Kotlin Symbol Processing in Action

Kotlin Symbol Processor

```kotlin
class MyProcessor : SymbolProcessor {
    override fun process(resolver: Resolver): List<KSAnnotated> {
        val files = resolver.getAllFiles()
        files.forEach { file ->
            // myVisitor will analyse every component on File-level
            val myVisitor = KSVisitorVoid()
            file.accept(myVisitor, Unit)
        }
        return files.toList()
    }
}
```

KSVisitor *Complete*

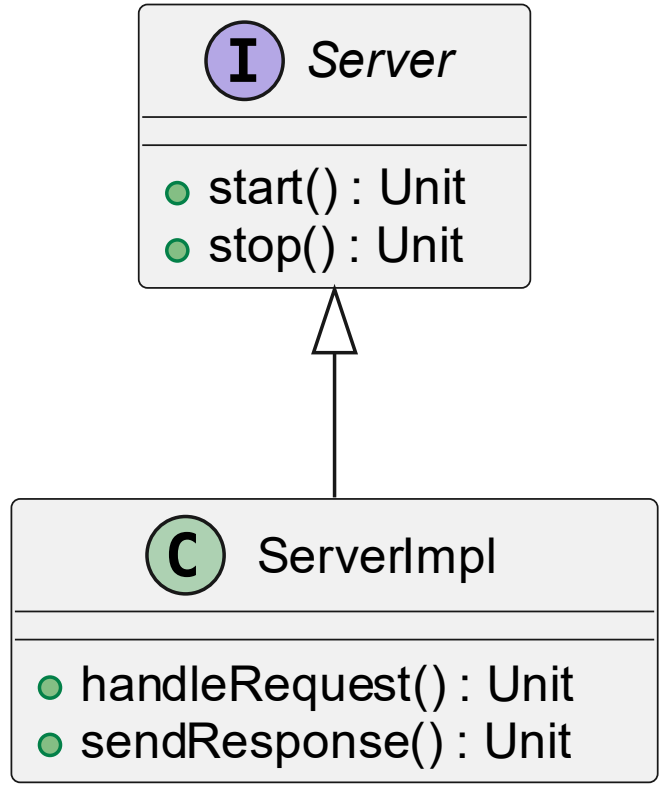
```
interface KSVisitor<D, R> {
    fun visitNode(node: KSNode, data: D): R

    fun visitAnnotated(annotated: KSAnnotated, data: D): R

    fun visitAnnotation(annotation: KSAnnotation, data: D): R

    fun visitModifierListOwner(modifierListOwner: KSModifierListOwner, data: D): R

    fun visitDeclaration(declaration: KSDeclaration, data: D): R

    fun visitDeclarationContainer(declarationContainer: KSDeclarationContainer, data: D): R

    fun visitDynamicReference(reference: KSDynamicReference, data: D): R

    fun visitFile(file: KSFile, data: D): R

    fun visitFunctionDeclaration(function: KSFunctionDeclaration, data: D): R

    fun visitCallableReference(reference: KSCallableReference, data: D): R

    fun visitParenthesizedReference(reference: KSParenthesizedReference, data: D): R

    fun visitPropertyDeclaration(property: KSPropertyDeclaration, data: D): R

    fun visitPropertyAccessor(accessor: KSPropertyAccessor, data: D): R

    fun visitPropertyGetter(getter: KSPropertyGetter, data: D): R

    fun visitPropertySetter(setter: KSPropertySetter, data: D): R

    fun visitReferenceElement(element: KSReferenceElement, data: D): R

    fun visitTypeAlias(typeAlias: KSTypeAlias, data: D): R

    fun visitTypeArgument(typeArgument: KSTypeArgument, data: D): R

    fun visitClassDeclaration(classDeclaration: KSClassDeclaration, data: D): R

    fun visitTypeParameter(typeParameter: KSTypeParameter, data: D): R

    fun visitTypeReference(typeReference: KSTypeReference, data: D): R

    fun visitValueParameter(valueParameter: KSValueParameter, data: D): R

    fun visitValueArgument(valueArgument: KSValueArgument, data: D): R

    fun visitClassifierReference(reference: KSClassifierReference, data: D): R
}
```

KSVisitor *Focused*

```kotlin
/**
 * A visitor for program elements
 */
interface KSVisitor<D, R> {
    fun visitFile(
        file: KSFile,
        data: D
    ): R

    fun visitFunctionDeclaration(
        function: KSFunctionDeclaration,
        data: D
    ): R

    fun visitPropertyDeclaration(
        property: KSPropertyDeclaration,
        data: D
    ): R

    fun visitTypeAlias(
        typeAlias: KSTypeAlias,
        data: D
    ): R

    fun visitClassDeclaration(
        classDeclaration: KSClassDeclaration,
        data: D
    ): R
}
```


We need a Server that can handle Requests and send Responses.
Ok.
We will do that internally and you can just start/stop it.

```kotlin
public interface Server {
    fun start(): Unit
    fun stop(): Unit
}

internal class ServerImpl : Server {
    override fun start(): Unit{}
    override fun stop(): Unit{}
    fun handleRequest(): Unit{}
    fun sendResponse(): Unit{}
}
```



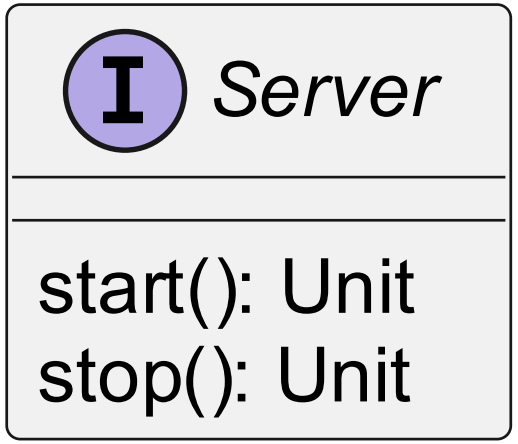
These properties can be used to analyse the class

```
interface KSClassDeclaration : KSDeclaration, KSDeclarationContainer {

    val classKind: ClassKind

    val primaryConstructor: KSFunctionDeclaration?

    val superTypes: Sequence<KSTypeReference>

    val isCompanionObject: Boolean

    fun getSealedSubclasses(): Sequence<KSClassDeclaration>

    fun getAllFunctions(): Sequence<KSFunctionDeclaration>

    fun getAllProperties(): Sequence<KSPropertyDeclaration>

    fun asType(typeArguments: List<KSTypeArgument>): KSType

    fun asStarProjectedType(): KSType
}
```


Ok.
Can you add Configuration please

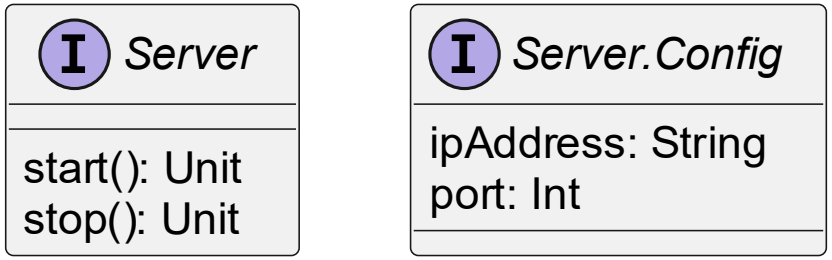
```kotlin
public interface Server {
    fun start(): Unit
    fun stop(): Unit

    interface Config {
        val ipAddress: String
        val port: Int
    }
}
```

Generated Documentation



```kotlin
public interface Server {
    fun start(): Unit
    fun stop(): Unit

    interface Config {
        val ipAddress: String
        val port: Int
    }
}
```

