Szegedi Tudományegyetem
Természettudományi és Informatikai Kar

Informatikai Intézet
Számítógépes Optimalizálás Tanszék (?)

Android oktatóalkalmazás Java programnyelvhez

Diplomamunka

Készítette: Gáspár Tamás
Programtervező Informatikus MSc hallgató

Témavezető: Dr. Csendes Tibor
Egyetemi tanár
Számítógépes Optimalizálás Tanszék

Szeged, 2021

Tartalomjegyzék

1. Bevezető

...

2. A diplomamunka célja

A cél egy olyan Android eszközökön futó alkalmazás, ami segítséget nyújt a *Java* programozási nyelv megtanulásához. Ehhez két komponensre van szükség: a konkrét alkalmazásra, mely képes valamilyen standard formában megadott tananyagot megjeleníteni, és magára a tananyagra.

2.1. Követelmények a tananyaggal kapcsolatban

A tananyagnak strukturáltnak kell lennie, hogy a felhasználó könnyen tudjon navigálni benne. A programozás alapjaitól kell kezdődnie, hogy teljesen kezdők számára is használható lehessen. Fontos, hogy a tananyag csak olyan ismeretekre hivatkozzon, amik már a korábbi fejezetekben be lettek mutatva. Lenniük kell számonkéréseknek is, amelyekkel a felhasználó megbizonyosodhat róla, hogy megértette-e az anyagot.

2.2. Az alkalmazás követelményei

Az alkalmazásnak képesnek kell lennie, hogy a tananyagot a struktúráját megőrizve megjelenítse. A programozás oktatásánál elengedhetetlenek a kódminták, ezeket formázottan és könnyen olvashatóan kell mutatni (tabulálás, színezés). A felhasználó által kitöltött számonkéréseket ki kell tudni értékelni, a helyes és helytelen válaszokat pedig jelölni.

3. A tananyag struktúrája

A tananyagot kurzusokra, fejezetekre, feladatokra és vizsgákra osztottam. A következő alfejezetekben részletesen leírom, hogy ezek mit takarnak.

3.1. XML

Az *XML* (*Extensible Markup Language*) egy jelölőnyelv, ami alkalmas strukturált adattárolásra. Az *XML* fájlok egymásba ágyazott *tag*ekbből állnak.

Egy példa *XML*-ben megadott adatra, ami egy azonosító és egy név *tag*et tartalmaz:

```
<root>
  <id>33</id>
  <name>XML</name>
</root>
```

A beépített *Android* erőforrástípusok *XML*-t használnak, ezért számomra is természetes volt, hogy ebben kódolom el a tananyag erőforrásfájljait.

A továbbiakban az *XML* dokumentumokban azokat a részeket, ahol nem konkrét adatot kell érteni [és] jelek közé fogom írni, ezáltal általánosabban adhatom meg a struktúrákat. A fenti *XML* így átírva:

```
<root>
  <id>[Azonosító]</id>
  <name>[Név]</name>
</root>
```

3.2. Kurzus

A kurzus a legnagyobb egység, a *Java* nyelv egy nagy területéhez kapcsolódó ismereteket tartalmazza. Ezek az ismeretek lehetnek egyszerűek (a tananyag elején), vagy magas szintűek, amelyek már a korábbi kurzusokra épülnek (jellemzően a későbbi kurzusok).

Külön kurzusba vettem például az adatszerkezeteket, vagy a generikus programozást. Természetesen ezek a területek önmagukban is rengeteg ismeretet tartalmaznak, ezért további felosztásra van szükség, ezért bevezettem a fejezeteket.

Minden kurzus tartalmaz egy nevet és egy egyedi azonosítószámot, méghozzá úgy, hogy ezek a számok növekvő sorrendbe állítva meghatározzák a kurzusok sorrendjét.

Ezen kívül minden kurzus tartalmazza, hogy mely fejezetek (3.3), feladatok (3.4) és vizsga (3.5) tartozik hozzájuk.

A fentiek alapján egy kurzust leíró *XML* fájl a következőképpen néz ki:

```
<resources>
  <coursedata>
    <id>[Kurzus azonosító]</id>
    <name>[Kurzusnév]</name>
    <finished>[Befejezettség jelölő]</finished>
  </coursedata>

  <chapter>[Fejezet azonosító]</chapter>
  ...
  <chapter>[Fejezet azonosító]</chapter>

  <task>[Feladat azonosító]</task>
  ...
  <task>[Feladat azonosító]</task>

  <exam>[Vizsga azonosító]</exam>
</resources>
```

Arról, hogy mit takar a befejezettség jelölő és a *<finished>* tag, a 3.7 alfejezetben írok.

3.3. Fejezet

A kurzus nem osztatlan egység, hanem fejezetekből épül fel. Ezek további kisebb, összefüggő részekre bontják a kurzus tartalmát, hogy az átláthatóbb és könnyebben elsajátítható legyen.

