



Miodrag Živković

RAZVOJ MOBILNIH APLIKACIJA

ANDROID JAVA PROGRAMIRANJE

Beograd, 2020.

UNIVERZITET SINGIDUNUM

Miodrag Živković

RAZVOJ MOBILNIH APLIKACIJA

Android Java programiranje

Prvo izdanje

Beograd, 2020.

RAZVOJ MOBILNIH APLIKACIJA

Android Java programiranje

Autor:

dr Miodrag Živković

Recenzenti:

Prof. Aleksandar Jevremović, redovni profesor, Univerzitet Singidunum

Prof. Nebojša Bačanin Džakula, vanredni profesor, Univerzitet Singidunum

Prof. Boško Nikolić, redovni profesor, Elektrotehnički fakultet Univerziteta u Beograd
u

Izdavač:

UNIVERZITET SINGIDUNUM

Beograd, Danijelova 32

www.singidunum.ac.rs

Za izdavača:

dr Milovan Stanišić

Priprema za štampu:

Miodrag Živković

Dizajn korica:

Aleksandar Mihajlović

Godina izdanja:

2020.

Tiraž:

700 primeraka

Štampa:

Caligraph, Beograd

ISBN: 978-86-7912-719-8

Copyright:

© 2020. Univerzitet Singidunum

Izdavač zadržava sva prava.

Reprodukacija pojedinih delova ili celine ove publikacije nije dozvoljena.

Sadržaj

Lista slika	VII
Predgovor	XIII
1. Uvod	1
1.1. Istorijat	2
1.1.1. Prve verzije	3
1.1.2. Android 1.5 Cupcake	4
1.1.3. Android 1.6 Donut.....	5
1.1.4. Android 2.0 Eclair	6
1.1.5. Android 2.2 Froyo.....	7
1.1.6. Android 2.3 Gingerbread.....	8
1.1.7. Android 3.0 Honeycomb.....	10
1.1.8. Android 4.0 Ice Cream Sandwich.....	12
1.1.9. Android 4.1 Jelly Bean	13
1.1.10. Android 4.4 KitKat	14
1.1.11. Android 5.0 Lollipop	16
1.1.12. Android 6.0 Marshmallow.....	17
1.1.13. Android 7.0 Nougat	19
1.1.14. Android 8.0 Oreo	20
1.1.15. Android 9.0 Pie	21
1.1.16. Android 10.....	23
1.2. Rasprostranjenost Android verzija na tržištu	25
1.3. Udeo Android operativnog sistema na tržištu mobilnih uređaja.....	27
1.4. Tržište mobilnih aplikacija	29
2. Android platforma.....	34
2.1. Arhitektura Android platforme	34
2.2. Android Runtime	36

2.3.	Android aplikacije	39
2.4.	Okruženje za razvoj	41
2.5.	Android uređaji i emulatori	43
3.	Android Studio.....	47
3.1.	Instalacija okruženja	47
3.2.	Kreiranje i struktura projekta	55
3.3.	Konfigurisanje Emulatora i pokretanje aplikacije.....	66
3.4.	SDK Manager	72
4.	Grafički korisnički interfejs	76
4.1.	Klasa View.....	77
4.2.	Standardne komponente grafičkog interfejsa.....	79
4.2.1.	TextView	82
4.2.2.	EditText.....	84
4.2.3.	Button.....	87
4.2.4.	CheckBox	90
4.2.5.	RadioButton.....	95
4.2.6.	Spinner.....	97
4.2.7.	Toast	101
4.2.8.	DatePicker i TimePicker	104
4.2.9.	Dialog.....	106
4.3.	Rasporedi komponenti (layout).....	110
4.3.1.	Linearni raspored.....	112
4.3.2.	Relativni raspored.....	116
4.3.3.	Ograničeni raspored	121
4.3.4.	ListView	122
4.4.	Layout editor	124
4.5.	Klasa R	126
4.6.	Rukovanje događajima	128

4.7.	Pitanja za vežbu.....	131
5.	Aktivnosti.....	133
5.1.	Konfiguriranje manifesta.....	134
5.2.	Životni ciklus aplikacije	135
5.3.	Životni ciklus aktivnosti	138
5.4.	Čuvanje stanja korisničkog interfejsa	146
5.5.	Promena orijentacije ekrana	149
5.6.	Intenti (namere)	153
5.6.1.	Intent objekat	154
5.6.2.	Eksplicitne i implicitne namere.....	155
5.6.3.	Intent filteri i rezolucija Intenta.....	160
5.7.	Back Stack.....	161
5.8.	Pitanja za vežbu.....	165
5.9.	Zadaci za vežbu.....	166
6.	Niti	181
6.1.	AsyncTask	183
6.2.	Pitanja za vežbu.....	186
6.3.	Zadaci za vežbu.....	187
7.	Dozvole	199
7.1.	Tipovi dozvola.....	199
7.2.	Deklaracija privilegija u manifestu	201
7.3.	Potvrda opasnih dozvola	202
7.3.1.	Potvrda za vreme izvršavanja	202
7.3.2.	Potvrda za vreme instalacije.....	203
7.4.	Provera dozvola.....	204
7.5.	Zahtevanje dozvola.....	204
7.6.	Pitanja za vežbu.....	208
7.7.	Zadaci za vežbu.....	209

8.	Fragmenti	222
8.1.	Životni ciklus fragmenta	223
8.2.	Iscrtavanje korisničkog interfejsa fragmenta	226
8.3.	Dodavanje fragmenta aktivnosti	226
8.3.1.	Deklaracija fragmenta unutar fajla sa rasporedom komponenti aktivnosti	227
8.3.1.	Programsko dodavanje fragmenta	228
8.4.	Transakcija fragmenta	230
8.5.	Komunikacija fragmenta sa aktivnosti domaćinom	231
8.6.	Pitanja za vežbu	236
8.7.	Zadaci za vežbu	237
9.	Perzistencija podataka	243
9.1.	SharedPreferences	243
9.2.	Interni prostor	245
9.3.	Eksterni prostor	246
9.4.	SQLite baza podataka	248
9.4.1.	Definisanje šeme i ugovora	248
9.4.2.	Kreiranje baze pomoću SQL pomoćne klase	249
9.4.3.	Ubacivanje podataka u bazu	251
9.4.4.	Čitanje podataka iz baze	252
9.4.5.	Brisanje podataka iz baze	253
9.4.6.	Ažuriranje podataka u bazi	254
9.4.7.	Perzistencija konekcije	255
9.5.	Room biblioteka	256
9.6.	Pitanja za vežbu	260
9.7.	Zadaci za vežbu	261
10.	BroadcastReceiver komponente	281
10.1.	Sistemska obaveštenja	281

10.2.	Prijem obaveštenja.....	282
10.2.1.	Deklaracija prijemnika kroz manifest	282
10.2.2.	Programska registracija prijemnika.....	284
10.3.	Slanje obaveštenja.....	286
10.4.	Sigurnosna razmatranja i praktične primene	287
10.5.	Pitanja za vežbu	289
10.6.	Zadaci za vežbu.....	290
11.	Servisi.....	308
11.1.	Životni ciklus servisa	308
11.2.	Deklaracija servisa u manifestu.....	311
11.3.	Startovani servis	312
11.3.1.	IntentService klasa.....	313
11.3.2.	Service klasa	314
11.3.3.	Pokretanje i zaustavljanje servisa.....	318
11.4.	Vezani servis	319
11.5.	Pitanja za vežbu	326
11.6.	Zadaci za vežbu.....	327
12.	Praktični primeri upotrebe	350
12.1.	Kamera.....	350
12.2.	Lokacija	353
12.3.	Senzori	356
12.4.	Rad sa Google mapama	360
	Literatura	369

Lista slika

Slika 1.1, prvi komercijalno dostupni Android telefon HTC Dream.....	2
Slika 1.2, Android 1.5 Cupcake glavni ekran.....	4
Slika 1.3, Android 1.6 Donut glavni ekran	5
Slika 1.4, Android 2.0 Eclair poboljšana navigacija i funkcija speech-to-text.....	7
Slika 1.5, Android 2.2 Froyo glavni ekran	8
Slika 1.6, Android 2.3 Gingerbread glavni ekran	9
Slika 1.7, Android 3.0 Honeycomb glavni ekran	11
Slika 1.8, Android 3.0 Honeycomb Recent apps.....	11
Slika 1.9, Android 4.0 Ice Cream Sandwich glavni ekran i Recent Apps.....	12
Slika 1.10, Android 4.1 Jelly Bean korisnički interfejs	14
Slika 1.11, Android 4.4 KitKat korisnički interfejs.....	15
Slika 1.12, Android 5.0 Lollipop korisnički interfejs i materijalni dizajn	16
Slika 1.13, Android 6.0 Marshmallow korisnički interfejs	18
Slika 1.14, Android 7.0 Nougat korisnički interfejs i nova opcija podeljenog ekrana.....	19
Slika 1.15, Android 8.0 Oreo korisnički interfejs i nova opcija slike u slici	21
Slika 1.16, Android 9.0 Pie nova opcija pozicioniranja u zatvorenom prostoru	22
Slika 1.17, Android 10 kontrola aplikacija koje pristupaju lokaciji	23
Slika 1.18, Android 10 podrška za savitljive ekrane	24
Slika 1.19, raspodela različitih verzija Androida na tržištu	25
Slika 1.20, raspodela veličine ekrana Android uređaja na tržištu	27
Slika 1.21, iOS, Android, BlackBerry i Symbian operativni sistemi.....	28
Slika 1.22, ukupan broj preuzimanja aplikacija na svim platformama	30
Slika 1.23, ukupan broj besplatnih i plaćenih preuzimanja u intervalu 2011-2017.....	31
Slika 1.24, uporedni broj preuzimanja sa Google Play i App Store	32
Slika 1.25, broj aplikacija dostupnih na Google Play Store	33
Slika 2.1, Android softverski stek	34
Slika 2.2, uporedni prikaz JVM i Dalvik VM	37
Slika 2.3, JIT arhitektura	38
Slika 2.4, izvršavanje Android aplikacija u zasebnim Linux procesima.....	40
Slika 2.5, struktura APK	41
Slika 2.6, Android Studio IDE	42
Slika 2.7, proces kreiranja aplikacije u Android Studio IDE	43
Slika 2.8, skrivena sekcija Developer options.....	44

Slika 2.9, Android Emulator sa dodatnim funkcionalnostima	45
Slika 3.1, verzija Android Studio okruženja korišćena u ovoj knjizi.....	47
Slika 3.2, početni ekran procesa instalacije.....	48
Slika 3.3, odabir komponenti za instalaciju	48
Slika 3.4, specifikacija lokacije gde će Android Studio biti instaliran	49
Slika 3.5, odabir Start Menu direktorijuma	49
Slika 3.6, proces instalacije Android Studio okruženja.....	50
Slika 3.7, završetak instalacije Android Studio okruženja	50
Slika 3.8, import podešavanja od prethodne verzije (ukoliko postoji).....	51
Slika 3.9, Android Studio splash screen.....	51
Slika 3.10, preuzimanje potrebnih SDK komponenti prilikom prvog pokretanja okruženja	51
Slika 3.11, Android Studio čarobnjak za podešavanje okruženja.....	52
Slika 3.12, odabir tipa instalacije okruženja	52
Slika 3.13, odabir teme korisničkog interfejsa okruženja.....	53
Slika 3.14, verifikacija odabranih opcija	53
Slika 3.15, čarobnjak preuzima i instalira SDK komponente, sa i bez prikaza detalja.....	54
Slika 3.16, Welcome to Android Studio ekran.....	54
Slika 3.17, odabir platforme i šablona aktivnosti za novi projekat	55
Slika 3.18, osnovno podešavanje projekta	56
Slika 3.19, odabir druge vrednosti minimalnog podržanog API nivoa	57
Slika 3.20, procenat uređaja koji će moći da pokrenu aplikaciju na osnovu zadatog minimalnog API nivoa	57
Slika 3.21, kreirani HelloWorld projekat	57
Slika 3.22, struktura standardne Android aplikacije.....	58
Slika 3.23, projektni fajlovi u Android pogledu	58
Slika 3.24, struktura glavnog prozora Android Studio okruženja	59
Slika 3.25, tab sa XML rasporedom komponenti u editoru.....	61
Slika 3.26, lokacija XML fajlova sa rasporedom u strukturi projekta (res/layout)	62
Slika 3.27, grafički editor za XML fajlove sa rasporedom komponenti	62
Slika 3.28, pregled sadržaja XML koda rasporeda komponenti	63
Slika 3.29, lokacija AndroidManifest.xml fajla u strukturi projekta	64
Slika 3.30, build.gradle fajl za modul app i njegova lokacija u strukturi projekta	65
Slika 3.31, sadržaj build.gradle fajla za modul app.....	66

Slika 3.32, sekcija Tools, gde se nalaze AVD i SDK Manager opcije.....	66
Slika 3.33, SDK Tools tab u okviru SDK Manager sekcije	67
Slika 3.34, AVD Manager u okviru sekcije Tools.....	68
Slika 3.35, Raspoloživi virtuelni uređaji u okviru opcije AVD Manager	68
Slika 3.36, odabir hardverskog profila novog virtualnog uređaja	69
Slika 3.37, odabir sistemske slike virtualnog uređaja	69
Slika 3.38, verifikacija odabrane konfiguracije	70
Slika 3.39, napredna podešavanja virtualnog uređaja	70
Slika 3.40, novi uređaj prikazan u AVD Manager ekranu	71
Slika 3.41, pokretanje aplikacije na dodatom virtualnom uređaju	71
Slika 3.42, pokrenuta aplikacija na virtualnom uređaju.....	72
Slika 3.43, lista svih dostupnih SDK platformi	73
Slika 3.44, lista dostupnih SDK platformi sa detaljima za svaku pojedinačnu platformu.....	73
Slika 3.45, lista svih dostupnih SDK alata	74
Slika 3.46, proces ažuriranja komponenti	75
Slika 4.1, primer strukture korisničkog interfejsa sa ViewGroup komponentama i pripadajućim View objektima.....	76
Slika 4.2, definicija pozicije View komponente	78
Slika 4.3, padding i margin komponente	79
Slika 4.4, prikaz tipične upotrebe TextView komponente	83
Slika 4.5, prikaz tipične upotrebe EditText komponente	86
Slika 4.6, različiti tipovi virtualne tastature za različite vrednosti inputType atributa	87
Slika 4.7, tipičan primer upotrebe dugmeta sa jednostavnom obradom događaja	90
Slika 4.8, komponente CheckBox	91
Slika 4.9, tipičan primer upotrebe CheckBox komponenti.....	95
Slika 4.10, tipičan primer upotrebe RadioButton komponente	97
Slika 4.11, tipičan primer upotrebe Spinner komponente.....	100
Slika 4.12, prikaz kratke Toast poruke u mail aplikaciji	101
Slika 4.13, primer upotrebe Toast poruke.....	103
Slika 4.14, TimePicker i DatePicker komponente	104
Slika 4.15, Izgled DatePicker komponente u okviru aplikacije	105
Slika 4.16, tipični primjeri Dialog komponenti	107
Slika 4.17, prikaz jednostavnog Dialog-a	109
Slika 4.18, hijerarhija komponenti sa pridruženim layout parametrima	110

Slika 4.19, najčešće korišćeni rasporedi komponenti	111
Slika 4.20, najčešće korišćeni rasporedi komponenti koji koriste adapter	112
Slika 4.21, linearni raspored komponenti	113
Slika 4.22, primer vertikalnog linearног rasporeda sa težinama	115
Slika 4.23, relativni raspored komponenti	116
Slika 4.24, primer relativnog rasporeda	120
Slika 4.25, dodavanje ograničenja komponenti u ograničenom rasporedu ...	122
Slika 4.26, tipičan primer upotrebe ListView rasporeda komponenti.....	124
Slika 4.27, Layout Editor - grafički editor za dizajn rasporeda komponenti....	125
Slika 4.28, podešavanje atributa komponente u grafičkom editoru.....	126
Slika 4.29, R.java klasa, lokacija i njen sadržaj.....	127
Slika 5.1, hijerarhija procesa po važnosti	136
Slika 5.2, životni ciklus aktivnosti sa callback metodama.....	139
Slika 5.3, dijagram stanja u kojima se aktivnost može naći.....	142
Slika 5.4, primer aktivnosti koja gubi fokus usled dolaznog poziva	144
Slika 5.5, uticaj promene rotacije uređaja na stanje aktivnosti	145
Slika 5.6, dodavanje alternativnog prikaza za Landscape režim	149
Slika 5.7, dodatni XML fajl za raspored komponenti u Landscape režimu.....	150
Slika 5.8, korisnički interfejs aplikacije sa odvojenim rasporedima za Portrait i Landscape režim	152
Slika 5.9, implicitna namera i poziv aktivnosti kamere	158
Slika 5.10, prosleđivanje implicitne namere	158
Slika 5.11, dijalog za odabir aplikacije	159
Slika 5.12, princip rada back stack-a.....	162
Slika 5.13, primer dva taska, pri čemu je task B u fokusu i vrši interakciju sa korisnikom, dok je task A u pozadini sa kompletno očuvanim back stack-om	163
Slika 5.14, slučaj više instanci jedne aktivnosti u okviru back stack-a	163
Slika 5.15, Prikaz aktivnosti koja preuzima podatke sa korisničkog interfejsa i ispisuje ih u tekstualnom polju.....	168
Slika 5.16, dinamički kreiran korisnički interfejs	173
Slika 5.17, primer aplikacije koja koristi eksplicitnu nameru za pokretanje druge aktivnosti u okviru iste aplikacije	180
Slika 6.1, Application not responding dijalog	181
Slika 6.2, dijagram toka AsyncTask-a.....	184
Slika 6.3, Preuzimanje slike pomoću AsyncTask.....	190
Slika 6.4, Prikaz dohvaćene liste fakulteta preko AsyncTask-a	198
Slika 7.1, dijalog traženja dozvola za vreme izvršavanja	203

Slika 7.2, dijalog traženja dozvola za vreme instalacije	203
Slika 7.3, dijagram procesa zahtevanja dozvola	207
Slika 7.4, korisnički interfejs aplikacije koja demonstrira dinamičku dodelu dozvola	209
Slika 7.5, Rad sa dozvolama, dinamička provera dozvola	214
Slika 7.6, Rad sa dozvolama, dinamičko traženje dozvola	215
Slika 7.7, Rad sa dozvolama, dinamička dodela dozvola i provera statusa dozvola za aplikaciju u okviru podešavanja.....	215
Slika 7.8, Rad sa dozvolama, dijalog ukoliko korisnik odbije da dodeli dozvole	216
Slika 7.9, dinamička dozvola pristupa eksternom prostoru	221
Slika 8.1, primer modularnog dizajna upotrebom fragmenata.....	223
Slika 8.2, životni ciklus fragmenta dok se njegova aktivnost domaćin izvršava	225
Slika 8.3, ubacivanje fragmenta u aktivnost deklaracijom u XML rasporedu aktivnosti	228
Slika 8.4, ubacivanje fragmenta programskim putem.....	229
Slika 8.5, zamena jednog fragmenta drugim.....	231
Slika 8.6, korisnički interfejs aplikacije sa dva fragmenta	242
Slika 9.1, Room - osnovni dijagram arhitekture i odnosa između komponenti	257
Slika 9.2, Korisnički interfejs aplikacije za čuvanje podataka	265
Slika 9.3, Prikaz svih unetih redova iz Notes SQLite baze podataka	279
Slika 10.1, publisher-subscriber šablon	287
Slika 10.2, prikaz rada BroadcastReceiver aplikacije koja prati obaveštenja o promenama nivoa baterije	293
Slika 10.3, aplikacija sa prijemnikom obaveštenja o bateriji - ulazak u Low Battery	297
Slika 10.4, aplikacija sa prijemnikom obaveštenja o bateriji – izlazak iz Low Battery	297
Slika 10.5, simulacija slanja poruka i poziva ka emulatoru.	302
Slika 10.6, presretanje SMS poruke pomoću BroadcastReceiver komponente	302
Slika 10.7, presretanje poziva pomoću BroadcastReceiver komponente.....	303
Slika 10.8, Primer aplikacije koja šalje sopstvena obaveštenja.....	306
Slika 11.1, životni ciklus servisa	309
Slika 11.2, red za čekanje i Looper	315

Slika 11.3, životni ciklus servisa koji je startovan i vezan u isto vreme.....	320
Slika 11.4, prikaz rada servisa koji pokreće muziku u pozadini.....	331
Slika 11.5, prikaz rada aplikacije sa IntentService	335
Slika 11.6, primer izvršavanja vezanog servisa.....	340
Slika 11.7, prikaz rada servisa koji obavlja Cezarovu šifru.	349
Slika 12.1, upotreba postojeće aplikacije za kameru upotrebom implicitne namere	352
Slika 12.2, primer dohvatanja lokacije uređaja	356
Slika 12.3, Koordinatni sistem koji koriste senzori pokreta na Android uređaju	357
Slika 12.4, senzor rotacije uređaja	359
Slika 12.5, primer rada akcelerometra	360
Slika 12.6, kreiranje Google Maps aktivnosti	361
Slika 12.7, aplikacija koja postavlja marker trenutne lokacije na mapi.....	367
Slika 12.8, prikaz markera na mapi sa dodatnim informacijama o lokaciji	367

Predgovor

Programski jezik Java, kao jedan od najpopularnijih programskih jezika u svetu, svoju popularnost duguje pre svega eleganciji, savremenim konceptima i jednostavnom objektno orijentisanom modelu. Programi napisani u Javi se mogu pronaći svuda, od Android mobilnih uređaja, preko klasičnih desktop aplikacija, pa sve do Internet strana i veb servisa. Zbog sve veće popularnosti mobilnih aplikacija i dostupnosti mobilnih uređaja, jedno od najtraženijih zanimanja danas je upravo razvoj mobilnih aplikacija za Android platformu, što predstavlja fokus ovog udžbenika.

Mobilni operativni sistemi poput Androida su danas dostupni ne samo za mobilne telefone, već i za tablete, kao i čitav niz drugih uređaja, poput pametnih satova i televizora. Samim tim je i tržiste na koje se mogu plasirati Android aplikacije značajno povećano. Sa druge strane, mobilne aplikacije imaju veći broj prednosti u odnosu na tradicionalne desktop aplikacije. Ovakav tip aplikacija je uvek uz korisnika, pošto mobilni telefon nosimo uvek sa sobom. Mobilni telefon nudi još dodatne mogućnosti u obliku pristupa Internetu, velikog broja senzora (poput akcelerometra ili kompasa), pristupa mapama, mogućnosti za pozicioniranje preko GPS i kamere, a veliki broj dostupnih aplikacija je besplatan.

Koncept razvoja mobilnih aplikacija za Android u programskom jeziku Java zahteva razumevanje paradigmi na kojima je zasnovan, uključujući dobro poznavanje objektno orijentisanog programiranja (OOP). OOP se bazira na premisi da se problem koji se rešava može modelovati objektima u domenu problema. OOP ima svoje specifične karakteristike, poput nasleđivanja, enkapsulacije i polimorfizma. Programiranje za Android naročito zahteva dobro poznavanje koncepta nasleđivanja, apstraktnih klasa i interfejsa, jer je većina standardnih Android komponenti realizovana upravo na ovaj način, odnosno od programera se zahteva da proširi odgovarajuće standardne Android klase i da implementira već postojeće interfejse. Dodatno, potrebno je imati u vidu i ograničenja na koja ranije nije obraćano previše pažnje. Na primer, veličina ekrana za prikaz je veoma ograničena i drastično je manja od veličine standardnih desktop monitora. Pošto je telefon mobilni uređaj, mora se обратити pažnja i na ograničen kapacitet baterije i na ograničenu procesorsku moć, koja je višestruko manja od modernih računara.

Programersko okruženje koje se koristi za pisanje Android aplikacija je isključivo Android Studio. U pitanju je okruženje koje je zasnovano na IntelliJ okruženju, i u potpunosti prilagođeno lakšem razvoju mobilnih aplikacija za Android, uz podršku i za Java i za Kotlin. Android Studio, dodatno, značajno olakšava razvoj korisničkog interfejsa, koji se može izvesti prostim prevlačenjem ponuđenih komponenti na virtuelni ekran. U ovom udžbeniku će čitalac detaljno biti upućen

u korišćenje ovog okruženja za razvoj aplikacija, uz veliki broj konkretnih primera.

Obim i sadržaj udžbenika prilagođeni su nastavnom programu istoimenog predmeta na Fakultetu za informatiku i računarstvo Univerziteta Singidunum u Beogradu. Podrazumeva se da čitaocima knjige ovo nije prvi susret sa programiranjem u Javi, tj. očekuje se poznavanje programskog jezika Java i koncepcata OOP.

Tekst je propraćen velikim brojem slika i detaljno objašnjenih primera pomoću kojih su ilustrovani novouvedeni pojmovi. Primeri su odabrani na takav način da budu što jednostavniji za razumevanje, ali sa druge strane i što realističniji i interesantniji, sa krajnjim ciljem da podstaknu čitaoce na dublje samostalno učenje i budu početna tačka za dalje eksperimentisanje. Iako je knjiga pisana kao udžbenik, ambicija autora jeste da ona posluži svima koji se prvi put susreću sa programiranjem mobilnih aplikacija za Android u programskom jeziku Java. Na kraju većine poglavlja data su pitanja koja mogu pomoći u utvrđivanju gradiva. Ova pitanja su i tipična pitanja koja se mogu očekivati na intervjuu za posao Android programera.

Autor se ovom prilikom zahvaljuje asistentu Milanu Tairu, koji je omogućio studentima veći broj primera za vežbu kroz video materijale dostupne na adresi: <http://zadatak.singidunum.ac.rs/videos/android/index.php>. Iako su video materijali stari nekoliko godina, veliki broj njih je i dalje aktuelan i direktno primenjiv u praksi (uz eventualne manje modifikacije). Neki od primera su poslužili autoru kao inspiracija za pojedine primere koda prikazane u ovom udžbeniku.

Sve primedbe, komentari, preporuke, pohvale i eventualno uočene greške se mogu poslati na adresu mzivkovic@singidunum.ac.rs.

Autor

Beograd, Srbija, 2020.

1. Uvod

Android je operativni sistem za mobilne uređaje koji se zasniva na Linux jezgru, razvijen od strane kompanije Google primarno za uređaje sa ekranom osetljivim na dodir, poput mobilnih telefona i tableta. Korisnički interfejs Androida se uglavnom zasniva na direktnoj manipulaciji putem dodira i različitih pokreta (prevlačenje – engl. *swipe*, zumiranje i odzumiranje – engl. *pinch* itd.) nad objektima koji se nalaze prikazani na ekranu uređaja, uz virtualnu tastaturu koja se koristi za unos teksta.

Nakon velikog uspeha Android platforme na mobilnim telefonima i tabletima, Google je proširio podršku i na druge uređaje, tako da su danas podržani Android TV za pametne televizore, Android Wear za pametne satove, kao i Android Auto kao podrška za automobile, pri čemu svaki od njih ima specifičan korisnički interfejs. Dodatno, Android je u različitim varijantama danas dostupan i na digitalnim kamerama, konzolama za igre i velikom broju drugih elektronskih uređaja.

Danas se Android nalazi u vlasništvu konzorcijuma Open Handset Alliance, u kome se uz Google nalaze još 83 kompanije. Google je bio predvodnik osnivanja ovog konzorcijuma 2007. godine, zajedno sa kompanijama koje proizvode mobilne uređaje, nekim mobilnim operatorima, proizvođačima čipova, kao i sa kompanijama koje se bave razvojem aplikacija. Istaknuti članovi ovog konzorcijuma, uz naravno Google, su Samsung, LG, HTC, Sony, Motorola, Qualcomm, Texas Instruments, ARM, T-mobile, Vodafone i mnogi drugi. Open Handset Alliance je zadužen za razvoj otvorenih standarda za mobilne uređaje, a vodeći (engl. *flagship*) softver ovog konzorcijuma je upravo Android, koji se zasniva na licenci otvorenog koda. U cilju promovisanja jedinstvene platforme, svim članicama Open Handset Alliance je zabranjeno da proizvode mobilne uređaje na kojima se nalaze nekompatibilne verzije Android sistema. Kompatibilnost se postiže time što se softver razdvaja od hardvera na kome se izvršava, čime se omogućava velikom broju heterogenih uređaja da izvršavaju istu aplikaciju. Prvi komercijalno dostupni model telefona sa Android operativnim sistemom bio je HTC Dream, koji je bio poznat i pod nazivom T-Mobile G1, koji se na tržištu pojavio krajem 2008. godine.

Svaka od verzija Androida ima dodeljen broj, koji se naziva API nivo (engl. *Application Programming Interface - API level*). Ova oznaka služi da označi koja verzija Android API (uključuje veliki broj klasa i biblioteka) je dostupna programerima pri razvoju aplikacije. Osim ove oznake, glavne (engl. *major*) verzije imaju i kodno ime, poput Lollipop, Marshmallow, Nougat, Oreo itd. Specifično je da su sva kodna imena zasnovana na engleskim nazivima za različite slatkiše i idu abecednim redosledom. Ova praksa zavšena je 2019.

godine, objavljinjem Android 10 platforme, uz najavu da će se i sve buduća imena zasnivati na brojevima. Moguće je da više verzija androida ima isti API nivo, poput verzija Gingerbread 2.3.3, 2.3.4, 2.3.5, 2.3.6 i 2.3.7, koje sve imaju isti API nivo 10. Praktično gledano, time se označava da sve pomenute Gingerbread verzije nisu imale promene samog API-ja koji je dostupan i vidljiv programeru, već da su u pitanju samo ispravke uočenih bagova i određena unapređenja same platforme.

1.1. Istorijat

Razvoj Androida započeo je 2003. godine, pod okriljem kompanije Android Inc. Ovu kompaniju je 2005. godine preuzeo Google, a od 2007. godine Android se nalazi u okviru konzorcijuma Open Handset Alliance. Prva beta verzija objavljena je 5. novembra 2007. godine, dok je prvi SDK (engl. *software development kit*) objavljen nedelju dana kasnije, 12. novembra 2007. godine. Usledilo je još nekoliko beta verzija samog SDK, koja su se zasnivala isključivo na softverskoj emulaciji, pošto još uvek nije postojao nijedan fizički uređaj kako bi se testirao sam operativni sistem. Prva zvanična i javno dostupna verzija Androida 1.0 usledila je u oktobru 2008. godine sa objavom prvog komercijalno dostupnog Android telefona – HTC Dream (slika 1.1). Android 1.0 i 1.1 nisu imali dodeljena specifična kodna imena. Interno su bili poznati pod nazivima Astro Boy i Bender, ali ta imena nisu izašla u javnost, tako da su zvanično ostali priznati pod jednostavnim imenima 1.0 i 1.1.



Slika 1.1, prvi komercijalno dostupni Android telefon HTC Dream (slika preuzeta sa <https://android.appsapk.com/product/htc-dream/>)

1.1.1. Prve verzije

Android verzija 1.0 (API nivo 1) je sadržavala veći broj tada veoma naprednih funkcionalnosti:

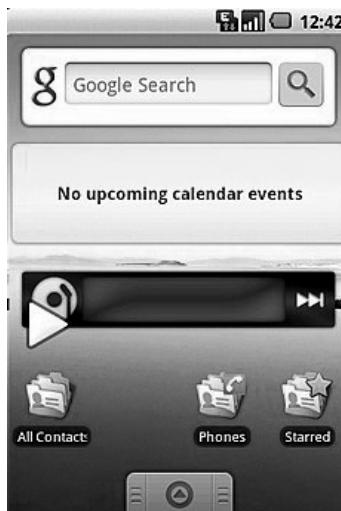
- Android Market aplikacija koja se koristila za download i update drugih aplikacija.
- Veb čitač za prikaz veb stranica, sa podržanom opcijom za zumiranje i prikazom više otvorenih stranica istovremeno kroz kartice.
- Kamera – uz ograničenja da nije bilo moguće promeniti rezoluciju i kvalitet slika.
- Folderi koji su omogućili grupisanje ikonica aplikacija u jedinstveni folder na glavnom ekranu.
- Pristup mejl serverima putem protokola POP3, IMAP4 i SMTP.
- Sinhronizacija Gmail naloga sa Gmail aplikacijom na telefonu.
- Google kontakti sinhronizovani sa People aplikacijom.
- Google kalendar sinhronizovan sa Calendar aplikacijom.
- Google Maps za prikaz mapa i satelitskih snimaka, uz podršku za navigaciju preko GPS (Global Positioning System).
- Google Sync za upravljanje sinhronizacijom aplikacija Gmail, People i Calendar.
- Google Search za pretragu Interneta i aplikacija na telefonu, kontakata, kalendara i slično.
- Google Talk za komunikaciju preko poruka.
- Instant poruke, tekstualne poruke (SMS) i multimedijalne poruke (MMS).
- Media Player za upravljanje i prikaz multimedijalnim fajlovima (uz nedostatak video i stereo Bluetooth podrške).
- Notifikacije koje se prikazuju u statusnoj liniji, sa različitim opcijama za podešavanje zvučnih signala telefona, LED svetala i vibracije.
- Voice Dialer za usmeno iniciranje poziva, bez ukucavanja imena ili broja.
- Wallpaper za postavljanje različitih pozadinskih slika ili fotografija iza ikonica i vidžeta na glavnom ekranu (Home screen).
- Inkorporirani YouTube plejer.
- Podrška za Wi-Fi i Bluetooth.
- Dodatne aplikacije, poput Alarm Clock, Calculator, Dialer, Gallery i Settings.

Sledeća verzija, Android 1.1 (API nivo 2), objavljena je u februaru 2009 za telefon HTC Dream. Ova verzija je ispravila neke uočene bagove i donela nekoliko novih funkcionalnosti, od kojih su najbitnije:

- Prilikom pretrage za lokalnim poslovnicama (poput restorana) na mapama, moguće je videti i detalje i komentare korisnika.
- Mogućnost čuvanja priloga (engl. *attachments*) u okviru poruka.
- Mogućnost sakrivanja/prikazivanja tastature za vreme poziva preko spikerfona.
- Podrška za *marquee* (nešto slično kajronu na televiziji, tekst koji skroluje preko ekrana) u rasporedu komponenti (engl. *layout*).

1.1.2. Android 1.5 Cupcake

U toku 2009. godine pojavili su se novi Android mobilni telefoni, kao i nekoliko verzija samog Androida. Prva od ovih verzija je Android 1.5 (API nivo 3), koji je objavljen u aprilu 2009, koja je prva imala zvanično kodno ime bazirano na slatkišu – Cupcake. Verzija je donela nekoliko novih funkcionalnosti, kao i izmena korisničkog interfejsa, čiji je izgled prikazan na slici 1.2.



Slika 1.2, Android 1.5 Cupcake glavni ekran
(slika preuzeta sa: https://en.wikipedia.org/wiki/Android_version_history)

Najbitnije nove funkcionalnosti su:

- Podrška za virtuelne tastature razvijane od treće strane (engl. *third party*), sa predikcijom teksta i rečnikom za korisnički definisane pojmove
- Podrška za vidžete (engl. *widgets* – male aplikacije sa minijaturnim prikazom, koje se mogu ugnjezditi u druge aplikacije, poput glavnog ekrana, i koje se mogu periodično ažurirati).
- Snimanje i plejbek video materijala u MPEG-4 i 3GP formatima.
- Bluetooth - automatsko uparivanje i podrška za stereo.

- Podrška za copy/paste u veb čitaču.
- Slike za omiljene kontakte.
- Animirane tranzicije između ekrana.
- Auto-rotacija.
- Upload video materijala na YouTube i slika na Picasa.

1.1.3. Android 1.6 Donut

Sledeća verzija Androida, koja je takođe izашla 2009. godine, u septembru, je bila verzija 1.6, poznata pod kodnim imenom Donut. Izgled korisničkog interfejsa prikazan je na slici 1.3.



Slika 1.3, Android 1.6 Donut glavni ekran
(slika preuzeta sa: https://en.wikipedia.org/wiki/Android_version_history)

Uvedene su brojne nove funkcionalnosti, od kojih su najbitnije:

- Tekstualna i glasovna pretraga je poboljšana tako da uključuje istoriju pretraživanja, obeležene stranice (engl. *bookmarks*) i veb.
- Mogućnost da programeri uključe svoj sadržaj u rezultate pretrage.
- Lakša pretraga aplikacija na Android Market-u, sa dodatim skrinšotovima aplikacija.
- Galerija i kamera su potpuno integrисани, sa bržim pristupom kamери.
- Mogućnost obeležavanja i brisanja više slika odjednom.
- Podrška za text-to-speech.

- Podrška za telekomunikacione tehnologije CDMA (engl. *Code-division multiple access*), EVDO (engl. *Evolution-Data Optimized*), VPN (engl. *Virtual Private Network*). Jedna od mreža koja koristi CDMA je Verizon u Sjedinjenim Američkim Državama, čime je drastično uvećano tržište.
- Proširena podrška za pokrete i gestove prstima.

1.1.4. Android 2.0 Eclair

Još jedna verzija Androida je puštena u javnost 2009. godine, krajem oktobra objavljen je Android 2.0 Eclair (API nivo 5). Izgled korisničkog interfejsa ove verzije prikazan je na slici 1.4. Već u decembru iste godine izašla je verzija Android 2.0.1 Eclair (API nivo 6), a u januaru 2010 i verzija Android 2.1 Eclair (API nivo 7). Ove dve verzije su sadržavale samo manje ispravke bagova i minorne promene API-ja. Eclair je uveo sledeće nove funkcionalnosti:

- Proširenje sinhronizacije naloga, tako da se korisnicima dozvoli da dodaju više naloga na uređaj koji se mogu sinhronizovati sa mejlom i kontaktima.
- Podrška za Microsoft Exchange, sa kombinovanim sandučetom gde se mogu videti mejlovi sa više naloga u okviru jedne strane.
- Podrška za Bluetooth 2.1.
- Dodir na fotografiju kontakta otvara opcije za iniciranje poziva, SMS ili mejl poruke.
- Brojna poboljšanja kamere, uključujući blic, digitalno zumiranje, makro fokus, različite kolor efekte i slično.
- Ubrzan rad virtuelne tastature, sa pametnijim imenikom koji uči reči koje korisnik upotrebljava, i uključuje imena kontakata u predloge.
- Osvežen korisnički interfejs veb čitača, uz dodatu podršku za HTML5 standard i za zumiranje dvostrukim dodirom.
- Poboljšane funkcionalnosti kalendara i Google Maps (glasovna navigacija).
- Optimizovana brzina sistema i podrška za veći broj veličina ekrana i različitih rezolucija, sa boljim odnosom kontrasta.
- Podrška za rukovanje multi-touch događajima.
- Podrška za žive pozadine kroz animaciju pozadine glavnog ekrana.
- Podrška za speech-to-text.



Slika 1.4, Android 2.0 Eclair poboljšana navigacija i funkcija speech-to-text (slika preuzeta sa: <https://www.computerworld.com/article/3235946/android-versions-a-living-history-from-1-0-to-today.html>)

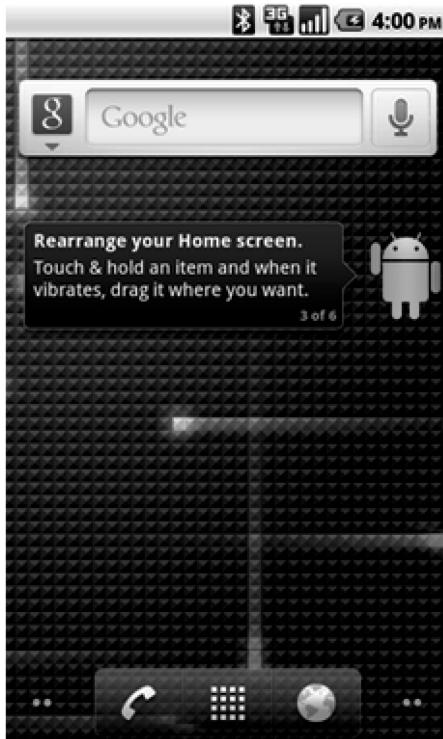
1.1.5. Android 2.2 Froyo

U maju 2010. godine usledila je verzija Android 2.2 Froyo (API nivo 8). Froyo je inače skraćeno od zamrznutog jogurta (engl. *Frozen Yogurt*). Izgled korisničkog interfejsa ove verzije prikazan je na slici 1.5. Ova verzija je unela brojna poboljšanja i optimizacije, među kojima su najbitnije:

- Brojne optimizacije brzine, korišćenja memorije i performansi.
- Dodatna ubrzavanja aplikacija kroz JIT kompilaciju (engl. *Just in Time*).
- Podrška za Android Cloud to Device Messaging servis, čime su omogućene push notifikacije.
- Deljenje mreže - USB *tethering* i Wi-Fi hotspot funkcionalnosti.
- Opcija za isključivanje prenosa podataka preko mobilne mreže.
- Ažurirana Android Market aplikacija, sa opcijom za automatsko ažuriranje aplikacija.
- Brz prelaz između različitih jezika na virtuelnoj tastaturi.

- Podrška za numeričke i alfanumeričke šifre.
- Podrška za Adobe Flash
- Veb čitač prikazuje sve frejmove animiranog GIF-a, umesto samo prvog

Froyo je imao još tri verzije u toku 2011. godine, 2.2.1, 2.2.2 i 2.2.3, koje su sve bile samo ispravke bagova, dodatna poboljšanja performansi i sigurnosne zakrpe.



Slika 1.5, Android 2.2 Froyo glavni ekran
(slika preuzeta sa: https://en.wikipedia.org/wiki/Android_version_history)

1.1.6. Android 2.3 Gingerbread

U decembru 2010. godine, usledila je i nova glavna verzija Androida, Android 2.3 Gingerbread (API nivo 9). Izgled korisničkog interfejsa ove verzije prikazan je na slici 1.6. Ova verzija počela je da uvodi i prvi pravi vizuelni identitet Androida. Svetlo zelena boja je oduvek bila boja Androidove maskote – robota, a sa Gingerbread verzijom, postala je integralni deo samog izgleda operativnog sistema. Crna i zelena boja su bile prisutne i vidljive u celom korisničkom interfejsu, čime je počelo utabavanje prepoznatljivog dizajna.



Slika 1.6, Android 2.3 Gingerbread glavni ekran
(slika preuzeta sa: <https://developer.android.com/about/versions/android-2.3-highlights>)

Osim prepoznatljivog interfejsa, ova verzija je takođe uvela veći broj novina i poboljšanja, među kojima su najbitniji:

- Podrška za veoma velike ekrane i rezolucije (WXGA i veće).
- Nativna podrška za SIP VoIP (engl. *voice over IP*)
- Brži i intuitivniji unos teksta preko virtualne tastature, sa poboljšanom preciznošću, boljim sugerisanjem teksta i boljim glasovnim unosom.
- Poboljšana copy/paste opcija, koja dozvoljava korisnicima da odaberu i kopiraju reč dodirom i držanjem.
- Podrška za NFC (engl. *Near Field Communication*), čime korisnici mogu da očitaju NFC tagove koji su embedovani u okviru različitih postera ili reklama.
- Bolji audio efekti, naročito pri upotrebi slušalica.
- Novi Download Manager, koji omogućava korisnicima lak pristup bilo kom fajlu koji je preuzet putem veb čitača, mejla ili neke druge aplikacije.

- Podrška za više kamere na uređaju, poput prednje kamere ukoliko je dostupna na samom uređaju.
- Podrška za nove video/audio formate.
- Poboljšani menadžment energije, koji aktivnije prati aplikacije koje predugo drže uređaj budnim i time troše mnogo energije.
- Konkurentno sakupljanje đubreta (engl. *Garbage Collection*) za poboljšanje performansi.
- Veliki broj audio i grafičkih poboljšanja za bolju podršku mobilnim video igrama.

Nakon ove verzije, usledilo je još nekoliko Gingerbread verzija. Verzije 2.3.1 i 2.3.2 (API nivo 9) su se uglavnom odnosile na poboljšanja i ispravke bagova za model Nexus S. Nexus S je bio telefon koji su zajedno razvili Google i Samsung, i bio je prvi uređaj koji je koristio Gingerbread 2.3, kao i prvi Android uređaj sa softverskom i hardverskom podrškom za NFC. U februaru 2011. godine izašla je i verzija 2.3.3, koja je takođe nosila oznaku Gingerbread, ali sa novim API nivoom 10. Od februara do septembra 2011. godine objavljeno je 5 verzija, 2.3.3 – 2.3.7, koje su uglavnom sadržavale ispravke uočenih grešaka (između ostalog spontano resetovanje nekih modela, bagovi sa Bluetooth i Wi-Fi, kao i bag u glasovnoj pretrazi), zatim određenja poboljšanja (mrežne performanse, bolja Gmail aplikacija, dodate animacije pri skrolovavanju lista, poboljšanja kamere), kao i dodavanje podrške za Google Walet sa verzijom 2.3.7.

1.1.7. Android 3.0 Honeycomb

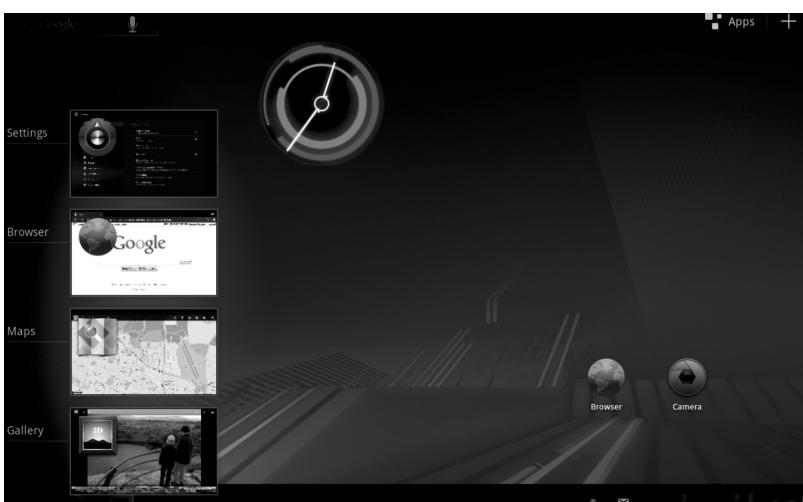
U februaru 2011 objavljena je verzija Android 3.0 Honeycomb (API nivo 11), koja je bila namenjena i optimizovana isključivo za uređaje sa većim ekranima - tablete. Prvi tablet koji je podržavao ovu verziju Androida bio je Motorola Xoom. Nakon verzije 3.0, iste godine usledile su verzije 3.1 (API nivo 12) i 3.2 (API nivo 13). Uveden je novi, tzv. hologramski dizajn, kao što je prikazano na slici 1.7. Ovakav dizajn je omogućio maksimalno iskorišćenje velikog ekrana tableta. Iako sam koncept interfejsa specifičnog isključivo za tablete nije potrajaо dugo, veliki broj ideja koje je uveo Honeycomb su postavile temelje za Android kakav danas poznajemo, poput softverskih dugmića za navigaciju na samom ekranu i koncepta prikaza nedavno korišćenih aplikacija u obliku kartica. Među ostalim novinama koje su unete sa ovom verzijom, izdvajaju se:

- Dodata sistemska linija (engl. *System bar*) sa brzim pristupom notifikacijama i statusu, kao i softverski navigacioni dugmići na dnu ekrana.
- Dodata akcionala linija (engl. *Action bar*) sa pristupom kontekstualnim opcijama, navigaciji i vidžetima.

- Pojednostavljen multitasking – dodir na nedavno korišćene aplikacije (engl. *Recent Apps*) omogućava korisniku da vidi slike (kartice) taskova koji se trenutno izvršavaju, i da brzo skače sa jedne aplikacije na drugu, kao što je prikazano na slici 1.8.
- Podrška za procesore sa više jezgara.
- Dvopanelni korisnički interfejsi za kontakte i mejlove, za lakšu organizaciju i prikaz kontakata i organizaciju mejlova.



Slika 1.7, Android 3.0 Honeycomb glavni ekran
(izvor: <https://developer.android.com/about/versions/android-3.0-highlights>)



Slika 1.8, Android 3.0 Honeycomb Recent apps
(izvor: <https://developer.android.com/about/versions/android-3.0-highlights>)

1.1.8. Android 4.0 Ice Cream Sandwich

U oktobru 2011. godine na tržište je izbačena još jedna glavna verzija - Android 4.0 Ice Cream Sandwich (API nivo 14). Smatra se da je prethodna verzija, Honeycomb, služila kao most za prelazak između starog i novog, a da je Ice Cream Sandwich zvanično uveo platformu u eru modernog dizajna. Ice Cream Sandwich je iskoristio i poboljšao sve vizuelne koncepte uvedene sa verzijom Honeycomb (softverski dugmići za navigaciju i pregled nedavnih aplikacija), i ujedinio tablete i telefone pod jedinstven korisnički interfejs, koji je prikazan na slici 1.9.



Slika 1.9, Android 4.0 Ice Cream Sandwich glavni ekran i Recent Apps
(slika preuzeta sa: <https://www.computerworld.com/article/3235946/android-versions-a-living-history-from-1-0-to-today.html>)

Tokom 2011. i 2012. godine objavljeno je još nekoliko verzija Ice Cream Sandwich. Verzije 4.0.1 i 4.0.2 su delile isti API nivo 14 kao i glavna verzija, dok su verzije 4.0.3 i 4.0.4 objavljene pod API nivoom 15. Naknadne verzije su uglavnom sadržavale ispravke bagova i manja poboljšanja. Osim izgleda korisničkog interfejsa, nove funkcionalnosti koje su uvedene sa Ice Cream Sandwich verzijom su brojne, a među najznačajnije spadaju:

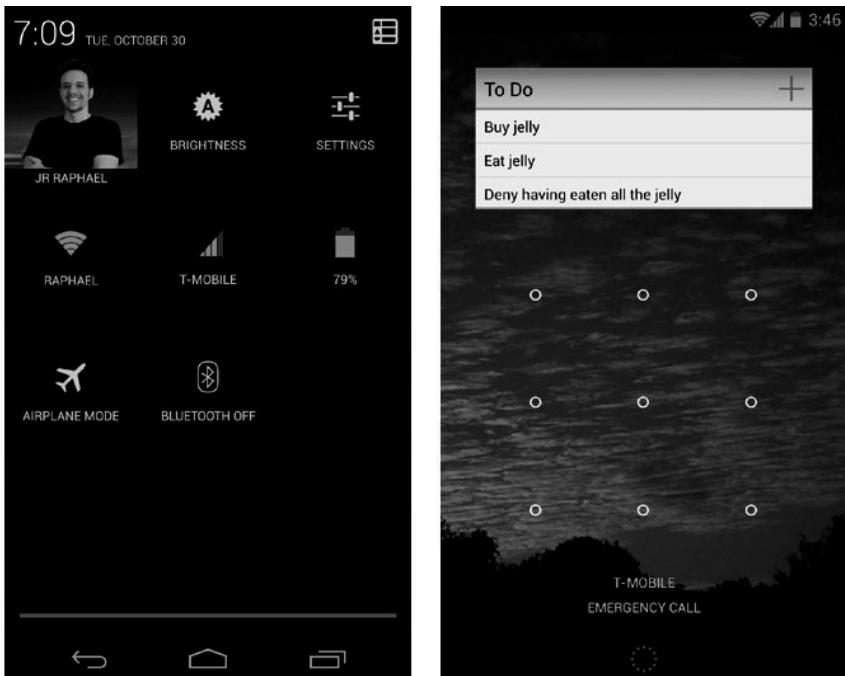
- Integrisana podrška za skrinšot (istovremeno držanje hardverskih dugmića *Power* i *Volume-down*).
- Poboljšano zumiranje pokretima prstiju u nekim aplikacijama poput kalendara.
- Poboljšan copy/paste.
- Poboljšano prepoznavanje lica, uvedena opcija otključavanja telefona sa Face Unlock.
- Dodata sekcija Data Usage u okviru podešavanja, koja upozorava korisnika kada se približi određenom limitu transfera podataka preko mobilne mreže, i onemogućava prenos podataka preko mobilne mreže kada se ovaj limit premaši.
- Pokretom prevlačenja (engl. *swipe*) se može ukloniti aplikacija iz liste nedavnih aplikacija ili odbaciti notifikacija.
- People aplikacija je poboljšana integracijom socijalnih mreža, statusnim porukama i slikama visoke rezolucije.
- Poboljšana kamera, dodat panorama režim rada, kao i mogućnost zumiranja za vreme snimanja video snimka.
- Glatke rotacije ekrana.

1.1.9. Android 4.1 Jelly Bean

U julu 2012. godine, na tržište je stigla verzija Android 4.1 Jelly Bean (API nivo 16). Tokom 2012. i 2013. godine izašle su još dve verzije Jelly Bean: 4.2 (API nivo 17) i 4.3 (API nivo 18). Ova verzija je dodatno nadgradila i optimizovala prethodnu verziju, čime je Android postao mnogo privlačniji prosečnim korisnicima. Izgled korisničkog interfejsa ove verzije prikazan je na slici 1.10. Velika poboljšanja u performansama i brzini rada Androida obezbeđena su, između ostalog, smanjenim kašnjenjem detekcije dodira, ubrzavanjem procesora i hardverski ubrzanim 2D renderovanjem. Notifikacije su sada postale proširive i interaktivne, poboljšana je i podrška za glasovnu pretragu i uveden je napredniji sistem za prikaz rezultata pretrage. Osim navedenih, neke od najznačajnijih novih funkcionalnosti koje je uveo Jelly Bean su:

- Mogućnost za ukidanje notifikacija na nivou pojedinačnih aplikacija.
- Prečice i vidžeti se mogu aranžirati i promeniti veličinu kako bi se dodali novi na glavnom ekranu.
- Dodatno poboljšan softver za kameru.
- Višekanalni i USB audio
- Brza podešavanja (engl. *Quick Settings*).
- Nova Clock aplikacija, sa ubaćenim svetskim satom, štopericom i tajmerom.

- Podrška za Bluetooth LE (engl. *Low energy*) i opšte gledano poboljšan Bluetooth stek.
- Poboljšana grafika za igre uvođenjem podrške za OpenGL ES 3.0.
- Omogućeno više korisničkih naloga.
- Podrška za jezike koji se pišu sa desna na levo.
- Nativna podrška za emoji.
- Veći broj sigurnosnih ispravki.
- Poboljšanje zaključanog ekrana, uključujući mogućnost da se putem poteza prevlačenja prstom direktno ode u aplikaciju kamere.



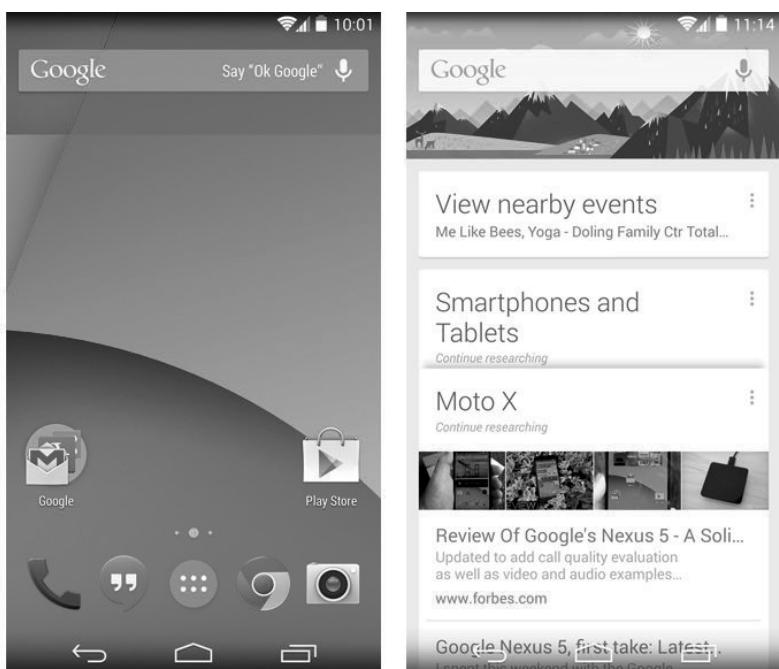
Slika 1.10, Android 4.1 Jelly Bean korisnički interfejs
 (slika preuzeta sa: <https://www.computerworld.com/article/3235946/android-versions-a-living-history-from-1-0-to-today.html>)

1.1.10. Android 4.4 KitKat

Sledeća verzija Androida, 4.4 KitKat (API nivo 19), zvanično je izašla na tržište u septembru 2013. Ova verzija je označila kraj "mračne" ere, pošto je pretežno crna boja prisutna u verziji Gingerbread, odnosno pretežno tamno plava koju je uveo Honeycomb zamjenjena svetlijim i neutralnijim pozadinama, sa transparentnom statusnom linijom i pretežno svetlim i belim ikonicama. Korisnički interfejs ove verzije Androida prikazan je na slici 1.11. KitKat je

dizajniran da se izvršava brzo i glatko na većem broju uređaja nego ikada pre, računajući veliki broj niskobudžetnih modela telefona sa svega 512MB RAM. Veliki broj optimizacija pomogao je u štednji raspoložive memorije, poput optimizacije Dalvik JIT keša, KSM (engl. *kernel samepage merging*) i prelaska na zRAM. Osim glavne verzije 4.4, do sredine 2014. godine na tržište je izašlo još četiri verzije KitKat-a, označenih sa 4.4.1 – 4.4.4, koje su uglavnom bile ispravke bagova, dodatne optimizacije, kao i poboljšanje sigurnosti. KitKat je uneo brojne novine, a najbitnije nove funkcionalnosti su, između ostalog:

- Aplikacije u imerzivnom modu (sakrivanje navigacije i statusne linije, aplikacija preuzima ceo ekran).
- Restrikcije aplikacija prilikom pristupa eksternoj memoriji (osim sopstvenih direktorijuma).
- Opcija za bežično štampanje.
- Dodatne opcije sa senzorima, poput detekcije i brojanja koraka korisnika.
- Umesto Dalvik virtuelne mašine započeto uvođenje Android Runtime (ART).
- Aplikacijama razvijenim od trećih strana onemogućen pristup statistici korišćenja baterije.

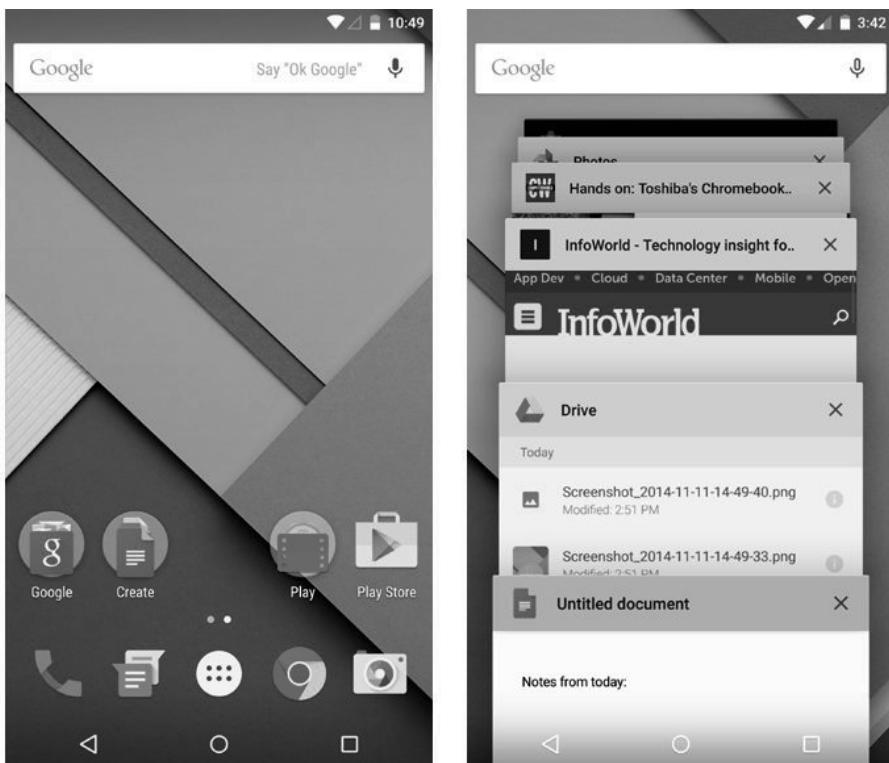


Slika 1.11, Android 4.4 KitKat korisnički interfejs
(slika preuzeta sa: <https://www.computerworld.com/article/3235946/android-versions-a-living-history-from-1-0-to-today.html>)

Osim glavne Android 4.4 verzije, 2014. godine na tržište je puštena i verzija Android 4.4W KitKat, koja je bila isključivo namenjena Android Wear uređajima, pretežno pametnim satovima.

1.1.11. Android 5.0 Lollipop

U junu 2014. godine, na tržište je izашla sledeća glavna verzija, Android 5.0 Lollipop (API nivo 21). Lollipop je uveo i danas prisutni standard materijalnog dizajna, koji je doneo novi izgled u sve delove Androida i aplikacije. Korisnički interfejs ove verzije Androida prikazan je na slici 1.12.



Slika 1.12, Android 5.0 Lollipop korisnički interfejs i materijalni dizajn (slika preuzeta sa: <https://www.computerworld.com/article/3235946/android-versions-a-living-history-from-1-0-to-today.html>)

Osim redizajniranog korisničkog interfejsa, poboljšane su i notifikacije, kojima se može pristupiti direktno sa zaključanog ekrana, a mogu biti prikazane i u gornjem baneru. Google je u ovoj verziji zvanično zamenio Dalvik virtuelnu mašinu sa Android Runtime, kako bi se doobile još bolje performanse i dodatno

optimizovala potrošnju baterije. Osim navedenih, najbitnije nove funkcionalnosti, među ostalog, uključuju:

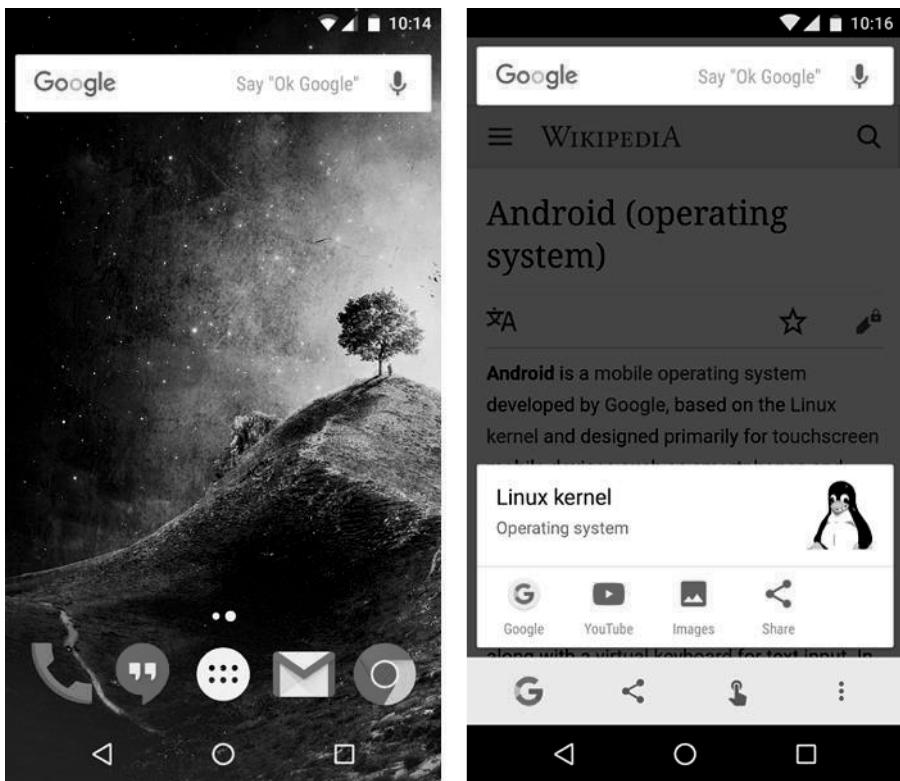
- Podrška za 64-bitne procesore.
- Vektorski grafički elementi (engl. *drawables*), koji se skaliraju bez gubitka kvaliteta.
- Projekat Volta, sa ciljem poboljšanja korišćenja baterije.
- Audio ulaz i izlaz kroz USB uređaje.
- Nedavno korišćene aplikacije su upamćene i posle restarta uređaja.
- Aplikacije razvijene od trećih strana ponovo dobijaju pristup podacima na SD kartici.
- Ubačena aplikacija Flashlight, koja koristi fleš kamere za osvetljavanje.

Osim glavne verzije 5.0, u toku decembra 2014. godine objavljene su još dve verzije Lollipop sistema, 5.0.1 i 5.0.2, sa istim API nivoom 21, uglavnom sa ispravkama uočenih bagova. U martu 2015. godine, izašao je Android 5.1 Lollipop sa API nivoom 22, koji je doneo nekoliko novina, poput zvanične podrške za više SIM kartica u jednom telefonu, dodatne zaštite u slučaju gubitka uređaja i prenosa glasa u visokoj definiciji između uređaja koji su kompatibilni sa 4G LTE. Nešto kasnije, u aprilu 2015. godine, verzija 5.1.1 je uglavnom ispravila uočene bagove.

1.1.12. Android 6.0 Marshmallow

U oktobru 2015 objavljena je nova glavna verzija Android sistema, 6.0 Marshmallow (API nivo 23). Ova verzija je započela trend da postoji jedna glavna verzija Androida godišnje, koja dobija svoj novi ceo broj (u ovom slučaju 6.0). Korisnički interfejs ove verzije Androida prikazan je na slici 1.13.

Jedna od glavnih novina koju je uveo Marshmallow je novi model dozvola (privilegija) sa mnogo većom granulacijom, gde korisnici dodeljuju dozvole za vreme izvršavanja aplikacije. Na ovaj način se korisnicima daje veća kontrola nad dozvolama, a olakšava proces instalacije aplikacija i njihovog ažuriranja. Korisnici u bilo kom trenutku mogu da daju ili povuku dozvole za instalirane aplikacije. Uvedene su i nove optimizacije za štednju energije kroz dva nova režima rada uređaja, Doze i App Standby. Još jedna od bitnih izmena je podrška za čitanje otiska prsta. U decembru 2015, objavljena je i verzija Marshmallow 6.0.1 sa sitnjim unapređenjima.



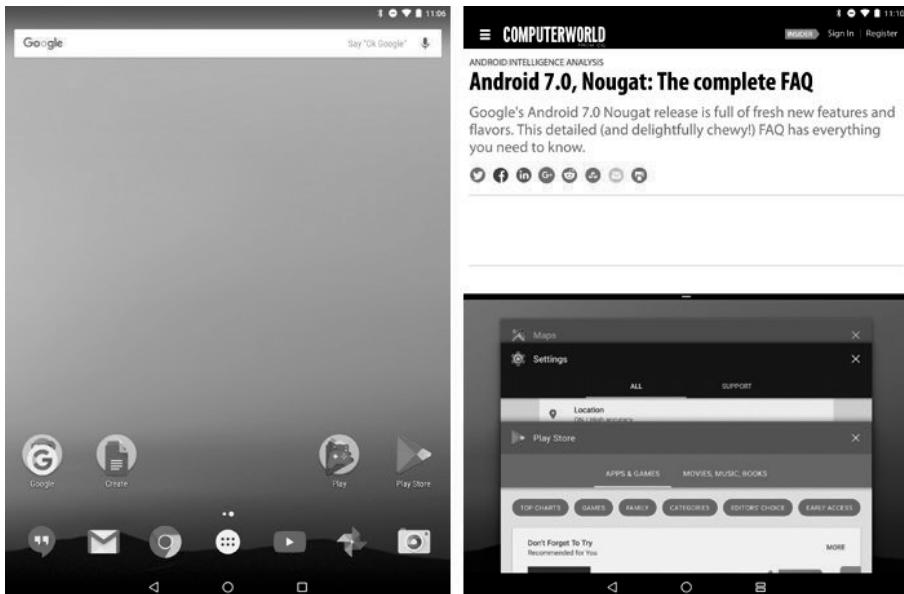
Slika 1.13, Android 6.0 Marshmallow korisnički interfejs
(slika preuzeta sa: <https://www.computerworld.com/article/3235946/android-versions-a-living-history-from-1-0-to-today.html>)

Osim već navedenih novih funkcionalnosti, Marshmallow je uveo veći broj novina, od kojih su najbitnije:

- Direct Share opcija.
- Automatski bekap aplikacija.
- 4k prikaz za aplikacije.
- Eksperimentalna podrška za više prozora.
- Unapređenja USB
- Unicode 7.0 i 8.0 emoji.
- Dvostruki pritisak na Power dugme otvara kameru.

1.1.13. Android 7.0 Nougat

Sledeća glavna verzija Androida izšla je na tržište u avgustu 2016, pod oznakom Android 7.0 Nougat (API nivo 24). Nešto kasnije u toku iste godine, izšla je i verija 7.1 Nougat (API nivo 25), kao i dve manje verzije 7.1.1 i 7.1.2. Glavna nova funkcionalnost koja je vidljiva korisnicima je podrška za podeljeni ekran (engl. *split screen*). Ova funkcionalnost, uz izgled korisničkog interfejsa, prikazana je na slici 1.14. Nougat je takođe uveo nova poboljšanja i optimizacije sa ciljem štednje energije i produžavanja životnog veka baterije, kao i smanjenje potrošnje RAM-a. Režim rada Doze, koji je uveden u Android 6.0, dodatno je poboljšan.



Slika 1.14, Android 7.0 Nougat korisnički interfejs i nova opcija podeljenog ekранa

(slika preuzeta sa: <https://www.computerworld.com/article/3235946/android-versions-a-living-history-from-1-0-to-today.html>)

Osim ovih najbitnijih izmena, druge funkcionalnosti koje je Nougat uveo su:

- Podrška za enkripciju fajlova.
- Novi Unicode 9.0 emoji.
- Dodata opcija Clear All u listi nedavno korišćenih aplikacija.
- Daydream platforma virtuelne stvarnosti (VR interfejs).
- Poboljšanja pregledača fajlova.
- Dodatne opcije u podešavanjima.

- Novi JIT kompajler, koji ubrzava instalacije aplikacija, a smanjuje veličinu kompajliranog koda.
- Slika u slici – podrška za Android TV.
- Vulkan 3D API za renderovanje.
- Poboljšane performanse ekrana osetljivog na dodir
- Podrška za tastaturu sa ubacivanjem slika
- Google asistent

1.1.14. Android 8.0 Oreo

Osma glavna verzija Android sistema je puštena na tržište u avgustu 2017. godine, pod nazivom Android 8.0 Oreo (API nivo 26). U decembru 2017. godine, puštena je još jedna manja verzija, Android 8.1.0 Oreo (API nivo 27). Jedna od najvažnijih novih opcija je specijalan oblik režima rada sa više prozora pod nazivom slika u slici (Picture-in-picture, PIP), koji je prikazan na slici 1.15. Ovaj režim rada je prvenstveno bio dostupan samo na Android TV uređajima, ali je sa Androidom 8.0 postao dostupan na svim Android uređajima. Još jedna od novih funkcionalnosti je mogućnost da se odlože (engl. *snooze*) notifikacije, kao i posebni kanali za notifikacije putem kojih je moguće podesiti na koji način želimo da nas neka aplikacija obavesti. Još jedna od značajnijih izmena je restrukturirana sekcija podešavanja, gde su podsekcije grupisane tako da slične opcije budu zajedno. Dodat je i veći broj novih API-ja, od kojih se izdvaja API za neuronske mreže. Osim navedenih, dodat je veći broj novih funkcionalnosti, od kojih su najznačajnije:

- Pojačana sigurnost kroz Google Play Protect.
- Wi-Fi asistent.
- Automatska aktivacija isključenog WiFi ukoliko se prepozna poznata mreža.
- Adaptivne ikonice.
- Autofill opcija na nivou celog sistema.
- Podrška za više displeja.
- Optimizacija aplikacija koje su u pozadini.
- Dva puta brže podizanje sistema u poređenju sa varijantom Nougat (prema Google, testirano na njihovom Pixel uređaju).
- Vizuelne izmene za Power Off i Restart opcije, sa novim ekranom.
- Vizuelna izmena Toast poruka.



Slika 1.15, Android 8.0 Oreo korisnički interfejs i nova opcija slike u slici (izvor: <https://developer.android.com/about/versions/oreo/android-8.0>)

1.1.15. Android 9.0 Pie

Deveta glavna verzija Androida, pod nazivom Android 9.0 Pie (API nivo 28), puštena je na tržište u avgustu 2018. Jedna od najinteresantnijih novih opcija je podrška za pozicioniranje u zatvorenom prostoru uz pomoć Wi-Fi RTT (*Wi-Fi Round Trip Time*), koja je prikazana na slici 1.16. Upotreboom RTT API-ja aplikacije mogu da izmere udaljenost do obližnjih Wi-Fi pristupnih tačka (koje imaju opciju RTT) i da na taj način odrede lokaciju uređaja (ovo zahteva da su servis lokacije i Wi-Fi skeniranje uključeni u podešavanjima sistema). Ukoliko uređaj uspe da izmeri udaljenost do tri ili više pristupnih tačaka, pomoću multilateracije moguće je proceniti poziciju uređaja sa tipičnom preciznošću od 1-2 metra. Poboljšane su i notifikacije i poruke, tako da je od Androida 9.0 moguće prikazati sliku u notifikaciji poruke. Još jedna od izmena je poboljšana podrška za više kamere na uređaju, tako da je moguće pristupiti istovremeno strimovima dve ili više fizičkih kamera.



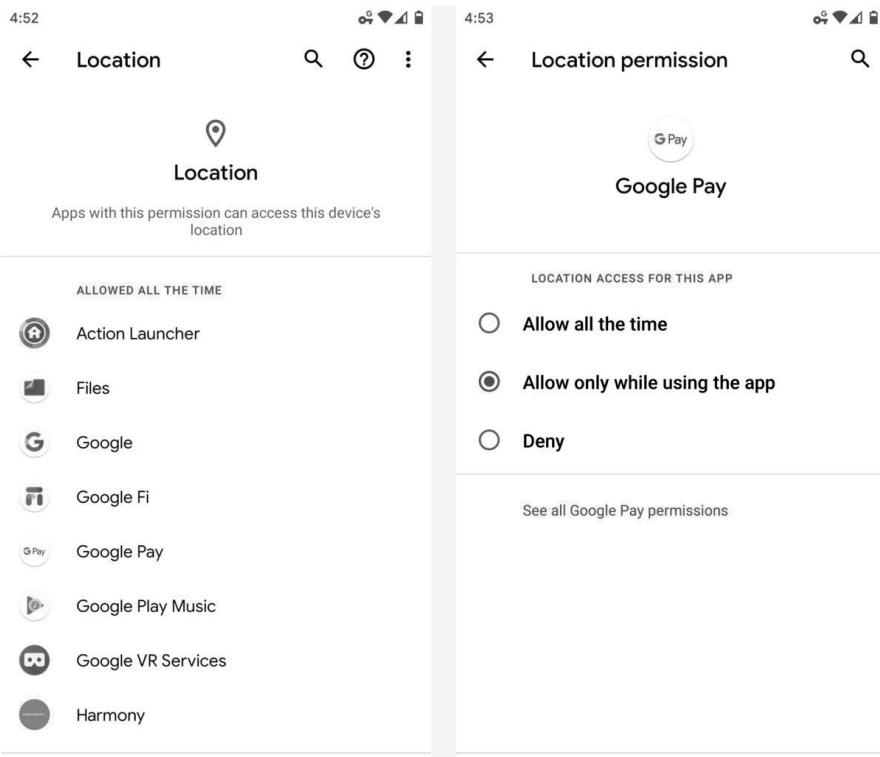
Slika 1.16, Android 9.0 Pie nova opcija pozicioniranja u zatvorenom prostoru (slika preuzeta sa: <https://developer.android.com/about/versions/pie/android-9.0>)

Osim već navedenih poboljšanja, Pie je uveo veći broj novih funkcionalnosti, među kojima se najviše ističu:

- Poboljšan korisnički interfejs.
- Nove tranzicije prilikom prelaska između različitih aplikacija, ili aktivnosti u okviru jedne aplikacije.
- Procenat baterije uvek prikazan na Always-On ekranu.
- Android Dashboard, koji prikazuje korisniku koliko vremena provodi na uređaju i u određenim aplikacijama, uz opciju da se postave vremenska ograničenja na aplikacije.
- Određen broj eksperimentalnih funkcionalnosti, poput opcije za automatsko uključivanje Bluetooth-a za vreme vožnje.
- Redizajnirani slajder za jačinu zvuka.
- Novi interfejs baziran na većoj upotrebi pokreta prevlačenja (swipe) u odnosu na prethodne verzije.
- Auto-Brightness koji modifikuje osvetljenje ekrana na osnovu navika korisnika.

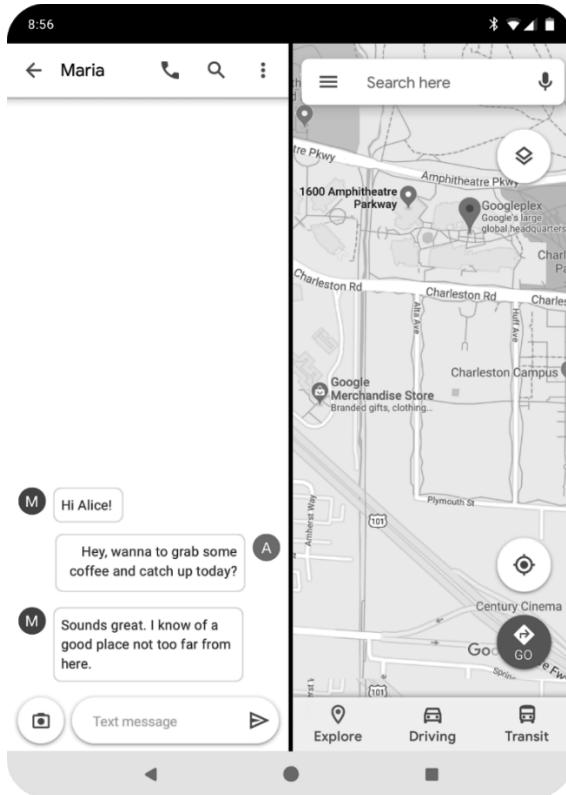
1.1.16. Android 10

Najnovija verzija Androida koja je trenutno aktuelna, dostupna je na tržištu od septembra 2019. godine i poznata je pod jednostavnim nazivom Android 10 (API nivo 29). Jedna od najbitnijih izmena po pitanju sigurnosti se nalazi u kontroli dozvola, gde se korisniku daje puna kontrola da odredi koja aplikacija, kada i kako sme da koristi lokaciju uređaja, kao što je prikazano na slici 1.17.



Slika 1.17, Android 10 kontrola aplikacija koje pristupaju lokaciji
(slika preuzeta sa: <https://www.computerworld.com/article/3235946/android-versions-a-living-history-from-1-0-to-today.html>)

Još jedna od novina je opcija za podršku telefona sa savitljivim ekranima, uz održavanje kontinuiteta prikaza aplikacija u toku savijanja ili ispravljanja ekrana, kao što je prikazano na slici 1.18. Moguće je konfigurisati i emulator virtuelnog uređaja sa savitljivim ekranom.



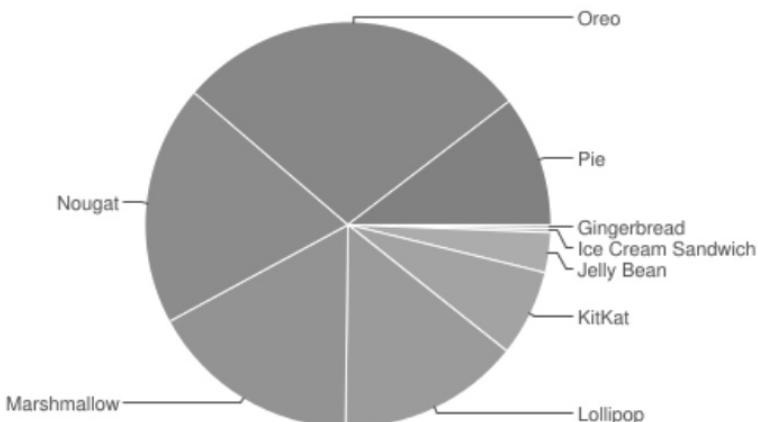
Slika 1.18, Android 10 podrška za savitljive ekrane
(slika preuzeta sa: <https://developer.android.com/about/versions/10/highlights>)

Još jedna značajna novina je podrška za 5G mobilne mreže, koje su trenutno u procesu uvođenja u većem broju država u svetu. Od drugih novih funkcionalnosti, izdvajaju se:

- Smart Reply opcija, koja se javlja u okviru notifikacije u zavisnosti od konteksta (na primer slanje pametnog odgovora u slučaju notifikacije za poruku, ili otvaranja mape u slučaju da u notifikaciji postoji adresa).
- Tamna tema za uslove lošeg osvetljenja čime se dodatno štedi baterija.
- Deljenje prečica, čime se omogućava direktno deljenje sadržaja sa kontaktom.
- Dinamička dubina za fotografije, čime se može izmeniti zamućenost pozadine nakon snimanja fotografije.
- Poboljšanja u biometrijskog autentifikaciji u aplikacijama.

1.2. Rasprostranjenost Android verzija na tržištu

Veoma važna informacija za programere prilikom razvoja aplikacija za Android je da poznaju situaciju na tržištu, odnosno rasprostranjenost verzija. Ova informacija je uvek dostupna na zvaničnoj stranici za Android programere, koja se nalazi na adresi: <https://developer.android.com/about/dashboards/index.html>. Na osnovu te informacije programeri mogu da odluče šta je ciljana grupa uređaja i koja je njihova verzija, kako bi maksimalno iskoristili tržište. Na primer, ukoliko je ciljana verzija Android 10, koji ima manje od 1% udela u tržištu, aplikacija možda neće dostići željeni nivo prodaje, jer nije izvesno da će moći da se izvršava na starijim verzijama. Sa druge strane, ukoliko je ciljana verzija na primer Android 4.1 Jelly Bean, biće pokriveno praktično 100% tržišta, odnosno aplikacija će moći da se izvršava na ogromnom broju telefona, ali za takvu aplikaciju nije moguće koristiti naprednije funkcionalnosti koje su uvedene u verzijama nakon Android 4.1. Google periodično radi snimanje trenutne raspodele svih verzija Androida (snimanjem svih aktivnih Android telefona u određenom vremenskom intervalu), a u trenutku pisanja ovog udžbenika dostupne su bile informacije iz početka maja 2019. godine. Grafički prikaz ove statistike prikazan je na slici 1.19, a nakon toga i tabelarno, u tabeli 1-1.



