

Pragmatic **Kotlin** ♥

Practical Tips to migrate your Android App to Kotlin

By Ravindra Kumar [@ravidark](#)

About Me

- Ravindra Kumar [@ravidsrk](#)
- Android Developer [@Fueled](#)
- **Proud Kannadiga** from *Bengaluru*
- Speaker at **Droidcon In, Jsfoo, TiConf**
- Creator of [AndroidStarters](#)
- Open source contributor [@ravidsrk](#)
- Author of [Android Testing Guide](#)

Agenda

- ~~Talk Intro~~
- ~~About Me~~
- ~~Agenda - *In Progress*~~
- Steps to Convert
- Common converter Issues
- Takeaways
- Eliminate all !! from your Kotlin code

Steps to **Convert**

Once you learn basics syntax of **Kotlin**

1. Convert files, one by one, via "`mv *.java *.kt`", make sure tests still pass

Steps to **Convert**

Once you learn basics syntax of **Kotlin**

1. Convert files, one by one, via "`mv *.java *.kt`", make sure tests still pass
2. Go over the Kotlin files and make them more [idiomatic](#).

Steps to **Convert**

Once you learn basics syntax of **Kotlin**

1. Convert files, one by one, via "`\u2192 \u2197 \u2699 K`", make sure tests still pass
2. Go over the Kotlin files and make them more **idiomatic**.
3. Repeat step 2 until you convert all the files.

Steps to **Convert**

Once you learn basics syntax of **Kotlin**

1. Convert files, one by one, via "`\u2192 \u2197 \u2692 K`", make sure tests still pass
2. Go over the Kotlin files and make them more [idiomatic](#).
3. Repeat step 2 until you convert all the files.
4. Ship it.

Common Converter **Issues**

- TypeCasting for the sake of Interoperability.

Common Converter **Issues**

- TypeCasting for the sake of Interoperability.
- *Companion* will add extra layer.

Common Converter **Issues**

- TypeCasting for the sake of Interoperability.
- *Companion* will add extra layer.
- If java method starting with getX(), converter looks for property with the name X.

Common Converter **Issues**

- TypeCasting for the sake of Interoperability.
- *Companion* will add extra layer.
- If java method starting with getX(), converter looks for property with the name X.
- Generics are hard to get it right on the first go.

Common Converter **Issues**

- TypeCasting for the sake of Interoperability.
- *Companion* will add extra layer.
- If java method starting with getX(), converter looks for property with the name X.
- Generics are hard to get it right on the first go.
- No argument captor.

Common Converter **Issues**

- TypeCasting for the sake of Interoperability.
- *Companion* will add extra layer.
- If java method starting with getX(), converter looks for property with the name X.
- Generics are hard to get it right on the first go.
- No argument captor.
- *git diff* If two developers are working on same java file and one guy converts it to Kotlin, it will be rework.

TypeCasting for the sake of Interoperability

Kotlin is not Interoperable right away, but you need to do a lot of work around to make it Interoperable

Here is the Java class:

```
public class DemoFragment extends BaseFragment implements DemoView {  
  
    @Override  
    public void displayMessageFromApi(String apiMessage) {  
        ...  
    }  
}
```

TypeCasting for the sake of Interoperability

```
// Kotlin class
```

```
class DemoResponse {  
    @SerializedName("message") var message: String? = null  
}
```

```
// Typecasting to String
```

```
mainView?.displayMessageFromApi(demoResponse.message as String)
```

Companion will add extra layer

Here is Java class:

```
public class DetailActivity extends BaseActivity implements DetailMvpView{
    public static final String EXTRA_POKEMON_NAME = "EXTRA_POKEMON_NAME";

    public static Intent getStartIntent(Context context, String pokemonName) {
        Intent intent = new Intent(context, DetailActivity.class);
        intent.putExtra(EXTRA_POKEMON_NAME, pokemonName);
        return intent;
    }
}
```


Companion will add extra layer

Converted Kotlin class:

```
class DetailActivity : BaseActivity(), DetailMvpView {
    companion object {
        val EXTRA_POKEMON_NAME = "EXTRA_POKEMON_NAME"

        fun getStartIntent(context: Context, pokemonName: String): Intent {
            val intent = Intent(context, DetailActivity::class.java)
            intent.putExtra(EXTRA_POKEMON_NAME, pokemonName)
            return intent
        }
    }
}
```

Companion will add extra layer

```
public class MainActivity extends BaseActivity implements MainMvpView {  
    private void pokemonClicked(Pokemon pokemon) {  
        startActivity(DetailActivity.Companion.getStartIntent(this, pokemon))  
    }  
}
```

Companion will add extra layer

```
public class MainActivity extends BaseActivity implements MainMvpView {  
    private void pokemonClicked(Pokemon pokemon) {  
        startActivity(DetailActivity.Companion.getStartIntent(this, pokemon))  
    }  
}
```

Remember: *you do not need to stress about migrating the entire codebase.

