

KSP in Action

Kotlin Symbol Processor

```
1 class MyProcessor : SymbolProcessor {
2     override fun process(resolver: Resolver): List<KSAnnotated> {
3         val files = resolver.getAllFiles()
4         files.forEach {
5             // Here each class can be interpreted
6             val myVisitor = KSVisitorVoid()
7             it.accept(myVisitor, Unit)
8         }
9         return files.toList()
10    }
11 }
12
```

KSVisitor

Complete

```
1
2 /**
3  * A visitor for program elements
4  */
5 interface KSVisitor<D, R> {
6     fun visitNode(node: KSNode, data: D): R
7
8     fun visitAnnotated(annotated: KSAnnotated, data: D): R
9
10    fun visitAnnotation(annotation: KSAnnotation, data: D): R
11
12    fun visitModifierListOwner(modifierListOwner: KSModifierListOwner, data: D): R
13
14    fun visitDeclaration(declaration: KSDeclaration, data: D): R
15
16    fun visitDeclarationContainer(declarationContainer: KSDeclarationContainer, data: D): R
17
18    fun visitDynamicReference(reference: KSDynamicReference, data: D): R
19
20    fun visitFile(file: KSFile, data: D): R
21
22    fun visitFunctionDeclaration(function: KSFunctionDeclaration, data: D): R
23
24    fun visitCallableReference(reference: KSCallableReference, data: D): R
25
26    fun visitParenthesizedReference(reference: KSParenthesizedReference, data: D): R
27
28    fun visitPropertyDeclaration(property: KSPropertyDeclaration, data: D): R
29
30    fun visitPropertyAccessor(accessor: KSPropertyAccessor, data: D): R
31
32    fun visitPropertyGetter(getter: KSPropertyGetter, data: D): R
33
34    fun visitPropertySetter(setter: KSPropertySetter, data: D): R
35
36    fun visitReferenceElement(element: KSReferenceElement, data: D): R
37
38    fun visitTypeAlias(typeAlias: KSTypeAlias, data: D): R
39
40    fun visitTypeArgument(typeArgument: KSTypeArgument, data: D): R
41
42    fun visitClassDeclaration(classDeclaration: KSClassDeclaration, data: D): R
43
44    fun visitTypeParameter(typeParameter: KSTypeParameter, data: D): R
45
46    fun visitTypeReference(typeReference: KSTypeReference, data: D): R
47
48    fun visitValueParameter(valueParameter: KSValueParameter, data: D): R
49
50    fun visitValueArgument(valueArgument: KSValueArgument, data: D): R
51
52    fun visitClassifierReference(reference: KSClassifierReference, data: D): R
53 }
```

KSVisitor *Focused*

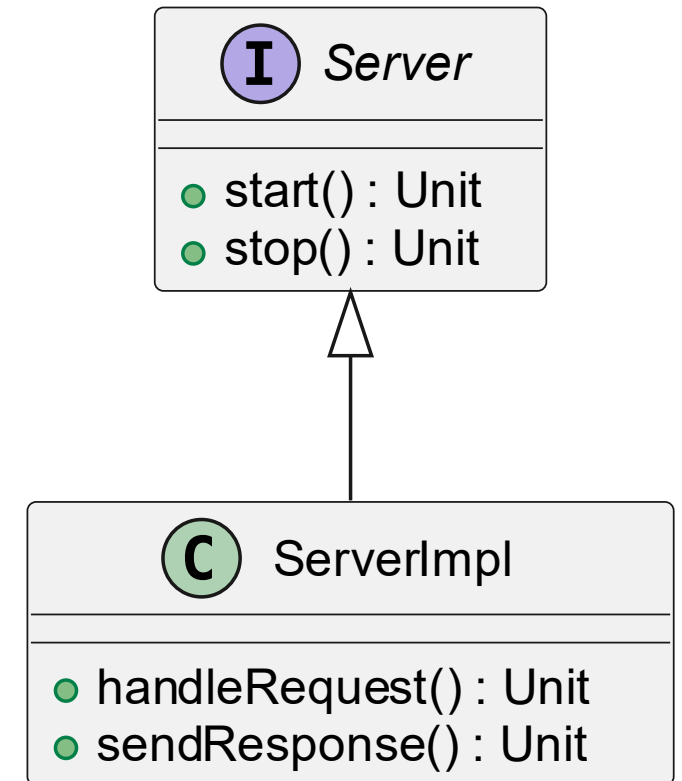
```
1
2 /**
3  * A visitor for program elements
4  */
5 interface KSVisitor<D, R> {
6     fun visitFile(file: KSFile, data: D): R
7
8     fun visitFunctionDeclaration(function: KSFunctionDeclaration, data: D): R
9
10    fun visitPropertyDeclaration(property: KSPropertyDeclaration, data: D): R
11
12    fun visitTypeAlias(typeAlias: KSTypeAlias, data: D): R
13
14    fun visitClassDeclaration(classDeclaration: KSClassDeclaration, data: D): R
15
16 }
```

Ok.
We will do that
internally and you
can just start/stop
it.

We need a Server that can
handle Requests and send
Responses.



```
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



Generated Documentation

```
public interface Server {  
    fun start(): Unit  
    fun stop(): Unit  
  
    interface Config {  
        val ipAddress: String  
        val port: Int  
    }  
}
```

I *Server*

start(): Unit
stop(): Unit

Expected Generated Documentation

```
public interface Server {  
    fun start(): Unit  
    fun stop(): Unit  
  
    interface Config {  
        val ipAddress: String  
        val port: Int  
    }  
}
```

I *Server*

start(): Unit
stop(): Unit


I *Server.Config*

ipAddress: String
port: Int

Why is the inner class missing ?

