ĆWICZENIE 2

Style i motywy – podstawy.
Menu aplikacji i akcje.

Mariusz Fraś

Cele ćwiczenia

1. Zapoznanie się z motywami i stylami widoków w aplikacji.
2. Opanowanie techniki tworzenia i obsługi różnego rodzaju menu.
3. Poznanie techniki obsługi paska aplikacji i akcji (ActionBar) i menu kontekstowego.
4. Zapoznanie się z techniką utrwalania danych w pamięci Shared Preferences. .
5. Poszerzenie umiejętności pracy z dokumentacją techniczną.

- **Pierwsza część ćwiczenia (Zadanie – część I)** jest instrukcją sposobu realizacji / implementacji omawianych funkcjonalności aplikacji – wprowadzenie do ćwiczenia omawianych technik.
- **Druga część ćwiczenia (Zadanie – część II)** wymaga przygotowania i samodzielnej realizacji. Jest do prezentacji gotowego rozwiązania lub do wykonania na kolejnych zajęciach (zgodnie z poleceniami prowadzącego).

Politechnika Wrocławska
Wrocław 2023

1. Style i motywy (*Themes*).

Style i motywy (*themes*) służą do definiowania wyglądu aplikacji (interfejsu - UI) w separacji od szczegółów implementacji i projektu. Style i motywy mają podobną strukturę (syntaktykę) w postaci kolekcji par **klucz-wartość (*key-value*)**, które **mapują atrybuty do zasobów**.

- ✓ Style specyfikują atrybuty poszczególnych elementów UI / widoków (*views*). Określenie wartości danego atrybutu precyzuje jego cechy (np. kolor, wielkość itp.). Style są specyficzne dla danego widżetu (*widget*), gdyż te mogą mieć różne atrybuty.
- ✓ Motyw to **zbiór nazwanych zasobów**, do których można odwoływać się za pomocą stylów, układów, widżetów itp. **Przypisują semantyczne nazwy zasobom** Androida, dzięki czemu można się do nich później odwoływać – np. *colorPrimary* to nazwa semantyczna danego koloru.
  - Nazwane zasoby to atrybuty motywu. Motywem jest kolekcja <atrybut motywu, zasób>.
  - Atrybuty motywu (w odróżnieniu od atrybutów widoków) nie są właściwościami specyficznymi dla konkretnego typu widoku, ale nazwanymi wskaźnikami do wartości, które mają szerokie zastosowanie w aplikacji. Czyli **motyw specyfikuje konkretne wartości dla nazwanych zasobów**. Np. dla atrybutu *colorPrimary* przypisuje konkretną wartość koloru.
  - **Motyw można przypisać dla przypisać komponentowi, który posiada kontekst** (*context*) – np. aplikacji, aktywności, widokowi. Te elementy otrzymują cechy takie jak określono w motywie. Różne motywy pozwalają różnie definiować UI.
- ✓ Można definiować nie tylko wygląd ale również inne cechy, np. występowanie danego elementu aplikacji – np. obecność paska aktywności (*AppBar*).
- ✓ Style i motywy są deklarowane w pliku zasobów styli, w gałęzi **/res/values**, w pliku **themes.xml** (lub **styles.xml**).

1.1. Style

- ✓ Style definiuje się w pliku **res/values/themes.xml** (lub styles.xml) z użyciem elementu **<style>**:

  <style name="*style_name***" parent="…">…</style>**

  Każdy atrybut stylu definiuje się w elemencie <item>:

  <item name="*atribute_name***">***some_value***</item>**

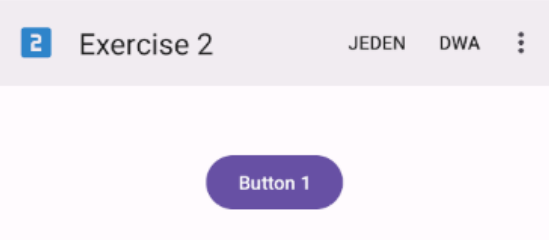
  Wartość tego elementu jest wartością atrybutu. Np. styl określający rozmiar i kolor tekstu:

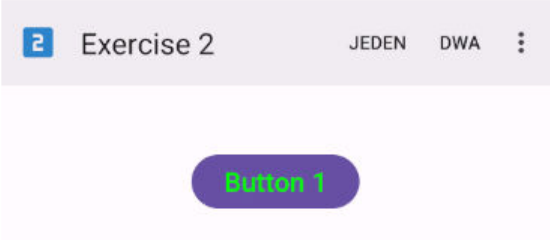
```
<style name="txtblue20" parent="">
    <item name="android:textSize">20sp</item>
    <item name="android:textColor">#0000FF</item>
</style>
```

- ✓ Przypisanie stylu uzyskuje się z użyciem atrybutu widoku **style** odwołując się do stylu (tu dla przycisku:

```
<Button
  android:id="@+id/button1"
  style="@style/txtblue20"
  .../>
```

Przykładowy efekt zastosowania stylu:





- ✓ Style można definiować rozszerzając/modyfikując styl dziedziczony. który specyfikuje się w atrybucie stylu **parent**:

  <style name="*style_name***" parent="***parent_style***">…</style>**

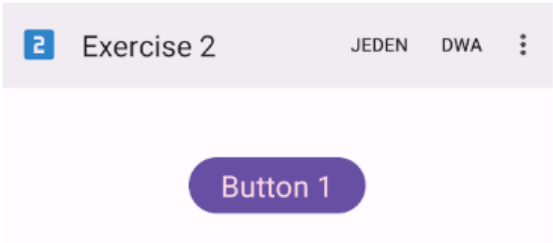
- ✓ Do wartości danego atrybutu używanego motywu (aby go wykorzystać w swoim stylu) można odwoływać się poprzez nazwę zasobu następującskładnią:

  <item name="…">?attr/*themeAttributeName*</item>

Np. dziedziczenie stylu tekstu z biblioteki wsparcia (AppCompat) oraz użycie koloru zdefiniowanego w motywie (tu: kolor kontenerów trzeciego rzędu z motywu Material3):

```
<style name="mytxtstyle" parent="TextAppearance.AppCompat.Large">
  <item name="android:textColor">?attr/colorTertiaryContainer</item>
</style>
```

Przykład po zastosowaniu dla motywu domyślnego Material3:



1.2. Motywy

Motywy określają szerokie spektrum własności aplikacji – zarówno cechy widoczne w UI jak i występowanie elementów UI – np. paska aktywności (tzw. *Application Action Bar*).

