ACADEMIA DE STUDII ECONOMICE DIN BUCUREȘTI

Facultatea de Cibernetică, Statistică și Informatică Economică

Departamentul de Informatică și Cibernetică Economică

DISPOZITIVE ȘI APLICAȚII MOBILE
2021-2022

Material didactic pentru ID

Drd. Florentin-Alexandru DIȚĂ

București

2021

CUPRINS

1 Dezvoltarea aplicațiilor mobile – principii, activități 3
1.1 Mediul de lucru 3
1.2 Ce este o activitate? 3
1.3 Ciclul de viață al unei activități 3
1.4 Care sunt resursele unei aplicații mobile? 3
1.5 Salvarea/restaurarea stării unei activități 4
1.6 Afișarea log-urilor în Android Studio 5
2 Activități, machete (layout-uri), controale vizuale 6
2.1 Tipuri de machete 6
2.2 Componente vizuale 6
2.3 Navigarea între activități 7
3 Controale vizuale, tratarea evenimentelor 9
3.1 Transfer de date între activități 9
3.2 Alte elemente aprofundate 11
4 Meniuri 12
4.1 Meniuri navigabile 12
4.2 Meniuri laterale clasice 12
5 Fragmente 14
6 Dezvoltarea aplicațiilor mobile - Adaptor personalizat 16
7 Accesul la rețea, prelucrare fișiere JSON/XML 18
7.1 Realizarea operațiilor asincrone 18
7.2 Conexiunea la rețea 18
7.3 Ce este JSON - JavaScript Object Notation? 19
7.4 Prelucrarea unui fișier JSON 19
8 Stocarea persistentă a datelor – Fișiere de preferințe 20
9 Stocarea persistentă a datelor – Baze de date locale 21
10 Implementarea aplicației mobile - Grafică 23
11 Tutoriale suport pentru curs 24
11.1 Tutorial 1 - Creare proiect cu meniu NavigationDrawer 24
11.2 Tutorial 2 - Adăugarea unui meniu clasic la nivelul unei activități 28
11.3 Tutorial 3 - Formular de adăugare transfer de date între activități 28
Bibliografie 31

1 Dezvoltarea aplicațiilor mobile – principii, activități

1.1 Mediul de lucru

Mediul de lucru pentru dezvoltarea aplicațiilor mobile este Android Studio. Disponibil aici.

1.2 Ce este o activitate?

O activitate este unitatea de baza dintr-un proiect Android. Orice aplicație mobila are cel puțin o activitate, care trebuie sa fie Main și Launcher. O activitate este formata din doua componente: o clasa Java - responsabila cu manipularea datelor; un fișier XML - responsabil cu construirea interfeței grafice. Metoda **setContentView** este utilizata pentru atașarea unui fișier din directorul res/layout unei clase Java care extinde AppCompatActivity(clasa părinte pentru marcarea unei activități). Parametrul de intrare reprezintă calea către layout (R.layout.<nume_fisier>).

1.3 Ciclul de viață al unei activități

Ciclul de viață reprezintă stările prin care o activitate trece pe parcursul unei execuții. Este format din următoarele metode:

- **onCreate()** - *singura metoda obligatorie pe care o activitate trebuie sa o implementeze. Este responsabila cu atașarea fișierul XML (din layout) corespunzător clasei Java*;
- **onPause()/onStop()** - *se pot executa de mai multe ori. Sunt declanșate în momentul în care activitatea nu se mai afla pe firul principal de execuție al telefonului(ex: introducem aplicația în background). În interiorul acestor metode se realizează închiderea diverselor conexiune*;
- **onResume()/onStart()** - *se pot executa de mai multe ori. Sunt declanșate în momentul în care activitatea revine pe firul principal de execuție al telefonului. În interiorul acestora se refac conexiunile externe*;
- **onDestroy** - *se executa o singura data, în momentul în care se dorește distrugerea activității. Ex: se închide aplicația, se apelează metoda finish()*

1.4 Care sunt resursele unei aplicații mobile?

Resursele un aplicații mobile pentru platforma Android sunt reprezentate de următoarele elemente:

- **res** - directorul care conține toate resursele proiectului. Este format din layout, drawable, mipmap, values(colors.xml, styles.xml, strings.xml), menu etc.
- **layout** - directorul care conține fișiere XML corespunzătoare unei activități
- **strings.xml** - conține toate constantele de tip text care apar pe ecranul dispozitivului mobil.

Utilizarea resurselor se realizează după următoarele reguli:

- **în fișierele xml**:
    - '@<nume_folder>/<nume_fisier>' - pentru toate directoarele mai puțin 'values';

- '@<nume_fisier>/<valoare_proprietate_name>' - pentru fișierele din directorul 'values';
- '@id/<valoare_proprietate_android:id>' - pentru componentele vizuale care au proprietatea android:id populata.

- **în Java este reprezentată de clasa R generată la compilare. În Figura 1 se poate observa o ierarhie a acestei clase**:
  - 'R.<nume_folder>.<nume_fisier>' - pentru toate directoarele mai puțin 'values';
  - 'R.<nume_fisier>.<valoare_proprietate_name>' - pentru fișierele din directorul 'values';
  - 'R.id.<valoare_proprietate_android:id>' - pentru componentele vizuale care au proprietatea android:id populata.

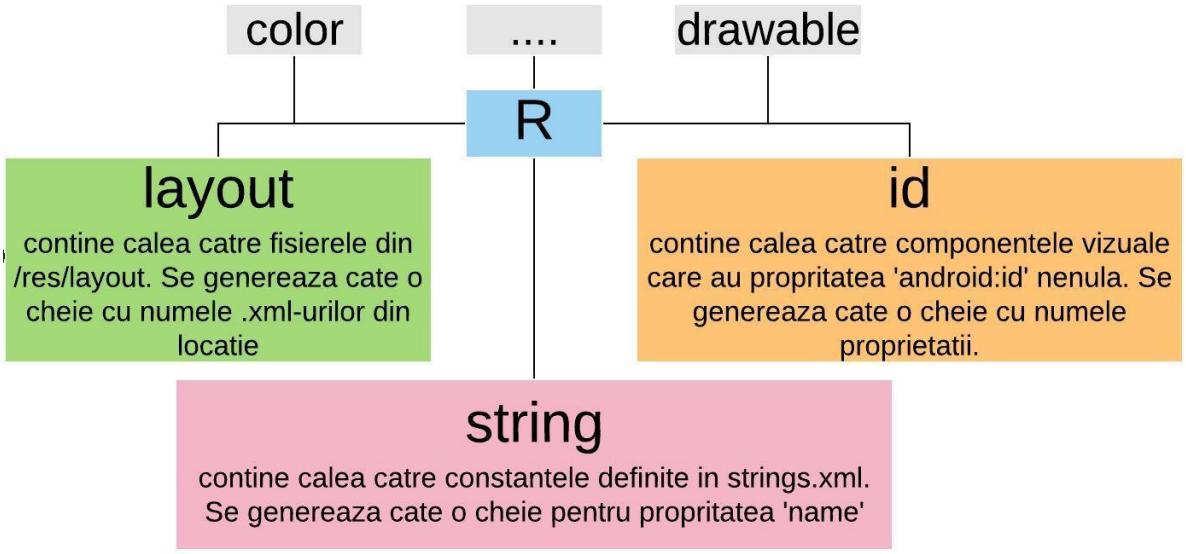

Figura 1 Ierarhia clasei R

1.5 Salvarea/restaurarea stării unei activități

Salvarea stării unei activități este gestionata în mod automat de către sistemul de operare Android prin intermediul obiectul savedInstanceState de tip Bundle, primit ca parametru de intrare de către metoda onCreate. Scopul acestei salvări este de a păstra pe ecranul dispozitivului mobil informațiile introduse de utilizator atunci când acesta executa anumite operații precum: rotirea dispozitivului mobil, introducerea în fundal a aplicației etc.