Például az adatszerkezetek kurzus fejezetei a következők:

1. Tömbök.
2. A tömbök hátrányai.
3. Listák.
4. Verem és sor.
5. Szótár.
6. Hasznos osztályok adatszerkezetek kezelésére.

A listából látszik, hogy vannak átkötő fejezetek is, melyek elmagyarázzák, hogy miért fontosak a korábbi, vagy későbbi fejezetek ismeretei. Például a *Tömbök* fejezet után a felhasználóban felmerülhet, hogy miért kell további adatszerkezetekkel foglalkozni. Ezért közbeiktattam egy plusz fejezetet, ami részletesen megmagyarázza, hogy milyen hátrányai vannak a tömböknek és hogy mely esetekben érdemes más adatstruktúrát választani.

A fejezetek is rendelkeznek azonosítóval és névvel. Ezen kívül tartalmazzák a fejezet törzsét, ami majd konkrétan megjelenik a felhasználónak, amikor megnyitja az adott fejezetet.

```
<resources>
  <chapterdata>
        <id>[Fejezet azonosító]</id>
    <name>[Fejezetnév]</name>
  </chapterdata>

  [Fejezet törzse]

</resources>
```

Arról, hogy mi tartozhat a törzsbe, és ezek hogyan vannak elkódolva, a 4. részben írok részletesen.

3.4. Feladat

A feladat egy önálló programozási munkát takar, ami mindig egy kurzushoz kapcsolódik. Ahhoz, hogy a felhasználó meg tudja oldani, szüksége lesz az adott kurzus és az azt megelőző kurzusok ismereteire is.

Egy kurzushoz több feladat is tartozhat, de mindegyikhez adott legalább egy. Továbbra is maradva az adatszerkezetek kurzusnál, ehhez például két feladatot mellékeltem:

- Tömb alapú lista implementálása.
- Láncolt lista implementálása.

Megjegyzem, hogy ezen feladatok megoldásának nem kell olyan hatékonynak vagy általánosnak lennie, mint egy valódi implementáció, a cél csak a két

lista alapelveinek megértése. Mivel a generikus programozás kurzus az adatszerkezetek után következik, ezért itt természetesen nem elvárás generikus listák programozása.

Egy-egy feladatnak olyan sokféle helyes implementációja van, hogy lehetetlen ellenőrizni, hogy a felhasználó megoldása megfelelő-e. Ennek ellenére szerettem volna biztosítani, hogy ha a felhasználó elakad, vagy csak összevetné a megoldását egy másikkal, akkor erre lehetősége legyen. Minden feladat mellé referencia implementációt mellékeltem, ami az adott feladat alatt megtekinthető.

A feladatok *XML* elkódolása így néz ki:

```
<resources>
  <taskdata>
    <id>[Feladat azonosító]</id>
    <name>[Feladatnév]</name>
  </taskdata>

  [Feladat törzs]

  <solution>

  [Referencia implementáció törzs]

  </solution>
</resources>
```

Arról, hogy mi tartozhat a törzsbe, és ezek hogyan vannak elkódolva, a 4. részben írok részletesen.

3.5. Vizsga

A vizsgák a tananyag valódi számonkérései, itt ugyanis a feladatokkal ellentétben lehetséges a megoldások pontos ellenőrzése. Minden kurzushoz egy vizsga tartozik, amelyben benne lehet bármi az adott kurzus fejezeteiből.

Azért, hogy az alkalmazás a válaszokat ellenőrizni tudja, megkötéseket kell tenni a kérdések típusára. Nem lehet például esszékérdés. Ezt figyelembe véve a következő kérdéstípusokat implementáltam:

- Feleletválasztós kérdés, egy választási lehetőséggel (*single choice*).
- Feleletválasztós kérdés, több választási lehetőséggel (*multi choice*).
- Szöveges kérdés, ahol a válasz egy szó vagy rövid kifejezés lehet.
- Igaz-hamis kérdés.

A vizsgákhoz kérdéshalmaz tartozik, melyből az alkalmazás véletlenszerűen választ annyi, amennyi kérdésből az adott vizsga áll (jellemzően 25-30 darabot). Időlimit is van, ami alatt a felhasználónak be kell fejeznie a kitöltést. Az idő lejárta esetén a kitöltés akkori állapota kerül kiértékelésre.

Sikeres vizsga esetén a következő kurzus elérhetővé válik. Amennyiben a vizsga nem sikeres, akkor néhány óráig nem lesz újra kitölthető. Ezt a korlátozást azért vezettem be, hogy ne lehessen a vizsgákat "*brute-force*" módon teljesíteni. Ha a felhasználó már korábban sikeresen elvégezte a vizsgát, akkor azt korlátozás nélkül bármikor újra elindíthatja. A vizsgán elért legmagasabb pontszám eltárolásra kerül.

