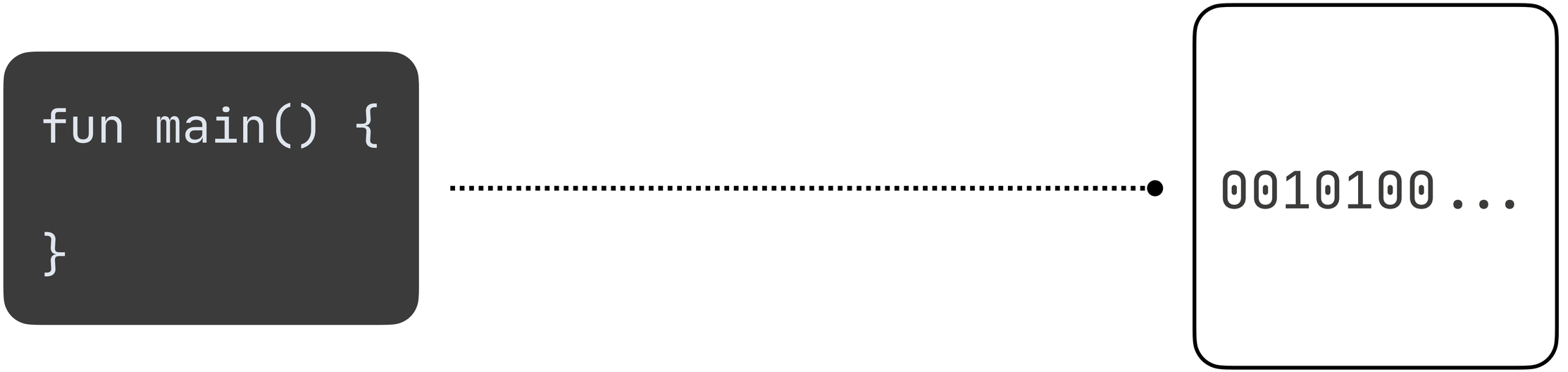
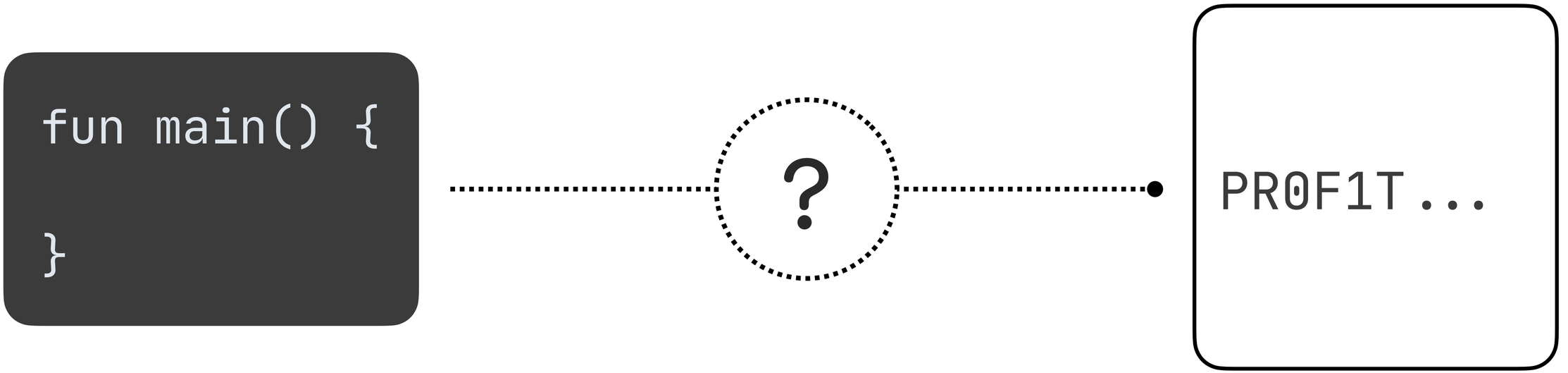
badoo

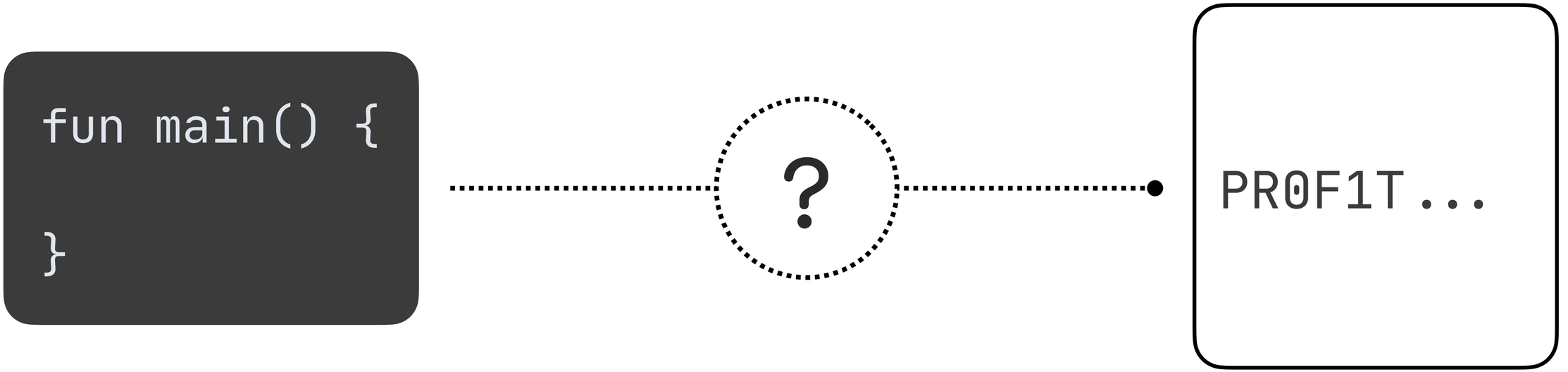
Магия расширений компилятора Kotlin

Или текущее положение вещей в мире плагинов компилятора и как нам с этим жить

Andrei Shikov

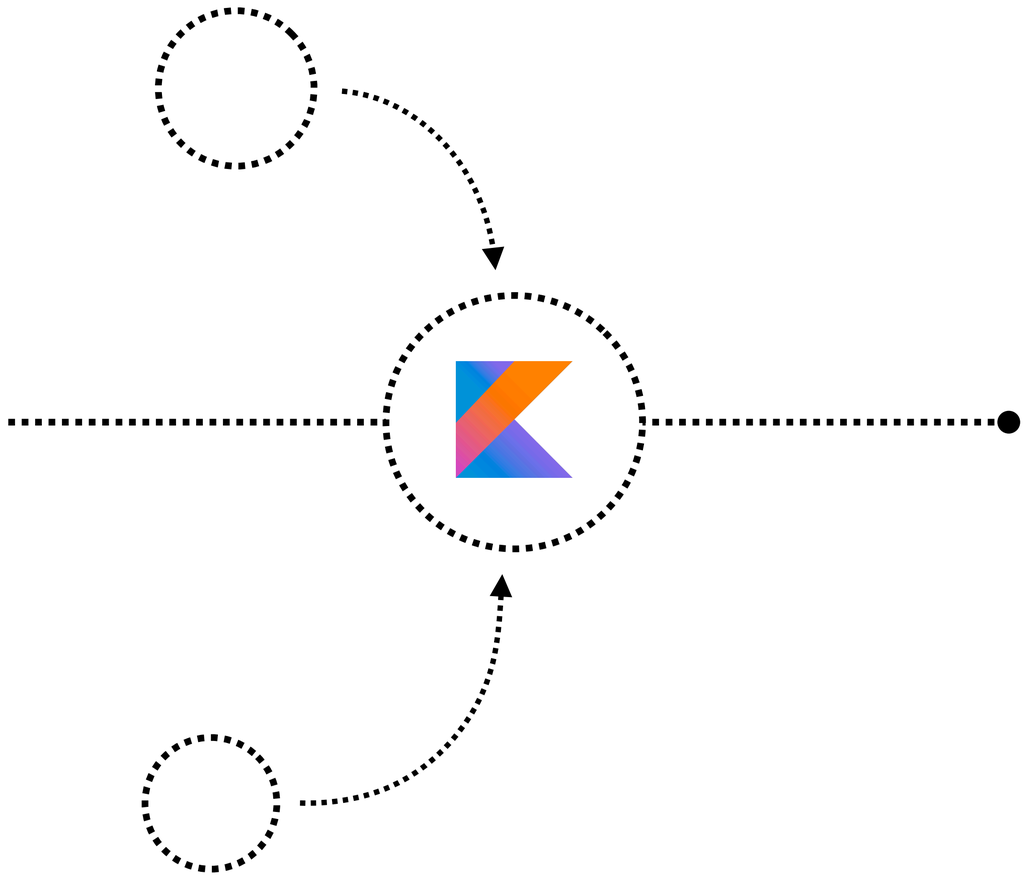

fun main() {
}
0010100...


fun main() {
}
?
PR0F1T...




```
gcc -Wall -O3 test.cc -o test
```

```
fun main() {

}
```



```
0010100...
```

