Experiences and lessons learned rewriting an Android app in Kotlin

I recently started rewriting a couple of my apps in Kotlin. Android Studio provides a tool to automatically convert Java files to Kotlin, but I opted to rewrite the apps manually, to get used to the syntax and to learn. After manually rewriting one app, I compared the results with the converter tool, which showed me a couple of additional improvements I could implement.

This article is about my experience and the few things I learned, rewriting the French Revolutionary Calendar app.

To give an idea of the effort in rewriting an app in Kotlin: This project had 2189 lines of Java code (excluding comments and whitespace, per cloc) prior to the rewrite, 1983 lines of Kotlin code after, and took about two full-time days to convert. During the rewrite, I didn't do any additional refactoring or significant improvements. For each class, I opened the Java and Kotlin files in split-screen mode in Android Studio, and wrote the Kotlin code line-by-line looking at the Java code.

I didn't really read much about Kotlin before jumping into the rewrite.

General impressions

- Kotlin is less verbose than Java, without losing readability. This project has about 10% fewer lines of code now. And the lines are shorter as well: No more need for `void` for example.
- Kotlin and Java can call each other, which makes a progressive migration possible. During the transition, there may be temporary code that will ultimately be removed once the project is completely in Kotlin. In my case, I had to temporarily add few keywords (`INSTANCE`, `CompanionObject`, and `JvmField`).
- The only crashes and issues requiring workarounds that I encountered were due to supporting a very old api level in my project.
- I haven't done any benchmarking to compare the app's performance in Java vs Kotlin.

Here are some of the specific features of Kotlin that made an impression.

Utility classes as "objects": Nice!

In java, when creating a utility class, you generally have to do the following, to please static code analysis tools like Sonar:

- make the class final
- create a private constructor with a comment "prevent instantiation"
- make each method `static final`

Example in Java:

```
public final class ApiHelper {
    private ApiHelper() {
        // prevent instantiation
    }

    public static int getAPILevel() {
        //noinspection deprecation
        return Integer.parseInt(Build.VERSION.SDK);
    }
}
```

In Kotlin, you can use an `object`. No need for a private constructor or `static final` qualifiers:

First iteration:

```
object ApiHelper {
    @Suppress("DEPRECATION")
    fun getApiLevel() : Int {
        return Integer.parseInt(Build.VERSION.SDK)
    }
}
```

If you're wondering about the deprecated API, keep reading :)

Data objects: Nice! (except for private constructors)

When you need a read-only class which is simply a collection of fields, in java you should:

- Create a constructor with the fields
- Optionally: make the fields private and add getters. Or you can leave the fields as `public final`.
- Override `equals()`, `hashCode()`, `toString()`
- You can use the IDE to generate these methods. In any case, you may end up suppressing warnings about complexity or magic numbers in these methods, because you don't want to focus on these generated methods when analyzing the code quality of your project.

Example in java, with plenty of boilerplate code:

```
public class Action {
    public final String title;
    @DrawableRes
    public final int iconId;
    public final Intent intent;
    public final PendingIntent pendingIntent;

    public Action(@DrawableRes int iconId, String title, Intent intent, PendingIntent pendingIntent) {
        this.iconId = iconId;
        this.title = title;
        this.intent = intent;
        this.pendingIntent = pendingIntent;
    }

    @SuppressWarnings({"checkstyle:npathcomplexity", "pmd:NPathComplexity", "squid:S1142"})
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Action action = (Action) o;

        if (iconId != action.iconId) return false;
        if (title != null ? !title.equals(action.title) : action.title != null) return false;
        if (intent != null ? !intent.equals(action.intent) : action.intent != null) return false;
        return pendingIntent != null ? pendingIntent.equals(action.pendingIntent) : action.pendingIntent
```

```
    }

    @SuppressWarnings({"checkstyle:magicnumber", "pmd:NPathComplexity"})
    @Override
    public int hashCode() {
        int result = title != null ? title.hashCode() : 0;
        result = 31 * result + iconId;
        result = 31 * result + (intent != null ? intent.hashCode() : 0);
        result = 31 * result + (pendingIntent != null ? pendingIntent.hashCode() : 0);
        return result;
    }

    @Override
    public String toString() {
        return "Action{" +
                "title='" + title + '\'' +
                ", iconId=" + iconId +
                ", intent=" + intent +
                ", pendingIntent=" + pendingIntent +
                '}';
    }
```