Expected Generated Documentation



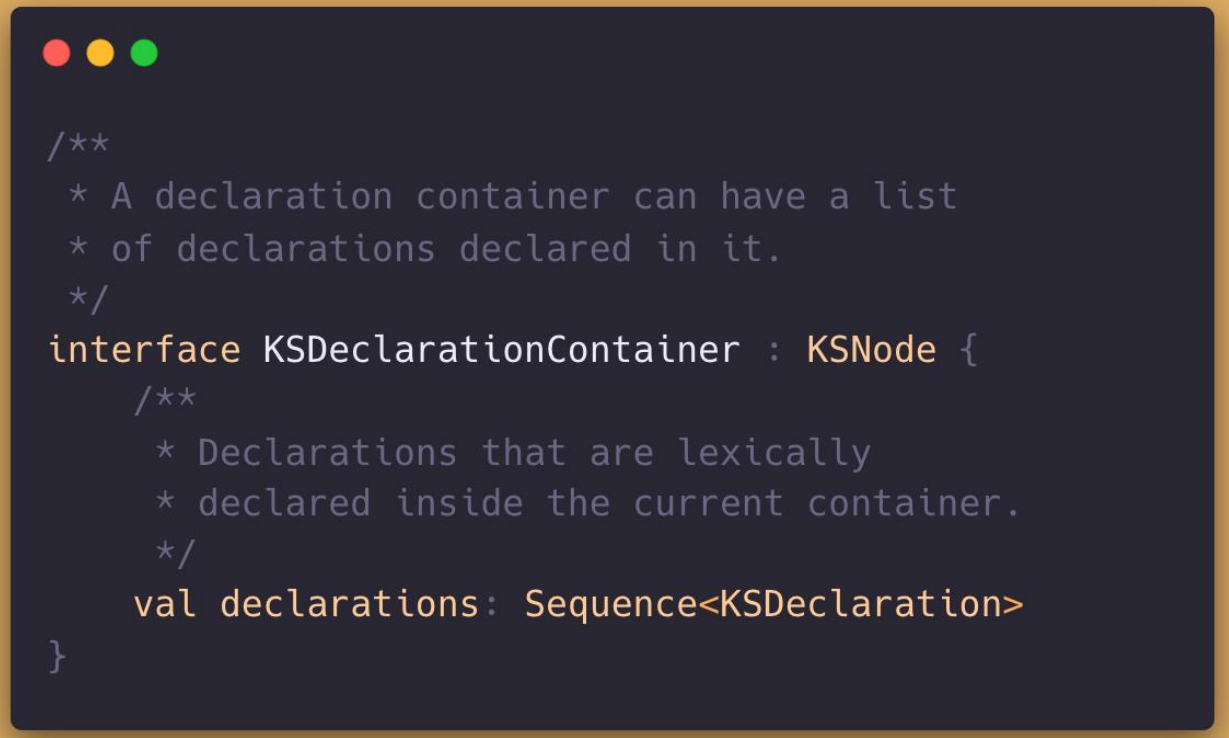
Why is the inner class missing ?

```
interface KSVisitor<D, R> {
    fun visitFile(file: KSFile, data: D): R

    fun visitFunctionDeclaration(
        function: KSFunctionDeclaration,
        data: D
    ): R

    fun visitPropertyDeclaration(
        property: KSPropertyDeclaration,
        data: D
    ): R

    fun visitTypeAlias(
        typeAlias: KSTypeAlias,
        data: D
    ): R

    fun visitClassDeclaration(
        classDeclaration: KSClassDeclaration,
        data: D
    ): R
}
```

- The Visitor is applied to the given File
- and its nodes
- The inner class *Config* was not visited
- If a node has further nodes that should be visited, the visitor must be passed within the node

```
1 // File Level (KSVisitor.visitFile)
2 public interface Server {
3     // Class Level (KSVisitor.visitClassDeclaration)
4
5     // Class Level (KSVisitor.visitFunctionDeclaration)
6     fun start(): Unit
7     fun stop(): Unit
8
9     // Class Level (KSVisitor.visitDeclaration)
10    // ! There is no KSVisitor.visitInnerClassDeclaration
11    interface Config {
12        val ipAddress: String
13        val port: Int
14    }
15 }
16
```

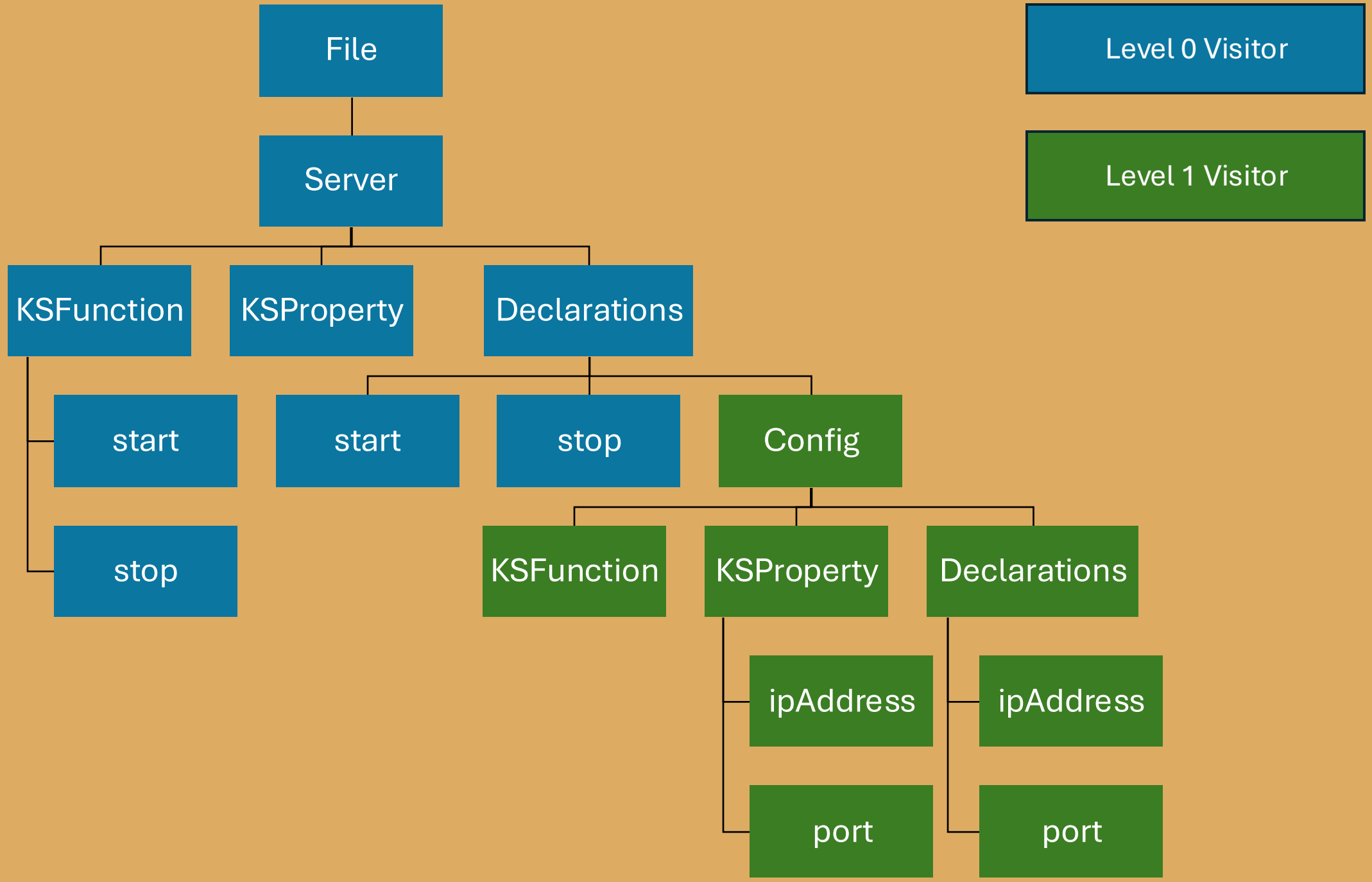
But to access the inner classes, *declarations* must be used