Slika 1.19, raspodela različitih verzija Androida na tržištu
(maj 2019. godine, izvor: <https://developer.android.com/about/dashboards>)

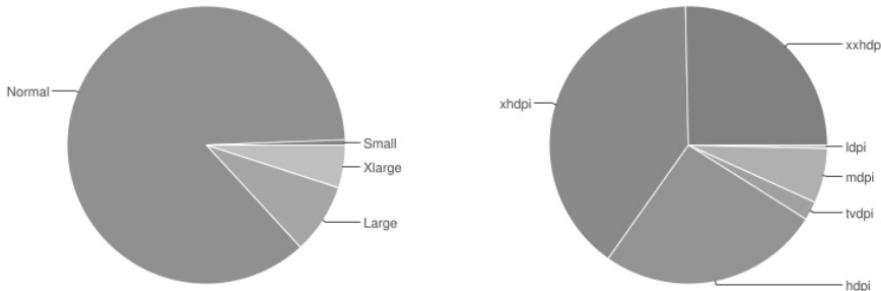
Potrebno je samo uočiti da se Android 10 ne nalazi prikazan u raspodeli, pošto je izašao na tržište nakon perioda kada je snimano stanje aktivnih Androida. Uopšteno gledano, najnovije verzije Androida često nude odlične i inovativne API-je za naše aplikacije, ali je obično neophodno podržati i starije verzije, bar dok se dovoljno uređaja ne ažurira na novije verzije. U narednim poglavljima će biti objašnjeno i kako se može iskoristiti prednost najnovijih API-ja, uz

kontinuiranu podršku starijih verzija, putem specificiranja minimalne SDK verzije i ciljane SDK verzije unutar AndroidManifest.xml fajla. U praksi se raspodela sa slike 1.19 veoma često koristi za definisanje ove dve vrednosti. Kao primer dobre prakse pokazalo se da bi trebalo podržati 90% aktivnih uređaja.

Verzija	Kodno ime	API	Udeo u tržištu
2.3.3 -	Gingerbread	10	0.30%
2.3.7			
4.0.3 -	Ice Cream Sandwich	15	0.30%
4.0.4			
4.1.x	Jelly Bean	16	1.20%
4.2.x		17	1.50%
4.3		18	0.50%
4.4	KitKat	19	6.90%
5	Lollipop	21	3.00%
5.1		22	11.50%
6	Marshmallow	23	16.90%
7	Nougat	24	11.40%
7.1		25	7.80%
8	Oreo	26	12.90%
8.1		27	15.40%
9	Pie	28	10.40%

Tabela 1-1, udeo različitih verzija Androida u tržištu (maj 2019. godine)

Osim informacije o verziji Androida na uređajima koji se nalaze na tržištu, još jedna bitna stavka je podatak o broju uređaja na tržištu koji imaju određenu konfiguraciju ekrana. Konfiguracija ekrana se definiše preko dva parametra, same veličine ekrana i gustine piksela. Android se izvršava na velikom broju različitih uređaja koji imaju različite veličine ekrana i gustine piksela. Sistem izvršava osnovno skaliranje kako bi adaptirao prikaz korisničkog interfejsa na različitim ekranima, ali kako bi ovaj proces prošao glatko programeri takođe moraju da posvete pažnju tome prilikom razvoja aplikacije. Podatak o konfiguracijama ekrana uređaja koji se nalaze na tržištu je takođe javno dostupan programerima, i može se proveriti na istoj adresi: <https://developer.android.com/about/dashboards>. Statistika iz maja 2019. godine prikazana je na slici 1.20, a nakon toga i tabelarno, u tabeli 1-2.



Slika 1.20, raspodela veličine ekrana Android uređaja na tržištu (maj 2019. godine, izvor: <https://developer.android.com/about/dashboards>)

Veličina/gustina	ldpi	mdpi	tvdpi	hdpi	xhdpi	xxhdpi	Ukupno
Mali	0.40%				0.10%	0.10%	0.60%
Normalni		0.90%	0.30%	24.00%	37.70%	23.60%	86.50%
Veliki		2.40%	1.90%	0.60%	1.60%	1.70%	8.20%
Veoma veliki		3.10%		1.30%	0.60%		5.00%
Ukupno	0.40%	6.40%	2.20%	25.90%	40.00%	25.40%	

Tabela 1-2, udeo različitih konfiguracija ekrana Androida u tržištu (maj 2019. godine)

1.3. Udeo Android operativnog sistema na tržištu mobilnih uređaja

Android operativni sistem nije jedini operativni sistem za mobilne uređaje na tržištu. U trenutku izlaska Androida na tržište, konkurenčija je bila veoma ozbiljna (slika 1.21). Tada suvereni lider bila je Nokia (Symbian), sa skoro dvotrećinskim udedom u tržištu mobilnih uređaja. Ozbiljnju konkurenčiju su dalje činili BlackBerry (RIM), koji je na svom vrhuncu imao i preko 20% udela na tržištu, naročito poslovnih korisnika, kao i Microsoft sa oko 10% i iOS (Apple) sa oko 10%. Operativni sistem Symbian je bio prvi popularni operativni sistem za mobilne telefone na svetu, i smatra se za prvi moderni operativni sistem za mobilne uređaje. Do 2009. godine, udeo Symbian-a je opao na oko 50% tržišta. BlackBerry je takođe bio jedan od pionira u oblasti operativnih sistema za mobilne uređaje, čija su ciljana grupa bili poslovni korisnici. Tada revolucionarna opcija Push email, koja je omogućavala da korisniku mejl bude dostavljen automatski sa servera na telefon, rezultirala je činjenicom da je veliki broj kompanija u svetu svojim zaposlenima davao neki od BlackBerry modela kao poslovni telefon. Iako su imali ogromnu početnu prednost, Symbian i BlackBerry su praktično zbrisani sa tržišta dolaskom novih operativnih sistema. Osim Androida, na tržište je ušao i Apple sa operativnim sistemom iOS i telefonom iPhone 2007. godine.



Slika 1.21, iOS, Android, BlackBerry i Symbian operativni sistemi

Apple je 2007. godine napravio revoluciju na tržištu mobilnih uređaja. Pre iPhone uređaja, pametni telefoni još uvek nisu bili masovni. Danas je iOS drugi najpopularniji operativni sistem za mobilne telefone, sa udelom na tržištu koji u poslednjih 10 godina varira između 14 i 21%. Android je od ulaska na tržište beležio konstantni rast. Oko 7 miliona Android telefona je prodato 2009. godine, da bi ta cifra prešla milijardu prodatih uređaja 2015. godine. Danas je Android ubedljivo prvi operativni sistem za mobilne uređaje u svetu, sa udelom koji poslednjih godina varira između 80 i 85%. Takav trend se očekuje i u narednih nekoliko godina, kao što se može videti iz tabele 1-3 (izvor: <https://www.idc.com/promo/smartphone-market-share/os>). Svi ostali operativni sistemi zajedno imaju zanemarljiv udio.

Godina	2017	2018	2019	2020	2021	2022	2023
Android	85.10%	85.10%	87.00%	87.00%	87.20%	87.30%	87.40%
iOS	14.70%	14.90%	13.00%	13.00%	12.80%	12.70%	12.60%
Ostali	0.20%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Ukupno	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%

Tabela 1-3, udio OS na tržištu sa očekivanjima za naredne godine (IDC)

Kada se pogleda udio tržišta prema proizvođačima mobilnih telefona, prema renomiranoj kompaniji za istraživanje tržišta IDC (International Data Corporation, veb sajt <https://www.idc.com/>) može se uočiti da dominiraju proizvođači sa dalekog istoka. U tabeli 1-4 prikazan je udio tržišta po proizvođačima po kvartalima, u periodu od poslednjeg kvartala 2017. do drugog kvartala 2019. godine. Ukupna godišnja prodaja mobilnih telefona iznosi oko 1.5 milijardi isporučenih uređaja. Samsung je već nekoliko godina lider na tržištu, sa otprilike 20% udela na tržištu, odnosno skoro 300 miliona prodatih telefona. Iako je potražnja za Samsung flagship telefonima opala, pošto većina kupaca traži

jeftinije opcije, srednja klasa predvođena A serijom je doprinela rastu prodaje ovog proizvođača. Iako postoje trgovinske tenzije između Sjedinjenih Američkih Država i Kine, Huawei je održao drugu poziciju sa preko 200 miliona prodatih uređaja. Huawei je vodeći proizvođač na kineskom tržištu, gde drži lidersku poziciju već nekoliko godina. Na trećem mestu proizvođača nalazi se Apple, čija prodaja obično skače u poslednjem kvartalu svake godine, kada izlazi novi model iPhone. Na četvrtom mestu nalazi se Xiaomi, koji najveći deo uređaja isporučuje u Kinu, Indiju i Indoneziju. Još jedan kineski proizvođač, Oppo, koji najveći broj uređaja isporučuje u Kinu i Indiju, nalazi se na petom mestu. Svi ostali proizvođači, među kojima su i giganti poput LG, imaju ideo manji od 6%.

Kvartal	2017Q4	2018Q1	2018Q2	2018Q3	2018Q4	2019Q1	2019Q2
Samsung	18.90%	23.50%	21.00%	20.30%	18.80%	23.00%	22.90%
Huawei	10.70%	11.80%	15.90%	14.60%	16.20%	18.90%	17.70%
Apple	19.60%	15.70%	12.10%	13.20%	18.30%	11.80%	10.20%
Xiaomi	7.10%	8.40%	9.50%	9.50%	6.70%	8.90%	9.70%
Oppo	6.90%	7.40%	8.60%	8.40%	7.90%	7.40%	8.90%
Svi ostali	36.80%	33.20%	32.90%	34.00%	32.00%	30.10%	30.60%
Ukupno	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%

Tabela 1-4, ideo proizvođača mobilnih uređaja na tržištu (IDC)

1.4. Tržište mobilnih aplikacija

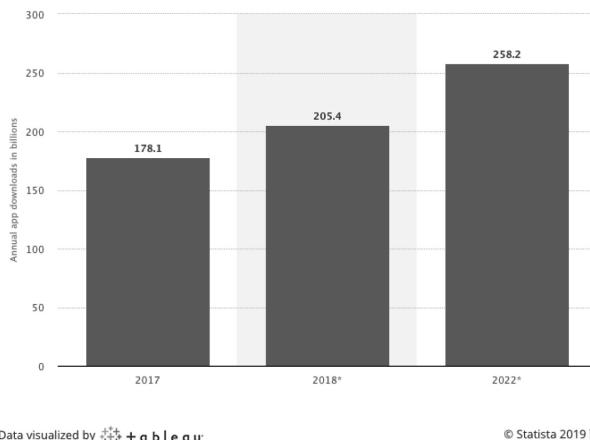
U svetu postoji oko 2.7 milijardi korisnika mobilnih uređaja, pa ne čudi da industrija mobilnih aplikacija cveta. Tržište mobilnih aplikacija već godinama raste stabilnom stopom, bez ikakvih znakova usporavanja u doglednoj budućnosti. Sve je potpuno jasno ukoliko podignemo pogled sa svog telefona i pogledamo oko sebe – većina ljudi takođe je zagledana u svoje mobilne uređaje. Prema nekim studijama, prosečna osoba proverava svoj telefon na svakih 12 minuta, dok svaka deseta osoba proverava telefon na 4 minuta. Mobilni telefoni koriste se svuda – na poslu, u kući, na ulici, dok jedemo, dok se odmaramo u fotelji, u krevetu, čak i u automobilima (što je naročito opasno i izaziva veliki broj saobraćajnih nesreća širom sveta). Od toga, 90% upotrebe mobilnih uređaja se odnosi na upotrebu različitih aplikacija.

Iako tolika vezanost za mobilne uređaje možda nije ohrabrujuća vest za društvo sa psihološkog i sociološkog aspekta, sa stanovišta razvoja mobilnih aplikacija to je izuzetno dobra vest za sve koji žele da se bave ovim poslom. Međutim, kako bismo bili uspešni u ovom poslu, potrebno je da se dese dve ključne stvari: korisnici moraju da preuzmu našu aplikaciju, i zatim da je koriste. Kako bismo

bolje razumeli tržište mobilnih aplikacija, potrebno je poznavati neke statističke podatke koji nam mogu ukazati na veličinu i vrednost samog tržišta:

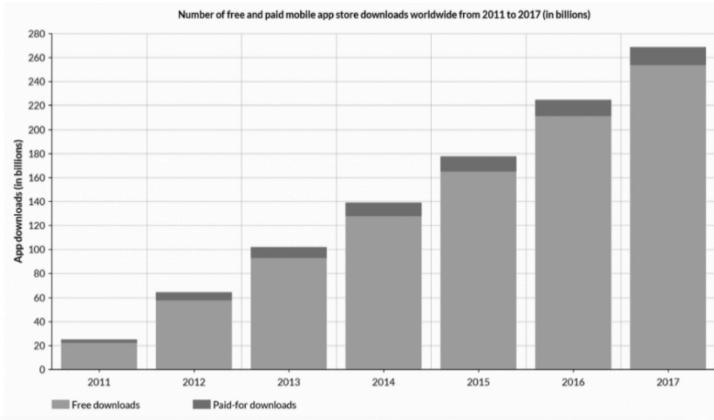
- Očekuje se da mobilne aplikacije donesu prihod od oko 190 milijardi dolara u 2020. godini.
- Google Play Store sadrži oko 2.8 miliona aplikacija za preuzimanje.
- Apple App Store sadrži oko 2.2 miliona aplikacija za preuzimanje.
- Veliki broj korisnika mlađih od 20 godina više od 50 puta dnevno otvori neku od aplikacija na svom uređaju (socijalne mreže, igre, aplikacije za razmenu poruka i slično).
- Polovina svih korisnika mobilnih telefona otvori neku aplikaciju više od 10 puta dnevno.
- Prosečni korisnik mobilnog telefona koristi prosečno 30 aplikacija svakog meseca.
- Prosečni korisnik mobilnog telefona proverava mobilni telefon čak i dok vozi automobil.

Kao izvor za statističke podatke tržišta mobilnih aplikacija korišćen je Statista veb sajt (<https://www.statista.com/>). Pogled na brojke dovoljno govori o popularnosti mobilnih aplikacija. Ukupan broj preuzimanja mobilnih aplikacija na svim platformama iznosi preko 200 milijardi godišnje, sa tendencijom rasta i u narednim godinama, kao što je prikazano na slici 1.22. Ukoliko se predviđanja eksperata pokažu kao tačna, od 2018. do 2022. godine broj preuzimanja će porasti za oko 25%, sa očekivanim brojem od skoro 260 milijardi preuzimanja aplikacija u 2022. godini. Najveći deo ovih preuzimanja naravno odlazi na Apple i Android, pošto sve ostale platforme imaju zanemarljiv uticaj na priložene brojke.



Slika 1.22, ukupan broj preuzimanja aplikacija na svim platformama
(izvor: <https://www.statista.com/>)

Još jedan bitan parametar jeste odnos preuzimanja besplatnih aplikacija, kao i onih koje se naplaćuju. Cilj razvoja mobilnih aplikacija je svakako zarada novca. Prilikom kreiranja aplikacije, potrebno je osmisliti i strategiju naplate. Jedna od mogućih strategija je naplata preuzimanja aplikacije, i to je verovatno ono što svakom programeru prvo padne na pamet. Međutim, postavlja se pitanje da li će korisnici preuzimati aplikaciju ukoliko se od njih očekuje da plate preuzimanje? Statistički podaci besplatnih i plaćenih preuzimanja aplikacija u posmatranom intervalu od 2011. godine zaključno sa 2018. godinom su prikazani na slici 1.23.



Slika 1.23, ukupan broj besplatnih i plaćenih preuzimanja u intervalu 2011-2017.

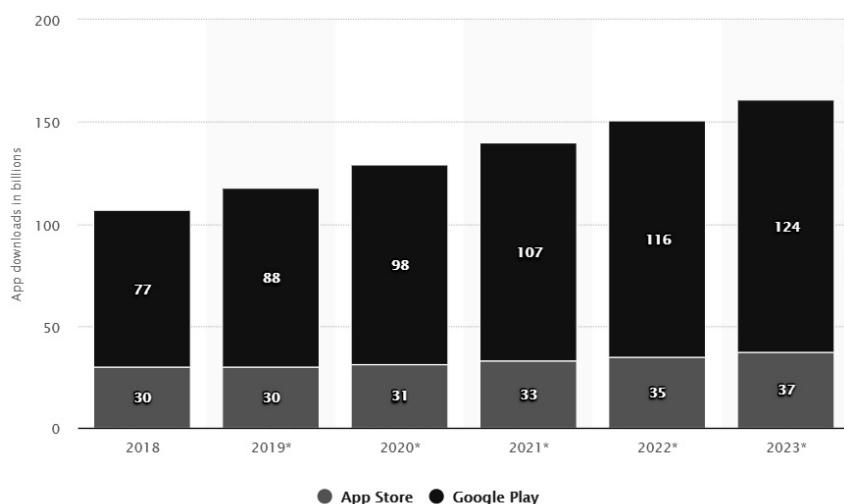
(izvor <https://www.statista.com/>).

Već na prvi pogled se može uočiti šta su korisnici skloni da preuzimaju, pošto se ogromna većina preuzimanja odnosi na besplatne aplikacije. Prema statistikama, 98% ukupne zarade koje generišu mobilne aplikacije dolazi od besplatnih aplikacija, pošto je jako mali broj korisnika spreman da plati preuzimanje aplikacije. To ne znači da ideja naplate preuzimanja aplikacije u potpunosti loša. U zavisnosti od tipa aplikacije, obe pomenute strategije mogu biti uspešne. Ukoliko se naplaćuje preuzimanje aplikacije, može se očekivati da će manji broj korisnika zaista preuzeti aplikaciju. Razlog tome leži u činjenici da su korisnici navikli da im se aplikacije ponude besplatno – u smislu bez naplate preuzimanja. Dovoljno je da pogledamo naše telefone i razmislimo da li smo nekada instalirali neku aplikaciju čije je preuzimanje bilo naplaćeno. Međutim, korisnici koji su platili preuzimanje aplikacije (iako su u manjini), su mnogo više posvećeni korišćenju aplikacije, pošto niko neće hteti da potroši novac na aplikaciju i da je zatim nikada više ne koristi.

Sa druge strane, nije neophodno naplatiti preuzimanje aplikacije kako bi se zaradio novac. Ukoliko na primer firma koja ima razrađen biznis ponudi aplikaciju koja će proširiti taj biznis, naročito u smeru poboljšanja zadovoljstva

korisnika, ta aplikacija će najverovatnije biti besplatna, a uvećanje zarade će se odraziti kroz više klijenata primarnog biznisa te firme. Dalje, moguće je zaraditi novac direktno od aplikacije, čak i ako je ponuđena sa besplatnim preuzimanjem. Postoji veći broj strategija, poput kupovine unutar aplikacije (engl. *in-app purchases*) ili preko reklama unutar aplikacije.

Kao što je ranije navedeno, ogromna većina preuzimanja mobilnih aplikacija odlazi na samo dve platforme – Apple i Android. Samim tim, postoje dva mesta odakle korisnici preuzimaju praktično sve aplikacije. Apple korisnici sa iOS uređajima preuzimaju aplikacije sa Apple App Store, dok korisnici Android uređaja svoje aplikacije preuzimaju sa Google Play Store. Na slici 1.24 prikazan je uporedni prikaz broja preuzimanja po platformama za 2018. i 2019. godinu, sa predviđanjima stručnjaka do 2023. godine. Može se uočiti da više preuzimanja dolazi sa Google Play, pošto Android ima više korisnika. Međutim, može se uočiti i da broj preuzimanja raste svake godine na obe platforme, sa tim što Google Play beleži porast od oko 13% u 2019. godini u odnosu na 2018. godinu, dok Apple App Store beleži značajno manji porast, od oko 3% godišnje.

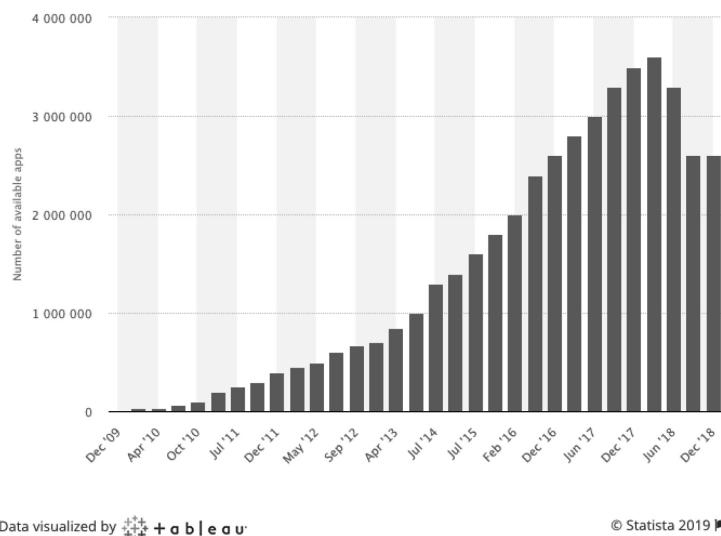


Slika 1.24, uporedni broj preuzimanja sa Google Play i App Store
(izvor: <https://www.statista.com/>)

Ukoliko se posmatraju najčešće preuzimane kategorije aplikacija na obe platforme, najviše preuzimanja imaju igre (oko 25% svih preuzimanja), zatim poslovne aplikacije (oko 10%), zatim edukacione (oko 8.5%), životni stil (oko 8.3%), zabavne aplikacije (oko 6%), dok sve ostale kategorije imaju udeo manji od 5%.

Kako bismo shvatili kolika je zapravo konkurenca, potrebno je pogledati ukupan broj aplikacija dostupnih za preuzimanje. Statistički podaci o broju

aplikacija koje su dostupne za preuzimanje na Google Play u periodu od 2009. godine do kraja 2018. godine su prikazani na slici 1.25. Sa grafika se može uočiti ogroman rast broja aplikacija, od skromnih početaka, preko prvih milion aplikacija (2014. godine), zatim cifre od dva miliona dostupnih aplikacija koja je dostignuta 2016. godine, sve do maksimuma od oko 3.5 miliona dostupnih aplikacija 2017. godine. Međutim, od druge polovine 2017. godine počinje da se beleži pad broja dostupnih aplikacija. Iako to možda deluje neočekivano, rezultati potvrđuju ogromnu konkureniju aplikacija na Google Play – neke aplikacije prosto nisu bile sposobne da prežive. Nasuprot tome, kao što je ranije navedeno, ukupan broj preuzimanja aplikacija raste svake godine. Trenutno je raspoloživo oko 2.5 miliona aplikacija na Google Play, i ne očekuje se dalji pad. Pretpostavlja se da će broj aplikacija ostati na ovom nivou, a zatim u narednim godinama početi ponovo da raste.



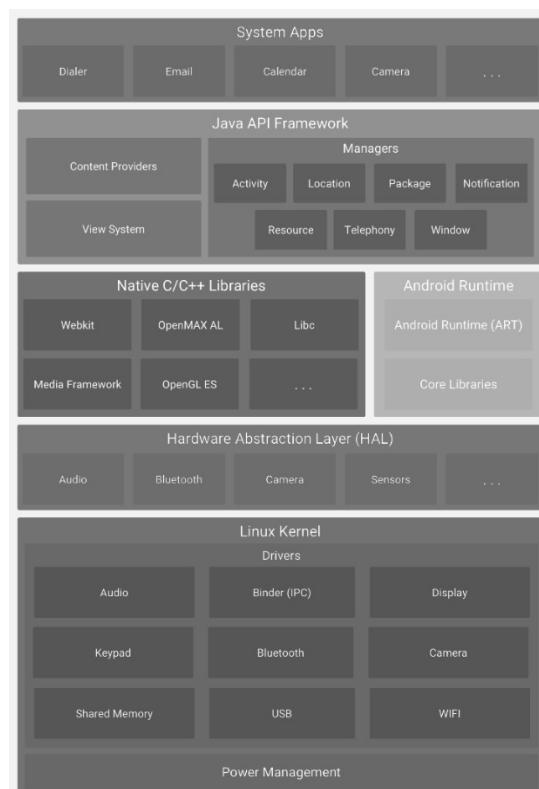
Slika 1.25, broj aplikacija dostupnih na Google Play Store
(izvor: <https://www.statista.com/>)

2. Android platforma

Programer bi trebalo dobro da upozna platformu sa kojom radi, jer isključivo sa dobrim razumevanjem sistema može izvući maksimum iz svojih aplikacija. Programiranje za Android se najčešće radi u programskom jeziku Java, međutim, postoje određene specifičnosti sa kojima se čisti Java programeri nisu prethodno susretali, i koje je neophodno uvesti i objasniti na početku. U ovom poglavlju opisana je arhitektura Android platforme, softverski stek, zatim izvršno okruženje (engl. *runtime*), kao i osnovni detalji Android aplikacija.

2.1. Arhitektura Android platforme

Android platforma je zasnovana na softverskom steku na čijem se dnu nalazi Linux jezgro. Na slici 2.1 prikazani su osnovni slojevi i komponente platforme.



Slika 2.1, Android softverski stek
(izvor: <https://developer.android.com/guide/platform>)

Osnova Android platforme leži u Linux jezgru. Ovaj modul zadužen je za upravljanje ključnim sistemskim servisima i resursima, poput memorije, procesa, niti, hardvera i potrošnje energije. ART (engl. *Android Runtime*) se u potpunosti oslanja na Linux kernel za ove ključne sistemske servise. Upotreba Linux jezgra omogućava Androidu da iskoristi njegove ključne sigurnosne funkcionalnosti i omogućava proizvođačima telefona da razvijaju hardverske drajvere za poznato jezgro.

Linux jezgro se koristi godinama u velikom broju različitih okruženja koja su osetljiva po pitanju sigurnosti, i danas se smatra da je to stabilno i sigurno jezgro kome veruje veliki broj korporacija. Sa stanovišta mobilnih uređaja, Linux jezgro pruža nekoliko ključnih sigurnosnih funkcija, od kojih su najvažniji model dozvola (privilegija) na nivou korisnika, izolacija procesa, kao i sprečavanje da jedan korisnik čita fajlove drugog korisnika, pošto je Linux višekorisnički operativni sistem.

Iznad Linux jezgra nalazi se sloj apstrakcije hardvera (engl. HAL – *hardware abstraction layer*). Ovaj sloj je zadužen da obezbedi set standardnih interfejsa koji otkrivaju hardverske karakteristike Java API okviru višeg nivoa. U ovom sloju se nalazi više biblioteka, a svaka biblioteka je zadužena da implementira interfejs ka određenoj hardverskoj komponenti, poput modula za kameru, Bluetooth ili nekog od senzora dostupnih na uređaju. Kada Java API napravi poziv da pristupi određenoj hardverskoj komponenti uređaja, Android sistem učitava biblioteku za traženu komponentu.

Iznad sloja apstrakcije hardvera nalazi se Android izvršno okruženje. Počev od Android verzije 5.0 (API nivo 21), svaka aplikacija se izvršava u svom procesu i sa svojom instancom ART. ART je implementiran na takav način da može da pokrene više virtuelnih mašina na uređajima sa ograničenom memorijom, izvršavanjem DEX fajlova. DEX je zapravo bajtkod dizajniran specijalno za Android, tako što je optimizovan za minimalni memorijski otisak. Pre Android 5.0, kao izvršno okruženje koristio se Dalvik. Ukoliko se aplikacija ispravno izvršava na ART, onda bi trebalo da radi bez problema i na Dalviku, ali obrnuto ne mora da važi. Android sadrži i skup ključnih izvršnih biblioteka, koje pružaju veći deo funkcionalnosti programskog jezika Java, koji je najbliži Java verziji 8.

Na ovom nivou se, pored ART, nalazi i skup izvornih C/C++ biblioteka. Mnoge sistemske komponente i servisi Androida su napisani u izvornom kodu koji zahteva ove izvorne biblioteke. Android platforma kroz Java API otkriva aplikacijama funkcionalnost nekih od ovih biblioteka. Na primer, moguće je pristupiti OpenGL ES modulu (Open Graphics Library – podrška za 2D i 3D grafiku visokih performansi) kroz Java OpenGL API kako bi se u aplikaciji dodala podrška za iscrtavanje i manipulisanje 2D i 3D grafikom. Moguće je i razvijati aplikacije direktno upotrebom C ili C++ koda, u tom slučaju se može koristiti Android NDK za pristup izvornim bibliotekama.

Sledeći sloj predstavlja Java API okvir, preko koga je programerima dostupan ceo skup funkcionalnosti Androida. Preko ovog Java okvira se na jednostavan način mogu izgraditi aplikacije, putem upotrebe modularnih sistemskih komponenta i servisa, od kojih su najbitniji:

- View System, koji je bogat i lako proširiv. Pomoću njega se jednostavno pravi korisnički interfejs aplikacije, upotrebom lista, rasporeda komponenti, tekstualnih polja, dugmića itd.
- Resource Manager, koji omogućava pristup resursima aplikacije koji nisu kod, poput lokalizovanih stringova, slika i razmeštaja komponenti.
- Notification Manager, koji omogućava svim aplikacijama da prikažu obaveštenja u statusnoj liniji.
- Activity Manager, koji je zadužen za menadžment životnog ciklusa aplikacije i obezbeđuje zajednički navigacioni stek (engl. *back stack*).
- Content Provider komponente, koje omogućavaju aplikacijama da pristupe podacima drugih aplikacija, odnosno omogućavaju deljenje podataka.

Potrebno je napomenuti da programeri prilikom razvoja svojih aplikacija koriste identični API okvir koji koriste i sistemske aplikacije Androida.

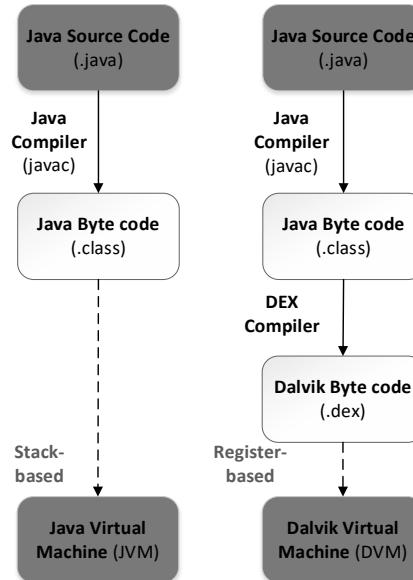
Na vrhu steka se nalazi nivo sistemskih aplikacija. To je skup predefinisanih osnovnih aplikacija koje dolaze zajedno sa Androidom, poput mejl klijenta, SMS poruka, kalendara, kamere, Internet pretraživača, kontakata, mape i slično. Ove preinstalirane aplikacije nemaju poseban status u odnosu na aplikacije koje korisnik kasnije odluči da instalira, tako da je moguće da se promeni podrazumevani veb čitač ili podrazumevana tastatura nekom aplikacijom razvijenom od treće strane, uz određene izuzetke poput sistemske aplikacije za podešavanja – Settings. Sistemske aplikacije, osim što služe kao korisničke aplikacije, pružaju ključne funkcionalnosti koje programeri mogu da iskoriste i pristupe im iz aplikacije koju razvijaju. Na primer, ukoliko je potrebno iz aplikacije poslati SMS poruku ili napraviti fotografiju kamerom, nije potrebno samostalno razviti tu funkcionalnost, već se može pozvati postojeća SMS aplikacija da prosledi poruku željenoj osobi, odnosno može se pozvati postojeća Android aplikacija kamera pomoću koje ćemo napraviti fotografiju.

2.2. Android Runtime

Izvršno okruženje koje koriste aplikacije i neki sistemski resursi na Androidu se naziva Android runtime (ART). Iako je ART već pomenut u prethodnom poglavljju, zaslužuje da mu se posveti malo više pažnje. Kao što je već navedeno, ART je dostupan počev od verzije Android 5.0. I ART i njegov prethodnik Dalvik

su kreirani specijalno u okviru projekta Android, prema specifičnim zahtevima mobilne platforme. ART izvršava DEX bajtkod (DEX – Dalvik Executable format). Sa stanovišta kompatibilnosti, i ART i Dalvik mogu da izvršavaju DEX bajtkod, tako da aplikacije koje su razvijene za Dalvik rade i kada se izvršavaju na ART, mada obrnuto ne važi.

Dalvik virtuelna mašina je veoma slična Java virtuelnoj mašini (JVM). U slučaju standardne Java, izvorni Java kod se prevodi u Java bajtkod, koga zatim interpretira JVM. Po istoj logici, Dalvik virtuelna mašina izvršava DEX bajtkod. Razlika između običnog i DEX bajtkoda leži u činjenici da je DEX optimizovan za mobilne uređaje (mali memorijski otisak, štednja energije, performanse). Uporedni prikaz standardne JVM i Dalvik virtuelne mašine, sa procesom prevodenja, prikazan je na slici 2.2. Androidova virtuelna mašina u potpunosti zavisi od Linux jezgra po pitanju upravljanja nitima i menadžmenta memorije. Svaka aplikacija se, uz pomoć Dalvik ili ART virtuelne mašine, izvršava u zasebnom procesu u okviru izvršnog okruženja.



Slika 2.2, uporedni prikaz JVM i Dalvik VM

ART je dodatno unapredio Dalvik virtuelnu mašinu. Uvođenjem AOT (Ahead-of-Time) prevodenja, poboljšane su performanse aplikacija, i smanjeno je vreme instalacije aplikacija. Za vreme instalacije, ART prevodi aplikacije upotrebom dex2oat alata. Ovaj alat prihvata .dex fajlove kao ulaz i generiše prevedeni izvršni fajl (.oat) za ciljani uređaj. Još jedna stvar koja je značajno poboljšana sa ART je sakupljanje đubreta (engl. *Garbage Collection* - GC). GC može drastično da degradira performanse aplikacije na mobilnim uređajima, što za rezultat ima loš

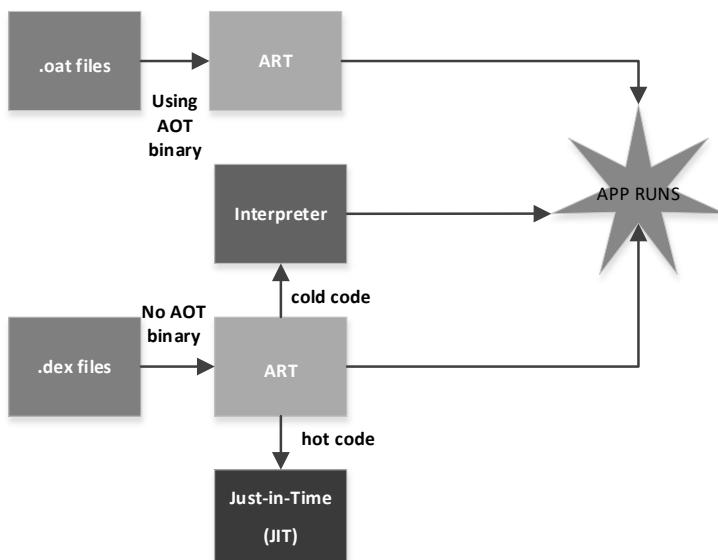
odziv korisničkog interfejsa i „sekanje“ prikaza. Poboljšanje je postignuto optimizacijama poput paralelnog procesiranja i kompaktnosti kako bi se smanjila potrošnja memorije. Na kraju, ART nudi veći broj funkcionalnosti koje olakšavaju razvoj i debagovanje aplikacija.

ART u sebi sadrži JIT (Just-in-Time) kompjajler, koji je komplementaran AOT kompjajleru. JIT poboljšava performanse samog izvršnog okruženja, čuva memoriju i ubrzava ažuriranje aplikacije. JIT dodatno poboljšava AOT kompjajler tako što se izbegava usporavanje sistema za vreme automatskog ažuriranja aplikacija. Prikaz arhitekture JIT prikazan je na slici 2.3.

Pojednostavljenog gledano, JIT u praksi funkcioniše na sledeći način. Kada korisnik pokrene aplikaciju, okida se ART koji učitava .dex fajl. U zavisnosti da li je .oat fajl (AOT verzija ,dex fajla) raspoloživ ili ne moguća su dva scenarija:

- Ukoliko je .oat fajl raspoloživ, ART ga koristi direktno. Iako se .oat fajlovi generišu redovno, to ne znači da uvek sadrže prevedeni kod.
- Ukoliko .oat fajl ne sadrži prevedeni kod, ART koristi JIT i interpreter kako bi izvršio .dex fajl.

JIT podaci se čuvaju u fajlu u sistemskom direktorijumu kome samo aplikacija može da pristupi. AOT kompjajler (dex2oat) parsira taj fajl i upravlja njegovom kompilacijom. Drugim rečima, kod uređaja koji koriste ART, veliki deo izvršnog bajtkoda se prevodi već u vreme instalacije. Prilikom narednog pokretanja aplikacije nije potrebno kompletno prevesti .dex izvršni bajtkod aplikacije, čime se značajno ubrzava rad.



Slika 2.3, JIT arhitektura

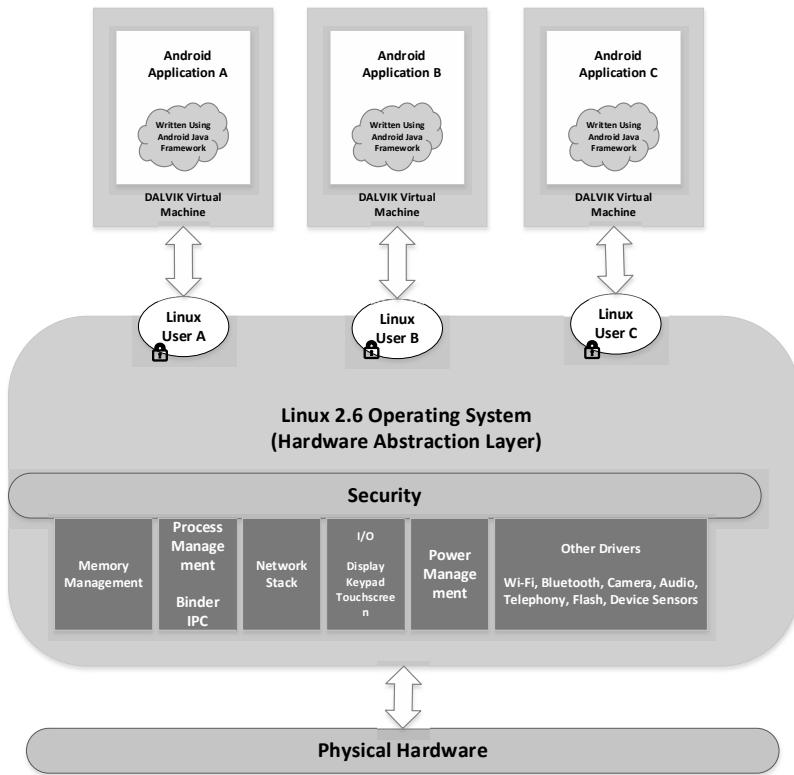
2.3. Android aplikacije

Android aplikacije se mogu pisati u programskim jezicima Java, Kotlin ili C++. Kotlin svakim danom dobija na popularnosti, međutim, još uvek je Java najčešće korišćeni jezik, pa je u ovoj knjizi fokus na razvoju aplikacija upotrebom Jave. Za razliku od standardne Jave SE, Android ne podržava neke biblioteke (na primer biblioteke za razvoj korisničkog interfejsa, AWT i Swing, za koje Android ima svoju posebnu biblioteku). Android SDK prevodi kod aplikacije, zajedno sa podacima i resursima u APK (Android Package), koji predstavlja arhivu sa .apk sufiksom. U APK fajlu se nalazi kompletan sadržaj koji je potreban uređaju da instalira tu aplikaciju.

Svaka Android aplikacija se izvršava u svom Linux procesu (slika 2.4), zaštićena Androidovim sigurnosnim funkcijama:

- Android OS je višekorisnički Linux sistem, u kome se svaka aplikacija posmatra kao drugi korisnik.
- Sistem svakoj aplikaciji dodeljuje jedinstveni korisnički Linux ID (ovaj ID koristi isključivo sistem i nije poznat samoj aplikaciji). Sistem postavlja privilegije pristupa svim fajlovima u aplikaciji, tako da samo korisnički ID dodeljen toj aplikaciji može da im pristupi. Drugim rečima, fajlovi jedne aplikacije su dostupni samo toj aplikaciji.
- Svaki proces poseduje svoju zasebnu virtuelnu mašinu, pa se aplikacija izvršava izolovano od drugih aplikacija.
- Android pokreće proces kada je potrebno izvršiti bilo koju komponentu aplikacije, a završava ga kada više nije potreban ili kada sistemu nedostaje memorije za nesmetan rad drugih aplikacija.

Android koristi princip najmanje privilegije. To znači da aplikacija podrazumevano ima pristup samo komponentama koje su joj potrebne za rad i ničemu više. Na taj način se kreira veoma sigurno okruženje u kome aplikacija ne može pristupiti delovima sistema za koje joj nije data dozvola. Ukoliko je aplikaciji potreban pristup nekom delu Android sistema, poput kamere, lokacije ili konekcije na Internet, ona može zahtevati dozvolu od korisnika, koji mora eksplicitno potvrditi (ili opovrgnuti) te dozvole.



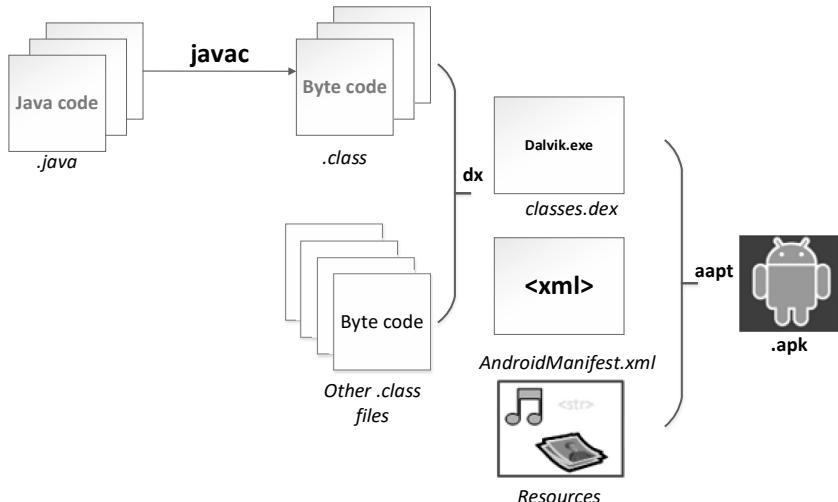
Slika 2.4, izvršavanje Android aplikacija u zasebnim Linux procesima

Svaka aplikacija se sastoji od komponenti. Postoje četiri osnovne komponente: aktivnosti, servisi, prijemnici obaveštenja (engl. *Broadcast receivers*) i komponente koje pružaju sadržaj (engl. *Content providers*). Svaki tip komponente ima svoju svrhu i svoj odgovarajući životni ciklus, koji definiše kako se komponenta kreira odnosno uništava. Osim Java fajlova koji sadrže kod, aplikacija sadrži i resurse poput dodatnih fajlova ili statičkog sadržaja koji kod aplikacije koristi. Ovi resursi mogu biti slike, definicije rasporeda komponenti (engl. *layout*), lokalizovani stringovi i slično. Android razdvaja resurse od koda, kako bi se mogli održavati nezavisno. Vrlo često se koriste i alternativni resursi za različite konfiguracije uređaja, koji se grupišu u odgovarajuće direktorijume. Android onda u vreme izvršavanja koristi odgovarajuće resurse na osnovu stvarne konfiguracije uređaja na kome se aplikacija trenutno izvršava. Svakom resursu se iz koda može pristupiti preko ID koji se generiše i čuva u R.java klasi.

Dodatno, svaka aplikacija mora da sadrži i posebni XML fajl pod strogo definisanim nazivom *AndroidManifest.xml*. U ovom fajlu su definisane osnovne informacije o aplikaciji koje su potrebne Android operativnom sistemu za ispravno izvršavanje aplikacije, poput imena aplikacije, definicije komponenti,

dozvola koje su potrebne aplikaciji za pristup zaštićenim delovima Android sistema i slično.

Android SDK prevodi sve ove komponente u APK, kao što je prikazano na slici 2.5. APK je format fajla koji Android koristi za distribuciju i instaliranje aplikacija, i sadrži sve što je potrebno kako bi Android ispravno instalirao aplikaciju na uređaju.



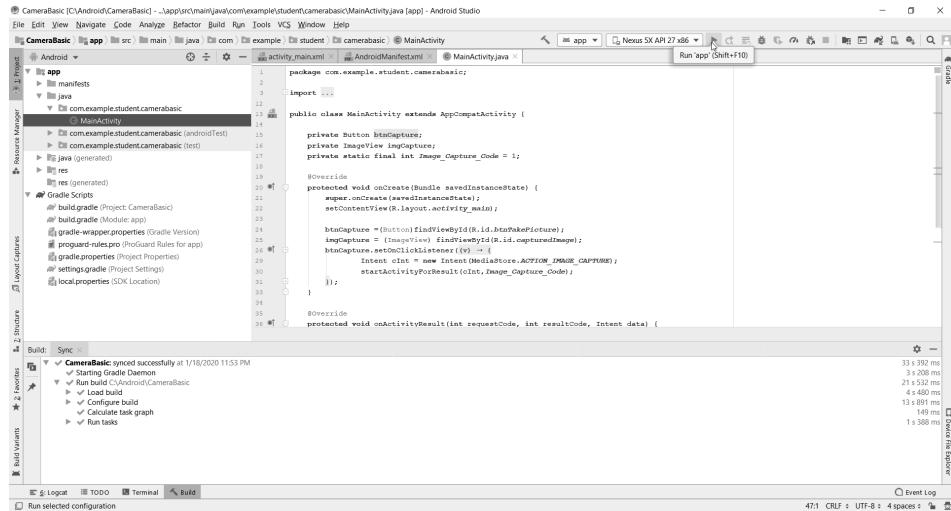
Slika 2.5, struktura APK

2.4. Okruženje za razvoj

Za razvoj Android aplikacija potrebno je odgovarajuće okruženje za razvoj (engl. IDE – *Integrated Development Environment*). Google je razvio veoma korisno okruženje za Android programere, pod nazivom Android Studio. Android Studio je zvanični IDE za razvoj Android aplikacija i jedini kojem je moguće razviti aplikacije za sve verzije Android operativnog sistema. Ranije je bilo moguće koristiti i Eclipse IDE za razvoj Android aplikacija upotrebom ADT (Android Developer Tools) proširenja, međutim Google je zvanično uskratio podršku ovom okruženju 2015. godine. To se poklapa sa izlaskom prve zvanične verzije Android Studio 1.0, koja je izašla na tržište u decembru 2014. Pošto Google nije nastavio dalje da razvija ADT, Eclipse IDE je i dalje moguće ograničeno koristiti i razvijati aplikacije zaključno sa verzijom Android 4.0, mada to nije preporučljivo.

Android Studio je okruženje bazirano na IntelliJ IDEA, a omogućava programiranje u programskim jezicima Java i Kotlin, uz podršku i za programske

jezik C++. Izgled okruženja prikazan je na slici 2.6. Okruženje je dizajnirano i organizovano na takav način da su najčešće upotrebljavane funkcije vidljive i lako dostupne, mada i programer sam može organizovati okruženje na način kako mu najviše odgovara.



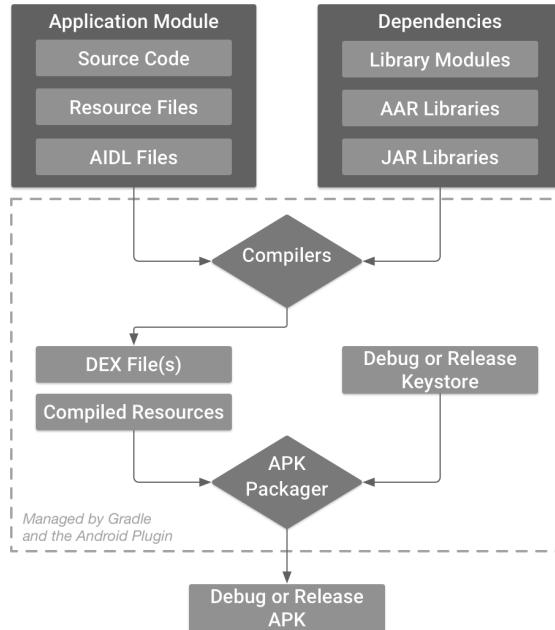
Slika 2.6, Android Studio IDE

Android studio kompajlira izvorni kod zajedno sa resursima aplikacije, pakuje ih u APK koji se može testirati, potpisati i distribuirati. Proces je prikazan na slici 2.7. Android Studio koristi Gradle za automatizaciju i vođenje ovog procesa. Sam proces pravljenja build-a aplikacije uključuje veći broj alata koji prevode projekat u APK. Tipičan proces uključuje sledeće opšte korake:

- Kompajleri su zaduženi da izvorni kod prevedu u DEX fajlove koji sadrže bajtkod koji se izvršava na Android uređaju, a sve ostalo u kompajlirane resurse.
- APK Packager komponenta spaja DEX fajlove i kompajlirane resurse u jedinstveni APK. Pre nego što se aplikacija instalira ili distribuira na Android uređaje, neophodno je da bude potpisana (engl. *signed*).
- APK Packager potpisuje APK upotrebom ili debug ili release ključa
 - Ukoliko je verzija aplikacije potrebna samo za testiranje, APK Packager potpisuje aplikaciju sa debug ključem (Android Studio automatski konfiguriše nove projekte sa debug ključem).
 - Ukoliko je verzija aplikacije namenjena eksternoj distribuciji (engl. *release version*), APK Packager potpisuje aplikaciju sa release ključem (koji se mora kreirati u okviru Android Studio okruženja).

- Pre generisanja finalne verzije APK, Packager dodatno optimizuje aplikaciju da koristi manje memorije prilikom izvršavanja na uređaju.

Na kraju ovog procesa, kao rezultat se dobija ili debug APK ili release APK koji se mogu koristiti za instalaciju, testiranje i distribuciju drugim korisnicima.



Slika 2.7, proces kreiranja aplikacije u Android Studio IDE
(izvor: <https://developer.android.com/studio/build>)

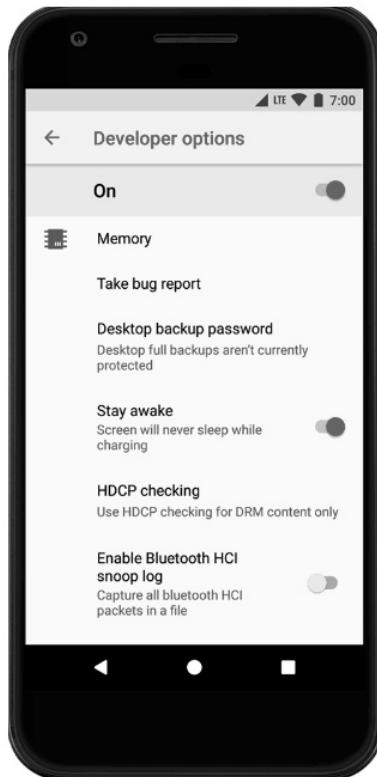
2.5. Android uređaji i emulatori

Razvoj mobilnih aplikacija za Android ima još jednu razliku u poređenju sa tradicionalnim Java aplikacijama. Kako bi se razvijena aplikacija mogla pokrenuti i testirati, potrebno je imati stvarni uređaj na kome se aplikacija biti instalirana. To je relativno jednostavno uraditi tako što se Android uređaj preko USB kabla poveže sa računarcem, a zatim se Android Studio preko ADB (Android Debug Bridge) poveže sa uređajem. ADB je alat koji omogućava komunikaciju sa uređajem i omogućava veliki broj akcija sa tim uređajem, poput instalacije i debagovanja aplikacija. Realizovan je kao klijent-server program, i sastoji se od sledećih komponenti:

- Klijent – izvršava se na računaru na kome se razvija aplikacija i šalje komande.

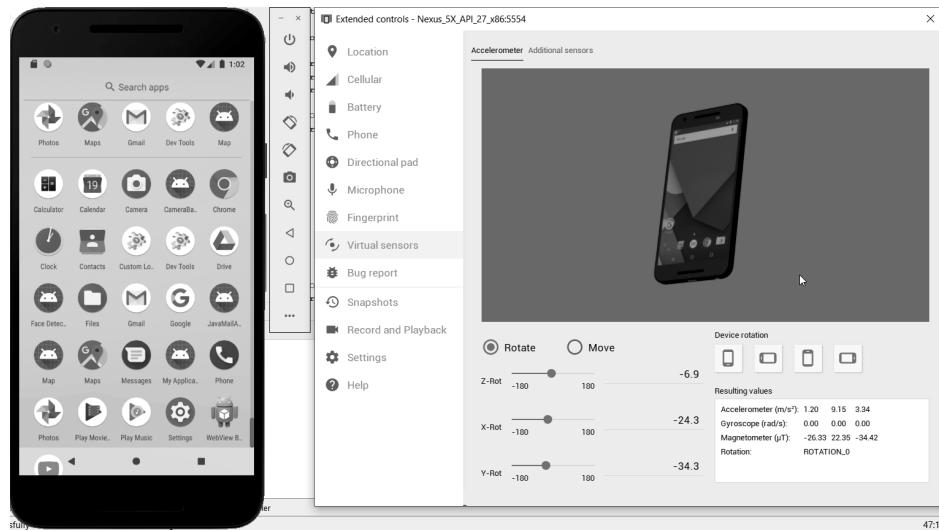
- Daemon (abdb) – izvršava se kao pozadinski proces na uređaju i izvršava komande na njemu.
- Server – služi za vođenje komunikacije između klijenta i daemon-a, izvršava se kao pozadinski proces na računaru na kome se razvija aplikacija.

Kako bi se na uređaju koji je povezan preko USB koristio ADB, potrebno je uključiti opciju USB debugging u okviru podešavanja uređaja, u skrivenoj sekciji Developer options (slika 2.8). Kako bi se ova sekcija aktivirala i postala vidljiva, na najvećem broju uređaja potrebno je otići na Settings – About phone, pronaći u okviru ove sekcije Build number i dodirnuti ga 7 puta. Prilikom sledećeg povratka u Settings sekciju, opcija Developer options će biti dostupna na dnu. Na nekim uređajima se ova opcija može nalaziti na drugom mestu i sa nešto drugaćijim imenom. Sada se prilikom povezivanja uređaja preko USB mogu izvršavati i ADB komande. Veliki broj Android programera se prilikom razvoja aplikacija odlučuje upravo za ovu varijantu, da debaseuje aplikaciju direktno na fizičkom uređaju.



Slika 2.8, skrivena sekcija Developer options
(izvor: <https://developer.android.com/studio/debug/dev-options>)

Međutim, fizički uređaj ponekada nije adekvatna opcija. Na primer, ukoliko ciljani uređaj nije dostupan iz bilo kog razloga (imamo uređaj sa Android 8.0, a ciljana platforma je Android 9.0), ili ukoliko želimo da proverimo aplikaciju na više različitih uređaja sa različitim API nivoima, pa nije praktično imati veliki broj fizičkih uređaja. U tom slučaju, moguće je koristiti Android Emulator, koji simulira Android uređaje na računaru. Emulator pruža skoro sve funkcionalnosti stvarnog Android uređaja. Moguće je simulirati pozive, tekstualne poruke, specificirati lokaciju uređaja (moguće je učitati rutu i simulirati kretanje), različite tipove mreže i jačinu signala, simulirati rotaciju i fizičke pokrete kako bi se testirali senzori i slično (slika 2.9). Debugovanje i testiranje aplikacije na emulatoru je iz više razloga brže i lakše nego u slučaju fizičkog uređaja – na primer, aplikacija se može instalirati i podaci se mogu prebaciti na emulator brže nego na uređaj povezan preko USB.



Slika 2.9, Android Emulator sa dodatnim funkcionalnostima

Emulator se isporučuje sa već predefinisanim konfiguracijama za različite Android telefone, tablete, satove i TV uređaje. Što se tiče telefona, ponuđeni su predefinisani Google telefoni tipa Nexus. Moguće je simulirati i uređaje drugih proizvođača podešavanjem odgovarajućih fizičkih karakteristika i upotreboom skinova. Potrebno je na kraju napomenuti da Android Emulator ima daleko veće hardverske zahteve od samog Android Studio okruženja. Konkretno, zahteva se 64-bitni procesor, minimum 8GB RAM (4GB RAM je u teoriji dovoljno da se pokrene Android Studio i Emulator, ali u praksi je neprihvatljivo sporo). Za hardversku akceleraciju dodatno je potrebno:

- Za Intel procesore: Intel procesor sa podrškom za Intel VT-x, Intel EM64T i Execute Disable (XD) Bit funkcionalnostima.

- Za AMD procesore: AMD procesor sa podrškom za AMD Virtuelizaciju (AMD-V).

Od juna 2019. godine počela je da se ukida podrška za 32-bitne Windows operativne sisteme. Osnovna podrška će trajati do kraja 2020. godine, uz ispravke kritičnih bagova, ali se nove funkcionalnosti neće dodavati. Zbog toga, preporuka je da se razvoj Android aplikacija i upotreba Android Emulatora vrši na uređaju sa 64-bitnim Windows okruženjem.

3. Android Studio

Android Studio je zvanični IDE za razvoj Android aplikacija, kao što je već pomenuto u prethodnom poglavlju. Okruženje se može preuzeti sa lokacije: <https://developer.android.com/studio/>. U trenutku pisanja ovog udžbenika, aktuelna verzija okruženja bila je 3.5.2, iz oktobra 2019. godine. Svi primjeri koda dati u nastavku ovog udžbenika implementirani su u ovoj verziji okruženja (slika 3.1).



Slika 3.1, verzija Android Studio okruženja korišćena u ovoj knjizi

U nastavku ovog poglavlja dato je detaljno uputstvo kako se instalira i konfiguriše okruženje, objašnjene su glavne komponente okruženja, opisano kako se konfiguriše emulator i dat je prvi jednostavan primer funkcionalne Android aplikacije.

3.1. Instalacija okruženja

Pre preuzimanja instalacionog fajla, potrebno je proveriti da li računar na kome se Android Studio instalira ispunjava hardverske zahteve. Za računar sa Windows operativnim sistemom zahtevi su sledeći:

- Operativni sistem Microsoft Windows 7/8/10 (preporučljivo 64-bitni).
- 4 GB RAM minimum, preporučljivo 8 GB RAM ili više (zbog emulatora).
- 2 GB dostupnog prostora na disku minimum (IDE + Android SDK).
- Minimalna rezolucija ekrana 1280 x 800.

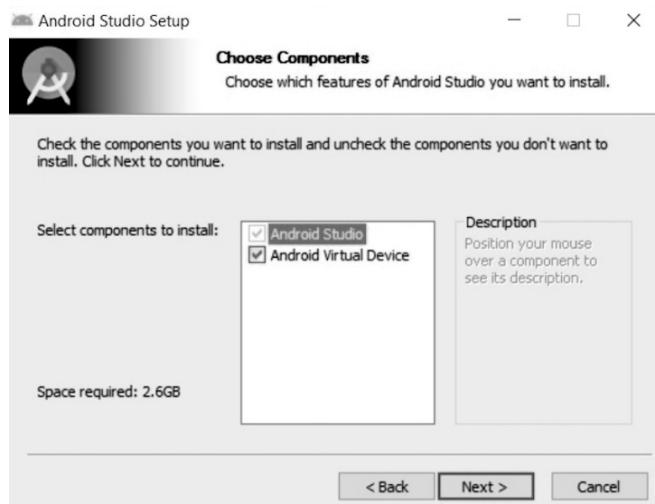
Prilikom instalacije okruženja na Windows računaru, nakon preuzimanja instalacionog fajla najnovije dostupne stabilne verzije sa adrese date na početku ovog poglavlja, potrebno je izvršiti dupli klik na .exe fajl. Sledeći koraci ilustruju proces instalacije koji je identičan za sve 3.x verzije Android Studio okruženja.

Instalacija započinje prikazom dijaloga dobrodošlice, gde se korisniku savetuje da isključi sve druge aktivne aplikacije za vreme procesa instalacije, kao što je prikazano na slici 3.2. Klikom na Next dugme pokreće se čarobnjak koji će voditi proces instalacije.



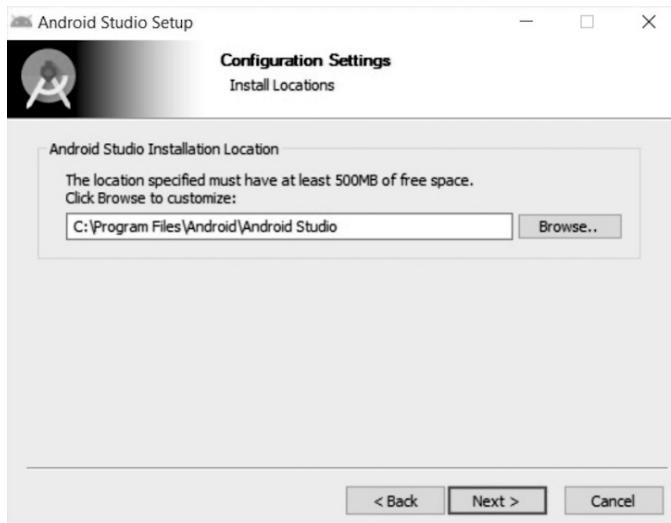
Slika 3.2, početni ekran procesa instalacije

Sledeći ekran nudi opciju za uključivanje/isključivanje komponenti koje će biti instalirane. Android Studio komponenta je obavezna, dok se opcionalno može isključiti Android Virtual Device. Preporučeno je ipak ostaviti podrazumevana podešavanja instalacije, kao što je prikazano na slici 3.3.

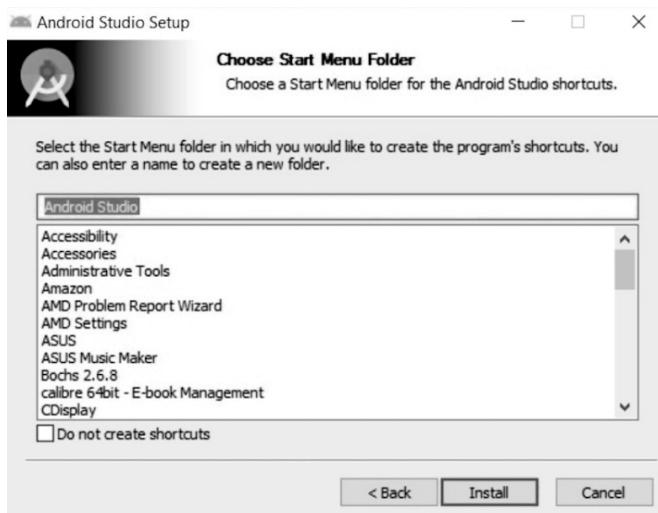


Slika 3.3, odabir komponenti za instalaciju

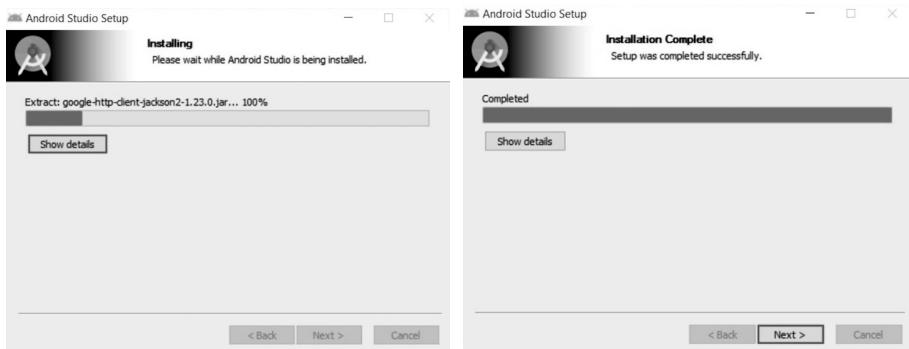
Sledeći panel, pod nazivom Configuration Settings, specificira lokaciju na kojoj će Android Studio biti instaliran (slika 3.4). Moguće je ostaviti podrazumevanu lokaciju, ili specificirati drugu. U svakom slučaju, specificirana lokacija mora imati bar 500 MB slobodnog prostora kako bi instalacija mogla biti izvršena. U sledećem meniju moguće je odabratи Start Menu direktorijum gde će biti smeštena prečica okruženja, kao što je prikazano na slici 3.5. Mogu se ostaviti podrazumevana podešavanja i kliknuti dugme Next, što započinje proces instalacije, kao što je prikazano na slici 3.6.



Slika 3.4, specifikacija lokacije gde će Android Studio biti instaliran



Slika 3.5, odabir Start Menu direktorijuma



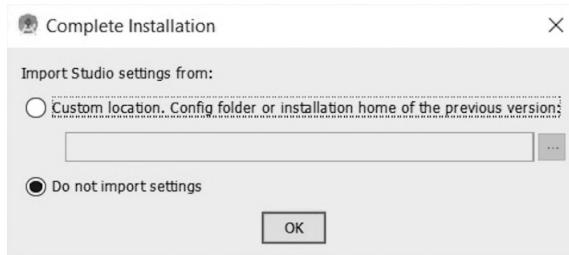
Slika 3.6, proces instalacije Android Studio okruženja

U toku instalacije moguće je kliknuti Show Details dugme, koje će prikazati detaljno koja se komponenta ili fajl trenutno instalira, kao i druge aktivnosti samog procesa instalacije. Kada se instalacija završi, pojaviće se panel Completing Android Studio Setup, kao što je prikazano na slici 3.7. Time se završava proces instalacije okruženja.



Slika 3.7, završetak instalacije Android Studio okruženja

Prilikom prvog pokretanja Android Studio okruženja, prikazaće se Complete Installation dijalog, koji nudi opciju importa podešavanja od prethodne instalacije koja je bila instalirana na tom računaru (ukoliko postoji), kao što je prikazano na slici 3.8. Ukoliko je ovo prva instalacija Android Studio okruženja na tom računaru, ili ukoliko se ne želi import podešavanja od prethodne verzije, bira se opcija Do not import settings. Nakon potvrde, počeće pokretanje okruženja, kao što je prikazano na slici 3.9.



Slika 3.8, import podešavanja od prethodne verzije (ukoliko postoji)



Slika 3.9, Android Studio splash screen

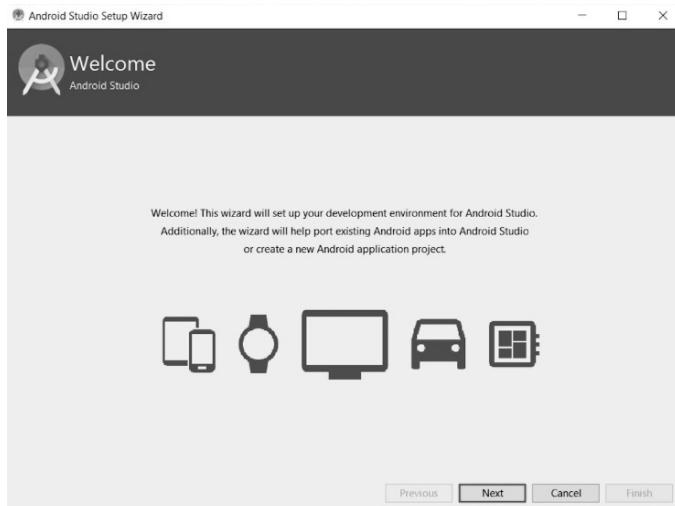
Prilikom prvog učitavanja okruženja, moguće je da će Android Studio u ovom trenutku početi da preuzima sa Interneta potrebne biblioteke, kao što je prikazano na slici 3.10. To su SDK komponente koje su dostupne i neophodne okruženju.



Slika 3.10, preuzimanje potrebnih SDK komponenti prilikom prvog pokretanja okruženja

Proces preuzimanja potrebnih SDK komponenti može da potraje nekoliko minuta u zavisnosti od brzine Internet konekcije, pa zbog toga prvo pokretanje okruženja traje malo duže u poređenju sa svakim sledećim pokretanjem. Kada su sve potrebne biblioteke preuzete, prikazaće se Android Studio ekran dobrodošlice, kao na slici 3.11. Ovaj ekran obaveštava korisnika da će čarobnjak sada podesiti okruženje za razvoj, i eventualno pomoći u portovanju već postojećih Android aplikacija (razvijanim u starijim verzijama okruženja) ili pokrenuti novi projekat sa novom Android aplikacijom. Na ekranu su grafički prikazane i sve različite

platforme koje su podržane od strane okruženja i za koje je moguće razvijati aplikacije (Android telefoni, satovi, TV i slično).

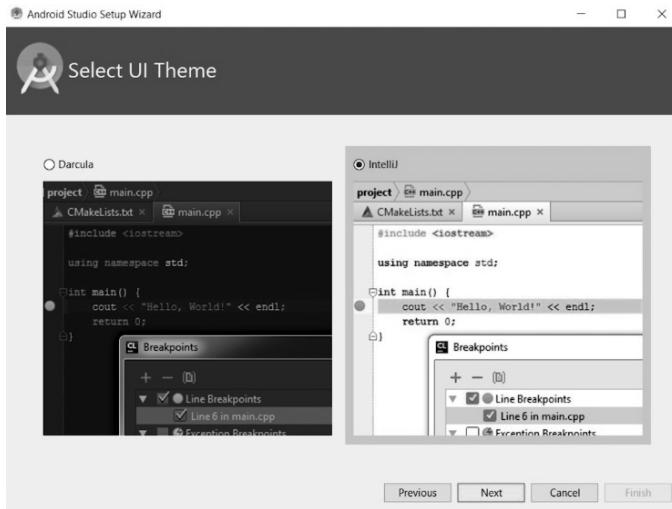


Slika 3.11, Android Studio čarobnjak za podešavanje okruženja

Sljedeća dva ekrana traže od korisnika da odabere tip instalacije (podrazumevana opcija u potpunosti odgovara), kao što je prikazano na slici 3.12, kao i opciju da se odabere tema korisničkog interfejsa okruženja, kao što je prikazano na slici 3.13. Ponuđene su dve teme, tamna (Darcula) i standardna svetla (IntelliJ), sa uzorkom prikaza obe teme. Odabir teme je u potpunosti individualan i ostavljen korisniku, pošto razlike u funkcionalnosti nema.

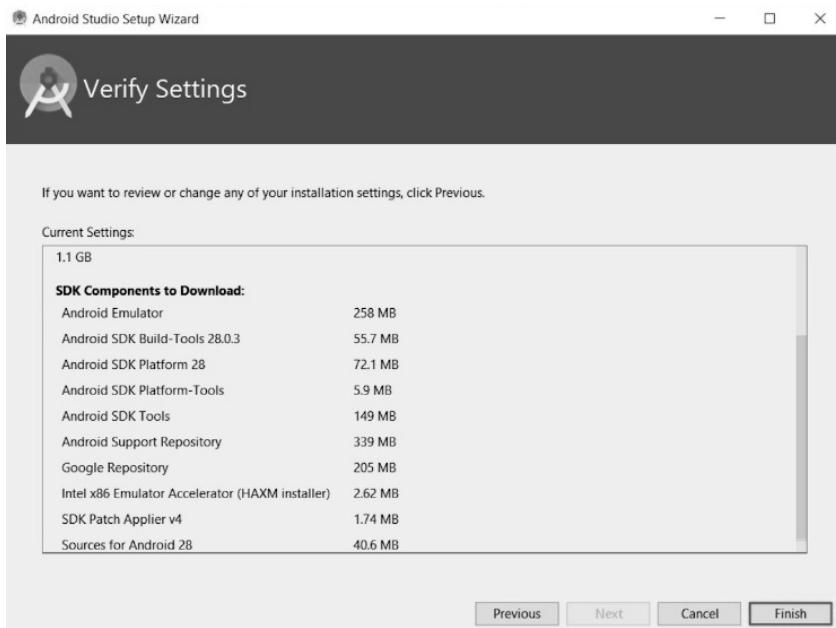


Slika 3.12, odabir tipa instalacije okruženja

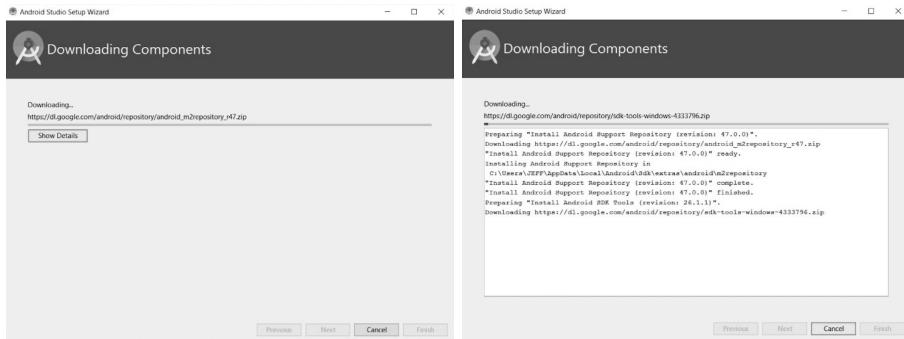


Slika 3.13, odabir teme korisničkog interfejsa okruženja

Nakon odabira tipa instalacije i teme, Android Studio nudi priliku da se verifikuju sva odabrana podešavanja i opcije, kao što je prikazano na slici 3.14. Na ovom ekraru su prikazane i sve dodatne komponente koje su identifikovane kao potrebne i koje će Android Studio početi da preuzima i instalira nakon potvrde opcija i klika na Finish dugme.

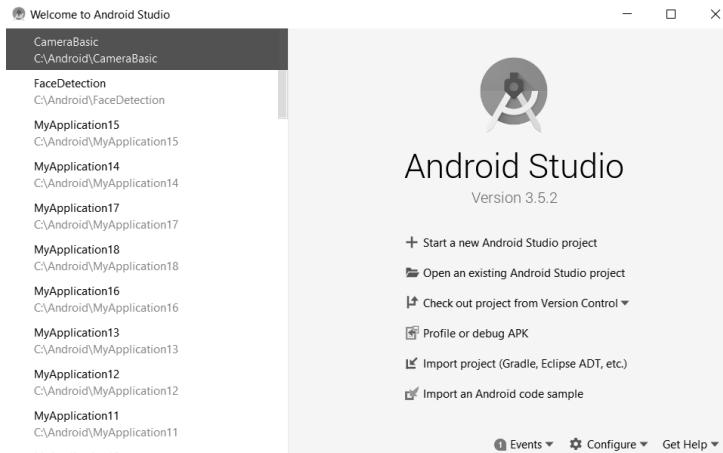


Slika 3.14, verifikacija odabralih opcija



Slika 3.15, čarobnjak preuzima i instalira SDK komponente, sa i bez prikaza detalja

Nakon verifikacije odabranih komponenti, čarobnjak počinje da preuzima, otpakuje i instalira sve neophodne SDK komponente. Ovaj proces može da potraje i desetak minuta, pa se dosada prilikom čekanja može delimično ublažiti klikom na Show Details. Na taj način korisnik može videti koji se svi različiti fajlovi trenutno preuzimaju, njihov proces otpakivanja, kao i lokaciju gde se instaliraju. Kada se konačno preuzmu sve neophodne komponente, prikazaće se ekran dobrodošlice kao na slici 3.16. Ovaj dijalog se može iskoristiti za pokretanje novog Android projekta, otvaranje već postojećeg projekta, kao i opcije za checkout projekta sa nekog alata za kontrolu verzije (npr. GIT), odnosno import projekta. Ukoliko ovo nije prvo pokretanje okruženja, prethodni projekti će biti izlistani sa leve strane ekrana, kao što je prikazano na slici 3.16.



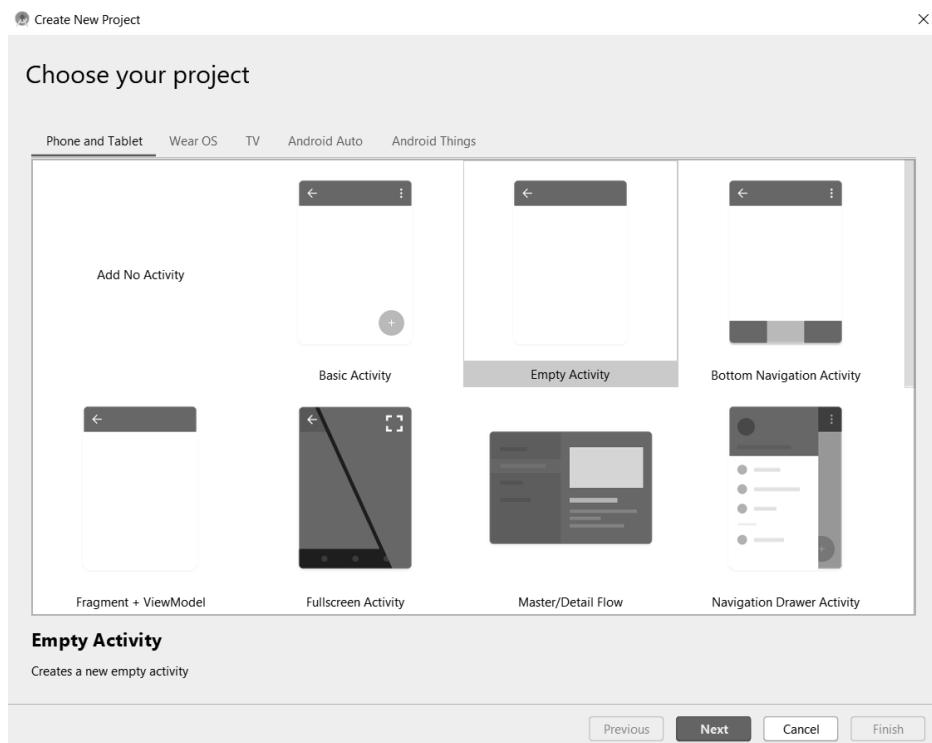
Slika 3.16, Welcome to Android Studio ekran

Strukturu i korisnički interfejs Android Studio okruženja je najlakše opisati kroz kreiranje prvog projekta, odnosno jednostavne Android aplikacije.

3.2. Kreiranje i struktura projekta

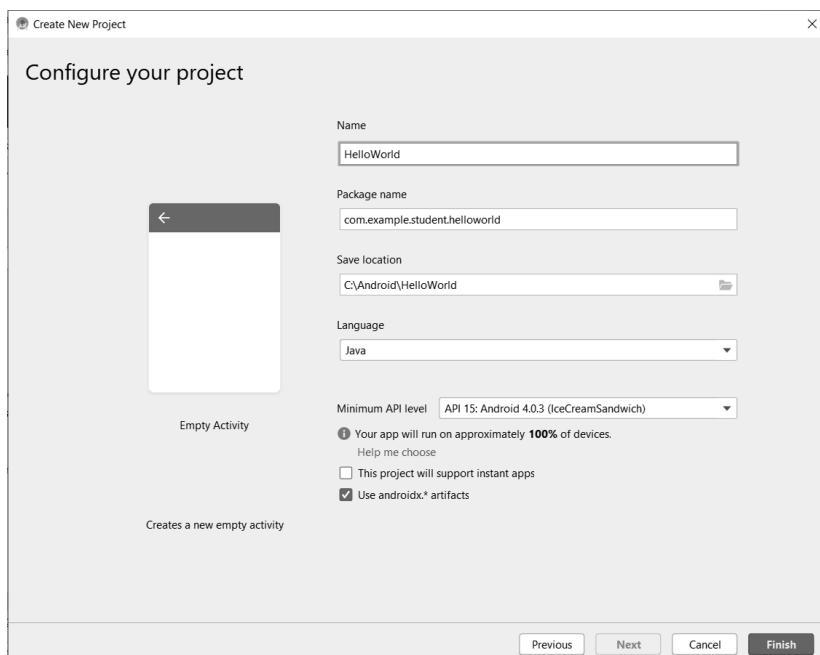
U ovom poglavlju opisan je proces kreiranja Android projekta, kroz koji ćemo objasniti osnovne alate koji se koriste u programiranju Android aplikacija. Prvi program koji se piše u bilo kom novom programskom jeziku ili okruženju obično je jednostavna „Hello World“ aplikacija, koju ćemo u ovom slučaju iskoristiti za opis korisničkog interfejsa okruženja. Nakon koraka opisanih u poglavlju 3.1, došli smo do ekrana sa slike 3.16 (Welcome to Android Studio ekran). Odabirom opcije **Start a new Android Studio project** započinje proces kreiranja novog projekta (ovaj proces, kao i redosled i sadržaj ekrana za konfiguraciju projekta se može malo razlikovati od verzije do verzije Android Studio okruženja). Proces je prikazan na verziji Android Studio 3.5.2.

Sledeći ekran koji se prikazuje nudi opcije za odabir platforme za koju se kreira aplikacija, kao i šablon na osnovu kojeg će biti kreirana početna aktivnost, kao što je prikazano na slici 3.17. Kao platforma u ovom primeru bira se telefon (prva opcija - **Phone and Tablet**), a kao šablon koristi se prazna aktivnost (**Empty Activity**).



Slika 3.17, odabir platforme i šablonu aktivnosti za novi projekat

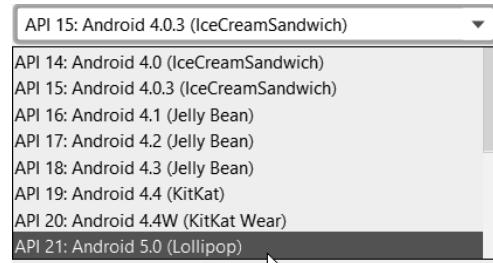
Nakon odabira platforme i šablona, otvara se forma za podešavanje projekta, kao što je prikazano na slici 3.18. Proces započinje unošenjem imena aplikacije u polje **Name**. Ime aplikacije će istovremeno služiti i kao ime samog projekta. Treba obratiti pažnju da se menjanjem imena aplikacije automatski menja i ime paketa koje će se koristiti u okviru projekta (polje sa imenom **Package name**). U slučaju da je potrebno drugačije ime paketa, može se naravno ručno izmeniti. Sledeće polje se odnosi na lokaciju na kojoj će se projekat čuvati (polje **Save location**), gde se može uneti željena putanja. U sledećem polju (polje sa imenom **Language**) bira se programski jezik u kome će se razvijati aplikacija. Dostupne opcije su Java i Kotlin. U okviru ovog udžbenika biće korišćena Java kao odabrani programski jezik.



