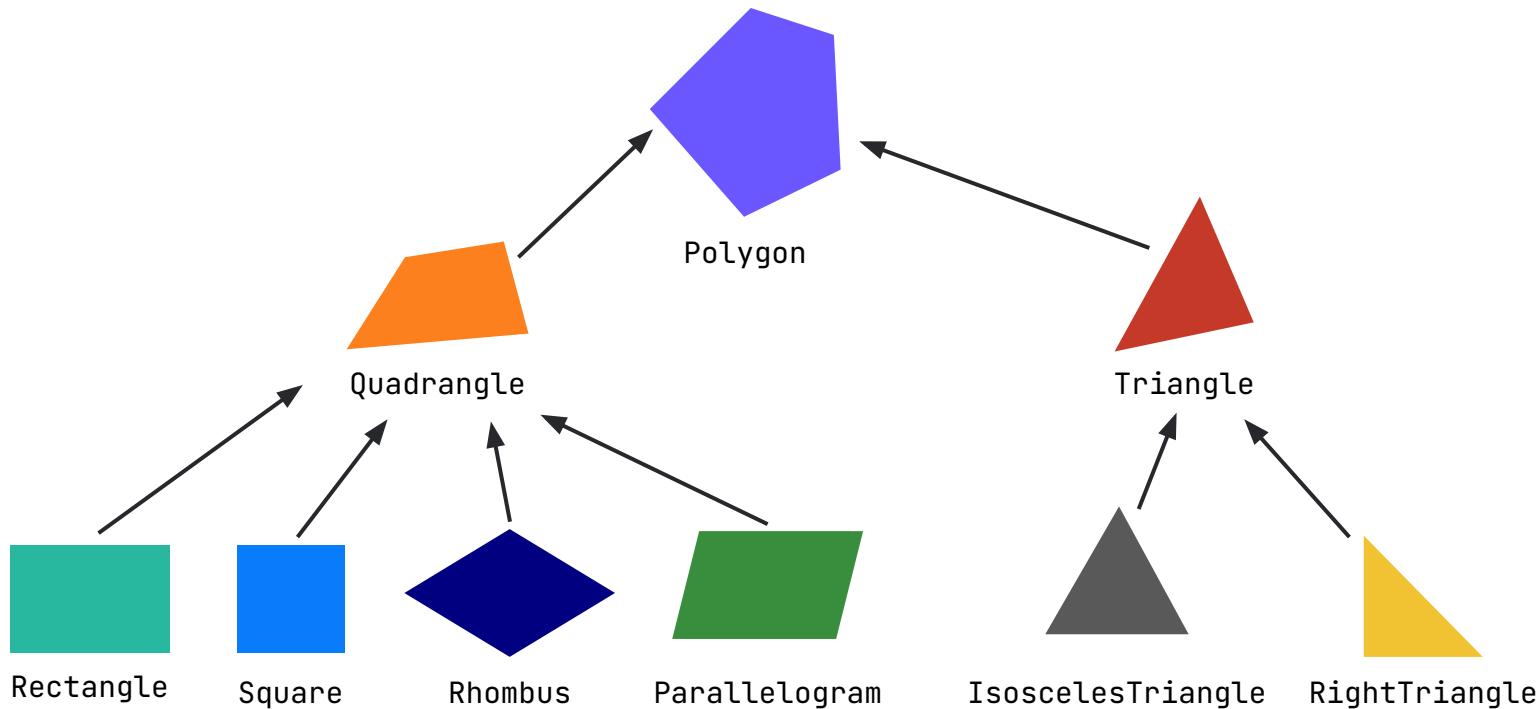
An object can belong to several types (classes) at the same time

- Eleanor – A student, a woman, a beer enthusiast, and the reigning UFC champion.
- Nate – A developer, a man, an anime lover, and a recreational swimmer.

Each type (class) defines an interface and expected behavior.

So, in our example, while Eleanor is a student, she will exhibit a set of expected behaviors (such as turning in homework, studying for tests, etc.). When Eleanor gets her degree, she will stop being a student and she may cease to exhibit the associated behaviors, but her overall identity will not change and the behaviors associated with her other properties will be unaffected.

Subtyping



Polymorphism

Polymorphism – A core OOP concept that refers to working with objects through their interfaces without knowledge about their specific types and internal structure.

- Inheritors can override and change the ancestral behavior.
- Objects can be used through their parents' interfaces.
 - The client code does not know (or care) if it is working with the base class or some child class, nor does it know what exactly happens “inside”.

Liskov substitution principle (LSP) – If for each object o_1 of type S , there is an object o_2 of type T , such that for all programs P defined in terms of T the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T .

OOP in Kotlin

```
class UselessClass

fun main() {
    val uselessObject = UselessClass() // () here is constructor invocation
}
```

Constructors

```
The primary constructor, which is used by default. If it is empty, the brackets can be omitted  
↓  
class Person(val name: String, val surname: String, private var age: Int) {  
  
    init {  
        findJob()  
    }  
  
    constructor(name: String, parent: Person) : this(name, parent.surname, 0)  
}  
  
↑  
The secondary constructor
```

The order of initialization: the primary constructor → the `init` block → the secondary constructor

Constructors

```
open class Point(val x: Int, val y: Int) {  
    constructor(other: Point) : this(other.x, other.y) { ... }  
  
    constructor(circle: Circle) : this(circle.centre) { ... }  
}
```

Constructors can be chained, but they should always call the primary constructor in the end.

A secondary constructor's body will be executed after the object is created with the primary constructor. If it calls other constructors, then it will be executed after the other constructors' bodies are executed.

Inheritor class must call parent's constructor:

```
class ColoredPoint(val color: Color, x: Int, y: Int) : Point(x, y) { ... }
```

init blocks

```
class Example(val value: Int, info: String) {  
    val anotherValue: Int  
    var info = "Description: $info"  
  
    init {  
        this.info += ", with value $value"  
    }  
  
    val thirdValue = computeAnotherValue() * 2  
  
    private fun computeAnotherValue() = value * 10  
  
    init {  
        anotherValue = computeAnotherValue()  
    }  
}
```

There can be several `init` blocks.

Values can be initialized in `init` blocks that are written after them.

Constructor parameters are accessible in `init` blocks, so sometimes you have to use `this`.

Abstraction

```
interface RegularCat {  
    fun pet()  
    fun feed(food: Food)  
}
```

```
interface SickCat {  
    fun checkStomach()  
    fun giveMedicine(pill: Pill)  
}
```

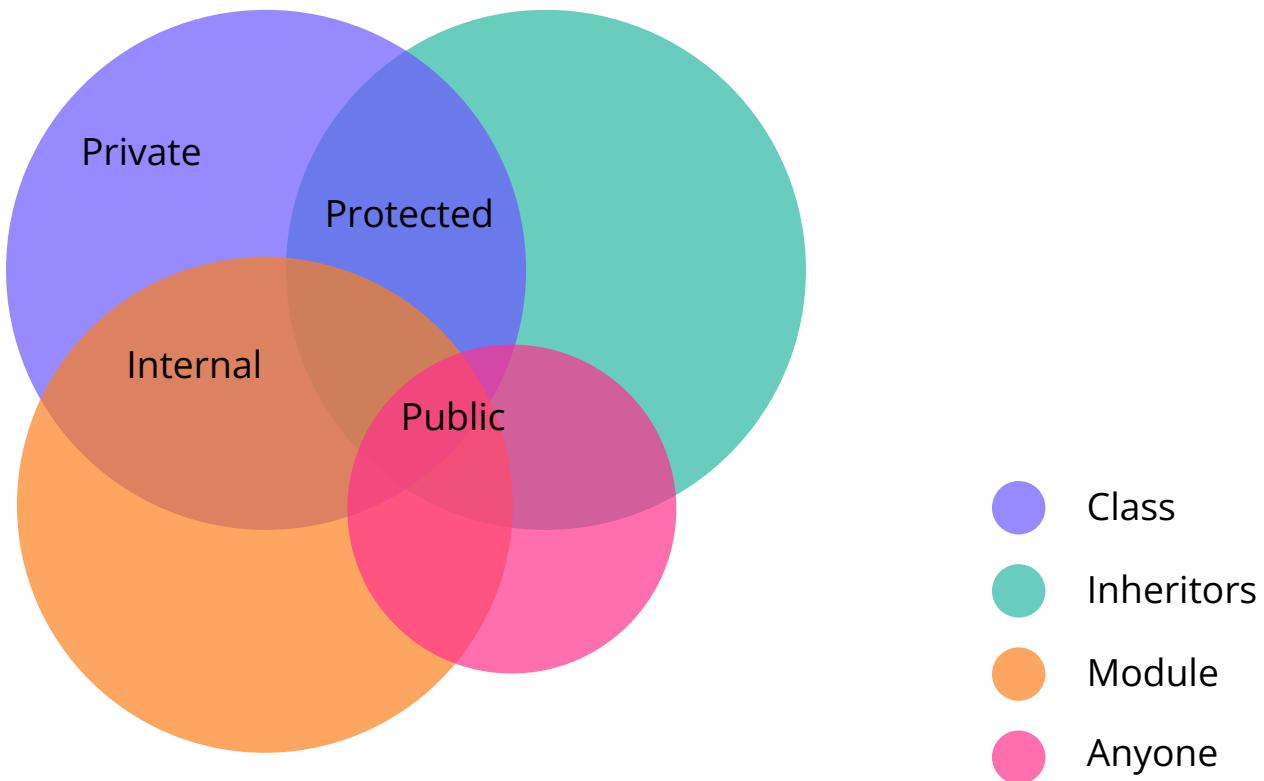
VS

```
abstract class RegularCat {  
    abstract val name: String  
  
    abstract fun pet()  
    abstract fun feed(food: Food)  
}  
  
abstract class SickCat {  
    abstract val location: String  
  
    abstract fun checkStomach()  
    fun giveMedicine(pill: Pill) {}  
}
```

Interfaces **cannot have** a state.
(We'll get back to this a bit later.)

Abstract classes cannot have an instance, but can **have** a state.

Encapsulation



Encapsulation

```
abstract class RegularCat {  
    protected abstract val isHungry: Boolean  
    private fun poop(): Poop { /* do the thing */ }  
    abstract fun feed(food: Food)  
}  
  
class MyCat : RegularCat() {  
    override val isHungry: Boolean = false  
    override fun feed(food: Food) {  
        if (isHungry) { /* do the thing */ }  
        else { poop() } // MyCat cannot poop  
    }  
}  
Cannot access 'poop': it is invisible (private in a supertype) in 'MyCat'
```

Inheritance

```
class SickDomesticCat : RegularCat(),  
CatAtHospital {  
    override var isHungry: Boolean = false  
        get() = field  
        set(value) {...}  
  
    override fun pet() {...}  
  
    override fun feed(food: Food) {...}  
  
    override fun checkStomach() {...}  
  
    override fun giveMedicine(pill: Pill)  
{...}  
}
```

To allow a class to be inherited by other classes, the class should be marked with the **open** keyword. (**Abstract** classes are always open.)

In Kotlin you can inherit only from **one class**, and from as many **interfaces** as you like.

When you're inheriting from a class, you have to call its constructor, just like how secondary constructors have to call the primary.

Why do you prohibit a cat from pooping?!

```
abstract class Cat {  
    /* final */ fun anotherDay() {  
        // various cat activities  
        digest(findFood())  
        poop(findWhereToPoop())  
    }  
    private fun poop(where: Place): Poop {...}  
    private fun digest(food: Food) {  
        // don't know how they work  
        poop(findWhereToPoop())  
    }  
    abstract fun feed(food: Food)  
    abstract fun findWhereToPoop(): Place  
    abstract fun findFood(): Food  
}
```

```
class DomesticCat(  
    val tray: Tray,  
    val bowl: Bowl  
) : Cat() {  
    override fun feed(food: Food) {  
        // place some food in the bowl  
    }  
  
    override fun findWhereToPoop() = tray  
    override fun findFood() {  
        return bowl.getFood() ?: run {  
            // find food somewhere else  
        }  
    }  
}
```

Polymorphism revisited

```
interface DomesticAnimal {  
    fun pet()  
}  
  
class Dog: DomesticAnimal {  
    override fun pet() {...}  
}  
  
class Cat: DomesticAnimal {  
    override fun pet() {...}  
}  
  
fun main() {  
    val homeZoo = listOf<DomesticAnimal>(Dog(), Cat())  
    homeZoo.forEach { it.pet() }  
}
```

Properties

```
class PositiveAttitude(startingAttitude: Int) {  
    var attitude = max(0, startingAttitude)  
    set(value) =  
        if (value ≥ 0) {  
            field = value  
        } else {  
            println("Only positive attitude!")  
            field = 0  
        }  
  
    var hiddenAttitude: Int = startingAttitude  
    private set  
    get() {  
        if (isSecretelyNegative) {  
            println("Don't ask this!")  
            field += 10  
        }  
        return field  
    }  
  
    val isSecretelyNegative: Boolean  
        get() = hiddenAttitude < 0  
}
```

Properties can optionally have an initializer, getter, and setter.

Use the **field** keyword to access the values inside the getter or setter, otherwise you might encounter infinite recursion.

Properties may have no (backing) field at all.

Properties

```
open class OpenBase(open val value: Int)

interface AnotherExample {
    /* abstract */ val anotherValue: OpenBase
}

open class OpenChild(value: Int) : OpenBase(value), AnotherExample
{
    override var value: Int = 1000
        get() = field - 7
    override val anotherValue: OpenBase = OpenBase(value)
}

open class AnotherChild(value: Int) : OpenChild(value) {
    final override var value: Int = value
        get() = super.value // default get() is used otherwise
        set(value) { field = value * 2 }
    final override val anotherValue: OpenChild = OpenChild(value)
// Notice that we use OpenChild here, not OpenBase
}
```

Properties may be [open](#) or [abstract](#), which means that their getters and setters might or must be overridden by inheritors, respectively.

Interfaces can have properties, but they are always [abstract](#).

You can prohibit further overriding by marking a property [final](#).

Operator overloading

```
class Example {  
    operator fun plus(other: Example): Example { ... }  
    operator fun dec() = this // return type has to be a subtype  
    operator fun get(i: Int, j: Int): SomeType { ... }  
    operator fun get(x: Double?, y: String) = this  
    operator fun <T> invoke(l: List<T>): SomeType { ... }  
}  
  
operator fun Example.rangeTo(other: Example): Iterator<Example> { ... }  
  
fun main() {  
    var ex1 = Example()  
    val ex2 = ex1 + --ex1 // -- reassigned ex1, so it has to be var  
    for (ex in ex1..ex2) {  
        ex[23, 42]  
        ex[null, "Wow"](listOf(1,2,3))  
    }  
}
```

Operator “overloading” is allowed.

Almost all operators can be overloaded.

Operators can be overloaded outside of the class.

Extensions

Kotlin provides the ability to extend a class or an interface with new functionality without having to inherit from the class or use forbidden magic (reflection)

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {  
    val tmp = this[index1] // 'this' is the given MutableList<T>  
    this[index1] = this[index2]  
    this[index2] = tmp  
}
```

If the extended class **already has** the new method with the same name and signature, the **original** one will be used.

Extensions under the hood

The class that is being extended does not change at all; it is simply a new function that can be called like a method. It cannot access private members, for example.

Extensions have static dispatch, rather than virtual dispatch by receiver type. An extension function being called is determined by the type of the expression on which the function is invoked, not by the type of the result from evaluating that expression at runtime.

```
open class Shape
class Rectangle: Shape()

fun Shape.getName() = "Shape"
fun Rectangle.getName() = "Rectangle"

fun printClassName(s: Shape) {
    println(s.getName())
}

printClassName(Rectangle()) // "Shape", not Rectangle
```

Infix functions

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

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

fun main() {
    val realHero = "Ryan" with "Gosling"
    val (real, bean) = realHero
}
```

ComponentN operator

```
class SomeData(val list: List<Int>) {  
    operator fun component1() = list.first()  
    operator fun component2() = SomeData(list.subList(1, list.size))  
    operator fun component3() = "This is weird"  
}  
  
fun main() {  
    val sd = SomeData(listOf(1, 2, 3))  
    val (head, tail, msg) = sd  
    val (h, t) = sd  
    val (onlyComponent1) = sd  
}
```

Any class can overload any number of componentN methods that can be used in destructive declarations.

Data classes have these methods by default.

Data classes

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

The compiler automatically derives:

- `equals()` and `hashCode()`
- `toString()` of the form `User(name=John, age=42)`
- `componentN()` [functions](#) corresponding to the properties in their order of declaration.
- `copy()` to copy an object, allowing you to alter some of its properties while keeping the rest unchanged

The standard library provides the `Pair` and `Triple` classes, but named data classes are a much better design choice.

Inline (value) classes

Occasionally you have to wrap a class, but wrapping always causes overhead in both memory and execution time. Inline classes may help you get the desired behavior without paying for it with a drop in performance.

```
interface Greeter {  
    fun greet(): Unit  
}  
  
class MyGreeter(var myNameToday: String) : Greeter {  
    override fun greet() = println("Hello, $myNameToday!")  
}  
  
@JvmInline  
/* final */ value class BadDayGreeter(val greeter: Greeter) : Greeter {  
    override fun greet() {  
        greeter.greet()  
        println("Having a bad day, huh?")  
    }  
}
```

```
var greeter: Greeter = MyGreeter("Cyr")  
if (today.isBad()) { greeter = BadDayGreeter(greeter)  
}  
greeter.greet()
```

Inline (value) classes

- An Inline class must have exactly one primary constructor parameter,
- Inline classes can implement interfaces, declare properties (no backing fields), and have `init` blocks.
- Inline classes are not allowed to participate in a class hierarchy, which is to say they are automatically marked with the "final" keyword.
- The compiler tries to use the underlying type to produce the most performant code.

```
@JvmInline
/* final */ value class Name(val name: String) : Greeter {
    init {
        require(name.isNotEmpty()) { "An empty name is absurd!" }
    }

    // val withABackingField: String = "Not allowed"

    var length: Int
        get() = name.length
        set(value) { println("What do you expect to happen?") }

    override fun greet() { println("Hello, $name") }
}
```

Inline (value) classes

Since inline classes are just wrappers and the compiler tries to use the underlying type, name mangling is introduced to solve possible signature clashing problems.

```
fun foo(name: Name) { ... } → public final void foo-<stable-hashcode>(name: String) { ... }

fun foo(name: String) { ... } → public final void foo(name: String) { ... }
```

If you want to call such a function from Java code, then you should use the `@JvmName` annotation.

```
@JvmName("fooName")
```

```
fun foo(name: Name) { ... } → public final void fooName(name: String) { ... }
```

Enum classes

```
enum class Direction {  
    NORTH, SOUTH, WEST, EAST  
}
```

Each enum constant is an object.

Each enum is an instance of the enum class, thus it can be initialized as:

```
enum class Color(val rgb: Int) {  
    RED(0xFF0000),  
    GREEN(0x00FF00),  
    BLUE(0x0000FF)  
}
```

Enum classes can have methods or even implement interfaces.

Kotlin example

```
val user = User("John", 23)

val (name, age) = user // destructuring declaration calls componentN()

val (onlyName) = user

val olderUser = user.copy(age = 42)

val g = Color.valueOf("green".uppercase())

when(g) {
    Color.RED → println("blood")
    Color.GREEN → println("grass")
    Color.BLUE → println("sky")
}
```

Sealed classes

```
sealed class Base {  
    open var value: Int = 23  
    open fun foo() = value * 2  
}  
  
open class Child1 : Base() {  
    override fun foo() = value * 3  
    final override var value: Int = 10  
        set(value) = run { field = super.foo() }  
}  
  
class Child2 : Base()  
  
  
val b: Base = Child1()  
when(b) {  
    is Child1 → println(1)  
    is Child2 → println(2)  
}
```

All of the inheritors of a `sealed` class must be known at compile time.

Can be used in `when` the same way as enums can be.

Not specific to `sealed` classes:

- Prohibit overriding an `open fun` or property by making it `final`.
- Access parents' methods through `super`.

Functional interfaces (SAM)

Single Abstract Method (SAM) interface

- Interface that has one abstract method.
- Kotlin allows us to use a lambda instead of a class definition to implement a SAM.

```
fun interface IntPredicate {  
    fun accept(i: Int): Boolean  
}
```

```
val isEven = object : IntPredicate {  
    override fun accept(i: Int): Boolean {  
        return i % 2 == 0  
    }  
}
```

VS

```
val isEven = IntPredicate { it % 2 == 0 }
```

```
fun main() {  
    println("Is 7 even? - ${isEven.accept(7)})  
}
```

Kotlin singleton

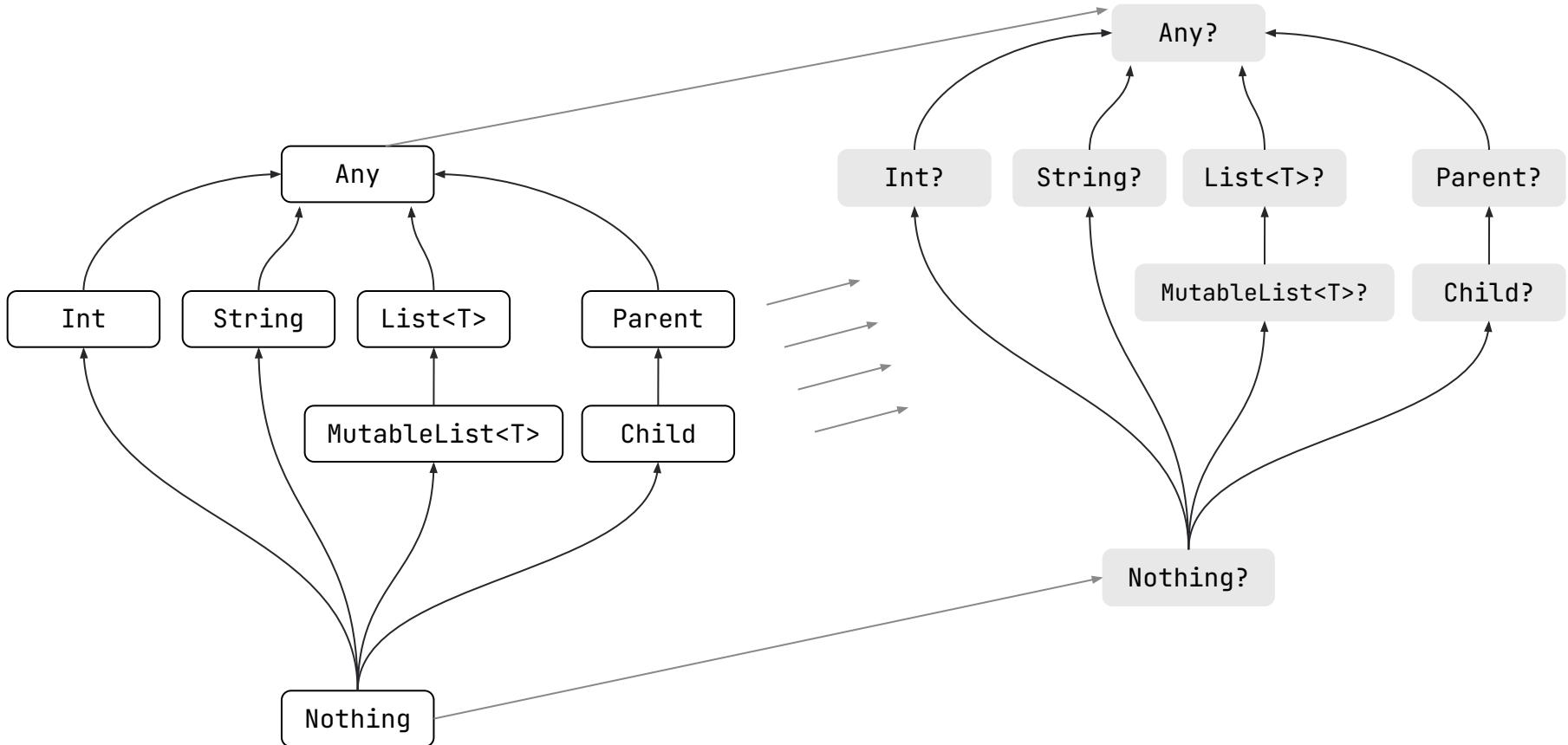
```
object DataProviderManager {  
    fun registerDataProvider(provider: DataProvider) {  
        // ...  
    }  
  
    val allDataProviders: Collection<DataProvider>  
        get() = // ...  
}  
  
DataProviderManager.registerDataProvider(...)
```

Companion objects

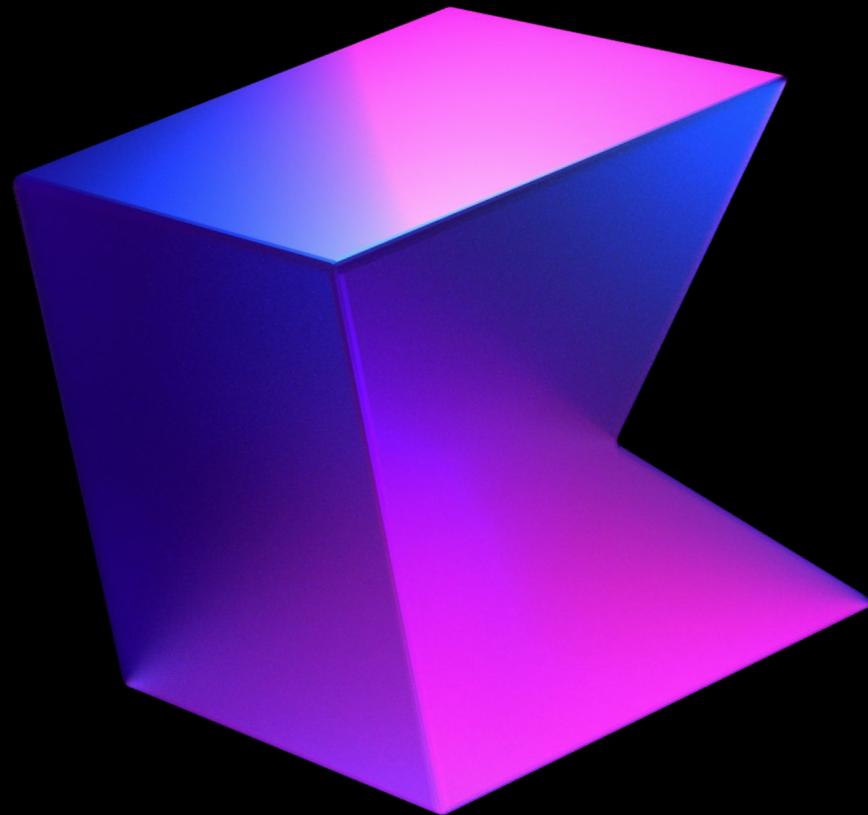
- An object declaration inside a class can be marked with the companion keyword.
- Companion objects are like static members:
 - The Factory Method
 - Constants
 - Etc.
- Visibility modifiers are applicable.
- Use @JvmStatic to go full static.

```
interface Factory<T> {  
    fun create(): T  
}  
  
class MyClass {  
    companion object : Factory<MyClass> {  
        private var counter: Int = 0  
        override fun create(): MyClass =  
            MyClass().also { counter += 1 }  
    }  
    // ... some code ...  
}  
  
val f: Factory<MyClass> = MyClass.Companion  
val instance1 = f.create()  
val instance2 = f.create()
```

Kotlin Type Hierarchy



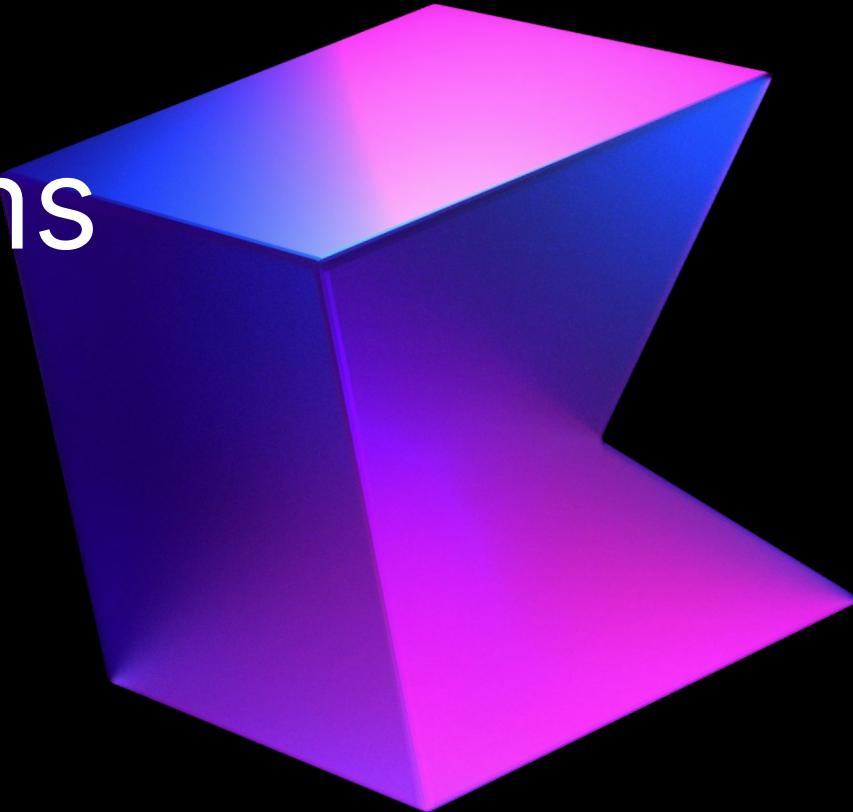
Thanks!



@kotlin



Build Systems



@kotlin

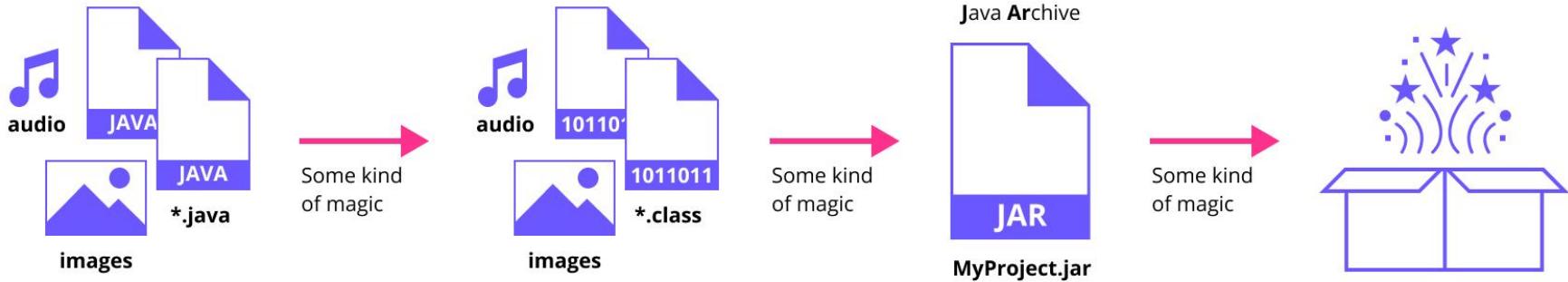
What? Why?

Build system – Software that automates the process of getting some kind of an artifact (executable, library) from the source code. Build systems can be used for:

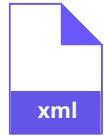
- Configuring your build once and using it forever (~~copy paste into new projects~~)
- Unifying builds and reusing logic in various projects
- Dependencies management*
- Testing and verification
- Incremental builds*



How?



Maven



pom.xml

Project Object Model

Declarative: You define the configuration without specifying how to achieve it.

Convention: You describe what you need with specific rules.

Lifecycle: It can support everything from compilation to tests and so on.

Plugins allow you to do the unconventional heavy-lifting.

Coordinates are located in pom.xml: *groupId, artifactId, version*.

Repositories: You can load (and cache) the dependencies on demand.

Learn more: search.maven.org (Maven Central)

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Gradle



build.gradle

settings.gradle

DSL: It uses Kotlin or Groovy instead of XML.

Tasks: You can define actions which might depend on each other and be quite complex.

Plugins provide unconventional predefined tasks to do the heavy-lifting.

Modules have independent compilation units. Each unit is built into a separate JAR (or some other kind of artifact).

Repositories: You can reuse Maven repositories.

Dependency Management: You can easily declare and resolve dependencies.

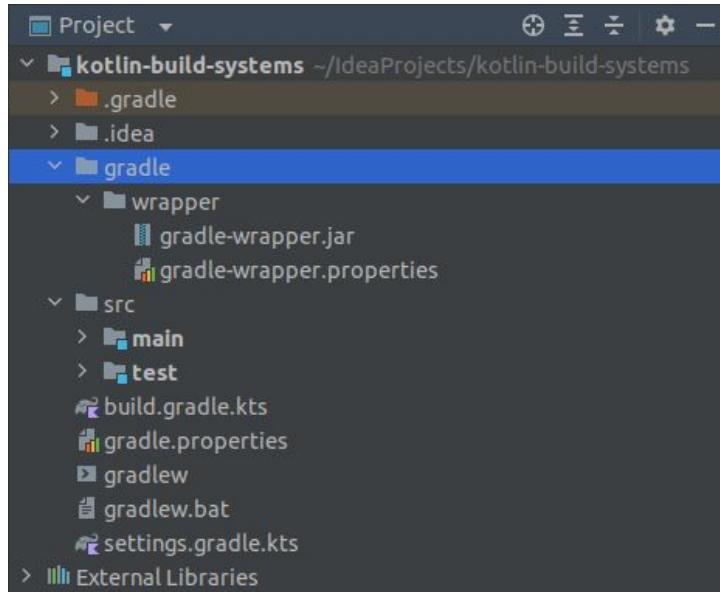
Language Agnostic: Gradle can be used for Kotlin, Java, Scala, C++, JS, and COBOL.

Learn more: docs.gradle.org

Gradle project structure

```
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── src
├── build.gradle.kts / build.gradle
├── gradle.properties
└── gradlew
    └── gradlew.bat
    └── settings.gradle.kts / settings.gradle
```

Don't push
to GitHub



- Gradle root project == IntelliJ IDEA project
- Gradle project != IntelliJ IDEA project
- Gradle module != IntelliJ IDEA module
- Gradle project ~ IntelliJ IDEA module
- Gradle root project might have subprojects that have subprojects and so on
- Tasks may be defined in any project

Gradle DSL

Fill out the **build.gradle** or **build.gradle.kts** file to set up the project.

```
plugins {
    kotlin("jvm") version "1.7.10"
}

repositories {
    mavenCentral()
}

dependencies {
    implementation(kotlin("stdlib"))
}

tasks {
    withType<JavaCompile> {
        targetCompatibility = "11"
    }
}
```

Gradle repositories

Specify where to find the libraries needed by the project. The search is carried out from top to bottom

```
repositories {  
    mavenCentral()  
    google()  
    maven {  
        url = uri("https://your.company.com/maven")  
        credentials {  
            username = "admin"  
            password = "12345"  
        }  
    }  
    flatDir {  
        dirs("libraries")  
    }  
}
```

Don't push the credentials to GitHub, please!
Use secrets, environmental variables, etc.

Gradle dependencies

- `compilationOnly` – Used only during compilation
- `runtimeOnly` – Used only during runtime
- `implementation` – Used in both
- `api` – Dependency “leaks”, meaning you can access its dependencies

- `testCompilationOnly`
- `testRuntimeOnly`
- `testImplementation`
- `testApi`

Gradle dependencies

```
val ktorVersion: String = "6.6.6"

dependencies {
    // string notation, e.g. group:name:version
    implementation("commons-lang:commons-lang:2.6")
    implementation("io.ktor:ktor-serialization-jackson:$ktorVersion")
    // map notation:
    implementation("org.jetbrains.kotlinx", "kotlinx-datetime", "7.7.7")
    // dependency on another project
    implementation(project(":neighborProject"))
    // putting all jars from 'libs' onto the compile classpath
    implementation(fileTree("libs"))
    // api dependency - internals are accessible
    api("io.ktor:ktor-server-content-negotiation:$ktorVersion")
    // test dependencies
    testImplementation("org.jetbrains.kotlin:kotlin-test-junit")
    testImplementation(kotlin("test"))
}
```

Gradle dependencies

```
dependencies {  
    implementation("org.hibernate:hibernate") {  
        version {  
            // If there is a version conflict, strictly select version "3.1" of hibernate  
            strictly("3.1")  
        }  
        exclude(module = "cglib") // by artifact name  
        exclude(group = "org.jmock") // by group  
        exclude(group = "org.unwanted", module = "buggyModule") // by both  
        // disabling all transitive dependencies of this dependency  
        isTransitive = false  
    }  
}
```

BOM

There are direct and transitive dependencies, which may lead to version conflicts.

myProject → thing:1.0 → anotherThing:1.1

myProject → thirdThing:1.0 → anotherThing:1.2

Maven's Bill Of Materials (BOM) offers a solution.

```
val ktorVersion: String = "2.0.0"

dependencies {
    implementation(enforcedPlatform("io.ktor:ktor-bom:$ktorVersion"))
    implementation(enforcedPlatform("io.ktor:ktor-server-core"))
    implementation(enforcedPlatform("io.ktor:ktor-server-netty"))
}
```

Gradle tasks

A task is a set of instructions for Gradle to perform:

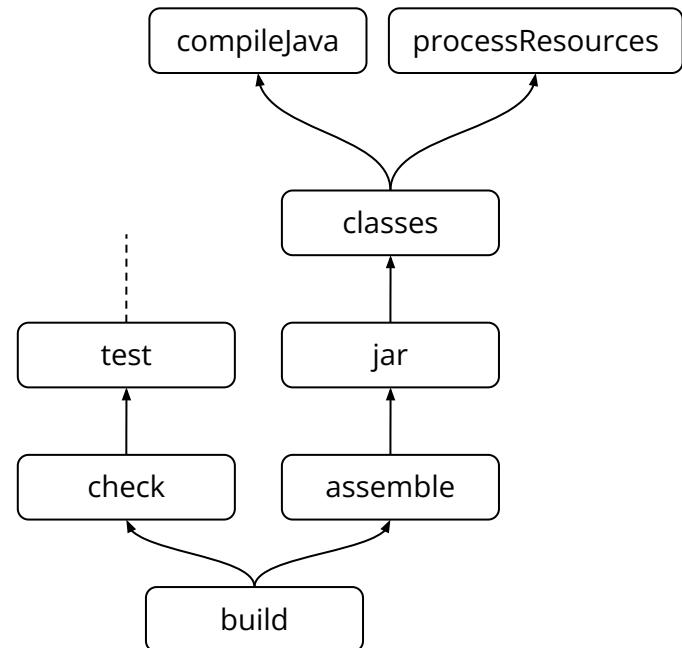
- Compile the source code
- Run tests
- Build a JAR
- Publish to Maven (or somewhere else)
- Etc.

There are default tasks, tasks from plugins, and your custom tasks.

The most popular tasks are `init`, `wrapper`, `build`, `clean`, `test`

For all available tasks, see `./gradlew tasks`

Partial task graph for a standard Java build



Gradle tasks

```
// build.gradle.kts - buildfile
tasks.create<Copy>("copy") {
    description = "Copies sources to the destination directory"
    group = "Custom"

    from("src")
    into("dst")
}
```

Older versions of Gradle only support the `create(...)` API, which eagerly creates and configures tasks when it is called and should be avoided. You can use `register(...)` instead.

Gradle tasks

Kotlin (and Groovy) are actually too powerful to be build configuration languages.

```
tasks.register("Fib") {  
    var first = 0  
    var second = 1  
    doFirst {  
        println("What is going on?")  
        for (i in 1..11) {  
            second += first  
            first = second - first  
        }  
    }  
    doLast {  
        println("Result = $first")  
    }  
}
```

```
> Task :Fib  
What is going on?  
Result = 89
```



Gradle tasks

```
abstract class FibonacciTask : DefaultTask() {  
    @get:Input  
    abstract val n: Property<Int>  
  
    @TaskAction  
    fun execute() {  
        if (n.get() < 0) {  
            throw StopExecutionException("n must be non-negative")  
        }  
        var first = 0  
        var second = 1  
        for (i in 1..n.get()) {  
            second += first  
            first = second - first  
        }  
        println("Result = $first")  
    }  
}  
  
tasks.register<FibonacciTask>("Fib_11") {  
    n.set(11)  
}
```

Gradle tasks

Tasks can have dependencies on each other.

```
tasks.withType<Test> {  
    dependsOn(tasks.withType<PublishToMavenLocal>{}, "jar_name")  
}
```

All PublishToMavenLocal tasks will be executed before **all** Test tasks

Gradle plugins

Plugin – A set of tasks that help deal with something specific: Kotlin, Java, Protobuf, etc.

Plugins block is compiled separately before everything else. Artifacts are placed in the classpath, which allows the IDE to provide auto-completion and other useful features.

``apply false`` is useful when you need the plugin for some subprojects.

```
plugins {  
    kotlin("jvm") version "1.7.10"  
    application // A plugin that runs a project as a Java application  
    id("org.jlleitschuh.gradle.ktlint") version "10.3.0" apply false  
}
```

Gradle plugins

```
// projectRoot/build.gradle.kts
class SamplePlugin : Plugin<Project> {
    override fun apply(target: Project) {
        target.tasks.register("pluginTask") {
            doLast { println("A plugin task was called") }
        }
    }
}

apply<SamplePlugin>()
```

Gradle plugins

```
// projectRoot/buildSrc/build.gradle.kts
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile

plugins {
    kotlin("jvm") version "1.7.10"
    `java-gradle-plugin`
}

gradlePlugin {
    plugins {
        create("samplePlugin") {
            id = "sample"
            implementationClass = "SamplePlugin"
        }
    }
}

repositories {
    mavenCentral()
}
```

```
// projectRoot/buildSrc/src/main/kotlin/SamplePlugin.kt
import org.gradle.api.XXX

abstract class SampleTask : DefaultTask() {
    init {
        description = "Just a sample template task"
        group = "custom"
    }

    @get:Input
    @get:Option(description = "Whom to greet?")
    abstract val username: Property<String>
    // property `name` is reserved ;^)

    @TaskAction
    fun greet() {
        logger.lifecycle("Name is: ${username.orNull}")
        println("Hello, ${username.orNull}!")
    }
}
```

Gradle plugins

```
// projectRoot/buildSrc/src/main/kotlin/SamplePlugin.kt [CONTINUED]

class SamplePlugin : Plugin<Project> {
    override fun apply(target: Project) {
        target.tasks.register("PluginTask", SampleTask::class.java) { task →
            task.username.set("world")
        }
    }
}
```

Gradle properties

Properties are used to configure the behavior of Gradle itself and specific projects.

From highest to lowest precedence:

- **Command-line flags**, such as `--build-cache`
- Properties stored in a local `gradle.properties` file.
- Properties stored in the `~/.gradle/gradle.properties` file.
- **Gradle properties**, such as `org.gradle.caching=true`, which are typically stored in a `gradle.properties` file in a project root directory or `GRADLE_USER_HOME` environment variable.
- **Environment variables**

For all available properties, see `./gradlew properties`

Gradle properties

```
// gradle.properties
kotlin.code.style=official
username=student

// build.gradle.kts
val username: String by project
val kotlinCodeStyle = project.property("kotlin.code.style") as String
tasks.register("printProps") {
    doLast {
        println(username)
        println(kotlinCodeStyle)
        println(System.getProperty("idea.version"))
    }
}
```

Gradle settings

Before Gradle assembles the projects for a build, it creates a `Settings` instance and loads the settings file into it. Only **one** settings file is stored in the Gradle project, and it is used to:

- add subprojects to the build
- modify the parameters from the command line, e.g., add a new project property
- access the global Gradle object to register lifecycle handlers

Gradle settings

```
// settings.gradle.kts
rootProject.name = "Project's name"
include(
    "Module1",
    "Module2"
)

// Include a repository from GitHub
sourceControl {
    gitRepository(URI.create("Repository URL")) {
        producesModule("Module")
    }
}
```

Gradle wrapper

A Gradle wrapper (gradlew) is a shell script that downloads and caches the required version of Gradle.

- gradlew – used in *nix
- gradlew.bat – used in Windows

The version is specified in projectRoot/gradle/wrapper/gradle-wrapper.properties:

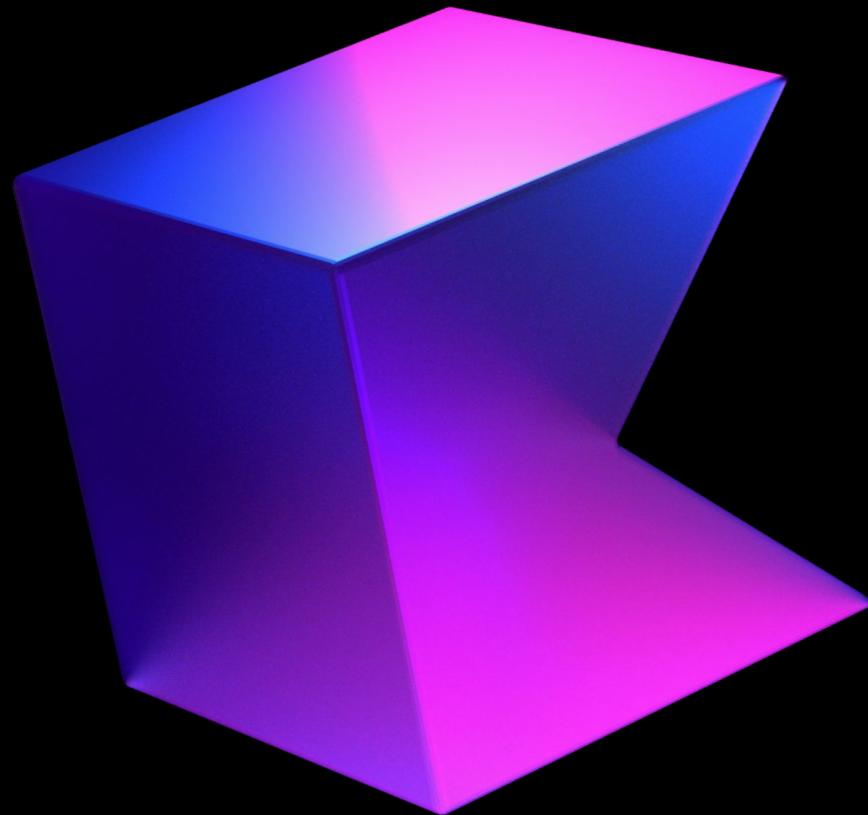
`distributionUrl=https://services.gradle.org/distributions/gradle-7.5.1-bin.zip`

Gradle can do so much more!

Gradle support many additional features which we won't be covering today:

- Caching
- Multi-module projects
- More blocks:
 - `allprojects { }` and `subprojects { }`
 - `publishing { }`
 - `artifacts { }`
- Compatibility
- Resolution strategies
- Source sets

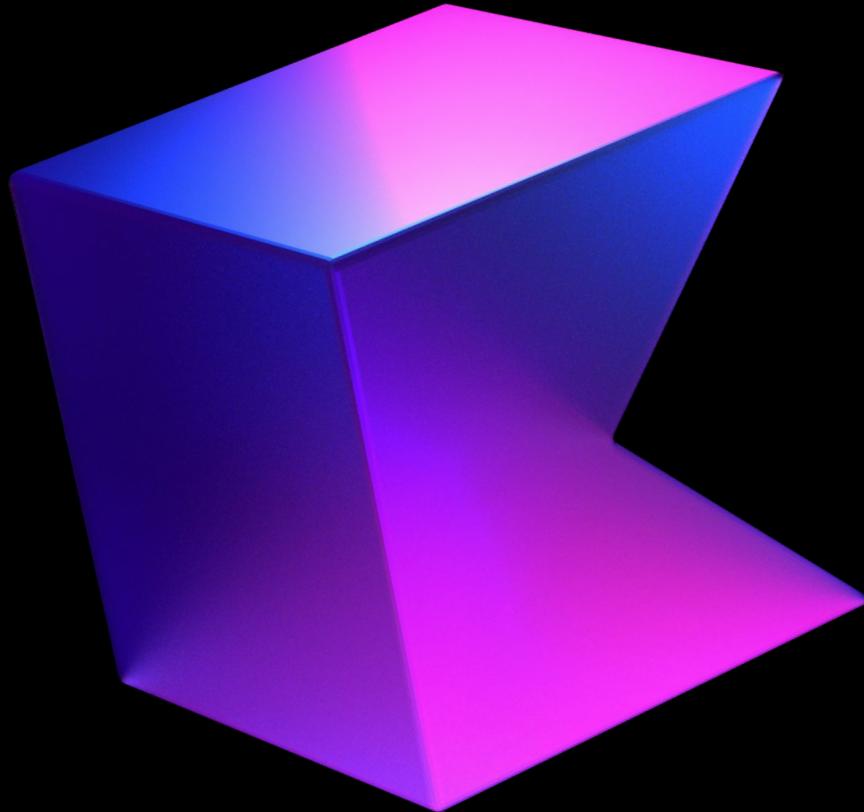
Thanks!



@kotlin



Generics



@kotlin

What? Why?

```
fun quickSort(collection: CollectionOfInts) { ... }
quickSort(listOf(1, 2, 3)) // OK
quickSort(listOf(1.0, 2.0, 3.0)) // NOT OK
```

```
fun quickSort(collection: CollectionOfDoubles) { ... } // overload (we'll get back to this a bit later)
quickSort(listOf(1.0, 2.0, 3.0)) // OK
quickSort(listOf(1, 2, 3)) // OK
```

Kotlin Number inheritors: Int, Double, Byte, Float, Long, Short

Do we need 4 more implementations of quickSort?

How?

Does the quickSort algorithm actually care what it is sorting? No, as long as it can compare two values against each other.

```
fun <T : Comparable<T>> quickSort(collection: Collection<T>): Collection<T> { ... }

quickSort(listOf(1.0, 2.0, 3.0)) // OK

quickSort(listOf(1, 2, 3)) // OK

quickSort(listOf("one", "two", "three")) // OK
```

How?

Generics allow you to write code that can work with any type or with types that should satisfy some rules (constraints) but are not limited in any other ways: type parameters.

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

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

```
val cupHolder = Holder("cup") // Generic parameter type can be inferred
```

Constraints

Sometimes we do not want to work with an arbitrary type and expect it to provide us with some functionality. In such cases type constraints in the form of upper bounds are used: upper bounds.

```
class Pilot<T : Movable>(val vehicle: T) {  
    fun go() { vehicle.move() }  
}  
  
val ryanGosling = Pilot<Car>(Car("Chevy", "Malibu"))  
val sullySullenberger = Pilot<Plane>(Plane("Airbus", "A320"))
```

Constraints continued

There can be several parameter types, and generic classes can participate in inheritance.

```
public interface MutableMap<K, V> : Map<K, V> { ... }
```

There can also be several constraints (which means the type parameter has to implement several interfaces):

```
fun <T, S> moveInAnAwesomeWayAndCompare(a: T, b: S) where T : Comparable<T>,  
S : Comparable<T>, T : Awesome, T : Movable { ... }
```

Star-projection

When you do not care about the parameter type, you can use *star-projection* * (Any? / Nothing).

```
fun printKeys(map: MutableMap<*, *>) { ... }
```

Let's go back

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

Nothing <: C <: B <: A <: Any

This means that the Any class is the *superclass* for all the classes and at the same time Nothing is a *subtype* of any type

What is next?

Consider a basic example:

```
interface Holder<T> {
    fun push(newValue: T) // consumes an element

    fun pop(): T // produces an element

    fun size(): Int // does not interact with T
}
```

What is next?

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

In Kotlin there are type projections:

```
G<T> // invariant, can consume and produce elements  
G<in T> // contravariant, can only consume elements  
G<out T> // covariant, can only produce elements  
G<*> // star-projection, does not interact with T
```

Several examples

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

interface Holder<T> {
    fun push(newValue: T) // consumes an element: OK

    fun pop(): T // produces an element: OK

    fun size(): Int // does not interact with T: OK
}
```

Several examples

```
G<in T> // contravariant, can only consume elements

interface Holder<in T> {
    fun push(newValue: T) // consumes an element: OK

    fun pop(): T // produces an element: ERROR: [TYPE_VARIANCE_CONFLICT_ERROR] Type
parameter T is declared as 'in' but occurs in 'out' position in type T

    fun size(): Int // does not interact with T: OK
}
```

Several examples

```
G<out T> // covariant, can only produce elements

interface Holder<out T> {
    fun push(newValue: T) // consumes an element: ERROR:
[TYPE_VARIANCE_CONFLICT_ERROR] Type parameter T is declared as 'out' but occurs in
'in' position in type T

    fun pop(): T // produces an element: OK

    fun size(): Int // does not interact with T: OK
}
```

Several examples

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

fun <T> foo1(holder: Holder<T>, t: T) {
    holder.push(t) // OK
}
fun <T> foo2(holder: Holder<*>, t: T) {
    holder.push(t) // ERROR: [TYPE_MISMATCH] Type mismatch. Required: Nothing. Found: T
}

fun <T> foo1(holder: Holder<Any>, t: Any) {
    holder.push(t) // OK
}
```

Subtyping

```
open class A
open class B : A()      ----> Nothing <: C <: B <: A <: Any
class C : B()

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

Holder<Nothing> ??? Holder<C> ??? Holder<B> ??? Holder<A> ??? Holder<Any>
```

Subtyping

```
open class A  
open class B : A()      ----> Nothing <: C <: B <: A <: Any  
class C : B()
```

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

Holder<Nothing> ~~<: B~~ Holder<C> ~~<: B~~ Holder ~~<: B~~ Holder<A> ~~<: B~~ Holder<Any>
Generics are invariant!!

