



David Vávra

[Follow](#)

Google Developer Expert for Android, Founder & CEO at Step Up Labs, GDG Prague organizer, early ...
May 3 · 3 min read

How to remove all !! from your Kotlin code

Null safety is one of the best features of Kotlin. It makes you think about nullability on the language level so you can avoid many hidden `NullPointerException`s which are so common in Java. However, when you automatically convert your Java code into Kotlin, you can see a lot of `!!` symbols there. It looks like you should not have any `!!` in your code unless it's a quick prototype. And I believe it's true, because `!!` basically means “you have a potentially unhandled `KotlinNullPointerException` here”. Plus it looks hacky.

Kotlin has some clever mechanisms how to avoid this, but figuring them out is not straightforward. Here are 6 ways how to do that:

1) Use val instead of var

Kotlin makes you think about immutability on the language level and that's great. `val` is read-only, `var` mutable. It's recommended to use as many read-only properties as you can. They are thread-safe and work great with functional programming. If you use them as immutables, you don't have to care about nullability. Just beware that `val` can actually be mutable.

2) Use lateinit

Sometimes you can't use immutable properties. For example, it happens on Android when some property is initialized in `onCreate()` call. For these situations, Kotlin has a language feature called `lateinit`.

It lets you replace this:

```
1 private var mAdapter: RecyclerView.Adapter<Transaction>? =
2
3 override fun onCreate(savedInstanceState: Bundle?) {
4     super.onCreate(savedInstanceState)
5     mAdapter = RecyclerView.Adapter(R.layout.item_transaction)
6 }
7
```

With this:

```

1  private lateinit var mAdapter: RecyclerView.Adapter<Transac
2
3  override fun onCreate(savedInstanceState: Bundle?) {
4      super.onCreate(savedInstanceState)
5      mAdapter = RecyclerView.Adapter(R.layout.item_transactio
6  }
7

```

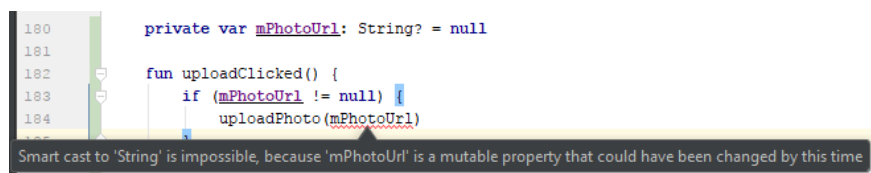
Be aware, that accessing non-initialized `lateinit` property will result in `UninitializedPropertyAccessException`.

`lateinit` sadly doesn't work with primitive data types like `Int`. For primitive types you can use delegates like this:

```
private var mNumber: Int by Delegates.notNull<Int>()
```

3) Use let function

This is a common compile-time error in Kotlin code:



```

180  private var mPhotoUrl: String? = null
181
182  fun uploadClicked() {
183      if (mPhotoUrl != null) {
184          uploadPhoto(mPhotoUrl)
185      }
186  }

```

Smart cast to 'String' is impossible, because 'mPhotoUrl' is a mutable property that could have been changed by this time

It annoyed me: I know that that this mutable property couldn't have been changed after the null check. And many developers quick-fix it with:

```

1  private var mPhotoUrl: String? = null
2
3  fun uploadClicked() {
4      if (mPhotoUrl != null) {
5          uploadPhoto(mPhotoUrl!!)
6      }
7  }

```

But there is an elegant solution using `let` function:

```
1     private var mPhotoUrl: String? = null
2
3     fun uploadClicked() {
4         mPhotoUrl?.let { uploadPhoto(it) }
```

4) Create global functions to handle more complex cases

`let` is a great replacement for a simple null check, but there might be more complex cases. For example:

```
1     if (mUserName != null && mPhotoUrl != null) {
2         uploadPhoto(mUserName!!, mPhotoUrl!!)
3     }
```

You could nest two `let` calls, but that wouldn't be very readable. In Kotlin, you can have globally accessible functions so you can easily build a function you need which is used like this:

```
1     ifNotNull(mUserName, mPhotoUrl) {
2         userName, photoUrl ->
3         uploadPhoto(userName, photoUrl)
4     }
```

Code of this function:

```
1     fun <T1, T2> ifNotNull(value1: T1?, value2: T2?, bothNo
2         if (value1 != null && value2 != null) {
3             bothNotNull(value1, value2)
4         }
```

5) Use Elvis operator

Elvis operator is great when you have a fallback value for the null case. So you can replace this:

```
1 fun getUsername(): String {  
2     if (mUserName != null) {  
3         return mUserName!!  
4     } else {  
5         return "Anonymous"  
6     }  
}
```

With this:

```
1 fun getUsername(): String {  
2     return mUserName ?: "Anonymous"  
3 }
```

6) Crash on your own terms

There are still cases when you know something can't be null, even though the type must be nullable. If it's null, it's a bug in the code and you should know about it. However, leaving `!!` there gives you a generic `KotlinNullPointerException` which is hard to debug. Use build-in functions `requireNotNull` or `checkNotNull` with accompanied exception message for easy debugging.

Replace this:

```
1 uploadPhoto(intent.getStringExtra("PHOTO_URL")!!)
```

With this:

```
1 uploadPhoto(requireNotNull(intent.getStringExtra("PHOTO
```

Conclusion

If you follow these 6 tips, you can remove all `!!` from your Kotlin code. Your code will be safer, more debuggable and cleaner. Let me know in the comments if you know about more ways how to deal with null safety.

**Let me know if there are any
news!**

yourname@example.com

Sign up