A vizsgákat definiáló *XML* fájlok a következőképpen néznek ki:

```
<resources>
  <examdata>
        <id>[Vizsga azonosító]</id>
        <questionAmount>[Kérdések mennyisége]</questionAmount>
        <timeLimit>[Időkorlát, perceben]</timeLimit>
    <finished>[Befejezettség jelölő]</finished>
  </examdata>

  [Vizsgakérdések]

</resources>
```

Arról, hogy mit takar a befejezettség jelölő és a *<finished>* tag, a 3.7. alfejezetben írok. A kérdéstípusok részletes definiálása a 4.5. részben történik.

3.6. Előrehaladás a tananyagban

Az alkalmazás kezdeti indításakor a felhasználó csak az első kurzus fejezeteihez és feladataihoz fér hozzá. A vizsga zárolva van. El kell olvasnia az összes

fejezetet ahhoz, hogy a vizsga kitölthető legyen. A feladatok elkészítése opcionális, de ajánlott.

A vizsga sikeres teljesítésével nyitható meg a következő kurzus, majd a folyamat ismétlődik addig, amíg van további kurzus.

Korábbi fejezetek, feladatok, és vizsgák bármikor újra megtekinthetőek és kitöltetőek.

3.7. Részben befejezett kurzusok

Az alkalmazás természetesen bővíthető új kurzusokkal. Azonban egy kurzus elkészítésé hosszú időt vehet igénybe, mivel ebbe beletartozik a tananyag megírásán kívül a feladatok és vizsgakérdések összeállítása is. Ezért az alkalmazás támogatja a részben befejezett kurzusokat és vizsgákat. Ezek segítségével az alkalmazás lényegében fejezetenként bővíthető.

Azt, hogy egy kurzus be van-e fejezve, az adott kurzus *XML* dokumentumában lévő *<finished>* tag mondja meg. Ha ennek értéke hamis, akkor a felhasználó jelzést kap arról, hogy a kurzus még bővítés alatt áll. Az *XML*-ben definiált vizsgák szintén tartalmaznak *<finished>* taget. Amennyiben ez hamis, akkor a vizsga nem lesz elinítható, még akkor sem ha a felhasználó teljesítette a kurzus összes fejezetét. Az indulás helyett értesítés jelenik meg, ami tájékoztat arról, hogy a vizsga csak egy későbbi frissítés után nyílik meg.

A fejezetek és feladatok nem támogatják ezt a funkciót, mivel ezek sokkal rövidebb terjedelműek.

4. Törzstartalom, kódminták és kérdések

Már leírtam a tananyag struktúráját és azt, hogy ez hogyan van *XML*-ben elkódolva. Most bemutatom, hogy miként vannak definiálva azok a komponensek, amelyeket a felhasználó a fejezetek, feladatok és vizsgák megnyitásakor lát és amelyek a tananyag törzsét alkotják.

Az ehhez használt egyik legfontosabb komponens a *TextView* osztály, amit Android alatt szöveges tartalom megjelenítésére lehet használni. A formázott szöveg megjelenítése is támogatott, ahol a formázási szabályokat *HTML* nyelven adhatjuk meg. Ugyan a *HTML*-nek csak egy kis része használható, de ennek az alkalmazásnak ez elég, mivel a szövegstílus (dőlt, félkövér) és a betűszín változtatható.

Például a következő módon lehet megjeleníteni egy szöveget, amelyben egy félkövér szó van:

```
String rawText = "Ez egy <b>félkövér</b> szó.";
Spanned formattedText = Html.fromHtml(rawText);
textView.setText(formattedText);
```

Ezt a funkcionalitást az alkalmazás erősen kihasználja, nem csak egy-egy szó kiemelésére vagy, linkek beillesztésére, hanem a kódminták megformázására is.

Nehézséget okoz viszont, hogy a szövegek *XML*-ben vannak megadva, mivel mind az *XML*, mind a *HTML* a < és > karaktereket használja. Azért, hogy ne legyenek ütközések, és az *XML* ne próbálja meg értelmezni a *HTML* tageket, a szöveget úgynevezett karakter-adat (*CDATA*) blokkba kell helyezni:

```
<text>
 <![CDATA[
  Ez egy <b>félkövér</b> szó.
 ]]>
</text>
```

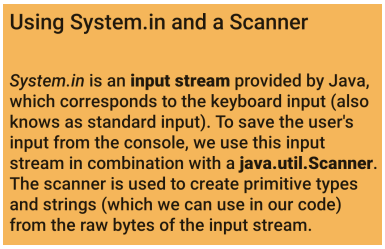
4.1. Szöveges tartalmak

A legegyszerűbb megjeleníthető komponens az egyszerű szöveg. Ilyet úgy helyezhetünk el egy fejezetben, vagy feladatban, hogy $< text >$ tagek közé helyezzük a formázott szöveget, használva az előző részben említett *CDATA* blokkot. A megadott szöveg formázottan fog megjelenni az alkalmazásban.

