KOTLIN GETS REFLECTION

Andrey.Breslav@JetBrains.com

Introspection is examination of one's own conscious thoughts and feelings.

Schultz, D. P.; Schultz, S. E. (2012).

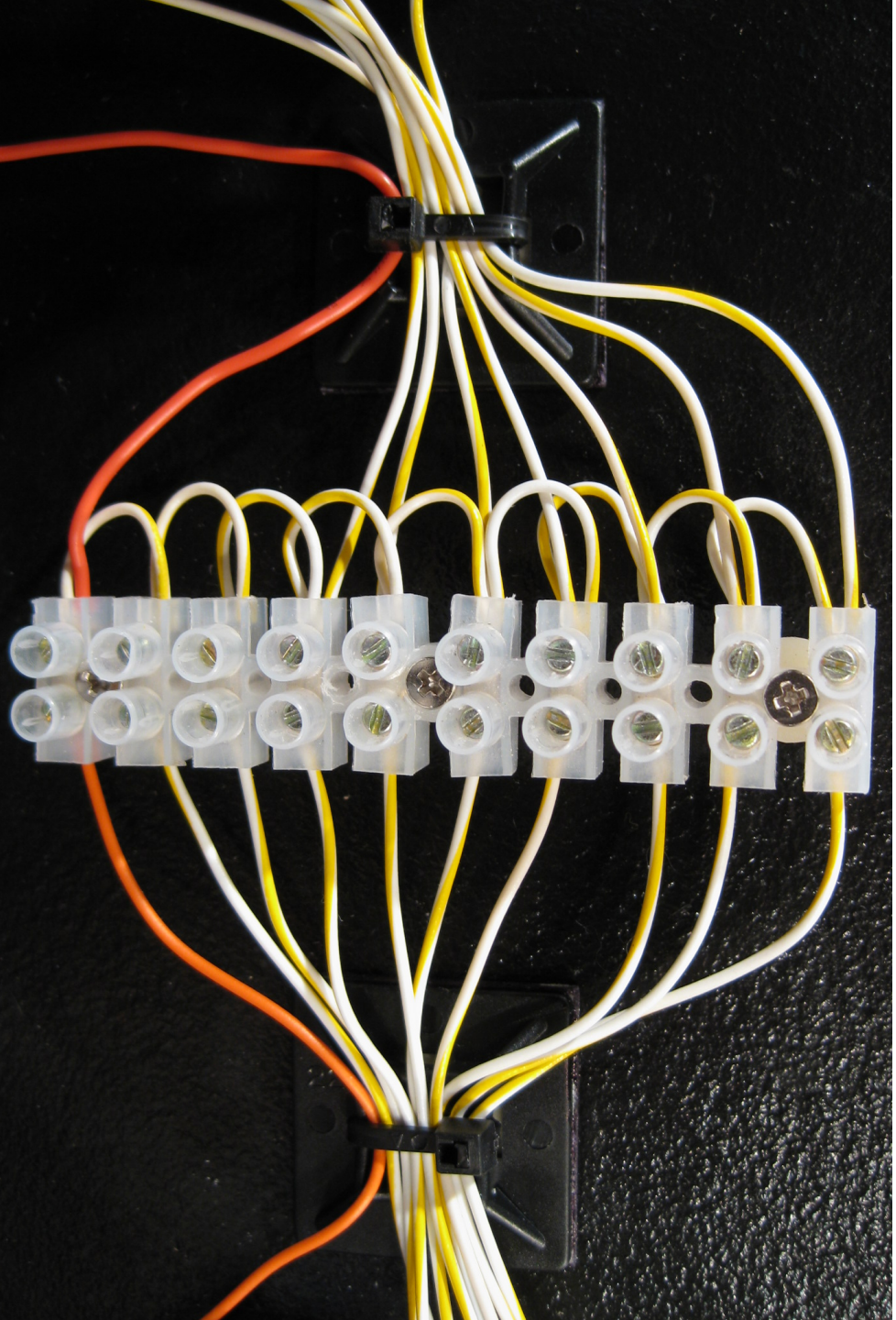
A history of modern psychology (10th ed.)

LEGAL

DON'T JUDGE STRICTLY

OUTLINE

- Intro
- Ways of Introspection
- Reflection API
- Reflection Literals
- Expression Trees
- Conclusion



USE CASES

Dependency Injection

Data binding

Convention over configuration

Hacks

Workarounds

Black Magic

INTROSPECTION

Instances

- what is your class?