Andrei does

Andrei loves

APT vs Plugins

Структура компилятора

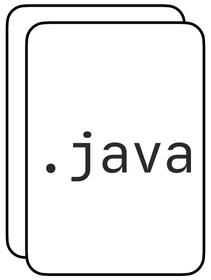
Плагины на примерах

APT vs Plugins

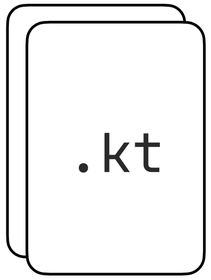
Структура компилятора

Плагины на примерах

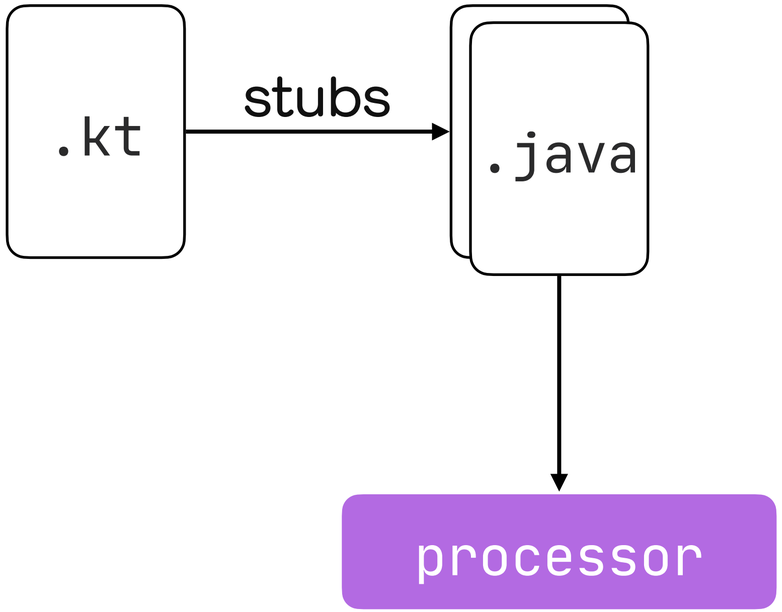
Java APT



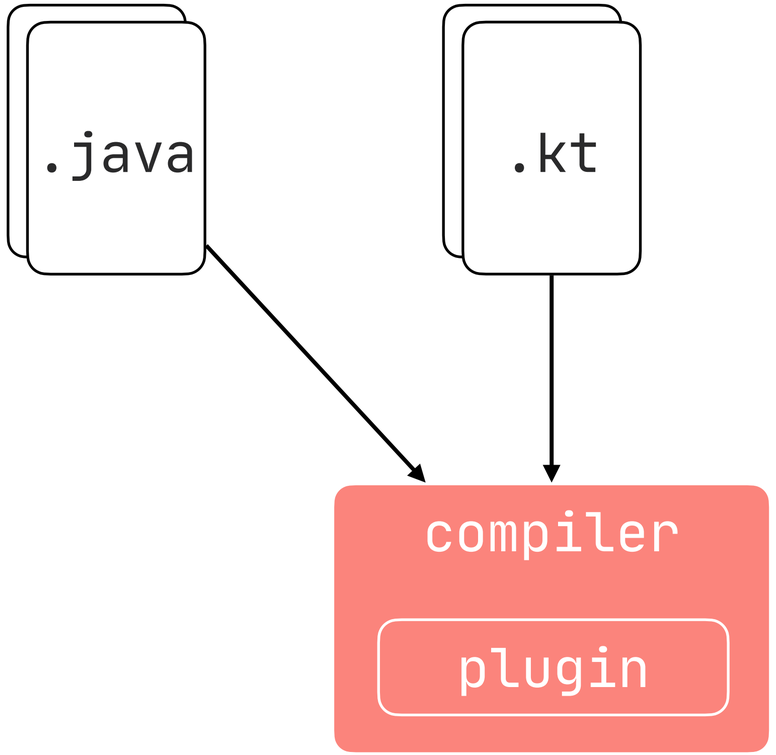
Kotlin plugins



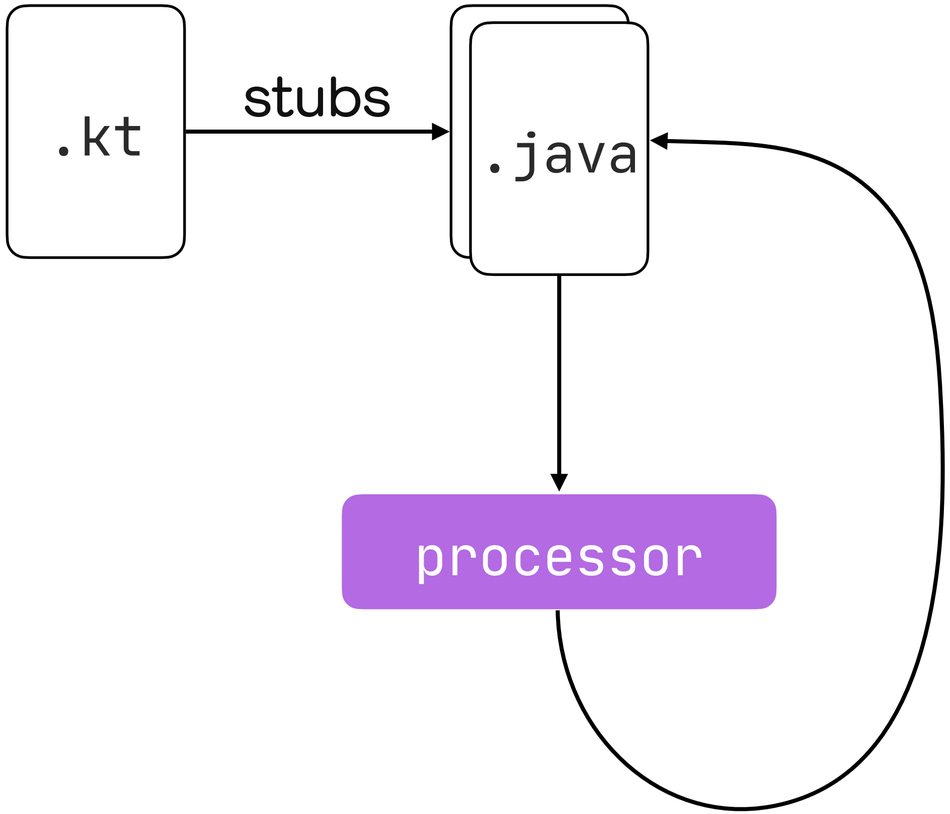
Java APT



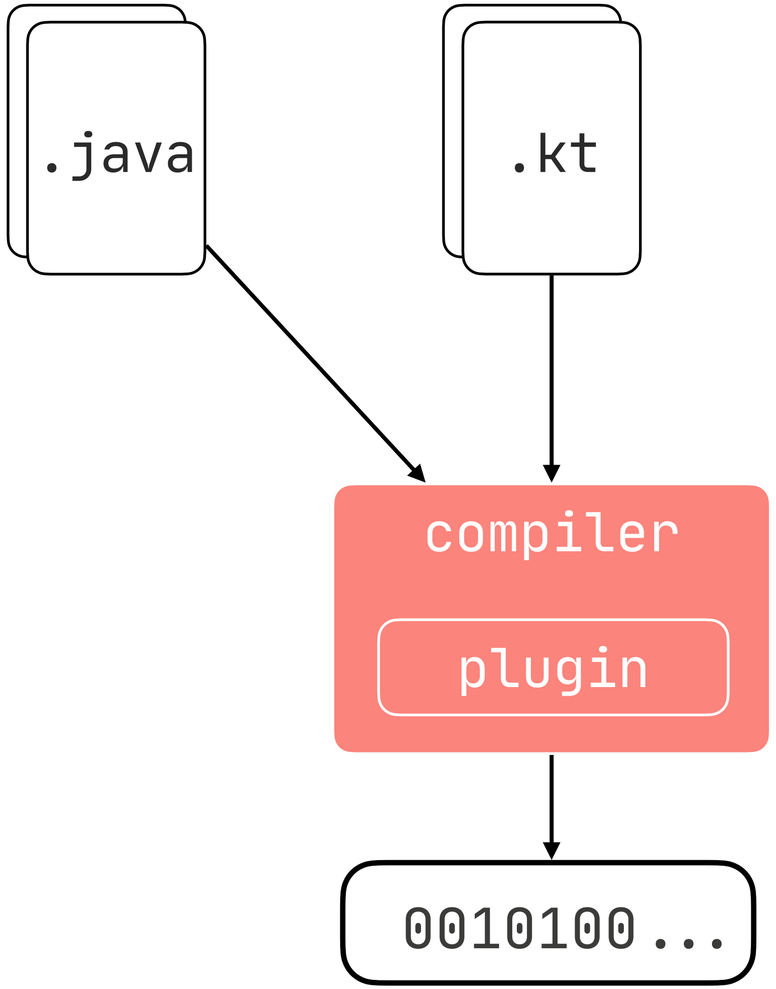
Kotlin plugins



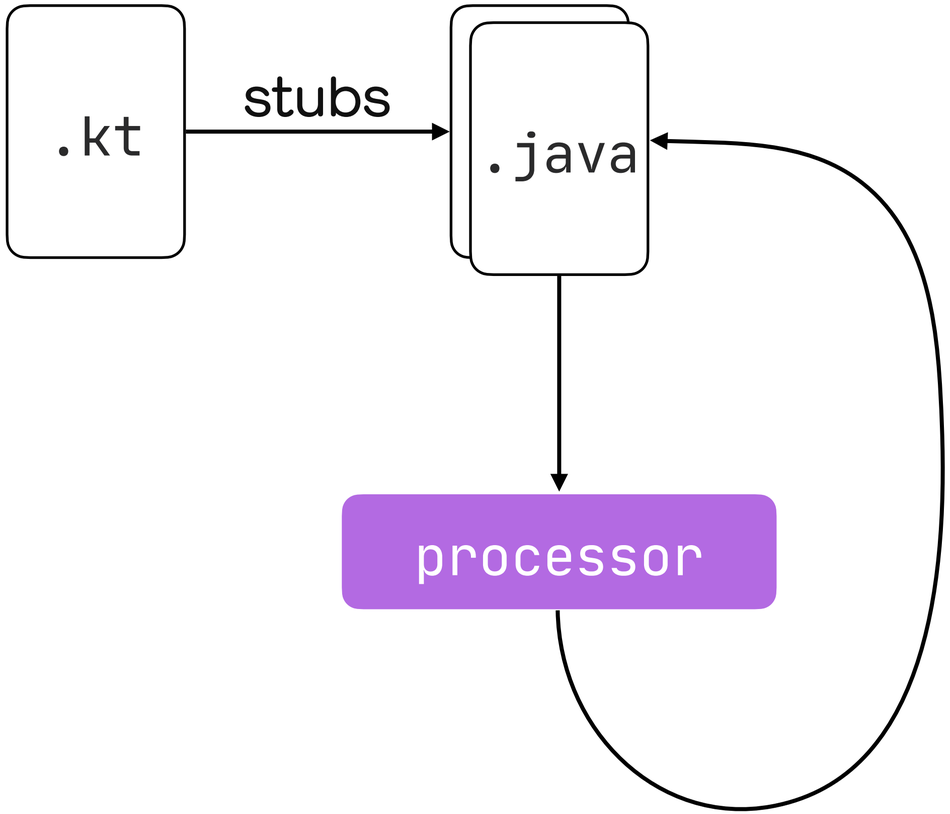
Java APT



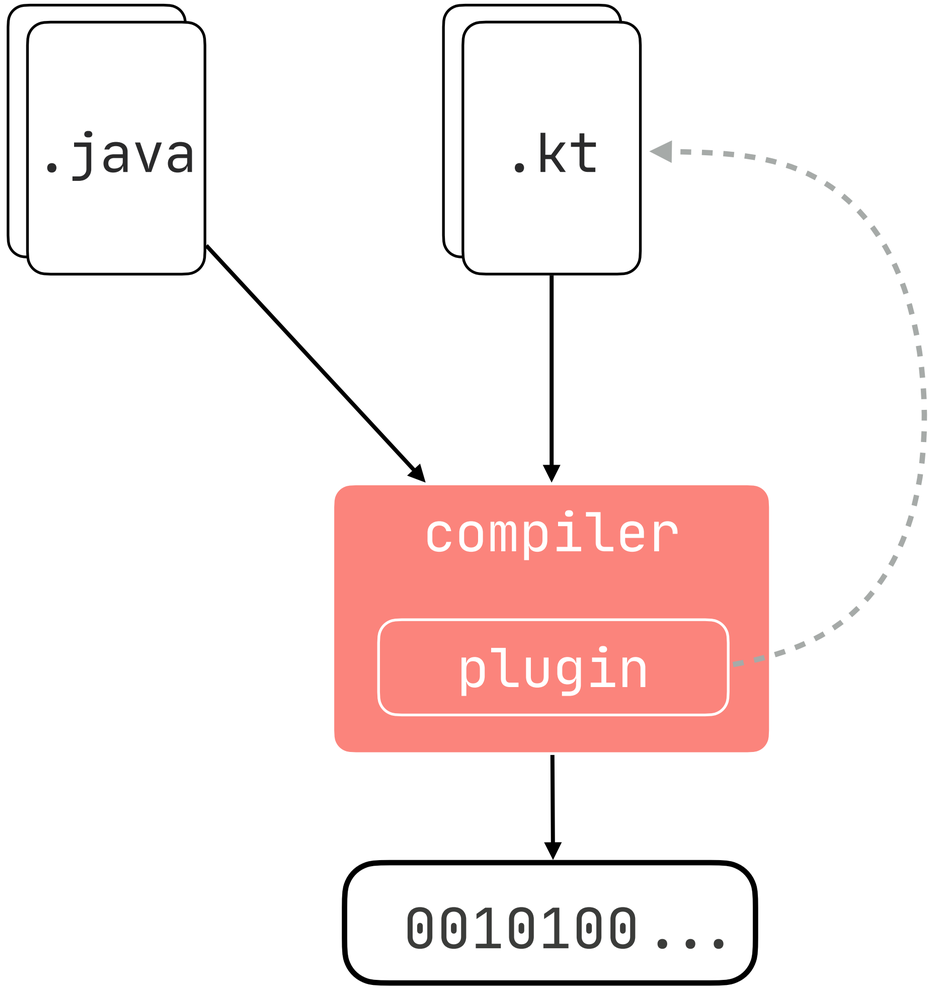
Kotlin plugins



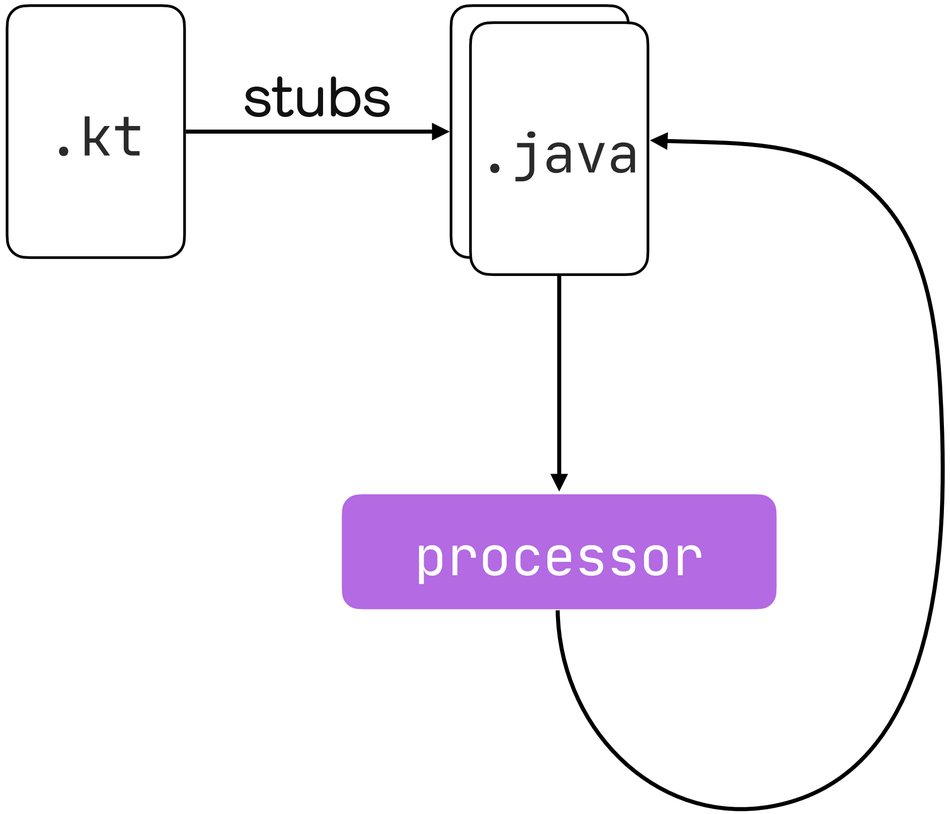
Java APT



Kotlin plugins

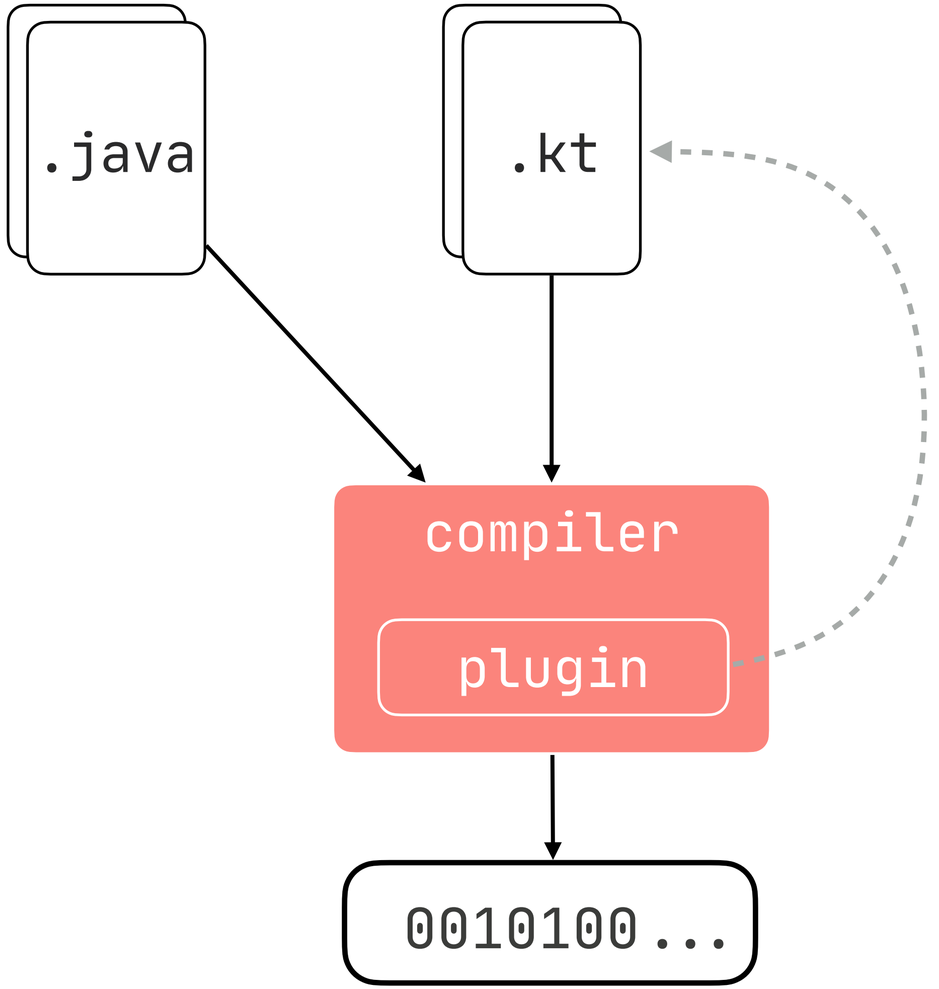


Java APT




```
/**
  * стабильное api
  * документация
  */
```

Kotlin plugins




```
/** ¯\_(ツ)_/¯ */
```

kapt = generate java stubs → java apt → compile

plugin = compile

Преимущества плагинов

Multiplatform

Изменение кода

Больше возможностей

APT vs Plugins

Структура компилятора

Плагины на примерах

```
fun main() {

}
```

0010100...

```
fun main() {

}
```

JVM

JS

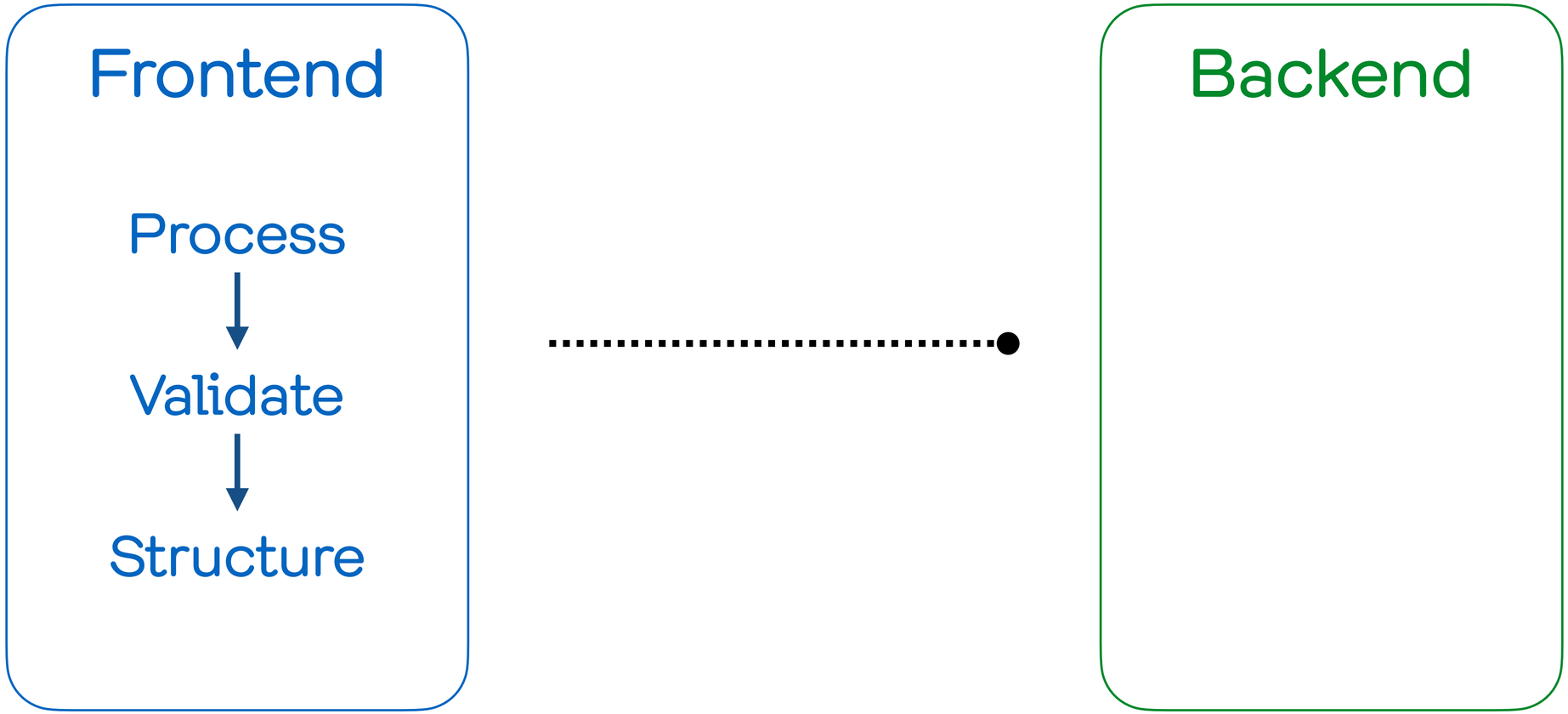
Native

Compiler

Frontend

Backend

Compiler



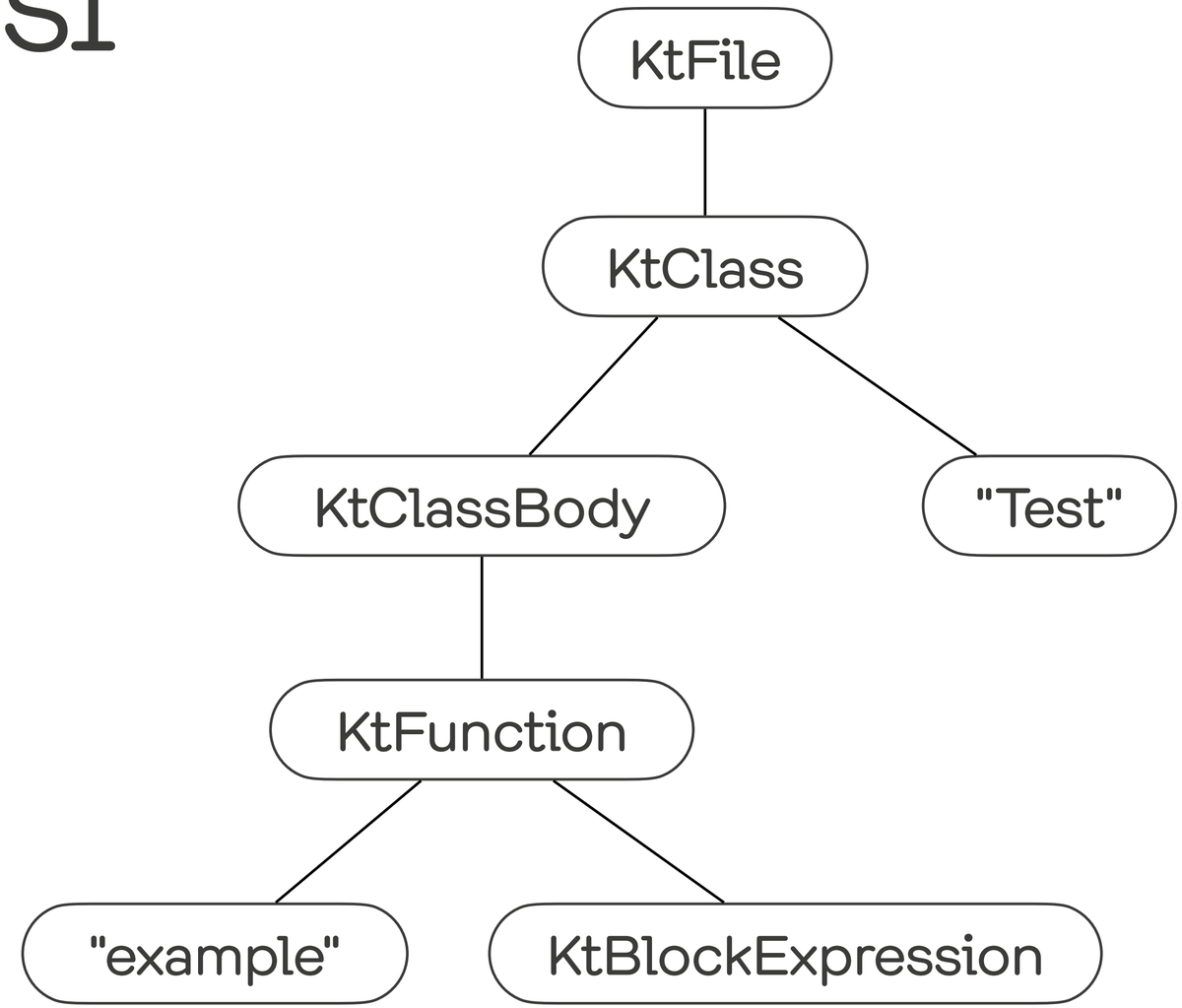
Compiler


Frontend
Process
Validate
Structure
внутреннее
представление
Backend
Optimize
Generate

```
class Test {
    fun example() {  }
}
```