Cu toate acestea exista câteva situații în care salvarea automata nu funcționează cum ar fi popularea componentelor vizuale de tip TextView cu informații dinamice încărcate în urma unor acțiune la execuție. În aceste situații ar trebui sa se suprascrie următoarele metode:

- **onSaveInstanceState** -> are ca parametru de intrare un obiect de tip Bundle utilizat pentru stocarea informațiilor pe care dorim sa le păstram. Prin urmare, scopul acestei metode este de a salva în memoria RAM informațiile utile pentru operația de restaurare

- **onRestoreInstanceState** -> are ca parametru de intrare același obiect de tip Bundle utilizat în metoda **onSaveInstanceState** pentru stocare și primit de către metoda **onCreate**. Scopul acestei metode este de a citi informațiile din Bundle și de a le repoziționa pe componentele vizuale corespunzătoare.
- **Bundle** -> clasa container responsabila cu stocarea diferitelor informații care se doresc a se transmite intre activități sau de la o activitate către un fragment. Aceasta clasa dispune de metode de tipul cheie-valoare a căror denumire începe cu 'put'. Metoda **onCreate()** prezinta un parametru de intrare de tip Bundle, denumit onSavedInstanceState fiind utilizat în salvarea stării unei activități. Daca valoarea parametrului **onSavedInstanceState** este null atunci utilizatorul a intrat prima data în activitatea respectivă.

1.6 Afișarea log-urilor în Android Studio

Clasa Log este utilizata pentru a scrie diverse mesaje în View-ul Logcat din Android Studio. Aceste mesaje sunt utile pentru depanarea aplicației.

2 Activități, machete (layout-uri), controale vizuale

2.1 Tipuri de machete

Layouturile definesc structura interfeței grafice în aplicațiile mobile. Rolul acestora este de a găzdui diferite componente vizuale (widgets) într-o anumita ordine.

- **LinearLayout** -> aliniază toate componentele vizuale într-o singura direcție (verticala/orizontala). Direcția este specificata cu ajutorul proprietății 'android:orientation'. De asemenea, permite distribuirea dinamica a spațiului intre componentele vizuale, prin intermediul proprietăților:
  - weightSum - este populata la nivelul tag-ului ; valoarea acesteia este un întreg (nu contează valoarea) ce reprezintă numărul de unități pe care containerul le poate împarți;
  - layout_weight - este populat la nivelul componentelor vizuale. Valoarea acesteia este în intervalul [0, weightSum]. Aceasta reprezentând proporția spațiului ocupat de componenta vizuala în container; Pentru folosirea acestor proprietăți este necesar ca layout_width sau layout_height sa aibă valoarea 0dp (aceasta decizie este luata în funcție de valoarea proprietății android:orientation, daca este vertical atunci layout_height este marcat cu 0, altfel layout_width).
- **RelativeLayout** -> afișează elementele vizuale în poziții relative. Poziția se poate stabili atât în funcție de părinte cat și de celelalte componente vizuale. Poziția este stabilita din punctul stânga-sus. Proprietăți specifice: layout_alignParentTop, layout_alignParentLeft, layout_alignParentRight, layout_alignParentBottom, layout_centerVertical, layout_centerHorizontal, layout_below, layout_above, layout_toRightOf, layout_toLeftOf.
- **ConstraintsLayout** -> reprezintă o combinație intre LinearLayout și RelativeLayout. Proprietăți specifice: layout_constraintStart_toStartOf, layout_constraintStart_toBottomOf, layout_constraintStart_toEndOf, layout_constraintStart_toTopOf, layout_constraintTop_toTopOf …, layout_constraintBottom_toBottomOf etc.
- **DrawerLayout** -> utilizat pentru desenul meniurilor navigabile.
- **FrameLayout** -> afișează o singura componenta în ecran. Este utilizat în special pentru afișează fragmentelor.
- **CoordinatorLayout** -> deriva din FrameLayout, fiind utilizat pentru afișarea Chrome-ului sau componentei NavigationDrawer.

2.2 Componente vizuale

Orice activitate este formată din mai multe componente vizuale adăugate atât la nivelul fișierelor xml din res/layout cât și la nivelul claselor Java care extind AppCompactActivity.

Mai jos puteți găsi o parte din paleta de componente vizuale pe care platforma Android le oferă:

- **RadioGroup** -> container special utilizat pentru găzduirea mai multor RadioButton-uri. Are aceleași proprietăți ca și LinearLayout. Acesta asigura crearea unui context special

pentru RadioButton-urile din interiorul sau, asigurând alegerea unei singure opțiuni dintr-un set.

- **RadioButton** -> O componenta vizuala ce are proprietatea checked pentru a stoca raspunsul utilizatorului.
- **TextInputLayout** -> este o componenta vizuală care este formată dintr-un EditText, iar în momentul în care utilizatorul completează câmpul aferent sugestiei controlului de tip EditText (proprietatea hint), este transformat într-un TextView.
- **Spinner** -> este o componenta vizuala ce asigura selecția unei opțiuni dintr-o lista derulanta. Lista se încarcă într-un Spinner prin intermediul unui Adapter.
- **FloatingActionButton** -> este o componenta vizuală ce a apărut în versiunile recente ale Android-ului, reprezentând o varianta îmbunătățita a Button-ului. Acesta are forma de cerc, și înlocuiește eticheta cu o imagine vectorial, ce se poate adaugă în res/drawable.

Alte metode utilitare pentru manipularea și atașarea de evenimente componentelor vizuale:

- **findViewById()** -> metoda ce aparține de AppCompatActivity. Este responsabila cu inițializarea unui obiect Java de tipul componentelor vizuale cu corespondentul din layout(fișiere xml). Are un singur parametru de intrare de tip int, care reprezintă identificatorul widget-ului(valoarea proprietății android:id din xml)
- **setOnClickListener** -> metoda ce aparține unor componente, precum: Button sau FloatingActionButton. Este declanșată în momentul în care utilizatorul apasă pe componenta vizuala. Primește ca parametru de intrare o instanță a unei clasei ce trebuie sa implementeze interfața View.OnclickListener. În momentul în care se implementează aceasta interfața este obligatoriu suprascrierea metodei onClick. În interiorul acesteia se adaugă codul care e dorește sa se execute în momentul declanșării acestui eveniment.
- **Adapter** -> O clasa utilitara care se comporta ca o punte intre componentele vizuale și o lista de elemente. Creează view-urile pentru fiecare element al unei liste de obiecte. De asemenea, oferă accesul din interfața la lista de obiecte.
- **ArrayAdapter** -> este un adaptor care folosește un vector de obiecte. Sursa de preluare a vectorului poate sa fie: în memoria Java, din baza de date, din rețea sau din strings.xml. Acest adaptor este responsabil sa încarce fiecare element al vectorului într-un TextView. De asemenea, Textview-ul este populat cu metoda toString() al obiectelor din vector.
- **string-array** -> tag utilizat pentru definirea unui vector static în res/values/strings.xml

2.3 Navigarea între activități

Într-o aplicație mobile Android navigarea între activitățile acesteia se realizează prin intermediul următoarelor metode/clase ajutătoare:

- **Intent:** este o clasa care permite deschiderea unei activități din interiorul alteia. De asemenea, este utilizata pentru transmiterea diferitelor tipuri de date intre activități. Permite transferul atât a tipurilor primitive - int, long, double, String, cat și a clasele personalizate - acestea trebuie sa implementeze Serializable sau Parcelable. Constructorul clasei primește Contextul aplicației și tipul clasei asociate activității pe care o deschidem.