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

VS

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

Subtyping

```
open class A
open class B : A()      ----> Nothing <: C <: B <: A <: Any
class C : B()

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

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

BUT

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

Subtyping

```
class Holder<T> (var value: T?) {  
    fun pop(): T? = value.also { value = null }  
    fun push(newValue: T?): T? = value.also { value = newValue }  
    fun steal(other: Holder<T>) { value = other.pop() }  
    fun gift(other: Holder<T>) { other.push(pop()) }  
}  
  
Holder<Nothing> <: Holder<C> <: Holder<B> <: Holder<A> <: Holder<Any>
```

```
val holderB: Holder<B> = Holder(B())  
val holderA: Holder<A> = Holder(null)  
holderA.steal(holderB) // ERROR: Type mismatch. Required: Holder<A>. Found: Holder<B>.  
holderB.gift(holderA) // ERROR: Type mismatch. Required: Holder<B>. Found: Holder<A>.
```

Type projection: `in`

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

Type projection: `other` is a restricted (projected) generic. You can only call methods that **accept** the type parameter `T`, which in this case means that you can only call `push()`.

This is contravariance:

```
Nothing <: C <: B <: A <: Any  
Holder<Nothing> :> Holder<C> :> Holder<B> :> Holder<A> :> Holder<Any>
```

Type projection: out

```
class Holder<T> (var value: T?) {  
    ...  
    fun steal(other: Holder<out T>) { value = other.pop() }  
}  
holderA.steal(holderB) // OK
```

Type projection: `other` is a restricted (projected) generic. You can only call methods that **return** the type parameter `T`, which in this case means that you can only call `pop()`.

This is covariance:

```
Nothing <: C <: B <: A <: Any  
Holder<Nothing> <: Holder<C> <: Holder<B> <: Holder<A> <: Holder<Any>
```

Type projections

```
class Holder<T> (var value: T?) {  
    fun steal(other: Holder<out T>) {  
        val oldValue = push(other.pop())  
        other.push(oldValue) // ERROR: Type mismatch. Required: Nothing?. Found: T?.  
    }  
    fun gift(other: Holder<in T>) {  
        val otherValue = other.push(pop())  
        push(otherValue) // ERROR: Type mismatch. Required: T?. Found: Any?.  
    }  
}
```

`out T` returns something that can be cast to `T` and accepts literally `Nothing`.

`in T` accepts something that can be cast to `T` and returns a meaningless `Any?`.

Type erasure

At runtime, the instances of generic types do not hold any information about their actual type arguments. The type information is said to be erased. The same byte-code is used in all usages of the generic as opposed to C++, where each template is compiled separately for each type parameter provided.

- Any `MutableMap<K, V>` becomes `MutableMap<*, *>` in the runtime*.
- Any `Pilot<T : Movable>` becomes `Pilot<Movable>`.

* Actually, in the **Kotlin/JVM** runtime we have just `java.util.Map` to preserve compatibility with Java.

Type erasure

As a corollary, you cannot override a function (**in Kotlin/JVM**) by changing generic type parameters:

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

Both become `quickSort(collection: Collection<*>)` and their signatures clash.

But you can use the `JvmName` annotation:

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

Nullability in generics

Contrary to common sense, in Kotlin a type parameter specified as `T` can be nullable.

```
class Holder<T>(val value: T) { ... } // Notice there is no `?`  
val holderA: Holder<A?> = Holder(null) // T = A? and that is OK
```

To prohibit such behavior, you can use a non-nullable `Any` as a constraint.

```
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?.
```

You may also find intersection helpful:

```
fun <T> elvisLike(x: T, y: T & Any): T & Any = x ?: y  
T & Any is populated with all values from T besides null
```

Inline functions

If they are used as first-class objects, functions are stored as objects, thus requiring memory allocations, which introduce runtime overhead.

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

Inline functions

```
fun foo(str: String, call: (String) → Unit) {  
    call(str)  
}  
  
public static final void foo(@NotNull String str, @NotNull Function1 call) {  
    Intrinsics.checkNotNullParameter(str, "str");  
    Intrinsics.checkNotNullParameter(call, "call");  
    call.invoke(str);  
}  
public static final void main() {  
    foo("Top level function with lambda example", (Function1)foo$call$lambda$1.INSTANCE);  
}
```

This call invokes the print function by passing the string as an argument.

Inline functions

```
public static final void foo(@NotNull String str, @NotNull Function1 call) {  
    Intrinsics.checkNotNullParameter(str, "str");  
    Intrinsics.checkNotNullParameter(call, "call");  
    call.invoke(str);  
}
```

“Under the hood” an instance of a Function class is created, i.e. allocated:

```
foo("...", new Function() {  
    @Override  
    public void invoke() {  
        ...  
    }  
});
```

Inline functions

We can use the `inline` keyword to inline the function, copying its code to the call site:

```
inline fun foo(str: String, call: (String) → Unit) {  
    call(str)  
}  
fun main() {  
    foo("Top level function with lambda example", ::print)  
}  
  
public static final void main() {  
    String str$iv = "Top level function with lambda example";  
    int $i$f$foo = false;  
    int var3 = false;  
    System.out.print(str$iv);  
}
```



Inline functions

`inline` affects not only the function itself, but also all the lambdas passed as arguments. If you do not want some of the lambdas passed to an `inline` function to be inlined (*for example, inlining large functions is not recommended*), you can mark some of the function parameters with the `noinline` modifier.

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

Inline functions

You can use `return` in inlined lambdas, this is called *non-local return*, which can lead to unexpected behaviour:

```
inline fun foo(call1: () → Unit, call2: () → Unit) {  
    call1()  
    call2()  
}  
  
fun main() {  
    println("Step#1")  
    foo({ println("Step#2")  
        return ,  
        { println("Step#3") } })  
    println("Step#4")  
}
```

→ Output:
Step#1
Step#2

Inline functions

To prohibit returning from the lambda expression we can mark the lambda as `crossinline`.

```
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")
}
```

`return@foo` is allowed and
fine, though

Inline functions

`crossinline` is especially useful when the lambda from an `inline` function is being called from another context, for example, if it is used to instantiate a `Runnable`:

```
inline fun drive(crossinline specialCall: (String) → Unit, call: (String) → Unit) {  
    val nightCall = Runnable { specialCall("There's something inside you") }  
    call("I'm giving you a nightcall to tell you how I feel")  
    thread { nightCall.run() }  
    call("I'm gonna drive you through the night, down the hills")  
}  
fun main() {  
    drive({ System.out.println(it) }) { println(it) }  
}
```

Inline reified functions

Sometimes you need to access a type passed as a parameter:

```
fun <T: Animal> foo() {  
    println(T::class) // ERROR: Cannot use 'T' as reified type parameter. Use a class instead  
    --- add a param: t: KClass<T>  
}
```

You can use the `reified` keyword with `inline` functions:

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

Note that the compiler has to be able to know the actual type passed as a type argument so that it can modify the generated bytecode to use the corresponding class directly.

```
open class A
class B : A()
class C : A() { fun consume(other: A): C = this }

fun <T, S : R, R> funny(
    source: Iterator<????>,
    target: MutableCollection<????>,
    base: ???,
    how: ????
) {
    var result: R = base
    for (value in source) {
        result = how(result, value)
        target.add(result)
    }
}

fun main() {
    val wtf = mutableListOf<A>()
    val src = mapOf(3.14 to B(), 2 to B(), "Hello" to B())
    val c = C()
    funny(src.values.iterator(), wtf, c) { r, t → r.consume(t) }
}
```

(in)Variance

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

open class A

open class B : A()

class C : B()
```

A hierarchy is in place (and don't forget about the same hierarchy with nullability):

Nothing → C → B → A → Any

Variance of Generics would give us another hierarchy:

Holder<Nothing> → Holder<C> → Holder → Holder<A> → Holder<Any>

But this is not the case, since Generics are invariant.

```
val holderC = Holder(C())

val holderB: Holder<B> = holderC // Error: Type mismatch. Required: Holder<B>. Found: Holder<C>.
```

NB: code below works, since C() passed as an argument is being cast to B, nothing to do with variance.

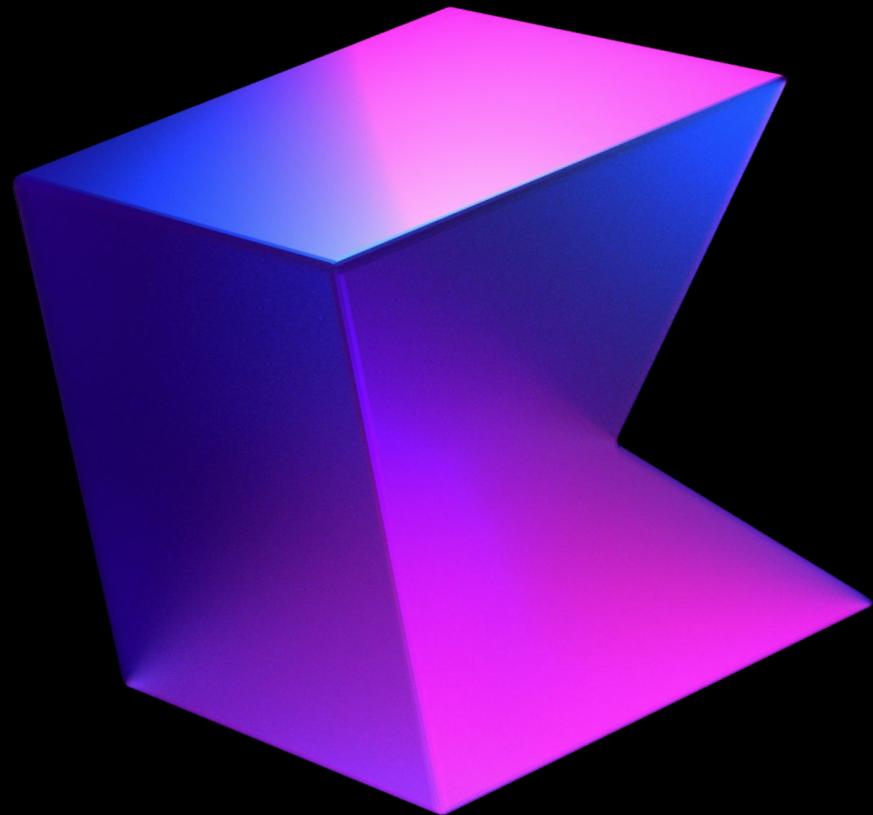
```
val kotlinIsSmart: Holder<B> = Holder(C())
```

More

```
class Holder<T> (var value: T?) {  
    fun pop(): T? = value.also { value = null }  
    fun push(newValue: T?): T? = value.also { value = newValue }  
    fun steal(other: Holder<T>) { value = other.pop() }  
    fun gift(other: Holder<T>) { other.push(pop()) }  
}  
  
val holderB: Holder<B> = Holder(B())  
val holderA: Holder<A> = Holder(null)  
holderA.steal(holderB) // Error: Type mismatch. Required: Holder<A>. Found: Holder<B>.  
holderB.gift(holderA) // Error: Type mismatch. Required: Holder<B>. Found: Holder<A>.
```

But why not? B can be easily cast to A inside `steal` or `gift` and everything should be fine.

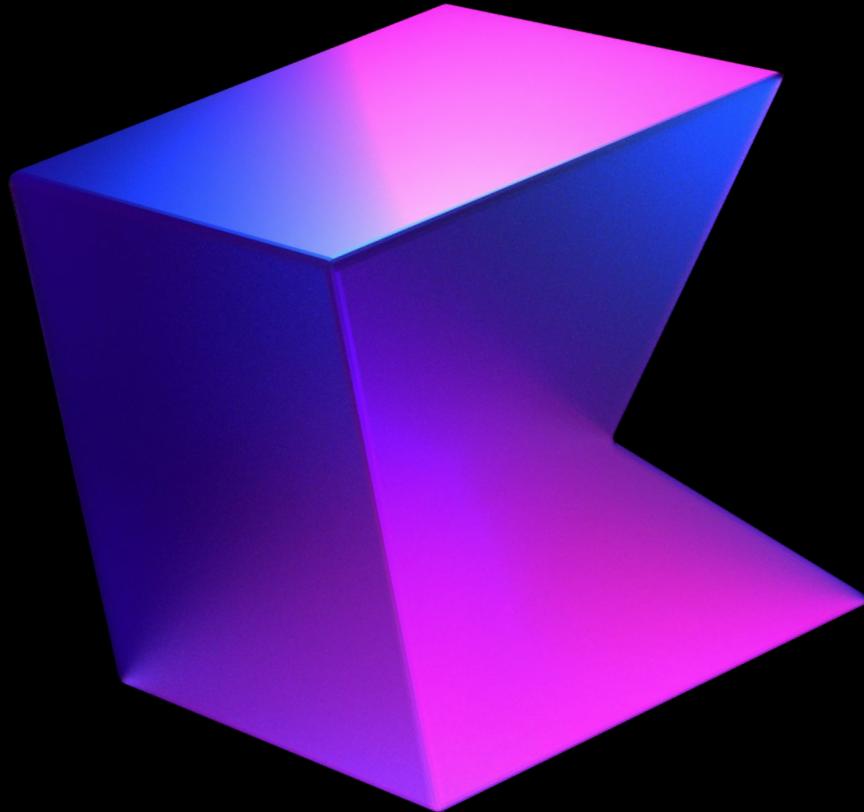
Thanks!



@kotlin



Collections and Co.



@kotlin

What are they?

A collection usually contains a number of objects (this number may also be zero) of the same type.

Objects in a collection are called elements or items.

- Lists are ordered collections with access to elements by indices – integer numbers that reflect their position. Elements can occur more than once in a list.
- Sets are collections of unique elements. They reflect the mathematical abstraction of “set”: a group of objects without duplicates.
- Maps (or dictionaries) are sets of key-value pairs. The keys are unique, and each of them maps to exactly one value, while the values can be duplicated.

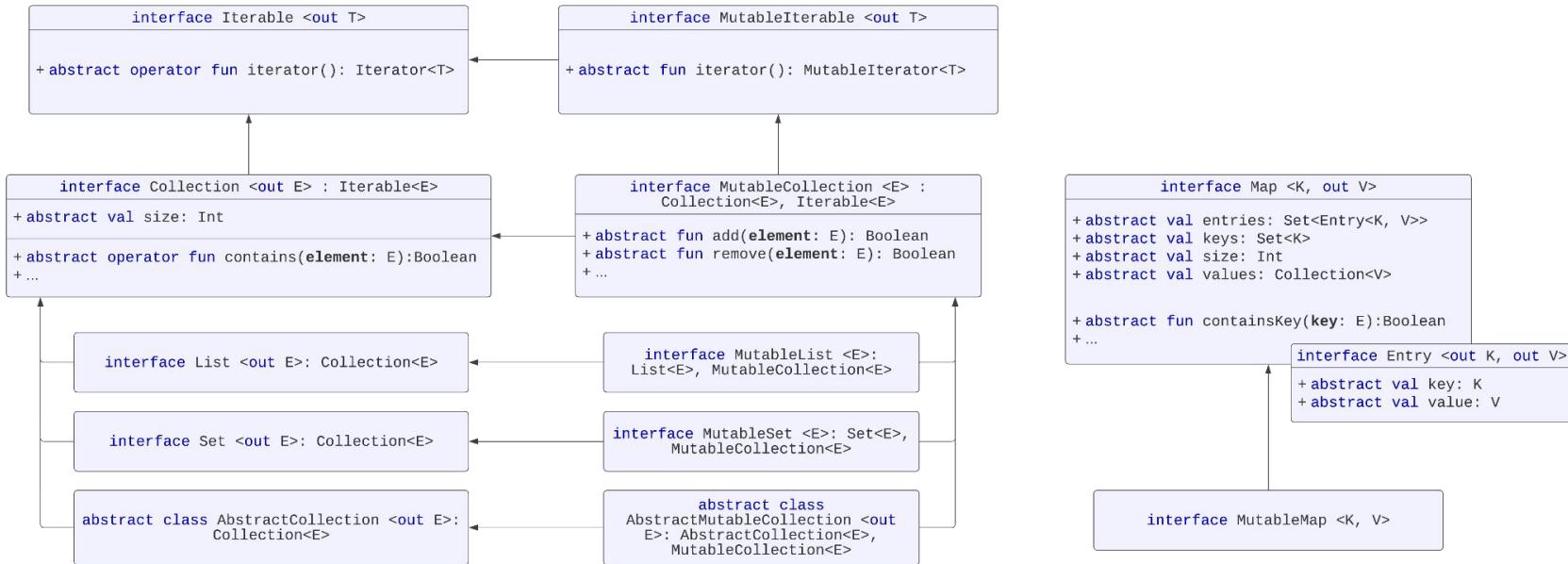
How can they be used?

Kotlin lets you manipulate collections independently of the exact types of objects stored in them.

In other words, you add a `String` to a list of `Strings` the same way as you would do with `Ints` or a user-defined class.

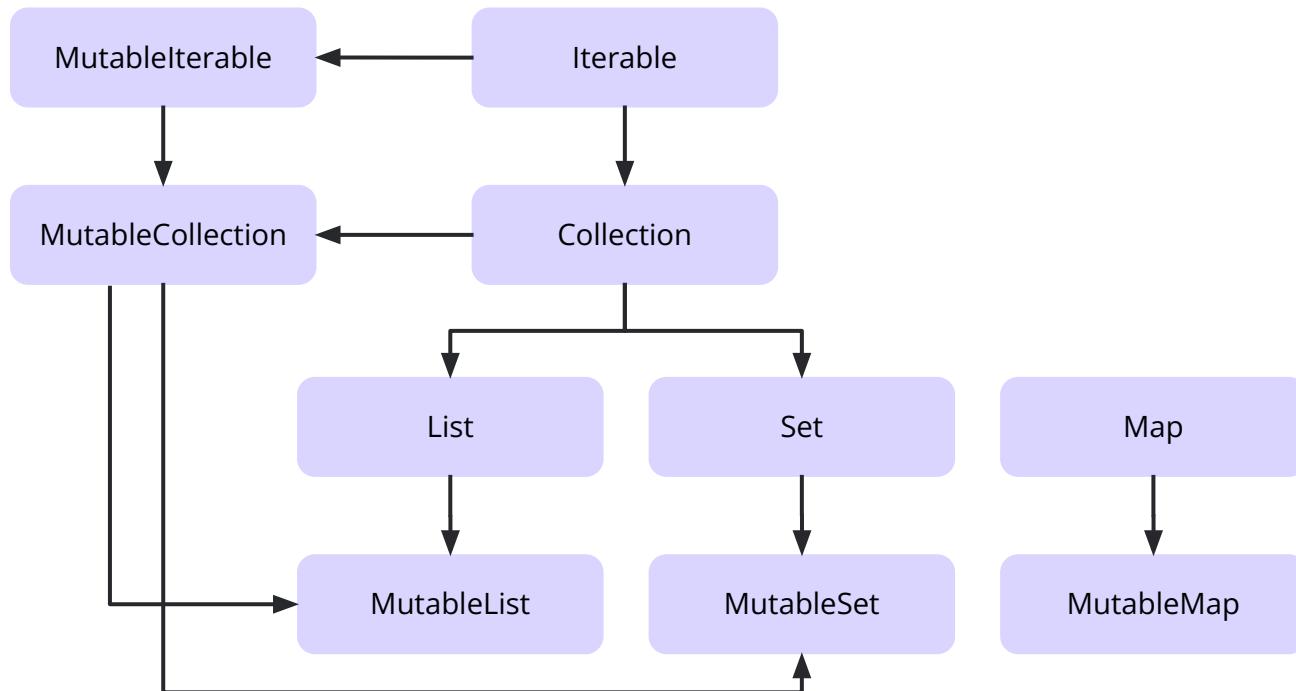
So, the Kotlin Standard Library offers generic interfaces, classes, and functions for creating, populating, and managing collections of any type.

Taxonomy of collections



Interfaces – Kotlin actually uses implementations from `java.util`

Taxonomy of collections



Iterable

All collections in Kotlin implement **Iterable** interface:

```
/*
 * Classes that inherit from this interface can be represented as a sequence of elements that can be
 * iterated over.
 *
 * @param T is the type of element being iterated over. The iterator is covariant in its element
 * type.
 */
public interface Iterable<out T> {
    // Returns an iterator over the elements of this object.
    public operator fun iterator(): Iterator<T>
}
```

Iterable

All collections in Kotlin are **Iterable**:

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

Iterable vs MutableIterable

All collections in Kotlin are Iterable:

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

But some of them are MutableIterable:

```
val iterator = myMutableIterableCollection.iterator()
while (iterator.hasNext()) {
    iterator.next()
    iterator.remove() // Because it is a mutable iterator
}
```

Different kinds of collections

There are 2 kinds of collections: `Collection` and `MutableCollection`. `Collection` implements only `Iterable` interface, while `MutableCollection` implements `Collection` and `MutableIterable` interfaces.

`Collection` allows you to read values and make the collection **immutable**.

`MutableCollection` allows you to change the collection, for example by adding or removing elements. In other words, it makes the collection **mutable**.

```
val readonlyCollection = listOf(1, 2, 3)  
readonlyCollection.add(4) // ERROR: Unresolved reference: add
```

```
val mutableCollection = mutableListOf(1, 2, 3)  
mutableCollection.add(4) // OK
```

Mutable Collection != Mutable Variable

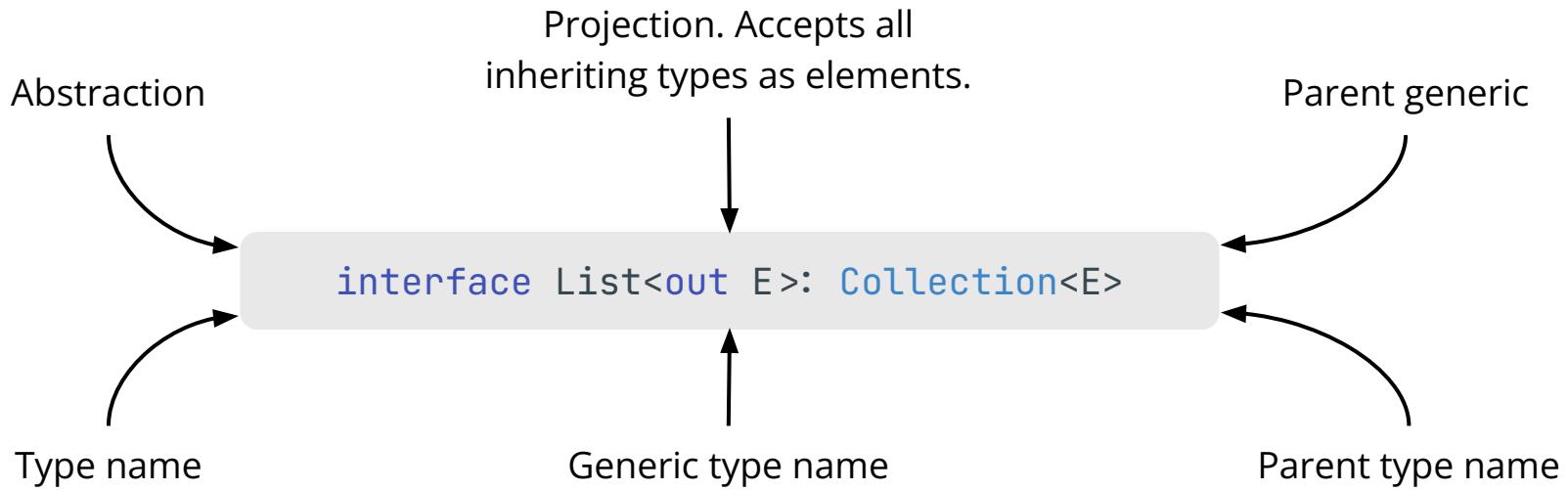
If you create a mutable collection, you **cannot** reassign the `val` variable.

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

But you can reassign `var`.

```
var mutableCollection = mutableListOf(1, 2, 3)
mutableCollection.add(4) // OK
mutableCollection = mutableListOf(4, 5, 6) // OK
```

The anatomy of a collection



The anatomy of a collection

Each collection has several **base** methods:

```
public interface Collection<out E> : Iterable<E> {  
    public val size: Int  
  
    public fun isEmpty(): Boolean ← Use this instead of size == 0  
  
    public operator fun contains(element: @UnsafeVariance E): Boolean  
  
    public fun containsAll(elements: Collection<@UnsafeVariance E>): Boolean  
    ...  
}
```

The anatomy of a collection

Actually there are **many** extensions:

```
public val Collection<*>.indices: IntRange  
    get() = 0..size - 1
```

Convenient to use in loops:

`for (i in collection.indices) { ... }`

```
public val <T> List<T>.lastIndex: Int  
    get() = this.size - 1
```

```
public inline fun <T> Collection<T>.isNotEmpty(): Boolean = !isEmpty()
```

...



Use this instead of `size ≠ 0`

Collections under the hood: List

```
public interface List<out E> : Collection<E> {  
    public operator fun get(index: Int): E  
    ...  
}
```

Convenient to use with []: collection[2]

```
public fun indexOf(element: @UnsafeVariance E): Int  
public fun lastIndexOf(element: @UnsafeVariance E): Int
```

```
public fun subList(fromIndex: Int, toIndex: Int): List<E>
```

Make a **referenced** copy:

```
val list1 = mutableListOf(1, 2, 3)  
val list2 = list1.subList(0, 1)  
list1[0] += 1  
println(list1) // [2, 2, 3]  
println(list2) // [2]
```

Collections under the hood: List

To create a new list you can use special **builders** (by default `ArrayList`):

```
val list1 = emptyList<Int>() // Builds the internal object EmptyList
val list2 = listOf<Int>() // Calls emptyList()
val list3 = listOf(1, 2, 3) // The type can be inferred

val list4 = mutableListOf<Int>() // But better: ArrayList<Int>()
val list5 = mutableListOf(1, 2, 3) // The type can be inferred
val list6 = buildList {
    // constructs MutableList<Int>
    add(5)
    addAll(0, listOf(1, 2, 3))
}
```

Collections under the hood: Set

```
public interface Set<out E> : Collection<E> {  
    abstract val size: Int  
  
    abstract fun contains(element: @UnsafeVariance E): Boolean  
  
    abstract fun containsAll(collection: Collection<E>): Boolean  
  
    abstract fun isEmpty(): Boolean  
  
    abstract fun iterator(): Iterator<E>  
}
```

A generic unordered collection of elements that does not support duplicate elements.

It compares objects via the equals method instead of checking if the objects are the *same*.

Collections under the hood: 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 a = A(1,1)
    val b = A(1,2)
    val set = setOf(a, b)
    println(set) // two elements
}
```

Collections under the hood: 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 a = B(1,1)
    val b = B(1,2)
    val set = setOf(a, b)
    println(set) // only one element
}
```

Collections under the hood: Set

To create a new set you can use special **builders** (by default `LinkedHashSet`):

```
val set1 = emptySet<Int>() // Builds the internal object EmptySet
val set2 = setOf<Int>() // Calls emptySet()
val set3 = setOf(1, 2, 3) // The type can be inferred

val set4 = mutableSetOf<Int>() // But better: LinkedHashSet<Int>() or HashSet<Int>()
val set5 = mutableSetOf(1, 2, 3) // The type can be inferred
val set6 = buildSet {
    // constructs MutableSet<Int>
    add(5)
    addAll(listOf(1, 2, 3))
}
```

Collections under the hood: Map

```
public interface Map<K, out V> {  
    public fun containsKey(key: K): Boolean  
  
    public fun containsValue(value: @UnsafeVariance V): Boolean  
  
    public operator fun get(key: K): V?  
  
    public fun getOrDefault(key: K, defaultValue: @UnsafeVariance V): V  
  
    public val entries: Set<Map.Entry<K, V>>  
    ...  
}
```

Convenient to use in loops:
`for ((key, value) in map.entries) { ... }`

Collections under the hood: Map

To create a new map you can use special **builders** (by default LinkedHashMap):

```
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 = mutableMapOf<Int, String>() // But better: LinkedHashMap<...>() or HashMap<...>()
val map5 = mutableMapOf(1 to "one", 2 to "two") // The type can be inferred
val map6 = buildMap {
    // constructs MutableMap<Int, String>
    put(1, "one")
    putAll(mutableMapOf(2 to "two"))
}
```

Array

- Not a collection and not iterable, but has an **iterator**.
- Has a **fixed** size, but its elements are **mutable**.

```
/**  
 * Represents an array (specifically a Java array when targeting the JVM platform).  
 * Array instances can be created using the [arrayOf], [arrayOfNulls], and [emptyArray] standard  
library functions.  
 */  
public class Array<T> {  
    public operator fun set(index: Int, value: T): Unit  
  
    ...  
}
```

Array

Kotlin also has classes that represent arrays of primitive types without boxing overhead: `ByteArray`, `ShortArray`, `IntArray`, and so on.

```
/**  
 * An array of ints. When targeting the JVM, instances of this class are represented as `int[]`.  
 */  
public class IntArray(size: Int) {  
    public operator fun set(index: Int, value: T): Unit  
  
    ...  
}
```

Ranges

Not collections, but there are defined *progressions* for standard types with **iterators**: CharProgression, IntProgression, LongProgression:

```
for (c in 'a'..'c') { ... } // CharProgression
for (i in 1..5) { ... }    // IntProgression
for (i in 1L..5L) { ... }  // LongProgression
```

There are a lot of ways to customize them:

```
for (i in 10 downTo 0 step 3) { ... }
```

downTo and step infix extension functions.

Sequence

Not a collection, but has an **iterator**:

```
/**  
 * A sequence that returns values through its iterator. The values are evaluated lazily, and the  
sequence is potentially infinite.  
 */  
public interface Sequence<out T> {  
    public operator fun iterator(): Iterator<T>  
}
```

Sequence

To create a new sequence you can use special **builders**:

```
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 + 2 } // `it` is the previous element
println(sequence5.take(5).toList()) // [1, 3, 5, 7, 9]
```

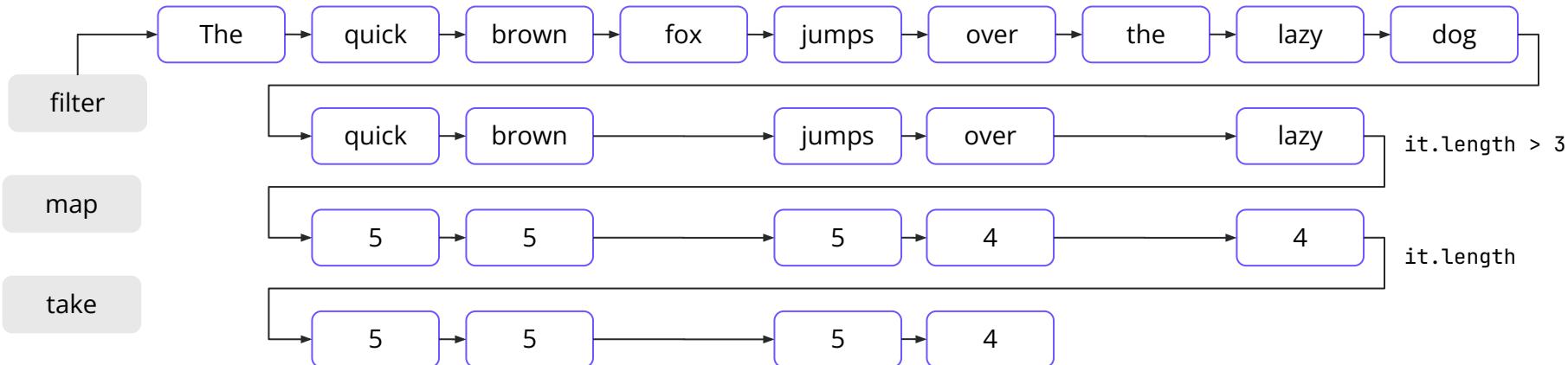
Sequence vs List

```
val words = "The quick brown fox jumps over the lazy dog".split(" ") // Returns a list
val lengthsList = words.filter { println("filter: $it"); it.length > 3 }
    .map { println("length: ${it.length}"); it.length }
    .take(4)

println("Lengths of first 4 words longer than 3 chars:")
println(lengthsList)
```

Sequence vs List

```
val words = "The quick brown fox jumps over the lazy dog".split(" ") // Returns a list
```



Sequence vs List

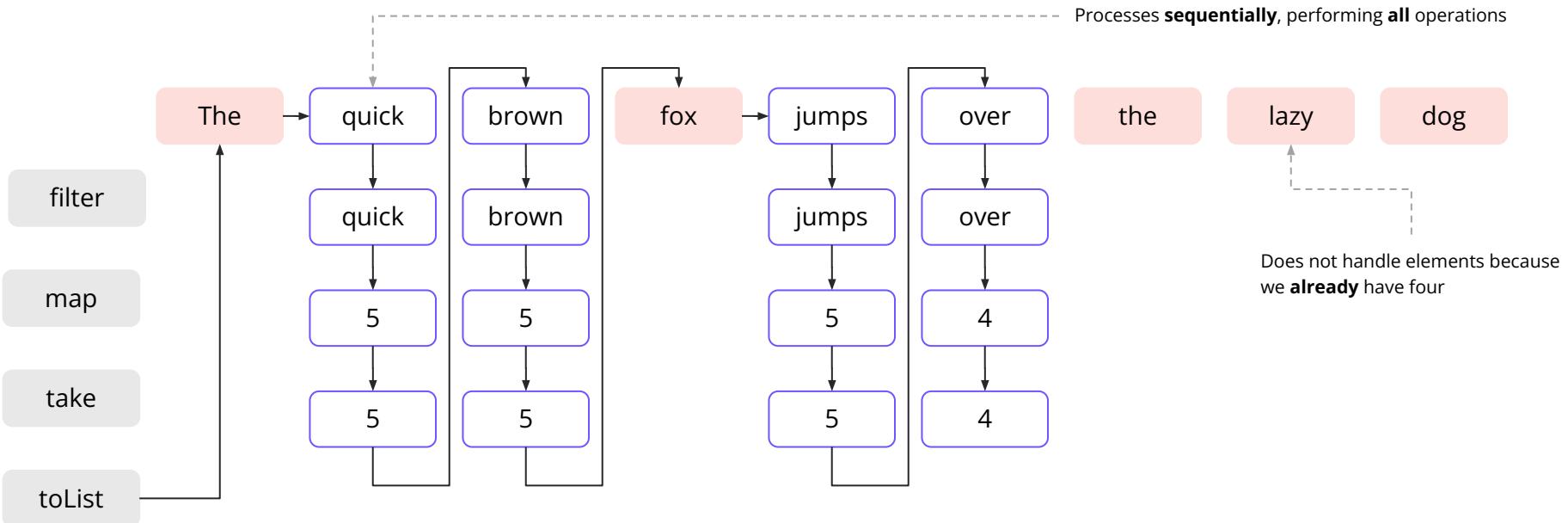
```
val words = "The quick brown fox jumps over the lazy dog".split(" ") // Returns a list
// Convert the List to a Sequence
val wordsSequence = words.asSequence()
val lengthsSequence = wordsSequence.filter { println("filter: $it"); it.length > 3 }
    .map { println("length: ${it.length}"); it.length }
    .take(4)

println(lengthsSequence) // prints `kotlin.sequences.TakeSequence@MEMORY_ADDR`

println("Lengths of first 4 words longer than 3 chars:")
// Terminal operation: obtaining the result as a List
println(lengthsSequence.toList()) // top code gets executed, then prints `[5, 5, 5, 4]`
```

Sequence vs List

```
val words = "The quick brown fox jumps over the lazy dog".split(" ") // Returns a list  
val wordsSequence = words.asList()
```



Collection operations

There are **many** different functions for working with collections. If you need to do something with a collection, Google it first. Most likely, the standard library already has the function you need, for example:

```
public fun <T : Comparable<T>> List<T?>.binarySearch(element: T?, fromIndex: Int = 0, toIndex: Int = size): Int
```

```
public actual fun <T : Comparable<T>> MutableList<T>.sort(): Unit
```

```
public inline fun <T, K, V> Iterable<T>.groupBy(keySelector: (T) → K, valueTransform: (T) → V): Map<K, List<V>>
```

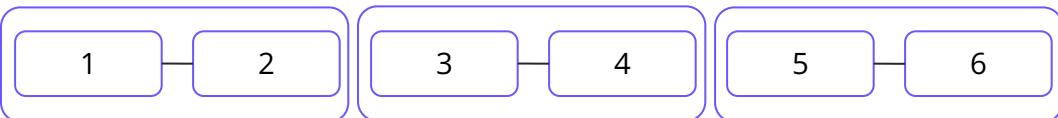
```
public inline fun <T> Iterable<T>.partition(predicate: (T) → Boolean): Pair<List<T>, List<T>>
```

Collection operations

```
val exampleList = listOf(1, 2, 3, 4, 5, 6)
```



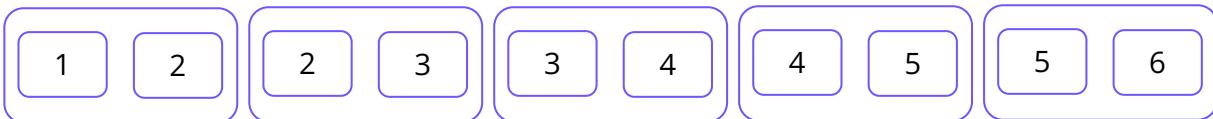
```
exampleList.chunked(2)
```



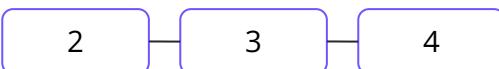
```
exampleList.chunked(2) { it.sum() }
```



```
exampleList.windowed(2)
```

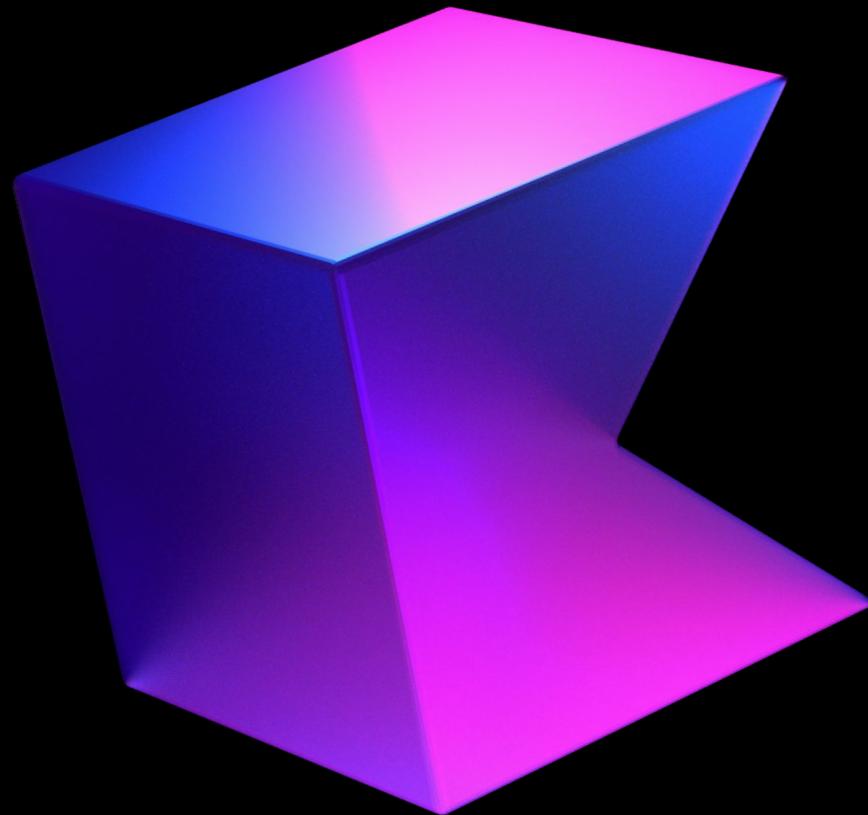


```
exampleList.drop(1).intersect(List(6) { it - 1 })
```



We will look into even more operations in our future lecture on Functional programming.

Thanks!



@kotlin



Functional Programming



What is it?

We are already familiar with object-oriented programming (OOP), but Kotlin also borrows concepts from functional programming (FP). FP is a programming paradigm where programs are constructed by **applying** and **composing functions**.

```
var sum = 0                                list.filter { it > 0 }.map { it * it }.sum()
for (item in list) {
    if (item > 0) {
        sum += item * item
    }
}
```

VS

Our approach

FP, like other concepts, has its advantages and disadvantages, but we will focus on its strengths.

Disclaimer: There won't be any deep math or Haskell examples in this lecture. We will look at what we consider to be the most important FP features that can be used in Kotlin

We already know that...

- In Kotlin you can pass functions as the arguments of other functions:

```
fun foo(bar: () → Unit): Unit { ... }
```

- If a function's last argument is a function, then it can be put outside the parentheses:

```
fun baz(start: Int, end: Int, step: (Int) → Unit): Unit { ... }  
baz(23, 42) { println("Magnificent!") }
```

- If a function's only argument is a function, then parentheses can be omitted altogether:

```
foo { println("Kotlin keeps on giving!") }
```

We already know that...

- Lambdas can be assigned to `vals` and reassigned in `vars`:

```
var lambda1: (Int) → Double = { r → r * 6.28 }  
val lambda2 = { d: Int → 3.14 * d.toDouble().pow(2) }  
lambda1 = lambda2
```

- Lambda expressions can be replaced with function syntax:

```
val sum = fun(a: Int, b: Int): Int = a + b  
val sum2 = { a:Int, b: Int → a + b }
```

- Declaring functions inside functions is allowed:

```
fun global() {  
    fun local() { ... }  
  
    ...  
    local()  
  
    ...  
}
```

Higher order functions (HOFs)

Functions that take other functions as **arguments** are called higher order functions.

In Kotlin you frequently encounter them when working with collections:

```
list.partition { it % 2 == 0 } OR list.partition { x → x % 2 == 0 }
```

Everything Kotlin allows you do with functions, which means that “functions in Kotlin are first-class citizens.”

Higher order functions (HOFs)

In functional programming, functions are designed to be pure. In simple terms, this means they cannot have a state. Loops have an iterator index, which is a state, so say goodbye to *conventional* loops.

```
fun sumIter(term: (Double) → Double, a: Double, next: (Double) → Double, b: Double): Double {  
    fun iter(a: Double, acc: Double): Double = if (a > b) acc else iter(next(a), acc + term(a))  
    return iter(a, 0.0)  
}  
  
fun integral(f: (Double) → Double, a: Double, b: Double, dx: Double): Double {  
    fun addDx(x: Double) = x + dx  
    return dx * sumIter(f, (a + (dx / 2.0)), ::addDx, b)  
}
```

(This is a LISP program transcribed to Kotlin; nobody actually writes like this)

Higher order functions (HOFs)

Often in the context of FP it is necessary to operate with the following functions: `map`, `filter`, and `fold`.

`map` allows us to perform a function over *each* element in a collection:

```
val list = listOf(1, 2, 3)  
list.map { it * it } // [1, 4, 9]
```

Higher order functions (HOFs)

Often in the context of FP it is necessary to operate with the following functions: `map`, `filter`, and `fold`.

`map` allows us to perform a function over *each* element in a collection.

```
val list = listOf(1, 2, 3)  
list.map { it * it } // [1, 4, 9]
```



What is the main difference between `map` and `forEach`?

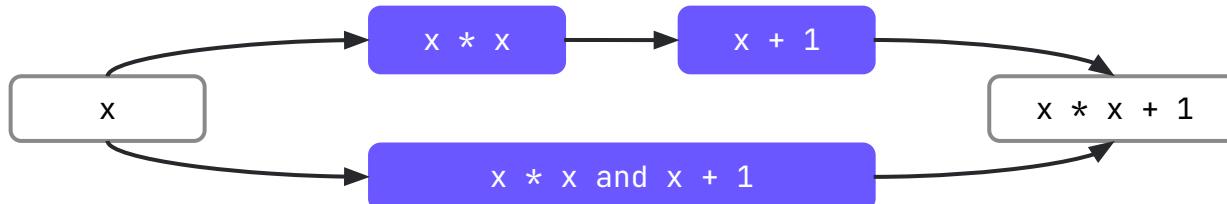
Higher order functions (HOFs)

You can **compose** the functions to perform both operations:

```
val l = listOf(1, 2, 3)
```

```
list.map { it * it }.map { it + 1 } // [2, 5, 10]
```

```
list.map { it * it + 1 } // [2, 5, 10]
```



NB: to compose complex functions by default you can use sequences, but be careful.

Higher order functions (HOFs)

`filter` returns a list containing only elements that match a given predicate:

```
val l = listOf(1, 2, 3)  
list.filter { it % 2 == 0 } // [2]
```

Higher order functions (HOFs)

Our third important function, `fold`, creates a mutable accumulator, which is updated on each round of the `for` and returns one value:

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

You can implement the `fold` function for any type, for example, you can fold a tree into a string representation.

Higher order functions (HOFs)

There are also right and left folds. They are equivalent if the operation is associative: $(a \circ b) \circ c = a \circ (b \circ c)$, but in any other case they yield different results.

```
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 + (3 + 0))) = 6

"PWD".fold("") { acc, x → "${acc}${acc}$x" }           // PPWPPWNPPWPPWND
"PWD".foldRight("") { x, acc → "${acc}${acc}$x" } // DDNDDNWDDNDNNWP
```

Make sure to: be careful with the order of your lambdas' arguments:

```
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
```

Higher order functions (HOFs)

```
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 }
    .eachCount()
    .toList()
    .sortedBy { (_, count) → count }
    .reversed()
```

Higher order functions (HOFs)

```
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())

[One, one, was, a, race, horse, , Two, two, was, one, too, , One, one, won,
one, race, , Two, two, won, one, too, , ]
```

Higher order functions (HOFs)

```
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() }

[One, one, was, a, race, horse, Two, two, was, one, too, One, one, won,
one, race, Two, two, won, one, too]
```

Higher order functions (HOFs)

```
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() }

[one, one, was, a, race, horse, two, two, was, one, too, one, one,
won, one, race, two, two, won, one, too]
```

Higher order functions (HOFs)

```
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 }
    .eachCount()
```

OR

```
string
    .split(...)
    .filter { it.isNotEmpty() }
    .groupBy({ it.lowercase() }, { it })
    .mapValues { (key, value) ->
        value.size
    }
```

{one=7, was=2, a=1, race=2, horse=1, two=4, too=2, won=2}

Higher order functions (HOFs)

```
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 }
    .eachCount()
    .toList()

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

Higher order functions (HOFs)

```
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 }
    .eachCount()
    .toList()
    .sortedBy { (_, count) → count }

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

Higher order functions (HOFs)

```
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(" ", "-", ".")
    .filter { it.isNotEmpty() }
    .map { it.lowercase() }
    .groupingBy { it }
    .eachCount()
    .toList()
    .sortedBy { (_, count) -> count }
    .reversed()

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

OR

```
string
    .allFunnyFuncs(...)
    .toList()
    .sortedWith { l, r ->
        r.second - l.second
    }
```

OR

```
string
    .allFunnyFuncs(...)
    .toList()
    .sortedByDescending { (_, c) ->
        c
    }
```

Higher order functions (HOFs)

Lambdas are not the only functions that can be passed as arguments to functions expecting other functions, as *references* to already defined functions can be as well:

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

```
val isEvenLambda = { x: Int → x % 2 = 0 }
```

Same results, different calls:

- `list.partition { it % 2 = 0 }`
- `list.partition(::isEven) // function reference`
- `list.partition(isEvenLambda) // pass lambda by name`

Lazy computations

Consider the following code:

```
fun <F> withFunction(  
    number: Int, even: F, odd: F  
): F = when (number % 2) {  
    0 → even  
    else → odd  
}  
  
withFunction(4, println("even"), println("odd"))
```

What will be printed to the console?

Lazy computations

Consider the following code:

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

```
withFunction(4, println("even"),  
println("odd"))
```

Arguments of the `withFunction` function will be evaluated **before** its body is executed (eager execution).

What will be printed into console? even odd

~~Lazy~~ Deferred computations

Consider the following code:

```
fun <F> withLambda(  
    number: Int, even: () -> F, odd: () -> F  
) : F = when (number % 2) {  
    0 -> even()  
    else -> odd()  
}  
  
withLambda(4, { println("even") }, { println("odd") })
```

It will print just even into the console because of the ~~lazy~~ deferred computations.

Operator overloading

Kotlin has extension functions that you can use to override operators, for example the `iterator`. That is, you do not need to create a new entity that inherits from the `Iterable` interface, as you would in OOP code.

```
class MyIterable<T> : Iterable<T> { // you need access to the sources of MyIterable
    override fun iterator(): Iterator<T> {
        TODO("Not yet implemented")
    }
}
```

VS

```
class A<T>
operator fun <T> A<T>.iterator(): Iterator<T> = TODO("Not yet implemented")
```

One last thing...

Is this code correct?

```
enum class Color {  
    WHITE,  
    AZURE,  
    HONEYDEW  
}  
  
fun Color.getRGB() = when (this) {  
    Color.WHITE → "#FFFFFF"  
    Color.AZURE → "#F0FFFF"  
    Color.HONEYDEW → "F0FFF0"  
}
```

One last thing...

Is this code correct? **Yes**, because the compiler knows **all** of the possible values.

```
enum class Color {  
    WHITE,  
    AZURE,  
    HONEYDEW  
}  
  
fun Color.getRGB() = when (this) {  
    Color.WHITE → "#FFFFFF"  
    Color.AZURE → "#F0FFFF"  
    Color.HONEYDEW → "F0FFF0"  
}
```

One last thing...

What is about this example?

```
sealed class Color

class WhiteColor: Color()
class AzureColor: Color()
class HoneydewColor: Color()

fun Color.getRGB() = when (this) {
    is WhiteColor → "#FFFFFF"
    is AzureColor → "#F0FFFF"
    is HoneydewColor → "F0FFF0"
}
```

One last thing...

What about this example? Once again, the answer is **yes**, because the compiler knows about **all** possible children of the `Color` class at the compilation stage and no new classes can appear.