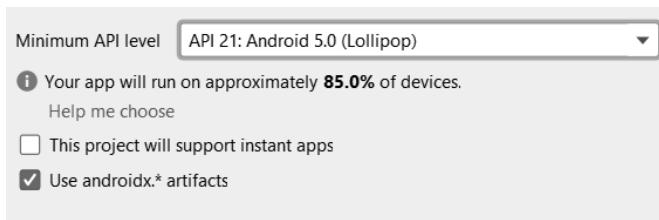
Slika 3.18, osnovno podešavanje projekta

Veoma bitna opcija je minimalni API nivo (dropdown lista **Minimum API level**). Ova vrednost označava minimalni API nivo na kojem će biti moguće instalirati i pokrenuti aplikaciju, pošto će uređaji sa nižim nivoom od zadatog minimalnog nivoa odbiti instalaciju aplikacije. Dodatno, Android Studio daje i statistički podatak koji opisuje koliko procenata mobilnih telefona na tržištu će moći da pokrene aplikaciju. Kao što se vidi sa slike 3.18, ukoliko se kao minimalni API nivo odabere Android 4.0.3 (Ice Cream Sandwich), broj podržanih uređaja je 100%, odnosno aplikacija će moći da se pokrene na praktično svim aktivnim Android uređajima. U slučaju odabira više verzije minimalnog API nivoa (slika 3.19), manji procenat uređaja će moći da pokrene

aplikaciju (slika 3.20). U ovom primeru koristićemo Android 5.0 Lollipop kao minimalni API nivo, odnosno aplikacija će moći da se pokrene na približno 85% aktivnih uređaja. Treba znati da što je viša vrednost minimalnog API nivoa, procenat uređaja drastično opada, tako da u slučaju odabira Android 9.0 (Pie) ovaj procenat je manji od 1%.

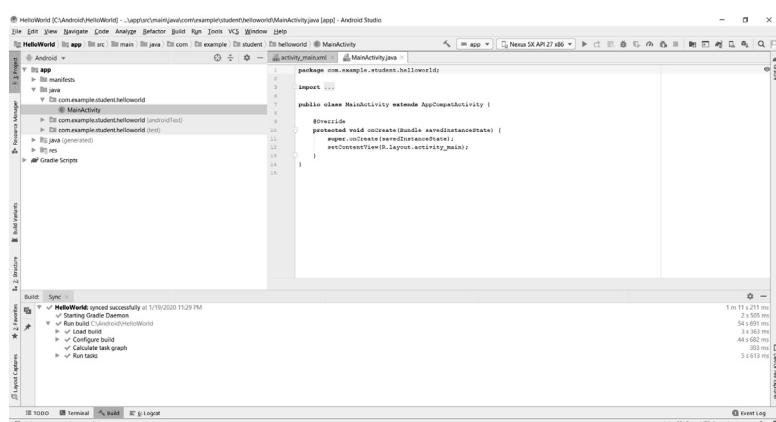


Slika 3.19, odabir druge vrednosti minimalnog podržanog API nivoa



Slika 3.20, procenat uređaja koji će moći da pokrenu aplikaciju na osnovu zadatog minimalnog API nivoa

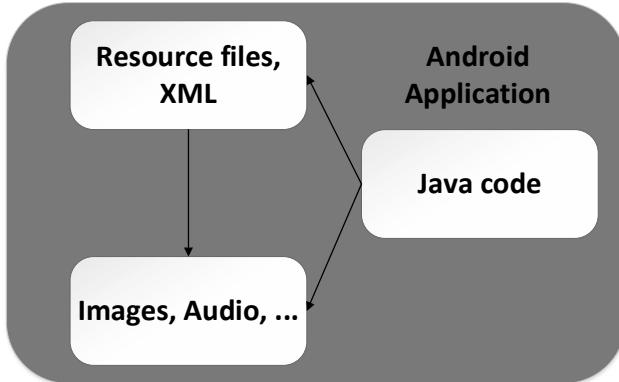
Nakon potvrde izabranih opcija klikom na dugme **Finish**, otvara se okruženje sa projektom pripremljenim prema prethodno odabranoj konfiguraciji, kao na slici 3.21.



Slika 3.21, kreirani HelloWorld projekat

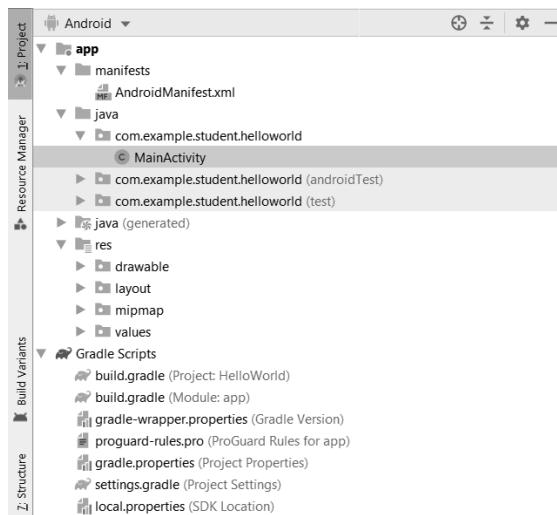
Treba još napomenuti da, ukoliko SDK za odabranu verziju Androida nije instaliran, Android studio može u tom slučaju početi da skida potrebne biblioteke.

Glavni princip razvoja mobilnih Android aplikacija leži u činjenici da se kod jasno razdvaja od resursa. Zbog toga se svaka Android aplikacija sastoji iz programskog koda i resursa, u koje spadaju XML fajlovi kao i drugi dodatni fajlovi poput slika, audio i video materijala i slično (slika 3.22).



Slika 3.22, stруктура standardne Android aplikacije

Na sličan način, svaki projekat u Android Studio okruženju sadrži jedan ili više modula koji sadrže izvorni kod i resursne fajlove. Podrazumevano, Android Studio prikazuje sve projektne fajlove u Android pogledu projekta, kao što je prikazano na slici 3.23. Ovaj pogled je dodatno organizovan po modulima, kako bi se omogućio brz pristup ključnim projektnim fajlovima.



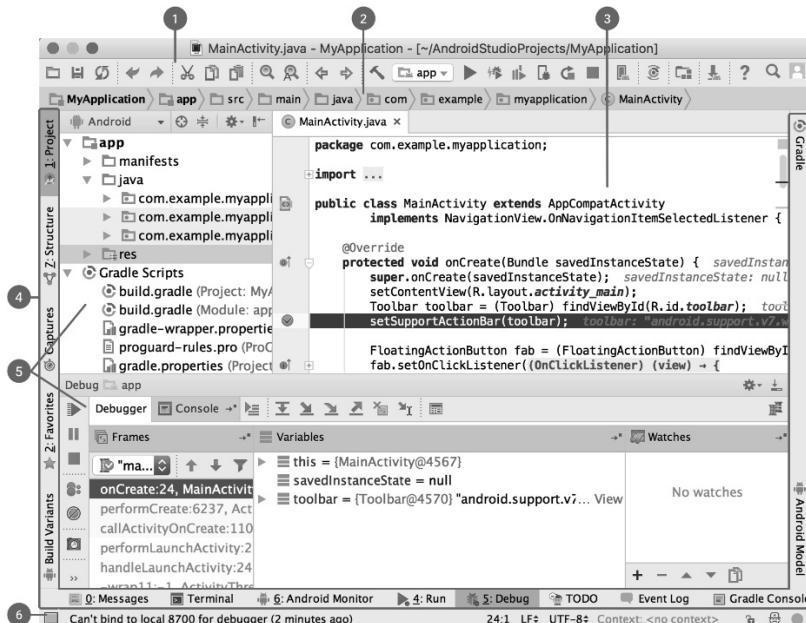
Slika 3.23, projektni fajlovi u Android pogledu

Aplikacioni modul (**app**) sadrži sledeće foldere:

- **manifests** – sadrži AndroidManifest.xml fajl.
- **java** – sadrži izvorne fajlove sa Java kodom, uključujući i JUnit testove.
- **res** – sadrži sve što nije kod, poput XML rasporeda komponenti (**layout**), korisničkih stringova, dimenzija, boja, tema i bitmap slika.

Modul **Gradle Scripts** sadrži gradle fajlove (konfiguracione fajlove Android projekta). Njih nije potrebno ručno pisati, pošto se automatski generišu pri kreiranju projekta, mada se po potrebi njihov sadržaj može menjati u toku razvoja same aplikacije. Gradle fajlovi sadrže podatke poput minimalne verzije API nivoa, ID aplikacije ili opisa eksternih biblioteka koje su potrebne aplikaciji. Gradle se može koristiti i van Android Studija, za potrebe drugih projekata koji su pisani drugaćijim programskim jezicima (poput Maven-a), odnosno nije isključivo alat koji se koristi samo u okviru Android Studio okruženja. Ukoliko je čitalac upoznat sa Maven alatom, videće da postoje velike sličnosti između dva alata.

Sam glavni prozor Android Studio okruženja se sastoji od više logičkih celina, označenih na slici 3.24 (slika preuzeta sa Google – developer.android web sajta). Programer može organizovati prikaz glavnog ekrana kako bi ga dodatno prilagodio sebi (u okviru ovog udžbenika mi ćemo koristiti podrazumevani prikaz glavnog ekrana).



Slika 3.24, struktura glavnog prozora Android Studio okruženja
(izvor: <https://developer.android.com/studio/intro/>)

Svaka od označenih celina ima svoju svrhu:

1. **Linija sa alatima (toolbar)** sadrži veliki opseg različitih akcija koje se mogu izvršiti, uključujući pokretanje izvršavanja aplikacije i pokretanje različitih Android alata.
2. **Navigaciona linija (navigation bar)** služi za navigaciju kroz projekat, kao i otvaranje fajlova za editovanje. Prikazuje apsolutnu putanju do fajla koji se nalazi fokusiran u editoru. Pruža mnogo kompaktniji pogled od strukture koja se vidi pod **Project** prozorom.
3. **Editor** je celina gde se piše i modifikuje kod. U zavisnosti od konkretnog tipa fajla koji je trenutno otvoren, sam editor se može promeniti. Na primer, kada se posmatra XML fajl razmeštaja komponenti, editor će prikazati grafički pregled trenutnog sadržaja XML fajla.
4. **Linija sa prozorima alata (tool window bar)** se nalazi po obodu ekrana okruženja, sadrži prečice pomoću kojih se može prikazati ili sakriti pojedinačni prozor sa konkretnim alatom.
5. **Prozor sa alatima (tool window)** daje pristup debageru, kao i posebnim zadacima poput pretrage, kontrole verzije i slično. Svaki od ovih alata pojedinačno se može prikazati ili sakriti.
6. **Statusna linija (status bar)** prikazuje status projekta i samog okruženja, uz eventualna upozorenja ili poruke.

Pogledajmo sada detaljnije sadržaj koji je Android Studio generisao za potrebe našeg projekta. U editoru možemo videti dva taba, jedan pod nazivom `MainActivity.java`, a drugi pod nazivom `activity_main.xml`. `MainActivity.java` je Java klasa koja predstavlja ulaznu tačku u našu aplikaciju. U pitanju je aktivnost, koja predstavlja jednu od četiri osnovne Android komponente. Aktivnost predstavlja osnovni blok korisničkog interfejsa Android aplikacije, analogna je jednoj stranici tradicionalne web aplikacije. Aplikacija obično ima više od jedne aktivnosti, a korisnik aplikacije se kreće kroz njih na sličan način kao kada se koristi web aplikacija sa više strana.

Prilikom programiranja Android aplikacija, potrebne klase se izvode iz predefinisanih Android baznih klasa, što se može uočiti i na našem primeru, jer je `MainActivity` izvedena iz Androidove klase `AppCompatActivity`. Automatski generisani sadržaj ove klase dat je sa:

```
package com.example.student.helloworld;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}

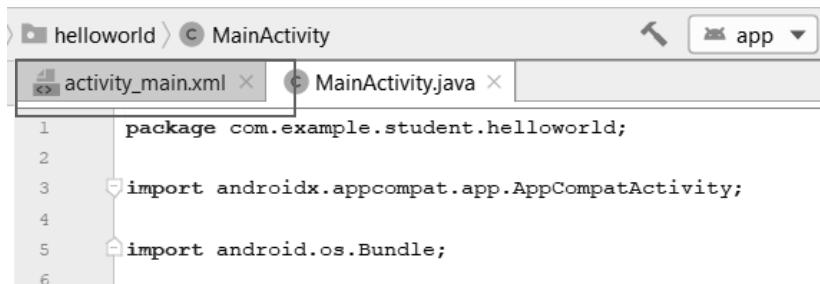
```

Može se uočiti da postoji samo jedna metoda, onCreate(), koja predstavlja nadjačanu metodu (anotacija @Override) iz bazne klase AppCompatActivity. Ono što je potrebno razumeti za sada jeste da ova metoda, pozivom metode setContentView() postavlja na ekran sadržaj aktivnosti koji je definisan u XML fajlu prosleđenom kao parametar ove metode – R.layout.activity_main.

Svaka aktivnost, osim Java koda, treba da ima i definiciju vizuelnog izgleda korisničkog interfejsa koji odgovara toj aktivnosti. Kao što smo ranije naveli, resursi (među koje spada i XML fajl sa rasporedom komponenti na ekrani – layout fajl) su potpuno odvojeni od Java koda, i nalaze se u folderu res/layout. Poziv metode setContentView() zapravo povezuje aktivnost i njen XML fajl sa rasporedom komponenti, u kome su definisane:

- sve komponente koje su potrebne na ekranu.
- atributi koji definišu vizuelni prikaz tih komponenti.
- raspored komponenti na ekranu.

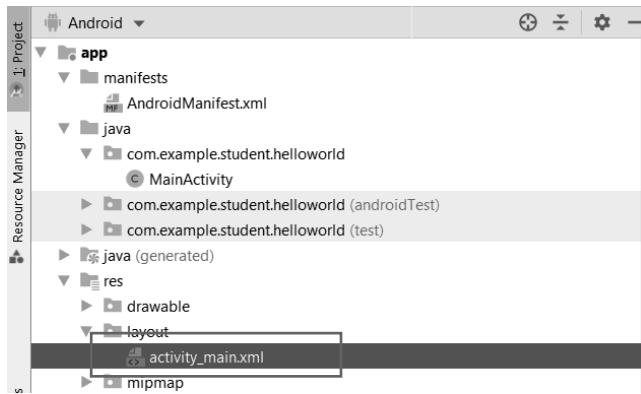
Jednoj Activity klasi se pridružuje jedan fajl sa rasporedom komponenti. Po konvenciji, ime XML fajla sa rasporedom treba da odgovara imenu same Activity klase na sledeći način – ako se aktivnost zove MainActivity, njen raspored se zove activity_main.xml. Na isti način je i okruženje automatski imenovalo naš raspored komponenti u primeru. Inače, konvencija imenovanja koja važi u programskom jeziku Java važi i u slučaju Android programiranja. Imena klasa se drže standardne Camel Case notacije (prvo slovo svake reči u imenu veliko, ostala slova mala), što se i vidi u slučaju imena Activity klase – MainActivity. Na prikaz XML fajla se dolazi klikom na tab u editoru sa imenom activity_main.xml, kao što je prikazano na slici 3.25.



Slika 3.25, tab sa XML rasporedom komponenti u editoru

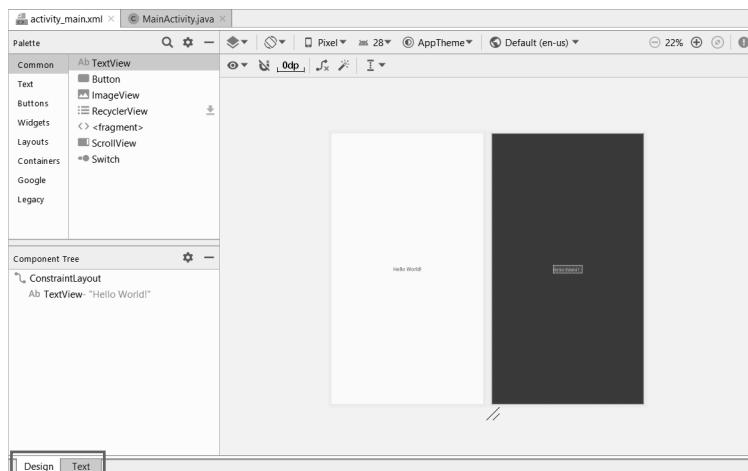
Ukoliko XML fajl nije učitan u editoru, može mu se pristupiti i kroz strukturu projekta. Kao što je navedeno ranije, XML fajlovi sa rasporedima se nalaze u

res/layout folderu, gde možemo pronaći i traženi raspored, kao što je prikazano na slici 3.26. Pošto jedna aplikacija može imati više aktivnosti, u ovom folderu se u opštem slučaju može nalaziti više fajlova sa rasporedima komponenti.



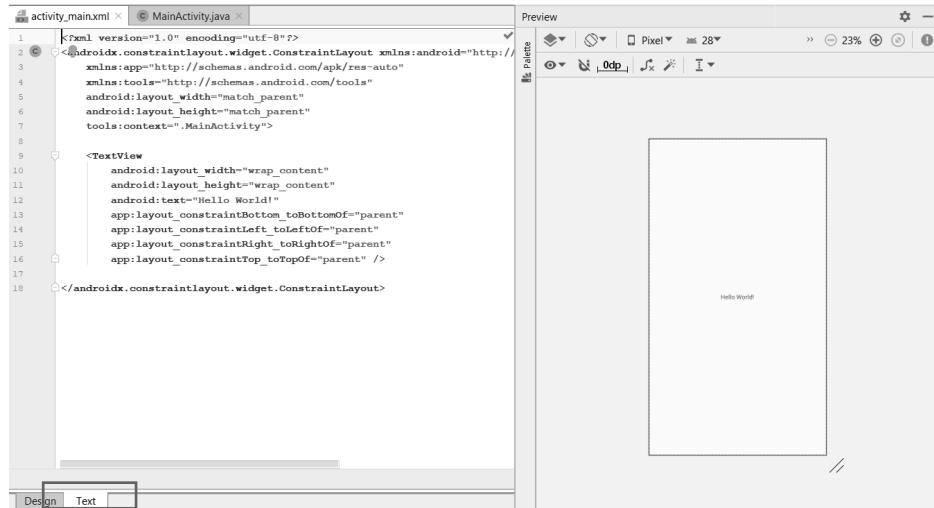
Slika 3.26, lokacija XML fajlova sa rasporedom u strukturi projekta (res/layout)

Nakon otvaranja fajla sa rasporedom komponenti, podrazumevano će se otvoriti ekran za grafičkim editorom za raspored komponenti. Ovaj editor omogućava veoma jednostavno kreiranje grafičkog interfejsa, uz podršku za drag & drop komponenti grafičkog interfejsa iz palete dostupnih komponenti. Svaka promena se trenutno prikazuje i vizuelno, kao što je prikazano na slici 3.27. U našem slučaju, automatski je generisan ekran sa jednom tekstualnom labelom (tip **TextView**), čiji je sadržaj string „Hello World!“. U narednim poglavljima ovog udžbenika biće prikazano kako se postavljaju nove komponente u raspored i kako se definišu njihovi vizuelni atributi. U ovom primeru, ostavićemo automatski generisani sadržaj.



Slika 3.27, grafički editor za XML fajlove sa rasporedom komponenti

Na slici 3.27, crvenom bojom su označena dva taba na dnu editora. Prvi tab, pod imenom **Design**, je podrazumevano selektovan, i omogućava vizuelni prikaz XML fajla. Drugi tab, sa imenom **Text**, omogućava pregled sadržaja samog XML fajla. Odabirom ovog taba, dobija se XML kod koji odgovara grafičkom prikazu iz grafičkog editora, kao što je prikazano na slici 3.28.



Slika 3.28, pregled sadržaja XML koda raspoređa komponenti

Izdvojen sadržaj generisanog fajla je dat u nastavku:

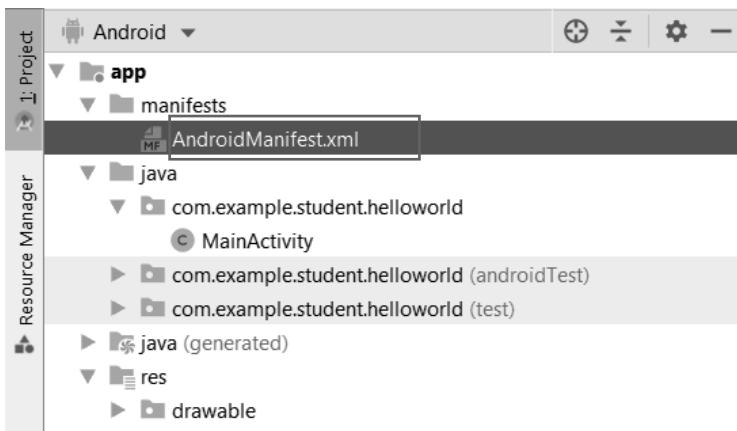
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"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

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

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

Iz datog koda XML fajla vidimo na koji način je definisano polje tipa TextView koje sadrži tekst „Hello World!“. Ovaj fajl definiše sadržaj ekrana naše aktivnosti MainActivity, i on će biti prikazan kada se pokrene aplikacija i kada se pokrene onCreate metoda naše aktivnosti.

Pogledajmo sada i sadržaj ostalih bitnih fajlova aplikacije. Kao što smo naveli, AndroidManifest.xml je jedan od najznačajnijih fajlova aplikacije, u kome se nalazi opis ključnih informacija o aplikaciji koje su potrebne Android build alatima i samom Android operativnom sistemu. To su informacije poput imena paketa, svih komponenti aplikacije (aktivnosti, servisi, itd.), kao i eventualnih dozvola koje su potrebne aplikaciji za rad. Fajl se može pronaći u folderu manifests, kao što je prikazano na slici 3.29.



Slika 3.29, lokacija AndroidManifest.xml fajla u strukturi projekta

Primer sadržaja AndroidManifest.xml fajla dat je u nastavku:

```
<?xml version="1.0" encoding="utf-8" ?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.student.helloworld">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Pogledajmo još za kraj šta se nalazi u build.gradle fajlu (Module:app). Njemu se može pristupiti takođe iz strukture samog projekta, kao što je prikazano na slici 3.30. Ovaj fajl se nalazi u okviru foldera Gradle Scripts, zajedno sa drugim skriptama koje su neophodne za uspešno build-ovanje aplikacije.



Slika 3.30, build.gradle fajl za modul app i njegova lokacija u strukturi projekta

Otvaranje ovog fajla će u editoru prikazati sadržaj same skripte, kao što je prikazano na slici 3.31. Ovde je moguće videti (i izmeniti) targetSdkVersion, koja je u ovom slučaju postavljena na vrednost 28. To znači da se kompajliranje aplikacije koristi verzija biblioteka Androida koja se nalazi u API nivo 28. Ovde je definisana i minSdkVersion, koja označava minimalnu verziju API nivoa na kojoj će biti moguće pokrenuti aplikaciju, i ona je u ovom projektu postavljena na API nivo 21 (Lollipop). U ovom fajlu je moguće videti i ID aplikacije, kao i njenu verziju. U sekciji **dependencies** mogu se videti sve eksterne biblioteke koje su podrazumevano uključene (naravno, moguće je uključiti i dodatne biblioteke po potrebi, na sličan način kao što je prikazano na slici). U našem primeru, ostavićemo podrazumevana podešavanja koja su automatski generisana prilikom kreiranja projekta.

```

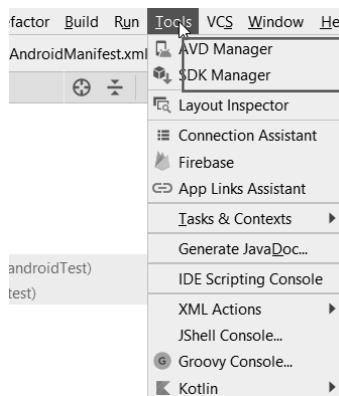
1  apply plugin: 'com.android.application'
2
3  android {
4      compileSdkVersion 28
5      defaultConfig {
6          applicationId "com.example.student.helloworld"
7          minSdkVersion 21
8          targetSdkVersion 28
9          versionCode 1
10         versionName "1.0"
11         testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
12     }
13     buildTypes {
14         release {
15             minifyEnabled false
16             proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
17         }
18     }
19 }
20
21 dependencies {
22     implementation fileTree(dir: 'libs', include: ['*.jar'])
23     implementation 'androidx.appcompat:appcompat:1.0.2'
24     implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
25     testImplementation 'junit:junit:4.12'
26     androidTestImplementation 'androidx.test.ext:junit:1.1.0'
27     androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.1'
28 }
29

```

Slika 3.31, sadržaj build.gradle fajla za modul app

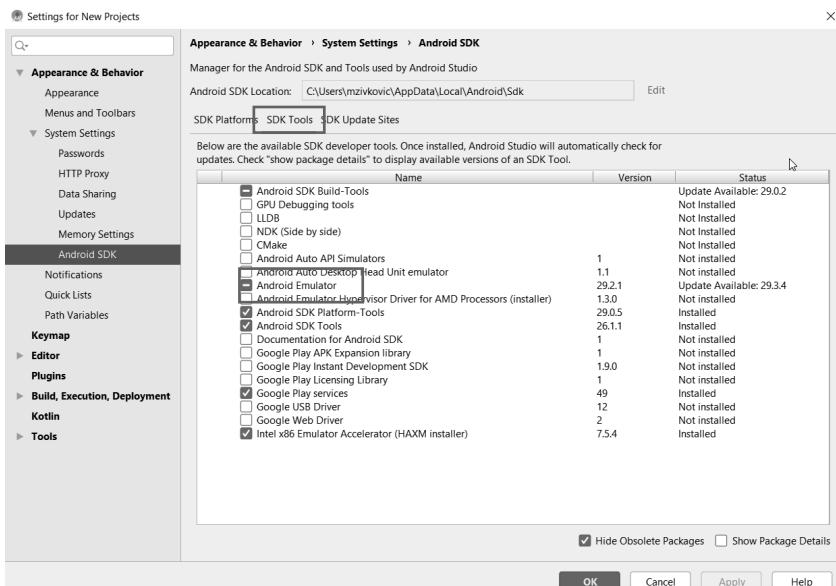
3.3. Konfigurisanje Emulatora i pokretanje aplikacije

Aplikacija kreirana u prethodnom poglavlju je u potpunosti funkcionalna. Kako bismo je pokrenuli, neophodno je da konfigurišemo **Android Emulator**. Već smo opisali da Emulator nudi skoro sve funkcionalnosti pravog, fizičkog uređaja. Moguće je simulirati veliki broj korisničkih scenarija, poput dolaznih poziva, lokacije uređaja, različitih performansi mobilne mreže, različitog nivoa baterije i slično. Postoji nekoliko stvari koje nije moguće simulirati putem virtuelnog uređaja, poput Bluetooth, NFC, slušalica prikačenih za uređaj i USB konekcije.



Slika 3.32, sekcija Tools, gde se nalaze AVD i SDK Manager opcije

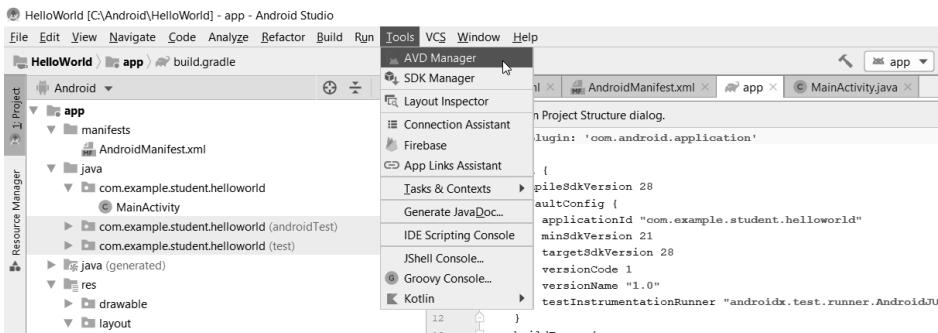
Za instalaciju Android Emulatora, potrebno je odabratи sekciju **Tools**, kao što je prikazano na slici 3.32, zatim **SDK Manager** opciju, a nakon toga u sledećem ekranu odabratи tab **SDK Tools**, kao što je prikazano na slici 3.33.



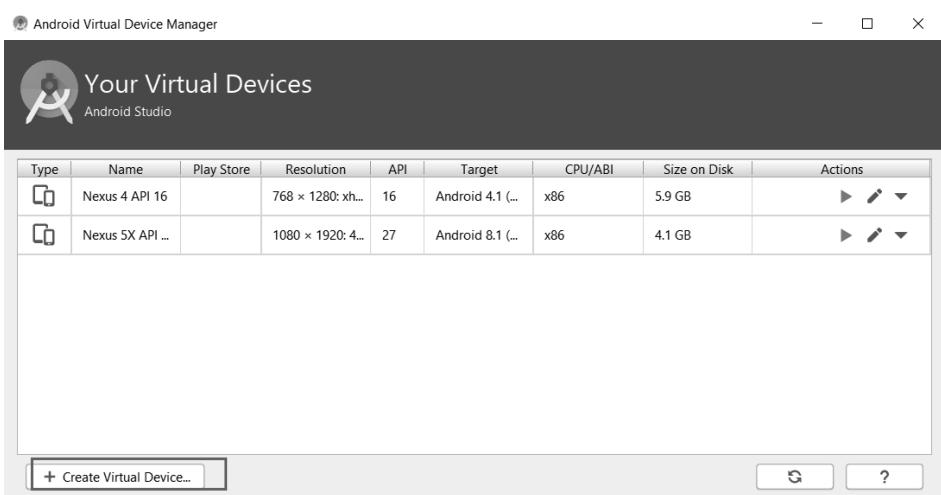
Slika 3.33, SDK Tools tab u okviru SDK Manager sekcije

Za računare sa Intel procesorom, preporučljivo je instalirati i opciju **Intel HAXM** (Intel Hardware Accelerated Execution Manager) selektovanjem poslednje opcije u listi sa slike 3.33. Na ovaj način se putem funkcije hardverskog ubrzavanja drastično mogu poboljšati performanse Emulatora kroz ubrzanje virtuelne mašine. Kako bi virtuelizacija radila, opcija virtuelizacije mora biti aktivirana kroz BIOS računara.

Svaka instanca Android Emulatora koristi Android virtuelni uređaj (engl. Android Virtual Device) kako bi specificirala verziju Androida i hardversku konfiguraciju simuliranog uređaja. Nakon instalacije Emulatora, moguće je dodati različite Android virtuelne uređaje (AVD) kroz **AVD Manager**, koji se aktivira kroz sekciju Tools, kao što je prikazano na slici 3.34. Nakon ulaska u **AVD Manager**, prikazuju se svi raspoloživi virtuelni uređaji (sa opcijama pokretanja virtuelnog uređaja, modifikacije i slično), kao i opcija za dodavanje novog virtuelnog uređaja, kao što je prikazano na slici 3.35. U praksi, preporučljivo je dodati AVD za svaku konfiguraciju koju aplikacija treba da podrži, na osnovu podešavanja `<uses-sdk>` u okviru `AndroidManifest.xml` fajla.



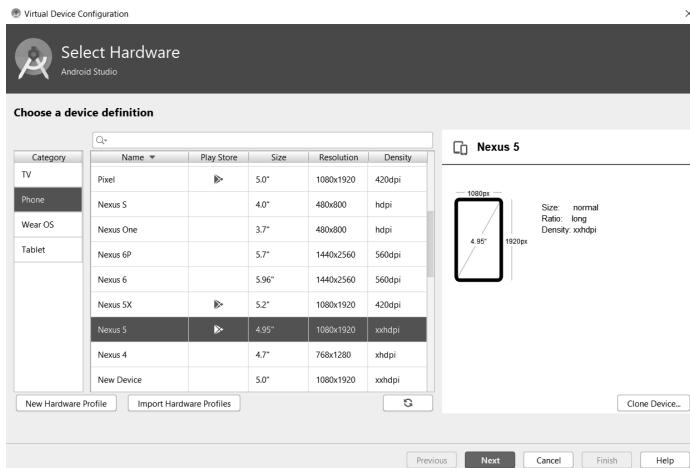
Slika 3.34, AVD Manager u okviru sekcije Tools



Slika 3.35, Raspoloživi virtuelni uređaji u okviru opcije AVD Manager

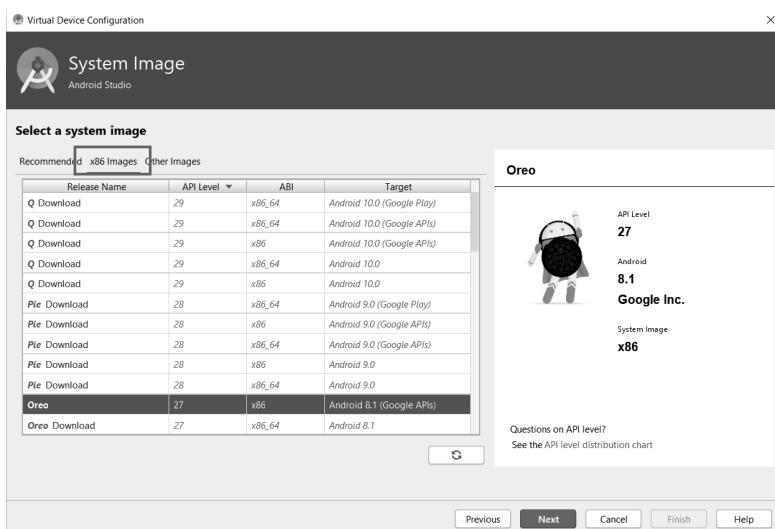
Svaki AVD sadrži hardverski profil, sistemska slika (engl. *system image*), memoriju za smeštanje podataka i druga svojstva. Hardverski profil definiše skup karakteristika fizičkog uređaja. **AVD Manager** dolazi sa predefinisanim hardverskim profilima (poput Pixel i Nexus uređaja). Prilikom kreiranja novog virtuelnog uređaja, prvo je potrebno odabrati hardverski profil, kao što je prikazano na slici 3.36. Treba obratiti pažnju da su samo neki uređaji označeni da uključuju **Play Store**. To su tzv. CTS komercijalno usaglašeni uređaji (Compatibility Test Suite – CTS), koji mogu koristiti odgovarajuće sistemske verzije i imaju oznaku u koloni Play Store. U ovom primeru kreiraćemo novi Nexus 5 uređaj.

Kada je odabran hardverski profil, prelazi se na sledeći ekran gde je potrebno odabrati sistemsku sliku. U slučaju da se koristi Intel hardverska akceleracija, potrebno je odabrati **x86 Images** tab, kako bi se doble što bolje performanse.



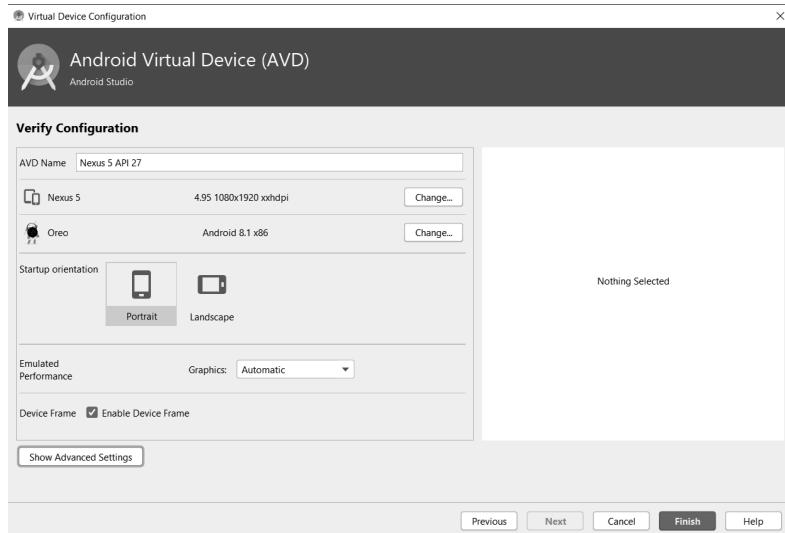
Slika 3.36, odabir hardverskog profila novog virtuelnog uređaja

Neke verzije slika imaju dodato **Google APIs** u nazivu, što označava mogućnost pristupima Google Play servisima. Odabir sistemske slike prikazan je na slici 3.37. Za potrebe ovog primera odabraćemo Android Oreo 8.1 API nivo 27 sa opcijom Google APIs. Međutim, za potrebe vežbanja, naročito na slabijim računarima (manje od 8 GB RAM), preporučljivo je uzeti neki slabiji hardverski profil (npr. Nexus 4) sa nešto starijom verzijom Androida (na primer Jelly Bean 4.1 API nivo 16), zbog znatno bržeg i komfornijeg rada i boljih performansi emulatora. Ukoliko pored oznake Android verzije piše **Download**, potrebno je prvo preuzeti sistemsku sliku pre nastavka dalje.

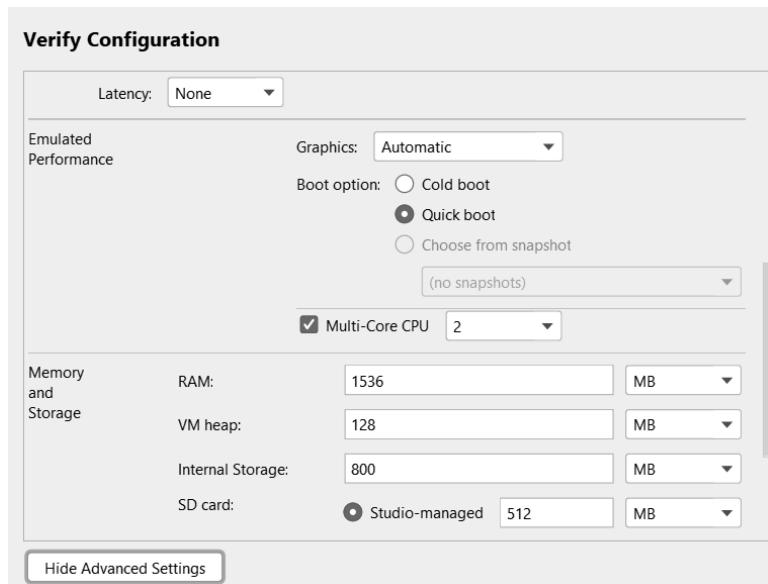


Slika 3.37, odabir sistemske slike virtuelnog uređaja

Potvrdom se prelazi na potvrdu odabrane konfiguracije. Moguće je dati posebno ime virtuelnoj mašini, odabrati orijentaciju uređaja nakon pokretanja emulatora, kao i prikazati napredna podešavanja (slika 3.39), gde je moguće odabrati skin, postaviti količinu memorije koju će uređaj koristiti, podesiti prednju i zadnju kameru i slično.

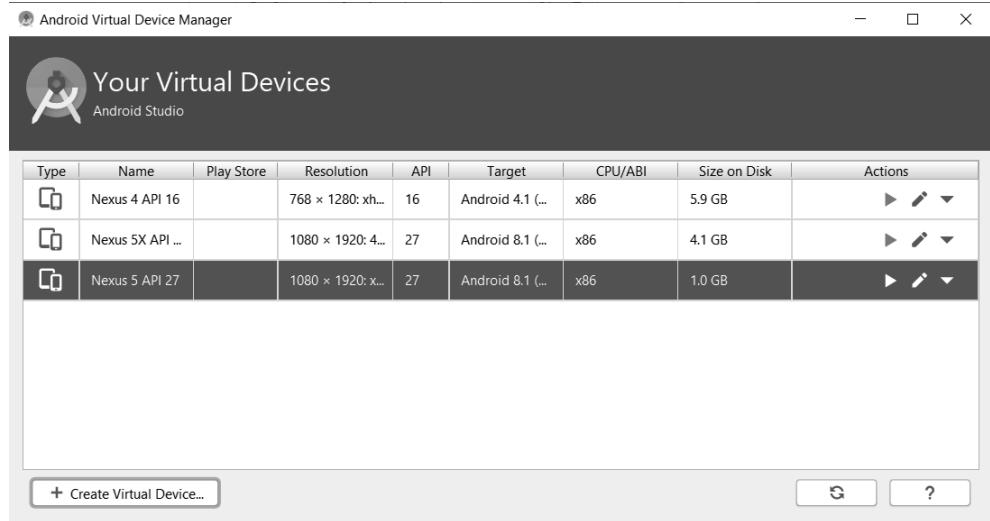


Slika 3.38, verifikacija odabrane konfiguracije



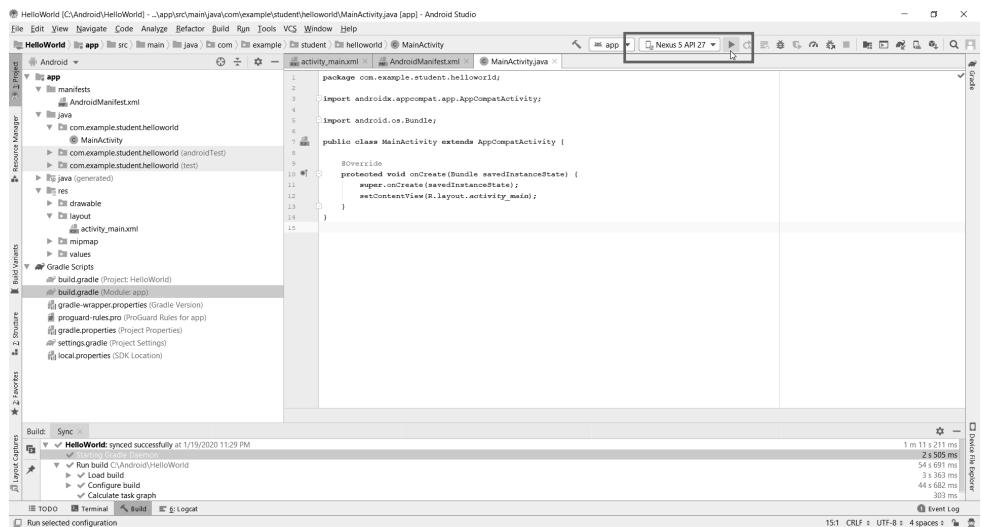
Slika 3.39, napredna podešavanja virtuelnog uređaja

Klikom na dugme **Finish** na slici 3.38 kreira se željeni virtualni uređaj. Odmah nakon kreiranja, uređaj je vidljiv u **AVD Manager** ekranu, kao što je prikazano na slici 3.40.



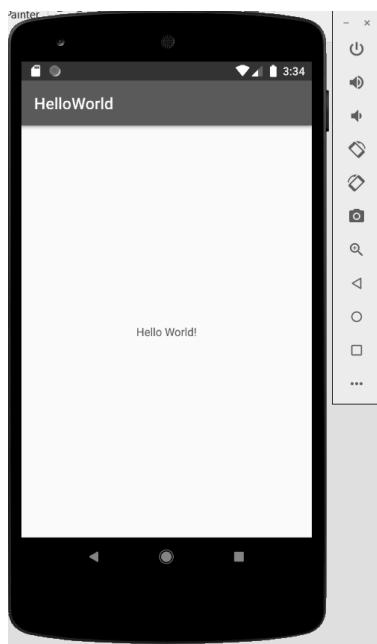
Slika 3.40, novi uređaj prikazan u AVD Manager ekranu

Kada je uređaj dodat, moguće je pokrenuti izvršavanje aplikacije na njemu klikom na dugme **Run app** (zelena strelica), kao što je prikazano na slici 3.41.



Slika 3.41, pokretanje aplikacije na dodatom virtuelnom uređaju

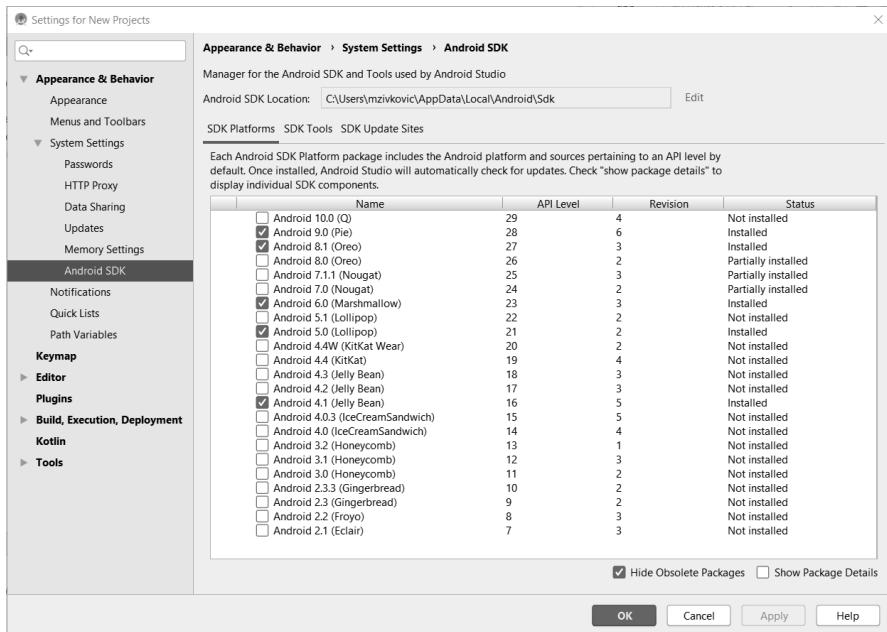
U zavisnosti od kompleksnosti virtuelnog uređaja, kao i od jačine računara na kojem se pokreće aplikacija, proces build-a aplikacije i pokretanja emulatora može da potraje nekoliko minuta. Zbog toga se preporučuje da kada se jednom podigne emulator, ne gasi dok se ne završi sa radom na aplikaciji, pošto će u tom slučaju svako sledeće pokretanje aplikacije ići znatno brže (pošto je emulator već podignut). „Hello World!“ aplikacija iz primera, pokrenuta na konfigurisanim virtuelnim uređaju, prikazana je na slici 3.42.



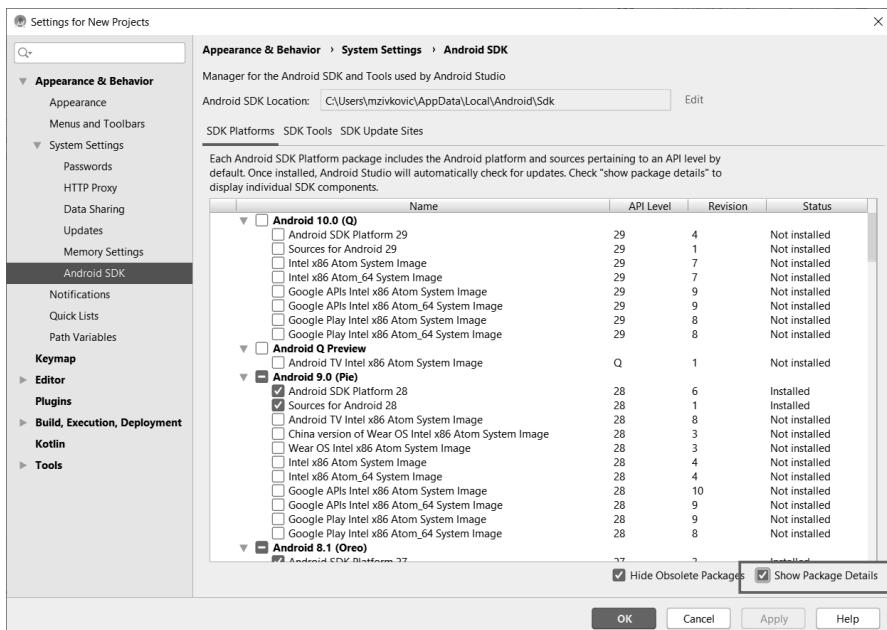
Slika 3.42, pokrenuta aplikacija na virtuelnom uređaju

3.4. SDK Manager

Android Studio nudi laku opciju preuzimanja novih SDK alata, platformi i drugih komponenti koje su potrebne za razvoj aplikacija, kao i njihovo lako održavanje i ažuriranje. U okviru sekcije **Tools**, odabirom stavke **SDK Manager** i taba **SDK Platforms**, moguće je videti listu svih dostupnih SDK, kao što je prikazano na slici 3.43. Neophodno je instalirati bar jednu verziju Android platforme u okruženju, kako bi bilo moguće kompajlirati aplikaciju. Dobra praksa je da se kao ciljna platforma koristi najnovija verzija SDK. Na taj način, aplikacija se kompajlira sa najnovijom verzijom biblioteka, čime se mogu koristiti najnovije funkcionalnosti koje mogu iskoristiti uređaji sa najnovijom verzijom Androida, a aplikacija se i dalje može izvršavati i na starijim verzijama Androida.

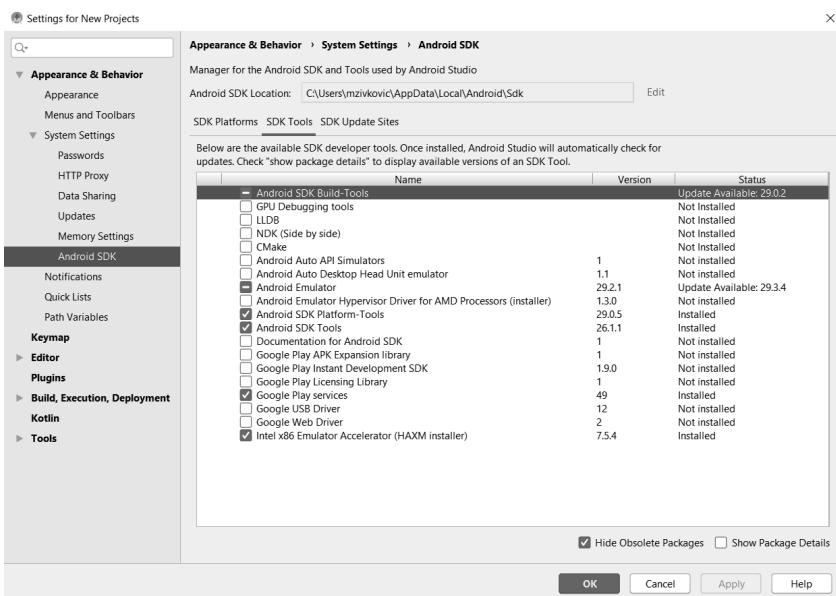


Slika 3.43, lista svih dostupnih SDK platformi



Slika 3.44, lista dostupnih SDK platformi sa detaljima za svaku pojedinačnu platformu

Za svaku dostupnu platformu moguće je videti njen status, odnosno da li je instalirana na računaru, kao i da li postoji dostupna novija verzija već instalirane platforme kroz opciju za ažuriranje. Takođe, moguće je videti detaljno sve komponente koje su raspoložive u okviru jednog SDK, klikom na dugme **Show Package Details**, kao što je prikazano na slici 3.44. Kroz ovaj prikaz moguće je instalirati samo one komponente koje su trenutno potrebne, a ne celu platformu. Željenu platformu ili komponentu je potrebno selektovati i zatim pritisnuti dugme **Apply**, čime će Android Studio započeti preuzimanje i instalaciju.



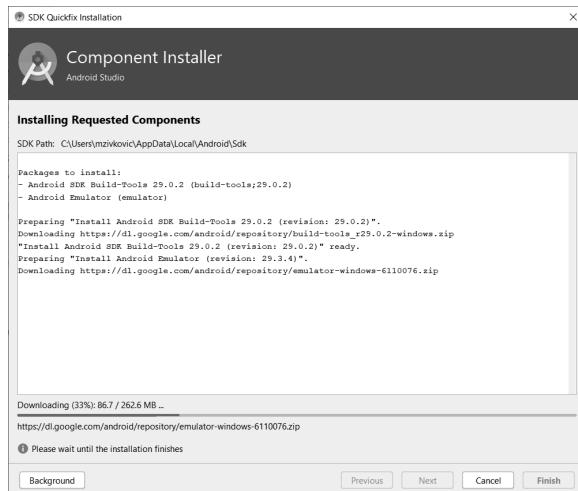
Slika 3.45, lista svih dostupnih SDK alata

U okviru taba **SDK Tools** moguće je videti listu svih dostupnih SDK razvojnih alata, kao što je prikazano na slici 3.45. Moguće je dodati nove alate, kao i ažurirati već instalirane ukoliko postoji nova verzija. Android Studio sam proverava dostupnost novih verzija, i označava sa crticom pakete koje je moguće ažurirati (na primer, na slici 3.45 vidi se da postoje nove verzije već instaliranih alata **Android SDK Build-Tools** i **Android Emulator**). Postoji veći broj dostupnih alata, manjeg ili većeg značaja, međutim, postoje alati koji su neophodni za normalan razvoj Android aplikacija koji bi trebalo da obavezno budu preuzeti, instalirani i dostupni u okruženju. Uz Android Emulator, u ove obavezne alate spadaju još:

- **Android SDK Build-Tools**, koji sadrži alate potrebne za build Android aplikacija.

- **Android SDK Platform-Tools**, koji sadrži veći broj važnih alata potrebnih Android platformi, poput adb.
- **Android SDK Tools**, sadrži ključne alate poput ProGuard (koji između ostalog služi za optimizaciju bajtkoda).

Ažuriranje i preuzimanje novih komponenti se prikazuje u obliku ekrana sa slike 3.46, a potrebno je napomenuti da je Android Studio nedostupan za vreme procesa ažuriranja.



Slika 3.46, proces ažuriranja komponenti

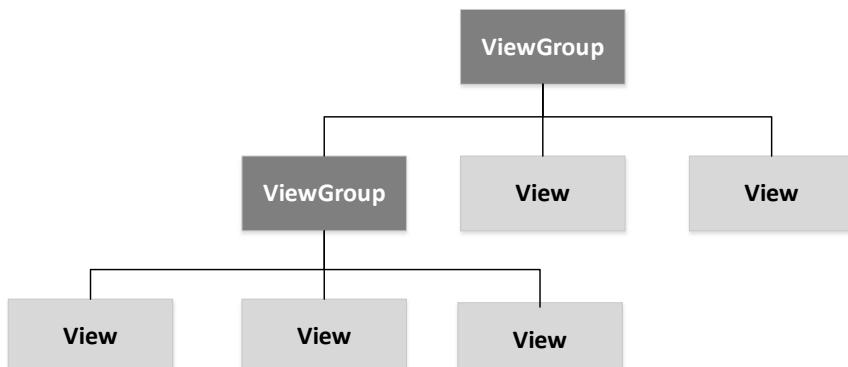
Kako bi se izvukao maksimum iz okruženja, neophodno je redovno ažurirati instalirane SDK i alate sa novim verzijama. Takođe, i samo okruženje treba redovno ažurirati, pošto će Android Studio obavestiti korisnika kada je dostupna nova verzija samog okruženja. Dobra praksa je da se nikada okruženje ne ažurira u toku jednog aktivnog projekta, pošto se dešava da izmene budu veće pa je neophodno utrošiti vreme u konfiguraciju i adaptaciju na novu verziju okruženja.

4. Grafički korisnički interfejs

Korisnički interfejs predstavlja sve što korisnik vidi na ekranu aplikacije i sa čime može da izvrši interakciju. To je prva i jedina tačka kontakta korisnika sa aplikacijom, kao i jedini deo aplikacije koji je vidljiv korisniku. Korisnički interfejs zbog toga mora biti funkcionalan i dopadljiv, zato što korisnici prilikom odabira između više sličnih aplikacija najčešće odluku donose samo na osnovu samog izgleda aplikacije (engl. *look and feel*). To je od naročitog značaja naročito u slučaju Android aplikacija, gde je konkurenčija na tržištu (Google Play) ogromna.

Kada se programira korisnički interfejs za Android aplikaciju, koriste se gotove Androidove komponente korisničkog interfejsa, poput strukturiranih rasporeda komponenti i različitih elemenata i kontrola. Android nudi i složenije module poput dijaloga, notifikacija i različitih oblika menija. Osnovne Android komponente koje služe za kreiranje korisničkog interfejsa su klase View i ViewGroup. Sve komponente od kojih se sastoji korisnički interfejs su instance ove dve klase. U opštem slučaju, kao što je prikazano na slici 4.1, jedan korisnički interfejs Android aplikacije je kreiran kao hijerarhija rasporeda komponenti (*layout*) i vidžeta (*widgets*):

- Rasporedi komponenti su objekti koji pripadaju klasama izvedenim iz klase ViewGroup, sa ulogom da kontrolišu kako se njihova deca (View komponente ili drugi ViewGroup objekti) pozicioniraju na ekranu.
- Vidžeti su objekti klasa izvedenih iz klase View, iscrtavaju se na ekranu i sa njima korisnik aplikacije može da vrši interakciju (na primer dugmići ili polja za unos teksta).



Slika 4.1, primer strukture korisničkog interfejsa sa ViewGroup komponentama i pripadajućim View objektima

Kao što smo ranije naveli, filozofija razvoja Android aplikacija jeste da se u potpunosti razdvoje programski kod i resursi. Zbog toga, Android nudi XML rečnik za sve ViewGroup i View klase, tako da se korisnički interfejs definiše kroz XML fajl. View komponente se mogu definisati i u programskom kodu u Javi, u slučaju da je potrebno dinamički dodati komponente. Ipak, preporučeni način je da se komponente grafičkoj interfejsa definisu u XML fajlu. Komponentama koje su definisane u XML fajlu može se pristupiti iz koda, upotrebom R.java klase, o čemu će biti reči kasnije.

4.1. Klasa View

Klasa View predstavlja osnovni gradivni blok za komponente korisničkog interfejsa. Iz ove klase izvedene su sve ostale klase korisničkog interfejsa, tzv. vidžetti, koji se koriste za pravljenje interaktivnih komponenti korisničkog interfejsa, poput dugmića i polja za unos teksta. ViewGroup klasa, koja je bazna klasa za sve rasporede komponenti, takođe je izvedena iz klase View.

Svi View objekti jednog ekrana su poređani u jedno stablo. Ovo stablo treba da ima tačno jedan koren element, koji je tipa ViewGroup. Kada je stablo kreirano i komponenta ubaćena u njega, postoji određen broj tipičnih operacija koje se najčešće izvode.

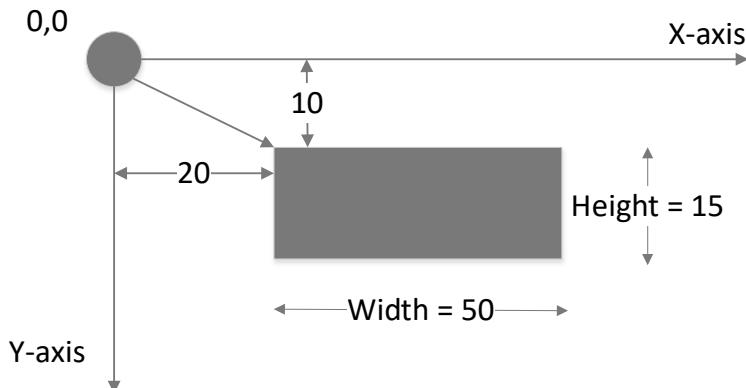
- Postavljanje atributa – poput postavljanja teksta u polje tipa TextView. Dostupne metode variraju u zavisnosti od konkretne izvedene klase.
- Postavljanje fokusa – pomeranje fokusa kao odgovor na neki korisnički unos. Fokus na određeno polje se može zahtevati pozivom metode requestFocus().
- Postavljanje osluškivača (engl. *listeners*) – View komponente dozvoljavaju postavljanje osluškivača, koji će biti obavešteni kada se desi nešto od interesa. Na primer, moguće je detektovati i obavestiti osluškivača kada komponenta dobije ili izgubi fokus. Izvedene klase imaju specijalizovane osluškivače, poput klase Button koja daje mogućnost da osluškivač bude obavešten kada se klikne na dugme.

View zauzima pravougaoni prostor na ekranu, i odgovoran je za crtanje komponente na ekranu, kao i za interakciju sa korisnikom, putem rukovanja događajima (engl. *event handling*). Svaki View ima svoju lokaciju na ekranu, koja je definisana pomoću para koordinata koje označavaju gornji levi čošak (x, y), i dve dimenzije koje označavaju širinu i visinu (*width* i *height*), kao što je prikazano na slici 4.2. Svaka komponenta koja je tipa View ima raspoložive geter metode koje dohvataju odgovarajuće elemente pozicije komponente na ekranu, tako da getLeft() i getTop() služe da dohvate levi čošak i vrate relativnu poziciju

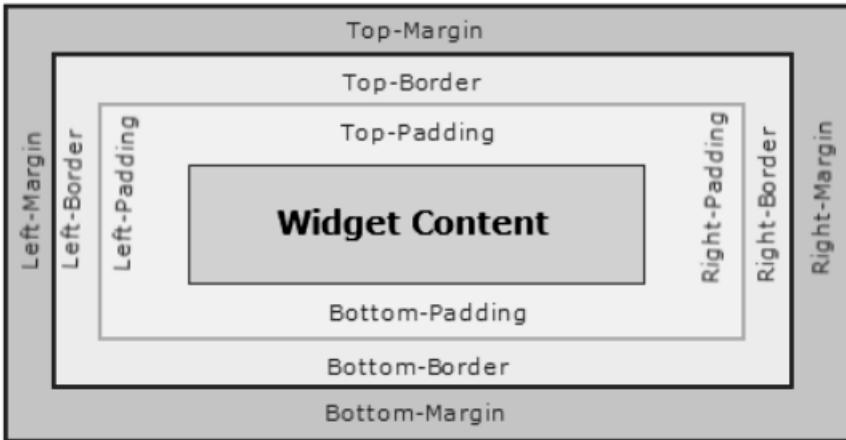
komponente u odnosu na roditeljsku komponentu. Analogno, `getRight()` i `getBottom()` služe za dohvatanje donjeg desnog čoška. Metode `getHeight()` i `getWidth()` dohvataju iscrtanu visinu i širinu komponente (pri čemu važi $\text{getLeft()} + \text{getWidth()} = \text{getRight}()$).

Osim pozicije, bitni parametri za prikaz svake View komponente su njena veličina (*size*), *padding* i margine. Veličina komponente se izražava kroz visinu i širinu. Svaki View zapravo sadrži dva para vrednosti za visinu i širinu. Uz već pomenute metode `getHeight()` i `getWidth()` koje dohvataju stvarnu veličinu iscrtane komponente na ekranu, postoji još dve metode, pod nazivom `getMeasuredHeight()` i `getMeasuredWidth()`, koje vraćaju izmerenu visinu i širinu. Ove dve vrednosti zapravo pokazuju koliko komponenta želi da bude velika u svojoj roditeljskoj komponenti, i one mogu, ali ne moraju da budu jednake iscrtanim vrednostima.

Prilikom merenja svojih dimenzija, View mora da uračuna i padding. Padding se izražava u pikselima od leve, gornje, desne i donje ivice komponente, i služi da opiše rastojanje od granica komponente do samog sadržaja koji se nalazi u komponenti. Može se postaviti metodom `setPadding(int, int, int, int)`, a dohvatiti get metodama `getPaddingLeft()`, `getPaddingTop()`, `getPaddingRight()` i `getPaddingBottom()`. Margine se sa druge strane odnose na prostor izvan granica komponente, odnosno definišu rastojanje od susednih komponenti (razlika između paddinga i margina prikazana je na slici 4.3). Margine se ne mogu podesiti na nivou pojedinačnih komponenti, već se postavljaju na nivou ViewGroup komponente koja služi za grupisanje komponenti.



Slika 4.2, definicija pozicije View komponente



Slika 4.3, padding i margine komponente

Svaki View može imati dodeljenu ID vrednost tipa integer. Ovi identifikatori se tipično dodeljuju u XML fajlu sa rasporedom komponenti postavljanjem atributa android:id, a služe za pronalaženje određene komponente u stablu. Ove vrednosti moraju biti jedinstvene. Na primer, u XML fajlu se može definisati dugme sa dodeljenim identifikatorom „mojeDugme“ na sledeći način:

```
<Button
    android:id="@+id/mojeDugme"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/moj_tekst"/>
```

@+id naredba iz prethodnog fragmenta koda govori da je potrebno dodati novi identifikator u R.java klasu. Nakon toga se dugmetu iz odgovarajuće aktivnosti može pristupiti na sledeći način (preko R.java klase, koristeći dodeljeni identifikator):

```
Button mojeDugme = findViewById(R.id.mojeDugme);
```

4.2. Standardne komponente grafičkog interfejsa

Android nudi veći broj predefinisanih komponenti koje programer može da koristi. Sve ove komponente su izvedene iz bazne klase View, a programer može modifikovati njihov prikaz na ekranu pomoću atributa. U predefinisane komponente spadaju tipovi prikazani u tabeli 4-1.

Tip	Funkcionalnost	Klasa
Label	Tekstualno polje za prikaz teksta	TextView
Button	Dugme na koje korisnik može da klikne kako bi izvršio neku akciju	Button
Text field	Polje za unos teksta, uz podršku za automatsku dopunu teksta i različite tipove unosa (broj, mejl adresa, lozinka itd.)	EditText, AutoCompleteTextView
Checkbox	Polje za potvrdu koje korisnik može da čekira, pri čemu su polja međusobno nezavisna	CheckBox
Radio button	Slično kao Checkbox, uz razliku da su opcije zavisne jedna od druge	RadioGroup, RadioButton
Toggle	Dugme sa dva stanja (on/off)	ToggleButton
Spinner	Drop down lista iz koje korisnik može odabratи vrednost	Spinner
Pickers	Dijalog gde se vrednost bira upotrebom dugmeta gore/dole ili pokreta	DatePicker, TimePicker

Tabela 4-1, standardne komponente grafičkog interfejsa

Prikaz svake komponente moguće je kontrolisati skupom atributa. Na primer, svaki View ima atribute koji kontrolisu njegovu veličinu. Oni se moraju definisati u okviru XML fajla za svaki individualni View. Primer podešavanja veličine TextView komponente dat je sledećim isečkom koda iz XML fajla.

```
<TextView
    android:layout_width="150dp"
    android:layout_height="wrap_content" />
```

Atribut android:layout_width služi da odredi širinu komponente, i on se obavezno mora definisati. Analogno, android:layout_height služi za određivanje visine komponente. Oba atributa se mogu definisati u nekoliko različitih formi:

- wrap_content – prilagođava širinu i visinu komponente veličini samog sadržaja te komponente.
- match_parent – prilagođava širinu i visinu komponente veličini roditeljske komponente.
- pomoću dimenzije u dp (engl. *density independent pixels*) – na primer 150dp.
- referenciranjem vrednosti iz dimens.xml fajla (u obliku @dimen/dim)

Ove vrednosti se mogu promeniti za vreme izvršavanja iz Java koda na nekoliko načina, na primer:

```
// Promena širine i visine
int novaSirina = 50; //u običnim pikselima
int novaVisina = 50;
view.setLayoutParams(new LayoutParams(novaSirina,
novaVisina));
// zahtevamo invalidaciju view kako bi se ponovo iscrtala
view.requestLayout();
```

Još jedan bitan atribut svake komponente je gravity, koji služi da se definiše na koju stranu treba da „gravitira“ sadržaj u okviru komponente. Dolazi u dva oblika:

- gravity – definiše smer u kome se sadržaj View komponente poravnava (veoma slično kao CSS text-align).
- layout_gravity – definiše smer u kome View gravitira unutar svoje roditeljske komponente (poput CSS float).

Dozvoljene vrednosti koje se mogu koristiti su: left, right, top, bottom, center, itd. Moguće je specificirati i više vrednosti, pri čemu se mora koristiti vertikalna crta za razdvajanje, na primer: android:gravity="top|right". Primer primene ovih atributa na komponentu tipa TextView dat je sledećim XML kodom:

```
<TextView
    android:gravity="left"
    android:layout_gravity="right"
    android:layout_width="150dp"
    android:layout_height="wrap_content"
    android:textSize="10sp"
    android:text="@string/hello_world" />
```

Klasa View ima veliki broj atributa koji se mogu primeniti na komponentu, kako bi se kontrolisali različiti aspekti komponente. Neki od najčešće korišćenih atributa su prikazani u tabeli 4-2. Ova lista nije konačna, već samo prikazuje pregled često korišćenih atributa. Uz svaki atribut, dat je i kratak opis, kao i primer vrednosti koja mu se može dodeliti.

Atribut	Opis	Primer vrednosti
android:background	Postavljanje pozadine za View	#ffffff
android:onClick	Metoda koja se poziva nakon klika na komponentu	onButtonClicked
android:visibility	Kontroliše prikaz komponente	invisible
android:hint	Hint tekst koji se prikazuje kada je komponenta prazna	@string/hint
android:text	Tekst koji se prikazuje u komponenti	@string/poruka
android:textColor	Boja teksta	#000000
android:textSize	Veličina teksta	21sp
android:textStyle	Stil formatiranja teksta	bold

Tabela 4-2, najčešće korišćeni atributi sa primerima vrednosti

Primer postavljanja više atributa za komponentu tipa TextView dat je sledećim isečkom XML koda. Ovim primerom se tekst koji se prikazuje u okviru komponente referencira iz strings.xml fajla, prikazuje se u beloj boji, na crnoj pozadini.

```
<TextView  
    android:layout_width="150dp"  
    android:layout_height="wrap_content"  
    android:text="@string/poruka"  
    android:background="#000"  
    android:textColor="#fff"  
/>
```

U nastavku su dati opisi osnovnih komponenti interfejsa opisanih u tabeli 4-1, sa tipičnim primerima upotrebe.

4.2.1. TextView

TextView je komponenta korisničkog interfejsa koja služi za prikaz teksta korisniku. Podrazumevano ponašanje TextView komponente jeste da tekst nije promenljiv, odnosno ponaša se kao labela na ekranu korisničkog interfejsa. Ukoliko je potrebno omogućiti korisniku da menja ili unosi tekst, koristi se EditText, koji je opisan nešto kasnije. Ključni atributi komponente tipa TextView (uz već navedene u prethodnom poglavlju) su:

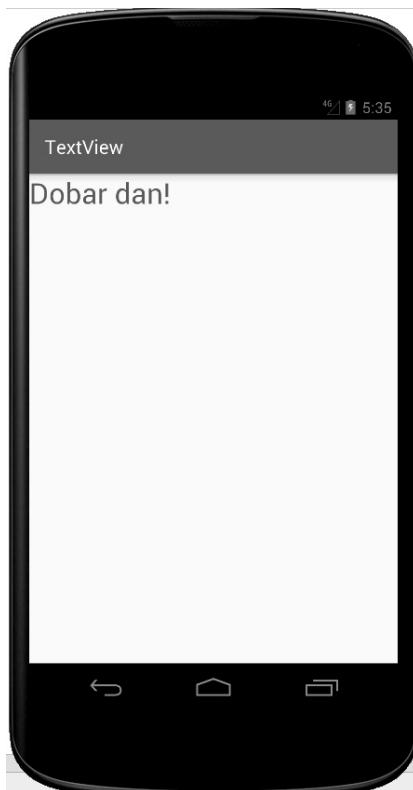
- android:id – opcionalo polje koje omogućava pristup ovoj komponenti iz Java koda, preko R.java klase, kao i referenciranje na ovu komponentu iz drugih komponenti u rasporedu.
- android:text – definiše se tekst koji će biti prikazan kao sadržaj komponente. Moguće je da bude direktno data („hard-coded“) fiksna vrednost, a može i da bude referenca na String resurs koji je definisan u strings.xml (@string naredba).

Tipičan primer upotrebe komponente tipa TextView dat je u sledećem primeru XML koda:

```
<LinearLayout  
  
    xmlns:android="http://schemas.android.com/apk/res/android"  
        android:layout_width="match_parent"  
        android:layout_height="match_parent">  
            <TextView  
                android:id="@+id/labelaText"  
                android:layout_height="wrap_content"  
                android:layout_width="wrap_content"
```

```
    android:textSize="30dp"
    android:text="Dobar dan!" />
</LinearLayout>
```

Ovako kreirana TextView komponenta će na ekranu biti iscrtana na način prikazan na slici 4.4.



Slika 4.4, prikaz tipične upotrebe TextView komponente

Komponenti se može pristupiti i iz programskog koda. Moguće je izmeniti sadržaj komponente definisane u XML fajlu tako što se željena komponenta referencira iz koda pomoću njenog ID uz pomoć R.java klase. Primer modifikovanja sadržaja dat je kroz jednostavnu aktivnost koja dohvata TextView polje i menja mu sadržaj novim Stringom koji se nalazi definisan u strings.xml fajlu. Primer podrazumeva da je sadržaj fajla strings.xml, koji se nalazi u res/values folderu, dat sa:

```
<resources>
    <string name="app_name">Korisnicki interfejs</string>
    <string name="nova_poruka">Dobrodosli na RMA!</string>
</resources>
```

U tom slučaju, kod MainActivity.java klase dat je sa:

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        TextView helloTextView =
findViewById(R.id.labelaText);
        helloTextView.setText(R.string.nova_poruka);
    }
}
```

Dimenzija teksta u prethodnom primeru zadata je atributom android:textSize. Ona može biti definisana na više načina, odnosno u više različitih mernih jedinica. Moguće jedinice koje se mogu koristiti su:

- px – veličina slova zadata u pikselima.
- in – veličina slova zadata u inčima.
- mm – veličina slova zadata u milimetrima.
- sp – veličina slova zadata u skaliranim pikselima baziranim na veličini slova koju je korisnik odabrao kao *preferred font size* u celom Android sistemu.
- dp – *density-independent pixels*.

Jedinica sa oznakom dp je apstraktna merna jedinica koja zavisi od fizičke gustine tačaka ekrana. Jedinica je zadata relativno u odnosu na 160 dpi (*dots per inch*) ekran, na kome važi da je 1dp ekvivalentan 1px. Kada se aplikacija izvršava na ekranu sa većom gustinom, broj piksela koji se koristi za crtanje 1dp se skalira naviše odgovarajućim faktorom koji zavisi od dpi ekrana. Analogno, ukoliko se aplikacija izvršava na ekranu sa manjom gustinom, broj piksela koji se koristi za crtanje 1dp se skalira naniže. U opštem slučaju, u vreme izvršavanja, dimenzija zadata u dp se preračunava po formuli $px = dp * (\text{dpi} / 160)$. Ovo je preporučeni način predstavljanja dimenzija komponenti, pošto će se ispravno iscrtavati na ekranima sa različitim gustinama tačaka.

4.2.2. EditText

EditText je komponenta korisničkog interfejsa koja služi za unos ili modifikaciju teksta. Direktno je izvedena iz klase TextView, a služi kao bazna klasa za klasu AutoCompleteTextView koja je izvedena iz nje. Kada se dodirne ovo polje, na njega se postavlja cursor i podiže virtualna tastatura u kojoj korisnik može da

uneset tekst, a dozvoljeno je da radi i copy/paste. Ključni atributi komponente EditText su:

- android:inputType – ovaj atribut konfiguriše tip virtuelne tastature koja će se prikazati, dozvoljene karaktere, kao i izgled polja. Na primer, ukoliko polje služi za unos neke tajne informacije, poput lozinke, ovaj atribut se može postaviti na vrednost textPassword, čime se otvara standardna virtuelna tastatura, a uneti karakteri se ne prikazuju na ekranu već se umesto njih prikazuju tačkice. Odabir number vrednosti za ovaj atribut će ograničiti unos na brojeve, a prikazaće se numerička tastatura. Druge moguće vrednosti uključuju textEmailAddress, phone, numericPassword itd. pri čemu svaka od opcija ima predefinisanu virtuelnu tastaturu.
- android:hint – prikaz pomoćnog teksta koji korisniku treba da sugerise šta treba uneti u polje.

Tipičan primer upotrebe EditText komponente dat je u sledećem primeru XML koda:

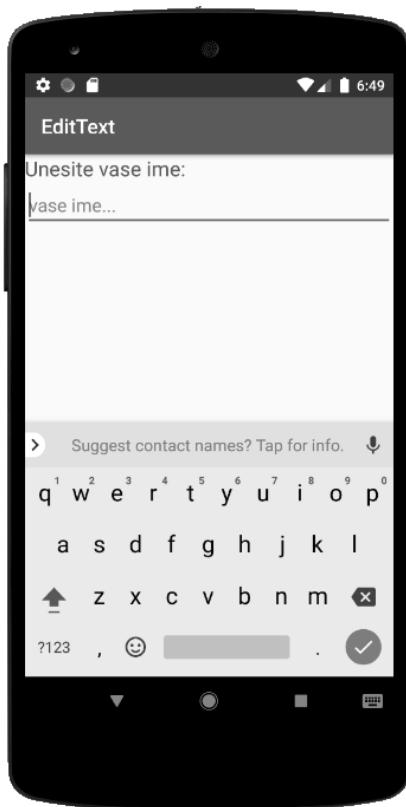
```
<LinearLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
        android:orientation="vertical"  
        android:layout_width="match_parent"  
        android:layout_height="match_parent">  
    <TextView  
        android:id="@+id/labelaPoruka"  
        android:layout_height="wrap_content"  
        android:layout_width="match_parent"  
        android:textSize="20dp"  
        android:text="Unesite vase ime:" />  
    <EditText  
        android:id="@+id/inputIme"  
        android:layout_height="wrap_content"  
        android:layout_width="match_parent"  
        android:hint="vase ime..."  
        android:inputType="text"/>  
</LinearLayout>
```

Komponenti se može pristupiti i iz programskog koda. Deo Java koda, koji bi se nalazio negde u MainActivity.java na mestu gde je potrebno dohvatiti unetu vrednost, i koji bi preko R.java klase pristupio sadržaju koji je korisnik uneo u polje i sačuvao ga u promenljivu tipa String, dat je sa:

```
EditText inputIme = findViewById(R.id.inputIme);  
String unetoIme = inputIme.getText().toString();
```

Nakon pokretanja aplikacije, ovako definisana komponenta biće iscrtana kao na slici 4.5. Dodir na polje za unos će otvoriti standardnu virtuelnu tastaturu, pošto je inputType polja definisan kao text. Drugačije definisana vrednost inputType prikazaće i drugaćiju tastaturu, kao što je prikazano na slici 4.6. Primer XML koda koji bi se koristio za definisanje EditText polja za unos telefona (inputType phone), dat je u nastavku:

```
<EditText  
    android:id="@+id/inputPhone"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:hint="@string/phone_hint"  
    android:inputType="phone" />
```



Slika 4.5, prikaz tipične upotrebe EditText komponente



Slika 4.6, različiti tipovi virtuelne tastature za različite vrednosti inputType atributa

4.2.3. Button

Button je komponenta korisničkog interfejsa koja prikazuje dugme. Dugme je objekat koji korisnik može da dodirne ili klikne na njega kako bi se izvršila neka akcija. Bitni atributi klase Button su:

- android:text – definiše tekst koji je isписан на dugmetu.
- android:onClick – opcionalno, definiše ime metode iz odgovarajuće aktivnosti čije će se izvršavanje okinuti kada se dugme klikne.

Tipičan primer definisanja dugmeta dat je u sledećem XML kodu.

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/labelaPoruka"
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:text="Pritisnite dugme" />
    <Button
        android:id="@+id/button"
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:text="Pritisni me!"/>
</LinearLayout>
```

Ovako definisano dugme će biti uredno prikazano na ekranu, međutim, ništa se neće dogoditi kada korisnik klikne na njega. Kako bi se nešto zaista desilo kada korisnik klikne na dugme, programer mora napisati metodu koja će predstavljati reakciju na klik. U ovom kontekstu, klik na dugme prestavlja događaj koji je korisnik generisao (engl. *event*). Kada se desi klik, objekat tipa Button će dobiti on-click događaj. Sledeća stvar koju programer treba da napiše jeste obrada događaja, koja je zapravo komad koda koji se piše unutar klase odgovarajuće aktivnosti. U opštem slučaju, to se može postići na dva načina.

Prvi način za obradu događaja jeste kroz Java kod odgovarajuće aktivnosti, i ovo je preferirani način obrade događaja. U tom slučaju, XML kod ostaje isti, dok se kompletna obrada događaja piše u kodu. Najpre je neophodno kreirati instancu klase View.OnClickListener, koja implementira istoimeni interfejs, koji je definisan u okviru klase View. U ovom primeru, koristi se objekat anonimne klase u pozivu metode setOnClickListener nad objektom tipa Button, čija je referenca dobijena preko R.java klase pozivom metode findViewById(). Pošto je u pitanju anonimna klasa koja implementira interfejs, neophodno je dati implementaciju nedostajućih metoda. Interfejs View.OnClickListener ima samo jednu deklarisanu apstraktну metodu, onClick(), koju je potrebno nadjačati i implementirati. To je metoda koja će biti pozvana kada se on-click događaj zaista desi, a ovakve metode su poznate kao callback metode. Unutar ove metode se samo poziva Toast preko kojeg će se korisniku prikazati kratka poruka na ekranu.

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button button = findViewById(R.id.button);
        button.setOnClickListener(new
View.OnClickListener() {
            public void onClick(View v) {
                // Kod koji se izvršava kada se klikne na
dugme
                Toast.makeText(getApplicationContext(),
"Klik!", Toast.LENGTH_LONG).show();
            }
        });
    }
}
```

Drugi način jeste da se obrada događaja definiše kroz XML fajl. U tom slučaju, prvo je neophodno malo izmeniti sam XML fajl. Potrebno je dodati atribut android:onClick, čija je vrednost zapravo ime metode koja treba da se pozove kada korisnik klikne na dugme. Izmenjeni XML fajl dat je u nastavku:

```

<LinearLayout

    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/labelaPoruka"
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:textSize="20dp"
        android:text="Pritisnite dugme" />
    <Button
        android:id="@+id/button"
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:onClick="prikaziPoruku"
        android:text="Pritisni me!"/>
</LinearLayout>

```

Na ovaj način definisano je kroz XML da kada se desi događaj klika na dugme, treba pozvati metodu čije je ime dato kao vrednost android:onClick atributa, odnosno u ovom slučaju metodu prikaziPoruku(). Ova metoda mora da se nalazi u aktivnosti kojoj odgovara ovaj XML fajl, i mora da ispunjava tri uslova:

- public modifikator pristupa – ukoliko bi bilo private, to znači da se metodi može pristupiti samo iz klase u kojoj je definisana, a ne iz pridruženog XML fajla.
- parametar tipa View – metoda mora imati ovaj parametar, jer će kroz njega biti prosleđen View koji je generisao događaj.
- povratni tip metode mora biti void.

Uz ova navedena dva uslova, java kod aktivnosti sada izgleda ovako:

```

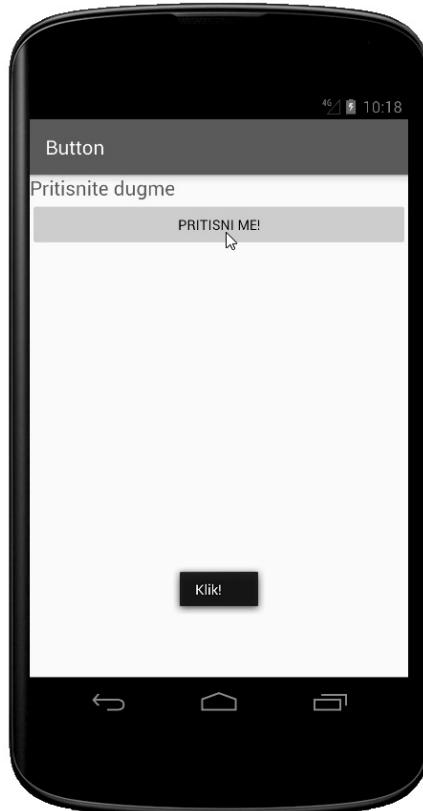
public class MainActivity extends AppCompatActivity {

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

    public void prikaziPoruku(View v) {
        Toast.makeText(getApplicationContext(), "Klik!", Toast.LENGTH_LONG).show();
    }
}

```

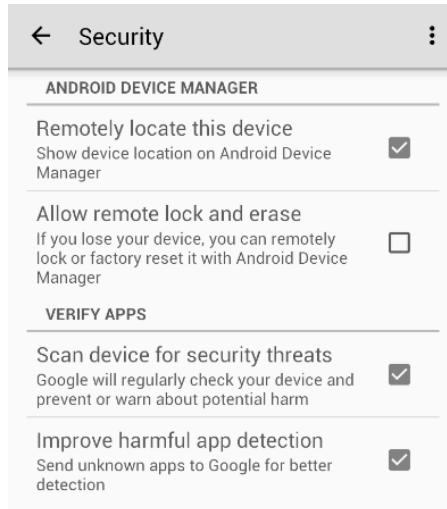
Kod metode prikaziPoruku je identičan kao kod onClick metode koji je upotrebljen u prvom opisanom pristupu. Svejedno koji se pristup koristi, rezultat je isti – nakon klika na dugme prikazaće se mali Toast sa tekstrom „Klik!“ na ekranu, kao što je prikazano na slici 4.7. Kao što je već napomenuto, iako se na oba načina može postići isti rezultat, preferirani način je da se to uradi na prvi način, kroz Java kod.