Method names starting with **get**

Here is the Java class:

```
public interface DemoService {  
    @GET("posts")  
    Observable<PostResponse> getDemoResponse();  
  
    @GET("categories")  
    Observable<CategoryResponse> getDemoResponse2();  
}
```

Method names starting with **get**

```
interface DemoService {  
    @get:GET("posts")  
    val demoResponse: Observable<PostResponse>  
  
    @get:GET("categories")  
    val demoResponse2: Observable<CategoryResponse>  
}
```

Expecting methods **demoResponse** and **demoResponse2**, They are being interpreted as getter methods, this will cause lots of issues.

No **ArgumentCaptor**

If you are using Mockito's **ArgumentCaptor** you will most probably get following error

```
java.lang.IllegalStateException: classCaptor.capture() must not be null
```

The return value of **classCaptor.capture()** is null, but the signature of **SomeClass#someMethod(Class, Boolean)** does not allow a *null* argument.

mockito-kotlin library provides supporting functions to solve this problem

Key Takeaways

- **annotationProcessor** must be replaced by **kapt** in build.gradle
- Configure tests to mock final classes
- If you are using android **data-binding**, include:

```
kapt com.android.databinding:compiler:3.0.0
```

- `@JvmField` to rescue while using ButterKnife `@InjectView` and Espresso `@Rule`

Eliminate all **!!** from your **Kotlin** code

1. Use **val** instead of **var**
2. Use **lateinit**
3. Use **let** function
4. User **Elivis** operator

Use **val** instead of **var**

- Kotlin makes you think about immutability on the language level and that's great.
- *var* and *val* mean "writable" and "read-only"
- If you use them as immutables, you don't have to care about nullability.

Use **lateinit**

```
private var adapter: RecyclerViewAdapter<Droids>? = null

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    mAdapter = RecyclerViewAdapter(R.layout.item_droid)
}

fun updateTransactions() {
    adapter!!.notifyDataSetChanged()
}
```

Use **lateinit**

```
private lateinit var adapter: RecyclerViewAdapter<Droids>

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    mAdapter = RecyclerViewAdapter(R.layout.item_droid)
}

fun updateTransactions() {
    adapter?.notifyDataSetChanged()
}
```

Use **let** function

```
private var photoUrl: String? = null
```

```
fun uploadClicked() {  
    if (photoUrl != null) {  
        uploadPhoto(photoUrl!!)  
    }  
}
```

Use **let** function

```
private var photoUrl: String? = null

fun uploadClicked() {
    photoUrl?.let { uploadPhoto(it) }
}
```

User **Elvis** operator

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

```
fun getUsername(): String {  
    if (mUserName != null) {  
        return mUserName!!  
    } else {  
        return "Anonymous"  
    }  
}
```

User **Elvis** operator

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

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

Kotlin Extensions

```
Toast.makeText(this, "GDG Ahmedabad", Toast.LENGTH_LONG).show()
```


Kotlin Extensions

```
Toast.makeText(this, "GDG Ahmedabad", Toast.LENGTH_LONG).show()
```

```
fun Context?.toast(text: CharSequence, duration: Int = Toast.LENGTH_LONG) =  
this?.let { Toast.makeText(it, text, duration).show() }
```

Final tip

Don't try to learn the whole language at once

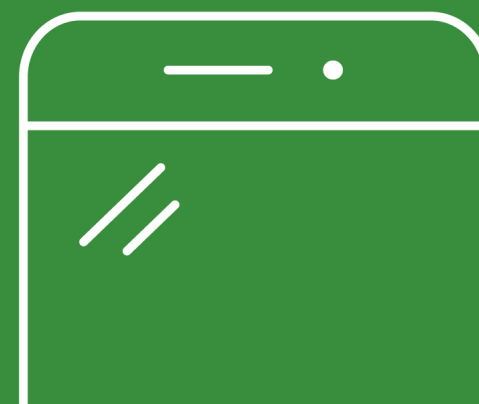
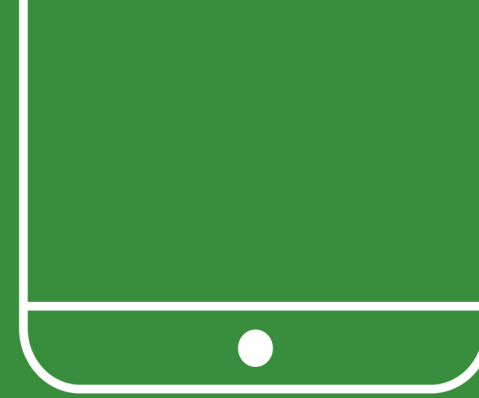
Questions?



GDGDevFest
Thank You
2017 Ahmedabad

Android Testing Guide

- Everything to start writing tests for Android App.
- 75% discount on my upcoming book use [DEVFESTAHM](#)
- <https://leanpub.com/android-testing/c/DEVFESTAHM>



Android Testing Guide

Practical tips and techniques for testing real-world android applications.

Ravindra Kumar