```
interface KSVisor<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 : KNode {
    /**
     * Declarations that are lexically
     * declared inside the current container.
     */
    val declarations: Sequence<KSDeclaration>
}
```

```
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)
    }

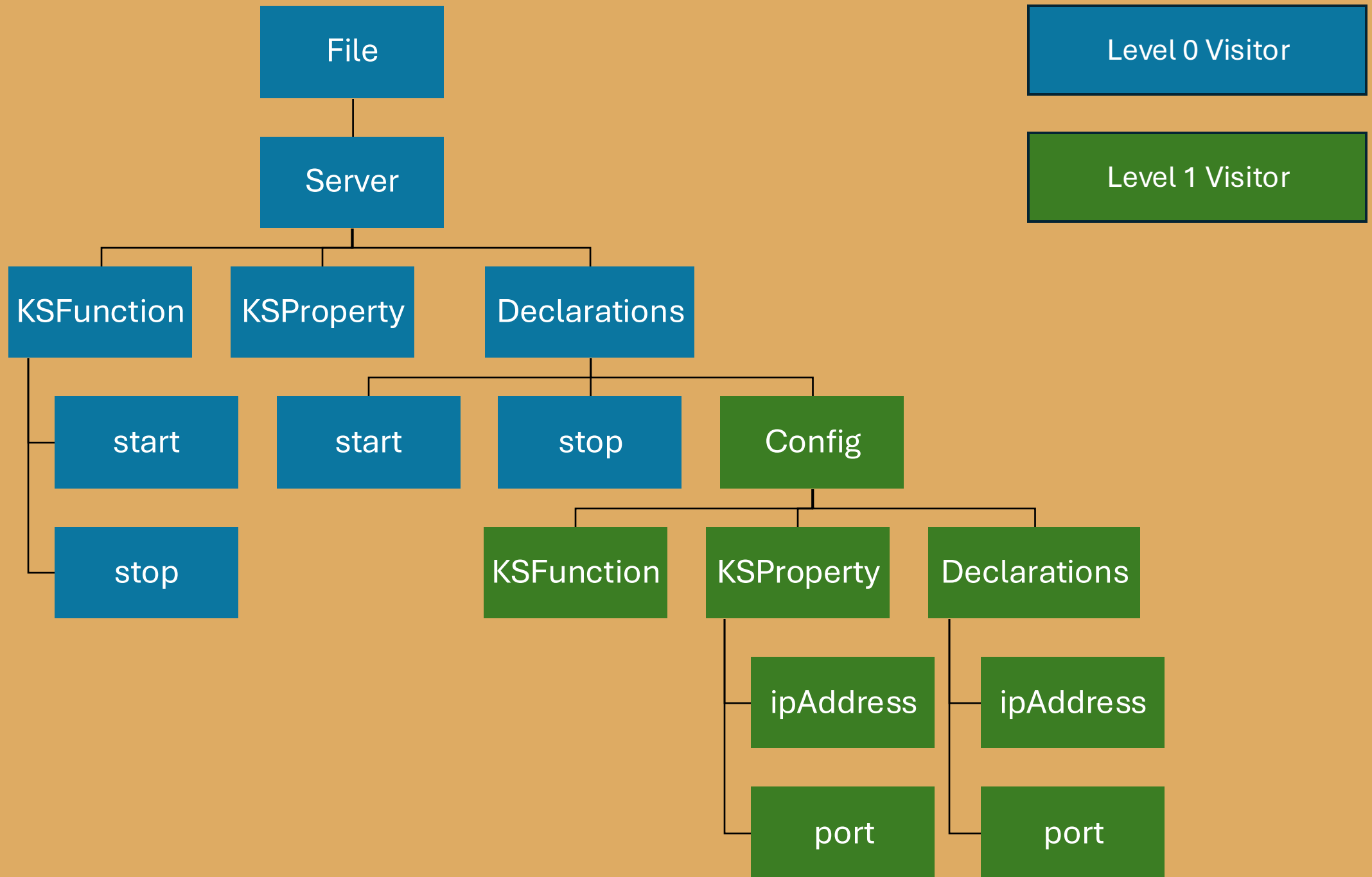
    // This is what I missed first
    classDeclaration.declarations
        .filterNot { it in classDeclaration.getDeclaredProperties() }
        .filterNot { it in classDeclaration.getDeclaredFunctions() }
        .filterIsInstance<KSClassDeclaration>()
        .forEach { declaration ->
            declaration.accept(this,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

```

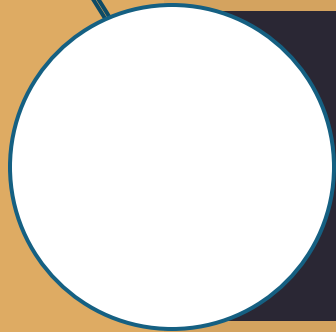
```

