Delegation

#KotlinGems

Kotlin

We use delegation whenever we want to add new functionality to our class.
Still, we don’t want to add more responsibilities to it, Since this will cause our class code to grow over time.

So one of the patterns to achieve this is the decorator pattern.

let's take an example,
Suppose we need to add logging functionality to the **MutableMap** class in our app whenever we **put** a pair or **get** the value by key.

```
fun main() {

    val testMap = mutableMapOf()<String, Int>()
    testMap["One"] = 1
    val four = testMap["Four"]
    testMap["Two"] = 2
    val one = testMap["One"]
    testMap["Three"] = 3

    println("Size:${testMap.size}")

}
```

One possibility to achieve this is extending the **MutableMap** and overriding the **put** and **get** functions with logging logic. Unfortunately, this also will require us to override all other functions, although we will not add new behavior to any of them.

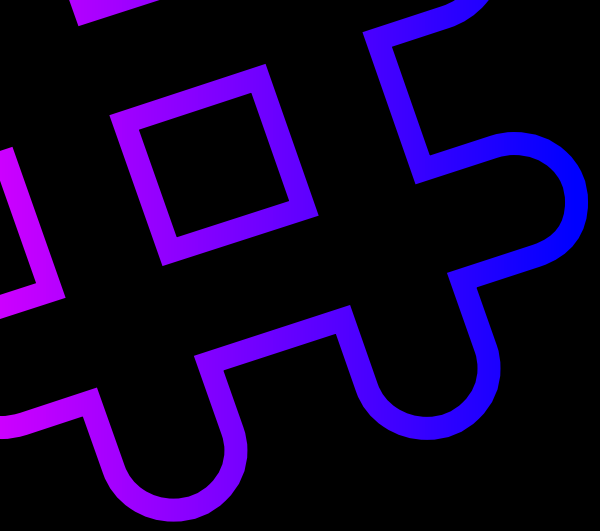
```
//Compile Error 💥💥: all MutableMap interface functions need to be
overridden
class MapWithLogging<K, V>(
    override val size: Int,
    override val entries: MutableSet<MutableMap.MutableEntry<K, V>>,
    override val keys: MutableSet<K>,
    override val values: MutableCollection<V>
) : MutableMap<K, V> {

}
```

A more straightforward solution is using The **Decorator pattern**; the whole idea is to create a new class stating the new functionality and delegating the unchanged behaviors to an internal object, sent through the constructor.

```kotlin
/**
 * Decorator pattern
 * Notice: [MapWithLogging] and [helperMap] have same type of the target class
 */
class MapWithLogging<K, V>(private val helperMap: MutableMap<K, V>) : MutableMap<K, V> {

    override val entries: MutableSet<MutableMap.MutableEntry<K, V>>
        //we don't need to change anything here, let's delegate it to [helperMap]
        get() = helperMap.entries

    override val keys: MutableSet<K>
        //Also here
        get() = helperMap.keys

    override val size: Int
        //You got the idea
        get() = helperMap.size


    //We add our logging logic here
    override fun put(key: K, value: V): V? {
        println("Putting Pair $key to $value")
        return helperMap.put(key, value)
    }

    //We add our logging logic here
    override fun get(key: K): V? {
        val value = helperMap.get(key)
        println("Putting Pair $key to $value")
        return value
    }

    //Other functions: We follow the same pattern
    .....
    ...
    ..

}
```

But applying this pattern would require us to write boilerplate code every time; Luckily, with **Kotlin**, we could use the **by keyword** to Delegate unchanged behavior to the internal object directly. And boilerplate code will be generated for us by the compiler.

```kotlin
/**
 * Delegation in Kotlin
 * Notice: Using the by keyword to delegate all overridden functions to helperMap
 */
class MapWithLogging<K, V>(private val helperMap: MutableMap<K, V>) : MutableMap<K, V> by helperMap {

    //We add our logging logic here
    override fun put(key: K, value: V): V? {
        println("Putting Pair $key to $value")
        return helperMap.put(key, value)
    }

    //We add our logging logic here
    override fun get(key: K): V? {
        val value = helperMap.get(key)
        println("Getting Pair $key to $value")
        return value
    }
}
```



follow

#KotlinGems