In Kotlin:

```
data class Action(@DrawableRes val iconId: Int,
                  val title: String,
                  val intent: Intent,
                  val pendingIntent: PendingIntent)
```

Wow! So much shorter.

However, in my project, this `Action` class actually had a private constructor, and factory methods to create instances, like this:

```
...
private Action(@DrawableRes int iconId, String title, Intent intent, PendingIntent pendingIntent) {
    this.iconId = iconId;
    this.title = title;
    this.intent = intent;
    this.pendingIntent = pendingIntent;
}

public static Action getLightSearchAction(Context context, FrenchRevolutionaryCalendarDate date) {
    return getSearchAction(context, date, R.drawable.ic_action_search);
}

public static Action getDarkSearchAction(Context context, FrenchRevolutionaryCalendarDate date) {
    return getSearchAction(context, date, R.drawable.ic_action_search_dark);
}

private static Action getSearchAction(Context context, FrenchRevolutionaryCalendarDate date, @DrawableR
    Intent intent = new Intent(Intent.ACTION_WEB_SEARCH);
    intent.putExtra(SearchManager.QUERY, date.getObjectOfTheDay());
    PendingIntent pendingIntent = PendingIntent.getActivity(context, R.id.action_search, intent, Pendin
```

```
    return new Action(iconId, context.getString(R.string.popup_action_search, date.getObjectOfTheDay())
}
...
```

I attempted to do the same in Kotlin, with a private constructor:

```
data class Action private constructor(@DrawableRes val iconId: Int,
                                      val title: String,
                                      val intent: Intent,
                                      val pendingIntent: PendingIntent) {
    companion object {
        fun getLightSearchAction(context: Context, date: FrenchRevolutionaryCalendarDate): Action = getS
        fun getDarkSearchAction(context: Context, date: FrenchRevolutionaryCalendarDate): Action = getSe

        private fun getSearchAction(context: Context, date: FrenchRevolutionaryCalendarDate, @DrawableRe
            var intent = Intent(Intent.ACTION_WEB_SEARCH)
            intent.putExtra(SearchManager.QUERY, date.objectOfTheDay)
            val pendingIntent = PendingIntent.getActivity(context, R.id.action_search, intent, PendingIn
            return Action(iconId, context.getString(R.string.popup_action_search, date.objectOfTheDay),
        }
```

The IDE showed a warning about the private constructor:

> Private data class constructor is exposed via the generated 'copy' method. This inspection reports private constructors of data classes because they are always exposed via the generated 'copy' method.

After reading discussions on this issue here and here, I decided to just have a public constructor and factory methods. You might prefer to use a normal class instead of a data class.

Compatibility with Java 1.5

Unless you're crazy, you probably don't care about backwards compatibility with Java 1.5. The Kotlin FAQ on the Android developer website says:

> Which versions of Android does Kotlin support?
>
> All of them! Kotlin is compatible with JDK 6, so apps with Kotlin safely run on older Android versions.

This doesn't specify *which* "older Android versions" are supported. For nostalgic (and masochistic) reasons, my app still works on Cupcake (Android 1.5, sdk level 3!). There were a couple of hiccups porting it to Kotlin, but in the end, there were no blockers, and I learned a couple of things about how Kotlin works on the way.