private fun analyseFunction(function: KSFunctionDeclaration) {}

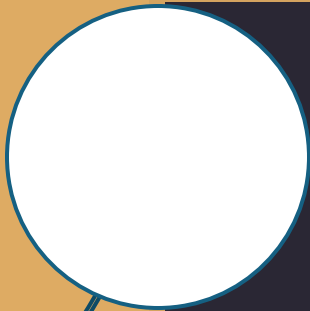
```

Tipp 1

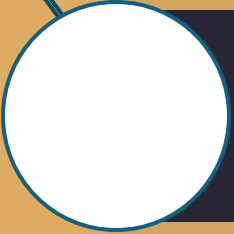


KSVisitor just visits the next level.

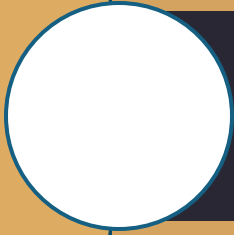


Pass it through to catch all levels

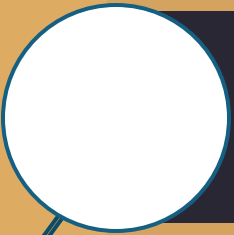
Tipp 2



Understand the KSClassContainer component

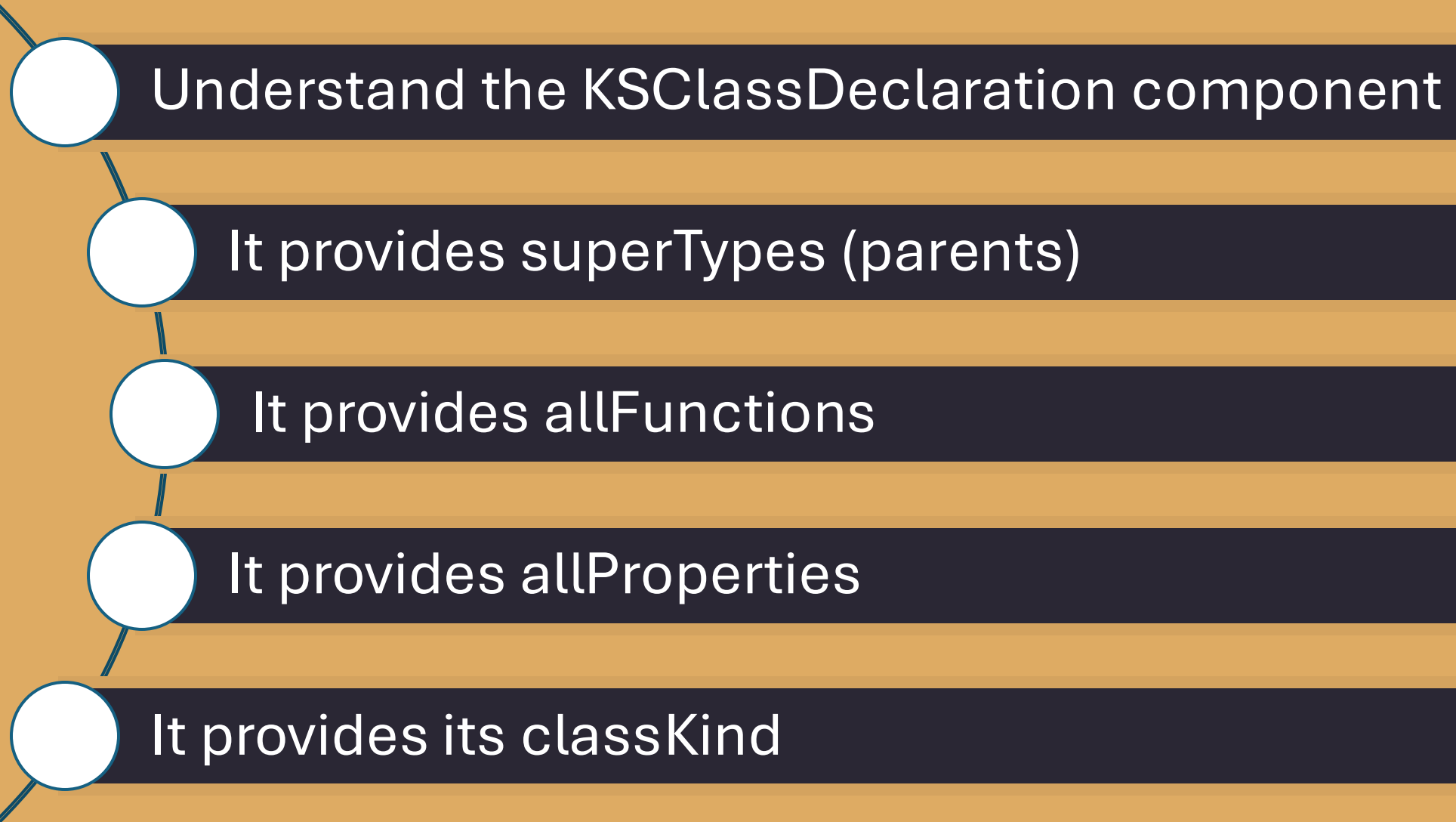


Inner *Classes*, *Interfaces*, *companion objects* can be retrieved by the *declarations* property



declarations contains every KSDeclaration, not only the ones for inner classes etc.

Tipp 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 ?





