Kotlin: Rules for writing classes

Rules for Writing classes

You have inferred your desire for stability. Mostly this means you should aim for immutability as gaining stability through notifying composition requires a lot of work, such as was done by the creation of the MutableState<T> class.

1. Do not use *var* as properties inside state holding classes

As these are mutable, but do not notify composition, they will make the composables which use them unstable.

Do:

```
data class InherentlyStableClass(val text: String)
```

Don’t:

```
data class InherentlyUnstableClass(var text: String)
```

2. Private properties still affect stability

As of the time of writing, it is uncertain if this is a design choice or a bug, but let’s slightly modify our stable class from above.

```
data class InherentlyStableClass(
    val publicStableProperty: String,
    private var privateUnstableProperty: String
)
```

The compiler report will mark this class as unstable:

```
unstable class InherentlyStableClass {
  stable val publicStableProperty: String
  stable var privateUnstableProperty: String
  <runtime stability> = Unstable
}
```

Looking at the results, it’s fairly obvious that the compiler struggles here. It marks both individual properties as stable, even though one is not, but marks the whole class as unstable.

```kotlin
data class InherentlyStableClass(
    val publicStableProperty: String,
    private var privateUnstableProperty: String
)
```

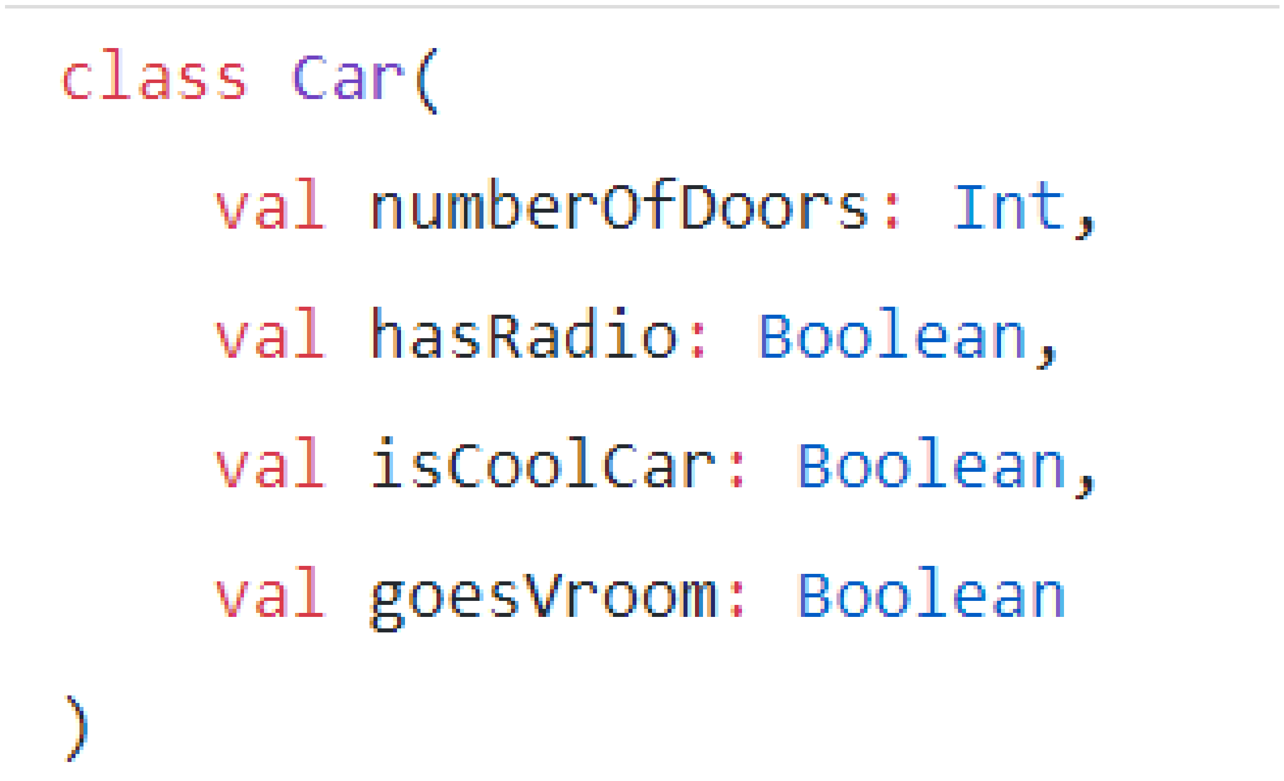
```
unstable class InherentlyStableClass {
  stable val publicStableProperty: String
  stable var privateUnstableProperty: String
  <runtime stability> = Unstable
}
```

```kotlin
class Car(
    val numberOfDoors: Int,
    val hasRadio: Boolean,
    val isCoolCar: Boolean,
    val goesVroom: Boolean
)
```

3. Do not use classes that belong to an external module to form state

Sadly, Compose can only infer stability for classes, interfaces, and objects that originate from a module compiled by the Compose Compiler. This means that any externally originated class will be marked as unstable, regardless of its true stability.