Passing varargs around

This is how a method signature with varargs looks like:

Java:

```
static void fitTextViewsVertically(Context context,
                                   @DimenRes int widgetHeightRes,
                                   @DimenRes int widgetTopMarginRes,
                                   @DimenRes int widgetBottomMarginRes,
                                   TextView... textViews) {
    ...
}
```

Kotlin:

```
fun fitTextViewsVertically(context: Context,
                           @DimenRes widgetHeightRes: Int,
                           @DimenRes widgetTopMarginRes: Int,
                           @DimenRes widgetBottomMarginRes: Int,
                           vararg textViews: TextView) {
    ...
}
```

This works fine even on Java 1.5. What doesn't work, is if a method passes the varargs argument to another method, as follows:

Java:

```
static void fitTextViewsVertically(Context context,
                                   @DimenRes int widgetHeightRes,
                                   @DimenRes int widgetTopMarginRes,
                                   @DimenRes int widgetBottomMarginRes,
                                   TextView... textViews) {
    float totalTextViewsAllowedHeight = ...

    fitTextViewsVertically(totalTextViewsAllowedHeight, textViews);
}

private static void fitTextViewsVertically(float totalHeight, TextView... textViews) {
    ...
}
```

Kotlin:

```
fun fitTextViewsVertically(context: Context,
                           @DimenRes widgetHeightRes: Int,
                           @DimenRes widgetTopMarginRes: Int,
                           @DimenRes widgetBottomMarginRes: Int,
                           vararg textViews: TextView) {
    val totalTextViewsAllowedHeight = ...

    fitTextViewsVertically(totalTextViewsAllowedHeight, *textViews)
}

private fun fitTextViewsVertically(totalHeight: Float, vararg textViews: TextView) {
    ...
}
```

Note that the `*` character is the spread operator, and allows you to pass a varargs parameter to another method. I discovered, thanks to crashes on Cupcake and `javap`, that under the hood, Kotlin uses the `Arrays.copyOf()` method to implement the spread operator. This function was added in java 1.6. In order to support my deprecated devices, I had to modify the second method to accept an array instead of a varargs, as follows:

```
fun fitTextViewsVertically(context: Context,
                           @DimenRes widgetHeightRes: Int,
                           @DimenRes widgetTopMarginRes: Int,
                           @DimenRes widgetBottomMarginRes: Int,
                           vararg textViews: TextView) {
    val totalTextViewsAllowedHeight = ...

    fitTextViewsVertically(totalTextViewsAllowedHeight, Array(textViews.size) { textViews[it] })
}

private fun fitTextViewsVertically(totalHeight: Float, textViews: Array<TextView>) {
    ...
}
```

`String.format()`

On Kotlin, I discovered that `String.format()` also uses `Arrays.copyOf()` under the hood. When you use `String` in Kotlin, you aren't actually using `java.lang.String`, but rather the `String` class from the Kotlin stdlib. When porting a project to Kotlin, it probably won't make a difference in most cases, unless you still have an emotional attachment to your ADP1 phone. In this case, the app will crash, similar to passing varargs around. The workaround for this is to explicitly invoke the Java String class:

```
java.lang.String.format(Locale.getDefault(), "%d:%02d", frenchDate.hour, frenchDate.minute)
```

Initializing arrays

After a few iterations, I found a concise way to initialize an array with a fixed size which is subsequently populated, using the `Array` constructor which takes an init function argument.

In Java:

```
private static String[] getMonthNames(Locale locale) {
    FrenchRevolutionaryCalendarLabels cal = FrenchRevolutionaryCalendarLabels.getInstance(locale);
    String[] result = new String[13];
    for (int i = 1; i <= 13; i++) {
        result[i - 1] = cal.getMonthName(i);
    }
    return result;
}
```

In Kotlin:

```
private fun getMonthNames(locale: Locale): Array<String> {
    val cal = FrenchRevolutionaryCalendarLabels.getInstance(locale)
    return Array(13) { i ->
        cal.getMonthName(i + 1)
    }
}
```

One-line functions: Nice!

If you have a simple function, you can write it on one line, instead of between curly braces.

