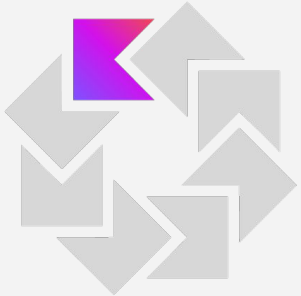


Kotlin
Vilnius
User Group

Kotlin Annotation Processing

Lead software engineer at EPAM Systems

10+ years of mobile development

Passionate to the best design in every aspect

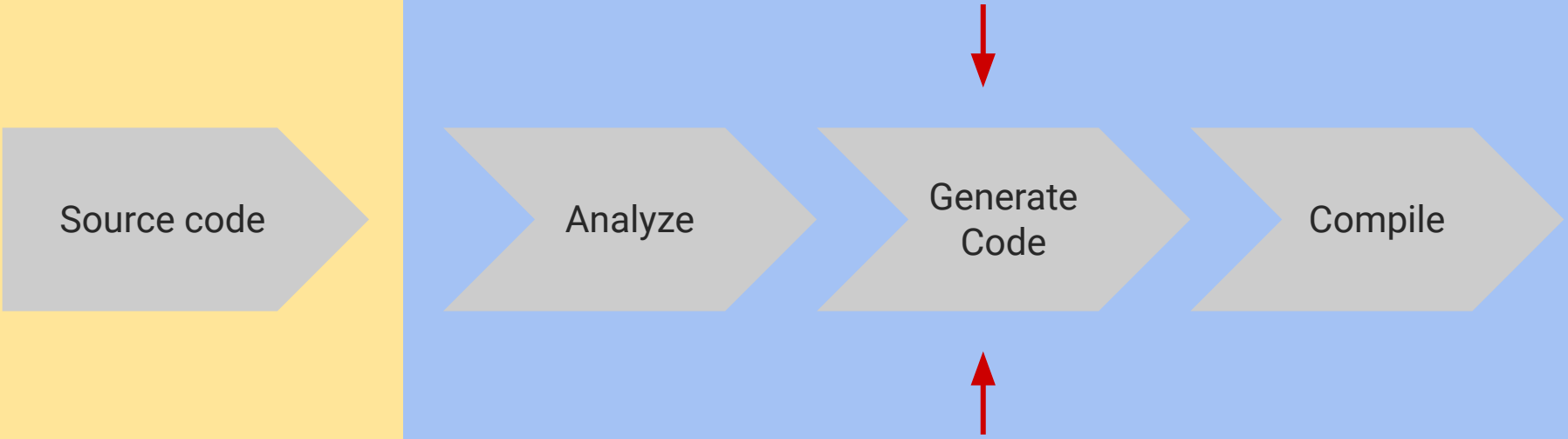
Agenda

- Annotation processing
- Java and Kotlin annotation processing
- Kotlin Symbolic Processing
- Best practices
- Conclusions
- Questions

Annotation processing

Code Editing

Code Compilation






Metaprogramming

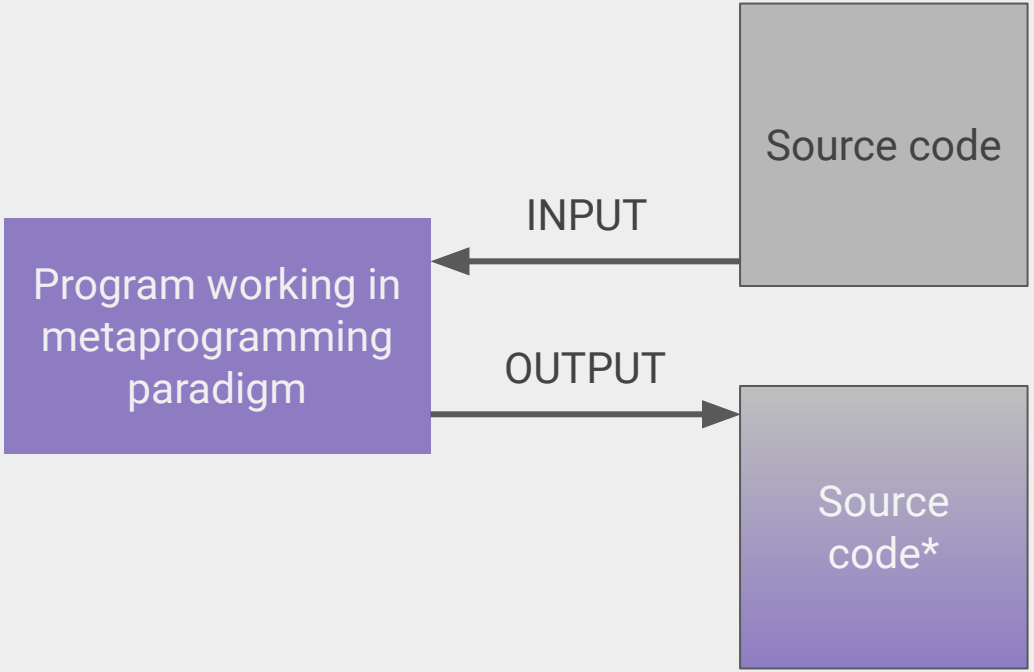
V · T · E

Types of programming languages [hide]

Paradigm

Actor-based · Array · Aspect-oriented · Class-based · Concatenative · Concurrent · Dataflow · Declarative · Differentiable · Domain-specific · Dynamic · Esoteric · Event-driven · Extensible · Functional · Imperative · Logic · Macro · **Metaprogramming** · Object-based · Object-oriented · Pipeline · Procedural · Prototype-based · Reflective · Rule-based · Scripting · Stack-oriented · Synchronous · Tactile · Template

Metaprogramming



“We’ve got some code for your code, so you can produce more code while writing your code…”