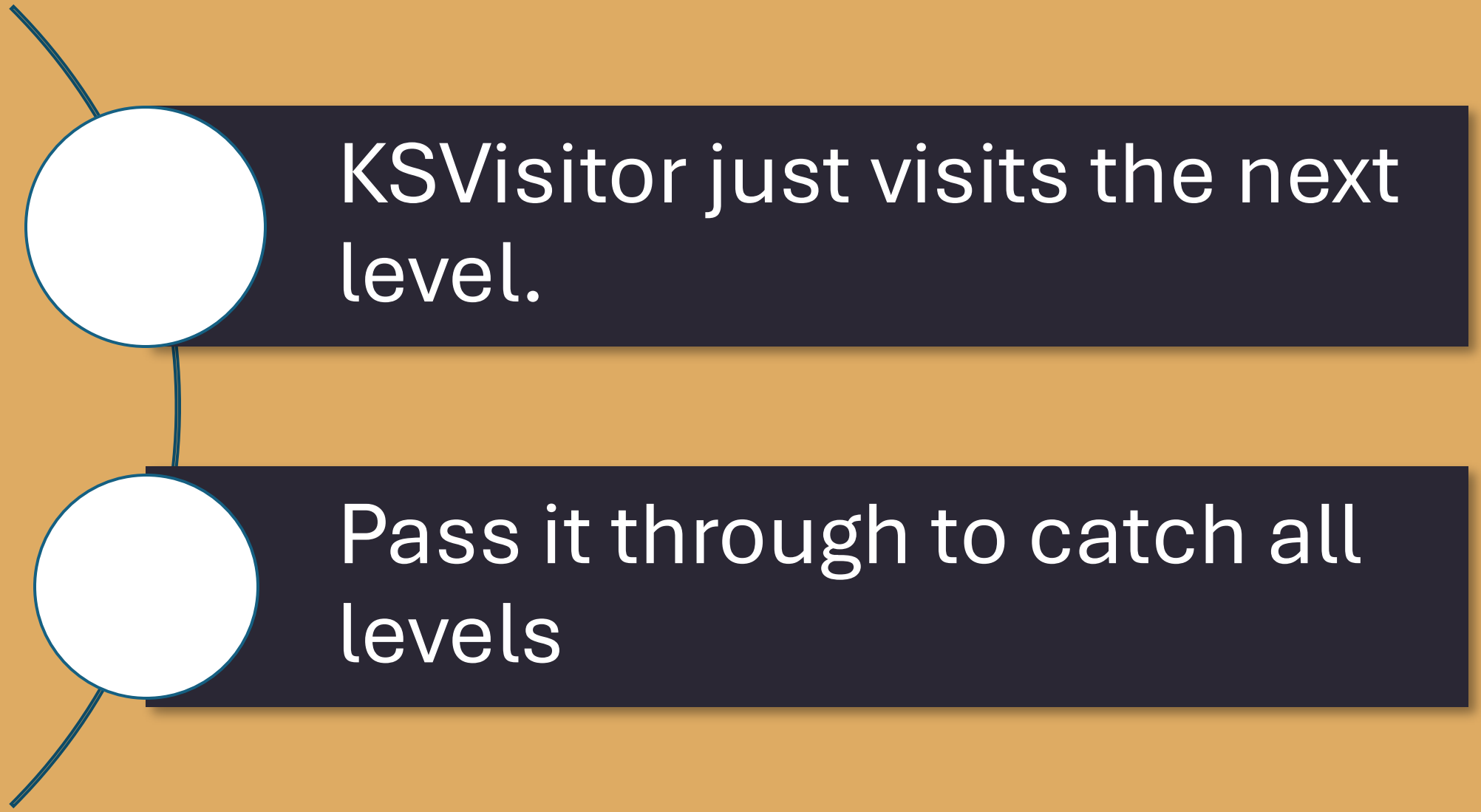
```
/**
 * A declaration container can have a list
 * of declarations declared in it.
 */
interface KSDeclarationContainer : KSNode {
    /**
     * Declarations that are lexically
     * declared inside the current container.
     */
    val declarations: Sequence<KSDeclaration>
}
```

```kotlin
class MyKSVisitor() : KSVisitorVoid() {

    override fun visitFile(
        file: KSFile,
        data: Unit
    ) {
        file.declarations.forEach { declaration ->
            declaration.accept(this, Unit)
        }
    }

    override fun visitClassDeclaration(
        classDeclaration: KSClassDeclaration,
        data: Unit
    ) {
        classDeclaration.getDeclaredFunctions().forEach { function ->
            analyseFunction(function)
        }
        classDeclaration.getDeclaredProperties().forEach { property ->
            analyseProperty(property)
        }

        // This is what I missed first
        classDeclaration.declarations
            .filterNot { it in classDeclaration.getDeclaredProperties() }
            .filterNot { it in classDeclaration.getDeclaredFunctions() }
            .filterIsInstance<KSClassDeclaration>()
            .forEach { declaration ->
                declaration.accept(this, Unit)
            }
    }

    private fun analyseFunction(function: KSFunctionDeclaration) {}
    private fun analyseProperty(function: KSPropertyDeclaration) {}
}
```

```
}

override fun visitClassDeclaration(
    classDeclaration: KSClassDeclaration,
    data: Unit
) {
    classDeclaration.getDeclaredFunctions().forEach { function ->
        analyseFunction(function)
    }
    classDeclaration.getDeclaredProperties().forEach { property ->
        analyseProperty(property)
    }

    // This is what I missed first
    classDeclaration.declarations
        .filterNot { it in classDeclaration.getDeclaredProperties() }
        .filterNot { it in classDeclaration.getDeclaredFunctions() }
        .filterIsInstance<KSClassDeclaration>()
        .forEach { declaration ->
            declaration.accept(this, Unit)
        }
}

private fun analyseFunction(function: KSFunctionDeclaration) {}
```


File
Server
KSFunction
KSProperty
Declarations
start
stop
start
stop
Config
KSFunction
KSProperty
Declarations
ipAddress
port
ipAddress
port
Level 0 Visitor
Level 1 Visitor

Advice 1


KSVisitor just visits the next level.
Pass it through to catch all levels

Advice 2

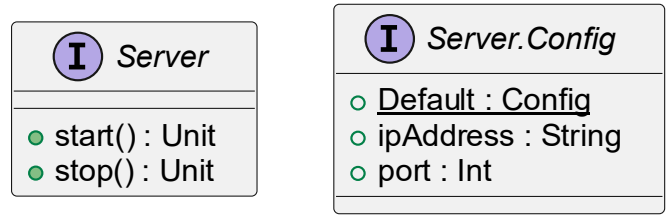
- Understand the KSClassContainer component
- Inner Classes, Interfaces, companion objects can be retrieved by the declarations property
- declarations contains every KSDeclaration, not only the ones for inner classes etc.