```
sealed class Color

class WhiteColor: Color()
class AzureColor: Color()
class HoneydewColor: Color()

fun Color.getRGB() = when (this) {
    is WhiteColor → "#FFFFFF"
    is AzureColor → "#F0FFFF"
    is HoneydewColor → "F0FFF0"
}
```

One last thing...

Consider the following code:

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

We have the common part in **all** classes and we know that these are the **only** possible subclasses.
Let's move this code into the base class.

One last thing...

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



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

Actually, we have **equivalent** classes, i.e. *each* function for the first version can be rewritten as the *second* one.

One last thing...

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



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

```
fun Color.getUserRGB() = when (this) {  
    is WhiteColor → "${this.name}: #FFFFFF"  
    is AzureColor → "${this.name}: #F0FFFF"  
    is HoneydewColor → "${this.name}: F0FFF0"  
}
```

```
fun NewColor.getUserRGB() = when (this) {  
    is WhiteColor → "${this.name}: #FFFFFF"  
    is AzureColor → "${this.name}: #F0FFFF"  
    is HoneydewColor → "${this.name}: F0FFF0"  
}
```

In the first function, we have smart casts, but in the second one we don't have them.

One last thing...

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



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

```
fun Color.getUserRGB() = when (this) {  
    is WhiteColor → "${this.name}: #FFFFFF"  
    is AzureColor → "${this.name}: #F0FFFF"  
    is HoneydewColor → "${this.name}: F0FFF0"  
}
```

```
fun NewColor.getUserRGB() = when (this) {  
    is WhiteColor → "${this.name}: #FFFFFF"  
    is AzureColor → "${this.name}: #F0FFFF"  
    is HoneydewColor → "${this.name}: F0FFF0"  
}
```

Math time! We can actually rewrite this in math terms:

$$\text{WhiteColor} * \text{String} + \dots + \text{HoneydewColor} * \text{String} = \text{String} * (\text{WhiteColor} + \dots + \text{HoneydewColor})$$

One last thing...

Math time! We can actually rewrite this in math terms:

$$\text{WhiteColor} * \text{String} + \dots + \text{HoneydewColor} * \text{String} \simeq \text{String} * (\text{WhiteColor} + \dots + \text{HoneydewColor})$$

This is possible because we are actually operating with **algebraic data types*** and can use their properties.

**This is not entirely true, but for most cases with sealed classes it works.*

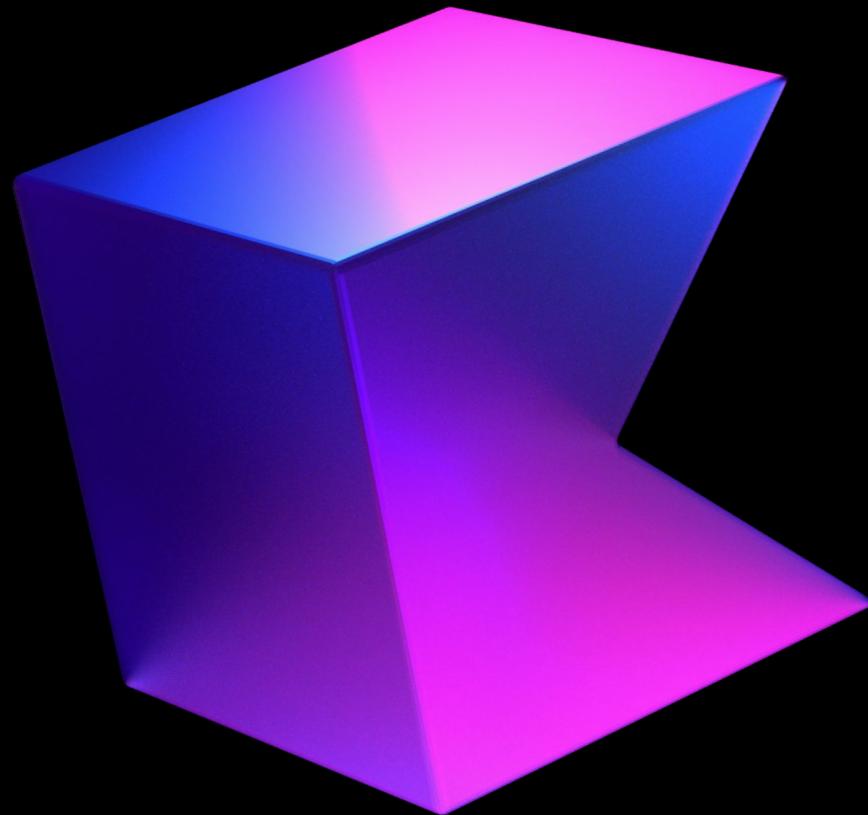
Final thought

FP in Kotlin does not kill OOP. Each of the concepts brings its own advantages and disadvantages, and it is important to combine them in order to get concise, readable and understandable code!

If you are interested in the topic of FP in Kotlin for a more detailed study, come here:

<https://arrow-kt.io/>

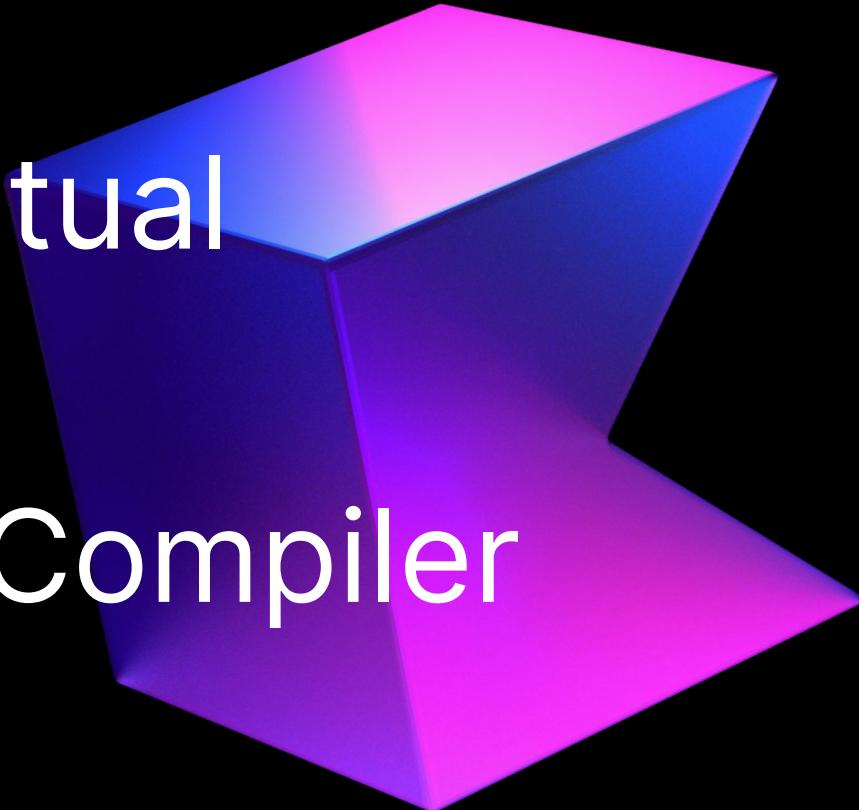
Thanks!



@kotlin



The Java Virtual Machine & the Kotlin Compiler



The Java language

- Was created in 1995.
- Is an OOP language with strong static typing.
- Has Just-in-time (JIT) compilation.
- Uses the Java Virtual Machine (JVM).
- Has a garbage collector, meaning you can allocate memory and it will be freed automatically.

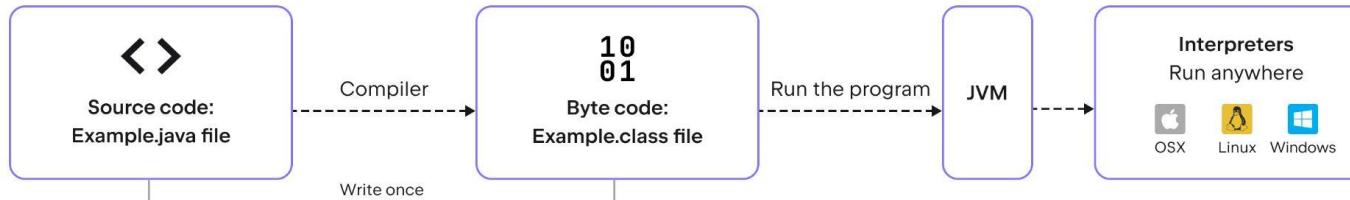
Compilation process – Java vs C

C compilation process



VS

Java compilation process



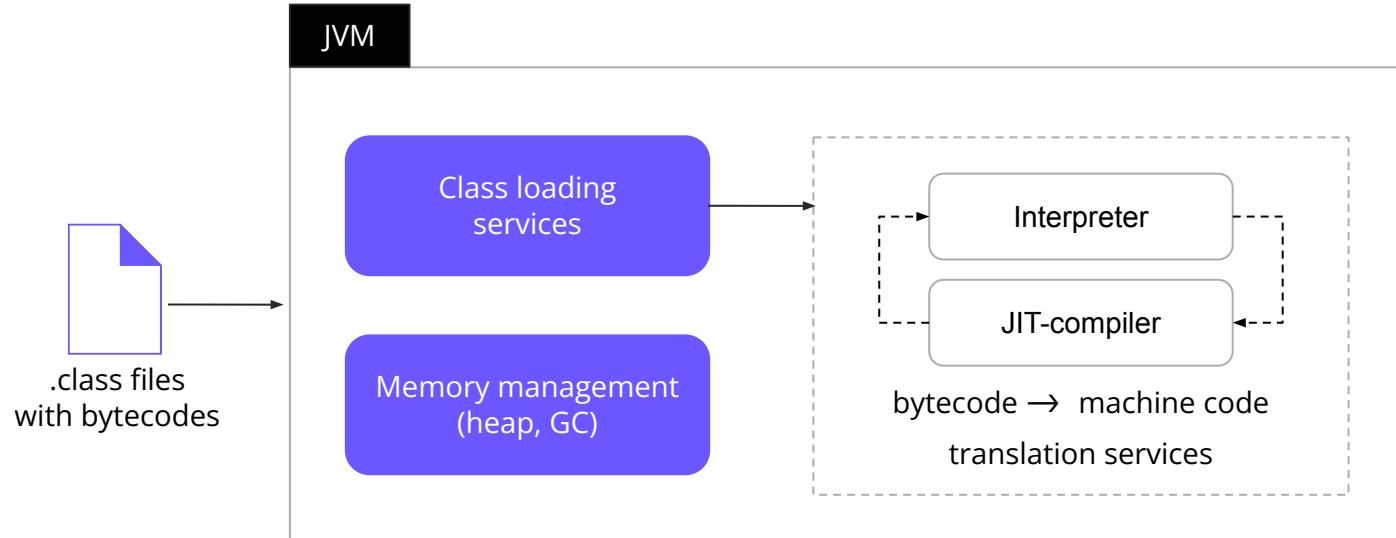
Java bytecode

```
public class Main {  
    public static void main(String[] args) {  
        System.out.print("Hello, World!");  
    }  
}
```

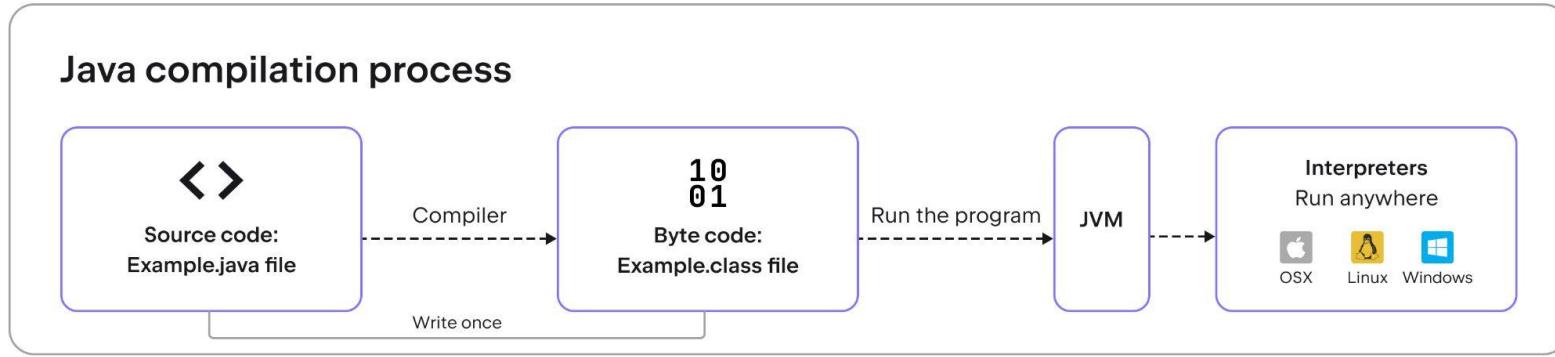


```
public class examples/Main {  
  
    public <init>()V  
        L0  
        LINENUMBER 3 L0  
        ALOAD 0  
        INVOKESPECIAL java/lang/Object.<init> ()V  
        RETURN  
    L1  
        LOCALVARIABLE this Lorg/examples/Main; L0 L1 0  
        MAXSTACK = 1  
        MAXLOCALS = 1  
  
    public static main([Ljava/lang/String;)V  
        L0  
        LINENUMBER 5 L0  
        GETSTATIC java/lang/System.out : Ljava/io/PrintStream;  
        LDC "Hello, World!"  
        ICONST_0  
        ANEWARRAY java/lang/Object  
        INVOKEVIRTUAL java/io/PrintStream.print  
            (Ljava/lang/String;[Ljava/lang/Object;)Ljava/io/PrintStream;  
        POP  
    L1  
        LINENUMBER 6 L1  
        RETURN  
    L2  
        LOCALVARIABLE args [Ljava/lang/String; L0 L2 0  
        MAXSTACK = 3  
        MAXLOCALS = 1  
    }  
}
```

The JVM under the hood



Just-in-time compilation



- Program profiling occurs at runtime.
- Pieces of code are compiled for a specific platform to optimize the execution time.

Interpreting a command is much slower than executing it directly on the processor.



Why, then, do we need the interpreter?

Just-in-time compilation



Why, then, do we need the interpreter?

Interpreter

- Starts working almost instantly.
- The performance of the executable code is poor.

VS

JIT-compiler

- Kicks in after a long delay (needs time for optimizations).
- The performance of the executable (compiled) code is high.

Just-in-time compilation



What sorts of JIT code are worth compiling?

Code that will take a long time to run or code that runs frequently, because the compilation overhead will be covered by the profit from having optimized execution.

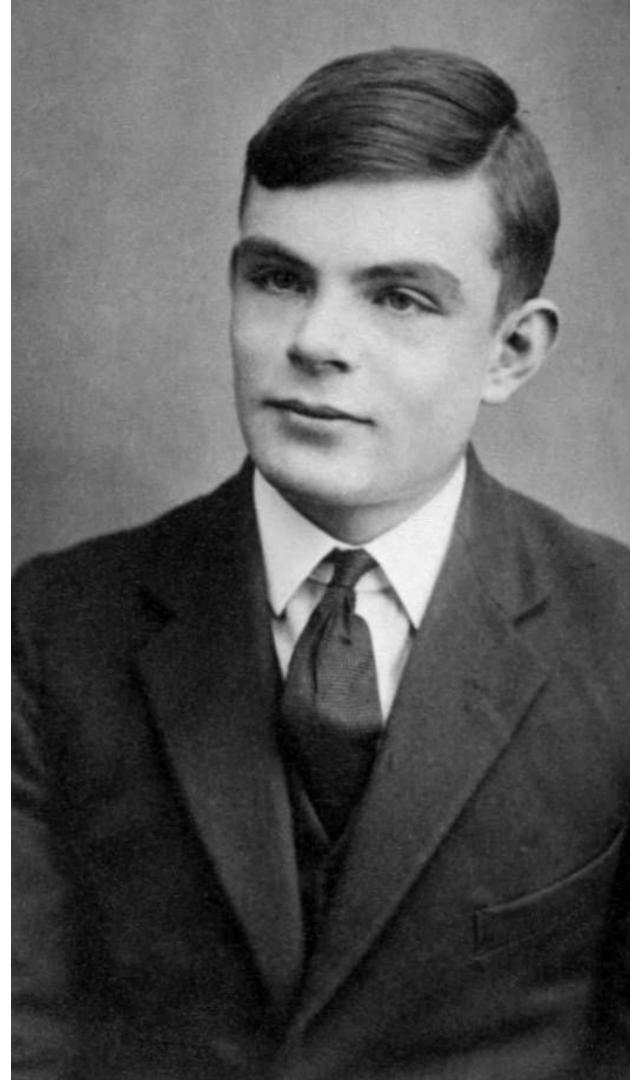
Just-in-time compilation



How can we understand which pieces of code will take a long time to execute?

Some guy named Alan Turing said it is **impossible**.

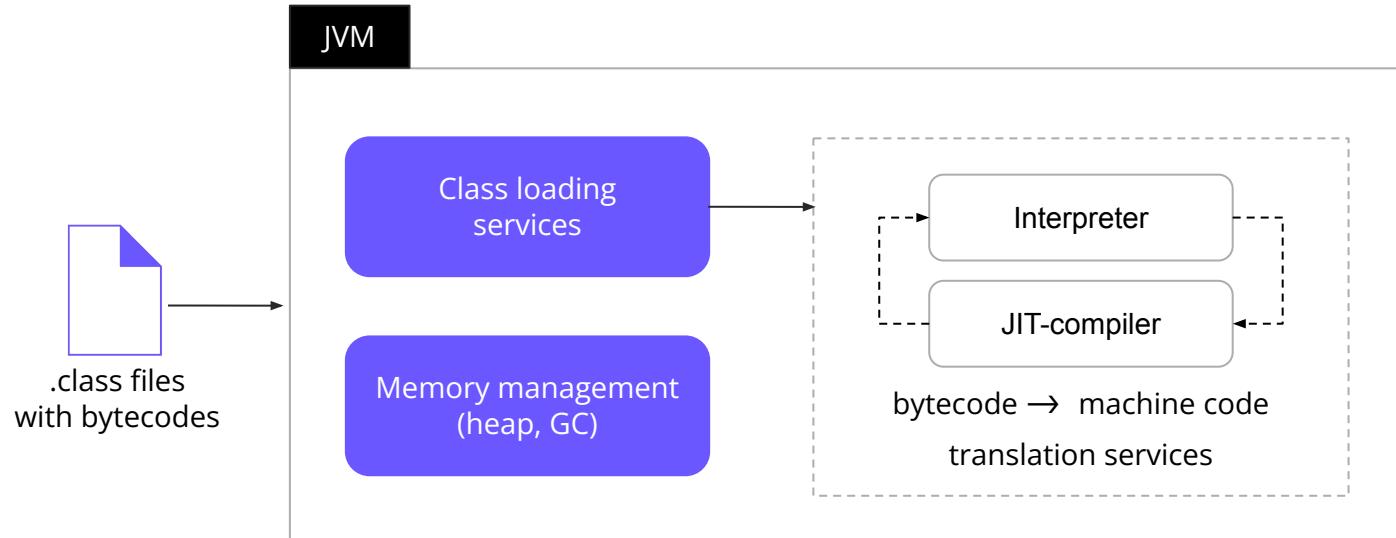
But empirically it is possible. If a piece of code took a long enough time to execute once, then most likely the same will be true in the future.



The JVM under the hood



Why does the compiler need to access the interpreter?



The JVM under the hood



Why does the compiler need to access the interpreter?

```
class PiUtils {  
    private static final double PI = 3.141592653589;  
  
    public static double getPiSquared() {  
        return PI * PI;  
    }  
}
```



```
public static double getPiSquared() {  
    return 9.869604401084375;  
}
```

The JVM under the hood



Why does the compiler need to access the interpreter?

```
class PiUtils {  
    private static final double PI = 4;  
  
    public static double getPiSquared() {  
        return PI * PI;  
    }  
}
```

↓ Reflection

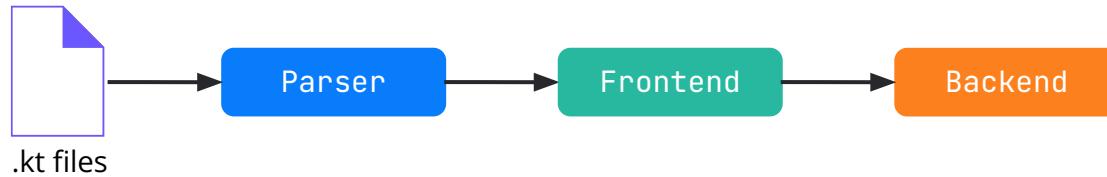
```
public static double getPiSquared() {  
    return 9.869604401084375;  
}
```



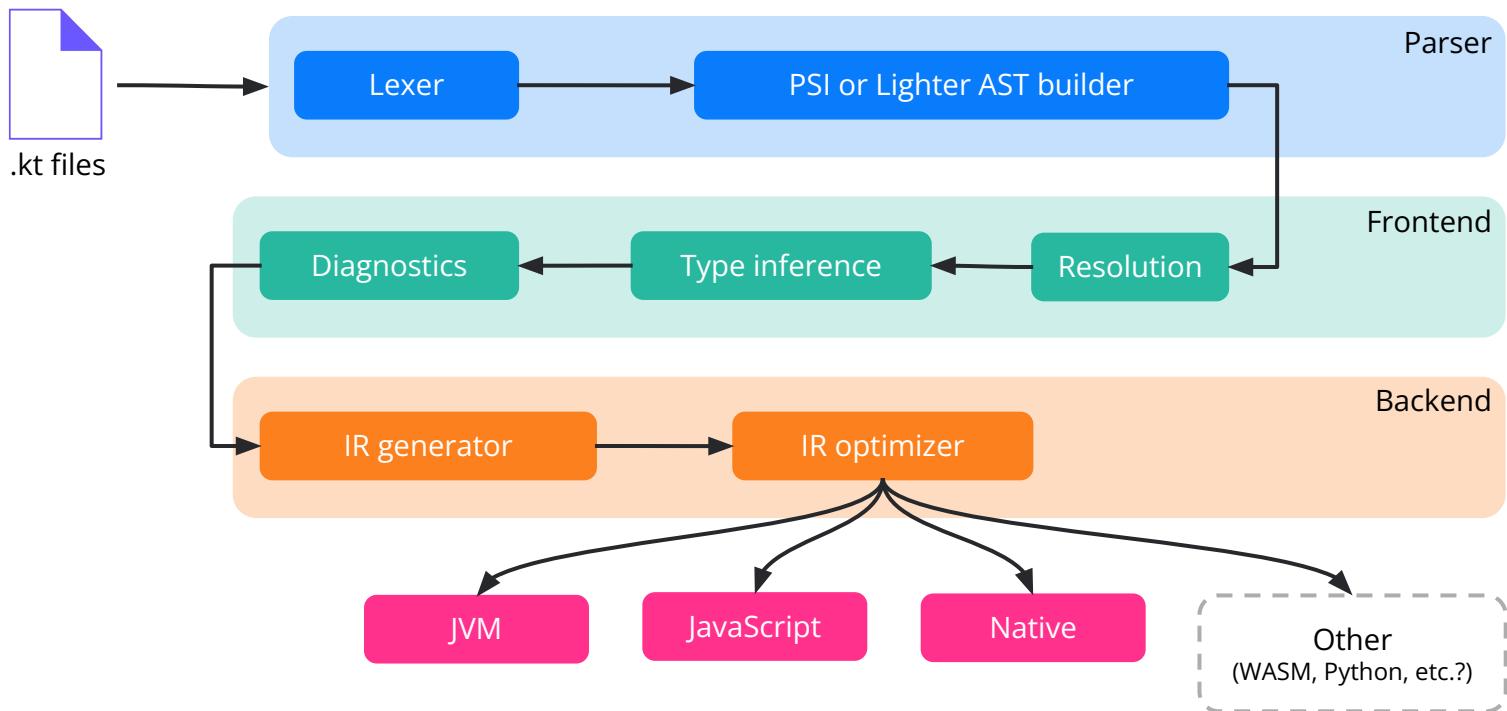
The Kotlin language

- Can be compiled into JVM bytecode.
- Is interoperable with Java.
 - Can work in Java projects.
 - Can use Java libraries.
- Is safe and concise.
- Provides the ability to integrate into the compilation process (compiler plugins).
- Is the main development language for Android applications, according to Google.

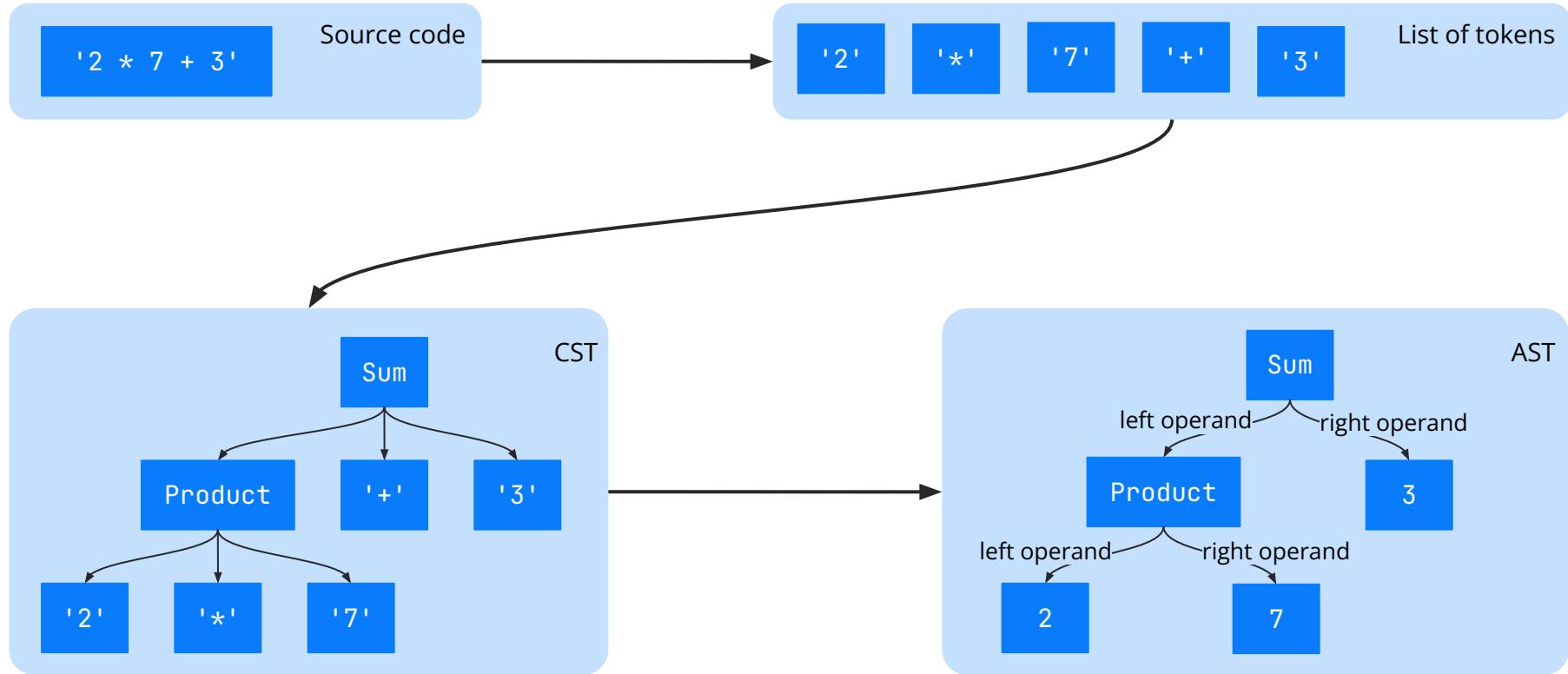
Kotlin compiler



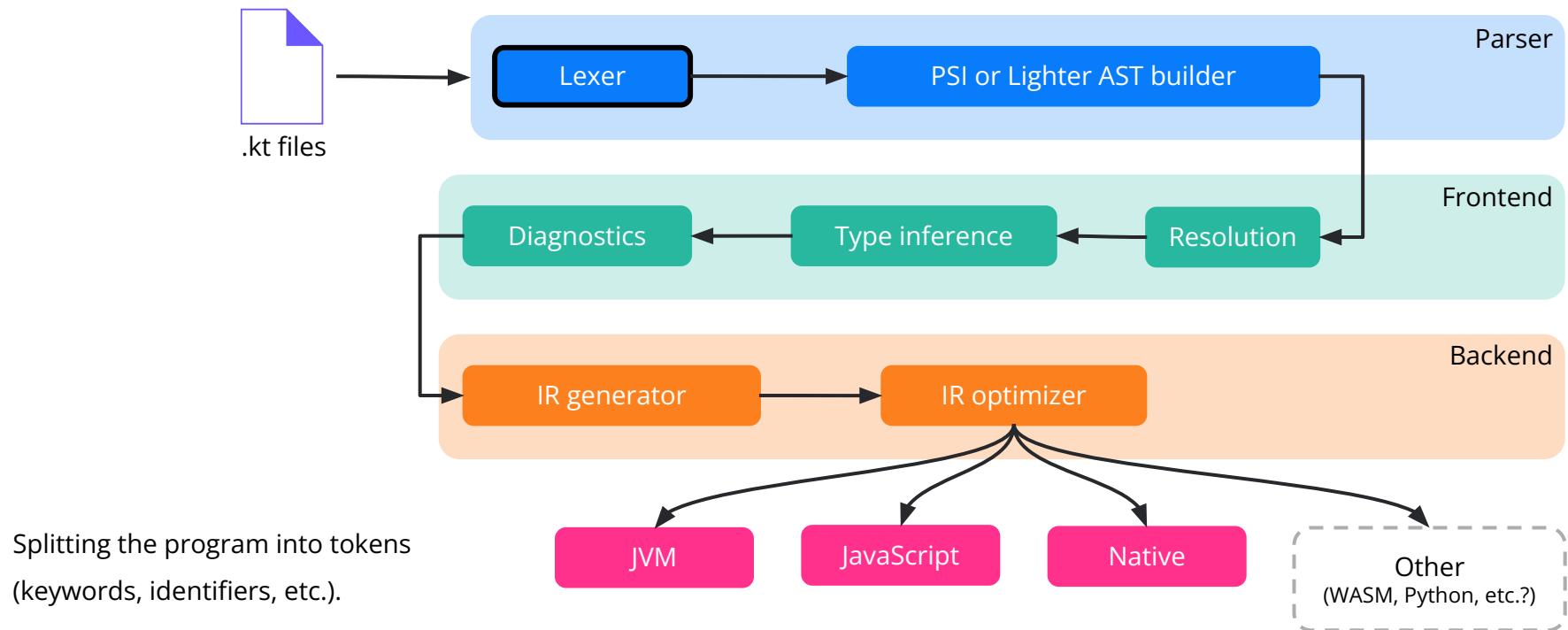
Kotlin compiler



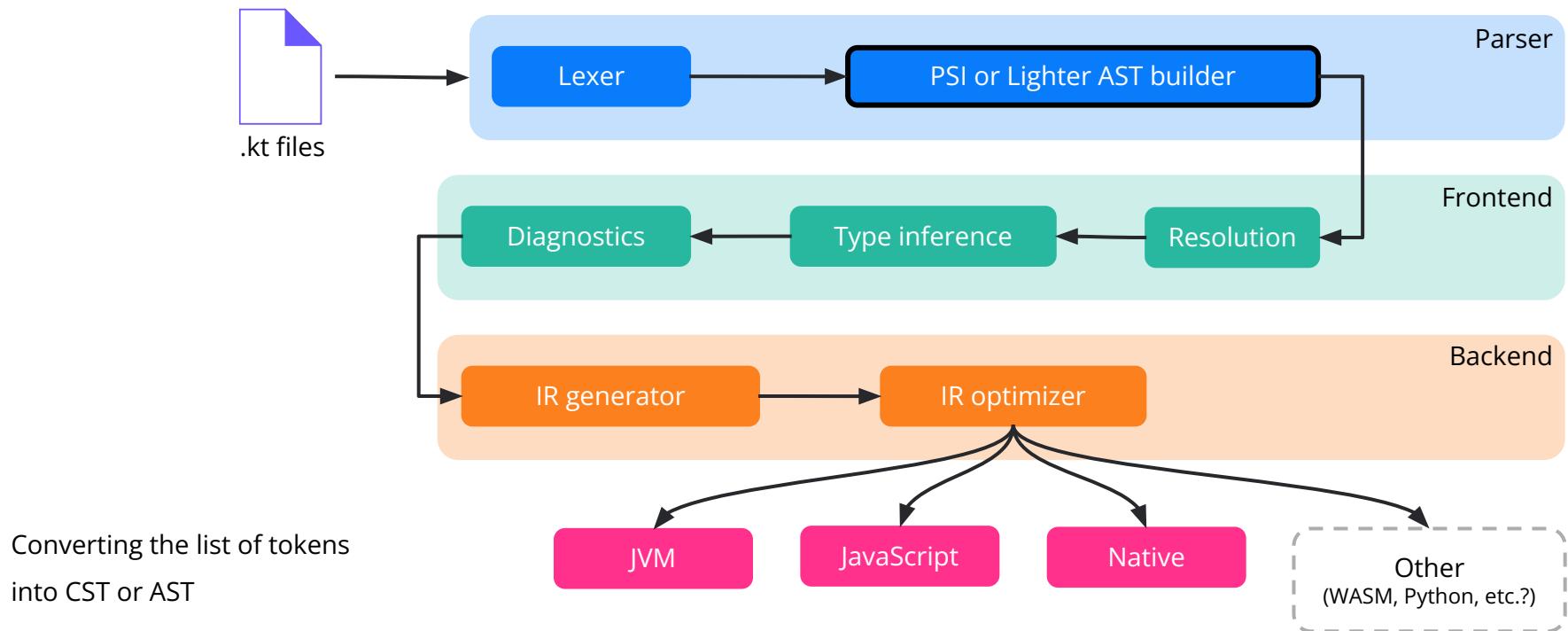
Parsing in a nutshell



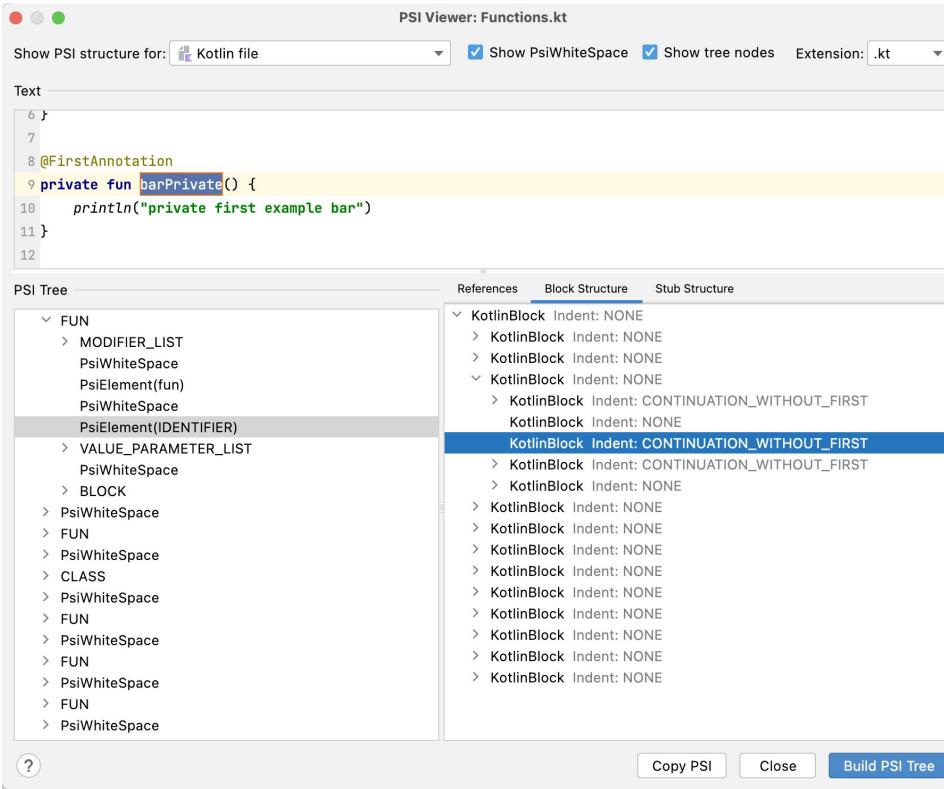
Kotlin compiler



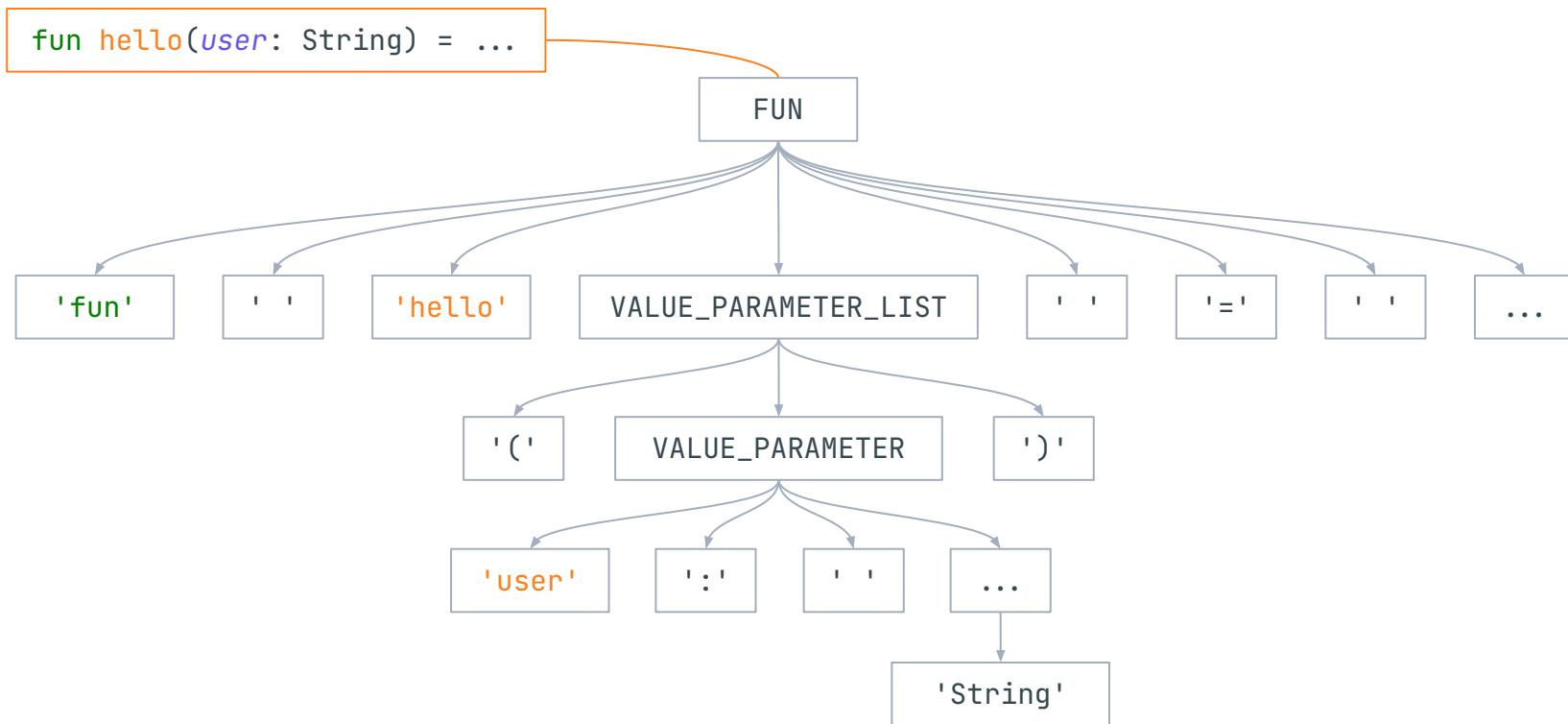
Kotlin compiler



Kotlin compiler: PSI

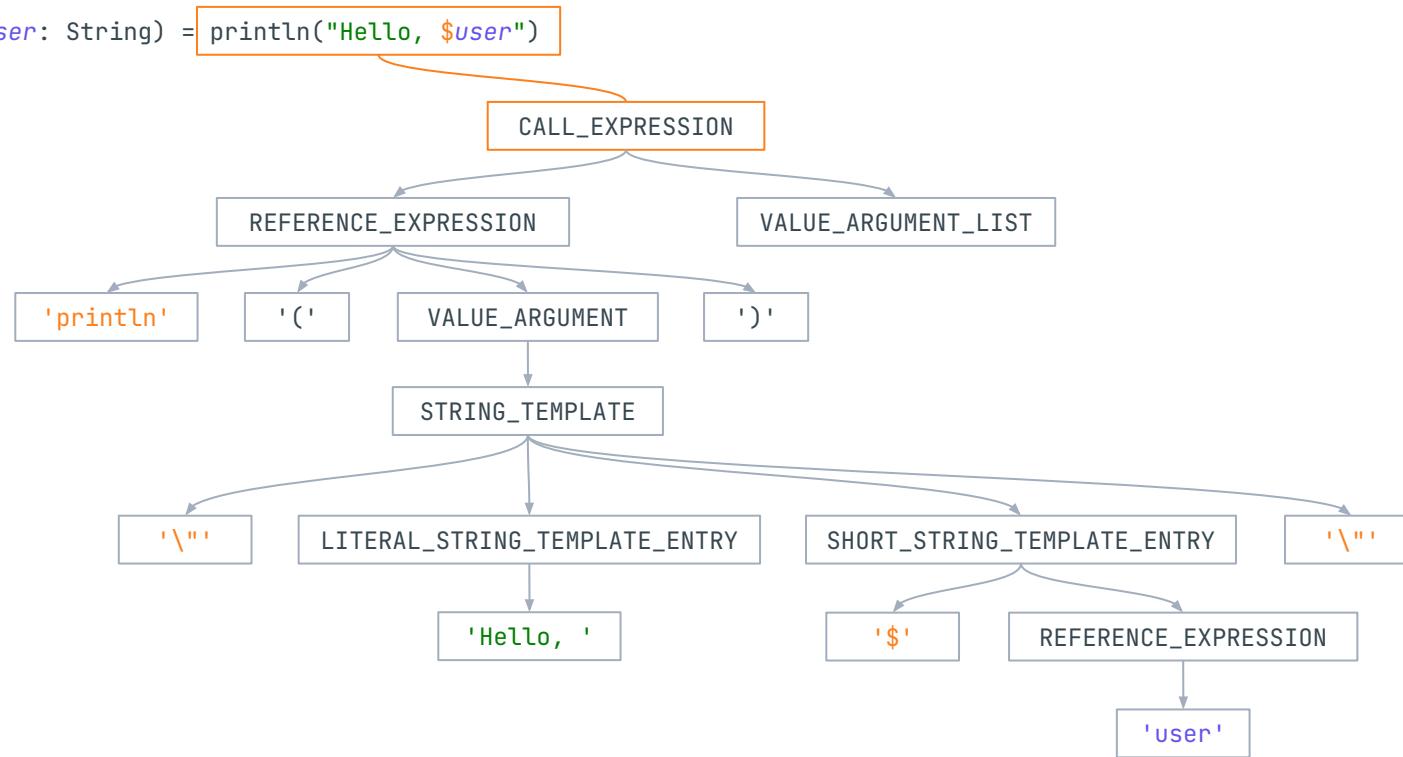


Kotlin compiler: PSI



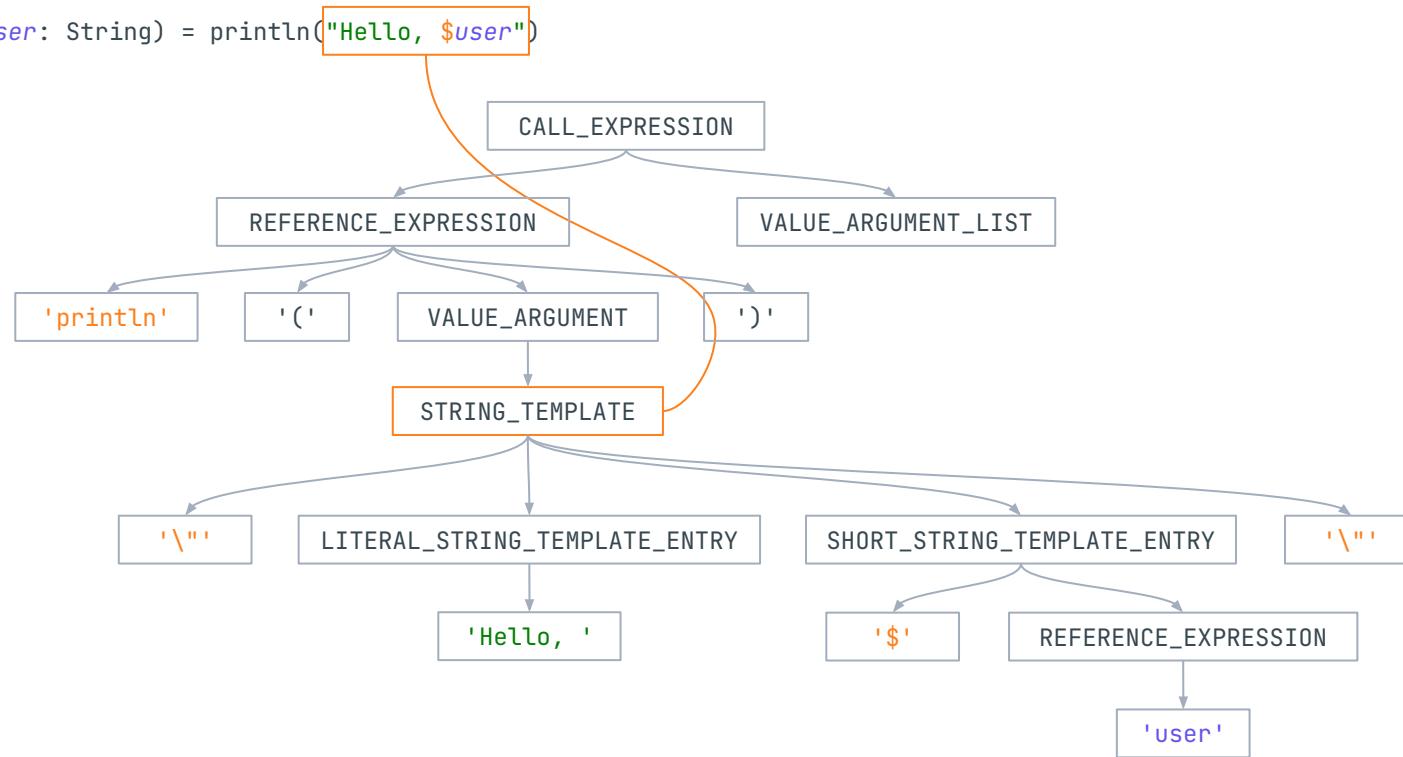
Kotlin compiler: PSI

```
fun hello(user: String) = println("Hello, $user")
```



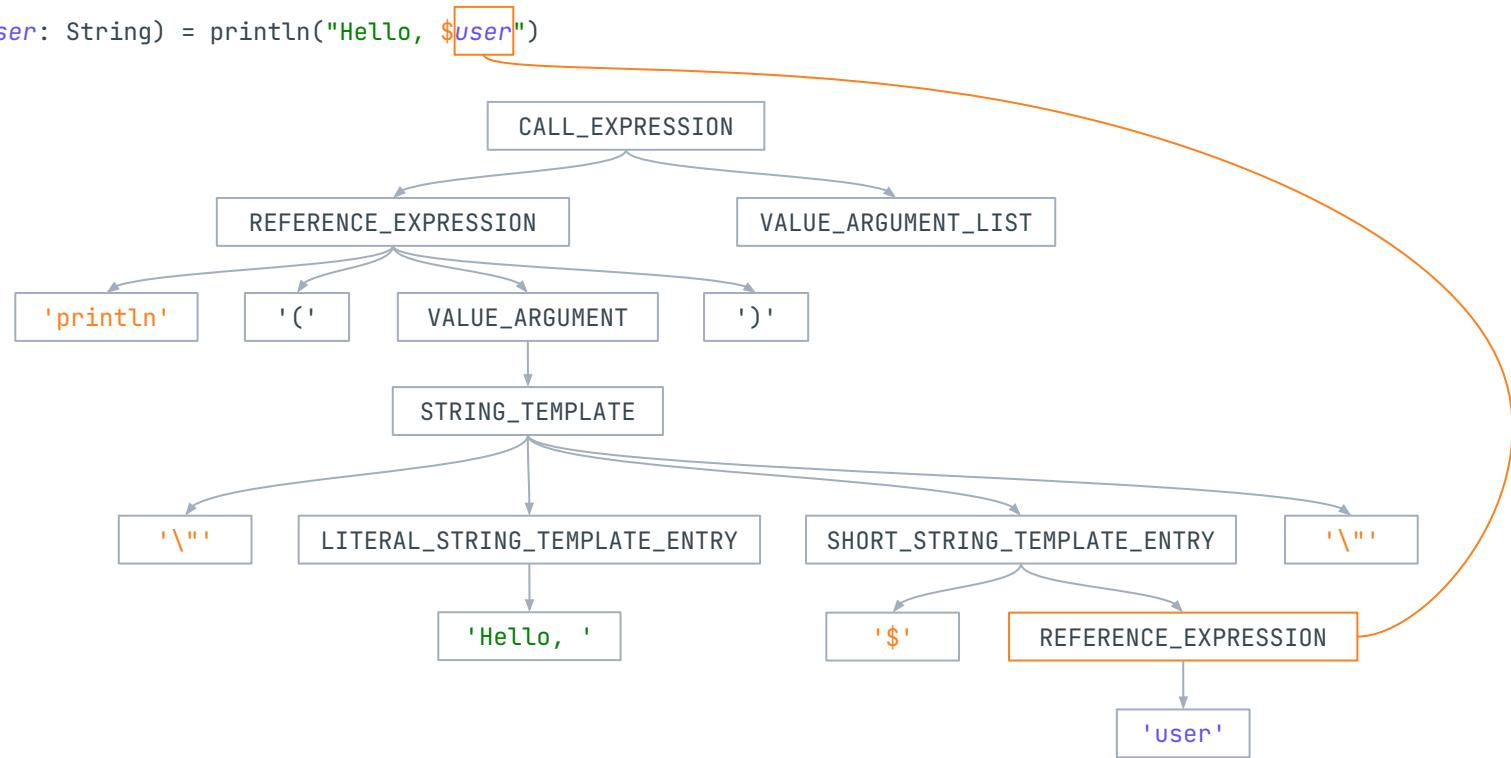
Kotlin compiler: PSI

```
fun hello(user: String) = println("Hello, $user")
```

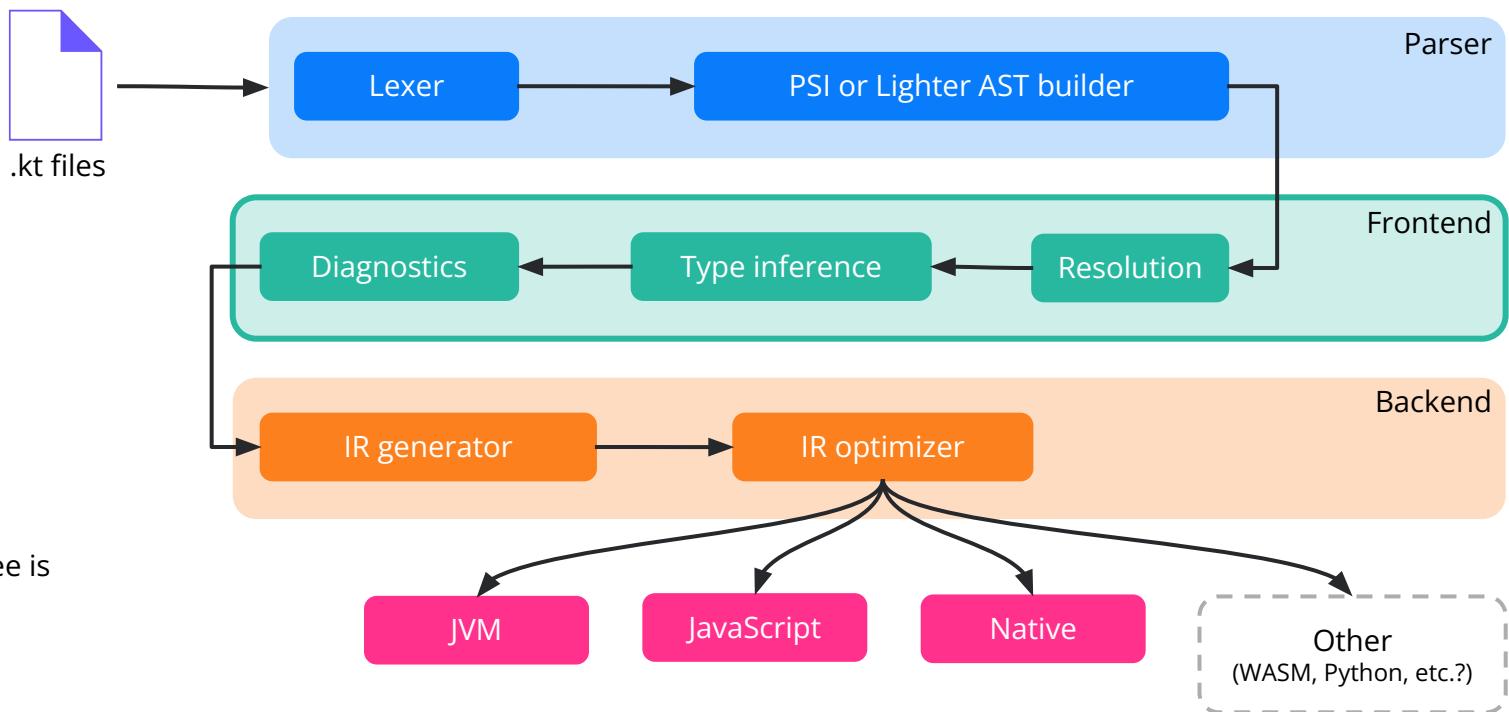


Kotlin compiler: PSI

```
fun hello(user: String) = println("Hello, $user")
```

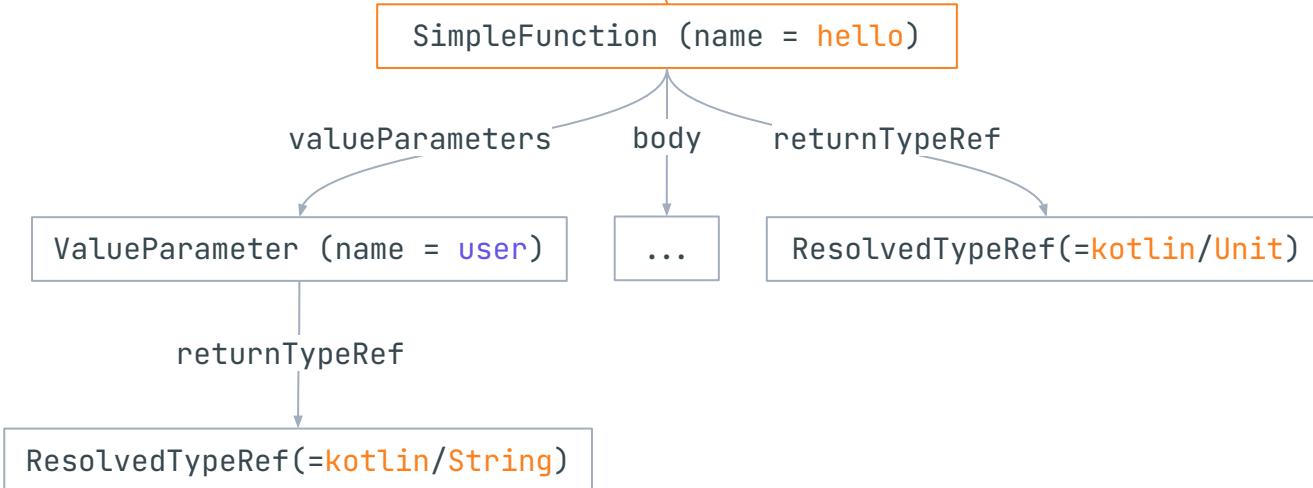


Kotlin compiler

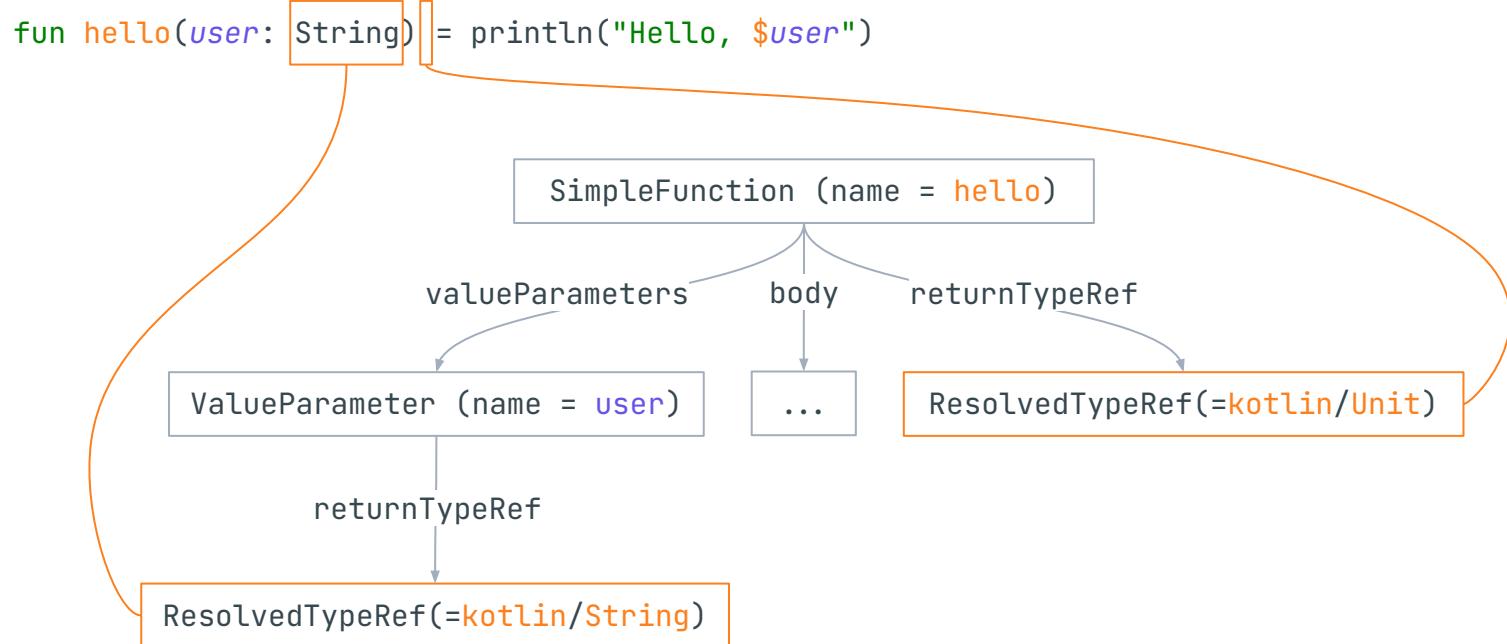


FIR? Another tree!