```
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  
            }  
        }  
    }  
}
```



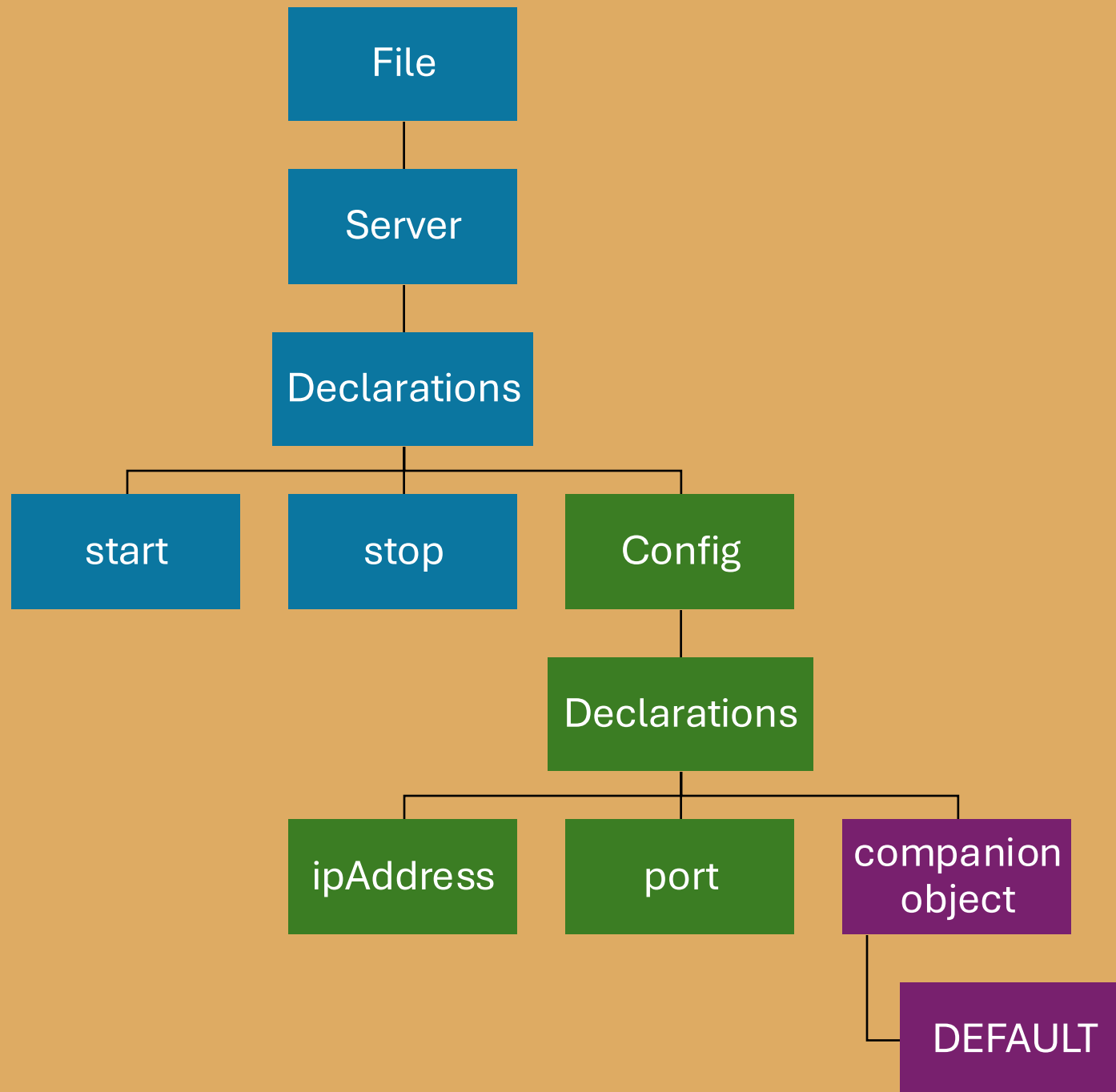
Server

- start(): Unit
- stop(): Unit



Server.Config

- Default : Config
- ipAddress : String
- port : Int



Level 0 Visitor

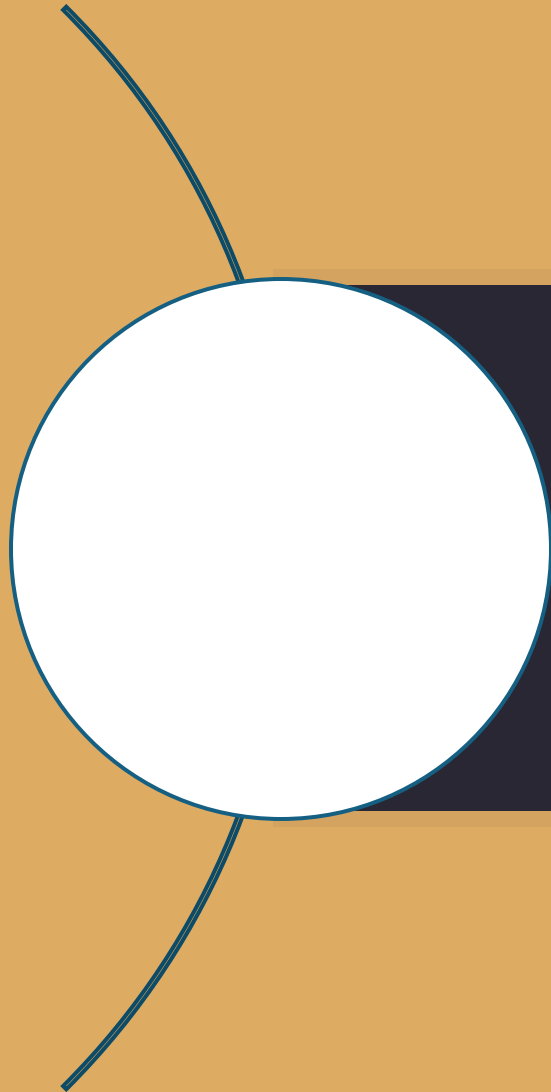
Level 1 Visitor

Level 2 Visitor



```
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  
}
```

```
/**  
 * 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")  
}
```


Tipp 4



Know the
KSClassKind enum
states

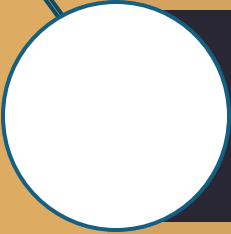