*Motywy są tworzone wg. koncepcji **Material Design**, która nakłada wiele reguł na kompozycję motywów (np. dobór kolorów). W ćwiczeniu nie będzie ona szeroko omawiana.*

Podczas tworzenia projektu aplikacji generowany jest domyślny motyw aplikacji – zdefiniowany w pliku zasobów **themes.xml**.

- ✓ Motyw aplikacji dziedziczony jest z predefiniowanego motywu bazowego. W najnowszej wersji Androida jest to jeden z motywów **Material3** (trzecia wersja standardu – tzw. M3) – np. domyślnie dla najprostszego projektu jest to **Theme.Material3.DayNight.NoActionBar**. Poprzednio były to motywy **MaterialComponents** (pierwsza i druga wersja – tzw. M2) – np. **Theme.MaterialComponents.DayNight.DarkActionBar**.

```
<style name="AppTheme" parent="Theme.Material3.DayNight.NoActionBar">
</style>
```

Definiowanie własnego motywu.

Definiowanie własnego motywu pozwala na zbudowanie (*theming*) specyficznego stylu aplikacji. Korzysta się, jak wspomniano, z nadrzędnego predefiniowanego motywu.

✓ Podstawowe sposoby definiowania/modyfikacji motywu są następujące:

1. Ustawienie nowej wartości atrybutu w motywie dziedziczącym.

Np. nowe wartości dla kolorów stosowanych dla elementów interfejsu:

```
<style name="AppTheme" parent="Theme.Material3.Light.NoActionBar">
  <item name="colorPrimary">@color/my_primary</item>
  <item name="colorOnPrimary">@color/my_onPrimary</item>
  ...
</style>
```

Tu: użyty kolor (zasób) zdefiniowany jest w pliku zasobów **res/values/color.xml**.

2. Ustawienie stylu dla atrybutu stylu danego elementu i zdefiniowanie tego nowego stylu dla tego atrybutu.

Np. kolor tła (o wartości @color/my_surfaceVariant) dla stylu paska aplikacji (atrybut **toolbarStyle** motywu i atrybut **android:background** stylu):

```
<style name="AppTheme" parent="Theme.Material3.Light.NoActionBar">
  ...
  <item name="toolbarStyle">@style/AppTheme1.Toolbar</item>
</style>
```

i zdefiniowanie tego stylu:

```
<style name="AppTheme1.Toolbar" parent="Widget.Material3.Toolbar">
    <item name="android:background">@color/my_surfaceVariant</item>
</style>
```

Nowy styl jest często definiowany w oparciu o odpowiedni styl bazowy (tu: stosowanego Toolbar'a w aplikacji).

✓ Stosowane atrybuty w motywach, nazwy, domyślne wartości, specyfikacja komponentów, system kolorów, itp. można znaleźć m.in. na następujących witrynach dokumentacji:

Dla Material3 (najnowsza wersja):

- **https://m3.material.io/components**
- **https://github.com/material-components/material-components-android/tree/master/docs/components**

Dla Material2 (poprzednia wersja):

- **https://m2.material.io/components?platform=android**

Przykładowe atrybuty dla kolorów stosowane w motywach Material3:

- *colorPrimary* – podstawowy kolor stosowany dla różnych komponentów interfejsu,
- *colorOnPrimary* – kolor elementów (np. tekstu) na elementach o kolorze *colorPrimary*,
- *colorSurface* – główny kolor tła dla różnych elementów interfejsu,
- *colorSecondaryContainer* – drugorzędny kolor kontenerów niektórych elementów interfejsu.

Inny przykład:

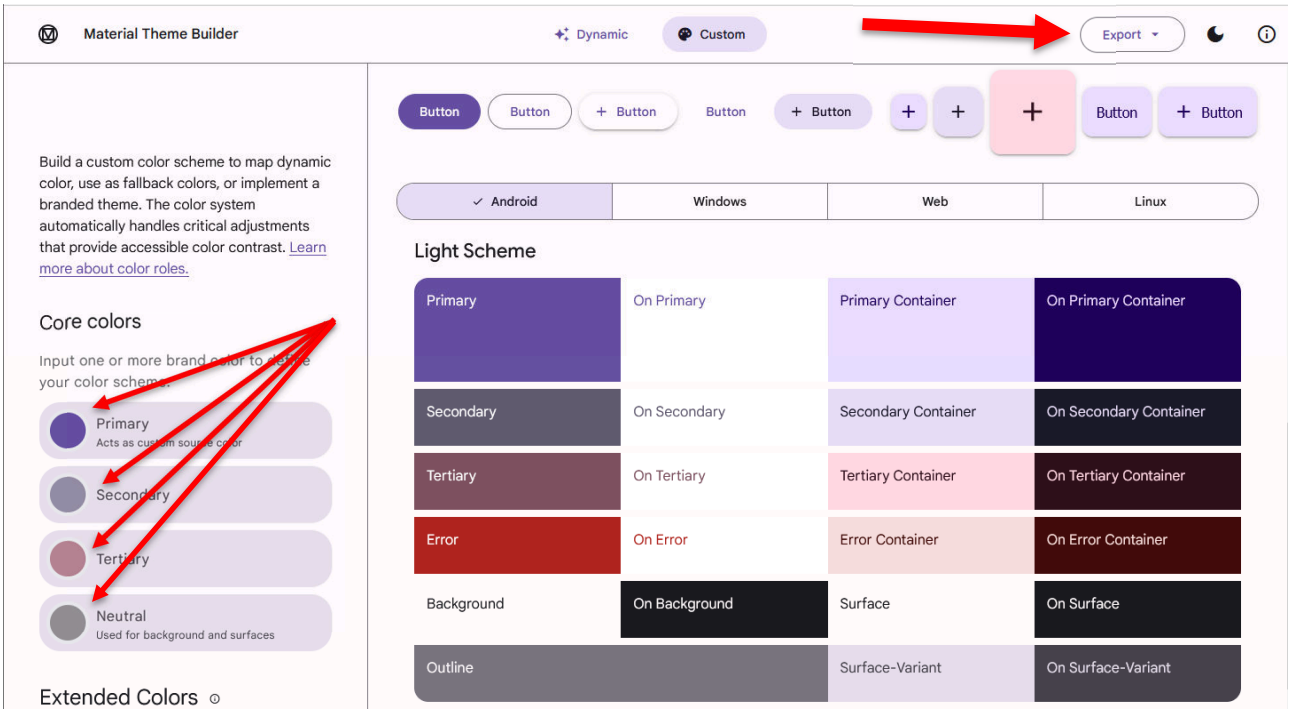
- atrybut *actionBarSize* – wysokość paska aplikacji (tytułu i akcji)

