INTERNATIONAL SCIENTIFIC JOURNAL OF ENGINEERING AND MANAGEMENT ISSN: 2583-6129
VOLUME: 03 ISSUE: 08 | AUG – 2024 DOI: 10.55041/ISJEM02103
AN INTERNATIONAL SCHOLARLY || MULTIDISCIPLINARY || OPEN ACCESS || INDEXING IN ALL MAJOR DATABASE & METADATA

Enhancing Class Functionality with Kotlin Extensions

Nilesh Jagnik
Los Angeles, USA
nileshjagnik@gmail.com

***Abstract*—Kotlin has many features that make code simpler to develop and read. One such feature is the support for extension functions. Extension functions allow extending the capability of a class or interface, without the need for coding pattern like the decorator pattern and utilities, which add a lot of complexity to code. In this paper we study the decorator pattern which is the primary tool used for class extension. We discuss the disadvantages of using the decorator pattern and how extensions can be used to overcome them. We review how extensions work and how they can used. Finally, we present some benefits and drawbacks of extension functions.**

Keywords—decorator pattern, class extension, code readability, code reuse

I. INTRODUCTION

Kotlin has many features that make development and maintenance of code simple and easy. This includes the support for lightweight coroutines, structured concurrency, Flows for streaming and Channels for inter-coroutine data transfer. Extensions are another feature in Kotlin that allow extending the capabilities of a class or interface in a simple manner.

Often in software development, it is required to alter or add functionality to preexisting classes and interfaces. This is done to support evolving requirements from applications or to improve or optimize runtime behavior, debuggability, error handling, etc.

In Java and other languages lacking extensions, when it is required to extend the functionality of a class, developers have to use the decorator pattern and composition to create wrappers around the base class. Over time, this can lead to increased code complexity and other issues associated with complexity.

```
interface Person {
  fun name(): String
}

class Male
    (private val name: String) : Person {
  override fun name() = name
}

abstract class PersonDecorator
    (private val person: Person) : Person {
  override fun name() = person.name()
}

class TitleMrDecorator
    (person: Person) : PersonDecorator(person) {
  override fun name() = "Mr. " + super.name()
}
```

Extensions in Kotlin provide the capability of extend the functionality of classes and interfaces in a simple way. However, the power of extensions come with some drawbacks. There certain scenarios they are not suited for. In this paper we take a look at Kotlin extensions, their uses, benefits and caveats to be aware of when using them.

II. DECORATOR PATTERN

The decorator pattern is a design pattern that allows adding functionality to classes and interfaces without altering their definition. This allows adding functionality to an object at runtime without affecting the other instances of its' class.

The decorator pattern usually involves creating wrapper classes following the same interface as the class being wrapped. This allows re-implementing the functionality of member functions that need to be modified. This wrapper can use composition to re-use logic from the base class wherever required. Fig. 1 shows an example of the decorator pattern. In this

example, an object of the Person interface has been decorated to add a title before the name.

```
fun Male.nameWithTitleMr() : String {
  return "Mr. " + this.name()
}

fun main() {
  val john = Male("John")

  // The decorator and extension print the same string.
  print(TitleMrDecorator(john).name())
  print(john.nameWithMrTitle())
}
```

Fig. 1. Using decorator pattern to modify object behavior.

III. PROBLEMS WITH THE DECORATOR PATTERN

A. *Complexity*

The overuse of decorators can lead to increased code complexity. This is because decorators add many layers of wrappers on the original class. This makes code quite hard to read and extend.

B. *Debugging*

A high number of nested decorators can create deep call stacks and subsequently make it harder to debug issues using stack traces.

C. *Decoration Order*

The order in which decorators are applied to an object can determine the end result. This can cause confusion and errors if the order is not properly documented and management.

```
fun Person?.nameWithTitleMr() : String {
  if (this == null) return "Null Person"
  return "Mr. " + this.name()
}
```

D. *Removal*

It can be quite difficult to remove a decorator if the code uses many decorators in a chain. Removal can cause unexpected behavior due to complex code structure.

E. *Performance*

Although small, each decorator adds a small performance overhead since it new objects are created after decoration. This could add up if several decorators are used.

IV. KOTLIN EXTENSION FUNCTIONS

Extension functions in Kotlin allow extending the functionality of a class or interface without the need for the decorator pattern. It is possible to write new functions for classes that are obtained from external libraries/code.

