



Technologie informatyczne w klasycznym i inteligentnym sterowaniu

1. Aplikacje mobilne

Opracował: Maciej Penar

Spis treści

1. Aplikacje mobilne - stan na dziś.....	3
2. Jak zacząć.....	4
Na urządzeniu	4
Przy instalacji Android Studio	4
Po instalacji Android Studio – sprawdź ADB	4
ADB.....	5
Mój pierwszy projekt.....	6
Odpalamy projekt	7
3. Budowa aplikacji Androidowej	9
Aplikacja z punktu widzenia architekta – dużo nieprawdy	9
! Activity.....	10
Cykle życia.....	11
Aplikacja z punktu widzenia Programisty.....	11
Napsujmy coś.....	15
Zamiana napisu HelloWorld – statyczne podejście	15
Zamiana napisu HelloWorld – statycznie, ale z zasobami	15
! Zamiana napisu HelloWorld – Dynamicznie w kodzie	16
! Otworzenie Nowej Aktywności.....	16
Aplikacja z punktu widzenia architekta – sprostowanie	17
4. Kotlin	18
5. Android	18
6. Zaliczenie	27

1. Aplikacje mobilne - stan na dziś

Na chwilę obecną mamy dwa główne systemy operacyjne na smartfony:

- Android – i wszystkie jego pochodne (MIUI/Lineage'y)
- iOS

W branży można też znaleźć obecność innych systemów, które albo odciskają piętno swoją obecnością (Tizen) albo dopiero zamierzają się ujawnić (Fuchsia). Każdy z systemów wypracowuje swój własny sposób developmentu aplikacji mobilnych. Oczywiście jak to w życiu bywa, nie wszystkim się to podoba i jak grzyby po deszcz pojawiają się różne „alternatywne” sposoby wytwarzania aplikacji mobilnych. Ogólnie są trzy metodyki:

1. **Aplikacje natywne** – pisane z wykorzystaniem dedykowanych SDK
 - a. W Androidzie z wykorzystaniem (kiedyś) Javy lub (dziś) Kotlin
 - b. W iOS z wykorzystaniem (kiedyś) Objective-C lub (dziś) Swift
2. **Aplikacje hybrydowe/webowe** – pisane jako aplikacja natywna zawierająca jeden ekran z jednym widgetem tzw. WebView – czyli komponentem przeglądarki. Aplikacje te otwierają specjalną stronę webową stanowiącą de-facto aplikację mobilną. Development odbywa się za pomocą javascriptu i css-a i HTML'a. Kiedyś istniał do tego React Native i PhoneGap/Cordova, ale nie wiem czy te frameworki istnieją, bo są ogólnie uznawane za kiepskie.
3. **Aplikacje hybrydowe/kompilowane** – pisane w dziwnych językach takich jak Dart/C#. Wymagają dołączenia środowisk wykonawczych do aplikacji mobilnych, przez co apki ważą **bardzo dużo**. Przedstawicielami są Xamarin (teraz mniej popularny) i Flutter (pitchowany przez Google'a jako zbawca mobilek).
4. **Aplikacje webowe** – dostępne z poziomu przeglądarki – dopisujemy CSS-y – to rozwiązanie nigdy nie działało dobrze. W założeniu miało działać dobrze na obu systemach, w praktyce nie działa na żadnym.
5. **PWA** – podejście numer dwa do sprzedawania aplikacji webowych jako mobilnych (powodem tej upartości jest trudny proces umieszczania aplikacji w Apple AppStore)

Na tych zajęciach **jedynym** sposobem developmentu który będzie nas interesował to aplikacje **natywne pisanie w Kotlinie**.

Wiem co teraz myślicie – nowy język, nowe problemy – ale Kotlin powstał po to żeby uprościć wytwarzanie aplikacji na Androida. I robi to bardzo skutecznie. Tam gdzie w Javie potrzebowalibyście napisać bardzo dużo linijek kodu, Kotlin robi dużo rzeczy 'pod spodem'.

Jestem zdania, że jeśli mobilki mają Wam się przydać to apki natywne to jedyna opcja dla Automatyków – być może znajdziecie sposób parowania mikrokontrolerami, czy pojawi się potrzeba pchania danych przez USB z komórki itp. (raz spotkałem się z apką która wypychała/pobierała dane z urządzenia zewnętrznego). Taką możliwość daje tylko natywny sposób pisania apek.

2. Jak zacząć

Żeby zacząć programowanie na Androida musicie posiadać:

- Urządzenie z Androidem (bądź symulator) + kabel USB
- Android Studio [link](#)

NA URZĄDZENIU

Co by odblokować urządzenie do developmentu należy:

1. Wejść w **Ustawienia**
2. Wejść w **Telefon – Informacje**
3. Wejść w **Informacje o oprogramowaniu**
4. Kliknąć 5 razy na Numer Wersji – System będzie powiadamiał o liczbie kliknięć
5. Wejść w Ustawienia -> Opcje Programisty, odblokować „Debugowanie USB”

Jak coś się nie zgadza to pooglądać jak to wygląda tu: [link](#)

PRZY INSTALACJI ANDROID STUDIO

Wybrać Kotlina

PO INSTALACJI ANDROID STUDIO – SPRAWDŹ ADB

Po instalacji Android Studio upewnijcie się, że macie zainstalowane **adb**. Na Windowsie binaria powinny być zainstalowane w:

\$FolderUżytkownika\AppData\Local\Android\Sdk\platform-tools

adb służy do komunikacji poprzez usb z urządzeniem, więc żeby sprawdzić czy wszystko jest ok to otwieramy shell:

Wpisujemy:	adb
Powinno wyświetlić:	Android Debug Bridge version 1.0.40 Version 4986621

Jeśli nie wyświetla to musimy wyeksportować w/w ścieżkę do zmiennej %PATH% - liczę, że to powszechna wiedza (na pewno było na Informatyce na pierwszym roku :P). Jeśli wyświetla to podłączcie telefon pod USB (w trybie MTP lub plików) i wpiszcie

Wpisujemy:	adb devices
Powinno wyświetlić:	List of devices attached:

Możliwe, że w tym momencie telefon poprosi o autoryzację wg. komputera (może prosić później). Zaakceptować.

ADB

Tu zostawiam info dla ciekawskich:

Dwie przydatne komendy ADB to:

- **adb start-server** - ensure that there is a server running
- **adb kill-server** - kill the server if it is running

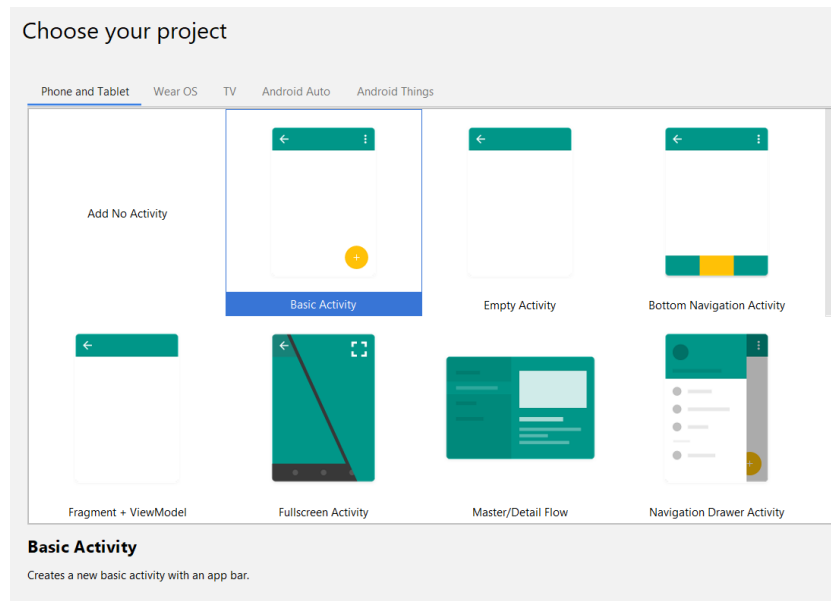
Które zazwyczaj wykorzystywane są gdy coś nakłapiecie i ADB się zatnie. Wtedy kombinacje kill-server + start-server resetuje ADB.