Kreator motywów Material.

Koncepcja Material Design jest wspierana przez kilka narzędzi. **Material Theme Builder** pozwala definiować motywy zgodnie ta koncepcją, zwłaszcza w obszarze kolorystyki aplikacji. Dostępne jest m.in. jako witryna pod adresem:

https://m3.material.io/theme-builder#/custom

Po wybraniu kluczowych kolorów motywu – **Primary**, **Secondary**, **Tertiary** i **Neutral** – narzędzie generuje paletę dla wielu elementów aplikacji i pozwala wyeksportować dane w postaci plików zasobów: **colors.xml** i **themes.xml**. Te dane można zastosować/skopiować w projekcie aplikacji



1.3. Pasek aplikacji *TopAppBar*

Jednym elementów aplikacji jest/był umieszczany u góry pasek tytułowy. Od API 11 pasek tytułu został zastąpiony przez **pasek akcji** (***Action Bar***, zwany czasem mniej precyzyjnie AppBar) który ma możliwość mi.in. udostępniania opcji menu (zwanymi **akcjami**) i dodaje też inne możliwości interakcji. Od Androida 5 (API 21) pasek akcji może być realizowany jako jedna z funkcjonalności **paska narzędzi** (***Toolbar***), który ma większe możliwości konfiguracji. Obecnie użycie **Toolbar** jest zalecanym sposobem realizacji aplikacji z paskiem tytułu i akcji.

- ✓ **Toolbar** definiuje się w pliku zasobów xml układu (layout) aktywności, najczęściej równorzędnie z głównym kontenerem na zawartość ekranu (np. ConstraintLayout), najczęściej powyżej.
- ✓ Z różnych powodów implementacyjnych przydaje się umieszczanie go wewnątrz kontenerów **CoordinatorLayout** i **AppBarLayout** (ewentualnie AppBarLayout bywa pomijany).

Przykładowa struktura układu z paskiem narzędzi:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout
  ...
  android:layout_height="match_parent">
  <com.google.android.material.appbar.AppBarLayout
    ...
    android:layout_height="wrap_content">
    <androidx.appcompat.widget.Toolbar
      ...
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:minHeight="?attr/actionBarSize">
    </androidx.appcompat.widget.Toolbar>
  </com.google.android.material.appbar.AppBarLayout>
  [tu treść główna ekranu – jakiś layout (np. Constraint) ]
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Domyślnie Toolbar ma min. wysokość o wartości określonej w motywie (*?attr/actionBarSize*). Może też mieć ustawiony atrybut koloru. Dla ujednolicenia koloru z innymi elementami można taki atrybut koloru zmienić/usunąć.

- ✓ Powyższe elementy (Toolbar, ApparLayout, …) można dodać za pomocą palety elementów okna *Design* dla układu danej aktywności.
- ✓ Aby toolbar był widoczny i dostępny trzeba w aktywności wywołać metodę:

```
setSupportActionBar(toolbar)
```

gdzie **toolbar** jest wskaźnikiem obiektu Toolbara (uzyskanym metodą findViewByID(…) lub z klasy binding).

1.4. Zmiana motywu w czasie wykonania

***Uwaga**: w Material3 pojawił się mechanizm tzw. **Dynamicznych kolorów** umożliwiający zmianę kolorów motywu w różnych sytuacjach. Tu nie będzie to omawiane.*

- ✓ W aplikacji można programowo wybrać wcześniej zdefiniowany motyw w czasie wykonania aplikacji. Aby ustawić dany motyw (tu o nazwie **MyTheme**), należy użyć metody aktywności:

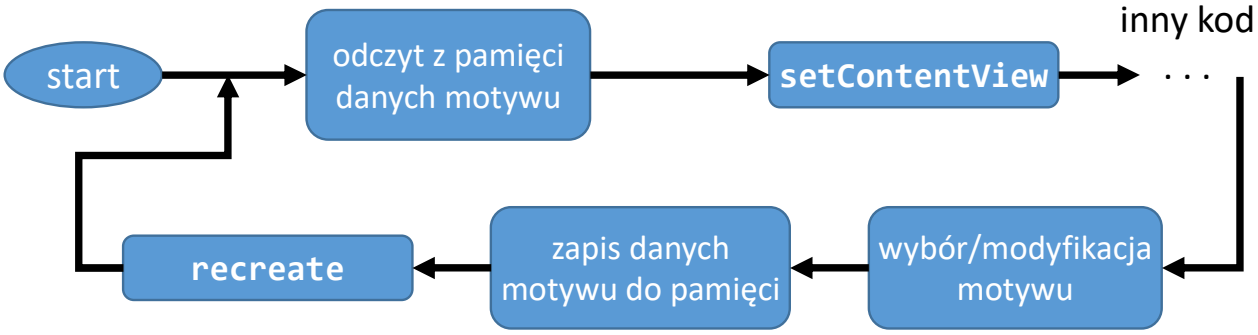
```
setTheme(R.style.MyTheme)
```

- ✓ Aby interfejs uwzględniał dany motyw to musi on być tworzony po wybraniu motywu. Instrukcja musi być wywołana przed utworzeniem instancji jakichkolwiek widoków (czyli przed wywołaniem metody `setContentView`).
- ✓ Aby programowo zastosować dany styl w motywie, można użyć instrukcji:

```
theme.applyStyle(R.style.myStyle,true)
```