Slika 4.7, tipičan primer upotrebe dugmeta sa jednostavnom obradom događaja

4.2.4. CheckBox

Komponente tipa CheckBox omogućavaju korisniku da selektuje jednu ili više opcija iz skupa opcija. Tipično se ove komponente redaju u vertikalnoj listi – na taj način ih sam Android koristi kroz sistem, kao na slici 4.8. Svaki pojedinačni CheckBox može biti u jednom od dva stanja – čekiran i nečekiran, nezavisno od stanja drugih CheckBox komponenti.



Slika 4.8, komponente CheckBox

Za kreiranje svake pojedinačne opcije, kreira se po jedan CheckBox. Pošto je moguće da korisnik odabere i više opcija u isto vreme, svaka opcija se mora pratiti pojedinačno, odnosno mora se registrovati osluškivač za svaku pojedinačnu opciju. Kada korisnik selektuje jednu opciju, taj CheckBox objekat dobija on-click događaj. Slično kao u primeru za Button, rukovanje događajem se može implementirati na dva načina – kroz XML i kroz Java kod.

Ukoliko se rukovanje događajem implementira kroz XML, potrebno je svakom pojedinačnom CheckBox elementu u XML fajlu dodati android:onClick atribut, čija je vrednost ime metode koju želimo da pozovemo kao odgovor na događaj. Naravno, aktivnost koja odgovara XML fajlu mora implementirati taj metod. XML kod i Java kod odgovarajuće aktivnosti dati su u nastavku.

```
<LinearLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
        android:orientation="vertical"  
        android:layout_width="match_parent"  
        android:layout_height="match_parent">  
    <TextView  
        android:id="@+id/labelaPoruka"  
        android:layout_height="wrap_content"  
        android:layout_width="match_parent"  
        android:textSize="20dp"  
        android:text="Odaberite dodatke za kafu" />  
    <CheckBox android:id="@+id/checkbox_mleko"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"
```

```

    android:text="Mleko"
    android:onClick="onCheckboxClicked"/>
<CheckBox android:id="@+id/checkbox_secer"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Secer"
    android:onClick="onCheckboxClicked"/>
</LinearLayout>

```

U slučaju klika na bilo koji od dve opcije, poziva se metoda `onCheckboxClicked()`, koja mora biti implementirana u odgovarajućoj aktivnosti. U samoj metodi neophodno je znati na koju konkretnu opciju je korisnik kliknuo. To se postiže proverom Id komponente v tipa View, koja je generisala dogadjaj i koja je prosleđena kao parametar u metodi. Id se dobija pozivom metode `v.getId()`, i zatim kroz switch naredbu poredi sa Id svake pojedinačne opcije. Kod odgovarajuće aktivnosti dat je u nastavku.

```

public class MainActivity extends AppCompatActivity {

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

    public void onCheckboxClicked(View v) {
        // Da li je view sada cekiran?
        boolean checked = ((CheckBox) v).isChecked();
        // Proveri koji checkbox je upravo kliknut
        switch(v.getId()) {
            case R.id.checkbox_mleko:
                if (checked) {
                    //dodaj mleko u kafu
                } else{
                    //nemoj stavljati mleko
                }
                break;
            case R.id.checkbox_secer:
                if (checked) {
                    //dodaj secer u kafu
                } else {
                    //nemoj stavljati secer
                }
                break;
            // TODO: druge opcije
        }
    }
}

```

Ukoliko se koristi drugi pristup, gde se rukovanje događajem kompletno implementira u kodu, XML fajl i kod odgovarajuće aktivnosti dati su u nastavku. XML fajl je skoro identičan, osim uklonjenih android:onClick atributa.

```
<LinearLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
        android:orientation="vertical"  
        android:layout_width="match_parent"  
        android:layout_height="match_parent">  
    <TextView  
        android:id="@+id/labelaPoruka"  
        android:layout_height="wrap_content"  
        android:layout_width="match_parent"  
        android:textSize="20dp"/>  
    <CheckBox android:id="@+id/checkbox_mleko"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="Mleko">  
    <CheckBox android:id="@+id/checkbox_secer"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="Secer"/>  
</LinearLayout>
```

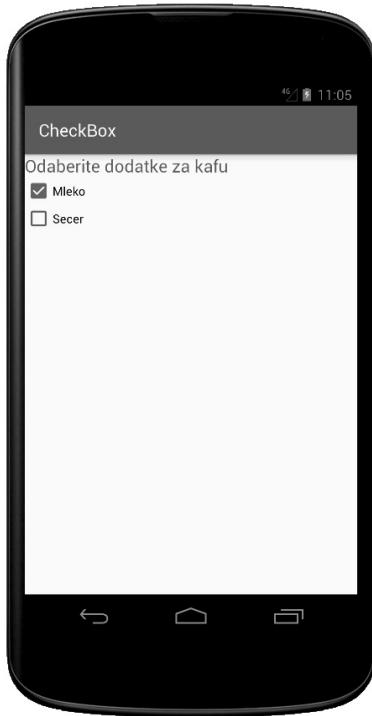
Odgovarajuća aktivnost je implementirana nešto drugačije. Programeri vrlo često žele da izbegnu rad sa anonimnim klasama, pa se često sama klasa aktivnosti proglaši za osluškivač. To se postiže tako što se u zagлавljtu klase proglaši da klasa implementira View.OnClickListener interfejs. Na taj način, prilikom postavljanja osluškivača na odgovarajuću komponentu, kao parametar metode setOnClickListener() aktivnost može proslediti samu sebe, prosleđivanjem vrednosti this. To je moguće pošto setOnClickListener() metoda očekuje objekat neke klase koja implementira interfejs View.OnClickListener, a naša aktivnost jeste takav objekat. Kako bi se ispoštovala implementacija interfejsa, u samoj klasi aktivnosti potrebno je implementirati onClick() metodu definisanu u interfejsu View.OnClickListener, tipa void i sa jednim parametrom tipa View – to je komponenta koja je generisala događaj.

```
public class MainActivity extends AppCompatActivity  
implements View.OnClickListener {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
  
        CheckBox checkBoxMleko =
```

```
findViewById(R.id.checkbox_mleko);
    checkBoxMleko.setOnClickListener(this);
    CheckBox checkBoxSecer =
findViewById(R.id.checkbox_secer);
    checkBoxSecer.setOnClickListener(this);
}

public void onClick(View v){
    // Da li je view sada cekiran?
    boolean checked = ((CheckBox) v).isChecked();
    // Proveri koji checkbox je upravo kliknut
    switch(v.getId()) {
        case R.id.checkbox_mleko:
            if (checked) {
                //dodaj mleko u kafu
            } else{
                //nemoj stavljati mleko
            }
            break;
        case R.id.checkbox_secer:
            if (checked) {
                //dodaj secer u kafu
            } else {
                //nemoj stavljati secer
            }
            break;
        // TODO: druge opcije
    }
}
}
```

U oba pristupa, ponašanje aplikacije je identično. Prikaz aplikacije dat je na slici 4.8.



Slika 4.9, tipičan primer upotrebe CheckBox komponenti

4.2.5. RadioButton

Komponente tipa RadioButton su komponente sa dva stanja (čekiran ili nečekiran), i veoma su slične CheckBox komponentama, uz bitnu razliku da RadioButton komponente rade u grupi. Unutar jedne grupe RadioButton komponenti samo jedna može biti odabrana u jednom trenutku. Klik na bilo koji drugi RadioButton u okviru iste grupe odčekiraće sve ostale RadioButton komponente te grupe. Grupa se definiše RadioGroup objektom.

RadioButton komponente omogućavaju korisniku da odabere jednu opciju iz skupa opcija. Koriste se u slučaju da je potrebno da opcije budu međusobno isključive, a želimo da sve ponuđene opcije budu prikazane jedna pored druge. Ukoliko nije potrebno da budu prikazane jedna pored druge već jedna ispod druge, preporučuje se da se koristi Spinner. Kreiraju se vrlo slično kao CheckBox komponente, ali se moraju grupisati zajedno unutar grupe – RadioGroup elementa. Na taj način se obezbeđuje da samo jedna komponenta iz jedne grupe može biti odabrana u jednom trenutku. RadioGroup je izvedena iz LinearLayout, i podrazumevano ima vertikalnu orientaciju ređanja komponenti. Tipičan primer upotrebe dat je u nastavku, sledećim XML fajlom:

```

<?xml version="1.0" encoding="utf-8"?>
<RadioGroup
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">
    <RadioButton android:id="@+id/radio_zvezda"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Zvezda"
        android:onClick="onRadioButtonClicked"/>
    <RadioButton android:id="@+id/radio_partizan"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Partizan"
        android:onClick="onRadioButtonClicked"/>
</RadioGroup>

```

U ovom primeru se od korisnika traži da iskaže svoje navijačko opredeljenje, a ponuđene opcije su isključive. Bilo koja RadioButton komponenta da se odabere, pozvaće se metoda onRadioButtonClicked, koja se mora implementirati u odgovarajućoj aktivnosti. I ovde se rukovanje događajem može implementirati na dva načina, veoma slično kao i kod CheckBox komponenti. Zbog izrazite sličnosti, prikazana je implementacija rukovanja događajem kroz XML, dok se implementacija kompletног rukovanja kroz Java kod realizuje analogno i preporučuje se čitaocu kao vežba.

Implementacija aktivnosti koja odgovara ovom XML fajlu data je u nastavku. Slično kao kod CheckBox komponenti, u metodi onRadioButtonClicked se prvo proverava koja komponenta v tipa View je generisala događaj, dohvatanjem vrednosti v.getId() i upotrebom naredbe switch da se ta vrednost uporedi sa Id svakog RadioButton elementa ponaosob. Izgled aplikacije nakon pokretanja prikazan je na slici 4.10.

```

public class MainActivity extends AppCompatActivity {

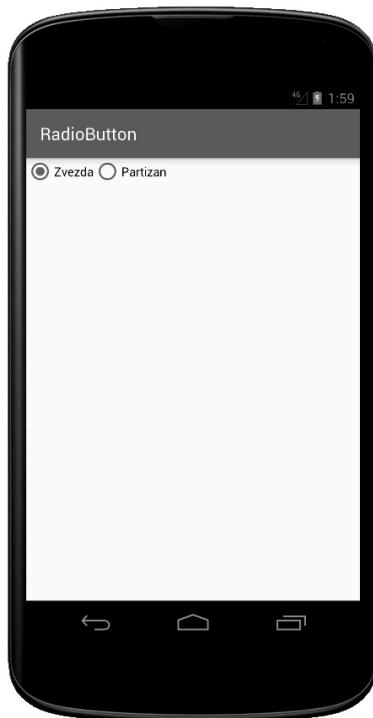
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void onRadioButtonClicked(View view) {
        // Da li je dugme sada cekirano?
        boolean checked = ((RadioButton) view).isChecked();

        // Proveri koje radio dugme je kliknuto
        switch(view.getId()) {

```

```
case R.id.radio_zvezda:  
    if (checked)  
        // Zvezda je najbolja!  
        break;  
case R.id.radio_partizan:  
    if (checked)  
        // Partizan je najbolji  
        break;  
}  
}  
}
```



Slika 4.10, tipičan primer upotrebe RadioButton komponente

4.2.6. Spinner

Spineri omogućavaju brz odabir jedne vrednosti iz zadatog skupa vrednosti. Spiner pokazuje trenutno odabranu vrednost kao podrazumevanu vrednost u prikazu. Dodir na Spiner prikazuje klasičan padajući meni sa svim dostupnim vrednostima, od kojih korisnik može odabrati jednu.

Spinner se jednostavno dodaje u XML rasporedu ubacivanjem Spinner elementa. Na primer, u sledećem XML fajlu dodat je spinner koji će kao vrednosti sadržavati sve planete Sunčevog sistema:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">
    <TextView
        android:id="@+id/labelPlanete"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Planete" />
    <Spinner
        android:id="@+id/spinner"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"/>
</LinearLayout>
```

Kako bi se spinner popunio sa listom opcija, u aktivnosti se mora specificirati SpinnerAdapter. Opcije kojima se popunjava spinner mogu doći iz bilo kog izvora, ali se moraju proslediti kroz jedan SpinnerAdapter, na primer kroz ArrayAdapter ukoliko su opcije dostupne u nizu vrednosti, ili kroz CursorAdapter ukoliko su opcije dostupne kao rezultat upita u bazu. Na primer, ukoliko su opcije za spinner predefinisane, one se mogu definisati u obliku niza Stringova u fajlu strings.xml. Primer sadržaja strings.xml fajla dat je sledećim isečkom XML koda:

```
<resources>
    <string name="app_name">Spinner</string>
    <string-array name="niz_planeta">
        <item>Merkur</item>
        <item>Venera</item>
        <item>Zemlja</item>
        <item>Mars</item>
        <item>Jupiter</item>
        <item>Saturn</item>
        <item>Uran</item>
        <item>Neptun</item>
    </string-array>
</resources>
```

Kod aktivnosti koja odgovara datom XML fajlu prikazan je u nastavku. Metoda createFromResource() omogućava kreiranje ArrayAdapter objekta iz niza Stringova. Treći argument ove metode predstavlja layout resurs koji definiše

kako će se odabrana opcija prikazati u spineru – simple_spinner_item je podrazumevani layout koji je obezbeđen od strane Android platforme, i koji treba da se koristi uvek (osim ukoliko programer ne želi da samostalno definiše layout za prikaz spinera). Nakon toga, nad objektom klase adapter poziva se metoda setDropDownViewResource(int), koja specificira layout koji će adapter koristiti da prikaže listu spinner opcija (simple_spinner_dropdown_item je takođe standardna opcija koju obezbeđuje Android). Nakon toga se poziva metoda setAdapter() nad objektom tipa Spinner kako bi se kreirani adapter primenio. Izgled samog spinera prikazan je na slici 4.11, gde se vidi prikaz odabrane komponente, kao i prikaz liste komponenti nakon dodira na komponentu spinner.

```
public class MainActivity extends AppCompatActivity {

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

        Spinner spinner = (Spinner)
        findViewById(R.id.spinner);
        // Kreira se ArrayAdapter koji koristi niz
        // stringova I podrazumevani spinner layout
        ArrayAdapter<CharSequence> adapter =
        ArrayAdapter.createFromResource(this,
            R.array.niz_planeta,
        android.R.layout.simple_spinner_item);
        // Specificira se layout koji se koristi kada se
        // pojavi lista opcija
        adapter.setDropDownViewResource(android.R.layout.
        simple_spinner_dropdown_item);
        // primenjuje se adapter na spinneru
        spinner.setAdapter(adapter);
    }
}
```

Dodatno, potrebno je navesti da kada korisnik odabere neku od opcija iz padajuće liste, Spinner objekat dobija on-item-selected događaj. Moguće je dodati u aktivnost kod kojim se rukuje ovim tipom događaja, implementacijom interfejsa osluškivača AdapterView.OnItemSelectedListener i dodavanjem odgovarajuće onItemSelected() callback metode. Potrebno je dodati i implementaciju onNothingSelected() metode, koju zahteva implementirani interfejs. Primer upotrebe ovog osluškivača je dat sa:

```
public class SpinnerActivity extends Activity implements  
OnItemSelectedListener {  
    ...  
    public void onItemSelected(AdapterView<?> parent, View  
view,  
        int pos, long id) {  
// Opcija je odabrana, i moze se dohvati upotrebom  
// parent.getItemAtPosition(pos)  
    }  
  
    public void onNothingSelected(AdapterView<?> parent) {  
        // Another interface callback  
    }  
}
```

Postavljanje ovog osluškivača na objekat tipa Spinner se radi na sledeći način:

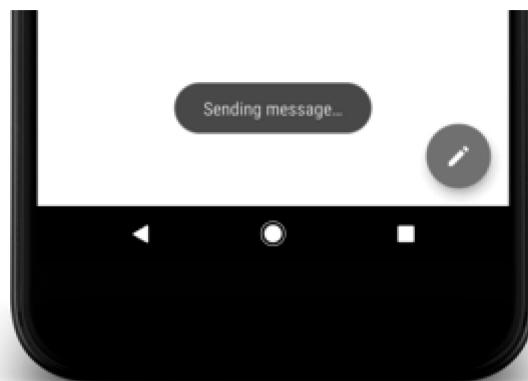
```
Spinner spinner = findViewById(R.id.spinner);  
spinner.setOnItemSelectedListener(this);
```



Slika 4.11, tipičan primer upotrebe Spinner komponente

4.2.7. Toast

Toast služi za prikaz jednostavnih poruka korisniku aplikacije putem malog pop-up prozora. Obično se koristi da se korisniku da povratna informacija o nekoj izvršenoj akciji. Na primer, ukoliko se u mejl aplikaciji klikne na Send, obično se prikaže kratka „Sending message...“ Toast poruka, kao što je prikazano na slici 4.12. Na Toast poruku nije moguće kliknuti. Ona prilikom prikaza na ekranu zauzima samo prostor potreban za ispis poruke, pri čemu trenutna aktivnost ostaje vidljiva i interaktivna. Toast automatski nestaje nakon određenog vremena.



Slika 4.12, prikaz kratke Toast poruke u mail aplikaciji
(slika preuzeta sa: <https://developer.android.com/guide/topics/ui/notifiers/toasts>)

Kreiranje Toast poruke je veoma jednostavno. Prvo, potrebno je napraviti instancu Toast klase pozivom makeText() metode. Ova metoda prihvata tri parametra: kontekst aplikacije, tekstualnu poruku i dužinu prikaza Toast poruke na ekranu, i kao rezultat vraća ispravno inicijalizovani Toast objekat. Prikaz Toast poruke na ekranu postiže se pozivom metode show() nad Toast objektom. Tipičan primer upotrebe Toast poruke dat je sledećim Java kodom.

```
Context context = getApplicationContext();
CharSequence text = "Zdravo, ovo je Toast!";
int duration = Toast.LENGTH_SHORT;

Toast toast = Toast.makeText(context, text, duration);
toast.show();
```

Obično se koristi još kraća i jednostavnija sintaksa. Moguće je koristiti ulančavanje metoda i izbeći potrebu za Toast promenljivom, na sledeći način:

```
Toast.makeText(getApplicationContext(), text,
Toast.LENGTH_LONG).show();
```

Navedeni primer je standardan način korišćenja Toast poruke, dovoljan za sve primene u aplikacijama. Moguće je uticati na dužinu prikaza sa dve opcije: Toast.LENGTH_SHORT za kratak prikaz, i Toast.LENGTH_LONG za duži prikaz. Retko kad je potrebno nešto dodatno uraditi. Na primer, može se uticati na poziciju gde će se Toast prikazati upotrebom setGravity() metode. Standardna Toast notifikacija se prikazuje pri dnu ekrana, centrirana horizontalno. Metoda setGravity() prihvata tri parametra: Gravity konstantu, pomeraj po x osi i pomeraj po y osi. Na primer, ukoliko bismo odlučili da Toast poruka treba da se prikaže u gornjem levom uglu ekrana, potreбно bi bilo uraditi sledeće:

```
toast.setGravity(Gravity.TOP|Gravity.LEFT, 0, 0);
```

Ukoliko bismo želeli da gurnemo poziciju prikaza poruke u desnu stranu, trebalo bi povećati drugi parametar. Ukoliko želimo da pomerimo poruku na dole, trebalo bi povećati treći parametar).

Primer jednostavne aplikacije koja prikazuje Toast poruku svaki put kad se klikne na dugme je dat u nastavku.

XML fajl:

```
<?xml version="1.0" encoding="utf-8" ?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">
    <TextView
        android:id="@+id/labelDemo"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="20dp"
        android:text="Toast primer" />
    <Button
        android:id="@+id/button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Prikazi Toast" />
</LinearLayout>
```

Java kod odgovarajuće aktivnosti, sa implementacijom osluškivača za on-click događaj je dat u nastavku. Osluškivač je realizovan tako što sama klasa aktivnosti implementira odgovarajući View.OnClickListener, pa je moguće prilikom postavljanja osluškivača na objekat tipa Button proslediti metodi setOnClickListener() parametar this u pozivu.

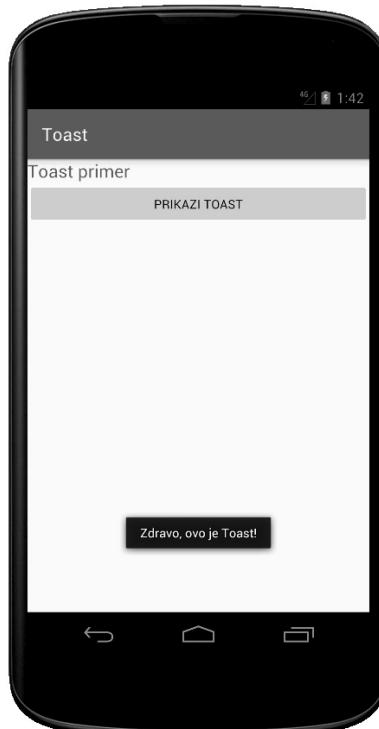
```
public class MainActivity extends AppCompatActivity
implements View.OnClickListener{

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

        Button button = findViewById(R.id.button);
        button.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        CharSequence text = "Zdravo, ovo je Toast!";
        Toast.makeText(getApplicationContext(), text,
        Toast.LENGTH_LONG).show();
    }
}
```

Primer izvršavanja ove aplikacije prikazan je na slici 4.13.



Slika 4.13, primer upotrebe Toast poruke

4.2.8. DatePicker i TimePicker

DatePicker klasa predstavlja vidžet za odabir datuma, i direktno nasleđuje klasu Picker. Analogno, TimePicker klasa predstavlja vidžet za odabir vremena, i takođe nasleđuje klasu Picker. Picker komponente omogućavaju korisniku aplikacije da odabere vrednost iz skupa vrednosti upotrebom pokreta. U slučaju DatePicker i TimePicker komponenti, moguće je pristupiti pojedinačnim delovima datuma i vremena, kao što je prikazano na slici 4.14.



Slika 4.14, TimePicker i DatePicker komponente

(slika preuzeta sa:

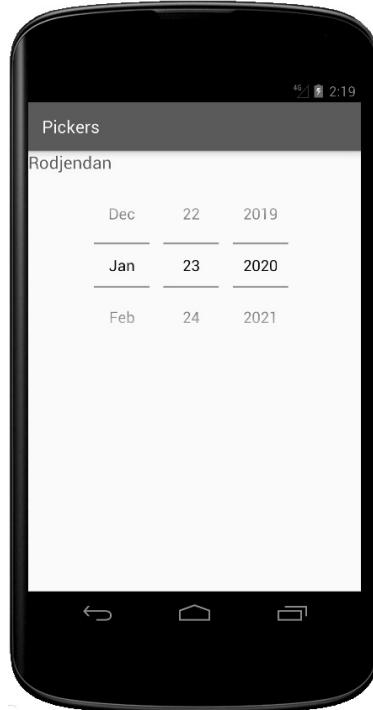
<https://developer.android.com/guide/topics/ui/controls/pickers>)

Svaki Picker pruža kontrole za odabir svakog pojedinačnog dela vremena (sat, minut, AM/PM) ili datuma (mesec, dan, godina). Upotrebom Picker komponenti se osigurava da će korisnik aplikacije odabrati validan datum koji je ispravno formatiran i lokalizovan. Picker se može dodati kroz XML fajl na sledeći način (dat je primer za DatePicker, pošto se sa TimePicker komponentom radi analogno).

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <TextView
        android:id="@+id/labelRojdanje"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="20dp"
        android:text="Rojdanje"/>
    <DatePicker
        android:id="@+id/rojdanje"
        android:layout_width="match_parent"
```

```
    android:layout_height="wrap_content"
    android:calendarViewShown="false"/>
</LinearLayout>
```

Izgled DatePicker komponente u okviru interfejsa aplikacije dat je na slici 4.15.



Slika 4.15, Izgled DatePicker komponente u okviru aplikacije

Iz aktivnosti koja odgovara datom XML fajlu, DatePicker komponenti se može pristupiti na sledeći način kako bi se, na primer, pojedinačne komponente datuma izvukle i formatirale u String koji se može dalje koristiti u okviru aplikacije:

```
DatePicker rodjendan = (DatePicker)
findViewById(R.id.rodjendan);
String rodjendan = String.format("%4d-%2d-%2d",
        rodjendan.getYear(), rodjendan.getMonth(),
        rodjendan.getDayOfMonth());
```

Preporučeni način upotrebe DatePicker ili TimePicker komponenti od strane Android zajednice jeste da se koristi DialogFragment kao nosilac ovih komponenti, i nešto je kompleksniji od prethodnog primera. Kako bi se definisao DialogFragment za DatePicker komponentu, neophodno je definisati onCreateDialog() metodu na takav način da vrati instancu klase DatePickerDialog. Nakon toga, potrebno je implementirati interfejs

DatePickerDialog.OnDateSetListener kako bi se dobio callback kada korisnik postavi novi datum. Primer ovakve upotrebe dat je u nastavku kroz Java kod odgovarajućeg fragmenta.

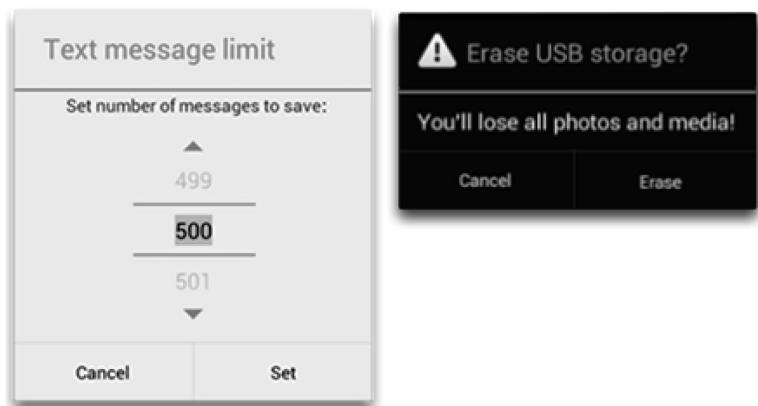
```
public static class DatePickerFragment extends  
DialogFragment implements  
DatePickerDialog.OnDateSetListener {  
  
    @Override  
    public Dialog onCreateDialog(Bundle savedInstanceState)  
{  
        // koristi se trenutni datum kao podrazumevani  
        final Calendar c = Calendar.getInstance();  
        int year = c.get(Calendar.YEAR);  
        int month = c.get(Calendar.MONTH);  
        int day = c.get(Calendar.DAY_OF_MONTH);  
  
        // kreiramo instancu DatePickerDialog i vracamo je  
        return new DatePickerDialog(getActivity(), this,  
year, month, day);  
    }  
  
    public void onDateSet(DatePicker view, int year, int  
month, int day) {  
        // Uraditi nesto sa datumom koji je korisnik  
postavio  
    }  
}
```

Datum definisan na ovakav način se može prikazati na ekranu pozivom metode:

```
public void showDatePickerDialog(View v) {  
    DialogFragment newFragment = new DatePickerFragment();  
    newFragment.show(getSupportFragmentManager(),  
"datePicker");  
}
```

4.2.9. Dialog

Dialog komponenta se odnosi na mali prozor koji se prikazuje korisniku i od njega traži da napravi neku odluku ili da unese dodatne informacije. Dialog komponenta ispunjava samo deo ekrana i obično se koristi za modalne događaje koji zahtevaju od korisnika da izvrši neku akciju pre nastavka rada. Primer Dialog komponenti dat je na slici 4.16.



Slika 4.16, tipični primeri Dialog komponenti
(slika preuzeta sa: <https://developer.android.com/guide/topics/ui/dialogs>)

Dialog klasa je bazna za sve dijaloge, ali preporuka je da se izbegava direktno instanciranje ove klase. Umesto toga, koriste se izvedene klase poput AlertDialog, ili već pomenutih DatePickerDialog i TimePickerDialog. Sve ove izvedene klase definišu svoj stil i strukturu. Na primer, AlertDialog je komponenta koja se sastoji od naslova, maksimalno tri dugmeta, liste stavki koje se mogu odabrat ili se može koristiti sopstveni layout fajl. U ovom udžbeniku će biti prikazan primer jednostavnog AlertDialoga, dok se za dodavanje sopstvenih dugmeća, listi i slično čitalac upućuje na dodatnu literaturu (<https://developer.android.com/guide/topics/ui/dialogs>).

Još jedna preporuka je da se za svaki Dialog koristi DialogFragment kao nosilac komponente. DialogFragment pruža sve potrebne kontrole za kreiranje Dialog komponente, kao i za njen izgled. Upotrebom DialogFragment klase se osigurava ispravno ponašanje Dialog komponente tokom njenog životnog veka i ispravno rukovanje dogadjajima koje generiše korisnik, poput klika na Back dugme ili rotiranja ekrana. DialogFragment omogućava i da se korisnički interfejs komponente iskoristi kao ubaćena komponenta u većem korisničkom interfejsu, kao u slučaju upotrebe tradicionalnih fragmenata.

Primer osnovnog AlertDialog-a, koji koristi DialogFragment kao nosioca, dat je u nastavku. U okviru onCreateDialog metode se koristi AlertDialog.Builder za kreiranje AlertDialog-a, pomoću kojeg je moguće postaviti poruku (setMessage()), kao i dugme sa pozitivnim i negativnim odgovorom (setPositiveButton() i setNegativeButton(), respektivno). Odgovarajuće onClick() metode obezbeđuju callback pozive kada korisnik klikne na neku od ponuđenih opcija.

```

public class FireMissilesDialogFragment extends
DialogFragment {
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState)
{
    // Builder klasa se koristi za udobniju
    konstrukciju dialoga
    AlertDialog.Builder builder = new
AlertDialog.Builder(getActivity());
    builder.setMessage("Fire missiles?")
        .setPositiveButton("Yes", new
DialogInterface.OnClickListener() {
            public void onClick(DialogInterface
dialog, int id) {
                // FIRE ZE MISSILES!
            }
        })
        .setNegativeButton("No", new
DialogInterface.OnClickListener() {
            public void onClick(DialogInterface
dialog, int id) {
                // Korisnik je otkazao dialog
            }
        });
    // Kreira se AlertDialog objekat i vraca kao
    rezultat
    return builder.create();
}
}

```

Kada želimo da prikažemo naš Dialog, potrebno je napraviti instancu DialogFragment klase i pozvati metodu show(), kojoj se prosleđuju FragmentManager i ime dijaloga kao parametri. FragmentManager se može dohvatiti na dva načina:

- pozivom getSupportFragmentManager() iz aktivnosti, ili
- pozivom getFragmentManager() iz fragmenta.

Primer poziva je dat sa:

```

FireMissilesDialogFragment fragment = new
FireMissilesDialogFragment();
fragment.show(getFragmentManager(), "missiles");

```

Kompletan kod koji ilustruje upotrebu ovako napravljenog AlertDialog-a dat je u nastavku. XML fajl je definisan na sledeći način:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">
    <TextView
        android:id="@+id/labelDialog"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="20dp"
        android:text="Fire the missiles"/>

    <Button
        android:id="@+id/button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Fire" />
</LinearLayout>

```

Aplikacija sadrži jedno dugme tipa Button, čijim se pritiskom otvara definisani Dialog. U onClick() callback metodi aktivnosti koja odgovara ovom XML fajlu se zbog toga poziva FireMissileDialogFragment. Kod aktivnosti je dat u nastavku, a izgled aplikacije i prikazanog dijaloga nakon pritiska na dugme prikazan je na slici 4.17.



Slika 4.17, prikaz jednostavnog Dialog-a

```

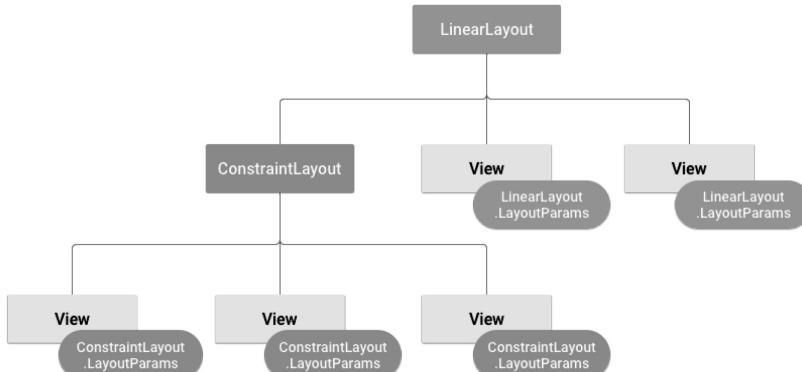
public class MainActivity extends Activity implements
View.OnClickListener{
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button button = findViewById(R.id.button);
        button.setOnClickListener(this);
    }
    public void onClick(View v){
        FireMissilesDialogFragment fragment = new
FireMissilesDialogFragment();
        fragment.show(getFragmentManager(), "missiles");
    }
}

```

4.3. Rasporedi komponenti (layout)

Često je potrebno grupisati komponente grafičkog interfejsa u grupe, kako bi se, na primer, iscrtali na odgovarajući način na ekranu telefona. ViewGroup je specijalna komponenta koja služi kao kontejner za grupisanje drugih komponenti – unutar ViewGroup komponente se mogu smestiti i View i ViewGroup komponente, koje su u tom slučaju njena deca.

Svaka klasa izvedena iz ViewGroup klase implementira ugnježdenu klasu koja je izvedena iz ViewGroup.LayoutParams. Ova podklasa sadrži atribute koji definišu veličinu i poziciju svakog deteta. Na slici 4.18 se može videti da svaka roditeljska ViewGroup komponenta definiše layout parametre za svako svoje dete (uključujući i druge ViewGroup komponente).



Slika 4.18, hijerarhija komponenti sa pridruženim layout parametrima
(slika preuzeta sa: <https://developer.android.com/guide/topics/ui/declaring-layout>)

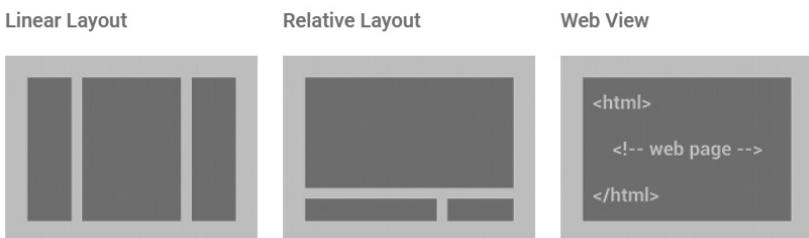
Svaka podklasa klase LayoutParams ima svoju sintaksu za postavljanje vrednosti. Svako dete element mora definisati LayoutParams koji su odgovarajući za njegovog roditelja, a takođe može definisati drugačije LayoutParams za svoju decu koje će oni morati da ispoštiju. Svaki ViewGroup sadrži dva obavezna atributa, širinu i visinu (layout_width i layout_height), a opcionalno mogu definisati i marge i okvire. Visina i širina se mogu specificirati sa tačnim merama (npr. u pikselima), ali se to ne radi često, već se obično specificiraju jednom od sledeće dve konstante:

- wrap_content – veličina komponente je određena dimenzijama sadržaja te komponente.
- match_parent – komponenta može da postane onoliko velika koliko će joj njena roditeljska komponenta dopustiti.

Uopšteno govoreći, specificiranje širine i visine rasporeda upotreboom apsolutnih jedinica poput piksela nije preporučljivo od strane Android zajednice. Umesto toga, bolji pristup je upotreba relativnih mera poput dp (density-independent pixels), wrap_content ili match_parent. Na taj način se obezbeđuje da se aplikacija iscrta na odgovarajući način na širokom opsegu uređaja sa različitim veličinama ekrana.

Svaka od podklasa klase ViewGroup prikazuje svoju decu na jedinstven način. Najčešće upotrebljavani tipovi rasporeda koji su ugrađeni u Android platformu su (slika 4.19):

- Linearni raspored (LinearLayout) – reda svoju decu linearno u jedan horizontalni ili vertikalni niz, a ukoliko prozor probije veličinu ekrana automatski kreira scrollbar.
- Relativni raspored (RelativeLayout) – dozvoljava specificiranje lokacije svoje dece relativno jedno u odnosu na drugo (dete A je levo od deteta B) ili u odnosu na roditeljsku komponentu (dete poravnato uz gornju granicu roditeljske komponente).
- Veb raspored (WebView) – služi za prikaz veb stranica.

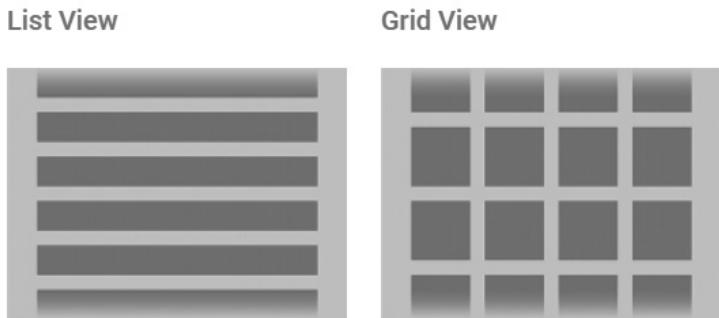


Slika 4.19, najčešće korišćeni rasporedi komponenti
(slika preuzeta sa: <https://developer.android.com/guide/topics/ui/declaring-layout>)

Kada raspored komponenti nije unapred poznat, već se dinamički određuje u vreme izvršavanja, može se koristiti jedan od rasporeda koji je izведен iz bazne klase AdapterView. Ovakvi rasporedi mogu da popune svoj layout sa komponentama u vreme izvršavanja upotrebljajom Adapter objekta koji služi da poveže podatke sa layout-om. Adapter služi kao komponenta u sredini između izvora podataka i AdapterView rasporeda tako što preuzima podatke (izvor može biti niz podataka, upit u bazu podataka ili podaci dovućeni sa Interneta) i pretvara svaki pojedinačni podatak u View komponentu koja se može dodati u AdapterView. Najčešće upotrebljavani rasporedi koji se zasnivaju na adapteru su (slika 4.20):

- ListView – omogućava dinamičko popunjavanje komponenti koje se smeštaju u listu sa jednom kolonom koja može da se skroluje.
- GridView – omogućava dinamičko popunjavanje komponenti koje se smeštaju u mrežu (sa kolonama i redovima) koja može da se skroluje.

U nastavku ovog poglavlja biće detaljno opisani najčešće korišćeni rasporedi, uz primere koji će ilustrovati tipične upotrebe.



Slika 4.20, najčešće korišćeni rasporedi komponenti koji koriste adapter (slika preuzeta sa: <https://developer.android.com/guide/topics/ui/declaring-layout>)

4.3.1. Linearni raspored

Linearni raspored (LinearLayout klasa) je raspored koji ređa svoju decu u jednom smeru, vertikalno ili horizontalno (slika 4.21). Smer se definiše atributom android:orientation, koji može da uzme jednu od dve ponuđene vrednosti: horizontal i vertical (podrazumevana vrednost je horizontal).



Slika 4.21, linearni raspored komponenti
(slika preuzeta sa:

<https://developer.android.com/guide/topics/ui/layout/linear.html>)

Sva deca linearog rasporeda se postavljaju jedno za drugim, tako da vertikalna lista ima samo jednu kolonu, odnosno po jedno dete u svakom redu (nebitno koliko su pojedina deca široka), a horizontalna lista ima samo jedan red (čija će visina biti određena visinom najvišeg deteta, uz padding). Linearni raspored uračunava i marge između dece, kao i gravity (right, center ili left poravnjanje) svakog pojedinačnog deteta.

Linearni raspored dozvoljava dodeljivanje težine svakom pojedinačnom detetu kroz atribut `android:layout_weight`. Ovaj atribut zapravo označava koliko je komponenta „bitna“ u smislu koliko prostora će zauzeti na ekranu. Veća vrednost težine dozvoljava komponenti da se proširi kako bi ispunila nepotpunjen prostor u roditeljskoj komponenti. Svako dete može specificirati težinu, a onda će preostali dostupni prostor biti proporcionalno podeljen komponentama na osnovu njihove deklarisane težine. Podrazumevana vrednost ovog atributa je 0.

Na primer, ukoliko se želi linearan raspored u kome svako dete komponenta uzima istu količinu prostora na ekranu, potrebno je postaviti atribut `android:layout_height` svakog deteta na `0dp` (za vertikalni raspored), odnosno `android:layout_width` svakog deteta na `0dp` (za horizontalni raspored), a nakon toga postaviti težinu svakog deteta postavljanjem atributa `android:layout_weight` na 1.

Ukoliko je potrebna nejednaka rasподела, moguće je napraviti linearni raspored na takav način da različita deca uzimaju različite veličine prostora na ekranu. Na primer:

- U rasporedu postoje tri `TextView` komponente, pri čemu prve dve deklarišu težinu 1, a trećoj nije dodeljena nikakva težina. U tom slučaju, treći `TextView` koji je bez dodeljene težine neće moći da raste. Zapravo,

on će zauzimati samo onoliko prostora koliko zahteva njegov sadržaj. Druge dve TextView komponente će se podjednako proširiti kako bi ispunile kompletan prostor koji preostaje nakon merenja sve tri komponente.

- Ukoliko prve dve komponente tipa TextView deklarišu težinu 1, a treća TextView komponenta deklariše težinu 2 (umesto 0), ona se posmatra kao važnija komponenta u poređenju sa prve dve, i preuzima polovinu preostalog prostora. Prve dve komponente dele ostatak prostora ravnomerno.

Kako bismo ilustrovali rad sa težinama, neka se posmatra konkretni primer gde je potrebno razviti korisnički ekran u obliku vertikalnog linearног rasporeda koji služi u aktivnosti za slanje poruke u okviru neke aplikacije. Neka je potrebno da postoje polja **Primalac**, **Naslov** i dugme **Pošalji** koji uzimaju onoliko prostora koliko im je potrebno. Preostali prostor visine ekrana ove aktivnosti treba da bude rezervisan za samu poruku. XML fajl koji ispunjava ove zahteve dat je u nastavku. Posebna pažnja treba da se obrati na EditText koji se koristi za unos teksta poruke. Pošto se želi da on zauzme kompletan preostali prostor koji ostane nakon merenja potrebnog prostora svih komponenti, potrebno je navesti da atribut `android:layout_height` bude postavljen na vrednost `0dp`, a `android:layout_weight` bude postavljen na vrednost 1.

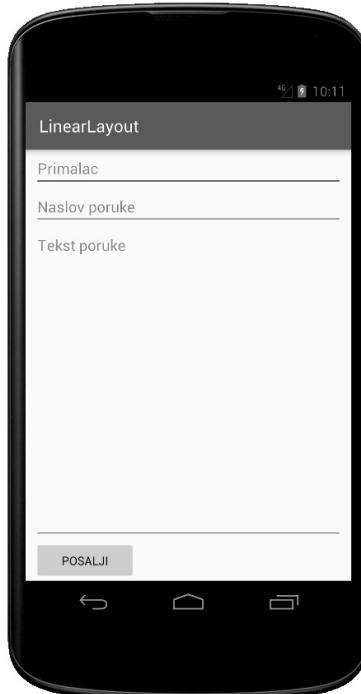
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:paddingLeft="10dp"
        android:paddingRight="10dp"
        android:orientation="vertical" >
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/primalac" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/naslov" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="top"
        android:hint="@string/tekst" />
    <Button
        android:layout_width="120dp"
```

```
    android:layout_height="wrap_content"
    android:layout_gravity="left"
    android:text="@string/posalji" />
</LinearLayout>
```

Ovaj primer podrazumeva da su Stringovi koji se koriste nisu hard-kodovani već da su definisani u strings.xml fajlu (preporučeni način, koji olakšava održavanje i lokalizaciju aplikacije). Sadržaj strings.xml fajla:

```
<resources>
    <string name="app_name">LinearLayout</string>
    <string name="primalac">Primalac</string>
    <string name="naslov">Naslov poruke</string>
    <string name="tekst">Tekst poruke</string>
    <string name="posalji">Posalji</string>
</resources>
```

Izgled rezultujućeg ekrana dat je na slici 4.22.



Slika 4.22, primer vertikalnog linearnog rasporeda sa težinama

4.3.2. Relativni raspored

Relativni raspored (RelativeLayout) je raspored koji svoju decu postavlja na relativne pozicije (slika 4.23). Pozicija svakog deteta može se postaviti relativno u odnosu na neku drugu komponentu dete (poput levo ili ispod druge komponente) ili relativno u odnosu na roditeljsku RelativeLayout komponentu (na primer poravnato sa dnom, levo, desno ili centrirano).



Slika 4.23, relativni raspored komponenti
(slika preuzeta sa:
<https://developer.android.com/guide/topics/ui/layout/relative.html>)

Relativni raspored je veoma moćan alat za dizajn korisničkog interfejsa pošto eliminiše potrebu za ugnježđenim linearnim rasporedima, čime se hijerarhija rasporeda održava ravnom (što može drastično uticati na performanse). Opšte pravilo je da ukoliko je za pravljenje zahtevanog rasporeda potrebno više ugnježđenih linearnih rasporeda, treba razmisliti i zameniti ih sa jednim relativnim rasporedom.

U relativnom rasporedu komponenta se pozicionira relativno u odnosu na roditeljsku komponentu ili u odnosu na neku drugu komponentu, pri čemu se komponente referenciraju putem svog ID. Dakle, moguće je poravnati nekoliko komponenti uz levu ivicu, ili staviti jednu komponentu ispod druge, centrirati komponentu i slično. Podrazumevano se sve komponente iscrtavaju u gornjem levom uglu rasporeda, pa je neophodno definisati poziciju svake komponente kako se ne bi iscrtavale jedna preko druge. Za ovo se koriste atributi definisani u RelativeLayout.LayoutParams. U ovoj klasi postoji veliki broj parametara, od kojih su neki:

- android:layout_alignParentTop – true vrednost ovog atributa će poravnati gornju ivicu ove komponente sa gornjom ivicom roditeljske komponente.

- android:layout_centerVertical – true vrednost ovog atributa će centrirati ovu komponentu vertikalno u okviru roditeljske komponente.
- android:layout_below – poravnava gornju granicu ove komponente ispod komponente referencirane sa svojim ID.
- android:layout_toRightOf – postavlja levu ivicu ove komponente desno od komponente referencirane sa svojim ID.

Detaljna lista svih opcija za relativno pozicioniranje je data u tabeli 4-3 (izvor <https://developer.android.com/reference/android/widget/RelativeLayout.LayoutParams.html>).

XML atributi	Opis
android:layout_above	Positions the bottom edge of this view above the given anchor view ID.
android:layout_alignBaseline	Positions the baseline of this view on the baseline of the given anchor view ID.
android:layout_alignBottom	Makes the bottom edge of this view match the bottom edge of the given anchor view ID.
android:layout_alignEnd	Makes the end edge of this view match the end edge of the given anchor view ID.
android:layout_alignLeft	Makes the left edge of this view match the left edge of the given anchor view ID.
android:layout_alignParentBottom	If true, makes the bottom edge of this view match the bottom edge of the parent.
android:layout_alignParentEnd	If true, makes the end edge of this view match the end edge of the parent.
android:layout_alignParentLeft	If true, makes the left edge of this view match the left edge of the parent.
android:layout_alignParentRight	If true, makes the right edge of this view match the right edge of the parent.
android:layout_alignParentStart	If true, makes the start edge of this view match the start edge of the parent.
android:layout_alignParentTop	If true, makes the top edge of this view match the top edge of the parent.
android:layout_alignRight	Makes the right edge of this view match the right edge of the given anchor view ID.
android:layout_alignStart	Makes the start edge of this view match the start edge of the given anchor view ID.
android:layout_alignTop	Makes the top edge of this view match the top edge of the given anchor view ID.

android:layout_alignWithParentIfMissing	If set to true, the parent will be used as the anchor when the anchor cannot be found for layout_toLeftOf, layout_toRightOf, etc.
android:layout_below	Positions the top edge of this view below the given anchor view ID.
android:layout_centerHorizontal	If true, centers this child horizontally within its parent.
android:layout_centerInParent	If true, centers this child horizontally and vertically within its parent.
android:layout_centerVertical	If true, centers this child vertically within its parent.
android:layout_toEndOf	Positions the start edge of this view to the end of the given anchor view ID.
android:layout_toLeftOf	Positions the right edge of this view to the left of the given anchor view ID.
android:layout_toRightOf	Positions the left edge of this view to the right of the given anchor view ID.
android:layout_toStartOf	Positions the end edge of this view to the start of the given anchor view ID.

Tabela 4-3, lista svih atributa za relativno pozicioniranje komponenti

Jedna stvar na koju je potrebno обратити пажњу јесте да се не доде до ситуације да се направи tzv. кружна зависност током дефинисања relativnih pozicija komponenti, односно да се компонента A постави у односу на компоненту B, а компонента B се постави у односу на компоненту A, што није дозвољено. Dodatno, може да се desi да ukoliko se направи грешка у дефинисању relativnih pozicija komponenti да једна компонента zakloni другу компоненту. На primer, ако су две View компоненте истовремено poravnate sa горњом и левом ivicom roditeljske компоненте, друга компонента ће zakloniti прву.

Tipičan primer upotrebe relativnog rasporeda приказан је кроз implementaciju ekrana koji predstavlja login formu. Moguće je уочити да XML težи да буде значајно veći nego u slučaju linearног rasporeda, zbog потребе за дефинисањем relacija između komponenti. XML je dat u sledećem isečku koda:

```
<?xml version="1.0" encoding="utf-8" ?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
        android:id="@+id/labelLogin"
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerInParent="false"
        android:text="@string/login"
        android:textSize="20dp"
        android:layout_marginTop="5dp" />
<TextView
        android:id="@+id/labelUsername"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/labelLogin"
        android:layout_marginTop="20dp"
        android:text="@string/username"
        android:textSize="22dp" />

<EditText
        android:id="@+id/inputUsername"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_alignTop="@+id/labelUsername"
        android:layout_toRightOf="@+id/labelUsername" />
<TextView
        android:id="@+id/labelPassword"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/labelUsername"
        android:layout_marginTop="20dp"
        android:text="@string/password"
        android:textSize="22dp" />
<EditText
        android:id="@+id/inputPassword"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_alignTop="@+id/labelPassword"
        android:layout_toRightOf="@+id/labelPassword"
        android:inputType="textPassword" />
<Button
        android:id="@+id/btnSubmit"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="false"
        android:layout_below="@+id/inputPassword"
        android:layout_centerInParent="true"
        android:text="@string/submit" />
<Button
        android:id="@+id/btnSignUp"
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:text="@string/signup"
        android:layout_centerHorizontal="true"/>
</RelativeLayout>
```

Stringovi su definisani u res/values folderu, u fajlu strings.xml, čiji je sadržaj dat u nastavku. Izgled rezultujućeg ekrana aplikacije prikazan je na slici 4.24.

```
<resources>
    <string name="app_name">RelativeLayout</string>
    <string name="login">LOGIN</string>
    <string name="username">Username :</string>
    <string name="password">Password :</string>
    <string name="submit">Submit</string>
    <string name="signup">Sign up</string>
</resources>
```



Slika 4.24, primer relativnog rasporeda

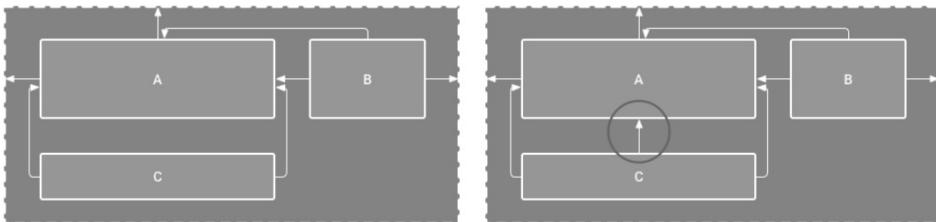
4.3.3. Ograničeni raspored

Ograničeni raspored (ConstraintLayout) je raspored komponenti veoma sličan relativnom rasporedu, sa tom razlikom što je fleksibilniji. Prema najavama najnovije verzije Androida, ovaj raspored spada u deo Androidovih biblioteka koje se više ne razvijaju, odnosno, neće više biti podržan dolaskom AndroidX platforme. Inicijalno, ConstraintLayout je dozvoljavao kreiranje velikih i kompleksnih rasporeda sa ravnom hijerarhijom (bez ugnježđavanja, kao kod relativnog rasporeda). Komponente se postavljaju na osnovu odnosa između samih komponenti i odnosa sa roditeljskom komponentom.

Osim što je nešto fleksibilniji od relativnog rasporeda, glavna prednost ograničenog rasporeda je direktna dostupnost kroz Layout Editor alat (grafički editor za kreiranje rasporeda komponenti), pošto su specijalno razvijeni jedno za drugo. Kada se kreira raspored komponenti pomoću ConstraintLayout, u potpunosti je moguće razviti ga pomoću drag & drop-a komponenti iz paleta, bez pisanja XML koda.

Kako bi se definisala pozicija komponente u ograničenom rasporedu, potrebno je dodati najmanje jedno horizontalno i jedno vertikalno ograničenje za tu komponentu. Svako ograničenje predstavlja konekciju ili poravnjanje sa drugom komponentom, roditeljskim rasporedom ili sa nevidljivom linijom vodiljom. Svako ograničenje definiše poziciju komponente na vertikalnoj i horizontalnoj osi, tako da je minimum potrebnih ograničenja jedno po osi, a vrlo često je potrebno i više. Kada se komponenta prevuče iz palete na ekran u grafičkom editoru, ona ostaje pozicionirana tamo gde je spuštena čak i ako nema ograničenja. Međutim, to je učinjeno samo da bi se olakšalo dalje modifikovanje. Ako komponenta nema ograničenja, prilikom izvršavanja na fizičkom uređaju biće iscrtana na poziciji (0,0), odnosno u gornjem levom ugлу ekrana.

Raspored komponenti sa leve strane slike 4.25 izgleda dobro u grafičkom editoru, međutim, ne postoji vertikalno ograničenje za komponentu C (ograničena je samo horizontalno tako da bude poravnata sa komponentom A). Prilikom crtanja ovog rasporeda na fizičkom uređaju, komponenta C će biti poravnata po levoj i desnoj ivici sa komponentom A, ali će biti prikazana na vrhu ekrana jer nema vertikalno ograničenje. Raspored komponenti sa desne strane slike 4.25 će biti ispravno prikazan na fizičkom uređaju, pošto postoji i vertikalno ograničenje komponente C u odnosu na komponentu A. Ukoliko neko ograničenje nedostaje, to neće izazvati grešku pri kompajliranju, međutim, biće označeno u grafičkom editoru kao greška. Grafički editor se može podesiti tako da automatski doda ograničenja prilikom pozicioniranja komponenti na ekranu.



Slika 4.25, dodavanje ograničenja komponenti u ograničenom rasporedu
 (izvor: <https://developer.android.com/training/constraint-layout>)

4.3.4. ListView

ListView raspored prikazuje vertikalnu kolekciju komponenti kroz koju je moguće skrolovati, pri čemu se svaka komponenta pozicionira neposredno ispod prethodne komponente u listi. Postoji i fleksibilnija opcija, RecyclerView (uz ograničenje da RecyclerView neće biti podržan od strane AndroidX platforme).

ListView raspored ćemo opisati na tipičnom primeru. Potrebno je prikazati listu predmeta na fakultetu, koji su za potrebe primera smešteni u niz (mogu se dovlačiti i sa Interneta ili iz baze). Pošto se potrebni podaci nalaze u nizu, može se koristiti ArrayAdapter. ArrayAdapter podrazumevano kreira komponentu za svaki element niza tako što poziva njegovu `toString()` metodu, i smešta dobijeni sadržaj u TextView. ArrayAdapter se inicijalizuje na sledeći način:

```
ArrayAdapter adapter = new ArrayAdapter<String>(this,
    R.layout.list_item, nizKurseva);
```

Parametri koji se prosleđuju konstruktoru su kontekst aplikacije (u većini slučajeva može se proslediti `this`), XML fajl sa rasporedom komponenti koji obezbeđuje prikaz pojedinačnog elementa iz niza, a poslednji argument je niz Stringova odakle će se izvlačiti podaci za popunjavanje svakog pojedinačnog TextView elementa.

ListView raspored se jednostavno deklariše u fajlu sa XML rasporedom. Sadržaj fajla `activity_main.xml`:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <ListView
        android:id="@+id/listaPredmeta"
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content" >
    </ListView>
</LinearLayout>
```

Potrebno je dodati još jedan fajl u res/layout direktorijum, gde će biti opisan raspored jedne komponente liste za prikaz. Dizajn pojedinačne komponente ListView rasporeda dat je fajlom list_item.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/label"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:padding="8dp"
        android:textSize="20dp"
        android:textStyle="bold" >
</TextView>
```

Kod aktivnosti MainActivity.java:

```
package com.example.student.listview;
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.ArrayAdapter;
import android.widget.ListView;

public class MainActivity extends AppCompatActivity {

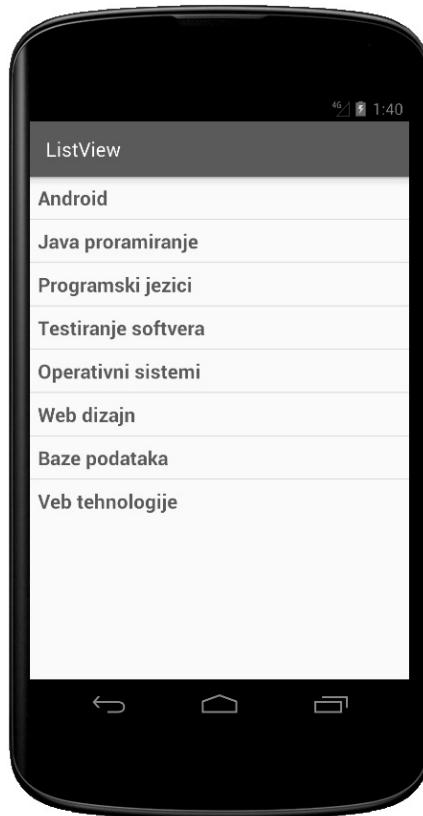
    // Niz kurseva na fakultetu...
    String[] nizKurseva = {"Android", "Java
proramiranje", "Programski jezici", "Testiranje softvera",
        "Operativni sistemi", "Web dizajn", "Baze
podataka", "Veb tehnologije"};

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

        ArrayAdapter adapter = new
ArrayAdapter<String>(this,
            R.layout.list_item, nizKurseva);

        ListView listView =
findViewById(R.id.listaPredmeta);
        listView.setAdapter(adapter);
    }
}
```

Nakon pokretanja aplikacije, njen prikaz na ekranu dat je na slici 4.26.



Slika 4.26, tipičan primer upotrebe ListView rasporeda komponenti

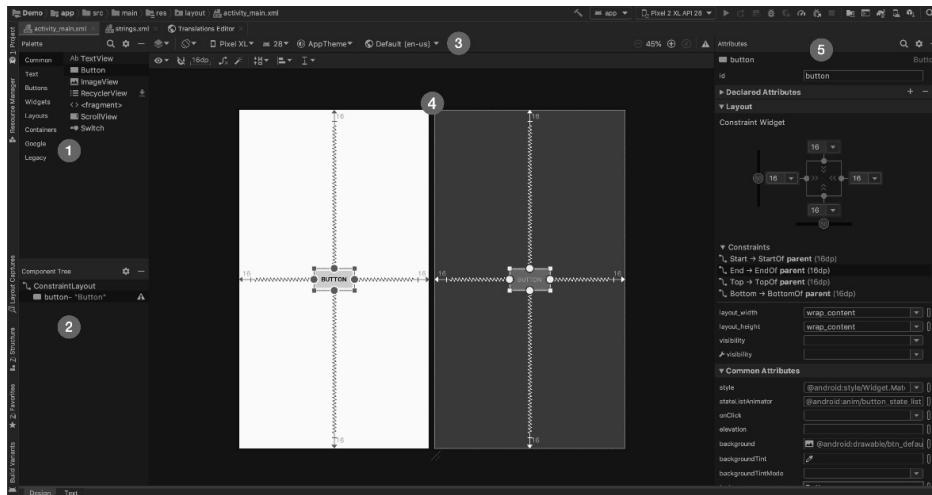
4.4. Layout editor

Android Studio sadrži grafički editor za dizajniranje korisničkog interfejsa, pod nazivom Layout Editor. Upotrebom editora može se veoma brzo napraviti raspored komponenti na ekranu, prostim prevlačenjem komponenti iz palete u radni prostor editora, čime se skoro u potpunosti eliminiše potreba da se komponente definišu ručno pisanjem XML fajla. Editor omogućava i trenutni pregled rasporeda, kao i prikaz na više različitih verzija Android uređaja i veličina ekrana, kako bi se raspored mogao dinamički prilagoditi. Na taj način se osigurava da će dizajn raditi dobro na različitim ekranima.

LayoutEditor se može koristiti sa svim rasporedima komponenti. Optimizovan je inicijalno za ograničeni raspored, međutim sasvim dobro radi i sa relativnim i sa

linearnim rasporedima komponenti. Editor će se automatski pojaviti čim se otvorи XML fajl sa rasporedom komponenti. Njegove glavne komponente su prikazane na slici 4.27 (izvor: <https://developer.android.com/studio/write/layout-editor.html#intro>):

1. Paleta – lista komponenti (View i ViewGroup) koje se mogu prevući na raspored.
2. Stablo komponenti – prikazuje hijerarhijski pogled na trenutni raspored.
3. Alati – skup opcija za konfiguraciju prikaza rasporeda u editoru.
4. Dizajn editor – raspored prikazan kao dizajn, plan ili oba istovremeno.
5. Atributi – prikaz svih atributa trenutno odabrane komponente, koji se mogu modifikovati po potrebi.



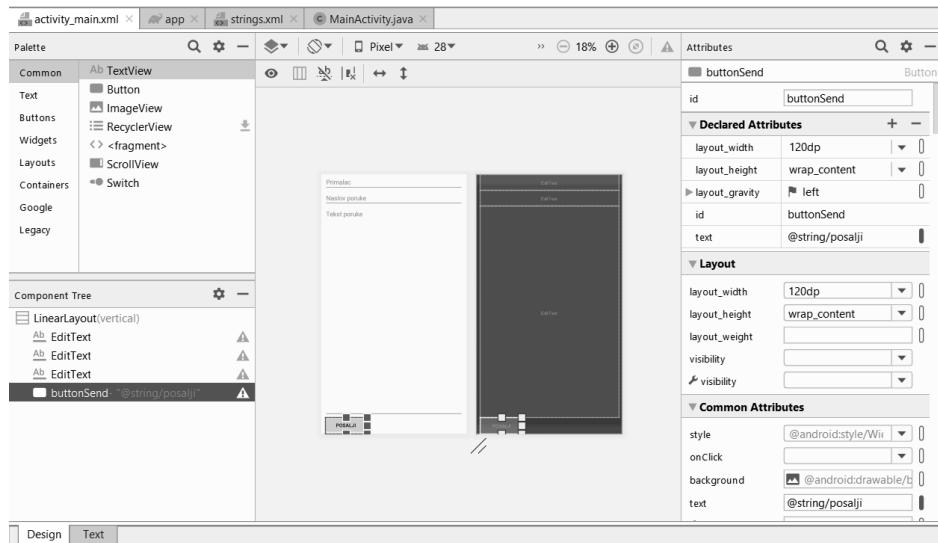
Slika 4.27, Layout Editor - grafički editor za dizajn rasporeda komponenti

Moguće je i dalje editovati XML kod. Ukoliko je potrebno prikazati XML kod, potrebno je odabratи Text tab na dnu prozora. Prozor sa atributima nije vidljiv dok se programer nalazi u tekstualnom editoru XML fajla.

Prilikom dodavanja novog rasporeda za aplikaciju, proces počinje dodavanjem novog XML fajla u res/layout direktorijum. Nakon toga se komponente dodaju prostim prevlačenjem iz palete komponenti. Atributi za svaku komponentu se mogu videti u prozoru Attributes, koji se nalazi sa desne strane editora (slika 4.28).

Atributi su podeljeni u nekoliko sekcija. Sekcija Declared Attributes se odnosi na atribute koji su već specificirani u rasporedu. Sekcija Layout se odnosi na atribute koji se odnose na prikaz komponente, odnosno njenu širinu, visinu, vidljivost i

slično. Nakon toga se nalazi sekcija Common Attributes koja obuhvata izdvojeno najčešće korišćene atributе za taj tip komponente. Na kraju, nalazi se sekcija All Attributes u kojoj su izlistani svi atributi komponente (ima ih jako veliki broj, zbog toga je i izdvojena sekcija Common Attributes gde je programeru koji još uvek nije naviknut na okruženje olakšano pronalaženje najbitnijih atributa). Svaka promena atributa se automatski unosi u XML kod rasporeda na kojem se trenutno radi.



Slika 4.28, podešavanje atributa komponente u grafičkom editoru

Uvođenje grafičkog editora u Android Studio znatno je olakšalo razvoj korisničkog interfejsa i drastično smanjilo potrebu za pisanjem XML fajlova. Međutim, i dalje je potrebno poznavati strukturu XML fajla, posebno zbog činjenice da nisu sve komponente dostupne u paleti. Na primer, DatePicker komponenta se ne nalazi u paleti, pa se njeno dodavanje u raspored mora odraditi ručno kroz XML kod.

4.5. Klasa R

Neki atributi su zajednički za sve komponente (View objekte). To su atributi definisani u baznoj klasi View, i koje nasleđuju sve izvedene klase, od kojih je najbitniji atribut id. To praktično znači da sve komponente grafičkog interfejsa imaju svoj id.

Kada se aplikacija kompajlira, svakom View objektu se dodeljuje jedinstvena celobrojna vrednost koja se koristi za identifikaciju, i smešta u poseban fajl po

imenu R.java. Ovaj fajl se automatski generiše prilikom build-a aplikacije, i smešta ove jedinstvene identifikatore u posebne grupe: drawables, strings, layout itd. Osnovna svrha ove klase je omogućavanje brzog pristupa resursima u projektu. Prilikom dodavanja novih resursa ili brisanja postojećih resursa, R.java fajl će se automatski ažurirati.

Svaka komponenta korisničkog interfejsa je definisana u XML fajlu. Na primer, posmatrajmo dugme definisano sledećim isečkom XML koda:

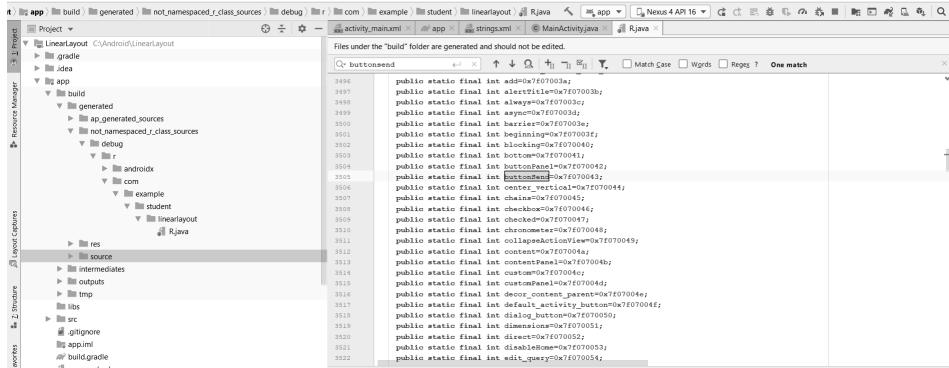
```
<Button  
    android:id="@+id/buttonSend"  
    android:layout_width="120dp"  
    android:layout_height="wrap_content"  
    android:layout_gravity="left"  
    android:text="@string/posalji" />
```

Simbol @ na početku stringa obaveštava XML parser da treba da parsira ostatak Id stringa i da ga identificuje kao Id resursa. Simbol + označava da je ovo novi resurs sa novim imenom koje se mora dodati u resurse aplikacije, odnosno u R.java fajl.

Kada je View jedinstveno identifikovan, može se referencirati iz Java izvornog koda. Na primer, dugme definisano u prethodnom XML kodu ima dodeljen Id "buttonSend", može se dohvatiti iz Java koda pozivom metode findViewById(). Ova metoda locira komponentu preko Id definisanog u XML fajlu, i koristi se na sledeći način:

```
Button button = findViewById(R.id.buttonSend);
```

Svi resursi se mogu adresirati na sličan način. Prilikom kompajliranja, oni se smeštaju u R.java klasu, koja se tipično nalazi u direktorijumu app/build/generated. Ovaj fajl nije preporučljivo ručno menjati. Primer sadržaja R.java klase (sa označenim buttonSend Id) dat je na slici 4.29.



Slika 4.29, R.java klasa, lokacija i njen sadržaj

4.6. Rukovanje događajima

Android nudi više načina za prihvatanje događaja koje korisnik generiše u interakciji sa aplikacijom. Svaki View objekat može generisati događaje koje je moguće kasnije obraditi u kodu aplikacije. Prilikom razvijanja interaktivnosti korisničkog interfejsa, uobičajen pristup je da se prihvate događaji od tačno određene View komponente nad kojom korisnik vrši interakciju.

Ukoliko se detaljnije pogleda kod različitih View klasa od kojih se pravi korisnički interfejs, moguće je primetiti nekoliko callback metoda koje su veoma korisne prilikom obrade korisničkih događaja. Ove metode poziva Android sistem kada se odgovarajuća akcija nad objektom izvrši. Na primer, kada se neki View (poput Button ili EditText) dodirne, biće pozvana `onTouchEvent()` metoda nad tim objektom. Kako bi se omogućilo programeru da reaguje na događaje, klasa View sadrži kolekciju interfejsa sa callback metodama koje se lako mogu implementirati. Ovi interfejsi su poznati pod zajedničkim imenom – osluškivači događaja (engl. *event listeners*),

Osluškivač događaja je dakle interfejs u View klasi koji sadrži jednu callback metodu. Na primer, `OnClick` osluškivač je unutar View klase definisan na sledeći način (preuzeto iz izvornog koda Android platforme):

```
/**  
 * Interface definition for a callback to be invoked  
 * when a view is clicked.  
 */  
public interface OnClickListener {  
    /**  
     * Called when a view has been clicked.  
     *  
     * @param v The view that was clicked.  
     */  
    void onClick(View v);  
}
```

Ovu metodu će pozvati Android Sistem u onom trenutku kada View objekat na kog je osluškivač registrovan dobije odgovarajući događaj od korisnika. Konkretni View objekat koji je generisao događaj se prosleđuje kao parametar metode. Najčešće korišćeni osluškivači, odnosno callback metode su:

- `onClick()` – nalazi se u interfejsu `View.OnClickListener`, poziva se kada korisnik dodirne komponentu.
- `onLongClick()` – nalazi se u interfejsu `View.OnLongClickListener`, poziva se kada korisnik dodirne i zadrži dodir na komponenti.

- onFocusChange() – nalazi se u interfejsu View.OnFocusChangeListener, poziva se kada korisnik navigira do ili od komponente upotrebom tastera za navigaciju.
- onKey() – nalazi se u interfejsu View.OnKeyListener, poziva se kada se korisnik fokusira na komponentu i pritisne/otpusti hardversko dugme.
- onTouch() – nalazi se u interfejsu View.OnTouchListener, poziva se kada korisnik izvrši akciju koja se može okvalifikovati kao dodir, na primer, pritisak na komponentu, otpuštanje komponente ili bilo koji pokret na ekranu unutar granica komponente.

Ove metode su jedini članovi odgovarajućih interfejsa. Ako se posmatraju povratne vrednosti ovih metoda, onClick() na primer je tipa void, odnosno nema povratnu vrednost. Sa druge strane, onLongClick() vraća vrednost tipa boolean, koja označava da li događaj treba da se obrađuje i dalje ili ne. Ako je vraćena vrednost true, time se označava da je događaj u potpunosti obrađen i zaustavlja se svaka dalja obrada. U suprotnom, vrednost false označava da događaj nije obrađen, odnosno da drugi osluškivači treba da preuzmu obradu.

Logika implementacije obrade događaja je slična za sve interfejse. Postoje dva osnovna pristupa pri implementaciji metode i obradi događaja. Oba su pokazana u nastavku, na primeru registrovanja osluškivača za onClick na komponenti tipa Button. Prvi pristup podrazumeva upotrebu anonimne klase, kroz čiju se direktnu definiciju daje i implementacija metode onClick(). Ovaj pristup prikazan je kroz sledeći isečak aktivnosti:

```
Button button = findViewById(R.id.buttonSend);
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // dugme je pritisnuto, ovde treba uraditi nesto
    }
});
```

Često je bolji i fleksibilniji pristup da se interfejs OnClickListener implementira na nivou aktivnosti, čime se izbegava dodatno učitavanje klasa i alokacija objekata. On podrazumeva da aktivnost mora da implementira metodu onClick() nasledenu iz interfejsa, a pri registraciji osluškivača aktivnost može proslediti samu sebe. Ovaj pristup dat je u primeru koda sledeće aktivnosti.

```
public class MainActivity extends AppCompatActivity
implements View.OnClickListener {

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

```
        Button button = findViewById(R.id.buttonSend);
        button.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        // dugmme je kliknuto, ovde treba uraditi nesto
    }
}
```

4.7. Pitanja za vežbu

- Koja je osnovna klasa u hijerarhiji komponenti grafičkog korisničkog interfejsa?
- Šta je dp?
- Šta je gustina ekrana?
- Koji je odnos nasleđivanja imeđu klase View i ViewGroup?
- Kojoj klasi pripadaju svi rasporedi komponenti u Androidu?
- Koji sve različiti rasporedi komponenti postoje?
- Kako se elementi redaju unutar Linear Layout komponente sa vertikalnom orijentacijom?
- Kako se zove komponenta koja se koristi za grupisanje elemenata jedan ispod drugog?
- Kako se sve mogu definisati pozicije komponenti u okviru relativnog rasporeda komponenti?
- Šta je to ograničeni raspored? Kako se u njemu postavljaju komponente?
- Po čemu se ograničeni raspored razlikuje od relativnog?
- Šta je ListView?
- U kom resursnom fajlu se definiše koje su komponente vidljive u korisničkom interfejsu i kakav je njihov raspored?
- Koliko korenih elemenata sme imati resursni fajl koji definiše raspored komponenti?
- Kako se zove funkcija koja se koristi da u Java kodu referencira element iz grafičkog interfejsa?
- Čemu služi atribut id?
- Koji sve još atributi postoje za različite komponente grafičkog interfejsa?
- Kojim atributima se kontrolišu dimenzije komponente?
- Koji osluškivač (listener) se dodeljuje dugmetu kako bi se upravljalo Click događajem nad njim?
- Koji još tipovi osluškivača (listenera) postoje?
- U Android aplikaciji potrebno je prikazati više radio dugmadi od kojih korisnik bira samo jedno. Koju komponentu treba koristiti za grupisanje radio dugmadi?
- Čemu služi klasa R.java u Androidu?
- Koje sve standardne komponente grafičkog interfejsa postoje?
- Šta je Toast?
- Šta je Spinner?
- Šta su adapteri i kako se koriste?
- Koje Picker komponente postoje?
- Kako se koriste dijalozi?

- Kako se zove grafički editor u kome se definiše raspored korisničkog interfejsa?
- Kako se može podesiti koju softversku tastaturu će Android izbaciti kada korisnik uđe u polje za unos teksta? Koje sve tastature postoje?
- Kako se u opštem slučaju rukuje događajima korisničkog interfejsa?

5. Aktivnosti

Activity klasa je ključna komponenta Android aplikacije, a način na koji se aktivnosti pokreću je jedan od osnovnih koncepta modela aplikacije na Android platformi. U klasičnom Java programiranju, aplikacija se pokreće pokretanjem main() metode (koju poziva JVM), dok se u Androidu aktivnost pokreće pozivom jedne od specifičnih callback metoda koje odgovaraju različitim fazama životnog ciklusa aktivnosti.

Mobilne aplikacije se razlikuju od tradicionalnih desktop aplikacija i po tome što mobilna aplikacija ne mora uvek da započne na istom mestu, za razliku od desktop aplikacija koje obično imaju jednu tačno definisani tačku ulaska (main() metoda). Kako bismo ilustrovali ovaj slučaj, posmatrajmo primer mejl aplikacije. Ukoliko se ovoj aplikaciji pristupi sa glavnog ekrana Androida pokreće se standardan korisnički interfejs sa prikazom liste mejlova. Međutim, ukoliko se iz neke druge aplikacije pokrene mejl aplikacija, moguće je da će se direktno otici na ekran za sastavljanje i slanje mejla. Još jedan tipičan primer je da kada direktno otvorimo neku aplikaciju socijalne mreže, videćemo feed sa novostima i statusima naših kontakata. Ukoliko se iz neke druge aplikacije, na primer galerije sa slikama, uradi share neke slike na tu socijalnu mrežu, direktno se ide na share ekran te aplikacije.

Razlog tome je u samom konceptu Android platforme. Kada iz jedne aplikacije zovemo drugu aplikaciju, ne pozivamo zapravo drugu aplikaciju ako atomsku celinu, već pozivamo jednu njenu aktivnost. To znači da aktivnost zapravo služi kao tačka ulaska u aplikaciju.

Najjednostavnije rečeno, aktivnost obezbeđuje jedan prozor po kome aplikacija može da iscrta svoj korisnički interfejs. Tipično, ovaj prozor ispunjava ceo ecran, ali može biti i manji od ekrana. Osnovni princip kreiranja Android aplikacija se svodi na to da jedna aktivnost implementira jedan ecran u aplikaciji. Aplikacija se može sastojati od više ekrana, a u tom slučaju će postojati aktivnost za svaki od pojedinačnih ekrana. Na primer, jedna aktivnost aplikacije može da traži logovanje korisnika na sistem, druga aktivnost može prikazati listu fotografija, treća aktivnost može biti ecran sa podešavanjima aplikacije, a četvrta aktivnost da omogući odabir pojedinačne fotografije.

Jedna aktivnost u aplikaciji se obično označi kao MainActivity (analogno sa main() metodom u tradicionalnim Java programima) – ova aktivnost odgovara prvom ekranu koji se pojavi kada korisnik pokrene aplikaciju. Nakon toga, ova aktivnost može pozvati neku drugu, ta druga aktivnost neku treću, kako bi se izvršile akcije koje korisnik zahteva. Ako se opet vratimo na primer jednostavne mejl aplikacije, MainActivity bi mogao da pruži osnovni ecran koji pokazuje prijemno sanduče sa listom primljenih mejlova. Iz ove aktivnosti, korisnik može

da pokrene druge aktivnosti, kako bi dobio druge ekrane na kojima može da izvrši akcije, poput ekrana za detaljan pregled pojedinačnog mejla ili ekrana za pisanje novog mejla. Aktivnosti jedne aplikacije rade zajedno kako bi formirali tečan i smislen korisnički interfejs, ali pojedinačne aktivnosti su samo labavo povezane jedne sa drugima. Labava sprega znači da postoji vrlo malo zavisnosti između pojedinačnih aktivnosti.

5.1. Konfigurisanje manifesta

Aktivnosti, zajedno sa određenim atributima, moraju biti deklarisane u manifestu aplikacije. Deklaracija se vrši u AndroidManifest.xml fajlu, dodavanjem <activity> elementa kao deteta <application> elementa. Primer je dat sledećim isečkom XML koda AndroidManifest fajla, koji deklariše aktivnost pod nazivom MainActivity:

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

            <category
                android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity
        android:name=".ConfirmationActivity"></activity>
</application>
```

Iz prethodnog primera se vidi da se osim activity elementa koristi još jedan, pod nazivom intent filter (filter namera). Intent filteri su veoma moćan alat Android platforme, koji omogućava da se aktivnost startuje ne samo na osnovu eksplicitnog zahteva, već i na osnovu implicitnog. Eksplicitna namera je kada se sistemu direktno kaže da želimo da odemo u Send Email aktivnost Gmail aplikacije. Dakle, precizno je definisano koja tačno aktivnost treba da se pokrene. Nasuprot tome, implicitni zahtev bi bio da kažemo da želimo da pošaljemo mejl i da tražimo od sistema da nam startuje Send Email ekran bilo koje aktivnosti koja može da nam završi posao.

5.2. Životni ciklus aplikacije

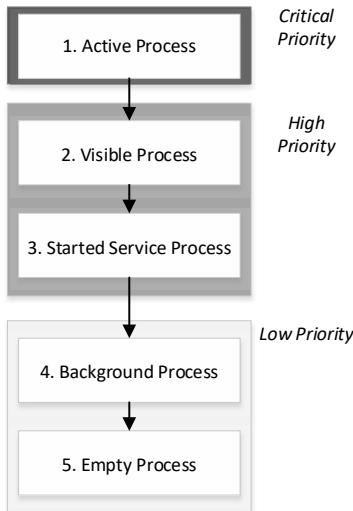
Kako bismo razumeli životni ciklus pojedinačnih aktivnosti, najpre je potrebno shvatiti životni ciklus aplikacije. Ovde prvi put jasno dolazi do izražaja razlika u programiranju klasičnih aplikacija za desktop računare i aplikacija za mobilne uređaje. Mobilni uređaj je veoma ograničen po pitanju resursa – nedovoljno memorije, limitirana procesorska moć, kao i ograničenja po pitanju energije (rad sa baterijom). Implikacija koje programer mora biti svestan u svakom trenutku razvoja aplikacije - kako bi upravljao ograničenim sistemskim resursima na što optimalniji način, Android sistem može da ubije bilo koju aplikaciju koja se izvršava.