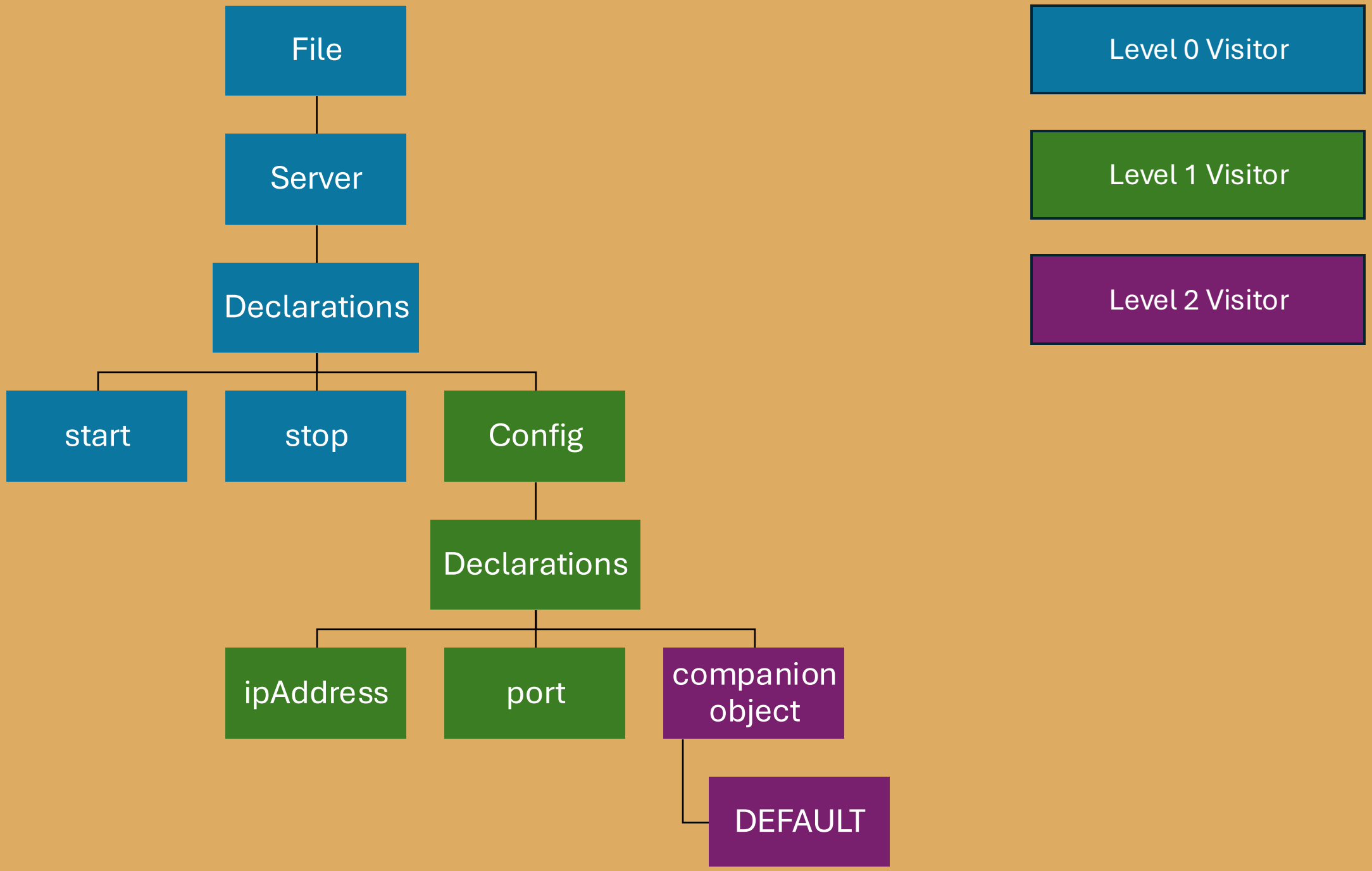
Advice 3

- Understand the KSClassDeclaration component
- It provides superTypes (parents)
- It provides allFunctions
- It provides allProperties
- It provides its classKind


Almost, I will add Default config
Done ?

```kotlin
public interface Server {
    fun start(): Unit
    fun stop(): Unit

    interface Config {
        val ipAddress: String
        val port: Int

        companion object {
            val Default: Config = object : Config {
                override val ipAddress: String = "127.0.0.1"
                override val port: Int = 8080
            }
        }
    }
}
```




File
Server
Declarations
start
stop
Config
Declarations
ipAddress
port
companion object
DEFAULT
Level 0 Visitor
Level 1 Visitor
Level 2 Visitor

```kotlin
interface KSClassDeclaration : KSDeclaration, KSDeclarationContainer {

    val classKind: ClassKind

    val primaryConstructor: KSFunctionDeclaration?

    val superTypes: Sequence<KSTypeReference>

    val isCcompanionObject: Boolean

    fun getSealedSubclasses(): Sequence<KSClassDeclaration>

    fun getAllFunctions(): Sequence<KSFunctionDeclaration>

    fun getAllProperties(): Sequence<KSPropertyDeclaration>

    fun asType(typeArguments: List<KSTypeArgument>): KSType

    fun asStarProjectedType(): KSType
}
```

```kotlin
/**
 * Kind of a class declaration.
 * Interface, class, enum class and object
 * are all considered a class declaration.
 */
enum class ClassKind(val type: String) {
    INTERFACE("interface"),
    CLASS("class"),
    ENUM_CLASS("enum_class"),
    ENUM_ENTRY("enum_entry"),
    OBJECT("object"),
    ANNOTATION_CLASS("annotation_class")
}
```

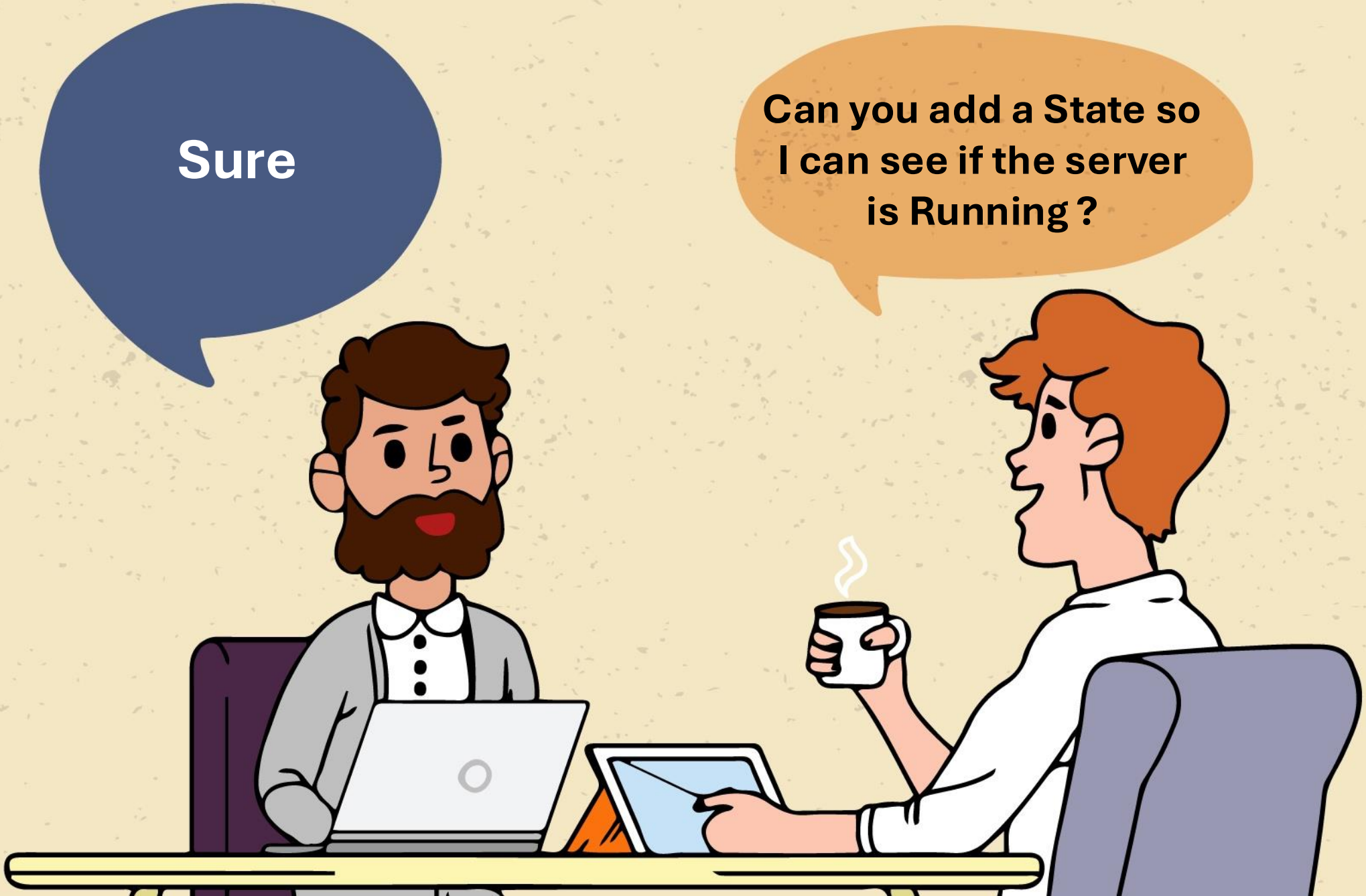
Advice 4

```
fun getCompanionObjectDeclaration(
    clazz: KSClassDeclaration
): KSClassDeclaration? {
    return clazz
        .declarations
        .filterIsInstance<KSClassDeclaration>()
        .filter { it.classKind == ClassKind.OBJECT }
        .filter { it.isCompanionObject }
        .firstOrNull()
}
```