Let’s say you have the following class which comes from an external module and is therefore unstable:



```kotlin
class Car(
    val numberOfDoors: Int,
    val hasRadio: Boolean,
    val isCoolCar: Boolean,
    val goesVroom: Boolean
)
```

A common way to build a state using it would be to do the following:

```
data class RegisteredCarState(
    val registration: String,
    val car: Car
)
```

However, this is troublesome. Now you've made our state class unstable and therefore unskippable. This could potentially cause performance issues.

Luckily there are multiple ways to get around this.

If you only need a few properties of *Car* to form *RegisteredCarState*, you may simply flatten it as follows:

```
data class RegisteredCarState(
    val registration: String,
    var numberOfDoors: Int,
    var hasRadio: Boolean
)
```

```kotlin
data class RegisteredCarState(
    val registration: String,
    val car: Car
)
```

```kotlin
data class RegisteredCarState(
    val registration: String,
    var numberOfDoors: Int,
    var hasRadio: Boolean
)
```

```kotlin
class CarState(
    val numberOfDoors: Int,
    val hasRadio: Boolean,
    val isCoolCar: Boolean,
    val goesVroom: Boolean
)
```

```kotlin
@Immutable
data class WrappedList(
    val list: List<String> = listOf()
)
```

However, this may not be appropriate in cases where you need the whole object with all of its properties.

In such cases, you may create a local stable counterpart such as:

```
class CarState(
    val numberOfDoors: Int,
    val hasRadio: Boolean,
    val isCoolCar: Boolean,
    val goesVroom: Boolean
)
```

The two are identical, but *CarState* is stable.

Because users might need to convert to and from these classes depending on which architectural layer they are dealing with, you should provide easy mapping functions going both ways such as:

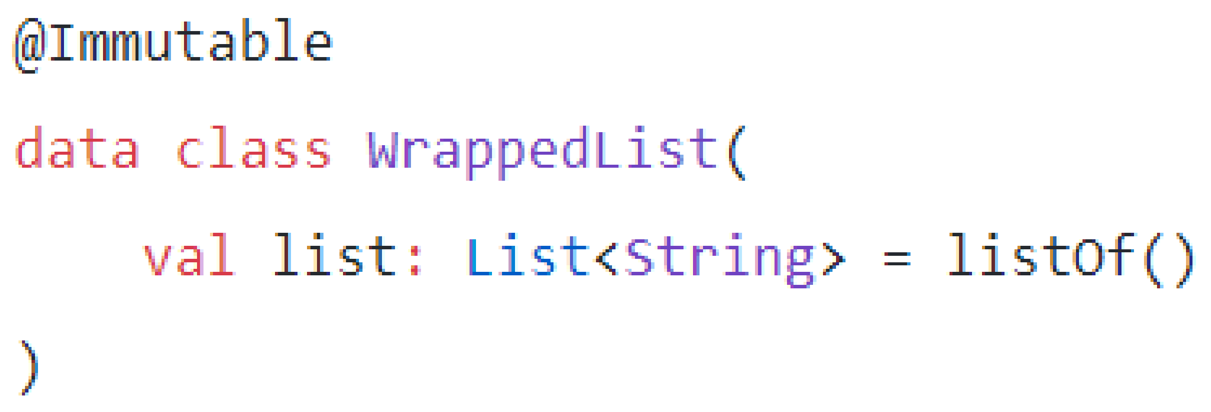
fun Car.toCarState(): CarState
fun CarState.toCar(): Car

4. Do not expect immutability from collections

Things such as *List*, *Set*, and *Map* might seem immutable at first, but they are not and the Compiler will mark them as unstable.

Currently, there are two alternatives, the more straightforward one includes using Kotlin's immutable collections. However, these are still pre-release and might not be viable.

The other solution, which is a technical hack and not officially advised but used by the community, is to wrap your lists and mark the wrapper class as @Immutable.



```
@Immutable
data class WrappedList(
    val list: List<String> = listOf()
)
```

5. Flows are unstable

Even though they might seem stable since they are observable, *Flow*s do not notify composition when they emit new values. This makes them inherently unstable. Use them only if absolutely necessary.

6. Inlined Composables are neither restartable nor skippable

As with all inlined functions, these can present performance benefits. Some common Composables such as *Column*, *Row*, and *Box* are all inlined. As such this is not an admonishment of inlining Composables, just a suggestion that you should be mindful when inlining Composables, or using inlined Composables and be aware of how they affect the parent scope recomposition.

it Agenturen