- **getApplicationContext**: metoda ce aparține clasei AppCompatActivity, fiind utilizată pentru obținerea contextului aplicației mobile. Contextul reprezintă mediul de care se leagă toate componentele vizuale și activitățile. Fiecare soluție mobila are un context definit, astfel încât sistemul de operare Android sa știe ce informații încarcă în ecranul utilizatorului într-un anumit moment de timp.
- **startActivity**: metoda ce aparține clasei AppCompatActivity, scopul metodei fiind deschidere unei activități din interiorul alteia. Constructorul acesteia.

3 Controale vizuale, tratarea evenimentelor

3.1 Transfer de date între activități

Pentru realizarea transferului de date între activități sunt utilizate următoarele metode:

- **ActivityResultCallback:**
  - este o interfață ce conține o singura metoda **onActivityResult()**
  - **onActivityResult**() este apelata de ActivityResultLauncher in momentul finalizării apelurilor **setResult**(), respectiv **finish**().
  - **onActivityResult**() primește un singur parametru de intrare de tipul **ActivityResult**
  - **ActivityResult** este format din doua atribute:
    - resultCode - trebuie sa fie egal cu cel din **setResult**();
    - data - reprezintă intentul lansat de **ActivityResultLauncher** prin apelul metode **launch**() si in același timp este preluat in activitatea secundara, îmbogățit cu informații pentru activitatea principala, iar in final trimis ca al doilea parametru al metodei **setResult**()
- **ActivityResultLauncher:**
  - este un obiect responsabil cu lansarea diferitelor acțiuni pentru aducerea informatiilor într-o activitate. Aceste acțiuni pot fi:
    - **StartActivityForResult** - utilizata pentru deschiderea unei activități secundare din cadrul aplicației pentru a prelua diferite informații de la utilizator
    - **PickContact** - utilizata pentru preluarea unui contact din agenda telefonului mobil
    - **TakePicture** - utilizata pentru preluarea unei poze
    - **TakeVideo** - utilizata pentru preluarea unui video
    - etc
  - apelează metoda **onActivityResult** din **ActivityResultCallback** cu rezultatul primit din activitatea deschisa pe baza acțiunii definite
- **launch** - metoda disponibila la nivelul clasei **ActivityResultLauncher**. In cazul acțiunii de tip **StartActivityForResult** primește ca parametru de intrare Intent-ul care deschide activitatea secundara
- **setResult()** -> metoda utilizata în clasa deschisa cu metoda **startActivityForResult**() pentru a întoarce rezultatele așteptate de activitatea apelator. Metoda primește doi parametri:
  - resultCode - se marchează tipul de răspuns (pentru un rezultat corect se utilizează RESULT_OK);
  - data - Mesajul cu care s-a deschis activitatea și care conține informațiile pe care dorim sa le transmitem.
- **finish()** -> metoda utilizata pentru a finaliza ciclul de viată al unei activități.
- **Parcel** -> Container de mesaje. Este folosit pentru a transmite obiecte Java intre activități.

- **Parcelable** -> Interfața implementata de clasele Java pe care dorim sa le transmitem intre activități prin intermediul unui Parcel. Permite scrierea și citirea dintr-un Parcel. Trebuie sa implementam doua metode: **writeToParcel()**, **describeContents()**. Trebuie definit un obiect public static numit CREATOR de tipul Parcelable.Creator pentru a se realiza operație de citire. Obiectul Creator implementează metoda **createFromParcel()** care realizează citirea. Implementarea are o singura regula, și anume ordinea de citire trebuie sa fie identica cu cea de scriere.
- **putExtra()** -> metoda disponibila la nivelul clasei Intent, fiind utilizata pentru salvarea unui obiect personalizat în Intent. Clasa obiectului salvat implementează interfața Parcelable/Serializable.
- **getParcelableExtra()** -> metoda disponibila la nivelul clasei Intent, fiind utilizata pentru preluare unui obiect personalizat din Intent. Clasa obiectului preluat implementează interfața Parcelable. Parametrul de intrare reprezintă cheia pe care s-a salvat obiectul prin apelul metodei **putExtra()**.
- **getSerializableExtra()** -> metoda disponibila la nivelul clasei Intent, fiind utilizata pentru preluare unui obiect personalizat din Intent. Clasa obiectului preluat implementează interfața Serializable. Parametrul de intrare reprezintă cheia pe care s-a salvat obiectul prin apelul metodei **putExtra()**.
- **getText()** -> metoda disponibila la nivelul componente vizuale de tip input, precum: EditText, TextInputEditText, RadioButton etc. Asigura preluarea informației introduse de utilizator.

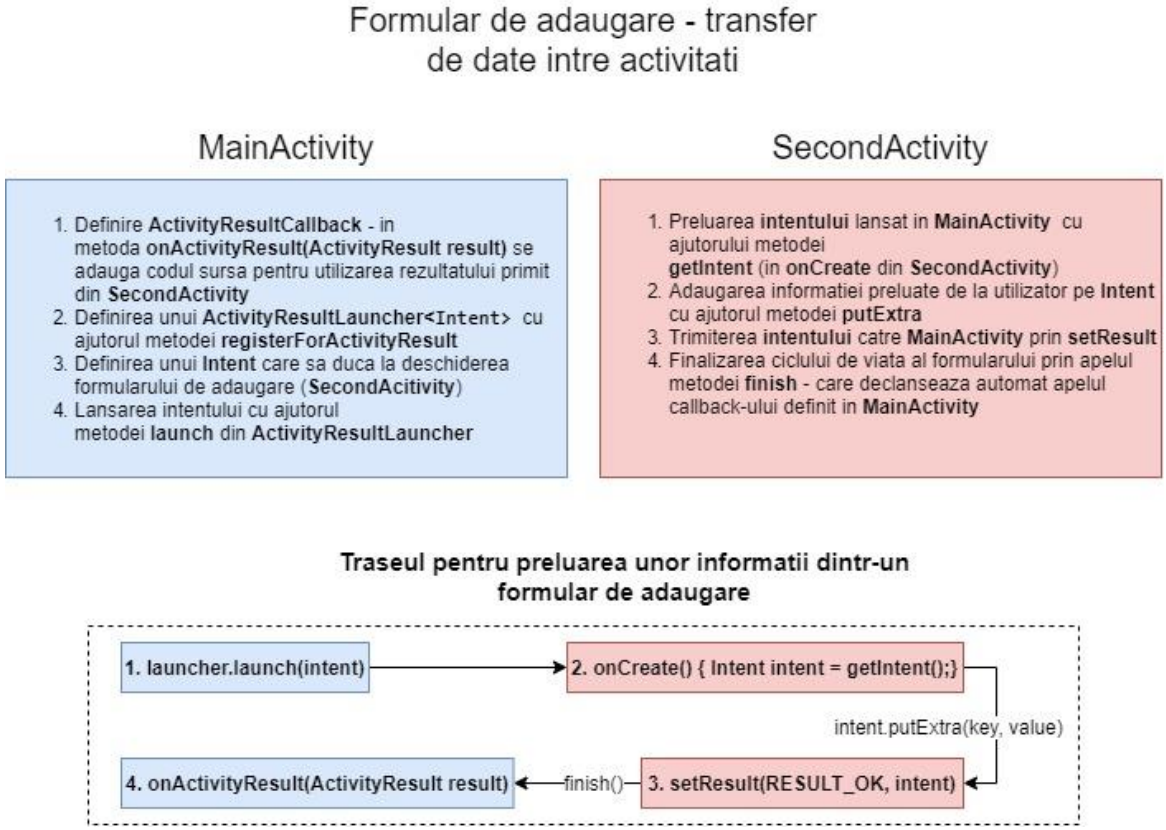

Figura 2 Mecanism transfer de date

În Figura 2 se observa mecanismul pentru realizarea transferului de date între activități.