- ✓ Aby zmiany były widoczne w czasie wykonywania programu, widoki muszą zostać unieważnione lub cała aktywność (widoki) utworzona ponownie. Np. instrukcją **`recreate()`**.

Schemat implementacji może być następujący:



Typowym miejscem tego typu danych jest pamięć **SharedPreferences**.

✓ W czasie działania można modyfikować motyw jakimś stylem (wcześniej styl musi być zdefiniowany). Modyfikację robi się metodą motywu (który można uzyskać metoda aktywności **getTheme()**):

```
getTheme().applyStyle(R.style.my_style, true)
```

lub inaczej:

```
theme.applyStyle(R.style.my_style, true)
```

Oczywiście jak poprzednio, zmianę można zobaczyć dopiero po ponownym tworzeniu widoków.

1.5. Pamięć trwała Shared Preferences

Aplikacja może mieć jedno lub więcej **repozytoriów** (**identyfikowanych przez nazwę**) danych zapisywanych w postaci **par <klucz_identyfikujący, wartość>**. Podstawowe operacje (tu: dla repozytorium o nazwie **my_prefs**) to:

- odczyt danych (tu: typu Int):

```
val data : SharedPreferences

data =getSharedPreferences("my_prefs", Context.MODE_PRIVATE)
var the_value = data.getInt("key_name", 0)
```

- zapis danych (tu: typu Int) z użyciem obiektu edytora uzyskanego z obiektu repozytorium:

```
val editor = data.edit()

editor.putInt("key_name", the_value)
editor.apply() // lub editor.commit()
```

Pamięć SharedPreferences jest dostępna dla całej aplikacji.

ĆWICZENIE

1) Podstawowe element aplikacji - projekt

- Utwórz projekt Empty Views Activity.
- Główna aktywność będzie dla ustawień preferencji wyglądu (m.in. motywów).
- Dodaj jeszcze dwie aktywności (podobnie jak w poprzednim ćwiczeniu) "lewa" (ActivityLeft) do wykorzystania później, "prawa" (ActivityRight) dla ustawień personalnych (też później).

- Zdefiniuj w ujładzie aktywności strukturę z Toolbar'em (najlepiej jak w opisanym schemacie) i włącz Toolbar.
  Uwaga: CoordinatorLayout nie występuje w palecie okna Design. Można go uzyskać poprzez zaznaczenie głównego kontenera (np. ConstraintLayout) i użycie opcji *Convert view…* (potem można dodać nowy ConstraintLayout).
- Zaimplementuj w **ActivityMain** trzy przyciski dla wyboru motywu dla aplikacji (po kliknięciu przycisku). Oraz przyciski do uruchomienia aktywności lewej i prawej (wykorzystane będą później).

2) Wygenerowanie motywów

- Za pomocą kreatora motywów wygeneruj 3 różne motywy. Dla każdego wybierz inny kolor *Primary*, tak aby motywy różniły się (pozostałe kolory są dobierane automatycznie w stosunku do *Primary*).

***Uwaga**: w ćwiczeniu będziemy używać tylko plików dla motywu dziennego (z gałęzi …/values). Motywy nocne robi się analogicznie.*

- W plikach **colors.xml** i **themes.xml** (który odwołuje się do zasobów w colors.xml) zróżnicuj nazwy atrybutów **name** kolorów (i analogicznie odwołania w themes.xml), tak aby się nie powtarzały (bo dla każdego wygenerowanego motywu są używane takie same nazwy).
  Np. zamień powtarzającą się część nazwy na ciąg specyficzny dla motywu – np. na theme1, theme2 i theme3 odpowiednio dla motywu 1, 2 i 3. Analogicznie ponumeruj nazwy motywów.
- Skopiuj zawartość z węzłów **<resources>** do plików projektu colors.xml i themes.xml.
  Po operacjach powinny być dostępne dodatkowe trzy motywy **AppTheme1**, **AppTheme2** i **AppTheme3**. Np. fragment pliku themes.xml:

```
<resources ...>
  ...
  <style name="AppTheme1" parent="Theme.Material3.Light.NoActionBar">
    <item name="colorPrimary">@color/theme1_light_primary</item>
    ...
  </style>
  ...
```

4) Implementacja zmiany motywu aplikacji

- Zdefiniuj w aktywności dodatkowa funkcję, która będzie zapisywała w pamięci SharedPreferences informację o wybranym motywie – np. jako wartość Int z numerem motywu. Przykładowy kod:

```
private fun setPrefs(themeNum : Int) {
  val data : SharedPreferences = getSharedPreferences("my_prefs",
                                                      Context.MODE_PRIVATE)
  val editor = data.edit()
  editor.putInt("theme_num", themeNum)
  editor.apply()
}
```

  Tu nazwa repozytorium to **my_prefs**, a klucz dla wartości to **theme_num**.