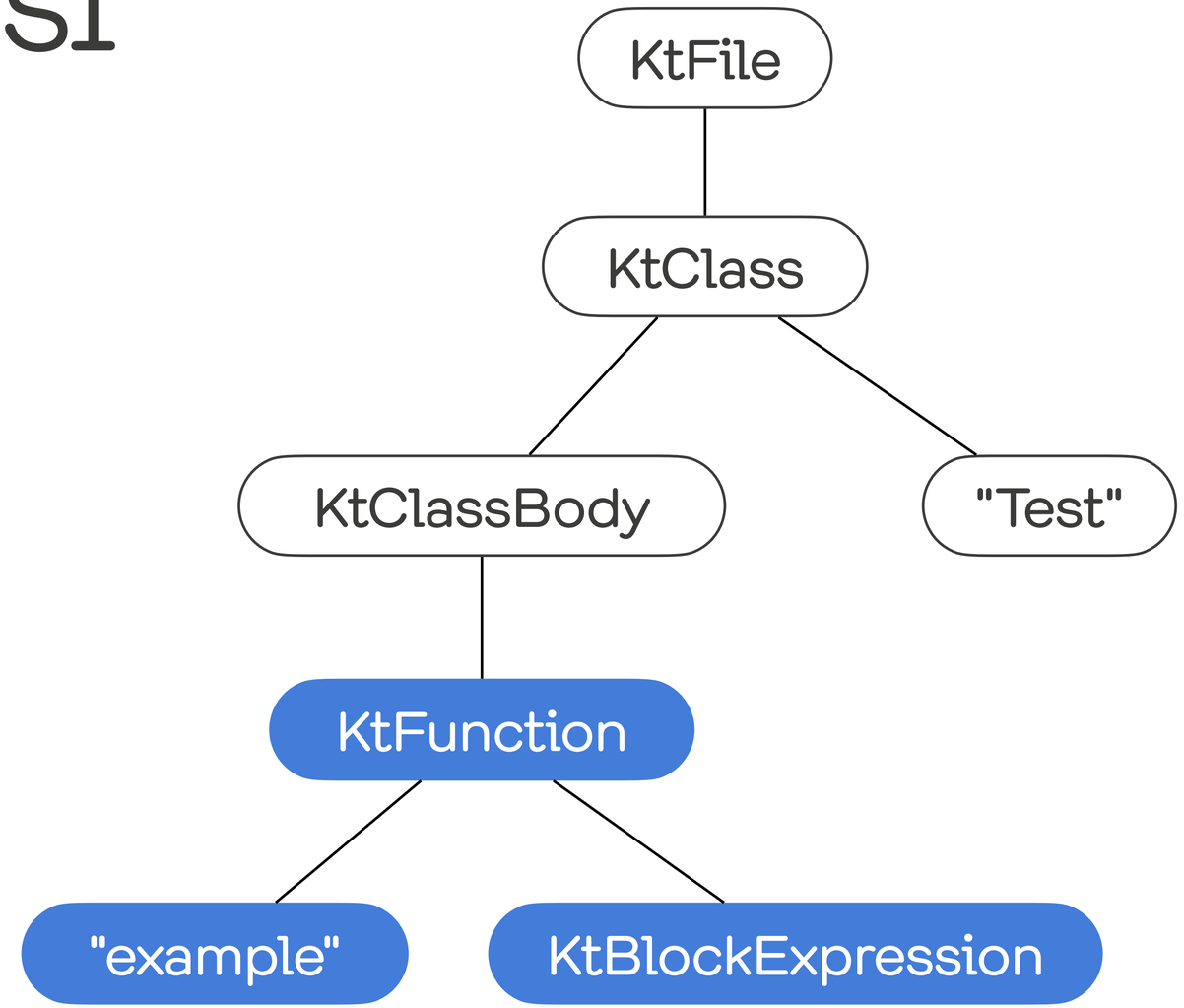
PSI

```
class Test {
    fun example() {  }
}
```



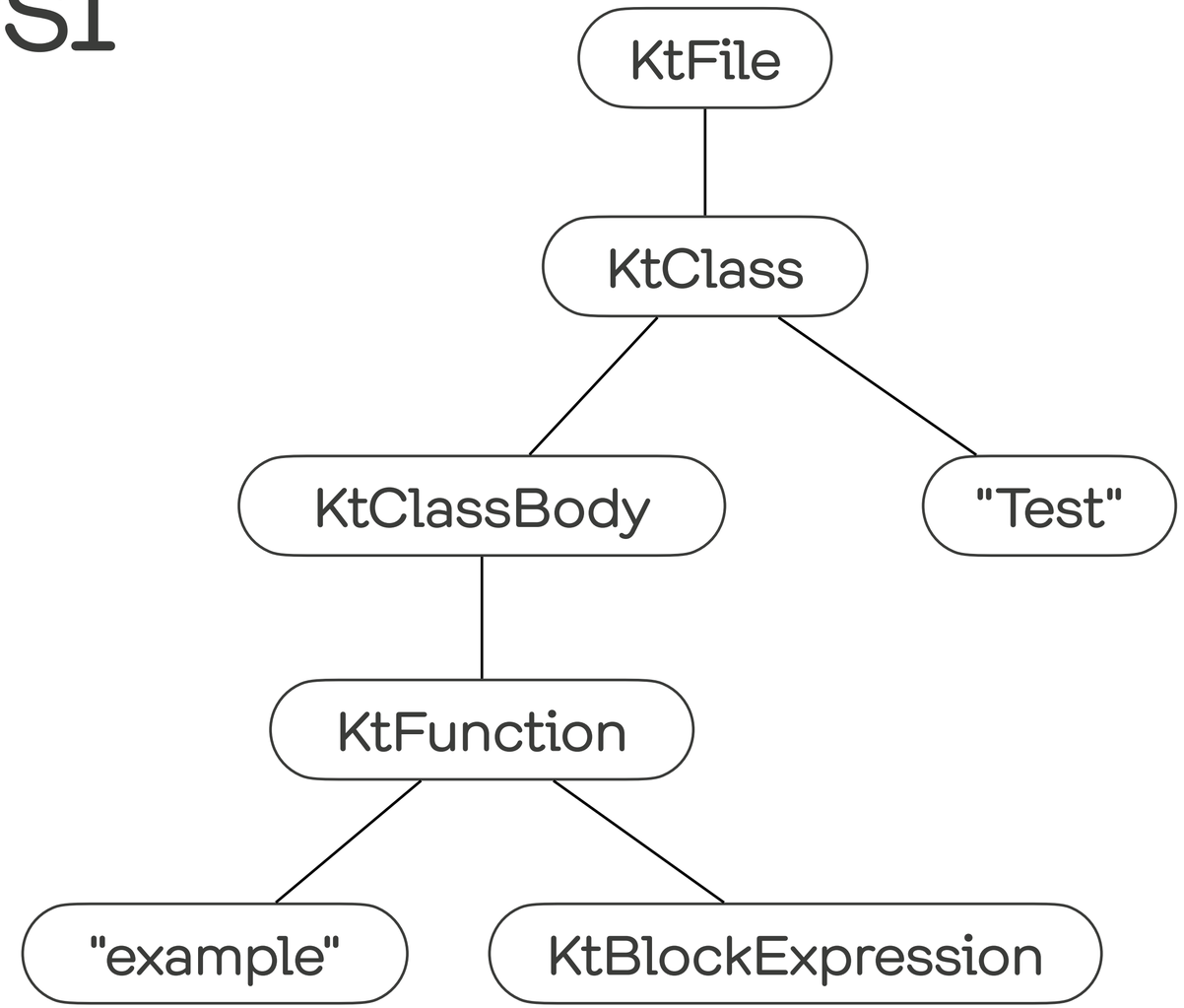
PSI

```
class Test {
    fun example() {  }
}
```



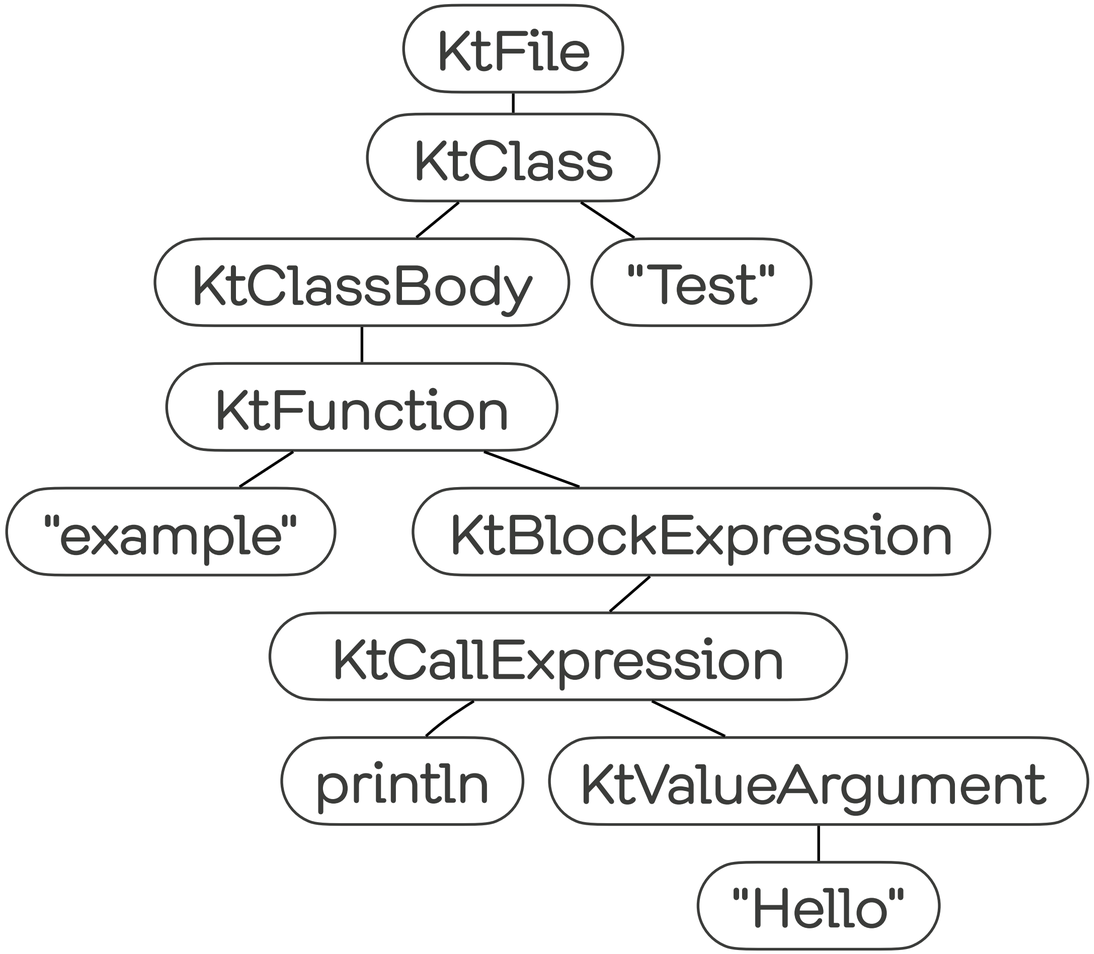
PSI

```
class Test {
    fun example() {  }
}
```

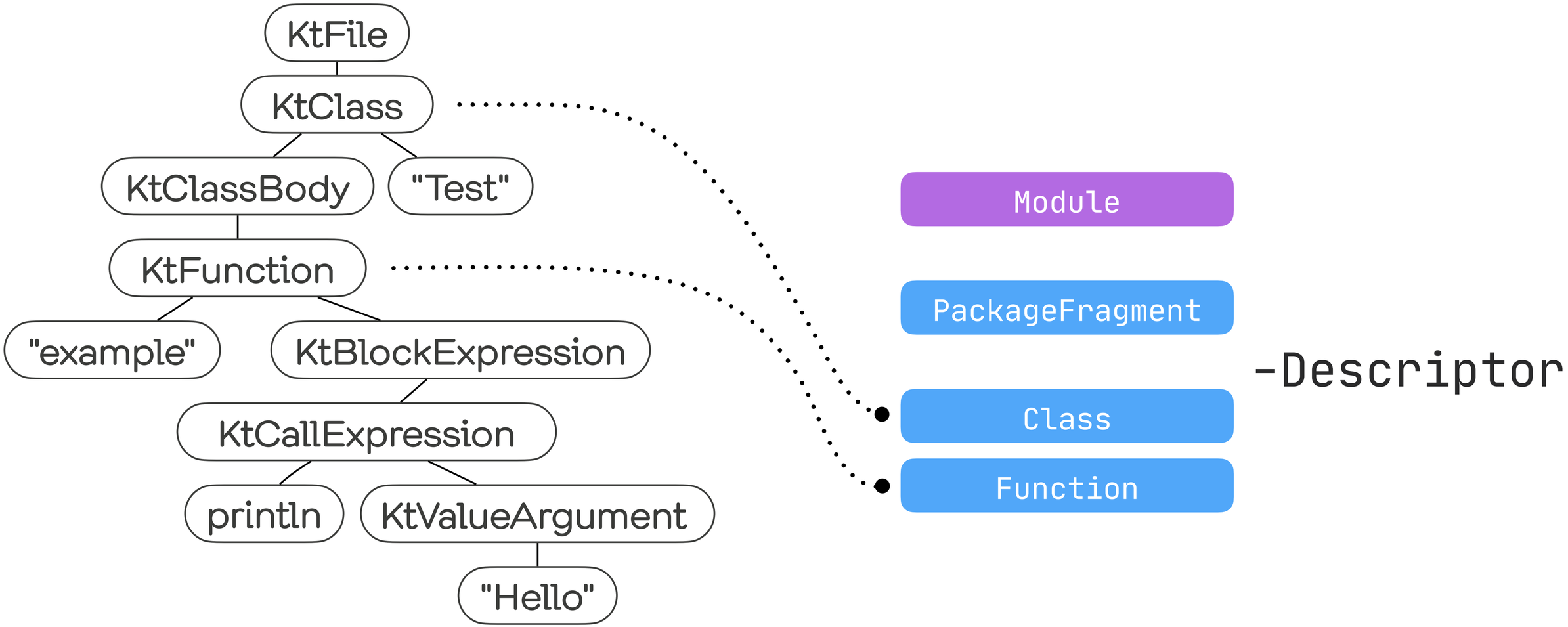


Descriptors

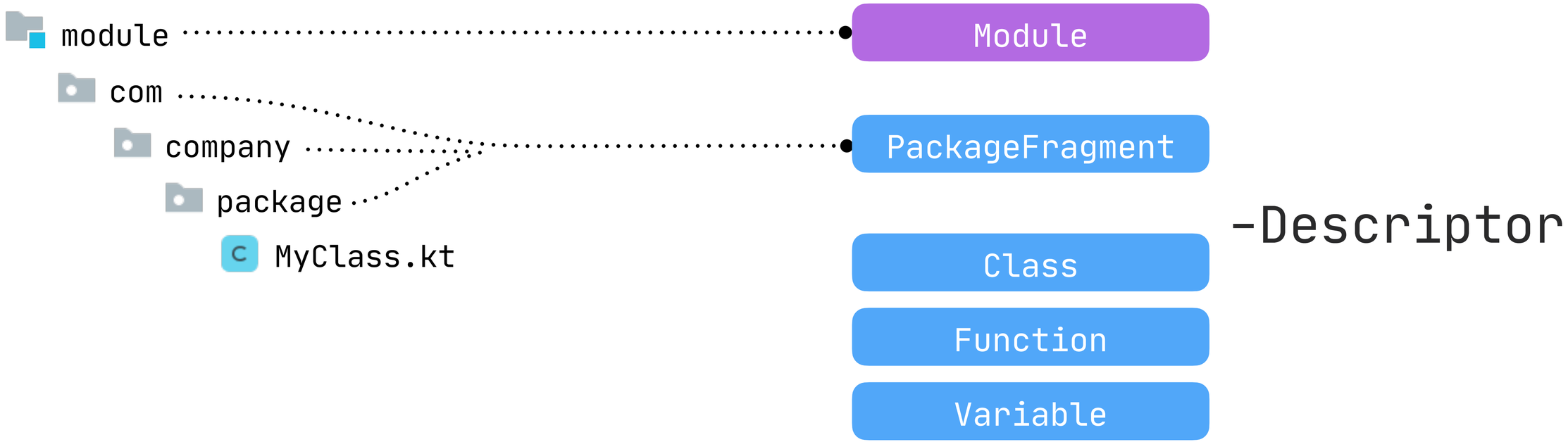
```
class Test {
    fun example() {
        println("Hello")
    }
}
```