Neka se sada posmatra sam način izvršavanja aplikacije pod Android sistemom. U najvećem broju slučajeva, svaka aplikacija se izvršava u svom Linux procesu. Proces se kreira u trenutku kada je potrebno izvršiti neki deo koda aplikacije, i ostaje aktivan sve dok više nije potreban i dok sistemu ne bude potrebno da mu oduzme memoriju kako bi je koristile druge aplikacije. To je jedna od neobičnih ali fundamentalnih osobina Androida – životni ciklus procesa aplikacije ne zavisi od same aplikacije, odnosno sama aplikacija ne može direktno kontrolisati svoj proces. Životni ciklus procesa zavisi od sistema, koji o tome odlučuje na osnovu sledećih nekoliko stavki:

- za koje komponente aplikacije sistem zna da se izvršavaju,
- koliko su te komponente bitne za korisnika, i
- koliko je memorije još dostupno sistemu.

Programeri bi trebalo da razumeju koliko različite komponente aplikacije (aktivnosti, servisi, broadcast receiveri) utiču na životni ciklus samog procesa aplikacije. Ukoliko se komponente ne koriste na adekvatan način, može doći do situacije da sistem ubije proces aplikacije dok ona izvršava neki važan posao.

Odluku o tome koji proces treba ubiti kada sistemu nedostaje memorija Android donosi na osnovu nekoliko pravila. Android najpre pravi hijerarhiju u koju reda procese na osnovu njihove važnosti (slika 5.1). Važnost procesa se određuje na osnovu komponenti koje se u njemu izvršavaju i njihovog stanja.



Slika 5.1, hijerarhija procesa po važnosti

Najviši prioritet imaju aktivni procesi (engl. *foreground process*). To su procesi aplikacije koja je u prvom planu i čije komponente trenutno vrše interakciju sa korisnikom, ili rade nešto što je korisniku trenutno potrebno. Proces se smatra da je u prvom planu ukoliko je ispunjen bilo koji od sledećih uslova:

- Aplikacija ima aktivnost koja je u prvom planu i sa kojom korisnik trenutno vrši interakciju (nakon poziva njene `onResume()` metode).
- Aplikacija ima `BroadcastReceiver` koji se trenutno izvršava (odnosno izvršava se njegova metoda `onReceive()`).
- Aplikacija ima servis koji trenutno izvršava kod u jednoj od svojih `callback` metoda (`onCreate()`, `onStart()`, `onDestroy()`).

U svakom trenutku ima svega nekoliko procesa u celom sistemu koji su u aktivnom stanju, i oni će biti ubijeni samo u krajnjem slučaju da je raspoloživa memorija toliko niska da ne može da održi ni ove procese da se ispravno izvršavaju.

Drugi nivo prioriteta su vidljivi procesi. Vidljiv proces izvršava neki posao i korisnik je svestan toga, pa bi ubijanje ovakvog procesa bilo primetno od strane korisnika i imalo bi negativan uticaj na korisničko iskustvo. Proces se smatra vidljivim u sledećim uslovima:

- Aplikacija izvršava aktivnost koja je vidljiva korisniku na ekranu, ali nije u prvom planu (pozvana je njena `onPause()` metoda). Tipičan primer je ukoliko je aktivnost koja je u prvom planu prikazana kao dijalog koji ne zauzima ceo ekran, već se vidi i deo prethodne aktivnosti iza nje.
- Aplikacija ima servis koji se izvršava u prvom planu.

- Aplikacija ima servis koju sistem koristi za određenu funkcionalnost koje je korisnik svestan, poput live pozadine i slično.

Ovakvi procesi se smatraju za ekstremno važne i sistem ih neće ubiti sve dok ima dovoljno memorije da izvršava sve procese koji su u prvom planu.

Treći nivo prioriteta su servisni procesi, odnosno procesi koji imaju pokrenut servis pozvan sa startService() metodom. Iako ovi procesi nisu direktno vidljivi korisniku, u opštem slučaju rade stvari koje su važne korisniku (preuzimanje podataka preko mreže u pozadini na primer), pa će se sistem truditi da održi ove procese u životu sve dok ima dovoljno memorije da se podrže svi aktivni procesi koji su u prvom planu i svi vidljivi procesi.

Servisima koji se izvršavaju dugo (30 minuta i više) može biti smanjen prioritet kako bi se dozvolilo da njihov proces padne u LRU listi (engl. *Least Recently Used*). Na taj način se izbegava situacija gde dugotrajni servisi sa eventualnim curenjem memorije mogu da zauzmu preveliku količinu RAM-a i ugroze performanse sistema.

Četvrti nivo prioriteta su pozadinski procesi. Pozadinski, odnosno keširan proces je takav proces koji trenutno nije potreban, pa sistem može da ga ubije po želji ukoliko je memorija potrebna drugim procesima. Ovakvi procesi tipično imaju aktivnosti koje nisu trenutno vidljive korisniku (pozvana metoda onStop()). Ukoliko je aktivnost dobro programirana sa ispravnim ponašanjem onStop() metode, ubijanje ovakvog procesa neće uticati na korisnika kada se vrati u aplikaciju, jer će aktivnost prilikom ponovnog pokretanja moći da restaurira svoje stanje. U normalno održavanom sistemu, samo ovi procesi ulaze u razmatranje za ubijanje. Dobro održavan sistem ima uvek nekoliko dostupnih keširanih procesa (kako bi se obezbedilo efikasno prelaženje između aplikacija), a redovno će ubijati najstarije keširane procese. Samo u slučaju kritičnih situacija sistem može doći do tačke da su svi keširani procesi ubijeni, pri čemu i dalje nema dovoljno memorije pa mora početi da ubija i servisne procese. Ovakvi procesi se drže u LRU listi, gde će poslednji proces u listi biti prvi ubijen kako bi se oslobođila memorija.

Najniži nivo prioriteta imaju prazni procesi – to su aplikacije bez i jedne aktivne komponente, na samom kraju svog životnog ciklusa.

Sada, kada je poznat životni ciklus same aplikacije, može se detaljno obratiti pažnja na životni ciklus pojedinačnih aktivnosti, koji je opisan u nastavku.

5.3. Životni ciklus aktivnosti

Tokom svog životnog ciklusa, svaka aktivnost prolazi kroz određeni broj stanja. Korisnik aplikacije može da se kreće kroz aplikaciju, da izade iz nje, da se ponovo vrati i slično. Za to vreme, aktivnosti aplikacije prolaze kroz različita stanja u svom životnom ciklusu. Activity klasa nudi nekoliko callback metoda koje omogućavaju aktivnostima da zna da joj se stanje promenilo i da po potrebi reaguje na tranziciju. Promene od značaja su kada sistem pokreće aktivnost, stopira aktivnost, nastavlja aktivnost, ili uništava proces kome aktivnost pripada.

Programer može da se „zakači“ na callback metode koje aktivnost pruža i definiše kako će se aktivnost ponašati u slučajevima kada korisnik aplikacije izade iz aktivnosti ili ponovo uđe u nju. Na primer, ukoliko aktivnost strimuje video sa Interneta, trebalo bi pauzirati video i isključiti mrežnu konekciju ukoliko se korisnik prebacuje na drugu aplikaciju. Kada se korisnik vrati u aplikaciju, potrebno je ponovo se povezati na mrežu i nastaviti video sa mesta gde je pauziran. Slično, ukoliko aplikacija prikazuje interaktivni grafički sadržaj (poput mobilnih igara), nema smisla ažurirati grafički displej ukoliko se korisnik prebacuje na drugu aplikaciju. Kada se korisnik vrati, potrebno je ponovo početi sa ažuriranjem interfejsa. Svaka callback metoda dozvoljava da se izvrši akcija koja je odgovarajuća datoj promeni stanja. Dobra aplikacija se svodi na to da je potrebno uraditi pravu akciju u pravo vreme kako bi se tranzicije između stanja izvršile korektno, čime i aplikacija postaje robusnija i generalno poboljšava svoje performanse. Ukoliko tranzicije nisu realizovane korektno, aplikacija se može susresti sa čitavim nizom problema, čime se drastično smanjuje njena upotrebljivost i snižava zadovoljstvo korisnika:

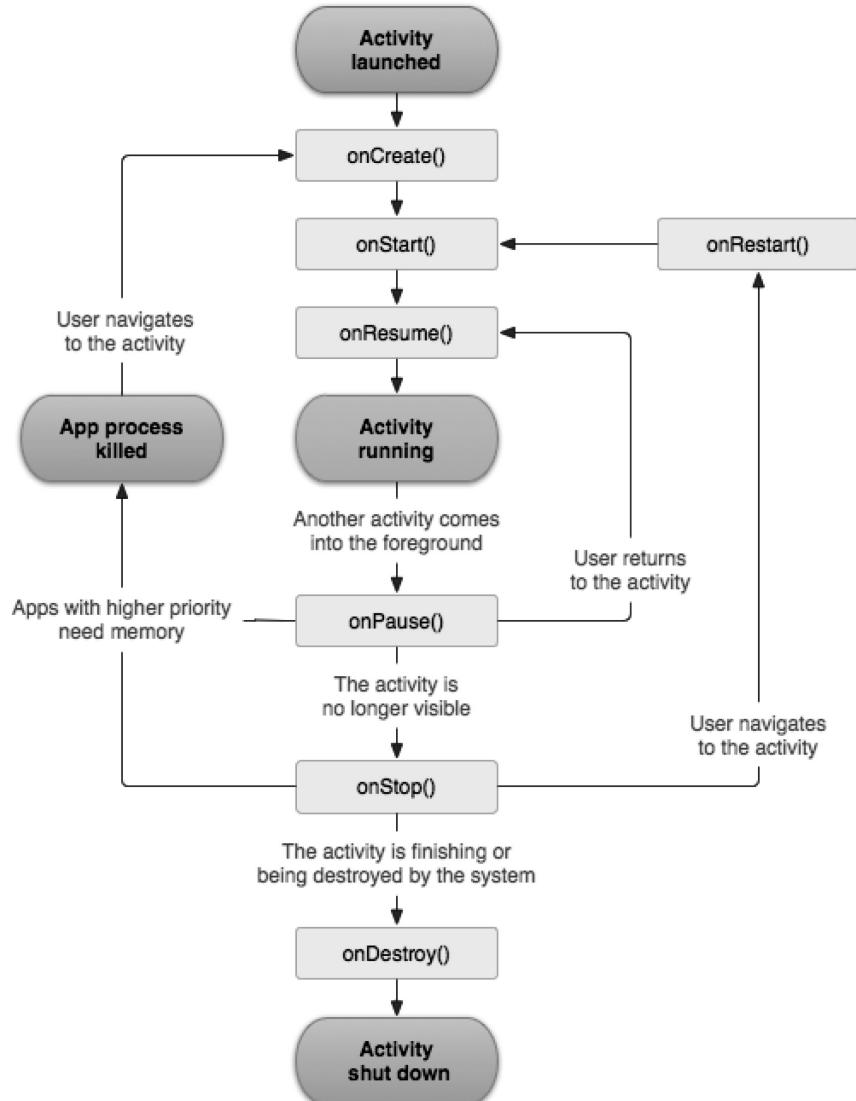
- Nasilni prekid rada aplikacije (engl. *crash*) ukoliko korisnik tokom korišćenja aplikacije primi telefonski poziv ili se prebacuje na drugu aplikaciju.
- Nerazumno trošenje vrednih i veoma ograničenih sistemskih resursa u trenutku dok korisnik ne koristi aktivno aplikaciju.
- Gubitak korisničkih podataka ukoliko korisnik napusti aplikaciju i kasnije je ponovo otvoriti.
- Nasilni prekid rada aplikacije ili gubitak korisničkih podataka ukoliko se ekran zarotira.

Klasa Activity nudi šest callback metoda kako bi omogućila korektnu tranziciju između stanja životnog ciklusa aktivnosti. Ove metode poziva Android sistem kada aktivnost pređe u novo stanje, i to su redom:

- `onCreate()`,
- `onStart()`,
- `onResume()`,

- onPause(),
- onStop(), i
- onDestroy.

Ovaj koncept prikazan je na slici 5.2, na kojoj su označena stanja i callback metode (izvor: <https://developer.android.com/guide/components/activities/activity-lifecycle>).



Slika 5.2, životni ciklus aktivnosti sa callback metodama

U zavisnosti od aplikacije koja se razvija i od same složenosti aktivnosti zavisi koje callback metode je potrebno implementirati. Najčešće nije potrebno implementirati svih šest metoda za svaku aktivnost, ali potrebno je razumeti ponašanje svake od ovih metoda i implementirati odgovarajuće kako bi se aktivnost ponašala onako kako korisnik aplikacije i očekuje. Dodatno, ove callback metode ne poziva programer, već ih sve okida Android sistem prilikom tranzicije između različitih stanja u kojima se aktivnost može naći.

Ključni aspekt koji je potrebno razumeti jeste trenutak kada korisnik počne da napušta aktivnost (na primer, otvorio novu aplikaciju). U tom trenutku Android sistem počinje da poziva metode koje vrše postepenu demontažu aktivnosti. Ponekad je ova demontaža samo privremena, jer kada korisnik otvorio novu aplikaciju, prethodna aktivnost i dalje postoji u memoriji, a korisnik je može lako ponovo vratiti u prvi plan. U tom slučaju aktivnost treba da nastavi tamo gde je stala pre gubitka fokusa.

onCreate()

Prva od metoda, koja je obavezni deo aktivnosti koji se mora implementirati, je `onCreate()` metoda. Ova metoda se okida u trenutku kada sistem kreira aktivnost. Tipični zadaci koji se obavljaju u ovoj metodi su inicijalizacija komponenti koje su potrebne aktivnosti, kreira se korisnički interfejs, povezuju podaci sa listama i slično. Time aktivnost prelazi u stanje `Created`. Aktivnost se ne zadržava dugo u ovom stanju. Čim se metoda `onCreate()` izvrši do kraja, aktivnost prelazi u stanje `Started`, a sistem poziva callback metode `onStart()` i `onResume()` jednu za drugom.

onStart()

Prilikom prelaska iz stanja `Created` u stanje `Started`, sistem poziva ovu metodu. Zadatak `onStart()` metode je da učini aktivnost vidljivom korisniku, jer se u ovom trenutku aktivnost prebacuje u prvi plan i postaje interaktivna. Tipična akcija koja se izvršava u ovoj metodi je inicijalizacija koda koji treba da opslužuje korisnički interfejs. Ova metoda se izvršava jako brzo, a aktivnost ne ostaje dugo u stanju `Started`. Odmah nakon završetka metode `onStart()` aktivnost prelazi u stanje `Resumed`, a sistem okida metodu `onResume()`.

onResume()

Prilikom ulaska u stanje `Resumed`, aktivnost dolazi u prvi plan, a sistem okida `onResume()` metodu. U ovom stanju aktivnost je vidljiva i interaktivna, odnosno korisnik može da vrši različite akcije nad komponentama korisničkog interfejsa te aktivnosti. Aktivnost ostaje u ovom stanju sve dok se ne desi nešto što će oduzeti fokus aplikaciji, poput dolazećeg poziva, pokretanja druge aplikacije ili gašenja ekrana. Kada se ovakav događaj desi, aktivnost prelazi u stanje `Paused`, a sistem okida `onPause()` metodu. Ukoliko se iz stanja `Paused` aktivnost vrati u

stanje Resumed, sistem ponovo poziva onResume() metodu. Zbog toga, onResume() treba da inicijalizuje one komponente koje su oslobođene u okviru onPause() metode.

onPause()

Poziv ove metode je prva indikacija da korisnik napušta ovu aktivnost. To znači da aktivnost više nije u prvom planu (iako je možda još uvek vidljiva, na primer, ukoliko je korisnik uključio režim rada sa više prozora). Tipična upotreba onPause() metode uključuje pauziranje ili prilagođavanje operacija koje nije potrebno nastaviti dokle god je aktivnost pauzirana. Aktivnost može biti pauzirana iz više razloga:

- Nešto je oduzelo fokus aplikaciji, poput dolazećeg poziva, pokretanja druge aplikacije ili gašenja ekrana – ovo je i najčešći razlog.
- Od Android 7.0 (API nivo 24) pa na dalje, više aplikacija se može istovremeno izvršavati u režimu rada sa više prozora. Pošto samo jedan od ovih prozora može imati fokus u bilo kom trenutku, ostale aplikacije su pauzirane.
- Nova aktivnost je delimično prekrila ekran, poput otvorenog dijaloga. Sve dok je aktivnost vidljiva, ali nije u fokusu, ona ostaje pauzirana.

Metoda onPause() obično otpušta sistemske resurse (poput senzora ili GPS), kao i sve ono što može negativno uticati na životni vek baterije dok je aktivnost pauzirana, odnosno dok korisniku ne treba. Međutim, pošto aktivnost u pauziranom stanju može i dalje biti vidljiva delimično ili u celosti (naročito ukoliko se koristi režim rada sa više prozora), ovde ne treba otpustiti resurse koji se odnose na korisnički interfejs, već ga i dalje treba ažurirati (to treba uraditi u onStop() metodi). Dodatno, pošto je vreme izvršavanja metode onPause() veoma kratko, preporučeno je da se u ovoj metodi ne čuvaju korisnički podaci, ne prave mrežni pozivi ili izvršavaju operacije nad bazom, pošto se možda neće izvršiti do kraja pre završetka metode. Ovakve dugotrajne operacije se izvršavaju u onStop() metodi.

Aktivnost ostaje u ovom stanju i nakon završetka onPause() metode, sve dok se aktivnost ne vrati u prvi plan, ili dok ne postane potpuno nevidljiva korisniku. Ukoliko se aktivnost nastavi i vrati u prvi plan, sistem ponovo okida onResume() metodu, a aktivnost prelazi u stanje Resumed. Sa druge strane, ukoliko aktivnost postane potpuno nevidljiva, sistem poziva onStop() metodu.

onStop()

Kada aktivnost više nije vidljiva korisniku, ona prelazi u Stopped stanje, a sistem okida metodu onStop(). To je slučaj kada je nova aktivnost prekrila ceo ecran. Sistem može pozvati ovu metodu i kada je aktivnost završila sa izvršavanjem i treba da se uništi. U ovoj metodi aktivnost treba da otpusti ili prilagodi resurse

koji nisu potrebni dok aktivnost nije vidljiva. Tipično, treba obustaviti ažuriranje korisničkog interfejsa (pošto korisnik ne vidi aktivnost, nema smisla ažurirati ekran) i izvršiti sve dugotrajne operacije poput snimanja podataka u bazu.

Aktivnost koja je stopirana se i dalje drži u memoriji, čuva svoje stanje i informacije. Ukoliko se aktivnost nastavi, svi podaci su i dalje tu. U tom slučaju, poziva se metoda `onRestart()`. Međutim, kada je aktivnost stopirana, sistem može uništiti proces kome pripada u bilo kom trenutku, ukoliko je potrebna memorija. Ako je aktivnost završila sa izvršavanjem, sistem poziva `onDestroy()` metodu.

onDestroy()

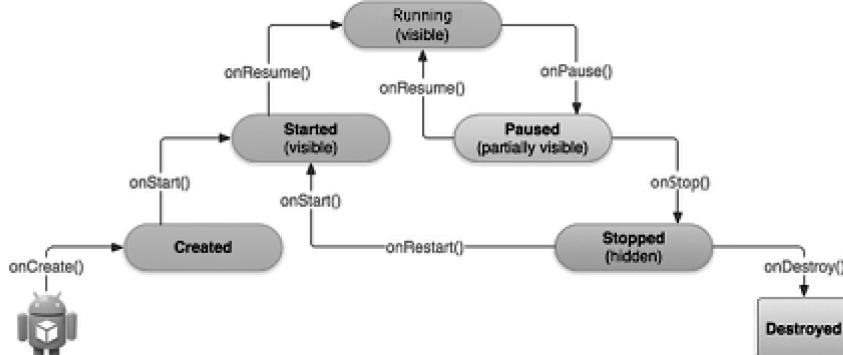
Sistem poziva ovu metodu neposredno pre uništavanja aktivnosti. To se može desiti u više slučajeva:

- Aktivnost se završila (korisnik je odbacio aktivnost ili je pozvana njena `finish()` metoda).
- Sistem je privremeno uništio aktivnost zbog promene u konfiguraciji, poput rotacije ekrana ili ulaska u režim rada sa više prozora.

U tom trenutku aktivnost treba da počisti za sobom i otpusti sve ostale resurse pre nego što bude uništена.

Najlakši način za razumevanje životnog ciklusa aktivnosti je kroz dijagram stanja sa prikazanim metodama koje se pozivaju prilikom prelaska iz jednog stanja u drugo (slika 5.3, preuzeta sa:

<https://developer.android.com/guide/components/activities/activity-lifecycle>).



Slika 5.3, dijagram stanja u kojima se aktivnost može naći

Kako bi se dodatno ilustrovali i demonstrirali pozivi callback metoda, neka se posmatra jednostavna aktivnost koja ima implementirane sve navedene callback metode iz životnog ciklusa aktivnosti. Log poruke se generišu pomoću `Log.d()`

metode, kako bi se ključni pozivi metoda ispisali u logu. Kod aktivnosti je dat na sledeći način:

```
package com.example.student.myapplication;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;

public class MainActivity extends Activity {

    String msg = "Android : ";

    /** Poziva se prilikom prvog kreiranja aktivnosti */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Log.d(msg, "Poziv onCreate() metode");
    }

    /** Poziva se kada aktivnost treba da postane vidljiva */
    @Override
    protected void onStart() {
        super.onStart();
        Log.d(msg, "Poziv onStart() metode");
    }
    /** Poziva se kada je aktivnost postala vidljiva */
    @Override
    protected void onResume() {
        super.onResume();
        Log.d(msg, "Poziv onResume() metode");
    }

    /** Poziva se kada aktivnost pocne da gubi fokus */
    @Override
    protected void onPause() {
        super.onPause();
        Log.d(msg, "Poziv onPause() metode");
    }

    /** Poziva se kada aktivnost vise nije vidljiva */
    @Override
    protected void onStop() {
        super.onStop();
        Log.d(msg, "Poziv onStop() metode");
    }
}
```

```

/** Poziva se neposredno pre unistavanja aktivnosti */
@Override
public void onDestroy() {
    super.onDestroy();
    Log.d(msg, "Poziv onDestroy() metode");
}
}

```

Nakon pokretanja aplikacije na emulatoru i pokretanja aplikacije, u sekciji LogCat se može videti sledeći redosled poziva metoda:

```

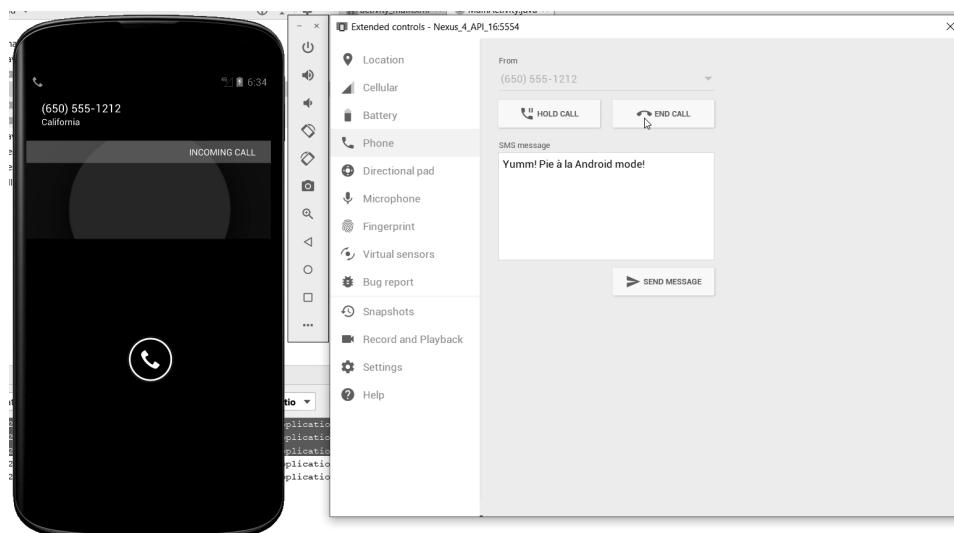
01-25 18:22:55.122 2362-
2362/com.example.student.myapplication D/Android :: Poziv
onCreate() metode

01-25 18:22:55.132 2362-
2362/com.example.student.myapplication D/Android :: Poziv
onStart() metode

01-25 18:22:55.132 2362-
2362/com.example.student.myapplication D/Android :: Poziv
onResume() metode

```

To je i očekivan redosled, pošto je tako definisano u životnom ciklusu aktivnosti. Posmatrajmo sada situaciju da u tom trenutku, dok je posmatrana aktivnost u prvom planu, stigne telefonski poziv na uređaj, što se može simulirati upotrebom emulatora kao što je prikazano na slici 5.4.



Slika 5.4, primer aktivnosti koja gubi fokus usled dolaznog poziva

U logu se u tom trenutku mogu videti sledeća dva poziva:

```
01-25 18:33:58.672 2362-
2362/com.example.student.myapplication D/Android :: Poziv
onPause() metode

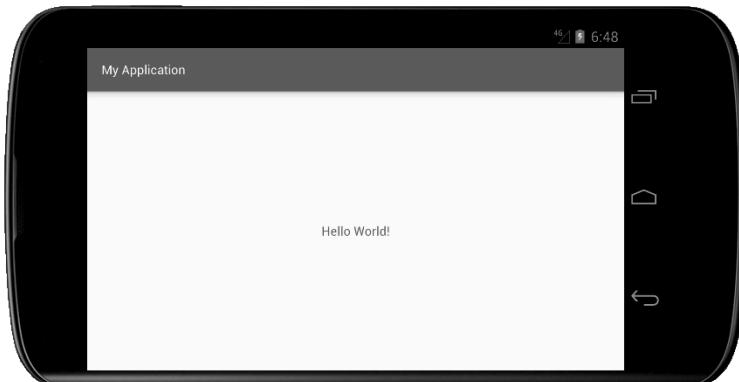
01-25 18:33:59.342 2362-
2362/com.example.student.myapplication D/Android :: Poziv
onStop() metode
```

U tom slučaju, pošto aktivnost gubi fokus jer telefonska aplikacija izlazi u prvi plan, pozivaju se prvo onPause(), a odmah nakon toga i onStop() metoda. Nakon završetka poziva, telefonska aplikacija se uklanja, a posmatrana aktivnost ponovo dolazi u prvi plan. Očekivano, nakon prekida poziva, u logu se ispisuju sledeća dva poziva (prvo poziv onStart() metode, a zatim i onResume() metode):

```
01-25 18:37:30.132 2362-
2362/com.example.student.myapplication D/Android :: Poziv
onStart() metode

01-25 18:37:30.132 2362-
2362/com.example.student.myapplication D/Android :: Poziv
onResume() metode
```

Interesantno je obratiti pažnju na to što se dešava u slučaju da korisnik promeni orijentaciju telefona, tj. ukoliko pređe iz standardnog vertikalnog položaja (engl. *Portrait*) u horizontalni položaj (engl. *Landscape*), kao što je prikazano na slici 5.5. U tom slučaju, potrebno je znati kako se Android sistem ponaša. Aktivnost koja se trenutno izvršava se restartuje, odnosno uništava i ponovo kreira (u logu će se videti novi poziv onCreate() metode). Ovakvo ponašanje je očekivano i implementirano namerno, kako bi se pomoglo aplikaciji da se adaptira na novu konfiguraciju ekrana automatskim restartom. Međutim, ukoliko programer ne očekuje ovakvo ponašanje, može doći do gubitka nekih korisničkih podataka koje sadrži aktivnost. Poželjno je koristiti neki oblik perzistencije kako bi se očuvalo stanje aktivnosti.



Slika 5.5, uticaj promene rotacije uređaja na stanje aktivnosti

Ukoliko se zarotira ekran, u logu su ispisani sledeći pozivi:

```
01-25 19:24:07.882 2676-
2676/com.example.student.myapplication D/Android :: Poziv
onPause() metode

01-25 19:24:07.882 2676-
2676/com.example.student.myapplication D/Android :: Poziv
onStop() metode

01-25 19:24:07.882 2676-
2676/com.example.student.myapplication D/Android :: Poziv
onDestroy() metode

01-25 19:24:07.942 2676-
2676/com.example.student.myapplication D/Android :: Poziv
onCreate() metode

01-25 19:24:07.942 2676-
2676/com.example.student.myapplication D/Android :: Poziv
onStart() metode

01-25 19:24:07.942 2676-
2676/com.example.student.myapplication D/Android :: Poziv
onResume() metode
```

Ukoliko korisnik izađe iz aktivnosti pritiskom na dugme Back, aktivnost se uništava, što se vidi u sledećim pozivima koji su ispisani u logu:

```
01-25 19:27:47.542 2676-
2676/com.example.student.myapplication D/Android :: Poziv
onPause() metode

01-25 19:27:48.102 2676-
2676/com.example.student.myapplication D/Android :: Poziv
onStop() metode

01-25 19:27:48.102 2676-
2676/com.example.student.myapplication D/Android :: Poziv
onDestroy() metode
```

5.4. Čuvanje stanja korisničkog interfejsa

U trenutku kada aktivnost počinje da prelazi u stopirano stanje, sistem automatski poziva metodu onSaveInstanceState(), čime se aktivnosti pruža šansa da sačuva informacije o svom stanju. To je neophodno jer ne postoji garancije da će aktivnost koja je stopirana ponovo biti pozvana u prvi plan, već postoji dobre šanse da će je Android sistem nakon određenog vremena uništiti. Podrazumevana implementacija ove metode čuva privremene podatke o stanju komponenti u

hijerarhiji ekrana, poput informacije o tekstu unetom u EditText komponentu, ili poziciju klizača (scroll) u ListView rasporedu. Ukoliko je potrebno sačuvati bilo koje dodatne podatke o stanju aktivnosti, mora se nadjačati onSaveInstanceState(). U nadjačanoj metodi treba dodati parove ključ-vrednost u Bundle objekat koji se čuva u slučaju da se aktivnost neočekivano uništi. Bundle objekti se često upotrebljavaju u Android programiranju, tipično za prenos podataka između različitih aktivnosti. Po implementaciji Bundle veoma liči na Java Map objekat, koji mapira String ključeve u vrednosti. Razlog zašto Android ne koristi stare Map objekte je da su mape previše fleksibilne i mogu da sadrže objekte koji ne mogu biti serijalizovani. Bundle ograničava tipove objekata koji se mogu dodati u Bundle objekat tako da je sadržaj Bundle objekta garantovano serijalizabilan.

Prilikom nadjačavanja metode onSaveInstanceState(), mora se pozvati verzija iz bazne klase ukoliko se želi i podrazumevano ponašanje, odnosno čuvanje stanja hijerarhije komponenti. Dodatna napomena se odnosi na sam Bundle objekat, koji je prihvatljivo rešenje samo u slučaju da je potrebno sačuvati veoma mali skup podataka (zbog toga što zahteva serijalizaciju i izvršava se u glavnoj niti aplikacije). Za čuvanje podataka većeg obima postoje drugi oblici persistencije, o kojima će biti reči kasnije.

Neka se kao primer posmatra aktivnost neke mobilne igre, u kojoj je neophodno sačuvati i podatke o trenutno ostvarenim poenima igrača i o njegovom nivou. U tom slučaju, potrebno je nadjačati onSaveInstanceState() metodu i u njoj sačuvati potrebne informacije u Bundle objektu na sledeći način:

```
static final String STANJE_POENI = "stanjePoeni";
static final String STANJE_NIVO = "stanjeNivo";

// ...

@Override
public void onSaveInstanceState(Bundle savedInstanceState)
{
    // Sacuvati trenutno stanje igre
    savedInstanceState.putInt(STANJE_POENI, trenutniPoeni);
    savedInstanceState.putInt(STANJE_NIVO, trenutniNivo);
    // Pozvati verziju iz bazne klase da sacuva hijerarhiju
    komponenti
    super.onSaveInstanceState(savedInstanceState);
}
```

Prilikom pokretanja aktivnosti nakon što je prethodno uništena, njen stanje se može restaurirati iz sačuvanog Bundle objekta koji će sistem proslediti aktivnosti. Dve callback metode prihvataju isti Bundle objekat koji sadrži informacije o prethodnom stanju:

- `onCreate()`,
- `onRestoreInstanceState()`.

Pošto se `onCreate()` metoda poziva i u slučaju da sistem kreira novu instancu aktivnosti i u slučaju da restaurira prethodno uništenu, pre pokušaja čitanja potrebno je prvo proveriti da li je Bundle objekat null. Ukoliko je null, to znači da sistem kreira novu instancu aktivnosti, i da nema šta da se restaurira.

Ako se posmatra prethodni primer mobilne igre gde se čuvaju trenutno ostvareni poeni i nivo korisnika, implementacija u slučaju da se restauracija radi kroz `onCreate()` metodu data je sledećim isečkom Java koda:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState); // Uvek pozvati prvo
metodu iz bazne klase

    // Proveriti da li restauriramo prethodno unistenu
instancu ili pravimo potpuno novu instancu
    if (savedInstanceState != null) {
        // Restaurirati sacuvane vrednosti iz Bundle objekta
        trenutniPoeni =
savedInstanceState.getInt(STANJE_POENI);
        trenutniNivo =
savedInstanceState.getInt(STANJE_NIVO);
    } else {
        // Kreira se nova instanca, potrebno je
inicijalizovati promenljive na pocetne vrednosti...
    }
    // ...
}
```

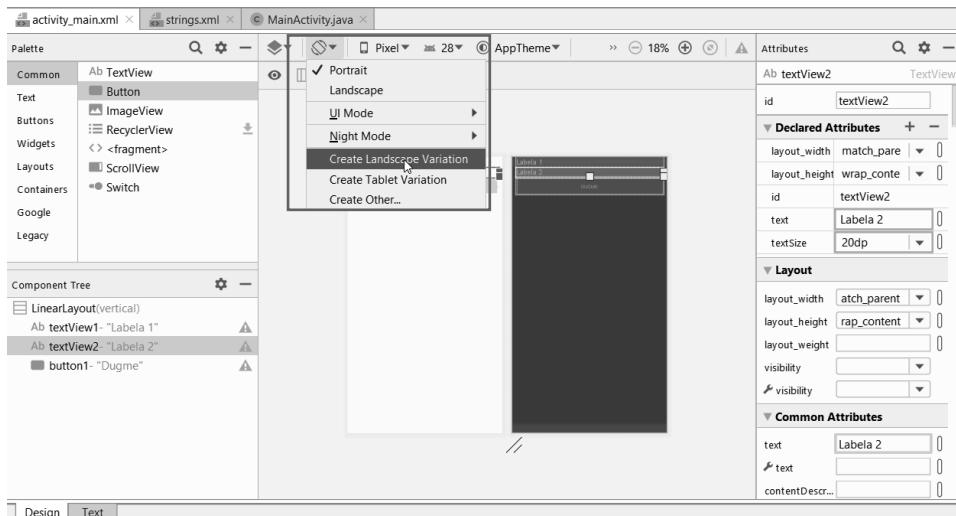
Uместо restauracije tokom metode `onCreate()`, može se implementirati callback `onRestoreInstanceState()` metoda, koju Android sistem poziva nakon `onStart()` metode samo u slučaju da postoji sačuvano stanje koje treba restaurirati. Samim tim, nije potrebno proveravati da li je Bundle objekat null. Ovakva implementacija data je sledećim isečkom Java koda:

```
public void onRestoreInstanceState(Bundle
savedInstanceState) {
    // Prvo pozvati verziju iz bazne klase da restaurira
hijerarhiju komponenti
    super.onRestoreInstanceState(savedInstanceState);
    // Restaurirati sacuvane vrednosti iz Bundle objekta
    trenutniPoeni =
savedInstanceState.getInt(STANJE_POENI);
    trenutniNivo = savedInstanceState.getInt(STANJE_NIVO);
}
```

5.5. Promena orijentacije ekrana

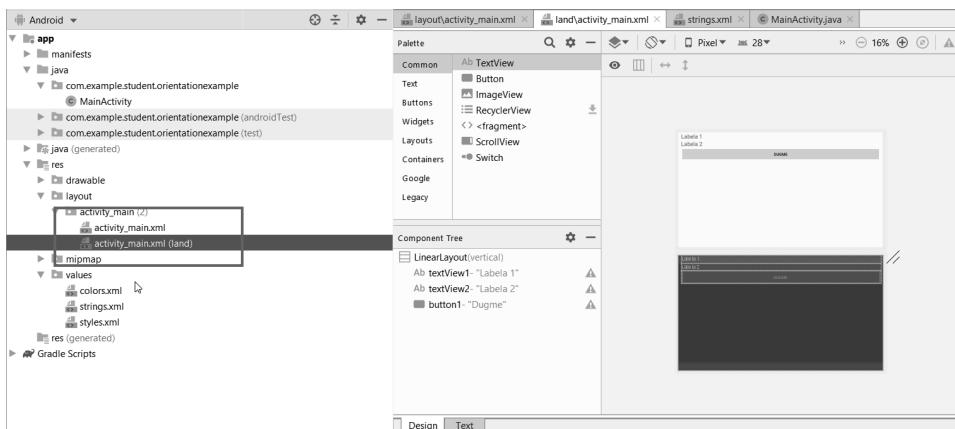
Konfiguracija uređaja se može promeniti u toku izvršavanja aplikacije, a tipični ovakvi primeri su promena orijentacije uređaja ili ulazak u režim rada sa više prozora. U slučaju da se ovakva situacija desi, Android restartuje aktivnost koja se trenutno izvršava tako što poziva prvo `onDestroy()`, a zatim i `onCreate()` metode. Ovakvo ponašanje je dizajnirano sa ciljem da se aplikaciji pomogne da se adaptira na novu konfiguraciju i automatski ponovo učita alternativne resurse koji odgovaraju novoj konfiguraciji. Kada se posmatra orijentacija ekrana, postoje Portrait orijentacija (standardni prikaz, kada se telefon drži vertikalno) i Landscape orijentacija (telefon u vodoravnom položaju).

Android sistem će prilikom promene orijentacije uređaja iz Portrait u Landscape režim sam iscrtati korisnički interfejs po pravilima koja su zadata u XML za podrazumevani (Portrait) režim. Ponekad to nije ono što programer želi, već je potrebno iscrtati drugačiji raspored komponenti. Alternativni resursi za Portrait režim se dodaje jednostavno u vizuelnom editoru odabirom opcije Create Landscape Variation, kao što je prikazano na slici 5.6.



Slika 5.6, dodavanje alternativnog prikaza za Landscape režim

Nakon dodavanja alternativnog prikaza za Landscape režim, biće dodat još jedan XML fajl sa rasporedom komponenti u okviru `res/layout` foldera, koji će imati isto ime kao i podrazumevani fajl za Portrait režim, uz oznaku (`land`), kao što je prikazano na slici 5.7.



Slika 5.7, dodatni XML fajl za raspored komponenti u Landscape režimu

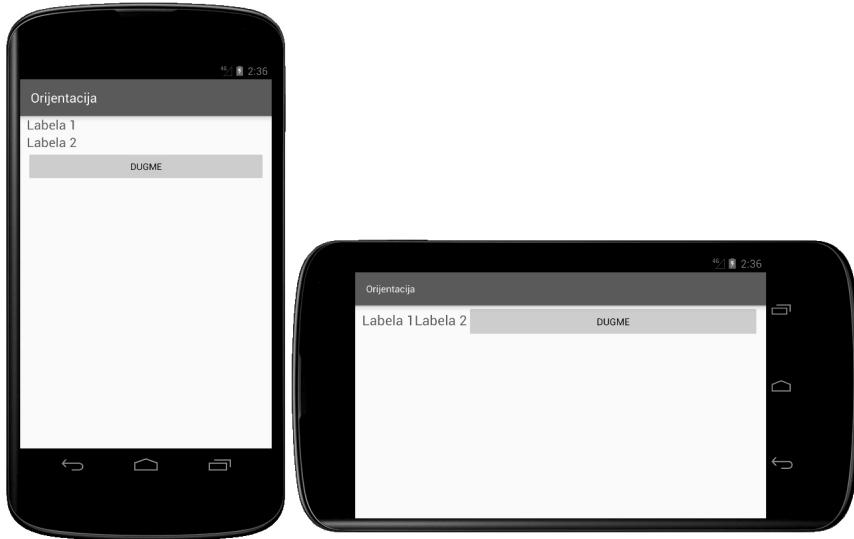
Neka se posmatra sledeći primer, gde raspored komponenti u Portrait režimu koristi linearni vertikalni raspored za prikaz komponenti. Neka je XML kod fajla activity_main.xml dat sa:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="10dp"
    android:paddingRight="10dp"
    android:orientation="vertical" >
    <TextView
        android:id="@+id/textView1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Labela 1"
        android:textSize="20dp" />
    <TextView
        android:id="@+id/textView2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Labela 2"
        android:textSize="20dp" />
    <Button
        android:id="@+id/button1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Dugme" />
</LinearLayout>
```

Pri prelasku u Landscape režim, ne želimo da se koristi isti raspored komponenti, već je potrebno da se koristi linearni horizontalni raspored. U tom slučaju, dodaje se još jedan XML fajl koji odgovara Landscape orijentaciji uređaja (activity_main.xml (land)), sa istim komponentama, ali organizovanim u drugačijem rasporedu, i njegov sadržaj dat je u nastavku:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="10dp"
    android:paddingRight="10dp"
    android:orientation="horizontal" >
    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Labela 1"
        android:textSize="20dp" />
    <TextView
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Labela 2"
        android:textSize="20dp" />
    <Button
        android:id="@+id/button1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Dugme" />
</LinearLayout>
```

Kada su XML fajlovi ovako definisani, Android sistem će prilikom rotacije uređaja restartovati aktivnost, koja će nakon toga tokom novog izvršavanja svoje onCreate() metode učitati Landscape XML fajl. Prikaz korisničkog interfejsa definisanog na ovaj način, u oba režima rada, prikazan je na slici 5.8.



Slika 5.8, korisnički interfejs aplikacije sa odvojenim rasporedima za Portrait i Landscape režim

Na promenu konfiguracije uređaja moguće je reagovati i iz Java koda odgovarajuće aktivnosti. Prilikom promene konfiguracije, Android sistem generiše događaj koji je moguće uhvatiti callback metodom `onConfigurationChange` u kodu aktivnosti i samostalno reagovati na promenu koja se desila. Ovoj metodi se kao parametar prosleđuje objekat tipa `Configuration` koji specificira novu konfiguraciju uređaja. Čitanjem odgovarajućih polja iz `Configuration` objekta, moguće je odrediti šta je promenjeno i reagovati ažuriranjem resursa koji se koriste u korisničkom interfejsu. Ipak, treba biti oprezan jer samostalno rukovanje promenama u konfiguraciji može drastično zakomplikovati i otežati upotrebu alternativnih resursa, zato što ih sistem u tom slučaju neće automatski primeniti umesto programera. Ova tehnika nije preporučena od strane Android zajednice.

Ukoliko je ipak potrebno samostalno rukovati promenama orijentacije uređaja u aktivnosti, mora se u manifestu aplikacije za tu aktivnost dodati atribut `android:configChanges`, i deklarisati sledeće vrednosti: "orientation", "screenSize" i "screenLayout". Na primer, sadržaj manifesta u slučaju da aktivnost treba da reaguje na promenu konfiguracije i u slučaju promene orijentacije, kao i u slučaju promene statusa dostupnosti tastature ("keyboardHidden" vrednost) bi bio sledeći:

```
<activity android:name=".MainActivity"
    android:configChanges=
"orientation|screenSize|screenLayout|keyboardHidden"
    android:label="@string/app_name">
```

U ovom slučaju, ukoliko se bilo koja od ovih promena konfiguracije desi, MainActivity se neće automatski restartovati, već će biti pozvana callback metoda onConfigurationChanged() u okviru MainActivity klase. Primer hvatanja ovog poziva u telu MainActivity, koji nakon toga proverava trenutnu orijentaciju uređaja dat je u nastavku:

```
@Override  
public void onConfigurationChanged(Configuration newConfig)  
{  
    super.onConfigurationChanged(newConfig);  
  
    // Provera orijentacije ekrana  
    if (newConfig.orientation ==  
        Configuration.ORIENTATION_LANDSCAPE) {  
        Toast.makeText(this, "landscape",  
        Toast.LENGTH_SHORT).show();  
    } else if (newConfig.orientation ==  
        Configuration.ORIENTATION_PORTRAIT) {  
        Toast.makeText(this, "portrait",  
        Toast.LENGTH_SHORT).show();  
    }  
}
```

5.6. Intenti (namere)

Skoro svaka Android aplikacija se sastoji od više aktivnosti. Tokom rada aplikacije, vrlo je verovatno da će korisnik da ulazi i izlazi iz jedne aktivnosti više puta, kao i da će biti potrebno da se iz jedne aktivnosti pokrene neka druga. To je slučaj kada aplikacija treba da pređe sa ekrana na kome se trenutno korisnik nalazi na novi ekran. Pokretanje nove aktivnosti se može postići pozivom jedne od dve moguće metode:

- startActivity(), ukoliko nova aktivnost ne treba da vratи rezultat,
- startActivityForResult(), ukoliko nova aktivnost treba da vratи rezultat.

Obe metode prihvataju kao argument objekat tipa Intent (engl. namera). Ovaj objekat definise ili tačno određenu aktivnost koja treba da se pokrene, ili tip akcije koja se želi izvršiti (pri čemu će sistem odrediti odgovarajuću aktivnost, koja može pripadati i nekoj drugoj aplikaciji). Osim toga, Intent objekat može da se iskoristi da se novoj aktivnosti prosledi manja količina podataka koji su joj potrebni.

Kada jedna aktivnost pokrene neku drugu aktivnost, obe će proći kroz promene stanja. Prva aktivnost prestaje sa radom i prelazi u pauzirano ili stopirano stanje, dok se druga aktivnost kreira. Redosled poziva callback metoda u svakoj

aktivnosti je jasno definisan, naročito u slučaju da su aktivnosti deo iste aplikacije. U slučaju da aktivnost A pokreće aktivnost B, redosled je sledeći:

- Aktivnost A izvršava onPause(),
- Aktivnost B izvršava redom onCreate(), onStart() i onResume() i dobija fokus,
- Ukoliko aktivnost A više nije vidljiva na ekranu, izvršava onStop() metodu.

Jasno definisan i poznat redosled poziva značajno olakšava implementaciju prelaska iz jedne aktivnosti u drugu.

5.6.1. Intent objekat

Intent objekat služi kao apstraktni opis neke operacije koja treba da se izvrši. Može da se koristi za:

- pokretanje nove aktivnosti (poziv startActivityForResult(Intent)),
- komunikaciju za pozadinskim servisom (poziv startService(Intent)),
- komunikaciju sa BroadcastReceiver komponentom (broadcastIntent()).

Osim što se objekat klase Intent koristi za pokretanje nove aktivnosti, može da posluži i za prenos informacija između različitih komponenti sistema. Informacije koje mogu da budu prenete su definisane atributima klase Intent:

- ime komponente (component) – opcioni parametar, koji se koristi prilikom pokretanja eksplizite namere. Definiše ime komponente kojoj se prosleđuje ovaj objekat. Ukoliko ovaj parametar nije unet, radi se o implicitnoj nameri, odnosno Android sistem treba na osnovu vrednosti ostalih atributa da odredi kome treba da isporuči ovaj objekat.
- akcija (action) – uopšten opis akcije koju je potrebno obaviti. Klasa Intent ima veći broj konstanti koji opisuje generičke akcije, poput ACTION_SEND (koristi se kada je potrebno deliti podatke kroz neku drugu aplikaciju) ili ACTION_VIEW (uopšten opis akcije kada je potrebno prikazati neke podatke korisniku). Moguće je definisati i svoje akcije.
- podaci (data i/ili type) – koristi se za prosleđivanje podataka kroz URI objekat koji može da ukazuje na same podatke i na njihov MIME tip.
- kategorija (category) - dodatne informacije o akciji koju treba izvršiti. Na primer CATEGORY_LAUNCHER označava da je aktivnost inicijalna aktivnost nekog zadatka i da se nalazi u listi aplikacija koje izlistava sistem, dok CATEGORY_BROWSABLE dozvoljava da

aktivnost bude pokrenuta kroz veb čitač kako bi se prikazali podaci na koje ukazuje data adresa.

- extras – Bundle objekat koji sadrži sve dodatne informacije u obliku parova ključ vrednost. Na primer, prilikom kreiranja Intent objekta za slanje mejla sa akcijom ACTION_SEND, moguće je specificirati primaoca sa ključem EXTRA_EMAIL, a naslov mejla ključem EXTRA SUBJECT. U samoj klasi Intent postoji veći broj konstanti u obliku EXTRA_ za standardne tipove informacija koje se prenose. Naravno, moguće je i samostalno definisati dodatne ključeve.

5.6.2. Eksplisitne i implicitne namere

Ukoliko nova aktivnost ne treba da vrati nikakav rezultat svog izvršavanja, može se pokrenuti prostim pozivom metode startActivity(). Ukoliko je potrebno iz MainActivity jednostavno pokrenuti aktivnost DrugaActivity, to se može postići sledećim delom Java koda:

```
Intent intent = new Intent(this, DrugaActivity.class);
startActivity(intent);
```

Ovakav tip poziva je poznat kao **eksplisitna namera** – tačno je navedeno koja aktivnost treba da se pokrene. Sa druge strane, ponekad je samo potrebno izvršiti neku akciju koja je opisno zadata, poput slanja mejla ili pravljenja fotografije. Naša aplikacija čak i ne mora imati aktivnosti koje mogu da izvrše takve akcije, pa se može od samog Android sistema zahtevati da pronađe neku aktivnost u okviru neke druge aplikacije na uređaju koja može uraditi traženu akciju. Ovakav tip namere poznat je kao **implicitna namera**, gde se samo opiše akcija koju je potrebno izvršiti, bez tačnog navođenja koju konkretnu aktivnost treba pozvati.

Kako bismo ilustrovali implicitnu nameru, neka se posmatra slučaj da je aplikaciji potrebna usluga kamere, pošto je potrebno da se iz aplikacije može napraviti fotografija. Jedna od mogućih opcija bila bi da se u potpunosti od nule razvije aktivnost u kojoj će biti implementirana kompletan funkcionalnost kamere. Međutim, ovakav zadatak nije nimalo jednostavan iz dva razloga:

- kod koji bi trebalo razviti je složen,
- na samom telefonu već postoji dostupna aplikacija kamera, koja radi tačno ono što nam je potrebno, pa bi razvoj naše verzije bilo izmišljanje tople vode.

Dakle, umesto da sami razvijamo aktivnost koja bi obavljala kompletan funkcionalnost kamere, možemo u objektu tipa Intent opisno navesti da nam je potrebna akcija snimanja fotografije i dozvoliti Android sistemu da pronađe odgovarajuću aktivnost za nas. Konkretno, akcija koju je potrebno navesti u

ovom slučaju je MediaStore.ACTION_IMAGE_CAPTURE. U ovom slučaju, potrebno je da se dobije rezultat kada se pozvana aktivnost završi – konkretno, napravljena fotografija. Običan poziv startActivity() nije dovoljan, već se mora koristiti poziv startActivityForResult(Intent, int), koji osim samog objekta tipa Intent prihvata kao parametar i celobrojnu vrednost koja služi kao identifikator poziva. Ovaj identifikator je neophodan u slučaju da je potrebno razlikovati više različitih poziva startActivityForResult() iz iste aktivnosti. Rezultat će se vratiti prvoj aktivnosti kroz callback metodu onActivityResult(int, int, Intent). Prva celobrojna vrednost odgovara identifikatoru poziva pod kojim je aktivnost pozvana. Drugi parametar je kod rezultata, koji najčešće ima standardne vrednosti RESULT_CANCELED i RESULT_OK. Poslednji parametar je Intent objekat koji može u sebi sadržati dodatne podatke (u našem primeru to je napravljena fotografija). Kompletan primer koda sa implicitnim pozivom kamere dat je u nastavku.

```
package com.example.student.camerabasic;

import android.content.Intent;
import android.graphics.Bitmap;
import android.provider.MediaStore;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.ImageView;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {

    private Button btnCapture;
    private ImageView imgCapture;
    private static final int IMAGE_CAPTURE_CODE = 1;

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

        btnCapture = findViewById(R.id.btnTakePicture);
        imgCapture = findViewById(R.id.capturedImage);
        btnCapture.setOnClickListener(new
View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent cInt = new
Intent(MediaStore.ACTION_IMAGE_CAPTURE);

```

```

startActivityForResult(cInt, IMAGE_CAPTURE_CODE);
    }
})
}
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == IMAGE_CAPTURE_CODE) {
        if (resultCode == RESULT_OK) {
            Bitmap bp = (Bitmap)
data.getExtras().get("data");
            imgCapture.setImageBitmap(bp);
        } else if (resultCode == RESULT_CANCELED) {
            Toast.makeText(this, "Cancelled",
Toast.LENGTH_LONG).show();
        }
    }
}

```

Dati kod podrazumeva da je u XML fajlu sa rasporedom komponenti definisano jedno dugme sa Id btnTakePicture, na čiji klik će se okinuti implicitni poziv aktivnosti kamere, kao i polje za prikaz uslikane fotografije koja će biti vraćena kao rezultat, sa Id capturedPicture. Kod activity_main.xml fajla dat je u nastavku:

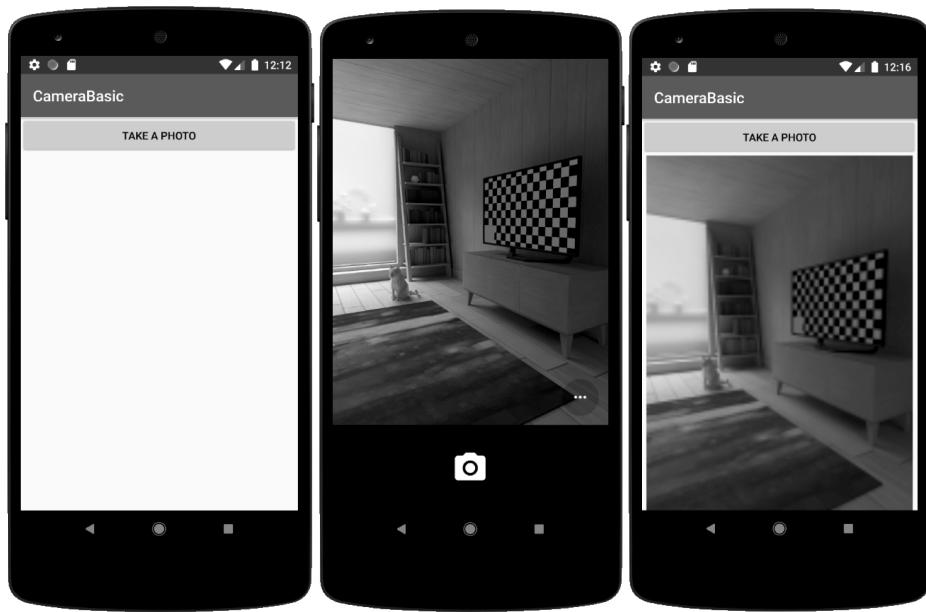
```

<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <Button
        android:id="@+id	btnTakePicture"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Take a photo" />
    <ImageView
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:id="@+id/capturedImage"/>
</LinearLayout>

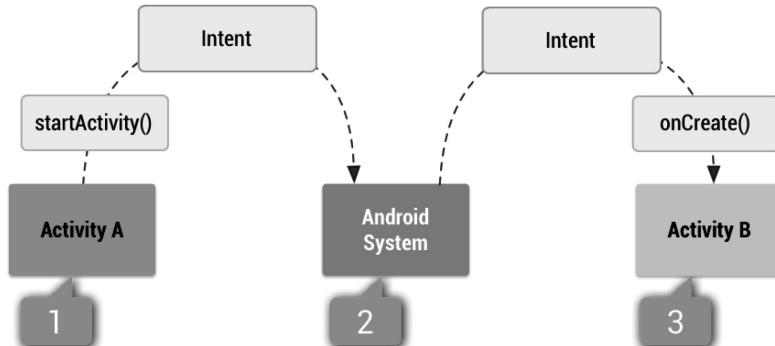
```

Prikaz rada ove jednostavne aplikacije dat je na slici 5.9. Na prvom delu slike se vidi prikaz implementirane aktivnosti, na drugom delu slike prikazana je pokrenuta aktivnost kamere nakon pritiska na dugme Take a photo, a na trećem delu slike prikazana je vraćena fotografija u prvoj aktivnosti.



Slika 5.9, implicitna namera i poziv aktivnosti kamere

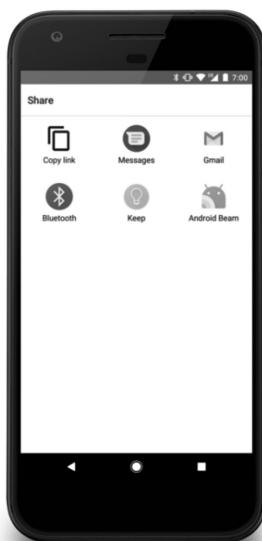
Na slici 5.10 prikazano je kako se koristi Intent objekat prilikom pokretanja nove aktivnosti. U slučaju eksplisitne namere, sistem odmah pokreće navedenu aktivnost. U slučaju implicitne namere, Android sistem pronalazi odgovarajuću komponentu poređenjem sadržaja Intent objekata sa Intent filterima deklarisanim u manifest fajlovima drugih aplikacija na uređaju. Ukoliko se pronađe poklapanje, sistem pokreće tu komponentu i prosledjuje joj Intent objekat. U slučaju da je pronađeno više kompatibilnih komponenti, sistem prikazuje dijalog u kome korisnik može odabratи aplikaciju koju želi da koristi.



Slika 5.10, prosleđivanje implicitne namere
(izvor: <https://developer.android.com/guide/components/intents-filters#Types>)

Ukoliko više aplikacija može da odgovori jednoj implicitnoj nameri, korisnik će možda želeti svaki put da odabere drugačiju aplikaciju, pa mu se može eksplisitno prikazati dijalog za odabir. Na primer, ukoliko se iz aplikacije uradi tzv. share upotrebot akcije ACTION_SEND, korisnik će možda u zavisnosti od situacije želeti da odabere drugačiju aplikaciju. Prikazani dijalog će izlistati sve aplikacije koje mogu da odgovore na zahtevanu akciju. Primer koda za prikazivanje dijaloga za odabir dat je u nastavku, a izgled ovog dijaloga prikazan je na slici 5.11.

```
Intent sendIntent = new Intent(Intent.ACTION_SEND);  
...  
  
// Uvek koristiti string resurse za UI text.  
// Ovaj resurs moze sadrzati na primer "Share this photo  
with"  
String title =  
getResources().getString(R.string.chooser_title);  
// Kreiranje intenta za prikaz chooser dialoga  
Intent chooser = Intent.createChooser(sendIntent, title);  
  
// Verifikacija da se originalni intent razresi na bar  
jednu aktivnost  
if (sendIntent.resolveActivity(getApplicationContext()) !=  
null) {  
    startActivityForResult(chooser);  
}
```



Slika 5.11, dijalog za odabir aplikacije
(izvor: <https://developer.android.com/guide/components/intents-filters#ForceChooser>)

5.6.3. Intent filteri i rezolucija Intenta

Ukoliko je potrebno proglašiti da aplikacija treba da prihvati implicitne namere, potrebno je definisati jedan ili više Intent filtera za svaku pojedinačnu komponentu aplikacije. To se postiže dodavanjem <intent-filter> elemenata ugnježdenog u element odgovarajuće komponente (na primer <activity>) u okviru manifest fajla. Svaki pojedinačni Intent filter definiše tip namera koje prihvata na osnovu akcije, podataka i kategorije primljenog Intent objekta. Android sistem prosledjuje implicitnu nameru aplikaciji samo u slučaju da Intent objekat prolazi jedan od filtera. Na primer, ukoliko je potrebno deklarisati da aktivnost može da prihvati Intent sa akcijom ACTION_SEND pri čemu je tip podatka text, u okviru elementa odgovarajuće aktivnosti u manifest fajlu bi bilo potrebno dodati:

```
<activity android:name="ShareActivity">
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category
    android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
</activity>
```

Android sistem testira svaki Intent objekat sa Intent filterom poređenjem tri elementa: akcije, podataka i kategorije. Da bi aktivnost prihvatile određeni Intent objekat, mora proći sva tri testa, inače Android sistem neće proslediti Intent objekat toj komponenti. Svaka komponenta može deklarisati i više filtera. Deo manifest fajla jedne socijalne aplikacije za deljenje različitog sadržaja bi mogao da izgleda ovako:

```
<activity android:name="MainActivity">
    <!--Ova aktivnost je inicijalna i ulaz u aplikaciju, treba
da se prikaze u app launcher-u -->
    <intent-filter>
        <action    android:name="android.intent.action.MAIN"
/>
        <category
    android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>

<activity android:name="ShareActivity">
    <!--Ova aktivnost prihvata akciju "SEND" sa tekstualnim
podacima -->
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
    </intent-filter>
</activity>
```

```

        <category
    android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
    <!--Ova aktivnost prihvata akcije "SEND" i
"SEND_MULTIPLE" sa multimedijom -->
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <action
    android:name="android.intent.action.SEND_MULTIPLE"/>
        <category
    android:name="android.intent.category.DEFAULT"/>
        <data
    android:mimeType="application/vnd.google.panorama360+jpg"/>
        <data android:mimeType="image/*"/>
        <data android:mimeType="video/*"/>
    </intent-filter>
</activity>
```

Prva aktivnost pod nazivom MainActivity je ulazna u aplikaciju, odnosno pokreće se kada korisnik pokrene aplikaciju dodirom na ikonicu. ACTION_MAIN označava da je u pitanju ulazna tačka u aplikaciju. CATEGORY_LAUNCHER označava da ikonica ove aktivnosti treba da se izlistava u listi aplikacija sistema. Ova dva atributa se moraju ovako upariti da bi se aktivnost prikazala u listi aplikacija. Druga aktivnost, ShareActivity, služi za deljenje teksta i multimedije. Korisnici mogu da dođu do ove aktivnosti ili navigacijom iz MainActivity, ili iz druge aplikacije prilikom implicitne namere koja prođe jedan od dva navedena Intent filtera.

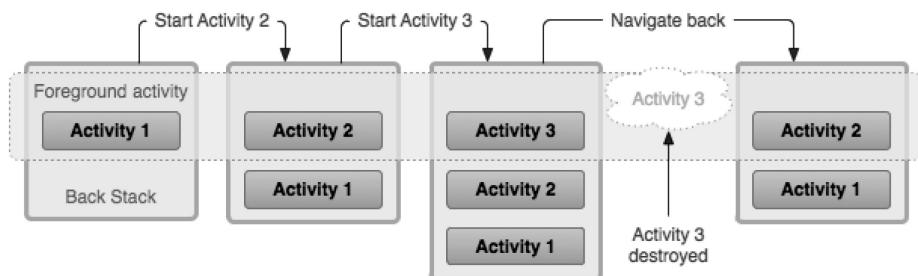
Kada Android sistem dobije implicitnu nameru, on traži najadekvatniju aktivnost za dati Intent objekat poređenjem tog objekta sa Intent filterima u manifest fajlovima aplikacija. Ovaj proces se naziva rezolucija intenta, i podrazumeva poređenje po tri osnovna atributa: action, data (i URI i type) i category.

5.7. Back Stack

Android aplikacije se u opštem slučaju sastoje od više aktivnosti. Aktivnosti se projektuju na takav način da svaka treba da obavi neki specifičan vid interakcije sa korisnikom. U interakciju sa korisnikom spada i pokretanje nove aktivnosti iz trenutne aktivnosti, pri čemu nova aktivnost može biti deo aplikacije, ali i ne mora, kao što je objašnjeno kroz implicitne namere.

Sve aktivnosti sa kojima korisnik vrši interakciju dok obavlja određen posao (na primer slanje i provera mejlova, pregled neke od socijalnih mreža ili nešto treće) se zapravo posmatraju kao jedna celina. Ovakva kolekcija aktivnosti se u Android sistemu naziva zadatak (engl. *task*). Sve ove aktivnosti su uređene u strukturi u

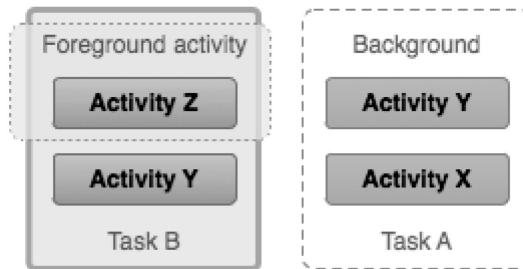
obliku steka, pod nazivom back stack, po redosledu pojedinačnog pokretanja svake aktivnosti. Stek je struktura koja je uređena po LIFO principu (engl. *Last In First Out*), odnosno poslednja dodata aktivnost će biti prva uklonjena iz steka. Kada korisnik otvori novu aplikaciju, kreira se novi task, pokreće se njena inicijalna aktivnost i stavlja na stek. Kada se iz te aktivnosti otvori nova aktivnost, ona se dodaje na vrh steka i dobija fokus. Prethodna aktivnost ostaje na steku u stopiranom stanju. Ukoliko korisnik pritisne Back dugme, trenutna aktivnost se uklanja sa vrha steka i uništava, a prethodna aktivnost dobija fokus i nastavlja sa interakcijom. Aktivnosti u steku se nikada ne preuređuju, već se samo mogu dodavati i uklanjati sa vrha steka. Na slici 5.12 prikazan je primer kretanja korisnika kroz aktivnosti i stanje steka u svakom trenutku.



Slika 5.12, princip rada back stack-a
 (izvor: <https://developer.android.com/guide/components/activities/tasks-and-back-stack>)

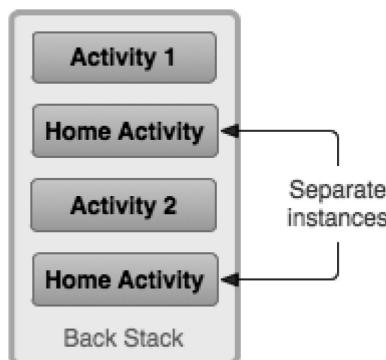
Kada korisnik pritisne Back dugme, aktivnost se uklanja sa vrha steka i uništava, pri čemu na vrh steka i u prvi plan izlazi prethodna aktivnost, sve dok se korisnik ne vrati na glavni ekran (ili bilo koju drugu aktivnost koja se izvršavala u trenutku pokretanja taska). Kada se sve aktivnosti uklone iz steka, task više ne postoji.

Jedan task se ponaša kao celina i pri prelasku u pozadinu, u slučajevima da korisnik odluči da pokrene novu aplikaciju koja izlazi u prvi plan, ili ukoliko ode na glavni ekran telefona pritiskom na Home dugme. Sve aktivnosti koje pripadaju tom tasku su u tom slučaju stopirane, ali back stack ostaje sačuvan, kako bi korisnik mogao da nastavi gde je stao kada odluči da vrati task u prvi plan (slika 5.13).



Slika 5.13, primer dva taska, pri čemu je task B u fokusu i vrši interakciju sa korisnikom, dok je task A u pozadini sa kompletno očuvanim back stack-om (izvor: <https://developer.android.com/guide/components/activities/tasks-and-back-stack>)

Posebna pažnja se mora obratiti na činjenicu da se aktivnosti u okviru back stack-a nikada ne preuređuju. Zbog toga se može desiti da u back stack-u postoji više instanci jedne iste aktivnosti u određenim okolnostima (slika 5.14). Primer je slučaj da aplikacija dozvoljava pokretanje određene aktivnosti iz više drugih aktivnosti, pri čemu se svaki put ta aktivnost kreira i stavlja na vrh back stack-a.



Slika 5.14, slučaj više instanci jedne aktivnosti u okviru back stack-a (izvor: <https://developer.android.com/guide/components/activities/tasks-and-back-stack>)

Opisano ponašanje back stack-a i taska je odgovarajući za većinu aplikacija. Ipak, moguće je napraviti odstupanje od definisanog načina rada, upotrebom atributa u manifestu aplikacije ili putem flag-ova u Intent objektu koji se prosleđuje aktivnosti. Bez dubljeg ulazeња u detalje, navodimo samo da su podržani sledeći scenariji:

- Nova aktivnost se pokreće u posebnom tasku umesto u već postojećem

- Prilikom pokretanja aktivnosti, podiže se već postojeća instanca umesto kreiranja nove na vrhu back stack-a.
- Kada korisnik napusti zadatak, sve aktivnosti se uklanjaju iz back stack-a, osim određene korene aktivnosti

5.8. Pitanja za vežbu

- Šta je aktivnost?
- Da li je potrebno deklarisati aktivnost u manifestu aplikacije?
- Koja su osnovna stanja u kojima aktivnost može da bude?
- Opisati ukratko životni ciklus aplikacije.
- Detaljno objasniti životni ciklus aktivnosti.
- Koje su callback metode životnog ciklusa?
- Između kojih callback metoda je aktivnost vidljiva?
- Koju metodu sistem poziva kada aktivnost počinje da gubi fokus?
- Koja metoda se poziva kada aktivnost više nije vidljiva?
- Šta se tipično radi u onCreate() metodi?
- Koji je redosled poziva callback metoda prilikom pokretanja aktivnosti?
- Kako se čuva stanje korisničkog interfejsa aktivnosti?
- Kako se rekreira sačuvano stanje korisničkog interfejsa aktivnosti?
- Šta je Bundle objekat i kako se koristi?
- Kako se stavlja vrednosti u Bundle objekat?
- Kako se vade vrednosti iz Bundle objekta?
- Šta se dešava prilikom promene orijentacije ekrana?
- Kako se iz jedne aktivnosti pokreće druga aktivnost?
- Šta je namera (Intent)?
- Koji tipovi namera postoje?
- Objasnitи detaljno razliku između eksplicitne i implicitne namerе.
- Kako se mogu podaci iz jedne aktivnosti proslediti u drugu aktivnost?
- Šta se sve može pokrenuti sa Intent objektom?
- Koja metoda se koristi za pokretanje nove aktivnosti?
- Koja metoda se koristi za pokretanje nove aktivnosti ukoliko nova aktivnost treba da vrati rezultat prvoj?
- Kako se deklariše da aktivnost prima implicitne namerе?
- Šta je IntentFilter?
- Kako se vrši rezolucija Intenta?
- Objasnitи princip rada Back Stack-a.

5.9. Zadaci za vežbu

Primer 1: Napraviti korisnički interfejs u kome korisnik unosi svoje ime, adresu i telefon. Raspored komponenti treba da bude linearni vertikalni. Na ekranu postoji i dugme na čiji klik aktivnost treba da preuzme sve unete podatke, i da ih prikaže u tekstualnom polju na ekranu. U aktivnosti dohvatiće reference na sve objekte korisničkog interfejsa, implementirati programski osluškivač za onclick događaj na dugmetu, preuzeti sve podatke sa korisničkog interfejsa i ispisati ih u pomenutom tekstualnom polju.

Rešenje:

Zadati korisnički interfejs se može implementirati na sledeći način, kako je definisano u activity_main.xml fajlu:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <EditText
        android:id="@+id/inputIme"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Unesi ime"/>
    <EditText
        android:id="@+id/inputAdresa"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Unesi adresu"/>
    <EditText
        android:id="@+id/inputTelefon"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Unesi telefon"/>
    <Button
        android:id="@+id/buttonPosalji"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Posalji"/>
    <TextView
        android:id="@+id/labelUnetiPodaci"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="20dp"
```

```
        android:lines="20" />
</LinearLayout>
```

Kod zadate aktivnosti, koja uzima reference na sve objekte i postavlja osluškivač u okviru posebno izdvojene initComponents() metode, a zatim nakon klika na dugme ispisuje sve podatke u tekstualnom polju, dat je sadržajem MainActivity.java klase:

```
package com.example.student.myapplication;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity
implements View.OnClickListener {

    private EditText inputIme;
    private EditText inputAdresa;
    private EditText inputTelefon;
    private Button buttonPosalji;
    private TextView labelUnetiPodaci;

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

    private void initComponents(){
        //dohvatanje referenci na objekte
        inputIme = findViewById(R.id.inputIme);
        inputAdresa = findViewById(R.id.inputAdresa);
        inputTelefon = findViewById(R.id.inputTelefon);
        labelUnetiPodaci =
        findViewById(R.id.labelUnetiPodaci);
        buttonPosalji = findViewById(R.id.buttonPosalji);
        //postavljanje listenera
        buttonPosalji.setOnClickListener(this);
    }

    @Override
    public void onClick(View view) {
        //kliknuto dugme, pokupiti sve unete vrednosti iz
```

```

polja
    String unetoIme = inputIme.getText().toString();
    String unetaAdresa =
inputAdresa.getText().toString();
    String unetiTelefon =
inputTelefon.getText().toString();
    //postaviti u prikaz
    String podaci = "Uneti su podaci:\n" +
        unetoIme + "\n" +
        unetaAdresa + "\n" +
        unetiTelefon + "\n";
    labelUnetiPodaci.setText(podaci);
}
}

```

Korisnički interfejs aplikacije nakon pokretanja prikazan je na slici 5.15 levo. Nakon unosa podataka i klika na dugme, uneti podaci se ispisuju u tekstualnom polju kao na slici 5.15 desno.



Slika 5.15, Prikaz aktivnosti koja preuzima podatke sa korisničkog interfejsa i ispisuje ih u tekstualnom polju.

Primer 2: Napraviti aplikaciju koja dinamički kreira korisnički interfejs u obliku liste kontakata. Kontakti mogu da budu jednog od četiri moguća tipa (telefonski broj, Viber, Facebook ili Mail), a za svaki od tipova je u dinamički kreiranoj listi potrebno postaviti drugačiju ikonicu. Potrebno je kreirati posebni raspored komponenti koji će definisati izgled svakog pojedinačnog unosa.

Rešenje: Na početku, potrebno je definisati izgled korisničkog interfejsa. U ovom slučaju, odabran je ScrollView, unutar kojeg se nalazi linearni vertikalni raspored u koji će se programski ubaciti pojedinačni kontakti. Pojedinačni kontakti su takođe definisani sopstvenim rasporedom komponenti, koji se nalazi u posebnom xml fajlu. Sadržaj activity_main.xml fajla dat je u nastavku:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"

    tools:context="com.example.mzivkovic.myapplication.MainActivity">

    <ScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <LinearLayout
            android:id="@+id/mainScrollView"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical" />
    </ScrollView>
</RelativeLayout>
```

Svaki pojedinačni kontakt je definisan sopstvenim rasporedom komponenti. Ovaj raspored sadrži jedan ImageView, u kome se prikazuje ikonica odgovarajućeg tipa kontakta, i dva tekstualna polja u kojima se nalaze ime kontakta i odgovarajuća vrednost (broj telefona, mail adresa ili slično, u zavisnosti od tipa kontakta). Sadržaj je definisan u contact_view.xml fajlu na sledeći način:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    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="100sp">
```

```

<ImageView
    android:id="@+id/imageIkonica"
    android:layout_width="100sp"
    android:layout_height="100sp"
    android:layout_alignParentLeft="true"
    android:layout_alignParentStart="true"
    android:layout_centerVertical="true"
    android:layout_margin="5sp"
    app:srcCompat="@drawable/user" />

<TextView
    android:id="@+id/labelaIme"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentEnd="true"
    android:layout_alignParentRight="true"
    android:layout_alignTop="@+id/imageIkonica"
    android:layout_toEndOf="@+id/imageIkonica"
    android:layout_toRightOf="@+id/imageIkonica"
    android:textSize="18sp" />

<TextView
    android:id="@+id/labelaVrednost"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignBottom="@+id/imageIkonica"
    android:layout_below="@+id/labelaIme"
    android:layout_marginTop="5sp"
    android:layout_toEndOf="@+id/imageIkonica"
    android:layout_toRightOf="@+id/imageIkonica" />
</RelativeLayout>
```

Pojedinačni kontakt u kodu je predstavljen objektom klase Kontakt.java. U pitanju je obična klasa koja predstavlja model podatka koji se koristi u okviru aktivnosti, a čiji je kod dat u nastavku:

```

package com.example.mzivkovic.myapplication;

public class Kontakt {

    public enum TipKontakta {EMAIL, PHONE, VIBER, FACEBOOK}

    private String ime, vrednost;
    private TipKontakta tipKontakta;

    public Kontakt(String ime, String vrednost, TipKontakta tipKontakta) {
```

```

        this.ime = ime;
        this.vrednost = vrednost;
        this.tipKontakta = tipKontakta;
    }

    public String getIme() {
        return ime;
    }

    public String getVrednost() {
        return vrednost;
    }

    public TipKontakta getTipKontakta() {
        return tipKontakta;
    }
}

```

U realnom scenariju, lista kontakata bi se dovlačila preko nekog oblika API-ja (dohvatanje preko Interneta, čitanje iz baze i slično). U ovom primeru, API je modelovan klasom KontaktAPI koja samo programski kreira listu kontakata i vraća je kao rezultat. Sadržaj ove klase dat je u nastavku:

```

package com.example.mzivkovic.myapplication;

import java.util.LinkedList;

public class KontaktAPI {

    public static LinkedList<Kontakt> getMyContacts () {
        LinkedList<Kontakt> lista = new LinkedList<>();
        lista.add(new Kontakt ("Pera Peric",
"+3811111111555", Kontakt.TipKontakta.PHONE));
        lista.add(new Kontakt ("Miodrag Zivkovic",
"Miodrag.Zivkovic.viber", Kontakt.TipKontakta.VIBER));
        lista.add(new Kontakt ("Jovan Jovanovic",
"jovan.jovanovic55@singidunum.ac.rs",
Kontakt.TipKontakta.EMAIL));
        lista.add(new Kontakt ("Mika Peric",
"+3811111311555", Kontakt.TipKontakta.FACEBOOK));
        lista.add(new Kontakt ("Miodrag Jovanovic",
"Miodrag.Jovanovic.viber", Kontakt.TipKontakta.VIBER));
        lista.add(new Kontakt ("Jovan Jovic",
"jovan.jovic32@singidunum.ac.rs",
Kontakt.TipKontakta.EMAIL));
        lista.add(new Kontakt ("Pera Petrovic",
"+3811111211555", Kontakt.TipKontakta.PHONE));
        lista.add(new Kontakt ("Miodrag Petrovic",

```

```

"mzz@singidunum.ac.rs", Kontakt.TipKontakta.EMAIL));
    lista.add(new Kontakt ("Jovan Jocic",
"jovan.jocic33@singidunum.ac.rs",
Kontakt.TipKontakta.EMAIL));
    return lista;
}
}

```

Na kraju, aktivnost treba da dohvati generisanu listu kontakata, i da za svaki pojedinačni kontakt generiše njegov prikaz i smesti ga u ScrollView u activity_main.xml fajlu. Ovaj primer podrazumeva da se u direktorijumu res/drawable nalaze odgovarajuće ikonice za svaki tip kontakta. Kod klase MainActivity.java dat je u nastavku:

```

package com.example.mzivkovic.myapplication;

import android.content.Context;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.widget.ImageView;
import android.widget.LinearLayout;
import android.widget.RelativeLayout;
import android.widget.TextView;

import java.util.LinkedList;

public class MainActivity extends AppCompatActivity {

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

        LinkedList<Kontakt> lista =
KontaktAPI.getMyContacts();

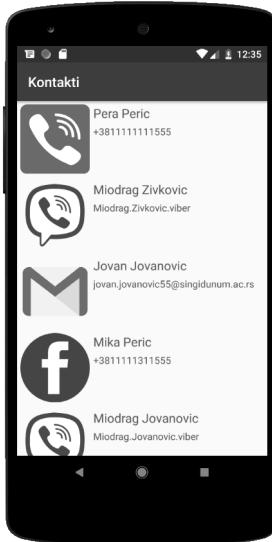
        LinearLayout mainScrollView =
findViewById(R.id.mainScrollView);
        LayoutInflator inflater = (LayoutInflator)
getSystemService(Context.LAYOUT_INFLATER_SERVICE);

        for (Kontakt kontakt : lista){
            RelativeLayout item = (RelativeLayout)
inflater.inflate(R.layout.contact_view, null);
            ((TextView)
item.findViewById(R.id.labelaIme)).setText(kontakt.getIme());
        }
    }
}

```

```
        ((TextView)
item.findViewById(R.id.labelavrednost)).setText(kontakt.
getVrednost());  
  
        ImageView imageIkonica = (ImageView)
item.findViewById(R.id.imageIkonica);
        switch (kontakt.getTipKontakta()) {
            case EMAIL:
imageIkonica.setImageResource(R.drawable.email);
            break;
            case PHONE:
imageIkonica.setImageResource(R.drawable.phone);
            break;
            case VIBER:
imageIkonica.setImageResource(R.drawable.viber);
            break;
            case FACEBOOK:
imageIkonica.setImageResource(R.drawable.facebook);
        }
        mainScrollView.addView(item);
    }
}
```

Nakon pokretanja aplikacije, u okviru onCreate() metode se dinamički kreira raspored komponenti za svakog pojedinačnog kontakta i smešta u korisnički interfejs. Izgled aplikacije je dat na slici 5.16.



Slika 5.16, dinamički kreiran korisnički interfejs

Primer 3: Potrebno je napraviti aplikaciju za studente, koja od korisnika u prvom ekranu traži da unese odgovarajuće podatke o sebi (korisničko ime, šifru, odabir fakulteta koji studira, status zaposlenja i datum rođenja). Nakon klika na dugme, svi ovi podaci se uzimaju iz korisničkog interfejsa, pakuju u Intent objekat i šalju drugoj aktivnosti koja će ih prihvati i prikazati na ekranu.

