ĆWICZENIE 3

Fragmenty i nawigacja

Mariusz Fraś

Cele ćwiczenia

1. Zapoznanie się z tworzeniem aplikacji z wykorzystaniem fragmentów.
2. Nabycie umiejętności operowania fragmentami.
3. Poznanie sposobu wymiany danych pomiędzy fragmentami oraz czynności z wykorzystaniem słuchaczy.
4. Zapoznanie się z techniką nawigacji z wykorzystaniem *Navigation Framework*.

- **Pierwsza część ćwiczenia (Zadanie – część I)** jest instrukcją sposobu realizacji, implementacji funkcjonalności, do wykonania według podanych informacji (oraz ewentualnych poleceń prowadzącego zajęcia laboratoryjne).
- **Druga część ćwiczenia (Zadanie – część II)** wymaga przygotowania i samodzielnej realizacji. Jest do prezentacji lub do wykonania na kolejnych zajęciach (wg polecenia prowadzącego).

Politechnika Wrocławska, Instytut Informatyki i Telekomunikacji
Wrocław 2023

ĆWICZENIE – część I – Aplikacja z fragmentami i nawigacją.

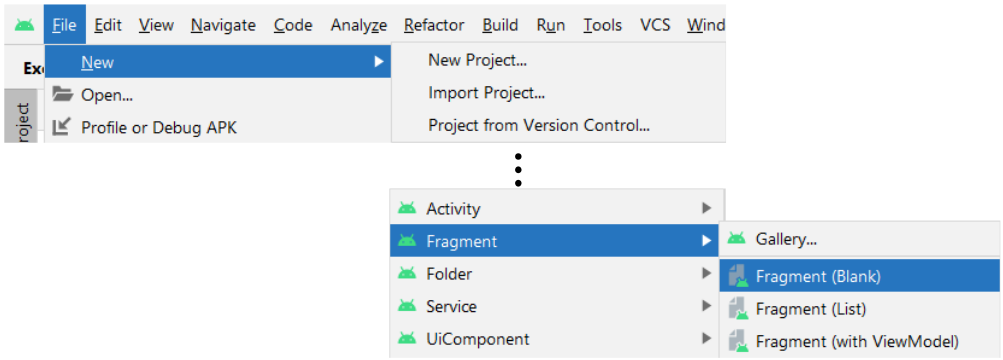
1. Implementacja fragmentów

W tej części zaimplementowana zostanie aplikacja zawierająca w jednej aktywności jeden fragment dodany statycznie oraz dwa fragmenty zagnieżdżone w pierwszym (child) dodawane dynamicznie. Ta część aplikacji stosuje schemat bezpośredniego zarządzania fragmentami. **Nowe podejście** z urzyciem frameworku nawigacji zostanie przedstawione w kolejnych rozdziałach.

- Utwórz w Android Studio nowy projekt aplikacji **Empty Views Activity**.

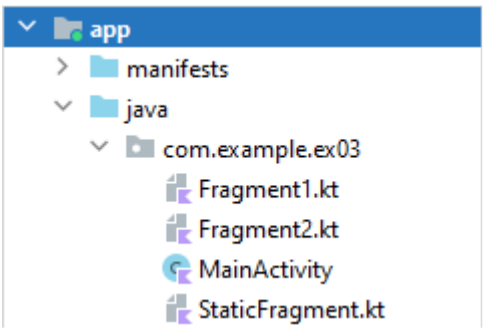
1.1. Dodawanie fragmentów

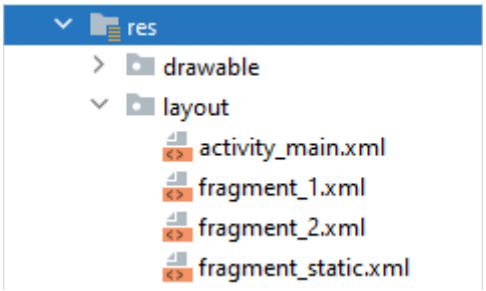
✓ Fragmenty są implementowane (a ściślej powinny być) jako oddzielne komponenty (osobne klasy) i mają własne pliku zasobów układu. Można je utworzyć za pomocą menu głównego AS lub opcji menu kontekstowego okna eksploratora projektu: *New…→Fragment→Fragment…*,



Automatycznie wygenerowany kod klasy fragmentu zależy od wybranego typu fragmentu. W ćwiczeniu zawsze zaczynamy od **Blank Fragment**.

- Dodaj trzy puste fragmenty do projektu (Fragment (Blank)): **FragmentCenter**, **Fragment1**, i **Fragment2**



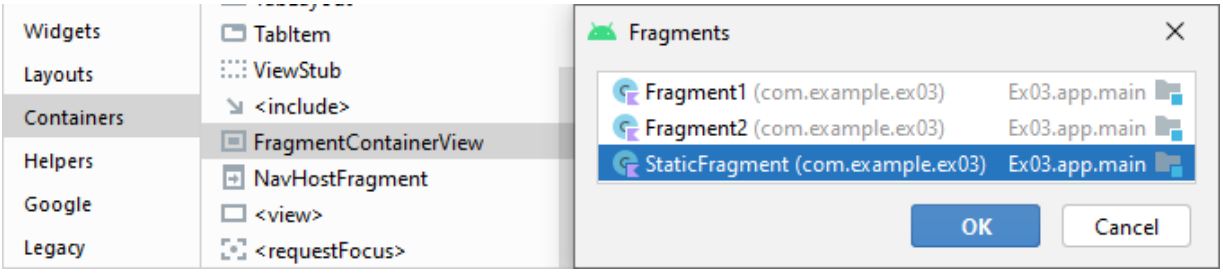


- Aby uzyskać lepszą kontrolę nad pozycjonowaniem elementów, możesz zmienić układy fragmentów z **`FrameLayout`** na np. **`LinearLayout`** lub **`ConstraintLayout`**.
- Aby rozróżnić widoki, wstaw do układów Fragment1 i Fragment2 pole tekstowe z tekstem informującym, który fragment jest wyświetlany oraz różne kontrolki – np. **`RatingBar`** do fragmentu 1 i **`SeekBar`** do fragmentu 2. Dodaj także przyciski (będą użyte później).