```
fun hello(user: String) = println("Hello, $user")
```

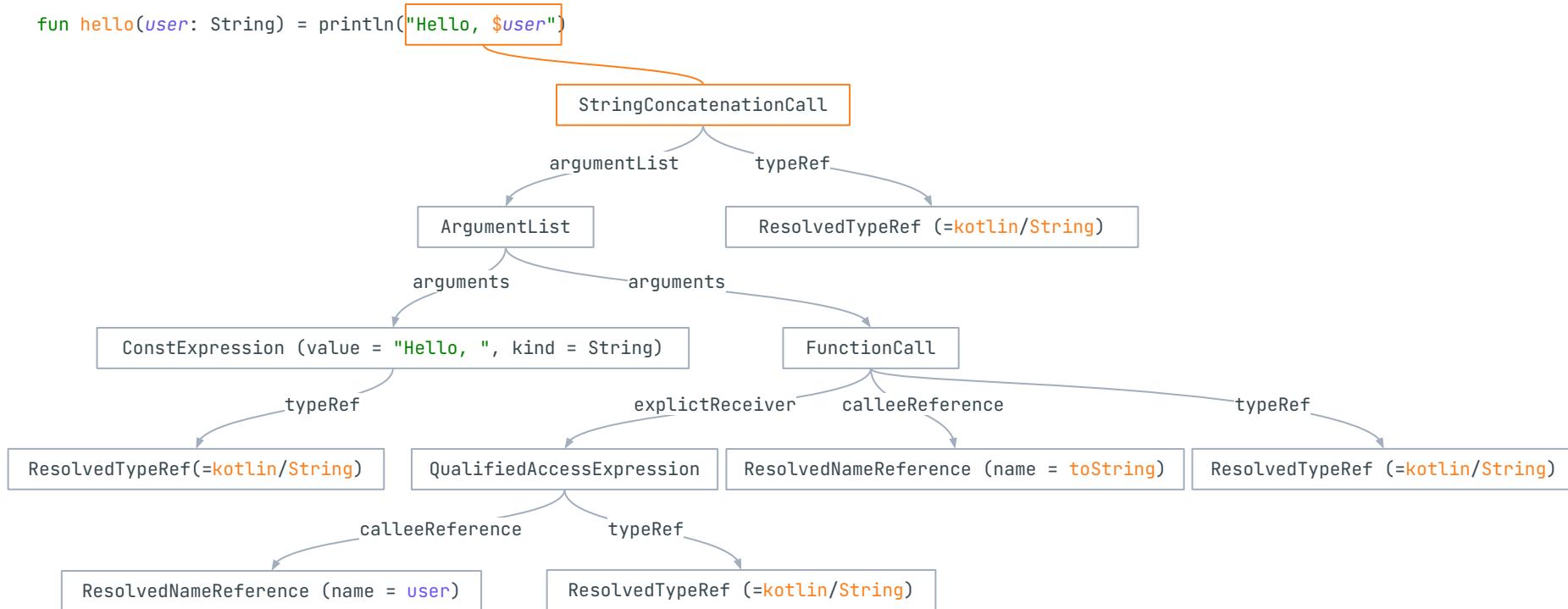


FIR? Another tree!



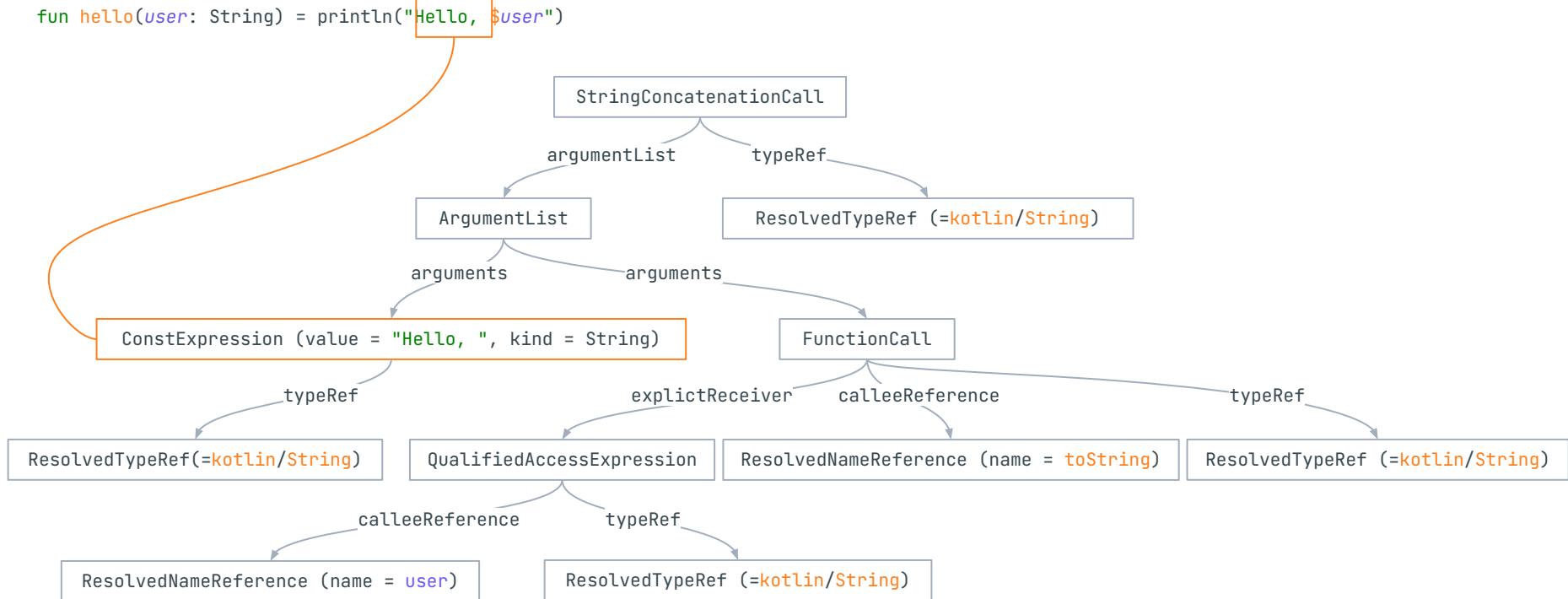
FIR? Another tree!

```
fun hello(user: String) = println("Hello, $user")
```



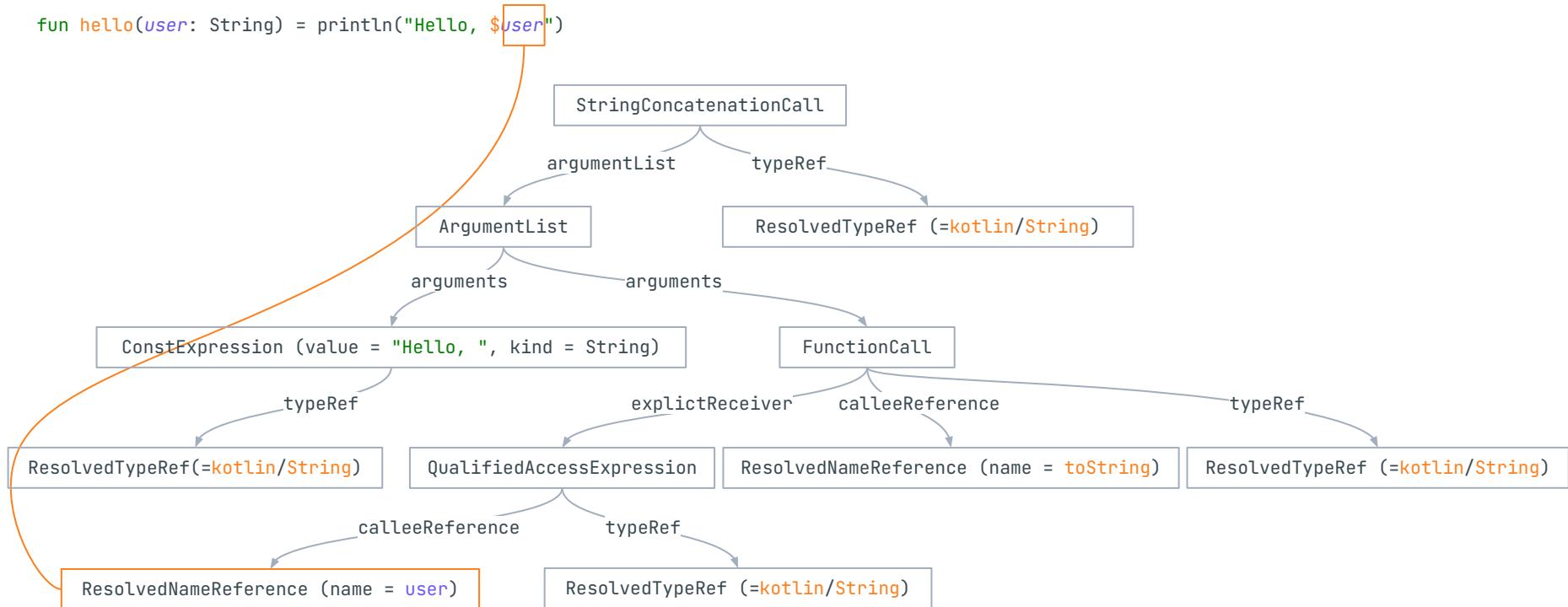
FIR? Another tree!

```
fun hello(user: String) = println("Hello, $user")
```



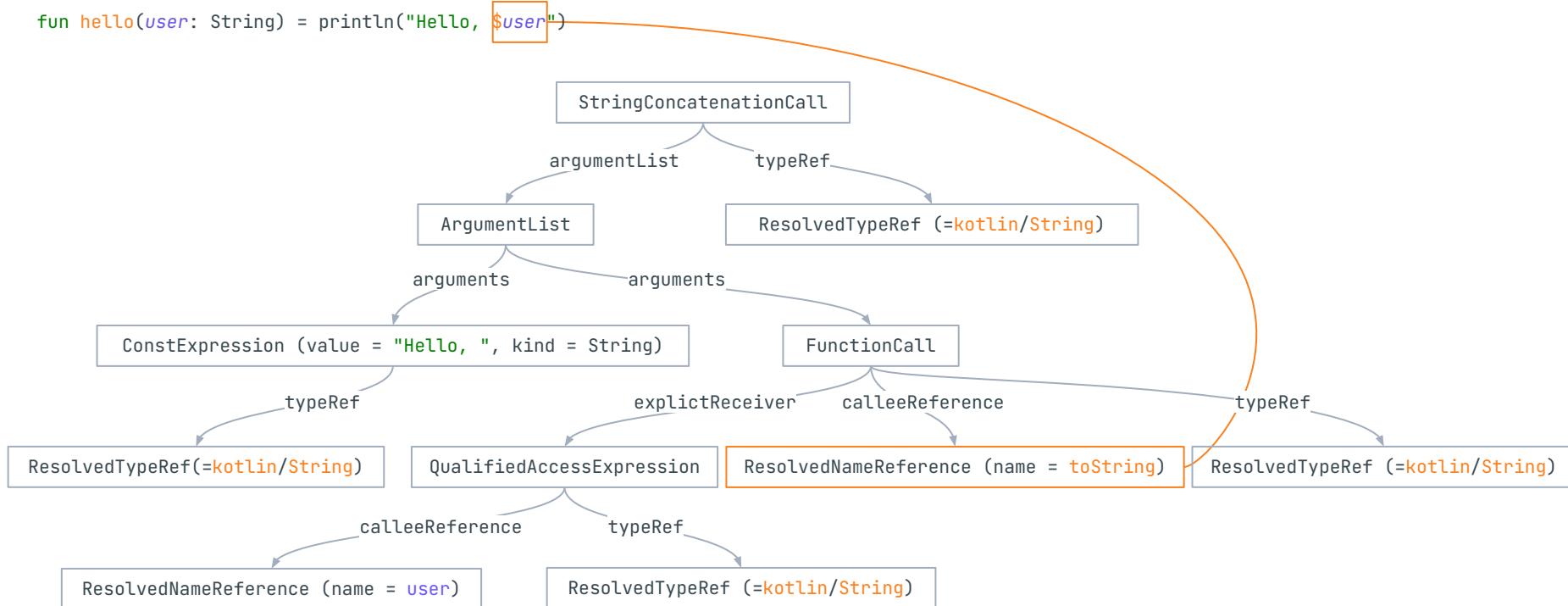
FIR? Another tree!

```
fun hello(user: String) = println("Hello, $user")
```



FIR? Another tree!

```
fun hello(user: String) = println("Hello, $user")
```



FIR: desugaring

```
if (b) {  
    println("Hello")  
}
```



```
when {  
    b → println("Hello")  
}
```

```
for (s in list) {  
    println(s)  
}
```



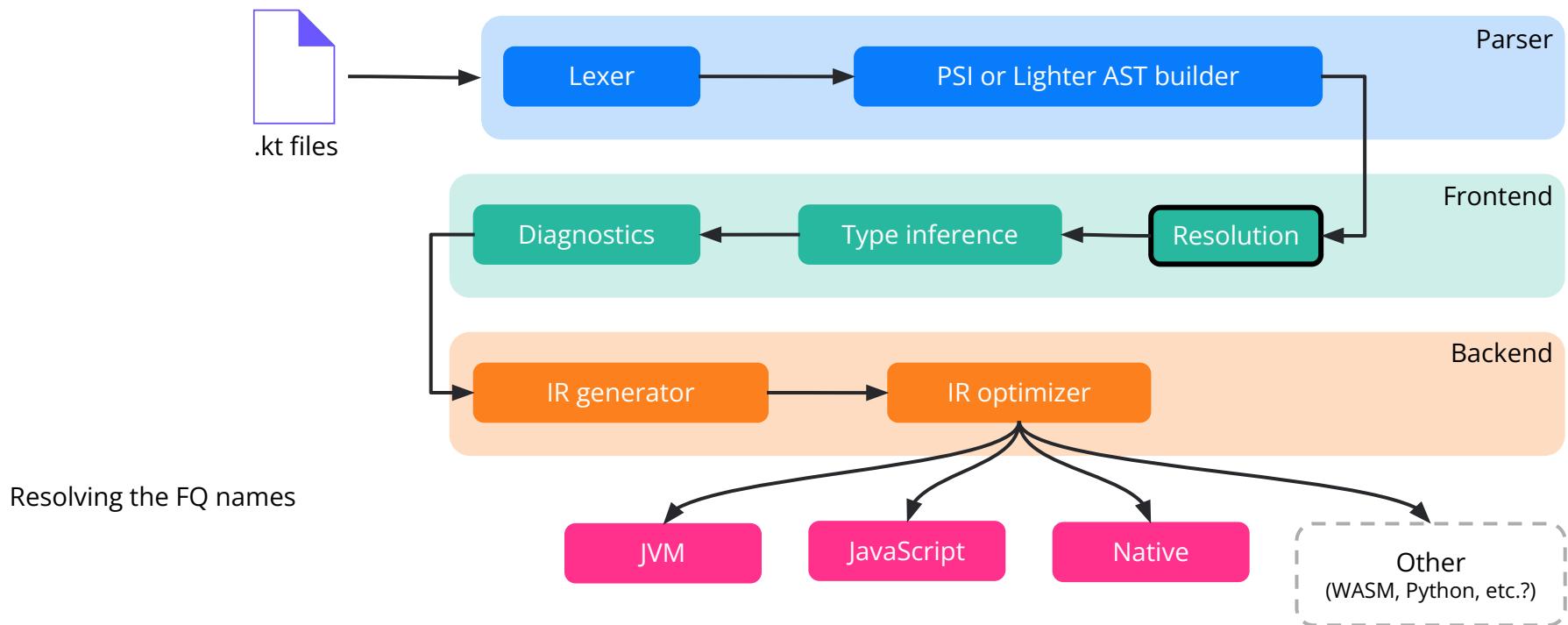
```
val <iterator> = list.iterator()  
while (<iterator>.hasNext()) {  
    val s = <iterator>.next()  
    println(s)  
}
```

```
val (a, b) = "a" to "b"
```



```
val <destruct> = "a" to "b"  
val a = <destruct>.component1()  
val b = <destruct>.component2()
```

Kotlin compiler



Kotlin compiler: resolve

```
fun myFunction() {  
}
```

Library A

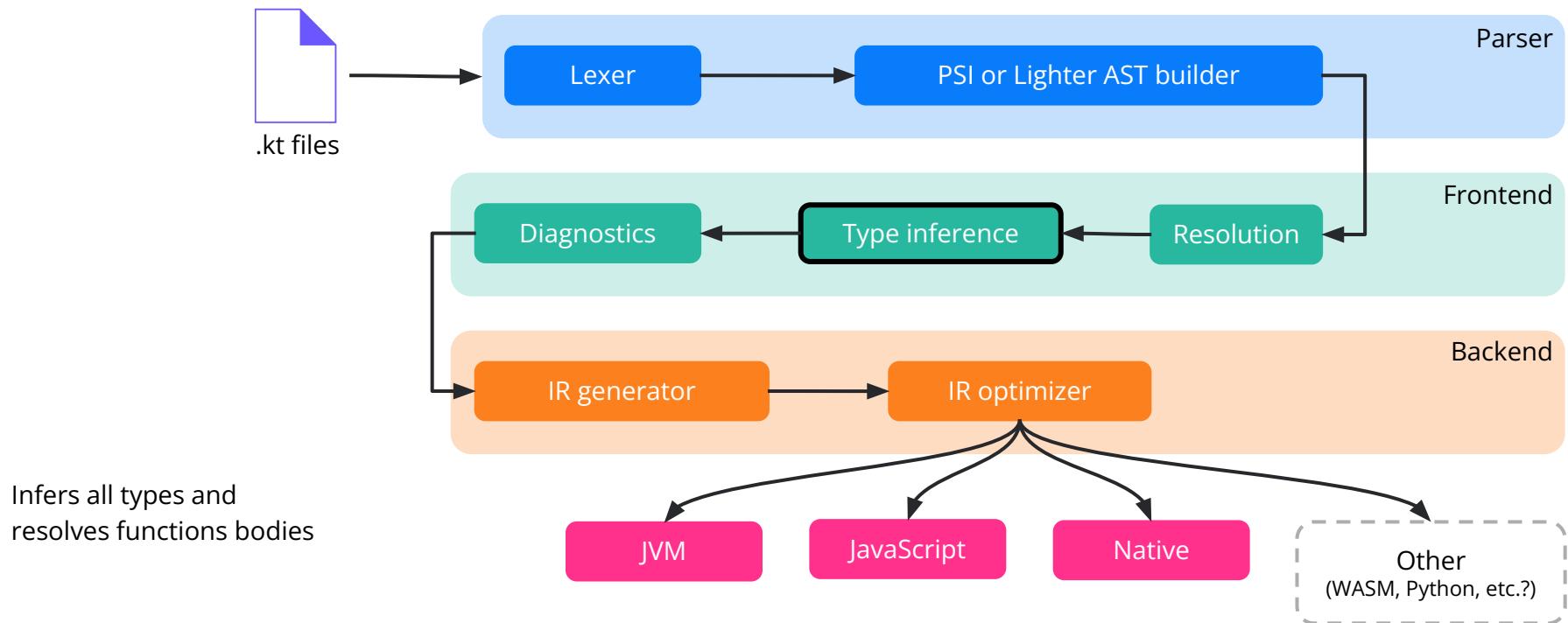
Short FQ name: myFunction
Resolved FQ name: org.libraryA.myFunction

```
fun myFunction() {  
}
```

Library B

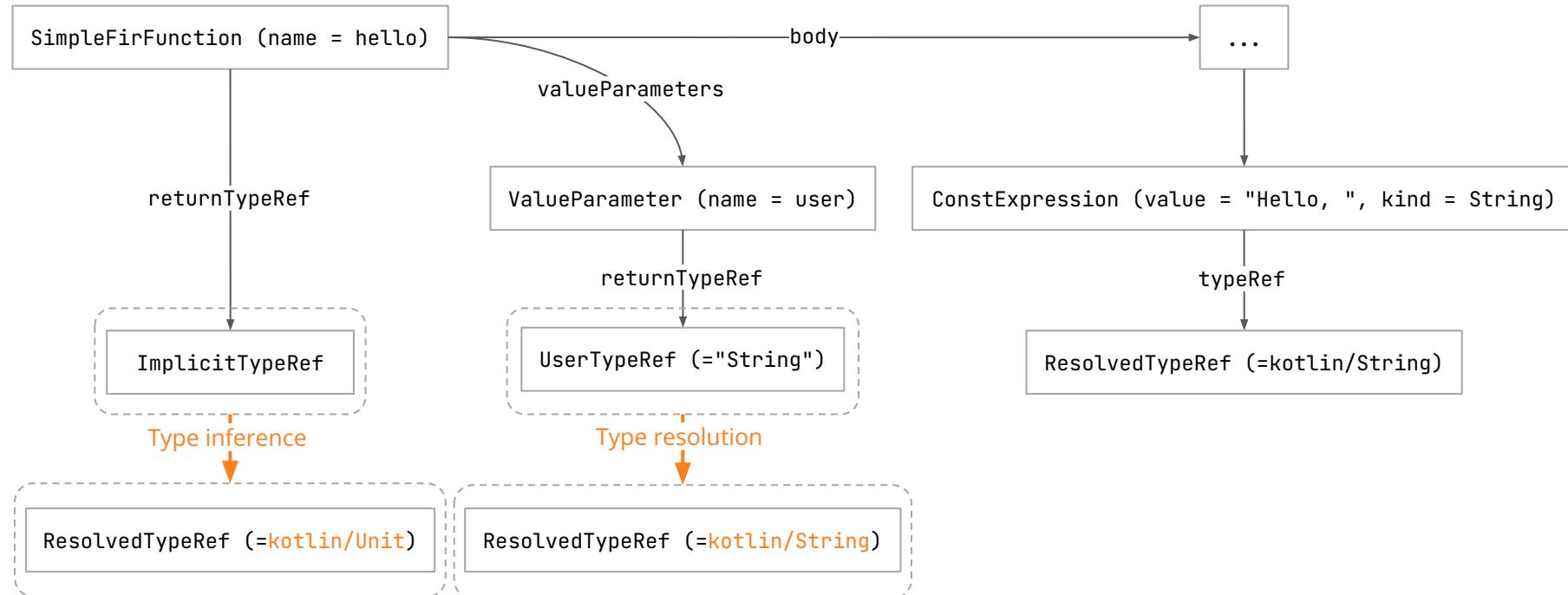
Short FQ name: myFunction
Resolved FQ name: org.libraryB.myFunction

Kotlin compiler



Kotlin compiler

```
fun hello(user: String) = println("Hello, $user")
```



Java interoperability: nullability

- Java nullable types in Kotlin

Java sources

```
public class Main {  
    public static String foo() {  
        // TODO  
    }  
}
```



Kotlin sources

```
var a: ??? = foo()
```

Java interoperability: nullability

- Java nullable types in Kotlin

Java sources

```
public class Main {  
    public static String foo() {  
        // TODO  
    }  
}
```



Kotlin sources

```
var a: ??? = foo()  
String!
```

Java interoperability: nullability

- Java nullable types in Kotlin
- `String!` is the type range: `[String .. String?]`

Java sources

```
public class Main {  
    public static String foo() {  
        // TODO  
    }  
}
```



Kotlin sources

```
var a: ??? = foo()  
  
String!
```

Java interoperability: nullability

- Java nullable types in Kotlin
- Nullability annotations `@NotNull` and `@Nullable`

Java sources

```
public class Main {  
    @NotNull  
    public static String foo() {  
        // TODO  
    }  
}
```



Kotlin sources

```
var a: String = foo()
```

Java interoperability: collection mapping

- Java collection types in Kotlin

Java sources

```
public class Main {  
    @NotNull  
    public static List<@NotNull String> foo() {  
        // TODO  
    }  
}
```



Kotlin sources

```
var a: ??? = foo()
```

Java interoperability: collection mapping

- Java collection types in Kotlin
- (Mutable)List<T> is the type range: [MutableList<T> .. List<T>]

Java sources

```
public class Main {  
    @NotNull  
    public static List<@NotNull String> foo() {  
        // TODO  
    }  
}
```



Kotlin sources

```
var a: ??? = foo()  
(Mutable)List<String>
```

Kotlin compiler: control- and data-flow analysis

- Variable initialization analysis
- Return analysis
- Smart cast analysis

Each variable is initialized before being used.

Each immutable variable is not reassigned after initialization.

```
val a: Int  
  
while(true) {  
    if (Random.nextBoolean()) {  
        a = 15  
        break  
    }  
}  
  
println(a) // It compiles!
```

Kotlin compiler: control- and data-flow analysis

- Variable initialization analysis
- Return analysis
- Smart cast analysis

If the return type is not `Unit`, then the function won't return control unless it returns something.

```
fun bar(): Int {  
    print("Again")  
    while (true) {  
        print(" and again")  
    }  
} // It compiles!  
  
fun baz(): Long {  
    error("YOLO! :)")  
} // It compiles!
```

Kotlin compiler: control- and data-flow analysis

- Variable initialization analysis
- Return analysis
- Smart cast analysis

If type check is successful, then the checked value is automatically casted to the corresponding type.

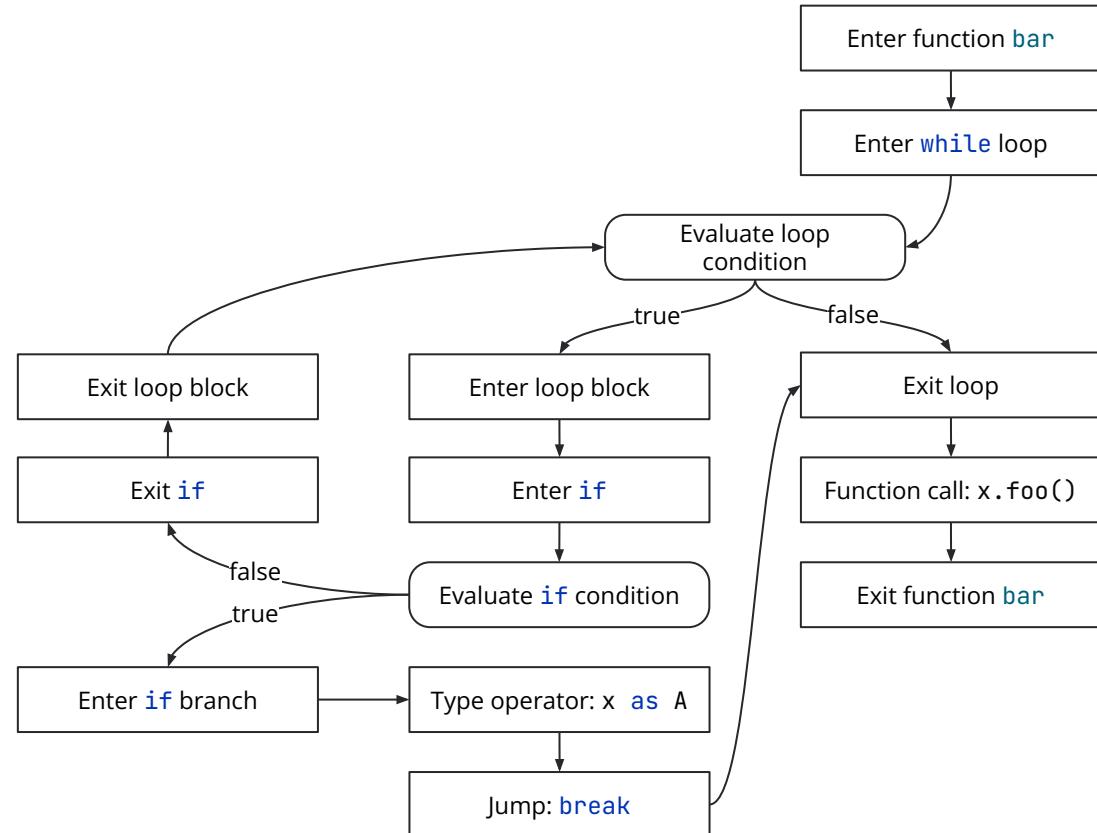
```
fun Any?.printFirstElement() {  
    when (this) {  
        is List<*> → get(0)  
        is Iterable<*> → iterator().next()  
    }  
}
```

```
fun String?.length(): Int =  
    if (this == null) 0  
    else length
```

```
fun Int?.isEven(): Boolean =  
    this ≠ null && this % 2 == 0
```

Kotlin compiler: control- and data-flow analysis

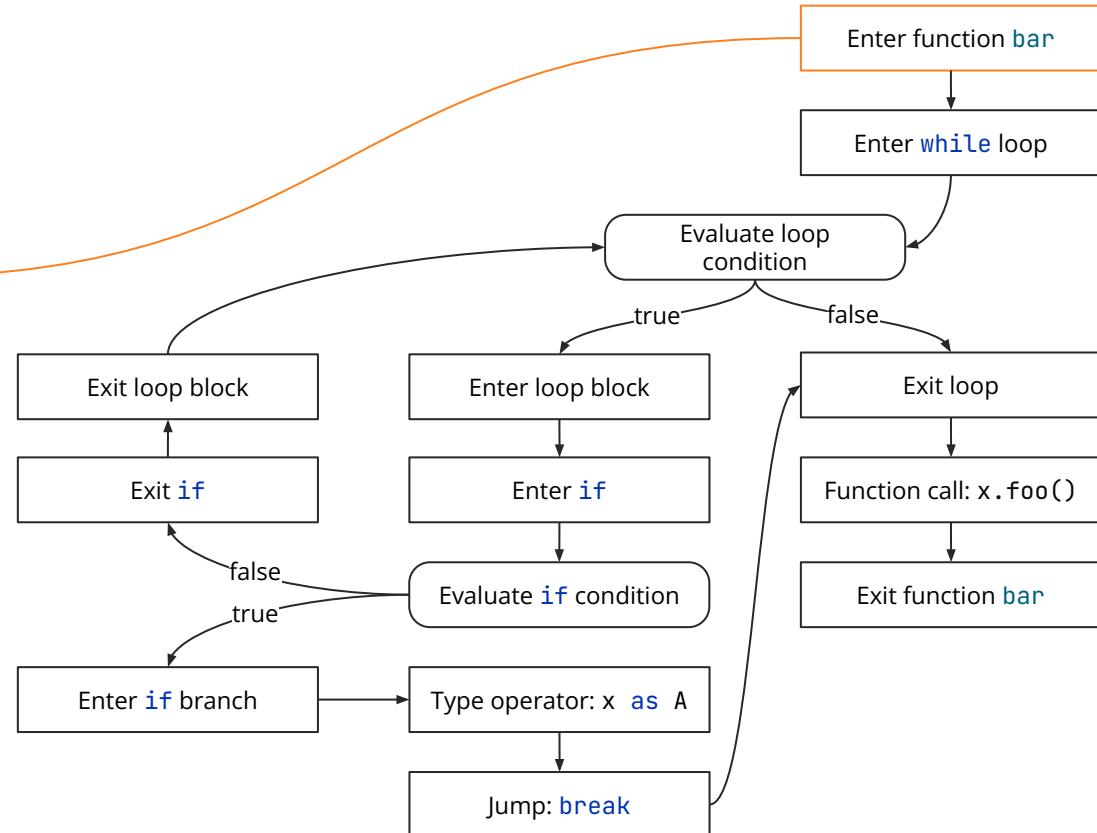
```
interface A {  
    fun foo()  
}  
  
fun bar(x: Any, b: Boolean) {  
    while (true) {  
        if (b) {  
            x as A  
            break  
        }  
    }  
    x.foo()  
}
```



Kotlin compiler: control- and data-flow analysis

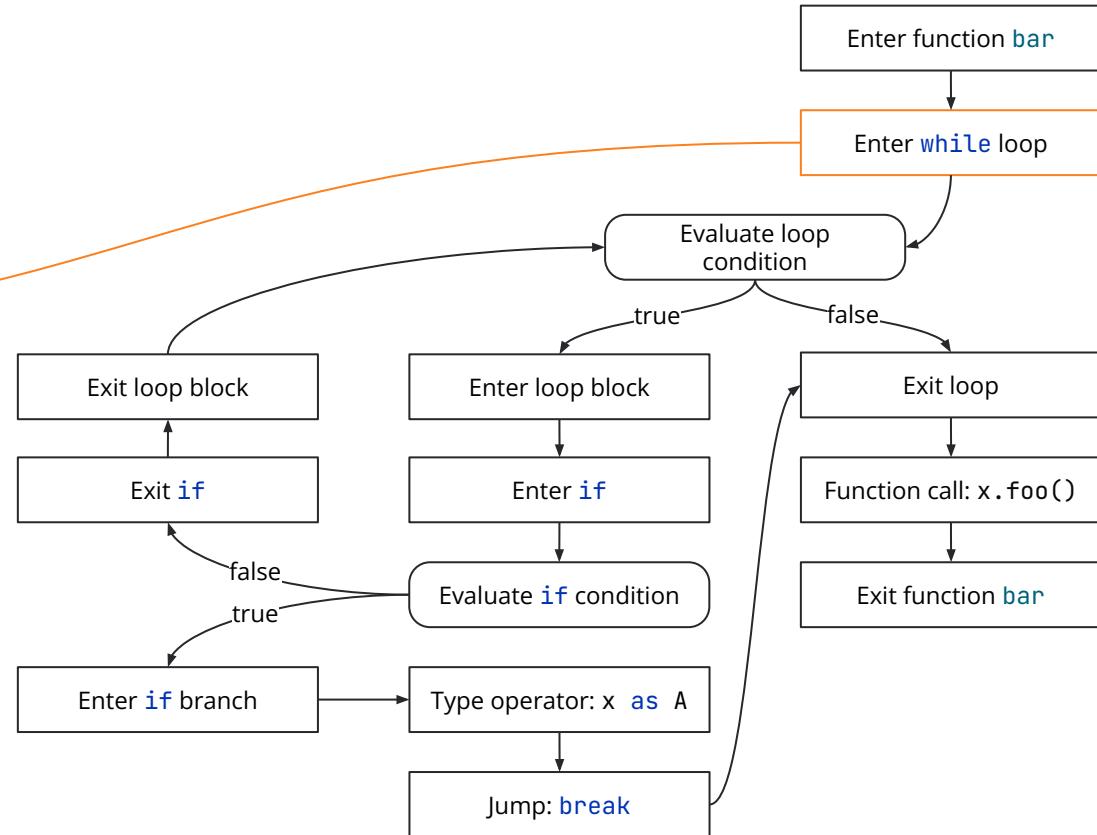
```
interface A {  
    fun foo()  
}
```

```
fun bar(x: Any, b: Boolean) {  
    while (true) {  
        if (b) {  
            x as A  
            break  
        }  
    }  
    x.foo()  
}
```



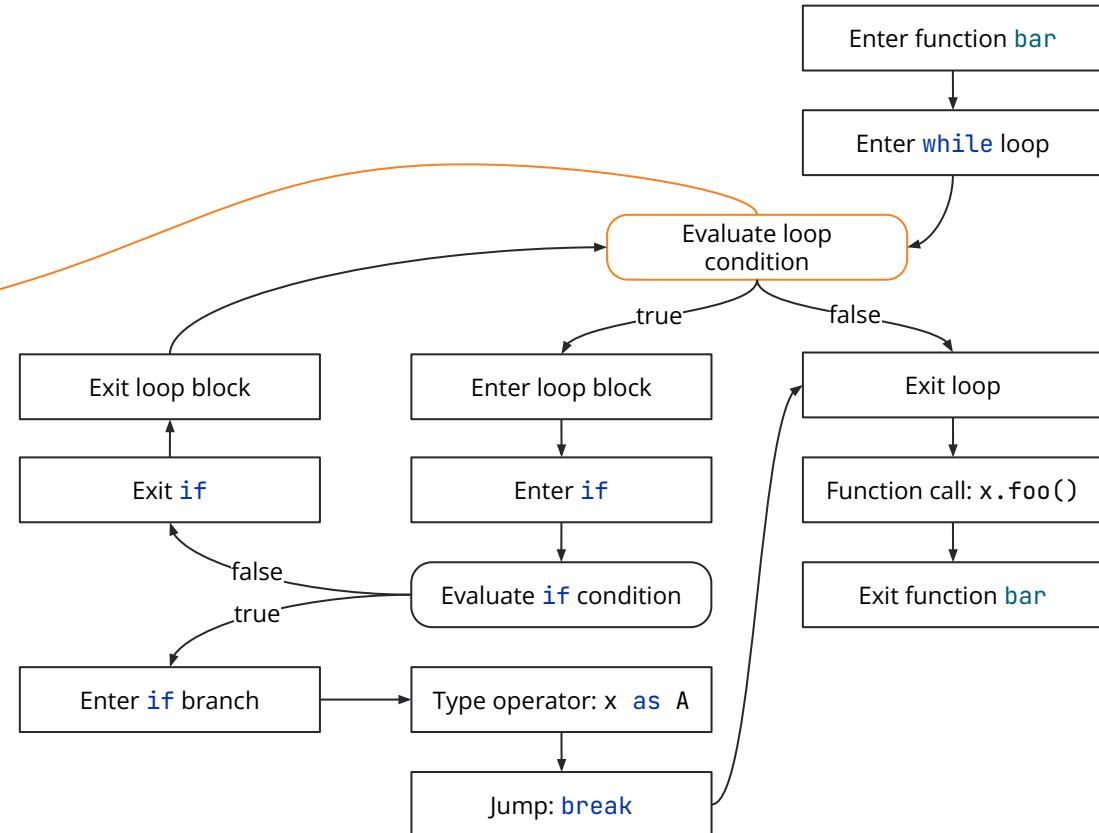
Kotlin compiler: control- and data-flow analysis

```
interface A {  
    fun foo()  
}  
  
fun bar(x: Any, b: Boolean) {  
    while (true) {  
        if (b) {  
            x as A  
            break  
        }  
    }  
    x.foo()  
}
```



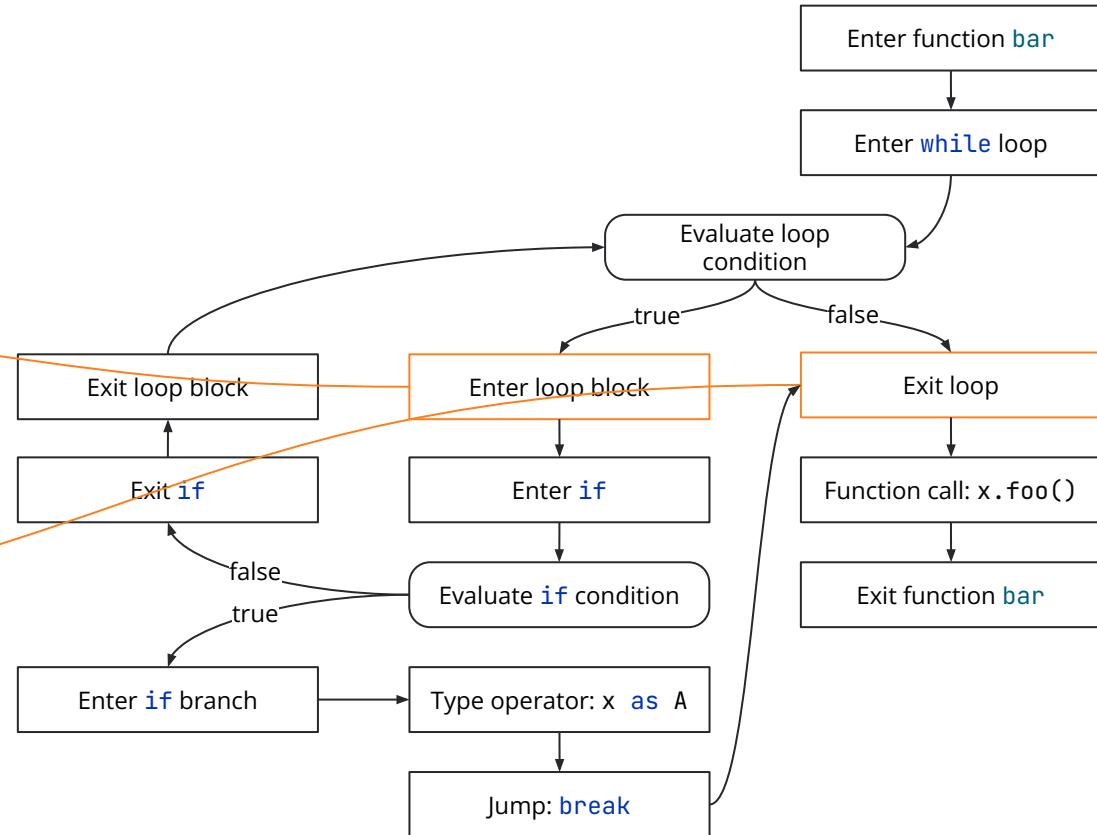
Kotlin compiler: control- and data-flow analysis

```
interface A {  
    fun foo()  
}  
  
fun bar(x: Any, b: Boolean) {  
    while (true) {  
        if (b) {  
            x as A  
            break  
        }  
    }  
    x.foo()  
}
```



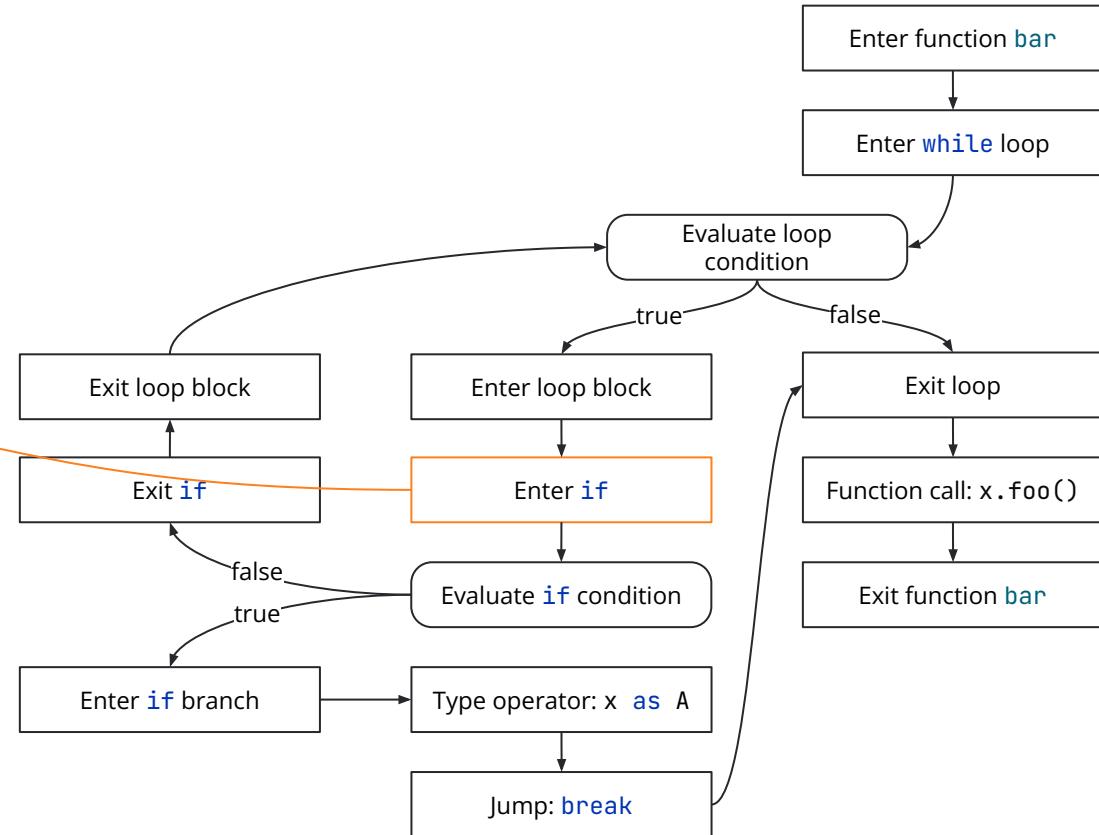
Kotlin compiler: control- and data-flow analysis

```
interface A {  
    fun foo()  
}  
  
fun bar(x: Any, b: Boolean) {  
    while (true) {  
        if (b) {  
            x as A  
            break  
        }  
    }  
    x.foo()  
}
```



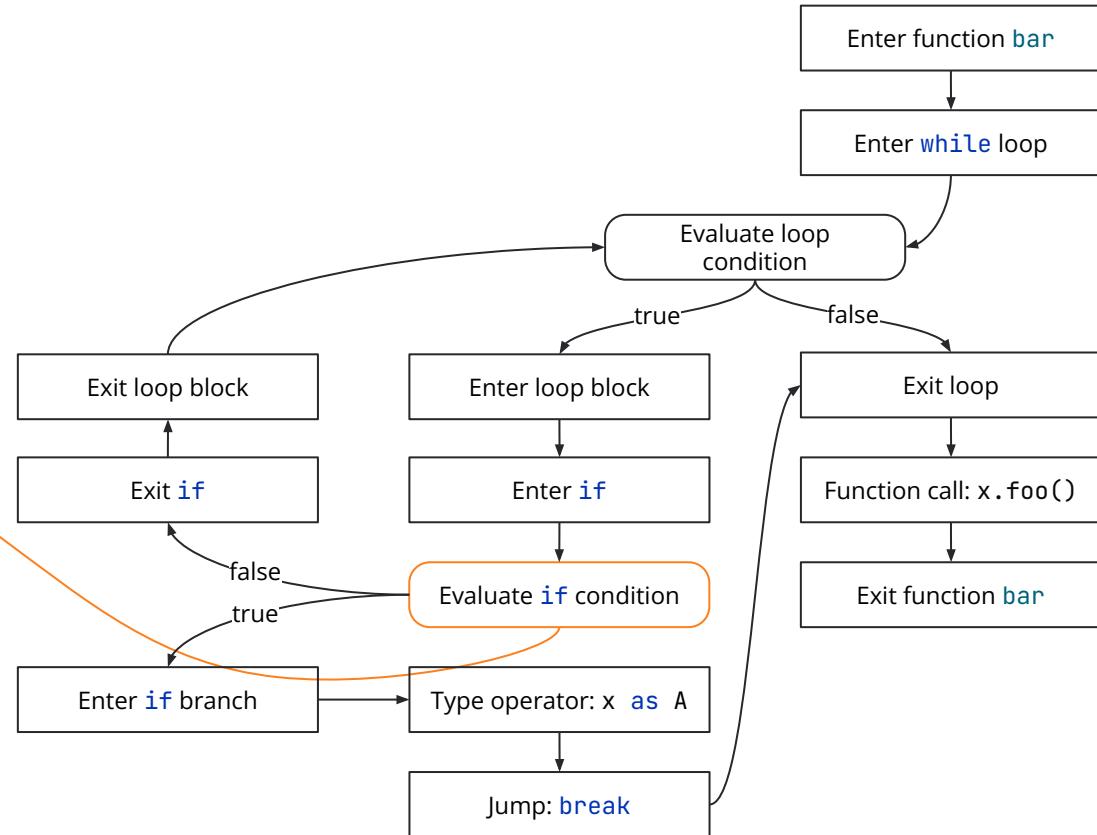
Kotlin compiler: control- and data-flow analysis

```
interface A {  
    fun foo()  
}  
  
fun bar(x: Any, b: Boolean) {  
    while (true) {  
        if (b) {  
            x as A  
            break  
        }  
    }  
    x.foo()  
}
```



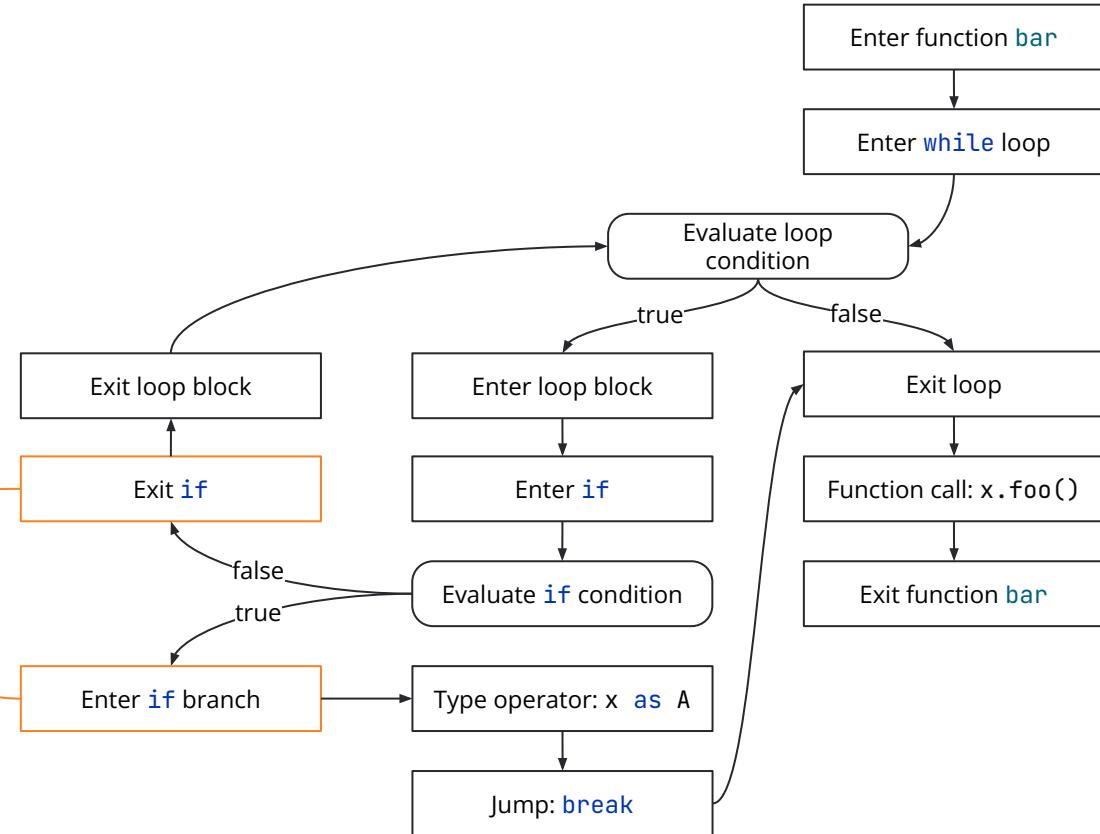
Kotlin compiler: control- and data-flow analysis

```
interface A {  
    fun foo()  
}  
  
fun bar(x: Any, b: Boolean) {  
    while (true) {  
        if (b) {  
            x as A  
            break  
        }  
    }  
    x.foo()  
}
```



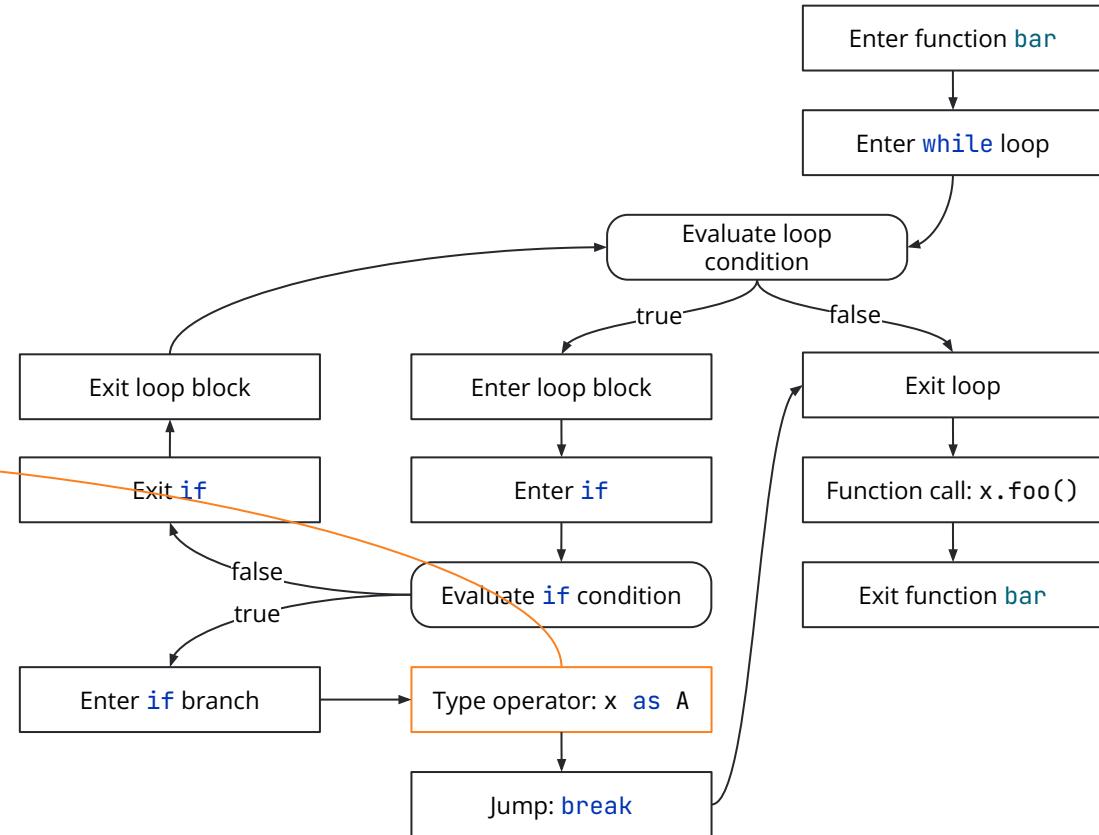
Kotlin compiler: control- and data-flow analysis