3.2 Alte elemente aprofundate

Alte elemente utilizate în Android pentru afișarea de mesaje și atașarea de evenimente:

- **Toast** -> Clasa Java prin intermediul căreia se afișează mesaje temporare pe ecranul dispozitivului mobil. Pentru a afișa mesajul pe ecran se apelează metoda **show()**, iar pentru construirea unei instanțe, se utilizează metoda **makeText()** care are trei date de intrare:
  - Contextul aplicației;
  - Identificatorul resursei din strings.xml care reprezintă mesajul ce se afișează;
  - durata de afișarea ce se măsoară în milisecunde.
- **getSelectedItem()** -> metoda disponibila la nivelul obiectelor de tip Spinner. Returnează valoarea selectata de utilizator din lista de opțiuni.
- **getCheckedRadioButtonId()** -> metoda disponibila la nivelul obiectelor de tip RadioGroup. Returnează identificatorul RadioButton-ului din interiorul grupului care are proprietatea **checked = true**.
- **SimpleDateFormat** -> este o clasa Java de bază utilizată pentru conversia de la String la Date, respectiv de la Date la String. Aceasta primește un format de data pe care sa-l aplice în timpul conversie. Metodele utilizate pentru conversie sunt:
  - **parse()** - utilizata pentru extragerea unui obiect Date dintr-un String. Execuția se realizează într-un block de try-catch, deoarece metoda arunca o excepție ce trebuie tratata;
  - **format()** - utilizata pentru transformarea unui obiect Date în String.
- **Paramentru în Strings.xml** -> În strings.xml se pot adăuga mesaje statice cu parametru. Regula de formare a acestora este: **%<nr_parametru>$<tip_date>**, unde nr_parametru - este o valoare > 0, reprezentând poziția parametrului din metoda **getString**; tip_date - se poate afișa s(String), i(int), d(double). Exemplu: *Buna %1$s*
- **getString** -> metoda pusa la dispoziție de pachetul de bază Android. Este utilizata pentru încărcarea unui text din strings.xml. Are doua implementări, prima accepta doar identificatorul resursei din strings.xml, iar cea de a doua accepta pe lângă identificatorul resurse și un numărul variabil de parametri care reprezintă valorile variabilelor dinamice declarate în strings.xml.

4 Meniuri

4.1 Meniuri navigabile

Pentru realizarea meniurilor navigabile sunt utilizate următoarele directoare, clase și metode din pachetul Android:

- **menu** -> directorul care conține fișiere XML corespunzătoare unui meniu
- **Toolbar** -> clasa Java corespunzătoare componentei vizuale. Reprezintă bara de acțiune dintr-o aplicație mobila.
- **setSupportActionBar()** -> metoda ce aparține de AppCompatActivity. Este utilizata pentru a atașa unei activități un obiect de tipul Toolbar. La nivel vizual, apare bara de acțiune. Are un singur parametru de intrare de tip Toolbar
- **DrawerLayout** -> clasa Java corespunzătoare containerului DrawerLayout. Este utilizata pentru construirea meniul navigabil și manipularea evenimentelor acestuia.
- **ActionBarDrawerToggle** -> clasa Java având următoarele responsabilități:
    - asigura asocierea dintre Toolbar (bara de acțiune) și DrawerLayout (meniul lateral);
    - afișează pe ecran în partea stânga-sus un burger menu, iar prin apăsarea acestuia este realizata acțiunea de deschidere/închidere a meniului lateral. Constructorul clasei primește o instanță de tip Activity, un DrawerLayout, un Toolbar și doi identificatori din strings.xml care ar trebui sa aibă următoarele denumiri: navigation_drawer_open, navigation_drawer_close (acestea reprezintă cele doua stări în care se poate afla ActionBarDrawerToggle). Daca cei doi identificatori nu sunt prezenți în fișierul strings.xml, aceste șiruri ar trebui adăugate.
- **syncState()** -> metoda de la nivelul ActionBarDrawerToggle. Asigura sincronizarea stării instanței apelate.
- -> componenta vizuala utilizata pentru implementarea meniului lateral. Container de care aparține este DrawerLayout. Este formata din doua elemente:
    - ***header*** - reprezintă partea superioara din meniu. Proprietate utilizata 'app:headerLayout' - valoarea acesteia fiind un fișier XML din res/layout;
    - ***menu*** - reprezintă opțiunile meniului lateral. Proprietate utilizată 'app:menu' - valoarea acesteia fiind un fișier XML din res/menu.
- **NavigationView** -> clasa Java corespunzătoare componentei vizuale reprezentând meniul lateral. În Java este utilizata pentru a interacționa cu opțiunile meniului lateral.
- **OnNavigationItemSelectedListener** -> Interfața ce aparține clasei NavigationView fiind responsabila cu interceptarea momentului în care una din opțiunile meniului lateral este apăsată.

4.2 Meniuri laterale clasice

Pentru atașarea unui meniu lateral clasic într-o aplicație Android sunt utilizate următoarele directoare, clase și metode:

- **onCreateOptionsMenu** - metoda implementata într-o activitate pentru atașarea unui meniu. Parametru de intrare obiect Menu (interfață Java de tip resursa corespunzătoare fișierului XML din res/menu).
- **MenuInflater** – clasă utilitara care asigura asocierea dintre un obiect Java de tip Menu cu fișierul XML corespunzător din res/menu. Metoda utilizata inflate, care primește calea către fișierul xml și instanța Java de tip Menu.
- **Paramentru în Strings.xml** - În strings.xml se pot adăuga mesaje statice cu parametru. Regula de formare a acestora este: **%<nr_parametru>$<tip_date>**, unde nr_parametru - este o valoare > 0, reprezentând poziția parametrului din metoda **getString**; tip_date - se poate afișa s(String), i(int), d(double). Exemplu: *Buna %1$s*
- **getString** - metoda pusa la dispoziție de pachetul de bază Android. Este utilizata pentru încărcarea unui text din strings.xml. Are doua implementări, prima accepta doar identificatorul resursei din strings.xml, iar cea de a doua accepta pe lângă identificatorul resurse și un numărul variabil de parametri care reprezintă valorile variabilelor dinamice declarate în strings.xml.