- Zdefiniuj w aktywności dodatkowa funkcję, która będzie odczytywała z pamięci SharedPreferences informację który motyw zastosować i ustawi go metodą **setTheme(…)**. Przykładowy kod:

```
private fun applyTheme() {
    val data = getSharedPreferences("my_prefs", Context.MODE_PRIVATE)
    var themeNum = data.getInt("theme_num", 0)
    when(themeNum) {
        1 -> setTheme(R.style.AppTheme1)
        2 -> setTheme(R.style.AppTheme2)
        3 -> setTheme(R.style.AppTheme3)
        else -> setTheme(R.style.Theme_Exercise2)
    }
}
```

  gdzie **Theme_Exercise2** to nazwa motywu pierwotnego (domyślnego) – zależna od nazwy projektu

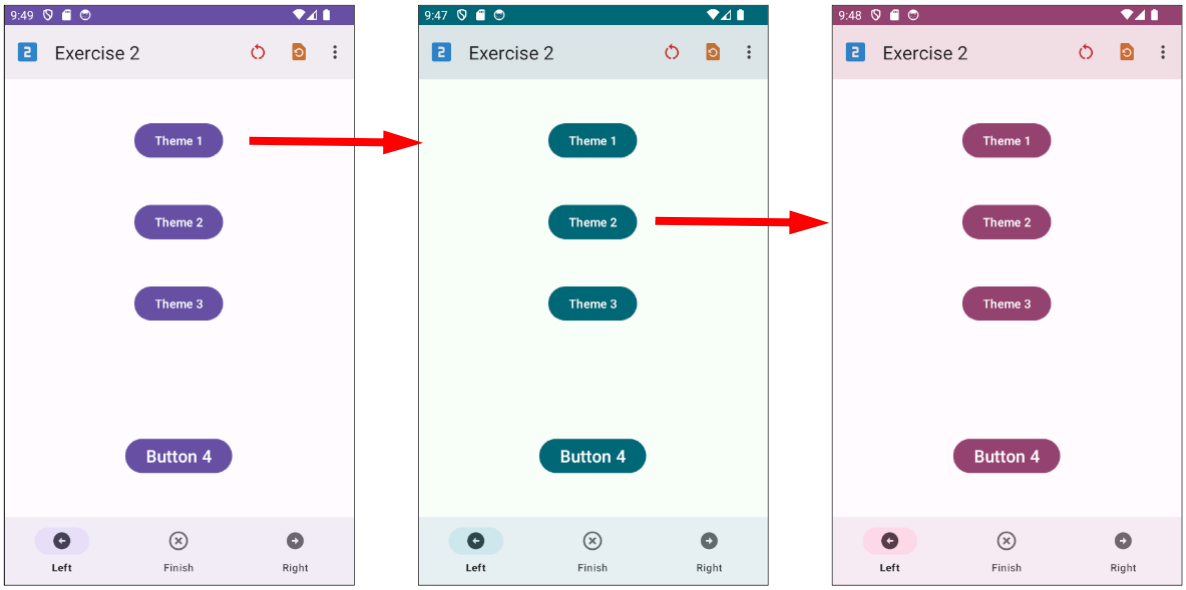
- W aktywności, przed utworzeniem widoków (na początku metody onCreate) wywołaj metodę **applyTheme()** (ustawienie motywu wg danych z pamięci trwałej).
- Dodaj w aktywności obsługę kliknięcia przycisków wyboru motywu. W każdym słuchaczu **OnClickListener** wywołaj metody:

```
setPrefs(X)
recreate()
```

  (gdzie X to numer 1, 2 lub 3, odpowiednio dla przycisku 1, 2 i 3)

  które ustawią odpowiedni motyw i wywołają ponowne zbudowanie elementów UI wg wybranego motywu.

- Uruchom aplikację i przetestuj działanie. Wynik powinien być podobny do poniższego:



W ilustracji: elementy paska aplikacji i dolny BottomNavigationView będą implementowane później.

2. Podstawy tworzenia i obsługi menu aplikacji

W tej części wykonywanie działań aplikacji będzie przeniesione do akcji różnego rodzaju menu aplikacji.

W większości przypadków tworzenie podstawowych opcji w dostępnych w różnego rodzaju menu aktywności jest podobne – nieznacznie różniące się w szczegółach inicjalizacji (np. aktywacji paska akcji).

- ✓ Różne rodzaje menu można tworzyć i obsługiwać na kilka sposobów – z użyciem opisu zasobów menu w XML oraz w kodzie Java/Kotlin. Najbardziej podstawowe rodzaje menu to:
  - standardowe menu rozwijalne (***Overflow Menu*** lub Options menu) dostępne z paska aplikacji,
  - menu kontekstowe.
  - menu trybu akcji kontekstowej (***Contextual Action Mode***),
  - akcje wywoływane z innych elementów (np. przycisk akcji FAB (***Floating Action Button***), ***PopUp Menu***).

Oddzielną kwestią jest realizacja akcji nawigacji z pomocą Android Navigation Component (Navigation Framework) – w tym ćwiczeniu nie jest to omawiane.

2.1. Klasy do wykonywania akcji z menu i elementów nawigacyjnych

- ✓ Częściami składowymi menu (obiekt klasy **`Menu`**) są elementy typu **`MenuItem`** (pozycje menu) i **`SubMenu`** (podmenu). Pozycje wyboru menu mogą być dostępne: z paska akcji (ActionBar a także Toolbar), jako pozycje menu rozwijalnego, z menu kontekstowego i kilku innych komponentów aplikacji.

  Podstawowe klasy dla obiektów wspierających menu i akcje to:

`Menu` – obiekt reprezentujący menu.

`Submenu` – element pozycji menu mogący zawierać podopcje menu.

`MenuItem` – pojedyncza pozycja menu

`ContextMenu` – menu kontekstowe – tworzone dynamicznie na skutek akcji użytkownika.

`PopupMenu` – menu wyskakujące (typu popup).

`ActionBar` – pasek akcji w którym umieszczane jest menu i jego pozycje.

`ToolBar` – kontener "rozszerzający" ActionBar – może realizować ActionBar i ma dodatkowe funkcjonalności.

`ActionMode` – specjalna klasa do realizacji trybu akcji kontekstowych (opis - patrz dalej).