- W układzie **FragmentCenter** wstaw **`RadioGroup`** i dwa przyciski wyboru **`RadioButton`**. Przypisz im odpowiednie teksty (np. Opcja 1 i Opcja 2).
- Dla przejrzystości wyśrodkuj i/lub zmień rozmiar poszczególnych elementów/widoków.
- Upewnij się, że wszystkie widoki mają przypisane identyfikatory (**!**)
  (jest to ważne dla przywracania widoków podczas ich ponownego rysowania).

1.2. Układy (Layouts) hostujące fragmenty

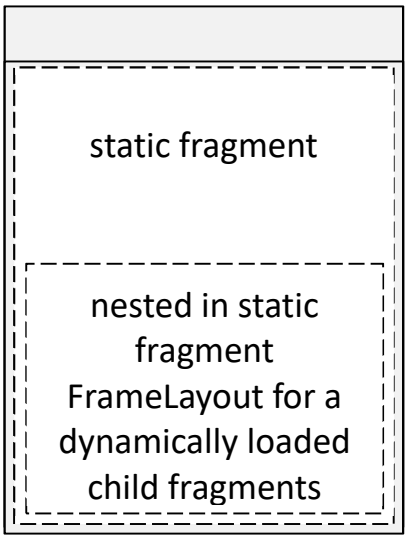
✓ Aby statycznie wstawić fragment do układu aktywności, można użyć *XML Resource Editor*. Dla wstawianego fragmentu można użyć kontenera **`Fragment`** lub **`FragmentContainerView`**. Przy dołączaniu edytor pyta o podaną, zdefiniowaną klasę fragmentu. Taki kontener ma określony statycznie fragment w atrybucie **android:name**.



***Uwaga**: w najnowszych wersjach AS kontener **`Fragment`** nie jest dostępny w palecie komponentów, jednakże nadal można go wstawić w trybie Code (kodowanie w pliku XML).*

✓ Aby dynamicznie (programowo w czasie wykonywania) dodać/wyświetlić fragment do układu aktywności lub do fragmentu nadrzędnego, układ aktywności (lub układ fragmentu, jeśli dodawany jest fragment potomny) musi zawierać odpowiedni kontener na ten fragment: **`FrameLayout`** lub **`FragmentContainerView`** bez atrybutu **android:name**. Następnie należy użyć Menedżera fragmentów i transakcji do dołączenia fragmentu dynamicznie, lub frameworka nawigacji (ten drugi sposób będzie przedstawiony później).

- Zdefiniuj fragmenty i skonfiguruj układy fragmentów i aktywności, aby zbudować następujący ekran aplikacji:

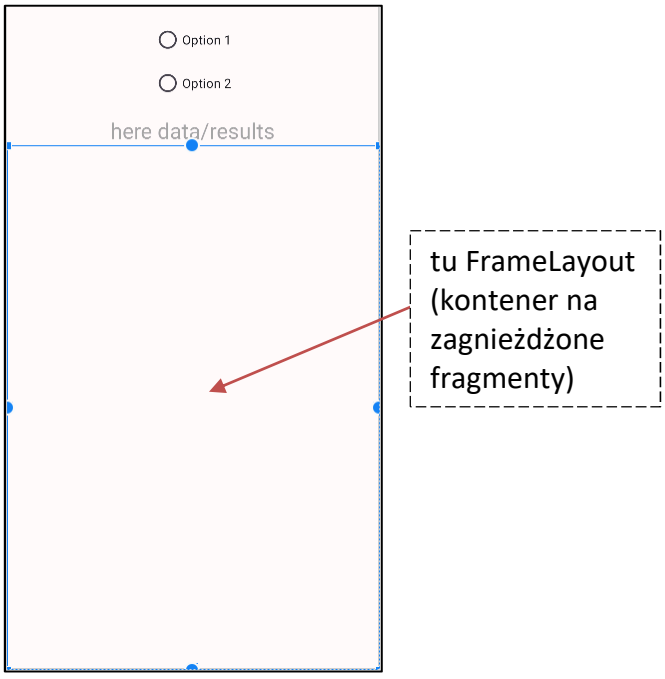


- Umieść **FragmentCenter** statycznie w głównej aktywności za pomocą Edytora zasobów XML – przeciągnij z palety do układu działania komponent **`FragmentContainerView`**.
- Ustaw poprawnie atrybuty **layout_height** i **layout_width**.

```
<androidx.constraintlayout.widget.ConstraintLayout ...>
    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/sfcontainer"
        android:name="com.example.ex03.FragmentCenter"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Można też dodać atrybut `tools:layout="@layout/fragment_center"` (odniesienie do układu fragment_center.xml), aby zobaczyć właściwy widok w edytorze układu w trybie *Design*.

- W układzie fragmentu (tutaj fragment_center.xml) oprócz grupy `RadioGroup`, dodaj poniżej `TextView` do wyświetlania później otrzymanych danych z innych fragmentów.
- Dodaj poniżej kontener **`FrameLayout`** dla zagnieżdżonych, dynamicznie dodawanych fragmentów.
- Ustaw poprawnie atrybuty **layout_height** i **layout_width** (oraz opcjonalnie **gravity**) dla wszystkich elementów. W razie potrzeby odpowiednio wycentruj i/lub zmień wielkość elementów. Układ fragmentu powinien wyglądać podobnie do poniższego:



  - Ustaw unikalne identyfikatory dla wszystkich widoków.
  - W razie potrzeby dodaj atrybut `Tools:layout`, aby poprawnie wyświetlić widoki układów w edytorze układu.

1.3. Implementacja kodu dotycząca fragmentów

Uwaga: *Tutaj prezentowana jest implementacja operacji we fragmencie. Można to również implementować w aktywności poprzez odpowiedniego słuchacza. Wtedy kod różni się w szczegółach (np. używa się* ***suportFragmentManager*** *zamiast* ***childFragmentManager****, itp.). Taka implementacja byłaby wskazana, gdyby dynamicznie dołączany fragment nie był zagnieżdżony (ale był bezpośrednio w aktywności – jak w poprzedniej wersji ćwiczenia).*

- ✓ Automatycznie wygenerowany kod klasy fragmentu zawiera przynajmniej metody **`onCreate(…)`** i **`onCreateView(…)`** oraz metodę do tworzenia instancji fragmentu **`newInstance(…)`** w obiekcie towarzyszącym (***companion object***), który jest swego rodzaju odpowiednikiem metody singleton. Metoda zawiera przykładowy kod odbioru przekazanych do fragmentu argumentów (danych).
  - W plikach Java/Kotlin fragmentów 1 i 2 sprawdź, czy metoda **`onCreateView(…)`** tworzy widoki ekranu fragmentu metodą **`inflate(…)`**.
- ✓ Aby fragment mógł działać, musi być powiązany z aktywnością (statycznie lub dynamicznie). Wszystkie widoki należą i są dostępne poprzez aktywność. Interoperacyjność pomiędzy fragmentem a aktywnością jest wspierana za pomocą:
  - Metoda dostępna we fragmencie, która zwraca referencję do aktywności, z którą fragment jest aktualnie powiązany:
    `public final getActivity() : FragmentActivity`
    lub lepiej (bo sprawdza dostępność aktywności):
    `public final requireActivity() : FragmentActivity`
  - Metoda dostępna w aktywności, która znajduje fragment powiązany z tą aktywnością:
    - identyfikowane poprzez podany identyfikator (zasobu XML lub kontenera):
      `public findFragmentById(int id) : Fragment`
    - identyfikowane poprzez podaną etykietę (*tag*):
      `public findFragmentByTag(String tag) : Fragment`

  Używanie etykiet pozwala zachować stan widoku, gdy fragment znika (jest odłączany) z ekranu.
- ✓ Obiekt klasy **`FragmentTransaction`** dostępny jest za pomocą metody **`beginTransaction()`** klasy **`SupportFragmentManager`** (lub **`FragmentManager`** w przypadku implementacji bez wsparcia starszych API). Za pomocą transakcji możliwe jest wykonywanie operacji:

  `instantiate(…)` – tworzy instancję fragmentu na podstawie podanej nazwy klasy,
  `add(…)` – dodaje fragment do aktywności
  `remove(…)` – usuwa fragment z aktywności
  `attach(…)` – dołacza (bind) fragment do aktywności
  `detach(…)` – odłacza fragment od aktywności bez usuwania
  `replace(…)` = **`remove`** (aktualny) oraz **`add`** (nowy) fragment.

Przykładowy kod:

```
val myFragMan:FragmentManager = getSupportFragmentManager()
val myTrans:FragmentTransaction = myFragMan.beginTransaction()
myTrans.operation(...)     //here is some of mentioned operation(s)
myTrans.commit()
```

Krótsza forma z wykorzystaniem składni dostępu do właściwości (*property access syntax*):

```
val myTrans:FragmentTransaction = supportFragmentManager.beginTransaction()
myTrans.operation(...)
myTrans.commit()
```

Implementacja kodu fragment (obiekty i operacje dla zmiany fragmentów).

- W klasie aktywności zadeklaruj zmienne globalne dla dynamicznie dołączanych fragmentów i dla transakcji służącej do manipulacji fragmentami:

```
lateinit var frag1: Fragment1
lateinit var frag2: Fragment2
lateinit var myTrans: FragmentTransaction
```

- W metodzie **onCreate(…)** utwórz obiekty fragmentu 1 i 2, utwórz obiekt transakcji i za jego pomocą dodaj te fragmenty do działania i odłącz je.

```
frag1 = Fragment1.newInstance(...)
frag2 = Fragment2.newInstance(...)
myTrans = childFragmentManager.beginTransaction()     //Uwaga: tu child
```

- Dodaj implementację słuchacza **onCheckedChangeListener** do obsługi wyboru opcji:
  - Dodaj dyrektywę dla klasy **FragmentCenter** :

    class ..., RadioGroup.OnCheckedChangeListener {
  - Dodaj wymaganą metodę **onCheckedChanged(...)** i sprawdź wybraną opcję w bloku **when** (czyli identyfikator wybranego obiektu (przycisk radiowy)), i w zależności od wyboru wygeneruj odpowiednie zdarzenie - czyli wykonaj odpowiednią transakcję.
  - W bloku **when(…) {…}** instrukcje:

    `myTrans.replace(R.id.dfcontainer, fragX)`

    gdzie:

    R.id.*dfcontainer* to identyfikator **FrameLayout** (kontener na dynamicznie umieszczane fragmenty w pliku XML układu aktywności) **android:id="@+id/dfcontainer"** ,

    fragX – to fragment, który ma zostać wyświetlony – użyj odpowiednich nazw zmiennych.
  - Użyj selekcji w **when(…)** dla wyboru 1 i wyboru 2 (fragment 1 lub 2).
  - Na koniec (na końcu metody) zatwierdzenie transakcji:

    `myTrans.commit();`

  Ostateczny kod powinien być podobny do:

```
when (checkedId) {
    R.id.sf_radioButton1 -> myTrans.replace(...)
    R.id.sf_radioButton2 -> myTrans.replace(...)
}
myTrans.commit()
```

- Dodaj metodę **`onViewCreated(…)`** i wewnątrz zaimplementuj:
  - wywołanie metody nadklasy (powinna zostać dodana automatycznie),
  - ustawienie grupy RadioButton fragmentu jako słuchacza dla wyboru opcji:

```
(requireActivity().findViewById(R.id.sfrag_options) as RadioGroup)
                  .setOnCheckedChangeListener(this)
```

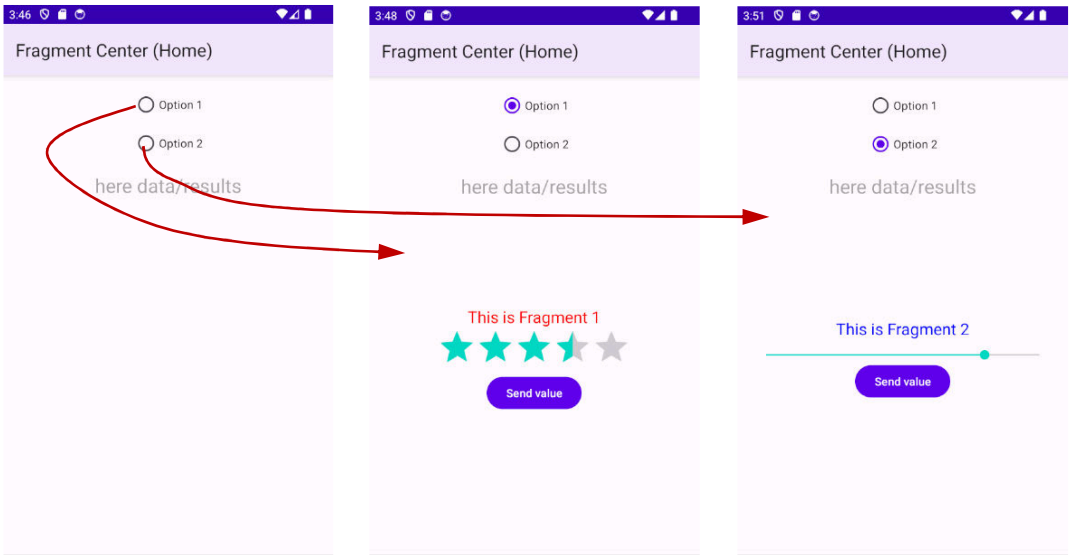
  Jeśli grupa przycisków wyboru nie ma określonego identyfikatora (tu: **sfrag_options**) ustaw go w pliku xml.

Sprawdzenie i test działania:

- Przejrzyj zawartość projektu.
- Zwróć uwagę na elementy kodu używane do implementacji interakcji pomiędzy fragmentami i aktywnością oraz na to, jak umieszczać i wyświetlać fragmenty w aktywnosci.
- Zuważ, że we fragmencie należy zastosować metodę **onViewCreated (...)**.
- Uruchom aplikację na podłączonym smartfonie i sprawdź działanie, ustawiając niektóre wartości kontrolek i fragmenty przełączania (wybierając opcje).
- Zobacz, co się stanie, gdy zmienisz orientację ekranu (obróć urządzenie o 90° - czyli ***configuration change***).

Zaobserwowany problem (brak zachowania ustawień) zostanie rozwiązany w dalszej części ćwicz.

Aplikacja powinna wyglądać podobnie do poniższego (ekran początkowy i ekrany po dokonaniu wyboru):



1.4. Modyfikacje dla zachowania stanu po obróceniu ekranu

✓ W wyniku tzw. **zmiany konfiguracji** (np. spowodowanej obrotem urządzenia) ponownie tworzona jest aktywność i fragmenty (zwłaszcza widoki). Aby przywrócić ustawienia widoków (np. ustawienie suwaka (*Slider*)) można przekazać dane do tworzonej instancji obiektu lub uniknąć usuwania widoków z pamięci i skorzystać z parametru **savedInstanceState**.

✓ Wiele metod cyklu życia aktywności i fragmentów pobiera parametr **savedInstanceState**. Przy pierwszym uruchomieniu aktywności/fragmentu ma on wartość **null**. Nie jest null, gdy

odtwarzane są fragmenty/aktywności. Dodatkowo fragmenty można oznaczyć **etykietą** znakową (***tag***). Użycie takiego identyfikatora pomaga zachować obiekt fragmentu w pamięci. Stany widoków są odtwarzane jeśli mają nadane identyfikatory.

Kod wspierający rotację urządzenia i prawidłowe wyświetlanie widoków fragmentów

Wspomniany problem przy rotacji można różnie rozwiązać. Tutaj używamy etykiet oraz metod **attach** i **detach** zamiast **replace**, aby zachować obiekty i stany obiektów we fragmentach podczas obracania urządzenia.

- Zdefiniuj w klasie fragmentu statyczne zmienne globalne dla etykiet obu fragmentów:

```
private val TAG_F1 = "Fragment1"
private val TAG_F2 = "Fragment2"
```

- W metodzie fragmentu **onCreate()** w bloku warunkowym **if** załącz kod tworzenia instancji fragmentów, oraz kod dodawania fragmentu do układu, oraz (początkowo) odłączenia ich:

```
if (savedInstanceState == null) {
  frag1 = Fragment1.newInstance()
  frag2 = Fragment2.newInstance()
  myTrans = supportFragmentManager.beginTransaction()
  myTrans!!.add(R.id. dfcontainer, frag1, this.TAG_F1)
  myTrans!!.detach(frag1!!)
  myTrans!!.add(R.id. dfcontainer, frag2, this.TAG_F2)
  myTrans!!.detach(frag2!!)
  myTrans!!.commit()
}
```

  Tutaj zamiast metod: **add(containerID:int, fragment:Fragment)** stosowane są metody **add(containerID:int, fragment:Fragment, tag:String)** przekazujące odpowiedni ***tag***.

- W metodzie fragment **onViewCreated** a dodaj w bloku warunkowym odzyskanie (np. po obrocie) referencji do obiektów fragmentu:

```
if (savedInstanceState != null) {
  frag1 = childFragmentManager.findFragmentByTag(this.TAG_F1) as Fragment1
  frag2 = childFragmentManager.findFragmentByTag(this.TAG_F2) as Fragment2
}
```

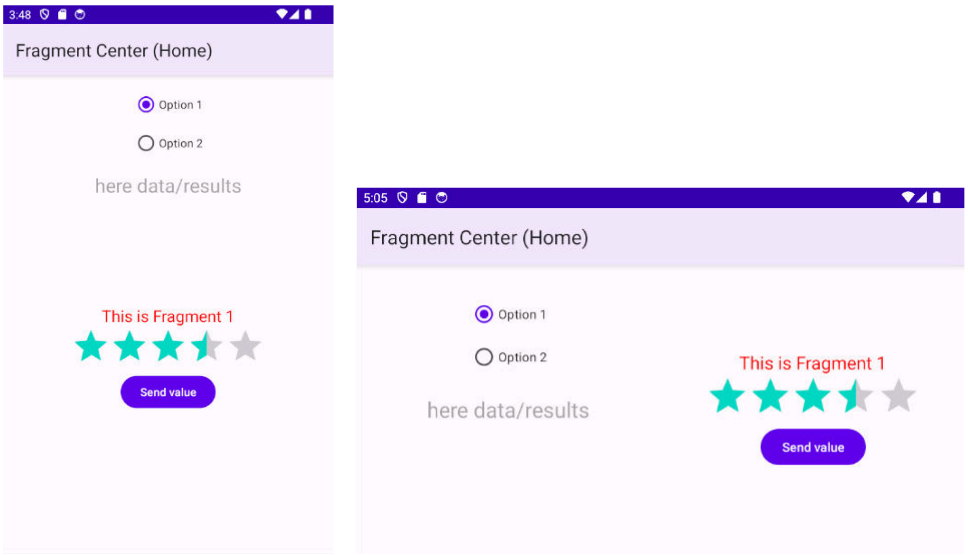
  Uwaga*: dodanie tu dodatkowej logiki pozwoliłoby również implementować aplikację z użyciem metody* ***replace****. Generalnie* ***replace*** *warto używać, gdy zazwyczaj od nowa tworzymy obiekty fragmentów, których może być wiele. Metody* ***attach*** *i* ***detach****, gdy chcemy odzyskać obiekty zachowane w pamięci.*

- W metodzie **onCheckedChanged(…)** , w bloku **when**, zamiast **replace(…)** użyj metod **attach(…)** i **detach(…)** :