Reminder of our first iteration of this function:

```
@Suppress("DEPRECATION")
fun getApiLevel() : Int {
    return Integer.parseInt(Build.VERSION.SDK)
}
```

Second iteration:

```
@Suppress("DEPRECATION")
fun getApiLevel() : Int = Integer.parseInt(Build.VERSION.SDK)
```

Properties: Nice, maybe

Kotlin allows you to access getter and setter methods as if you were accessing a field directly. In my project, I found a couple of use cases for this. The first case was for this now famous `getApiLevel()` method. (By the say, I need this because `Build.VERSION.SDK_INT` doesn't exist in api level 3...)

This is how other parts of the project were initially accessing this method:

```
if (ApiHelper.getAPILevel() >= Build.VERSION_CODES.M) {
    ...
}
```

The third iteration changed this function into a property:

```
object ApiHelper {
    @Suppress("DEPRECATION")
    val apiLevel: Int get() = Integer.parseInt(Build.VERSION.SDK)
}

...

if (ApiHelper.apiLevel >= Build.VERSION_CODES.M) {
...
}
```

This is more concise and easier to read. However, you have to keep in mind when using it, that you're not just reading a field: in this case, each time this property is read, an `Integer.parseInt()` is being executed. I guess the usage of properties should be limited to code which is not very expensive to execute, or which isn't accessed very often.

Another use case for properties is a custom date picker widget class: it has separate `NumberPicker` fields for day, month, and year, but has getters and setters for a `FrenchRevolutionaryCalendarDate` object which accesses the three:

```
class FRCDatePicker : LinearLayout {

    private lateinit var mDayPicker: NumberPicker
```

```
    private lateinit var mMonthPicker: NumberPicker
    private lateinit var mYearPicker: NumberPicker
    ...
    var date
        get(): FrenchRevolutionaryCalendarDate? =
            FrenchRevolutionaryCalendarDate(
                    mYearPicker.value,
                    mMonthPicker.value,
                    mDayPicker.value)
        set(frcDate)  {
            if (frcDate != null) {
                mYearPicker.value = frcDate.year
                mMonthPicker.value = frcDate.month
                mDayPicker.value = frcDate.dayOfMonth
            }
        }
```

Read usage:

```
val selectedDate = mFrcDatePicker.date
```

Write usage:

```
mFrcDatePicker.date = FrenchRevolutionaryCalendarDate(1, 1, 1)
```

Constants in companion objects: a bit annoying

Constants, like the `TAG` constant used for logging, or `EXTRA_XYZ` constants for `Intents`, cannot be defined directly inside the class. Instead, they must go inside a `companion object`, as follows:

```
class FRCDatePicker : LinearLayout {
    companion object {
        private val TAG = Constants.TAG + FRCDatePicker::javaClass.javaClass.simpleName
        private const val EXTRA_YEAR = "year"
        private const val EXTRA_MONTH = "month"
        private const val EXTRA_DAY = "day"
    }
   ...
}
```

They are accessed directly, as in Java:

```
override fun onStart() {
    super.onStart()
    Log.v(TAG, "onStart")
    ...
}
```

Singletons

If you're not using a dependency injection library, you may have singletons in your project. In an Android project, it's possible that your singleton's constructor needs a `Context` parameter, even if it doesn't hold on to it.

My first attempt was essentially a direct port of the Java code, but it didn't compile in Kotlin:

```
@Volatile
private var sInstance: FRCPreferences? = null

fun getInstance(context: Context): FRCPreferences {
    synchronized(this) {
        if (sInstance == null) {
            sInstance = FRCPreferences(context)
        }
        return sInstance
    }
}
```

The error is at the `return` statement:

> Smart cast to FRCPreferences is impossible, because 'sInstance' is a mutable property that could have been changed by this time.

Note, this error occurs whether or not `sInstance` is volatile or the `synchronized` keyword exists.