`BottomAppBar` i `BottomNavigation` – komponenty pozwalające realizować akcje na dole ekranu.

`DrawerLayout` and `NavigationView` – komponenty umożliwiające wykonywanie akcji w wysuwanym (zwykle z lewej strony) elemencie interfejsu.

2.2. Tworzenie i obsługa menu standardowego

✓ Aktywność jest powiązana tylko z jednym menu standardowym. Tworzy się je implementując metodę, która jest wywoływana automatycznie w aplikacji tylko jeden raz:

```
fun onCreateOptionsMenu(menu: Menu!): Boolean
```

✓ Elementy menu można tworzyć na dwa sposoby:

- Tworząc obiekt typu **Menu** i wywołując metodę **Menu.add** zwracającą obiekt typu **MenuItem**:.

  ```
  add(groupId: Int, itemId: Int, order: Int, title: CharSequence!): MenuItem!
  ```

  gdzie: **groupId=NONE** jeśli nie mamy grup, **itemId** - unikalny id pozycji w menu, **order** – określa kolejność w menu, **title** – nazwa opcji, napis (ciąg znaków).
- definiując menu w pliku XML

✓ Jeśli menu definiowane jest w pliku XML to w metodzie **onCreateOptionsMenu(…)** musimy utworzyć instancje obiektów menu za pomocą obiektu klasy **MenuInflater**:

```
override fun onCreateOptionsMenu(menu: Menu) {
    getMenuInflater().inflate(R.menu.app_menu, menu)
    return true
}
```

W tym przypadku plik XML umieszczony w katalogu projektu res/menu/ składa się z elementów **<menu>**, **<item>** i **<group>**. Np. dla 3 pozycji menu:

```
<menu ...>
    <item
        android:id="@+id/mi_id_1"
        android:title="@string/mi_title_1"
        android:icon="@drawable/mi_ic_1"
        app:showAsAction="always " />
    <item
        android:id="@+id/ mi_id_2"
        android:title="@string/mi_title_2"
        android:icon="@drawable/mi_ic_2" />
    <item
        android:id="@+id/ mi_id_3"
        android:icon="@drawable/mi_ic_3"
        android:title="Tytul 3 pozycji" />
    <item
```

✓ Elementy menu mogą zawierać wiele innych atrybutów, oprócz wymienionych powyżej. M.in.:

- **showAsAction** – definiuje występowanie pozycji menu bezpośrednio na pasku; możliwe wartości to np. `always`, `ifRoom`, `never`, `withText`.
- **titleCondensed** – definiuje skrócony tekst do wyświetlania.

Uwaga: pomimo definicji w aplikacji, ostatecznie to urządzenie/system decyduje jak będzie wyświetlane menu (np. czy pozycja będzie bezpośrednio na pasku i czy obok ikony będzie tekst).

Element **<group>** pozwala formatować grupy elementów w menu (np. ustawiać ich widzialność, lub specyficzne cechy – przykład będzie dalej.

✓ Obsługę akcji wyboru pozycji w menu można zrobić na kilka sposobów. Zalecanym sposobem jest implementacja metody:

```
fun onOptionsItemSelected(item: MenuItem): Boolean.
```

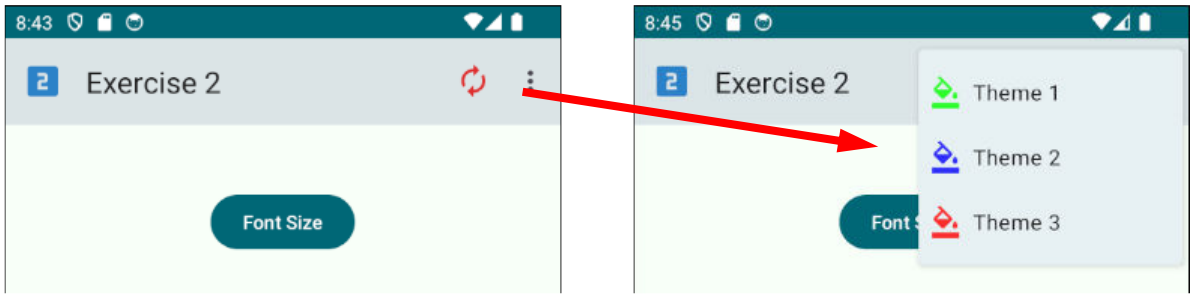
Krótki przykład:

```
override fun onOptionsItemSelected(item: MenuItem) {
    return when (item.itemId) {
        case R.id.item1 -> {
            // here define what to do
            true;
            }
        case R.id.item2 -> {
            ...
            true }
        ...
        else -> super.onContextItemSelected(item)
    }
}
```

✓ Podmenu (*Submenu*) można tworzyć w kodzie Java/Kotlin lub w pliku XML.
W XML, podmenu tworzy się zagnieżdżając w menu element **<menu>**, a w nim umieszczając elementy **<item>** będące pozycjami podmenu.

ĆWICZENIE

- Dodaj w res/menu plik zasobu xml menu – opcja *New → Menu Resource File*
  W pliku zdefiniuj 4 pozycje menu:
  - pierwsza dla wyboru pierwotnego motywu aplikacji,
  - pozostałe trzy dla wyboru zdefiniowanych dodatkowo motywów,
  Dla pierwszej opcji dobierz ikonę i ustaw wyświetlanie opcji bezpośrednio na pasku.
- Zaimplementuj w głównej aktywności metodę **onCreateOptionsMenu(…)** , w której tworzy się menu na podstawie pliku xml menu.
- "Przenieś" zapisywanie wyboru motywu ze słuchaczy przycisków do akcji menu - zaimplementuj w głównej aktywności metodę **onOptionsItemselected(…)**, w której:
  - w instrukcji wyboru wywołuje się metoda **setPrefs(nr)** (nr odpowiedni dla danej opcji menu),
  - na końcu metody wtwołaj funkcję **recreate()** (aby odświerzyć ekran).
- Uruchom aplikację i przetestuj działanie. Wynik powinien być podobny do poniższego:



Działanie aplikacji powinno być takie samo.