Inne fajne komendy to:

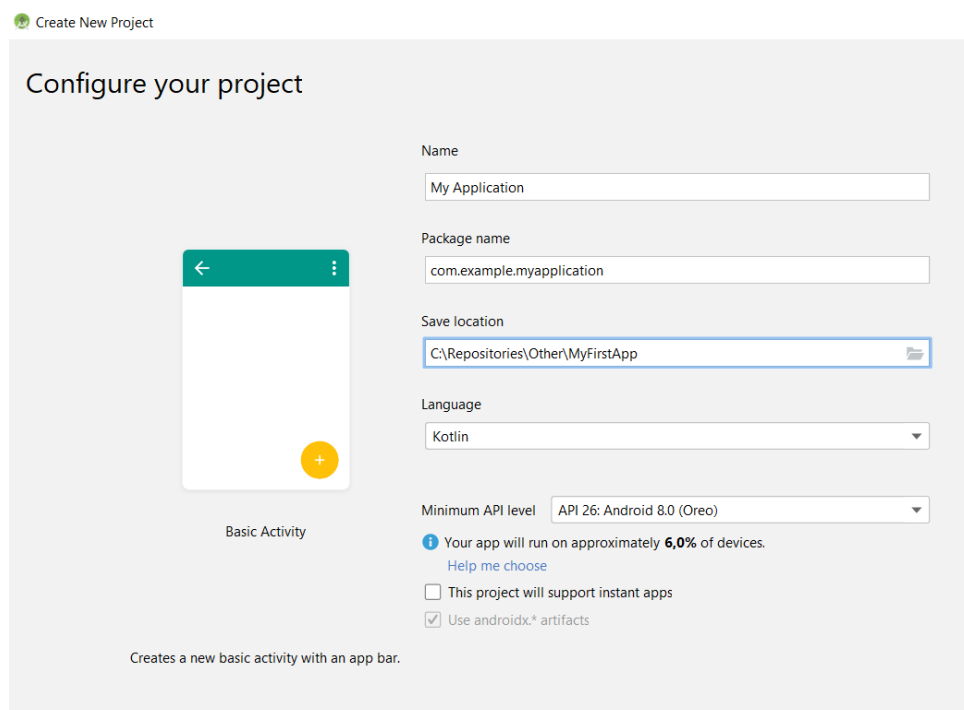
- **adb root** - restart adbd with root permissions
- **adb shell** - run remote shell command (interactive shell if no command given)
- **adb push** - copy local files/directories to device
- **adb pull** - copy files/dirs from device

MÓJ PIERWSZY PROJEKT

1. Otworzyć Android Studio i utworzyć projekt na szablonie **Basic Activity**: File -> New -> New Project

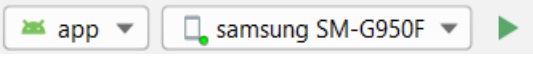
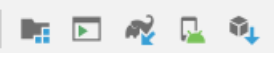


2. Wybrać język Kotlin i odpowiednią nazwę pakietu (Package name) – w przypadku gdybyście wystawiali apke do sklepu to pakiet musi być **globalnie unikatowy**. Wybrać odpowiedni API level – zaraz o tym więcej.





Teraz Android Studio zacznie indeksować pliki i po chwili – jeśli wszystko poszło zgodnie z planem – i Wasza komórka obsługuje wybrany poziom API – to powinniście móc projekt odpalić, jeśli nie możecie to nie stresujcie się, tylko czytajcie dalej.


Istotnymi miejscami w Android Studio są:

1. Opcja File->Sync Project with Gradle Files – służy do synchronizacji środowiska budującego z plikami projektowymi, zazwyczaj ta operacja wykonuje się automatycznie, ale w razie fiaska trzeba uruchamiać ten proces pod podaną ścieżką
2. Odpalenie projektu:  - jeśli sprzęt podłączyliście pod USB i jest widoczne za pomocą **adb devices** – jeśli macie odpalony symulator to w dropdownie też możecie go wybrać
3. Po prawej na górze powinniście widzieć: . Dwie ostatnie ikony to:
 - a. AVD manager – służący do tworzenia symulatorów smartfonów
 - b. SDK manager – służący do instalacji różnych poziomów API.


ODPALAMY PROJEKT

Odpalmy projekt, podpinamy urządzenie pod USB:

1. Jeśli przycisk wygląda  to odpalamy i cieszymy się apką na telefonie.
2. Jeśli przycisk robi problemy to Google'amy wg. komunikatów:
 - a. Jest ryzyko, że nie adb nie wykrywa urządzenia. To możecie zdiagnozować przez **adb devices**
 - b. Jest ryzyko, że nie zaakceptowaliście urządzenia jako zaufanego – Android Studio będzie wyświetlać je jako [UNAUTHORIZED]
 - c. Jest ryzyko, że nie macie zainstalowanego odpowiedniego SDK, tj. Wasz sprzęt jest stary. W tym celu:
 - i. Klikamy  i instalujemy odpowiedni pakiet dla naszego telefonu. Pakiety ważą dużo – nie instalujcie wszystkich jak leci.
 - ii. W AS znajdujemy pliki o nazwie build.gradle


▼  Gradle Scripts

 build.gradle (Project: My Application)

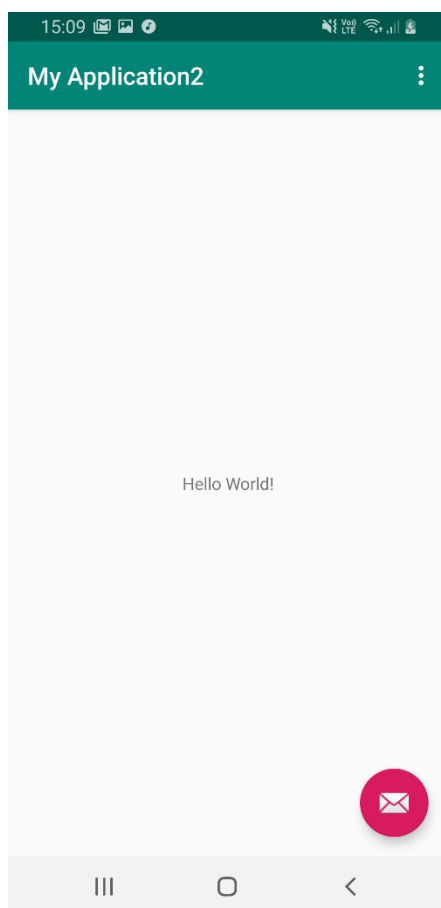
 build.gradle (Module: app)

Wybieramy plik należący do modułu **app** i modyfikujemy sekcję:

```
defaultConfig {
    applicationId "com.example.myapplication"
    minSdkVersion 20
    targetSdkVersion 29
    versionCode 1
    versionName "1.0"
    testInstrumentationRunner
    "androidx.test.runner.AndroidJUnitRunner"
}
```

Tak by wartość minSdkVersion odpowiadała poziomowi który zainstalowaliśmy w zakładce: .

Po odpaleniu projektu powinniście zobaczyć na telefonie:

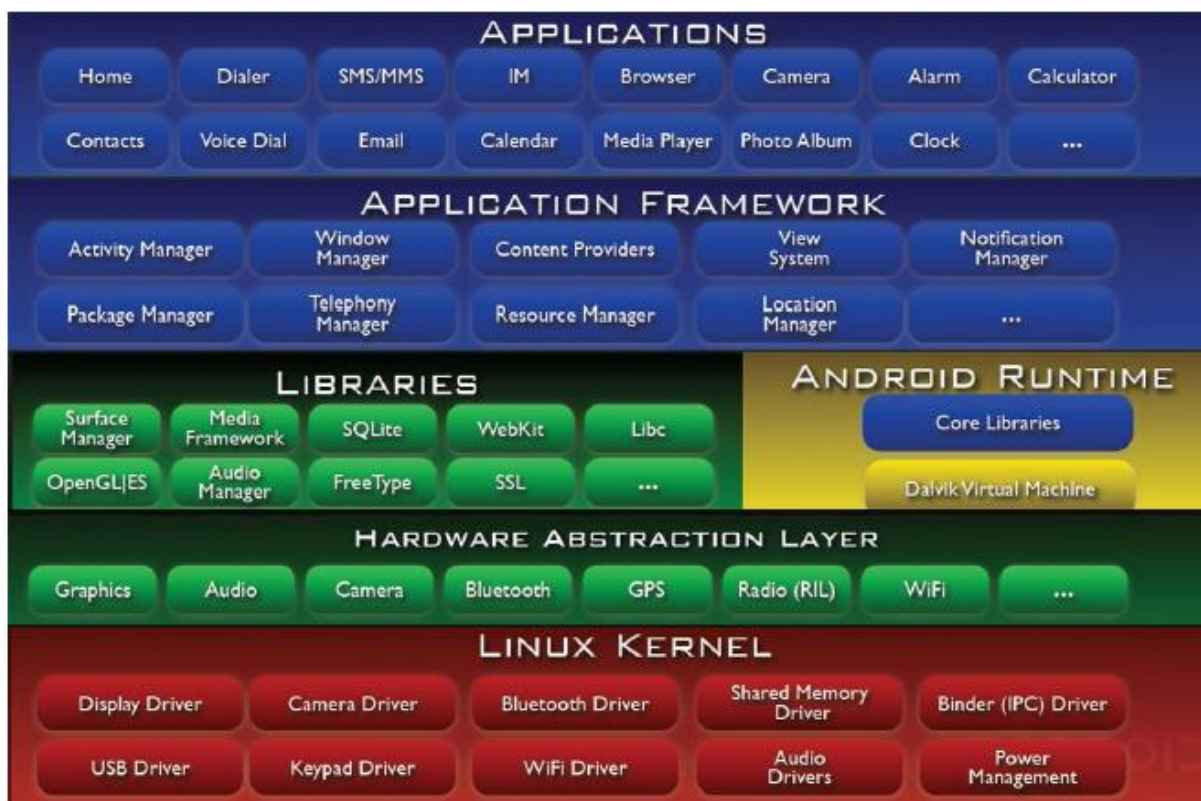


A na pulpicie powinniście widzieć ikone apki.



3. Budowa aplikacji Androidowej

Zamieszczam słynny rysunek architektury Androida obrazujący jego możliwości:



Na diagramie zaznaczono różne funkcjonalne obszary które Android (i jego SDK) oferuje użytkownikom. Dla nas oznacza to tyle, że dowolna funkcjonalność z tego diagramu jest do wykorzystania w przystępny sposób. Na przykład: nie programujemy Maila / wysyłki SMS-ów / przeglądarki internetowej / SQLite'a / GPS-a: istnieją specjalne klasy umożliwiające wykorzystanie tych funkcji. Im wyżej na diagramie, tym „łatwiejsze” do wykorzystania są one.

APLIKACJA Z PUNKTU WIDZENIA ARCHITEKTA – DUŻO NIEPRAWDY

Ikona na pulpicie Androida reprezentuje **aplikację**. **Aplikacja** możemy podzielić na ekrany zwane **aktywnościami**. Każdy ekran musi być jawnie zarejestrowany w pliku manifestu aplikacji tzw. AndroidManifest.xml. Na aplikację mogą składać się też dwa inne komponenty podlegające obowiązkowi rejestracji: **Broadcast Receivery** oraz **serwisy** – oba komponenty realizują długotrwałe/krótkotrwałe działania w tle.

Z punktu widzenia programisty:

- Aplikacja reprezentowana jest przez klasę Application ([link](#))
- **Aktywność reprezentowana jest przez klasę Activity** ([link](#))
- Serwis reprezentowany jest przez klasę Service ([link](#))
- Broadcast reprezentowany jest przez klasę BroadcastReceiver ([link](#))

Bez klasy Activity ani rusz. Klasy Application, Service oraz Broadcasty można sobie na razie podarować. Aktywności stanowią funkcjonalności uruchamiane za pośrednictwem systemu operacyjnego – **Intencja** opisuje ... coś... intencję uruchomienia aktywności. Ekran składa się z użyciem widжетów (widoków) (ang. View).

TO JEST BARDZO WAŻNA SEKCJA.

Aktywność, czyli to co jest wyświetlane użytkownikowi, podlega tzw. cyklowi życia. Obrazuje go następujący diagram:

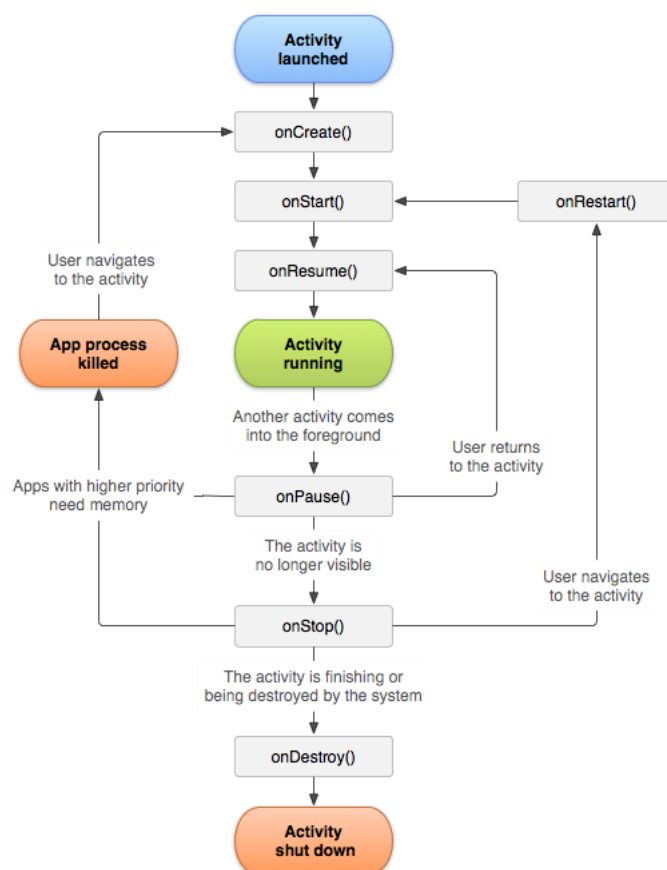


Diagram przedstawia w jakiej kolejności i w jakich warunkach wywoływane są odpowiednie metody klasy Activity. W momencie w którym aplikacja jest uruchomiona, „odpowiednia” aktywność wywołuje metodę `onCreate()`, potem `onStart()` – w tym momencie następuje wyświetlenie widoku użytkownikowi – po czym `onResume()` i użytkownik może cieszyć się aplikacją.

Jeśli użytkownik wejdzie na kolejny ekran – **aktywność nie jest domyślnie usuwana** – dzięki temu przycisku cofnij nie musimy oprogramowywać (mechanizm ten jest obsługiwany przez tzw. BackStack [link](#)). Najpierw wywoływana jest metoda `onPause()`, potem gdy już użytkownik ma zaprezentowany widok nowej aktywności `onStop()`.

Dopiero gdy aplikację ubijemy ręcznie wywoływane jest `onDestroy()` (lub w przypadku nieobsłużonego wyjątku).

No dobra, czym jest „odpowiednia” aktywność:

- Jeśli otwieramy aplikację z pulpitu, która nie działała w tle, to otwierana jest aktywność opatrzona w pliku AndroidManifest.xml następującym filtrem:

```
<intent-filter>
    <action            android:name="android.intent.action.MAIN"           />
    <category          android:name="android.intent.category.LAUNCHER"      />
</intent-filter>
```