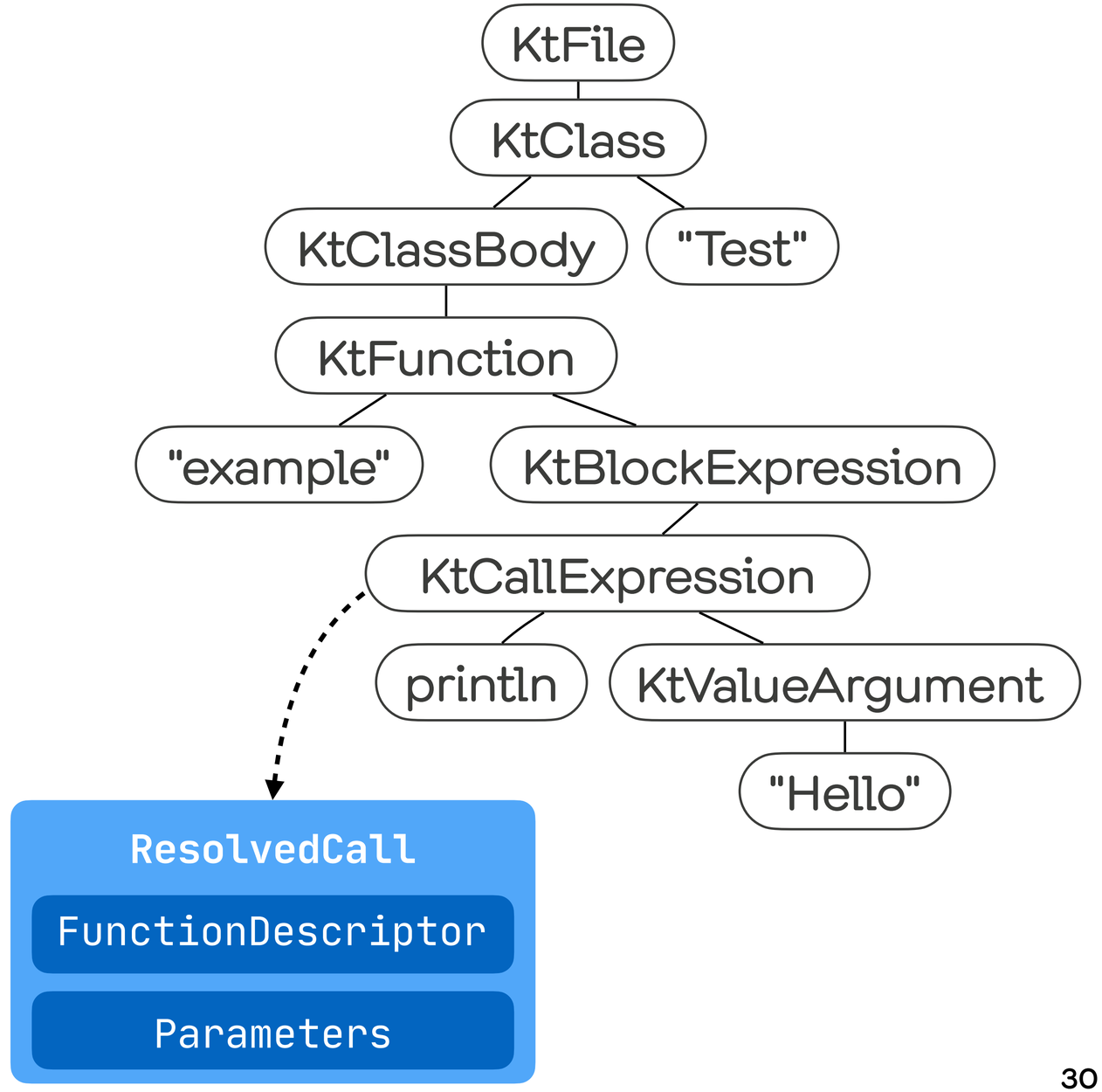
Descriptors



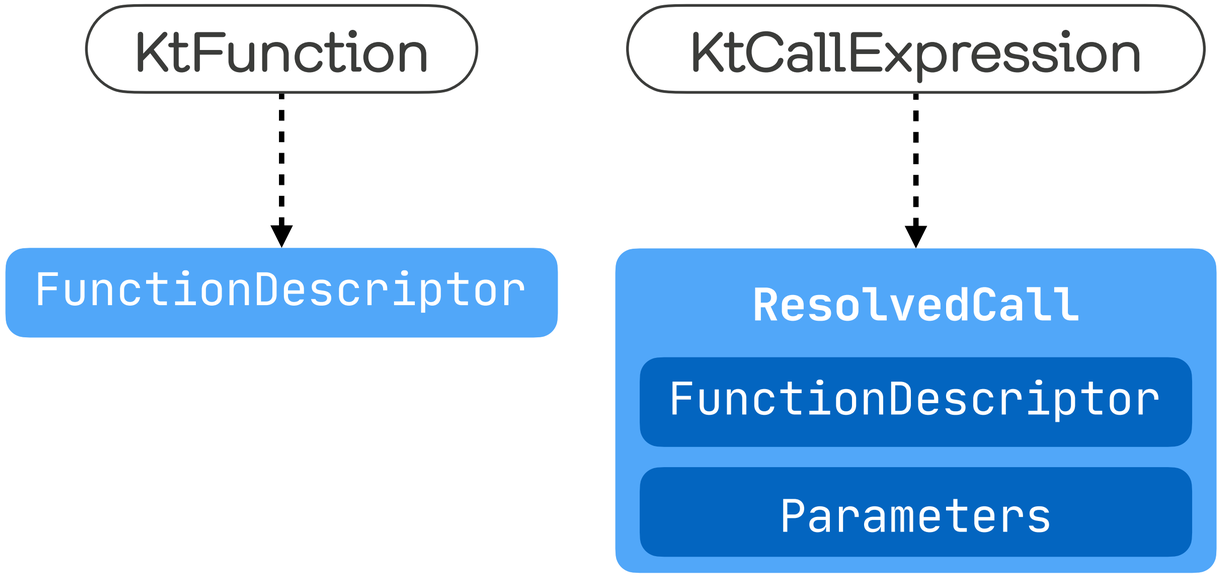
Descriptors



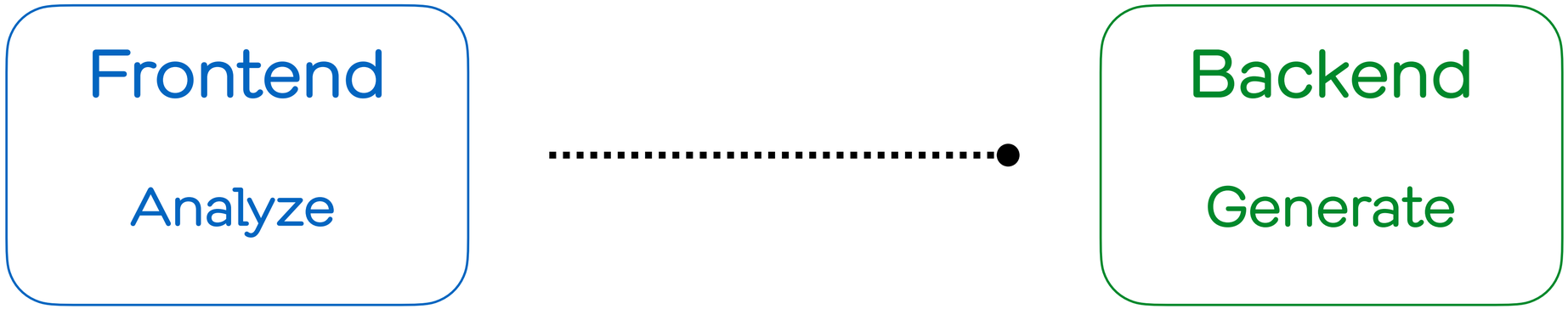
```
class Test {
    fun example() {
        println("Hello")
    }
}
```