```
interface A {  
    fun foo()  
}  
  
fun bar(x: Any, b: Boolean) {  
    while (true) {  
        if (b) {  
            x as A  
            break  
        }  
    }  
    x.foo()  
}
```



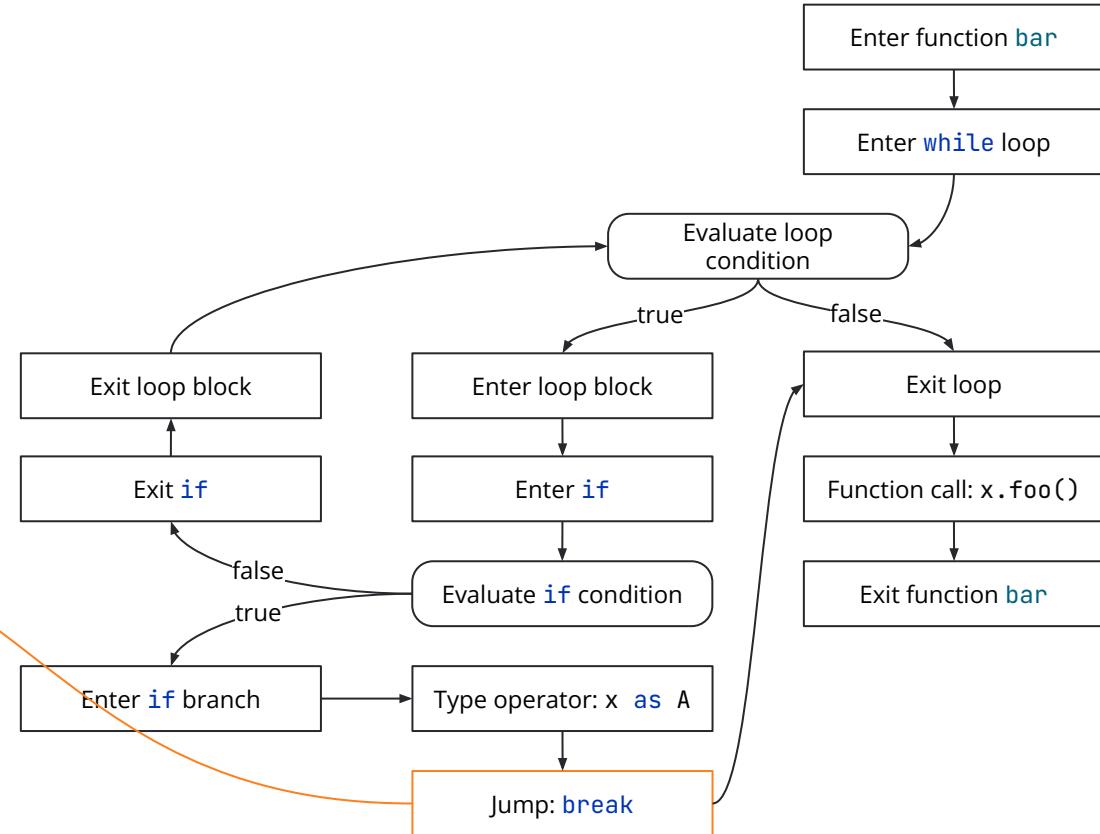
Kotlin compiler: control- and data-flow analysis

```
interface A {  
    fun foo()  
}  
  
fun bar(x: Any, b: Boolean) {  
    while (true) {  
        if (b) {  
            x as A  
            break  
        }  
    }  
    x.foo()  
}
```



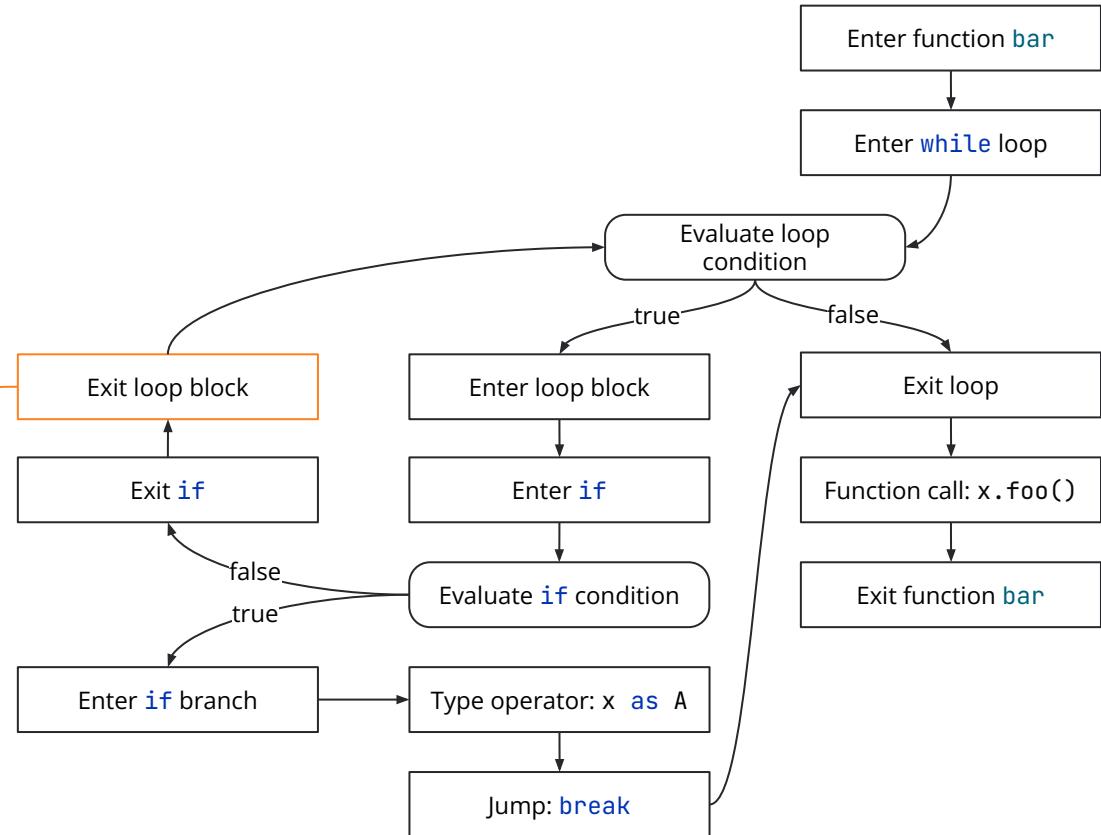
Kotlin compiler: control- and data-flow analysis

```
interface A {  
    fun foo()  
}  
  
fun bar(x: Any, b: Boolean) {  
    while (true) {  
        if (b) {  
            x as A  
            break  
        }  
    }  
    x.foo()  
}
```



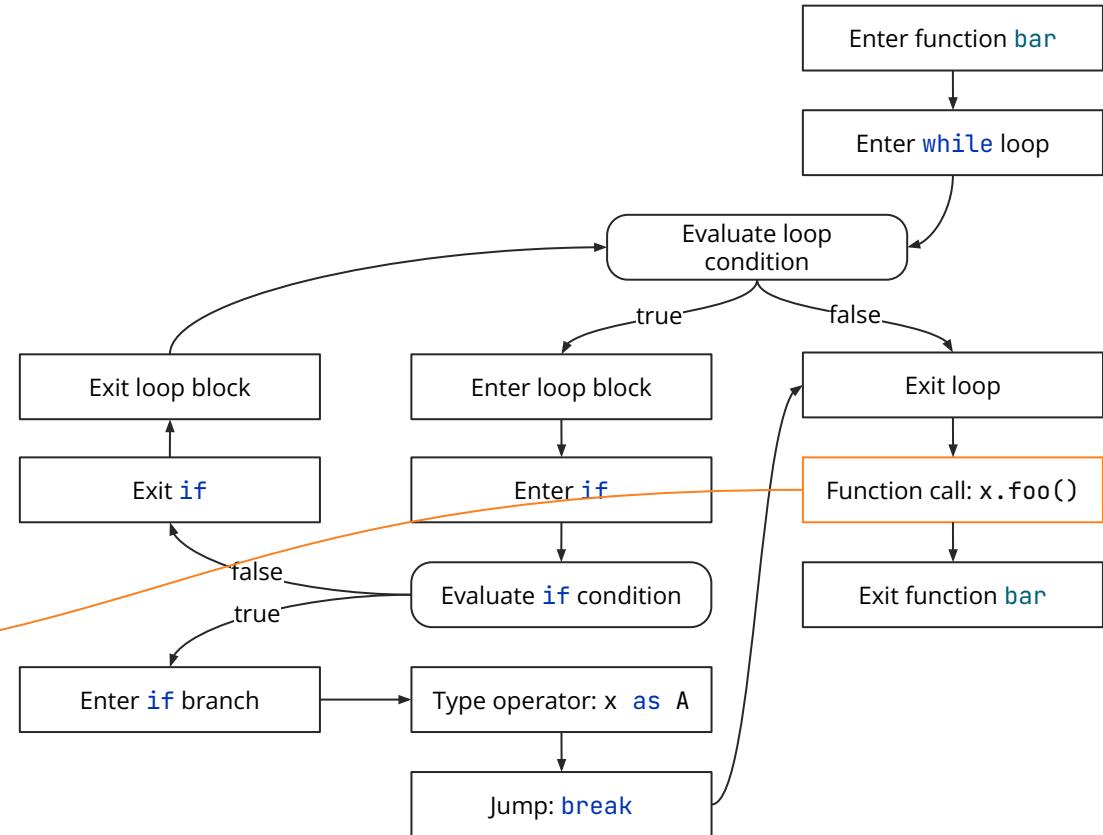
Kotlin compiler: control- and data-flow analysis

```
interface A {  
    fun foo()  
}  
  
fun bar(x: Any, b: Boolean) {  
    while (true) {  
        if (b) {  
            x as A  
            break  
        }  
    }  
    x.foo()  
}
```



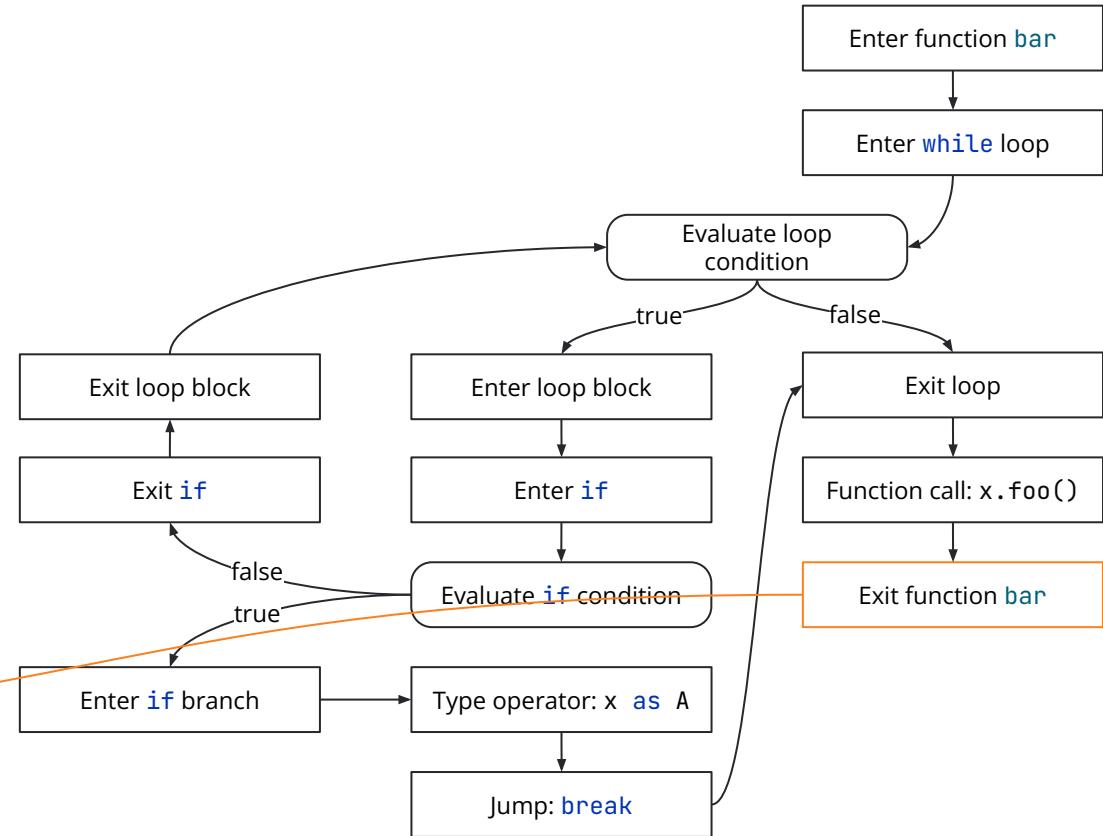
Kotlin compiler: control- and data-flow analysis

```
interface A {  
    fun foo()  
}  
  
fun bar(x: Any, b: Boolean) {  
    while (true) {  
        if (b) {  
            x as A  
            break  
        }  
    }  
    x.foo()  
}
```



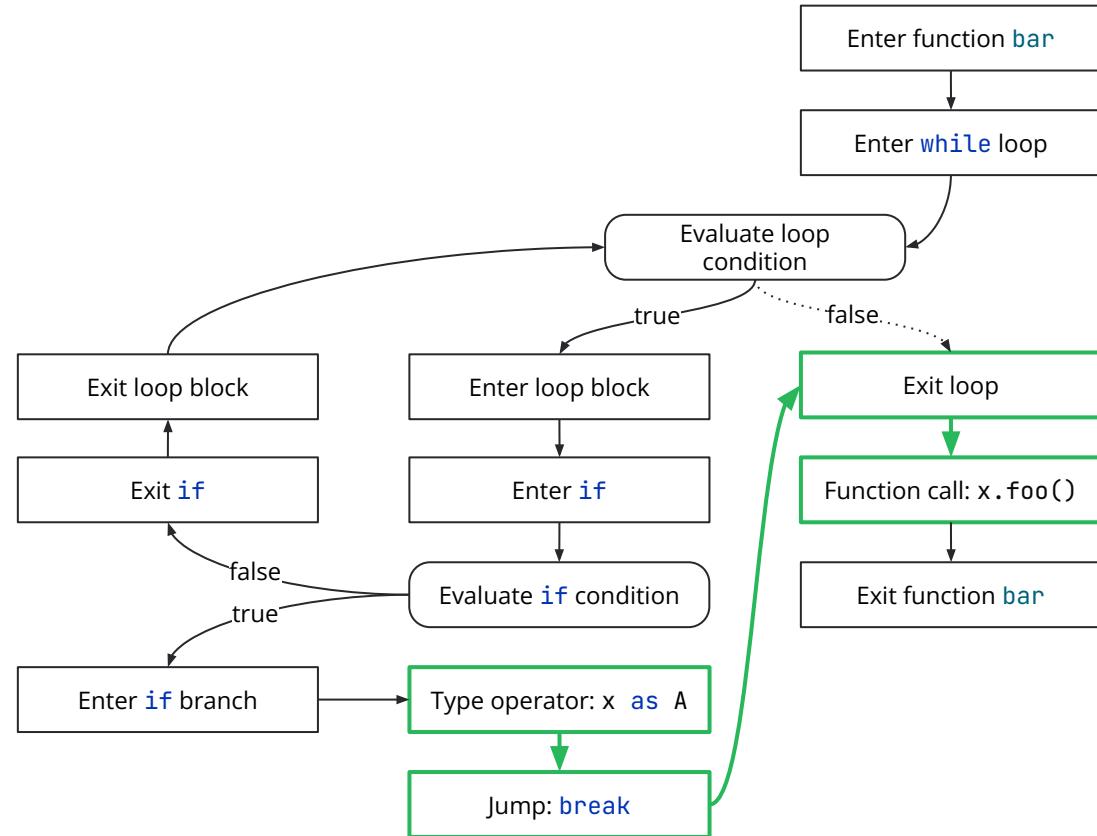
Kotlin compiler: control- and data-flow analysis

```
interface A {  
    fun foo()  
}  
  
fun bar(x: Any, b: Boolean) {  
    while (true) {  
        if (b) {  
            x as A  
            break  
        }  
    }  
    x.foo()  
}
```

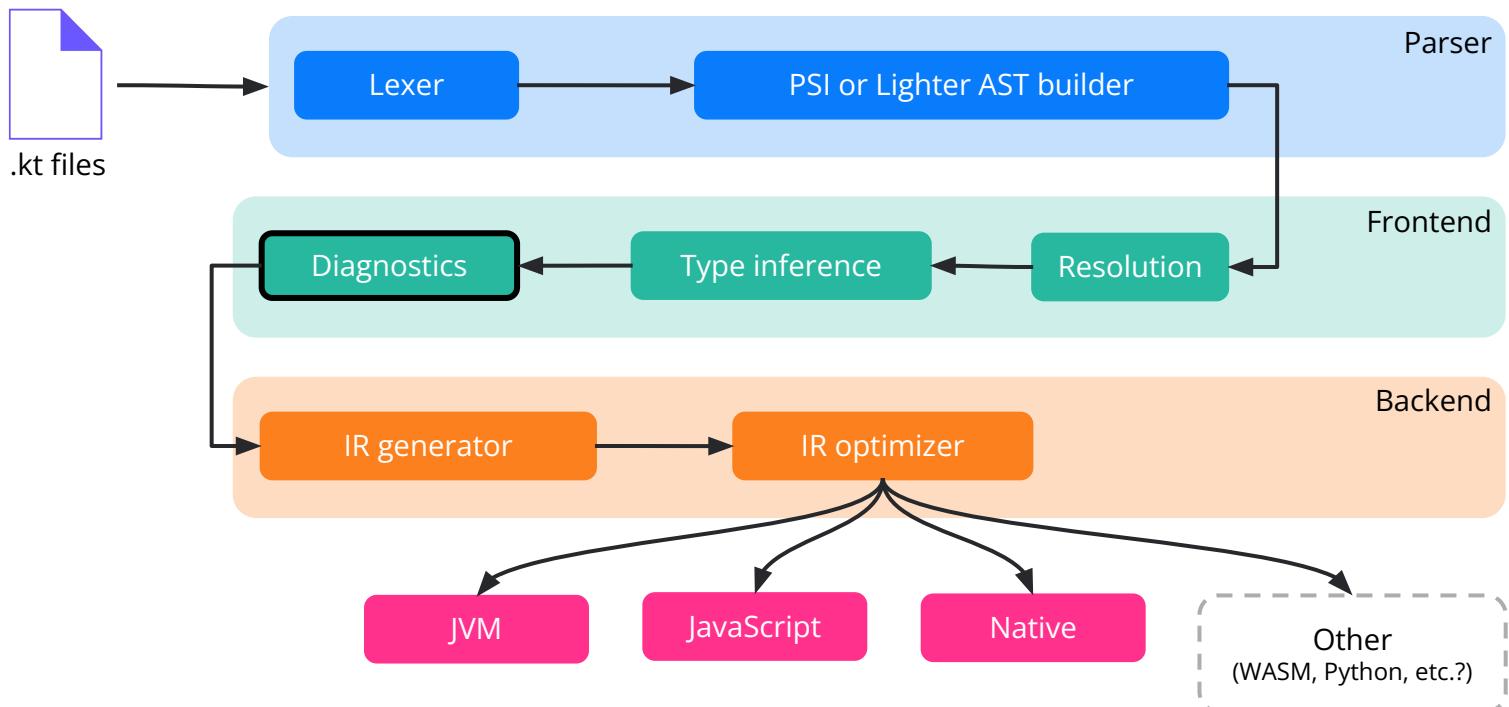


Kotlin compiler: control- and data-flow analysis

```
interface A {  
    fun foo()  
}  
  
fun bar(x: Any, b: Boolean) {  
    while (true) {  
        if (b) {  
            x as A  
            break  
        }  
    }  
    x.foo()  
}
```



Kotlin compiler



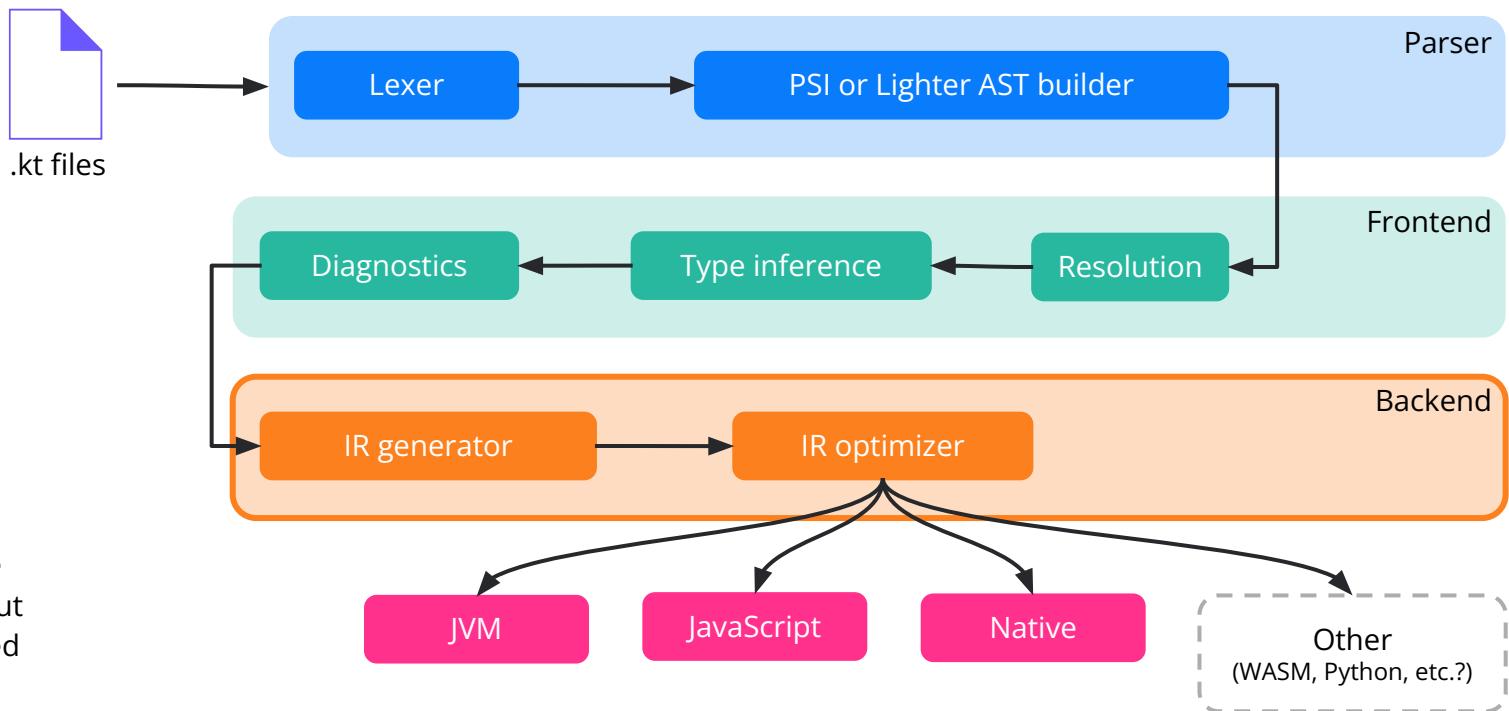
Kotlin compiler: diagnostics

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

Only one class may appear in a supertype list :
Remove supertype Alt+Shift+Enter More actions... Alt+Enter

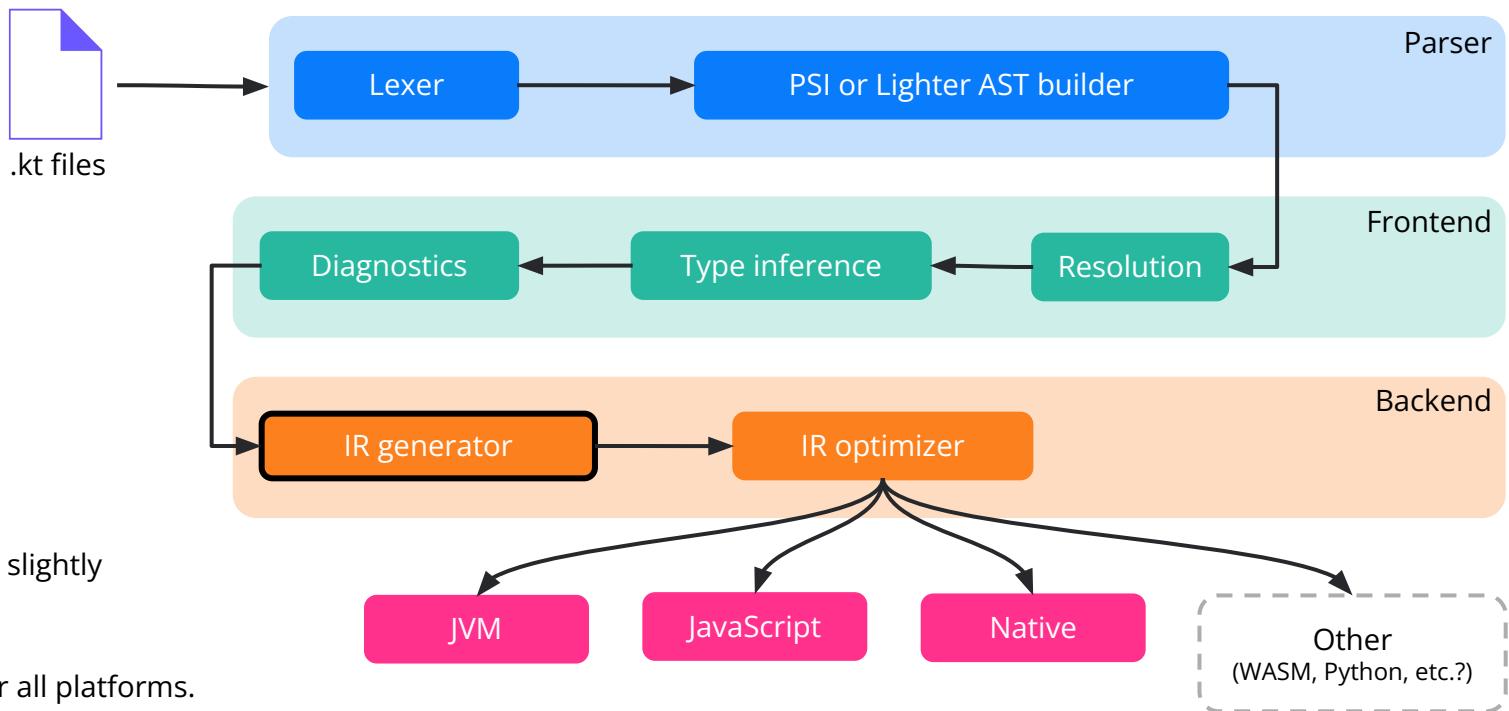
```
public open class B  
main.kt  
kotlinTest.wasmMain
```

Kotlin compiler



On the backend, we
DO NOT resolve, but
only use the received
information

Kotlin compiler

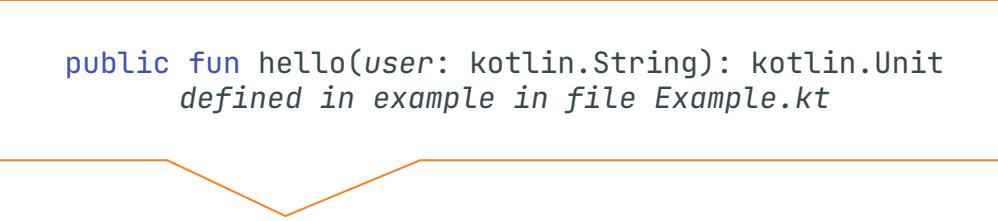


Kotlin compiler: resolved tree

```
fun hello(user: String) = println("Hello, $user")
```

Kotlin compiler: resolved tree

```
public fun hello(user: kotlin.String): kotlin.Unit  
defined in example in file Example.kt
```



```
fun hello(user: String) = println("Hello, $user")
```

Kotlin compiler: resolved tree

Reference to *value-parameter user*: kotlin.String
defined in example.hello

```
fun hello(user: String) = println("Hello, $user")
```

Kotlin compiler: resolved tree

```
fun hello(user: String) = println("Hello, $user")
```

Type: kotlin.String

Kotlin compiler: resolved tree

```
Type: kotlin.String  
fun hello(user: String) = println("Hello, $user")
```

Kotlin compiler: resolved tree

Type: kotlin.Unit

```
fun hello(user: String) = println("Hello, $user")
```

Kotlin compiler: resolved tree

Call: kotlin.io.println(kotlin.String)

```
fun hello(user: String) = println("Hello, $user")
```

IR? Yet another tree!

The code:

```
// file 'src/kotlin/example.kt'

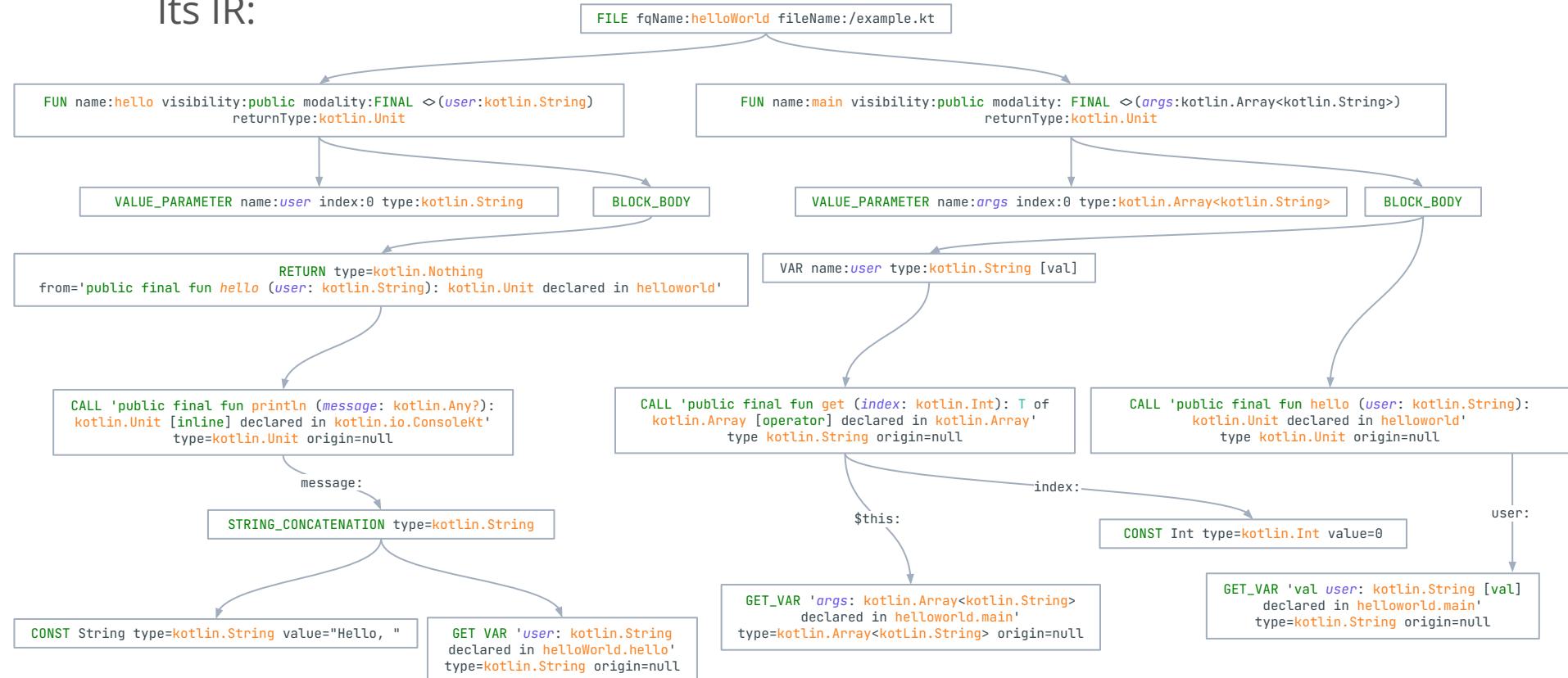
package helloworld

fun hello(user: String) = println("Hello, $user")

fun main(args: Array<String>) {
    val user = args[0]
    hello(user)
}
```

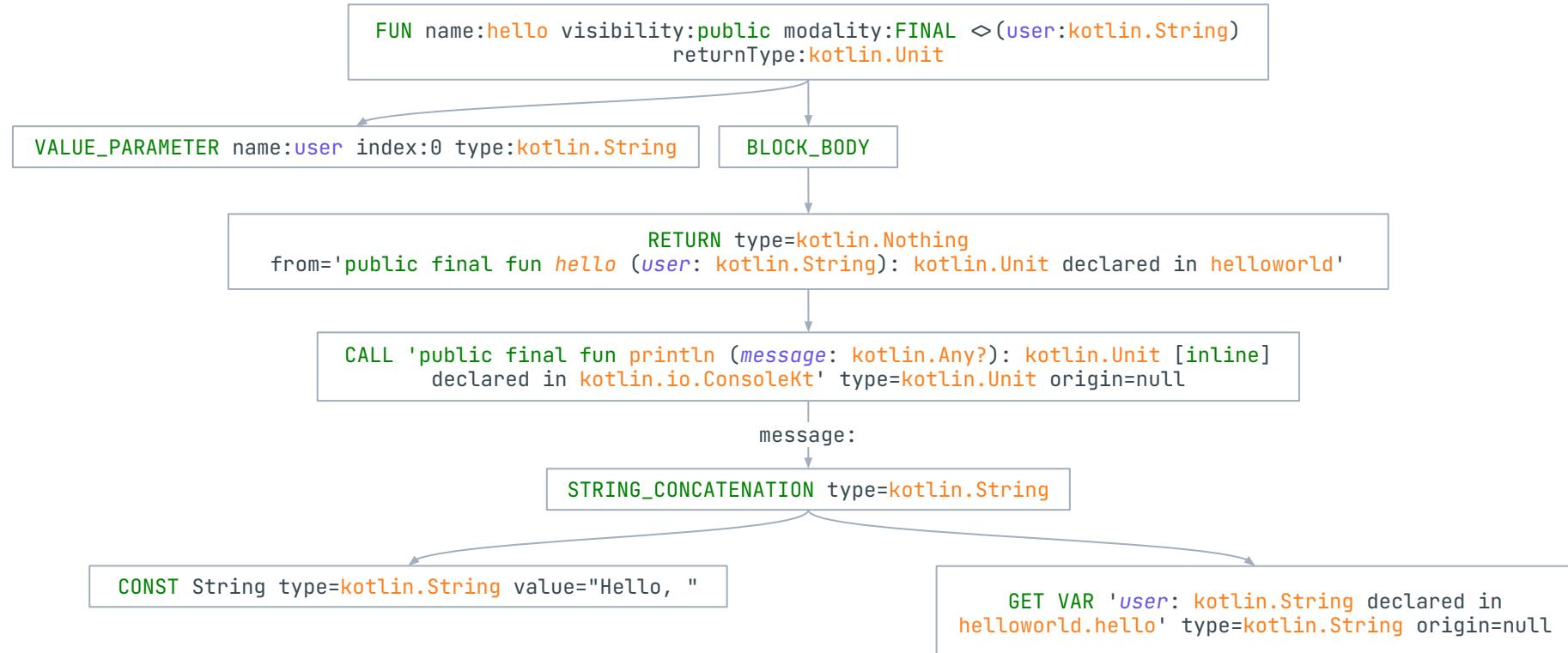
IR? Yet another tree!

Its IR:

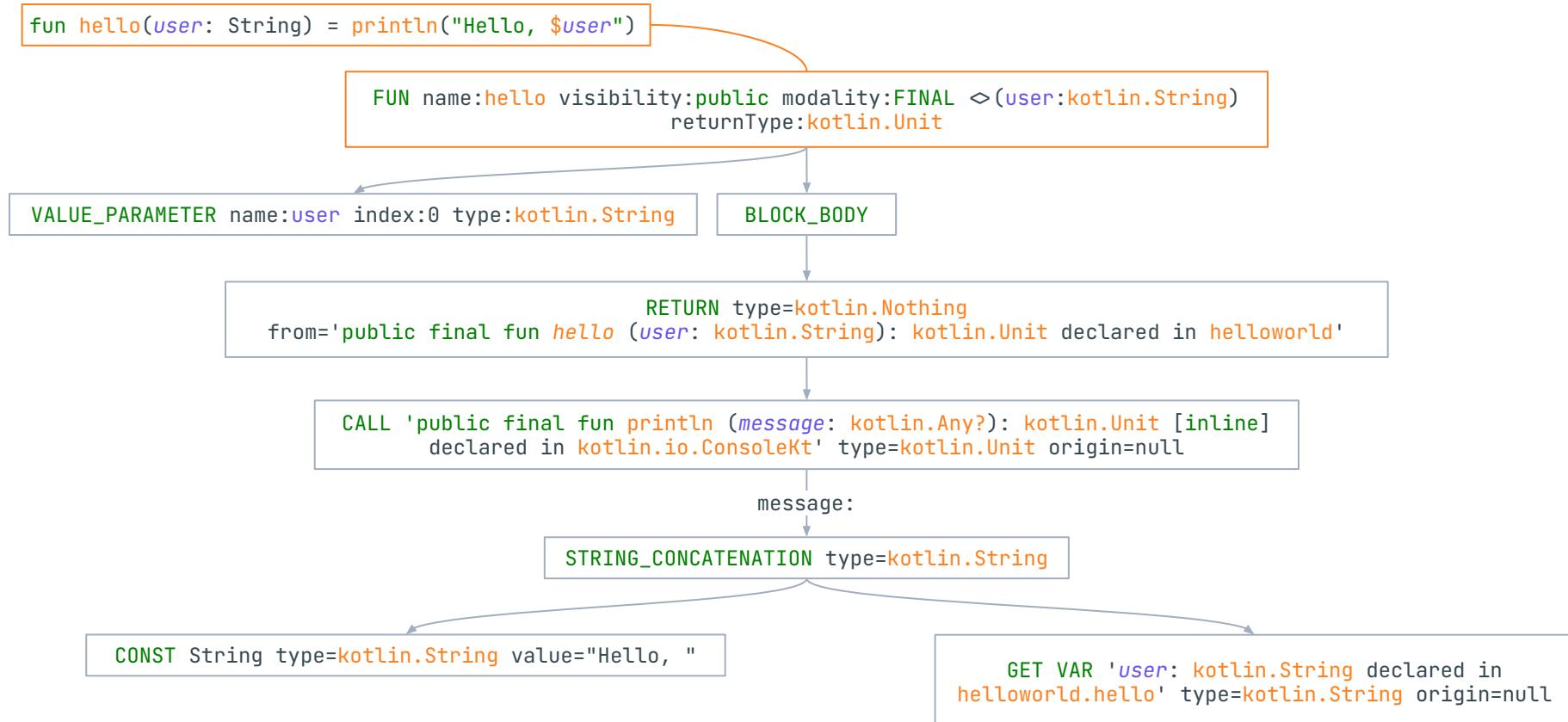


Back-end intermediate representation: a closer look

```
fun hello(user: String) = println("Hello, $user")
```



Back-end intermediate representation: a closer look



Back-end intermediate representation: a closer look

```
fun hello(user: String) = println("Hello, $user")
```

```
FUN name:hello visibility:public modality:FINAL ◇(user:kotlin.String)  
returnType:kotlin.Unit
```

```
VALUE_PARAMETER name:user index:0 type:kotlin.String
```

```
BLOCK_BODY
```

```
RETURN type=kotlin.Nothing  
from='public final fun hello (user: kotlin.String): kotlin.Unit declared in helloworld'
```

```
CALL 'public final fun println (message: kotlin.Any?): kotlin.Unit [inline]  
declared in kotlin.io.ConsoleKt' type=kotlin.Unit origin=null
```

```
message:
```

```
STRING_CONCATENATION type=kotlin.String
```

```
CONST String type=kotlin.String value="Hello, "
```

```
GET VAR 'user: kotlin.String declared in  
helloworld.hello' type=kotlin.String origin=null
```

Back-end intermediate representation: a closer look

```
fun hello(user: String) = println("Hello, $user")
```

```
FUN name:hello visibility:public modality:FINAL ◇(user:kotlin.String)  
returnType:kotlin.Unit
```

```
VALUE_PARAMETER name:user index:0 type:kotlin.String
```

```
BLOCK_BODY
```

```
RETURN type=kotlin.Nothing  
from='public final fun hello (user: kotlin.String): kotlin.Unit declared in helloworld'
```

```
CALL 'public final fun println (message: kotlin.Any?): kotlin.Unit [inline]  
declared in kotlin.io.ConsoleKt' type=kotlin.Unit origin=null
```

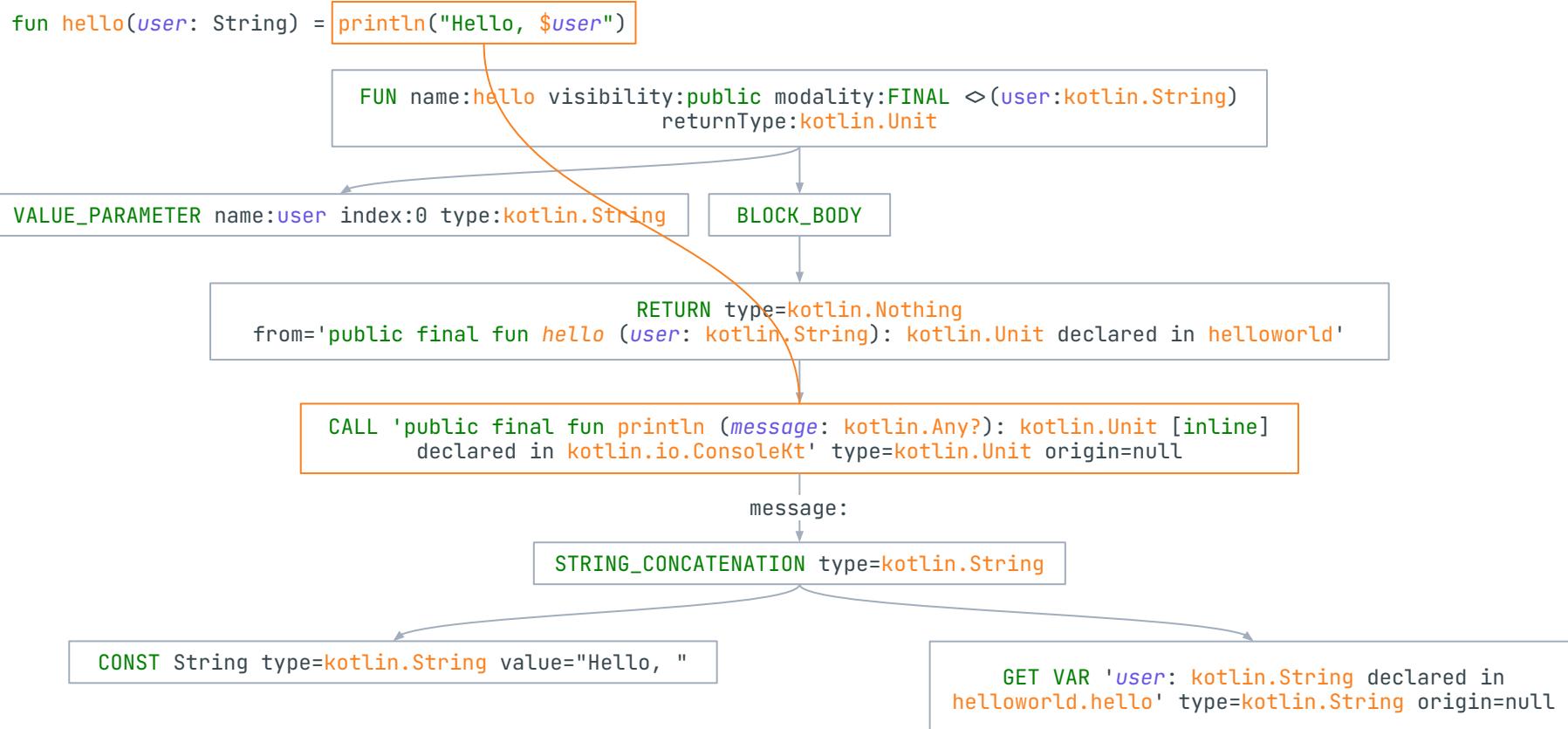
```
message:
```

```
STRING_CONCATENATION type=kotlin.String
```

```
CONST String type=kotlin.String value="Hello, "
```

```
GET VAR 'user: kotlin.String declared in  
helloworld.hello' type=kotlin.String origin=null
```

Back-end intermediate representation: a closer look



Back-end intermediate representation: a closer look

```
fun hello(user: String) = println("Hello, $user")
```

```
FUN name:hello visibility:public modality:FINAL ◇(user:kotlin.String)  
returnType:kotlin.Unit
```

```
VALUE_PARAMETER name:user index:0 type:kotlin.String
```

```
BLOCK_BODY
```

```
RETURN type=kotlin.Nothing  
from='public final fun hello (user: kotlin.String): kotlin.Unit declared in helloworld'
```

```
CALL 'public final fun println (message: kotlin.Any?): kotlin.Unit [inline]  
declared in kotlin.io.ConsoleKt' type=kotlin.Unit origin=null
```

```
message:
```

```
STRING_CONCATENATION type=kotlin.String
```

```
CONST String type=kotlin.String value="Hello, "
```

```
GET VAR 'user: kotlin.String declared in  
helloworld.hello' type=kotlin.String origin=null
```

Back-end intermediate representation: a closer look

```
fun hello(user: String) = println("Hello, $user")
```

```
FUN name:hello visibility:public modality:FINAL ◇(user:kotlin.String)  
returnType:kotlin.Unit
```

```
VALUE_PARAMETER name:user index:0 type:kotlin.String
```

```
BLOCK_BODY
```

```
RETURN type=kotlin.Nothing  
from='public final fun hello (user: kotlin.String): kotlin.Unit declared in helloworld'
```

```
CALL 'public final fun println (message: kotlin.Any?): kotlin.Unit [inline]  
declared in kotlin.io.ConsoleKt' type=kotlin.Unit origin=null
```

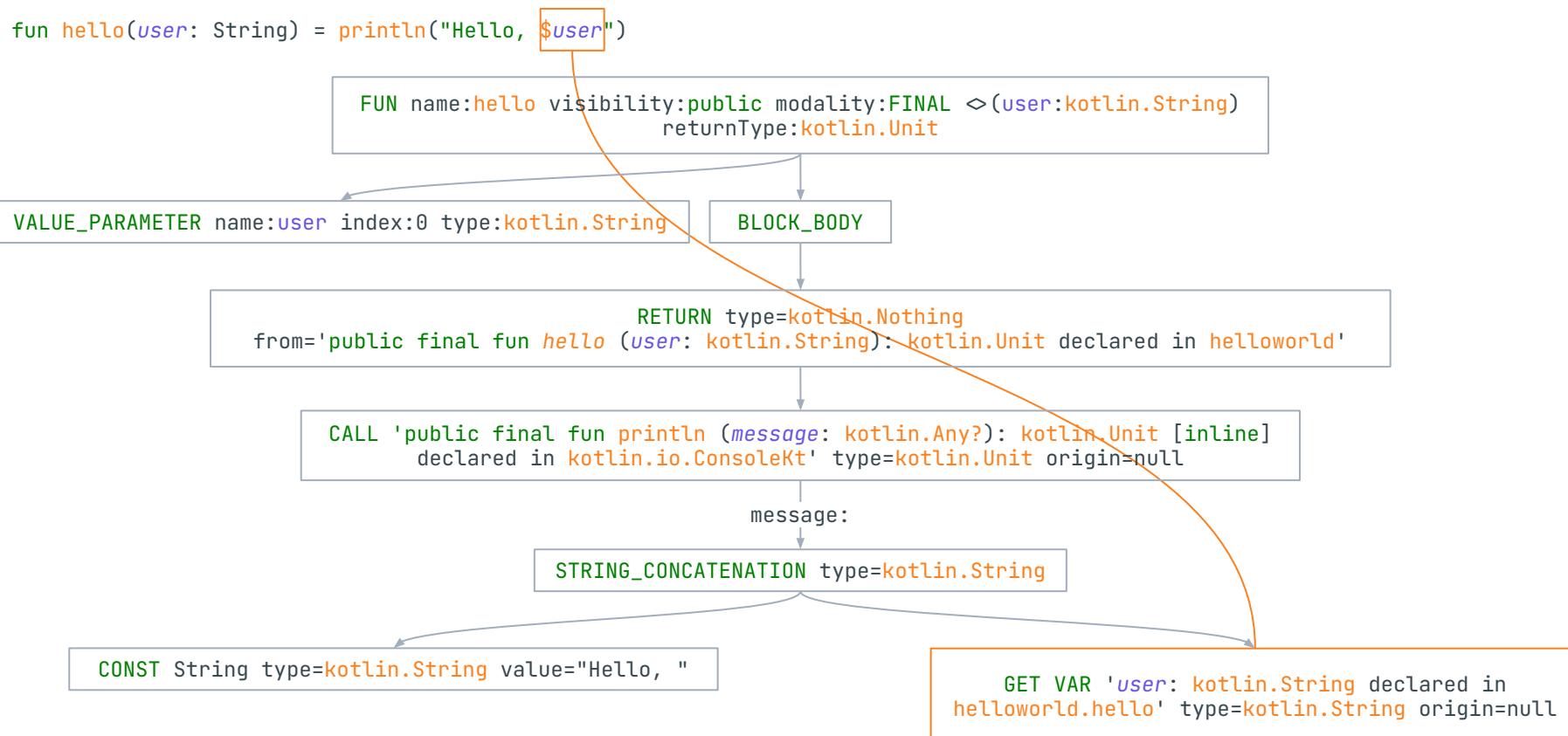
```
message:
```

```
STRING_CONCATENATION type=kotlin.String
```

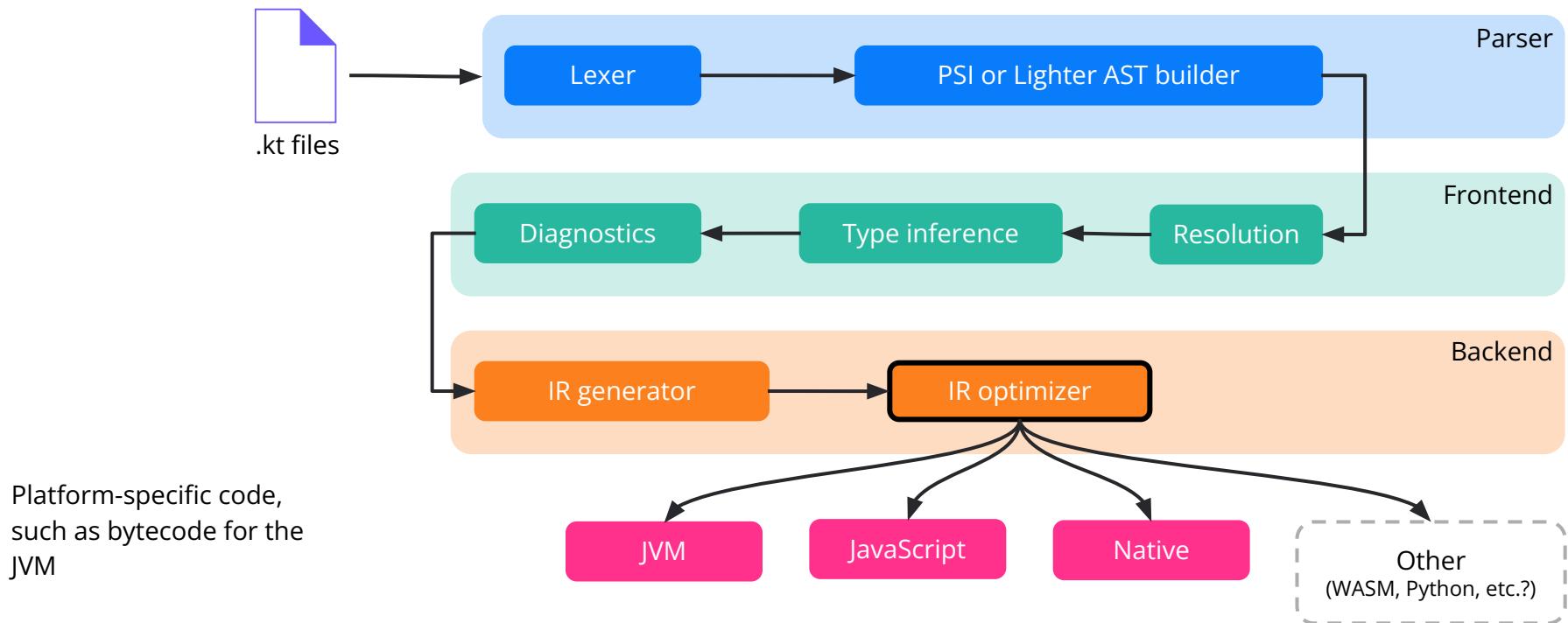
```
CONST String type=kotlin.String value="Hello, "
```

```
GET VAR 'user: kotlin.String declared in  
helloworld.hello' type=kotlin.String origin=null
```

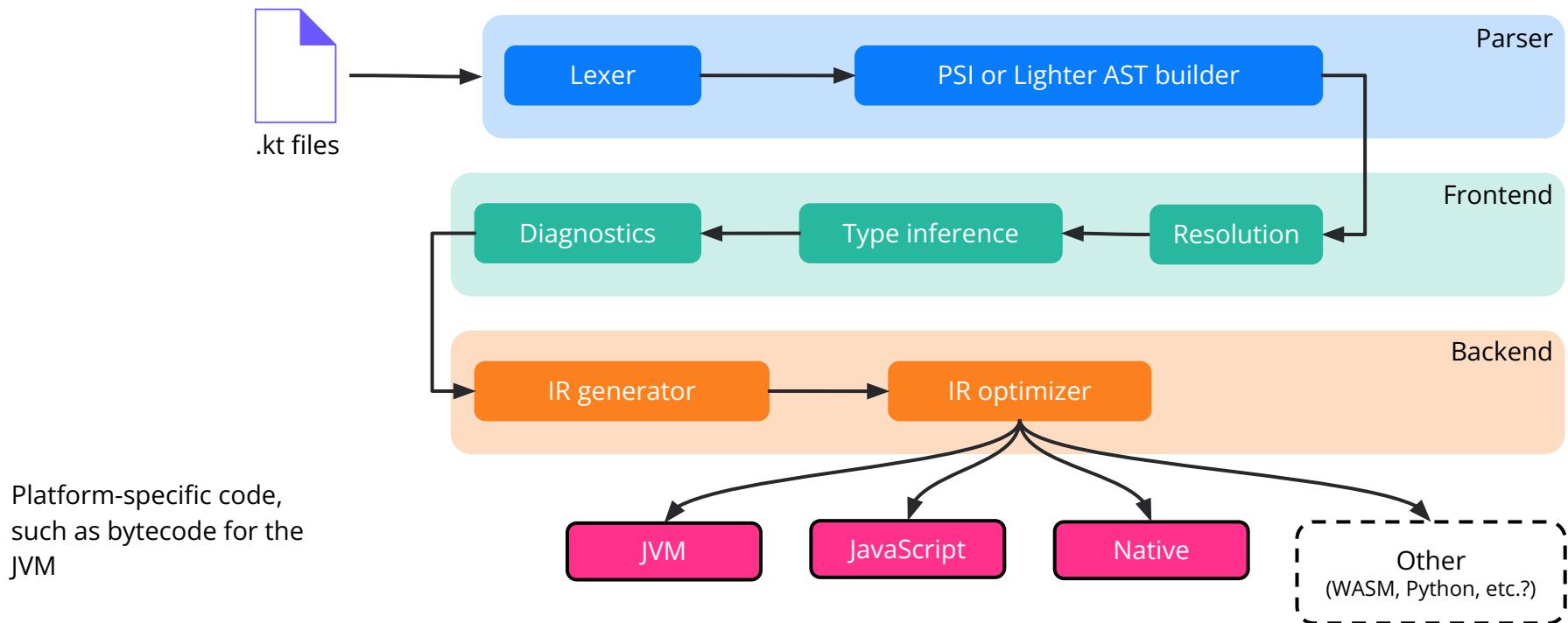
Back-end intermediate representation: a closer look



Kotlin compiler

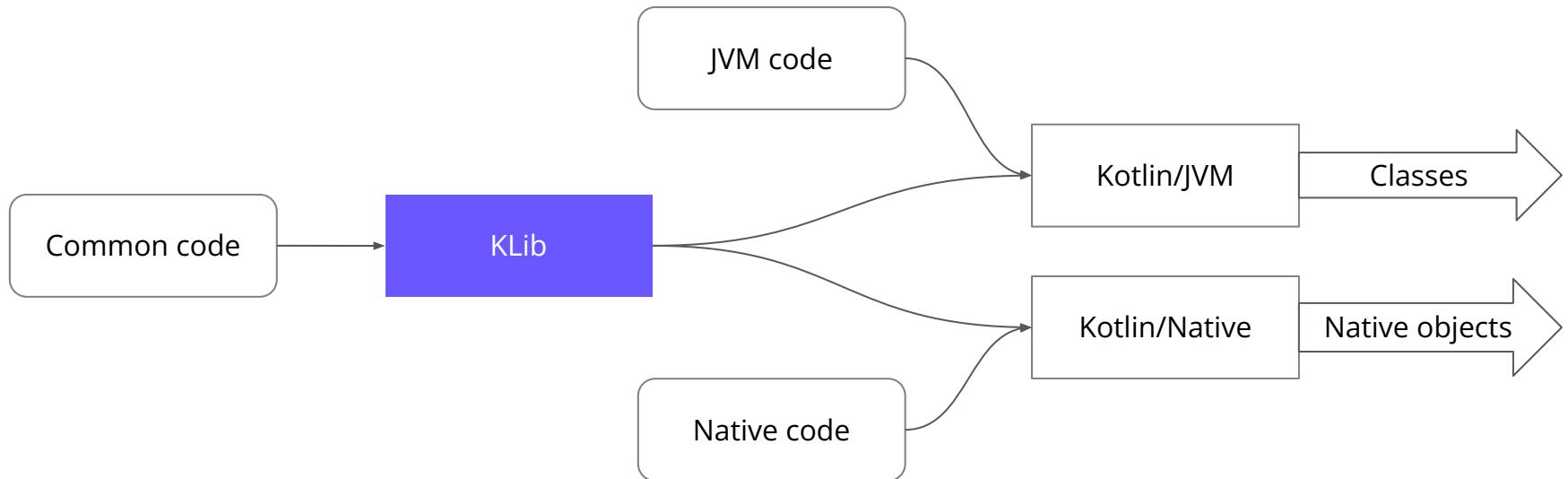


Kotlin compiler



KLibs

JAR analogues – store a serialized IR for the subsequent use of cross-platform libraries.



Compiler plugins

You can extend **any** compile phase via compiler plugins. To do so, you need to implement compiler extensions and register them.

To register extensions, you need to inherit from [CompilerPluginRegistrar](#) ([ComponentRegistrar](#) for previous versions).

Don't forget to use `supportsK2 = true` if you need to support the FIR frontend

Compiler plugins: FIR extensions

To get the full list of the extensions, please see: [package org.jetbrains.kotlin.fir.extensions](#) ([link](#)).

Consider an example:

```
/*
 * Generates top level class
 *
 * package foo.bar
 *
 * public final class MyClass {
 *     fun foo(): String = "Hello world"
 * }
 */
```

Compiler plugins: FIR extensions

```
class SimpleClassGenerator(session: FirSession) : FirDeclarationGenerationExtension(session) {
    companion object {
        // foo.bar.MyClass
        val MY_CLASS_ID = ClassId(
            FqName.fromSegments(listOf("foo", "bar")),
            Name.identifier("MyClass")
        )
        // foo.bar.MyClass.foo
        val FOO_ID = CallableId(MY_CLASS_ID, Name.identifier("foo"))
    }
    override fun generateClassLikeDeclaration(classId: ClassId): FirClassLikeSymbol<*>? {
        ...
    }
    ...
}
```

Compiler plugins: FIR extensions

```
class SimpleClassGenerator(session: FirSession) : FirDeclarationGenerationExtension(session) {  
    companion object {  
        // foo.bar.MyClass  
        val MY_CLASS_ID = ClassId(  
            FqName.fromSegments(listOf("foo", "bar")),  
            Name.identifier("MyClass")  
        )  
        // foo.bar.MyClass.foo  
        val FOO_ID = CallableId(MY_CLASS_ID, Name.identifier("foo"))  
    }  
    override fun generateClassLikeDeclaration(classId: ClassId): FirClassLikeSymbol<*>? {  
        ...  
    }  
    ...  
}
```

Compiler plugins: FIR extensions

```
class SimpleClassGenerator(session: FirSession) : FirDeclarationGenerationExtension(session) {
    companion object {
        // foo.bar.MyClass
        val MY_CLASS_ID = ClassId(
            FqName.fromSegments(listOf("foo", "bar")),
            Name.identifier("MyClass")
        )
        // foo.bar.MyClass.foo
        val FOO_ID = CallableId(MY_CLASS_ID, Name.identifier("foo"))
    }
    override fun generateClassLikeDeclaration(classId: ClassId): FirClassLikeSymbol<*>? {
        ...
    }
    ...
}
```

Compiler plugins: FIR extensions

```
class SimpleClassGenerator(session: FirSession) : FirDeclarationGenerationExtension(session) {
    companion object {
        // foo.bar.MyClass
        val MY_CLASS_ID = ClassId(
            FqName.fromSegments(listOf("foo", "bar")),
            Name.identifier("MyClass")
        )
        // foo.bar.MyClass.foo
        val FOO_ID = CallableId(MY_CLASS_ID, Name.identifier("foo"))
    }
    override fun generateClassLikeDeclaration(classId: ClassId): FirClassLikeSymbol<*>? {
        ...
    }
    ...
}
```

Compiler plugins: FIR extensions

```
class SimpleClassGenerator(session: FirSession) : FirDeclarationGenerationExtension(session) {
    companion object {
        // foo.bar.MyClass
        val MY_CLASS_ID = ClassId(
            FqName.fromSegments(listOf("foo", "bar")),
            Name.identifier("MyClass")
        )
        // foo.bar.MyClass.foo
        val FOO_ID = CallableId(MY_CLASS_ID, Name.identifier("foo"))
    }
    override fun generateClassLikeDeclaration(classId: ClassId): FirClassLikeSymbol<*>? {
        ...
    }
    ...
}
```

Compiler plugins: FIR extensions

You use a special key to mark **everything** generated by the compiler and can transfer **any** information between frontend and backend. So it also helps to find the new declaration to generate it's IR.

