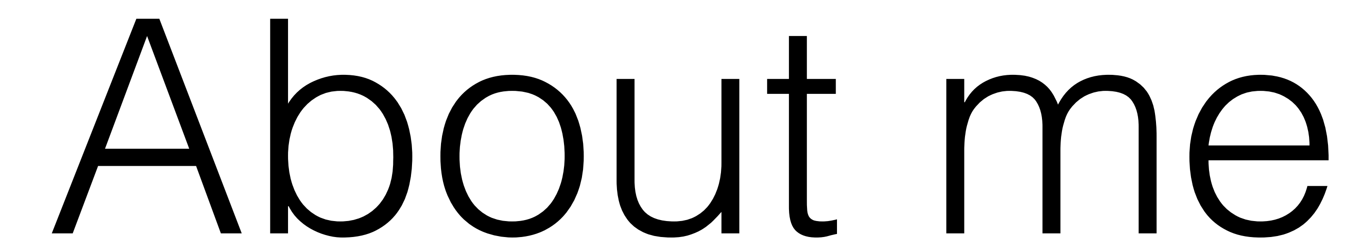
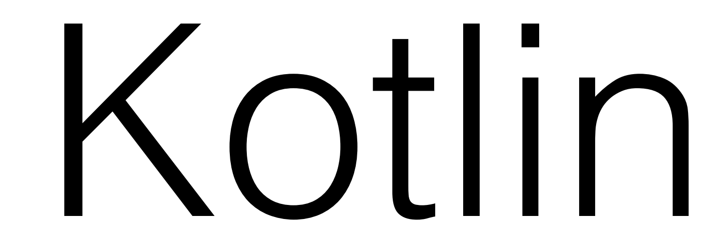
A Deep Dive Into Kotlin

By 技术小黑屋



- 技术小黑屋(droidyue.com)博主
- @Flipboard China（红板报）
- GDG线下分享讲师

微信扫一扫关注



- **An official language for Android recently**
- **Powered by Jetbrains**

- **Concise**
- **Safe**
- **interoperable**
- **tool-friendly**

- **How to interoperate with Java**
- **How to write the idiomatic Kotlin code**
- **What are the pitfalls and how to avoid them**
- **The common best practices**

```kotlin
//nullable and non-null are two types
//to make it possible to check nullability at compile time
var id: String = ""
id = null //compile error

var title: String? = ""
title = null
```

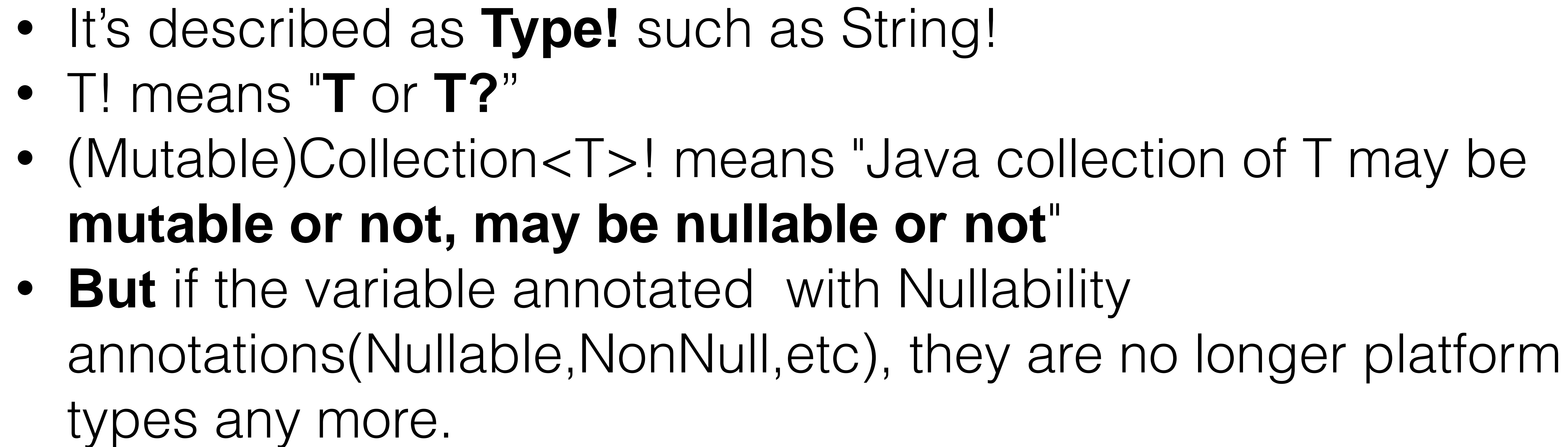
We have less NPEs
But we could not avoid them

```
fun test(value: String?) {
    value?.length
    value!!.length // treat the value as non-null
}
```

```
user.contactInfo!!.emailContacts!!.primaryEmail
//Exception in thread "main" kotlin.KotlinNullPointerException

//a better way
user.contactInfo?.emailContacts?.primaryEmail
```

```
fun getItemNames(items: List<Item?>) {
    items.filter {
        it != null
    }.map {
        it!!.title // not smart here, we need to recheck nullability
    }

    //If we change the filter criteria, we may encounter npe

    items.filterNotNull().map {
        it.title  //smart here
    }
}
```