Use cases

Code generation	Validation	Configuration	Performance optimization

Java Annotation Processing

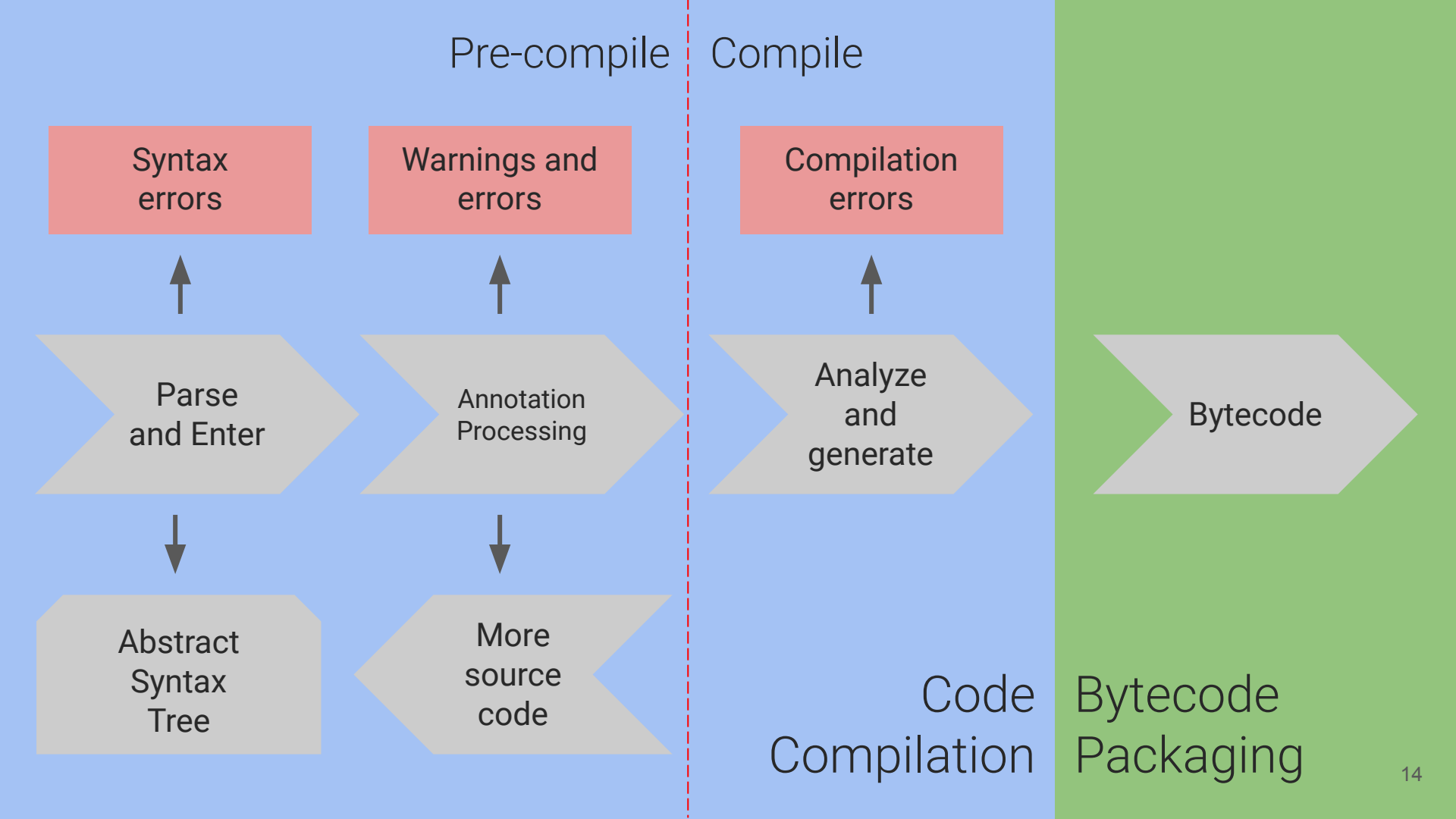
Code Editing

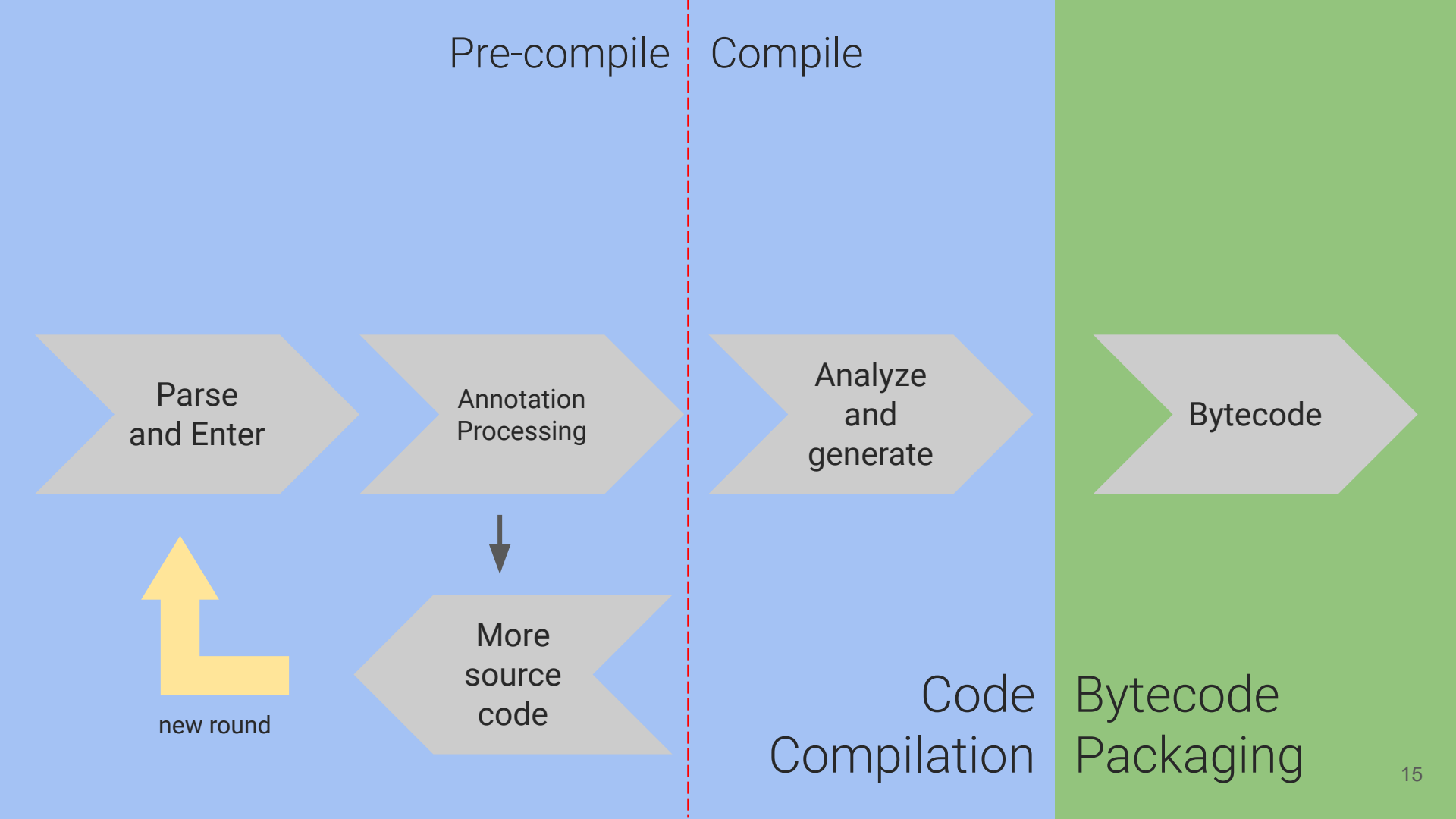
Code Compilation

Code Editing

Code Compilation

javac

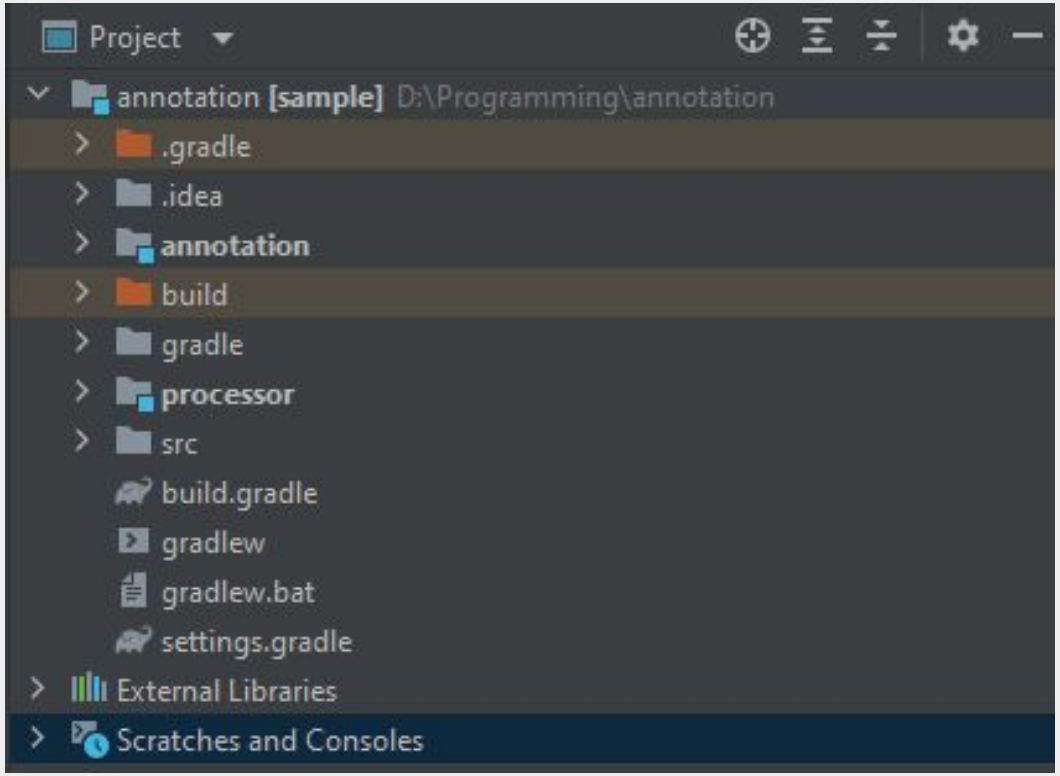

Pre-compile
Compile
Syntax errors
Warnings and errors
Compilation errors
Parse and Enter
Annotation Processing
Analyze and generate
Bytecode
Abstract Syntax Tree
More source code
Code Compilation
Bytecode Packaging


Pre-compile
Compile
Parse and Enter
Annotation Processing
Analyze and generate
Bytecode
More source code
new round
Code Compilation
Bytecode Packaging

Building blocks

1. define your custom annotation(s)
2. extend `javax.annotation.processing.AbstractProcessor`
3. override `getSupportedAnnotationTypes()` and `getSupportedSourceVersion()` methods
4. override method `process()` to provide your custom processing logic
5. register a `Processor` in `META-INF/services` directory
6. build a JAR and import it into the project
7. configure the build tool to use your JAR dependency as annotation processor

Building blocks



1. define your custom annotation(s);

```
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.SOURCE)
public @interface VilniusKugPrinter {}
```

2. extend `javax.annotation.processing.AbstractProcessor`;

```
public class VilniusKugPrinterProcessor extends AbstractProcessor {

    private Elements elementUtils;
    private Types typeUtils;
    private Filer filer;