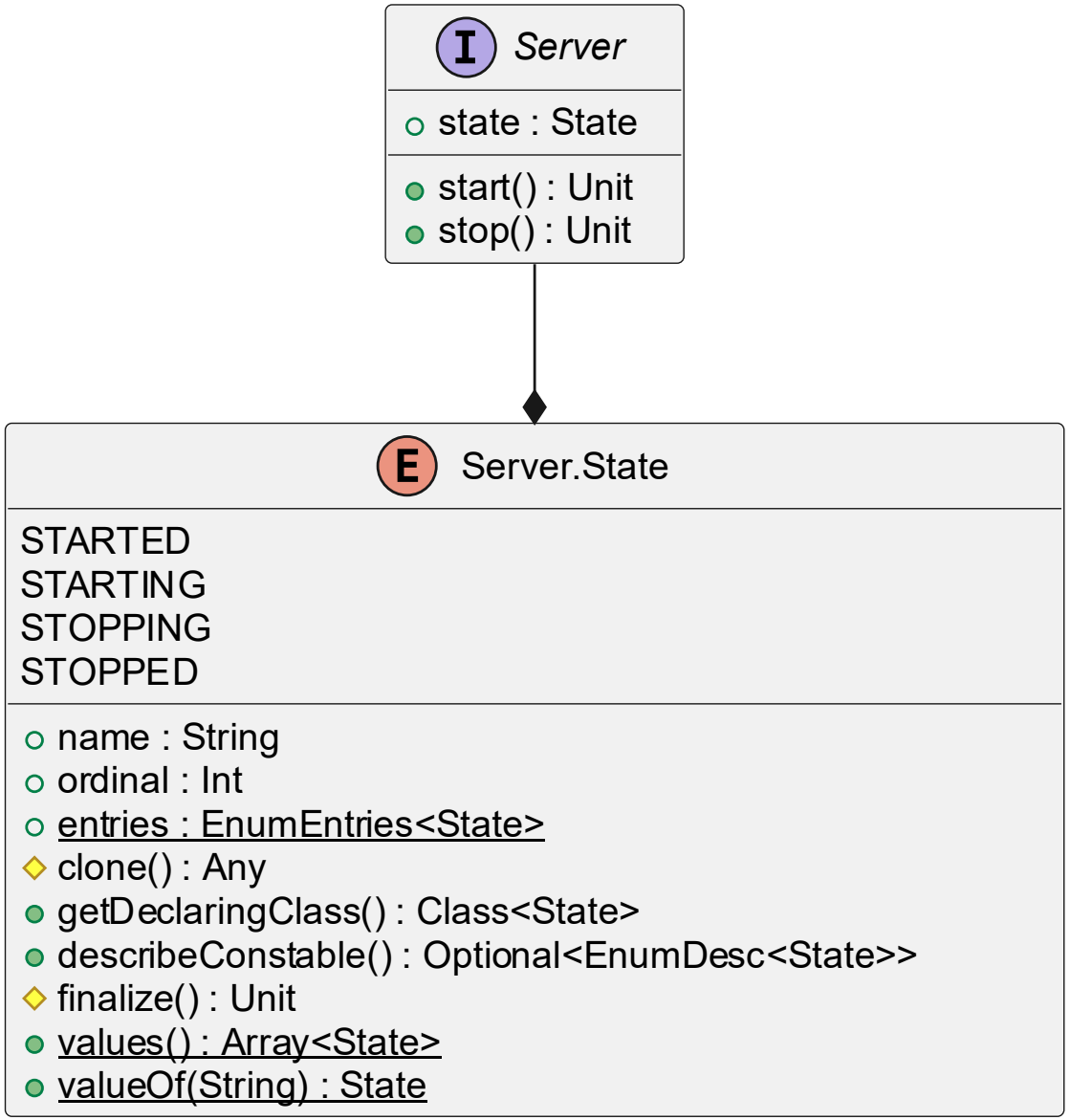
```
fun getCompanionObjectProperties(clazz: KSClassDeclaration) : List<KSPropertyDeclaration>{
    return getCompanionObjectDeclaration(clazz)?.getAllProperties()?.toList() ?: emptyList()
}

fun getCompanionObjectFunctions(clazz: KSClassDeclaration) : List<KSFunctionDeclaration>{
    return getCompanionObjectDeclaration(clazz)?.getAllFunctions()?.toList() ?: emptyList()
}
```

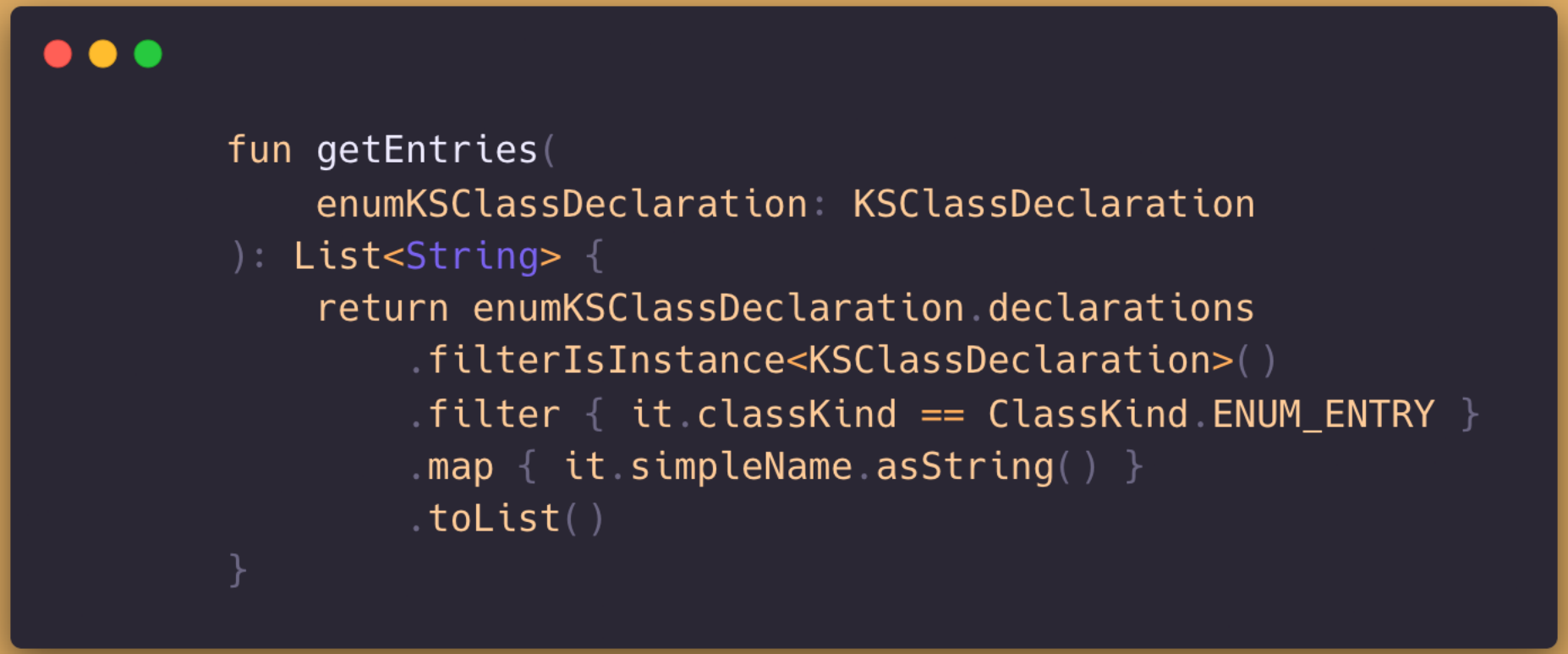
Advice 5

- Every class can have max. 1 companion object
- All functions and properties of this a companion object can be intepreted like Java static
- To find the companion object of a class, use declarations and filter on the Object classKind + KSClassDeclaration isCompanionObject property


