Reflection

Reflection

Allows the inspection of code structures at *runtime*

- class declarations
- class components: properties, methods, etc
- return types
- constructors
- type relationships
- access modifiers
- other modifiers (lateinit, nested classes, final, sealed, etc)

Why it's useful

- invoking code dynamically at runtime
- adding functionality that is not known (or is tedious to write) at compile time
- surfacing information about code structure that is only known at runtime (e.g. based on configs)

Class Reflection

Can inspect the structure of classes and all their non-erased internals

```kotlin
val personClass: KClass<Person> = Person::class
```

A KClass has access to

- its name at runtime
- modifiers: final, nested, sealed, abstract
- constructors
- properties
- methods
- nested classes
- companions

```kotlin
println("Class name: ${personClass.simpleName}")
println("JVM name: ${personClass.jvmName}")
println("Qualified name: ${personClass.qualifiedName}")

// class type
println("Class is final: ${personClass.isFinal}")

// visibility
println("access modifier: ${personClass.visibility}")
```

Property Reflection

Can inspect the definition of a class property

```kotlin
val properties = personClass.declaredMemberProperties

properties.forEach { property ->
    println("Name: ${property.name}" +
            "Type: ${property.returnType.classifier}" +
            "Nullable: ${property.returnType.isMarkedNullable}"
    )
}
```

A KProperty can offer

- the name of the property
- the return type - may be a class, but it can also be something else, e.g. a type parameter
- access modifiers
- whether it is final, nullable, open, lateinit, suspend, etc

Method Reflection

Can inspect the entire structure of a method

```kotlin
val functions = personClass.declaredFunctions
functions.forEach { fn ->
    val fnName = fn.name
    val params = fn.parameters
    val returnType = fn.returnType
    println("Function $fnName: ${params.joinToString(",") { it.type.toString() }} -> $returnType")
}
```

A KFunction can offer

- the entire signature: name, return type, parameter list (name + type)
- boolean flags: open, final, suspend, abstract, inline, etc.
- the ability to invoke it on an instance, with a parameter list known at runtime

Type Reflection

Can inspect the structure of a declared type

```kotlin
fun processList(list: List<*>, type: KType) {
    if (type.isSubtypeOf(typeOf<List<String>>())) {
        println("Processing a list of strings")
    } else if (type.isSubtypeOf(typeOf<List<Int>>())) {
        println("List of ints")
    } else {
        println("not supported")
    }
}
```

A KType can show

- the *kind* of type it is: type parameter, class, interface, etc
- its type parameters, if any
- subtype relationships

Powerful combo with reified types

- avoids most problems of type erasure

Annotation Reflection

Annotations can add metadata to declarations

```
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.RUNTIME)
annotation class TestAnnotation(val value: String)
```

Meta-annotations

- @Target – specifies the kind of declaration the annotation can be attached to (default: everything)
- @Retention – where this information can be stored (default: RUNTIME)
  - SOURCE – keep it just during the compiler phase (useful for symbol processing)
  - BINARY – keep it in the bytecode (useful for post-compile tasks)
  - RUNTIME – keep it in the bytecode, accessible via reflection

```
annotation class Table(val name: String)

// ... later
val tableAnnotation: Table? = clazz.findAnnotation<Table>()
```

Kotlin rocks