- Any reference in Java may be null
- It’s not practical to keep null-safety for these from Java
- Types of Java declarations are treated specially in Kotlin and called platform types



- It’s described as **Type!** such as String!
- T! means "**T** or **T?**”
- (Mutable)Collection<T>! means "Java collection of T may be **mutable or not, may be nullable or not**"
- **But** if the variable annotated with Nullability annotations(Nullable,NonNull,etc), they are no longer platform types any more.

- Use annotations for Java variables and functions
- It would be better to treat T! as T? for safety

```
// In JavaMain.java
public static List<String> getList() {
        String[] seasons = {"Spring, Summer, Autumn, Winter"};
        return Collections.unmodifiableList(Arrays.asList(seasons));
}

//Kotlin code
fun mutableListPlatformTypes() {
    //wrong way
    // NPE or UnsupportedOperationException
    JavaMain.getList().add("Kotlin")


    val list = mutableListOf<String>()
    JavaMain.getList()?.let {//maybe null
        list.addAll(it) //use a new list
    }
    list.add("Kotlin")
}
```

```kotlin
//need to check null(especially data from servers)
data class IPInfoResponse(val status: Int, val ip: String) {
    fun isValid(): Boolean {
        return status == 0 && !TextUtils.isEmpty(ip)
    }
}
```

- val is read-only
- var is readable and writable

```kotlin
fun getItemTitle(item: Item) {
    var title: String?
    if (item.isValid()) {
        title = item.title
    } else {
        title = "Not found"
    }


    //the var title may be re-assigned to other value
    //value could be returned through if/else clause
    val title2 = if (item.isValid()) {
        item.title
    } else {
        "Not Found"
    }

}
```

```
val item = getItem(position)
//value returned by when clause
return when(item?.type) {
    "hotmix" -> ITEM_TYPE_HOTMIX
    "promoted" -> ITEM_TYPE_PROMOTED
    "list" -> ITEM_TYPE_ITEM_LIST
    "sudoku" -> ITEM_TYPE_SUDOKU
    else -> ITEM_TYPE_SEARCHBOX
}
```

- It's quick to implement Singleton
- Companion object stores properties and functions of Class
- Do not use Object to store static methods, Use the top-level functions

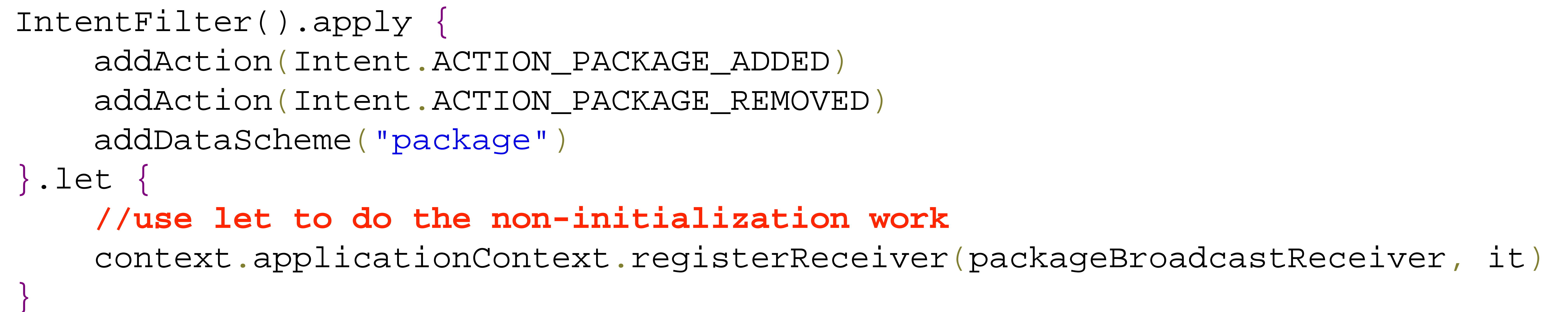
```
class SSOServiceManager {
      companion object  {
            private val APPID = ""
            private val SSO_SCOPE = "get_simple_userinfo"
            val RESULT_OK = -1
            val instance = QQServiceManager()
      }
}
```