Rešenje: Ovaj primer ilustruje eksplizitnu nameru, pošto se svi podaci uneti u prvoj aktivnosti prosleđuju drugoj aktivnosti, koju glavna aktivnost pokreće nakon klika na dugme. Korisnički interfejs prve aktivnosti dat je u fajlu activity_main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.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"
    tools:context="com.example.mzivkovic.myapplication.MainActivity">
    <ScrollView
        android:id="@+id/scrollView2"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:layout_marginBottom="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent">

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical" >

            <TextView
                android:id="@+id/labelUsername"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:text="Username" />

            <EditText
                android:id="@+id/inputUsername"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
```

```
        android:ems="10"
        android:inputType="textPersonName" />

    <TextView
        android:id="@+id/labelPassword"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="20sp"
        android:text="Password" />

    <EditText
        android:id="@+id/inputPassword"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:inputType="textPassword" />

    <TextView
        android:id="@+id/labelFaculty"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="20sp"
        android:text="Faculty" />

    <Spinner
        android:id="@+id/inputFaculty"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/labelEmployed"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="20sp"
        android:text="Employed" />

    <Switch
        android:id="@+id/inputEmployed"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Are you employed?" />

    <TextView
        android:id="@+id/labelBirthdate"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="20sp"
        android:text="Birth date" />
```

```

<DatePicker
    android:id="@+id/inputBirthdate"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android: datePickerMode="spinner"
    android:calendarViewShown="false" />

<Button
    android:id="@+id/buttonSend"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="20sp"
    android:text="Send" />
</LinearLayout>
</ScrollView>
</android.support.constraint.ConstraintLayout>

```

Glavna aktivnost nakon klika na dugme sakuplja sve unete vrednosti od strane korisnika, pakuje ih u Intent objekat i nakon toga startuje drugu aktivnost. U pitanju je eksplicitna namera, pošto se druga aktivnost poziva eksplicitno po imenu aktivnosti u Intent objektu koji se prosleđuje kao argument startActivity() metode. Kod MainActivity.java klase dat je u nastavku:

```

package com.example.mzivkovic.myapplication;

import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.DatePicker;
import android.widget.EditText;
import android.widget.Spinner;
import android.widget.Switch;
import android.widget.TextView;
import java.util.ArrayList;

public class MainActivity extends AppCompatActivity
    implements View.OnClickListener {

    private EditText inputUsername;
    private EditText inputPassword;
    private Spinner inputFaculty;
    private Switch inputEmployed;
    private DatePicker inputBirthday;
    private Button buttonSend;

```

```

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

    initComponents();
}

public void initComponents() {
    //inicijalizacija liste fakulteta za spinner
    inputFaculty = findViewById(R.id.inputFaculty);
    ArrayList<String> facultyList = new ArrayList<>();
    facultyList.add("Fakultet za informatiku i
racunarstvo");
    facultyList.add("Tehnicki fakultet");
    facultyList.add("Poslovni fakultet");

    ArrayAdapter<String> adapter = new
ArrayAdapter<String>(this,
    android.R.layout.simple_spinner_dropdown_item,
    facultyList);

    inputFaculty.setAdapter(adapter);

    inputUsername = findViewById(R.id.inputUsername);
    inputPassword = findViewById(R.id.inputPassword);
    inputEmployed = findViewById(R.id.inputEmployed);
    inputBirthday = findViewById(R.id.inputBirthdate);
    buttonSend = findViewById(R.id.buttonSend);
    buttonSend.setOnClickListener(this);
}

@Override
public void onClick(View view) {
    if (view.getId()== R.id.buttonSend) {
        Intent intent = new Intent(this,
ConfirmationActivity.class);

        String username =
inputUsername.getText().toString();
        String password =
inputPassword.getText().toString();
        String faculty = (String)
inputFaculty.getSelectedItem();
        boolean employed = inputEmployed.isChecked();

        String birthdayString = String.format("%4d-%2d-%2d",
        Integer.parseInt(inputYear.getText().toString()),
        Integer.parseInt(inputMonth.getText().toString()),
        Integer.parseInt(inputDay.getText().toString()));
    }
}

```

```

%2d", inputBirthday.getYear(), inputBirthday.getMonth() +
1, inputBirthday.getDayOfMonth());
        Bundle extras = new Bundle();
        extras.putString("username", username);
        extras.putString("password", password);
        extras.putString("faculty", faculty);
        extras.putBoolean("employed", employed);
        extras.putString("birthday", birthdayString);

        intent.putExtras(extras);
        startActivity(intent);
    }

}
}

```

Druga aktivnost takođe ima svoj fajl sa rasporedom komponenti. Ovaj fajl je u ovom primeru jednostavan, sadrži samo tekstualno polje gde se prikazuju svi podaci koji su primljeni kroz Intent objekat. Raspored komponenti druge aktivnosti dat je sadržajem activity_confirmation.xml fajla u nastavku:

```

<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.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"

tools:context="com.example.mzivkovic.myapplication.ConfirmationActivity">

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <TextView
            android:id="@+id/labelMessage"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:lines="10"
            android:text="Message" />

        <Button
            android:id="@+id/buttonConfirm"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentEnd="true"

```

```

        android:layout_alignParentRight="true"
        android:layout_below="@+id/labelMessage"
        android:layout_marginTop="30sp"
        android:text="Confirm" />
    </RelativeLayout>
</android.support.constraint.ConstraintLayout>
```

Kod same aktivnosti dat je u klasi ConfirmationActivity.xml:

```

package com.example.mzivkovic.myapplication;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.TextView;

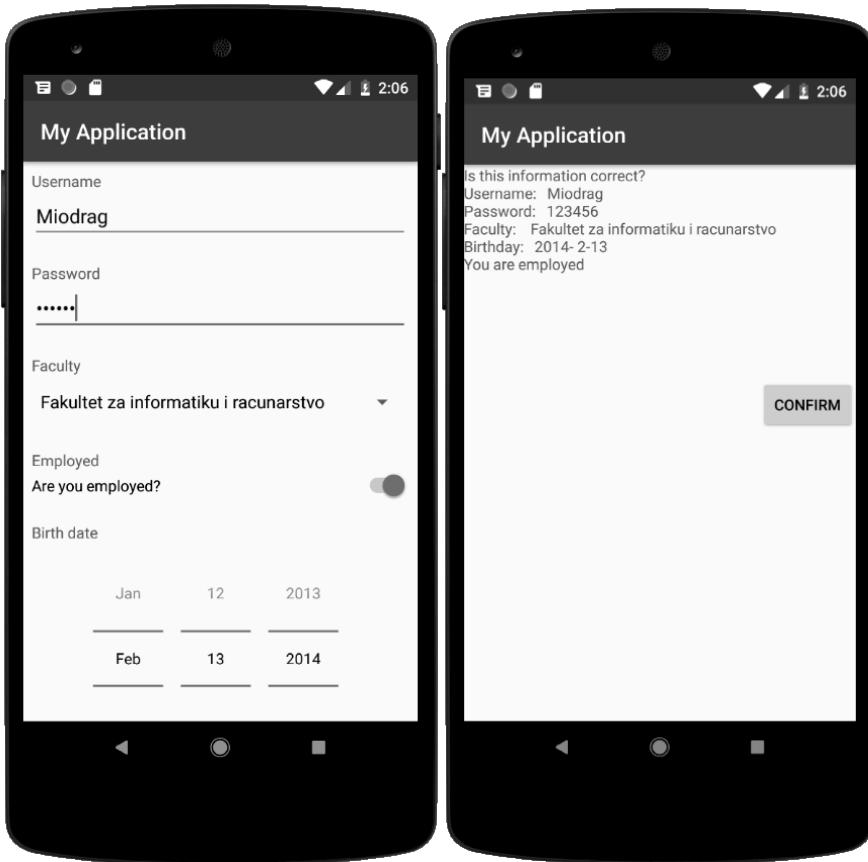
public class ConfirmationActivity extends AppCompatActivity
{

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

        Bundle extras = getIntent().getExtras();
        String username = extras.getString("username");
        String password = extras.getString("password");
        String faculty = extras.getString("faculty");
        Boolean employed = extras.getBoolean("employed");
        String birthday = extras.getString("birthday");
        String message = "Is this information correct? \n";
        message += "Username: " + username + "\n";
        message += "Password: " + password + "\n";
        message += "Faculty: " + faculty + "\n";
        message += "Birthday: " + birthday + "\n";
        message += (employed) ? "You are employed \n" : "You
are not employed \n";
        ((TextView)
        findViewById(R.id.labelMessage)).setText(message);

    }
}
```

Prikaz rada aplikacije dat je na slici 5.17. Sa leve strane slike prikazana je prva aktivnost, koja od korisnika traži da unese sve zahtevane informacije, a klikom na dugme pokreće drugu aktivnost i šalje joj sve unete podatke. Sa desne strane iste slike prikazana je druga aktivnost, koja ispisuje sve podatke koje primila kroz Intent objekat od prve aktivnosti.

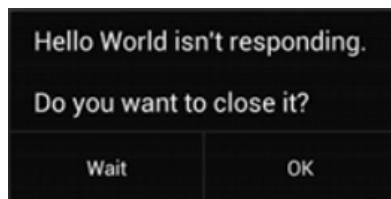


Slika 5.17, primer aplikacije koja koristi eksplicitnu nameru za pokretanje druge aktivnosti u okviru iste aplikacije

6. Niti

Svaka Android aplikacija se, nakon pokretanja, pokreće u okviru novog Linux procesa sa jednom niti u kojoj se ta aplikacija izvršava. Ova nit se naziva glavna nit (engl. *main thread*), i u njoj se izvršavaju sve komponente aplikacije. Poznata je još pod nazivom korisnička nit (engl. *UI thread* - nit korisničkog interfejsa), zbog toga što je zadužena za obradu događaja koje korisnik generiše prilikom interakcije sa korisničkim interfejsom aplikacije, kao i za iscrtavanje događaja. Drugim rečima, ova nit je veoma bitna zato što se u okviru nje odvija kompletan interakciju aplikacije sa korisnikom, odnosno sa komponentama Android UI toolkit (komponente definisane u android.widget i android.view paketima).

Android sistem je dizajniran na takav način da se sve komponente istog procesa instanciraju i izvršavaju u jednoj – korisničkoj niti. Sistem neće samostalno pokrenuti novu odvojenu nit za novu instancu neke komponente. Dodatno, sve callback metode (poput onClick(), onKeyUp() itd.) se okidaju i izvršavaju u ovoj niti. Ako bi aplikacija u okviru ovih callback metoda izvršavala neki dug i složen posao, to bi drastično uticalo na performanse same aplikacije. Uopšteno gledano, bilo kakva dugotrajna operacija koja se izvršava u korisničkoj niti, poput dohvatanja podataka preko mreže ili pristupa bazi podataka, će blokirati kompletno korisnički interfejs. Kada je korisnička nit blokirana, to znači da se ne generišu niti obrađuju događaji, kao i da se korisnički interfejs ne iscrtava – drugim rečima ekran aplikacije je zamrznut, a korisnik ne može da izvrši nikakvu interakciju sa njim. Situaciju dodatno pogoršava sam Android sistem – ukoliko se primeti da je korisnička nit neke aplikacije blokirana više od pet sekundi, korisniku će biti prikazan „Application not responding“ dijalog (slika 6.1). Ovaj zloglasni dijalog je poznat i pod skraćenim imenom ANR. Nije potrebno dodatno naglašavati koliko ANR može imati loše posledice po zadovoljstvo korisnika, koji može da odluči i da obriše aplikaciju. Dakle, programer mora uvek biti svestan činjenice da će Android sistem prikazati ANR u slučaju da aplikacija ne može da odgovori na korisnički unos.



Slika 6.1, Application not responding dijalog

Još jedno ograničenje je postavljeno od strane samog Android UI toolkit-a, koji nije bezbedno koristiti u višenitnom sistemu (nije *thread-safe*). Posledica po programera se ogleda u tome da se elementima korisničkog interfejsa ne sme

pristupati iz neke druge niti, već isključivo iz korisničke niti. Navedeni problemi se mogu sumirati u dva veoma jednostavna pravila koje programer mora poštovati u svakom trenutku:

- Korisnička nit se ne sme blokirati,
- Komponentama interfejsa se ne sme pristupati van korisničke niti.

Kako se izbegava ANR? Svaka dugotrajna operacija koju aplikacija obavlja u korisničkoj niti može izazvati ANR dijalog. Zbog toga, sve metode koje se izvršavaju u okviru korisničke niti moraju da budu što brže i da obavljaju što je moguće manje posla u samoj korisničkoj niti. Naravno, to ne znači da aplikacija uopšte ne sme da obavlja dugotrajne operacije, koje su vrlo često neophodne i predstavljaju samu srž i osnovnu funkcionalnost date aplikacije. Takve operacije treba izvršiti u odvojenoj niti, kako se ne bi ugrozila brzina odziva korisničkog interfejsa aplikacije. Iz drugih niti nije dozvoljeno ažurirati korisnički interfejs – to se sme obavljati samo iz korisničke niti. Android nudi nekoliko opcija kako se iz drugih niti može pristupiti korisničkoj niti, kroz sledeće metode:

- `Activity.runOnUiThread(Runnable)` – izvršava zadatu akciju u korisničkoj niti. Ukoliko je trenutna nit korisnička nit, akcija se izvršava odmah. Ukoliko trenutna nit nije korisnička nit, akcija se stavlja u red za čekanje na izvršavanje u korisničkoj niti.
- `View.post(Runnable)` – dodaje akciju u message queue, pri čemu će se Runnable akcija izvršiti u korisničkoj niti.
- `View.postDelayed(Runnable, long)` - dodaje akciju u message queue, pri čemu će se Runnable akcija izvršiti u korisničkoj niti nakon isteka zadatog vremena.

Na primer, implementacija data u nastavku je bezbedna u okruženju više niti – dugotrajna operacija obrade slike se vrši u odvojenoj niti, dok se ažuriranje ImageView komponente obavezno vrši iz korisničke niti.

```
public void onClick(View v) {  
    new Thread(new Runnable() {  
        public void run() {  
            // obrada koja može dugo trajati  
            final Bitmap bitmap =  
                obradiSliku("slika1.png");  
            imageView.post(new Runnable() {  
                public void run() {  
                    imageView.setImageBitmap(bitmap);  
                }  
            });  
        }  
    }).start();  
}
```

Ovo je klasičan primer tzv. radne niti (engl. *worker thread*), čiji je posao da obavi dugotrajnu operaciju odvojeno od niti korisničkog interfejsa. Iako nema ništa pogrešno u ovoj implementaciji, to ipak nije preporučeni način rada sa nitima, bar u okviru Android sistema. Ovakav kod teži da se usložnjava ukoliko se poveća kompleksnost operacije koju treba da izvrši, a samim tim postaje težak za održavanje. Kako bi se malo olakšao razvoj, može se koristiti Handler objekat unutar radne niti za isporuku rezultata operacije korisničkoj niti. Najbolje rešenje, međutim, je upotreba klase AsyncTask, koja drastično olakšava izvršavanje radne niti koja u nekom trenutku mora da komunicira sa korisničkim interfejsom. Ovaj pristup je preporučen od strane Android zajednice i opisan je u nastavku ovog poglavlja.

6.1. AsyncTask

Klasa AsyncTask dozvoljava izvršavanje asinhronih zadataka na korisničkom interfejsu aplikacije. Ona objedinjuje dva procesa time što izvršava blokirajuću operaciju u radnoj niti, a objavljuje rezultate na korisničkoj niti, bez potrebe da programer samostalno rukuje nitima. To je preporučeni način korišćenja niti za obavljanje dužih pozadinskih operacija koje je kasnije potrebno objaviti na ekranu. Sama klasa je osmišljena kao pomoćna klasa bazirana na Thread i Handler klasama. Treba napomenuti da za dugotrajne pozadinske operacije (koji traju duže od desetak sekundi), Android zajednica i dalje preporučuje tradicionalan pristup i upotrebu java.util.concurrent paketa.

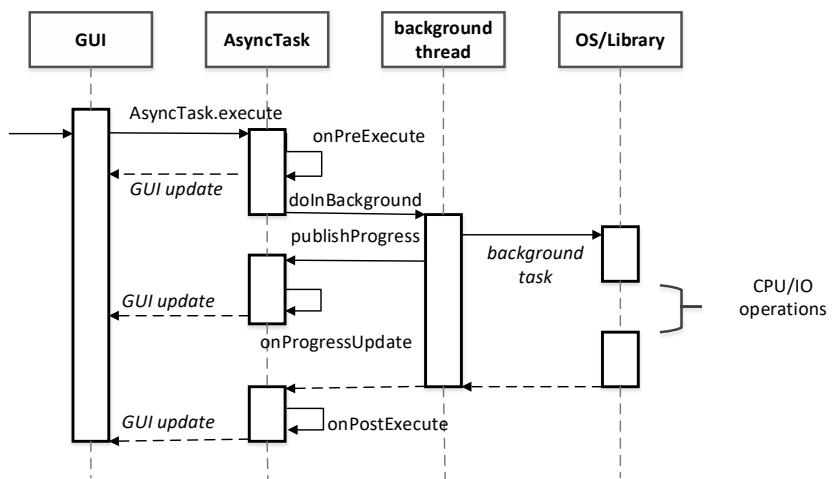
Praktično gledano, AsyncTask se koristi tako što se kreira nova klasa, izvedena iz AsyncTask klase, a zatim implementira nekoliko callback metoda. Sam AsyncTask je definisan sa tri generička tipa i četiri koraka (metode koje je potrebno implementirati). Generički tipovi se definišu u obliku AsyncTask<Params, Progress, Result>, pri čemu je njihova uloga sledeća:

- Params – označava tip ulaznih parametara koje AsyncTask prihvata pre izvršavanja.
- Progress – označava tip jedinice koja služi da se objavi progres za vreme izvršavanja samog taska.
- Result – označava tip rezultata operacije koja se izvršava u pozadini.

Nije uvek potrebno koristiti sve tipove, ali se to mora označiti sa tipom Void. Na primer, AsyncTask<String, Void, Bitmap> označava da task uzima String kao ulazni parametar, ne prikazuje progres i vraća Bitmap kao rezultat rada nakon završetka operacije. Osim potrebnih parametara, AsyncTask je određen i sa četiri koraka kroz koje prolazi dok se izvršava. Svaki od koraka je povezan i sa metodom koju je potrebno implementirati. Koraci su, redom:

- `onPreExecute()` – metoda koja se poziva u okviru korisničke niti, pre izvršavanja samog taska. U ovom koraku potrebno je pripremiti sve što je tasku potrebno, poput inicijalizacije početnog progrusa na vrednost 0.
- `doInBackground(Params...)` – metoda se izvršava u pozadinskoj niti, a poziva se neposredno nakon izvršavanja metode `onPreExecute()`. Prosleđuju joj se parametri zadati u definiciji `AsyncTask<Params, Progress, Result>`. Ova metoda izvršava dugotrajanu operaciju, čiji se rezultat mora vratiti u okviru ovog koraka, i koji će biti prosleđen poslednjem koraku, odnosno metodi `onPostExecute(Result)`. Metoda `doInBackground(Params...)` može da koristi metodu `publishProgress(Progress...)` za objavljuvanje progrusa izvršavanja operacije.
- `onProgressUpdate(Progress...)` – ova metoda se okida u okviru korisničke niti nakon poziva metode `publishProgress(Progress...)`. Tačno vreme izvršavanja ove metode nije definisano. Pošto se izvršava u korisničkoj niti, ona sme da pristupa korisnickom interfejsu i da prikaže u nekom obliku progres pozadinske operacije, poput progresne linije, procента ili slično.
- `onPostExecute(Result)` – metoda se poziva u okviru korisničke niti nakon završetka pozadinske operacije. Rezultat operacije joj se prosleđuje kao parametar.

Task se pokreće pozivom metode `execute(Params...)` iz korisničke niti. Tok jednog asinhronog taska prikazan je na slici 6.2. Jednom pokrenut task se po potrebi može otkazati pozivom `cancel(boolean)` metode. U tom slučaju će, umesto `onPostExecute()` metode, biti pozvana metoda `onCancelled()`. Status otkazivanja se može po potrebi proveriti pozivom metode `isCancelled()`.



Slika 6.2, dijagram toka `AsyncTask`-a

Kako bi se AsyncTask ispravno koristio, potrebno je pridržavati se nekoliko pravila:

- Instanca AsyncTask se mora kreirati u okviru korisničke niti.
- execute(Params...) se mora pozvati u okviru korisničke niti.
- Nije potrebno ručno pozivati metode onPreExecute, doInBackground, onProgressUpdate i onPostExecute.
- Task se može izvršiti samo jednom, svaki naknadni pokušaj izvršavanja istog taska će rezultirati izbacivanjem izuzetka.

Tipičan primer AsyncTacka jeste preuzimanje fajlova preko mreže. Ovakva operacija može da traje nepredviđeno dugo, zbog lošeg kvaliteta mreže. U slučaju da se ovakva operacija radi u okviru korisničke niti, za sve vreme dok traje preuzimanje fajlova korisnik ne bi mogao da vrši interakciju sa aplikacijom. Tipična implementacija preuzimanja fajlova data je u nastavku. Definisana je klasa ZadatakPreuzimanjeFajlova, koja proširuje AsyncTask. Parametri AsyncTacka dati su u obliku AsyncTask<URL, Integer, Long>. URL objekti su potrebni za preuzimanje fajlova, Integer parametar služi za objavljivanje progrusa operacije, a Long predstavlja tip povratnog rezultata. Ovaj primer implementacije podrazumeva da postoji implementirana klasa Downloader koji u sebi sadrži statičku metodu downloadFile(Uri), čijim pozivom se preuzima fajl na lokaciji zadatoj pomoću vrednosti Uri parametra. Takođe, pretpostavlja se da ova metoda kao povratnu vrednost vraća broj preuzetih bajtova.

```
private class ZadatakPreuzimanjeFajlova extends
AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int brojFajlova = urls.length;
        long ukupnaVelicina = 0;
        for (int i = 0; i < brojFajlova; i++) {
            ukupnaVelicina +=
        Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float)
brojFajlova) * 100));
            // prekinuti operaciju ukoliko je pozvana
metoda cancel()
            if (isCancelled()) break;
        }
        return ukupnaVelicina;
    }
    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }
    protected void onPostExecute(Long rezultat) {
        showDialog("Preuzeto " + rezultat + " bajtova");
    }
}
```

6.2. Pitanja za vežbu

- U koliko niti se u okviru svog procesa podrazumevano izvršava Android aplikacija?
- Šta je korisnička nit (main thread)? Šta se sve radi u okviru ove niti?
- Šta se dešava ukoliko se korisnička nit blokira? Detaljno objasniti slučaj da je nit na kratko blokirana (manje od 5 sekundi), i duže blokirana (duže od 5 sekundi).
- Šta je ANR dijalog?
- Da li se komponentama korisničkog interfejsa sme pristupati van korisničke niti?
- Koja su dva osnovna pravila po pitanju upotrebe korisničke niti, kojih se programeri obavezno moraju pridržavati?
- Navesti primere tipičnih operacija koje se izvršavaju u odvojenim nitima.
- Koje načine Android nudi drugim nitima za pristup korisničkoj niti?
- Objasniti koncept radne niti (worker thread).
- Koji je preporučeni način korišćenja niti u Androidu?
- Šta je AsyncTask i čemu služi?
- Objasniti detaljno svaki od genetičkih tipova definisanih zaglavljem AsyncTask<Params, Progress, Result>.
- Koji su osnovni koraci kroz koje AsyncTask prolazi? Koje metode im odgovaraju?
- Objasniti metodu onPreExecute().
- Objasniti metodu doInBackground().
- Objasniti metodu onPostExecute().
- Kako se objavljuje progres izvršavanja asinhronog zadatka?
- Da li je moguće prekinuti već pokrenuti AsyncTask?
- Da li je moguće reciklirati AsyncTask, odnosno, da li je moguće ponovo pokrenuti jednom izvršeni AsyncTask?
- U okviru koje niti se kreira instanca AsyncTask?
- Da li se metode onPreExecute, doInBackground, onProgressUpdate i onPostExecute pozivaju ručno ili ne?
- Navesti neke tipične primere kada je pogodno upotrebiti AsyncTask.

6.3. Zadaci za vežbu

Primer 1: Napraviti aplikaciju koja preuzima sliku sa Interneta upotrebom AsyncTask. Preuzimanje slike se inicira klikom na odgovarajuće dugme, a nakon preuzimanja slika se pokazuje na korisničkom interfejsu.

Rešenje: Korisnički interfejs se sastoji od jednog dugmeta kojim se inicira preuzimanje slike, i jednog ImageView objekta u koji će biti smeštena slika. Sadržaj activity_main.xml fajla je dat u nastavku:

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

    xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:orientation="vertical" >
    <Button
        android:id="@+id/buttonDownload"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Preuzmi sliku" />
    <ImageView
        android:id="@+id/imageView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:contentDescription="Ovde će biti prikazana
slika" />
</LinearLayout>
```

Pošto aplikacija treba da preuzme sliku sa Interneta, potrebna joj je odgovarajuća dozvola u manifestu aplikacije (android.permission.INTERNET). Sadržaj AndroidManifest.xml fajla dat je sa:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.example.student.AsyncTask">
    <uses-permission
        android:name="android.permission.INTERNET"/>
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
```

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

    <category
        android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
</application>
</manifest>
```

Aktivnost preuzima sliku preko AsyncTask-a, pošto se može desiti da ova akcija potraje. Ako bi se preuzimanje vršilo u okviru korisničke niti, moglo bi doći do blokiranja korisničkog interfejsa i ANR dijaloga. Kod MainActivity.java dat je sa:

```
package com.example.student.AsyncTask;

import android.app.AlertDialog;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.AsyncTask;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.ImageView;

import java.io.InputStream;
import java.net.HttpURLConnection;

public class MainActivity extends AppCompatActivity
    implements View.OnClickListener {
    private ImageView preuzetaSlika;
    private ProgressDialog dijalogZaCekanje;
    private Button buttonPreuzmiSliku;
    private static String url =
"https://repository.singidunum.ac.rs/images/2014/06/thumb/singidunum-logo.png";

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

        initComponents();
    }

    private void initComponents() {
```

```

        buttonPreuzmiSliku =
findViewById(R.id.buttonDownload);
        preuzetaSlika = findViewById(R.id.imageView);
        buttonPreuzmiSliku.setOnClickListener(this);
    }

@Override
public void onClick(View v) {
    new TaskPreuzimanjeSlike().execute(url);
}

private class TaskPreuzimanjeSlike extends
AsyncTask<String,Void,Bitmap> {
    @Override
    protected Bitmap doInBackground(String... param) {
        try{
            java.net.URL url = new
java.net.URL(param[0]);
            HttpURLConnection connection =
(HttpURLConnection) url.openConnection();
            connection.setDoInput(true);
            connection.connect();
            InputStream input =
connection.getInputStream();
            Bitmap myBitmap =
BitmapFactory.decodeStream(input);
            input.close();
            return myBitmap;
        } catch (Exception e) {
            // rukovanje izuzetkom
        }
        return null;
    }
    @Override
    protected void onPreExecute() {
        dijalogZaCekanje =
ProgressDialog.show(MainActivity.this,
                    "Cekanje", "Slika se preuzima");
    }

    @Override
    protected void onPostExecute(Bitmap result) {
        preuzetaSlika.setImageBitmap(result);
        dijalogZaCekanje.dismiss();
    }
}
}

```

Asinhroni task je definisan u obliku `AsyncTask<String,Void,Bitmap>`. Prvi parametar je `String`, kroz koji se prosleđuje url adresa na kojoj se slika za preuzimanje nalazi. Drugi parametar je `Void`, pošto se progres ne objavljuje, već se samo vrti dijalog za čekanje. Treći parametar je tipa `Bitmap`, pošto se slika nakon preuzimanja dekoduje u `Bitmap` i tako vraća kao rezultat.

Nakon pokretanja aplikacije, korisniku se prikazuje interfejs kao na slici 6.3 (levo). Klikom na dugme za preuzimanje slike se inicira asinhrono preuzimanje pomoću prethodno definisanog `AsyncTask`-a. Dok se slika preuzima korisniku se prikazuje dijalog za čekanje, koji je prikazan na slici 6.3 u sredini. Nakon preuzimanja slike, ona se smešta u `ImageView` i prikazuje korisniku (6.3 desno). Lokacija odakle se slika preuzima zadata je statičkom `String` promenljivom `url`. Ukoliko korisnik želi da preuzme neku drugu sliku, dovoljno je staviti njen URL u ovu promenljivu.



Slika 6.3, Preuzimanje slike pomoću `AsyncTask`

Primer 2:

Implementirati Android aplikaciju koja će preko `AsyncTask`-a kontaktirati API na adresi <http://199.188.100.46/singidunum/android/faculties.json>. GET zahtev na ovu adresu vraća listu fakulteta u obliku JSON fajla, koji je prikazan sledećim listingom:

```
[ {
    "name": "Fakultet za informatiku i računarstvo",
    "acronym": "FIR",
    "website": "http://fir.singidunum.ac.rs"
}, {
    "name": "Poslovni fakultet u Beogradu",
    "acronym": "PFB",
    "website": "http://pfb.singidunum.ac.rs"
}, {
    "name": "Fakultet za turistički i hotelijerski
menadžment",
    "acronym": "FTHM",
    "website": "http://fthm.singidunum.ac.rs"
}, {
    "name": "Tehnički fakultet",
    "acronym": "TF",
    "website": "http://tf.singidunum.ac.rs"
} ]
```

Kada se dohvati lista fakulteta, potrebno je parsirati JSON u domenske objekte (klasa Faculty), i nakon toga prikazati korisniku na ekranu.

Rešenje: Korisnički interfejs aplikacije treba da sadrži jedno dugme, čijim klikom se asinhrono učitava lista fakulteta sa API-ja. Osim dugmeta, potrebno je i jedno tekstualno polje za prikaz dovućenih rezultata. Kod activity_main.xml fajla dat je sa:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
    android:layout_width="match_parent"
        android:layout_height="match_parent">
    <Button
        android:id="@+id/buttonFaculties"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Dohvati listu fakulteta"/>
    <TextView
        android:id="@+id/labelFaculties"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="15dp"
        android:lines="20" />
</LinearLayout>
```

Pošto je za kontaktiranje API-ja neophodno da aplikacija ima pristup internet, mora se zahtevati odgovarajuća dozvola u okviru AndroidManifest.xml fajla:

```

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

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

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

```

Dohvatanje liste fakulteta se odvija u odvojenoj niti u okviru AsyncTask. Za komunikaciju između više niti može se koristiti objekat klase Handler. Putem Handler objekta se iz jedne niti može zakazati akcija koju je potrebno izvršiti u drugoj niti. Handler se zasniva na mehanizmu prosleđivanja poruka. U ovom primeru se koristi naša klasa ReadDataHandler, koja je izvedena iz Handler klase. Proširena klasa dodatno sadrži polje kroz koje će se glavnoj korisničkoj niti proslediti odgovor dobijen od servera. Sadržaj ReadDataHandler.java fajla je dat u nastavku:

```

package com.example.student.myapplication;

import android.os.Handler;

public class ReadDataHandler extends Handler {
    private String json;

    public String getJson() {
        return json;
    }
}

```

```
public void setJson(String json) {
    this.json = json;
}
}
```

Dalje, potrebno je implementirati klasu koja će predstavljati domenske objekte. Podaci iz JSON-a nisu previše laki za manipulaciju ukoliko bi se čuvali samo kao Stringovi. Potrebna je klasa čiji će objekat sadržati sva polja koja se nalaze u primljenom JSON-u, pošto se objektima u Java kodu mnogo lakše manipuliše. Klasa Faculty.java predstavlja domenske objekte, a dodatno pruža i uslužne metode za parsiranje jednog fakulteta, kao i parsiranje niza fakulteta. Kod ove klase dat je u nastavku:

```
package com.example.student.myapplication;

import org.json.JSONArray;
import org.json.JSONObject;
import java.util.LinkedList;

public class Faculty {

    private String name, acronym, website;

    public Faculty() {

    }

    public Faculty(String name, String acronym, String website) {
        this.name = name;
        this.acronym = acronym;
        this.website = website;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAcronym() {
        return acronym;
    }

    public void setAcronym(String acronym) {
        this.acronym = acronym;
    }
}
```

```

}

public String getWebsite() {
    return website;
}

public void setWebsite(String website) {
    this.website = website;
}

//parsiranje jednog objekta
public static Faculty parseJSONObject (JSONObject
object){
    Faculty faculty = new Faculty();
    try {
        if (object.has("name")) {
            faculty.setName(object.getString("name"));
        }
        if (object.has("acronym")) {

faculty.setAcronym(object.getString("acronym"));
        }
        if (object.has("website")){
            faculty.setWebsite(object.getString("website"));
        }
    }
    catch (Exception e) {

    }
    return faculty;
}

//parsiranje niza objekata, koje se zapravo zasniva na
iterativnom parsiranju pojedinacnih objekata
public static LinkedList<Faculty> parseJSONArray
(JSONArray array){
    LinkedList<Faculty> list = new LinkedList<>();
    try {
        for (int i = 0; i < array.length(); i++) {
            Faculty faculty =
parseJSONObject(array.getJSONObject(i));
            list.add(faculty);
        }
    }
    catch (Exception e){ }
    return list;
}
}

```

Za komunikaciju sa API-jem kreira se klasa Api, koja će kroz poziv metode getJSON() kreirati asinhroni poziv ka serveru kako bi se dohvatala lista fakulteta. Dohvaćeni podaci se smeštaju u prosleđeni parametar ReadDataHandler, koji će moći da bude procitan u korisničkoj niti. Kod klase Api.java dat je u nastavku:

```
package com.example.student.myapplication;

import android.os.AsyncTask;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class Api {

    public static void getJSON (String url, final
ReadDataHandler rdh){
        AsyncTask<String,Void,String> task = new
AsyncTask<String, Void, String>() {
            @Override
            protected String doInBackground(String...
strings) {
                String response = "";

                //treba nam http konekcija
                try {
                    URL link = new URL(strings[0]);
                    HttpURLConnection con =
(HttpURLConnection) link.openConnection();

                    BufferedReader br = new
BufferedReader(new
InputStreamReader(con.getInputStream()));
                    //cita se red po red iz br
                    String red;
                    while ((red = br.readLine())!=null){
                        response += red + "\n";
                    }

                    br.close();
                    con.disconnect(); // ne mora se pisati
- Android sam ubija konekciju

                } catch (Exception e){
                    response = "[ ]";
                }
            return response;
        }
    }
}
```

```

        }

    @Override
    protected void onPostExecute(String response) {
        rdh.setJson(response);
        rdh.sendEmptyMessage(0);
    }
};

task.execute(url);
}

}

```

Na kraju, u glavnoj aktivnosti se klikom na dugme okida poziv statičke metode Api klase getJSON(), kojoj se prosleđuje objekat ReadDataHandler, kroz koji će stići odgovor kada podaci budu asinhrono skinuti sa servera, pri čemu se koristi mehanizam poruka. Kod MainActivity.java dat je u nastavku:

```

package com.example.student.myapplication;

import android.os.Message;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

import org.json.JSONArray;

import java.util.LinkedList;

public class MainActivity extends AppCompatActivity
implements View.OnClickListener{

    private Button buttonFaculties;
    private TextView labelFaculties;

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

        initComponents();
    }

    private void initComponents() {

```

```

        buttonFaculties =
findViewById(R.id.buttonFaculties);
        labelFaculties = findViewById(R.id.labelFaculties);
        buttonFaculties.setOnClickListener(this);
    }

    @Override
    public void onClick(View view) {

Api.getJSON("http://199.188.100.46/singidunum/android/faculties.json", new ReadDataHandler() {
    @Override
    public void handleMessage(Message msg) {
        String odgovor = getJson();
        try {
            JSONArray array = new
JSONArray(odgovor);
            LinkedList<Faculty> faculties =
Faculty.parseJSONArray(array);

labelFaculties.setText("Fakulteti\n\n");
            for (Faculty f : faculties) {
                String prikaz = "[" +
f.getAcronym() + "] " + f.getName() + "\n" + f.getWebsite() +
"\n\n";
                labelFaculties.append(prikaz);
            }
        } catch (Exception e) {
        }
    }
});

labelFaculties.setText("Ucitava se...");

}
}

```

Metoda handleMessage() objekta tipa ReadDataHandler se poziva kao odgovor na poziv metode sendEmptyMessage() u okviru onPostExecute() metode AsyncTask-a, čime se signalizira da je odgovor servera spreman i da se nalazi u polju json ReadDataHandler objekta. Dalje, u kodu handleMessage() metode se parsira dobijeni odgovor pozivom uslužnih metoda iz Faculty klase, a parsirani

odgovor smešta u objekte tipa Faculty. Na kraju se parsirana lista objekata prikazuje na korisničkom interfejsu, kao što je prikazano na slici 6.4.



Slika 6.4, Prikaz dohvaćene liste fakulteta preko AsyncTask-a

7. Dozvole

Privatnost korisnika je jedan od najbitnijih sigurnosnih aspekata svakog sistema. Mobilni uređaji su naročito osetljivi, pošto korisnici na njima čuvaju veliki broj privatnih informacija, poput liste kontakata, poruka, slika, video zapisa, dokumenata i slično. Pri tome, korisnici svakodnevno preuzimaju veliki broj aplikacija i instaliraju ih na te uređaje. Ukoliko ne bi postojala nikakva sigurnosna provera, svaka instalirana aplikacija bi mogla da pristupi svim korisničkim podacima koji se nalaze na uređaju, i tako bi drastično ugrozila privatnost korisnika. Zbog toga je u Android sistemu uveden sistem dozvola, odnosno privilegija (engl. *permissions*), koji služi da se aplikacijama ograniči pristup privatnim podacima korisnika.

Svrha sistema dozvola u Androidu jeste da se zaštiti privatnost korisnika. Svaka Android aplikacija mora da traži dozvolu ukoliko želi da pristupi osetljivim korisničkim podacima, poput liste kontakata, lokacije korisnika ili SMS poruka. Neki delovi i funkcionalnosti Android sistema su takođe zaštićeni, pa tako aplikacija mora da traži dozvolu i za pristup Internetu ili kameri. U zavisnosti od posmatrane funkcionalnosti, Android sistem može ili da automatski da dozvoli aplikaciji, ili da zahteva od korisnika da potvrди (ili odbije) davanje dozvole.

Sigurnosna arhitektura platforme je zasnovana na tome da nijedna aplikacija nema podrazumevanu dozvolu za izvršavanje bilo koje operacije koja bi na bilo koji način mogla da ugrozi korisnika, druge aplikacije koje se nalaze na uređaju ili sam operativni sistem.

7.1. Tipovi dozvola

Ukoliko se posmatraju navedeni primeri iz uvoda ovog poglavlja, može se uočiti da nisu sve akcije podjednako rizične po korisnika. Na primer, mnogo je gore da aplikacija ima direktni pristup SMS porukama i kontaktima korisnika, nego pristup Internetu. Zbog toga i sam Android dozvole za ove akcije tretira drugačije. Sistemske dozvole su podeljene u dve osnovne grupe, na osnovu rizika koji predstavljaju po privatnost korisnika:

- Normalne dozvole – dozvole koje ne predstavljaju preveliki rizik po pitanju privatnosti korisnika ili Android sistema. Ovakve dozvole sistem automatski dodeljuje aplikaciji, ukoliko su specificirane u manifestu aplikacije.
- Opasne dozvole – dozvole koje mogu direktno ugroziti privatnost korisnika (direktni pristup porukama, kontaktima, kalendaru i slično), ili

mogu ugroziti normalan rad Android sistema. Ovakve dozvole korisnik mora eksplicitno prihvatići, iako su navedene u manifestu aplikacije.

Pregled normalnih dozvola dat je u tabeli 7-1, dok je potpuna lista opasnih dozvola data u tabeli 7-2.

Normalne dozvole		
ACCESS_LOCATION_EXTRA_COMMANDS	GET_PACKAGE_SIZE	SET_ALARM
ACCESS_NETWORK_STATE	INSTALL_SHORTCUT	SET_TIME_ZONE
ACCESS_NOTIFICATION_POLICY	INTERNET	SET_WALLPAPER
ACCESS_WIFI_STATE	KILL_BACKGROUND_PROCESSES	SET_WALLPAPER_HINTS
BLUETOOTH	MODIFY_AUDIO_SETTINGS	TRANSMIT_IR
BLUETOOTH_ADMIN	NFC	UNINSTALL_SHORTCUT
BROADCAST_STICKY	READ_SYNC_SETTINGS	USE_FINGERPRINT
CHANGE_NETWORK_STATE	READ_SYNC_STATS	VIBRATE
CHANGE_WIFI_MULTICAST_STATE	RECEIVE_BOOT_COMPLETED	WAKE_LOCK
CHANGE_WIFI_STATE	REORDER_TASKS	WRITE_SYNC_SETTINGS
DISABLE_KEYGUARD	REQUEST_IGNORE_BATTERY_OPTIMIZATIONS	
EXPAND_STATUS_BAR	REQUEST_INSTALL_PACKAGES	

Tabela 7-1, pregled normalnih dozvola

Opasne dozvole			
Grupa	Dozvola	Grupa	Dozvola
CALENDAR	READ_CALENDAR	PHONE	READ_PHONE_STATE
	WRITE_CALENDAR		CALL_PHONE
CAMERA	CAMERA		READ_CALL_LOG
CONTACTS	READ_CONTACTS	SMS	WRITE_CALL_LOG
	WRITE_CONTACTS		ADD_VOICEMAIL
	GET_ACCOUNTS		USE_SIP
LOCATION	ACCESS_FINE_LOCATION		PROCESS_OUTGOING_CALLS
	ACCESS_COARSE_LOCATION		SEND_SMS
MICROPHONE	RECORD_AUDIO		RECEIVE_SMS
STORAGE	READ_EXTERNAL_STORAGE		READ_SMS
	WRITE_EXTERNAL_STORAGE		RECEIVE_WAP_PUSH
SENSORS	BODY_SENSORS		RECEIVE_MMS

Tabela 7-2, Lista opasnih dozvola

Ovde je potrebno napomenuti da u budućnosti nove verzije Android SDK mogu da premeste neku od dozvola iz jedne grupe u drugu, pa ne treba bazirati logiku aplikacije na datoј strukturi grupa. Dalje, dozvole READ_EXTERNAL_STORAGE i WRITE_EXTERNAL_STORAGE spadaju u istu grupu, ali pravo čitanja ne garantuje i pravo upisa, dok pravo upisa podrazumeva i pravo čitanja (aplikacija koja ima WRITE dozvolu automatski dobija i READ, obrnuto ne važi).

7.2. Deklaracija privilegija u manifestu

Aplikacija mora javno deklarisati sve dozvole koje su joj potrebne (i normalne i opasne) u manifestu aplikacije. Za to služi element <uses-permission>, u okviru kojeg se eksplicitno navodi svaka dozvola pojedinačno. Na primer, aplikacija koja želi da koristi dozvolu za slanje SMS poruka bi imala sledeći sadržaj manifesta:

```
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    ...>

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

    <application ...>
        ...
    </application>
</manifest>
```

Aplikacija može tražiti i više dozvola. Na primer, aplikacija kojoj je potrebno da koristi Internet i da pristupa eksternim fajlovima bi imala sledeći sadržaj manifesta:

```
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    ...>
    <uses-permission
        android:name="android.permission.INTERNET"/>
    <uses-permission
        android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <!-- dodatne dozvole navesti ovde -->

    <application ...>
        ...
    </application>
</manifest>
```

Sve normalne dozvole koje su navedene u manifestu Android sistem će automatski dodeliti aplikaciji. Sa druge strane, svaku opasnu dozvolu navedenu u manifestu korisnik mora eksplisitno potvrditi.

7.3. Potvrda opasnih dozvola

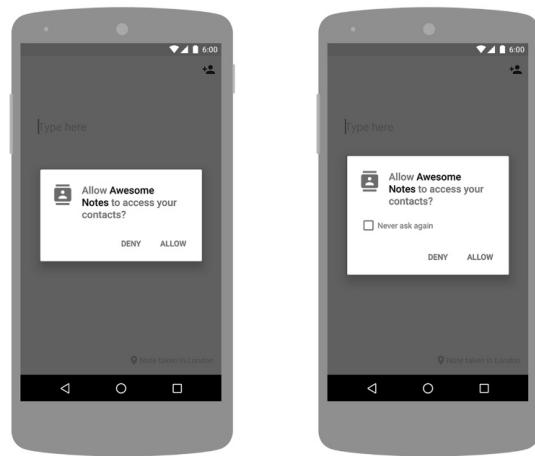
Korisnik mora eksplisitno potvrditi opasne dozvole, bez obzira što su deklarisane u manifestu, nezavisno od verzije Android sistema. Međutim, način na koji Android zahteva od korisnika da potvrdi opasne dozvole zavisi od verzije Androida koji se izvršava na uređaju, kao i od ciljnog SDK aplikacije. Promena ponašanja je uvedena sa Android 6.0 (API nivo 23), tako da razlikujemo sledeća dva slučaja:

- Potvrda za vreme izvršavanja aplikacije (Android 6.0 i noviji),
- Potvrda za vreme instalacije aplikacije (Android 5.1.1 i stariji).

7.3.1. Potvrda za vreme izvršavanja

Jedna od novih funkcionalnosti uvedenih sa Android 6.0 (API nivo 23) je dinamička dodata dozvola. Na uređajima sa Android 6.0 ili novijom verzijom, prilikom instalacije aplikacije se ne traži od korisnika da dodeli dozvole. Aplikacija mora da traži opasne dozvole koje su joj potrebne za vreme izvršavanja (slika 7.1). Korisniku se prikazuje dijalog u kome mu se navodi kojoj grupi dozvola aplikacija želi da pristupi, uz opcije da korisnik prihvati ili odbije davanje dozvole (leva slika). Ukoliko korisnik odbije da dodeli dozvolu aplikaciji, sledeći put kad aplikacija bude tražila dozvolu u dijalogu će biti ponuđena još jedna opcija kroz dodatni CheckBox, koji je indikacija da korisnik ne želi više da bude pitan za ovu dozvolu (desna slika).

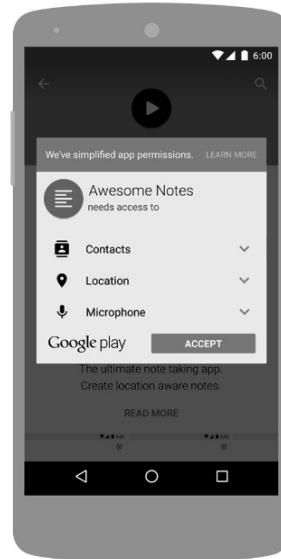
Zbog činjenice da korisnik može u bilo kom trenutku da dodeli ili oduzme dozvole za svaku pojedinačnu aplikaciju kroz sistemska podešavanja, nije moguće oslanjati se na to da aplikacija uvek ima određenu dozvolu koja joj je potrebna, čak i u slučaju da je korisnik inicijalno odobrio tu dozvolu. Kako bi se obezbedio ispravan rad aplikacije, neophodno je uvek proveravati i po potrebi tražiti dozvole za vreme izvršavanja aplikacije. Ukoliko korisnik uskrati bilo koju potrebnu dozvolu u nekom trenutku, aplikacija može da nastavi da radi i dalje, uz ograničenu funkcionalnost.



Slika 7.1, dijalog traženja dozvola za vreme izvršavanja
(izvor: <https://developer.android.com/guide/topics/permissions/overview>)

7.3.2. Potvrda za vreme instalacije

Na starijim verzijama Android sistema, odnosno od verzije Android 5.1.1 (API nivo 22) i starijim, ili ukoliko je ciljna verzija SDK 22 ili niža, sistem zahteva od korisnika da dodeli sve opasne dozvole aplikaciji već prilikom instalacije (slika 7.2).



Slika 7.2, dijalog traženja dozvola za vreme instalacije
(izvor: <https://developer.android.com/guide/topics/permissions/overview>)

Korisnik mora da prihvati sve dozvole da bi se aplikacija uspešno instalirala. Ukoliko korisnik odbije da dodeli dozvole, sistem neće dozvoliti instalaciju.

7.4. Provera dozvola

U slučaju opasnih dozvola, aplikacija mora da proveri da li poseduje dozvolu svaki put kada pokuša rizičnu operaciju. Počev od Android 6.0, korisnici mogu ukinuti prethodno date dozvole bilo kojoj aplikaciji u bilo kom trenutku, tako da aplikacija mora da proveri da li još uvek ima dozvolu bez obzira na to što je prethodno radila kako treba. Za to se koristi metoda ContextCompat.checkSelfPermission(). Na primer, deo koda koji pokazuje kako se proverava da li aktivnost ima dozvolu da koristi kameru dat je u nastavku:

```
if (ContextCompat.checkSelfPermission(thisActivity,
Manifest.permission.CAMERA)
    != PackageManager.PERMISSION_GRANTED) {
    // Dozvola nije dodeljena, potrebno je traziti od
korisnika
}
```

Ukoliko aplikacija trenutno ima potrebnu dozvolu, metoda vraća PERMISSION_GRANTED, a aplikacija može neometano da nastavi sa radom. U suprotnom slučaju, ukoliko aplikacija trenutno nema potrebnu dozvolu, mora eksplicitno pitati korisnika da je dodeli.

7.5. Zahtevanje dozvola

Ukoliko je prilikom provere da li aplikacija ima određenu dozvolu pozivom metode.checkSelfPermission() dobijen rezultat PERMISSION_DENIED, aplikacija mora da pita korisnika da joj dodeli tu dozvolu. To se postiže pozivom metode requestPermissions(), koja će prikazati standardni Android dijalog u kome se traži potrebna dozvola.

Ponekad je potrebno i dodatno objasniti korisniku zašto aplikacija traži određenu dozvolu. Na primer, ako posmatramo neku aplikaciju koja se bavi fotografijom, korisniku neće biti čudno ukoliko ta aplikacija prilikom pokretanja traži dozvolu za upotrebu kamere. Međutim, korisniku možda neće biti jasno zašto ta ista aplikacija traži pristup kontaktima, listi poziva ili lokaciji. Prilikom traženja dozvole, moguće je dati korisniku objašnjenje zašto je ta dozvola potrebna, ali uz oprez da korisnika ne treba zatravljati objašnjenjima (da ne bi došlo do suprotnog efekta, da korisnik izgubi želju da koristi našu aplikaciju).

Preporučeni pristup od strane Android zajednice je da se dodatno objašnjenje ponudi samo u slučaju da je korisnik prethodno odbio zahtev da dodeli određenu dozvolu. U tom slučaju koristi se metoda `shouldShowRequestPermissionRationale()`, koja će vratiti true u slučaju da je korisnik odbio prethodni zahtev, a false u slučaju da je korisnik odbio prethodni zahtev i dodatno odabralo opciju **Don't ask again**, kao što je prikazano na slici 7.1 desno.

Prilikom poziva metode `requestPermissions()`, prosleđuju se dozvole koje aplikacija želi, kao i celobrojna vrednost koja označava kod zahteva (engl. *request code*), kako bi bilo moguće jednoznačno identifikovati taj zahtev. Ovaj poziv se odvija asinhrono, odnosno, vratiće se odmah kontrola mestu poziva. Kada korisnik odgovori na zahtev, Android sistem poziva callback metodu `onRequestPermissionsResult()` sa rezultatom, pri čemu prosleđuje i isti kod zahteva koji je prosleđen u `requestPermissions()`. Proces zahtevanja dozvole ilustrovan je sledećim primerom, gde je aplikaciji potrebna dozvola za čitanje liste kontakata. Logika je praktično identična i za sve druge opasne dozvole. Prvo se proverava da li aplikacija ima potrebnu dozvolu. Ukoliko nema, proverava se da li je potrebno prikazati dodatno objašnjenje korisniku zbog čega aplikacija traži tu dozvolu, a ukoliko objašnjenje nije potrebno, traži se dozvola. Prilikom prikazivanja dijaloga, sistem navodi grupu dozvola kojoj aplikacija traži pristup, a ne specifičnu dozvolu.

```
// ovde se this odnosi na trenutnu aktivnost
if (ContextCompat.checkSelfPermission(this,
        Manifest.permission.READ_CONTACTS)
    != PackageManager.PERMISSION_GRANTED) {
    // Aktivnost nema dozvolu, potrebno je traziti od
    // korisnika
    // Da li treba pokazati i objasnjenje?
    if
        (ActivityCompat.shouldShowRequestPermissionRationale(this,
            Manifest.permission.READ_CONTACTS)) {
            // Prikazati obavestenje korisniku asinhrono - bez
            // blokiranja korisnicke niti dok se ceka odgovor od korisnika
            // Kada korisnik vidi objasnjenje, zahtevati ponovo
            potrebnu dozvolu
        } else {
            // Nije potrebno objasnjenje, zahteva se dozvola
        ActivityCompat.requestPermissions(this,
            new
String[] {Manifest.permission.READ_CONTACTS},
            ZAHTEV_READ_CONTACTS);
            // ZAHTEV_READ_CONTACTS se ovde odnosi na
            // int konstantu. Callback metoda dobija rezultat
            zahteva}
```

```

        // pod ovim kodom.
    }
} else {
    // Aktivnost vec poseduje dozvolu
}

```

Kao što je ranije navedeno, ovaj poziv je asinhron. U nekom trenutku, korisnik će odgovoriti na zahtev za davanje dozvole aplikaciji kroz dijalog koji će mu Android sistem prikazati na ekranu. Nakon odgovora korisnika, Android sistem poziva callback metodu onRequestPermissionsResult() sa rezultatom odgovora i istim kodom zahteva koji je prosleđen u metodi requestPermissions(). Aktivnost mora nadjačati ovu metodu i proveriti da li je dozvola dodeljena pre nego što pokuša da koristi zaštićenu funkcionalnost. Ukoliko dozvola nije dodeljena, aplikacija može da nastavi sa radom, ali sa umanjenom funkcionalnosti. Na primer, ova metoda bi u prethodnom primeru mogla da se nadjača na sledeći način:

```

@Override
public void onRequestPermissionsResult(int requestCode,
    String[] permissions, int[] grantResults) {
    switch (requestCode) {
        case ZAHTEV_READ_CONTACTS: {
            // Ukoliko je zahtev otkazan, rezultujući niz
            // je prazan
            if (grantResults.length > 0
                && grantResults[0] ==
PackageManager.PERMISSION_GRANTED) {
                // Dozvola je odobrena, moze se uraditi
                // potreban posao sa kontaktima.
            } else {
                // Dozvola je odbijena! Onesposobiti
                // funkcionalnost koja zavisi od ove
                // dozvole.
            }
            return;
        }
        // drugi 'case' slucajеви ukliko
        // aplikacija trazi i neke druge dozvole.
    }
}

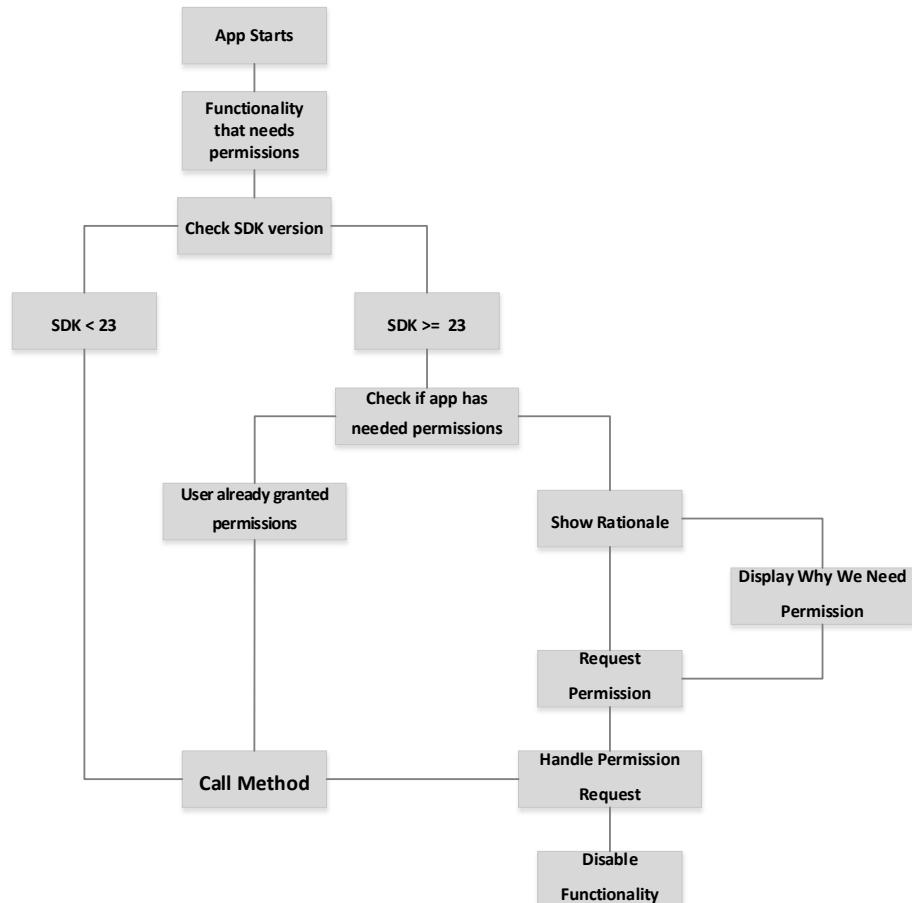
```

Jedna aplikacija se može izvršavati na različitim tipovima uređaja sa različitim verzijama Android sistema. Ukoliko se ta aplikacija izvršava na Android 5.1.1 ili starijem, nije potrebno pitati za dozvole. Ta ista aplikacija mora da pita za dozvole ukoliko se izvršava na Android 6.0 ili novijem. Zbog toga se u kodu mora uvesti provera da li je potrebno zahtevati dozvole ili ne, na osnovu verzije

Android sistema na kom se aplikacija trenutno izvršava. To se postiže vrlo jednostavno, proverom trenutne verzije Android sistema (Build.VERSION.SDK_INT) i poređenjem da li je veća od Android 6.0 Marshmallow (Build.VERSION_CODES.M). Jednostavan primer metode koja vraća true u slučaju da je potrebno dinamički tražiti dozvole, odnosno false ukoliko nije potrebno, dat je sa:

```
public boolean shouldAskPermission() {  
    return (Build.VERSION.SDK_INT >=  
Build.VERSION_CODES.M);  
}
```

Kompletna opisana logika procesa zahtevanja dozvola se može sumirati dijagramom na slici 7.3.



Slika 7.3, dijagram procesa zahtevanja dozvola

7.6. Pitanja za vežbu

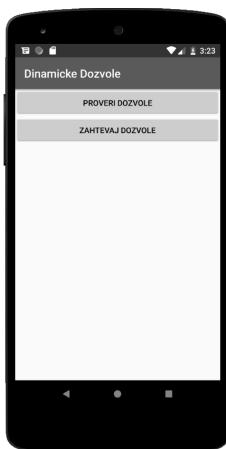
- Čemu služi sistem dozvola u Androidu?
- Gde se dozvole koje su potrebne aplikaciji deklarišu?
- Koja je razlika između opasnih i normalnih dozvola?
- Navesti neke karakteristične primere normalnih dozvola.
- Navesti neke karakteristične primere opasnih dozvola.
- Šta se smatra za osetljive podatke korisnika?
- Koja je osnovna razlika između modela dozvola koji je uveden u Android 6.0 u odnosu na prethodni model?
- Da li korisnik može da povuče prethodno dodeljene dozvole nekoj aplikaciji? Ukoliko može, kako aplikacija treba da reaguje na to?
- Čemu služi metoda ContextCompat.checkSelfPermission()?
- Čemu služi metoda shouldShowRequestPermissionRationale()?
- U kojim slučajevima treba objasniti korisniku aplikacije zašto je potrebna određena dozvola?
- Opisati ceo proces provere da li aplikacija ima određenu dozvolu.
- Opisati proces zahtevanja određene dozvole.
- Kroz koju callback metodu sistem vraća odgovor korisnika na zahtev za dozvolom? Kako se tipično implementira ova metoda?
- Šta su grupe dozvola?
- Da li prilikom dodele jedne dozvole iz grupe, sistem automatski dodeljuje i ostale dozvole iz te grupe?
- Ukoliko aplikacija treba da podrži i starije uređaje, koji ne zahtevaju dinamičku proveru dozvola, kao i novije, opisati kako se programski rešava taj problem.

7.7. Zadaci za vežbu

Primer 1: Kreirati aplikaciju koja demonstrira rad sa dinamičkim dozvolama. Aplikacija treba dinamički da zahteva opasne dozvole – pristup kameri i pristup lokaciji korisnika. Korisnički interfejs treba da sadrži dva dugmeta, prvo dugme služi za proveru da li aplikacija ima potrebne dozvole, a pomoću drugog dugmeta aplikacija zahteva potrebne dozvole.

Rešenje: Korisnički interfejs se sastoji od dva dugmeta, koja služe za proveru trenutnog statusa dozvola, kao i za zahtevanje potrebnih dozvola. Korisnički interfejs prikazan je na slici 7.4, a implementiran je u activity_main.xml fajlu:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
    android:layout_width="match_parent"
        android:layout_height="match_parent">
    <Button
        android:id="@+id/buttonProveriDozvole"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Proveri dozvole"/>
    <Button
        android:id="@+id/buttonZahtevajDozvole"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Zahtevaj dozvole"/>
</LinearLayout>
```



Slika 7.4, korisnički interfejs aplikacije koja demonstrira dinamičku dodelu dozvola

U manifestu aplikacije potrebno je naglasiti koje dozvole su aplikaciji potrebne (u ovom slučaju pristup kameri i lokaciji uređaja). Potrebno je naglasiti i opasne i normalne dozvole. U ovom primeru obe navedene dozvole su opasne. Sadržaj AndroidManifest.xml fajla je dat sa:

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

    <uses-permission
        android:name="android.permission.CAMERA" />
    <uses-permission
        android:name="android.permission.ACCESS_FINE_LOCATION" />

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

                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

U kodu aktivnosti se radi provera dozvola, kao i zahtevanje istih na standardan način opisan u poglavljima 7.4 i 7.5. Ukoliko klijent ne dodeli dozvole, putem dijaloga se prikazuje odgovarajuća poruka. Kod MainActivity.java klase je dat sa:

```
package com.example.student.dinamickedozvole;
import android.Manifest;
import android.content.DialogInterface;
import android.content.pm.PackageManager;
import android.os.Build;
import android.support.v4.app.ActivityCompat;
import android.support.v4.content.ContextCompat;
import android.support.v7.app.AlertDialog;
import android.support.v7.app.AppCompatActivity;
```

```

import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity
implements View.OnClickListener{

    //kod pod kojim se zahtev salje
    //i pod kojim ce stici odgovor
    private static final int KOD_ZAHTEVA_DOZVOLE = 1;

    private Button buttonProveriDozvole;
    private Button buttonZahtevajDozvole;

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

        initComponents();
    }

    private void initComponents() {
        buttonProveriDozvole =
findViewById(R.id.buttonProveriDozvole);
        buttonZahtevajDozvole =
findViewById(R.id.buttonZahtevajDozvole);
        buttonProveriDozvole.setOnClickListener(this);
        buttonZahtevajDozvole.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.buttonProveriDozvole:
                if (checkPermission()) {
                    Toast.makeText(this, "Dozvole su vec
dodeljene.", Toast.LENGTH_LONG).show();
                } else {
                    Toast.makeText(this, "Potrebno je
zahtevati dozvole posto nisu dodeljene.",
                    Toast.LENGTH_LONG).show();
                }
                break;
            case R.id.buttonZahtevajDozvole:
                if (!checkPermission()) {

```

```

        requestPermission();
    } else {
        Toast.makeText(this, "Dozvole su vec
dodeljene", Toast.LENGTH_LONG).show();
    }
    break;
}

}

private boolean checkPermission() {
    int resultLocation =
ContextCompat.checkSelfPermission(getApplicationContext(),
Manifest.permission.ACCESS_FINE_LOCATION);
    int resultCamera =
ContextCompat.checkSelfPermission(getApplicationContext(),
Manifest.permission.CAMERA);

    return resultLocation ==
PackageManager.PERMISSION_GRANTED && resultCamera ==
PackageManager.PERMISSION_GRANTED;
}

private void requestPermission() {

    ActivityCompat.requestPermissions(this, new
String[]{Manifest.permission.ACCESS_FINE_LOCATION,
Manifest.permission.CAMERA}, KOD_ZAHTEVA_DOZVOLE);

}

@Override
public void onRequestPermissionsResult(int requestCode,
String permissions[], int[] grantResults) {
    switch (requestCode) {
        case KOD_ZAHTEVA_DOZVOLE:
            if (grantResults.length > 0) {

                boolean locationAccepted =
grantResults[0] == PackageManager.PERMISSION_GRANTED;
                boolean cameraAccepted =
grantResults[1] == PackageManager.PERMISSION_GRANTED;

                if (locationAccepted && cameraAccepted)
                    Toast.makeText(this, "Dozvole
dodeljene, moze se pristupiti lokaciji i kameri. ",
Toast.LENGTH_LONG).show();
            else {

```

```

        Toast.makeText(this, "Dozvole nisu
dodeljene, nije moguc pristup lokaciji i kameri",
Toast.LENGTH_LONG).show();

        if (Build.VERSION.SDK_INT >=
Build.VERSION_CODES.M) {
            if
(shouldShowRequestPermissionRationale(
Manifest.permission.ACCESS_FINE_LOCATION)) {
                showMessageOKCancel("Potrebno je dodeliti obe
dozvole",
                    new DialogInterface.OnClickListener() {
                        @Override
                        public void onClick(DialogInterface dialog, int
which) {
                            if
(Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
                                requestPermissions(new
String[]{Manifest.permission.ACCESS_FINE_LOCATION,
Manifest.permission.CAMERA}, KOD_ZAHTEVA_DOZVOLE);
                            }
                        }
                    });
            }
            return;
        }
    }

}
break;
}

}

private void showMessageOKCancel(String message,
DialogInterface.OnClickListener okListener) {
    new AlertDialog.Builder(MainActivity.this)
        .setMessage(message)
        .setPositiveButton("OK", okListener)
        .setNegativeButton("Cancel", null)
        .create()
        .show();
}
}

```

Prilikom pokretanja aplikacije i pritiska na dugme za proveru dozvola, na ekranu će se pojaviti Toast sa obaveštenjem da aplikacija nema potrebne dozvole, odnosno da je neophodno zahtevati ih, kao što je prikazano na slici 7.5. Ukoliko

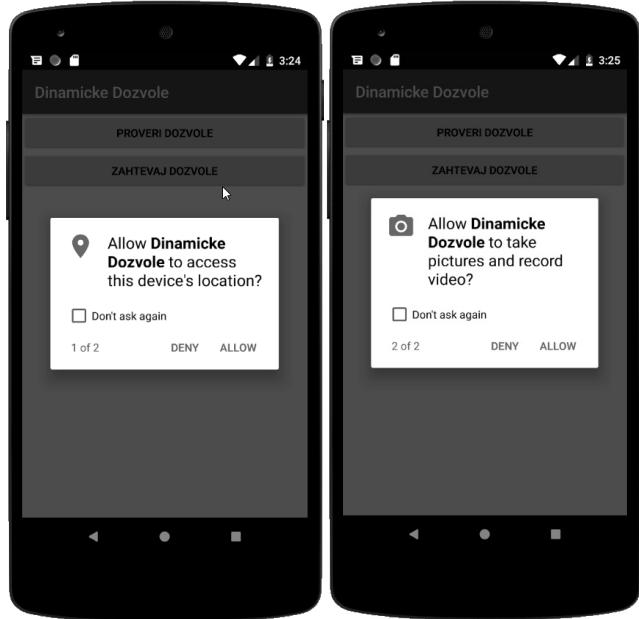
korisnik pritisne dugme za traženje dozvola, aplikacija će od njega tražiti da dodeli potrebne dozvole jednu za drugom, kao što je prikazano na slici 7.6. U zavisnosti od izbora korisnika, moguća su dva scenarija.

U prvom scenariju, korisnik prihvata da dodeli obe dozvole. U tom slučaju se korisniku prikazuje Toast sa obaveštenjem da su dozvole odobrene, a isto se može potvrditi proverom statusa dozvola aplikacije u okviru sistemskih podešavanja, kao što je prikazano na slici 7.7. U sekciji sistemskih podešavanja korisnik može u bilo kom trenutku da povuče prethodno odobrene dozvole, pa je zbog toga svaki put potrebno proveravati da li aplikacija i dalje ima potrebne dozvole. Pošto su obe potrebne dozvole odobrene, aplikacija bi mogla u tom trenutku da pristupi kameri i lokaciji uređaja.

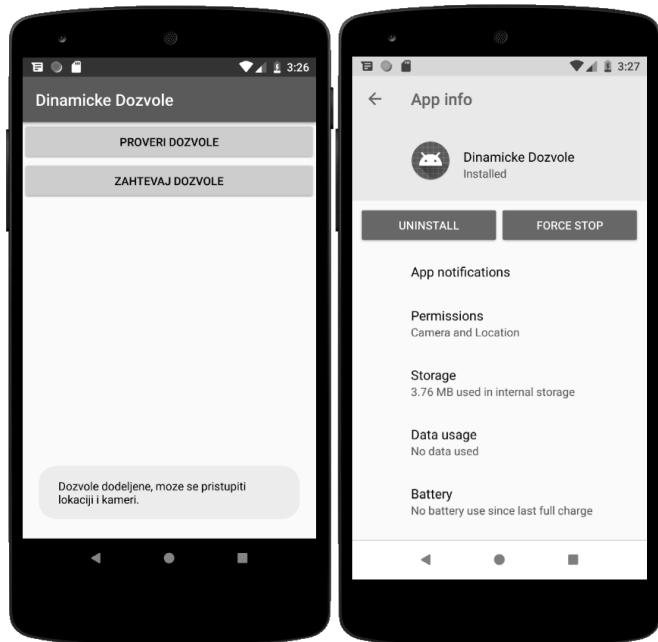
U drugom scenariju, korisnik odbija da dodeli dozvole, kao što je prikazano na slici 7.8. U tom slučaju se korisniku prikazuje dijalog sa dodatnim objašnjenjem zbog čega su dozvole potrebne. Ukoliko korisnik i tada odbije da dodeli dozvole, aplikacija može da nastavi sa radom, ali sa umanjenim funkcionalnostima u smislu da neće moći da koristi lokaciju uređaja i kameru.



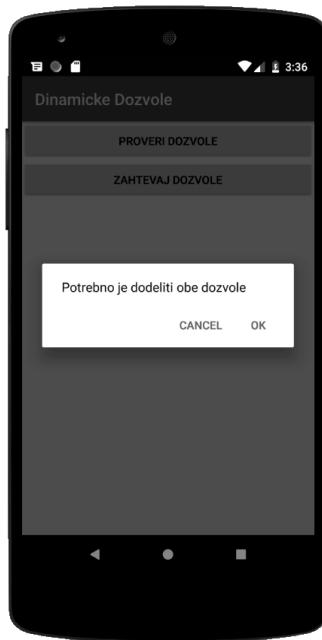
Slika 7.5, Rad sa dozvolama, dinamička provera dozvola



Slika 7.6, Rad sa dozvolama, dinamičko traženje dozvola



Slika 7.7, Rad sa dozvolama, dinamička dodata dozvola i provera statusa dozvola za aplikaciju u okviru podešavanja



Slika 7.8, Rad sa dozvolama, dijalog ukoliko korisnik odbije da dodeli dozvole

Primer 2: Napraviti jednostavnu aplikaciju koja će čuvati fajl u okviru eksternog prostora aplikacije. Ovaj primer podrazumeva da se u folderu res/drawable nalazi slika sa imenom slika1.jpg, koju je klikom na dugme na korisničkom interfejsu potrebno sačuvati u okviru eksternog prostora aplikacije.

Rešenje: Korisnički interfejs aplikacije sadrži samo jedno dugme, čijim se klikom slika čuva u eksternom prostoru. Sadržaj activity_main.xml fajla je dat sa:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <Button
        android:id="@+id/buttonSacuvaj"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Sacuvaj sliku"/>
</LinearLayout>
```

Kako bi aplikacija smela da pristupi eksternom prostoru, mora se zahtevati dozvola za to u manifestu aplikacije, čiji je sadržaj dat sa:

```

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

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

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

                    <category
    android:name="android.intent.category.LAUNCHER" />
                </intent-filter>
            </activity>
        </application>

</manifest>

```

U okviru glavne aktivnosti, klik na dugme pokreće čuvanje fajla. Najpre se mora proveriti da li je verzija Android uređaja na kome se aplikacija trenutno izvršava veća ili jednaka 23. Ukoliko jeste, mora se dinamički tražiti dozvola. Nakon toga se pristupa eksternom prostoru i slika čuva u root direktorijumu. Kod glavne aktivnosti je dat sa:

```

package com.example.student.dozvolaeksterniprostor;

import android.content.pm.PackageManager;
import android.graphics.Bitmap;
import android.graphics.drawable.BitmapDrawable;
import android.graphics.drawable.Drawable;
import android.os.Build;
import android.os.Environment;
import android.support.v4.app.ActivityCompat;
import android.support.v4.content.ContextCompat;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;

```

```

import android.widget.Toast;

import java.io.File;
import java.io.FileOutputStream;

public class MainActivity extends AppCompatActivity
implements View.OnClickListener{

    private static final int
WRITE_EXTERNAL_PERMISSION_REQUEST_CODE = 15;
    private Button buttonSacuvaj;
    private Bitmap bitmap;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        initComponents();
    }

    private void initComponents() {
        Drawable drawable =
getResources().getDrawable(R.drawable.slikal);
        bitmap = ((BitmapDrawable)drawable).getBitmap();
        buttonSacuvaj = findViewById(R.id.buttonSacuvaj);
        buttonSacuvaj.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        if (Environment.getExternalStorageState() .
equals(Environment.MEDIA_MOUNTED)) {
            if (Build.VERSION.SDK_INT >= 23) {
                if (checkPermission()) {
                    String path =
Environment.getExternalStorageDirectory().toString();
                    File file = new File(path,
"Slon"+".jpg");
                    if (!file.exists()) {
                        try {
                            FileOutputStream fos = new
FileOutputStream(file);
                            bitmap.compress(Bitmap.CompressFormat.JPEG, 100, fos);
                            fos.flush();
                            fos.close();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                }
            }
        }
    }
}

```

```
        }
    }
} else {
    requestPermission(); // potrebno je
zahtevati dozvolu
}
} else {
    //u pitanju je Android < 23 API, nije
potrebno dinamicki proveravati dozvolu
    String path =
Environment.getExternalStorageDirectory().toString();
    File file = new File(path, "Slon"+".jpg");
    if (!file.exists()) {
        FileOutputStream fos = null;
        try {
            fos = new FileOutputStream(file);
        }
        bitmap.compress(Bitmap.CompressFormat.JPEG, 100, fos);
            fos.flush();
            fos.close();
        } catch (java.io.IOException e) {
            e.printStackTrace();
        }
    }
}

private boolean checkPermission() {
    int result =
ContextCompat.checkSelfPermission(MainActivity.this,
android.Manifest.permission.WRITE_EXTERNAL_STORAGE);
    if (result == PackageManager.PERMISSION_GRANTED)
        return true;
    } else {
        return false;
    }
}

private void requestPermission() {
    if
(ActivityCompat.shouldShowRequestPermissionRationale(
MainActivity.this,
android.Manifest.permission.WRITE_EXTERNAL_STORAGE) ) {
    Toast.makeText(MainActivity.this, "Da bi
aplikacija mogla da sacuva sliku u eksternom prostoru
potrebna je dozvola", Toast.LENGTH_LONG).show();
    } else {
}
```

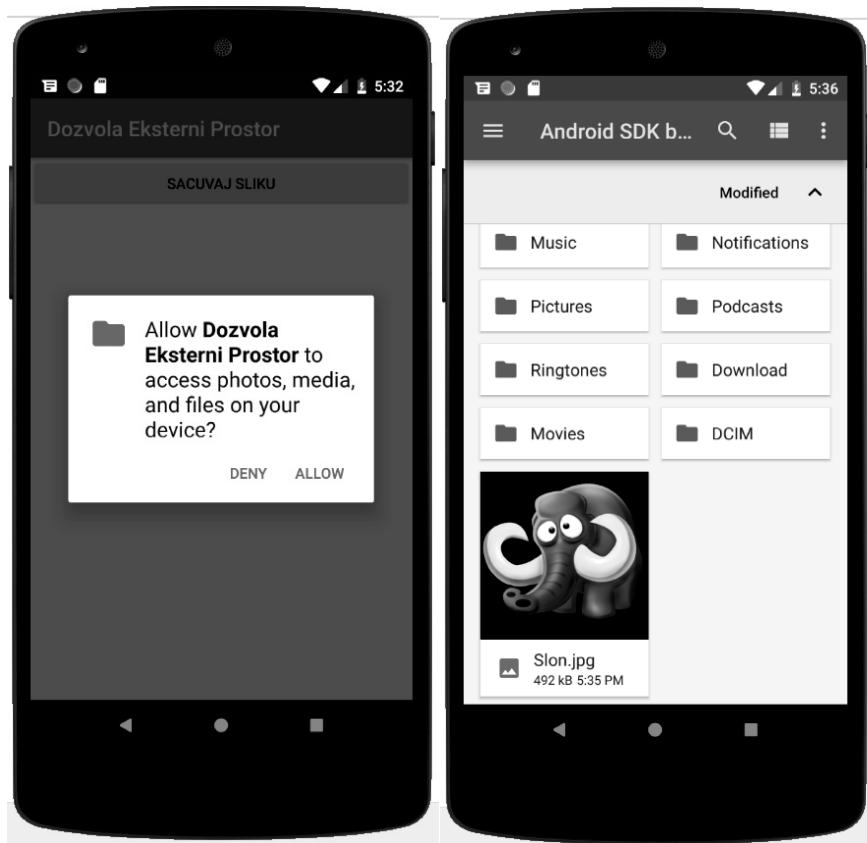
```

ActivityCompat.requestPermissions(MainActivity.this, new
String[]{android.Manifest.permission.WRITE_EXTERNAL_STORAGE
}, WRITE_EXTERNAL_PERMISSION_REQUEST_CODE);
}

@Override
public void onRequestPermissionsResult(int requestCode,
String permissions[], int[] grantResults) {
    switch (requestCode) {
        case WRITE_EXTERNAL_PERMISSION_REQUEST_CODE:
            if (grantResults.length > 0 &&
grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                Toast.makeText(this, "Dodeljena dozvola,
moze se koristiti eksterni prostor",
Toast.LENGTH_LONG).show();
            } else {
                Toast.makeText(this, "Odbijena dozvola,
ne moze se koristiti eksterni prostor",
Toast.LENGTH_LONG).show();
            }
            break;
    }
}
}

```

Nakon pokretanja aplikacije i klika na dugme za čuvanje, aplikacija će tražiti od korisnika potrebnu dozvolu (slika 7.9, levo). Ukoliko korisnik dodeli potrebnu dozvolu, aplikacija će sačuvati zadatu sliku u root direktorijumu eksternog prostora, kao što se vidi na slici 7.9 desno. Ukoliko korisnik ne bi dozvolio pristup eksternom prostoru, aplikacija ne bi sačuvala sliku, ali bi nastavila da radi.



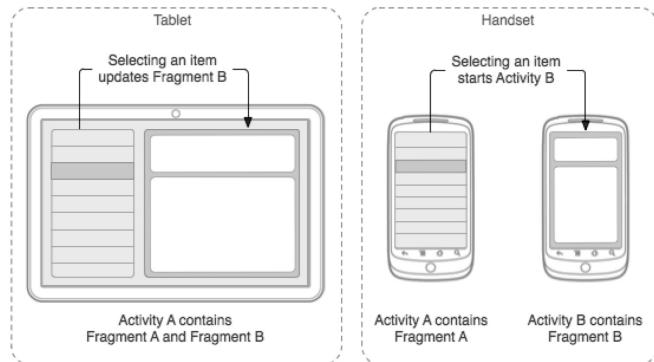
Slika 7.9, dinamička dozvola pristupa eksternom prostoru

8. Fragmenti

Fragmenti su veoma korisni modularni „delovi“ aktivnosti koji omogućavaju veoma laku ponovnu upotrebu u drugim aktivnostima. Jedan fragment opisuje ponašanje jednog dela korisničkog ekrana u FragmentActivity komponenti. Na taj način, omogućeno je kombinovanje više fragmenata u okviru jedne aktivnosti, čime se dobija višepanelni korisnički interfejs. Fragment se po potrebi može dodavati ili uklanjati za vreme izvršavanja aktivnosti), a može se i ponovo upotrebiti u nekoj drugoj aktivnosti. Fragment uvek mora biti ugnježđen u aktivnosti domaćinu. Fragmenti su uvedeni u Android sistem sa verzijom 3.0 (API nivo 11).

Fragment je deo aktivnosti, ali ima svoj životni ciklus i prihvata svoje korisničke unose. Međutim, njegov životni ciklus je pod direktnim uticajem životnog ciklusa same aktivnosti kojoj pripada. Na primer, ukoliko aktivnost izgubi fokus i pređe u pauzirano stanje, i svi fragmenti koji joj pripadaju prelaze u pauzirano stanje, a prilikom uništavanja aktivnosti, uništavaju se i svi fragmenti koji joj pripadaju. Dok je aktivnost aktivna, međutim, svakim fragmentom se može upravljati nezavisno, što podrazumeva i njihovo dodavanje i uklanjanje dok se aktivnost izvršava. Time je omogućen dinamički i fleksibilan korisnički interfejs, koji je naročito pogodan za velike ekrane poput tableta. Pošto je ekran tableta značajno veći od ekrana telefona, dizajn koji je namenjen telefonu ne mora izgledati dobro i na tabletu (jedna aktivnost može imati premalo sadržaja da ispuni veliki ekran, pa može izgledati „šupljikavo“). Na velikim ekranima tableta se zbog toga često kombinuje sadržaj više komponenti, a takav dizajn je omogućen upravo fragmentima. Dizajn podrazumeva da se raspored komponenti aktivnosti podeli u fragmente, a zatim se prikaz aktivnosti menja za vreme izvršavanja.

Tipičan primer upotrebe fragmenata je aplikacija koja služi za prikaz vesti. Ukoliko se izvršava na telefonu, zbog malog ekrana, potrebna je jedna aktivnost da prikaže listu vesti, a druga za prikaz pojedinačne vesti. Na tabletu se veći prostor ekrana može iskoristiti na drugačiji način, pa se u okviru jednog fragmenta na levoj strani može prikazati lista vesti, a u drugom fragmentu na desnoj strani sadržaj trenutno odabrane vesti, pri čemu se oba fragmenta prikazuju u jednoj aktivnosti, jedan pored drugog, kao što se može videti na slici 8.1.



Slika 8.1, primer modularnog dizajna upotreboom fragmenata
(slika preuzeta sa: <https://developer.android.com/guide/components/fragments>)

Svaki pojedinačni fragment se dizajnira tako da bude upotrebljiv više puta, odnosno da se može iskoristiti u više aktivnosti. Tako se postiže podrška i za telefone i za tablete, pošto se fragmenti mogu kombinovati i ponovo upotrebljavati u više različitih konfiguracija rasporeda komponenti, kako bi se maksimalno iskoristila dostupna veličina ekrana i optimizovalo korisničko iskustvo. Na primeru aplikacije za prikaz vesti, dva fragmenta se mogu ugnezdit u jednu aktivnost za prikaz na tabletu, kako bi se iskoristio veliki ekran (aktivnost A sadrži i fragment A i fragment B). Pošto na telefonu nema prostora za istovremeni prikaz oba fragmenta, aktivnost A sadrži samo fragment A koji prikazuje listu vesti, a kada korisnik odabere neku konkretnu vest, pokreće se aktivnost B koja sadrži fragment B koji prikazuje sadržaj konkretnе vesti. Ovako dizajnirana aplikacija podržava izvršavanje i na tabletima i na telefonima.

8.1. Životni ciklus fragmenta

Fragmenti se kreiraju tako što se napravi klasa izvedena iz klase `Fragment`. Klasa `Fragment` veoma liči na klasu `Activity`. Ukoliko se pogleda struktura klase `Fragment`, uočavaju se slične callback metode: `onCreate()`, `onStart()`, `onPause()` i `onStop()`. To već nagoveštava da je moguće konvertovati aktivnost u fragment jednostavnim premeštanjem koda koji se nalazi u callback metodama aktivnosti u odgovarajuće callback metode fragmenta. U najvećem broju praktičnih slučajeva potrebno je implementirati sledeće metode životnog ciklusa fragmenta:

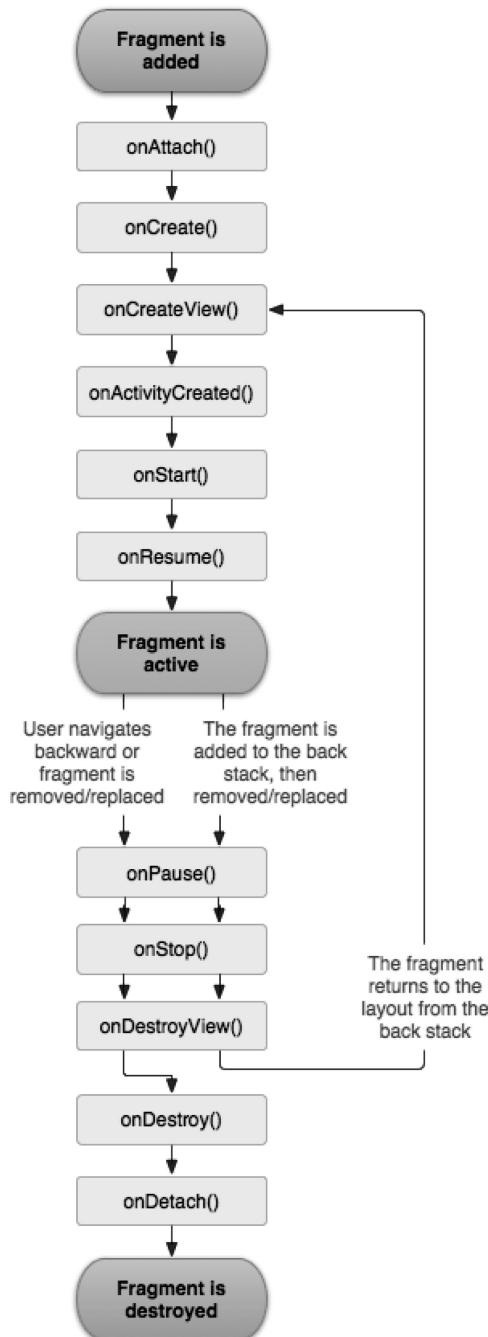
- `onCreate()` – slično kao i kod aktivnosti, ovu metodu poziva Android sistem prilikom kreiranja fragmenta. U okviru ove metode se inicijalizuju osnovne komponente fragmenta koje je potrebno zadržati kada fragment bude pauziran ili stopiran, a zatim nastavljen.

- `onCreateView()` – sistem poziva ovu metodu u trenutku kada je potrebno da fragment prvi put iscerta svoj korisnički interfejs. Kao rezultat ove metode potrebno je vratiti View komponentu koja je na vrhu hijerarhije rasporeda komponenti fragmenta. Ukoliko fragment nema korisnički interfejs, može se vratiti null.
- `onPause()` – opet slično kao kod aktivnosti, sistem poziva ovu metodu kao prvi nagoveštaj da korisnik napušta fragment (to ne znači da će fragment uvek biti uništen, ali postoji šansa da hoće). Poželjno je u okviru ove metode sačuvati sve što je potrebno, pošto se korisnik možda neće vratiti na fragment.

Svaka aplikacija koja koristi fragmente treba da implementira bar ove tri metode za svaki fragment. Postoje i druge callback metode koje se mogu koristiti za rukovanje različitim fazama života fragmenta. Na slici 8.2 prikazan je životni ciklus fragmenta kroz faze koje prolazi u okviru svoje aktivnosti domaćina. Na istoj slici su prikazane i sve callback metode koje je moguće implementirati u okviru svakog fragmenta. Može se uočiti da postoji veliki broj sličnosti sa životnim ciklусom aktivnosti.

Nije uvek neophodno praviti fragment kreiranjem klase koja je izvedena direktno iz klase Fragment. Moguće je koristiti još neke klase kao bazne klase za klasu koja se trenutno piše, u zavisnosti od željenog osnovnog prikaza samog fragmenta:

- `DialogFragment` – proširivanje ove klase omogućava kreiranje fragmenta u obliku pop-up dijaloga.
- `ListFragment` – proširivanje ove klase omogućava kreiranje fragmenta koji prikazuje listu stavki kontrolisanih nekim adapterom. Pruža i nekoliko veoma korisnih metoda za manipulisanje listom, poput callback metode `onListItemClick()` kojom se rukuje on-click događajima nad elementima liste.



Slika 8.2, životni ciklus fragmenta dok se njegova aktivnost domaćin izvršava
 (izvor: <https://developer.android.com/guide/components/fragments>)

8.2. IsCRTavanje korisničkog interfejsa fragmenta

Fragment se obično koristi kao deo korisničkog interfejsa aktivnosti, pri čemu doprinosi svoj sopstveni raspored komponenti. Raspored komponenti se daje kao rezultat metode `onCreateView()`, koju Android sistem poziva kada je vreme da se fragment iscrtava na ekranu. Raspored komponenti se definiše na standardan način, kao XML resurs. Ukoliko je raspored definisan u resursu `primer_fragment.xml` koji se nalazi u `res/layout` direktorijumu, `onCreateView()` bi mogao da bude definisan na sledeći način:

```
public static class PrimerFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater,
    ViewGroup container, Bundle savedInstanceState) {
        // Pripremi raspored za ovaj fragment
        return inflater.inflate(R.layout.primer_fragment,
    container, false);
}
```

Parametar `container` koji se prosleđuje u pozivu `onCreateView()` metode je roditeljska `ViewGroup` komponenta (iz rasporeda komponenti aktivnosti domaćina) u koju se umeće raspored komponenti fragmenta. Metoda `inflate()` služi za iscrtavanje rasporeda, i prihvata tri argumenta:

- ID rasporeda komponenti koji je potrebno iscrtati. Odgovara identifikatoru u okviru `R.java` fajla.
- `ViewGroup` komponenta roditelja u kojoj će se iscrtati komponenta na vrhu hijerarhije rasporeda fragmenta.
- boolean parametar koji služi kao indikacija da li je potrebno prikačiti raspored komponenti fragmenta za `ViewGroup` komponentu koja je prosleđena kao drugi parametar (u primeru iznad se prosleđuje `false`, zato što će sistem automatski ubaciti iscrtani raspored u `container`).

8.3. Dodavanje fragmenta aktivnosti

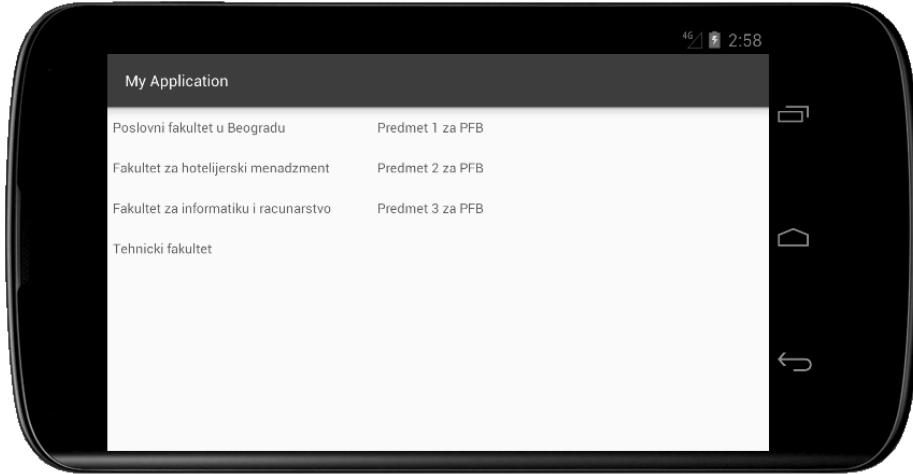
Pošto fragment doprinosi deo korisničkog interfejsa aktivnosti domaćina, potrebno je ugnezdati ga u hijerarhiju komponenti aktivnosti. To se može postići na dva načina – deklaracijom fragmenta unutar XML fajla sa rasporedom komponenti aktivnosti domaćina, ili programskim putem, dodavanjem fragmenta u već postojeću `ViewGroup` komponentu.

8.3.1. Deklaracija fragmenta unutar fajla sa rasporedom komponenti aktivnosti

Ukoliko se koristi ovaj pristup, raspored komponenti fragmenta se dodaje kao obična komponenta u okviru XML fajla sa rasporedom komponenti aktivnosti koja je domaćin fragmentu. Na primer, raspored komponenti aktivnosti koja sadrži dva fragmenta bi mogao biti definisan na sledeći način:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
    <fragment
        android:id="@+id/fragmentFakulteta"
        android:name=
"com.example.mzivkovic.myapplication.FakultetiFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="0.4" />
    <fragment
        android:id="@+id/fragmentPredmeta"
        android:name=
"com.example.mzivkovic.myapplication.PredmetiFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="0.6" />
</LinearLayout>
```

Atribut android:name u okviru <fragment> elementa u rasporedu specificira klasu izvedenu iz klase Fragment koja se koristi. U ovom primeru, podrazumeava se da postoje dve klase izvedene iz klase Fragment – FakultetiFragment i PredmetiFragment. Kada sistem kreira raspored komponenti za ovu aktivnost, napraviće instancu svakog fragmenta specificiranog u rasporedu, a nakon toga će pozvati odgovarajuće onCreateView() metode svakog pojedinačnog fragmenta, kako bi dohvatio raspored komponenti tog fragmenta. Sistem će isertati View komponentu vraćenu od strane onCreateView() metode direktno na mesto <fragment> elementa. Ovaj primer se odnosi na horizontalni prikaz, odnosno landscape režim rada uređaja (android:orientation atribut ima vrednost horizontal). Pošto su dodeljene težine atributima android:layout_weight za oba fragmenta, FakultetiFragment će zauzeti dve petine raspoloživog ekrana, a PredmetiFragment preostale tri petine (slika 8.3).