2.3. Menu kontekstowe

Menu kontekstowe kojarzy się z obiektami klasy **`View`** (i pochodnych), czyli widokami (jak pole tekstowe, przycisk (buton), element listy itp.). Takie menu jest wywoływane przy dłuższym przyciśnięciu widoku (*longclick*).
Menu kontekstowe tworzy się i obsługuje bardzo podobnie. Różnice są m.in. następujące:

- nie może zawierać podmenu,
- nie wyświetla ikon,
- trzeba je zarejestrować (zazwyczaj w metodzie **`onCreate()`**),

✓ menu kontekstowe tworzy metoda:

```
fun onCreateContextMenu(ctxmenu: ContextMenu,
                        myview: View,
                        ctxmi: ContextMenuInfo),
```

✓ Akcje (wybór pozycji menu) obsługuje się w metodzie:

```
fun onContextItemSelected(item: MenuItem): Boolean.
```

W metodzie `onCreateContextMenu` należy sprawdzić jaki to jest widok (przekazywane jako drugi parametr) i w zależności od tego utworzyć odpowiednie menu (takim samym sposobem jak dla menu standardowego).
W metodzie `onContextItemSelected` akcje obsługuje się tak samo jak dla menu standardowego

✓ Aby włączyć menu kontekstowe dla danego widoku trzeba zarejestrować je dla dla tego widoku metodą aktywności:

```
registerForContextMenu(View widok)
```

2.4. Zaznaczane (*checkable*) pozycje menu

✓ Menu z pozycjami zaznaczanymi – tzw. ***checkable menu*** – definiuje się specyfikując definiując grupę pozycji węzłem **`<group>`** i atrybut **`checkableBehavior`**:

```
<group android:checkableBehavior="xxx">
    <item
     ... />
    ...
</group>
```

Możliwe wartości dla **xxx** to:
- **single** – wybór/zaznaczenie jednej opcji (jak radio buton)
- **all** - wybór/zaznaczenie wielu opcji
- **none** – brak zaznaczania (domyślne).

Należy samemu oprogramować odpowiednie zaznaczanie i odznaczanie pozycji (menu same tego nie zrobi). Zatem w metodzie obsługi akcji wyboru pozycji dla każdej pozycji *checkable* musi znaleźć się odpowiedni kod w rodzaju poniższego:

```
...
R.id.item_id -> {
    if (item.isChecked)
        item.setChecked=false
    else
        item.setChecked=true
    true
...
```

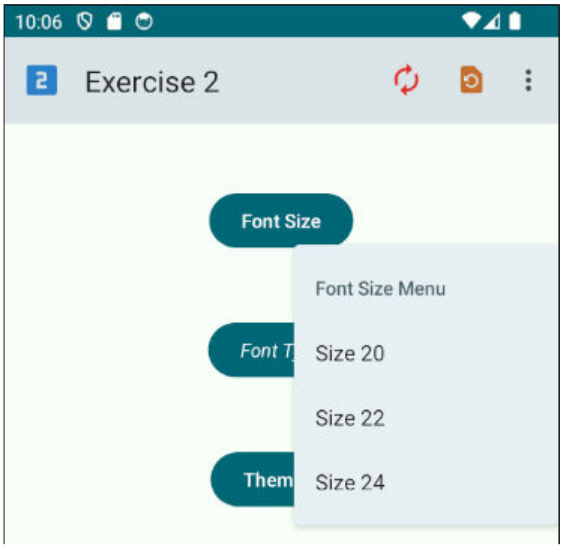
W przypadku **gdy menu jest kontekstowe**, to wybory **należy zapamiętywać w zmiennych** i **odtwarzać zaznaczenia przy ponownym wywołaniu menu**, gdyż menu kontekstowe jest na nowo tworzone za każdym jego wywołaniem.

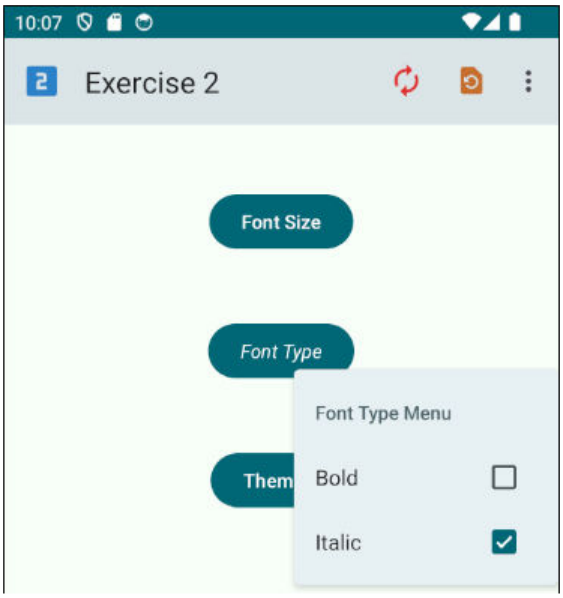
ĆWICZENIE

W ćwiczeniu wykonane będzie menu kontekstowe dla pierwszego i drugiego przycisku. Dla pierwszego ustawienie rozmiaru czcionki (przycisk będzie ilustrował zmiany). Dla drugiego ustawienie typu czcionki (normal, bold, italic).