- let
- apply
- with
- run

Let

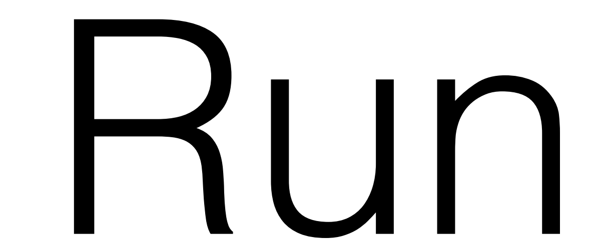
```
//let is a scoping function
DbConnection.getConnection().let { connection ->
}
// connection is no longer visible here


//let is also an alternative for null testing
item?.let {
    //code here
}
```

```
//without apply
val intentFilter = IntentFilter()
intentFilter. addAction(Intent.ACTION_PACKAGE_ADDED)
intentFilter.addAction(Intent.ACTION_PACKAGE_REMOVED)
intentFilter.addDataScheme("package")

//Use apply to group object initialization statements to allow for cleaner, easier to read code.
IntentFilter().apply {
    addAction(Intent.ACTION_PACKAGE_ADDED)
    addAction(Intent.ACTION_PACKAGE_REMOVED)
    addDataScheme("package")
}
```



```
IntentFilter().apply {
    addAction(Intent.ACTION_PACKAGE_ADDED)
    addAction(Intent.ACTION_PACKAGE_REMOVED)
    addDataScheme("package")
}.let {
    //use let to do the non-initialization work
    context.applicationContext.registerReceiver(packageBroadcastReceiver, it)
}
```

```
val w = Window()
with(w) {
  setWidth(100)
  setHeight(200)
  setBackground(RED)
}
```




```
//run it's a combination of let and apply.

val a = "HelloWorld".replace("World","")
a.run {
    println(length)
}
```

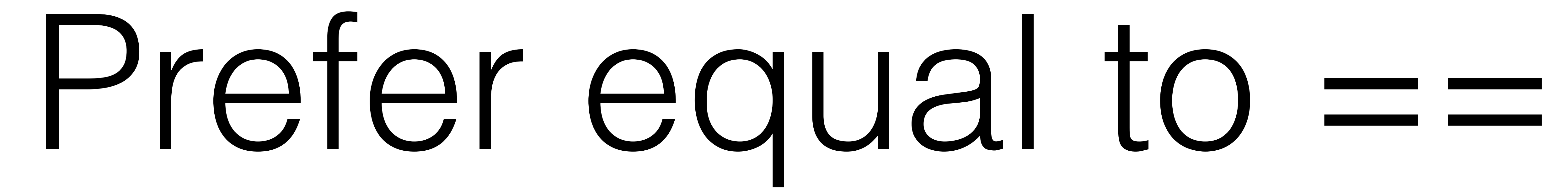
- Inline is a method of optimization(less methods)
- It’s not the JIT inline

```kotlin
inline fun <T> T?.runIfNotNull(block: (T) -> Unit): T? {
    if (this != null) {
        block(this)
    }
    return this
}

fun testInline(arg: Any?) {
    println("testInlineStart")
    arg.runIfNotNull {
        println("arg is not null")
    }
    println("testInlineStop")
}

//after inlined
fun testInline(arg: Any?) {
    println("testInlineStart")
    if (arg != null) {
        println("arg is not null")
    }
    println("testInlineStop")
}
```

- Inline works best for functions with lambda parameter(avoid inner class)
- If a little method is often called, You can inline it.
- Too many inline functions would increase the compiler’s pressure