5 Fragmente

Utilizarea fragmentelor în aplicațiile mobile se realizează prin intermediul următoarelor clase și metode:

- **Bundle** -> clasa container responsabila cu stocarea diferitelor informații care se doresc a se transmite intre activități sau de la o activitate către un fragment. Din punct de vedere tehnic are aceeași structura ca un Map. Aceasta clasa dispune de metode de tipul cheie-valoare a căror denumire începe cu 'put'. Metoda **onCreate()** prezinta un parametru de intrare de tip Bundle, denumit onSavedInstanceState fiind utilizat în salvarea stării unei activități.
- **Fragment** -> reprezintă o parte reutilizabila dintr-o interfață. Fragmentul este asemănător unei activități fiind compus din: o clasa Java - ce extinde *Fragment*; un fișier XML - ce se regăsește în directorul res/layout reprezentând interfața fragmentului. Prezinta ciclu de viată propriu, dar este dependent de cel al activității în care este folosit. Prin urmare, un fragment nu se poate utiliza decât în interiorul unei activități. Metodele ciclului de viată sunt:
    - onAttach() - este apelata în momentul în care fragmentul este atașat unei activități;
    - onCreate() - crearea în memorie a clasei Java;
    - onCreateView() - atașează clasei Java de tip Fragment fișierul XML corespunzător;
    - onStop() - la fel ca la activități;
    - onPause() - la fel ca la activități;
    - onResume() - la fel ca la activități;
    - onStart() - la fel ca la activități;
    - onDestroyView() - asigura eliminarea legăturii dintre clasa Java de tip Fragment și XML-ul corespunzător;
    - onDestroy() - distruge clasa Java de tip Fragment;
    - onDetach() - este apelata atunci când fragmentul nu mai este legat de o activitate.

O activitate poate utiliza fragmente daca conține o componenta vizuala de tip FrameLayout. De asemenea, în cadrul unei activități se pot adaugă mai multe fragmente în paralel, condiția fiind sa existe cate un FrameLayout pentru fiecare dintre ele. Avantajul obținut prin utilizarea fragmentelor este încărcarea unui conținut dinamic într-o activitate/activități..

- **FragmentManager** -> clasa utilitara folosita pentru adăugarea unui fragment în cadrul unei activități.
- **beginTransaction()** -> metoda disponibila la nivelul clasei **FragmentManager**. Apelul acesteia specifică activității ca urmează sa i se atașeze un fragment.
- **replace()** -> metoda disponibila la nivelul clasei **FragmentManager**. Apelul acesteia asigura înlocuirea componentei cu conținutul unui fragment. Prezinta doua date de intrare, și anume:
    - identificatorul componentei din cadrul activității pe care dorim sa o înlocuim,
    - instanța fragmentului.

- **commit()** -> metoda disponibila la nivelul clasei **FragmentManager**. Apelul acesteia asigura afișarea fragmentului pe ecranul dispozitivului mobil.
- **setArguments()** -> metoda disponibila la nivelul clasei **Fragment**. Asigura transferul de informații de la o activitate către fragment. Prezinta un singur parametru de intrare de tip Bundle.

6 Dezvoltarea aplicațiilor mobile - Adaptor personalizat

Adaptorul personalizat este utilizat pentru a modifica aspectul implicit de afișare a liniilor dintr-o componentă vizuală de tip listă (ListView, Spinner, Gallery, RecycleView). Pentru implementarea unui astfel de adaptor este necesar definirea unui fișier XML în directorul res/layout care să reprezinte aspectul vizual al unei singure înregistrări și utilizarea următoarelor metode:

- **notifyDataSetChanged()** -> metoda disponibila la nivelul clasei ArrayAdapter. Este responsabila cu reconstruirea adaptorului asociat unei componente vizuale de tip listare. Este utilizata în momentul în care lista Java folosita în construirea adaptorului își modifica conținutul (adăugare/ștergere/modificare obiecte)
- **getView()** -> metoda disponibila la nivelul clasei BaseAdapter (extinsă de ArrayAdapter) care este responsabila cu transformarea unui obiect Java într-un control vizual configurat prin intermediul fișierelor XML din layout (fie personalizat definit în **res/layout** din proiect, fie predefinit în **res/layout** din framekwork-ul Android)
- **setOnItemClickListener** -> metoda disponibila la nivelul clasei ListView, fiind utilizata pentru atașarea evenimentul de click pe un element din lista vizuala. Oferă ca parametru de intrare poziția pe care s-a apăsat.
- **setSelection** -> metoda disponibila la nivelul clasei Spinner, fiind utilizata pentru afișarea pe ecranul dispozitivului mobil a elementului de pe poziția 'i' din lista de opțiuni.
- **check** -> metoda disponibila la nivelul clasei RadioGroup, fiind utilizata pentru selectarea RadioButton-ului cu identificatorul specificat ca parametru de intrare.
- **hasExtra** -> metoda disponibila la nivelul clasei Intent, fiind utilizata pentru a se verifica daca mesajul conține un parametru cu numele specificat.

Pentru afișarea mesajelor permanente pe ecranul dispozitivelor mobile se utilizează AlertDialog. Metodele specifice acestei clase sunt definite mai jos:

- **AlertDialog** -> clasa pusa la dispoziție de pachetul Android, fiind utilizata pentru afișare mesajelor de tip pop-up pe ecranul dispozitivului mobil. Pentru a defini o instanță de tip **AlertDialog** este necesar utilizarea clasei **Builder**, care primește ca parametru de intrare contextul aplicației.
- **setTitle** -> metoda pusa la dispoziție de către **AlertDialog.Builder**, pentru a adaugă numele ce apare pe bara de acțiune a popup-ului. Are doua forme de implementare, și anume: acceptă String ca parametru de intrare, respectiv identificatorul unei resurse din strings.xml
- **setMessage** -> metoda pusa la dispoziție de către **AlertDialog.Builder**, pentru a adaugă mesajul ce apare pe popup. Are doua forme de implementare, și anume: accepta String ca parametru de intrare, respectiv identificatorul unei resurse din strings.xml
- **setPositiveButton** -> metoda pusa la dispoziție de catre **AlertDialog.Builder**, pentru a adaugă un buton, care reprezintă acțiunea ce trebuie executata daca utilizatorul **este de acord** cu mesajul afișat. Metoda așteaptă ca parametru de intrare eticheta butonului, precum și o implementare a evenimentului de click asociat.
- **setNegativeButton** -> metoda pusa la dispoziție de către **AlertDialog.Builder**, pentru a adaugă un buton, care reprezintă acțiunea ce trebuie executata daca utilizatorul **nu este de**

acord cu mesajul afișat. Metoda așteaptă ca parametru de intrare eticheta butonului, precum și o implementare a evenimentului de click asociat.

- **create** -> metoda ce aparține de **AlertDialog.Builder**, iar rolul acesteia este de a inițializa în memorie o variabile de tip **AlertDialog**.
- **show** -> metoda ce aparține de **AlertDialog** asigurând afișarea popup-ului pe ecranul dispozitivului mobil.

7 Accesul la rețea, prelucrare fișiere JSON/XML

7.1 Realizarea operațiilor asincrone

Pentru realizarea operațiilor paralele se vor utiliza următoarele clase:

- **Executor** -> este o clasa din pachetul Java.io responsabila cu gestionarea rulării diferitelor fire de execuție pe care aplicațiile doresc sa le ruleze. Aceasta clasa asigura decuplarea momentului în care sistemul apelează pornirea unui fir de execuție (new Runnable(){...} sau new Thread()) și momentul efectiv în care acesta începe procesarea informațiilor (apelează metoda run()). Prin urmare, cu ajutorul acestei clase se obține o utilizare mai eficienta a resurselor sistemului de operare.
- **Handler** -> clasa specifica aplicațiilor Android care gestionează o coada de mesaje de tip Runnable. Este utilizata pentru a transmite diferite informații care sunt procesate pe fire de execuție paralele către cel principal pe care rulează o activitate. Aceasta clasa realizează o analiza a resurselor disponibile la nivelul sistemului de operare și decide momentul oportun când sa trimită informația dorita către firul principal.