    private Messager messager;

    @Override
    public synchronized void init(ProcessingEnvironment processingEnv) {
        super.init(processingEnv);
        elementUtils = processingEnv.getElementUtils();
        typeUtils = processingEnv.getTypeUtils();
        filer = processingEnv.getFiler();
        messager = processingEnv.getMessager();
    }
}
```

3. override `getSupportedAnnotationTypes()` and `getSupportedSourceVersion()` methods;

```
@Override
public Set<String> getSupportedAnnotationTypes() {
    return Set.of(VilniusKugPrinter.class.getCanonicalName());
}

@Override
public SourceVersion getSupportedSourceVersion() {
    return SourceVersion.latestSupported();
}
```

4. override method `process()` to provide your custom processing logic;

```
@Override
public boolean process(
    Set<? extends TypeElement> annotations,
    RoundEnvironment roundEnv
) {
    List<Element> elements = roundEnv
        .getElementsAnnotatedWith(VilniusKugPrinter.class);
    for (Element annotated : elements) {
       System.out.print(annotated.getSimpleName());
    }

    return true;
}
```

5. register a `Processor` in `META-INF/services` directory;

//Processor module's build.gradle:

```
dependencies {
    implementation project(":annotation")
    compileOnly("com.google.auto.service:auto-service:1.0.1")
    annotationProcessor("com.google.auto.service:auto-service:1.0.1")
}
```

```
@AutoService(Processor.class)
public class VilniusKugPrinterProcessor extends AbstractProcessor {
```