BindingContext

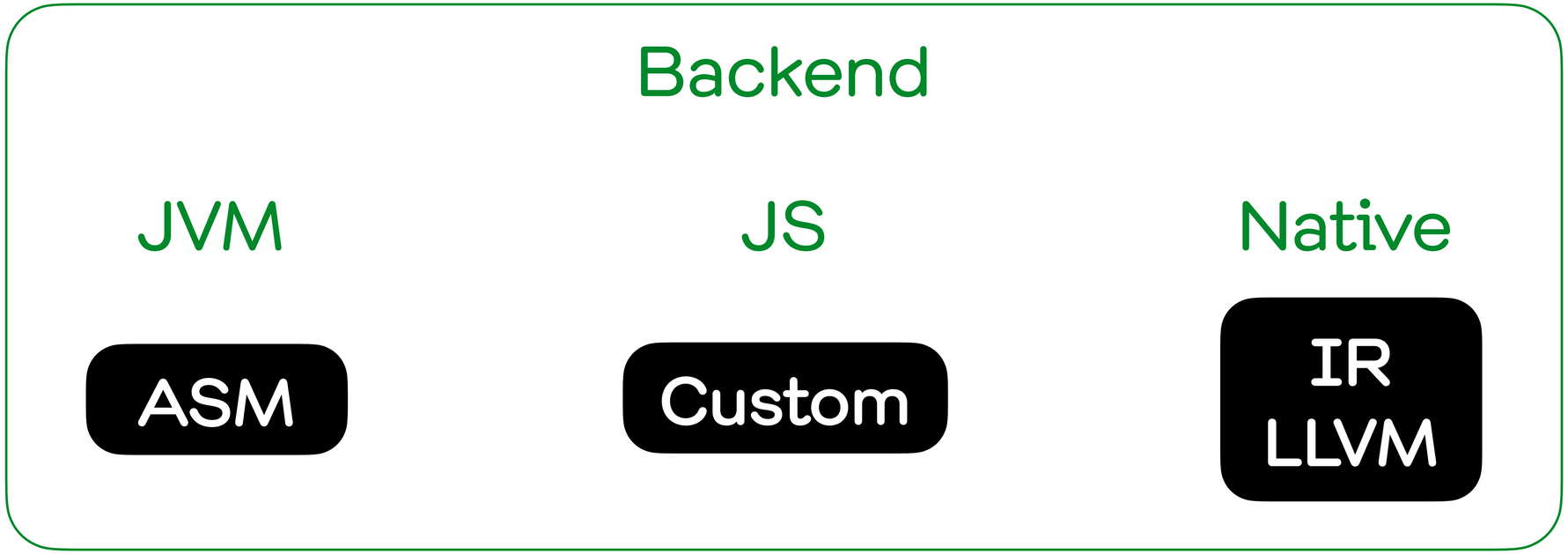


Compiler



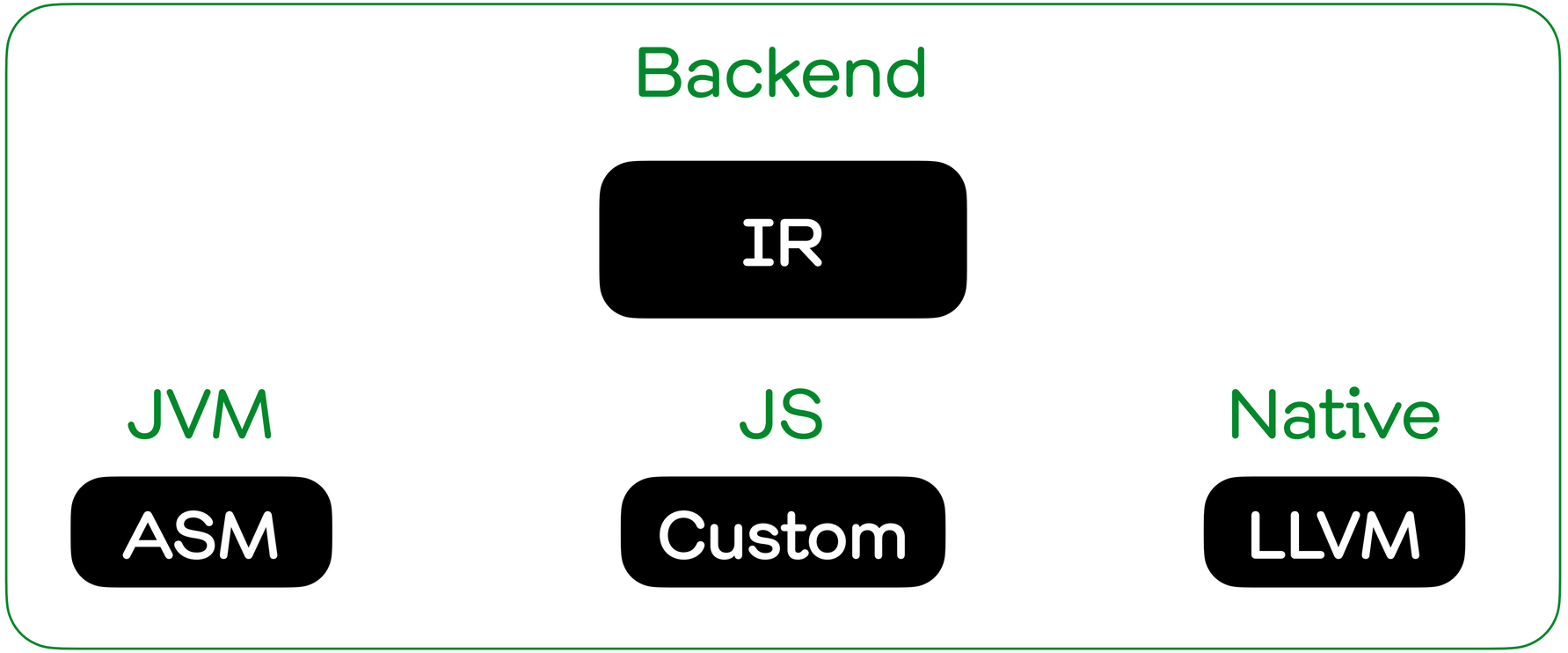
Compiler

Compiler



1.3.72

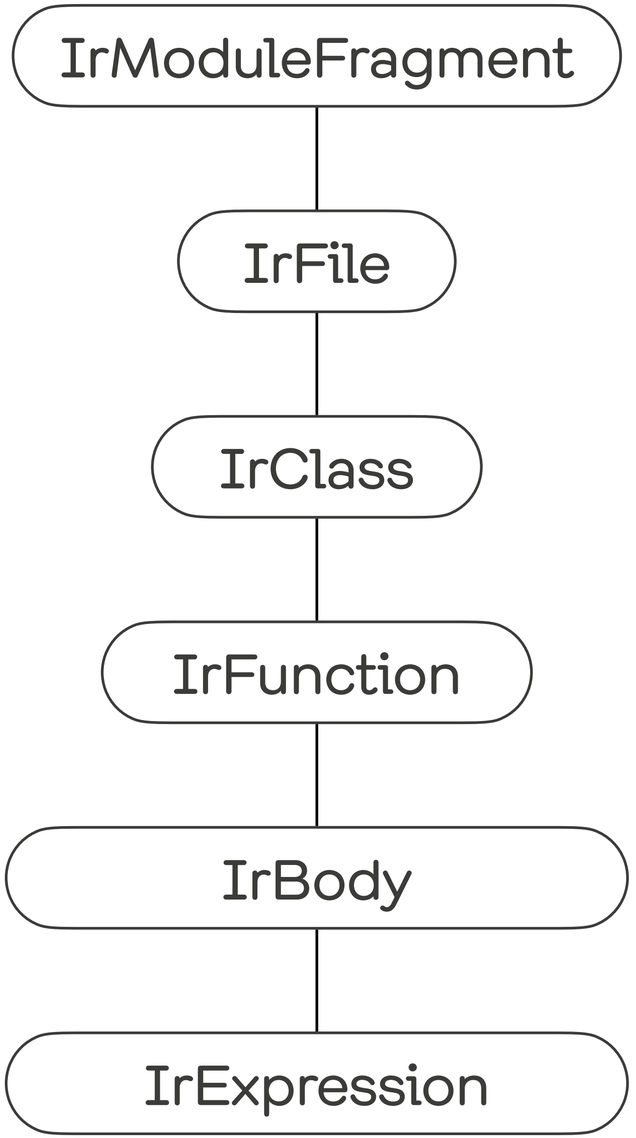
Compiler



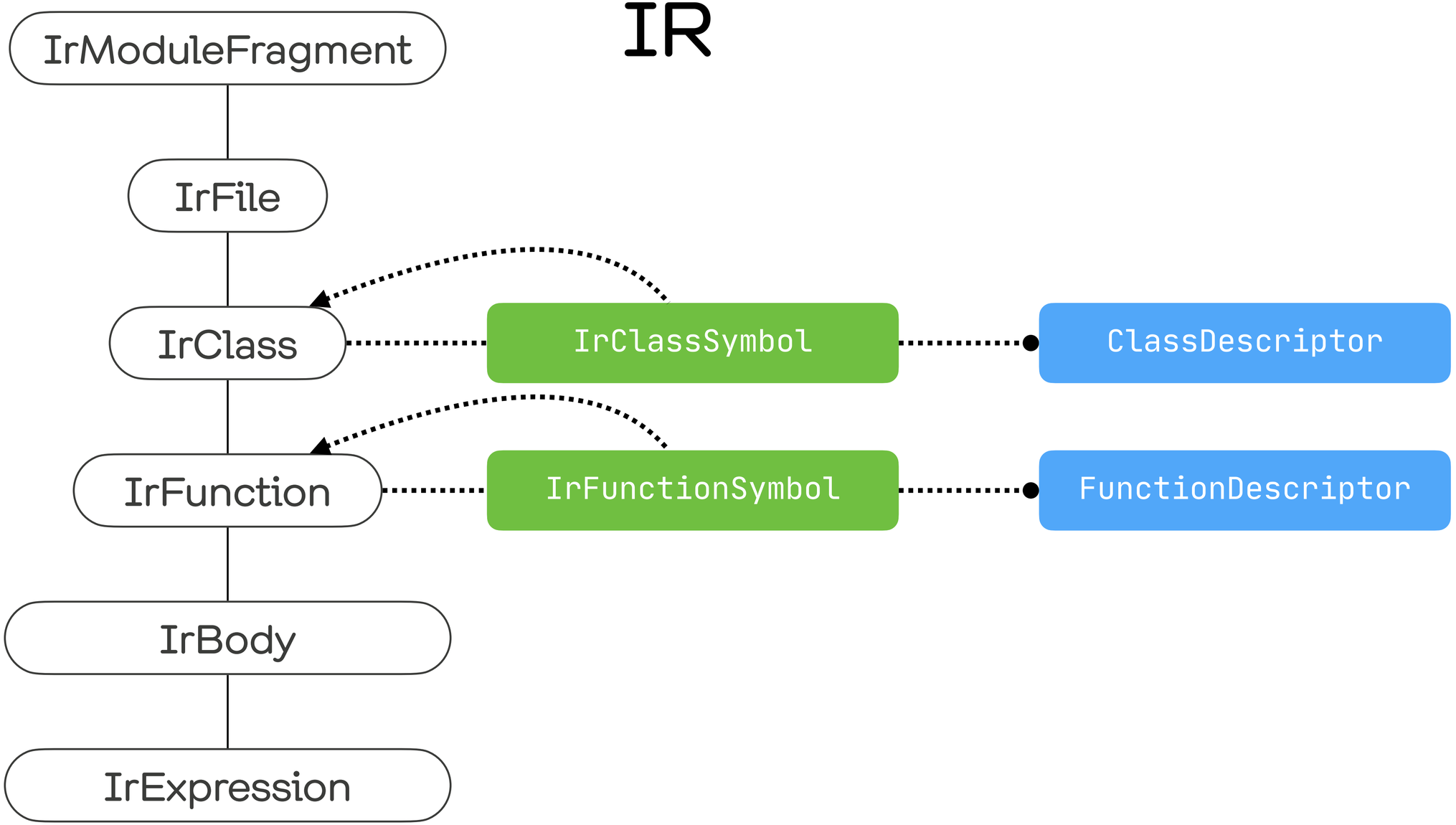
1.4

IR

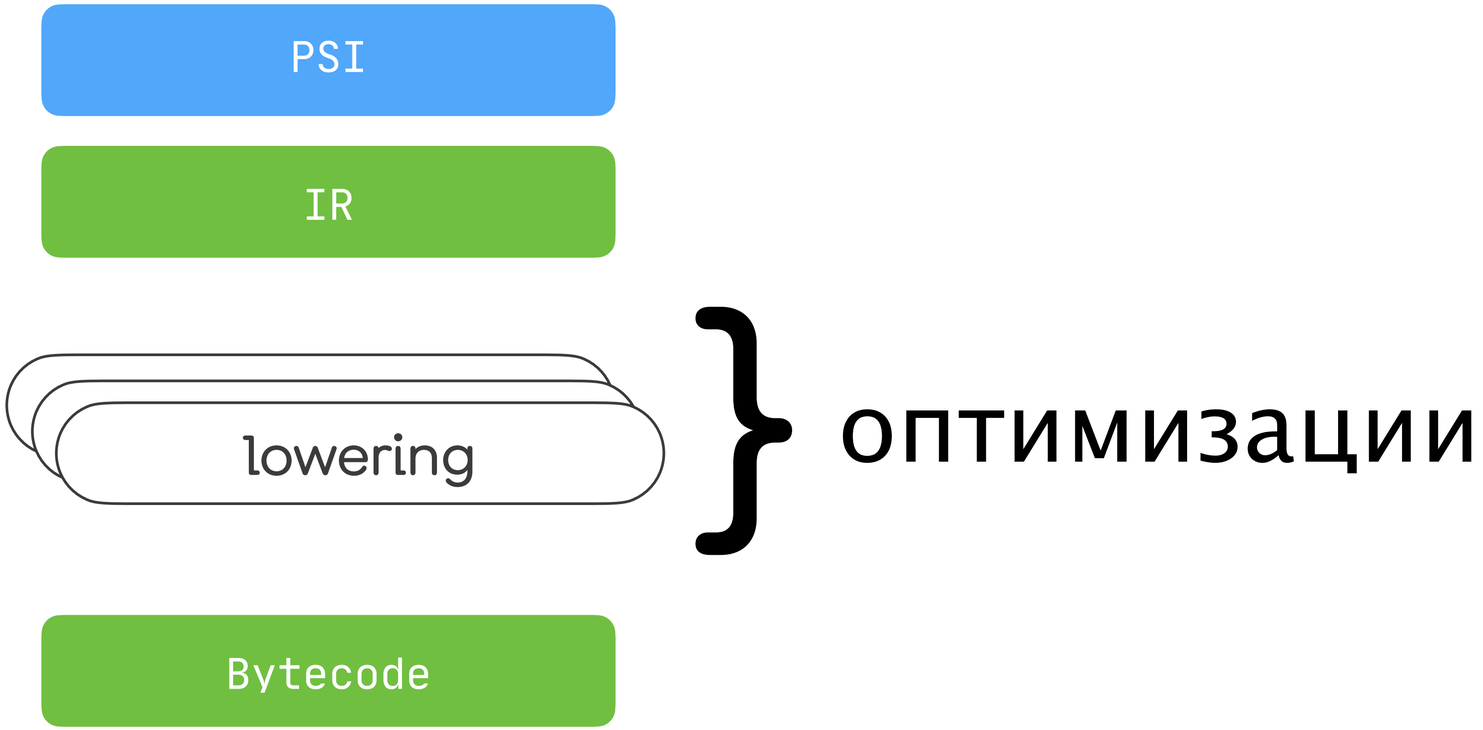
```
class Test {
    fun example() {  }
}
```



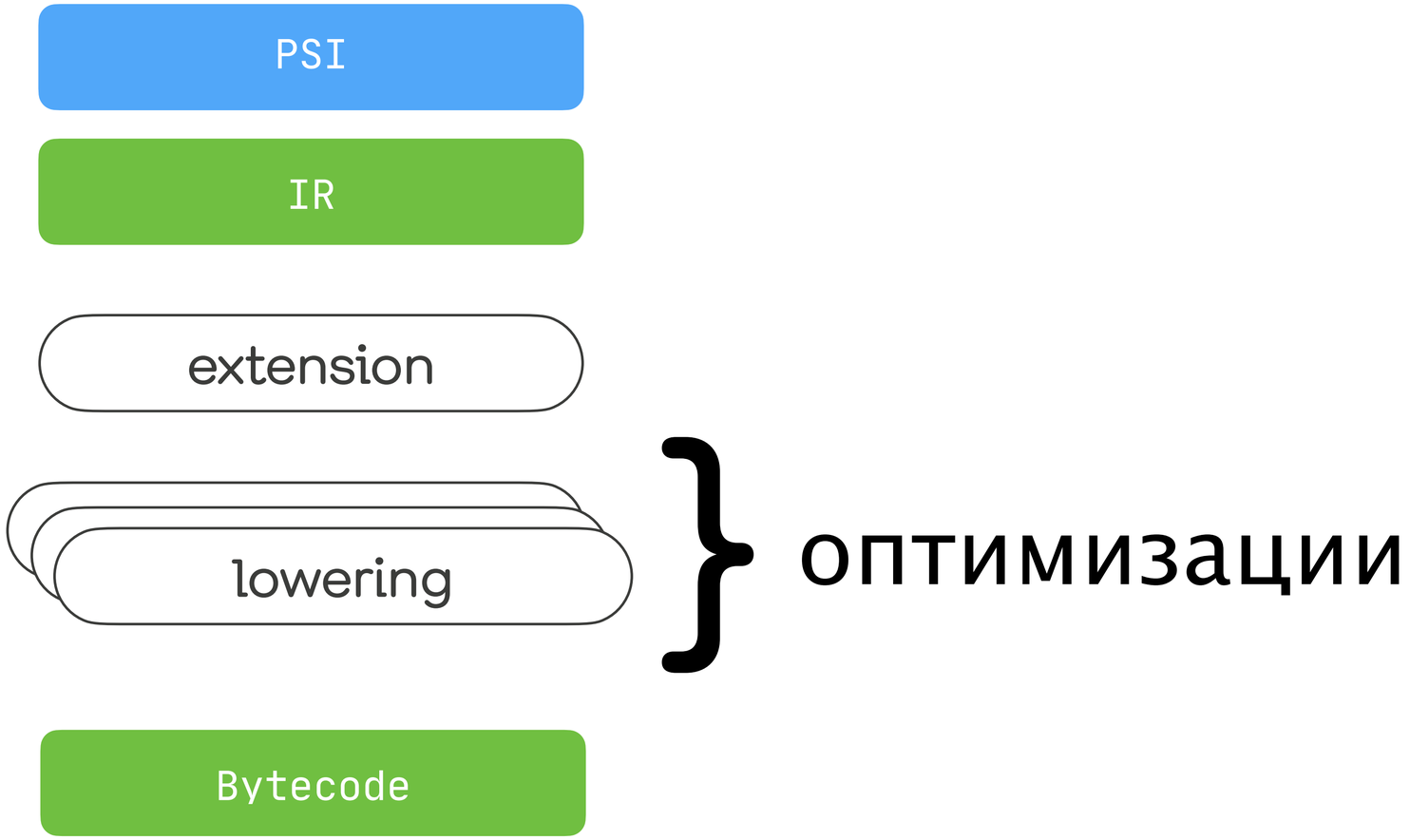
IR


IrModuleFragment
IrFile
IrClass
IrClassSymbol
ClassDescriptor
IrFunction
IrFunctionSymbol
FunctionDescriptor
IrBody
IrExpression

IR



IR



APT vs Plugins

Структура компилятора

Плагины на примерах

Экспериментальное API

```
compileKotlin {
    kotlinOptions {
        freeCompilerArgs += ["-Xplugin=your_plugin.jar"]
    }
}
```

```
compileKotlin {
    kotlinOptions {
        freeCompilerArgs += ["-Xplugin=your_plugin.jar"]
    }
}

// OR
dependencies {
    kotlinCompilerPluginClasspath 'your-plugin'
    kotlinNativeCompilerPluginClasspath 'your-plugin'
}
```

```
@AutoService(ComponentRegistrar::class)
class MyPlugin : ComponentRegistrar {
    override fun registerProjectComponents(
        project: MockProject,
        configuration: CompilerConfiguration
    ) {

    }
}
```

```kotlin
@AutoService(CommandLineProcessor::class)
class MyCLProcessor : CommandLineProcessor {
    override val pluginId: String = "me.test"
    override val pluginOptions = listOf(TEST_OPTION)

    companion object {
        val TEST_OPTION =
            CliOption(
                optionName = "testOption",
                valueDescription = "<path>",
                description = "My test option",
                required = true,
                allowMultipleOccurrences = false
            )
    }
}
```

```
-P plugin:${pluginId}:${optionName}=<value>
```

```kotlin
@AutoService(CommandLineProcessor::class)
class MyCLProcessor : CommandLineProcessor {
    ...

    override fun processOption(
        option: AbstractCliOption,
        value: String,
        configuration: CompilerConfiguration
    ) {
        if (option == TEST_OPTION) {
            configuration.put(MY_KEY, File(option.value))
        }
    }

    companion object {
        val MY_KEY = CompilerConfigurationKey.create<File>("myKey")
    }
}
```

```
open class ProjectExtensionDescriptor<T> {
    fun registerExtension(
        project: Project,
        extension: T
    )
}



interface SomeExtension {
    companion object : ProjectExtensionDescriptor<SomeExtension>()
}
```

```
interface SomeExtension {
    companion object : ProjectExtensionDescriptor<SomeExtension>()
}
```

```
class MyPlugin : ComponentRegistrar {
    override fun registerProjectComponents(
        project: MockProject,
        configuration: CompilerConfiguration
    ) {
        SomeExtension.registerExtension(
            project,
            SomeExtensionImpl(configuration[MY_KEY])
        )
    }
}
```

Примеры

kotlinx-serialization

compose

kapt in compiler

kotlinx-serialization

compose

kapt in compiler

```
@Serializable
data class Data(val a: Int, val b: String = "42")
```

```
@Serializable
data class Data(val a: Int, val b: String = "42") {
    companion object {
        fun serializer(): KSerializer<Data> = $serializer
    }

    @Deprecated(level = DeprecationLevel.HIDDEN)
    object $serializer : GeneratedSerializer<Data> {
        override fun childSerializers(): Array<KSerializer<*>> =
            arrayOf(IntSerializer, StringSerializer)

        override fun serialize() = ...
    }
}
```

Add serializer	Validate it is correct	Generate serializer code
		`ExpressionCodegen`
`SyntheticResolve`	`DeclarationChecker`	`IrGeneration`
		`JsSyntheticTranslate`

Add serializer

Validate it is correct

Generate serializer code

`SyntheticResolve`

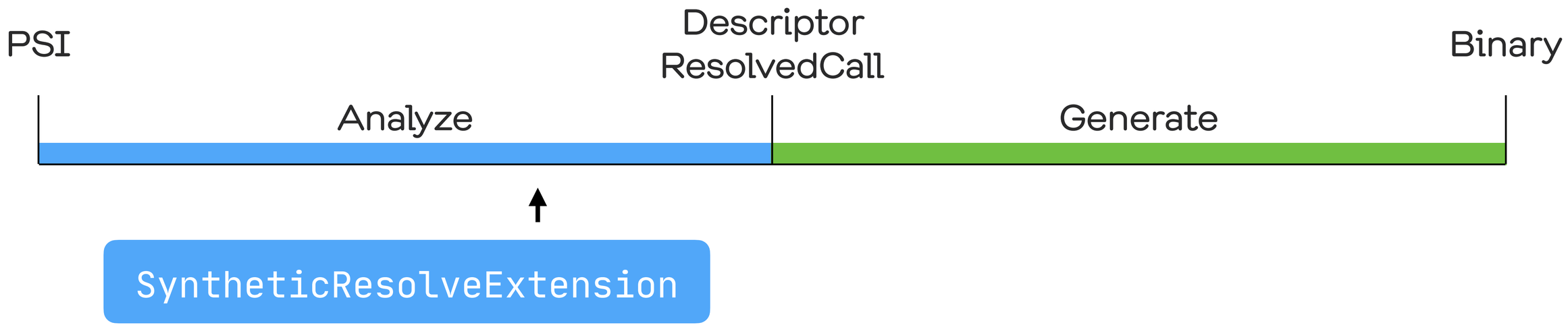
`DeclarationChecker`

`ExpressionCodegen`

`IrGeneration`

`JsSyntheticTranslate`

SyntheticResolve



```
interface SyntheticResolveExtension {

    fun getSyntheticNestedClassNames(thisDescriptor: ClassDescriptor): List<Name>
    fun generateSyntheticClasses(
        thisDescriptor: ClassDescriptor,
        name: Name,
        ...
        result: MutableSet<ClassDescriptor>
    ) { }

    fun getSyntheticFunctionNames(thisDescriptor: ClassDescriptor): List<Name>
    fun generateSyntheticMethods(
        thisDescriptor: ClassDescriptor,
        name: Name,
        ...
        result: MutableSet<SimpleFunctionDescriptor>
    ) { }

    ...
}
```

```kotlin
@Serializable
data class Data(val a: Int, val b: String = "42") {

    // generated

    companion object {
        fun serializer(): KSerializer<Data> = $serializer
    }

    private object $serializer : KSerializer<Data> { .. }
}
```

```
// companion object {
//     fun serializer(): KSerializer<Data>
// }

class SerializationResolveExtension : SyntheticResolveExtension {
    override fun getSyntheticFunctionNames(thisDescriptor: ClassDescriptor): List<Name> =
        if (thisDescriptor.isCompanionObject && thisDescriptor.isSerializableCompanion) {
            listOf(Name.identifier("serializer"))
        } else {
            emptyList()
        }
}
```

```
// companion object {
//     fun serializer(): KSerializer<Data>
// }

class SerializationResolveExtension : SyntheticResolveExtension {
    override fun getSyntheticFunctionNames(thisDescriptor: ClassDescriptor): List<Name> =
        if (thisDescriptor.isCompanionObject && thisDescriptor.isSerializableCompanion) {
            listOf(Name.identifier("serializer"))
        } else {
            emptyList()
        }

    val ClassDescriptor.isSerializableCompanion get() =
        isCompanionObject &&
            (containingDeclaration as ClassDescriptor).hasSerializableAnnotation

    val ClassDescriptor.hasSerializableAnnotation get() =
        hasAnnotation(FqName("kotlinx.serialization.Serializable"))
}
```

```
// companion object {
//     fun serializer(): KSerializer<Data>
// }

class SerializationResolveExtension : SyntheticResolveExtension {
    override fun generateSyntheticMethods(
        thisDescriptor: ClassDescriptor,
        name: Name,
        bindingContext: BindingContext,
        fromSupertypes: List<SimpleFunctionDescriptor>,
        result: MutableCollection<SimpleFunctionDescriptor>
    ) {
        val classDescriptor = getSerializableForCompanion(thisDescriptor) ?: return

        if (name.asString() == "serializer") {
            result.add(createSerializerGetterDescriptor(thisDescriptor, classDescriptor))
        }
    }

}
```

```
// companion object {
//     fun serializer(): KSerializer<Data>
// }

fun createSerializerGetterDescriptor(
    companionClass: ClassDescriptor,
    serializableClass: ClassDescriptor
): SimpleFunctionDescriptor {
    val f = SimpleFunctionDescriptorImpl.create(
        ...
    )

    val returnType = ...
    f.initialize(
       ...
    )
    return f
}
```

```
// companion object {
//     fun serializer(): KSerializer<Data>
// }

private fun createSerializerGetterDescriptor(
    companionClass: ClassDescriptor,
    serializableClass: ClassDescriptor
): SimpleFunctionDescriptor {
    val f = SimpleFunctionDescriptorImpl.create(
        companionClass,
        Annotations.EMPTY,
        Name.identifier("serializer"),
        CallableMemberDescriptor.Kind.SYNTHESIZED,
        companionClass.source
    )

    val returnType = ...
    f.initialize(
       ...
    )
    return f
}
```

```
// fun serializer(): KSerializer<Data>

private fun createSerializerGetterDescriptor(
    companionClass: ClassDescriptor,
    serializableClass: ClassDescriptor
): SimpleFunctionDescriptor {
    val f = SimpleFunctionDescriptorImpl.create(...)

    val returnType = KotlinTypeFactory.simpleNotNullType(...) // KSerializer<Data>

    f.initialize(
       /* extensionReceiver */ null,
       /* dispatcherReceiver */ companionClass.thisAsReceiverParameter,
       /* typeArgs */ emptyList(),
       /* valueArgs */ emptyList(),
       returnType,
       Modality.FINAL,
       Visibilities.PUBLIC
    )
    return f
}
```