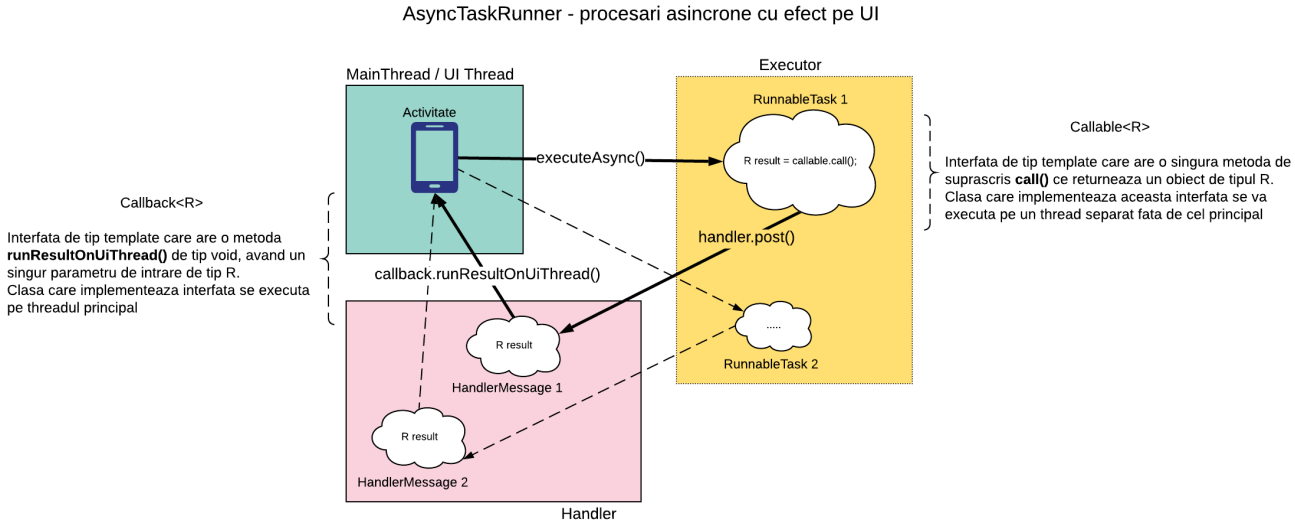

Figura 3 Mecanism pentru realizarea procesărilor paralele

În Figura 3 este detaliat mecanismul pentru realizarea procesărilor paralele.

7.2 Conexiunea la rețea

Pentru a se realiza conexiunea la rețea sunt utilizate următoarele clase:

- **URL** -> clasa utilizata pentru validarea unui URL și pentru obținerea unei conexiuni.
- **HttpURLConnection** -> reprezintă o conexiune cu o anumita adresa URL. Asigura preluarea informațiilor de la adresa dorita.
- **InputStream** -> clasa utilizata pentru preluare unor bucăți de informații dintr-o anumita sursă.

- **InputStreamReader** -> asigura împărțirea unui **InputStream** în unități mai mici de procesare.
- **BufferedReader** -> utilizarea pentru împărțirea unui **InputStreamReader** în unități mici, astfel încât sa se evite apariția erorilor de tip TimeOut sau OutOfMemory.

7.3 Ce este JSON - JavaScript Object Notation?

JSON este o cale de stocare a informațiilor într-o forma organizata. Formatul JSON poate avea structură omogena (reprezentare obiectelor din POO) sau neomogena. Tipurile de date pe care le suporta un JSON sunt: numeric (int, double, float), boolean și String.

Elemente cheie JSON:

- **{}** -> reprezentarea unui obiect. La nivel de Java se poate defini o clasa care sa conțină atributele din interiorul acestor paranteze
- **[]** -> reprezentarea unui vector. În Java se pot construi vectori de valori sau daca simbolul [este urmat de { este vorba despre un vector/lista de obiecte.
- **:** -> Are dubla reprezentare, și anume: ce se afla în partea stângă este numele atributului, iar ce se afla în partea dreapta este valoarea pe care o are acel atribut.
- **,** -> separator de atribute în cadrul unei obiect sau vector.

7.4 Prelucrarea unui fișier JSON

Pentru a realizarea prelucrarea unui fișier JSON la obiecte Java sunt folosite următoarele clase:

- **JSONObject** -> clasa utilizata pentru încărcarea în memoria Java a unui obiect JSON primit sub forma de String. În memorie este încărcat sub forma de Map. Asigura validare JSON -ului prin verificarea existentei pe poziției 0 a caracterului '{'. Oferă metode de tip get pentru extragerea de informații, și anume: getString, getBoolean, getJSONObject(String var), getJSONArray(String var) etc.
- **JSONArray** -> clasa utilizata pentru încărcarea în memoria Java a unui vector JSON primit sub forma de String. În memorie este încărcat sub forma de Map, unde cheia este de tip int, reprezentând indexul din vector. Asigura validarea JSON-ului primit prin verificarea existentei pe poziției 0 a caracterului '['. Permite extragerea unor obiect de tip JSONObject prin intermediul indexului.

8 Stocarea persistentă a datelor – Fișiere de preferințe

Pentru integrarea fișierelor de preferință într-o aplicație mobilă ar trebui utilizate următoarele clase:

- **SharedPreferences** -> clasa utilizata pentru gestionarea fișierelor de preferințe. Aceste fișiere au structura tabelara, având doua coloane, și anume:
    - cheia - variabila String care reprezintă numele unei valori stocate;
    - valoare - variabila de tip String/Integer/Float/Double/Set/Boolean.

Aceste fișiere reprezintă o modalitate de stocare persistenta. Durata de viată este echivalenta cu existenta aplicației pe dispozitivul mobil. Obiectivele clasei SharedPreferences sunt:

- încărcarea în memorie a fișierelor de preferințe;
- citirea informațiilor din fișier - cu ajutorul metodelor de tip get;
- crearea obiectelor de tip Editor.
- **Editor** -> clasa utilizata pentru scrierea în fișierele de proprietăți. Prezinta metode de tip put care permit scrierea în fișier.
- **getSharedPreferences** -> metoda disponibila la nivelul unei activități. Asigura crearea obiectelor de tip **SharedPreferences**. Are doi parametri de intrare, și anume:
    - numele fișierului
    - modul de acces (implicit ar trebui utilizat MODE_PRIVATE - care permite accesul asupra fișierului doar aplicației care l-a definit).
- **apply** -> metoda disponibila la nivelul clasei **Editor** asigurând salvarea efectiva în fișierul de proprietăți.