Slika 8.3, ubacivanje fragmenta u aktivnost deklaracijom u XML rasporedu aktivnosti

8.3.1. Programsko dodavanje fragmenta

Ukoliko se fragment dodaje programski, to se može učiniti u bilo kom trenutku izvršavanja aktivnosti domaćina. Potrebno je samo naznačiti u koju ViewGroup komponentu treba smestiti fragment. Za bilo koju manipulaciju fragmentom u okviru aktivnosti, poput dodavanja, uklanjanja ili zamene fragmenta, mora se koristiti API FragmentTransaction. Instanca objekta FragmentTransaction u okviru FragmentActivity aktivnosti se dobija na sledeći način:

```
FragmentManager fragmentManager =  
getSupportFragmentManager();  
FragmentTransaction fragmentTransaction =  
fragmentManager.beginTransaction();
```

Nakon toga, fragment se dodaje upotrebom add() metode i prosleđivanjem dva parametra. Potrebno je naznačiti koji fragment treba dodati i u koju ViewGroup komponentu ga treba ubaciti, što se postiže sledećim isečkom Java koda:

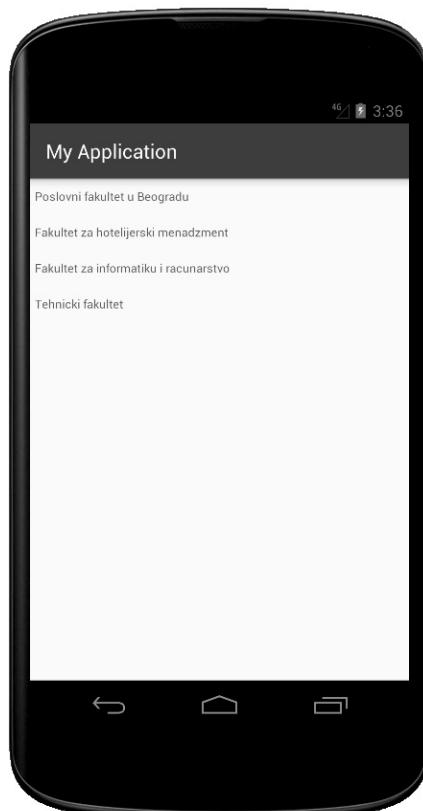
```
FragmentManager fragmentManager =  
getSupportFragmentManager();  
FragmentTransaction fragmentTransaction =  
fragmentManager.beginTransaction();  
FakultetiFragment fakulteti = new FakultetiFragment();  
fragmentTransaction.add(R.id.holder_fragmenta,fakulteti);  
fragmentTransaction.commit();
```

Prvi parametar metode add() predstavlja ViewGroup komponenta u koju se fragment smešta, u ovom slučaju to je FrameLayout komponenta sa

identifikatorom R.id.holder_fragmenta, koja je definisana u odgovarajućem XML fajlu sa rasporedom komponenti. Drugi parametar je nova instanca FakultetiFragment klase. XML fajl sa rasporedom komponenti bi u ovom slučaju izgledao ovako:

```
<RelativeLayout  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
    <FrameLayout  
        android:layout_width="match_parent"  
        android:layout_height="match_parent"  
        android:id="@+id/holder_fragmenta">  
    </FrameLayout>  
</RelativeLayout>
```

U ovom slučaju, samo jedan fragment bi bio dodat u raspored komponenti aktivnosti, što bi odgovaralo portrait orientaciji uređaja, kao što je prikazano na slici 8.4.



Slika 8.4, ubacivanje fragmenta programskim putem

8.4. Transakcija fragmenta

Velika prednost upotrebe fragmenata u aktivnosti je mogućnost dodavanja, uklanjanja ili zamene fragmenta kao odgovor na interakciju sa korisnikom. Svaka od mogućih akcija koja se komituje u aktivnost se naziva transakcija, a izvršava se pomoću FragmentTransaction API-ja.

Svaka transakcija može da bude skup promena koje je potrebno primeniti u isto vreme. Moguće je pripremiti sve potrebne promene u transakciji upotrebom metoda add(), remove() i replace(), a zatim primeniti transakciju na aktivnost pozivom metode commit(). Moguće je dodati transakciju na back stack transakcija pozivom metode addToBackStack() pre poziva metode commit(). Time se omogućava da se korisnik vrati na prethodno stanje fragmenta prostim pritiskom na Back dugme. Kod koji menja jedan fragment drugim, konkretno, na prethodnom primeru, kada korisnik klikne na neki od fakulteta dok je u portrait režimu, kako bi mu se pokazala lista predmeta na tom fakultetu, dat je u nastavku sledećim Java kodom:

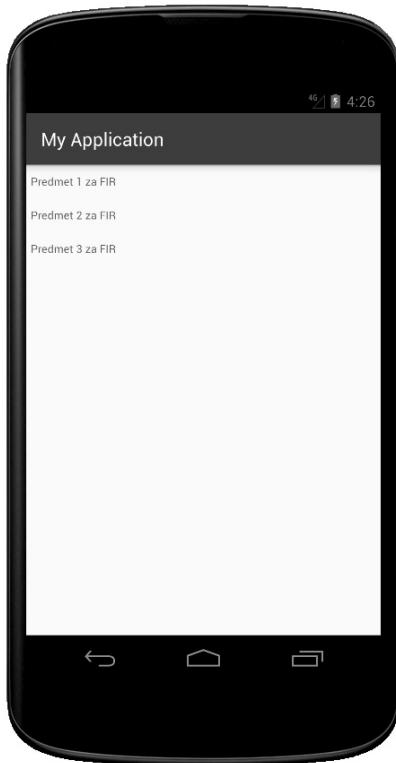
```
PredmetiFragment noviPredmeti = new PredmetiFragment();
FragmentTransaction transaction =
getSupportFragmentManager().beginTransaction();
// Zameniti sta god se nalazi u kontejneru fragmenta sa
ovim //novim fragmentom, i dodati transakciju na back stack
transaction.replace(R.id.holder_fragmenta, noviPredmeti);
transaction.addToBackStack(null);
// Commit transakcije
transaction.commit();
```

Na ovaj način, fragment fakulteta prikazan na slici 8.4 bio bi zamenjen fragmentom koji sadrži listu predmeta, koji je prikazan na slici 8.5.

Ukoliko se doda više promena u okviru jedne transakcije i pozove addToBackStack(), sve promene pre poziva commit() se dodaju na back stack kao jedna transakcija, a pritisak na Back dugme ih vraća sve zajedno. Sam redosled promena dodatih u jednu transakciju nije bitan, osim dva pravila:

- Poziv commit() metode mora biti poslednji.
- Ukoliko se dodaje više fragmenata u jedan kontejner, redosled dodavanja u transakciju određuje i redosled prikazivanja u hijerarhiji.

Na kraju je potrebno naglasiti da sam poziv commit() metode ne izvršava transakciju momentalno, već zakazuje njen izvršavanje u korisničkoj niti čim nit bude mogla da to učini.



Slika 8.5, zamena jednog fragmenta drugim

8.5. Komunikacija fragmenta sa aktivnosti domaćinom

Fragment se implementira nezavisno od aktivnosti, i kao takav se može koristiti u više aktivnosti. Međutim, data konkretna instanca fragmenta je direktno vezana za aktivnost koja joj je domaćin. Fragment može vrlo jednostavno dohvatiti instancu FragmentActivity kojoj pripada pozivom metode getActivity(), a nakon toga obaviti zadatke poput traženja određene View komponente u rasporedu komponenti aktivnosti na sledeći način:

```
View listView = getActivity().findViewById(R.id.list);
```

U suprotnom smeru, aktivnost jednostavno može pozivati metode iz fragmenta nakon što prvo dohvati referencu na Fragment upotrebom FragmentManager klase i metode findFragmentById(), kao što je prikazano sledećim primerom:

```
FakultetiFragment fragment = (FakultetiFragment)
getSupportFragmentManager().
findFragmentById(R.id.fragmentFakulteta);
```

U nekim slučajevima potrebno je da fragment podeli neki događaj sa svojom aktivnosti ili sa drugim fragmentima kojima je ta aktivnost takođe domaćin. Dva fragmenta ne mogu da komuniciraju direktno, već to moraju obaviti kroz aktivnost koja im je domaćin. Posmatrajmo opisan primer sa dva fragmenta, jednim koji sadrži listu fakulteta, i drugim koji prikazuje listu predmeta za odabrani fakultet. FakultetiFragment mora nekako obavestiti aktivnost koji fakultet je korisnik odabrao, a aktivnost onda mora obavestiti PredmetiFragment koje predmete treba da prikaže. To se postiže kreiranjem interfejsa u okviru fragmenta i callback metode u okviru aktivnosti koja će hvatati odgovarajući događaj.

```
public class FakultetiFragment extends Fragment {  
    ...  
    // Aktivnost domacin mora implementirati ovaj interfejs  
    public interface OnFragmentInteractionListener{  
        void odabranFakultet(String fakultet);  
    }  
    ...  
}
```

Nakon toga aktivnost koja je domaćin ovom fragmentu mora implementirati dati interfejs i implementirati metodu odabranFakultet(), u kojoj će obavestiti PredmetiFragment o događaju koji se desio u fragmentu FakultetiFragment (da je korisnik kliknuo na neki od fakulteta, i da treba prikazati predmete za taj fakultet). Kako bi se osiguralo da aktivnost domaćin implementira ovaj interfejs, u metodi onAttach() fragmenta FakultetiFragment (koju okida sistem prilikom dodavanja fragmenta aktivnosti koja će mu biti domaćin) se kreira objekat OnFragmentInteractionListener interfejsa (promenljiva mListener) i dodeljuje mu se Aktivnost koja je prosleđena u metodi onAttach():

```
public class FakultetiFragment extends Fragment {  
  
    private OnFragmentInteractionListener mListener;  
  
    ...  
    @Override  
    public void onAttach(Context context) {  
        super.onAttach(context);  
        if (context instanceof OnFragmentInteractionListener) {  
            mListener = (OnFragmentInteractionListener) context;  
        } else {  
            throw new RuntimeException(context.toString()  
                + " mora implementirati odgovarajući  
OnFragmentInteractionListener");  
        }  
    }
```

```

}

@Override
public void onDetach() {
    super.onDetach();
    mListener = null;
}
...
// Aktivnost domaćin mora implementirati ovaj interfejs
public interface OnFragmentInteractionListener{
    void odabraniFakultet(String fakultet);
}
...
}

```

Ukoliko prosleđena aktivnost nije implementirala odgovarajući interfejs, izbacuje se izuzetak. U slučaju pozitivnog ishoda, mListener promenljiva sadrži referencu na aktivnost koja implementira OnFragmentInteractionListener, čime se omogućava da FakultetiFragment može da obavesti aktivnost o događaju pozivom metoda definisanih u pomenutom interfejsu. Odnosno, ukoliko korisnik klikne na neki od fakulteta, sistem poziva onClick() događaj u okviru fragmenta, koji nakon toga treba da pozove odabraniFakultet() metodu interfejsa kako bi obavestio aktivnost o događaju.

Implementacija ove logike u FakultetiFragment klasi je data u nastavku. Kako bi klasa FakultetiFragment hvatala onClick() događaje, modifikovana je da implementira interfejs View.OnClickListener.

```

public class FakultetiFragment extends Fragment implements
View.OnClickListener{

    private OnFragmentInteractionListener mListener;

    ...
@Override
public void onAttach(Context context) {
    super.onAttach(context);
    if (context instanceof
OnFragmentInteractionListener) {
        mListener = (OnFragmentInteractionListener)
context;
    } else {
        throw new RuntimeException(context.toString()
                + " mora implementirati odgovarajući
OnFragmentInteractionListener");
    }
}

```

```

@Override
public void onDetach() {
    super.onDetach();
    mListener = null;
}

...
@Override
public void onClick(View view) {
    //Kada se dogodi klik, treba da obavestimo nas
    listener, da je odabran fakultet
    String fakultet = (String) view.getTag();
    if (mListener != null){
        mListener.odabraniFakultet(fakultet);
        //da bi aktivnost dobila informaciju, ona mora
da implementira odgovarajuci interfejs
    }
}

...
// Aktivnost domacin mora implementirati ovaj interfejs
public interface OnFragmentInteractionListener{
    void odabraniFakultet(String fakultet);
}
...
}

```

Na kraju, potrebno je još dati implementaciju callback metode odabraniFakultet() u okviru aktivnosti domaćina. Ova aktivnost u toj metodi treba da pripremi PredmetiFragment sa odgovarajućom listom predmeta, i da zameni fragment fakulteta sa pripremljenim fragmentom predmeta. To se postiže na sledeći način (prikazana je pojednostavljena implementacija, samo za portrait režim rada):

```

public class MainActivity extends AppCompatActivity
implements FakultetiFragment.OnFragmentInteractionListener
{

    ...
    @Override
    public void odabraniFakultet(String fakultet) {
        ...
        //Portrait rezim rada
        //kada je odabran fakultet, treba odraditi zamenu
        fragmenta
        PredmetiFragment noviPredmetiFragment =
        PredmetiFragment.newInstance(fakultet);
    }
}

```

```
//prosledjujemo dodatnu informaciju, fakultet za koji  
zelimo instance predmeta  
        getSupportFragmentManager().beginTransaction()  
replace(R.id.holder_fragmenta, noviPredmetiFragment).  
addToBackStack(null).commit();  
        ...  
        //Landscape rezim rada  
        ...  
    }  
}
```

8.6. Pitanja za vežbu

- Šta je fragment?
- Koja je razlika između fragmenta i aktivnosti?
- Kada je najbolje primeniti fragmente? Objasniti.
- Kako se definiše raspored komponenti fragmenta (layout)?
- Da li se fragment može višestruko koristiti?
- Da li fragment mora uvek biti ugnježden u okviru aktivnosti? Objasniti.
- Objasniti životni ciklus fragmenta.
- Koje callback metode u okviru životnog ciklusa fragmenta bi trebalo obavezno implementirati?
- Da li je životni ciklus fragmenta zavisan od životnog ciklusa aktivnosti?
- Ako aktivnost domaćin pređe u stopirano stanje, šta se dešava sa fragmentom koji se u njoj nalazi?
- Čemu služi metoda inflate()? Objasniti koje parametre prima.
- Na koje se sve načine fragment može dodati u aktivnost?
- Čemu služi poziv addBackStack()?
- Šta se dešava sa fragmentom ukoliko se pritisne Back dugme?
- Kako se mogu sačuvati podaci prilikom promene konfiguracije uređaja (npr. prelazak iz Portrait u Landscape režim rada)?
- Šta je transakcija fragmenta?
- Koje metode se mogu koristiti u transakciji fragmenta?
- Da li je moguće da jedna transakcija u sebi sadrži više izmena?
- Koja je razlika između dodavanja i zamene fragmenta?
- Čemu služi FragmentManager?
- Kako aktivnost može da komunicira sa fragmentom kome je domaćin?
- Kako fragment može da komunicira sa aktivnosti koja mu je domaćin?
- Da li dva fragmenta koji su delovi iste aktivnosti domaćina mogu da komuniciraju direktno jedan sa drugim? Objasniti odgovor.

8.7. Zadaci za vežbu

Primer: Realizovati aplikaciju čiji se ekran sastoji od dva fragmenta, koji zauzimaju po pola raspoloživog prostora. Na levom fragmentu, nalazi se lista verzija Android operativnih sistema. Klikom na bilo koji od elemenata iz liste, u desnem fragmentu se ispisuju detalji o toj verziji operativnog sistema. Realizovati komunikaciju fragmenta sa aktivnosti pomoću interfejsa.

Rešenje:

U korisničkom interfejsu glavne aktivnosti postoje dva fragmenta, koje možemo deklarisati unutar fajla sa rasporedom komponenti. Svaki od fragmenata zauzima 50% raspoloživog prostora. Kod activity_main.xml fajla dat je sa:

```
<LinearLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
        android:layout_width="match_parent"  
        android:layout_height="match_parent"  
        android:orientation="horizontal"  
        android:weightSum="1.0">  
    <fragment  
        android:layout_height="match_parent"  
        android:layout_width="match_parent"  
  
        class="com.example.mzivkovic.myapplication.MeniFragment"  
            android:id="@+id/fragmentMeni"  
            android:layout_weight="0.5"/>  
    <fragment  
        android:layout_width="match_parent"  
        android:layout_height="match_parent"  
  
        class="com.example.mzivkovic.myapplication.DetaljiFragment"  
            android:id="@+id/fragmentDetalji"  
            android:layout_weight="0.5"/>  
</LinearLayout>
```

Svaki od dva fragmenta (MeniFragment koji sadrži listu Android verzija i DetaljiFragment koji pruža detalje o svakoj platformi) je definisan svojim rasporedom komponenti. MeniFragment svoj raspored ima definisan u fajlu list_fragment.xml:

```
<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
        android:orientation="vertical"  
        android:layout_width="match_parent"  
        android:layout_height="match_parent">  
    <ListView
```

```
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+android:id/list" />
</LinearLayout>
```

DetaljiFragment svoj raspored drži u fajlu text_fragment.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:gravity="center"
        android:background="#00008b"
        android:layout_height="match_parent">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textColor="#ffffff"
        android:layout_gravity="center"
        android:textSize="35px"
        android:id="@+id/labelOs"/>
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textColor="#ffffff"
        android:layout_gravity="center"
        android:textSize="25px"
        android:id="@+id/labelVersion"/>
</LinearLayout>
```

Kod klase DetaljiFragment.java dat je u nastavku:

```
package com.example.mzivkovic.myapplication;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

public class DetaljiFragment extends Fragment {
    private TextView osTitle;
    private TextView version;

    @Override
    public View onCreateView(LayoutInflater inflater,
```

```

ViewGroup container, Bundle savedInstanceState) {

    View view =
inflater.inflate(R.layout.text_fragment, container, false);
    osTitle = view.findViewById(R.id.labelOs);
    version = view.findViewById(R.id.labelVersion);
    return view;
}

public void change(String os, String ver) {
    osTitle.setText(os);
    version.setText(ver);
}
}

```

Klasa MeniFragment je izvedena iz klase ListFragment, pošto svakako treba da bude u obliku liste. Ovaj fragment treba da komunicira sa svojom aktivnosti koja mu je domaćin, a to je implementirano kroz interfejs OnFragmentInteractionListener(), koji pruža metodu kojom ćemo aktivnosti proslediti podatke o tome na koji element liste je kliknuto, kako bismo kasnije ažurirali drugi fragment. Kod klase MeniFragment.java dat je u nastavku:

```

package com.example.mzivkovic.myapplication;

import android.support.v4.app.ListFragment;
import android.content.Context;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ListView;

public class MeniFragment extends ListFragment {

    private OnFragmentInteractionListener mListener;

    String[] AndroidOS = new String[] { "Android
10", "Pie", "Oreo", "Nougat", "Marshmallow", "Lollipop", "KitKat",
"Jelly Bean", "Ice Cream Sandwich" };
    String[] Version = new String[] {"10.0", "9.0", "8.0-
8.1", "7.0-7.1.2", "6.0-6.0.1", "5.0-5.1.1", "4.4", "4.1-
4.3", "4.0" };
    @Override

    public View onCreateView(LayoutInflater inflater,
ViewGroup container, Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.list_fragment,

```

```

container, false);
        ArrayAdapter<String> adapter = new
ArrayAdapter<String>(getActivity(),
                    android.R.layout.simple_list_item_1,
AndroidOS);
        setListAdapter(adapter);
        return view;
    }

@Override
public void onAttach(Context context) {
    super.onAttach(context);
    if (context instanceof
OnFragmentInteractionListener) {
        mListener = (OnFragmentInteractionListener)
context;
    } else {
        throw new RuntimeException(context.toString()
                + " mora implementirati odgovarajući
OnFragmentInteractionListener");
    }
}
@Override
public void onDetach() {
    super.onDetach();
    mListener = null;
}

@Override
public void onListItemClick(ListView l, View v, int
position, long id) {

getListView().setSelector(android.R.color.holo_blue_dark);
    if (mListener != null) {
        mListener.menuItemSelected(AndroidOS[position],
"Version : " + Version[position]);
    }
}

public interface OnFragmentInteractionListener {
    void menuItemSelected (String os, String Version);
}
}

```

Glavna aktivnost mora implementirati definisani OnFragmentInteractionListener, pomoću koga će primiti informaciju o tome koji element liste sa leve strane je odabran, a zatim pozvati DetaljiFragment i

proslediti mu informacije koje treba da prikaže. MainActivity.java je dat na sledeći način:

```
package com.example.mzivkovic.myapplication;

import android.support.v4.app.FragmentActivity;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

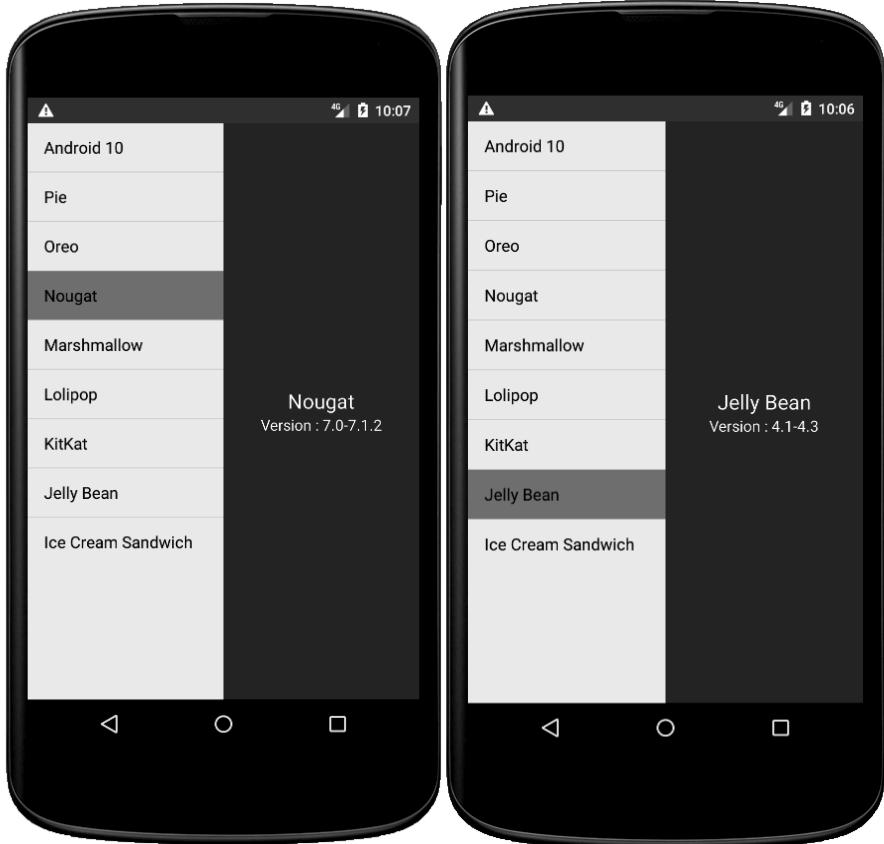
public class MainActivity extends FragmentActivity
implements MeniFragment.OnFragmentInteractionListener{

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

    @Override
    public void menuItemSelected(String os, String version)
{
    DetaljiFragment detalji = (DetaljiFragment)
getFragmentManager().findFragmentById(R.id.fragmentDetalji);
;
    detalji.change(os, version);

}
}
```

Nakon pokretanja, izgled aplikacije prikazan je na slici 8.6.



Slika 8.6, korisnički interfejs aplikacije sa dva fragmenta

9. Perzistencija podataka

Veliki broj Android aplikacija ima potrebu da trajno čuva neke podatke. Android sistem koristi sistem fajlova koji se sličan sistemima fajlova na diskovima koje koriste druge platforme. Postoji nekoliko različitih načina na koji Android aplikacija može čuvati podatke:

- Prostor za smeštanje podataka vidljivih samo aplikaciji – služi za čuvanje fajlova koje treba da upotrebljava samo aplikacija, ili u okviru dodeljenog direktorijuma u internom prostoru ili u odgovarajućim dodeljenim direktorijumima u okviru eksternog prostora. Interni prostor se može koristiti za smeštanje osetljivih informacija kojima druge aplikacije ne treba da pristupaju.
- Deljeni prostor – služi za smeštanje fajlova koje aplikacija deli sa drugim aplikacijama, poput multimedijalnog sadržaja ili dokumenata.
- Preferencije – služe za smeštanje privatnih primitivnih podataka u obliku parova ključ – vrednost.
- Baze podataka – služe za čuvanje strukturiranih podataka u privatnoj bazi podataka na uređaju.

Odabir odgovarajućeg rešenja za konkretnu aplikaciju zavisi od specifičnih potreba te aplikacije. Na primer, interni prostor za smeštanje podataka ima ograničen prostor, pa nije pogodan za smeštanje veće količine podataka. Ukoliko se smeštaju podaci kojima i druge aplikacije mogu pristupati, može se koristiti deljeni prostor. Za strukturirane podatke se mogu koristiti ili preferencije (parovi ključ-vrednost) ili baza podataka (ukoliko podaci sadrže više od dve kolone). Na kraju, ukoliko se smeštaju osetljivi podaci kojima druge aplikacije ne treba da pristupaju, može se koristiti interni prostor, preferencije ili baza.

9.1. SharedPreferences

U slučaju da je potrebno čuvati malu količinu podataka u obliku parova ključ-vrednost, preporučuje se upotreba SharedPreferences API-ja. SharedPreferences objekat pokazuje na fajl u kome se čuvaju parovi ključ-vrednost i pruža jednostavne metode kojima se može vršiti upis ili čitanje. Kreiranje novog fajla ili pristup postojećem se može postići pozivom jedne od dve dostupne metode:

- `getSharedPreferences()` – koristi se u slučaju da je potrebno više fajlova identifikovanih po imenu koje se prosleđuje kao prvi parametar metode. Ova metoda se može pozvati iz bilo kog konteksta u okviru aplikacije.

- `getPreferences()` – koristi se iz aktivnosti ukoliko je potreban samo jedan fajl za tu aktivnost. Pošto vraća podrazumevani fajl koji pripada toj aktivnosti, nije potrebno proslediti ime.

Na primer, pristup `SharedPreferences` fajlu čije se ime nalazi definisano u String resursima pod identifikatorom `R.string.preference_kljuc`, i otvaranje tog fajla u privatnom režimu tako da je dostupan samo aplikaciji se može postići na sledeći način:

```
Context context = getActivity();
SharedPreferences sharedPref = context.getSharedPreferences(
getString(R.string.preference_kljuc),
Context.MODE_PRIVATE);
```

Prilikom imenovanja fajla za `SharedPreferences`, potrebno je odabratim ime koje je jedinstveno. Zbog toga se često imenu fajla daje prefiks sa identifikatorom aplikacije, poput: "com.example.myapp.PREFERENCE_KLJUC".

Ukoliko je potreban samo jedan fajl za aktivnost, može se koristiti alternativna metoda `getPreferences()` u sledećem obliku:

```
SharedPreferences sharedPref =
getActivity().getPreferences(Context.MODE_PRIVATE);
```

Bitna napomena se odnosi na tip pristupa. Uvek se bira `MODE_PRIVATE`, koji označava privatni pristup samo iz date aplikacije. Tipovi `MODE_WORLD_READABLE` i `MODE_WORLD_WRITEABLE` se smatraju zastarelim počevši od API nivoa 17. Od Android 7.0, odnosno API nivoa 24, Android čak izbacuje `SecurityException` pri pokušaju upotrebe ovih vrednosti.

Ukoliko se `SharedPreferences` API koristi za čuvanje podešavanja aplikacije, preporučuje se poziv `getDefaultSharedPreferences()` koji vraća podrazumevani fajl za celu aplikaciju.

Nakon dohvatanja fajla, upis se vrši kreiranjem `SharedPreferences.Editor` objekta pozivom metode `edit()` nad `SharedPreferences` objektom. Ključevi i vrednosti se upisuju pozivom metoda poput `putInt()` ili `putString()`. Na kraju se poziva metoda `commit()` koja će sačuvati sve izmene:

```
SharedPreferences sharedPref =
getActivity().getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(getString(R.string.ostvarenipoeni_kljuc),
ostvarenipoeni);
editor.commit();
```

Metoda `commit()` upisuje podatke sinhrono, što može dovesti do pauziranja korisničkog interfejsa. Asinhroni upis se može postići upotrebom metode `apply()` umesto metode `commit()`.

Čitanje iz fajla se postiže pozivom metoda `getInt()` i `getString()`, pri čemu se prosleđuje ključ vrednosti koju je potrebno dohvatiti, uz opciono podrazumevanu vrednost ukoliko se ključ ne pronađe u fajlu:

```
SharedPreferences sharedPref =  
getActivity().getPreferences(Context.MODE_PRIVATE);  
int podrazumevanaVrednost = 0;  
int ostvarenipoeni =  
sharedPref.getInt(getString(R.string.ostvarenipoeni_kljuc)  
, podrazumevanaVrednost);
```

9.2. Interni prostor

Interni prostor čine direktorijumi kojima pristup ima samo aplikacija, odnosno sistem sprečava sve druge aplikacije da pristupe ovim lokacijama. Dodatno, ove lokacije su kriptovane, počev od Android 10 (API nivo 29). Ove karakteristike čine ove direktorijume dobrom mestom za smeštanje podataka kojima samo aplikacija treba da pristupi.

Android sistem svakoj aplikaciji dodeljuje direktorijume u okviru internog prostora gde aplikacija može čuvati svoje podatke. Jedan direktorijum se odnosi na fajlove koji se trajno čuvaju, dok se drugi direktorijum koristi za keširane fajlove. Aplikaciji nisu potrebne nikakve sistemske dozvole za čitanje i upis u ove direktorijume. Bitne osobine ovih direktorijuma su:

- Drugim aplikacijama je pristup ovim direktorijumima zabranjen.
- Ove lokacije su relativno malog kapaciteta, pa nisu pogodne za smeštanje većih količina podataka.

Trajni (perzistentni) fajlovi se čuvaju u direktorijumu `filesDir`, kome se može pristupiti pomoću `getFilesDir()` metode `Context` objekta. Za pristup fajlovima može se koristiti File API, ali preporučeni način je upotreba toka podataka (engl. *stream*). Pozivom `openFileOutput()` metode dohvata se `FileOutputStream` koji upisuje u fajl u okviru `filesDir` direktorijuma. Primer upisa teksta u fajl pomoću toka podataka dat je sledećim isečkom Java koda:

```
String imeFajla = "moj_fajl";  
String sadrzajFajla = "Zdravo!";  
try (FileOutputStream fos = context.openFileOutput(imeFajla,  
Context.MODE_PRIVATE)) {  
    fos.write(sadrzajFajla.toByteArray());  
}
```

Pristup fajlu i čitanje sadržaja pomoću toka podataka se postiže metodom `openFileInput()`, kao što je prikazano sledećim isečkom Java koda:

```

FileInputStream fis = context.openFileInput(imeFajla);
InputStreamReader inputStreamReader =
        new InputStreamReader(fis, StandardCharsets.UTF_8);
StringBuilder stringBuilder = new StringBuilder();
try (BufferedReader reader = new
BufferedReader(inputStreamReader)) {
    String line = reader.readLine();
    while (line != null) {
        stringBuilder.append(line).append('\n');
        line = reader.readLine();
    }
} catch (IOException e) {
    // Doslo je do greske prilikom pokusaja citanja iz
fajla.
} finally {
    String sadrzajFajla = stringBuilder.toString();
}

```

9.3. Eksterni prostor

Ukoliko interni prostor ne pruža dovoljno prostora, može se koristiti eksterni prostor. Android sistem nudi direktorijume u okviru eksternog prostora gde aplikacija može čuvati svoje podatke. Slično kao kod internog prostora, jedan direktorijum se koristi za trajne podatke aplikacije, dok drugi služi za čuvanje keširanih fajlova aplikacije. Ipak, treba donekle biti oprezan, jer ovi direktorijumi nisu garantovano dostupni, na primer ukoliko se SD kartica ukloni sa uređaja. Ukoliko funkcionalnost same aplikacije zavisi od ovih fajlova, bolje je čuvati ih u okviru internog prostora.

Počev od Androida 4.4 (API nivo 19), aplikacija ne mora da traži posebne dozvole za pristup ovim direktorijumima. Fajlovi smešteni u ovim direktorijumima se uklanjaju prilikom deinstalacije aplikacije.

Pošto se eksterni prostor nalazi na fizičkom medijumu poput SD kartice koju korisnik može da ukloni, potrebno je proveriti da li je eksterni prostor dostupan pre pokušaja upisa ili čitanja. Provera stanja se vrši pozivom Environment.getExternalStorageState(). Ukoliko ova metoda vrati stanje MEDIA_MOUNTED, moguće je vršiti i upis i čitanje fajlova u okviru eksternog prostora. Ukoliko je vraćeno stanje MEDIA_MOUNTED_READ_ONLY, dozvoljeno je samo čitanje ovih fajlova. Često se zbog toga koriste sledeće dve metode koje utvrđuju da li je eksterni prostor dostupan:

```

// Provera da li je eksterni prostor dostupan
// i za citanje i za pisanje.
private boolean isExternalStorageWritable() {

```

```

        return Environment.getExternalStorageState() == Environment.MEDIA_MOUNTED;
    }

// Provera da li je eksterni prostor dostupan bar za citanje.
private boolean isExternalStorageReadable() {
    return Environment.getExternalStorageState() == Environment.MEDIA_MOUNTED ||
           Environment.getExternalStorageState() == Environment.MEDIA_MOUNTED_READ_ONLY;
}

```

Za pristup aplikacijnim fajlovima koji se nalaze u eksternom prostoru koristi se poziv metode `getExternalFilesDir()`, kao u sledećem isečku Java koda:

```

File appSpecificExternalDir = new
File(context.getExternalFilesDir(), imeFajla);

```

Na sličan način se radi i sa keširanim fajlovima, samo što se u tom slučaju koristi poziv metode `getExternalCacheDir()`. Dodavanje i uklanjanje privremenih fajlova se radi na način opisan sledećim isečkom Java koda:

```

File externalCacheFile = new
File(context.getExternalCacheDir(), imeFajla);

...
externalCacheFile.delete();

```

Ukoliko aplikacija radi sa multimedijalnim fajlovima koji su od značaja korisniku samo u okviru te aplikacije, najbolje rešenje je da se smeste u predefinisane direktorijume u okviru eksternog prostora koji pripadaju aplikaciji, kao što je prikazano sledećim primerom Java koda:

```

@Nullable
File getAppSpecificAlbumStorageDir(Context context, String imeAlbuma) {
    // Postavljanje direktorijuma sa slikama u okviru direktorijuma koji pripada aplikaciji a nalazi se u okviru eksternog prostora.
    File file = new File(context.getExternalFilesDir(
        Environment.DIRECTORY_PICTURES), imeAlbuma);
    if (file == null || !file.mkdirs()) {
        Log.e(LOG_TAG, "Direktorijum nije kreiran uspesno");
    }
    return file;
}

```

DIRECTORY_PICTURES je jedna od konstanti koju pruža Environment API, i osigurava da sistem tretira fajlove u okviru ovih direktorijuma na odgovarajući način. Predefinisane sistemske vrednosti direktorijuma deklarisanih u okviru Environment API-ja koje se najčešće upotrebljavaju su:

- DIRECTORY_PICTURES – standardni direktorijum u kome se smeštaju slike dostupne korisniku.
- DIRECTORY_DOCUMENTS – standardni direktorijum u kome se smeštaju dokumenti koje korisnik kreira.
- DIRECTORY_DOWNLOADS – standardni direktorijum u kome se smeštaju fajlovi koje je korisnik preuzeo sa Interneta.
- DIRECTORY_MUSIC – standardni direktorijum u kome se smeštaju audio fajlovi koji treba da budu vidljivi u regularnoj muzičkoj listi koja je dostupna korisniku.
- DIRECTORY_DCIM – tradicionalna lokacija za slike i video snimljene kamerom.

9.4. SQLite baza podataka

Ukoliko je potrebno čuvati strukturirane podatke može se koristiti SQLite baza podataka. Primeri strukturiranih podataka su kontakti, knjige, proizvodi, odnosno uopšteno gledano svaki složeni tip podatka koji ima više kolona. U nastavku ovog poglavlja smatra se da je čitalac upoznat sa SQL bazama podataka i da poznaje osnovne termine. API potreban za rad sa bazom u Androidu je dostupan u paketu android.database.sqlite. Potrebno je samo naglasiti da Android zajednica preporučuje upotrebu Room biblioteke u radu sa bazom na Androidu, o čemu će takođe biti više reči u nastavku.

9.4.1. Definisanje šeme i ugovora

Osnovni princip svake SQL baze je njena šema, koja predstavlja formalnu deklaraciju organizacije same baze. Praktično, šema je predstavljena kroz SQL naredbe koje se koriste za kreiranje baze. Jedan od najčešćih pristupa koji može pomoći u definisanju šeme je kreiranje klase ugovora, koja eksplicitno opisuje šemu na sistematičan način. Klasa ugovor je zapravo nosilac konstanti koje definišu imena tabela i kolona. Upotreba konstanti je značajna pošto se one mogu potom koristiti u svim drugim klasama u okviru istog paketa. Time se dobija jedno centralno mesto za održavanje baze – dovoljno je na primer izmeniti ime kolone samo u klasi ugovora, što će biti automatski propagirano kroz kod.

Dobar način organizacije ove klase je da se globalne definicije koje važe za celu bazu stave u koreni nivo klase. Potom se može kreirati unutrašnja klasa za svaku pojedinačnu tabelu u bazi, koja će definisati odgovarajuće kolone te tabele. Dodatno, implementacijom interfejsa BaseColumns, unutrašnja klasa nasleđuje polje primarni ključ sa identifikatorom _ID, koje je veoma pogodno u radu sa nekim Android klasama koje ga očekuju, poput CursorAdapter klase. Na primer, sledeći kod definiše klasu ugovora za bazu kontakata:

```
public final class ContactsDBContract {
    //sprecavanje slučajnoginstanciranja ove klase
    private ContactsDBContract() {}

    /* Unutrašnja klasa koja definise sadrzaj tabele */
    public static class ContactEntry implements BaseColumns {
        public static final String TABLE_NAME = "contact";
        public static final String COLUMN_NAME = "name";
        public static final String COLUMN_EMAIL = "email";
        public static final String COLUMN_PHONE = "phone";
    }
}
```

9.4.2. Kreiranje baze pomoću SQL pomoćne klase

Android bazu podataka smešta u okviru privatnog direktorijuma aplikacije, poput fajlova koje čuva u okviru internog prostora. Time su podaci potpuno zaštićeni, pošto ovom direktorijumu druge aplikacije nemaju pristup. Android nudi API koji je veoma koristan u održavanju baze podataka, a nalazi se u okviru klase SQLiteOpenHelper. Upotreba ove klase je praktično neophodna za ispravan rad sa SQLite bazom i preporučuje se od strane Android zajednice.

Nakon definisanja šeme tabele, najčešće je potrebno implementirati metode koje služe za održavanje baze podataka. Koriste se tipične SQL naredbe, poput naredbi za kreiranje i uklanjanje tabele, čiji su primjeri dati u nastavku:

```
private static final String SQL_CREATE_ENTRIES =
    "CREATE TABLE " + ContactEntry.TABLE_NAME + " (" +
        ContactEntry._ID + " INTEGER PRIMARY KEY," +
        ContactEntry.COLUMN_NAME + " TEXT," +
        ContactEntry.COLUMN_EMAIL + " TEXT," +
        ContactEntry.COLUMN_PHONE + " TEXT)";

private static final String SQL_DELETE_ENTRIES =
    "DROP TABLE IF EXISTS " + ContactEntry.TABLE_NAME;
```

SQLiteOpenHelper se koristi za dohvatanje reference nad bazom podataka. Potencijalno dugotrajne operacije poput kreiranja ili ažuriranja baze podataka izvršava sam sistem, i to samo kada je neophodno i ne prilikom pokretanja aplikacije. SQLiteOpenHelper nudi metode getWritableDatabase() i getReadableDatabase() pomoću kojih se može dohvatiti referenca na bazu podataka u modu za upis ili čitanje podataka. Ove operacije mogu biti dugotrajne, pa predstavljaju dobre kandidate za implementaciju kroz AsyncTask koji se odvija u pozadinskoj niti.

Kako bi se koristio SQLiteOpenHelper, potrebno je napraviti izvedenu klasu koja će implementirati callback metode onCreate() i onUpgrade(). Moguće je implementirati i metode onDowngrade() i onOpen(), ali to nije neophodno. Primer implementacije koji koristi SQL komande definisane ranije dat je u nastavku:

```
public class ContactDatabase extends SQLiteOpenHelper {

    // Prilikom promene seme, mora se inkrementirati
    // verzija
    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME =
    "Contacts.db";

    public ContactDatabase (Context context) {
        super(context, DATABASE_NAME, null,
    DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase sqLiteDatabase) {
        sqLiteDatabase.execSQL(SQL_CREATE_ENTRIES);
    }

    @Override
    public void onUpgrade(SQLiteDatabase sqLiteDatabase,
    int oldVersion, int newVersion) {
        // Zbog jednostavnosti prilikom upgrade prosto se
        // brise sve i poziva onCreate
        sqLiteDatabase.execSQL(SQL_DELETE_ENTRIES);
        onCreate(sqLiteDatabase);
    }
}
```

Nakon kreiranja, bazi se pristupa pomoću objekta upravo definisane klase:

```
ContactDatabase dbHelper = new
ContactDatabase(getContext());
```

U svakoj bazi postoje četiri osnovne operacije: ubacivanje novih podataka, čitanje podataka, ažuriranje podataka i brisanje podataka. Ove operacije su opisane u nastavku.

9.4.3. Ubacivanje podataka u bazu

Najosnovnija operacija sa bazom podataka jeste unos podataka u bazu. Za to se koristi insert() metod, koji kao parametar prihvata ContentValues objekat. Ovaj objekat služi kao nosač vrednosti koje je potrebno uneti u bazu, a funkcioniše po principu mape (parovi ključ-vrednost) u kojoj su imena kolona ključevi. Metoda koja bi služila za unos jednog kontakta u bazu data je sledećim delom Java koda:

```
public void addContact (String name, String email, String phone) {
    //id ne treba, automatski ce se dohvatiti
    //moramo dohvatiti bazu sa dozvolom za upis sadzaja
    SQLiteDatabase db = this.getWritableDatabase();
    //da bismo dodali nesto u bazu, potreban nam je set
    vrednosti - ContentValues
    //imena kolona se koriste kao kljuccevi
    ContentValues cv = new ContentValues();

    //sada isto kao bundle
    cv.put(ContactsDBContract.ContactEntry.COLUMN_NAME,
name);
    cv.put(ContactsDBContract.ContactEntry.COLUMN_EMAIL,
email);
    cv.put(ContactsDBContract.ContactEntry.COLUMN_PHONE,
phone);

    db.insert(ContactsDBContract.ContactEntry.TABLE_NAME,
null, cv);
}
```

Metoda insert uzima tri argumenta. Prvi argument predstavlja ime tabele u koju se vrši unos. Drugi parametar govori sistemu šta da radi ukoliko je ContentValues objekat prazan (nisu unete nikakve vrednosti). Kada je specificirano null (kao u primeru), sistem neće ubaciti red u tabelu ukoliko je ContentValues objekat prazan. Metoda insert() kao rezultat vraća ID upravo kreiranog zapisa u bazi ili -1 ukoliko je došlo do greške pri unosu (na primer konflikt sa već postojećim podacima u bazi).

9.4.4. Čitanje podataka iz baze

Druga osnovna operacija nad bazom podataka jeste čitanje podataka. U tu svrhu koristi query() metoda, kojoj se prosleđuju kriterijum selekcije i odabране kolone. Rezultati upita se vraćaju u obliku Cursor objekta. Parametri selection i selectionArgs u query() metodi se kombinuju zajedno kako bi se formirala where klauzula. Pošto se argumenti dostavljaju odvojeno od samog upita (vrši se *escape* pre njihovog spajanja), može se sasvim bezbedno smatrati da je ovaj način potpuno zaštićen od SQL injekcije. Primer metode koja čita kontakte iz baze podataka i vraća sve kolone za kontakta čije je ime prosleđeno kao parametar metode dat je u nastavku sledećim delom Java koda.

```
public void readContact(String myName) {
    SQLiteDatabase db = this.getReadableDatabase();
    // Projekcija koja specificira koje kolone iz baze
    // su potrebne.
    String[] projection = {
        BaseColumns._ID,
        ContactsDBContract.ContactEntry.COLUMN_NAME,
        ContactsDBContract.ContactEntry.COLUMN_EMAIL,
        ContactsDBContract.ContactEntry.COLUMN_PHONE
    };
    // Filtriranje rezultata WHERE "name" = 'myName'
    String selection =
        ContactsDBContract.ContactEntry.COLUMN_NAME + " = ?";
    String[] selectionArgs = { myName };
    // Sortiranje rezultata u rezultujućem Cursor objektu
    String sortOrder =
        ContactsDBContract.ContactEntry.COLUMN_PHONE +
        " DESC";
    Cursor cursor = db.query(
        ContactsDBContract.ContactEntry.TABLE_NAME, // tabela
        projection, // kolone koje se vracaju (null vraca sve)
        selection, // kolone za WHERE klauzulu
        selectionArgs, // vrednostu WHERE klauzule
        null, // ne grupisi redove
        null, // ne filtriraj dodatno redove po grupama
        sortOrder // sortiranje
    );
    ...
    // uraditi nesto sa Cursor objektom
}
```

Cursor objekat sadrži rezultujući skup podataka. Kako bi se dohvatio jedan red iz Cursor objekta, koristi se jedna od metoda koje pomeraju kurSOR (neophodno je

pozvati jednu od ovih metoda zbog toga što kurzor počinje na poziciji -1). Na primer, moveToNext() postavlja poziciju za čitanje na prvi red rezultujućeg skupa podataka. Skup bitnih metoda za pomeranje kurzora uključuje:

- move(int pomeraj) – pomera kurzor za zadati pomeraj od trenutne pozicije.
- moveToFirst() – pozicionira kurzor na prvi red rezultujućeg skupa podataka.
- moveToLast() – pozicionira kurzor na poslednji red rezultujućeg skupa.
- moveToNext() – pomera kurzor na sledeći red.
- moveToPosition(int pozicija) – pomera kurzor na zadatu poziciju.
- moveToPrevious() – pomera kurzor na prethodni red.

Vrednosti se „čupaju“ iz svake kolone reda pozivom odgovarajućih Cursor metoda, poput getString() ili getInt(). Kolona iz koje se čita vrednost se precizira svojim indeksom, koji se može dobiti pozivom getColumnIndex(). Kada se završi sa iteracijom kroz rezultate, poziva se close() metoda koja na kraju oslobađa resurse. Sledeći deo koda pokazuje kako bi se dohvatali brojevi telefona svih kontakata koji se nalaze u rezultujućem kurzoru i smestili u listu:

```
List<String> numbers = new ArrayList<>();
while(cursor.moveToFirst()) {
    String number = cursor.getString(
cursor.getColumnIndex(ContactContract.
ContactEntry.COLUMN_PHONE));
    numbers.add(number);
}
cursor.close();
```

9.4.5. Brisanje podataka iz baze

Prilikom brisanja podataka iz tabele, potrebno je dati kriterijum odabira na osnovu koga se identifikuju redovi za brisanje i proslediti ga delete() metodi. Mehanizam radi identično kao u slučaju kriterijuma selekcije kod query() metode. I ovde se razdvaja specifikacija selekcije na klauzulu i na argumente selekcije. Klauzula definiše koje kolone se traže, a argumenti predstavljaju vrednosti koje se traže u tim kolonama. Isto kao kod query(), može se smatrati da je ovaj način zaštićen od SQL injekcije. Primer brisanja kontakta prema zadatom imenu dat je u nastavku:

```
public int deleteContact(String name) {
    SQLiteDatabase db = this.getWritableDatabase();
    // definisati 'where' klauzulu.
    String selection =
```

```

ContactsDBContract.ContactEntry.COLUMN_NAME + " LIKE ?";
    // specificirati argumente za where klauzulu.
    String[] selectionArgs = { name };
    // izvršiti SQL naredbu.
    int deletedRows =
db.delete(ContactsDBContract.ContactEntry.TABLE_NAME,
selection, selectionArgs);
    return deletedRows;
}

```

Metoda delete() vraća celobrojnu vrednost koja označava broj redova na koje je SQL naredba uticala (engl. *rows affected*).

9.4.6. Ažuriranje podataka u bazi

Prilikom ažuriranja podataka u bazi koristi se metoda update(). Ažuriranje tabele koristi objekat tipa ContentValues koji sadrži nove vrednosti koje je potrebno upisati za već postojeće podatke u bazi. Sama sintaksa veoma liči na mešavinu sintakse insert() metode sa objektom ContentValues i sintakse where klauzule metode delete(). Primer ažuriranja kontakta u bazi dat je sledećim isečkom koda:

```

public int updateContact(String name, String newPhone) {
    SQLiteDatabase db = this.getWritableDatabase();
    // Nova vrednost za jednu kolonu
    ContentValues values = new ContentValues();
    values.put(ContactsDBContract.ContactEntry.COLUMN_PHONE,
newPhone);
    // Red koji se azurira se pronađi po imenu kontakta
    String selection =
ContactsDBContract.ContactEntry.COLUMN_NAME + " LIKE ?";
    String[] selectionArgs = { name };
    int count = db.update(
        ContactsDBContract.ContactEntry.TABLE_NAME,
        values,
        selection,
        selectionArgs);
    return count;
}

```

Metoda update() vraća celobrojnu vrednost koja označava broj redova na koje je SQL naredba uticala (engl. *rows affected*).

9.4.7. Perzistencija konekcije

Metode `getWritableDatabase()` i `getReadableDatabase()` se smatraju skupim operacijama za pozivanje ukoliko je baza zatvorena. Zbog toga, dobra praksa je da se konekcija ka bazi drži otvorenom sve dok postoji šansa da će aplikaciji biti potreban pristup. Tipično, optimalni trenutak zatvaranja konekcije ka bazi je u callback metodi `onDestroy()` aktivnosti koja komunicira sa bazom. Primer implementacije `onDestroy()` metode sa zatvaranjem konekcije dat je u nastavku:

```
@Override  
protected void onDestroy() {  
    dbHelper.close();  
    super.onDestroy();  
}
```

9.5. Room biblioteka

SQLite API je veoma moćan, ali ima nekoliko nedostataka. Rad sa SQLite bazom podataka je na niskom nivou, i zahteva dosta vremena za razvoj. Osim toga, ne postoji verifikacija sirovih SQL upita u vreme kompajliranja, pa se SQL upiti moraju manuelno održavati, što je podložno greškama. Još jedan nedostatak je upotreba viška ponovljenog koda (engl. *boilerplate code*) kako bi se uradila konverzija između SQL upita i programskih objekata.

Room pruža nivo apstrakcije iznad SQLite baze kako bi se omogućio jednostavan i tečan pristup bazi podataka, uz očuvanje pune sirove snage SQLite. To je od velikog značaja za aplikacije koje rukuju većim količinama strukturiranih podataka, koje obično čuvaju te podatke lokalno (kao vid keširanja, odnosno privremenog čuvanja). U slučaju da uređaj ne može da pristupi mobilnoj mreži ili Internetu, korisnik i dalje može da pristupi podacima. Ukoliko korisnik promeni neke podatke, oni se sinhronizuju sa serverom kada se uređaj ponovo poveže sa mrežom. Room se brine o svim ovim aspektima čuvanja podataka.

Room je zvanično objavljen 2016. godine kao jedan od najbitnijih alata u okviru Android Architectural Components (kolekcija biblioteka koje omogućavaju dizajn robusnih aplikacija), a se koristi za manipulaciju podacima u okviru Android aplikacija. Pruža veoma jednostavan način za rad sa podacima pri čemu osigurava sigurnost i integritet podataka. Kako bi se koristio Room u okviru aplikacije, neophodno je dodati zavisnosti u build.gradle fajlu aplikacije. Takođe, mora se dodati Google Maven rezervorijum. Sve zavisnosti koje su potrebne date su sledećim delom koda build.gradle fajla:

```
dependencies {
    def room_version = "2.2.3"
    implementation "androidx.room:room-runtime:$room_version"
    annotationProcessor "androidx.room:room-
compiler:$room_version" // For Kotlin use
    // optional - Kotlin Extensions and Coroutines
    implementation "androidx.room:room-ktx:$room_version"

    // optional - RxJava support for Room
    implementation "androidx.room:room-rxjava2:$room_version"

    // optional - Guava support for Room
    implementation "androidx.room:room-guava:$room_version"

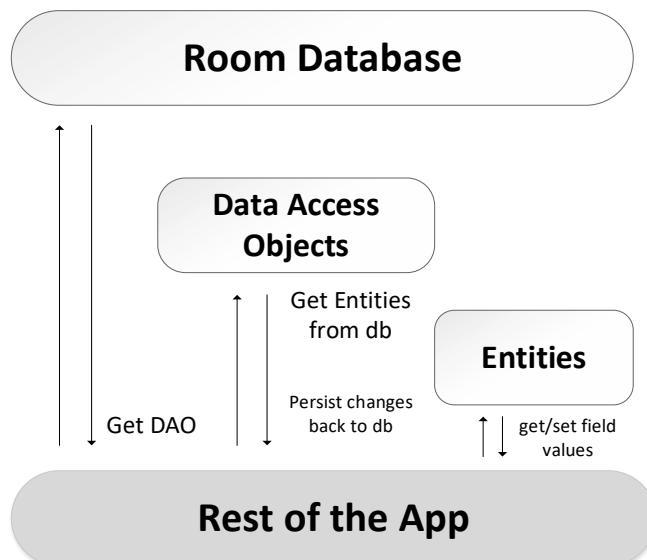
    // Test helpers
    testImplementation "androidx.room:room-
testing:$room_version"
}
```

Room nije ORM (engl. *Object-Relational Mapping*) već kompletna biblioteka koja omogućava programerima kreiranje i manipulaciju SQLite baza podataka na lak i siguran način. Upotreboom anotacija mogu se definisati baze, tabele i operacije. Room automatski prevodi anotacije u SQLite upite kako bi izvršio odgovarajuće operacije nad bazom podataka.

Room se sastoji od tri glavne komponente:

- Database: služi kao nosilac same baze podataka i predstavlja glavnu tačku pristupa aplikaciji ka konekciji. Ova klasa se obeležava anotacijom `@Database` i treba da ispunjava sledeće uslove:
 - Mora biti apstraktna klasa koja proširuje klasu `RoomDatabase`.
 - Sadrži listu entiteta koji se nalaze u okviru anotirane baze.
 - Sadrži apstraktnu metodu koja ima 0 argumenata i koja vraća objekat klase anotirane sa `@Dao`.
- Entity: označava tabelu u bazi podataka.
- DAO: skraćeno od Data Access Object, sadrži metode koje se koriste za pristup bazi podataka.

Aplikacija koristi Room bazu podataka kako bi dohvatile DAO objekte povezane sa tom bazom podataka. Nakon toga, aplikacija koristi DAO objekte da dohvati entitete iz baze podataka i sačuva eventualne promene nad tim entitetima nazad u bazi podataka. Na kraju, aplikacija koristi entitet da manipuliše vrednostima koje odgovaraju poljima u kolonama tabele u okviru baze podataka. Osnovni koncept odnosa između različitih komponenti dat je na slici 9.1.



Slika 9.1, Room - osnovni dijagram arhitekture i odnosa između komponenti

Upotreba Room alata ilustrovana je primerom sa jednostavnom bazom korisnika (tabela user), pri čemu su potrebni jedan entitet i jedan DAO objekat. Java kod koji opisuje entitet User dat je na sledeći način:

```
@Entity
public class User {
    @PrimaryKey
    public int uid;

    @ColumnInfo(name = "first_name")
    public String firstName;

    @ColumnInfo(name = "last_name")
    public String lastName;
}
```

Klasa User predstavlja entitet koji direktno odgovara jednom zapisu u bazi podataka. Može se uočiti da je klasa označena sa više različitih anotacija. Na primer, sama klasa se anotira sa `@Entity`, primarni ključ se anotira sa `@PrimaryKey`, dok se kolone kojima odgovaraju polja klase anotiraju sa `@ColumnInfo`, pri čemu se daje i dodatni atribut (ime kolone u bazi).

UserDao klasa predstavlja DAO objekat, koji ima ulogu da dohvata entitete iz baze podataka. Ona je anotirana sa `@Dao`, i sadrži upite koji se koriste u radu sa bazom podataka, anotirane sa `@Query`. Konkretan primer DAO klase omogućava dohvatanje svih korisnika, dohvatanje korisnika po Id, kao i po imenu i prezimenu. Dodatno, moguće je uraditi insert novih korisnika, kao i brisanje konkretnog korisnika iz baze. Implementacija ove klase data je sledećim Java kodom:

```
@Dao
public interface UserDao {
    @Query("SELECT * FROM user")
    List<User> getAll();

    @Query("SELECT * FROM user WHERE uid IN (:ids)")
    List<User> getAllByIds(int[] ids);

    @Query("SELECT * FROM user WHERE first_name LIKE :first
AND " + "last_name LIKE :last LIMIT 1")
    User findByName(String first, String last);

    @Insert
    void insertAll(User... users);

    @Delete
    void delete(User user);
}
```

Kod klase koja predstavlja nosioca same baze podataka obavezno sadrži anotaciju @Database, kao i listu svih entiteta (u ovom slučaju samo User). Ova klasa obavezno mora da bude izvedena iz klase RoomDatabase i da bude deklarisana kao apstraktna. Primer implementacije dat je sledećim Java kodom:

```
@Database(entities = {User.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract UserDao userDao();
}
```

Nakon kreiranja svih navedenih fajlova, instanca baze se može dohvatiti sledećim kodom:

```
AppDatabase db =
Room.databaseBuilder(getApplicationContext(),
    AppDatabase.class, "database-name").build();
```

Preporuka je da se prilikom kreiranja instance AppDatabase objekta koristi Java šablon Singleton, pošto je svaka instanca RoomDatabase skupa po pitanju resursa, a retko je potrebno imati više od jedne instance u okviru jednog procesa u kome se aplikacija izvršava.

9.6. Pitanja za vežbu

- Koji sve načini prezistencije podataka postoje u Android platformi?
- Od čega zavisi odabir načina prezistencije podataka za konkretnu aplikaciju?
- Koji je najbolji način za čuvanje primitivnih tipova podataka u obliku ključ – vrednost?
- Šta je interni prostor?
- Da li druge aplikacije mogu pristupati podacima u internom prostoru?
- Šta je eksterni prostor i gde se on nalazi?
- Da li su aplikaciji potrebne određene dozvole u radu sa eksternim prostorom?
- Kako se koristi SharedPreferences API?
- Kako se dohvata editor za SharedPreferences?
- Koja je razlika između commit() i apply()?
- Da li je moguće koristiti neki drugi mod za upis osim privatnog moda?
- Koje su najčešće korišćene konstante iz Environment API koje označavaju tipične direktorijume u eksternom prostoru?
- Kako se čuvaju privremeni fajlovi u okviru internog i eksternog prostora?
- Da li fajlovi u okviru internog i eksternog prostora dodeljenog aplikaciji ostaju na uređaju i nakon deinstalacije aplikacije?
- Kada je pogodno koristiti SQLite bazu podataka?
- Kako se definiše šema baze i ugovor za SQLite bazu podataka?
- Koja se pomoćna klasa SQLite API-ja koristi za kreiranje baze podataka?
- Kada se koristi getReadableDatabase() metoda, a u kojim slučajevima se mora pozvati getWritableDatabase()?
- Šta je Cursor objekat?
- Koje metode Cursor objekat pruža za kretanje kroz rezultate u skupu podataka?
- Šta je Room biblioteka i čemu služi?
- Da li je bolje raditi direktno sa SQLite, ili upotrebiti Room? Prodiskutovati prednosti i mane oba pristupa.
- Kako se dodaje Room u projekat aplikacije?
- Šta je entitet?
- Čemu služi DAO (Data Access Object)?
- Objasniti kako se u klasama koriste anotacije potrebne za Room biblioteku.
- Objasniti dijagram arhitekture Room-a, kao i odnose između različitih komponenti.

9.7. Zadaci za vežbu

Primer 1: Napraviti jednostavnu aplikaciju koja od korisnika traži da unese tri podatka, svoje ime i prezime, svoju adresu i svoj broj telefona. Potrebno je sačuvati unete podatke pomoću SharedPreferences API-ja. Obezbediti dva dugmeta, jedno za ručno čuvanje unetih podataka, i drugo, za čitanje sačuvanih podataka i njihov prikaz u predefinisanim poljima za unos. Takođe, obezbediti da se prilikom poziva onStop() metode trenutno uneti podaci sačuvaju, a prilikom onStart() metode pročitaju i postave u predefinisana polja kako bi korisnik mogao da nastavi sa popunjavanjem tamo gde je stao.

Rešenje: Korisnički interfejs se sastoji od tri labele (koje označavaju šta je potrebno uneti u odgovarajuće polje), tri polja za unos podataka, kao i dva dugmeta, za čuvanje i čitanje podataka. Odgovarajući fajl activity_main.xml ima sledeći sadržaj:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
    android:layout_width="match_parent"
        android:layout_height="match_parent">
    <TextView
        android:id="@+id/labelaImePrezime"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="16dp"
        android:text="Ime i prezime" />
    <EditText
        android:id="@+id/inputIme"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="textPersonName" />
    <TextView
        android:id="@+id/labelaAdresa"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="16dp"
        android:text="Vasa adresa" />
    <EditText
        android:id="@+id/inputAdresa"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="textPostalAddress" />
    <TextView
        android:id="@+id/labelaTelefon"
        android:layout_width="match_parent"
```

```

        android:layout_height="wrap_content"
        android:textSize="16dp"
        android:text="Vas broj telefona" />
<EditText
        android:id="@+id/inputTelefon"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="text|phone" />
<Button
        android:id="@+id/buttonSacuvajPodatke"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Sacuvaj podatke" />
<Button
        android:id="@+id/buttonProcitajPodatke"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Procitaj podatke" />
</LinearLayout>

```

Odgovarajuća aktivnost koja koristi SharedPreferences za čuvanje podataka u onStop(), odnosno čitanje u onStart(), kao i upis i čitanje klikom na odgovarajuće dugme, data je u klasi MainActivity.java, čiji je sadržaj dat u nastavku:

```

package com.example.student.sharedpreferences;

import android.content.SharedPreferences;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;

public class MainActivity extends AppCompatActivity
implements View.OnClickListener {

    //jedinstveno na nivou aktivnosti
    private final static String SHARED_PREFERENCES_PREFIX =
"MainActivitySharedPreferencesPrefix";
    private final static String SHARED_PREFERENCES_KEY_IME
= "ime";
    private final static String
SHARED_PREFERENCES_KEY_ADRESA = "adresa";
    private final static String
SHARED_PREFERENCES_KEY_TELEFON = "telefon";

    private EditText inputIme;
    private EditText inputAdresa;

```

```

private EditText inputTelefon;
private Button buttonSacuvajPodatke;
private Button buttonProcitajPodatke;

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

    initComponents();
}

private void initComponents() {
    inputIme = findViewById(R.id.inputIme);
    inputAdresa = findViewById(R.id.inputAdresa);
    inputTelefon = findViewById(R.id.inputTelefon);
    buttonSacuvajPodatke =
findViewById(R.id.buttonSacuvajPodatke);
    buttonProcitajPodatke =
findViewById(R.id.buttonProcitajPodatke);

    buttonProcitajPodatke.setOnClickListener(this);
    buttonSacuvajPodatke.setOnClickListener(this);
}

//kada se aplikacija pokrene, zelimo da prikazemo
eventualno ranije unete podatke
@Override
protected void onStart() {
    super.onStart();
    procitajPodatke();
}

//kada se aplikacija stopira, zelimo da sacuvamo
eventualno unete podatke
@Override
protected void onStop() {
    super.onStop();
    sacuvajPodatke();
}

@Override
public void onClick(View view) {
    switch(view.getId()){
        case R.id.buttonSacuvajPodatke:
            sacuvajPodatke();
            break;
        case R.id.buttonProcitajPodatke:
            procitajPodatke();
    }
}

```

```

        break;
    }

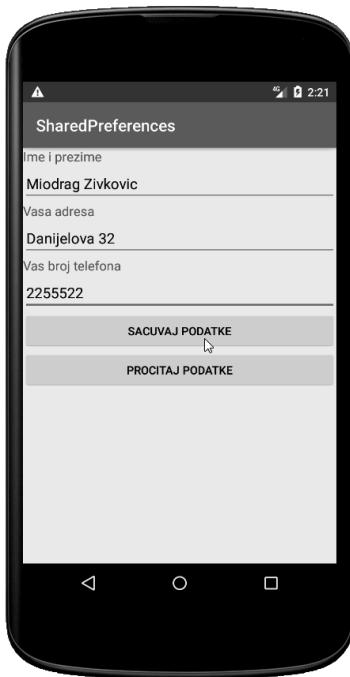
}

private void sacuvajPodatke() {
    String ime = inputIme.getText().toString();
    String adresa = inputAdresa.getText().toString();
    String telefon = inputTelefon.getText().toString();
    //dozvoljen je samo private mode
    Sharedpreferences sharedpreferences =
getSharedpreferences(SHARED_PREFERENCES_PREFIX, 0);
    Sharedpreferences.Editor editor =
sharedpreferences.edit();
    //cuvaju se parovi kljuc / vrednost
    editor.putString(SHARED_PREFERENCES_KEY_IME, ime);
    editor.putString(SHARED_PREFERENCES_KEY_ADRESA,
adresa);
    editor.putString(SHARED_PREFERENCES_KEY_TELEFON,
telefon);
    editor.commit(); //moze i editor.apply()
}

private void procitajPodatke() {
    Sharedpreferences sharedpreferences =
getSharedpreferences(SHARED_PREFERENCES_PREFIX, 0);
    String imePrezime =
sharedpreferences.getString(SHARED_PREFERENCES_KEY_IME,
"");
    String email =
sharedpreferences.getString(SHARED_PREFERENCES_KEY_ADRESA,
"");
    String telefon =
sharedpreferences.getString(SHARED_PREFERENCES_KEY_TELEFON,
"");
    //procitane vrednosti postavljamo u polja za unos
imena i adrese
    inputIme.setText(imePrezime);
    inputAdresa.setText(email);
    inputTelefon.setText(telefon);
}
}

```

Prikaz rada aplikacije dat je na slici 9.2. Klikom na dugme sačuvaj podatke, sve unete vrednosti se uzimaju i čuvaju u Sharedpreferences. Klikom na dugme pročitaj, sve sačuvane vrednosti se čitaju iz Sharedpreferences i postavljaju nazad u polja za unos.



Slika 9.2, Korisnički interfejs aplikacije za čuvanje podataka

Primer 2: Potrebno je razviti identičnu aplikaciju kao u primeru 1, sa razlikom da se za čuvanje unetih vrednosti koristi interni prostor.

Rešenje: Korisnički interfejs je identičan kao u primeru 1, i prikazan na slici 9.2. Jedino što se razlikuje je kod klase MainActivity.java (konkretno, sadržaj metoda sacuvajPodatke() i procitajPodatke()). Kod ove izmenjene aktivnosti je dat u nastavku:

```
package com.example.student.interniprostor;

import android.content.Context;
import android.content.SharedPreferences;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStreamReader;
import java.io.PrintWriter;
```

```

public class MainActivity extends AppCompatActivity
implements View.OnClickListener {

    //jedinstveno na nivou aktivnosti
    private final static String INTERNAL_APP_DATA_PREFIX =
"MainActivityInternalData";

    private EditText inputIme;
    private EditText inputAdresa;
    private EditText inputTelefon;
    private Button buttonSacuvajPodatke;
    private Button buttonProcitajPodatke;

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

        initComponents();
    }

    private void initComponents() {
        inputIme = findViewById(R.id.inputIme);
        inputAdresa = findViewById(R.id.inputAdresa);
        inputTelefon = findViewById(R.id.inputTelefon);
        buttonSacuvajPodatke =
findViewById(R.id.buttonSacuvajPodatke);
        buttonProcitajPodatke =
findViewById(R.id.buttonProcitajPodatke);

        buttonProcitajPodatke.setOnClickListener(this);
        buttonSacuvajPodatke.setOnClickListener(this);
    }

    //kada se aplikacija pokrene, zelimo da prikazemo
    eventualno ranije unete podatke
    @Override
    protected void onStart() {
        super.onStart();
        procitajPodatke();
    }

    //kada se aplikacija stopira, zelimo da sacuvamo
    eventualno unete podatke
    @Override
    protected void onStop() {
        super.onStop();
    }
}

```

```

        sacuvajPodatke();
    }

    @Override
    public void onClick(View view) {
        switch (view.getId()) {
            case R.id.buttonSacuvajPodatke:
                sacuvajPodatke();
                break;
            case R.id.buttonProcitajPodatke:
                procitajPodatke();
                break;
        }
    }

    private void sacuvajPodatke() {
        String ime = inputIme.getText().toString();
        String adresa = inputAdresa.getText().toString();
        String telefon = inputTelefon.getText().toString();

        //interna fajlovi su dostupni samo ovoj aplikaciji
        //mod ponovo mora da bude private
        //za upis se koristi metoda openFileOutput
        try {
            FileOutputStream fos =
openFileOutput(INTERNAL_APP_DATA_PREFIX,
Context.MODE_PRIVATE);
            PrintWriter pw = new PrintWriter(fos);
            pw.println(ime);
            pw.println(adresa);
            pw.println(telefon);
            pw.flush();
            pw.close();
            fos.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    //za citanje se koristi metoda openFileInput
    private void procitajPodatke() {
        try{
            //prvi put svakako mora doci do greske
            FileInputStream fis =
openFileInput(INTERNAL_APP_DATA_PREFIX);
            BufferedReader br = new BufferedReader(new
InputStreamReader(fis));

```

```
        String ime = br.readLine();
        inputIme.setText(ime);
        String adresa = br.readLine();
        inputAdresa.setText(adresa);
        String telefon = br.readLine();
        inputTelefon.setText(telefon);
        br.close();
        fis.close();

    } catch (Exception e) {
        //ignorisacemo problem u slucaju prvog
pokretanja aplikacije
    }
}
```

Primer 3: Potrebno je razviti identičnu aplikaciju kao u primeru 1, sa razlikom da se za čuvanje unetih vrednosti koristi eksterni prostor.