Add serializer

Validate it is correct

Generate serializer code

`ExpressionCodegen`

`SyntheticResolve`

`DeclarationChecker`

`IrGeneration`

`JsSyntheticTranslate`

Add serializer

Validate it is correct

Generate serializer code

`ExpressionCodegen`

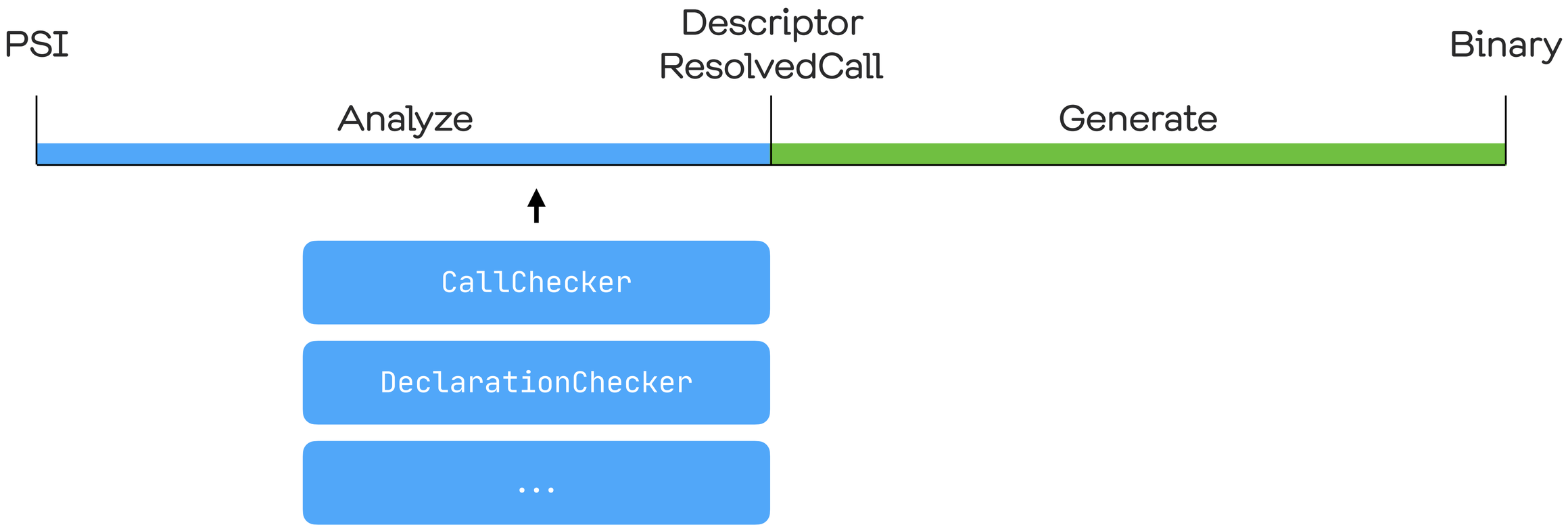
`SyntheticResolve`

`DeclarationChecker`

`IrGeneration`

`JsSyntheticTranslate`

Checkers


PSI
Descriptor
ResolvedCall
Binary
Analyze
Generate
CallChecker
DeclarationChecker
...

```
// ComponentRegistrar

StorageComponentContainerContributor.registerExtension(
    project,
    object : StorageComponentContainerContributor {
        override fun registerModuleComponents(
            container: StorageComponentContainer,
            platform: TargetPlatform,
            moduleDescriptor: ModuleDescriptor
        ) {
            container.useInstance(SerializationDeclarationChecker())
        }
    }
}
```

```
// ComponentRegistrar

StorageComponentContainerContributor.registerExtension(
    project,
    object : StorageComponentContainerContributor {
        override fun registerModuleComponents(
            container: StorageComponentContainer,
            platform: TargetPlatform,
            moduleDescriptor: ModuleDescriptor
        ) {
            container.useInstance(SerializationDeclarationChecker())
        }
    }
}
```

```
class SerializationDeclarationChecker : DeclarationChecker {
    override fun check(
        declaration: KtDeclaration,
        descriptor: DeclarationDescriptor,
        context: DeclarationCheckerContext
    ) {
        if (descriptor !is ClassDescriptor) return
        if (!descriptor.hasSerializableAnnotation) return

        if (descriptor.isInline) {
            trace.reportOnSerializableAnnotation(
                descriptor,
                SerializationErrors.INLINE_CLASSES_NOT_SUPPORTED
            )
            return
        }
    }
}
```

```
class SerializationDeclarationChecker : DeclarationChecker {
    override fun check(
        declaration: KtDeclaration,
        descriptor: DeclarationDescriptor,
        context: DeclarationCheckerContext
    ) {
        if (descriptor !is ClassDescriptor) return
        if (!descriptor.hasSerializableAnnotation) return

        if (descriptor.isInline) {
            trace.reportOnSerializableAnnotation(
                descriptor,
                SerializationErrors.INLINE_CLASSES_NOT_SUPPORTED
            )
            return
        }
    }
}
```

```
val INLINE_CLASSES_NOT_SUPPORTED =
    DiagnosticFactory0.create<PsiElement>(Severity.ERROR)

object SerializationPluginErrorsRendering : DefaultErrorMessages.Extension {
    private val _map = DiagnosticFactoryToRendererMap("SerializationPlugin")
    override fun getMap(): DiagnosticFactoryToRendererMap = _map

    init {
        _map.put(
            INLINE_CLASSES_NOT_SUPPORTED,
            "Inline classes are not supported by kotlinx.serialization yet"
        )
    }
}
```

```
val INLINE_CLASSES_NOT_SUPPORTED =
    DiagnosticFactory0.create<PsiElement>(Severity.ERROR)
```

```
object SerializationPluginErrorsRendering : DefaultErrorMessages.Extension {
    private val _map = DiagnosticFactoryToRendererMap("SerializationPlugin")
    override fun getMap(): DiagnosticFactoryToRendererMap = _map

    init {
        _map.put(
            INLINE_CLASSES_NOT_SUPPORTED,
            "Inline classes are not supported by kotlinx.serialization yet"
        )
        _map.put(
             SERIALIZER_NOT_FOUND,
             "Serializer has not been found for type ''{0}''.",
             Renderers.RENDER_TYPE_WITH_ANNOTATIONS
         )
    }
}
```

```
private fun BindingTrace.reportOnSerializableAnnotation(
    descriptor: ClassDescriptor,
    error: DiagnosticFactory0<in KtAnnotationEntry>
) {
    descriptor.findSerializableAnnotation()?.let {
        reportFromPlugin(
            error.on(it),
            SerializationPluginErrorsRendering
        )
    }
}
```

```
e: test.kt: (5, 1): Inline classes are not supported by kotlinx.serialization yet
```

Add serializer

Validate it is correct

Generate serializer code

`ExpressionCodegen`

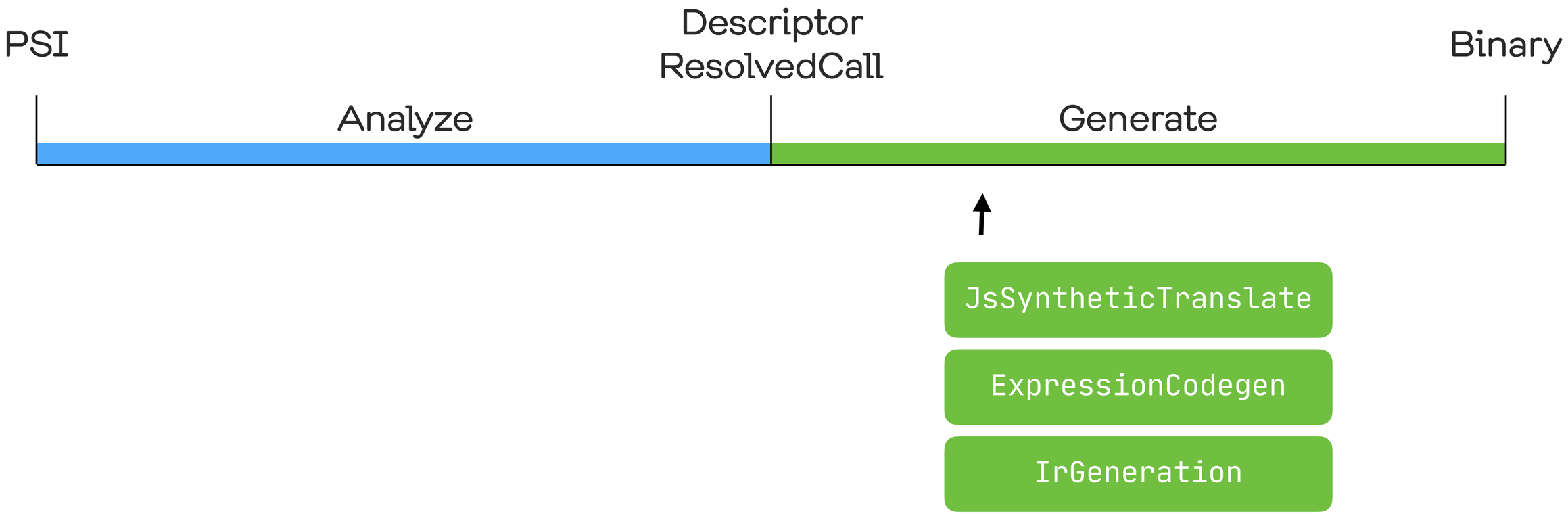
`SyntheticResolve`

`DeclarationChecker`

`IrGeneration`

`JsSyntheticTranslate`

Generation


PSI
Descriptor
ResolvedCall
Binary
Analyze
Generate
JsSyntheticTranslate
ExpressionCodegen
IrGeneration

```
interface IrGenerationExtension {
    fun generate(
        moduleFragment: IrModuleFragment,
        pluginContext: IrPluginContext
    )
}
```

```
class SerializationLoweringExtension : IrGenerationExtension {
    fun generate(
        moduleFragment: IrModuleFragment,
        pluginContext: IrPluginContext
    ) {
         SerializerClassLowering(pluginContext).lower(moduleFragment)
    }
}
```

```
class SerializationLoweringExtension : IrGenerationExtension {
    fun generate(
        moduleFragment: IrModuleFragment,
        pluginContext: IrPluginContext
    ) {
         SerializerClassLowering(pluginContext).lower(moduleFragment)
    }
}

private class SerializerClassLowering(
    val context: IrPluginContext
) : ClassLoweringPass {
    override fun lower(irClass: IrClass) {
        generate(irClass, context, context.bindingContext)
    }
}
```

```
kotlinOptions {
    freeCompilerArgs = [
        "-Xuse-ir",
        "-Xdump-directory=${buildDir}/ir/",
        "-Xphases-to-dump-after=ValidateIrBeforeLowering"
    ]
}
```

```
FILE fqName:<root>
  CLASS CLASS name:Data modality:FINAL visibility:public [data] superTypes:[kotlin.Any]
    $this: VALUE_PARAMETER INSTANCE_RECEIVER name:<this> type:<root>.Data
    CONSTRUCTOR visibility:public <> (a:kotlin.Int, b:kotlin.String) returnType:<root>.Data [primary]
      VALUE_PARAMETER name:a index:0 type:kotlin.Int
      VALUE_PARAMETER name:b index:1 type:kotlin.String
        EXPRESSION_BODY
          CONST String type=kotlin.String value="42"
      BLOCK_BODY
        DELEGATING_CONSTRUCTOR_CALL 'public constructor <init> () [primary] declared in kotlin.Any'
        INSTANCE_INITIALIZER_CALL classDescriptor='CLASS CLASS name:Data modality:FINAL visibility:public [data] superTypes:[kotlin.Any]
    PROPERTY name:a visibility:public modality:FINAL [val]
      FIELD PROPERTY_BACKING_FIELD name:a type:kotlin.Int visibility:private [final]
        EXPRESSION_BODY
          GET_VAR 'a: kotlin.Int declared in <root>.Data.<init>' type=kotlin.Int origin=INITIALIZE_PROPERTY_FROM_PARAMETER
      FUN DEFAULT_PROPERTY_ACCESSOR name:<get-a> visibility:public modality:FINAL <> ($this:<root>.Data) returnType:kotlin.Int
        correspondingProperty: PROPERTY name:a visibility:public modality:FINAL [val]
        $this: VALUE_PARAMETER name:<this> type:<root>.Data
        BLOCK_BODY
          RETURN type=kotlin.Nothing from='public final fun <get-a> (): kotlin.Int declared in <root>.Data'
            GET_FIELD 'FIELD PROPERTY_BACKING_FIELD name:a type:kotlin.Int visibility:private [final]' type=kotlin.Int origin=null
              receiver: GET_VAR '<this>: <root>.Data declared in <root>.Data.<get-a>' type=<root>.Data origin=null
    PROPERTY name:b visibility:public modality:FINAL [val]
      FIELD PROPERTY_BACKING_FIELD name:b type:kotlin.String visibility:private [final]
        EXPRESSION_BODY
          GET_VAR 'b: kotlin.String declared in <root>.Data.<init>' type=kotlin.String origin=INITIALIZE_PROPERTY_FROM_PARAMETER
      FUN DEFAULT_PROPERTY_ACCESSOR name:<get-b> visibility:public modality:FINAL <> ($this:<root>.Data) returnType:kotlin.String
        correspondingProperty: PROPERTY name:b visibility:public modality:FINAL [val]
        $this: VALUE_PARAMETER name:<this> type:<root>.Data
        BLOCK_BODY
          RETURN type=kotlin.Nothing from='public final fun <get-b> (): kotlin.String declared in <root>.Data'
            GET_FIELD 'FIELD PROPERTY_BACKING_FIELD name:b type:kotlin.String visibility:private [final]' type=kotlin.String origin=null
              receiver: GET_VAR '<this>: <root>.Data declared in <root>.Data.<get-b>' type=<root>.Data origin=null
```

```
fun generate(
    irClass: IrClass,
    context: IrPluginContext,
    bindingContext: BindingContext
) {
    if (!irClass.descriptor.hasSerializableAnnotation) return

    val serializerCls = generateSerializer(irClass, context, bindingContext)
    generateCompanionFactory(irClass, serializerCls, context, bindingContext)
}
```

```
private object `$serializer` : KSerializer<Data>
```

```
CLASS OBJECT name:$serializer modality:FINAL visibility:private superTypes:[kotlinx.serialization.KSerializer<<root>.Data>]
  $this: VALUE_PARAMETER INSTANCE_RECEIVER name:<this> type:<root>.Data.$serializer
  CONSTRUCTOR visibility:private <> () returnType:<root>.Data.$serializer [primary]
    BLOCK_BODY
      DELEGATING_CONSTRUCTOR_CALL 'public constructor <init> () [primary] declared in kotlin.Any'
      INSTANCE_INITIALIZER_CALL classDescriptor='CLASS OBJECT name:$serializer'
```

```
/**
CLASS OBJECT
    name:$serializer
    modality:FINAL
    visibility:private
    superTypes:[kotlinx.serialization.KSerializer<<root>.Data>]
*/

val kserializerSymbol = pluginContext.referenceClass(FqName("kotlinx.serialization.KSerializer"))
val kserializerType = kserializerSymbol.createType(
    hasQuestionMark = false,
    arguments = listOf(irClass.defaultType)
)

val serializerCls = buildClass {
    name = Name.identifier("\$serializer")
    kind = ClassKind.OBJECT
    modality = Modality.FINAL
    visibility = Visibilities.PRIVATE
}
```

```
/**
CLASS OBJECT
    name:$serializer
    modality:FINAL
    visibility:private
    superTypes:[kotlinx.serialization.KSerializer<<root>.Data>]
*/

val kserializerSymbol = pluginContext.referenceClass(FqName("kotlinx.serialization.KSerializer"))
val kserializerType = kserializerSymbol.createType(
    hasQuestionMark = false,
    arguments = listOf(irClass.defaultType)
)

val serializerCls = buildClass {
    name = Name.identifier("\$serializer")
    kind = ClassKind.OBJECT
    modality = Modality.FINAL
    visibility = Visibilities.PRIVATE
}
irClass.addChild(serializerCls)
serializerCls.superTypes.add(kserializerType)
```

```
/**
CONSTRUCTOR visibility:private <> () returnType:<root>.Data.$serializer [primary]
    BLOCK_BODY
*/

val primaryConstructor = serializerCls.addConstructor {
    returnType = serializerClsType
    isPrimary = true
    visibility = Visibilities.PRIVATE
}
primaryConstructor.body = pluginContext.body(primaryConstructor.symbol) {
    ...
}
```

```
/**
CONSTRUCTOR visibility:private <> () returnType:<root>.Data.$serializer [primary]
    BLOCK_BODY
*/

val primaryConstructor = serializerCls.addConstructor {
    returnType = serializerClsType
    isPrimary = true
    visibility = Visibilities.PRIVATE
}
primaryConstructor.body = pluginContext.body(primaryConstructor.symbol) {
    ...
}

private fun IrPluginContext.body(
    symbol: IrSymbol,
    block: IrBlockBodyBuilder.(symbolTable: SymbolTable) -> Unit
): IrBlockBody =
    DeclarationIrBuilder(pluginContext, symbol).run {
        irBlockBody {
            block(this@withScope)
        }
    }
```

```
/**
BLOCK_BODY
  DELEGATING_CONSTRUCTOR_CALL 'public constructor <init> () [primary] declared in kotlin.Any'
  INSTANCE_INITIALIZER_CALL classDescriptor='CLASS OBJECT name:$serializer'
*/

primaryConstructor.body = pluginContext.body(primaryConstructor.symbol) {
    + irDelegatingConstructorCall(
        pluginContext.irBuiltIns.unitType,
        anyConstructor
    )
    + irInstanceInitializerCall(
        serializerCls.symbol,
        serializerClsType
    )
}
```

https://github.com/ShikaSD/compiler-plugin-talk-example

Add serializer

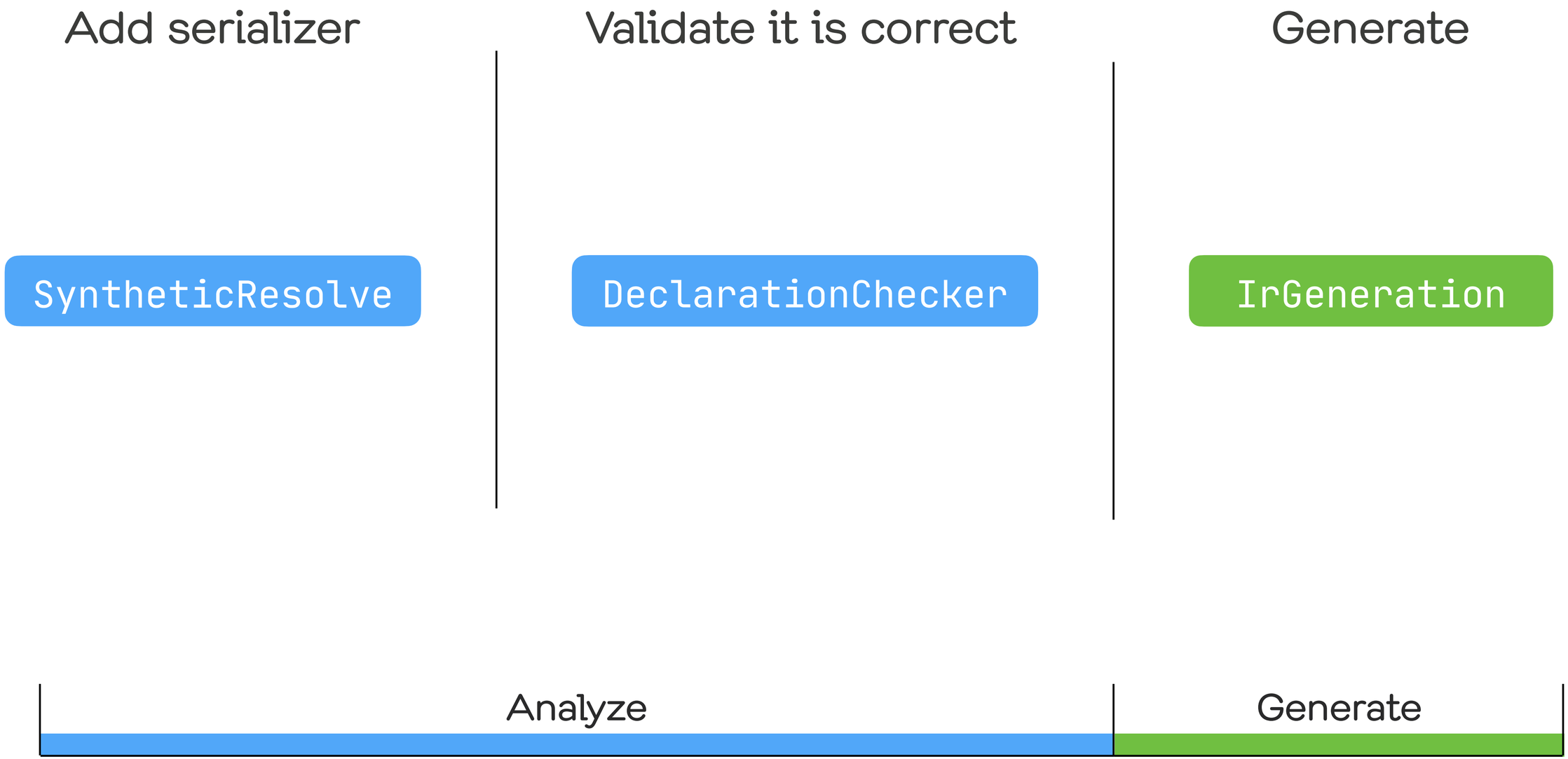
`SyntheticResolve`

Add serializer

Validate it is correct

`SyntheticResolve`

`DeclarationChecker`


Add serializer
Validate it is correct
Generate
SyntheticResolve
DeclarationChecker
IrGeneration
Analyze
Generate