9 Stocarea persistentă a datelor – Baze de date locale

La nivelul dispozitivelor mobile există baza de date SQLite pe care aplicațiile mobile o pot folosi ca alternativă pentru stocarea îndelungată a informațiilor introduse de utilizator. Clasele și metodele specifice lucrului cu baza de date SQLite sunt următoarele:

- **SQLite** -> platforma Android utilizează SQLite, ca și sistem de gestiune al bazelor de date. Aceasta este o baza de date SQL open source, care permite stocarea datelor într-un fișier text pe dispozitivul mobil. SQLite este direct integrata în sistemul de operare, prezentând toate caracteristicile unei baze de date relaționale, iar pentru utilizarea acesteia de catre aplicațiile mobile nu este necesar folosirea unui conector, precum JDBC sau ODBC. Tipurile de date acceptate de SQLite sunt: TEXT, NUMERIC, REAL, INTEGER, BLOB.
- **Room** -> Este un framework care adaugă un nivel de abstractizare peste Sqlite pentru a facilita accesul aplicației mobile către aceasta baza de date. Framework pune la dispoziția utilizatorilor o serie de adnotări ce se pot aplica la nivelul claselor, atributelor sau metodelor. Rolul acestor adnotări este de a genera codul necesar pentru realizarea operațiilor de tip DDL și DML asupra bazei de date. Din punctul de vedere al arhitecturii Room este formata din trei componente principale, și anume: ***database*** --> este o clasa abstracta ce extinde RoomDatabase, fiind responsabila cu proiectarea bazei de date și deschiderea conexiunilor dintre aplicația mobila și SQLite. De asemenea, conține lista tabelelor și o serie de metode abstracte care returnează instanțe ale claselor adnotate cu @Dao; ***entity*** --> este reprezentata de acele clase Java adnotate cu @Entity, acestea fiind tabelele bazei de date; ***Dao*** --> este reprezentata de acele interfețe din Java adnotate cu @Dao. Scopul lor este de a realiza operațiile de tip DML asupra entităților definite anterior;
- **@Database** -> adnotare utilizata la nivelul clasei abstracte care reprezintă managerul de proiectare a bazei de date. Aceasta adnotare are trei proprietăți, și anume: entities - conține lista de clase Java adnotate cu @Entity care reprezintă tabelele bazei de date; exportSchema - variabila boolean, care asigura crearea unui fișier la nivelul dispozitivului mobil ce conține schema bazei de date, daca valoarea acesteia este true; version - variabila int ce reprezintă versiunea bazei de date.
- **@Dao** -> adnotare utilizata asupra unei interfețe Java, nu conține proprietăți.
- **@Entity** -> adnotare utilizata asupra claselor Java reprezentând structura tabelei din baza de date. Conține o proprietate de tip String ce reprezintă numele tabelei.
- **@ColumnInfo** -> este aplicata asupra câmpurilor din clasa adnotata cu @Entity, reprezentând coloana unei tabele.
- **@Insert** -> este aplicata asupra unei metode din cadrul interfeței adnotate cu @Dao, prin intermediul căreia la execuție compilatorul extrage scriptul SQL pentru operația de inserare.
- **@Update** -> este aplicata asupra unei metode din cadrul interfeței adnotate cu @Dao, prin intermediul căreia la execuție compilatorul extrage scriptul SQL pentru operația de modificare.
- **@Delete** -> este aplicata asupra unei metode din cadrul interfeței adnotate cu @Dao, prin intermediul căreia la execuție compilatorul extrage scriptul SQL pentru operația de ștergere.

- **@Query** -> este aplicata asupra unei metode din cadrul interfeței adnotate cu @Dao. Are o singura proprietate, care este obligatorie, ce conține cod SQL, reprezentând comanda de selecție pe care o apelează Room la execuție.
- **@TypeConverter** -> este aplicata asupra unor metode. Asigura conversia intre doua tipuri de date neomogene. Este utilizata în special pentru conversia de la Date la Long/int/String și invers.
- **@TypeConverters** -> este aplicata asupra clasei abstracte care extinde RoomDatabase. Are o singura proprietate care prezinta lista de convertori pe care Room ar trebui sa-i aplice în momentul executării operațiilor de tip select, insert și update.

10 Implementarea aplicației mobile - Grafică

Pentru implementarea unui grafic bidimensional sunt necesare următoarele metode/obiecte:

- **onDraw**-> metoda ce aparține clasei View. Asigura desenarea componentelor vizuale pe ecranul dispozitivului mobil. Este metoda suprascrisa în momentul în care se dorește definirea unei componente vizuale personalizate.
- **canvas** -> obiect utilizat pentru desenarea graficelor. Este inițializat prin clasa View, fiind parametru de intrare al metodei onDraw
- **getHeight** -> metoda disponibila la nivelul clasei Canvas. Oferă informații despre înălțimea dispozitivului mobil
- **getWitdh** -> metoda disponibila la nivelul clasei Canvas. Oferă informații despre lățimea dispozitivului mobil

Aplicațiile android pot utiliza hărțile puse la dispoziție de Google.

11 Tutoriale suport pentru curs

11.1 Tutorial 1 - Creare proiect cu meniu NavigationDrawer

Scopul acestui tutorial este de a realiza o aplicație mobila care sa conțină toate setările necesare pentru implementarea unui meniu navigabil. Pentru realizarea acestuia să se parcurgă următorii pași:

1. Creare proiect

Crearea proiectul se poate face prin File->New->New Project.

2. Adăugare activitate de tip meniu

In ecranul pentru selecția tipului activității principale să se aleagă opțiunea 'Navigation Drawer Activity'.

3. Detalii aplicație mobila

Completați următoarele câmpuri:

- 'Application Name' - **DamNavigationDrawer**, reprezentând numele aplicației;
- 'Package name' - **eu.ase.ro.dam**, fiind numele pachetului principal în care se vor adaugă toate clasele Java ale proiectului.

La final apăsați **Finish**.

4. Modificare MainActivity

Deschideți clasa java/eu/ase/ro/dam/MainActivity. Modificați conținutul acesteia astfel:

```
public class MainActivity extends AppCompatActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
  }
}
```

5. Ștergere pachet ui [Optional]

Din java/eu/ase/ro/dam sa se elimine pachetul **ui**, daca exista.

6. Ștergere navigation [Optional]

Din res sa se elimine directorul **navigation**, daca exista.

7. Ștergere fișier main.xml

Din res/menu sa se elimine fișierul **main.xml**.

8. Ștergere fișiere fragment_*.xml

Din res/layout sa se elimine fișierele, dacă există:

- fragment_gallery.xml
- fragment_home.xml

- fragment_share.xml
- fragment_send.xml
- fragment_slideshow.xml
- fragment_tools.xml

9. Conținut activity_main.xml

Deschideți fișierul **activity_main.xml** din res/layout. Conținutul acestuia ar trebui sa fie:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.drawerlayout.widget.DrawerLayout
 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:id="@+id/drawer_layout" android:layout_width="match_parent"
 android:layout_height="match_parent" android:fitsSystemWindows="true" tools:openDrawer="start">
  <include
   layout="@layout/app_bar_main"
   android:layout_width="match_parent"
   android:layout_height="match_parent"/>

  <com.google.android.material.navigation.NavigationView
   android:id="@+id/nav_view"
   android:layout_width="wrap_content"
   android:layout_height="match_parent"
   android:layout_gravity="start"
   android:fitsSystemWindows="true"
   app:headerLayout="@layout/nav_header_main"
   app:menu="@menu/activity_main_drawer"/>
</androidx.drawerlayout.widget.DrawerLayout>
```

10. Conținut app_bar_main.xml

Deschideți fișierul **app_bar_main.xml** din res/layout. Conținutul acestuia ar trebui sa fie:

```
<?xml version="1.0" encoding="utf-8"?>
<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>

  <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" />

 <include layout="@layout/content_main" />
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

11. Modificare content_main.xml

Deschideți fișierul **content_main.xml** din res/layout. Conținutul acestuia ar trebui sa fie:

```
<?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:showIn="@layout/app_bar_main">
 <FrameLayout
  android:id="@+id/main_frame_container"
  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>
```