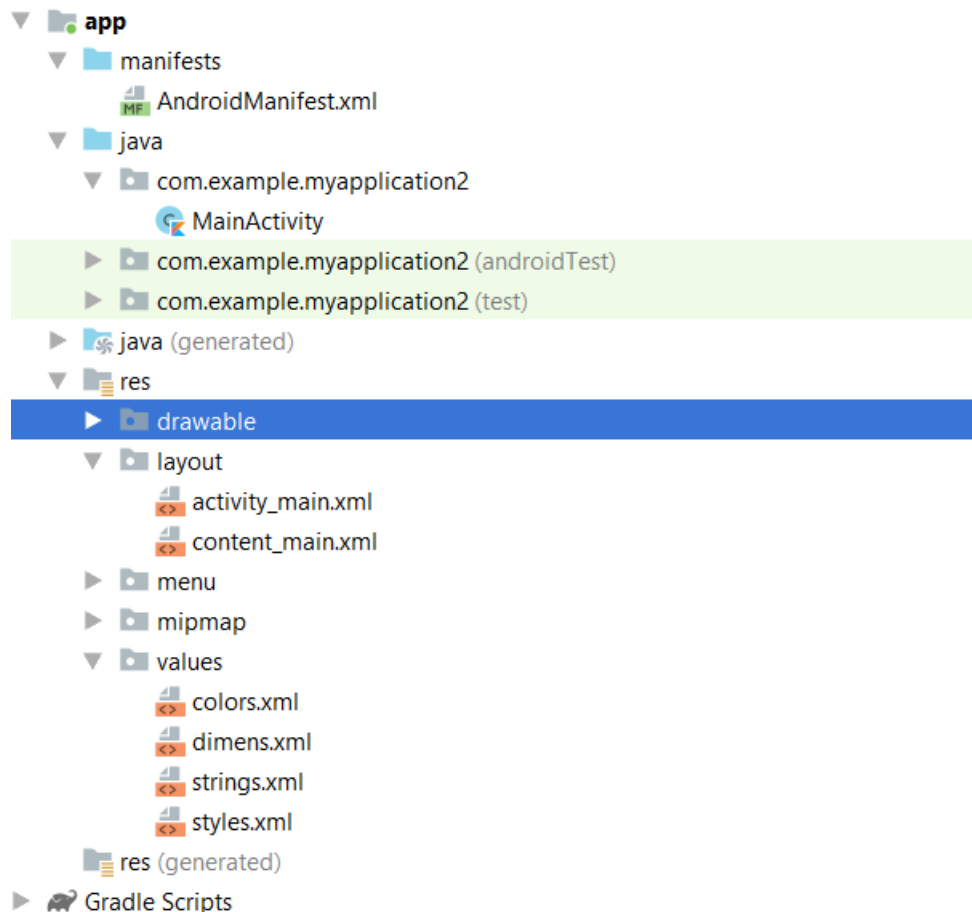
- Jeśli otwieramy aplikację z pulpitu, która działała w tle, to prezentowana jest ostatnia aktywność i cykl przechodzi od onRestart()
- Każdy inny przypadek wywołania aktywności wymaga podania jej nazwy ręcznie

CYKLE ŻYCIA

Tak jak aktywność podlega cyklom życia, tak inne komponenty też są poddane podobnym cyklom. Nazwy konkretnych stanów mogą się oczywiście różnić – odsyłam do dokumentacji. Najbardziej istotne jest to, że definiowanie własnej klasy **Application** wykonujemy gdy chcemy wpłynąć na działanie aplikacji w którymś z jej cykli. Analogicznie sprawa wygląda z klasą **Fragment** – ale o niej na razie cicho sza.

APLIKACJA Z PUNKTU WIDZENIA PROGRAMISTY

Moje drzewko projektu utworzone z szablonu wygląda w następujący sposób:



Możemy wyróżnić 4 główne obszary na których pracujemy:

- **Plik AndroidManifest.xml** – który opisuje co nasza aplikacja może / powinna zrobić i na otwieranie jakich aktywności zezwalamy jako programiści
- **Folder Java/nazwa pakietu** – tu kodujemy w plikach Kotlinowych – **centralne miejsce pracy**
- **Folder res** – używany do przechowywania różnych stałych np. kolorów, ciągów znaków.... **Ale też do przechowywania widoków. Odwołania do zasobów odbywają się z kodzie za pomocą klasy statycznej R.** Na przykład R.layout.xxxxx albo R.strings.aaaaa.
- **Gradle Scripts** – pliki dla środowiska budującego Gradle. Prosty projekt ma zazwyczaj dwa: jeden konfiguruje projekt (w skład projektu może wchodzić wiele aplikacji mobilnych) oraz plik dla już konkretnej aplikacji mobilnej (Modułowy). Tak naprawdę nie ma potrzeby żebyście się tymi plikami bawili – jeśli ktoś znajdzie jakieś fajne libki, to w modułowym w sekcji dependencies może zamieścić odpowiednie wpisy. Warto tutaj zaznaczyć, że Kotlin kompiluje się do kodu bajtowego Javy i też z Javowych libek można korzystać, więc nic tylko grzebać w <https://mvnrepository.com/>. Np. jeśli chcecie mieć dostęp do lepszej libki obsługującej daty ([link](#)) to można wpisać:
compile group: 'joda-time', name: 'joda-time', version: '2.10.5'
Lub po nowemu
implementation 'joda-time:joda-time:2.10.5'

Jak te obszary się spinają u mnie:

Sablon zawiera prostą implementację klasy **aktywności** w folderze Java/com/example/myapplication2 **MainActivity.kt**. Jej definicja wygląda mniej więcej tak:

```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
    }  
  
}
```

Podczas uruchamiania tej aktywności, onCreate() ustawia widok z **res/layout/activity_main.xml** (z kody odwołujemy się przez klasę R: R.layout.**activity_main**) za pomocą **bardzo istotnej metody setContentView()**.

Patrząc na zawartość (nieprzyjemnego) pliku XML **activity_main.xml** widzimy jego definicję:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context=".MainActivity">

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">

        <androidx.appcompat.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />

    </com.google.android.material.appbar.AppBarLayout>

    <include layout="@layout/content_main" />

    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="@dimen/fab_margin"
        app:srcCompat="@android:drawable/ic_dialog_email" />

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Składa się on z kontenera **CoordinatorLayout** wewnątrz którego znajdują się trzy elementy:

- AppBar (pasek narzędzi): **AppBarLayout**
- Właściwy layout: **<include layout="@layout/content_main" />**
- Przycisk na dole ekranu tzw. FAB: **FloatingActionButton**

Taki szablon tak naprawdę nigdy nie ulega zmianie – „mięso” widoku zamieszczone zostało w osobnym pliku: **content_main.xml** wstrzyknętem za pomocą `<include layout="@layout/content_main" />`.

Gdy popatrzymy w środek tego pliku zobaczymy:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:context=".MainActivity"
    tools:showIn="@layout/activity_main">

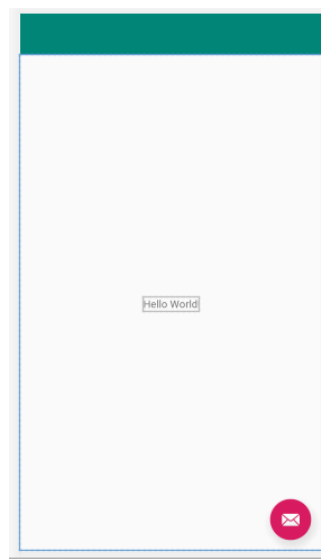
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Właściwy layout składa się tutaj z klasy kontenera **ConstraintLayout** oraz z widgeta wyświetlającego tekst: **TextView**.

Ogólnie rzecz biorąc zrobili w tym szablonie przerost formy nad treścią – nie przejmować się jak te pliki xml-owe wyglądają. One stają się po pewnym czasie bardziej czytelne. Tutaj prezentuję w jaki sposób można sobie dzielić po prostu widoki.

Android Studio renderuje ten widok jak:



Ostatnią rzeczą jaka trzeba obejrzeć jest plik AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapplication2">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER"
            />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

W skład którego wchodzi element **application** którego wartość atrybut **label** to nazwa wyświetana na pulpicie, **icon** to ikona na pulpicie. Wewnątrz węzła application znajduje się jedyna nasza aktywność: MainActivity która została oznaczona intencją : **android.intent.category.LAUNCHER** dzięki której po naciśnięciu ikony na pulpicie uruchamia się aktywność MainActivity.kt.