```
fun getCompanionObjectDeclaration(  
    clazz: KSClassDeclaration  
): KSClassDeclaration? {  
    return clazz  
        .declarations  
        .filterIsInstance<KSClassDeclaration>()  
        .filter { it.classKind == ClassKind.OBJECT }  
        .filter { it.isCompanionObject }  
        .firstOrNull()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

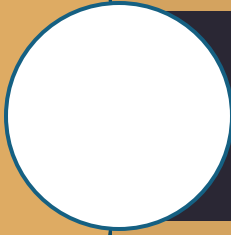


```
fun getCompanionObjectProperties(clazz: KSClassDeclaration) : List<KSPropertyDeclaration>{  
    return getCompanionObjectDeclaration(clazz)?.getAllProperties()?.toList() ?: emptyList()  
}  
  
fun getCompanionObjectFunctions(clazz: KSClassDeclaration) : List<KSFunctionDeclaration>{  
    return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

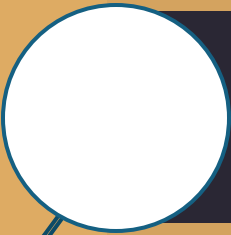
Tipp 5



Every class can have max. 1 companion object



All functions and properties of this a *companion object* can be interpreted 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 ?**

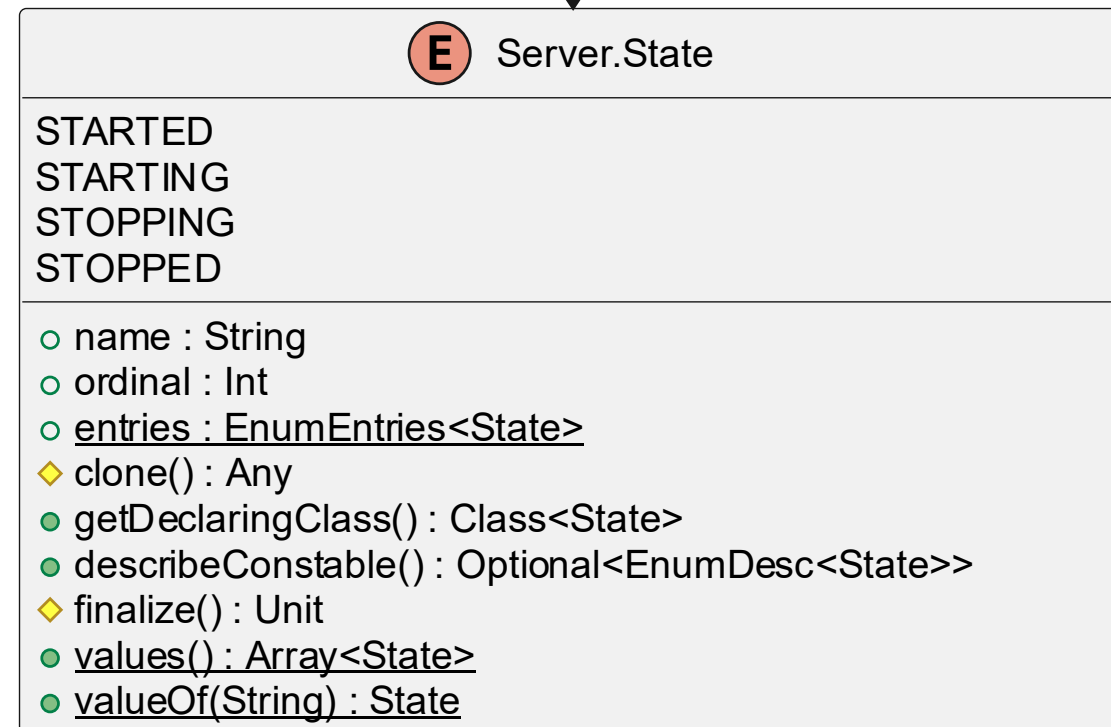
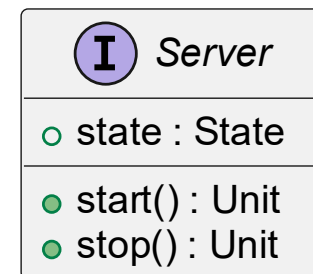