```
-> {
    myTrans.detach(fragY!!);
    myTrans.attach(fragX!!);
   )
```

  gdzie *fragX* jest fragmentem do wyświetlenia, *fragY* jest fragmentem, który ma zniknąć – użyj odpowiednich nazw zmiennych.

- Uruchom aplikację I sprawdź działanie. Widoki powinny zostać zachowane również po obrocie. Dodaj alternatywny widok dla położenia horyzontalnego.



1.5. Wymiana danych miedzy fragmentami – słuchacze komunikatów

✓ Wymiana danych pomiędzy fragmentami możliwa jest poprzez **rejestrację słuchaczy** dla określonych komunikatów (**rozróżnialnych po identyfikatorze** komunikatu) przy wykorzystaniu odpowiedniego menedżera fragmentów oraz z drugiej strony przesłanie danych do menedżera fragmentów (następnie są one przekazywane do słuchacza) z odpowiednim identyfikatorem.

✓ Wiadomości zawierają dane w formie Bundle – należy umieścić/pobrać dane do/z obiektu Bundle.

✓ Stosowane metody menedżera fragmentów to:

- aby zarejestrować słuchacza:

```
setFragmentResultListener(requestId : String,
                          owner : LifecycleOwner,
                          listener : FragmentResultListener)
```

LifecycleOwner reprezentuje cykl życia widoku fragmentu; w wielu przypadkach można użyć tu metody fragmentu: **getViewLifecycleOwner()**,

- aby wysłać dane (komunikat):

```
setFragmentResult(requestId : String,
                  result : Bundle)
```

Tutaj w aplikacji pobieramy dane z fragmentów potomnych (zagnieżdżonych) i wyświetlamy dane we fragmencie nadrzędnym.

- W klasie **`FragmentCenter`** w metodzie **`onViewCreated(…)`** należy wpisać kod ustawiający słuchacza dla wiadomości o podanym ID (tutaj ID to **msgfromchild**), a w słuchaczu pobierać dane.

  Tu wysyłamy/odbieramy prosty ciąg znaków (klucz w pakiecie to **msg1**) i ustawiamy ten ciąg w widoku tekstowym (wystarczy wyświetlić ciąg). Ogólnie można wysyłać złożone dane.

```
childFragmentManager.setFragmentResultListener("msgfromchild", viewLifecycleOwner)
{  key, bundle ->
    val result = bundle.getString("msg1")
    (requireActivity().findViewById<View>(R.id.tv_cfrag) as TextView)
                      .setText(result)
}
```

Zauważ, że:

– **childFragmentManager** służy do pobierania danych z fragmentu potomnego,

– **viewLifecycleOwner** to składnia dostępu równoważna dla metody **getLifecycleOwner()**,

– słuchacz jest definiowany za pomocą wyrażenia lambda.

- , wyślij dane (tekst zawierający ustawioną wartość paska oceny) w słuchaczu przycisku zawartego we fragmencie:
- W metodzie **onViewCreated(…)** w **Fragment1** zdefinuj słuchacza przycisku (*buton*) a w nim wyślij dane (tekst z ustawioną wartością paska oceny) – wysyłany po kliknięciu przycisku:

```
view.findViewById<View>(R.id.button_f1).setOnClickListener { _ ->
  var value = view.findViewById<RatingBar>(R.id.rbar_f1).rating
  parentFragmentManager.setFragmentResult("msgfromchild",
                        bundleOf("msg1" to ("value from child = " + value)))
}
```

Zauważ, że:

– **parentFragmentManager** jest użyty do komunikacji z fragmentem nadrzędnym (*parent*),

– ten sam identyfikator dla danych (**msgfromchild** i **msg1**) jest użyty

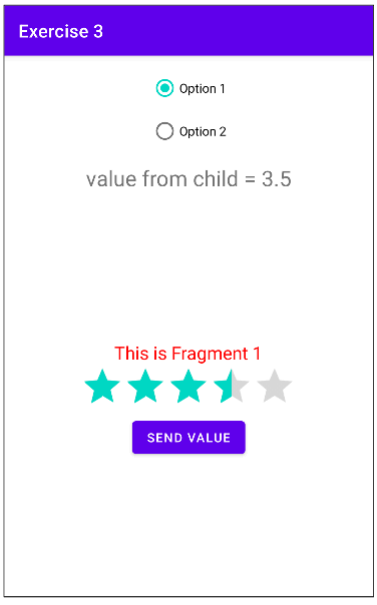
– **bundleOf** tworzy obiekt typu **Bundle** z przekazanymi danymi,

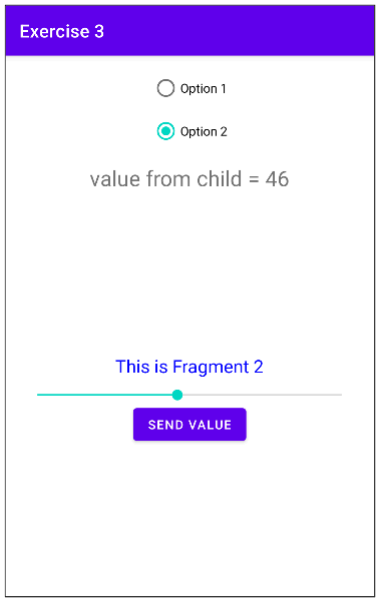
– intrukcja **… to …** w **bundleOf** tworzy parę **(keyname,value)**.

- Zaimplementuj to samo we fragmencie **Fragment2** dla suwaka *(SeekBar*).

Sprawdzenie i test działania:

- Uruchom aplikację i sprawdź działanie. Efekty działania po kliknięciu przycisków powinny być podobne do poniższych:





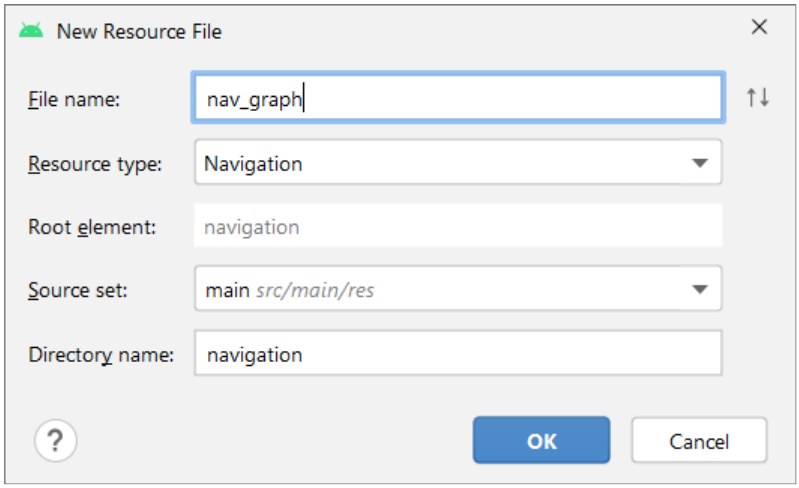
***Uwaga**: zauważ ostatni problem: po ustawieniu i wysłaniu danych, po obrocie te dane nie są odtwarzane w polu TextView. Mechanizm trwałości takich danych będzie poruszony oddzielnie.*

2. Navigating using Navigation Framework

The Navigation Framework supports navigation between app components (activities and fragments) with use of the following elements:

- navigation graph defined by **navigation** resource – the xml resource file with **navigation node** component,
- **destination** elements (fragments or activities) defined in navigation graph,
- **action** elements (nodes) used in navigation graph to specify path (transition) from one destination to another (or general actions defining only destinations),
- **navigation host** – an specific container where destinations are placed – an element derived from **NavHost**.
- **navigation controller** – a NavController object with use of which we activate path/transition from one destination to another (i.e. trigger navigation).

The navigation graph can be created using *File* → *New…* → *Android Resource file* menu option or *New…* → *Android Resource file* context menu option of *app/res* directory:

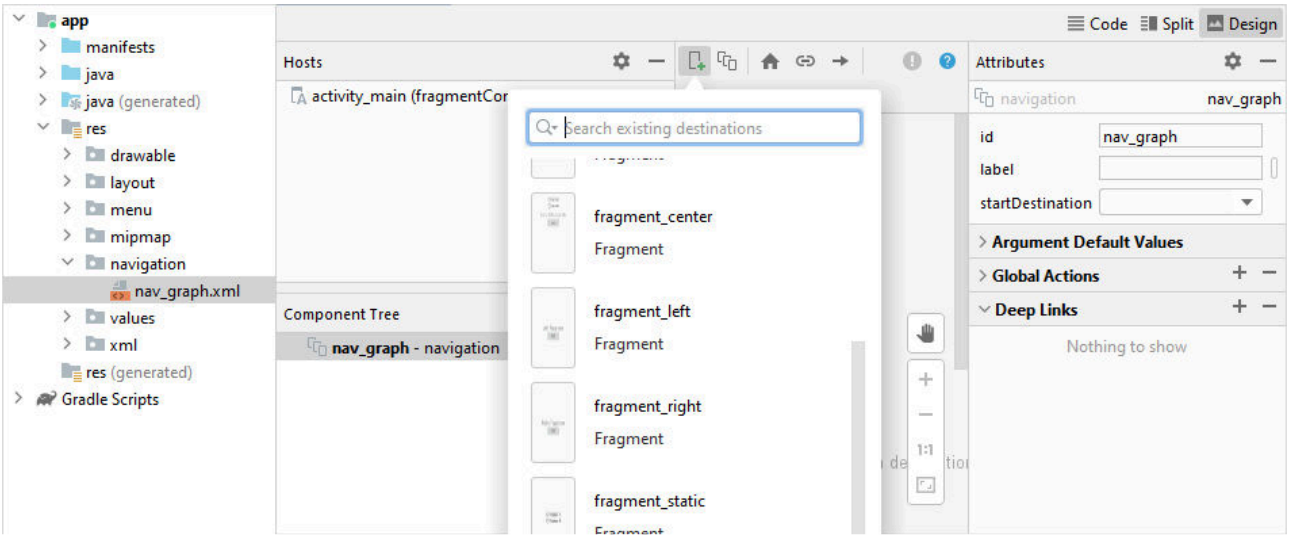


You must choose **Navigation** *Resource type* and set the unique name. The resource file is located by default in **app/res/navigation** folder.

The navigation editor is available in Design mode when you select (double click) the navigation graph xml resource file.

It supports adding new destinations and defining paths (adding actions). When clicking add icon (see the figure) you can select from already defined app components (activities or fragments).

You can define start destination with use of **app:startDestination** attribute of **navigation** component.

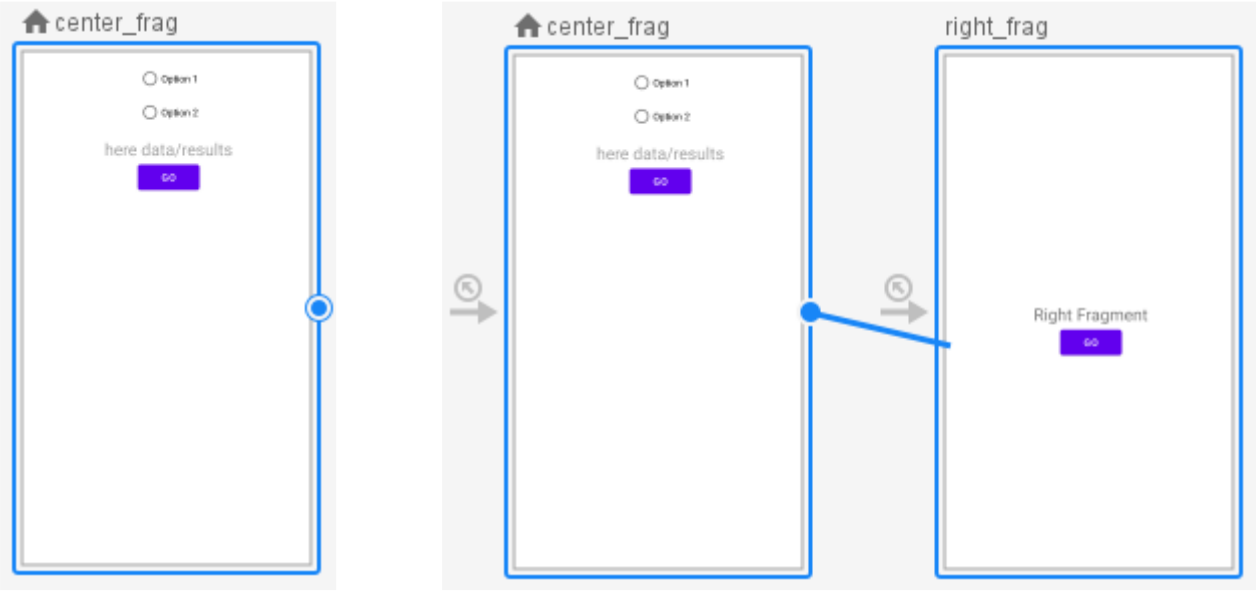


The added destination is represented as the node with ID, name (specifies the destination class), and label that can be displayed on app bar when we go to destination. E.g. for Fragment Center:

```
<fragment
  android:id="@+id/center_frag"
  android:name="com.example.ex03.FragmentCenter"
  android:label="Fragment Center"
  tools:layout="@layout/fragment_center" >
</fragment>
```

The added tools:layout attribute is for visibility the content in the editor.

The destination-to-destination paths can be added by selecting the destination and dragging from exposed point to other destination.



The actions are represented in the xml file as **<action>** node:

```
<fragment
  <action
    android:id="@+id/action_to_fragCenter"
    app:destination="@id/center_frag" />
</fragment>
```

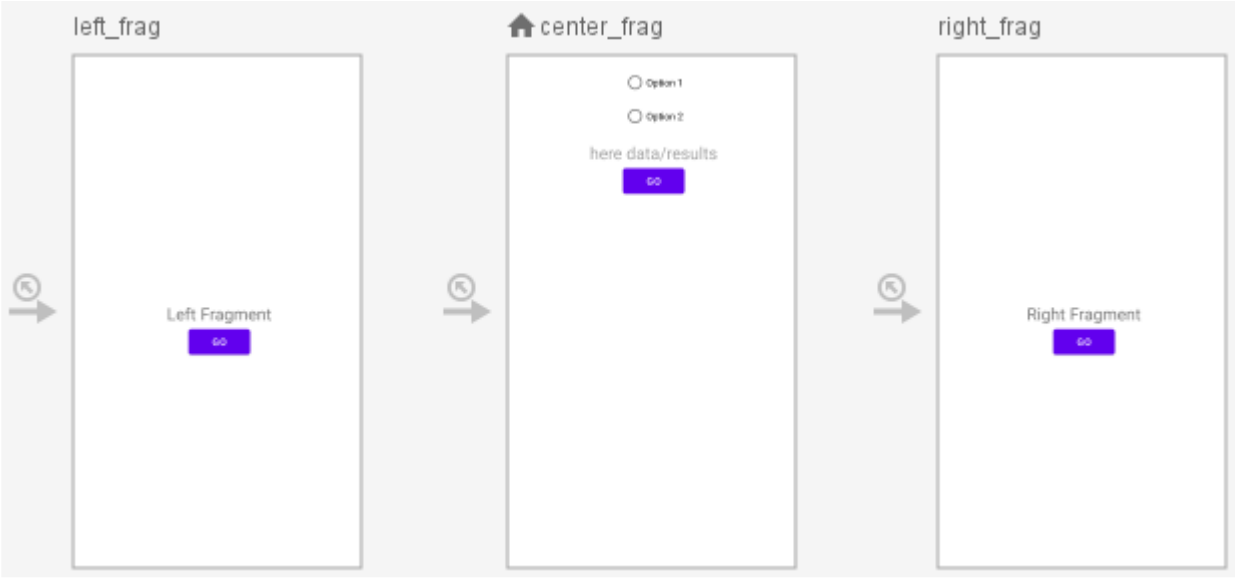
The **app:destination** attribute specify the destination to go. It can be added also manually in the xml code. You can define also **global actions** defined inside navigation node but outside any destination. They permit navigation from any component to defined in action destination.
The two important attributes you can define for actions are:

- **launchSingleTop** – specifies that one instance of destination exists if navigating to it one by one,
- **popUpTo** – specifies the destination up to which pop other visited destinations off the back stack.

Defining navigation graph and actions:

Here we replace and add moving from given destinations (activity or fragment) to other one.

- Add two simple fragments (here named **FragmentLeft** and **FragmentRight**) in the project. Include in the center of them descriptive text view.
- Add the navigation graph resource file **nav_graph** to the project (as described above).
- Add three destinations – three fragments: Left, Center, and Right – with use of editor as shown in the figure (here with ids: **left_frag**, **center_frag**, **right_frag**).



- Define three global actions to the added destinations (here with ids: **action_global_to_fragLeft**, **action_global_to_fragCenter**, **action_global_to_fragRight**,).

The navigation host can be a **NavHostFragment** or androidx.fragment.app.**Fragment** included in the activity or fragment layout. One such host must be a part of the activity (or fragment) layout if you want use Navigation framework.

The NavHostFragment is de facto **FragmentContainerView** component with specified attribute:

```
android:name="androidx.navigation.fragment.NavHostFragment"
```

I should point to defined navigation graph xml resource file, i.e. contain attribute **app:navGraph**:

```
app:navGraph="@navigation/the_name_of_nav_graph_resouce_file"
```

To trigger moving (displaying) other destination the navigation controller must be used. It can be received using one of the method:

```
Fragment.findNavController()
View.findNavController()     (or as propertyaccess: View.navControler)
Activity.findNavController(viewId: Int)
```

If a NavHost is a **FragmentContainerView** included in activity layout to get the controller you must use the code:

```
val navHostFragment=supportFragmentManager.findFragmentById(R.id.navContainer)
                                                    as NavHostFragment
val navController = navHostFragment.navController
```

where: navContainer is ID of NavHostFragment (it's FragmentContainerView).

Otherwise an exception is thrown.

In any destination you can do the navigation using the method:

```
NavController.navigate(R.id.Xxx)
```

Where *Xxx* in action identifier or destination identifier in some situations.

Modify activity to use Navigation Framework:

- In the main activity layout remove previous container and add **NavHostFragment** and point it to previously defined navigation graph. Constrain the container appropriately on the whole screen.

…or:

- In the main activity layout change and add two attributes of existing **FragmentContainerView** container:

```
android:name="androidx.navigation.fragment.NavHostFragment"
app:defaultNavHost="true"
app:navGraph="@navigation/nav_graph"
```

and remove previous `tools:layout` attribute if exists.

- In the main activity layout add the **BottomNavigationView** component, define for the bottom navigation the menu layout xml resource file with three items Left, Center, and Right. Define labels and some icons.

***Attention**: the IDs of menu items **must be the same** as IDs of destinations in navigation graph for one of possible navigation methods. It is always worth to follow this rule.*

The view will be used to trigger navigation to proper fragments.

The final layout view should be similar to the next:

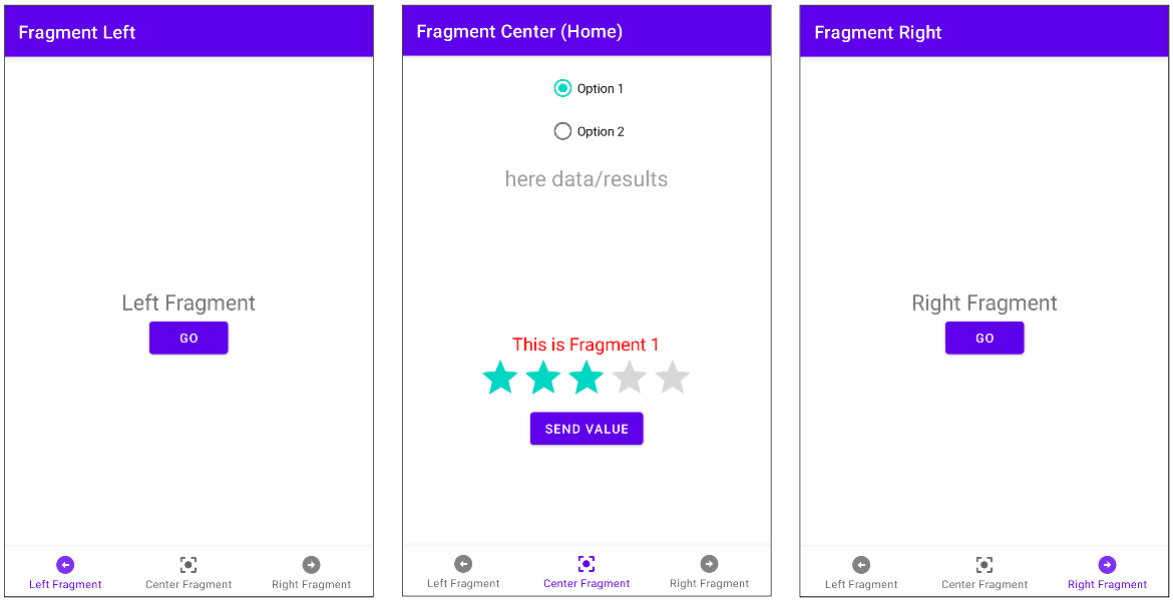
- In the main activity in **onCreate(…)** method:
  - Get the reference to **NavHostFragment** and then to **NavController**.
  - Get the reference to **BottomNavigationView**.
  - Set for BottomNavigation the **OnItemSelected** listener with **setOnItemSelectedListener** method, and in the listener the code for navigation depending on clicked BottomNavigation item:

```
R.id.left_frag -> navController.navigate(R.id.AAA)
R.id.center_frag -> navController.navigate(R.id.BBB)
R.id.right_frag -> navController.navigate(R.id.CCC)
```

    where AAA, BBB, CCC are proper action IDs to start **FragmentLeft**, **FragmentCenter**, and **FragmentRight** respectively.

    Note: item IDs are the same as destination IDs in graph (however here it is not necessary).

- Run the application and check the operation. The app look should be similar to the following:



  - Check the operation when you several times select different and the same destinations, and then click the back button (as many times as needed) to exit application.

- Set up action's attribute settings (**launchSingleTop**, **popUpTo**, etc.) to avoid many instances of fragments and better navigation – e.g. to always back to home (firstly displayed) fragment and then on next back exit application.

Performing navigation with BottomNavigation UI component can be done in other way. This permits to use a little bit less code and define some additional behavior.

- Remove (or better make as a comment) setting the `OnItemSelection` listener for BottomNavigation view, and instead enter the code (in activity's onCreate):

```
val appBarConfig : AppBarConfiguration =
 AppBarConfiguration(setOf(R.id.left_frag,R.id.center_frag,R.id.right_frag))
setupActionBarWithNavController(naviController,appBarConfig)
bottom_navigation.setupWithNavController(naviController)
```

- Run the application and check the operation. Try to notice the difference of app behavior.

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.

Wymagane umiejętności – elementy programistyczne do opanowania/realizacji.

1. Defining fragments and using them in the UI.
2. Dynamical management of fragments.
3. Implementing data exchange between fragments.
4. Defining navigation graph and using it for navigation.
5. Composing bottom navigation view with navigation elements.

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