After going through a few iterations and reading posts about this here, here, and here, I ended up following an example in the android-architecture-components project on GitHub. In my project, it looks like this:

```
class FRCPreferences private constructor(context: Context) {
    companion object {
        private lateinit var mSharedPrefs: SharedPreferences
        ...
        @Volatile
        private var sInstance: FRCPreferences? = null

        fun getInstance(context: Context): FRCPreferences =
                sInstance ?: synchronized(this) {
                    sInstance ?: FRCPreferences(context).also { sInstance = it }
                }
    }
    init {
        mSharedPrefs = PreferenceManager.getDefaultSharedPreferences(context)
    }
    ...
}
```

It's not too verbose, but the `also` and `it` syntax doesn't look that intuitive. It may be simply a question of getting used to Kotlin syntax.

`when` instead of `switch/case`: Yes!!

99.99999999999999999% of the time I've implemented a `switch/case` block, there was a `break;` after each `case`. These `break;` statements annoy me so much I've sometimes implemented `if/else` branches just to avoid them. Thankfully Kotlin offers a more concise way to do `switch/case`: the `when` expression. Here's a before/after:

Java:

```
switch (key) {
    case FRCPreferences.PREF_METHOD:
        updatePreferenceSummary(key, R.string.setting_method_summary);
        break;
    case FRCPreferences.PREF_LANGUAGE:
        updatePreferenceSummary(key, R.string.setting_language_summary);
        break;
    case FRCPreferences.PREF_CUSTOM_COLOR_ENABLED:
        updatePreferenceSummary(key, 0);
        break;
    case FRCPreferences.PREF_SYSTEM_NOTIFICATION_PRIORITY:
        updatePreferenceSummary(key, R.string.setting_system_notification_priority_summary);
        break;
}
```

Kotlin:

```
when (key) {
    FRCPreferences.PREF_METHOD ->
        updatePreferenceSummary(key, R.string.setting_method_summary)
    FRCPreferences.PREF_LANGUAGE ->
        updatePreferenceSummary(key, R.string.setting_language_summary)
    FRCPreferences.PREF_CUSTOM_COLOR_ENABLED ->
        updatePreferenceSummary(key, 0)
    FRCPreferences.PREF_SYSTEM_NOTIFICATION_PRIORITY ->
        updatePreferenceSummary(key, R.string.setting_system_notification_priority_summary)
}
```

Tips and tricks

Here a few things caused me to lose a little bit of time. If you're reading this, you can avoid the same mistakes :)

- *Auto-complete wasn't working for Kotlin classes.* For example, I had a class which extended `Activity`, and the `onCreate` method (or any of the `Activity` methods) was not visible in the IDE. The problem was that I placed the kotlin file in `src/main/kotlin` but forgot to add this to the sourceSet in the `build.gradle` file:

```
android {
    ....
    sourceSets {
        main.java.srcDirs += 'src/main/kotlin'
    }
}
```

- *Activity class not found*: My app compiled just fine and installed on the device, but it crashed upon launching, with an error about the activity class not being found. The problem was that I forgot the `package` declaration at the top of the activity's kotlin file. In Java, this would have been a compile error: as the path of the java file must match the package of the class. This isn't the case in Kotlin.
- *Star imports*: Android Studio was adding `import java.util.*` whenever I just needed one class like `import java.util.Locale`. This can be disabled in Preferences -> Code Style -> Kotlin -> Packages to Use with Import with '*'

Conclusion

Rewriting the app manually was quite a tedious task, taking much longer than I had anticipated, but it allowed me to get used to Kotlin syntax and discover a few details about how Kotlin works. I may consider porting other apps to Kotlin as well. I'm undecided at this point if I will use the converter tool and review the output, or rewrite the code from scratch. My other apps are a bit bigger than the French Revolutionary Calendar project, so a complete manual migration will not be possible in one weekend :) I may do a step-by-step migration starting with utility classes, then data classes, and then other classes by technical type or feature.

The next time I start a new app, I will likely start it in Kotlin.