Sure
Can you add a State so I can see if the server is Running ?

```kotlin
public interface Server {
    fun start(): Unit
    fun stop(): Unit
    val state: State

    enum class State {
        STARTED,
        STARTING,
        STOPPING,
        STOPPED
    }
}
```





```
fun getEntries(
    enumKSClassDeclaration: KSClassDeclaration
): List<String> {
    return enumKSClassDeclaration.declarations
        .filterIsInstance<KSClassDeclaration>()
        .filter { it.classKind == ClassKind.ENUM_ENTRY }
        .map { it.simpleName.asString() }
        .toList()
}
```

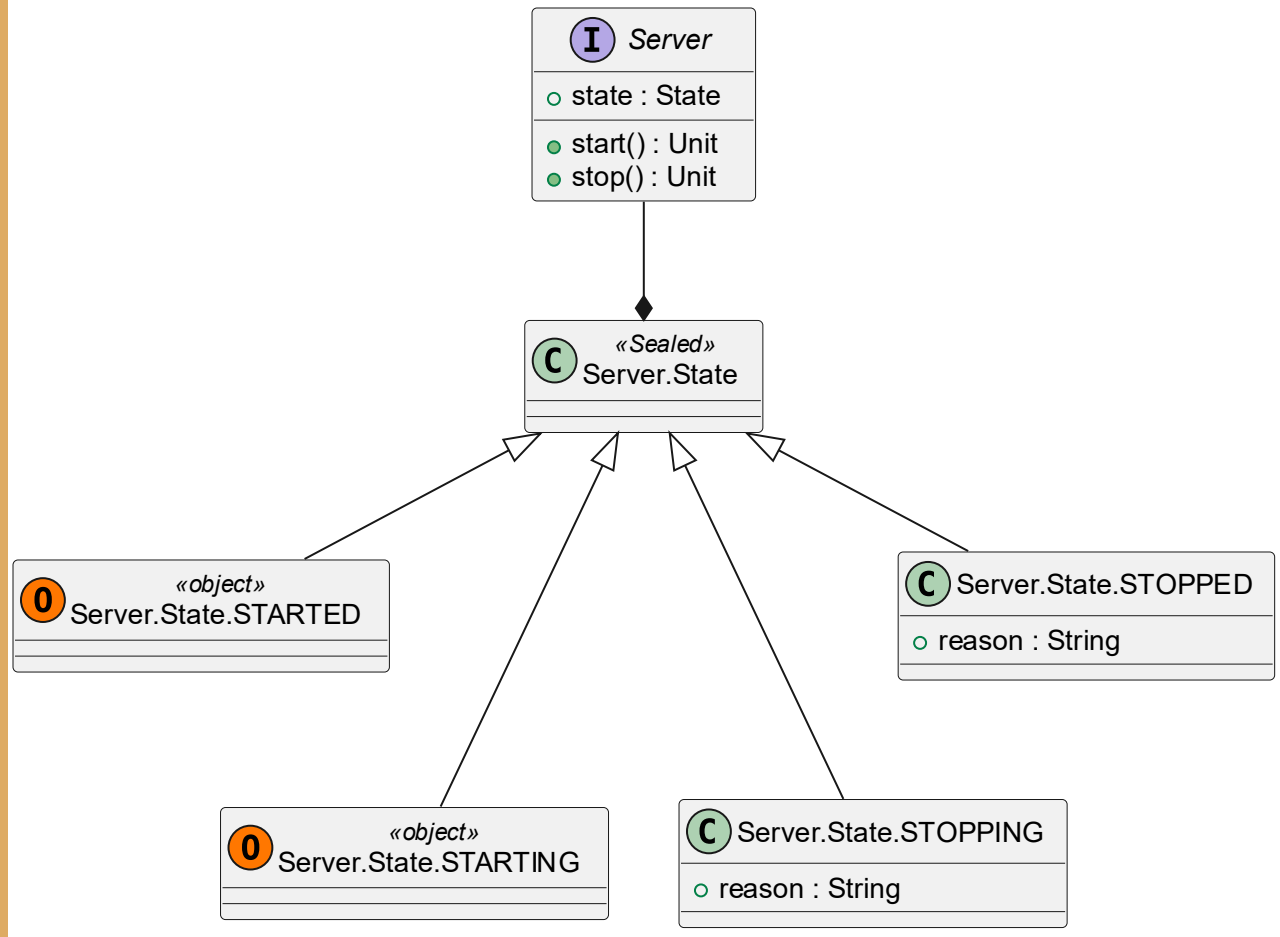
Advice 6

Each Enum has a superType: *Enum <T>*

Enum entries can be retrieved by the EnumEntry classes in its *declarations*


Sure
Yesterday it got turned off, but I have no Idea why, can you add that to the State ?

```kotlin
public interface Server {
    fun start(): Unit
    fun stop(): Unit
    val state: State

    sealed class State() {
        object STARTED : State()
        object STARTING : State()
        class STOPPING(val reason: String) : State()
        class STOPPED(val reason: String) : State()
    }
}
```