Lehetőség van címsor beszúrására. Ez a fejezetek vagy feladatok további kisebb, összefüggő részekre való felbontását teszi lehetővé. Címsort a $< title >$ taggel szúrhatunk be, tartalmát pedig egy attribútummal lehet szabályozni:

```
<title text="Második paragrafus"/>
```

Itt látható egy rész az *input beolvasása* fejezetből, ami egy címsorból és szövegblokkból áll:



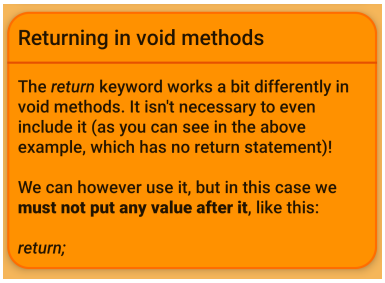
Már ezzel a két komponenssel meg lehet jeleníteni a tananyag nagy részét, de azért hogy a fejezetek kevésbé legyenek monotonok, további komponenseket is implementáltam, melyek "megtörik" a szöveget.

4.1.1. Bekeretezett szöveg

Ez a komponens lényegében egy szövegdoboz, ami eltérő háttérszínnel és saját címmel rendelkezik, ezáltal jobban felhívja a felhasználó figyelmét. Definiálása a $<boxed>$ taggel történik:

```
<boxed title="[Szövegdoboz címe]">
 <![CDATA[
   [Formázott szöveg]
 ]]>
</boxed>
```

Például a metódusok fejezetben ilyen szövegdobozba raktam a visszatérési érték nélküli metódusokkal kapcsolatos részt. Az adott fejezetet megnyitva ez a rész így fog megjelenni:



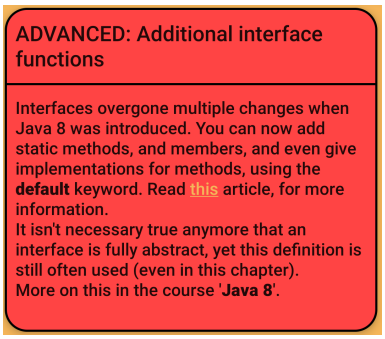
4.1.2. Magas szintű tartalom

Ez a komponens hasonló a bekeretezett szöveghez, viszont speciálisan a nehezebb anyagrészek, kiegészítő információk megjelenítésére terveztem. Az

< *advanced* > taggel adható meg. Élénkpiros színnel fog megjelenni, és a felhasználó tudomására hozza, hogy az adott tartalom nehezebb, vagy csak egy későbbi kurzusban kerül részletes bemutatásra.

```
<advanced title="[Cím]">
 <![CDATA[
   [Formázott szöveg]
 ]]>
</advanced>
```

Ilyen jelzővel láttam el például a *absztrakt osztályok és interfészek* fejezetben azt a rész, ahol a megemlítem, hogy milyen változásokon mentek keresztül az interfészek a *Java 8* megjelenésével. Ez részletesebben egy későbbi kurzusban, a *Java 8* nevűben van leírva.



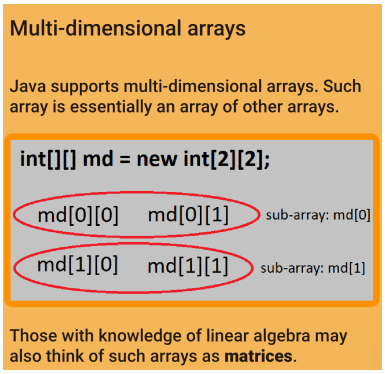
Ebben a komponensben még egy link is található, ami a megfelelő oldalra viszi a felhasználót (a telefon alapértelmezett böngészőjében).

4.2. Grafikus tartalmak

Lehetőség van képek beillesztésére a fejezetekbe és feladatokba, az < *image* > tag segítségével. Egy attribútumban kell megadni, hogy melyik kép jelenjen meg. Arról, hogy hol tárolom a tananyagban felhasznált képeket részletesen a 6 részben írok. Az *XML* struktúra a következőképpen néz ki:

```
<image name="[Kép neve]"/>
```

Például itt látható egy részlet a *tömbök* fejezetből, ahol az anyag megértését egy képpel segítem elő:



4.3. Kódminták

A programozás oktatásánál elengedhetetlen, hogy kódrészletekkel segítsük a megértést. Az alkalmazásban lévő mintáknak egységes formázásúnak kell lenniük. Ezt úgy értem el, hogy előre definiáltam a színeket, amelyekkel a kód részeit kijelöltem. A következők kaptak saját kijelölőszínt:

- A *Java* nyelv kulcsszavai.
- A *Java* nyelv primitívei. Ide soroltam a *void*-ot is, habár az valójában csak kulcsszó.
- Szöveg és karakter konstansok.
- Numerikus konstansok.
- Osztálynevek.
- Metódusok és adattagok nevei.
- Annotációk.
- Kommentek.