Classes & Types

- what are your members? supertypes? create an instance

Methods/Fields

- what are your parameters/types/etc? run with these args.

Expressions

- ???

JAVA.LANG.REFLECT?

Top-level functions

Default arguments

Properties

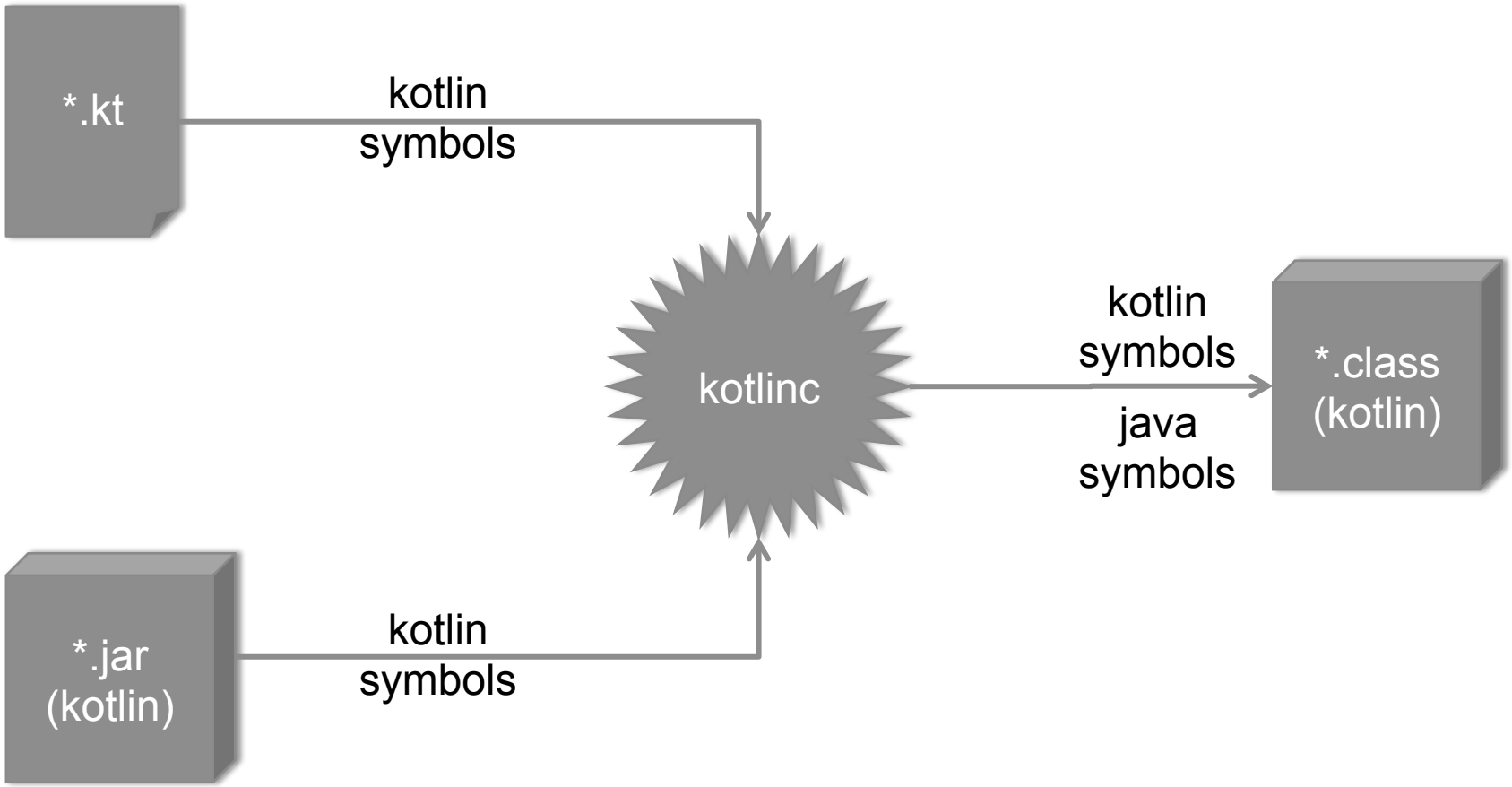
Nullable types

Special types (Nothing, (Mutable)List, etc)