- Dodaj w res/menu plik zasobu xml menu dwóch menu contekstowych:
  - cm_fontsize.xml – dla ustawień rozmiaru czcionki,
  - cm_fonttype.xml – dla ustawień typu czcionki,

  W plikach zdefiniuj odpowiednie pozycje menu – np. dla rozmiarów 20, 22 i 24 w pierwszy, i bold, italic w drugim.
- Zaimplementuj w aktywności metodę **`onCreateContextMenu(…)`** , w której tworzy się menu na podstawie pliku xml menu.
  - Sprawdzaj dla jakiego widoku wywołano funkcję (jest przekazywany w parametrze).
  - W zależności od tego twórz odpowiednie menu.
- Zaimplementuj w aktywności metodę **`onContextItemselected(…)`**, w której:
  - w instrukcji wyboru ustawia się odpowiedni atrybut dla widoku metodami **view.setTextSize(…)** oraz np.: **view.settypeface(null, …)**
- Uruchom aplikację i przetestuj działanie. Wynik powinien być podobny do poniższego:





2.5. Menu trybu kontekstowego (*Contextual Action Mode*)

Tryb akcji kontekstowej funkcjonalnie odpowiada menu kontekstowemu. Różnica jest taka, że opcje menu kontekstowego tymczasowo zastępują opcje w pasku akcji (ActionBar).
Obiekt **ActionMode** wspiera implementację tego meny kontekstowego. Implementacja menu trybu akcji kontekstowej wymaga utworzenia w odpowiednim słuchaczu obiektu typu **ActionMode** i implementacji w aktywności interfejsu **ActionMode.Callback** – obiektu klasy z metodami do realizacji i obsługi menu trybu akcji kontekstowej:

```
val myAMCallback: ActionMode.Callback = object: ActionMode.Callback() {
   ... // tu kod wywoływanej metody
}
```

Interfejs **ActionMode.Callback** wymaga implementacji 4 metod:

- **onCreateActionMode**(mode: **ActionMode**!, menu: **Menu**!): **Boolean**
  w której tworzymy menu identycznie jak w metodzie **onCreateOptionsMenu(…)**.
- **onActionItemClicked**(mode: **ActionMode**!, menu: **Menu**!): **Boolean**
  w której obsługujemy zdarzenie wyboru opcji menu tak jak dla metody **onOptionsItemSelected(item: Item**. Jedynie dodatkowo przed powrotem z metody (przed return) należy zamknąć tymczasowe menu metodą:
  mode.finish()
- **onPrepareActionMode**(mode: **ActionMode**!, menu: **Menu**!): **Boolean**
  w której możemy ewentualnie wykonać dodatkowe działania przy uruchamianiu trybu akcji kontekstowej – jeśli nic nie robimy to metoda pozostaje pusta.
- **onDestroyActionMode**(mode: **ActionMod**!)
  {
  myAM = **null**;
  }

– metoda wywoływana przy wyjściu z trybu akcji kontekstowej – usuwanie obiektu `ActionMode`.

– tu: `myAM` – obiekt typu `ActionMode` zadeklarowany w aktywności.

Menu trybu akcji kontekstowej tworzy się i aktywuje w słuchaczu jakiegoś zdarzenia – np. w słuchaczu dla długiego przyciśnięcia jakiegoś widoku (komponentu/kontrolki na ekranie). Np. dla kontrolki `view` (np. `TextView`) schemat aktywacji w słuchaczu jest następujący:

```
view!!.setOnLongClickListener(OnLongClickListener { view ->
    // Called when the user long-clicks on someView
    if (myAM != null) {
        return@OnLongClickListener false
    }
    // Start the CAB using the ActionMode.Callback defined above
    myAM = startActionMode(myAMCallback)
    //.maybe some more job
    true
})
```

gdzie: **`myAM`** – to zadeklarowany w aktywności obiekt typu ActionMode,

`myAMCallback` – to zdefiniowany w aktywności obiekt klasy implementującej interfejs **`ActionMode.Callback`**.

2.6. *Floating Action Button* (FAB)

Pływający przycisk akcji (**FAB**) to przycisk (zwykle w kształcie koła z ikoną pośrodku), który uruchamia akcję podstawową. FAB należy dodać w układzie – węzeł:

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/myFAB"
    app:srcCompat="@android:drawable/ic_input_add"
    .../>
```

Atrybut **app:srcCompat** pozwala zdefiniować widok (ikonę).

Aby wykonać dowolną akcję za pomocą FAB, po prostu definiuje się słuchacza dla FAB:

```
val fab = findViewById<FloatingActionButton>(R.id.myFAB)
fab.setOnClickListener { view ->
    //do your job here
}
```

ĆWICZENIE

Ta część może być jeszcze uzupełniona.

3. Uwagi dodatkowe

a. Gdy używa się **`AppCompatActivity`** (co jest regułą) pamiętać, że:
- atrybut **`showAsAction`** (definiujący występowanie pozycji menu na pasku) powinien być specyfikowany nie w przestrzeni nazw *android*, tylko przestrzeni nazw aplikacji (domyślnie **`app:showAsAction="…"`**) **.**

b. W androidzie motywy (***Theme***) określające różne aspekty wizualne i cechy aktywności. co jakiś czas zmieniają się. Mają one również wpływ na wygląd i zachowanie się menu (i paska akcji).
Od API 11 domyślne motywy aplikacji zawierają wbudowany `ActionBar` (z wyjątkiem tych z nazwą kończącą się na .NoActionBar).
`ActionBar` aktywowany w `ToolBar`'ze i wbudowany `ActionBar` aktywności wykluczają się!

c. Aby standardowe menu aplikacji (*Options Menu*) współdziałało z *Navigation Drawer* (szuflada nawigacji) metoda **onOptionsItemSelected(…)** obsługująca wybrane opcje **musi na koniec zwracać false**. To pozwala przechwytywać następnie zdarzenia szufladzie nawigacji – inaczej kliknięcia ikony szuflady nie dadzą efektu.

4. Zadanie – część II.

A. Wykonać na zajęciach elementy aplikacji wg wymagań i wskazówek przekazanych przez prowadzącego.

lub

B. Program przygotowawczy do samodzielnego wykonania "w domu":
Napisać aplikację składającą się z kilku aktywności, spełniającą wymagania wyspecyfikowane przez prowadzącego.

Uwaga: na zaliczenie wymagana jest znajomość kodu i umiejętność modyfikacji aplikacji dla osiągania żądanych zmian.