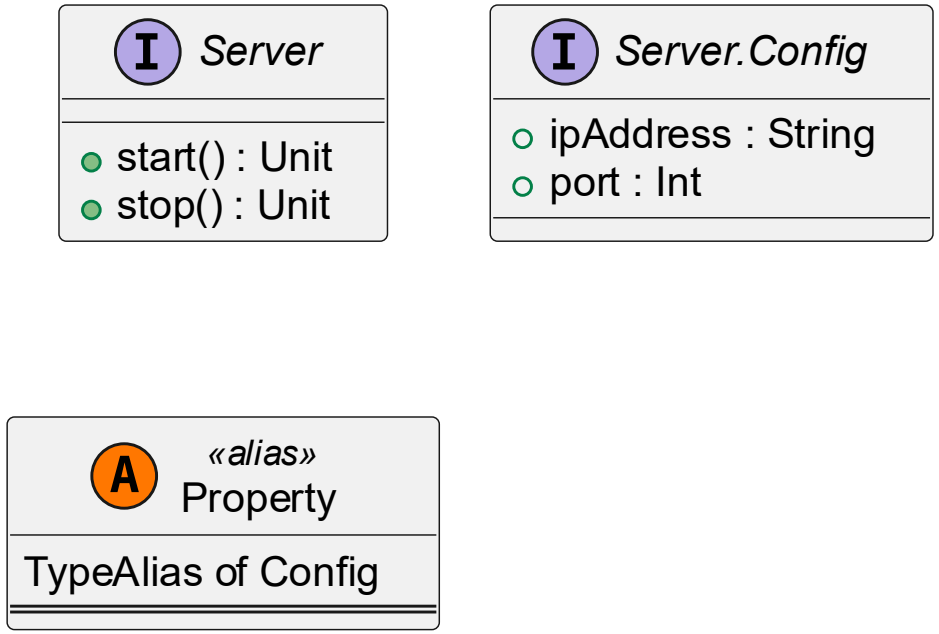
Advice 7

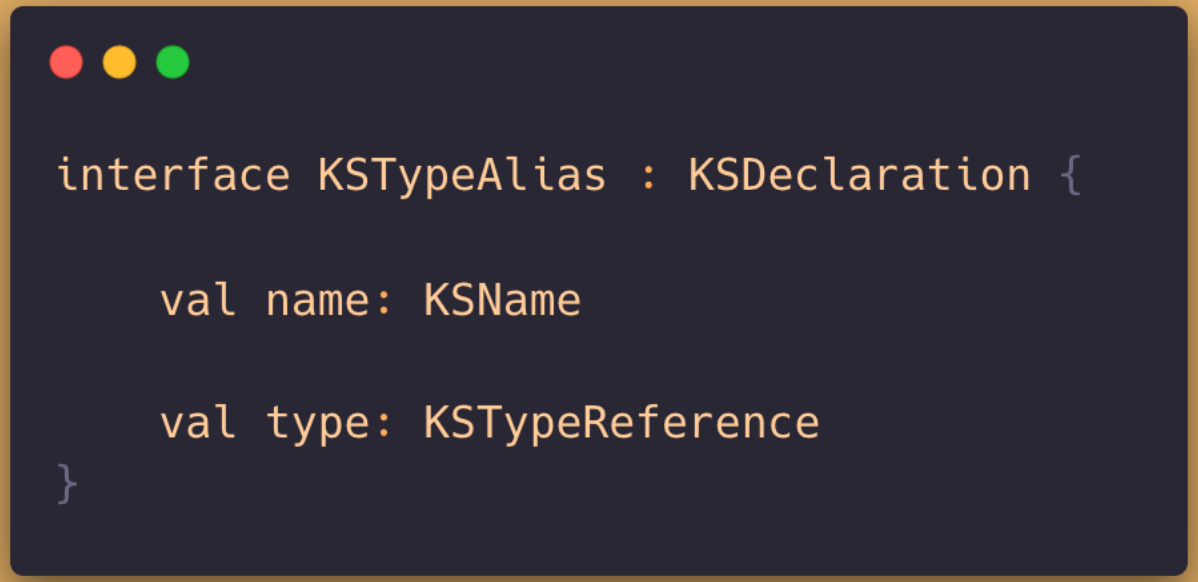
A sealed class can be identified by the *modifiers* property

Sealed classes can be interpreted like a group of classes that inherit a shared parent class


Do you mind if we do internally ? If not I will provide an TypeAlias !
We dont say Config anymore. We call it Properties now.

```kotlin
public interface Server {
    fun start(): Unit
    fun stop(): Unit

    interface Config {
        val ipAddress: String
        val port: Int
    }
}

typealias Property = Server.Config
```




interface KSTypeAlias : KSDeclaration {
val name: KSName
val type: KSTypeReference
}

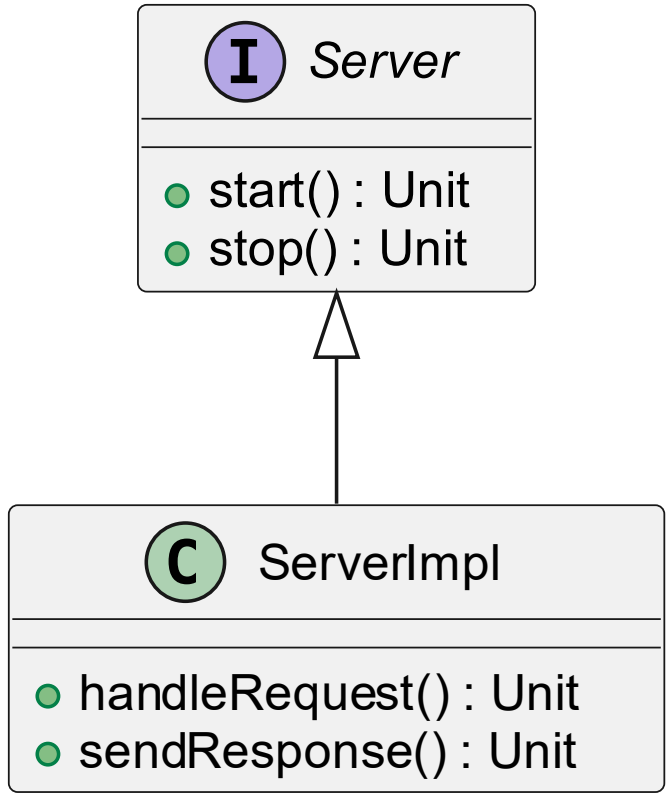
Advice 8

- Typealias are not KSClassDeclarations
- Extension functions can be applied on TypeAlias


Ok I‘ll try
The other team mentioned, you only provide an Interface. Can you show me how you implement the server internally ?

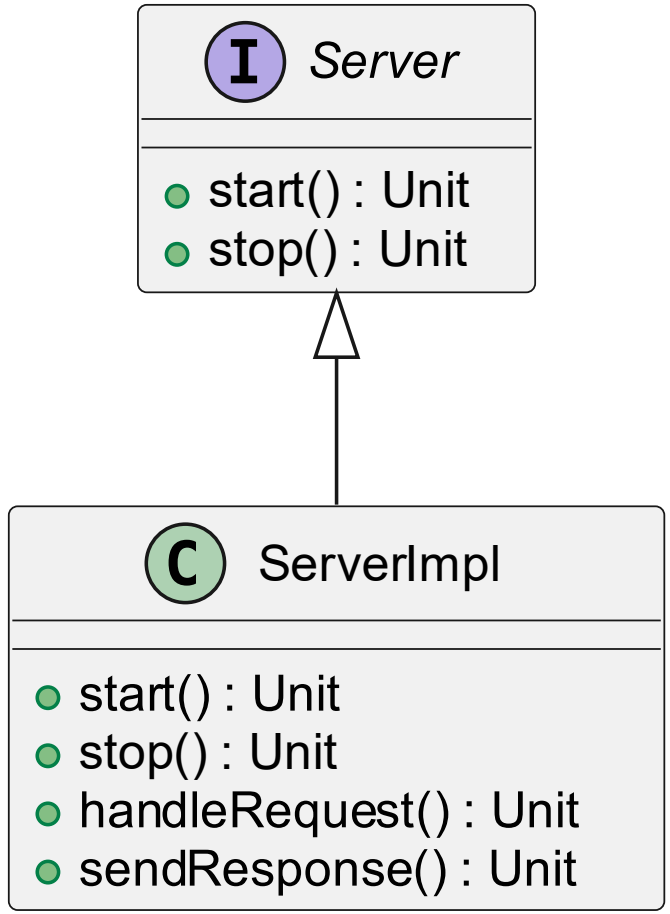
```kotlin
public interface Server {
    fun start(): Unit
    fun stop(): Unit
}

internal class ServerImpl : Server {
    override fun start(): Unit{}
    override fun stop(): Unit{}
    fun handleRequest(): Unit{}
    fun sendResponse(): Unit{}
}
```

Expected Generated Documentation



```kotlin
public interface Server {
    fun start(): Unit
    fun stop(): Unit
}

internal class ServerImpl : Server {
    override fun start(): Unit{}
    override fun stop(): Unit{}
    fun handleRequest(): Unit{}
    fun sendResponse(): Unit{}
}
```

Actual Generated Documentation



```kotlin
override fun visitClassDeclaration(
    classDeclaration: KSClassDeclaration,
    data: Unit
) {
    val functions = classDeclaration
        .getAllFunctions()
        .filterNot { it.isInhertied() }

    val properties = classDeclaration
        .getAllProperties()
        .filterNot { it.isInhertied() }
}
```

```kotlin
fun KSFunctionDeclaration.isInherited(): Boolean{
    if (this.modifiers.contains(Modifier.OVERRIDE)) {
        return true
    }
    // ... more checks
    return true
}

fun KSPropertyDeclaration.isInherited(): Boolean{
    if (this.modifiers.contains(Modifier.OVERRIDE)) {
        return true
    }
    // ... more checks
    return true
}
```