6. build a JAR and import it into the project
7. configure the build tool to use your JAR dependency as annotation processor

//main module's build.gradle:

```
dependencies {
   compileOnly project(":annotation")
   annotationProcessor(":processor")
}
```

```java
public class Main {
    public static void main(String[] args) {
        System.out.println(args);
    }
    private void doNothing(
            @VilniusKugPrinter int h,
            @VilniusKugPrinter int e,
            @VilniusKugPrinter int l,
            @VilniusKugPrinter int lo,
            @VilniusKugPrinter int world) {
        return;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
```

Run: annotation [:Main.main()]

- annotation [:Main.main()]: succe 763 ms
  - :annotation:compileJava UP-TC 11 ms
  - :annotation:processResources
  - :annotation:classes UP-TO-DATE
  - :annotation:jar UP-TO-DATE 10 ms
  - :processor:compileJava 235 ms
  - :processor:processResources
  - :processor:classes
  - :processor:jar 34 ms
  - :compileJava 70 ms
  - :processResources
  - :classes 1 ms
  - :Main.main() 78 ms

```
> Task :compileJava
AP for compileJava is [D:\Programming\
helloworld
```

Kotlin Annotation Processing

Code Editing

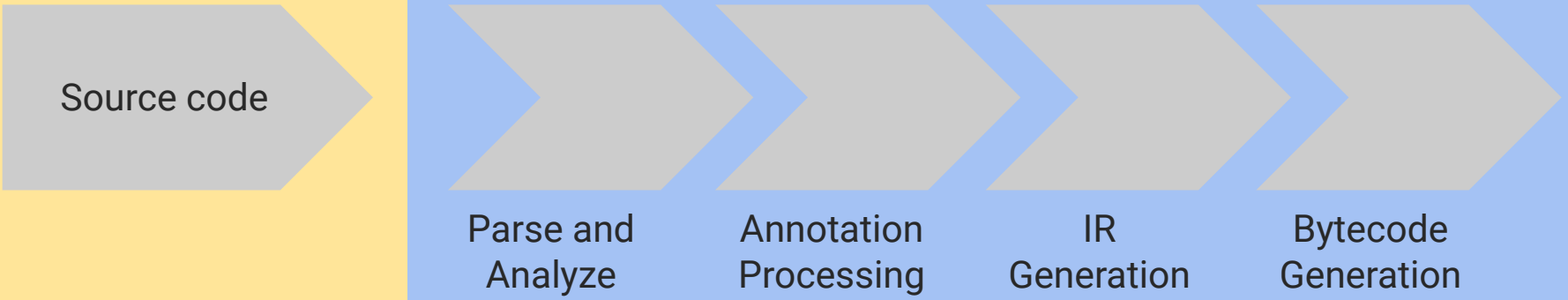
Code Compilation

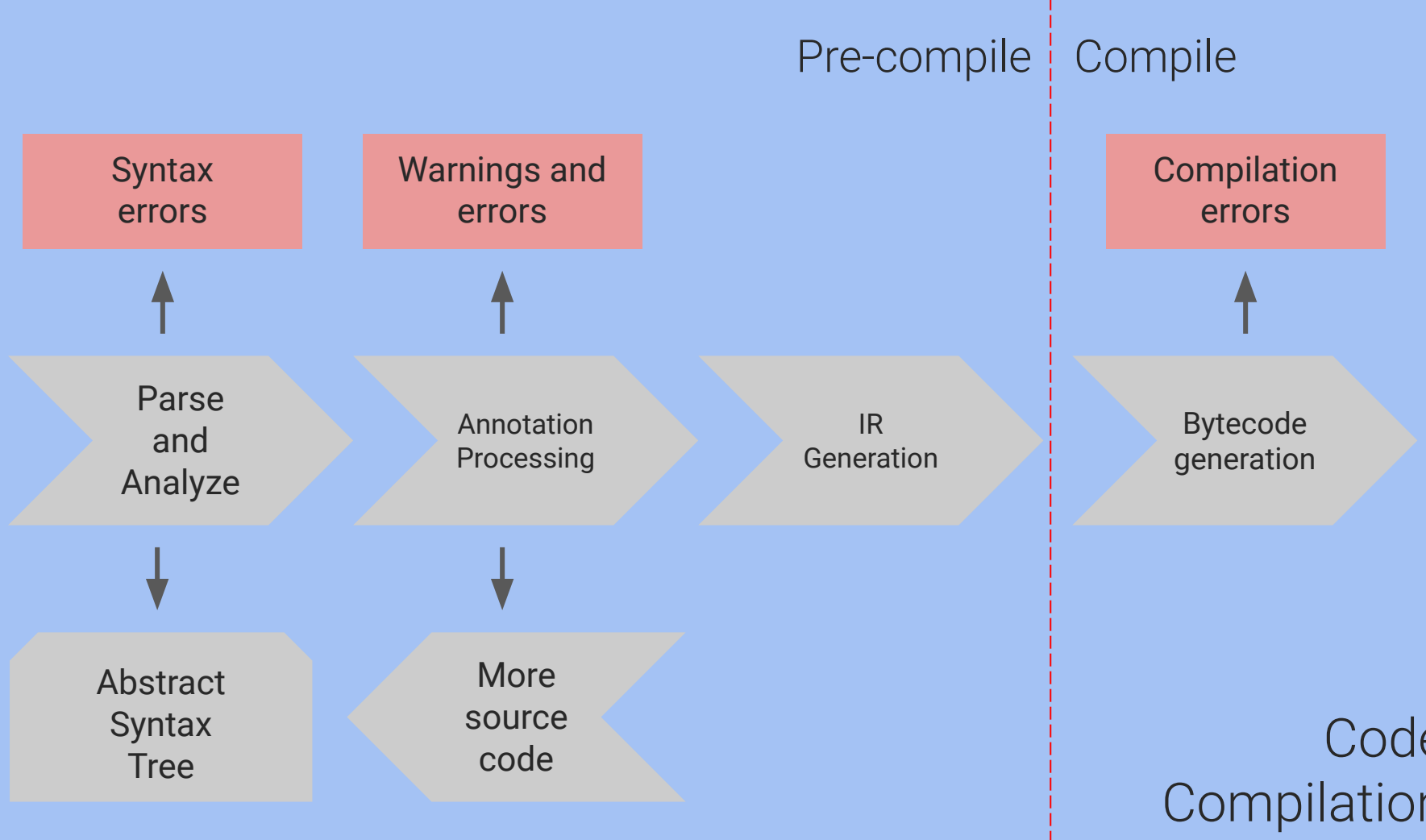
javac

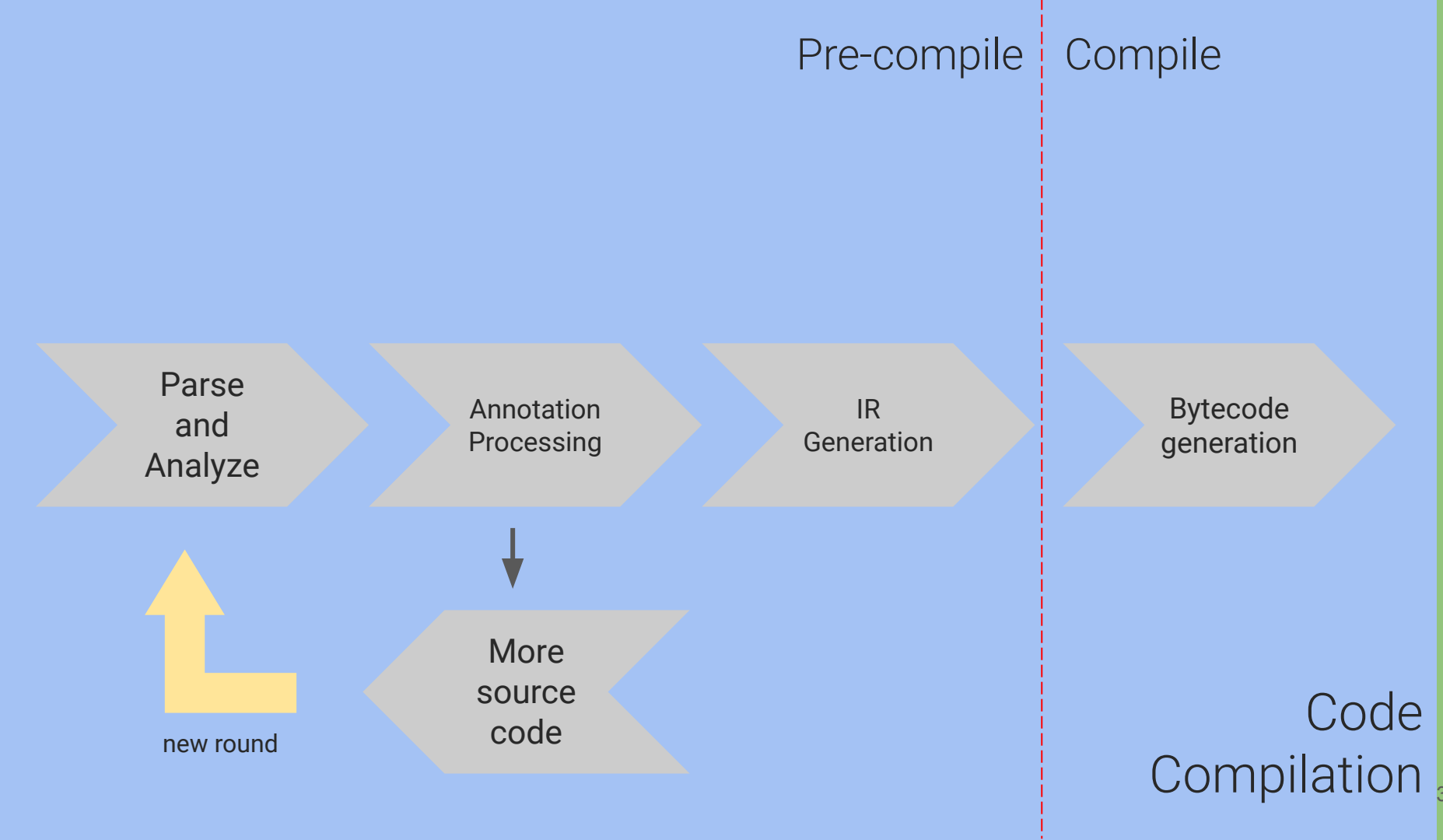
Code Editing

Code Compilation

kotlinc




Pre-compile
Compile
Syntax errors
Warnings and errors
Compilation errors
Parse and Analyze
Annotation Processing
IR Generation
Bytecode generation
Abstract Syntax Tree
More source code
Code Compilation


Pre-compile
Compile
Parse and Analyze
Annotation Processing
IR Generation
Bytecode generation
More source code
new round
Code Compilation

Building blocks

1. define your custom annotation(s); use Kotlin's target and retention annotations
2. extend `javax.annotation.processing.AbstractProcessor`;
3. override `getSupportedAnnotationTypes()` and `getSupportedSourceVersion()` methods;
4. override method `process()` to provide your custom processing logic;
5. register a `Processor` in `META-INF/services` directory
6. build a JAR and import it into the project
7. configure the build tool to use your JAR dependency as **KAPT** annotation processor

AP for Java & Kotlin

Pros:

- part of the compiler (javac, kotlinc)
- flexibility and access to AST

Cons:

- it makes builds slower
- build tools can’t easily do optimizations (like caching for example) for AP
- can’t handle kotlin specific cases

Kotlin Symbolic Processing

faster up to

x2

based on

compiler plugin API

Rational pros & cons

Pros:

- faster (but don’t expect x2 boost; depends on the use case)
- “understands” Kotlin-only semantics
- simplified API
- not tied to JVM