- @JvmStatic mark elements as static
- @JvmField mark val/var as Java field

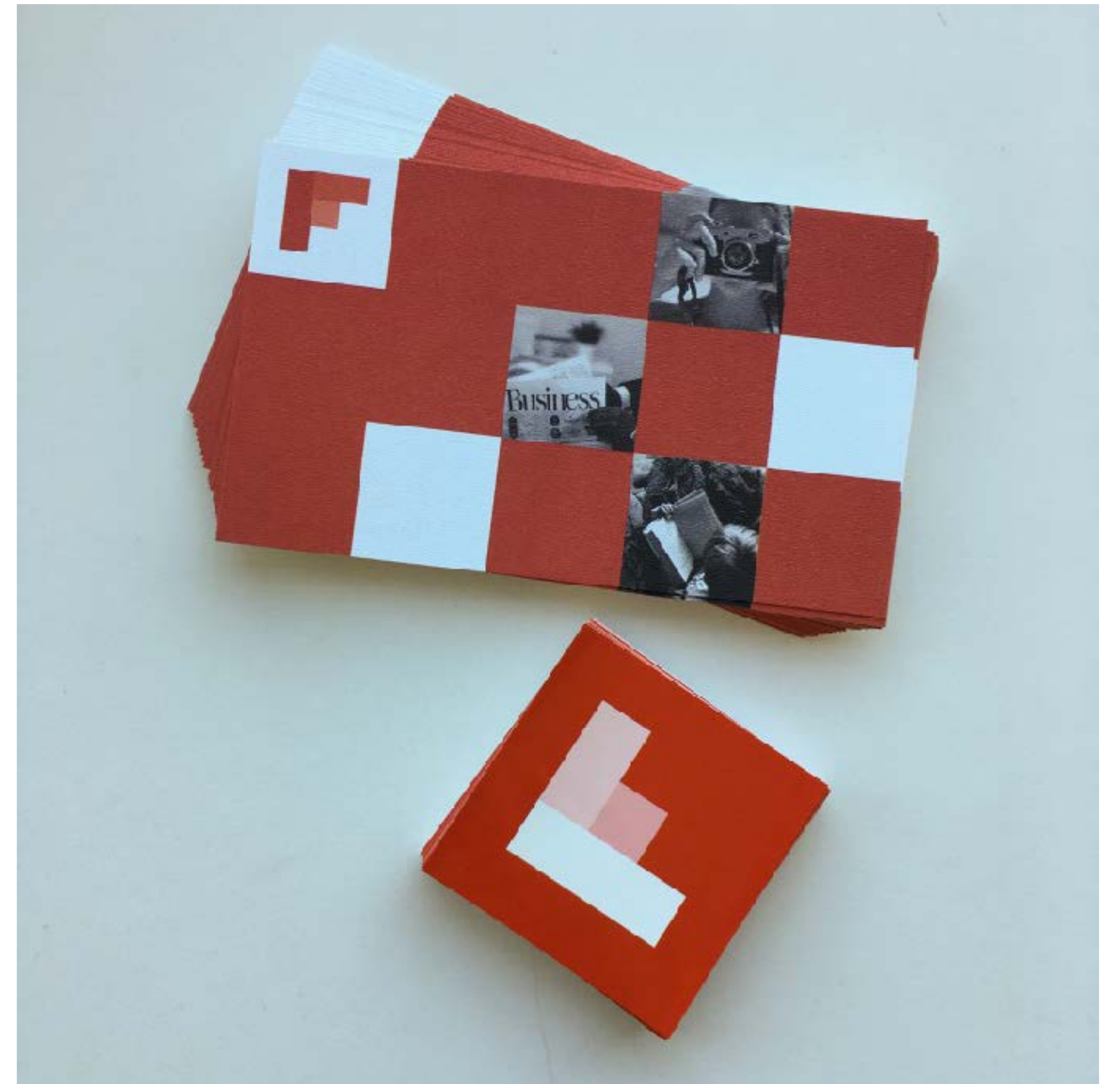
```
object SocialShareHelper {
    @JvmField
    val TARGET_WECHAT = 1

    val TARGET_WEIBO = 2
}

int target = getShareTarget();
    switch (target) {
        case SocialShareHelper.TARGET_WECHAT:
                break;
        case SocialShareHelper.INSTANCE.getTARGET_WEIBO():// error
Kotlin’s pitfall
                break;
}
```



- In Kotlin, == is used to check Strings equality
- But it will be confused to use it in a mixed project

No Checked Exceptions

```
//Checked Exceptions decreased productivity and little or no increase in
code quality.
//we could try-catch exceptions depending on our experience.
fun PackageManager.isPackageExisting(targetPkg: String): Boolean {
    return try {
        getPackageInfo(targetPkg, PackageManager.GET_META_DATA)
        true
    } catch (e: PackageManager.NameNotFoundException) {
        e.printStackTrace()
        false
    }
}
```



Gifts