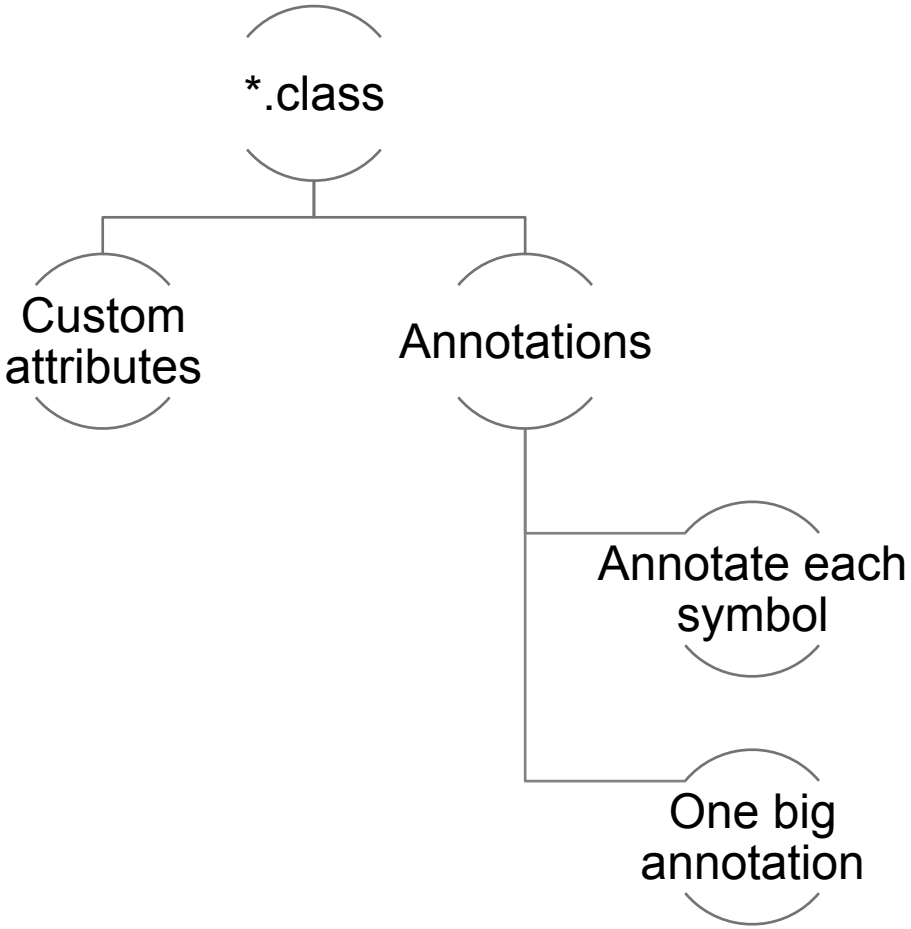
…

+ modules (classes in a package)