```

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

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

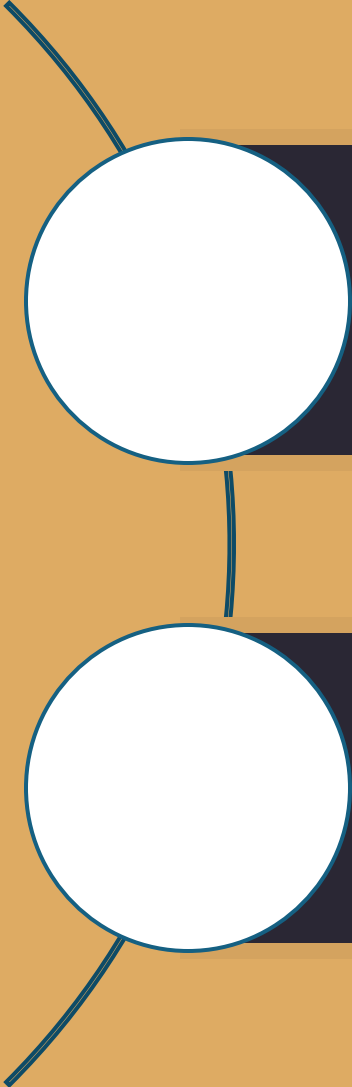
```





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


Tipp 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 ?

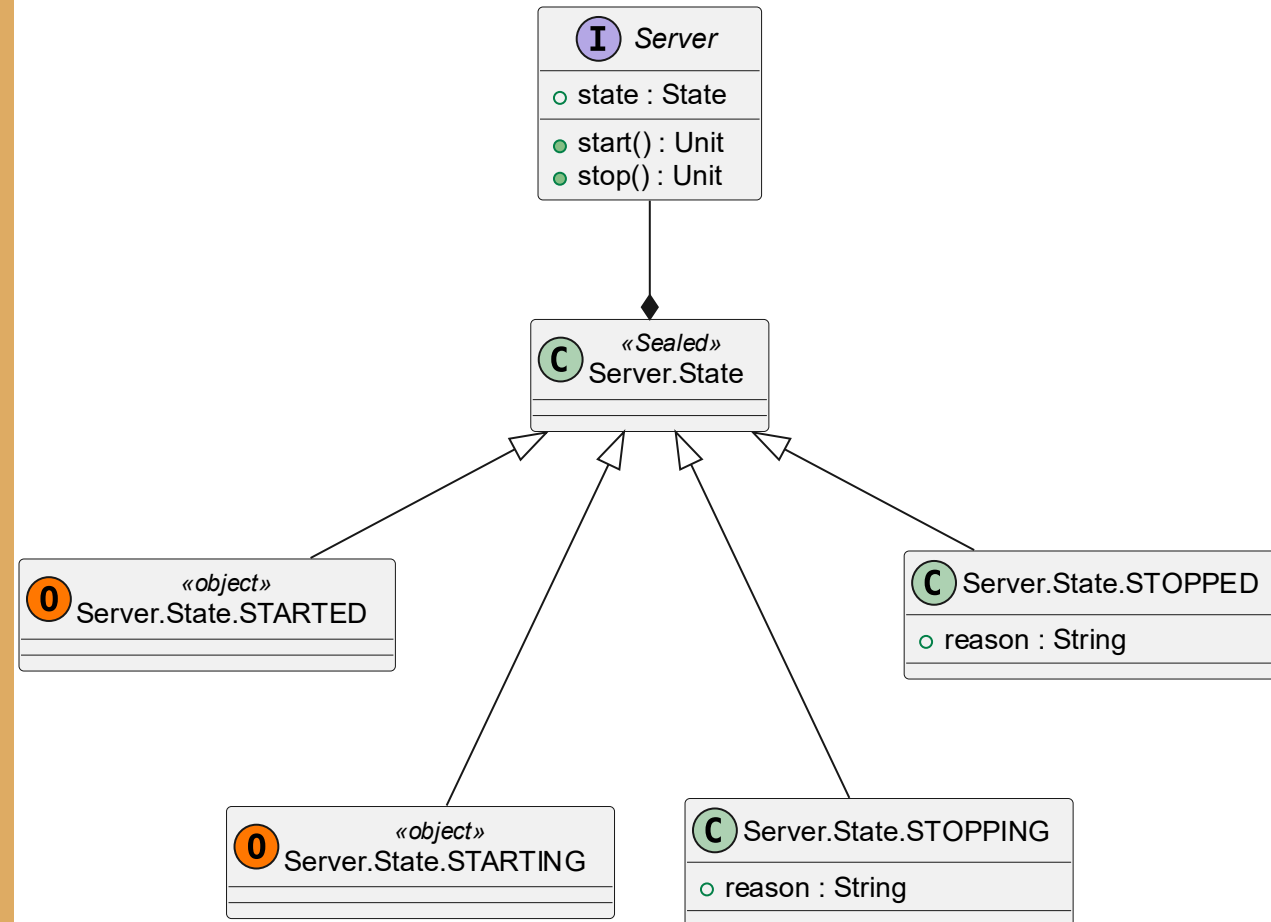


```

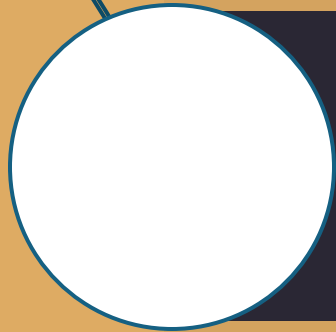
public interface Server {
    fun start(): Unit
    fun stop(): Unit
    val state: State

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

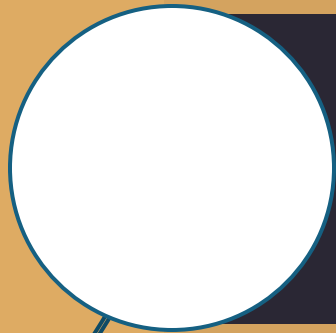
```



Tipp 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.**



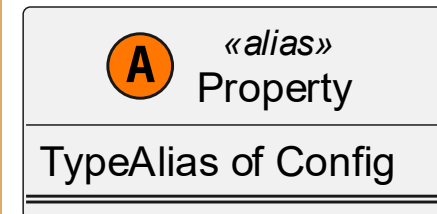
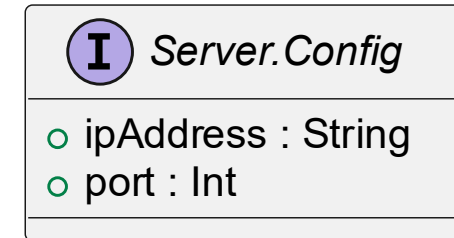
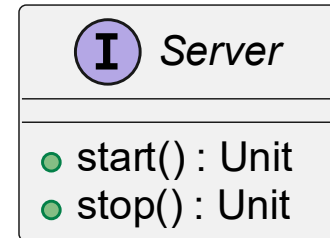
```