```
class SimpleClassGenerator(session: FirSession) : FirDeclarationGenerationExtension(session) {  
    object Key : GeneratedDeclarationKey()  
    ...  
}
```

Compiler plugins: FIR extensions

Don't forget to register all new declarations.

```
class SimpleClassGenerator(session: FirSession) : FirDeclarationGenerationExtension(session) {  
    ...  
    override fun topLevelClassIds(): Set<ClassId> {  
        return setOf(MY_CLASS_ID)  
    }  
  
    override fun hasPackage(packageFqName: FqName): Boolean {  
        return packageFqName == MY_CLASS_ID.packageFqName  
    }  
}
```

Compiler plugins: IR extensions

Actually, the compiler has only one extension for IRs: [IrGenerationExtension](#).

You just need to implement some *transformers* and accept them:

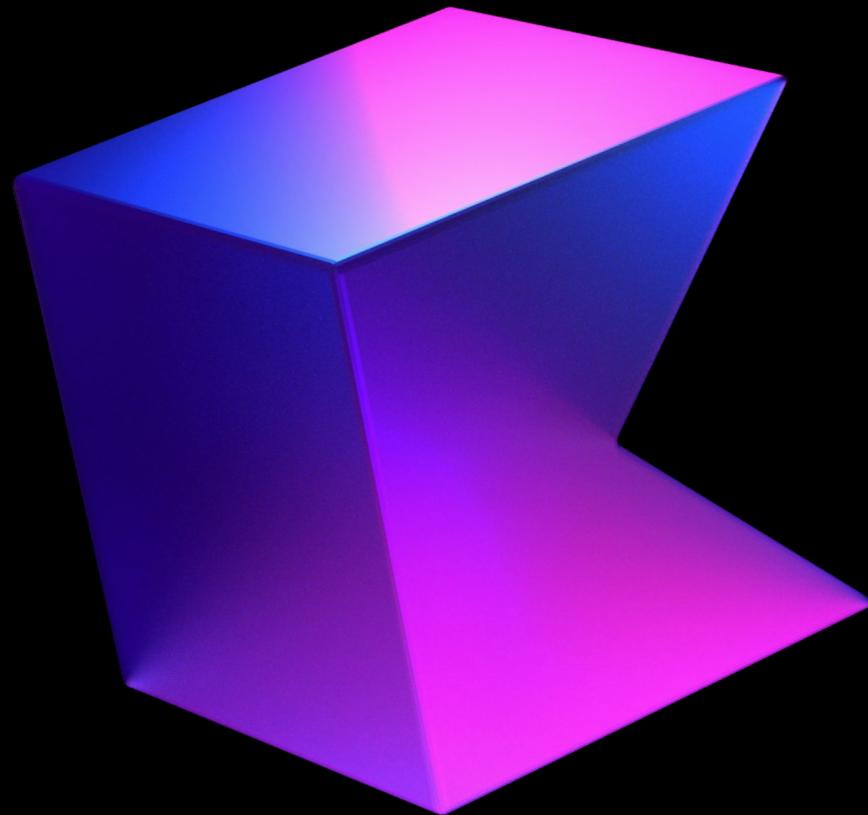
```
class SimpleIrGenerationExtension: IrGenerationExtension {
    override fun generate(moduleFragment: IrModuleFragment, pluginContext: IrPluginContext) {
        val transformers = listOf(SimpleIrBodyGenerator(pluginContext))
        for (transformer in transformers) {
            moduleFragment.acceptChildrenVoid(transformer)
        }
    }
}
```

Don't forget to check the key (use the `interestedIn` function)!

Compiler plugins: popular plugins

- [`kotlinx.serialization`](#) — Generates visitor code for serializable classes.
- [`all-open`](#) — Marks all classes as open classes.
- [`kapt`](#) — An annotation processor.
- [`ksp`](#) — An API for developing lightweight compiler plugins.
- [`Jetpack Compose`](#) — Generates efficient UI from its declarative description.
- [`Arrow Meta`](#) — Compiler plugin API that empowers all Arrow libraries.
- ...

Thanks!



@kotlin



Parallel & Concurrent Programming



Definition

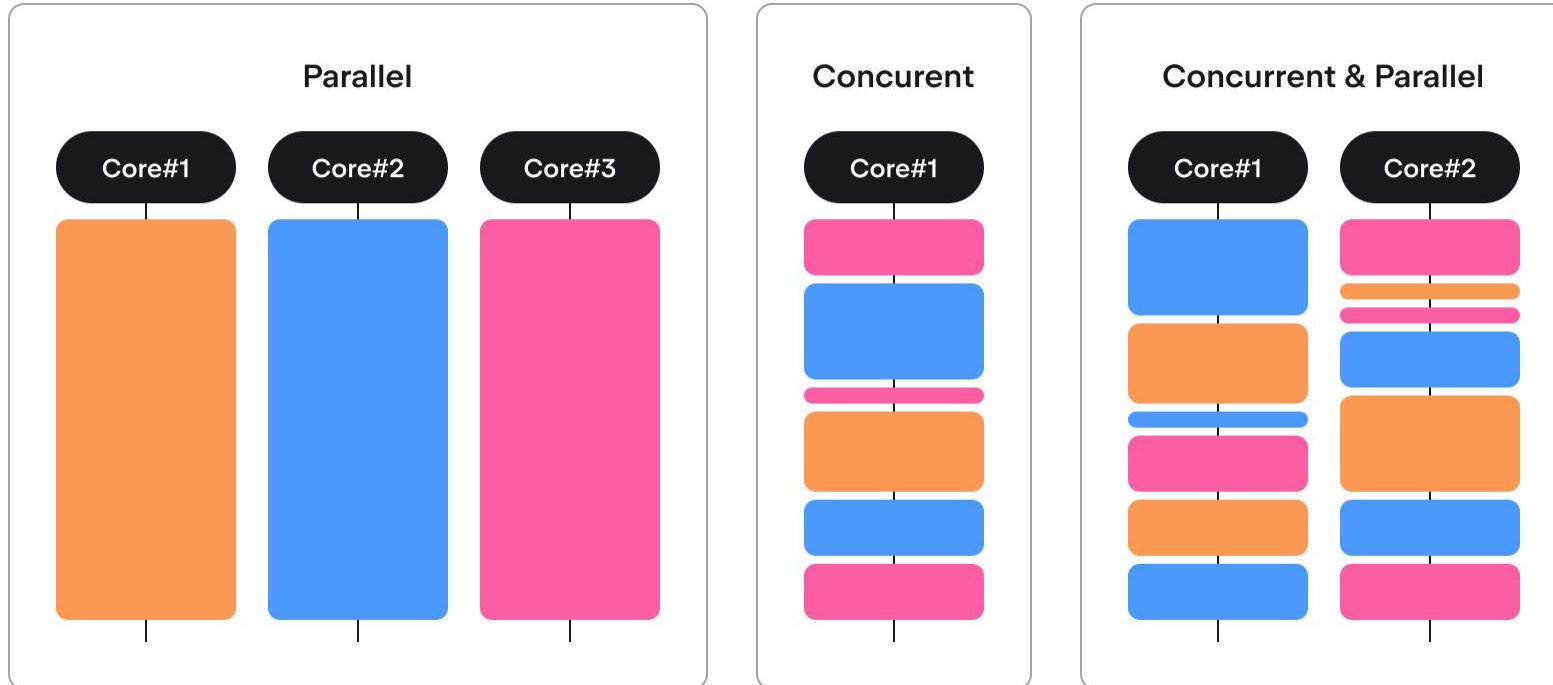
According to Wikipedia:

- **Parallel** computing is a type of computing “in which many calculations or processes are carried out **simultaneously**”.
- **Concurrent** computing is a form of computing in which several computations are executed **concurrently** – in overlapping time periods – instead of sequentially.
- It is possible to have parallelism without concurrency, and concurrency without parallelism.

Motivation

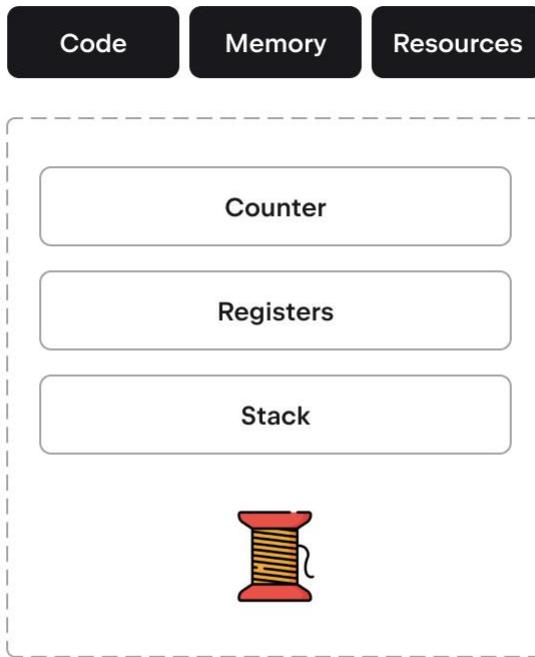
- Faster runtime
- Improved responsiveness

Parallelism vs concurrency

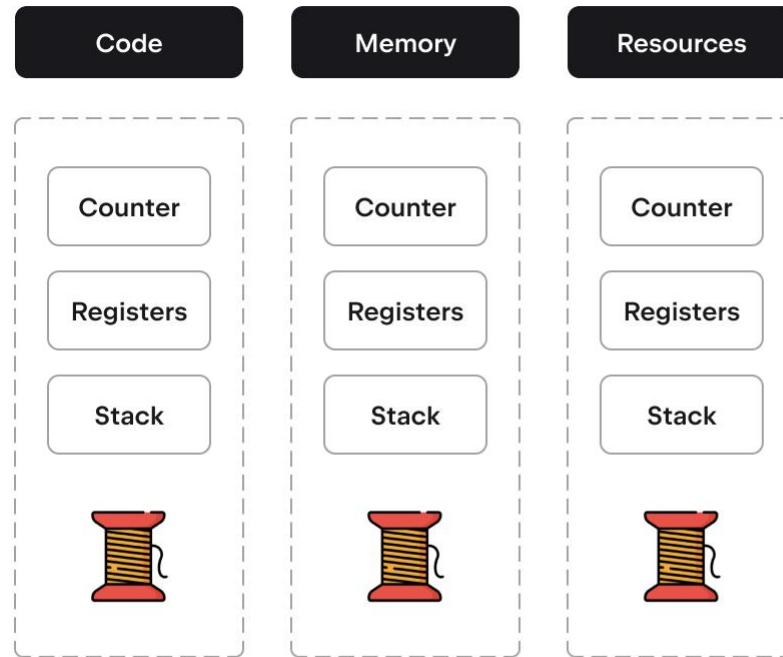


Concurrency: processes vs threads

Single-threaded process

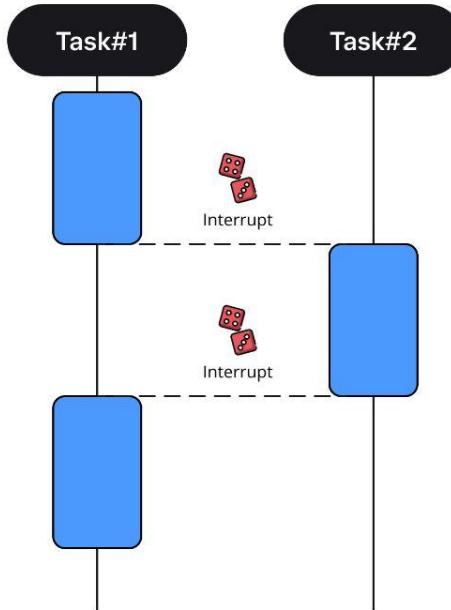


Multi-threaded process

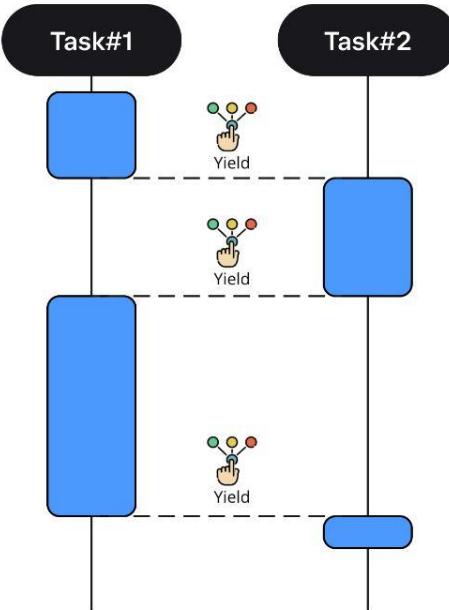


Preemptive vs cooperative scheduling

Preemptive: OS interrupts tasks



Cooperative: task yields control



Parallel and concurrent Programming in the JVM

- The JVM has its own scheduler
 - It is independent from the OS scheduler
 - A JVM thread \neq an OS thread
 - \Rightarrow Multithreaded JVM apps can run on a single-threaded OS
- (DOS) JVM threads are either daemons or user threads.
- The app stops when all **user threads** are done.
- The JVM does not wait for daemon threads to finish.

Parallel programming in the JVM

2 Java packages

- `java.lang` contains basic primitives: `Runnable`, `Thread`, etc
- `java.util.concurrent` contains synchronization primitives and concurrent data structures

Kotlin package

- `kotlin.concurrent` — Wrappers and extensions for Java classes

Throwback: Single abstract method interfaces

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

Interface with a single method. We can instantiate it with a lambda.

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

Ways to create threads

You can inherit from the Thread class, which also implements Runnable.

```
class MyThread : Thread() {  
    override fun run() {  
        println("${currentThread()} is running")  
    }  
}  
  
fun main() {  
    val myThread = MyThread()  
    myThread.start()  
}
```

run vs start

Never call Thread.run()!

run will execute on *your* thread, while start will create a new thread where run will be executed.

```
fun main() {  
    val myThread1 = MyThread()  
    myThread1.start() // OK  
    val myThread2 = MyThread()  
    myThread2.run() // Current thread gets blocked  
}
```

Ways to create threads

You can implement the `Runnable` interface and pass it to a thread. You can pass the same `Runnable` to several threads.

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

Ways to create threads

Kotlin has an even simpler way to create threads, but under the hood the same old thread is created and started.

```
import kotlin.concurrent.thread

fun main() {
    val kotlinThread = thread {
        println("I start instantly, but you can pass an option to start me later")
    }
}
```

This is the preferable way to create threads.

Thread properties

A thread's properties cannot be changed after it is started.

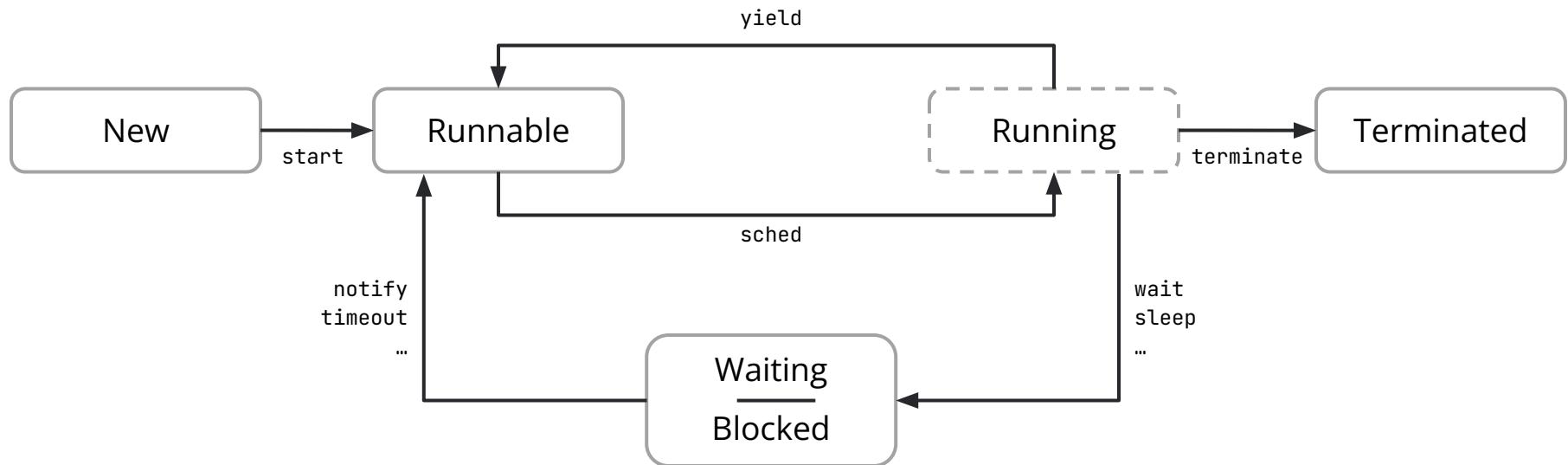
Main properties of a thread:

- `id`: Long — This is the thread's identifier
- `name`: String
- `priority`: Int — This can range from 1 to 10, with a larger value indicating higher priority
- `daemon`: Boolean
- `state`: Thread.state
- `isAlive`: Boolean

State of a thread

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

State of a thread



Ways to manipulate a thread's state

- `val myThread = thread { ... }` — Creates a new thread
- `myThread.start()` — Starts a thread
- `myThread.join()` — Causes the current thread to wait for another thread to finish
- `sleep(...)` — Puts the current thread to sleep
- `yield()` — *Tries* to step back
- `myThread.interrupt()` — *Tries* to interrupt a thread
- `myThread.isInterrupted()` — Checks whether thread was interrupted
- `interrupted()` — Checks and clears the interruption flag

sleep, join, yield, interrupt

- The `sleep` and `yield` methods are only applicable to the current thread, which means that you cannot suspend another thread.
- All blocking and waiting methods can throw `InterruptedException`

Classic worker

```
class ClassicWorker : Runnable {  
    override fun run() {  
        try {  
            while (!Thread.interrupted()) {  
                // do stuff  
            }  
        } catch (e: InterruptedException) {} // absolutely legal empty catch block  
    }  
}
```

Parallelism and shared memory: Examples of problematic interleaving

Parallel threads have access to the same shared memory.

This often leads to problems that cannot arise in a single-threaded environment.

```
class Counter {  
    private var c = 0  
  
    fun increment() {  
        c++  
    }  
    fun decrement() {  
        c--  
    }  
    fun value(): Int {  
        return c  
    }  
}
```

Both operations on c are single, simple statements.

However, even simple statements can be translated into multiple steps by the virtual machine, and those steps can be interleaved.

Parallelism and shared memory: Examples of problematic interleaving

Parallel threads have access to the same shared memory.

This often leads to problems that cannot arise in a single-threaded environment.

```
class Counter {  
    private var c = 0  
  
    fun increment() {  
        c++  
    }  
    fun decrement() {  
        c--  
    }  
    fun value(): Int {  
        return c  
    }  
}
```

Suppose both Thread#1 and Thread#2 invoke increment at the same time. If the initial value of c is 0, their interleaved actions might follow this sequence:

- T#1: Read value 0 from c.
- T#2: Read value 0 from c.
- T#1: Increment value — result is 1.
- T#1: Write result 1 to c.
- T#2: Increment value — result is 1.
- T#2: Write result 1 to c.

Synchronization mechanisms

- Mutual exclusion, such as Lock and the synchronized keyword
- Concurrent data structures and synchronization primitives
- Atomics, which work directly with shared memory (**DANGER ZONE**)

Locks

```
class LockedCounter {  
  
    private var c = 0  
  
    private val lock = ReentrantLock()  
  
    fun increment() {  
        lock.withLock { c++ }  
    }  
  
    // same for other methods  
    ...  
}
```

The lock interface

- `lock.lock()` — Acquires the lock
- `lock.tryLock()` — Tries to acquire the lock
- `lock.unlock()` — Releases the lock
- `lock.withLock { }` — Executes a lambda with the lock held (has try/catch inside)
- `lock.newCondition()` — Creates a *condition variable* associated with the lock

Conditions

```
class PositiveLockedCounter {
    private var c = 0
    private val lock = ReentrantLock()
    private val condition = lock.newCondition()

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

    fun decrement() {
        lock.withLock {
            while (c == 0) {
                condition.await()
            }
            c--
        }
    }

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

A condition allows a thread holding a lock to *wait* until another thread *signals* it about a certain event. Internally, the `await` method releases the associated lock upon call, and acquires it back before finally returning it again.

The ReentrantLock class

- `ReentrantLock` – Allows the lock to be acquired multiple times by the same thread
- `lock.getHoldCount()` – Gets the number of holds on this lock by the current thread
- `lock.queuedThreads()` – Gets a collection of the threads waiting on this lock
- `lock.isFair()` – Checks the *fairness* of the lock

The synchronized statement

In the JVM, every object has an *intrinsic* lock associated with it (aka a *monitor*).

```
class Counter {  
    private var c = 0  
  
    fun increment() {  
        synchronized(this) { c++ }  
    }  
    ...  
}
```

Synchronized method

Java

```
public class SynchronizedCounter {  
    private int c = 0;  
  
    public synchronized void increment() {  
        c++;  
    }  
  
    ...  
}
```

Kotlin

```
class SynchronizedCounter {  
    private var c = 0  
  
    @Synchronized  
    fun increment() {  
        c++  
    }  
  
    ...  
}
```

The ReadWriteLock class

ReadWriteLock allows multiple readers to access a resource concurrently but only lets a single writer modify it.

- `rwLock.readLock()` – Returns the read lock
- `rwLock.writeLock()` – Returns the write lock
- `rwLock.read { ... }` – Executes lambda under a read lock
- `rwLock.write { ... }` – Executes lambda under a write lock

The ReadWriteLock Class

```
class PositiveLockedCounter {  
    private var c = 0  
    private val rwLock = ReadWriteReentrantLock()  
  
    fun increment() {  
        rwLock.write { c++ }  
    }  
  
    fun decrement() {  
        rwLock.write { c-- }  
    }  
  
    fun value(): Int {  
        return rwLock.read { c }  
    }  
}
```

Concurrent blocking collections

`java.util.concurrent` is a Java package that implements both blocking and non-blocking concurrent collections, such as:

- `SynchronousQueue` – One-element rendezvous channel
- `ArrayBlockingQueue` – Fixed-capacity queue
- `LinkedBlockingQueue` – Unbounded blocking queue
- `PriorityBlockingQueue` – Unbounded blocking priority queue

Concurrent non-blocking collections

`java.util.concurrent` is a Java package that implements both blocking and non-blocking concurrent collections, such as:

- `ConcurrentLinkedQueue` – Non-blocking unbounded queue
- `ConcurrentLinkedDeque` – Non-blocking unbounded deque
- `ConcurrentHashMap` – Concurrent unordered hash-map
- `ConcurrentSkipListMap` – Concurrent sorted hash-map

Synchronization primitives

`java.util.concurrent` also implements concurrent data structures and synchronization primitives.

- Exchanger – Blocking exchange
- Phaser – Barrier synchronization

Java Memory Model: Weak behaviors

There are no guarantees when it comes to ordering!

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

Possible outputs:

- 0, 0
- 0, 1
- 1, 1
- 1, 0

Java Memory Model: Weak behaviors

There are no guarantees when it comes to progress!

```
class ProgressTest {  
    var flag = false  
    fun test() {  
        thread {  
            while (!flag) {}  
            println("I am free!")  
        }  
        thread { flag = true }  
    }  
}
```

Possible outputs:

- "I am free!"
- ...
- ...
- ...
- hang!

Java Memory Model: Weak behaviors

There are no guarantees when it comes to progress!

```
class ProgressTest {  
    var flag = false  
    fun test() {  
        thread {  
            while (true) {}  
            println("I am free!")  
        }  
        thread { flag = true }  
    }  
}
```

Possible outputs:

- "I am free!"
- ...
- ...
- ...
- hang!

JMM: Data-Race-Freedom Guarantee

But what does JMM guarantee?

Well-synchronized programs have **simple interleaving semantics**.

JMM: Data-Race-Freedom Guarantee

But what does JMM guarantee?

Well-synchronized programs have **simple interleaving semantics**.

Well-synchronized = Data-race-free

Simple interleaving semantics = Sequentially consistent semantics

Data-race-free programs have **sequentially consistent** semantics

JMM: Volatile fields

Volatile fields can be used to restore **sequential consistency**.

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

```
class ProgressTest {
    @Volatile var flag = false
    fun test() {
        thread {
            while (!flag) {}
            println("I am free!")
        }
        thread { flag = true }
    }
}
```

JMM: Volatile fields

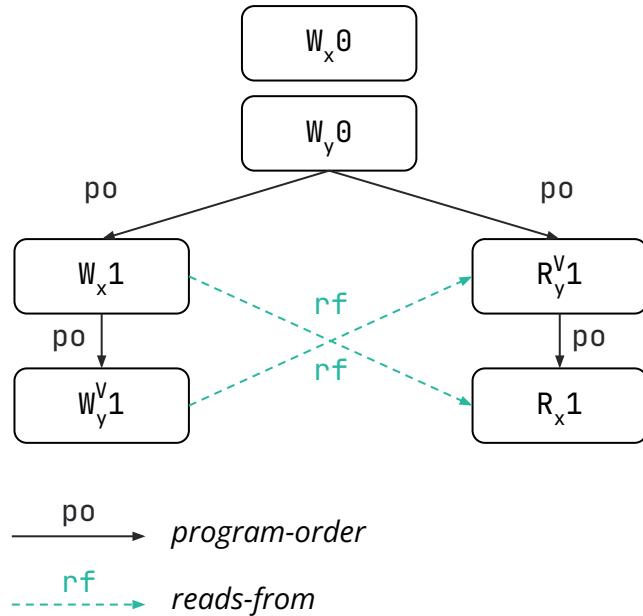
Volatile variables can be used for synchronization.

```
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")  
        }  
    }  
}
```

How do we know there is enough synchronization?

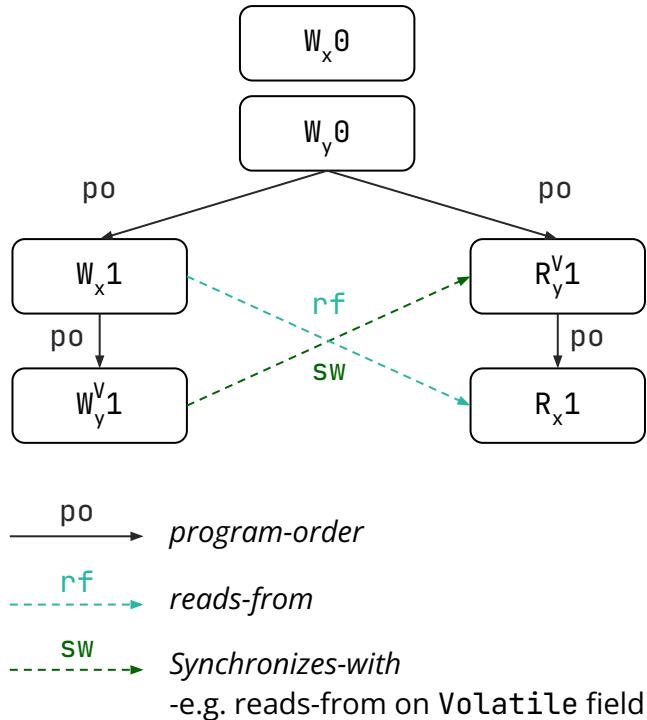
JMM: Happens-before relation

```
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")  
        }  
    }  
}
```



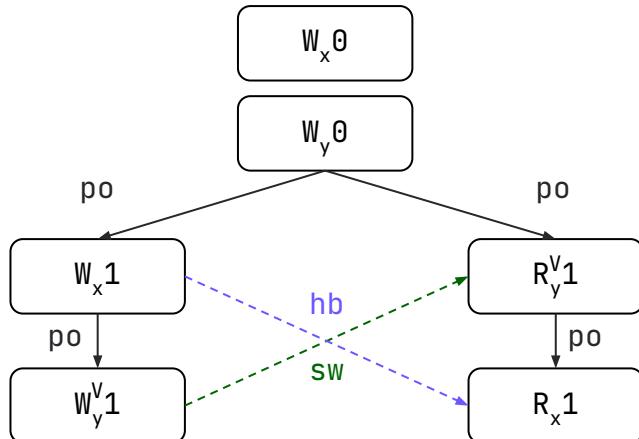
JMM: Happens-before relation

```
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")  
        }  
    }  
}
```



JMM: Happens-before relation

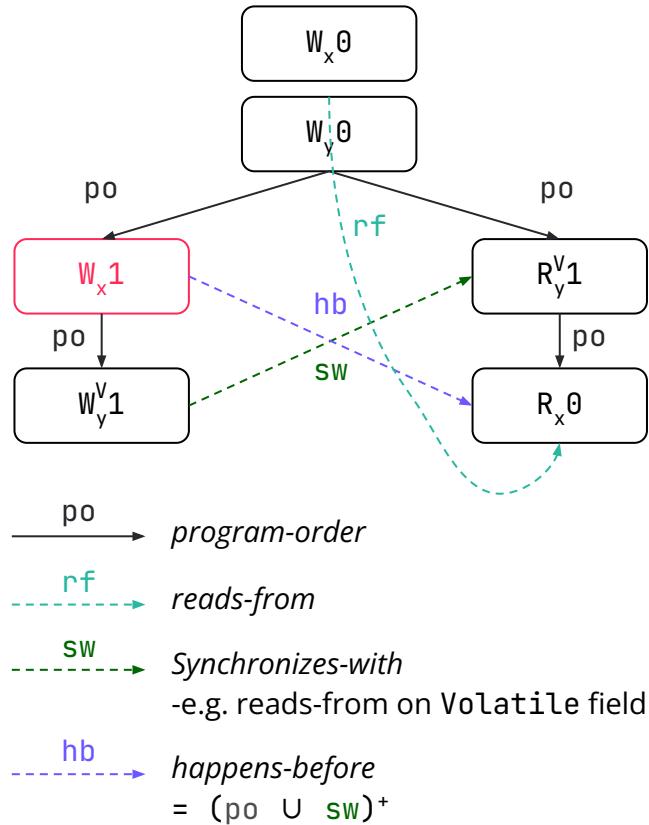
```
class OrderingTest {  
    var x = 0  
    @Volatile var y = 0  
    fun test() {  
        thread {  
            x = 1  
            y = 1  
        }  
        thread {  
            val a = y  
            val b = x  
            println("$a, $b")  
        }  
    }  
}
```



- $\xrightarrow{\text{po}}$ program-order
- $\xrightarrow{\text{rf}}$ reads-from
- $\xrightarrow{\text{sw}}$ Synchronizes-with
-e.g. reads-from on Volatile field
- $\xrightarrow{\text{hb}}$ happens-before
 $= (\text{po} \cup \text{sw})^+$

JMM: Happens-before relation

```
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")  
        }  
    }  
}
```



JMM: Synchronizing actions

- Read and write for volatile fields
- Lock and unlock
- Thread run and start, as well as finish and join

JMM: DRF-SC again

Two events form a data race if:

- Both are memory accesses to the same field.
- Both are **plain** (non-atomic) accesses.
- At least one of them is a `write` event.
- They are not related by *happens before*.

Data-race-free programs have **sequentially consistent** semantics

A program is data-race-free if, for every possible execution of this program, no two events form a data race.

JMM: Atomicics

But what about atomic operators on shared variables?

```
class Counter {  
    private val c = AtomicInteger()  
  
    fun increment() {  
        c.incrementAndGet()  
    }  
  
    fun decrement() {  
        c.decrementAndGet()  
    }  
  
    fun value(): Int {  
        return c.get()  
    }  
}
```

JMM: Atomics

Atomic classes from package the `java.util.concurrent.atomic` package:

- `AtomicInteger`
- `AtomicLong`
- `AtomicBoolean`
- `AtomicReference`

And their array counterparts:

- `AtomicIntegerArray`
- `AtomicLongArray`
- `AtomicReferenceArray`

JMM: Atomics

- `get()` – Reads a value with volatile semantics
- `set(v)` – Writes a value with volatile semantics
- `getAndSet(v)` – Atomically exchanges a value
- `compareAndSet(e, v)` – Atomically compares a value of atomic variable with the expected value, e, and if they are equal, replaces content of atomic variable with the desired value, v; returns a boolean indicating success or failure.
- `compareAndExchange(e, v)` – Atomically compares a value with an expected value, e, and if they are equal, replaces with the desired value, v; returns a read value.
- `getAndIncrement()`, `addAndGet(d)`, etc – Perform Atomic arithmetic operations for • numeric atomics (`AtomicInteger`, `AtomicLong`).
- ...

JMM: Atomicics

Methods of atomic classes:

- ...
- `getXXX()`
- `setXXX(v)`
- `weakCompareAndSetXXX(e, v)`
- `compareAndExchangeXXX(e, v)`

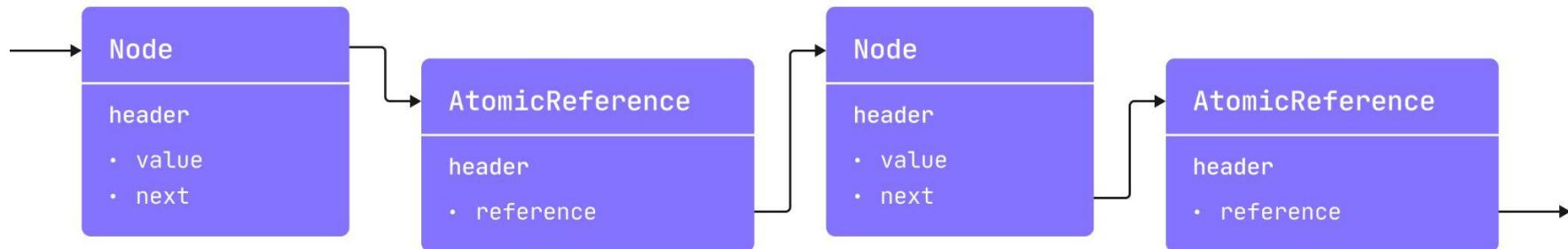
In these cases, XXX is an access mode: Acquire, Release, Opaque, Plain

You can learn more about Java Access Modes here:

<https://gee.cs.oswego.edu/dl/html/j9mm.html>

JMM: Atomics Problem

```
class Node<T>(val value: T) {  
    val next = AtomicReference<Node<T>>()  
}
```



JMM: Atomic field updaters

Use `AtomicXXXFieldUpdater` classes to directly modify volatile fields:

```
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 {
        return updater.get(this)
    }
}
```

Starting from JDK9, there is also the `VarHandle` class, which serves a similar purpose.

Kotlin: AtomicFU

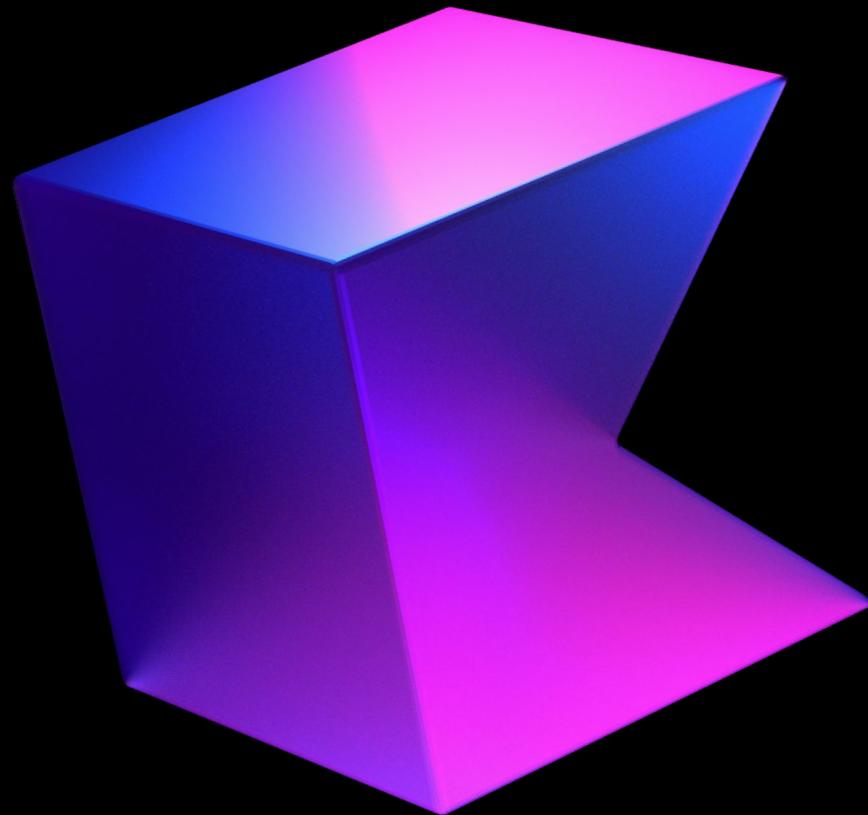
The AtomicFU library is a recommended way to use atomic operations in Kotlin:

<https://github.com/Kotlin/kotlinx-atomicfu>

```
class Counter {  
    private val c = atomic(0)  
    fun increment() {  
        c += 1  
    }  
    fun decrement() {  
        c -= 1  
    }  
    fun value(): Int {  
        return c.value  
    }  
}
```

- It provides `AtomicXXX` classes with API similar to Java atomics.
- Under the hood *compiler plugin* replaces usage of atomics to `AtomicXXXFieldUpdater` or `VarHandle`.
- It also provides convenient extension functions, e.g.
`c.update { it + 1 }`

Thanks!



@kotlin



Asynchronous Programming in Kotlin

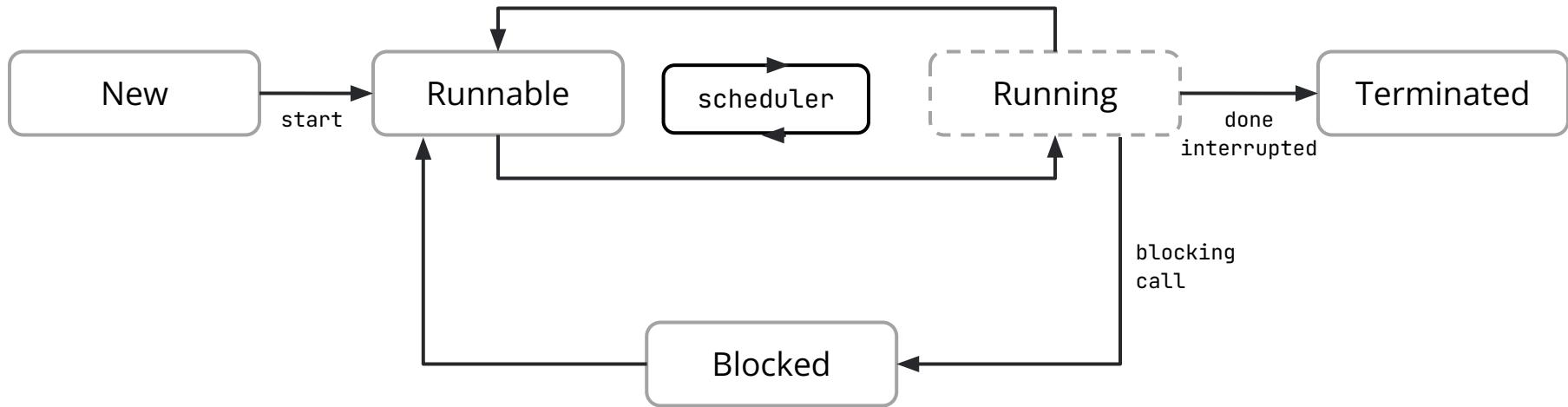


@kotlin

What we'll cover

- Parallel and asynchronous programming
- The history of coroutines
- Kotlin coroutines
- Inside CoroutineScope
- Channels
- More

Parallel programming



Parallel programming

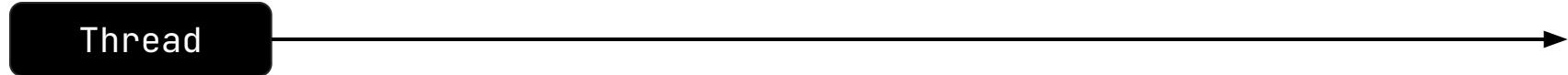
- In reality, programs (threads) spend a lot of time waiting for data to be fetched from disk, network, etc.
- The number of threads that can be launched is limited by the underlying operating system (each takes some number of MBs).
- Threads aren't cheap, as they require context switches which are costly.
- Threads aren't always available. Some platforms, such as JavaScript, do not even support them.
- Working with threads is hard. Bugs in threads (which are extremely difficult to debug), race conditions, and deadlocks are common problems we suffer from in multi-threaded programming.
- Threads terminating due to exceptions is a problem that deserves to be a separate point.

An example

```
fun postItem(item: Item) {  
    val token = preparePost()  
    val post = submitPost(token, item)  
    processPost(post)  
}  
  
fun preparePost(): Token { // requestToken  
    // makes a request and consequently blocks the execution thread  
    return token  
}
```

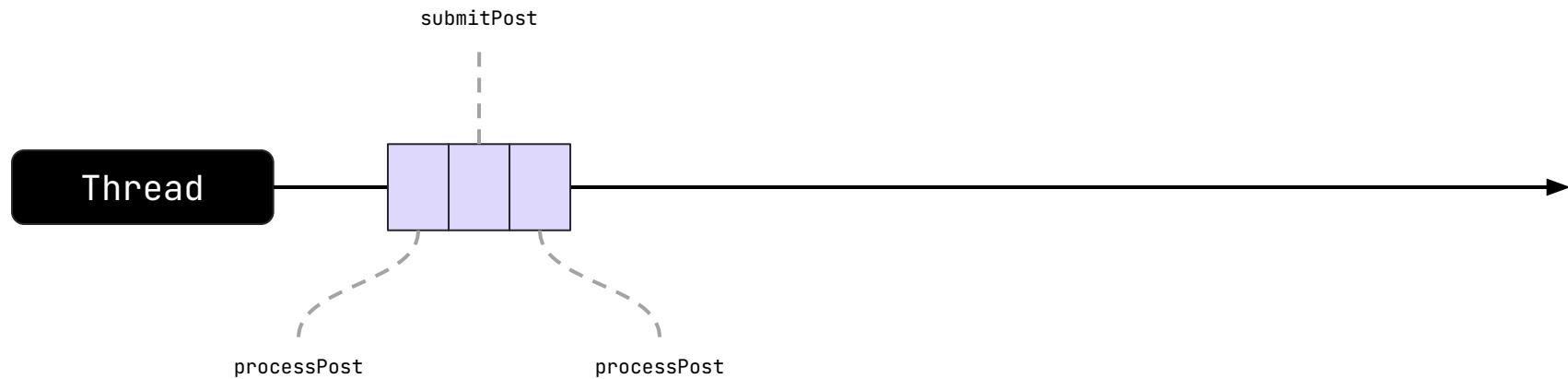
An example

How this code gets executed on a single thread



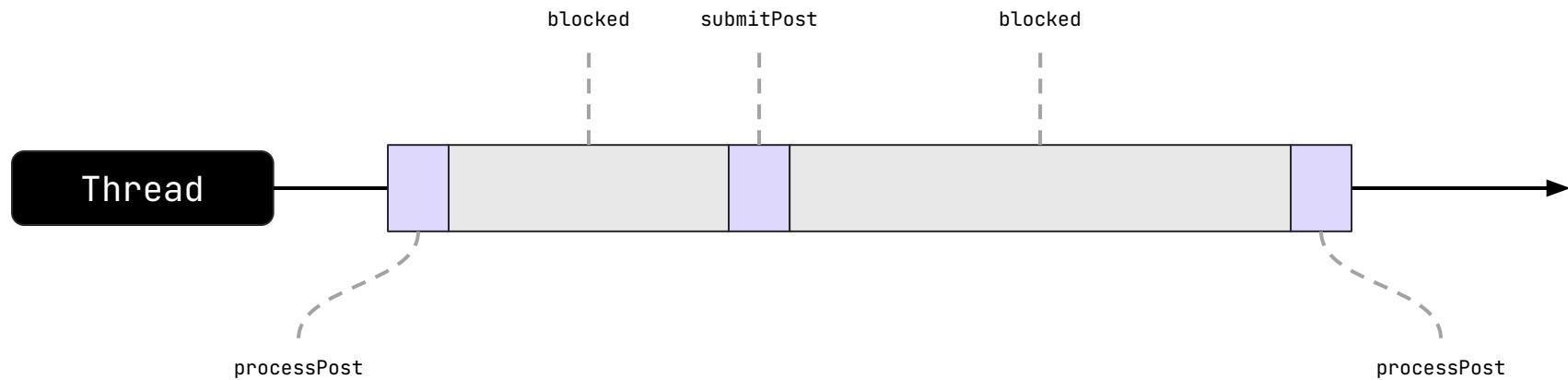
An example

What we want



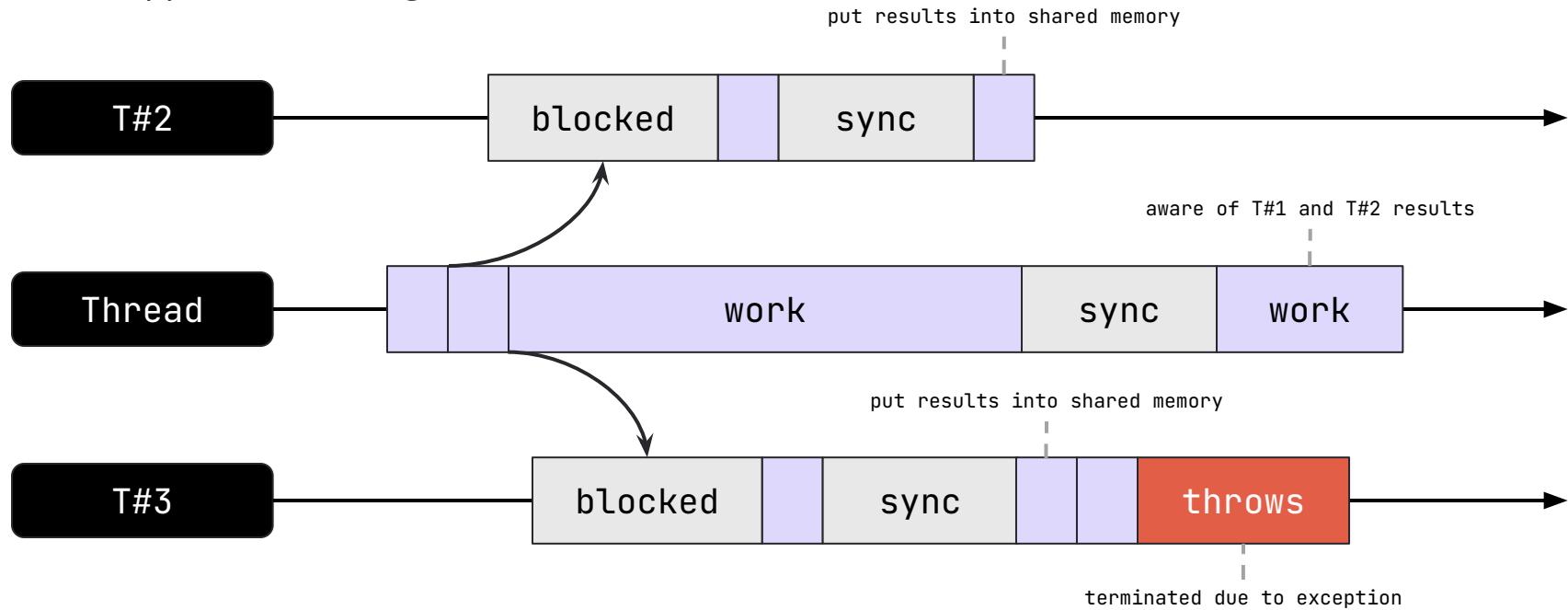
An example

What we get



An example

What happens when we go multi-threaded



Asynchronous Programming

Continuation passing style

```
fun preparePostAsync(callback: (Token) → Unit) {  
    // make request and return immediately  
    // arrange callback to be invoked later  
}
```

With callbacks, the idea is to pass one function as a parameter to another function and have this one invoked once the process has completed.

```
fun postItem(item: Item) {  
    preparePostAsync { token →  
        submitPostAsync(token, item) { post →  
            processPost(post)  
        }  
    }  
}
```

Continuation passing style

```
fun postItem(item: Item) {  
    preparePostAsync { token →  
        submitPostAsync(token, item) {  
            post → processPostAsync(post) {  
                ...  
            }  
        }  
    }  
}
```

- The } ladder is the ~~Stairway to Heaven~~ Highway to “Callback Hell”.
- Where is the error handling?
- **Callbacks are not asynchronous “by nature”.**

Futures, promises, and other approaches

`Promise<T>` encapsulates the callback.

```
fun preparePostAsync(): Promise<Token> {
    // makes request and returns a promise that is completed later
    return promise
}
```

Futures, promises, and other approaches

Promise<T> encapsulates the callback.

```
fun postItem(item: Item) {  
    preparePostAsync()  
        .thenCompose { token → submitPostAsync(token, item) }  
        .thenAccept { post → processPost(post) }  
        ...  
}
```

- This model differs from the typical top-down imperative approach.
- There are different APIs, which vary across libraries, frameworks, and platforms.
- It employs the Promise<T> return type instead of the actual we need.
- Each thenCompute/Accept/Handle creates a new object.
- Error handling can be complicated.