A megjelenő kódmintáknak ugyanolyan sortörésekkel és tabulálással kell megjelenni az alkalmazásban, mint ahogy megírásra kerültek. Ezért a minták szövegébe a sorok végére $< br >$ sortörő taget kell elhelyezni, és a tabulálást pedig a * *; szimbólummal kellet jelölni. A korábban bemutatott komponenseknél ez nem volt lényeges, ezeket az Android rendszer úgy tördelheti, ahogy a képernyő méretei megengedik, azonban a kódmintáknál fontos a helyes tördelés.

További formázást igényelt, hogy a *HTML* nyelv esetén a < szimbólumot speciálisan kell jelölni, hogy az értelmező ne vegye egy új tag kezdetének. A megoldás az *&lt* karaktersorozattal történő helyettesítés. Ezért az olyan kódminták, melyek például összehasonlítást tartalmaznak mind külön formázást igényelnek.

Látható, hogy rengeteg formázási szabályt vezettem be. Ha ehhez hozzávesszük a kódminták nagy számát, akkor látszik, hogy a manuális formázás nem célszerű. Ezért erre célra egy külön formázó programot készítettem, ami egy megadott mappában lévő *Java* kódot átalakít olyan *HTML* kóddá, ami a fent említett összes szabálynak eleget tesz. Ez a program reguláris kifejezések alapján azonosítja a listában látható kifejezéseket és elhelyezi körülöttük a színezést és a tördelést biztosító tageket.

A fejezetekben és feladatokban kódmintát a < *code* > tag segítségével definiálok. A tag belsejébe kell helyezni, azt a formázott kódot, amit az előző részben említett segédprogram ad eredményül.

```
<code>
 <![CDATA[
  [Formázott kód]
 ]]>
</code>
```

A megjelenítésnél nehézséget okoz, hogy a kódmintákban kosszú sorok lehetnek, amelyen nem minden eszköz képernyőjén férnek el egy sorban, viszont a mintákat a rendszer nem tördelheti szabadon. Ezért az alkalmazás kódmintái vízszintesen görgethetőek, ha vannak bennül olyan sorok, amelyek nem férnek el a képernyőn.

CODE

```
//create a thread using an anonymous runn
Thread t1 = new Thread(new Runnable() {
    @Override //override the run method he
    public void run() {
        System.out.println("Hello from an an
    }
});
//start the thread
t1.start();
```

CODE

```
for(int i=0; i<10; i++) {
    System.out.println(i);
}
```

A képek bal alsó sarkában lévő ikonokkal a kódminták betűmérete állítható. A jobb alsó sarokban lévő gomb aktiválja a *ClipSync* funckiót, melyről részletesebben a 8 részben írok.

4.4. Interaktív kódminták

Azért, hogy a tananyagot interaktívabbá tegyem, implementáltam a kódmintáknak olyan változatát is, ahol egyes szavak, vagy kifejezések hiányosak, kitöltésüket a felhasználó egy feladatkiírás alapján elvégezheti. Ez a komponens is *XML*-ben van definiálva, és formázott kódot vár. A formázás folyamatáról a **??**. részben írtam.

Az interaktív kódmintának meg kell adni, hogy hol legyenek benne módosítható részek, mezők, melyek tartalmát a felhasználó átírhatja. Ezek megadásához a formázott kódba három alulvonás jelet kell beszúrni. Ahol a beolvasás során ezt a karaktersorozatot találja az alkalmazás, oda egy módosítható rész fog kerülni. Például így lehet megjeleníteni egy változódeklarálást, ahol a változó értéke módosítható lesz:

```
int x = ___ ;
```

Az olvashatóság miatt itt nem tüntettem fel a formázást, egy valódi minta esetén *HTML* tagek is szerepelnek a formázott kódban.

A mezők azonosítása indexelés alapján történik. Mindegyik mezőhöz meg kell adni, hogy oda milyen megoldásokat vár az alkalmazás. Szintén megadható alapértelmezett tartalom, ilyenkor az adott mező nem üresen, hanem a megadott tartalommal jelenik meg. Ez felhasználható például olyan interaktív minta megadására, ahol a felhasználó dolga a kódrészlet kijavítása, nem pedig egyszerűen a kitöltése.