Serialization

1. KotlinConf 2019 → Design of Kotlin Serialization
2. Source code → Kotlin / Kotlinx.serialization
3. Источник информации / примеров

kotlinx-serialization

compose

kapt in compiler

kotlinx-serialization

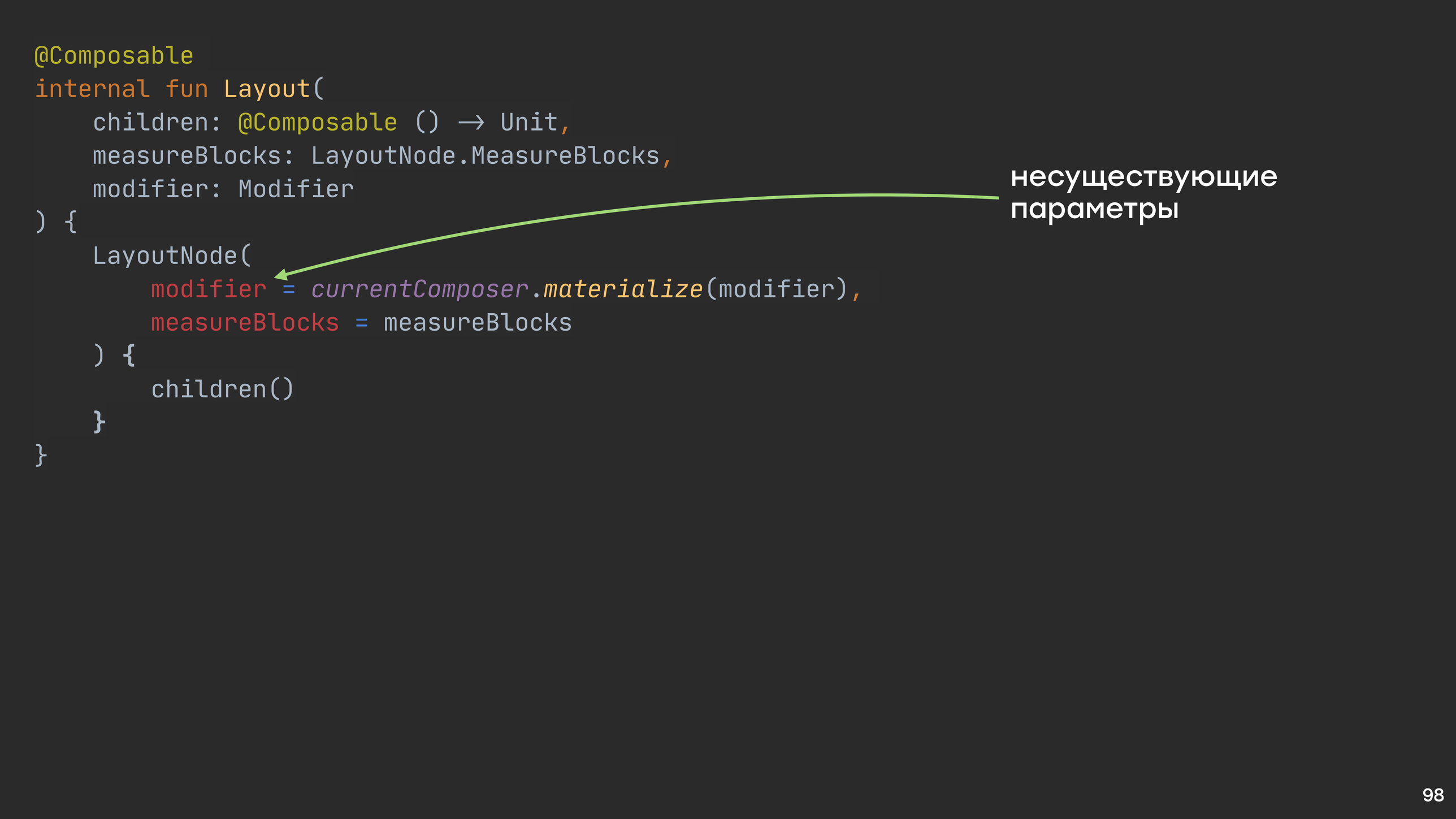
compose

kapt in compiler

```
class LayoutNode : Measurable {
    /**
     * The children of this LayoutNode, controlled by [insertAt], [move], and [removeAt].
     */
    val children: List<LayoutNode> get() = _children

    fun insertAt(index: Int, instance: LayoutNode) { ... }

    fun removeAt(index: Int, count: Int) { ... }

    fun move(from: Int, to: Int, count: Int) { ... }

}
```

```
@Composable
internal fun Layout(
    children: @Composable () → Unit,
    measureBlocks: LayoutNode.MeasureBlocks,
    modifier: Modifier
) {
    LayoutNode(
        modifier = currentComposer.materialize(modifier),
        measureBlocks = measureBlocks
    ) {
        children()
    }
}
```

```
@Composable
internal fun Layout(
    children: @Composable () → Unit,
    measureBlocks: LayoutNode.MeasureBlocks,
    modifier: Modifier
) {
    LayoutNode(
        modifier = currentComposer.materialize(modifier),
        measureBlocks = measureBlocks
    ) {
        children()
    }
}
```



```
@Composable
internal fun Layout(
    children: @Composable () → Unit,
    measureBlocks: LayoutNode.MeasureBlocks,
    modifier: Modifier
) {
    LayoutNode(
        modifier = currentComposer.materialize(modifier),
        measureBlocks = measureBlocks
    ) {
        children()
    }
}
```

несуществующие параметры

параметр с детьми

```kotlin
@Composable
internal fun Layout(
    children: @Composable () → Unit,
    measureBlocks: LayoutNode.MeasureBlocks,
    modifier: Modifier
) {
    LayoutNode(
        modifier = currentComposer.materialize(modifier),
        measureBlocks = measureBlocks
    ) {
        children()
    }
}
```

несуществующие параметры

параметр с детьми

0 параметров / инит блоков

```kotlin
class LayoutNode : Measurable {
    ...
}
```

```kotlin
@Composable
internal fun Layout(
    children: @Composable () → Unit,
    measureBlocks: LayoutNode.MeasureBlocks,
    modifier: Modifier
) {
    LayoutNode(
        modifier = currentComposer.materialize(modifier),
        measureBlocks = measureBlocks
    ) {
        children()
    }
}
```

несуществующие параметры

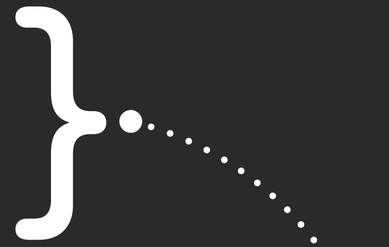
параметр с детьми

0 параметров / инит блоков

```kotlin
class LayoutNode : Measurable {
    ...
}
```

```
class LayoutNode : Measurable {

    fun insertAt(index: Int, instance: LayoutNode) { ... }

    fun removeAt(index: Int, count: Int) { ... }

    fun move(from: Int, to: Int, count: Int) { ... }

}
```

```
class LayoutNode : Measurable {

    fun insertAt(index: Int, instance: LayoutNode) { ... }
    fun removeAt(index: Int, count: Int) { ... }
    fun move(from: Int, to: Int, count: Int) { ... }

}
```



```
interface ApplyAdapter<N> {
    fun N.start(instance: N)

    fun N.insertAt(index: Int, instance: N)
    fun N.removeAt(index: Int, count: Int)
    fun N.move(from: Int, to: Int, count: Int)

    fun N.end(instance: N, parent: N)
}
```

```
class UiComposer(...): Composer<Any>(
    Applier(
        root,
        UiApplyAdapter()
    ),
    ...
) {
    inline fun <T : LayoutNode> emit(
        key: Any,
        ctor: () -> T,
        update: UiUpdater<T>.() -> Unit,
        children: () -> Unit
    ) { ... }
}
```

```
interface ApplyAdapter<N> {
    fun N.insertAt(index: Int, instance: N)
    fun N.removeAt(index: Int, count: Int)
    fun N.move(from: Int, to: Int, count: Int)
}
```

```
class UiComposer(...): Composer<Any>(
    Applier(
        root,
        UiApplyAdapter()
    ),
    ...
) {
    inline fun <T : LayoutNode> emit(
        key: Any,
        ctor: () -> T,
        update: UiUpdater<T>.() -> Unit,
        children: () -> Unit
    ) { ... }
}
```

x4

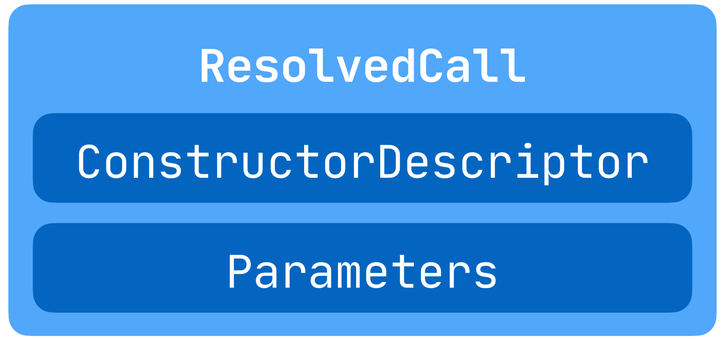
```
inline fun <T : LayoutNode> emit(
    key: Any,
    ctor: () -> T,
    update: UiUpdater<T>.() -> Unit,
    children: () -> Unit
) { ... }
```

```
inline fun <T : LayoutNode> emit(
    key: Any,
    ctor: () -> T,
    update: UiUpdater<T>.() -> Unit,
    children: () -> Unit
) { ... }
```

```
LayoutNode(
    modifier = currentComposer.materialize(modifier),
    measureBlocks = measureBlocks
) {
    children()
}
```

Resolving calls

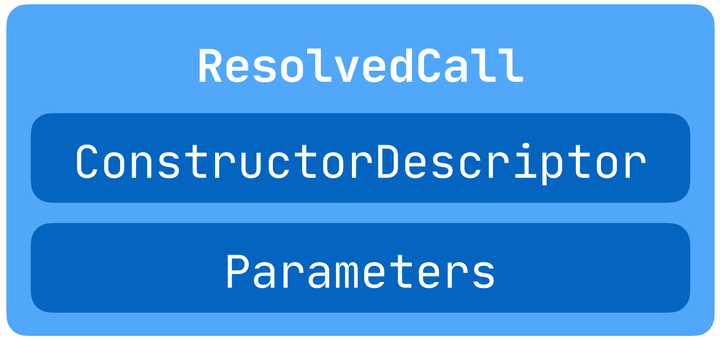
```
LayoutNode(
    modifier = ...
    measureBlocks = measureBlocks
) {
    children()
}
```



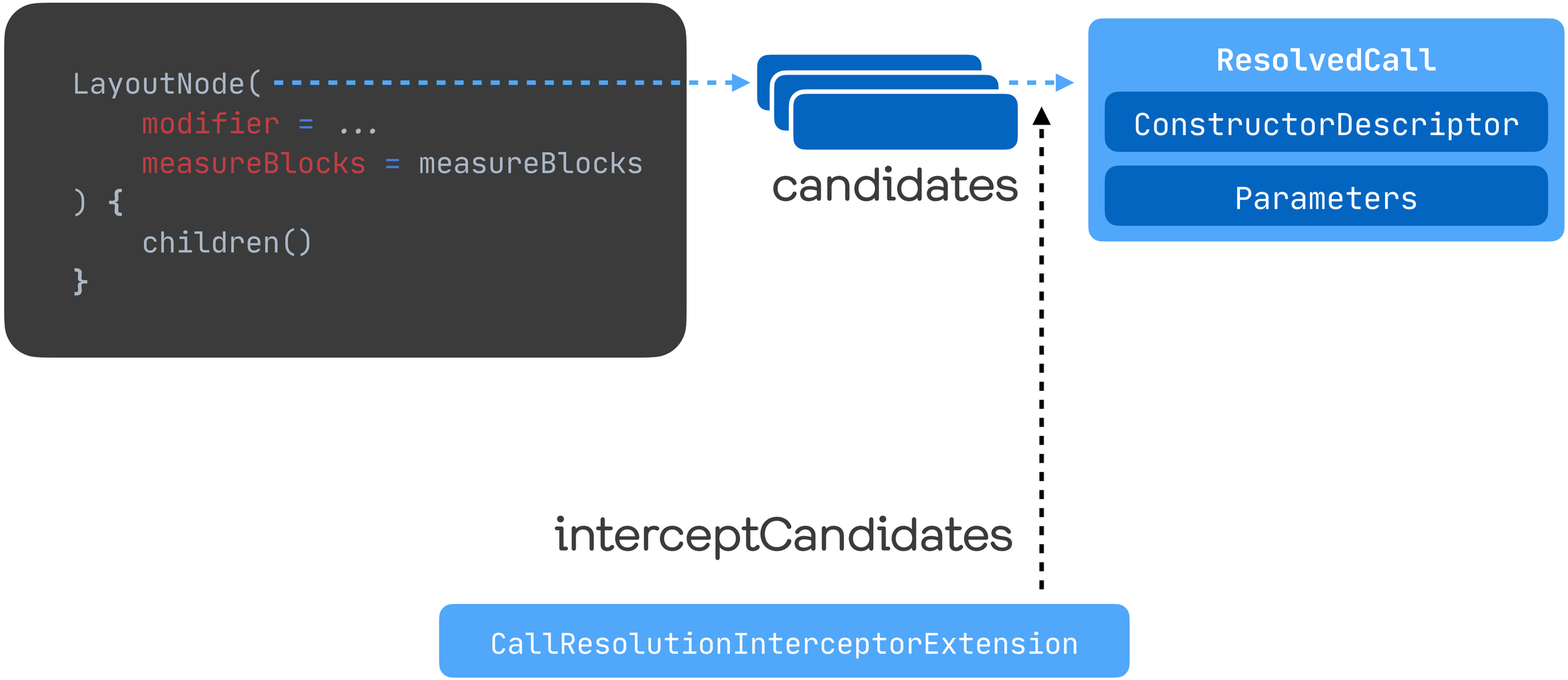
Resolving calls

```
LayoutNode(
    modifier = ...
    measureBlocks = measureBlocks
) {
    children()
}
```

candidates



Resolving calls



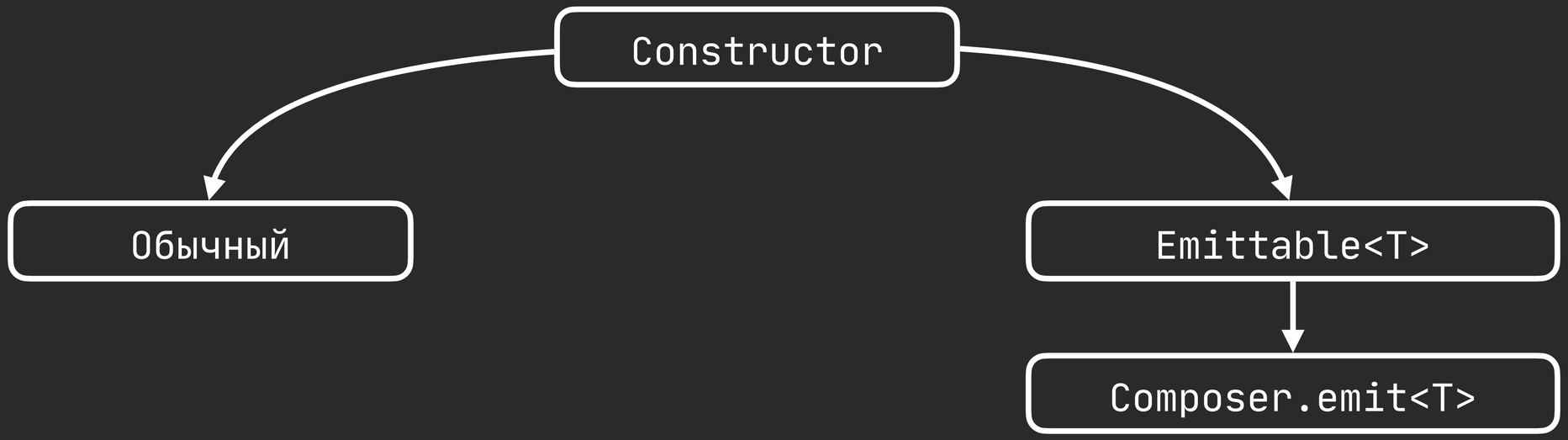
WARNING!!!

```
@Experimental(level = Experimental.Level.ERROR)
@Retention(AnnotationRetention.BINARY)
internal annotation class InternalNonStableExtensionPoints
```

```
interface CallResolutionInterceptorExtension {
    fun interceptCandidates(
        candidates: Collection<FunctionDescriptor>,
        ...,
        name: Name,
        ...
    ): Collection<FunctionDescriptor> = candidates
}
```