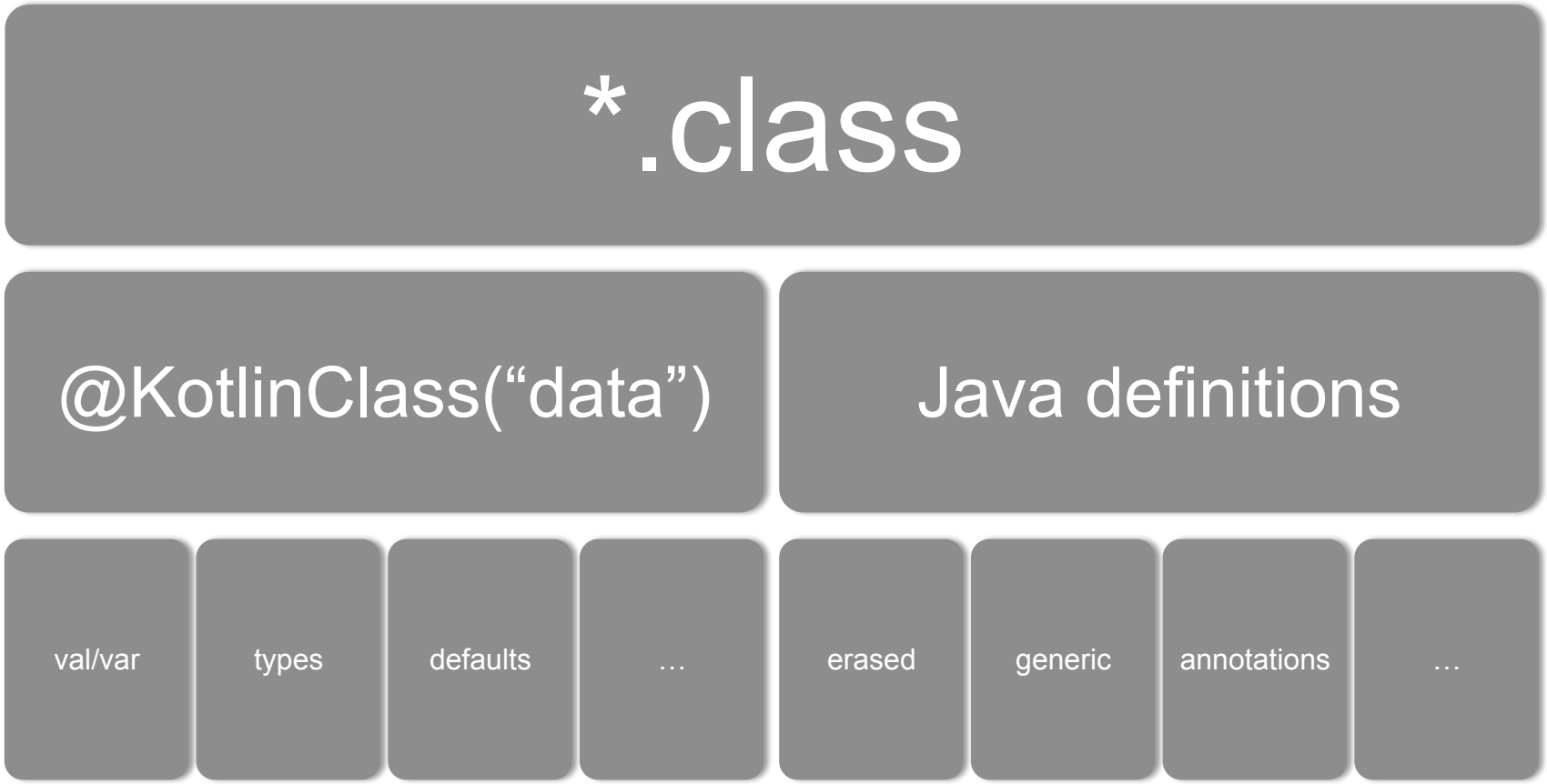
METADATA


*.kt
kotlin
symbols
kotlinc
kotlin
symbols
java
symbols