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

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

typealias Property = Server.Config

```





```
interface KTypeAlias : KSDeclaration {  
    val name: KSName  
  
    val type: KTypeReference  
}
```


Tipp 8



Typealiases are not
KSClassDeclarations

Extension functions can be
applied on TypeAlias

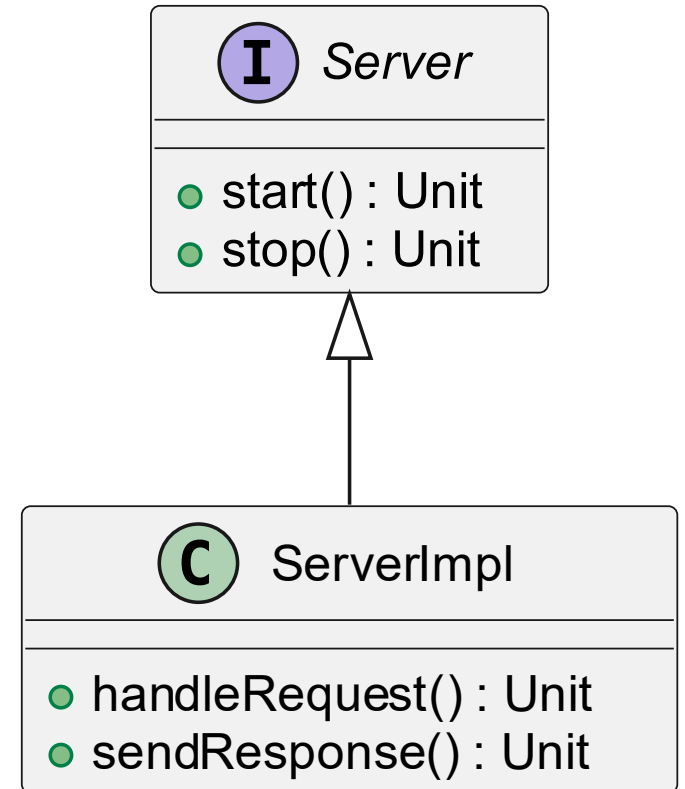
Ok I'll try

**The other
team mentioned, they only
have get the Interface. Can
you show me how you
implement the server
internally ?**



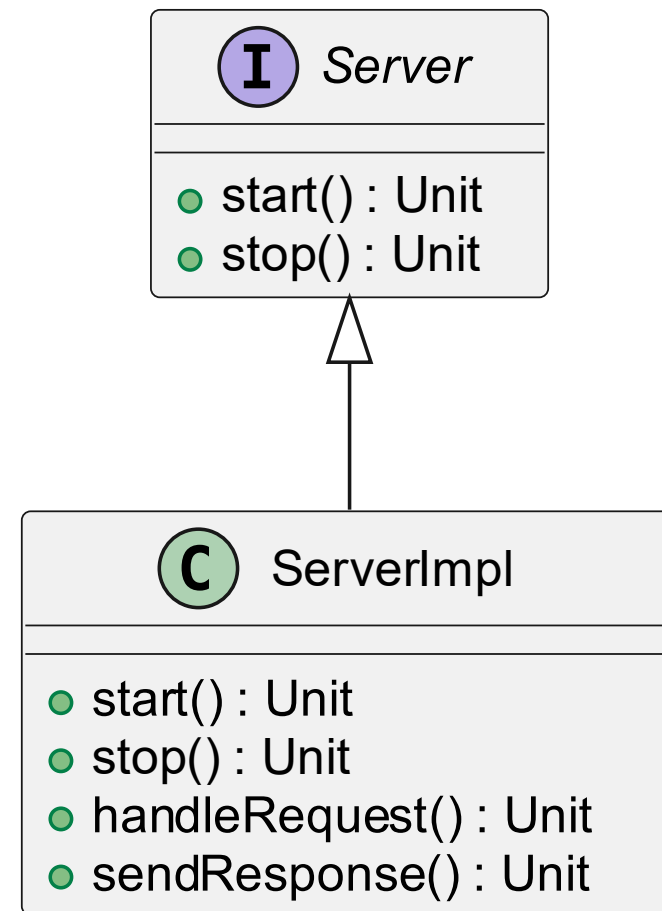
Expected Generated Documentation

```
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

```
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{}  
}
```



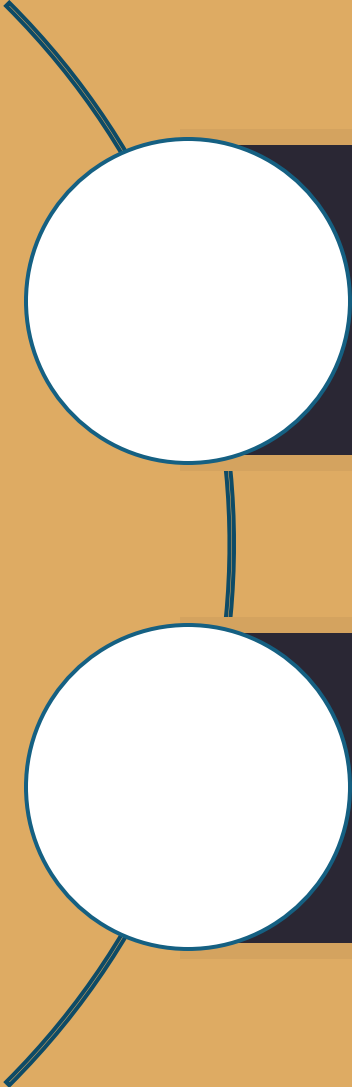


```
override fun visitClassDeclaration(  
    classDeclaration: KSClassDeclaration,  
    data: Unit  
) {  
    val functions = classDeclaration  
        .getAllFunctions()  
        .filterNot { it.isInherited() }  
  
    val properties = classDeclaration  
        .getAllProperties()  
        .filterNot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