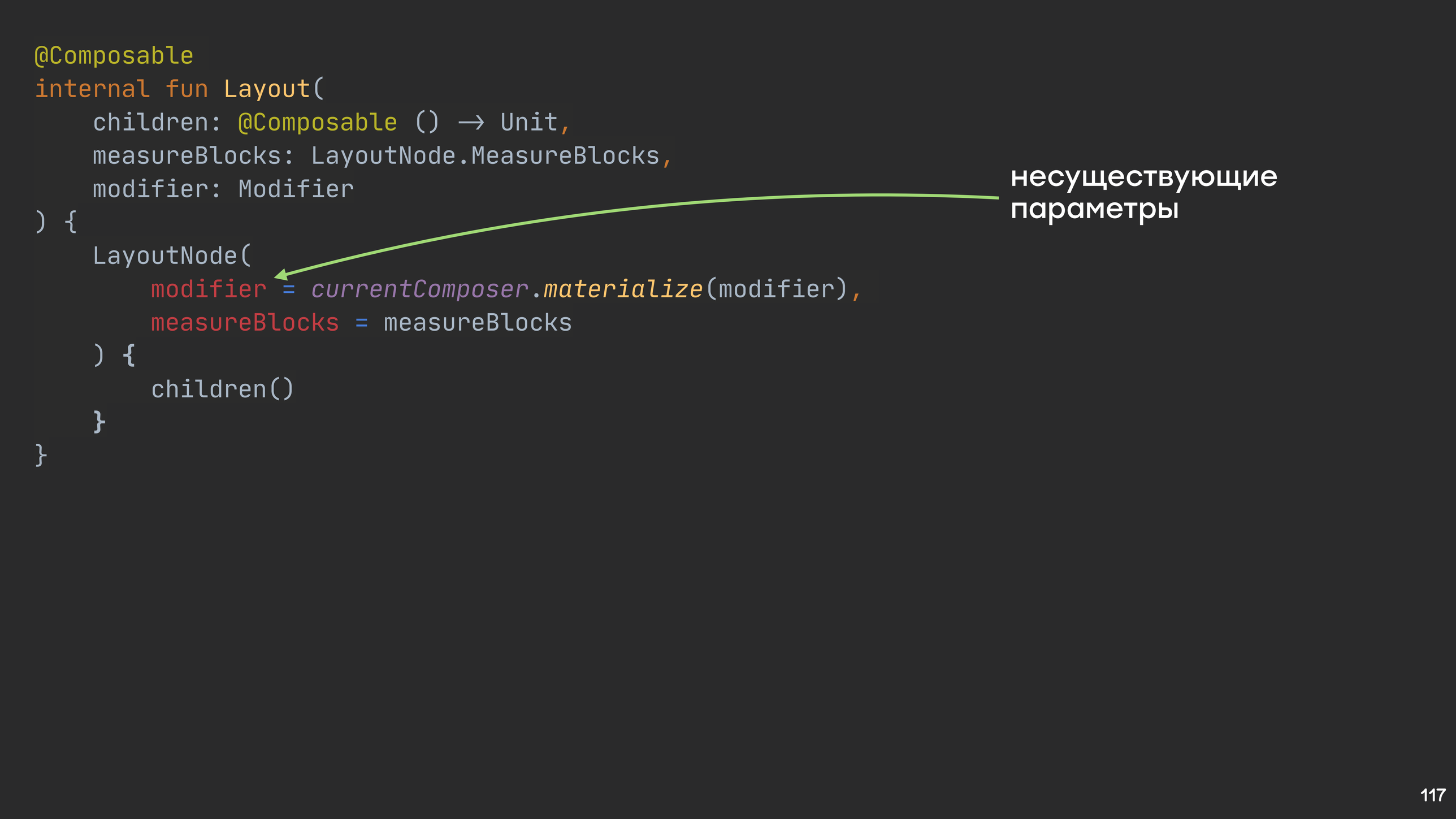
```
class ComposeCallResolutionInterceptorExtension : CallResolutionInterceptorExtension {
    override fun interceptCandidates(
        candidates: Collection<FunctionDescriptor>,
        ...
    ): Collection<FunctionDescriptor> {

    ...

    for (candidate in candidates) {
        when {
            candidate.hasComposableAnnotation() →
                composables.add(candidate)
            candidate is ConstructorDescriptor →
                constructors.add(candidate)
            else →
                nonComposablesNonConstructors.add(candidate)
        }
    }

    ...
}
```


Constructor
Обычный
Emittable<T>


Constructor
Обычный
Emittable<T>
Composer.emit<T>

```
class ComposeCallResolutionInterceptorExtension : CallResolutionInterceptorExtension {
    override fun interceptCandidates(
        candidates: Collection<FunctionDescriptor>,
        ...
    ): Collection<FunctionDescriptor> {

    ...

    val emittables = constructors.filter { composerMetadata.isEmittable(it.returnType) }
    val emitCandidates = emitResolver.resolveCandidates( ... )

    return nonComposablesNonConstructors +
        composables.map { ComposableFunctionDescriptor(it) } +
        constructors.filter { !composerMetadata.isEmittable(it.returnType) } +
        emitCandidates

}
```

```
@Composable
internal fun Layout(
    children: @Composable () → Unit,
    measureBlocks: LayoutNode.MeasureBlocks,
    modifier: Modifier
) {
    LayoutNode(
        modifier = currentComposer.materialize(modifier),
        measureBlocks = measureBlocks
    ) {
        children()
    }
}
```



```
private fun IrBlockBuilder.irComposableEmitBase(
    original: IrCall,
    getComposer: () -> IrExpression,
    emitMetadata: ComposableEmitMetadata
): IrExpression {
    /*

    TextView(text="foo")

    // transforms into

    val attr_text = "foo"
    composer.emit(
        key = 123,
        ctor = { context -> TextView(context) },
        update = { set(attr_text) { text -> this.text = text } }
    )
    */

    ...
}
```

```
private fun IrBlockBuilder.irComposableEmitBase(
    original: IrCall,
    getComposer: () -> IrExpression,
    emitMetadata: ComposableEmitMetadata
): IrExpression {
    /*

    TextView(text="foo")

    // transforms into

    val attr_text = "foo"
    composer.emit(
        key = 123,
        ctor = { context -> TextView(context) },
        update = { set(attr_text) { text -> this.text = text } }
    )
    */

    ...
}
```

```
private fun IrBlockBuilder.irComposableEmitBase(
    original: IrCall,
    getComposer: () -> IrExpression,
    emitMetadata: ComposableEmitMetadata
): IrExpression {
    /*

    TextView(text="foo")

    // transforms into

    val attr_text = "foo"
    composer.emit(
        key = 123,
        ctor = { context -> TextView(context) },
        update = { set(attr_text) { text -> this.text = text } }
    )
    */

    ...
}
```

```
private fun IrBlockBuilder.irComposableEmitBase(
    original: IrCall,
    getComposer: () -> IrExpression,
    emitMetadata: ComposableEmitMetadata
): IrExpression {
    /*

    TextView(text="foo")

    // transforms into

    val attr_text = "foo"
    composer.emit(
        key = 123,
        ctor = { context -> TextView(context) },
        update = { set(attr_text) { text -> this.text = text } }
    )
    */

    ...
}
```

```
LayoutNode(
    modifier = currentComposer.materialize(modifier),
    measureBlocks = measureBlocks
) {
    children()
}
```

```
inline fun <T : LayoutNode> emit(
    key: Any,
    ctor: () -> T,
    update: UiUpdater<T>.() -> Unit,
    children: () -> Unit
) { ... }
```

```
LayoutNode(
    modifier = currentComposer.materialize(modifier),
    measureBlocks = measureBlocks
) {
    children()
}
```

```
inline fun <T : LayoutNode> emit(
    key: Any,
    ctor: () -> T,
    update: UiUpdater<T>.() -> Unit,
    children: () -> Unit
) { ... }
```

```
val modifier = currentComposer.materialize(modifier)
val measureBlocks = measureBlocks

composer.emit(
    key = 123,
    ctor = { LayoutNode() }
)
```

```
LayoutNode(
    modifier = currentComposer.materialize(modifier),
    measureBlocks = measureBlocks
) {
    children()
}
```

```
inline fun <T : LayoutNode> emit(
    key: Any,
    ctor: () → T,
    update: UiUpdater<T>.() → Unit,
    children: () → Unit
) { ... }
```

```
val modifier = currentComposer.materialize(modifier)
val measureBlocks = measureBlocks

composer.emit(
    key = 123,
    ctor = { LayoutNode() },
    update = {
        set(modifier) { modifier → this.modifier = modifier }
        set(measureBlocks) { blocks → this.measureBlocks = blocks }
    }
)
```

Compose

1. KotlinConf 2019 → The Compose Runtime, Demystified
2. Source code → AOSP, androidx-master-dev
3. #compose on kotlinlang slack

kotlinx-serialization

compose

kapt in compiler

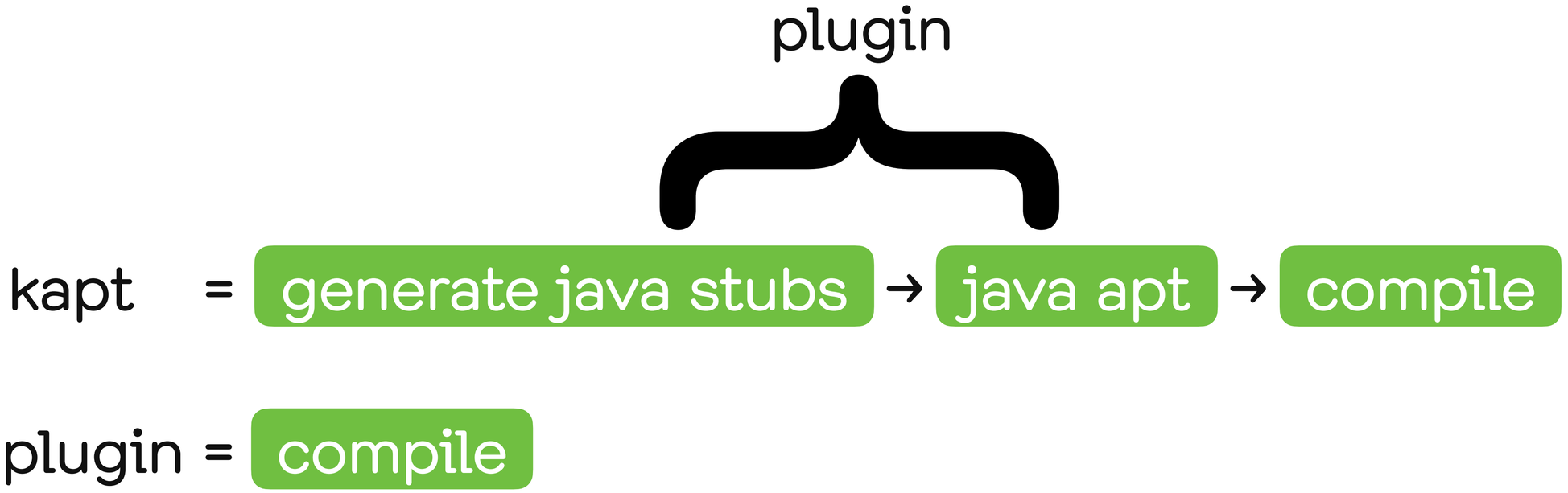
kotlinx-serialization

compose

kapt in compiler

kapt = generate java stubs → java apt → compile

plugin = compile

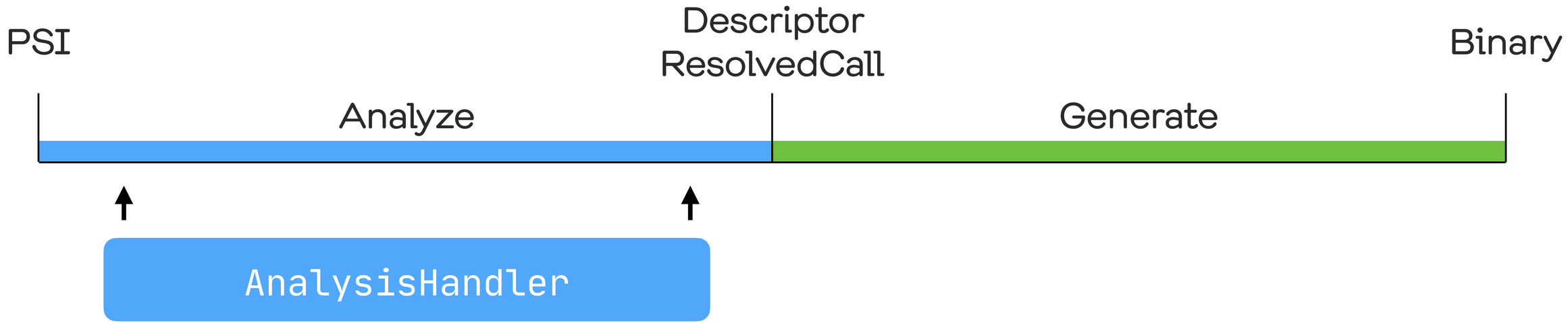

plugin
kapt = generate java stubs → java apt → compile
plugin = compile

-KaptMode=stubsAndApt

kapt = generate java stubs → java apt → compile

plugin = compile

AnalysisHandler



```
interface AnalysisHandlerExtension {
    fun doAnalysis(
        project: Project,
        module: ModuleDescriptor,
        ...,
        files: Collection<KtFile>
    ): AnalysisResult? = null

    fun analysisCompleted(
        project: Project,
        module: ModuleDescriptor,
        ...,
        files: Collection<KtFile>
    ): AnalysisResult? = null
}
```

```
class AnalysisResult {
    fun success(
        shouldGenerateCode: Boolean
    ): AnalysisResult = ...

    fun compilationError(): AnalysisResult = ...

    class RetryWithAdditionalRoots(
        additionalJavaRoots: List<File>,
        additionalKotlinRoots: List<File>
    )
}
```

```
@Mobius
class Talk {
    init {
        MyTalk()
    }
}
```

```kotlin
class MobiusExt(val sourcesDir: File) : AnalysisHandlerExtension {
    private var generated = false

    override fun analysisCompleted(
        project: Project,
        bindingTrace: BindingTrace,
        files: Collection<KtFile>
    ): AnalysisResult? {
        if (generated) { return null }
        generated = true

        files.forEach {
            it.accept(classRecursiveVisitor {
                val descriptor = bindingTrace[CLASS, it] ?: return
                generateClass(descriptor)
            })
        }
        return AnalysisResult.RetryWithAdditionalRoots(
            ...,
            additionalKotlinRoots = listOf(sourcesDir)
        )
    }
}
```

```
class MobiusExt(val sourcesDir: File) : AnalysisHandlerExtension {
    private var generated = false

    override fun analysisCompleted(
        project: Project,
        bindingTrace: BindingTrace,
        files: Collection<KtFile>
    ): AnalysisResult? {
        if (generated) { return null }
        generated = true

        files.forEach {
            it.accept(classRecursiveVisitor {
                val descriptor = bindingTrace[CLASS, it] ?: return
                generateClass(descriptor)
            })
        }
        return AnalysisResult.RetryWithAdditionalRoots(
            ...,
            additionalKotlinRoots = listOf(sourcesDir)
        )
    }
}
```

```
class MobiusExt(val sourcesDir: File) : AnalysisHandlerExtension {
    private var generated = false

    override fun analysisCompleted(
        project: Project,
        bindingTrace: BindingTrace,
        files: Collection<KtFile>
    ): AnalysisResult? {
        if (generated) { return null }
        generated = true

        files.forEach {
            it.accept(classRecursiveVisitor {
                val descriptor = bindingTrace[CLASS, it] ?: return
                generateClass(descriptor)
            })
        }
        return AnalysisResult.RetryWithAdditionalRoots(
            ...,
            additionalKotlinRoots = listOf(sourcesDir)
        )
    }
}
```

```
class MobiusExt(val sourcesDir: File) : AnalysisHandlerExtension {
    private var generated = false

    override fun analysisCompleted(
        project: Project,
        bindingTrace: BindingTrace,
        files: Collection<KtFile>
    ): AnalysisResult? {
        if (generated) { return null }
        generated = true

        files.forEach {
            it.accept(classRecursiveVisitor {
                val descriptor = bindingTrace[CLASS, it] ?: return
                generateClass(descriptor)
            })
        }
        return AnalysisResult.RetryWithAdditionalRoots(
            ...,
            additionalKotlinRoots = listOf(sourcesDir)
        )
    }
}
```

```
fun generateClass(descriptor: ClassDescriptor) {
    if (!descriptor.annotations.hasAnnotation(MOBIUS_ANNOTATION_FQNAME)) {
        return
    }

    writeClass(descriptor.fqNameSafe, "My" + descriptor.name)
}
```

```
class MobiusExt(val sourcesDir: File) : AnalysisHandlerExtension {
    private var generated = false

    override fun analysisCompleted(
        project: Project,
        bindingTrace: BindingTrace,
        files: Collection<KtFile>
    ): AnalysisResult? {
        if (generated) { return null }
        files.forEach {
            it.accept(classRecursiveVisitor {
                val descriptor = bindingTrace[CLASS, it] ?: return
                generateClass(descriptor)
            })
        }
        return AnalysisResult.RetryWithAdditionalRoots(
            ...,
            additionalKotlinRoots = listOf(sourcesDir)
        )
    }
}
```

```
@Mobius
class Talk {
    init {
        MyTalk() // found! + sources are in sourcesDir
    }
}
```

KSP

https://github.com/android/kotlin/tree/ksp/libraries/tools/kotlin-symbol-processing-api

https://github.com/ShikaSD/kotlin-dagger-reflect-compiler

Экспериментальное API

Много возможностей

Экспериментальное API

Много возможностей

Поддержка мультиплатформы

Экспериментальное API

Много возможностей

Поддержка мультиплатформы

badoo

Thank you!

 @shikasd

 @shikasd_

habr.com/company/badoo

tech.badoo.com