Az interaktív kódmintákat a következő *XML* struktúrával lehet megadni:

```
<interactive instruction=[Kitöltési instrukció]>
 <data>
  [Formázott kód, mezőket jelölő karakterekkel]
 </data>
 <answer place=[Index]>[Válasz]</answer>
 ...
 <answer place=[Index]>[Válasz]</answer>
</interactive>
```

Egy indexhez több $< answer >$ tag is tartozhat, ilyenkor az adott mezőnek több megoldása is van. Az opcionális alapértelmezett tartalmakat $< default >$ tagek közt kell megadni:

```
<default place=[Index]>[Alapértelmezett érték]</default>
```

Amikor a fent megadott módon elkezdtem az interaktív minták megírását, hamar rájöttem, hogy ez az eszköztár sokszor még a legegyszerűbb feladatok megadásához is kevés. Vegyük a következő példát:

Az utasítás az, hogy egészítsük ki a kapott kódmintát, méghozzá úgy, hogy a változó végső értéke legyen 8!

```
int num = ___ ;
num = num ___ 5 ;
```

Látható, hogy több megoldás is van:

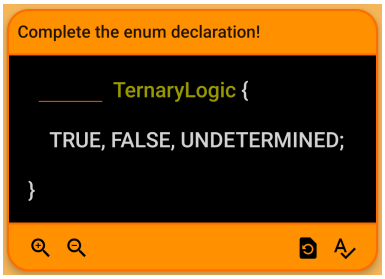
- Az első mezőbe 3, a másodikba + beírásával az eredmény 8 lesz.
- A 13 és a - jel kombinációja is helyes.
- Nyolcat kapunk továbbá a 40 és a % megadásával is.

A probléma azonban nem oldható meg annyival, hogy lehetséges megoldásként az első mezőhöz felvesszük a 3, 13 és 40 értékeket, a másodikhoz pedig a +, - és % szimbólumokat. Ebben az esetben ugyanis ezek helytelen kombinációját is elfogadná az alkalmazás, pedig csak a listában megadott kombinációk helyesek.

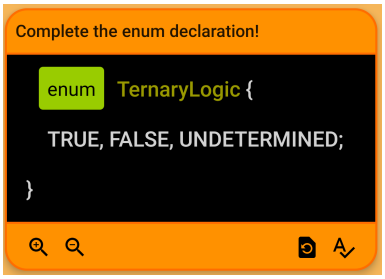
Ezt a nehézséget úgy hidaltam át, hogy csoportosíthatóvá tettem a válaszokat. Egy csoportba tartozó válaszok csak együtt lesznek elfogadva. Az előző példa válaszait így a következőképpen lehet elkódolni:

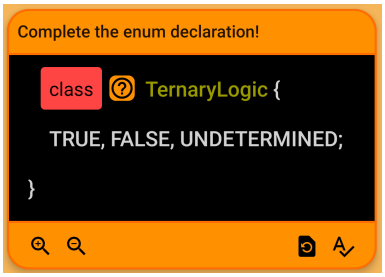
```
<answer place="0" group="add">3</answer>
<answer place="1" group="add">+</answer>
<answer place="0" group="subtract">13</answer>
<answer place="1" group="subtract">-</answer>
<answer place="0" group="div">40</answer>
<answer place="1" group="div">/</answer>
```

Most bemutatom, hogy miként jelenik meg egy interaktív minta az alkalmazáson belül. Az itt látható példa az egyszerűbbek közé tartozik, a felhasználónak egy mezőt kell benne kitöltenie:



A bal alsó sarokban lévő ikonok a 4.3 fejezetből már ismert betűméret állítók, a jobb alsó sarokban lévő gombokkal pedig a felhasználó elkérheti a megoldást, vagy alaphelyzete állíthatja a komponenst. A helyes és helytelen megoldások kijelölésre kerülnek:





4.5. Kérdések

...

4.5.1. feleletválaszós kérdés egy válaszlehetőséggel

...

4.5.2. feleletválaszós kérdés több válaszlehetőséggel

...

4.5.3. Szöveges kérdés

...

4.5.4. Igaz-hamis kérdés

...

5. Beolvasási folyamat

...

6. A tananyag tárolása

A tananyagot a forráskódtól el kell különíteni, viszont futásidőben elérhetőnek kell lennie, hogy a felhasználó kéréseit ki lehessen szolgálni. Fontos az egységes elkódolás. Például minden kurzust leíró erőforrásfájlnak azonos felépítésűnek kell lennie: meg kell mondaniuk mely fejezetek, feladatok és vizsga tartozik hozzájuk.

6.1. Android erőforráskezelés

Az *Android* alapelve, hogy az erőforrásfájlok legyenek elkülönítve a kódtól, és erre több eszközt is kínál. Az első a *Resource System*. Ebben olyan erőforrásfájlok tárolhatóak, melyekre a legtöbb alkalmazásnak szüksége van:

- Grafikus elemek (*drawable*): lehetnek ikonok, képek, hátterek, stb.
- Szövegek (*string*): a felhasználói felületen megjelenő feliratok.
- Elrendezések (*layout*): a felhasználói felület elrendezését tárolják.
- Még sok további előre definiált kategória: stílusok, színek, stb.