NAPSUJMY COŚ

ZAMIANA NAPISU HELLOWORLD – STATYCZNE PODEJŚCIE

Jeśli chcielibyśmy zmienić napis w **content_main.xml** na coś innego niż „HelloWorld” to możemy wpisać docelowy tekst prosto w atrybut **text** widgetu **TextView**.

```
android:text="Lubie placki!"
```

ZAMIANA NAPISU HELLOWORLD – STATYCZNIE, ALE Z ZASOBAMI

Jeśli chcielibyśmy zmienić napis w **content_main.xml** na coś innego niż „HelloWorld” to możemy wpisać docelowy tekst w pliku **strings.xml** w zasobach. Tu dodam tekst „FishyFish” identyfikowany kluczem „nothing_fishy”.

```
<resources>
    <string name="app_name">My Application2</string>
    <string name="action_settings">Settings</string>
    <string name="nothing_fishy">FishyFish</string>
</resources>
```

Teraz w pliku widoku mogę **content_main.xml** odwołać się do zasobu – w XML'ach odwołuje się przez notację @[typ zasobu]/[klucz].

```
android:text="@string/nothing_fishy"
```

! ZAMIANA NAPISU HELLOWORLD – DYNAMICZNIE W KODZIE

Każdemu widokowi można przyporządkować identyfikator umożliwiający odwołanie się do elementu w XML, albo w kodzie. Na przykład modyfikując węzeł **TextView** w **content_main.xml** w taki sposób (dodano atrybut **id**):

```
<TextView
    android:id="@+id/view_my_first_view"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/nothing_fishy"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Mogę odwołać się teraz w kodzie za pomocą nadanej nazwy **view_my_first_view** do elementu klasy **TextView**.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    view_my_first_view.text = Date().toString()
}
```

Ważne jest to, że do widoków możemy odwołać się tylko wtedy gdy aktywność wywoła **setContentView()**. Teraz po wywołaniu **onCreate()** nastąpi modyfikacja tekstu i zostanie tam wpisana data.

! OTWORZENIE NOWEJ AKTYWNOŚCI

By otworzyć kolejny ekran naszej aplikacji musimy:

- Utworzyć klasę aktywności
- Utworzyć layout (albo reużyć stary)
- Zarejestrować aktywność w manifeście
- Ustalić w kodzie jak odpalić aktywność

Zacznijmy od utworzenia prostego layoutu:

```
res/layout/activity_secondary.xml
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="New Activity Has Launched" />
</FrameLayout>
```


Teraz tworzymy klasę która w `setContentView()` odwoła się do tego layoutu:

```
class SecondaryActivity : Activity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_secondary)  
    }  
  
}
```

W `AndroidManifest.xml` rejestrujemy aktywność:

```
<activity android:name=".SecondaryActivity"></activity>
```

I na koniec modyfikujemy klasę `MainActivity` – ma ona już `Floating Button` o id: `fab`. Wykorzystajmy to:

```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        fab.setOnClickListener {  
            startActivity(Intent(this, SecondaryActivity::class.java))  
        }  
    }  
  
}
```

Klasa `Button` ma metodę `setOnClickListener()` przyjmującą jako argument funkcję wywoływaną przy aktywacji widoku (przy naciśnięciu). W przypadku gdy jednym argumentem funkcji jest inna funkcja – Kotlin dopuszcza skróconą notację: za pomocą nawiasów klamrowych `{ }` (wyrażenie lambda).

Odpalenie aktywności następuje na wskutek wywołania metody: `startActivity()` która przyjmuje dwa argumenty: kontekst (który komponent systemu prosi o wywołanie aktywności) oraz docelową klasę aktywności. Kontekstem może być zarówno obiekt Aktywności jak i Aplikacji (czy Fragmentu) – tu jest to aktywność reprezentowana przez wywołującego czyli `this`.

APLIKACJA Z PUNKTU WIDZENIA ARCHITEKTA – SPROSTOWANIE

Ikona na pulpicie Androida reprezentuje **aktywność** w ramach **aplikacji** oznaczoną **intencją**: `"android.intent.category.LAUNCHER"`. Aplikacja dzielona jest na zwane **aktywnościami**, które podlegają obowiązkowi rejestracji w pliku `AndroidManifest.xml`. **Fragmenty** reprezentują części ekranu odpowiadające stałym obszarom funkcjonalnym aplikacji – jedna **aktywność** może wyświetlać wiele **fragmentów**. Na **aplikację** mogą składać się też trzy inne komponenty podlegające obowiązkowi rejestracji: **Broadcast Receiver**, **serwisy** (oba komponenty realizują działania w tle) oraz **Content Resolver** – egzotyczny komponent do wymiany danych pomiędzy aplikacjami systemu.

Z punktu widzenia programisty:

- Aplikacja reprezentowana jest przez klasę `Application` ([link](#))
- **Aktywność** reprezentowana jest przez klasę `Activity` ([link](#))
- Serwis reprezentowany jest przez klasę `Service` ([link](#))
- `Broadcast` reprezentowany jest przez klasę `BroadcastReceiver` ([link](#))
- Fragmenty ([link](#))

4. Kotlin

Strona zawierająca referencyjne materiały jest tu: [link](#)

Kotlin jest językiem ogólnego przeznaczenia, który kompilowany jest do kodu dającego uruchomić się na maszynie wirtualnej javy (JVM). Oznacza to, że możemy korzystać z dowolnych libek napisanych w Javie w programie Kotlinowym (i na odwrót). Pliki z kodem kotlina mają rozszerzenie .kt.

HELLO WORLD

Kotlinowy HelloWorld wygląda tak:

```
fun main(args : Array<String>) {  
    println("Hello World")  
}
```

FUNKCJE

Kotlin jest językiem z silną naleciałością paradygmatu programowania funkcyjnego – tj. na ogół w parametrach przekazywane są funkcje. Nie jest to widzimisię języka, a raczej naturalna konsekwencja faktu, że język ten ma głównie zastosowanie do programowania aplikacji mobilnych – a proces ten jest (nie)stety szablonowy.

Funkcje w kotlinie zapisujemy:

```
fun nazwa(argument : Typ, ..., argument : Typ) : Typ
```

Typu zwracanego (fragment `: Typ` na końcu) nie musimy podawać w dwóch przypadkach:

- Jeśli nie chcemy podawać – wtedy domyślnie podany jest typ **Unit** (void w innych językach)
- Jeśli typ wyniku z definicji funkcji – np. funkcje jako wyrażenia

Przykładem funkcji-wyrażenia jest:

```
fun quadric(x : Double) = x * x
```

ZMIENNE

W ciele funkcji pewnie znajdzie potrzeba deklarowania zmiennych mutowalnych/nietutowalnych.

Deklaracja zmiennej mutowalnej – takiej którą możemy nadpisać wygląda tak:

```
val mutable = 1
```

Deklaracja zmiennej nietutowalnej – takiej której nie możemy nadpisać wygląda tak:

```
val nonMutable = 1
```

Oczywiście w obu przypadkach możemy użyć jawnej deklaracji typu, ale nie róbcie tego:

```
val nonMutable : Int = 1  
var mutable : Int = 1
```

Różnica objawia się w środowisku:

```
val nonMutable : Int = 1  
nonMutable = 10 // Nie wolno  
var mutable : Int = 1  
mutable = 10 // Wolno
```

Wyrażenia sterujące opisane są bardzo dobrze tu: [link](#)

LISTY I TABLICE

Do obsługi list i tablic mamy typy: `List<T>` oraz `Array<T>` - wraz z pomocniczymi funkcjami do powoływania obiektów tych klas:

- `listOf()`
`val list = listOf(1, 2, 3, 4, 5, 6)`
- `arrayListOf()`
`val list = arrayListOf(1, 2, 3, 4, 5, 6)`
- `arrayOf()`
`val arr = arrayOf(20, 23, 1, 23, 2, 3)`

! FUNKCJE JAKO ARGUMENTY

Będzie brzmiało skomplikowanie, ale bear-with-me.

Typem może być też funkcja. W kotlinie stosujemy do tego zapis:

`(Typ, Typ, Typ, ..., Typ) -> Typ`

Na przykład dla typów iterowalnych (tablice, listy) zdefiniowana jest funkcja:

```
fun <T> Iterable<T>.filter(predicate: (T) -> Boolean): List<T>
```

Funkcja ta zwyczajowo służy do zwrócenia nowej kolekcji, takich elementów które spełniają pewne kryterium (predykat). Predykat ocenia każdy element z osobna: dlatego funkcja o nazwie argumentu *predicate* operuje na typie T (reprezentujący typ kolekcji).