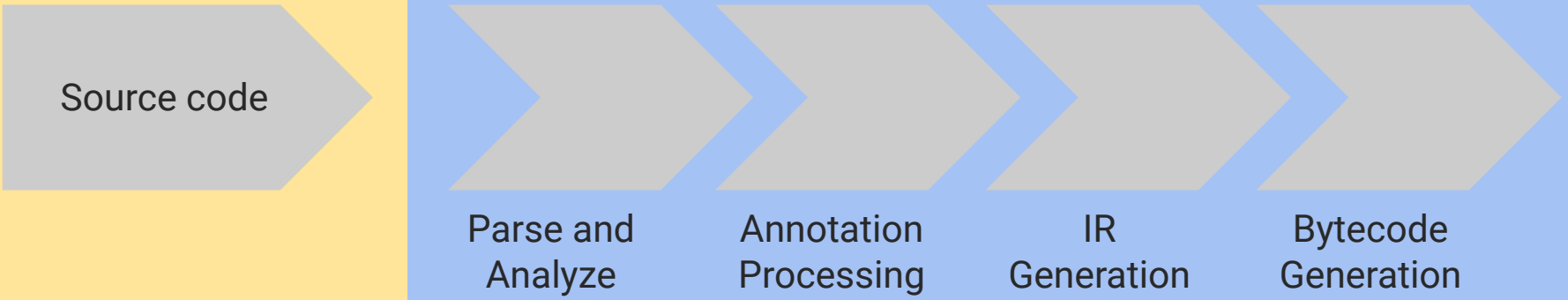
Cons:

- requires knowledge of KSP API
- doesn’t provide the same flexibility as a compiler plugin