*.class
(kotlin)
*.jar
(kotlin)
kotlin
symbols

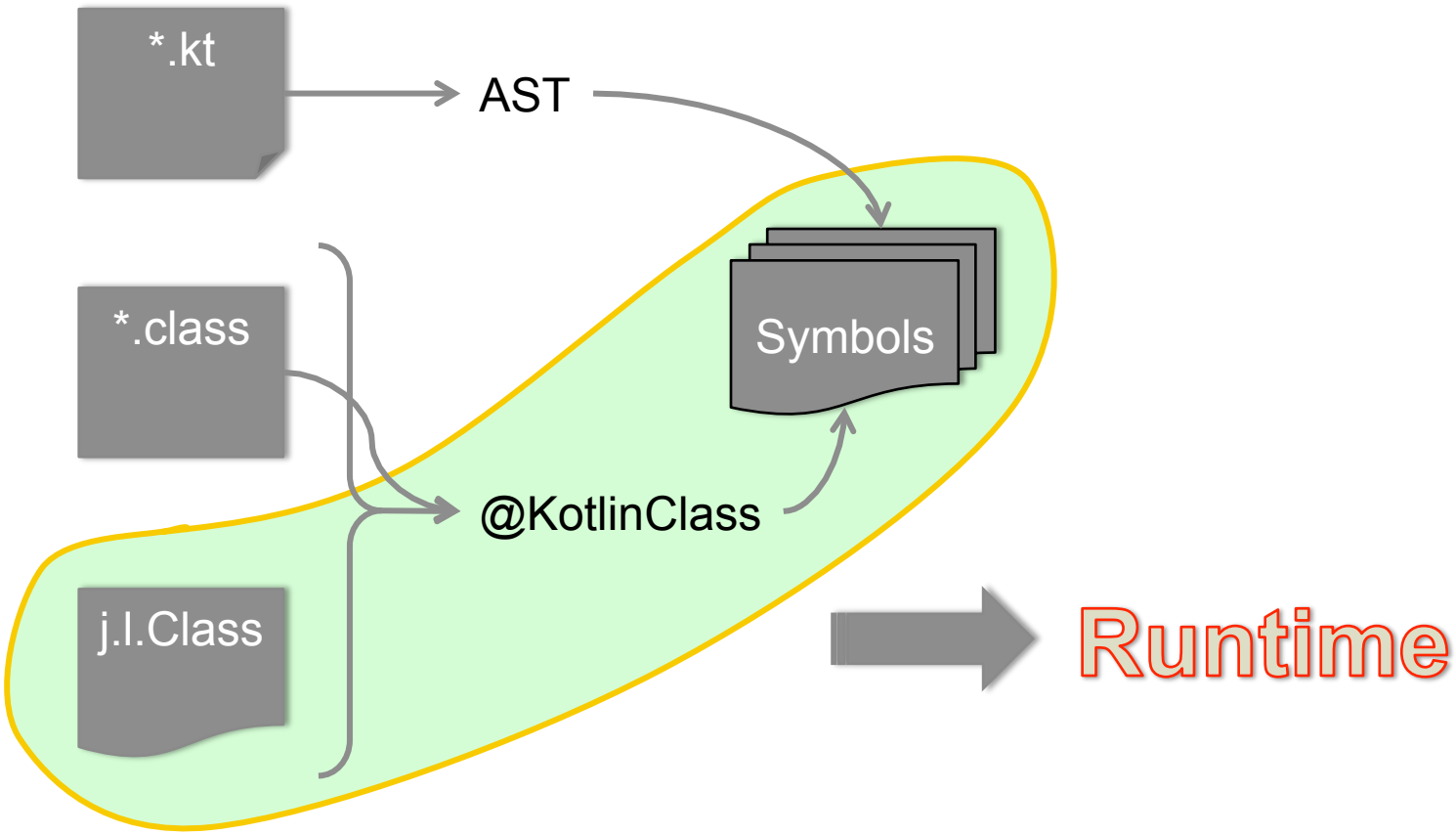
HOW TO STORE METADATA?