```
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  
}
```

Tipp 9

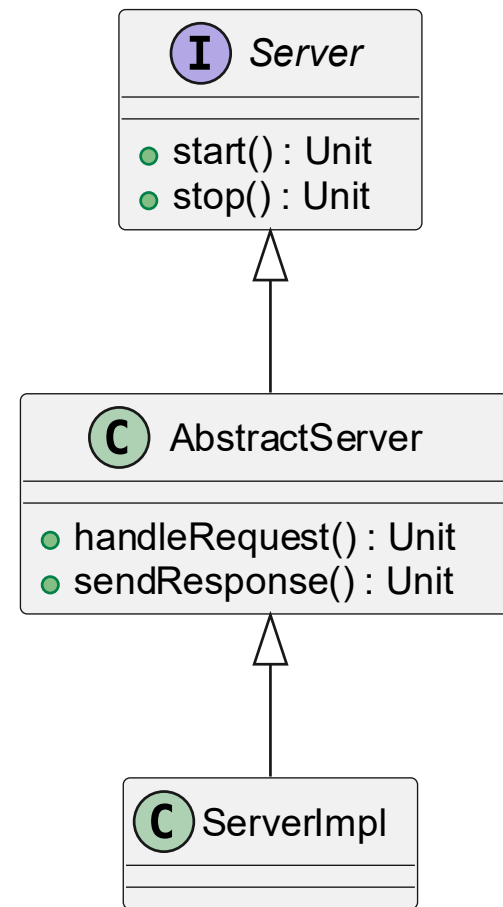


To find the functions that are provided by a parent interface, the modifier can be checked

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

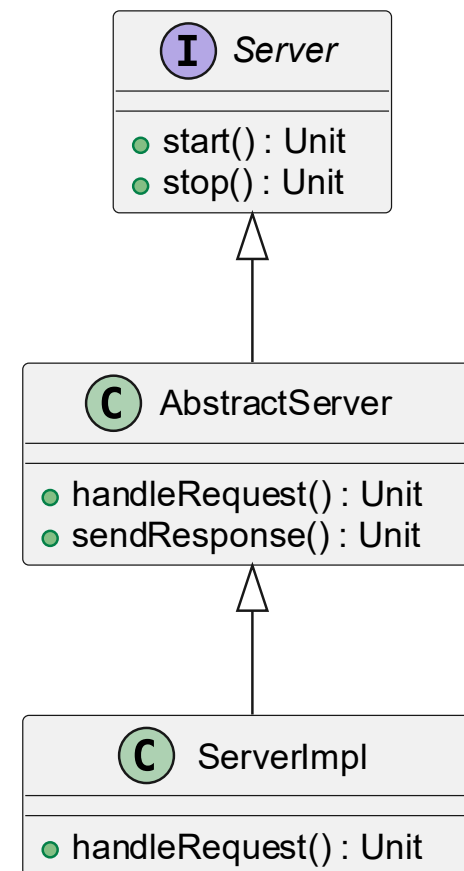
```
public interface Server {  
    fun start(): Unit  
    fun stop(): Unit  
}  
  
public abstract class AbstractServer() : Server {  
    override fun start(): Unit {}  
    override fun stop(): Unit {}  
    fun handleRequest(): Unit {}  
    fun sendResponse(): Unit {}  
}  
  
internal class ServerImpl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

Expected Generated Documentation



```
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




```

// 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
}

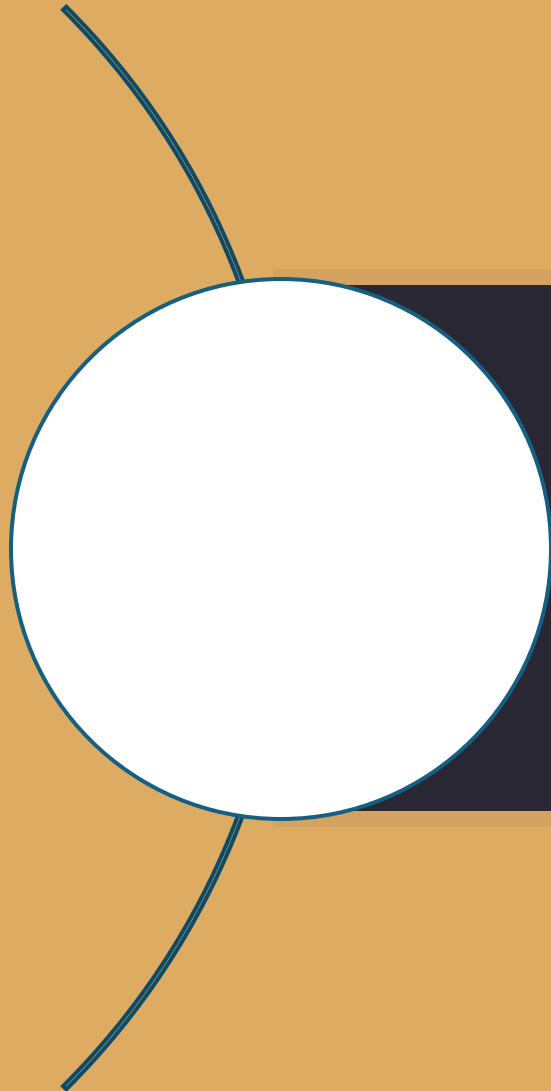
```

```

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
}

```

Tipp 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
they 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 know only
the have of the
story**

Amazing