Code Editing

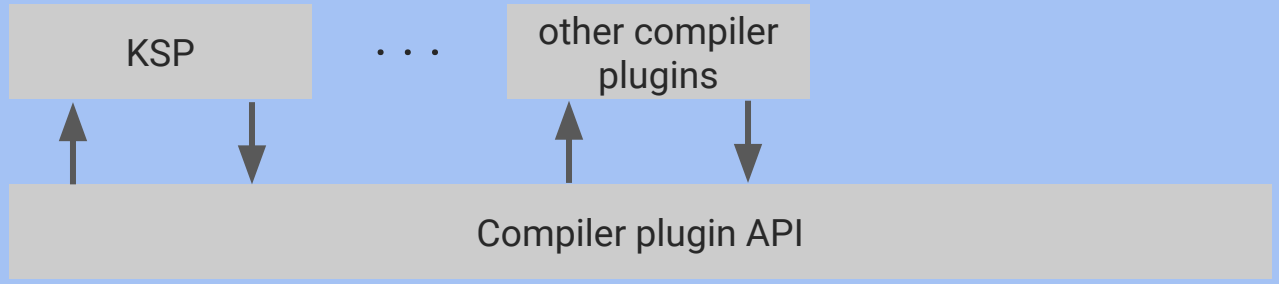
Code Compilation

kotlinc



Code Editing

Code Compilation



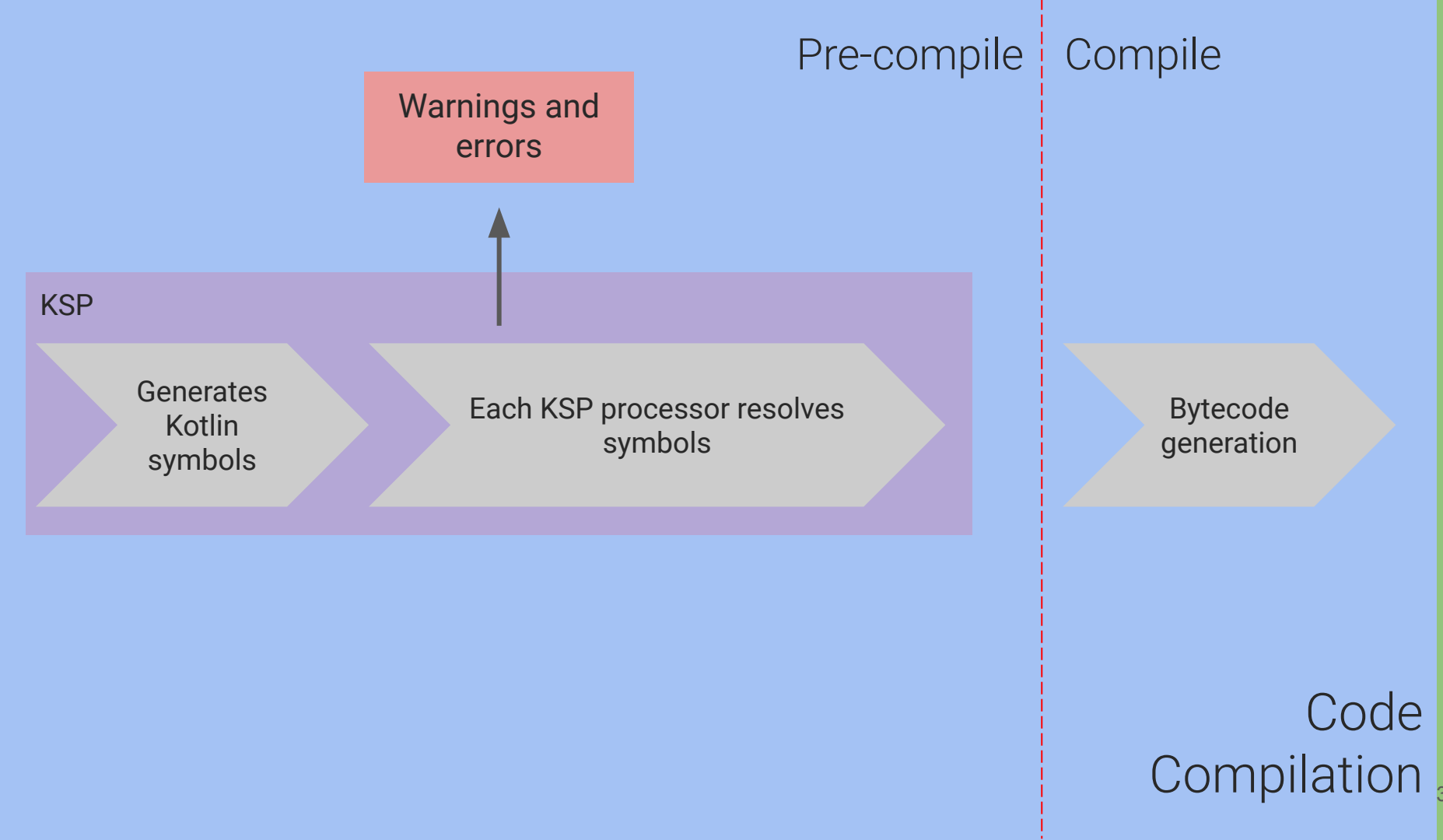
kotlinc

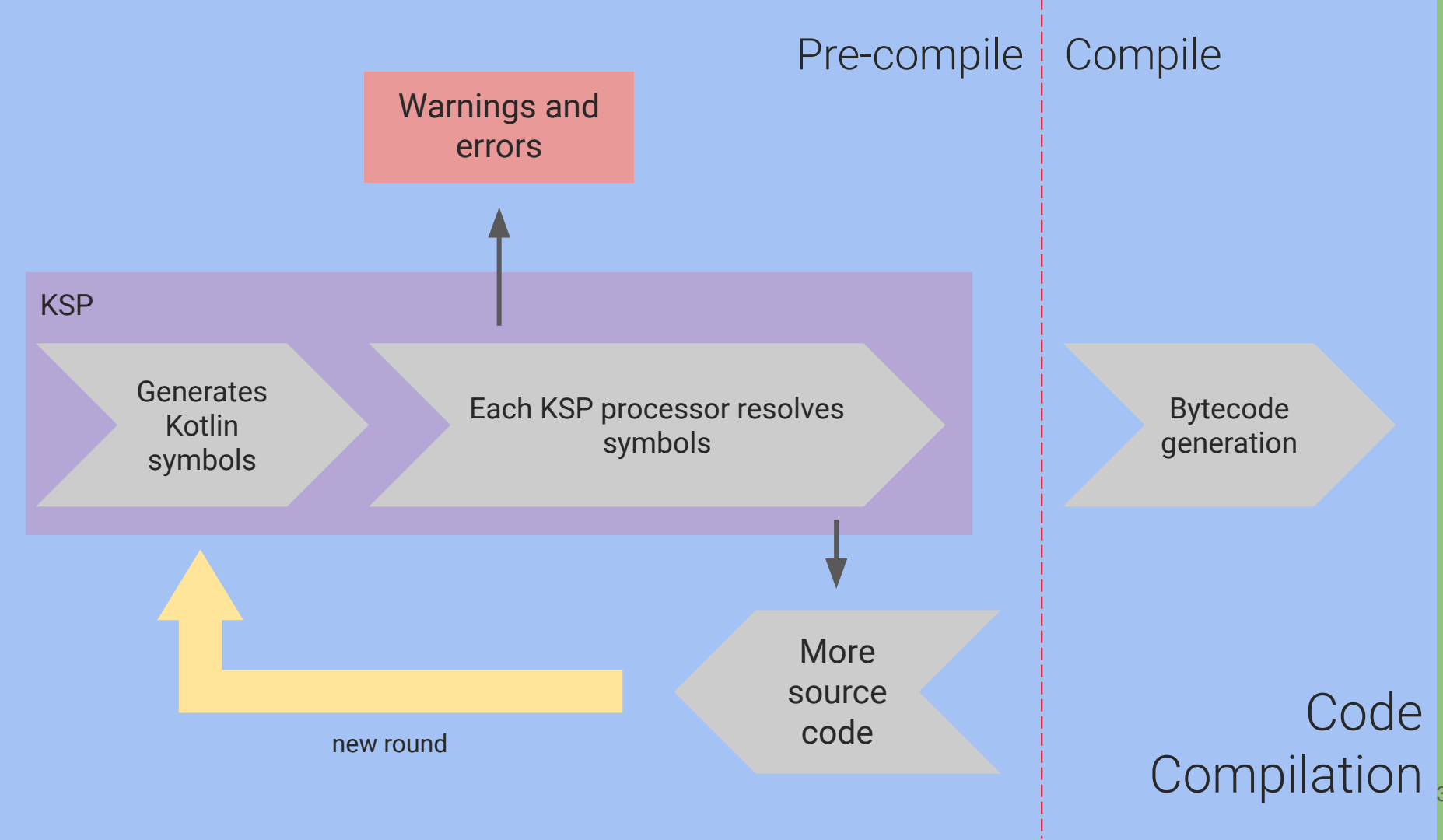
Parse and Analyze

Annotation Processing

IR Generation

Bytecode Generation


Pre-compile
Compile
Warnings and errors