Kotlin coroutines

`suspend` – a keyword in Kotlin marking suspendable function.

```
suspend fun submitPost(token: Token, item: Item): Post {
```

```
    ...
```

```
}
```

```
suspend fun postItem(item: Item) {
```

→ `val token = preparePost()`

→ `val post = submitPost(token, item)`

```
    processPost(post)
```

```
}
```

This looks and feels sequential, allowing you to focus on the logic of your code.

→ marks suspension points in IntelliJ IDEA.

The history of coroutines

History and definition

- Melvin Conway coined the term “coroutine” in 1958 for his **assembly** program.
- Coroutines were first introduced as a language feature in Simula'67 with the **detach** and **resume** commands.
- A coroutine can be thought of as an instance of a **suspendable** computation, i.e. one that can suspend at some point and later resume execution, possibly even on another thread.
- Coroutines calling each other (and passing data back and forth) can form the machinery for **cooperative multitasking**.
- Go'09, C#'12, Kotlin'17, C++'20, OpenJDK, Project Loom.

Kotlin

Coroutines came to Kotlin in version 1.1, and they became stable in version 1.3.

- `suspend` – A keyword for marking suspendable functions.
- `kotlin.coroutines` – A tiny part of the standard library.
- `kotlinx.coroutines` – A library with all the necessary functionality. It is not a part of the standard library, meaning there are no additional requirements for the host platform, facilitating multiplatform development.

A coroutine is an instance of suspendable computation. It is conceptually similar to a thread in the sense that it takes a block of code to run and has a similar life-cycle. It is created and started, but it is not bound to any particular thread. It may suspend its execution in one thread and resume in another one. Moreover, like a future or a promise, it can complete with some result (which is either a value or an exception).

Kotlin coroutines

Under the hood

The compiler turns your suspend function:

```
suspend fun submitPost(token: Token, item: Item): Post {...}
```

Into:

```
fun submitPost(token: Token, item: Item, cont: Continuation<Post>) {...}
```

Where:

```
public interface Continuation<in T> {  
    public val context: CoroutineContext  
    public fun resumeWith(result: Result<T>)  
}
```

Continuation<in T> ~ Generic callback

Under the hood

Code with suspending calls:

```
// code inside postItem
// suspend call 0
val token = preparePost()
// suspend call 1
val post = submitPost(token, item)
// suspend call 2
processPost(post)
```

Is compiled into (simplified version)

→

Under the hood

Code with suspending calls:

```
// code inside postItem  
// suspend call 0  
val token = preparePost()  
// suspend call 1  
val post = submitPost(token, item)  
// suspend call 2  
processPost(post)
```

Is compiled into (simplified version)

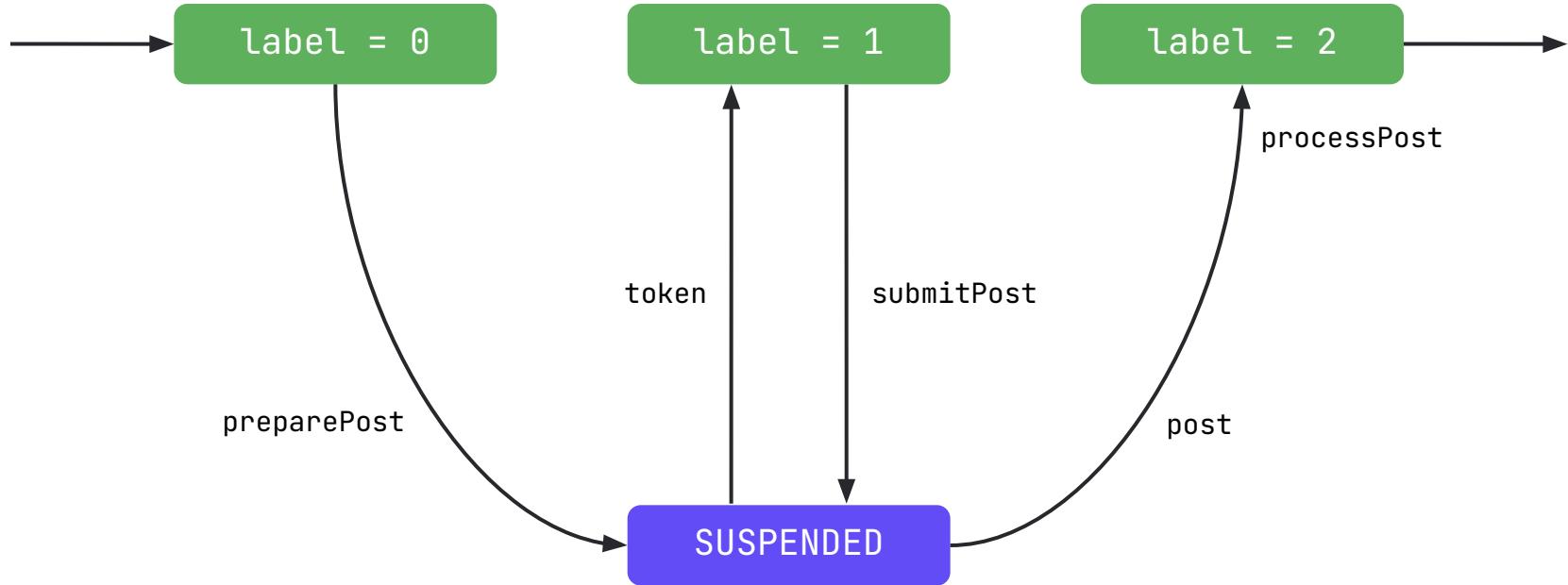
→

~~A large switch???~~ A state machine!

Each label marks a suspension point.

```
// some code here, continuation is created  
when(continuation.label) {  
    0 → { // suspend call 0  
        cont.label = 1;  
        preparePost(cont);  
    }  
    1 → { // suspend call 1  
        val token = prevResult;  
        cont.label = 2;  
        submitPost(token, item, cont);  
    }  
    2 → { // suspend call 2  
        val post = prevResult;  
        processPost(post, cont);  
    }  
}  
// more code here
```

State of a coroutine



Practice

Now we can finally post items without blocking the execution thread!

```
fun nonBlockingItemPosting(...) {  
    ...  
    postItem(item)  
}
```

Practice

Now we can finally post items without blocking the execution thread!

```
fun nonBlockingItemPosting(...) {  
    ...  
    postItem(item)  
}
```

The suspending function `postItem` should be called only from a coroutine or another suspending function.

One cannot just walk into a suspending function.

Inside CoroutineScope

Practice

`suspend` functions can be called from other `suspend` functions or within `CoroutineScope`.

```
fun main() = runBlocking { // this: CoroutineScope
    launch { // launch a new coroutine and continue
        delay(1000L) // non-blocking delay for 1 second (default time unit is ms)
        ↪       println("World!") // print after delay
    }
    print("Hello ") // main coroutine continues while the previous one is delayed
}
```

HOFs like `launch` are called *coroutine builders*.

Sophisticated practice

```
val jobs: List<Job> = List(1_000_000) {  
    launch(Dispatchers.Default + CoroutineName("#$it")  
        + CoroutineExceptionHandler { context, error →  
            println("${context[CoroutineName] ?.name}: $error")  
        }, // CoroutineContext  
        CoroutineStart.LAZY // do not start instantly  
    ) {  
        delay(Random.nextLong(1000))  
        if (it % 10 == 0) { throw Exception("No comments") }  
        println("Hello from coroutine $it!")  
    }  
}  
  
jobs.forEach { it.start() }
```

Now we are going to cover all of this step by step.

Scope and context

```
public interface CoroutineScope {  
    public val coroutineContext: CoroutineContext  
}
```

Easy, isn't it?

Scope and context

```
public interface CoroutineScope {  
    public val coroutineContext: CoroutineContext  
}  
  
public interface CoroutineContext {  
    public operator fun <E : Element> get(key: Key<E>): E?  
    ...  
}  
  
public interface Element : CoroutineContext {  
    public val key: Key<*>  
    ...  
}  
}
```

You can think of context like `Map<Key<Element>, Element>`

Inside CoroutineScope

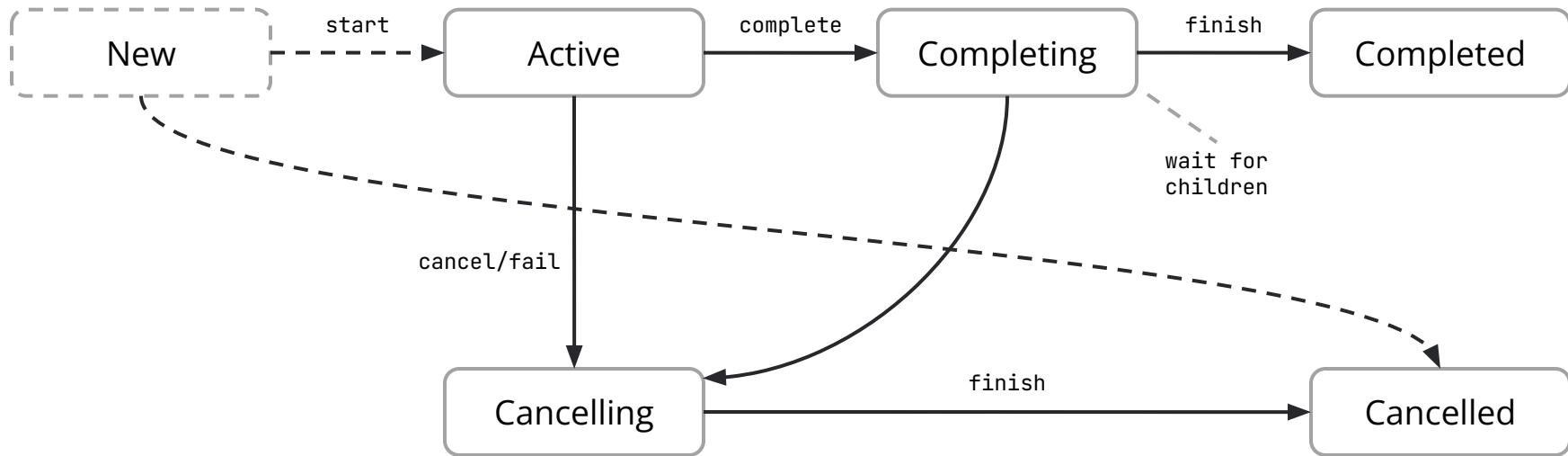
Job

Job

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

- A Job is work that is executed in the background.
- It is a cancellable work item with a life-cycle that culminates in its completion.
- Jobs can be arranged into parent-child hierarchies.
- A child's failure immediately cancels its parent along with all its other children. This behavior can be customized using SupervisorJob.

Job States



Job states

state	isAlive	isCompleted	isCancelled
New	false	false	false
Active	true	false	false
Completing	true	false	false
Cancelling	false	false	true
Cancelled	false	true	true
Completed	false	true	false

Inside CoroutineScope

Dispatchers

Dispatchers

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

- `Dispatchers.Default` – A shared pool of background threads, at least 2, depending on the default number of CPU cores. It is an appropriate choice for compute-intensive coroutines.
- `Dispatchers.IO` – A shared pool of on-demand created threads and is designed for offloading IO-intensive blocking operations (such as file/socket IO).

Dispatchers

- `Dispatchers.Main` – A dispatcher that is confined to the `Main` thread operating with UI objects. Usually single-threaded, it is not present in `core`, but is instead provided by packages like `android`, `swing`, etc.
- `Dispatchers.Unconfined` – The unconfined dispatcher should not normally be used in code.
- Private thread pools can be created with `newSingleThreadContext` and `newFixedThreadPoolContext`. (Both are `@ExperimentalCoroutinesApi`.)
- A view of a dispatcher with the guarantee that no more than `parallelism` coroutines are executed at the same time can be created via:

```
// method of public abstract class CoroutineDispatcher
@ExperimentalCoroutinesApi
public open fun limitedParallelism(parallelism: Int): CoroutineDispatcher
{ ... }
```

Dispatchers

- An arbitrary ExecutorService can be converted into a dispatcher with the asCoroutineDispatcher extension function.

```
interface ExecutorService : Executor {  
    fun execute(command: Runnable) // Executor is a SAM with this method  
    ...  
}  
  
val myExecutorService: ExecutorService = ...  
val myDispatcher = myExecutorService.asCoroutineDispatcher()
```

A peek under the hood

```
internal class GlobalQueue : LockFreeTaskQueue<Task>(singleConsumer = false)

internal class CoroutineScheduler(
    @JvmField val corePoolSize: Int, @JvmField val maxPoolSize: Int,
    @JvmField val idleWorkerKeepAliveNs: Long = ...,
    @JvmField val schedulerName: String = ...
) : Executor, Closeable {
    ...
    val globalCpuQueue = GlobalQueue()

    val globalBlockingQueue = GlobalQueue()
    ...
}
```

A peek under the hood

```
internal class CoroutineScheduler(...) : Executor, Closeable {  
    ...  
    val workers = AtomicReferenceArray<Worker?>(maxPoolSize + 1)  
  
    fun dispatch(  
        block: Runnable,  
        taskContext: TaskContext = NonBlockingContext,  
        tailDispatch: Boolean = false  
    ) {  
        ...  
    }  
}
```

A peek under the hood

```
internal inner class Worker private constructor() : Thread() {  
    ...  
    val localQueue: WorkQueue = WorkQueue()  
    var state = WorkerState.DORMANT  
    fun findTask(scanLocalQueue: Boolean): Task? {  
        // localQueue → globalBlockingQueue  
        return task ?: trySteal(blockingOnly = true)  
    }  
}
```

Inside CoroutineScope

Coroutines vs threads

Adding contexts

```
val jobs: List<Job> = List(1_000_000) {  
    launch(  
       BaseContext  
        + SupervisorJob()  
        + CoroutineName("#$it")  
        + CoroutineExceptionHandler { context, error →  
            println("${context[CoroutineName]?.name}: $error")  
        }, // launch's first argument is CoroutineContext, which is a sum here  
        ...  
    ) { ... }  
}
```

- Contexts can be added together. In this case, the rightmost value for a Key is taken as the resulting context.
- Since each Element implements CoroutineContext, this looks like a sum of elements.

Context switching

```
suspend fun preparePost(): Token = withContext(Dispatchers.IO) { ... }

// submitPost also withContext(Dispatchers.IO)

suspend fun processPost(post: Post) =
withContext(Dispatchers.Default) { ... }

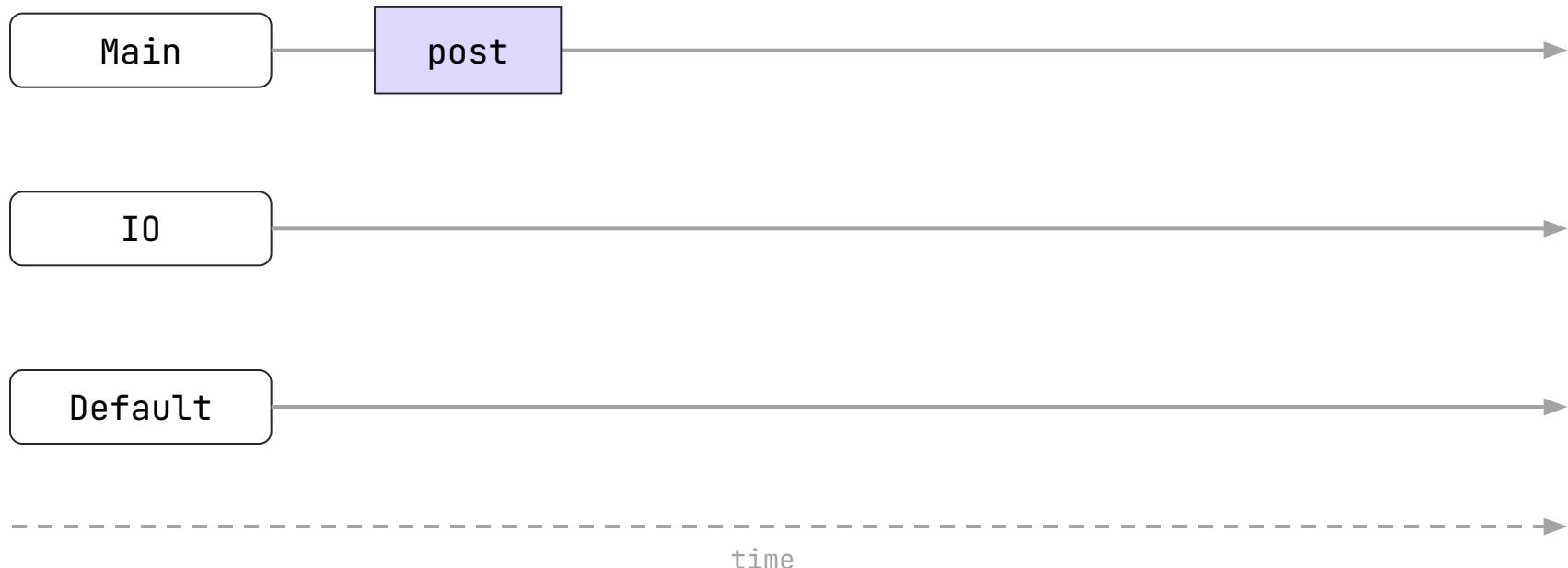
suspend fun postItem(item: Item) {
    ↗ val token = preparePost()
    ↗ val post = submitPost(token, item)
    ↗ processPost(post)
}

// somewhere in our application's code there is a View and a CoroutineScope related to it
viewScope.launch {
    postItem(someItem)
    // show the result in the UI somehow
}
```

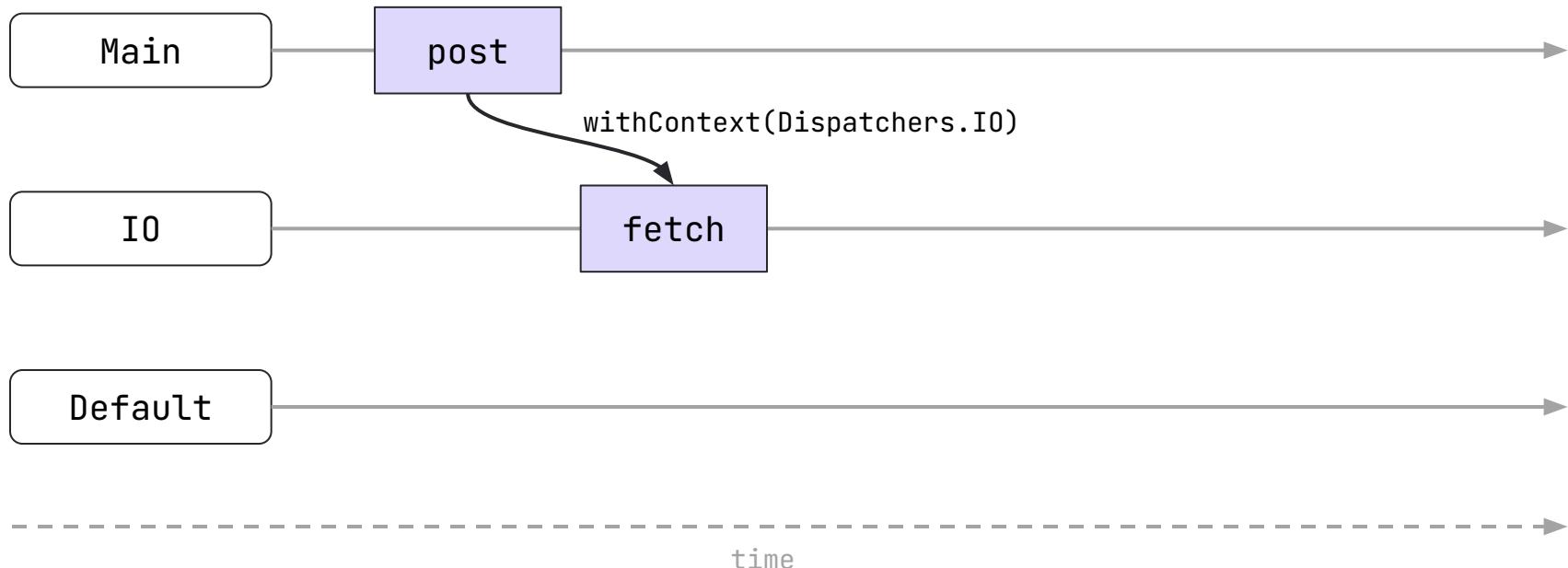
How is this actually better than threads?



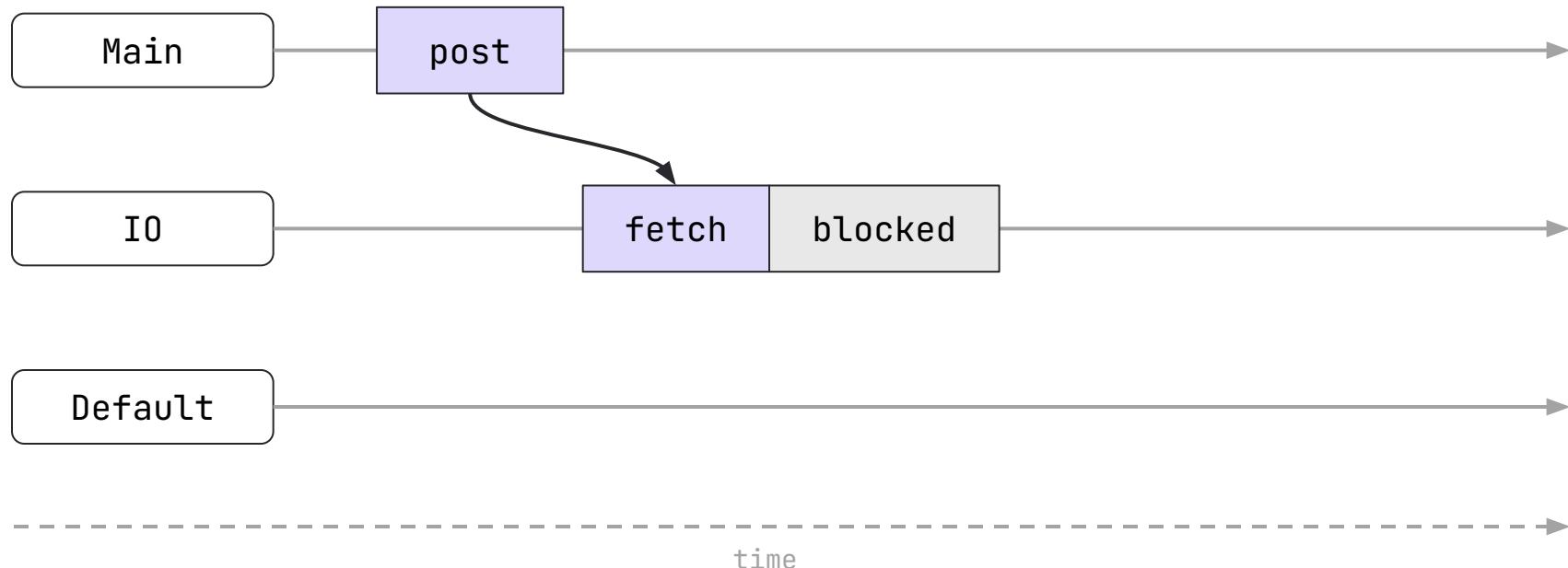
How is this actually better than threads?



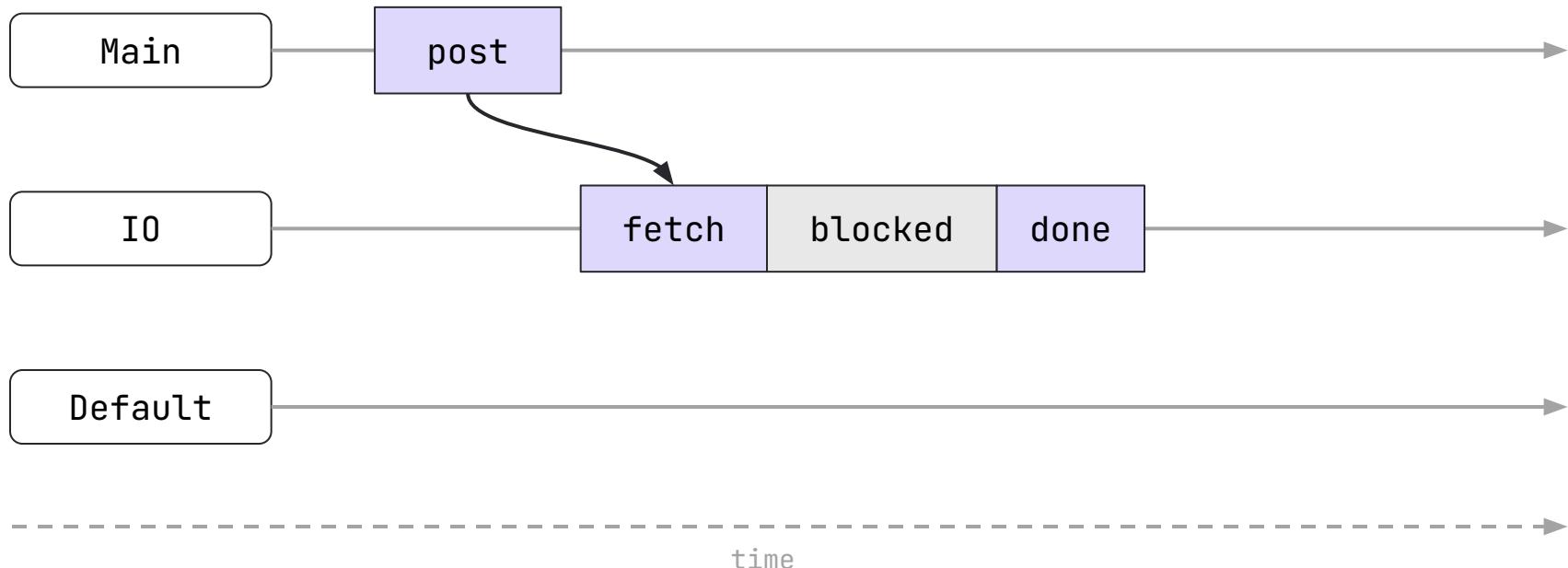
How is this actually better than threads?



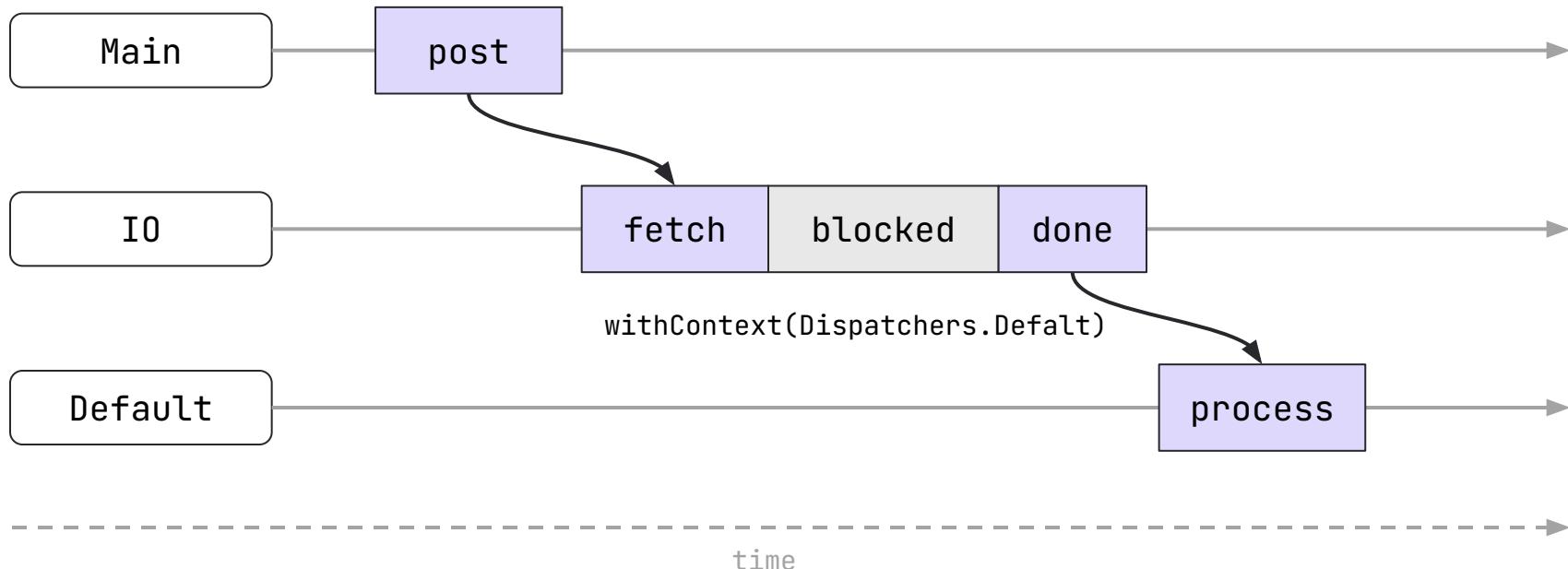
How is this actually better than threads?



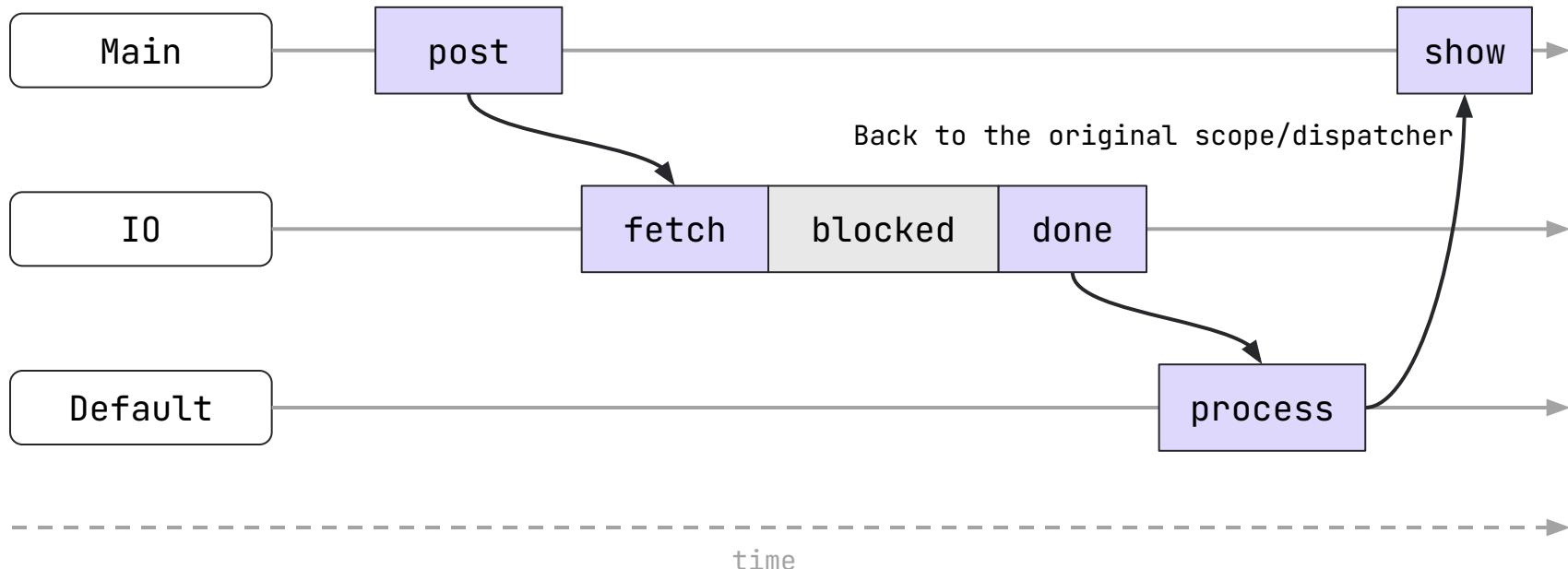
How is this actually better than threads?



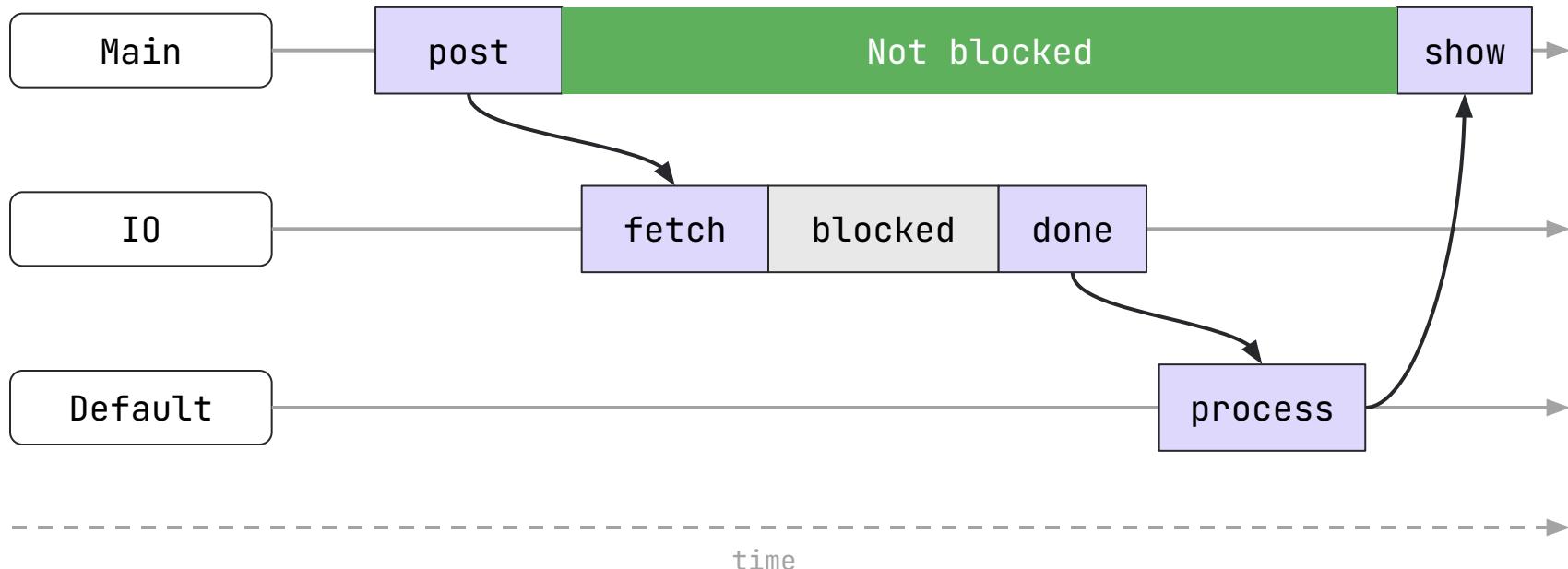
How is this actually better than threads?



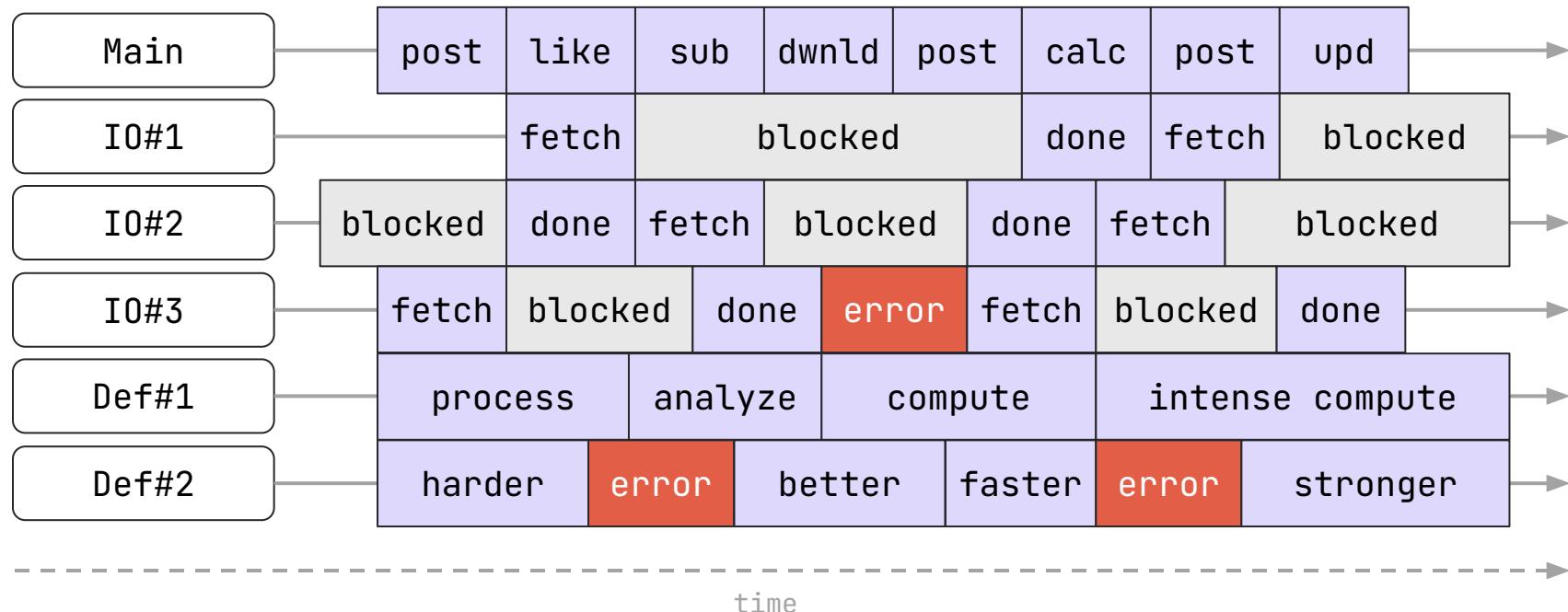
How is this actually better than threads?



How is this actually better than threads?



How is this actually better than threads?



Coroutines - fibers - threads

```
fun main(): Unit = runBlocking {
    repeat(1_000_000) { // it: Int
        delay(Random.nextLong(1000))
        println("Hello from coroutine $it!")
    }
}
```

Coroutines - fibers - threads

```
fun main(): Unit = runBlocking {
    repeat(1_000_000) { // it: Int
        delay(Random.nextLong(1000))
        println("Hello from coroutine $it!")
    }
}
```

WRONG!

The default behavior is sequential, you have to ask for concurrency.

Coroutines - fibers - threads

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

Coroutines are like light-weight threads.

Coroutines - fibers - threads

```
fun main(): Unit {  
    repeat(1_000_000) { // it: Int  
        thread { // new thread  
            sleep(1000L)  
            println("Hello from thread $it!")  
        }  
    }  
}
```

Coroutines - fibers - threads

```
fun main(): Unit {  
    repeat(1_000_000) { // it: Int  
        thread { // new thread  
            sleep(1000L)  
            println("Hello from thread $it!")  
        }  
    }  
}
```

Exception in thread "main" java.lang.OutOfMemoryError: unable to create native thread: possibly out of memory or process/resource limits reached.

Coroutines - fibers - threads

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

Coroutines are like light-weight threads.

Inside Coroutine Scope

Thread switching problem

An important non-guarantee

It is not guaranteed that the coroutine is going to be resumed on the same thread, so be very careful about calling suspending function while holding any monitor.

```
val lock = ReentrantLock()

suspend fun russianRoulette() {
    lock.lock()
    ↳ pullTheTrigger()
    lock.unlock()
}
```

Unlock might happen on **another thread**.

Murphy's law: "Anything that can go wrong will go wrong."

Then unlock will throw `IllegalMonitorStateException`.

Mutual Exclusion

Mutual Exclusion \implies Mutex.

```
val mutex = Mutex() // .lock() suspends, .tryLock() does not suspend
var counter = 0

suspend fun withMutex() {
    repeat(1_000) {
        launch {
            // protect each increment with lock
            mutex.withLock { counter++ }
        }
    }
    println("Counter = $counter") // Guaranteed `1000`
}
```



Inside CoroutineScope

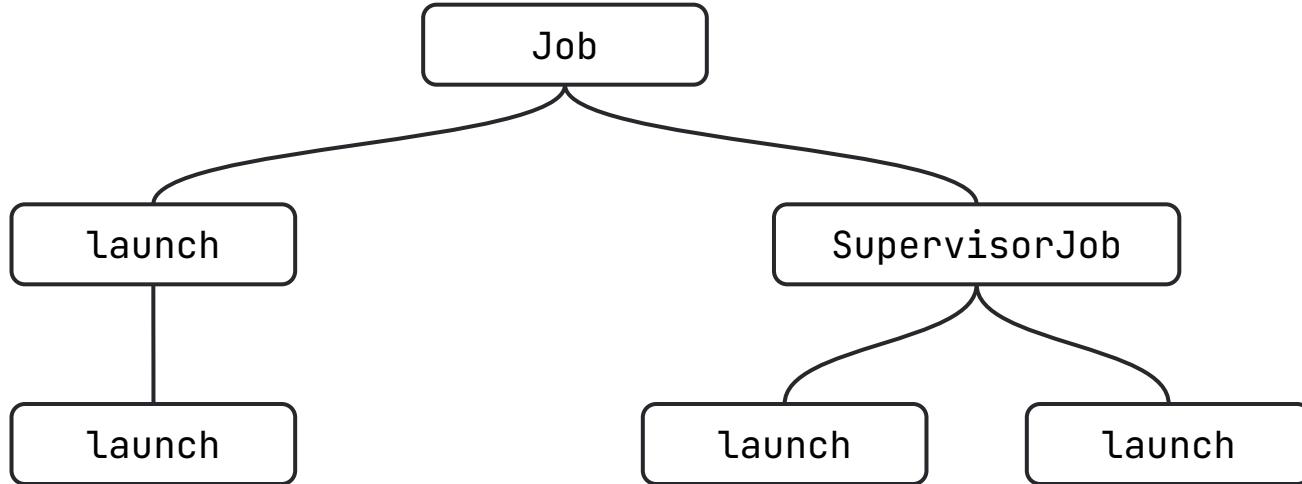
Exceptions

Exception handling

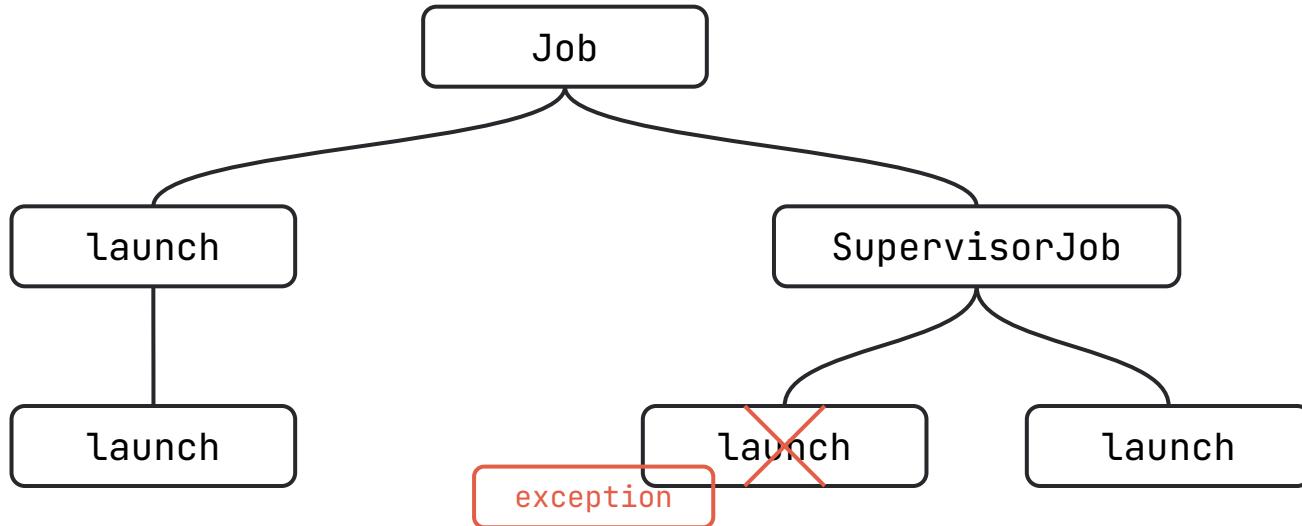
```
public interface CoroutineExceptionHandler : CoroutineContext.Element {  
    public companion object Key : CoroutineContext.Key<...>  
  
    public fun handleException(context: CoroutineContext, exception: Throwable)  
}
```

- Children coroutines delegate handling to their parents.
- Coroutines running with SupervisorJob do not propagate exceptions to their parents.
- CancellationExceptions are ignored.
- If there is a Job in the context, then Job.cancel is invoked.
- *All instances of CoroutineExceptionHandler found via ServiceLoader are invoked.*
- The current thread's Thread.uncaughtExceptionHandler is invoked.