Zastosowanie tej funkcji jest np. takie: wybierzemy z listy elementy większe od 10:

```
val list = listOf(20, 23, 1, 23, 2, 3)
val output = list.filter {element -> element > 10}
println(output)
```

Podawanie takich mikro-funkcji – nazywanych **lambdami** (Funkcje te nie posiadają nazw, są anonimowe) – wymaga opakowania ich w nawiasy `{}`. W tym zapisie nazwa **element** jest typu ogólnego **T** – wynika to z sygnatury funkcji filter. **Z kolei T jest dla tej konkretnej listy typu Int – oznacza to, że element jest typu Int.**

Funkcje które przyjmują 1 argument który jest funkcją są bardzo, bardzo, bardzo, bardzo, bardzo, bardzo, bardzo, bardzo, bardzo, bardzo, bardzo, bardzo częstym zjawiskiem.

Dlatego, jeśli wywołujecie taką funkcję, możecie ominąć nawiasy okrągłe `()`.

```
val list = listOf(20, 23, 1, 23, 2, 3)
val output = list.filter {element -> element > 10}
println(output)
```

I Kotlin to zrozumie, a nawet będzie na to nalegał.

Idąc krok dalej – funkcje które przyjmują jednoargumentowe funkcje są bardzo częstym zjawiskiem – dlatego Kotlin oferuje skróconą nazwę parametru: **it**.

```
val list = listOf(20, 23, 1, 23, 2, 3)
val output = list.filter { it > 10 }
println(output)
```

W ramach dygresji – zaznaczę tylko, że liczba funkcji którymi operujemy na lista jest długa i daje potężne możliwości. Dla Was oznacza to, że mając listę danych – a z Bazy Danych zazwyczaj otrzymuje się listę – nie trzeba używać konstrukcji takich jak pętle.

W praktyce jeśli chciałbym wyświetlić zawartość tej listy to zastosowałbym taki zapis:

```
val list = listOf(20, 23, 1, 23, 2, 3)
list.filter { it > 10 }.forEach { println(it) }
```

A nawet można iść o krok dalej:

```
listOf(20, 23, 1, 23, 2, 3).filter { it > 10 }.forEach { println(it) }
```

Bo nie potrzebujemy zmiennych – **zmienne są złe**.

Dlatego programowanie przycisku w Androidzie wyglądało tak:

```
fab.setOnClickListener {
    startActivity(Intent(this, SecondaryActivity::class.java))
}
```

NULLPOINTER EXCEPTION

Nullpointer – jaki jest w Javie, wszyscy wiemy. Kotlin stara się minimalizować ryzyko wystąpienia NullPointerException. Oznacza to, że **każdy typ w kotlinie nie przyjmuje wartości null**. Jeśli chcemy dopuścić null jako wartość musimy wybrany typ oznaczyć jako ``?``. Na przykład:

```
val nonnull : Int
val nullable : Int? = null
```

Ze zmiennej nonnull możemy korzystać bezpiecznie. Z kolei zmienna nullable to inna para kaloszy. Jeśli chcemy mieć gwarancję wykonania jakiegokolwiek operacji – musimy użyć notacji ``?.`` zamiast ``.``.

```
val list : List<Int>? = listOf(20, 23, 1, 23, 2, 3)
list.filter { it > 10 }.forEach { println(it) } // Już nie wolno - podkreślenie na czerwono
list?.filter { it > 10 }?.forEach { println(it) } // Tylko tak
```

Problem w tym, że nie mamy gwarancji, że lista nie jest nulem (bo typ to List<Int>?) – z tego względu linijka ...

```
list?.filter { it > 10 }?.forEach { println(it) }
```

... wcale wykonać się nie musi.

Jeśli jesteście pewni, że zmienna nie zawiera wartości null to możecie użyć operatora `!!`. – ale jeśli jednak tam null będzie to dostaniecie NullPointerException – zwróćcie uwagę, że nie ma drugiego `?.`.

```
list!!filter { it > 10 }.forEach { println(it) }
```

Na wypadek gdybyście pisali ryzykowany nullable kod, to kotlin ma kilka feature-ów np. tzw. smart-cast. Jeśli wykonamy sprawdzenie w `if`, to w ciele `if` sprawdzenie będzie respektowane:

```

val list : List<Int>? = listOf(20, 23, 1, 23, 2, 3)
if(list != null){
    // Niby typ List<Int>?, ale sprawdzenie już było
    list.filter { it > 10 }.forEach{ println(it) }
}

```

Analogiczne feature-y dostępne są dla sprawdzenia typu obiektowego.

KLASY

Składnia deklaracji klas dostępna jest tu: [link](#)

Uczulam tylko na istnienie specjalnego typu klas tzw. **data class** [link](#) które można używać wymiennie ze zwykłymi klasami do przechowywania danych.

SCOPE FUNCTIONS

Żeby **uprościć** (pewnie jak to czytacie to myślicie ha-ha) w Kotlinie dostępne są specjalne funkcje które służą zazwyczaj do skrócenia zapisu tzw. scope functions [link](#)

Osochodzi: na każdej obiektowej zmiennej, możemy wywołać jedną z funkcji **let**, **run**, **with**, **apply**, i **also**.

Te funkcje działają **bardzo** podobnie. Na przykład funkcja **let** służy do tego żeby wywołującego przekazać do argumentu lambdy – dzięki czemu możemy odwołać się do niego poprzez **it**.

Założmy zestaw bezsensownych klas. Obiekt klasy C zawiera obiekt klasy B, a ten obiekt klasy A, a ten 2 pola typu Int:

```

class A(var x : Int, var y : Int)
class B(val someField : A)
class C(val myField : B)

```

Następujący kod:

```

val c = C(B(A(1,1)))
c.myField.someField.x = 1
c.myField.someField.y = 2

```

Za pomocą funkcji **let** można zamienić na:

```

val c = C(B(A(1,1)))
c.myField.someField.let {
    it.x = 1
    it.y = 2
}

```

Dzięki temu fragment **c.myField.someField** powtarza się tylko raz. Dzięki funkcji **let** odwołujemy się do niego poprzez **it**.

Analogicznie działa **apply** – tylko przenosi kontekst wywołania to przekazany obiekt jest wywołującym (**this**). Dzięki temu możemy zastosować taki zapis:

```

c.myField.someField.apply {
    x = 1
    y = 2
}

```

Bardzo przydatne dla skłapanych libek w Javie które wymagają pisania tzw. pociągów ([link](#)) – które są **antywzorcem**.

5. Android

TWORZENIE WIDOKÓW

Ogólnie rzecz biorąc widoki możemy podzielić na dwie kategorie:

- Widoki / Widżety – komponenty mające graficzną reprezentację i mające pewną funkcjonalność
- Layouty (grupy widoków / kontenery) – służące do organizacji gdzie widżety powinny być wyświetlone

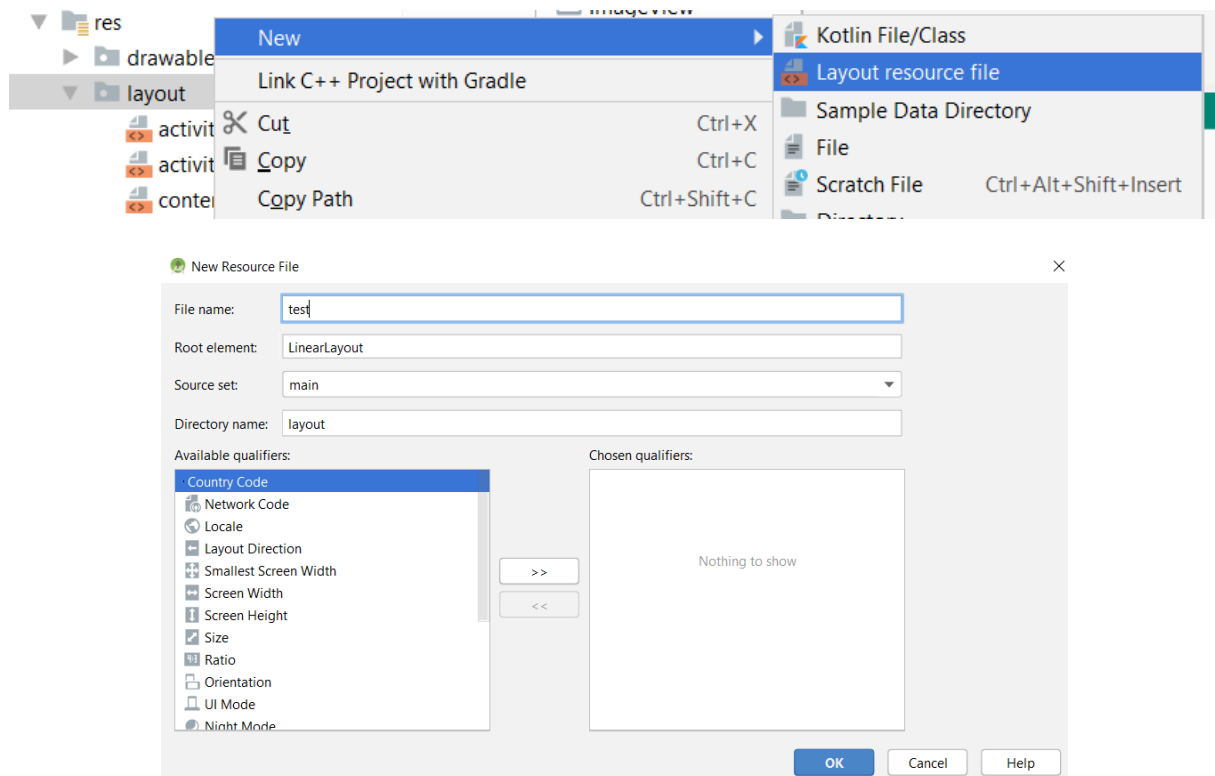
Do popularnych widżetów zaliczamy:

- TextView – komponent do wyświetlania tekstu
- EditText – komponent do wpisywania tekstu
- Button - przycisk
- ImageView – komponent do wyświetlania grafiki
- RecyclerView – komponent do wyświetlania kolekcji danych
- Switch - odpowiednik checkboxa

Popularnymi kontenerami są:

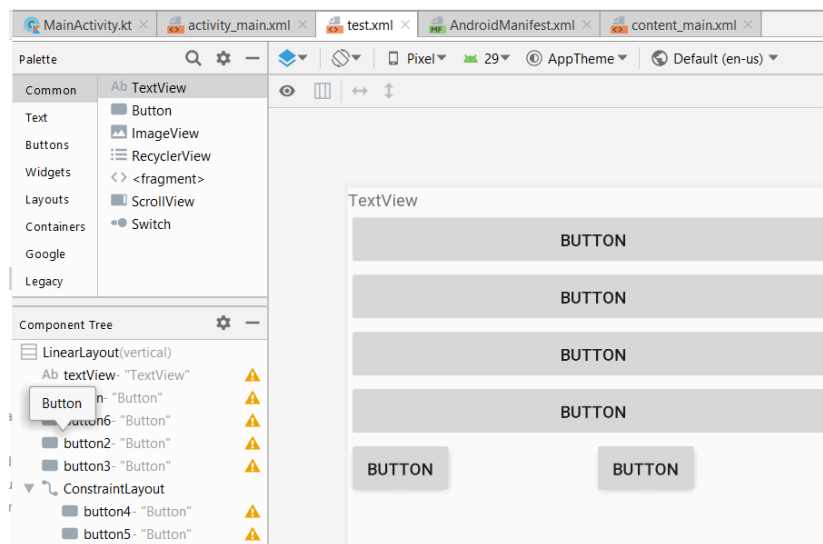
- LinearLayout – wszystkie dzieci tego layoutu zajmują kolejne pozycje (od góry do dołu, albo od lewej do prawej)
- FrameLayout – potomek zajmuje cały obszar tego komponentu
- ConstraintLayout – dzieci tego komponentu umieszczane są według specjalnych ograniczeń

Nowe widoki (ekran) można dodać poprzez:



Layout podany w RootElement można zmienić w dowolnej chwili.

Po utworzeniu layoutu warto pobawić się kreatorem:



W oknie Palette mamy wszystkie dostępne widoki, które za pomocą Drag&Dropa możemy umieszczać wedle naszego widzimisię w oknie Design. Osobiście wolę edytować widoki zakładce Text.

Tworzenie widoków - Jetpack

Istnieje nowy sposób tworzenia ekranów. Szczegóły można znaleźć tu: [link](#)

SurfaceView

Szczególnym widokiem jest SurfaceView, służy on do rysowania dowolnych rzeczy na płótnie (Canvas). Odsyłam do internetu jak korzystać z tego komponentu. Ogólna zasada wygląda tak:

```
class MainActivity : AppCompatActivity(), SurfaceHolder.Callback {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        view_surface.holder.addCallback(this)
    }

    override fun surfaceChanged(surface: SurfaceHolder?, format: Int,
        width: Int, height: Int) {
        /* Rysowanie ... */
    }

    override fun surfaceDestroyed(surface: SurfaceHolder?) {
    }

    override fun surfaceCreated(surface: SurfaceHolder?) {
        surface?.lockCanvas()?.let {
            /* Rysowanie czegokolwiek */
            it.drawRGB(255, 255, 255);
            it.drawOval(RectF(0.0f, 0.0f, 300.0f, 300.0f), Paint().apply {
                color = Color.BLUE })
            surface.unlockCanvasAndPost(it)
        }
    }
}
```

AKCJE

Pamiętajcie, że Android jest systemem który intensywnie korzysta z dotyka. Z tego względu większość komponentów ma różne rodzaje „słuchaczy” (Listenerów) które odpalają zarejestrowane lambdy – tak jak to było w przypadku przycisku. Każdy komponent warto sprawdzić pod kątem metod **setOnXXXXXListener()**. I tak bardzo często możecie znaleźć:

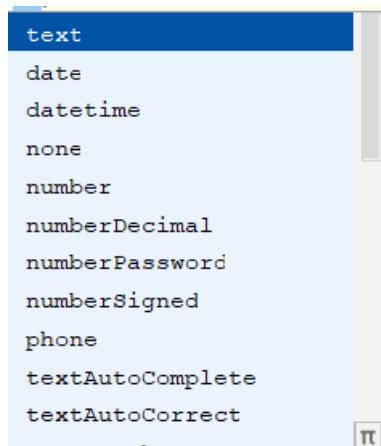
- `setOnClickListener()` – akcja na krótki klik
- `setOnLongClickListener()` – akcja na długi klik
- `setOnCheckedChangeListener()` / `setChangeListener` – akcja gdy wartość komponentu ulega zmianie

WŁAŚCIWOŚCI KOMPONENTÓW (NA PRZYKŁADZIE EDITVIEW)

Każdy dostępny widok daje się skonfigurować. Szkoda życia na programowanie większości funkcjonalności 😊. Przykładowo `EditText` posiada właściwość:

`android:inputType`

Niektóre z wartości jakie można tam podać to:



Wybór odpowiedniej opcji wpływa na to jaka klawiatura jest wyświetlana po otrzymaniu focusa (numeryczna, kalendarz), jak dane są wyświetlane oraz wykonywana jest walidacja.

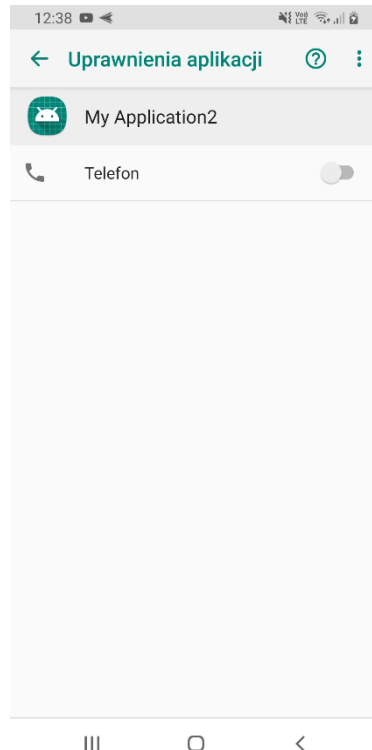
INTENCJE I UPRAWNIENIA

Domyślnie system będzie próbował blokować wszystkie podejrzane aktywności jakie aplikacja chce wykonać. Jeśli nasza aplikacja przewiduje korzystanie z jakichś systemowych funkcjonalności np. wykonywanie połączeń telefonicznych, to w pliku AndroidManifest.xml powinna znaleźć się odpowiednia deklaracja uprawnień.