KSP
Generates Kotlin symbols
Each KSP processor resolves symbols
Bytecode generation
Code Compilation

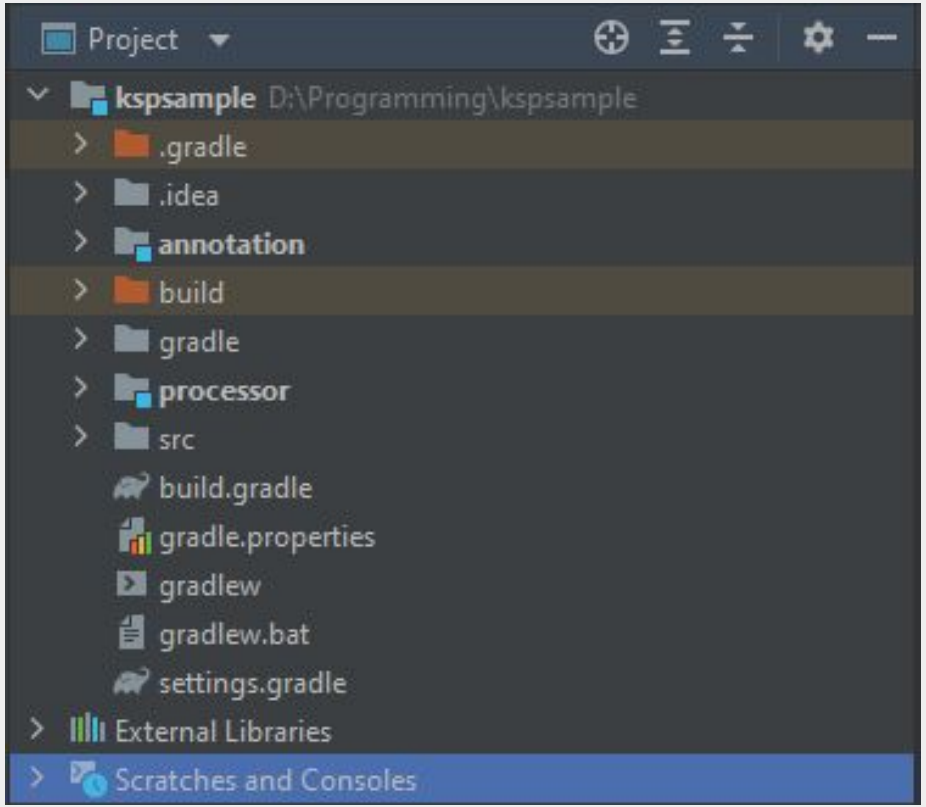

Pre-compile
Compile
Warnings and errors
KSP
Generates Kotlin symbols
Each KSP processor resolves symbols
Bytecode generation
More source code
new round
Code Compilation

KSP. How to?