ONE BIG ANNOTATION



RE-USE



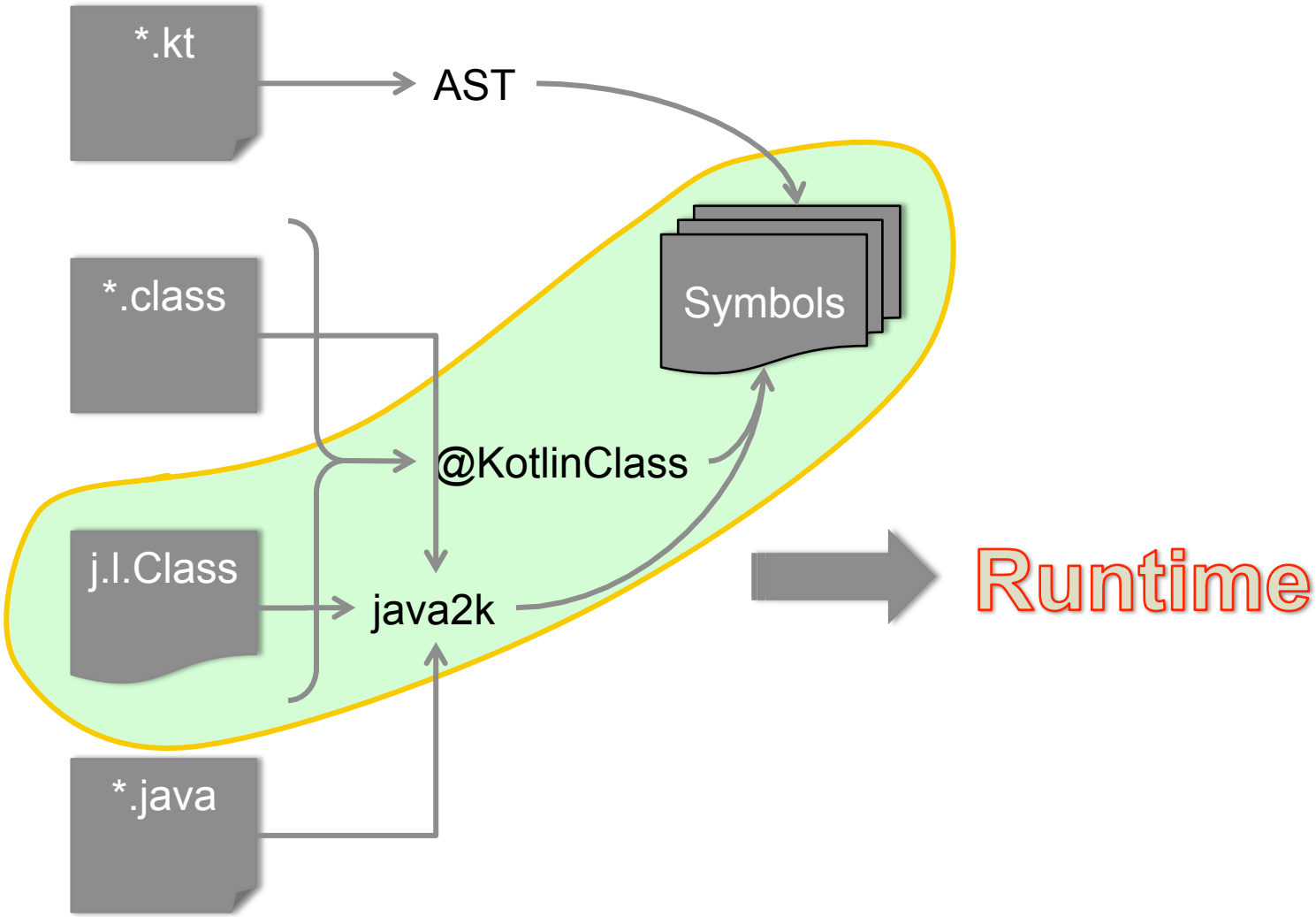
DISCREPANCY 1

java.lang.annotation.Annotation

vs

org.jetbrains.kotlin.internal….AnnotationDescriptor

PURE JAVA CLASSES?