A *Resource System* nem a legmegfelelőbb mód a tananyag tárolására, mivel az egyetlen gyakori, előre definiált kategóriába sem esik.

Biztosított továbbá az *Asset System*, mely annyiban tér el az előbb bemutatott rendszertől, hogy itt nincsenek kategóriák, be tud fogadni bármilyen erőforrásfájlt, amelyre egy alkalmazásnak szüksége lehet. Ez a legalkalmasabb módszer a tananyag tárolására.

7. Adatbázis

Az alkalmazásnak képesnek kell lennie eltárolnia felhasználó előrehaladását a tananyagban. A következő adatokat kell elmenteni:

- Teljesített vizsgák: Ez az információ a legfontosabb, mivel meghatározza, hogy melyik kurzusok elérhetőek.
- Elolvasott fejezetek: Mivel egy kurzus vizsgája csak akkor nyílik meg, ha a felhasználó minden fejezetet elolvas az adott kurzusból.
- Teljesített feladat: A feladatok teljesítése opcionális, de az alkalmazás ezt is eltárolja.

A 6 fejezetben említett *Resource System* és *Asset System* futásidőben csakis fájl olvasást tesznek lehetővé, írást nem. Ezért egy relációs adatbázist használok. *NoSQL* adatbázis is lehetne, arról hogy miért a választott módszer mellett döntöttem, a 7.3 fejezetben írok.

7.1. Státusz annotáció

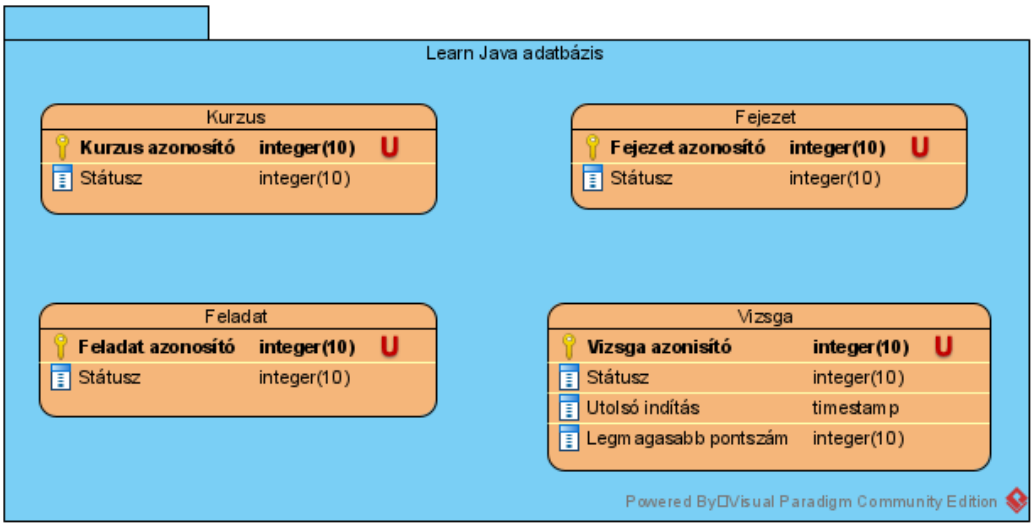
Az előrehaladást a programban és az adatbázisban is egységesen jelölöm. Erre egy saját *Java* annotációt használok:

```
@IntDef
public @interface Status {

        int LOCKED = 0;

        int UNLOCKED = 1;

        int COMPLETED = 2;
}
```

Az *@IntDef* annotáció listázza, hogy mik lehetnek a lehetséges értékei egy olyan változónak, melyek *@Status* annotációval látunk el. Ez fordítási időben ellenőrzésre kerül. Például ha a felhasználó először teljesít egy vizsgát, akkor annak státusza *UNLOCKED*-ról *COMPLETED*-re for változni az adatbázisban.

7.2. Struktúra

Az adatbázis szerkezete egyszerű. A tananyag négy egységéhez (kurzus, fejezete, feladat, vizsga) rendel egy-egy táblát, és ezen táblák rekordjai tárolják az előrehaladást. Így néz ki az egyed-kapcsolat diagram:



Táblák közti kapcsolatra, külső kulcsokra nincs szükség. Az, hogy mely fejezetek és feladatok tartoznak egy-egy kurzushoz nem az adatbázisban van eltárolva, hanem a tananyagot definiáló *XML* dokumentumokban (mivel nem változik dinamikusan).

Az azonosítók megegyeznek az *XML* dokumentumokban használt azonosítókkal. A státusz mezők mindig a 7.1 részben definiált annotáció valamelyik értékét tartalmazzák.

A vizsgák esetén vannak további dinamikusan változó adatok:

- Az utolsó indítás időpontja: Ez szükséges, hogy a 3.5 részben említett indítási korlátot be lehessen tartani.
- Legmagasabb pontszám: Dinamikusan változó, ezért az adatbázisban kell tárolni.