Przykładowo, deklaracja wymogu uprawnień dot. wykonywania połączeń wygląda tak:

```
<uses-permission android:name="android.permission.CALL_PHONE" />
```

Po umieszczeniu takiej deklaracji w ustawieniach aplikacji zobaczymy:



I już mamy możliwość nadania uprawnień czy aplikacja może dzwonić. Oczywiście warto zapytać o to użytkownika ([link](#)).

Gdy już nadamy uprawnienia do pożądanej funkcjonalności, powinniśmy wywołać odpowiednią aktywność. W przypadku dzwonienia intencja zapisana jest w statycznym polu `Intent.ACTION_CALL` i oczekuje URI w postaci „tel:xxxxxxx”.

```
startActivity(  
    Intent(Intent.ACTION_CALL).apply { data = Uri.parse("tel:111222333" ) }  
)
```

ROOM – SKŁADOWANIE DANYCH

Room to ORM który komunikuje się z SQLitem dobrze opisany jest tu: [link](#) oraz tu: [link](#)

Dlatego ograniczę się do kilku komentarzy:

1) ROOM WYMAGA APT

W dokumentacji jest to zaznaczone, ale we fragmentach kodu nie:

Note: For Kotlin-based apps, make sure you use `kapt` instead of `annotationProcessor`. You should also add the `kotlin-kapt` plugin.

Jeśli tego nie zrobicie, to budowanie projektu nie wygeneruje klas realizujących komunikację z BD.

Więc, trzeba w pliku build.gradle (moduł) dopisać w dowolnym miejscu (najlepiej na górze):
apply **plugin: 'kotlin-kapt'**

Oraz w sekcji dependencies:

kapt **"androidx.room:room-compiler:\$room_version"**

2) IO TYLKO W TLE #BOWYDAJNOŚĆ

Komunikacja sieciowa i komunikacja z BD w Androidzie jest blokowana jeśli dzieje się na głównym wątku.

Jest na to kilka obejść:

- Async Task: [link](#)
- Korutyny
- RxKotlin: [link](#)

Korutyny w lamerskiej postaci wyglądają tak: odczyt / zapis wyrzucamy do osobnej funkcji (**bardzo sierniężny** przykład):

```
suspend fun readFromDatabase() {  
    withContext(Dispatchers.IO) {  
        val db = Room.databaseBuilder(  
            applicationContext,  
            AppDatabase::class.java, "database-name"  
        ).build()  
        val dao = db.userDao()  
        dao.getAll().forEach{ dao.delete(it) }  
        dao.insertAll(User(1, "A", "B"), User(2, "c", "d"))  
        dao.getAll().forEach{ println(it) }  
    }  
}
```

Wywołanie opakowujemy w funkcję **launch**:

```
GlobalScope.launch {  
    suspend {  
        readFromDatabase()  
    }.invoke()  
}
```

3) PAMIĘTAJCIE, ŻE BAZY DANYCH SĄ TRWAŁE!

Czyli jak posadzicie schemat, to już zmiana musi „wyczyścić” w jakiś sposób dane. To oznacza, że dla tej klasy z dokumentacji (User) dodanie nowej kolumny może być dla Waszej aplikacji niewidoczne. Przynajmniej dopóki nie podniesiecie wersji BD – wg. **wersji którą macie zainstalowaną na komórce**. Żeby to zrobić modyfikujemy adnotacje @Database zmieniając atrybut version.

```
@Database(entities = arrayOf(User::class), version = 2)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

Alternatywnie możemy usunąć aplikację.

6. Zaliczenie

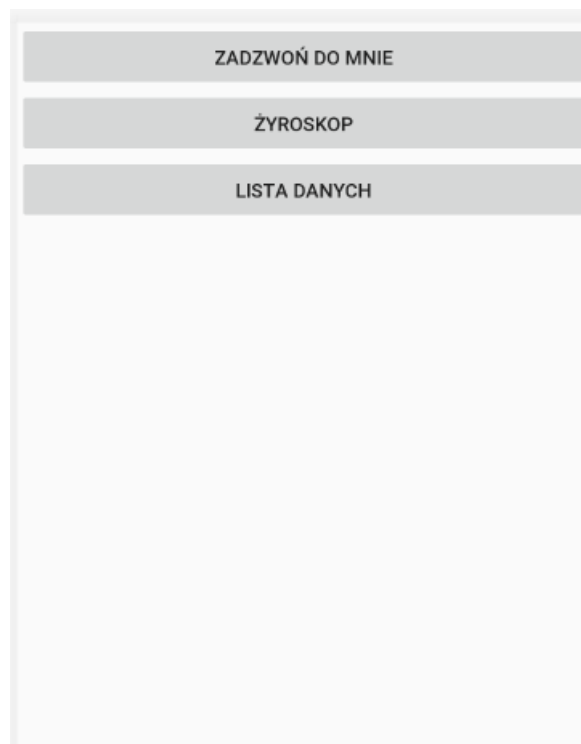
Na zaliczenie potrzebuje dwóch projektów:

1. Projekt uproszczony, dowodzący że cośtam przeczytaliście o pisaniu aplikacji mobilnych, skompilowaliście jakiś szablon i odpaliliście na sprzęcie/emulatorze + sprawozdanie
2. Projekt zaliczeniowy, będący przykładem rzeczywistej aplikacji + sprawozdanie

PROJEKT UPROSZCZONY

Napisać aplikację zawierającą ekran z trzema opcjami:

1. Zadzwoń do mnie
2. Żyroskop
3. Lista danych



1) ZADZWOŃ DO MNIE

Powinna wyświetlać mniej więcej coś takiego:



1	2	3
4	5	6
7	8	9

Telefon

<<

DZWOŃ

Przyciski 1-9 (brakuje 0/+ i # -proszę dorobić) po naciśnięciu powinny zapamiętywać wpisany ciąg i wyświetlać go w polu „Telefon”. Przycisk << usuwa ostatni wprowadzony znak. Przycisk „Dzwoń” dzwoni pod wybrany numer telefonu.

2) ŻYROSKOP

Skorzystać z SurfaceView i utworzyć widok:

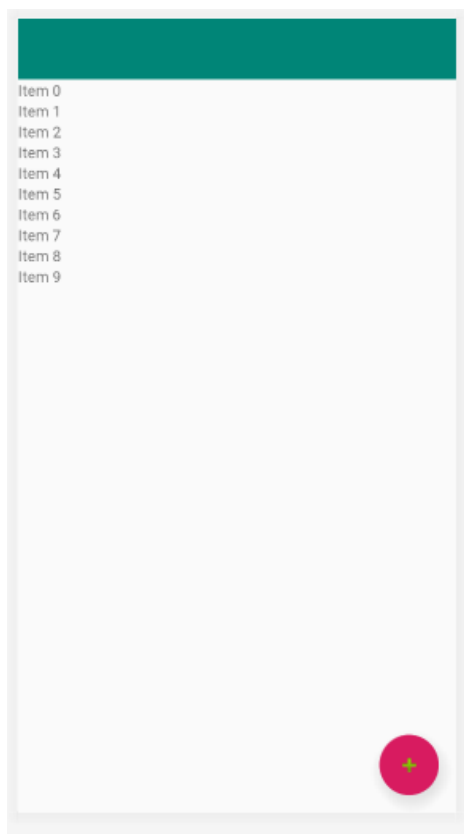


Śledzić położenie koła (albo czegoś innego – bo można rysować sprite'y) – koło powinno poruszać się zgodnie ze wskazaniami żyroskopu i odbić lub blokować na krawędziach ekranu – użyć akcelerometru ([link](#))

```
val sensorManager = getSystemService(Context.SENSOR_SERVICE) as
SensorManager
val sensor: Sensor? =
sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
```

3) LISTA DANYCH

Przygotować widok:



Na każdej pozycji powinny pojawiać się dane użytkowników z przykładu Room-a w formacie „{firstName} {lastName}”.

```
data class User(  
    @PrimaryKey val uid: Int,  
    @ColumnInfo(name = "first_name") val firstName: String?,  
    @ColumnInfo(name = "last_name") val lastName: String?  
)
```

Po naciśnięciu przycisku + powinien uruchomić się widok z dwoma polami edytowalnymi, ich uzupełnienie i zatwierdzenie – w dowolny sposób – powinno zapisać dane do BD i powrócić do poprzedniej aktywności (**finishActivity**).

!!! Skorzystać z RecyclerView