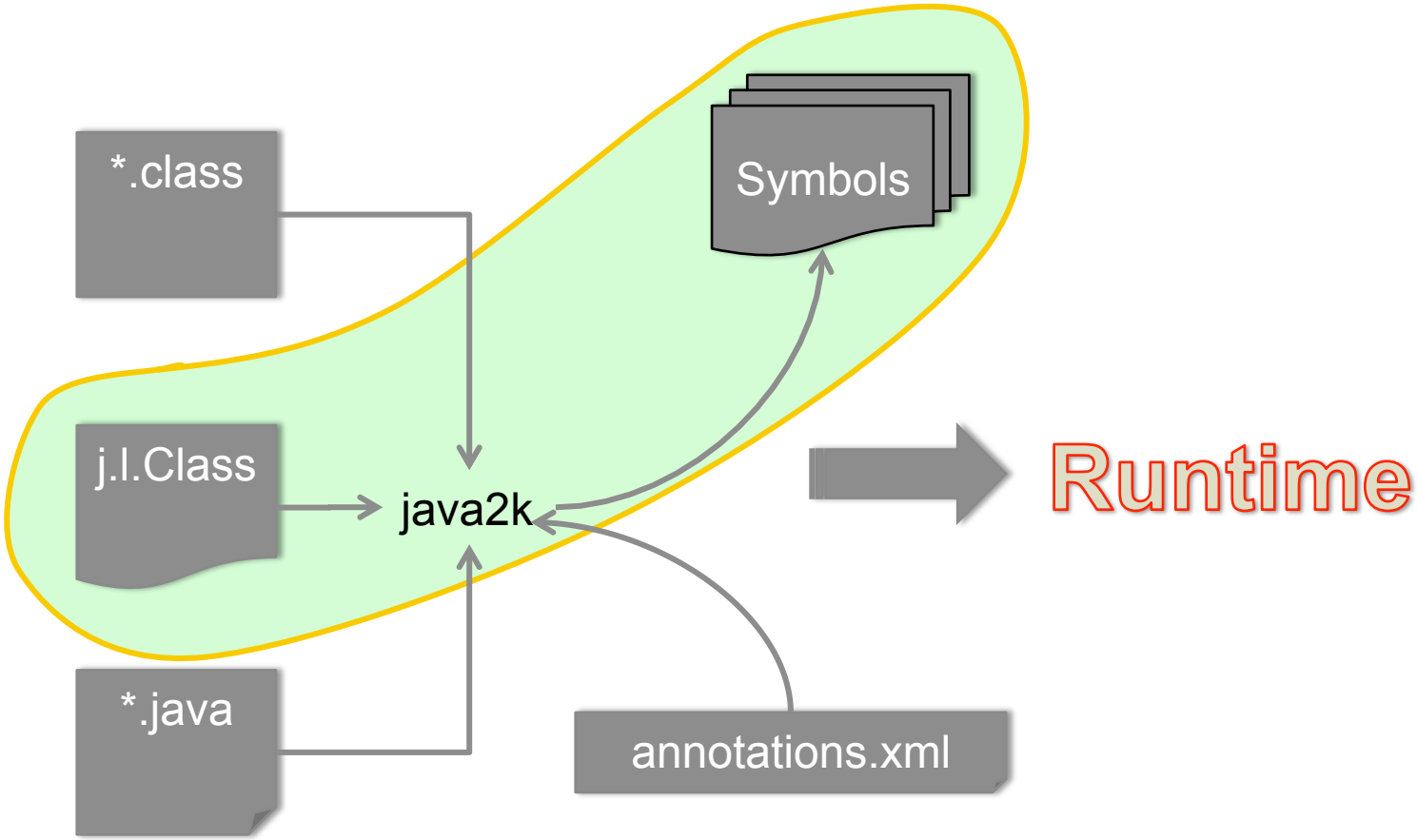


DISCREPANCY 2

```
// Java
class Foo {
    @NotNull
    String getFirstName() { … }

    String getMiddleName() { … }
}
```

```
// Corresponding Kotlin
class Foo {

    fun getFirstName(): String

    fun getMiddleName(): String?
}
```

DISCREPANCY 2



SUMMARY 1

Kotlin-specific reflection API

- works for Java as well

Metadata representation

- one big annotation
- re-use code from the compiler

Problems

- representing annotations
- nullable/mutable types

ENTRY POINTS

SYNTAX (TENTATIVE)

Foo::class

expr::class

List<String>::member

org.sample.bar::member

expr::member

expr::type (maybe)

USE CASES: PASSING CODE AROUND

```
list.filter(Item::isValid)
```

* Why not a lambda?

```
foo {a, b, c -> bar(a, b, c) }
```

USE CASES: CONSTRUCTORS

```
fun <T: Tag> tag(
    create: () -> T, init: T.() -> Unit
) { … }



tag(::DIV) {        // kara.tags::DIV
    …
}
```

USE CASES: DATA BINDING

```
class Model {
      var userName: String by observable()
}



bind(view.textField, model::userName)
```

GENERICS?

```
fun foo(s: String): Foo

                ::foo : (String) -> Foo



fun <T> foo(t: T): T { … }

                  ::foo : ∀T.(T) -> T
```

RANK-2 POLYMORPHISM

ENCODING FOR FUNCTIONS

```
f: (Foo) -> Bar     is      Function1<Foo, Bar>
```

```
interface Function1<P1, R> {
    R invoke(P1 p1);
}
```

GENERIC FUNCTIONS

```
f: <T>(List<T>) -> T
```

- `GenericFunction<List<T>, T> ???`
- `GenericFunction<T, List<T>, T> ???`

```
interface GenericFunction1_1<P1, R> {
    <T> R<T> invoke(P1<T> p1);
}
```

+ Kinds

GENERICS

```
class Foo<T> {
    fun <R> bar(t: T): R { … }
}
```

```
Foo<T'>::bar<R'> : (T') -> R'
```

EXPR::TYPE

```
val x: Any = listOf(1, 2, 3)
x::class   -> java.util.ArrayList
x::type    -> java.util.ArrayList<Unknown>
```

vs

```
val x = listOf(1, 2, 3)      // x: List<Int>
x::class   -> java.util.ArrayList
x::type    -> java.util.ArrayList<Int>
```

DELEGATED PROPERTIES

```
val foos: List<Foo> by Lazy { foos.find(…) }


class Lazy<T>(compute: () -> T) {
    private var value: T? = null

    fun get(me: Any, p: PropertyMetadata): T {
        if (value == null) value = compute()
        return value
    }
}
```

DELEGATED PROPERTIES

```
val foos: List<Foo> by Lazy { foos.find(…) }

::foos::delegate   : Lazy<Foo>
::foos             : Property<List<Foo>>
```