7.3. Megvalósítás: SQLite és Room

Az Android rendszer minden alkalmazás számára biztosít egy *SQLite* relációs adatbázist. Számomra egyértelmű volt, hogy ezt fogom használni. Az

SQLite egy olyan adatbázis kezelő, amely nem kliens-szerver alapú, hanem közvetlenül az őt használó alkalmazásba van beágyazva.

Ennek előnye, hogy nem kell hálózati kapcsolat az adatbázis eléréséhez, és nem kell szervert üzemeltetni. Hátránya viszont, hogy minden adat lokálisan kerül eltárolásra. Ennél az alkalmazásnál ez nem jelent gondot, mivel viszonylag kevés rekord keletkezik (a tananyag minden eleméhez egy).

Az alkalmazás saját adatbázisának közvetlenül is küldhetünk üzenetet, ez azonban több okból sem javasolt:

- Szintaktikus hibák: ha a kiadott *SQL* utasításban szintaktikus hibát vétünk, az a program helytelen működéséhez, összeomlásához vezet. Ez nincs ellenőrizve.
- Érvénytelen utasítások: ha olyan utasítást adunk ki, amelyet az *SQLite* nem tud végrehajtani, akkor is helytelen viselkedés fog történni. Például, ha olyan rekordot szúrunk be, ami sérti az egyediség feltételt.
- Adatbázis frissítés a főszálon: ha a főszálon adunk ki *SQL* utasítást, akkor ez blokkolni fog, és a felhasználói felület nem lesz válaszképes.

Pontos tervezéssel és helyes implementációval ez mind elkerülhető, de javasolt egy olyan keretrendszer használata, mely az adatbázis kezelés veszélyeit kezeli. Egy ilyen keretrendszer a *Room*, amely az Android fejlesztői könyvtár (*AndroidX*) része.

A *Room* elrejti a programozó elől az *SQLite* adatbázist, helyette két komponenst biztosít, melyekkel a struktúrája és tartalma módosítható. Ezen kívül meg fogja tiltani, hogy a főszálról adatbázis műveletet végezzünk (ilyen esetben kivétel dobódik).

7.3.1. Room entitások

Az entitások olyan *@Entity* annotációval jelölt Java osztályok, melyek megadják az adatbázis tábláinak struktúráját. Például a fejezet táblát a következőképpen definiáltam:

```
@Entity(tableName = "chapter_status")
public class ChapterStatus {

 @PrimaryKey //külső kulcs jelző
```

```
 @ColumnInfo(name = "chapter_id")
 private int chapterId;

 @ColumnInfo(name = "status")
 @Status //saját státusz annotáció
 private int status;

 //konstruktor...
}
```

Látható, hogy annotációk segítségével adhatjuk meg, hogy milyen mezői legyenek az adott táblának. Az alkalmazás indításakor ez a tábla létre fog jönni az adatbázisban, anélkül, hogy egyetlen *SQL* utasítást is írnunk kellene. Azokat a *Room* fogja összeállítani, az általunk megadott adatok alapján, és garantáltan helyesek lesznek.

A későbbiekben ezt az osztályt példányosíthatjuk, ezek az objektumok rekordokat fognak reprezentálni.

7.3.2. Room adatkezelők

Az adatkezelő objektumok olyan *@Dao* annotációval jelölt Java interfészek, amikkel az előző részben definiált táblák tartalmát módosíthatjuk. Például a fejezet táblához a következő módosító interfészt készítettem:

```
@Dao
public interface ChapterDao {

 @Insert //a beszúró SQL utasítás automatikusan generálódik
 void addChapterStatus(ChapterStatus chapterStatus);

 @Update //a frissítő SQL utasítás automatikusan generálódik
 void updateChapterStatus(ChapterStatus chapterStatus);

 @Query("SELECT * FROM chapter_status WHERE chapter_id == :chapterId")
 ChapterStatus queryChapterStatus(int chapterId);
}
```

Bizonyos egyszerű műveletekre vannak előre adott annotációk, melyek automatikusan fogják generálni a a megfelelő *SQL* utasítást (beszúrás, frissítés, törlés). Ezek azonban nem elegendőek, ezért a *@Query* annotációval

saját utasítást írhatunk. Ennek a tartalma fordítási időben ellenőrizve lesz. Megfigyelhető, hogy még arra is lehetőség van, hogy a kapott paraméterek értékét behelyettesítsük az *SQL* utasításba.

Egy példa arra, hogy hogyan kombinálhatjuk ezeket az osztályokat az adatbázis módosítására:

```
//egy új szálon
new Thread(() -> {
 ChapterDao dao = ... ;
 ChapterStatus cs = new ChapterStatus(id, Status.UNLOCKED);
 //beszúrás
 dao.addChapterStatus(cs);
}).start();
```

8. ClipSync funkció

...