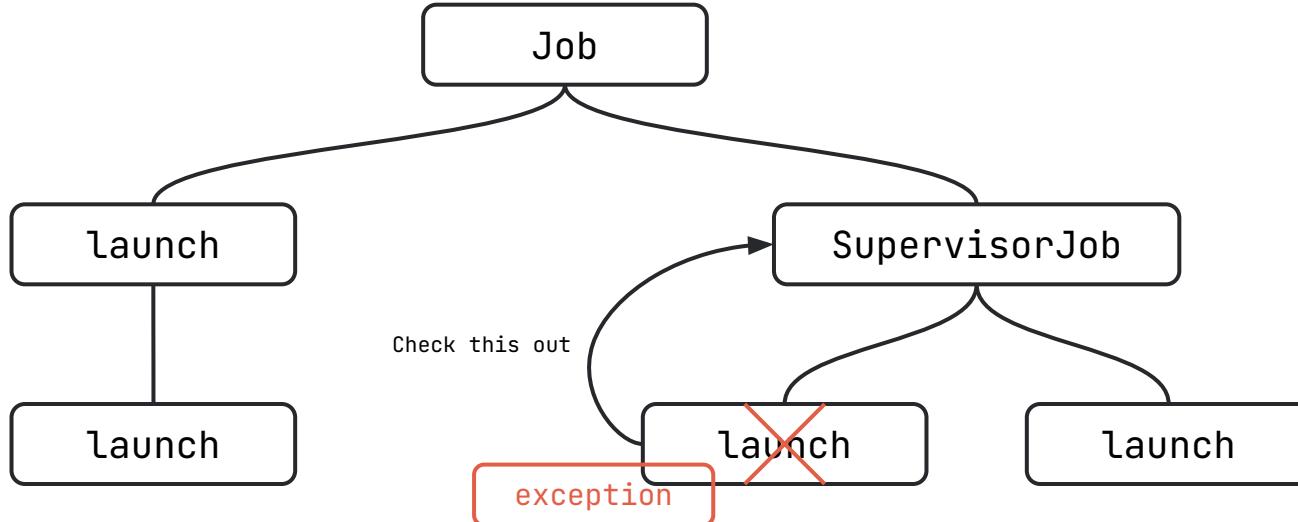
Exception propagation



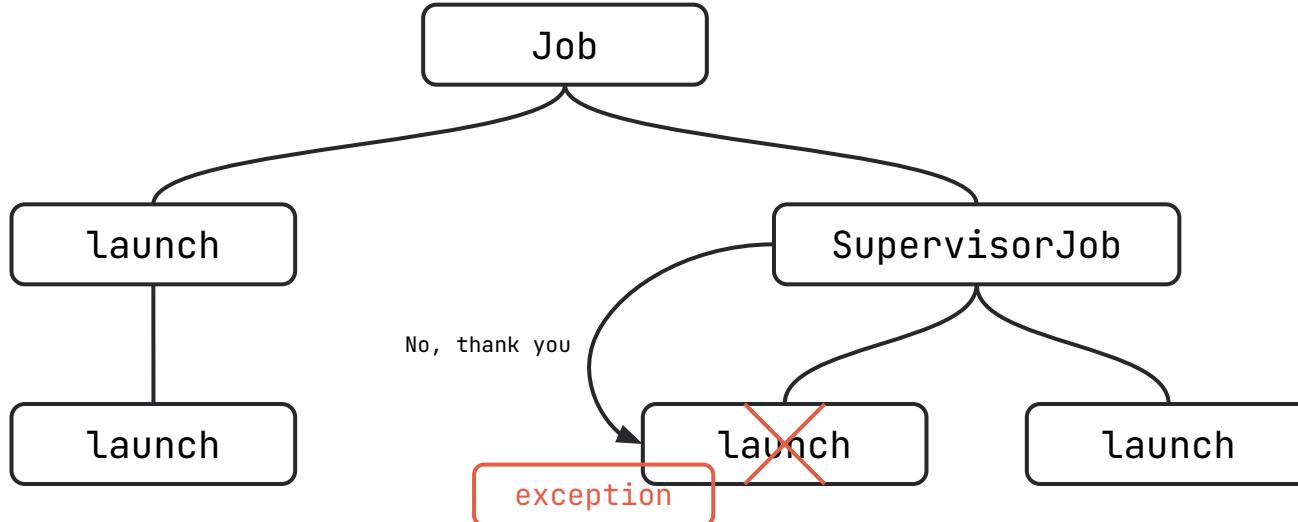
Exception propagation



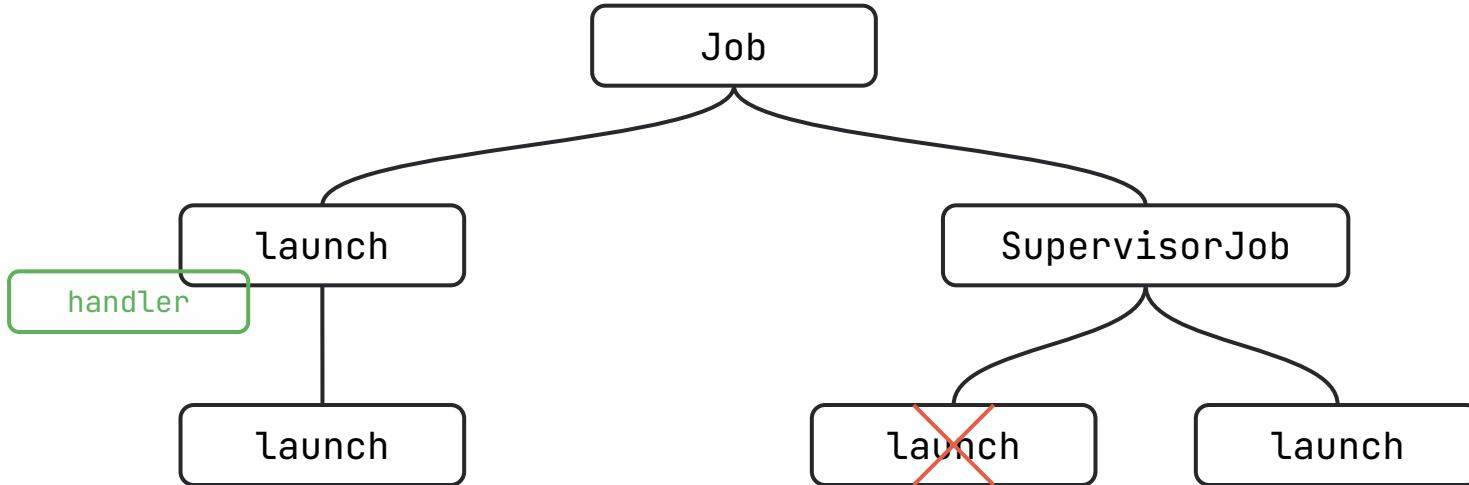
Exception propagation



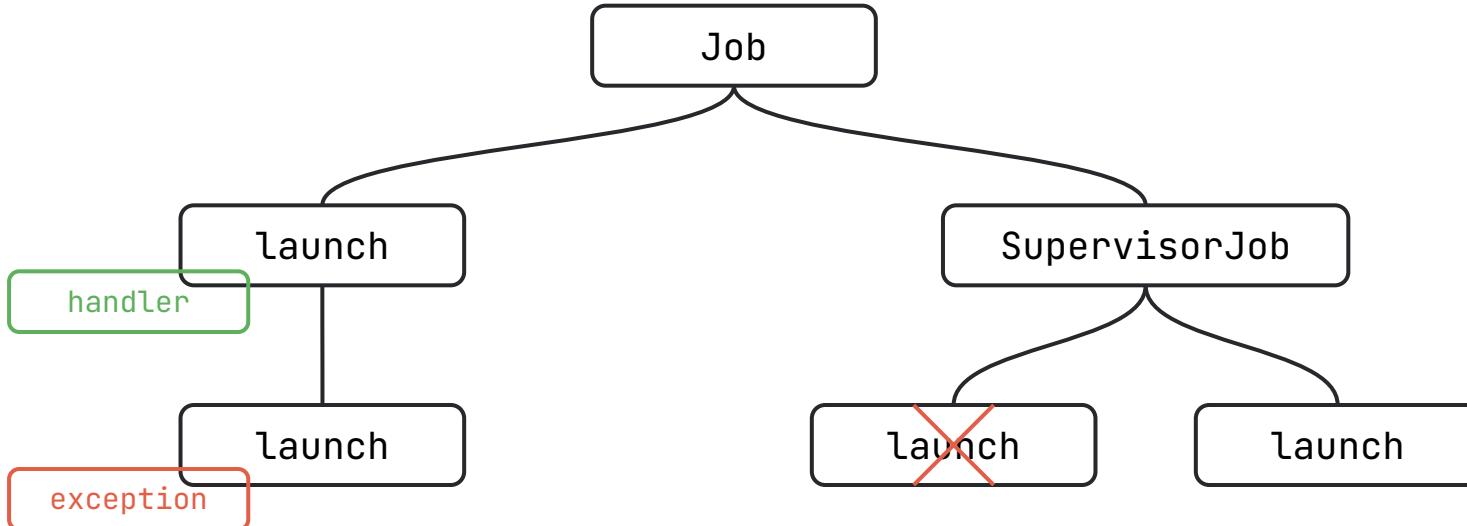
Exception propagation



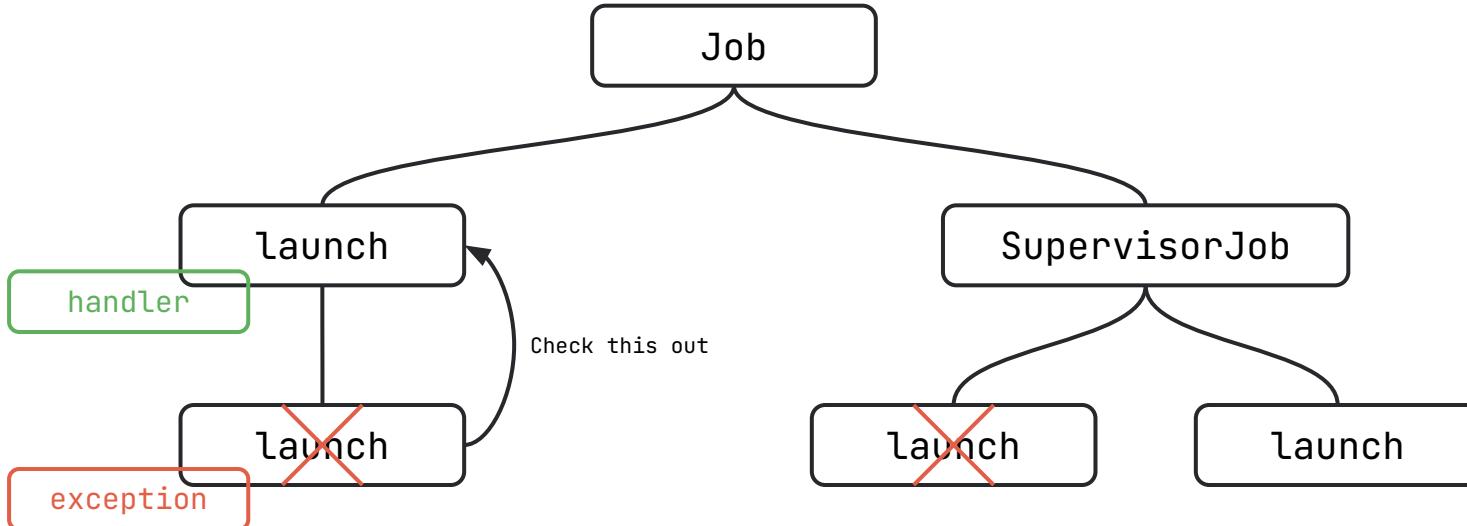
Exception propagation



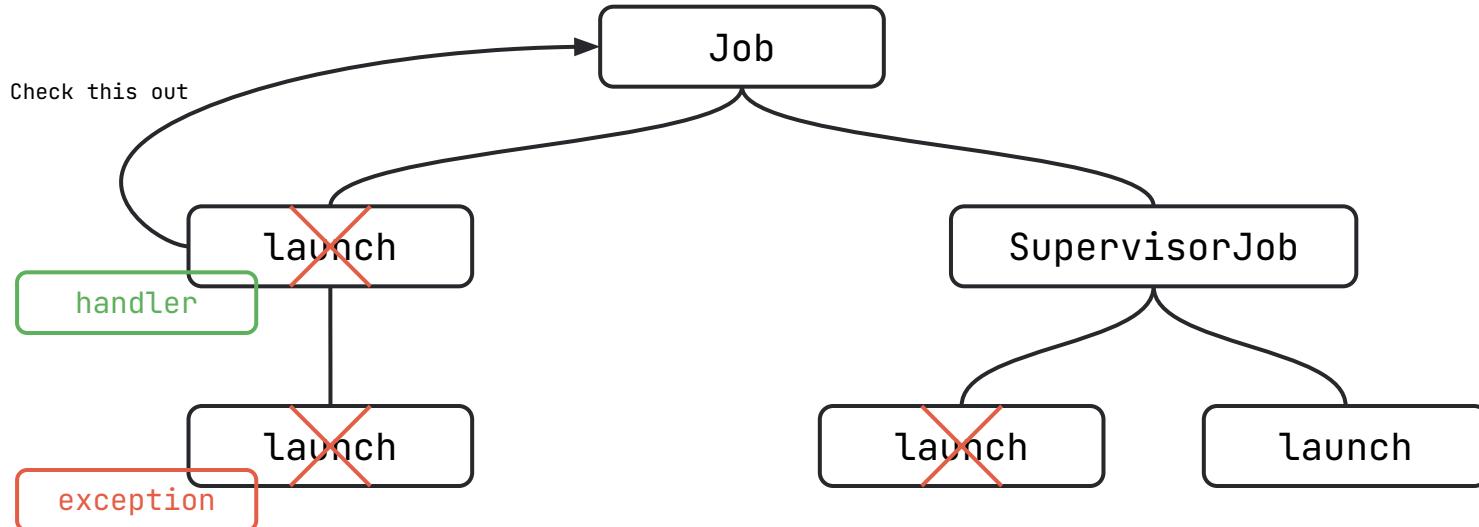
Exception propagation



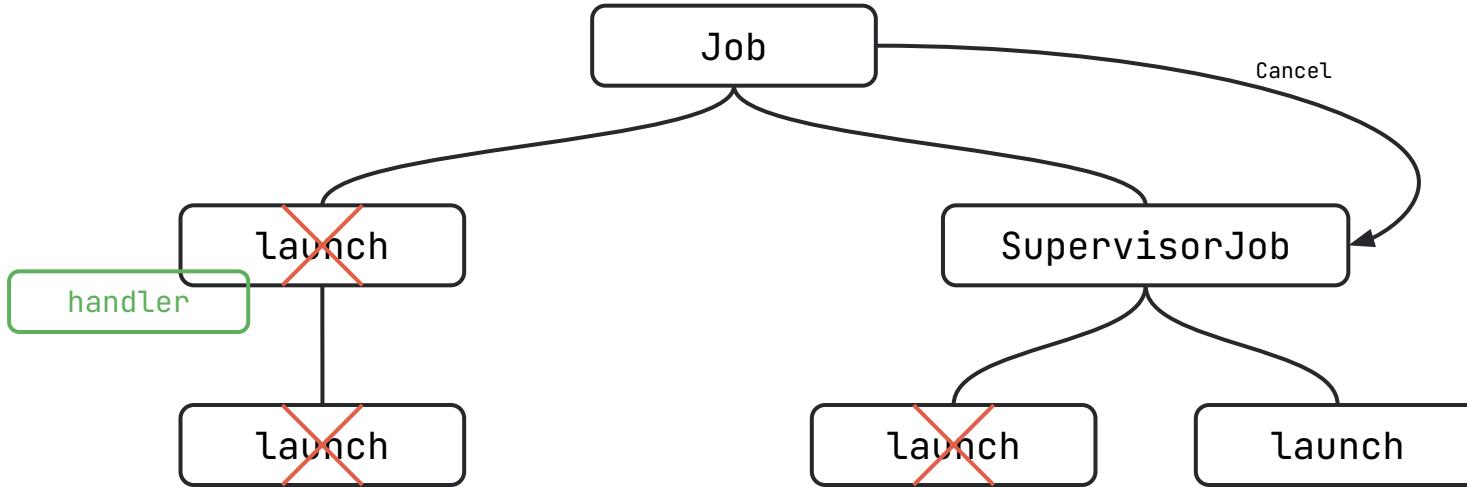
Exception propagation



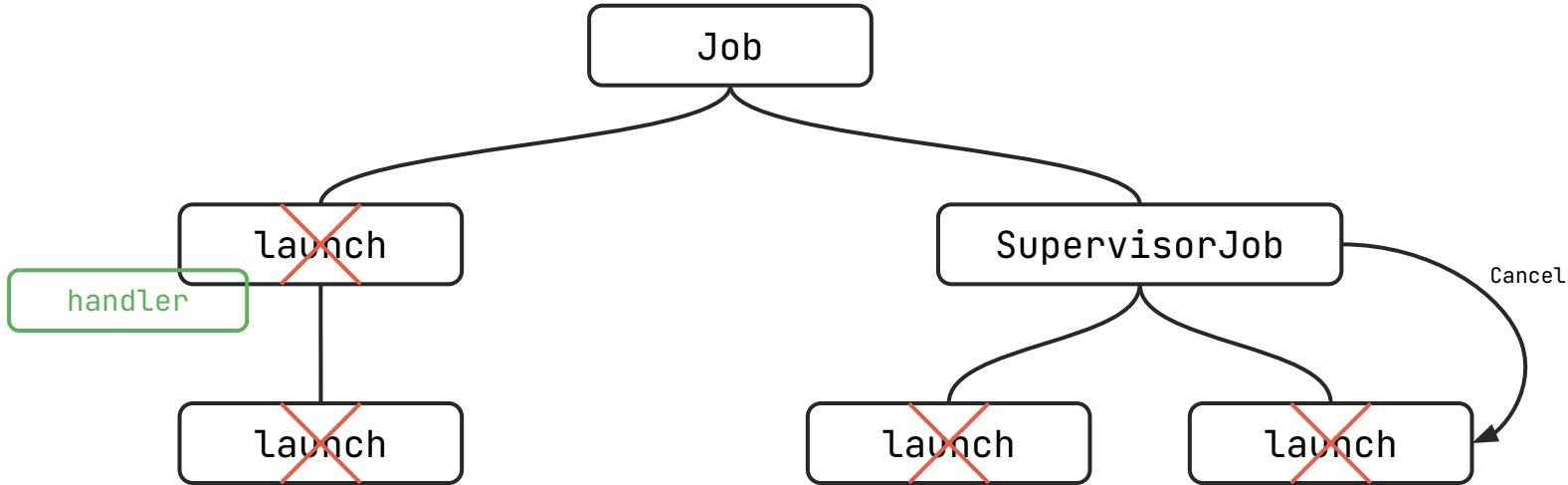
Exception propagation



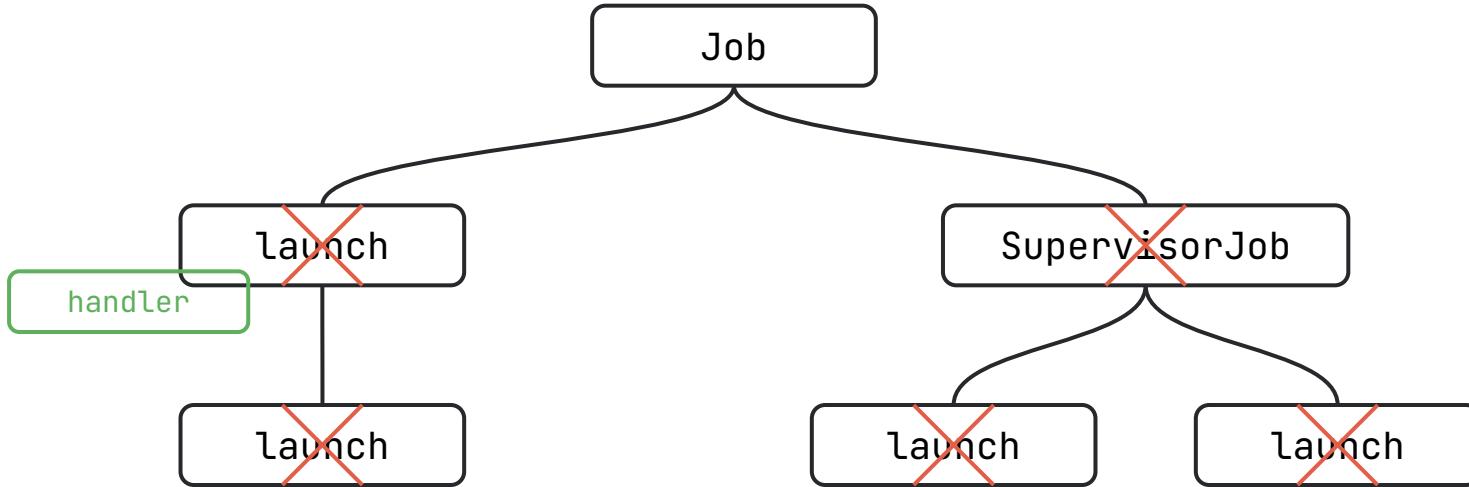
Exception propagation



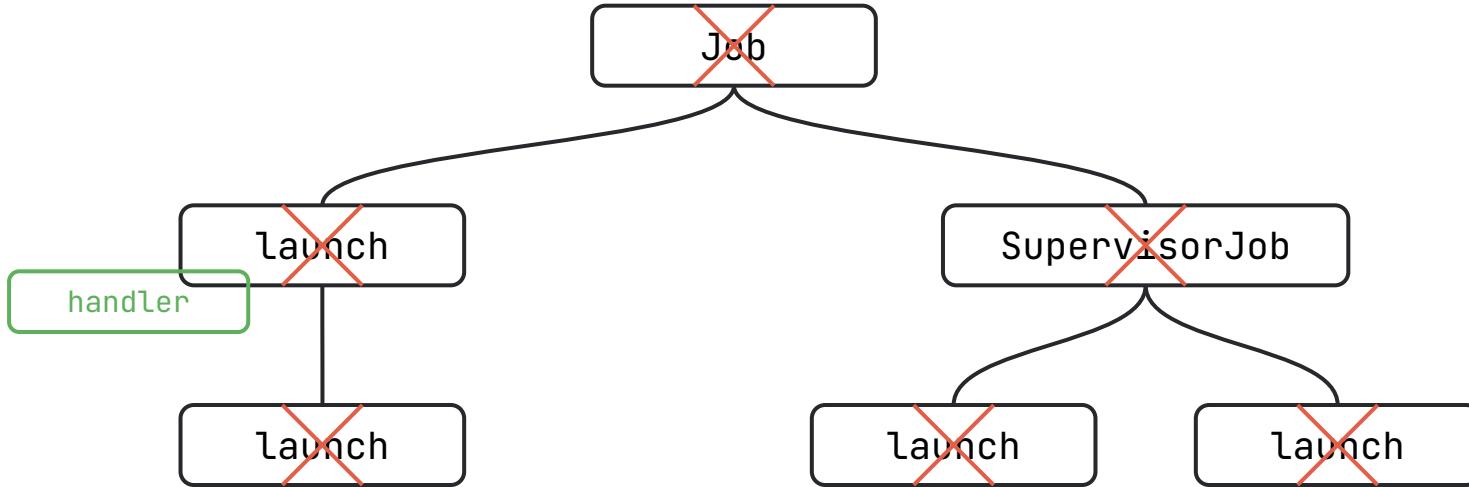
Exception propagation



Exception propagation



Exception Propagation



Now you see it

```
val jobs: List<Job> = List(1_000_000) {  
    launch(Dispatchers.Default + CoroutineName("#$it")  
        + CoroutineExceptionHandler { context, error →  
            println("${context[CoroutineName] ?.name}: $error")  
        },  
        CoroutineStart.LAZY  
    ) {  
        -$→ delay(Random.nextLong(1000))  
        if (it % 10 == 0) { throw Exception("No comments") }  
        println("Hello from coroutine $it!")  
    }  
}  
  
jobs.forEach { it.start() }
```

This exception handler is useless if this code is not inside a `SupervisorJob`.

Error handling

```
fun main() = runBlocking { // root coroutine
    val job1 = launch {
        delay(500)
        throw Exception("Some jobs just want to watch the world burn")
    }
    val job2 = launch {
        println("Going to do something extremely useful")
        delay(10000)
        println("I've done something extremely useful")
    }
}
```

Exception in job1 → propagate to parent → job2 gets cancelled

Error handling

```
fun main() {  
    val scope = CoroutineScope(Dispatchers.Default + SupervisorJob())  
    with(scope) {  
        val job1 = launch {  
            throw Exception("Some jobs just want to watch the world burn")  
        }  
        val job2 = launch {  
            delay(3000)  
            println("I've done something extremely useful")  
        }  
    }  
    scope.coroutineContext[Job]?.let { job →  
        runBlocking { job.children.forEach { it.join() } }  
    } // `job1.join()` will throw, so `it.join()` should actually be in a `try/catch` block  
}
```



Error handling

```
fun main() {  
    // `someScope: CoroutineScope` already exists  
    someScope.launch { // this coroutine is a child of someScope  
        supervisorScope { // SupervisorJob inside  
            val job1 = launch {  
                throw Exception("Some jobs just want to watch the world burn")  
            }  
            val job2 = launch {  
                println("Going to do something extremely useful")  
                delay(3000)  
                println("I've done something extremely useful")  
            }  
        }  
    }  
    ...  
}
```



Error handling

```
fun main() {  
    val scopeWithHandler = CoroutineScope(CoroutineExceptionHandler {  
        context, error → println("root handler called")  
    })  
    scopeWithHandler.launch {  
        supervisorScope {  
            launch { throw Exception() }  
            launch(CoroutineExceptionHandler { context, error →  
                println("personal handler called")  
            }) { throw Exception() }  
        }  
    }  
    ...  
}
```

Exceptions are not propagated to parents meaning you can override the handler.

Inside CoroutineScope

Structured concurrency

Error handling (revisited)

```
fun processReferences(refs: List<Reference>) {  
    for (ref in refs) {  
        val location = ref.resolveLocation()  
        GlobalScope.launch {  
            val content = downloadContent(location)  
            processContent(content)  
        }  
    }  
}
```



- Downloads are launched in the background.
- `GlobalScope` – *This is a delicate API and its use requires care. Make sure you fully read and understand the documentation of any declaration that is marked as a delicate API.* The delicate part is that no Job is attached to the `GlobalScope`, making its use dangerous and inconvenient.
- Any `downloadContent` or `processContent` crash results in a coroutines leak.

Structured concurrency

```
suspend fun processReferences(refs: List<Reference>) {  
    coroutineScope { // new scope with outer context, but a new Job  
        for (ref in refs) {  
            val location = ref.resolveLocation()  
            launch { // child of the coroutineScope above  
                val content = downloadContent(location)  
                processContent(content)  
            }  
        }  
    }  
}
```



In the event of a `downloadContent` or `processContent` crash, the exception goes to the `coroutineScope`, which stores links to all child coroutines and will cancel them. This is an example of structured concurrency, a concept that is not present in threads.

A helpful convention

This function takes a long time and waits for something:

```
suspend fun work(...) { ... }
```

This function launches more background work and quickly returns:

```
fun CoroutineScope.backgroundWork(...) {
    launch { ... }
}
```

or:

```
fun CoroutineScope.moreWork(...): Job = launch { ... }
```

Not:

```
suspend fun CoroutineScope.dontDoThisPlease()
```

A helpful convention

```
fun CoroutineScope.processReferences(refs: List<Reference>) {  
    for (ref in refs) {  
        val location = ref.resolveLocation()  
        launch { // child of coroutineScope  
            val content = downloadContent(location)  
            processContent(content)  
        }  
    }  
}
```



Inside CoroutineScope

Coroutine cancellation

Cancelling coroutines

```
val job = launch(Dispatchers.Default) {  
    repeat(5) {  
        println("job: I'm sleeping $it...")  
        Thread.sleep(500) // simulate blocking work  
    }  
}  
↳ yield() // lets the childJob work  
println("main: I'm tired of waiting!")  
job.cancel() // cancels the `job`  
↳ job.join() // waits for `job`'s completion  
println("main: Now I can quit.")
```

- The coroutine (job) does not know that somebody is trying to cancel it.
- Cancellation is *cooperative*.

Cancelling coroutines

```
val job = launch(Dispatchers.Default) {  
    repeat(5) {  
        try {  
            println("job: I'm sleeping $it...")  
            delay(500)  
        } catch (e: CancellationException) {  
            println("job: I won't give up $it")  
        }  
    }  
}  
yield()  
println("main: I'm tired of waiting!")  
job.cancelAndJoin() // cancel + join  
println("main: Now I can quit.")
```

Cancelling coroutines

```
val job = launch(Dispatchers.Default) {  
    var i = 0  
    while (isActive && i < 5) { // check Job status  
        println("job: I'm sleeping ${i++}...")  
        Thread.sleep(500)  
    }  
}  
→ delay(1300L)  
    println("main: I'm tired of waiting!")  
→ job.cancelAndJoin()  
    println("main: Now I can quit.")
```

Cancelling coroutines

```
val job = launch {
    try {
        repeat(1_000) {
            println("job: I'm sleeping $it...")
            delay(500L)
        }
    } finally {
        withContext(NonCancellable) {
            println("job: I'm running finally")
            delay(1000L)
            println("job: Delayed for 1 sec thanks to NonCancellable")
        }
    }
}
...
job.cancelAndJoin()
```

Channels

Communicating sequential processes

`Channel` is like `BlockingQueue`, but with suspending calls instead of blocking ones.

- Blocking put → suspending send
- Blocking take → suspending receive
- No shared mutable state!
- Channels are still **experimental**

```
public interface Channel<E> : SendChannel<in E>, ReceiveChannel<out E> {  
    ... suspend fun send(element: E)  
    ... suspend fun recieve(): E  
    ...  
}
```

There are also `trySend` and alike that do not wait.

Practice

```
fun main() = runBlocking {  
    val channel = Channel<Int>()  
    launch {  
        for (x in 1..5)  
            channel.send(x * x)  
    }  
    repeat(5) {  
        println(channel.receive())  
    }  
    println("Done!")  
}
```

Prime Numbers

```
fun CoroutineScope.numbersFrom(start: Int) = produce<Int> {
    var x = start
    while (true) send(x++) // infinite stream of integers from start
}

fun CoroutineScope.filter(numbers: ReceiveChannel<Int>, prime: Int) =
    produce<Int> { for (x in numbers) if (x % prime != 0) send(x) }

fun main() = runBlocking {
    var cur = numbersFrom(2)
    repeat(10) {
        println(cur.receive())
        cur = filter(cur, prime)
    }
    coroutineContext.cancelChildren()
}
```

Fan-in and fan-out

```
fun <T> CoroutineScope.production(ch: SendChannel<T>, msg: T) =  
    launch { while (true) { delay(Random.nextLong(23)); ch.send(msg) } }  
  
fun <T> CoroutineScope.processing(ch: ReceiveChannel<T>, name: String) =  
    launch { for (msg in ch) { println("$name: received $msg") } }  
  
fun main() = runBlocking {  
    val channel = Channel<String>()  
    listOf("foo", "bar", "baz").forEach { production(channel, it) }  
    repeat(8) { processing(channel, "worker #$it") }  
    delay(700)  
    coroutineContext.cancelChildren(CancellationException("Enough!"))  
}
```

Details

- Channels are still experimental.
- Channels are fair, meaning that send and receive calls are served in a first-in first-out order.
- By default, channels have RENDEZVOUS capacity: no buffer at all. This behavior can be tweaked: The user can specify buffer capacity, what to do when buffer overflows, and what to do with undelivered items.

Select (experimental!)

```
suspend fun selector(
    channel1: ReceiveChannel<String>,
    channel2: ReceiveChannel<String>
): String = select<String> {
    // onReceive clause in select fails when the channel is closed
    channel1.onReceive { it: String → "b → '$it'" }
    channel2.onReceiveCatching { it: ChannelResult<String> →
        val value = it.getOrNull()
        if (value ≠ null) {
            "a → '$value'"
        } else {
            "Channel 'a' is closed" // Select does not stop!
        }
    }
}
```

Miscellaneous

Miscellaneous

Beyond asynchronous programming

Sequences

```
val fibonacci = sequence { // A coroutine builder!
    var cur = 1
    var next = 1
    while (true) {
        yield(cur) // A suspending call!
        cur += next
        next = cur - next
    }
}

val iter = fibonacci.iterator() // nothing happens yet
println(iter.next()) // process up to the first yield → 1
println(iter.next()) // wake up and continue → 1
println(iter.next()) // 2 and then to infinity and beyond
```



Miscellaneous

Under the hood: advanced

Under the hood

Remember this code?

```
suspend fun postItem(item: Item) {  
    val token = preparePost()  
    val post = submitPost(token, item)  
    processPost(post)  
}
```

Now that we know much more, let's get a better *approximation* of what's going on under the hood.

Under the hood

```
fun postItem(item: Item, completion: Continuation<Any?>) {  
  
    class PostItemStateMachine(  
        completion: Continuation<Any?>?,  
        context: CoroutineContext?  
    ): ContinuationImpl(completion) {  
        var result: Result<Any?> = Result(null)  
        var label: Int = 0  
  
        var token: Token? = null  
        var post: Post? = null  
        ...  
    }  
}
```

Under the hood

```
fun postItem(item: Item, completion: Continuation<Any?>) {  
  
    class PostItemStateMachine(...): ... {  
        ...  
        override fun invokeSuspend(result: Result<Any?>) {  
            this.result = result  
            postItem(item, this)  
        }  
    }  
  
    val continuation = completion as? PostItemStateMachine ?: PostItemStateMachine(completion)  
    ...  
}
```

Under the hood

```
...
when(continuation.label) {
    0 → { ... }
    1 → {
        continuation.token = continuation.result.getOrThrow() as Token
        continuation.label = 2
        submitPost(continuation.token!!, continuation.item!!, continuation)
    }
    2 → { ... }
    3 → {
        continuation.finalResult = continuation.result.getOrThrow() as FinalResult
        continuation.completion.resume(continuation.finalResult!!)
    }
    else → throw IllegalStateException(...)
}
...
...
```

More

Continuation as generic callback

Continuation

Here's a refresher on what Continuation looks like:

```
public interface Continuation<in T> {  
    public val context: CoroutineContext  
    public fun resumeWith(result: Result<T>)  
}
```

We are given:

```
suspend fun suspendAnswer() = 42  
suspend fun suspendSqr(x: Int) = x * x
```

How can we run `suspendSqr(suspendAnswer)` without `kotlinx.coroutines`?

Continuation

Continuation is a generic callback, so we can go back to the continuation passing style:

```
fun main() {
    ::suspendAnswer.startCoroutine(object : Continuation<Int> {
        override val context: CoroutineContext
            get() = CoroutineName("Empty Context Simulation")

        override fun resumeWith(result: Result<Int>) {
            val prevResult = result.getOrThrow()
            ::suspendSqr.startCoroutine(
                prevResult,
                Continuation(CoroutineName("Only name Context")) {
                    it: Result<Int> → println(it.getOrNull())
                }
            )
        }
    } // Oh no!
}) // Closing brackets are coming!
} // Please help! I am being dragged into Callback Hell!!!
```

Miscellaneous

To wrap existing async code or to implement your own?

To wrap existing async code or to implement your own?

```
suspend fun AsynchronousFileChannel.aRead(b: ByteBuffer, p: Int = 0) =
    // Scheme: call-with-current-continuation; call/cc
    suspendCoroutine { cont →
        // CompletionHandler ~ Continuation
        read(b, p.toLong(), Unit, object : CompletionHandler<Int, Unit> {
            override fun completed(bytesRead: Int, attachment: Unit) {
                cont.resume(bytesRead)
            }

            override fun failed(exception: Throwable, attachment: Unit) {
                cont.resumeWithException(exception)
            }
        })
    }
```

To wrap existing async code or to implement your own?

```
fun main() = runBlocking {
    val readJob = launch(Dispatchers.IO) {
        val fileName = ...
        val channel = AsynchronousFileChannel.open(Paths.get(fileName))
        val buf = ByteBuffer.allocate(...)
        channel.use { // syntactic sugar for `try { ... } finally { channel.close() }` 
            while (isActive) {
                ... = it.aRead(buf)
                ...
            }
        }
    }
    ...
}
```

To wrap existing async code or to implement your own?

```
suspend fun cancellable(...) =  
    suspendCancellableCoroutine { cancellableCont →  
        cancellableCont.invokeOnCancellation { throwable: Throwable? →  
            // release resources, etc.  
            ...  
        }  
        ...  
        cancellableCont.cancel(...)  
    }
```

Miscellaneous

async / await

async / await in Kotlin

```
async Task PostItem(Item item) {  
    Task<Token> tokenTask = PreparePost();  
    Post post = await SubmitPost(tokenTask.await(), item);  
    ProcessPost();  
}
```

- `async` and `await` are keywords in C#.
- Awaiting does not block heavy OS thread.
- `await` is an explicit suspension point.
- `await` is a single function, but depending on its environment it can result in 2 different behaviours.
- The C# approach was a great inspiration for the Kotlin team when they were designing coroutines, as it was for Dart, TS, JS, Python, Rust, C++...

async / await in Kotlin

```
fun CoroutineScope.preparePostAsync(): Deferred<Token> = async<Token> { ... }
```

```
suspend fun postItem(item: Item) {
    coroutineScope {
        -> val token = preparePost().await()
        -> val post = submitPost(token, item).await()
        processPost(post)
    }
}
```

Deferred<T> : Job is a Job that we can get some result from. async is just another coroutine builder. You can write exactly the same in Kotlin!

But why would you do that? This is not idiomatic Kotlin.

async / await in Kotlin

```
suspend fun postItemAsyncAwait(item: Item) {  
    coroutineScope {  
        val deferredToken = async { preparePost() }  
        // some work  
        val token = deferredToken.await()  
        val deferredPost = async { submitPost(token, item) }  
        // more work  
        val post = deferredPost.await()  
        processPost(post)  
    }  
}
```

Miscellaneous

Coroutine builders

A zoo of them!

```
public fun CoroutineScope.launch(
    context: CoroutineContext,
    start: CoroutineStart,
    block: suspend CoroutineScope.() -> Unit // suspend lambda
): Job
public fun <T> future(...): CompletableFuture<T> // jdk8/experimental
public fun <T> CoroutineScope.async(...): Deferred<T>
public fun <T> runBlocking(...): T // Avoid using it
public fun <E> CoroutineScope.produce(
    context: CoroutineContext,
    capacity: Int,
    @BuilderInference block: suspend ProducerScope<E>.() -> Unit
): ReceiveChannel<E>
```

And many more! Like actor.

Actor

Actor ~ coroutine + channel

```
// Message types for counterActor - Command pattern
sealed class CounterMsg
// one-way message to increment
object IncCounter : CounterMsg() counter
// a request with a reply
class GetCounter(val response: CompletableDeferred<Int>) : CounterMsg()
```

Actor

```
// This function launches a new counter actor
fun CoroutineScope.counterActor() = actor<CounterMsg> {
    var counter = 0 // actor state
    for (msg in channel) { // iterate over incoming messages
        when (msg) {
            is IncCounter → counter++
            is GetCounter → msg.response.complete(counter)
        }
    }
}
```



Frequently encapsulated into a separate class.

Miscellaneous

Android

Android

Check out developer.android.com to learn how coroutines are used (extensively) in modern Android development.

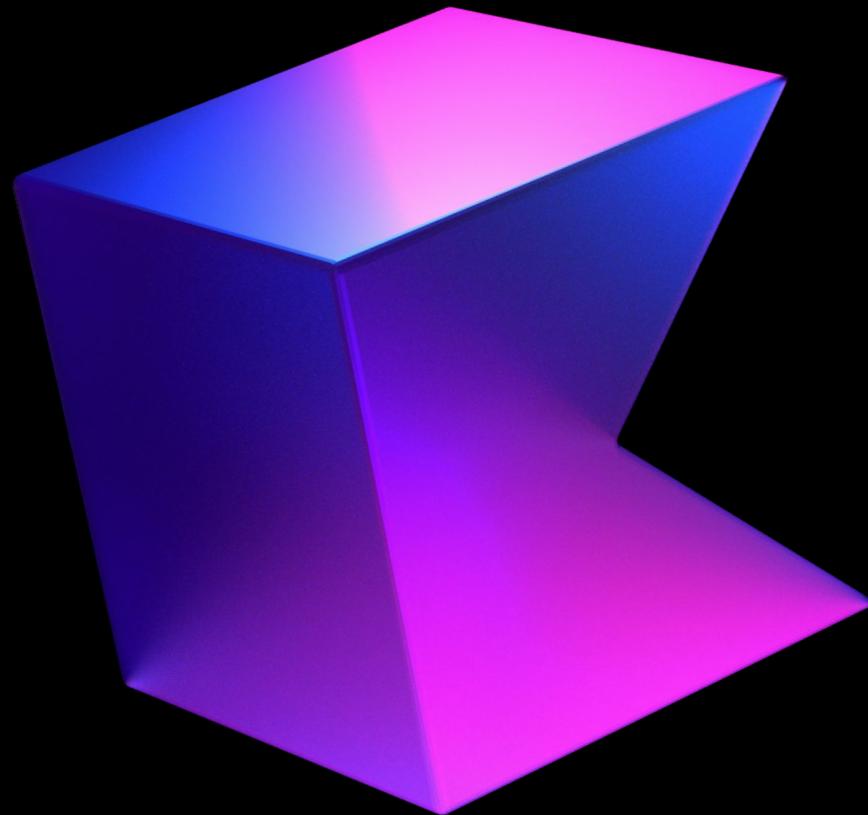
```
class MyViewModel: ViewModel() {  
    init {  
        viewModelScope.launch { ... }  
    }  
}
```

- A `ViewModelScope` is defined for each `ViewModel` in your app.
- A `LifecycleScope` is defined for each `Lifecycle` object.
- `Flow`, which is not covered in this lecture, is common in Android.

Further Reading

- github.com/Kotlin/KEEP/ – Kotlin design proposals, including coroutines
- kotlinlang.org – “Coroutines overview” and “Official libraries/kotlinx.coroutines”
- github.com/Kotlin/kotlinx.coroutines – A nicely documented resource
- Roman Elizarov’s talks on YouTube and posts on medium
- [Flow<T>](#) – Asynchronous Flow
- Kotlin sources at github.com/JetBrains/kotlin/
- All of the code from this presentation can be found in the coroutines folder at github.com/bochkarevko/kotlin-things/

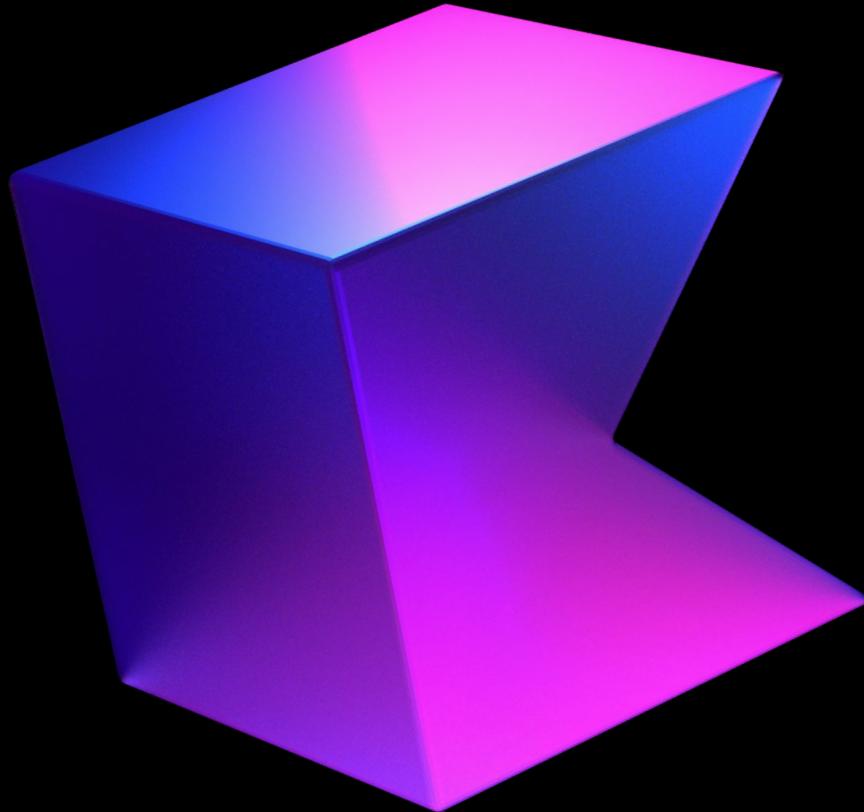
Thanks!



@kotlin



Exceptions



What? Why?

An exception signals that something went *exceptionally* wrong.

- Development mistakes
- Errors produced by external (to the program) resources
- System errors

Why use exceptions:

- To separate error-handling code from regular code
- To propagate errors up the call stack – maybe someone knows how to deal with the error
- To group and differentiate error types

Do NOT use exceptions for:

- Control flow
- Manageable errors

How?

```
fun main() {  
    throw Exception("Hello, world!")  
}
```

Or even better:

```
fun main() {  
    val nullableString: String? = null  
    println("Hello, NPE! ${nullableString!!}")  
}  
  
Exception in thread "main" java.lang.NullPointerException
```

Example

```
fun main() {  
    try {  
        throw Exception("An exception", RuntimeException("A cause"))  
    } catch (e: Exception) {  
        println("Message: ${e.message}")  
        println("Cause: ${e.cause}")  
        println("Exception: $e") // toString() is called "under the hood"  
        e.printStackTrace()  
    } finally {  
        println("Finally always executes")  
    }  
}
```

Message: An exception
Cause: java.lang.RuntimeException: A cause
Exception: java.lang.Exception: An exception
Finally always executes
java.lang.Exception: An exception
at MainKt.main(Main.kt:3)
at MainKt.main(Main.kt)
Caused by: java.lang.RuntimeException: A cause
... 2 more

Another meaningful example

```
data class Person(val name: String, val surname: String, val age: Int) {  
    init {  
        if (age < 0) {  
            throw IllegalStateException("Age cannot be negative")  
        }  
        if (name.isEmpty() || surname.isEmpty()) {  
            throw IllegalArgumentException("For blank names/surnames use -")  
        }  
    }  
}
```

Dealing with exceptions

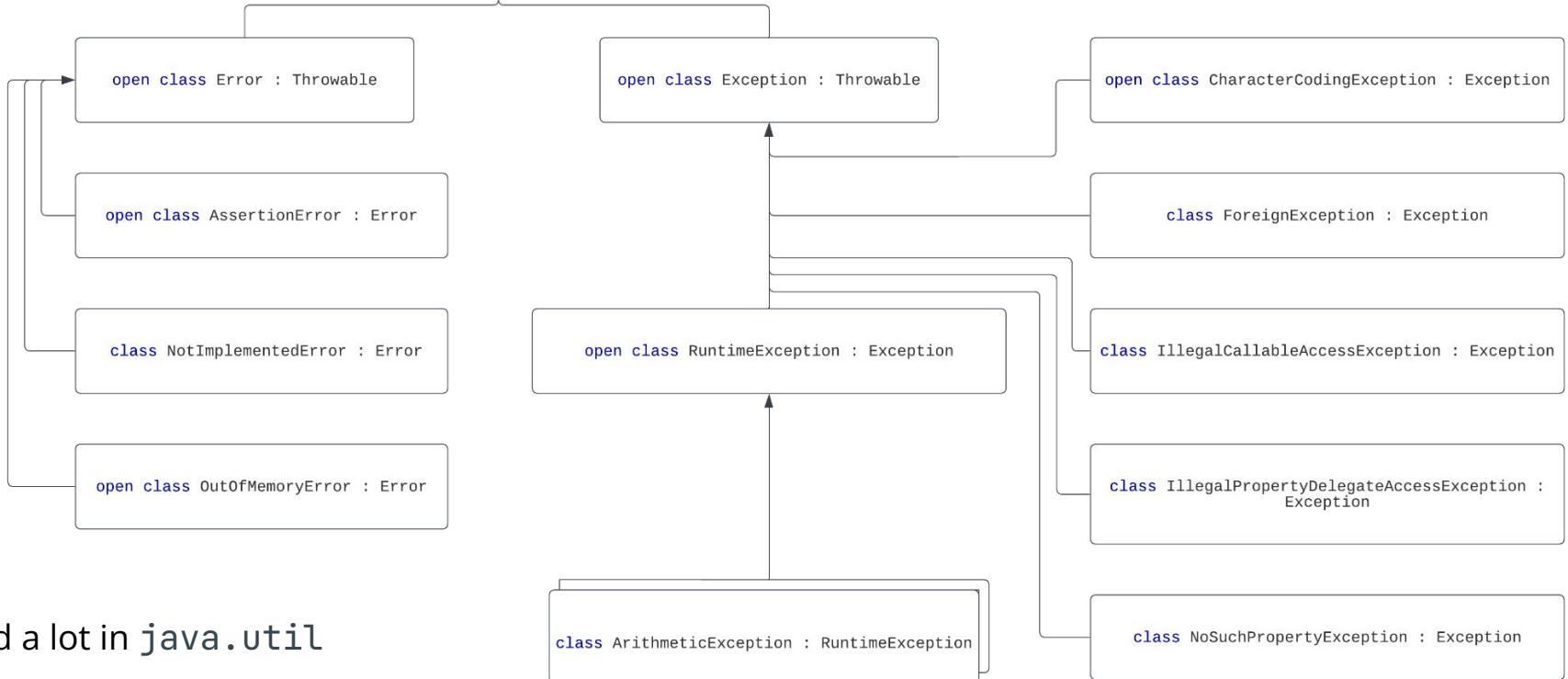
```
try {
    val (n, s, a) = readLine()!!.split('/')
    val person = Person(n, s, a.toInt())
    addToDataBase(person)
} catch (e: IllegalStateException) {
    println("You've entered a negative age! Why?")
} catch (e: IllegalArgumentException) {
    println(e.message)
} catch (e: NullPointerException) {
    println("NPE ;^)")
} catch (e: Exception) {
    println("Something else went wrong")
    throw Exception("Failed to add to the database", e)
} finally {
    println("See you in the next episodes!")
}
```

You might:

- Handle the error properly and continue execution
- Handle something on your side and re-throw the exception



```
open class Throwable  
  
+ open val message: String?  
+ open val cause: Throwable?  
  
+ fun getStackTrace(): Array<String>  
+ fun printStackTrace()  
+ open fun toString(): String
```



And a lot in `java.util`

Kotlin sugar

`try` is an expression:

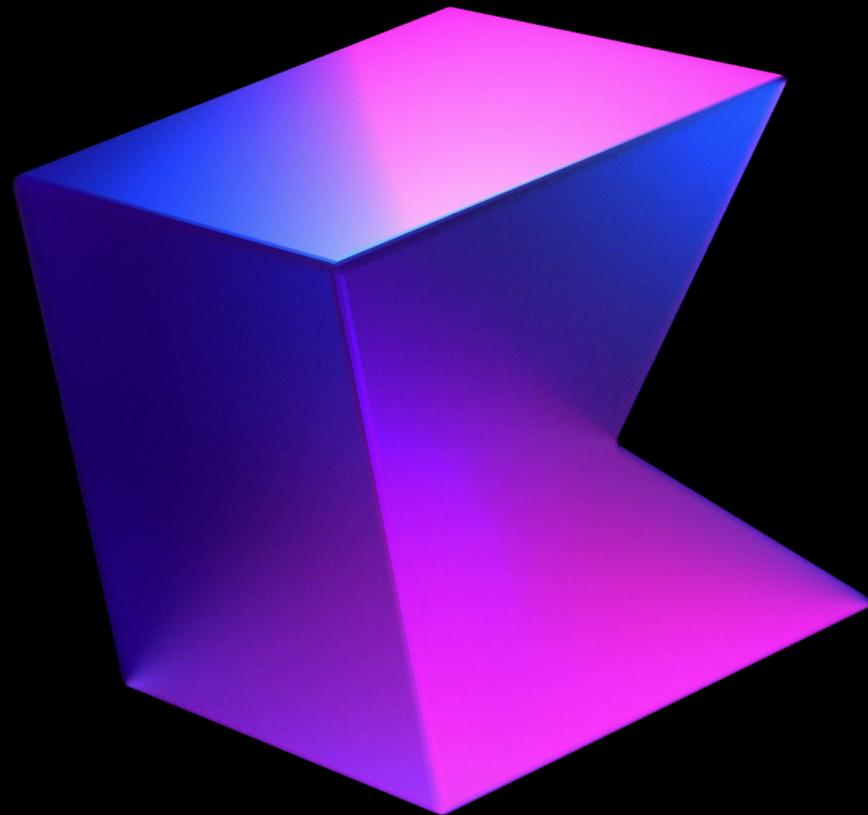
```
val a: Int? = try { input.toInt() } catch (e: NumberFormatException) { null }
```

More sugar:

```
require(count ≥ 0) { "Count must be non-negative, was $count" }
// IllegalArgumentException
```

```
error("Error message")
// IllegalStateException
```

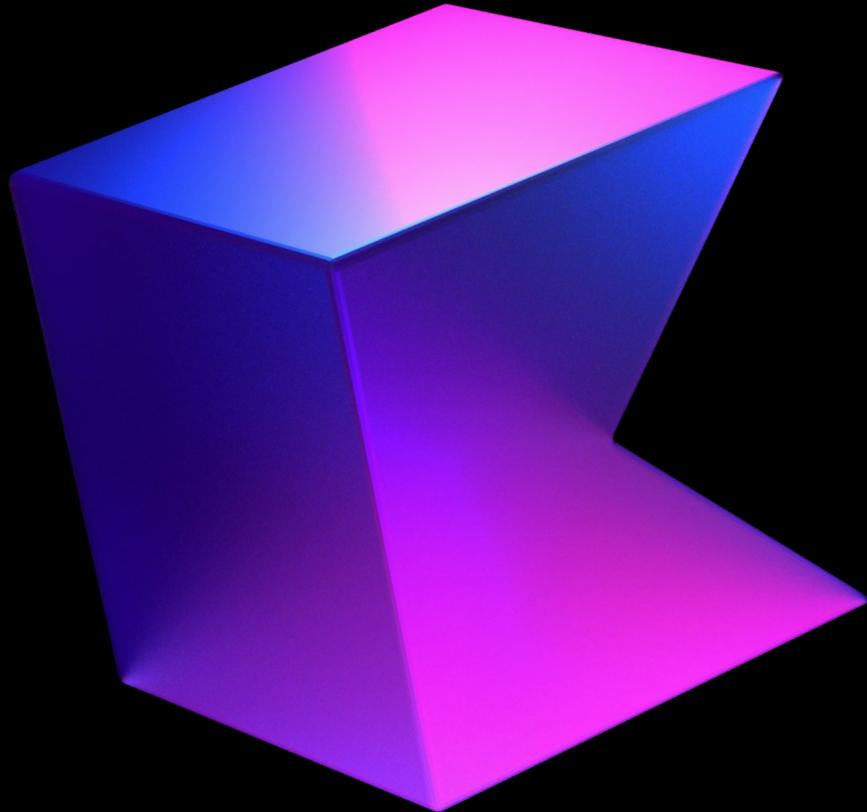
Thanks!



@kotlin



Testing



@kotlin

Testing

The software development experience we've accumulated over the many years unfortunately tells us that

- Every program contains bugs.
- If a program does not contain bugs, the algorithm that it implements contains them.
- If neither the program nor the algorithm contains bugs, no one needs the program (almost always).

Testing

However, everything is relative:

- A bug in a website might not really hurt anybody
- A bug in rocket launch calculations can result in terrible consequences and expenses ([Ariane 5](#))
- A bug in a radiation-treatment device can lead to deaths ([Therac-25](#))



Ariane 5

Testing: history

- 60s: Do exhaustive testing.
- Early 70s: Show that the program works correctly.
- Late 70s: Show that the program does not work correctly.
- 80s: Prevent defects throughout development.

Testing: goals

- Correct behavior of the product in all conditions.
- Compliance with the requirements.
- Information about the current state of the product.
- Error prevention and detection.
- Development cost reduction.

Testing: principles

- Testing demonstrates the presence of defects, but it does not prove their absence.
- The earlier the better.
- The absence of bugs is not an absolute goal.
- Many more.

Testing: types

- Functional – Checking the behavior given in the specifications.

Testing: types

- Functional – Checking the behavior given in the specifications.
- Load – Simulating a real load (for example, a certain number of users on the server).

Testing: types

- Functional – Checking the behavior given in the specifications.
- Load – Simulating a real load.
- Stress – Checking the system's operation under abnormal conditions (for example, a power outage or a huge number of operations per second).

Testing: types

- Functional – Checking the behavior given in the specifications.
- Load – Simulating a real load.
- Stress – Checking the system's operation under abnormal conditions.
- Configuration – Checking software using different system configurations.

Testing: types

- Functional – Checking the behavior given in the specifications.
- Load – Simulating a real load.
- Stress – Checking the system's operation under abnormal conditions.
- Configuration – checking software using different system configurations.
- Regression – Making sure new changes did not break anything that had worked previously.

Testing: types

- Functional – Checking the behavior given in the specifications.
- Load – Simulating a real load.
- Stress – Checking the system's operation under abnormal conditions.
- Configuration – Checking software using different system configurations.
- Regression – Making sure new changes did not break anything that had worked previously.

Others, e.g. security, compliance, etc.

Testing: levels

- Unit testing – Testing components separately (checking modules, classes, functions).

Testing: levels

- Unit testing – Testing components separately (checking modules, classes, functions).
- Integration testing – Checking the interaction of components and program modules.

Testing: levels

- Unit testing – Testing components separately (checking modules, classes, functions).
- Integration testing – Checking the interaction of components and program modules.
- System testing – Checking the entire system.

Testing: levels

- Unit testing – Testing components separately (checking modules, classes, functions).
- Integration testing – Checking the interaction of components and program modules.
- System testing – Checking the entire system.
- Acceptance testing – Verifying system compliance with all the client requirements.

Unit testing

For each non-trivial function, their own tests are written that check that the method works correctly:

- Frequent launch expected ⇒ should run fast
- One test ⇒ one use case

Unit testing in Kotlin

The [JUnit5](#) framework is the most popular way to test Java and Kotlin programs.

```
dependencies {  
    ...  
    testImplementation(platform("org.junit:junit-bom:5.8.2"))  
    testImplementation("org.junit.jupiter:junit-jupiter:5.8.2")  
}
```

```
tasks.test {  
    useJUnitPlatform()  
}
```

To run tests: ./gradlew test

Unit testing in Kotlin

```
class MyTests {  
    @Test  
    @DisplayName("Check if the calculator works correctly")  
    fun testCalculator() {  
        Assertions.assertEquals(  
            3,  
            myCalculator(1, 2, "+"),  
            "Assertion error message"  
        )  
    }  
}
```

Unit testing in Kotlin

```
class MyParametrizedTests {
    companion object {
        @JvmStatic
        fun calculatorInputs() = listOf(
            Arguments.of(1, 2, "+", 3),
            Arguments.of(0, 5, "+", 5),
        )
    }
    @ParameterizedTest
    @MethodSource("calculatorInputs")
    fun testCalculator(a: Int, b: Int, op: String, expected: Int) {
        Assertions.assertEquals(expected, myCalculator(a, b, op), "Assertion error message")
    }
}
```

Unit testing in Kotlin

There are many annotations for tests customization.

- `@BeforeEach` – Methods with this annotation are run before each test.
- `@AfterEach` – Methods with this annotation are run after each test.
- `@BeforeAll` – Methods with this annotation are run before all tests in the class.
- `@AfterAll` – Methods with this annotation are run after all tests in the class.

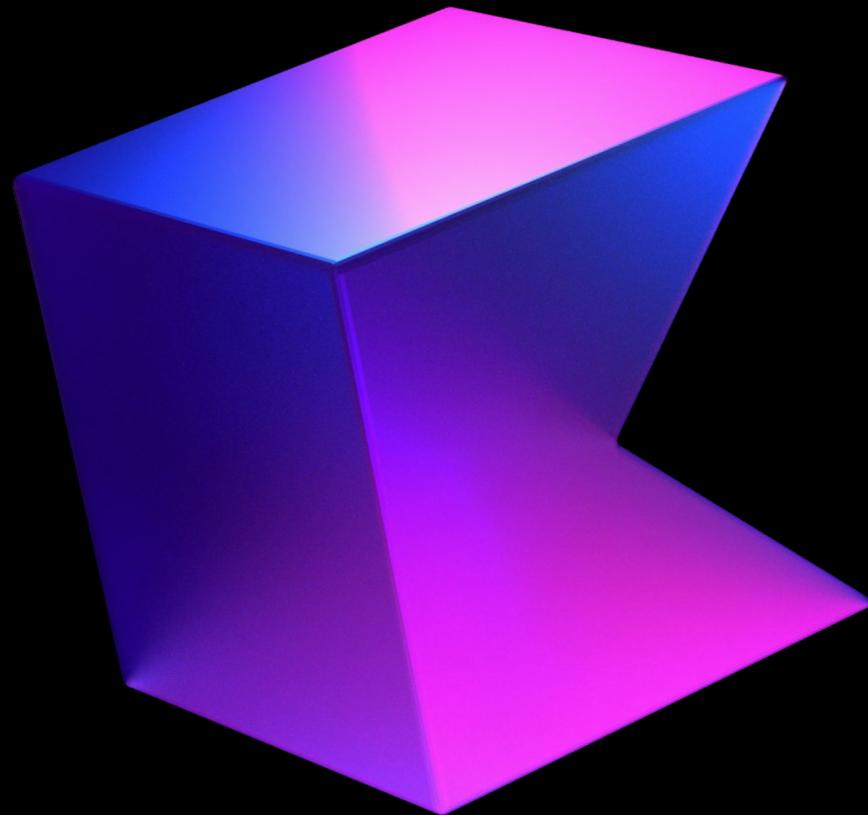
Code quality

Often, testing includes checking not only the correctness of the functionality but also the **quality** of the code itself.

Static code analyzers such as [detekt](#), [ktlint](#), and [diktat](#) exist to help you avoid having to do this manually.

The build of static analyzers should also be green.

Thanks!



@kotlin