or

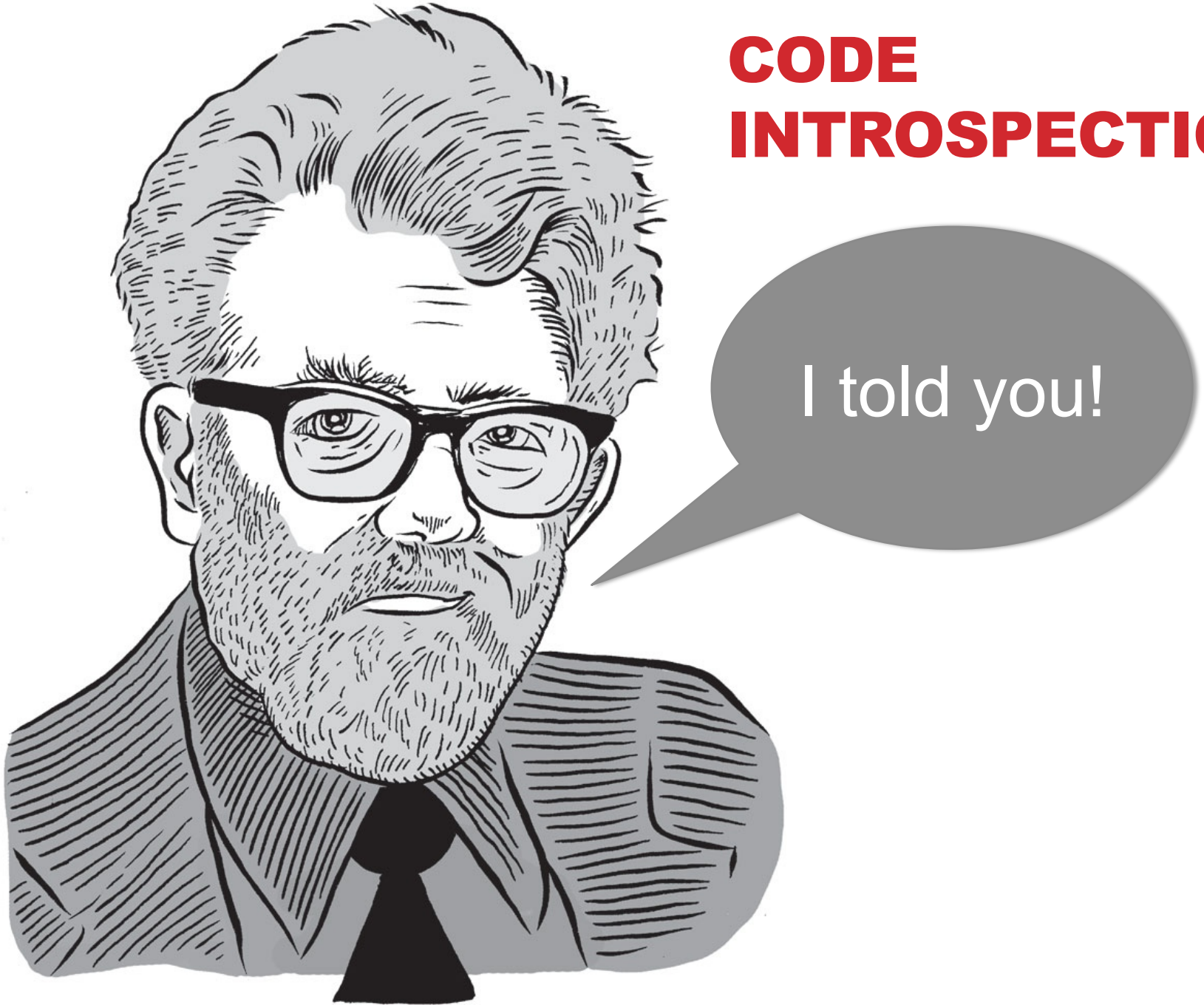
```
::foos.delegate    : Lazy<Foo>
::foos             : DelegatedProperty<
                         List<Foo>,
                         Lazy<Foo>
                       >
```

SUMMARY 2

Reflection literals

- **Static name lookup & typing**
- **Generics are hard, as usual**

CODE INTROSPECTION



USE CASES: LINQ

```
db
    .selectFrom(::User)
    .where { lastName.startsWith(“A”) }
    .orderBy { lastName + firstName }
```

USE CASES: WEB

```
html {
                                                    server
    body {
        onLoad {
            ...                                     client
        }

        ...
                                                    server
    }
}
```

EXPRESSION TREES

```
fun onLoad(ast: Expression<() -> Unit>) {
    ...
}

onLoad {
    ... // compiled to factory calls
}
```

At run time the tree can be translated to JS

SUMMARY

Kotlin's introspection facilities

- **Reflection API**
  - Java interop is the hardest issue
- **Reflection literals**
  - Issues with generics
- **Expression trees**
  - Compiler as a service