A. *Declaration*

Kotlin provides in-built support for extensions. To declare an extension, the type of the class being extended is prepended to the name of the function. Fig. 2 shows an example of an extension function. We use extensions to achieve the same result as we did with decorators in Fig. 1.

Fig. 2. Using extensions in Kotlin.

The “this” keyword can be used inside the function to refer to the object at runtime on which the extension function is invoked. Extension functions can also be declared for generic classes.

B. *Static Resolution*

One thing to note about extensions do not actually alter the definition of the class. The simply static methods that are callable with the dot notation on a class instance. So, if a class has a subclass and both have a similar extension, the compile time object type will determine which extension function will be called.

C. *Nullable Object Extension*

It is possible to define an extension with the receiver object being nullable. In this case, the extension function should perform null checks. Fig. 3 shows an example of an extension with a nullable receiver.

Fig. 3. Extension function with nullable receiver type.

D. *Extension Properties*

Similar to extension function, extension properties can also be defined. These properties do not have a backing field however, and thus cannot be used in initializer blocks. Developers have to create setter and getters for these properties.

E. Companion Objects

Extensions can also be defined for the companion object of a class. These can be called using only the class name much like other companion object members.

F. Extension Scope

Extensions should be defined directly under packages at the top of a file. To use extension outside of the package where it is declared, simply import where it needs to be used.

G. Visibility

An important thing to note is that visibility for extensions work similar to visibility for normal functions. E.g., an extension defined outside the class definition can not access the private members of the class.

V. BENEFITS OF KOTLIN EXTENSION FUNCTIONS

A. Improved Readability and Conciseness

Extension functions reduce the boilerplate of decorators and utility classes needed for extending functionality of objects. The syntax of extensions is also more natural and concise, leading to improved readability.

B. Debuggability

As opposed to decorators, extensions do not create deep call stacks. This leads to easier debugging.

C. Code Reuse

Extensions once built can be re-used in multiple packages across the code base by importing them. This increases code reuse and reduces duplication.

D. Extending External Libraries

Extensions can be used to enhance functionality of objects created by external libraries not owned by your project.

E. Performance

Since extension functions are statically resolved, they are more performant in comparison to decorators.

VI. CAVEATS OF KOTLIN EXTENSION FUNCTIONS

A. Name Conflicts

It is possible to have conflicts between member functions and extensions for a class being extended. E.g., if an extension was created for a class, but then later a member function with the same signature was defined. In this case, the member function is executed and not the extension. Change in base class functions could lead to confusion and unexpected behavior.

B. Static Dispatching

Because extension functions are statically resolved, depending on the variable declarations in code, it is possible that the extension function of the base class will be invoked and not the subclass (even though the object is of the subclass type).

C. Private Member Visbility

If extension functions are declared outside the class file, they will not have access to private members. They may also not have access to internal members if they are defined outside the class module.

CONCLUSION

Kotlin extension functions are a powerful tool that can make it quite simple to extend functionality of classes and interfaces, even those that are created by external libraries. They offer a simpler alternative to decorators and utilities which are often used for adding and modifying object functionality. They require minimal boilerplate and fulfill quite a lot of requirements related to class extension. However, they do have several nuances to be aware of. Improper or excessive use of extensions can lead to unexplained behavior and runtime issues. So, it is best to use extension functions with care and in moderation.

REFERENCES

[1] Rahul Chowdhury, "The Ugly Truth about Extension Functions in Kotlin (Jun 2017)," https://medium.com/android-news/the-ugly-truth-about-extension-functions-in-kotlin-486ec49824f4

[2] "Extensions (Jul 2021)," https://kotlinlang.org/docs/extensions.html

[3] Diego Torres, "The Decorator Pattern in Kotlin (Mar 2024)," https://www.baeldung.com/kotlin/decorator-pattern

[4] Sena Zincircioğlu, "Extension Functions in Kotlin (Dec 2022)," https://medium.com/@senazincircioglu/extension-functions-of-kotlin-c66862737868

[5] Sahil Thakar, "Kotlin Extension functions under the hood. (Apr 2024)," https://medium.com/@sahilthakar10/extension-function-under-the-hood-a081ddf02567

[6] Nidhi Sorathiya, "Kotlin Extension Function: Enhancing Classes, Streamlining Code (May 2024)," https://www.dhiwise.com/post/kotlin-extension-function-enhancing-classes-streamlining-code

[7] Graham Cox, "Extension Functions in Kotlin (Apr 2024)," https://www.baeldung.com/kotlin/extension-methods