12. Conținut nav_header_main.xml

Deschideți fișierul **nav_header_main.xml** din res/layout. Conținutul acestuia ar trebui sa fie:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto" android:layout_width="match_parent"
 android:layout_height="@dimen/nav_header_height" android:background="@drawable/side_nav_bar"
 android:gravity="bottom" android:orientation="vertical"
 android:paddingLeft="@dimen/activity_horizontal_margin"
 android:paddingTop="@dimen/activity_vertical_margin"
 android:paddingRight="@dimen/activity_horizontal_margin"
 android:paddingBottom="@dimen/activity_vertical_margin"
 android:theme="@style/ThemeOverlay.AppCompat.Dark">
 <ImageView
  android:id="@+id/imageView"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:contentDescription="@string/nav_header_desc"
  android:paddingTop="@dimen/nav_header_vertical_spacing"
  app:srcCompat="@mipmap/ic_launcher_round" />

 <TextView
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:paddingTop="@dimen/nav_header_vertical_spacing"
```

```
    android:text="@string/nav_header_title"
    android:textAppearance="@style/TextAppearance.AppCompat.Body1" />

  <TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/nav_header_subtitle" />
</LinearLayout>
```

13. Modificați activity_main_drawer.xml

Deschideți fișierul **activity_main_drawer.xml** din res/menu. Conținutul acestuia ar trebui sa fie:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools" tools:showIn="navigation_view">
  <group android:checkableBehavior="single">
    <!-- adaugati <item> pentru fiecare optiune pe care o doriti în meniul lateral-->
    <item
      android:id="@+id/main_nav_home"
      android:icon="@android:drawable/ic_menu_camera"
      android:title="@string/main_nav_home_title" />
  </group>
</menu>
```

14. Adăugare meniu în MainActivity

Deschideți clasa MainActivity din pachetul eu.ase.ro.dam. Modificați conținutul astfel:

```
public class MainActivity extends AppCompatActivity {

  DrawerLayout;


  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    configNavigation();
  }

  private void configNavigation() {
    Toolbar = findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);
    drawerLayout = findViewById(R.id.drawer_layout);
    ActionBarDrawerToggle = new ActionBarDrawerToggle(
        this,
        drawerLayout,
        toolbar,
        R.string.navigation_drawer_open,
        R.string.navigation_drawer_close);
    drawerLayout.addDrawerListener(actionBarDrawerToggle);
    actionBarDrawerToggle.syncState();
  }
}
```

15. Rulați aplicația

Aplicația o puteți instala fie pe un emulator, fie pe un dispozitiv mobil cu sistem de operare Android.

11.2 Tutorial 2 - Adăugarea unui meniu clasic la nivelul unei activități

Pentru adăugarea unui meniu clasic la nivelul unei activități să se parcurgă următorii pași:

1. Click dreapta pe folder-ul res -> New -> Android Resource Directory -> Resource type: menu
2. Click dreapta pe directorul menu (aflat în res) -> New -> Menu resource file -> în File name introduceți numele fișierului
3. In fișierul creat anterior adăugați item-urile dorite.
4. In clasa Java asociata activității se suprascrie metoda onCreateOptionsMenu (CTRL+O pentru căutarea metodei și suprascrierea ei în clasa)
5. In onCreateOptionsMenu: getMenuInflate().inflate(<calea către fișierul din res/menu>, menu);

11.3 Tutorial 3 - Formular de adăugare transfer de date între activități

Scopul acestui tutorial este de a prezenta pașii necesari pentru preluarea informatiilor de la utilizator prin intermediul unui formular de adăugare si transferul acestora catre o activitate principala.

Definiții

- **Formular de adăugare:**
  - este o activitate normala care are un singur scop, si anume: preluare tuturor informatiilor de la utilizator necesare unei alte activități din aplicație.
  - ciclul de viață este scurt
  - este formata din componente vizuale necesare preluării de informații precum: TextInputEditText, Spinner, RadioButton, CheckBox, EditText etc.
  - este deschisa prin intermediul unui eveniment declanșat în cadrul unei activități

Nota

- **MainActivity** - denumim activitatea la nivelul căreia dorim sa încărcăm informațiile preluate de la utilizator
- **SecondActivity** - denumim formularul de adăugare deschis din MainActivity si care conține toate componentele vizuale necesare pentru preluarea informatiilor dorite

1. Creare ActivityResultLauncher

Deschideți **MainActivity** pentru a defini un obiect de tipul *ActivityResultLauncher* astfel: private ActivityResultLauncher activityResultLauncher;

Pentru înregistrarea unui Launcher este necesar sa se definească următoarele elemente: **ActivityResultContracts.StartActivityForResult** si **ActivityResultCallback**

1.1. Creare ActivityResultContracts.StartActivityForResult

```
ActivityResultContracts.StartActivityForResult contract = new ActivityResultContracts.StartActivityForResult();
```

1.2. Creare ActivityResultCallback

```
ActivityResultCallback<ActivityResult> callback = new ActivityResultCallback<ActivityResult>() {
    @Override
    public void onActivityResult(ActivityResult result) {
        if (result.getResultCode() == RESULT_OK && result.getData() != null) {
          //TODO procesati infromatiile din result.getData (Intent) care au fost trimise din SecondActivity
        }
    }
};
```

1.3. Inregistrare Launcher

```
activityResultLauncher = registerForActivityResult(contract, callback);
```

2. Pornire SecondActivity prin intermediul Launcherului

In **MainActivity** se identifica un eveniment la nivelul căruia se pornește **SecondActivity**.In cadrul acestui tutorial se folosește onClickListener.

```
@NonNull
private View.OnClickListener getOnClickListener() {
    return new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Intent = new Intent(getApplicationContext(), SecondActivity.class);
            //pornire activitate secundara
            //TODO adaugare informatii extra daca este cazul
            //intent.putExtra("KEY", value);
            addStudentLauncher.launch(intent);
        }
    };
}
```

3. Preluare Intent in SecondActivity

Se deschide activitatea de tip formular **SecondActivity**. La nivelul acesteia trebuie preluat intent-ul trimis in metoda *launch* din **MainActivity** pentru:

- a citi informațiile pasate din activitatea principala (daca este cazul)
- pentru a trimite mai târziu informațiile preluate de la utilizator

```
private Intent intent;

@Override
```

```
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_add);
   //TODO initializare componente
   intent = getIntent();
   //TODO preluare informatii din intent daca este cazul.
}
```

4. Transmitere informatii catre MainActivity

Într-un eveniment atașat unui buton din SecondActivity se vor adaugă următoarele linii pentru a transmite obiectul creat pe baza informatiilor preluate de la utilizator.

```
public static final String SERIALIZABLE_OBJECT_KEY = "SERIALIZABLE_OBJECT_KEY";

@NonNull
private View.OnClickListener getSaveClickListener() {
   return new View.OnClickListener() {
      @Override
      public void onClick(View v) {
         if (isValid()) {
            SerializableObject object = buildSerializableObject();
            //Atasarea obiectului create pe intent-ul primit din MainActivity
            intent.putExtra(SERIALIZABLE_OBJECT_KEY, object);
            //trimiterea intentul catre MainActivity impreuna cu un resultCode
            setResult(RESULT_OK, intent);
            //distrugerea formularului si revenirea catre MainActivity
            finish();
         }
      }
   };
}
```

Bibliografie

1. M. L. Murphy, The Busy Coder's Guide to Android Development, CommonsWare, 2018
2. P. Pocatilu, Programarea dispozitivelor mobile, Editura ASE, 2012
3. P. Pocatilu, I. Ivan ș.a. – Programarea aplicațiilor Android, Editura, ASE, 2015
4. Android Developers, https://developer.android.com