Building blocks

1. define your custom annotation(s)
2. extend `com.google.devtools.ksp.processing.SymbolProcessor`
3. override `process()` function to provide your custom processing logic
4. extend `com.google.devtools.ksp.processing.SymbolProcessorProvider`
5. override `create()` to provide your custom `SymbolProcessor` implementation
6. register `SymbolProcessorProvider` in `META-INF/services` directory
7. build a JAR and import it into the project
8. configure the build tool to use your JAR dependency as ksp dependency

Building blocks



1. define your custom annotation(s);

```
@Target(AnnotationTarget.VALUE_PARAMETER)
@Retention(AnnotationRetention.SOURCE)
annotation class VilniusKugPrinter {}
```

2. extend `com.google.devtools.ksp.processing.SymbolProcessor`
3. override `process()` function to provide your custom processing logic

```
class VilniusKugPrinterProcessor(
    private val logger: KSPLogger,
) : SymbolProcessor {

   override fun process(resolver: Resolver): List<KSAnnotated> {
       val annotationName: String = VilniusKugPrinter::class.qualifiedName
           ?: throw IllegalStateException()

       resolver.getSymbolsWithAnnotation(annotationName)
           .forEach { annotated ->
             logger.info(annotated.toString())
           }

       return emptyList()
   }
}
```

4. extend `com.google.devtools.ksp.processing.SymbolProcessorProvider`
5. override `create()` to provide your custom `SymbolProcessor` implementation
6. register `SymbolProcessorProvider` in `META-INF/services` directory

```
@AutoService(SymbolProcessorProvider::class)
class VilniusKugPrinterProcessorProvider : SymbolProcessorProvider {

   override fun create(environment: SymbolProcessorEnvironment):
SymbolProcessor {
       return VilniusKugPrinterProcessor(
           logger = environment.logger,
       )
   }
}
```

//Processor module's build.gradle:

```
dependencies {
  implementation "com.google.auto.service:auto-service:1.0.1"
  ksp "dev.zacsweers.autoservice:auto-service-ksp:1.0.0"
  implementation "com.google.auto.service:auto-service-annotations:1.0.0"
}
```

```kotlin
fun main(args: Array<String>) {
    println("Hi!")
}

fun doNothing(
    @VilniusKugPrinter h: Int,
    @VilniusKugPrinter e: Int,
    @VilniusKugPrinter l: Int,
    @VilniusKugPrinter lo: Int,
    @VilniusKugPrinter world: Int,
) {
    return
}
```

./gradlew build --info

```
fun main(arg
    println("
}

fun doNothin
    @VilniusK
    @VilniusK
    @VilniusK
    @VilniusK
    @VilniusK
) {
    return
}
```

```
> Task :kspKotlin
Caching disabled fo sk ':kspKotlin' because:
  Build cache is disabl
Task ':kspKotlin' is not up -date because:
  Value of input property 'kotlinJavaToolchainProvider.javaVersion' has changed for task ':kspKotlin'
The input changes require a full rebuild for incremental task ':kspKotlin'.
Transforming kotlin-stdlib-1.8.10.jar with StructureTransformAction
Transforming annotation-1.0.jar with StructureTransformAction
Transforming kotlin-stdlib-common-1.8.10.jar with StructureTransformAction
Transforming annotations-13.0.jar with StructureTransformAction
'compileJava' task (current target is 18) and 'kspKotlin' task (current target is 1.8) jvm target compatibili
By default will become an error since Gradle 8.0+! Read more: https://kotl.in/gradle/jvm/target-validation
Consider using JVM toolchain: https://kotl.in/gradle/jvm/toolchain

Kotlin source files: D:\Programming\kspsample\src\main\kotlin\com\kavaliou\ksp\sample\Main.kt
Java source files:
Script source files:
Script file extensi .
[KOTLIN] Kotlin c pilation 'jdkHome' argument: E:\Program Files (x86)\jdk
i: found daem n port 17797 (1309508 ms old), trying to connect
i: connecte to the daemon
i: [ksp] loaded provider(s): [com.kavaliou.ksp.processor.VilniusKugPrinterProcessorProvider]
i: [ksp] h
i: [ksp] e
i: [ksp] l
i: [ksp] lo
i: [ksp] world
Resolve mutations for :compileKotlin (Thread[included builds,5,main]) started.
:compileKotlin (Thread[included builds,5,main]) started.
Resolve mutations for :processResources (Thread[Execution worker Thread 2,5,main]) started.
:processResources (Thread[Execution worker Thread 2,5,main]) started.
```

Best practices

Best practices

1. Pay attention at `Dependencies` object – its configuration affects build cache
2. Limit number of traversals through the source code
3. Use KSP class diagram picture from documentation while designing your processing code
4. Use kotlin poet to make kotlin code generation easier
5. Be aware of several rounds of processing
6. Cover processing logic with tests

Conclusions

Decision-making framework

<u>If there is a need:</u>

A. To replace boilerplate code with automated codegen and it's *not possible/reasonable to do using template engines*
B. For *each X* to *create/change* some *Y(s)*
C. To validate something that is *not possible/reasonable to validate with Lint*

<u>then at least **ONE of this questions** should have **"YES"** as answer:</u>

- Will it take less time implement a custom annotation processor than manually do the thing X times?
- Does the subject to be processed always have the same procedure and the only changing pieces are its inputs?
- Does the subject of such automatization really coupled with some other parts of the code base in a way that if the subject changes than the other parts should be changed too?

Sources

- Java AP documentation
- Kotlin AP documentation
- KSP documentation
- Annotation Processing 101
- Kodeco large KSP tutorial
- Tests for KSP project
- Article revealing AP in Lombok
- Article about general usage of AP

- Java APT sample project
- Kotlin KAPT sample project
- KSP sample project

Thanks!

[LinkedIn](#) 

Images and descriptions for several slides were created with the help of “Bing” and “DALL-E”.