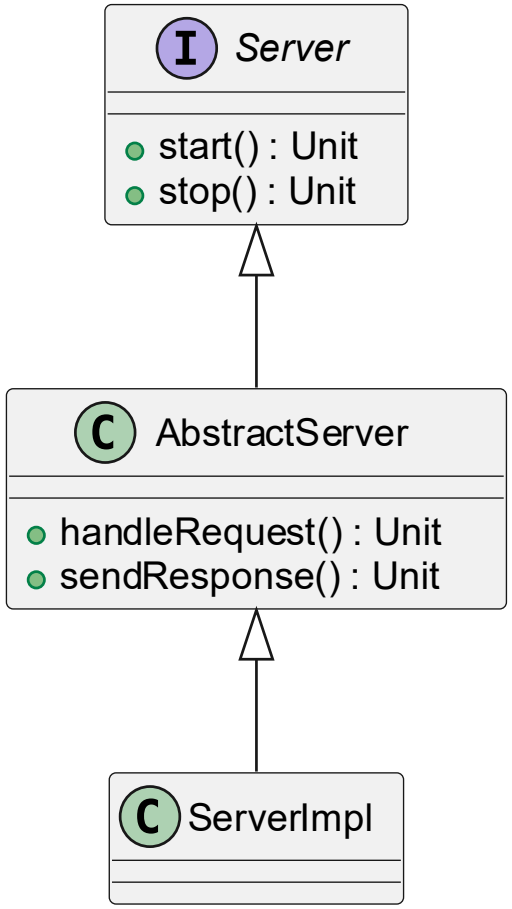
Advice 9

To find the functions that are provided by a parent interface, the modifier can be checked

If the parent is an abstract class, the override modifier might not be present

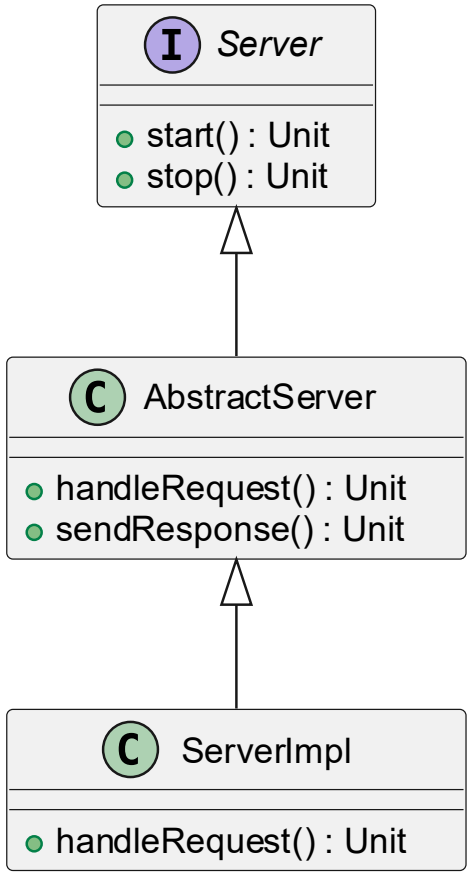
```kotlin
public interface Server {
    fun start(): Unit
    fun stop(): Unit
}

public abstract class AbstractServer() : Server {
    override fun start(): Unit {}
    override fun stop(): Unit {}
    fun handleRequest(): Unit {}
    fun sendResponse(): Unit {}
}

internal class ServerImpl : AbstractServer {
    override fun sendResponse(): Unit {}
}
```

Expected Generated Documentation



```kotlin
public interface Server {
    fun start(): Unit
    fun stop(): Unit
}

public abstract class AbstractServer() : Server {
    override fun start(): Unit {}
    override fun stop(): Unit {}
    fun handleRequest(): Unit {}
    fun sendResponse(): Unit {}
}

internal class ServerImpl : AbstractServer {
    override fun sendResponse(): Unit {}
}
```

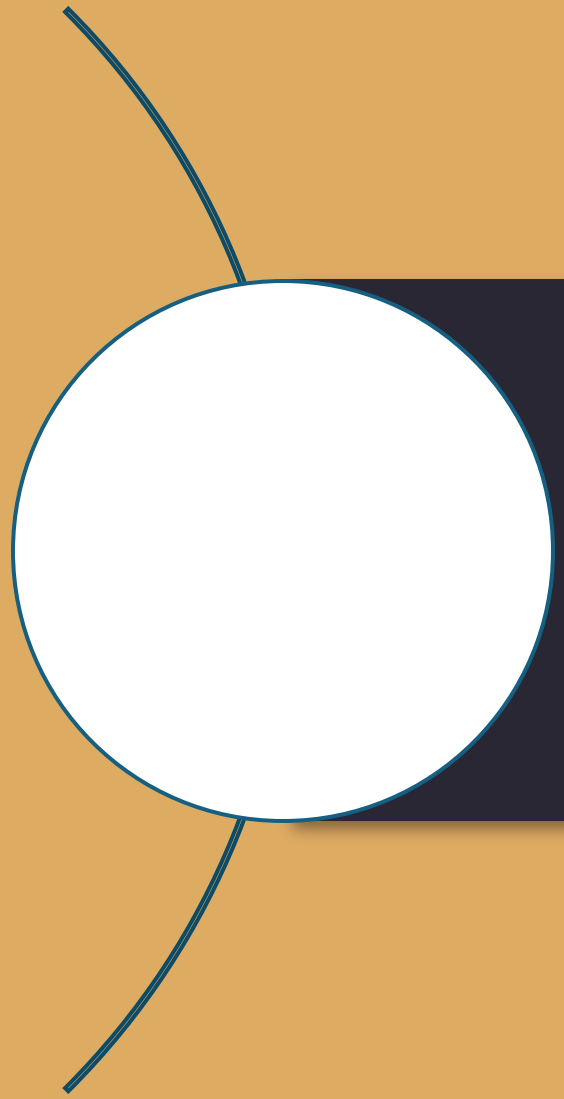
Actual Generated Documentation



```kotlin
// fullqualifiedName = Server
public interface Server {
    // fullqualifiedName = Server.start
    fun start(): Unit
    // fullqualifiedName = Server.stop
    fun stop(): Unit
}

// fullqualifiedName = AbstractServer
public abstract class AbstractServer() : Server {
    // fullqualifiedName = AbstractServer.start
    override fun start(): Unit {}
    // fullqualifiedName = AbstractServer.stop
    override fun stop(): Unit {}
    // fullqualifiedName = AbstractServer.handleRequest
    fun handleRequest(): Unit {}
    // fullqualifiedName = AbstractServer.sendResponse
    fun sendResponse(): Unit {}
}

// fullqualifiedName = ServerImpl
internal class ServerImpl : AbstractServer {
    // fullqualifiedName = ServerImpl.sendResponse
    override fun sendResponse(): Unit {}
    // fullqualifiedName = AbstractServer.start
    // fullqualifiedName = AbstractServer.stop
    // fullqualifiedName = AbstractServer.handleRequest
}
```

```kotlin
fun KSFunctionDeclaration.isInherited(): Boolean {
    if (this.modifiers.contains(Modifier.OVERRIDE)) {
        return true
    }
    val ownerOfThisFunction = this.parentDeclaration as? KSClassDeclaration
    val parents: List<KSClassDeclaration> = ownerOfThisFunction
        ?.superTypes
        ?.toList()
        ?.map { it.resolve().declaration }
        ?.filterIsInstance<KSClassDeclaration>() ?: emptyList()
    val parentFunctions: List<String> = parents
        .flatMap { it.getAllFunctions().toList() }
        .mapNotNull { it.qualifiedName?.asString() }
    if (this.qualifiedName?.asString() in parentFunctions) {
        return true
    }
    // ... more checks
    return true
}
```

Advice 10



To find the functions that are provided by a parent class, the full qualified name can be compared


...
Btw. The
other Team added an extension
function on your Server.
Depending on the Server state
the have a property
„isRunning“.


I thought so already.
Ahh one more thing:
We would love to see what Requests have been received and what Responses have been sent


Thank you.
But you only know the half of the story
Amazing