Rešenje: Korisnički interfejs je identičan kao u primeru 1, i prikazan na slici 9.2. Jedino što se razlikuje je kod klase MainActivity.java (konkretno, sadržaj metoda sacuvajPodatke() i procitajPodatke()), kao i dozvole za čitanje i upis u eksterni prostor, koji se moraju dodati u manifestu aplikacije. AndroidManifest.xml je dat sledećim kodom:

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

    <uses-permission
        android:name="android.permission.READ_EXTERNAL_STORAGE">
    </uses-permission>
    <uses-permission
        android:name="android.permission.WRITE_EXTERNAL_STORAGE">
    </uses-permission>

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

```

        <action
    android:name="android.intent.action.MAIN" />

        <category
    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Kod izmenjene aktivnosti je dat u nastavku:

```

package com.example.student.eksterniprostor;

import android.content.Context;
import android.content.SharedPreferences;
import android.os.Environment;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStreamReader;
import java.io.PrintWriter;

public class MainActivity extends AppCompatActivity
implements View.OnClickListener {

    private final static String EXTERNAL_APP_DATA_PREFIX =
"MainActivityInternalData";
    private final static String APP_DATA_FILE_NAME =
"/data.txt";

    private EditText inputIme;
    private EditText inputAdresa;
    private EditText inputTelefon;
    private Button buttonSacuvajPodatke;
    private Button buttonProcitajPodatke;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```

        initComponents();
    }

    private void initComponents() {
        inputIme = findViewById(R.id.inputIme);
        inputAdresa = findViewById(R.id.inputAdresa);
        inputTelefon = findViewById(R.id.inputTelefon);
        buttonSacuvajPodatke =
findViewById(R.id.buttonSacuvajPodatke);
        buttonProcitajPodatke =
findViewById(R.id.buttonProcitajPodatke);

        buttonProcitajPodatke.setOnClickListener(this);
        buttonSacuvajPodatke.setOnClickListener(this);
    }

    //kada se aplikacija pokrene, zelimo da prikazemo
    eventualno ranije unete podatke
    @Override
    protected void onStart() {
        super.onStart();
        procitajPodatke();
    }

    //kada se aplikacija stopira, zelimo da sacuvamo
    eventualno unete podatke
    @Override
    protected void onStop() {
        super.onStop();
        sacuvajPodatke();
    }

    @Override
    public void onClick(View view) {
        switch(view.getId()){
            case R.id.buttonSacuvajPodatke:
                sacuvajPodatke();
                break;
            case R.id.buttonProcitajPodatke:
                procitajPodatke();
                break;
        }
    }

    private void sacuvajPodatke(){
        String ime = inputIme.getText().toString();
        String adresa = inputAdresa.getText().toString();
    }
}

```

```

String telefon = inputTelefon.getText().toString();

//kada je u pitanju sd kartica, ne mozemo uvek
znati putanju (mount, storage... )
//ne mozemo znati ni da li je dostupna
// oslanjamo se na environmental variable
//radimo sa direktorijumom koji je maticni za nasu
aplikaciju u okviru javnog direktorijuma za dokumenta
//za novije androide, nakon 4.4
//File dir = new File
(Environment.getExternalStoragePublicDirectory(Environment.
DIRECTORY_DOCUMENTS), EXTERNAL_APP_DATA_PREFIX);

//pravimo poddirektorijum
File dir = new File
(Environment.getExternalStorageDirectory() + "/Documents",
EXTERNAL_APP_DATA_PREFIX); //pravimo poddirektorijum
//ova putanja verovatno ne postoji
dir.mkdirs();
//sada direktorijum postoji, pravimo konkretan fajl
File file = new File(dir.getAbsolutePath() +
APP_DATA_FILE_NAME); //ovo moze da ide kao konstanta
try {
    PrintWriter pw = new PrintWriter(file);
    pw.println(ime);
    pw.println(adresa);
    pw.println(telefon);
    pw.flush();
    pw.close();
} catch (Exception e) {
    //...
}
}

//za citanje se koristi metoda openFileInput
private void procitajPodatke() {
    File dir = new File
(Environment.getExternalStorageDirectory() + "/Documents",
EXTERNAL_APP_DATA_PREFIX);
    File file = new File(dir.getAbsolutePath() +
"/podaci-korisnika.txt");
    try {
        BufferedReader br = new BufferedReader(new
InputStreamReader(new FileInputStream(file)));
        String ime = br.readLine();
        inputIme.setText(ime);
    }
}

```

```
        String adresa = br.readLine();
        inputAdresa.setText(adresa);
        String telefon = br.readLine();
        inputTelefon.setText(telefon);
        br.close();
    }catch (Exception e) {
    //ignorisacemo problem u sluazu prvog
pokretanja aplikacije
}
}
```

Primer 4: Kreirati jednostavnu SQLite bazu podataka za čuvanje beleški. Tabela u koju se beleške čuvaju se zove notes, i ima kolone id, title i content. Obezbediti osnovnu CRUD funkcionalnost (Create, Read, Update i Delete). U aktivnosti jednostavno demonstrirati rad baze tako što se prvo unosi nekoliko redova u bazu, a zatim čitaju svi redovi i prikazuju na korisničkom interfejsu.

Rešenje: Prilikom kreiranja SQLite baze podataka, prvo se pristupa kreiranju šeme i ugovora baze, gde su definisane sve tabele baze. Sadržaj klase NotesDBContract.java je dat sledećim kodom:

```
package com.example.student.sqlitenotes;

import android.provider.BaseColumns;

public final class NotesDBContract {

    //sprecavanje slucajnjoginstanciranja ove klase
    private NotesDBContract() {}

    /* Unutrasnja klasa koja definise sadrzaj tabele */
    public static class NoteEntry implements BaseColumns {
        public static final String TABLE_NAME = "notes";
        public static final String COLUMN_TITLE = "title";
        public static final String COLUMN_CONTENT =
"content";
    }
}
```

Nakon toga, pristupa se definisanju modela podatka, koji je implementiran klasom Note.java, i odgovara jednom redu u bazi podataka. Ova klasa sadrži samo getere i `toString()` metodu za lakši ispis.

```

package com.example.student.sqlitenotes;

public class Note {
    //model za jedan red u bazi
    private int noteId;
    private String title, content;

    public Note(int noteId, String title, String content) {
        this.noteId = noteId;
        this.title = title;
        this.content = content;
    }

    public int getNoteId() {
        return noteId;
    }

    public String getTitle() {
        return title;
    }

    public String getContent() {
        return content;
    }

    @Override
    public String toString() {
        return "Note{" +
            "noteId=" + noteId +
            ", title='" + title + '\'' +
            ", content='" + content + '\'' +
            '}';
    }
}

```

Implementacija same baze podataka data je klasom NotesDatabase.java. U ovoj klasi su implementirane sve funkcionalnosti potrebne za CRUD, dakle, čitanje jedne beleške, čitanje svih beleški, unos beleške, brisanje beleške i ažuriranje beleške.

```

package com.example.student.sqlitenotes;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.provider.BaseColumns;

```

```

import java.util.ArrayList;
import java.util.List;

public class NotesDatabase extends SQLiteOpenHelper {

    private static final String SQL_CREATE_NOTES =
            "CREATE TABLE " +
    NotesDBContract.NoteEntry.TABLE_NAME + " (" +
            NotesDBContract.NoteEntry._ID + " "
    INTEGER PRIMARY KEY," +
                    NotesDBContract.NoteEntry.COLUMN_TITLE
+ " TEXT," +
NotesDBContract.NoteEntry.COLUMN_CONTENT + " TEXT" + ")";

    private static final String SQL_DELETE_NOTES =
            "DROP TABLE IF EXISTS " +
    NotesDBContract.NoteEntry.TABLE_NAME;

    // Prilikom promene seme, mora se inkrementirati
verzija
    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "notes.db";

    public NotesDatabase (Context context) {
        super(context, DATABASE_NAME, null,
DATABASE_VERSION);
    }
    @Override
    public void onCreate(SQLiteDatabase sqLiteDatabase) {
        sqLiteDatabase.execSQL(SQL_CREATE_NOTES);
    }

    @Override
    public void onUpgrade(SQLiteDatabase sqLiteDatabase,
int oldVersion, int newVersion) {
        sqLiteDatabase.execSQL(SQL_DELETE_NOTES);
        onCreate(sqLiteDatabase);
    }

    public void addNote (String title, String content){
        //id ne treba, automatski ce se dohvati
        //moramo dohvatiti bazu sa dozvolom za upis sadzaja
        SQLiteDatabase db = this.getWritableDatabase();
        //da bismo dodali nesto u bazu, potreban nam je set
vrednosti - ContentValues
        //imena kolona se koriste kao kljucevi
    }
}

```

```

        ContentValues cv = new ContentValues(); //sada isto
        kao bundle
        cv.put(NotesDBContract.NoteEntry.COLUMN_TITLE,
        title);
        cv.put(NotesDBContract.NoteEntry.COLUMN_CONTENT,
        content);

        db.insert(NotesDBContract.NoteEntry.TABLE_NAME,
        null, cv);

    }

public Note readNote(String findTitle){
    SQLiteDatabase db = this.getReadableDatabase();
    // Projekcija koja specificira koje kolone iz baze
    // su potrebne.
    String[] projection = {
        BaseColumns._ID,
        NotesDBContract.NoteEntry.COLUMN_TITLE,
        NotesDBContract.NoteEntry.COLUMN_CONTENT,
    };
    // Filtriranje rezultata WHERE "title" =
    'findTitle'
    String selection =
NotesDBContract.NoteEntry.COLUMN_TITLE + " = ?";
    String[] selectionArgs = { findTitle };
    // Sortiranje rezultata u rezultujućem Cursor
objektu
    String sortOrder =
                NotesDBContract.NoteEntry.COLUMN_TITLE + " "
DESC";
    Cursor cursor = db.query(
                NotesDBContract.NoteEntry.TABLE_NAME, ///
tabela
                projection, // niz kolona koje
se vracaju (null vraca sve)
                selection, // kolone za WHERE
klauzulu
                selectionArgs, // vrednostu WHERE
klauzule
                null, // ne grupisi
redove
                null, // ne filtriraj
dodatno redove po grupama
                sortOrder // sortiranje
    );
    Note note = null;
}

```

```

    if(cursor.moveToFirst()) {
        int id = cursor.getInt(
            cursor.getColumnIndex(BaseColumns._ID));
        String title = cursor.getString(
            cursor.
        getColumnIndex(NotesDBContract.NoteEntry.COLUMN_TITLE));
        String content = cursor.getString(
        cursor.
        getColumnIndex(NotesDBContract.NoteEntry.COLUMN_CONTENT));
        note = new Note(id, title, content);
    }
    cursor.close();
    return note;
}

public List<Note> readAllNotes() {
    SQLiteDatabase db = this.getReadableDatabase();
    // Projekcija koja specificira koje kolone iz baze
    // su potrebne.
    String[] projection = {
        BaseColumns._ID,
        NotesDBContract.NoteEntry.COLUMN_TITLE,
        NotesDBContract.NoteEntry.COLUMN_CONTENT,
    };

    Cursor cursor = db.query(
        NotesDBContract.NoteEntry.TABLE_NAME,           // 
    tabela
        projection,                                // niz kolona koje
    se vracaju (null vraca sve)
        null,                                     // kolone za WHERE
    klauzulu
        null,                                     // vrednosti WHERE klauzule
        null,                                     // ne grupisi
    redove
        null,                                     // ne filtriraj
    dodatno redove po grupama
        null                                       // sortiranje
    );
}

List<Note> notes = new ArrayList<>();
cursor.moveToFirst();
while(!cursor.isAfterLast()) {
    int id = cursor.getInt(
        cursor.getColumnIndex(BaseColumns._ID));
    String title = cursor.getString(

```

```

        cursor.
getColumnIndex(NotesDBContract.NoteEntry.COLUMN_TITLE));
        String content = cursor.getString(
cursor.
getColumnIndex(NotesDBContract.NoteEntry.COLUMN_CONTENT));
        notes.add( new Note(id, title, content));
        cursor.moveToNext();
    }
    cursor.close();
    return notes;
}

public int deleteNote(String title){
    SQLiteDatabase db = this.getWritableDatabase();
    // definisati 'where' klauzulu.
    String selection =
NotesDBContract.NoteEntry.COLUMN_TITLE + " LIKE ?";
    // specificirati argumete za where klauzulu.
    String[] selectionArgs = { title };
    // ivrsiti SQL naredbu.
    int deletedRows =
db.delete(NotesDBContract.NoteEntry.TABLE_NAME, selection,
selectionArgs);
    return deletedRows;
}

public int updateNote(String title, String newContent){
    SQLiteDatabase db = this.getWritableDatabase();
    // Nova vrednost za jednu kolonu
    ContentValues values = new ContentValues();

values.put(NotesDBContract.NoteEntry.COLUMN_CONTENT,
newContent);
    // Red koji se azurira se pronalazi po imenu
beleske
    String selection =
NotesDBContract.NoteEntry.COLUMN_TITLE + " LIKE ?";
    String[] selectionArgs = { title };
    int count = db.update(
        NotesDBContract.NoteEntry.TABLE_NAME,
        values,
        selection,
        selectionArgs);
    return count;
}
}

```

Korisnički interfejs u ovom primeru je jednostavan, i sadrži samo jedno tekstualno polje u kome se ispisuju rezultati. Sadržaj activity_main.xml fajla dat je sa:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
    android:layout_width="match_parent"
        android:layout_height="match_parent">

    <TextView
        android:id="@+id/labelPrikaz"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="15dp"
        android:lines="20" />
</LinearLayout>
```

Aktivnost je takođe jednostavna, rad sa bazom se demonstrira kroz dodavanje nekoliko unosa, a zatim prikaz svih redova iz baze podataka u tekstualnom polju na korisničkom interfejsu. Sadržaj MainActivity.java klase:

```
package com.example.student.sqlitenotes;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.TextView;

import java.util.Arrays;
import java.util.List;

public class MainActivity extends AppCompatActivity {

    private TextView labelPrikaz;

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

    private void initComponents() {
        labelPrikaz = findViewById(R.id.labelPrikaz);

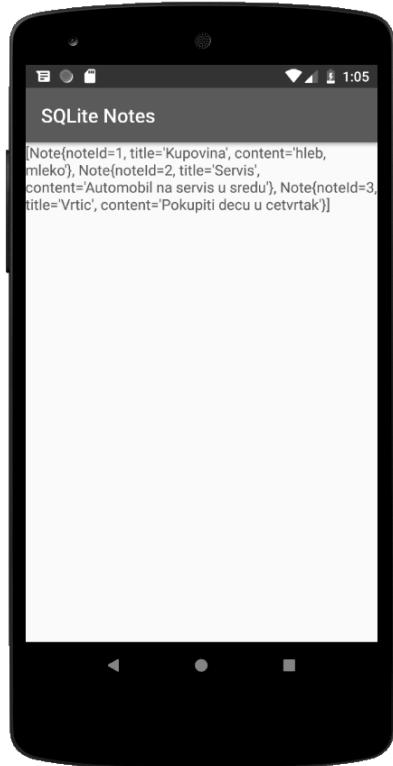
        NotesDatabase db = new NotesDatabase(this);
```

```
        db.addNote("Kupovina", "hleb, mleko");
        db.addNote("Servis", "Automobil na servis u
sredu");
        db.addNote("Vrtic", "Pokupiti decu u cetvrtak");

    List<Note> lista = db.readAllNotes();
    String listaString =
Arrays.toString(lista.toArray());

    labelPrikaz.setText(listaString);
}
}
```

Nakon pokretanja aplikacije, svi uneti redovi će biti izlistani kao što je prikazano na slici 9.3. Kreirana SQLite baza je potpuno funkcionalna, a čitalac kao vežbu može implementirati korisnički interfejs koji će omogućiti detaljan rad sa ovom bazom.



Slika 9.3, Prikaz svih unetih redova iz Notes SQLite baze podataka

10. BroadcastReceiver komponente

Android aplikacije mogu da šalju ili primaju obaveštenja (engl. *broadcast* poruke) od Android sistema ili od drugih aplikacija, na veoma sličan način kako funkcioniše publisher-subscriber Java šablon. Komponenta BroadcastReceiver je jedna od četiri osnovne komponente Android aplikacija, koja je zadužena za prijem ovih obaveštenja. Nakon što aplikacija primi obaveštenje, može po potrebi reagovati – na primer, ukoliko je primljeno obaveštenje da je nivo baterije nizak, aplikacija može smanjiti svoje aktivnosti.

Obaveštenja se šalju kada se desi neki događaj od interesa (ne samo za aplikaciju, već ponekad i za ceo Android sistem). Android sistem, na primer, šalje obaveštenja prilikom različitih sistemskih događaja, poput promene stanja baterije, prelaska ekrana u ugašeno stanje, kačenja uređaja na punjač, ulaska u režim za letenje i slično. Aplikacije takođe mogu da šalju obaveštenja kako bi obavestila druge aplikacije da se desio neki događaj koji im može biti od interesa, na primer da su neki novi podaci preuzeti sa Interneta. Dakle, Android sistem je pun obaveštenja koja se redovno šalju i mogu bitiinicirana ili od sistema ili od samih aplikacija. Ko „čuje“ ova obaveštenja, odnosno kako neka aplikacija može da se „pretplati“ na neko specifično obaveštenje? Odgovor je jednostavan, aplikacija se mora registrovati kako bi dobijala specifična obaveštenja. Kada se neko obaveštenje pošalje, sistem automatski obaveštava sve aplikacije koje „slušaju“, odnosno koje su se registrovale da primaju taj određeni tip obaveštenja.

10.1. Sistemska obaveštenja

Android sistem šalje veliki broj obaveštenja zbog velikog broja različitih događaja koji se dešavaju u njemu, poput promene statusa baterije, ulaska u režim rada za letenje ili izlaska iz tog režima, gašenja ekrana i slično. Sistem šalje ova obaveštenja svim aplikacijama koje su se registrovale da primaju ta obaveštenja. Obaveštenje se pakuje u Intent objekat čiji atribut akcija označava događaj koji se desio (na primer, android.intent.action.AIRPLANE_MODE). Intent objekat može da sadrži i dodatne informacije zapakovane u Bundle objektu u polju extra. Na primer, režim rada za letenje dodatno sadrži boolean extra koji je indikacija da li je Airplane Mode uključen ili ne, dok prilikom promene statusa baterije Intent sadrži celobrojni extra koji predstavlja trenutni nivo baterije.

Sa razvojem Android platforme, periodično se menja lista obaveštenja koje sistem šalje, kao i samo ponašanje obaveštenja. Na primer, od Android verzije 7.0 (API nivo 24) sistem ne šalje obaveštenja ACTION_NEW_PICTURE ili ACTION_NEW_VIDEO. Takođe, ukoliko aplikacija želi da prima

CONNECTIVITY_ACTION obaveštenje mora se koristiti programska registracija prijemnika, pošto deklaracija u manifestu više nije podržana. Od Android verzije 8.0 (API nivo 26), sistem postavlja dodatna ograničenja na prijemnike deklarisane u manifestu. Za većinu sistemskih obaveštenja je potrebno koristiti programsku registraciju prijemnika.

10.2. Prijem obaveštenja

Aplikacija se mora registrovati na prijem tačno određenog tipa obaveštenja, ukoliko želi da u daljem radu dobija od Android sistema obaveštenja tog tipa. Postoje dva načina za registraciju aplikacije:

- Deklaracija prijemnika kroz manifest aplikacije.
- Dinamička programska registracija prijemnika.

Ranije je napomenuto da Android od API nivoa 26 ograničava deklaraciju prijemnika u manifestu aplikacije, odnosno nije moguće prijaviti se na obaveštenja koja ne ciljaju tačno samu aplikaciju. Programski registrovani prijemnici za sada nemaju dodatna ograničenja, što ne znači da ograničenja neće biti u budućnosti. U nastavku ovog poglavlja opisane su obe tehnike registracije prijemnika.

10.2.1. Deklaracija prijemnika kroz manifest

Prvi način registracije prijemnika je deklaracija u okviru manifesta aplikacije. Ako je prijemnik registrovan na ovakav način, sistem će pokrenuti aplikaciju prilikom slanja obaveštenja (ukoliko aplikacija nije već pokrenuta). Sam proces registracije prijemnika se u ovom slučaju vrši na sledeći način.

U okviru manifesta aplikacije potrebno je dodati `<receiver>` element. Prijemnici su komponente aplikacije poput aktivnosti, tako da `<receiver>` element stoji na istom nivou u hijerarhiji XML fajla kao i element `<activity>`, kao direktno dete `<application>` elementa. Primer deklaracije prijemnika koji se prijavljuje da prima obaveštenja o promeni statusa baterije dat je u nastavku:

```
<receiver android:name=".BatteryStateReceiver">
    android:exported="true">
        <intent-filter>
            <action
                android:name="android.intent.action.BATTERY_CHANGED"/>
        </intent-filter>
</receiver>
```

Ukoliko se struktura deklaracije pogleda detaljnije, može se uočiti da se aplikacija registruje na sistemske poruke specificirane akcijama u okviru intent filtera. Jedan prijemnik se može prijaviti i na više različitih akcija (poruka), na primer:

```
<receiver android:name=".MyReceiver">
    android:exported="true">
        <intent-filter>
            <action
                android:name="android.intent.action.BOOT_COMPLETED"/>
            <action
                android:name="android.intent.action.INPUT_METHOD_CHANGED"
            />
        </intent-filter>
</receiver>
```

Bitni atributi koje je moguće definisati u okviru deklaracije prijemnika u manifestu su:

- android:enabled – označava da li se prijemnik može instancirati od strane Android sistema. Podrazumevana vrednost je true.
- android:exported – označava da li prijemnik može da primi poruke od izvora van same aplikacije. Podrazumevana vrednost je true.
- android:name: označava ime klase koja implementira prijemnik (mora biti izvedena iz klase BroadcastReceiver).
- android:permission: označava dozvolu koju izvor poruke mora da ima kako bi poslao poruku ovom prijemniku.
- android:process: označava ime procesa u kome se prijemnik izvršava.

Nakon deklaracije prijemnika u okviru manifesta aplikacije, potrebno je implementirati ga kao klasu u kodu aplikacije. Ova klasa mora biti izvedena iz klase BroadcastReceiver i implementirati callback metodu onReceive(Context, Intent). Takođe, klasa mora biti nazvana na isti način kako je definisano u atributu android:name u okviru manifesta. Na primer, prijemnik koji prihvata obaveštenja o promenama stanja baterije bi mogao biti implementiran na sledeći način:

```
public class BatteryStateReceiver extends BroadcastReceiver {
    private final static String BATTERY_LEVEL = "level";
    @Override
    public void onReceive(Context context, Intent intent) {
        int nivo = intent.getIntExtra(BATTERY_LEVEL, 0);
        //sada se moze na primer prikazati stanje na ekranu
        ...
    }
}
```

U navedenom primeru, kada se stanje baterije promeni, Android sistem će pozvati metodu `onReceive()`, kojoj kao drugi parametar prosleđuje Intent objekat. U ovom Intent objektu se nalazi opisana akcija (kako bi bilo moguće prihvati istim prijemnikom više različitih obaveštenja i odgovoriti drugačije na svako obaveštenje na osnovu akcije) i Bundle extra sa dodatnim vrednostima od interesa. Prilikom promene nivoa baterije, u Bundle objektu će biti celobrojna vrednost pod ključem „level“, u kojoj se nalazi vrednost novog nivoa baterije. Naravno, sadržaj Bundle objekta se razlikuje u slučaju različitih obaveštenja.

Kada je prijemnik definisan u aplikaciji, Android sistem će ga pri instalaciji aplikacije registrovati. Prijemnik tada postaje odvojena tačka ulaska u aplikaciju, što znači da sistem može da pokrene aplikaciju kako bi joj prosledio obaveštenje. Što se tiče životnog veka samog objekta prijemnika, odnosno instance klase koja je izvedena iz klase `BroadcastReceiver`, sistem kreira novi objekat za svaku poruku koju je potrebno isporučiti. Ovaj objekat postoji samo za vreme poziva `onReceive()` metode, odnosno prestaje da bude aktivan čim se `onReceive()` metoda izvrši. Android sistem potom može ubiti proces u bilo kom trenutku kako bi oslobođio memoriju.

10.2.2. Programska registracija prijemnika

Drugi način registracije prijemnika je dinamička programska registracija. U tom slučaju, u kontekstu aktivnosti se mora ručno napraviti instanca klase `BroadcastReceiver`. Na primeru `BatteryStatusReceiver` klase, instanca se kreira na sledeći način:

```
BroadcastReceiver receiver = new BatteryStatusReceiver();
```

Nakon toga, potrebno je napraviti `IntentFilter` i dodati akcije koje označavaju na koja obaveštenja se prijemnik registruje. Registracija se zatim izvršava pozivom metode `registerReceiver(BroadcastReceiver, IntentFilter)`:

```
IntentFilter intentFilter = new  
IntentFilter(Intent.ACTION_BATTERY_CHANGED)  
this.registerReceiver(receiver, intentFilter);
```

Ovako definisan prijemnik prihvata obaveštenja sve dok je kontekst u kome je registrovan validan. Na primer, ukoliko je definisan u okviru aktivnosti, obaveštenja se prihvataju sve dok aktivnost ne bude uništena. Ukoliko se definiše u okviru konteksta aplikacije, obaveštenja se prihvataju sve dok se aplikacija izvršava. Kada je potrebno prestati sa prihvatanjem obaveštenja, može se pozvati metoda `unregisterReceiver()` kojom se prijemnik prosleđen kao parametar odjavljuje od daljeg prijema obaveštenja.

Ako se registracija prijemnika vrši u okviru aktivnosti, treba obratiti pažnju u kom trenutku se registruje, odnosno odjavljuje prijemnik. U slučaju da se prijemnik registruje u okviru `onCreate()` metode, trebalo bi ga odjaviti u `onDestroy()` metodi, kako ne bi procureo van konteksta same aktivnosti. Drugi način je registracija prijemnika u `onResume()` metodi, pri čemu se odjavljivanje radi u `onPause()` metodi. Na taj način se sprečava registracija prijemnika više puta i obezbeđuje da se obaveštenja ne primaju dok je aktivnost pauzirana. Analogno, ako se prijemnik registruje u `onStart()` metodi, treba ga odjaviti u `onStop()` metodi. U ovom slučaju prijemnik dobija obaveštenja i dok je aktivnost pauzirana. Primer implementacije prijemnika u okviru aktivnosti sa registracijom u `onStart()` metodi i odjavljivanjem u `onStop()` metodi dat je u nastavku:

```
public class MainActivity extends AppCompatActivity {

    ...
    private BroadcastReceiver mReceiver;

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

    private void initComponents() {
        ...
        mReceiver = new BatteryStatusReceiver();
    }

    @Override
    protected void onStart() {
        registerReceiver(mReceiver, new
IntentFilter(Intent.ACTION_BATTERY_CHANGED));
        super.onStart();
    }

    @Override
    protected void onStop() {
        unregisterReceiver(mReceiver);
        super.onStop();
    }

    ...
}

}
```

10.3. Slanje obaveštenja

Aplikacija nije ograničena samo da prima obaveštenja, već po potrebi može i da ih šalje. Postoje tri osnovna načina kako aplikacija može poslati obaveštenje:

- `sendOrderedBroadcast(Intent, String)` – ova metoda šalje obaveštenje prijemnicima serijski, odnosno jednom po jednom. Svaki prijemnik u tom slučaju prima obaveštenje, može da doda međurezultat u obaveštenje i da ga propagira sledećem prijemniku, ili može da otkaže dalje slanje obaveštenja.
- `sendBroadcast(Intent)` – ova metoda šalje obaveštenje svim prijemnicima u neodređenom redosledu. Ovo je standardan način slanja obaveštenja, poznat i pod nazivom „normalno obaveštenje“ (engl. *Normal broadcast*). Ovakvo slanje obaveštenja je efikasnije, ali prijemnici ne mogu dohvati međurezultate drugih prijemnika, niti da propagiraju, odnosno otkažu dalje slanje obaveštenja.
- `LocalBroadcastManager.sendBroadcast()` – metoda služi za slanje obaveštenja prijemnicima koji se nalaze u istoj aplikaciji u kojoj se nalazi i pošiljalac obaveštenja. Ovaj oblik slanja obaveštenja je najefikasniji (naravno, upotrebljiv samo u slučaju da su i prijemnik i pošiljalac deo iste aplikacije, odnosno procesa).

Najčešći oblik upotrebe predstavlja normalno obaveštenje, koje se svodi na kreiranje Intent objekta, postavljanje akcije i eventualno dodatnih podataka kroz Bundle extra, i slanje obaveštenja pozivom metode `sendBroadcast(Intent)`. Primer implementacije dat je sledećim delom Java koda:

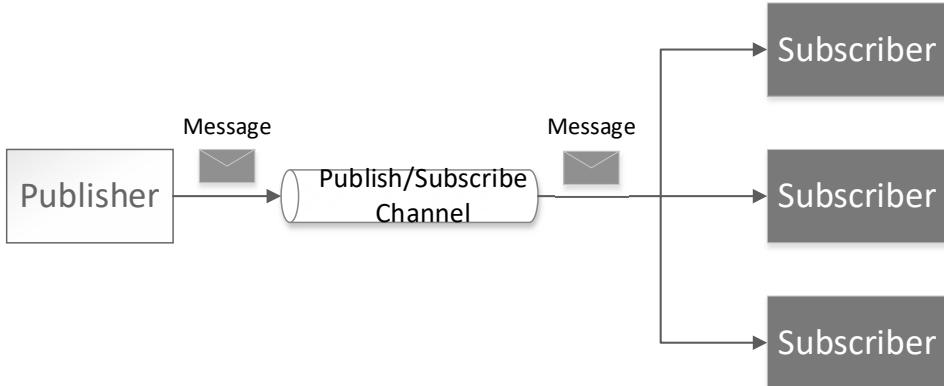
```
Intent intent = new Intent();
intent.setAction("com.example.obavestenja.MOJA_PORUKA");
intent.putExtra("poruka", "Zdravo!");
sendBroadcast(intent);
```

Akcija pod kojom se šalje obaveštenje može biti proizvoljno definisana od strane programera, ali treba da bude jedinstvena (zbog čega se često uključuje i celo ime paketa kao deo imena akcije). Da bi prijemnik mogao da prima ova obaveštenja, mora se registrovati na prijem obaveštenja tipa `"com.example.obavestenja.MOJA_PORUKA"`. Kada prijemnik prihvati obaveštenje, može uzeti poruku iz Intent objekta preko ključa „`poruka`“ pod kojim je string stavljen u Intent extras.

10.4. Sigurnosna razmatranja i praktične primene

Prijemnici su vrlo specifične komponente, pošto se mogu koristiti za razmenu obaveštenja i komunikaciju između različitih aplikacija, čime se otvara potencijalni prostor za zloupotrebe. Zbog toga treba biti posebno oprezan u radu sa ovim tipom komponenti. Kako bi se osigurao bezbedan rad aplikacije, kao i celog Android sistema, postoji nekoliko opštih smernica kojih bi se trebalo pridržavati prilikom razvoja aplikacije koja podržava slanje i primanje poruka.

Najpre je potrebno razmotriti da li aplikacija šalje obaveštenja samo drugim komponentama te iste aplikacije. Ukoliko nijedno obaveštenje ne ide nekom prijemniku van same aplikacije, preporuka je da se koristi LocalBroadcastReceiver. Upotreba ovakvog prijemnika se smatra efikasnijom i sigurnijom, čime se programer oslobođa brige o potencijalnim sigurnosnim problemima (zato što nema interprocesne komunikacije). Pošto su česte situacije kada aplikacija treba da funkcioniše po principu publisher-subscriber šablonu kao modela komunikacije između komponenti, preporučena implementacija u obliku lokalnih poruka predstavlja rešenje koje ne opterećuje kompletan Android sistem nepotrebnim porukama. Model publisher-subscriber šablonu prikazan je na slici 10.1. Ovaj šablon je primer mehanizma za razmenu poruka, u kome publisher komponente objavljaju poruke koje se šalju registrovanim subscriber komponentama. Na taj način dobija se sistem događaja i notifikacija, naročito koristan u slučaju distribuiranih aplikacija, ili aplikacija koje su izdeljene u manje komponente (kakva je obično struktura Android aplikacije).



Slika 10.1, publisher-subscriber šablon

Još jedan aspekt koji je neophodno razmotriti prestavlja slučaj kada je veliki broj aplikacija registrovan na prijem istog tipa obaveštenja kroz manifest aplikacije. To može dovesti do ekstremne situacije, kada sistem pokreće veliki broj aplikacija u isto vreme, što ima značajan negativan uticaj na performanse sistema,

kao i loše korisničko iskustvo. Preporuka Android zajednice stoga je da se prilikom implementacije preferira dinamička programska registracija prijemnika, kojom se ovakav slučaj izbegava.

Dodatne preporuke Android zajednice, naročito po pitanju sigurnosti, se mogu sumirati sledećom listom:

- Nije preporučljivo slati poruke putem implicitne namere, naročito u slučajevima da nose osetljive informacije. Ovako poslate poruke mogu pročitati sve aplikacije na uređaju koje su registrovane na taj tip poruka. Problem se rešava upotrebom lokalnih poruka ili ograničavanjem poruka putem dozvola.
- Nakon registracije prijemnika, aplikacija postaje ranjiva, pošto joj bilo koja druga aplikacija prisutna na uređaju može poslati potencijalno zlonamernu poruku. Ograničavanje poruka koje aplikacija prima se može postići upotrebom lokalnih poruka, specifikacijom dozvola ili postavljanjem android:exported atributa na false u manifestu aplikacije.
- Potrebno je biti svestan da je imenski prostor za poruke globalni na nivou sistema, odnosno imena akcija koja se koriste moraju biti jednoznačna kako ne bi došlo do kolizije sa drugim aplikacijama.

10.5. Pitanja za vežbu

- Šta je BroadcastReceiver komponenta, i čemu služi?
- Navesti nekoliko primera sistemskih obaveštenja.
- Koji su mogući načini implementacije BroadcastReceiver komponente?
- Da li je neophodno deklarisati BroadcastReceiver komponentu u manifestu aplikacije?
- Šta znači termin registracija prijemnika?
- Kako se prijemnik definiše kroz XML fajl?
- Koji su bitni atributi koje je moguće definisati kroz XML fajl?
- Objasniti čemu služe atributi enabled i exported. Objasniti razliku između ova dva atributa.
- Čemu služi atribut name, i da li je to obavezan atribut?
- Kako se prijemnik definiše dinamički, odnosno programski?
- U objekat kog tipa se pakuje obaveštenje?
- Šta označava akcija u kontekstu BroadcastReceiver komponenti?
- Da li obaveštenje može sa sobom nositi i dodatne informacije? Ukoliko da, kako se prosleđuju te informacije?
- Da li je dozvoljeno da dva i više prijemnika imaju deklarisanu istu akciju u svojim manifest fajlovima?
- Da li je prilikom implementacije prijemnika potrebno proširiti neku postojeću Android klasu, i ukoliko da, koju?
- Čemu služi onReceive() metoda?
- Da li novije verzije Android sistema uvode neka ograničenja u upotrebi BroadcastReceiver komponenti?
- Čemu služi LocalBroadcastManager?
- Koji java šablon je primjenjen u dizajnu BroadcastReceiver komponenti?
- Koji su sigurnosni problemi do kojih može doći kada se koriste BroadcastReceiver komponente?
- Koja je prednost lokalnih obaveštenja?
- Koji su mogući načini da aplikacija sama pošalje obaveštenje? Prodiskutovati sve načine i uočiti prednosti i mane svakog.
- Koji je najefikasniji način slanja obaveštenja?

10.6. Zadaci za vežbu

Primer 1: Napraviti BroadcastReceiver komponentu koja će pratiti sistemska obaveštenja o nivou baterije (akcija na koju je potrebno da se prijemnik prijavi je Intent.ACTION_BATTERY_CHANGED). Prijavu prijemnika na obaveštenje realizovati dinamički - programski. Prikazati u realnom vremenu promene u nivou baterije kroz tekstualni prikaz i progres bar u okviru korisničkog interfejsa aplikacije (čim stigne obaveštenje o novom nivou baterije, uzeti iz obaveštenja novu vrednost i ažurirati korisnički interfejs).

Rešenje: Traženi korisnički interfejs dat je sadržajem activity_main.xml fajla:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
    android:layout_width="match_parent"
        android:layout_height="match_parent">
    <TextView
        android:id="@+id/labelNivoBaterije"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="20dp"
        android:text="Nivo baterije" />
    <ProgressBar
        android:id="@+id/progressBarBaterija"
        style="?android:attr/progressBarStyleHorizontal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
</LinearLayout>
```

U okviru aktivnosti potrebno je držati referencu na objekat tipa BroadcastReceiver, koji se registruje u okviru onStart() metode, a odjavljuje u okviru onStop() metode. Sadržaj MainActivity.java dat je sledećim Java kodom:

```
package com.example.mzivkovic.myapplication;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.ProgressBar;
import android.widget.TextView;
```

```

public class MainActivity extends AppCompatActivity {

    private TextView nivoBaterijeTekst;
    private ProgressBar nivoBaterijeProgres;
    private BroadcastReceiver receiver;

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

    private void initComponents() {
        nivoBaterijeTekst =
findViewById(R.id.labelNivoBaterije);
        nivoBaterijeProgres =
findViewByIcon(R.id.progressBarBaterija);
        receiver = new BatteryBroadcastReceiver();
    }

    @Override
    protected void onStart() {
        registerReceiver(receiver, new
IntentFilter(Intent.ACTION_BATTERY_CHANGED));
        super.onStart();
    }

    @Override
    protected void onStop() {
        unregisterReceiver(receiver);
        super.onStop();
    }

    private class BatteryBroadcastReceiver extends
BroadcastReceiver {
        private final static String BATTERY_LEVEL =
"level";
        @Override
        public void onReceive(Context context, Intent
intent) {
            int nivo = intent.getIntExtra(BATTERY_LEVEL, 0);
            nivoBaterijeTekst.setText("Nivo baterije" + " "
+ nivo);
            nivoBaterijeProgres.setProgress(nivo);
        }
    }
}

```

Sadržaj AndroidManifest.xml fajla dat je sa:

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

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

                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Nakon pokretanja, aplikacija će dobijati obaveštenja o svakoj promeni nivoa baterije, koja će odmah po prijemu obaveštenja biti prikazana i ažuriranjem tekstualnog opisa i progres bara u korisničkom interfejsu, kao što je prikazano na slici 10.2.



Slika 10.2, prikaz rada BroadcastReceiver aplikacije koja prati obaveštenja o promenama nivoa baterije

Primer 2: Način prikazan u prethodnom primeru je funkcionalan, ali nije optimalan. Uopšteno govoreći, konstantno praćenje nivoa baterije ima veći negativni uticaj na bateriju (ironično) u odnosu na normalno ponašanje aplikacije. Sa stanovišta štednje energije, dobra praksa je da se prate samo značajne promene u nivou baterije, naročito slučaj kada uređaj ulazi ili izlazi iz stanja veoma niskog nivoa baterije (Low Battery, uređaj obično i sam daje ovo upozorenje korisniku). Još jedan događaj od značaja je prepoznavanje da li se uređaj trenutno puni ili ne. Potrebno je da se prethodna aplikacija preradi tako da prima obaveštenja samo za navedene događaje. BroadcastReceiver podesiti tako da reaguje na svaki događaj pojedinačno, kako bi se po potrebi moglo reagovati (na primer, smanjiti mrežne aktivnosti ukoliko je nivo baterije veoma nizak, povećati ih kada uređaj izade iz tog stanja i slično).

Rešenje: Korisnički interfejs aplikacije je prerađen tako da pokazuje samo status baterije (nizak nivo ili ok nivo), kao i da li se uređaj trenutno puni ili ne. Kod je dat u activity_main.xml fajlu:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
    android:layout_width="match_parent"
        android:layout_height="match_parent">
    <TextView
        android:id="@+id/labelStatusBaterije"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="20dp"
        android:text="Status baterije: " />
    <TextView
        android:id="@+id/labelPunjenje"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="20dp"
        android:text="Status punjenja: " />
</LinearLayout>
```

U MainActivity.java klasi, prijemnik se registruje na sva četiri obaveštenja. U implementaciji prijemnika (klasa BatteryBroadcastReceiver) se iz primljenog Intent objekta vadi akcija, na osnovu koje se odlučuje koji se događaj desio (switch naredba) i u okviru pojedinačnih case klauzula implementira odgovarajuće ponašanje sistema. U primeru, ažuriraju se statusne informacije na korisničkom interfejsu aplikacije. Implementacija ove aktivnosti data je u nastavku:

```
package com.example.mzivkovic.myapplication;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.BatteryManager;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.ProgressBar;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    private TextView statusBaterije;
    private TextView punjenjeBaterije;
    private BroadcastReceiver receiver;
```

```

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

    initComponents();
}

private void initComponents() {
    statusBaterije =
findViewById(R.id.labelStatusBaterije);
    punjenjeBaterije =
findViewById(R.id.labelPunjenje);
    receiver = new BatteryBroadcastReceiver();
}

@Override
protected void onStart() {
    IntentFilter intentFilter = new IntentFilter();
    //ulazak u stanje nizak nivo baterije
    intentFilter.addAction(Intent.ACTION_BATTERY_LOW);
    //izlazak iz stanja nizak nivo baterije
    intentFilter.addAction(Intent.ACTION_BATTERY_OKAY);
    //detektovanje da li je telefon nakacen na punjac

    intentFilter.addAction(Intent.ACTION_POWER_CONNECTED);
    //detektovanje skidanja uredjaja sa punjaca

    intentFilter.addAction(Intent.ACTION_POWER_DISCONNECTED);
    registerReceiver(receiver, intentFilter);
    super.onStart();
}

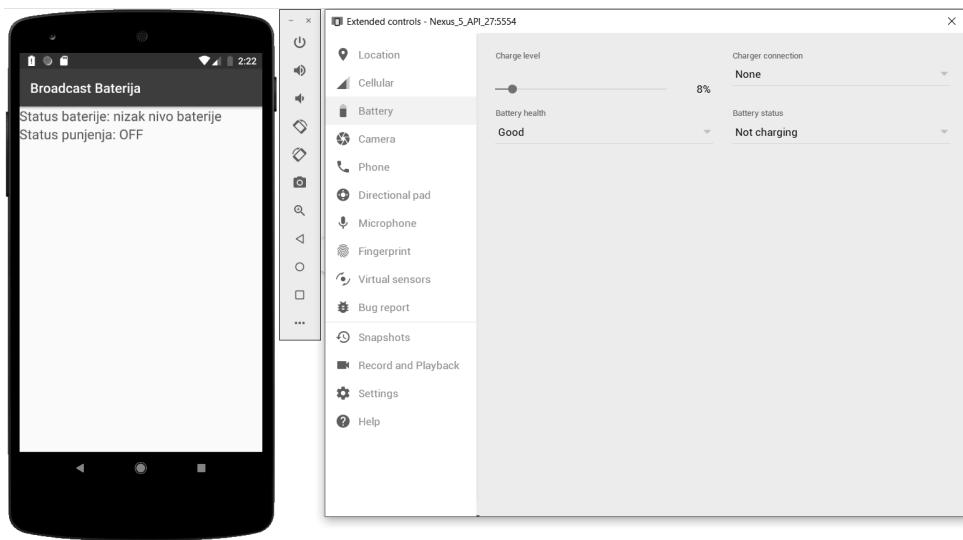
@Override
protected void onStop() {
    unregisterReceiver(receiver);
    super.onStop();
}

private class BatteryBroadcastReceiver extends
BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent
intent) {
        switch(intent.getAction()){
            case Intent.ACTION_BATTERY_LOW:
                statusBaterije.setText("Status
baterije: " + "nizak nivo baterije");
    }
}

```

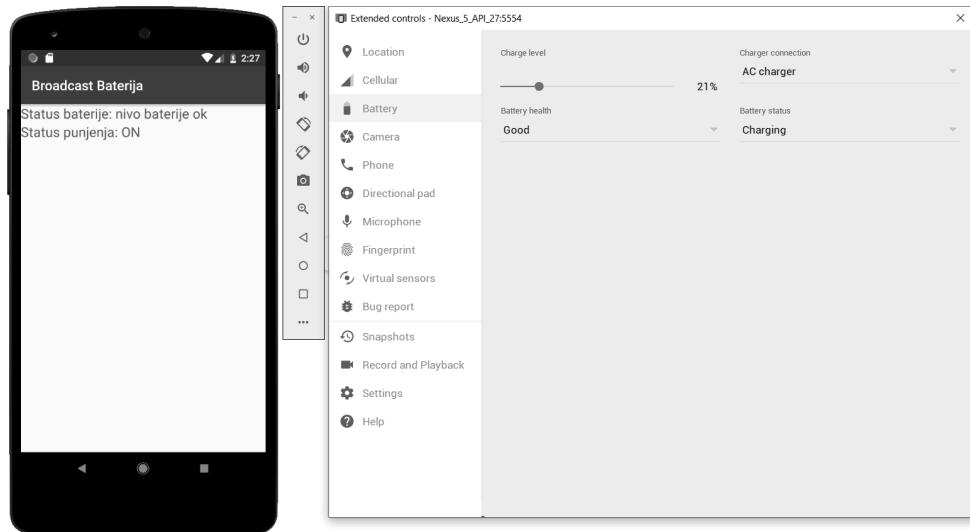
```
        break;
    case Intent.ACTION_BATTERY_OKAY:
        statusBaterije.setText("Status
baterije: " + "nivo baterije ok");
        break;
    case Intent.ACTION_POWER_CONNECTED:
        punjenjeBaterije.setText("Status
punjenja: " + "ON");
        // Moguce je odrediti i nacin punjenja
        // Kako se uredjaj puni odredjuje se na
sledeci nacin.
        // Nije prikazano na korisnickom
interfejsu u ovom primeru.
        int chargePlug =
intent.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
        boolean usbCharge = chargePlug ==
BatteryManager.BATTERY_PLUGGED_USB;
        boolean acCharge = chargePlug ==
BatteryManager.BATTERY_PLUGGED_AC;
        break;
    case Intent.ACTION_POWER_DISCONNECTED:
        punjenjeBaterije.setText("Status
punjenja: " + "OFF");
        break;
    }
}
}
```

Primer rada aplikacije je dat na slikama 10.3 i 10.4. Nivo baterije i status punjenja emulatora se može podešavati kroz dodatna podešavanja. Kada nivo baterije opadne ispod 20% sistem šalje obaveštenje Low Battery, kao što je prikazano na slici 10.3.



Slika 10.3, aplikacija sa prijemnikom obaveštenja o bateriji - ulazak u Low Battery

Kada se uređaj vrati u stanje baterije OK (iznad 20%), sistem će poslati novo obaveštenje. Sistem takođe šalje obaveštenja i u slučaju da se uređaj priključi / isključi sa punjača. Nakon obaveštenja da je sistem izašao iz stanja Low Battery, korisnički interfejs se ažurira, a njegov izgled je prikazan na slici 10.4.



Slika 10.4, aplikacija sa prijemnikom obaveštenja o bateriji – izlazak iz Low Battery

Primer 3: Realizovati aplikaciju koja presreće primljene pozive i primljene SMS poruke preko BroadcastReceiver komponente (napomena: od Android API nivoa 26 sistem vrši dodatnu restrikciju za neka od sistemskih obaveštenja, pa navedeni primer možda neće raditi ispravno na najnovijim verzijama Androida, ali lepo ilustruje princip rada prijemnika). Pokazati kako bi oba prijemnika bila realizovana sa registracijom kroz manifest aplikacije (napomena: najnovije verzije Android sistema zahtevaju da registracija za navedena obaveštenja bude implementirana dinamički).

Rešenje: Ovaj primer je namerno odabran kako bi ilustrovao problem ukoliko se nekoj aplikaciji bez razmišljanja daju dozvole. Pristup SMS porukama i pozivima korisnika spada u opasne dozvole – aplikacija koja ih ima može da zloupotrebi pristup i dohvati privatne korisničke informacije. Ukoliko to radi u pozadini, korisnik možda i neće biti svestan toga. U ovom primeru ćemo to ilustrovati tako što ćemo sadržaj primljene poruke prikazati u okviru Toast poruke. Maliciozna aplikacija bi mogla taj sadržaj poruke mogla preko nekog servisa u pozadini da pošalje na neku mrežnu lokaciju, a da korisnik uopšte nije svestan. Ovi problemi, naročito sa pozadinskim servisima, su rešeni uvođenjem restrikcija za servise i BroadcastReceiver komponente koje su uvedene u najnovijim verzijama Android sistema. Malo starije verzije, međutim, koje i dalje čine većinu na tržištu, nisu toliko zaštićene.

U ovom primeru je pokazana registracija prijemnika kroz manifest fajl. Sa najnovijim verzijama Androida, zahteva se da za većinu sistemskih obaveštenja prijemnik bude registrovan programski.

Ukoliko bi prijemnici bili registrovani kroz manifest, AndroidManifest.xml bi izgledao ovako:

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

        <uses-permission
    android:name="android.permission.READ_PHONE_STATE" />
        <uses-permission
    android:name="android.permission.INTERNET"/>
        <uses-permission
    android:name="android.permission.RECEIVE_SMS" />
        <application
            android:allowBackup="true"
            android:icon="@mipmap/ic_launcher"
            android:label="@string/app_name"
            android:roundIcon="@mipmap/ic_launcher_round"
```

```

        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action
                    android:name="android.intent.action.MAIN" />
                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <receiver android:name=".IncomingCallReceiver"
        android:enabled="true">
            <intent-filter>
                <action
                    android:name="android.intent.action.PHONE_STATE" />
            </intent-filter>
        </receiver>
        <receiver android:name=".SMSReceiver"
        android:enabled="true">
            <intent-filter>
                <action
                    android:name="android.provider.Telephony.SMS_RECEIVED">
                </action>
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

Oba prijemnika su deklarisana u okviru <receiver> elemenata, sa definisanim intent-filter elementima. U tom slučaju, nije potrebno registrovati prijemnike ručno kroz kod. Dodatno, pošto su prijemnici automatski prijavljeni, glavna aktivnost ne mora da uopšte bude svesna postojanja prijemnika. Kod MainActivity.java klase je ostavljen potpuno prazan:

```

package com.example.mzivkovic.myapplication;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

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

Prijemnici su realizovani kao odvojene klase. SMS prijemnik u okviru svoje onReceive metode dohvata sadržaj primljene SMS poruke i prikazuje ga kao Toast. SMS prijemnik se nalazi definisan u klasi SMSReceiver.java, čiji je sadržaj dat sa:

```
package com.example.mzivkovic.myapplication;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.telephony.SmsMessage;
import android.widget.Toast;

public class SMSReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        //dohvatanje sadrzaja primljene poruke
        Bundle pudsBundle = intent.getExtras();
        Object[] pdus = (Object[]) pudsBundle.get("pdus");
        SmsMessage messages
        =SmsMessage.createFromPdu((byte[]) pdus[0]);
        Toast.makeText(context, "Primljeni SMS:
" +messages.getMessageBody(),
                    Toast.LENGTH_LONG).show();
    }
}
```

Implementacija klase IncomingCallReceiver.java detektuje sve bitne faze poziva (dolazni poziv, poziv prihvaćen, poziv završen), i u svakom novom stanju izbacuje Toast sa odgovarajućim obaveštenjem. Programer bi na ovim mestima mogao da doda kod za logovanje, ili neki drugi vid reakcije na promenu stanja. Kod klase dat je u nastavku:

```
package com.example.mzivkovic.myapplication;

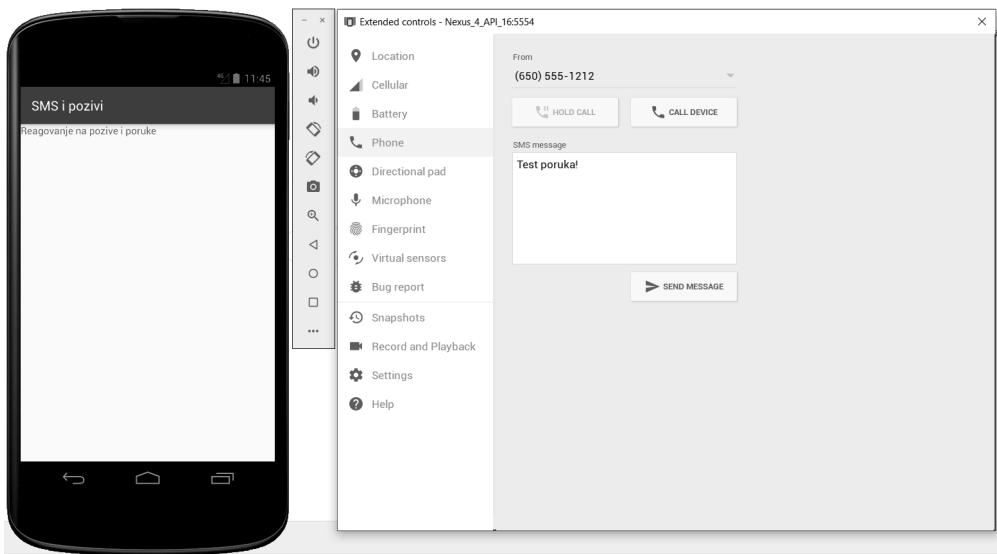
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.telephony.TelephonyManager;
import android.widget.Toast;

public class IncomingCallReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        try {

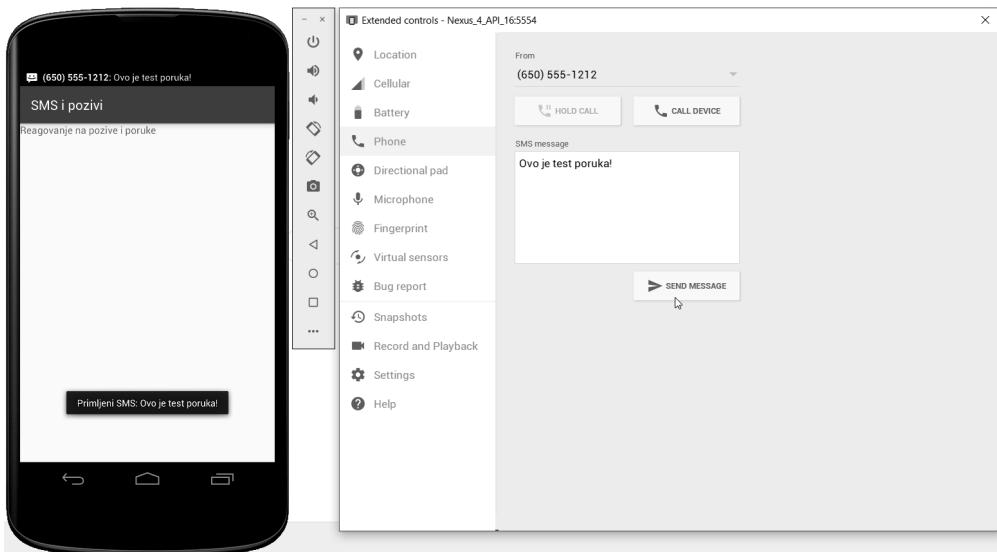
```

```
String state =  
intent.getStringExtra(TelephonyManager.EXTRA_STATE);  
  
if(state.equals(TelephonyManager.EXTRA_STATE_RINGING)) {  
    Toast.makeText(context, "Telefon zvoni",  
Toast.LENGTH_LONG).show();  
    // dalja implementacija ponasanja prilikom  
ovog dogadjaja  
}  
  
if(state.equals(TelephonyManager.EXTRA_STATE_OFFHOOK)) {  
    Toast.makeText(context, "Poziv prihvacen",  
Toast.LENGTH_LONG).show();  
    // dalja implementacija ponasanja prilikom  
ovog dogadjaja  
}  
if  
(state.equals(TelephonyManager.EXTRA_STATE_IDLE)) {  
    Toast.makeText(context, "Telefon je u idle  
stanju", Toast.LENGTH_LONG).show();  
    // dalja implementacija ponasanja prilikom  
ovog dogadjaja  
}  
}  
catch(Exception e) {  
    //rukovanje izuzetkom  
}  
}  
}  
}
```

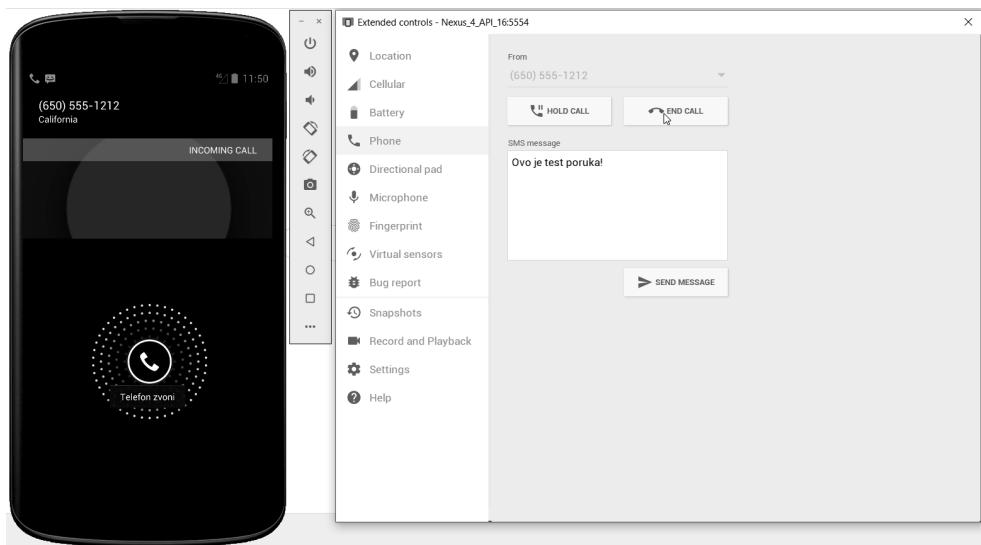
Ekran sa dodatnim podešavanjima emulatora omogućava simuliranje dolaznog poziva ili dolazne poruke, kroz ekran prikazan na slici 10.5. Ponašanje aplikacije prilikom dolazne poruke i dolaznog poziva prikazano je na slikama 10.6 i 10.7, respektivno. Sve ovo je omogućeno dozvolama koje bi korisnik dodelio aplikaciji na početku. Kao što se vidi, aplikacija ima potpuni pristup SMS porukama korisnika (maliciozna aplikacija ne bi prikazivala Toast poruke, već bi u pozadini sakupljala i slala sadržaj poruka na neku lokaciju na mreži), što ne bi bilo moguće ukoliko bi korisnik odbio da potvrди dozvolu za čitanje SMS. Zbog toga je jako važno da korisnik razmisli da li su dozvole koje aplikacija zahteva zaista potrebne. Na primer, nije logično da digitronu treba pristup porukama i listi poziva korisnika, pa bi trebalo razmisliti o opravdanosti zahteva tih dozvola, zbog toga što posledice mogu biti velike.



Slika 10.5, simulacija slanja poruka i poziva ka emulatoru.



Slika 10.6, presretanje SMS poruke pomoću BroadcastReceiver komponente



Slika 10.7, presretanje poziva pomoću BroadcastReceiver komponente

Primer 4: Emitovanje sopstvenog obaveštenja – kreirati aplikaciju koja emituje obaveštenje, u kome se nalazi poruka koju korisnik unese preko korisničkog interfejsa. Obaveštenje se emituje pritiskom na dugme. U okviru iste aplikacije definisati i implementirati prijemnik obaveštenja. Registraciju prijemnika implementirati deklaracijom u manifest fajlu aplikacije (za Android ispod verzije 8.0).

Rešenje: Korisnički interfejs ove aplikacije je jednostavan, treba da sadrži samo polje za unos teksta i dugme kojim će se emitovati obaveštenje. Sadržaj activity_main.xml fajla dat je u nastavku:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <EditText
        android:id="@+id/inputObavestenje"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Unesi obavestenje..."/>
    <Button
        android:id="@+id/buttonPosaljiObavestenje"
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        android:text="Posalji obavestenje"/>
</LinearLayout>
```

BroadcastReceiver komponenta realizovana je klasom PrijemnikObavestenja. U metodi onReceive() se iz pristiglog obaveštenja vadi poruka koja je smeštena pod ključem definisanim konstantom CUSTOM_KEY, i nakon toga prikazuje kao Toast. Kod klase PrijemnikObavestenja.java dat je u nastavku:

```
package com.example.student.sopstvenibroadcast;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.widget.Toast;

public class PrijemnikObavestenja extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        String poruka =
        intent.getStringExtra(MainActivity.CUSTOM_KEY);
        Toast.makeText(context, poruka,
        Toast.LENGTH_LONG).show();
    }
}
```

Prijemnik je registrovan kroz manifest aplikacije. Sadržaj fajla AndroidManifest.xml dat je u nastavku:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.student.sopstvenibroadcast">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action
                    android:name="android.intent.action.MAIN" />
                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

```

<receiver android:name=".PrijemnikObavestenja">
    <intent-filter>
        <action android:name=
"com.example.student.sopstvenibroadcast.VAZNO_OBAVESTENJE">
        </action>
    </intent-filter>
</receiver>
</application>
</manifest>

```

Jednostavna aktivnost koja koristi prethodno definisani korisnički interfejs, i nakon klika na dugme uzima poruku smeštenu u polje za unos, pakuje je u Intent i šalje kao obaveštenje, data je u klasi MainActivity.java, čiji je sadržaj:

```

package com.example.student.sopstvenibroadcast;

import androidx.appcompat.app.AppCompatActivity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity
implements View.OnClickListener {

    private EditText inputObavestenje;
    private Button buttonPosaljiObavestenje;

    public static final String CUSTOM_ACTION =
"com.example.student.sopstvenibroadcast.VAZNO_OBAVESTENJE";
    public static final String CUSTOM_KEY = "obavestenje";

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

    private void initComponents() {
        inputObavestenje =
findViewById(R.id.inputObavestenje);
        buttonPosaljiObavestenje =
findViewById(R.id.buttonPosaljiObavestenje);
        buttonPosaljiObavestenje.setOnClickListener(this);
    }
}

```

```
@Override
public void onClick(View v) {
    Intent intent = new Intent();
    intent.setAction(CUSTOM_ACTION);
    //dohvati unetu poruku i smesti u obavestenje
    String poruka =
inputObavestenje.getText().toString();
    intent.putExtra(CUSTOM_KEY, poruka);
    sendBroadcast(intent);
}
```

Rad aplikacije je prikazan na slici 10.8. Nakon unosa teksta u polje za unos (leva slika) i klika na dugme pošalji obaveštenje, prijemnik prihvata obaveštenje i prikazuje ga kao Toast poruku (desna slika).



Slika 10.8, Primer aplikacije koja šalje sopstvena obaveštenja

11. Servisi

Servisi su jedna od četiri osnovne komponente Android aplikacija. Oni su zaduženi da obavljaju dugotrajne pozadinske operacije, pri čemu ne pružaju korisnički interfejs korisniku aplikacije. Ove dugotrajne operacije mogu biti preuzimanje podataka sa Interneta ili druge mrežne operacije, ulazno izlazne operacije ili puštanje muzike u pozadini.

Servis se pokreće od strane neke druge komponente aplikacije – na primer, aktivnost može pokrenuti servis koji će puštati muziku u pozadini. U opštem slučaju, servis nastavlja sa izvršavanjem čak i u slučaju da se korisnik tokom korišćenja uređaja prebací na neku drugu aplikaciju. Po načinu izvršavanja razlikuju se tri osnovna tipa servisa:

- Servis koji se izvršava u prvom planu – ovakvi servisi obavljaju neki zadatak koji je vidljiv korisniku.
- Pozadinski servis – servisi ovog tipa obavljaju zadatke koje korisnik ne može direktno da primeti. Mora se napomenuti da je Android od API nivoa 26 ograničava izvršavanje servisa u pozadini ukoliko sama aplikacija nije u prvom planu.
- Vezani servisi – komponenta aplikacije se može vezati za servis pozivom metode bindService(). Na taj način se dobija struktura nalik na klijent-server arhitekturu, odnosno komponenta može slati zahteve servisu ili primati rezultate od njega. Životni vek ovako definisanog servisa direktno zavisi od komponente vezane za njega. Pošto je dozvoljeno vezati više komponenti za jedan servis, on će se izvršavati sve dok postoji bar jedna komponenta koja je još uvek vezana za njega. Nakon što poslednja komponenta pozove metodu unbindService(), servis će biti uništen.

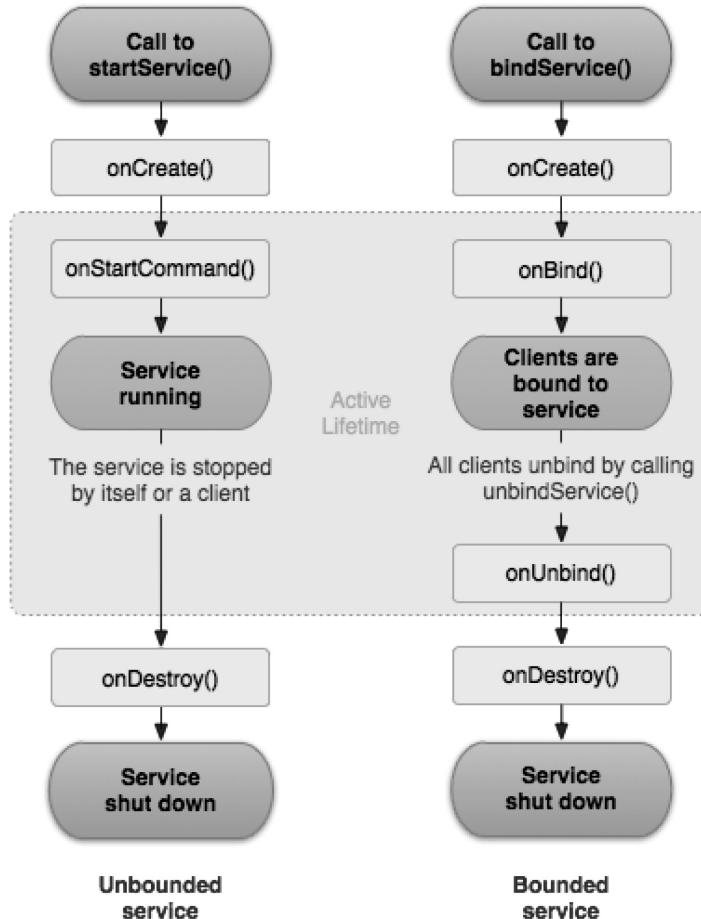
Servis se podrazumevano izvršava u korisničkoj niti svog procesa, odnosno ne kreira novu odvojenu nit izvršavanja automatski. To znači da je u slučaju dugotrajnih operacija potrebno kreirati novu nit u okviru samog servisa kako se ne bi opteretila korisnička nit i blokirao korisnički interfejs.

11.1. Životni ciklus servisa

Poput aktivnosti, servisi imaju svoj životni ciklus. U poređenju sa životnim ciklусом aktivnosti, servis je jednostavniji, ali zahteva posebnu pažnju - servis se može izvršavati u pozadini, a korisnik toga čak ne mora biti ni svestan. Neprimetan rad u pozadini može trošiti sistemske resurse i usporiti rad sistema,

pogotovo u slučaju nepravilne implementacije servisa. Zbog toga se kreiranje i uništavanje servisa moraju naročito pažljivo implementirati, kako ne bi došlo do bilo kakvog nepredviđenog ponašanja.

Za razliku od aktivnosti, servisi imaju dve moguće putanje kroz životni ciklus, u zavisnosti od toga na koji su način kreirani. Obe moguće putanje prikazane su na slici 11.1.



Slika 11.1. životni ciklus servisa

(slika preuzeta sa: <https://developer.android.com/guide/components/services>)

Servis se može kreirati na jedan od dva načina, i od načina kreiranja u potpunosti zavisi i način uništavanja servisa:

- Pokrenuti servis – u ovom slučaju servis je kreiran kada je neka druga komponenta aplikacije pozvala metodu `startService()`. Ovakav servis će

se izvršavati sve dok se samostalno ne zaustavi pozivom metode stopSelf(), ili dok ga neka druga komponenta ne zaustavi pozivom metode stopService(). Ukoliko se zaustavljanje (bilo samostalno bilo od druge komponente) ne implementira ispravno, servis se može izvršavati neodređeno dugo.

- Vezani servis – servis se kreira kada druga komponenta aplikacije pozove metodu bindService(). Ta komponenta se ponaša kao klijent, a sa servisom može da komunicira kroz IBinder interfejs. Klijent može zatvoriti vezu sa servisom pozivom metode unbindService(). Pošto je u opštem slučaju moguće da je više komponenti vezano za isti servis, on nastavlja da radi dokle god poslednja vezana komponenta ne pozove unbindService(), kada ga sistem automatski uništava. Veoma bitna osobina je da nije potrebno zaustaviti ovaj tip servisa.

Može se uočiti da su naročito problematični startovani servisi, jer je njih neophodno zaustaviti kako se ne bi izvršavali neodređeno dugo. Potrebno je napomenuti i da je moguće da se neka komponenta veže na već startovan servis. U tom slučaju, poziv stopSelf() ili stopService() neće imati efekta sve dok i poslednja vezana komponenta ne ukine vezu pozivom unbindService().

Prilikom implementacije servisa, mora se kreirati klasa izvedena iz klase Service, ili neke od njenih postojećih podklasa. Nakon toga se moraju nadjačati callback metode iz životnog ciklusa servisa kako bi se implementiralo odgovarajuće ponašanje i eventualno obezbedio mehanizam na koji se druge komponente mogu vezati za servis, ukoliko je to potrebno. Najvažnije callback metode koje bi uvek trebalo nadjačati su:

- onStartCommand() – sistem poziva ovu metodu kada neka druga komponenta aplikacije inicira startService(). Nakon izvršavanja metode onStartCommand(), servis je pokrenut i može se neodređeno dugo izvršavati u pozadini. Potpuna odgovornost programera je da zaustavi servis kada on završi svoj posao, pozivom jedne od metoda stopSelf() odnosno stopService(). Nije neophodno implementirati ovu metodu ukoliko se kreira vezani servis.
- onBind() – ako neka komponenta aplikacije inicira bindService(), sistem poziva metodu onBind(). Prilikom implementiranja ove metode mora se dati interfejs kroz koji će klijentska komponenta moći da komunicira sa servisom, a to se postiže vraćanjem IBinder objekta. Ova metoda se uvek mora implementirati, a u slučaju da nije potrebno omogućiti vezivanje komponenti, implementira se tako da vrati vrednost null.
- onCreate() – sistem poziva ovu metodu da izvrši inicijalizaciju, pre poziva onStartCommand() ili onBind().

- `onDestroy()` – sistem poziva ovu metodu prilikom uništavanja servisa, kada je potrebno osloboditi sve eventualno zauzete resurse, prekinuti niti ili odjaviti registrovane osluškivače.

11.2. Deklaracija servisa u manifestu

Poput drugih komponenti, i servise je neophodno deklarisati u okviru manifesta aplikacije. Za tu svrhu koristi se `<service>` element, koji se postavlja kao direktno dete `<application>` elementa. Primer definisanja servisa implementiranog klasom `MojServis` dat je sledećim isečkom XML koda:

```
<manifest ... >
    ...
    <application ... >
        <service android:name=".MojServis" />
        ...
    </application>
</manifest>
```

Servis element ima veći broj atributa koji se mogu koristiti za preciznije definisanje ponašanja servisa, a sintaksa definisanja data je sledećom šemom:

```
<service android:description="string resource"
         android:directBootAware=["true" | "false"]
         android:enabled=["true" | "false"]
         android:exported=["true" | "false"]
         android:foregroundServiceType=["connectedDevice" |
"dataSync" | "location" | "mediaPlayback" |
"mediaProjection" | "phoneCall"]
         android:icon="drawable resource"
         android:isolatedProcess=["true" | "false"]
         android:label="string resource"
         android:name="string"
         android:permission="string"
         android:process="string" >

    ...
</service>
```

Najbitniji atributi od navedenih su:

- `name`: ovaj atribut direktno imenuje klasu izvedenu iz `Service` klase, u kojoj je sam servis implementiran.
- `enabled`: atribut označava da li sistem može pokrenuti sistem.
- `permission`: dozvola koju komponenta mora da poseduje kako bi pokrenula ovaj servis ili se vezala na njega.

- **exported:** atribut označava da li komponente drugih aplikacija mogu pozivati ovaj servis (vrednost true označava da mogu).
- **foregroundServiceType:** ovim atributom se specificira ukoliko servis pripada određenom tipu servisa koji se izvršavaju u prvom planu.

Jedini obavezan atribut je name, ostali se mogu izostaviti prilikom deklaracije servisa u manifestu. U okviru servisa se može dodati i intent filter. Preporuka je, međutim, da se intent filter izostavi i da se uvek koriste eksplicitne namere kada se pokreće servis, jer je na taj način aplikacija bezbednija. Od Android 5.0, odnosno API nvio 21, sistem ne dozvoljava poziv bindService() metode sa implicitnom namerom. Servisi su pogotovo nezgodni zbog izvršavanja u pozadini, tako da ukoliko bi neka druga aplikacija pokrenula servis, korisnik vrlo verovatno toga ne bi bio svestan. Za najviši nivo sigurnosti preporučuje se da se dodatno postavi android:exported atribut na vrednost false, čime se servis čini dostupnim samo aplikaciji u okviru koje je definisan, a onemogućava druge aplikacije da ga pokrenu, čak i upotrebot eksplicitne namere.

11.3. Startovani servis

Prvi tip servisa je startovani servis. To su servisi koje druge komponente pokreću pozivom metode startService(), što rezultira pozivom metode onStartCommand(). Kroz metodu onStartCommand() servis dobija Intent objekat koji je pozivajuća komponenta prosledila kroz startService() metodu. Ovakvi servisi nakon pokretanja imaju potpuno odvojen životni ciklus od komponente koja ih je pokrenula, pa treba biti posebno oprezan. Čak i ako komponenta koja je pokrenula servis prestane da potoji, servis nastavlja da se izvršava sve dok se ne pozove jedna od metoda stopSelf() (ukoliko se servis sam zaustavlja kada završi svoj zadatak) ili stopService() (ukoliko ga zaustavlja neka druga komponenta).

Klasa koja implementira startovani servis treba da bude izvedena iz jedne od sledećih baznih klasa:

- **Service** – bazna klasa na vrhu hijerarhije, iz koje su izvedeni svi servisi. Servis čija je klasa implementirana izvođenjem iz Service klase se podrazumevano izvršava u glavnoj korisničkoj niti, pa je neophodno kreirati novu nit u okviru koje će servis obaviti svoj zadatak.
- **IntentService** – ova klasa je izvedena iz klase Service, sa razlikom što koristi radnu nit za prihvatanje zahteva za startovanje servisa (zahteve prihvata i obrađuje jedan po jedan). Ovo je i preporučeni način kreiranja startovanog servisa ukoliko je jedna nit i serijska obrada zahteva dovoljna za završetak potrebnog posla.

Oba pomenuta načina implementacije objašnjena su u nastavku poglavlja.

11.3.1. IntentService klasa

Prilikom izvođenja klase iz klase IntentService, ključni aspekt je implementacija onHandleIntent() metode, čiji zadatok je da prihvati intent za svaki zahtev za startovanje servisa koji se primi. Ovi zahtevi se obrađuju serijski, jedan po jedan, što je dovoljno za praktično sve moguće implementacije servisa (paralelna simultana obrada višestrukih zahteva se čak smatra za opasan scenario).

IntentService klasa obezbeđuje i kreira podrazumevanu radnu nit koja prihvata i izvršava sve Intent objekte koji su prosledeni u onStartCommand() metodi. Intent objekti se smeštaju u red za čekanje koji prosleđuje jedan po jedan intent metodi onHandleIntent(). Nakon završetka obrade svih pristiglih zahteva, IntentService klasa zaustavlja servis (nije potrebno pozivati stopSelf() metodu). IntentService dodatno pruža podrazumevanu implementaciju za sledeće metode:

- onBind() – podrazumevano vraća null.
- onStartCommand() – šalje intent u red za čekanje, a kad dođe na red prosleđuje se onHandleIntent() metodi

Implementacija klase koja je izvedena iz IntentService klase je veoma jednostavna. Potrebno je implementirati onHandleIntent() metodu i obezbediti konstruktor servisa. Jednostavan primer implementacije IntentService dat je sledećim isečkom Java koda:

```
public class MojPrimerIntentService extends IntentService {  
  
    /*  
     * Obavezno se mora dodati konstruktor koji poziva super  
     IntentService(String) konstruktorom.  
     * String parameter označava ime radne niti.  
     */  
    public MojPrimerIntentService() {  
        super("PrimerIntentService");  
    }  
  
    /*  
     * Intent service poziva metodu onHandleIntent iz radne  
     niti i prosledjuje joj intent objekat kojim je servis  
     startovan  
     * Kada se ova metoda završi i vrati, Intent Service  
     * zaustavlja servis, kao što je i zahtevano.  
     */  
    @Override
```

```

protected void onHandleIntent(Intent intent) {
    // U okviru ove metode treba da se izvrsti zahtevani
    posao.
    // U ovom primeru, nit samo spava 10 sekundi.
    try {
        Thread.sleep(10000);
    } catch (InterruptedException e) {
        // Zahtevani oporavak u slucaju prekida niti
    }
}
}

```

Kao što se iz priloženog primera vidi, ovaj način implementacije je veoma jednostavan. Dovoljno je implementirati onHandleIntent() metodu i dati jednostavni konstruktor. Nije neophodno nadjačati ostale callback metode obezbeđene u okviru životnog ciklusa servisa (onCreate(), onStartCommand(), kao i onDestroy()) pošto klasa IntentService već ima podrazumevanu implementaciju koja je zadovoljavajuća za skoro sve primene. Ukoliko se programer ipak odluči da implementira ove metode, potrebno je u okviru svake pojedinačne callback metode pozvati super() metodu nasleđenu iz IntentService klase, kako bi se obezbedio normalan rad servisa. Jedino u okviru metode onBind() nije potrebno zvati super() varijantu. Primer ispravnog nadjačavanja callback metode prikazan je u nastavku, isečkom Java koda koji pokazuje nadjačavanje onStartCommand() metode sa vraćanjem super() varijante iz klase IntentService:

```

@Override
public int onStartCommand(Intent intent, int flags, int
startId) {
    //prilagodjena metoda
    ...
    //mora pozvati na kraju super:
    return super.onStartCommand(intent,flags,startId);
}

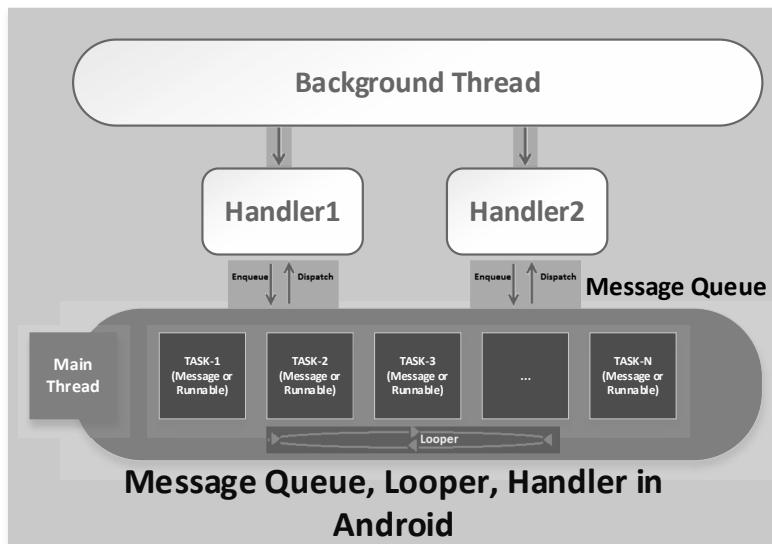
```

11.3.2. Service klasa

Service klasa se takođe može koristiti kao bazna za izvođenje klase koja treba da implementira servis. Za razliku od IntentService klase, koja značajno olakšava implementaciju servisa u slučaju da je jedna radna nit dovoljna, implementacija pomoću klase Service je dosta složenija i zahteva znatno više koda. Zbog toga se i favorizuje implementacija sa IntentService klasom. Međutim, ukoliko servis mora da izvršava više zahteva paralelno (višenitno izvršavanje), neophodno je koristiti klasu Service.

Za razliku od klase IntentService, gde je bilo dovoljno implementirati samo konstruktor i metodu onHandleIntent(), realizacija servisa pomoću klase Service zahteva implementaciju callback metoda životnog ciklusa servisa (kao što je navedeno u poglavlju 11.1). To su redom metode onCreate(), onDestroy(), onStartCommand() i onBind(). Pogledajmo kako bi izgledala implementacija jednostavnog primera datog u prethodnom poglavlju ukoliko bi se umesto IntentService klase koristila klasa Service.

Kako bi rešenje bilo potpuno u duhu Androida, koristi se nekoliko uslužnih klasa koje omogućavaju rad sa nitima. Klasa Handler omogućava slanje Message ili Runnable objekata u red za čekanje koji pripada datoј niti i njihovu kasniju obradu. Red za čekanje u Androidu predstavljen je klasom MessageQueue – ovo je klasa niskog nivoa koja sadrži listu poruka koje treba da budu otpremljene. Android listu poruka drži u glavnoj niti. Handler dodaje poruke u MessageQueue, dok je za otpremljivanje poruka zadužena Looper klasa. Looper opslužuje red za čekanje tako što kruži kroz poruke koje se u njemu nalaze i prosleđuje ih odgovarajućem Handler objektu na obradu. Svaka instanca Handler klase je povezana sa samo jednom niti i sa njenim redom za čekanje (MessageQueue) – isporučivaće poruke u red za čekanje i obradivati ih nakon što izađu iz reda za čekanje. Princip rada prikazan je na slici 11.2.



Slika 11.2, red za čekanje i Looper

Radna nit se implementira pomoću klase HandlerThread. Ova klasa kreira novu radnu nit koja izvršava zadatke sekvencijalno. Predstavlja korisno rešenje ukoliko je potrebna jedna pozadinska nit koja obrađuje poruke u redosledu u kome dolaze. Preko HandlerThread objekta se može instancirati Looper u okviru

niti, koji će obrađivati poruke u redu za čekanje. Za ubacivanje poruka (Message objekti) u red za čekanje Looper objekta koristi se Handler.

Nakon uvedenih pojmova, servis se može implementirati na sledeći način:

```
public class MojPrimerService extends Service {
    private Looper looper;
    private ServiceHandler handler;

    //Handler sluzi da prihvata poruke od niti
    private final class ServiceHandler extends Handler {
        public ServiceHandler(Looper looper) {
            super(looper);
        }
        @Override
        public void handleMessage(Message msg) {
            // Ovde treba obaviti zahtevani zadatok servisa
            // koji je obicno neki dugotrajni posao
            // U ovom primeru, nit samo spava 10 sekundi.
            try {
                Thread.sleep(10000);
            } catch (InterruptedException e) {
                // Zahtevani oporavak u slucaju prekida niti
            }
            // Zaustavljanje servisa preko ID, kako se ne bi
            // zaustavio servis koji obavlja neki drugi
            // zadatok
            stopSelf(msg.arg1);
        }
    }

    @Override
    public void onCreate() {
        // Pokrece se nit u kojoj se servis izvrsava. Nova nit
        // se koristi zato sto se servis podrazumevano izvrsava
        // u glavnoj niti procesa, koja se ne sme blokirati.
        Dodeljuje
        // se background prioritet kako zahtevni posao ne bi
        // ometao korisnicki interfejs
        HandlerThread thread = new
        HandlerThread("ServiceStartArguments",
                      Process.THREAD_PRIORITY_BACKGROUND);
        thread.start();

        // Dohvatanje Looper objekta HandlerThread-a i
        // iniciranje Handler-a
```

```

        looper = thread.getLooper();
        handler = new ServiceHandler(looper);
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int
startId) {
    Toast.makeText(this, "Servis je pokrenut",
Toast.LENGTH_LONG).show();

    // Za svaki request salje se poruka kako bi se
    // pokrenuo zadatak, pri cemu se prosledjuje i
    // ID kako bismo znali koji zahtev treba da se
    // zaustavi kada se zadatak završi
    Message msg = handler.obtainMessage();
    msg.arg1 = startId;
    handler.sendMessage(msg);

    // Ukoliko se nit ubije, nakon povratka uraditi
    restart
    return START_STICKY;
}

@Override
public IBinder onBind(Intent intent) {
    // U ovom primeru nema vezivanja, vraca se null
    return null;
}

@Override
public void onDestroy() {
    Toast.makeText(this, "Servis je zavrsen",
Toast.LENGTH_LONG).show();
}

```

Iz ovog jednostavnog primera bi trebalo da bude veoma očigledno koliko je upotreba IntentService klase lakša, naravno u slučaju da je potrebna jedna nit. Implementacija klase Service zahteva mnogo više posla, ali ima i svojih prednosti. Konkretno, programer ima mogućnost da implementira onStartCommand() na takav način da se može opslužiti i više zahteva paralelno. U datom primeru bi bilo potrebno kreirati novu nit za svaki zahtev i izvršiti je odmah, umesto čekanja da se prethodni zahtev završi. Još jedna važna stvar je celobrojna povratna vrednost metode onStartCommand(). Ova vrednost definiše kako sistem treba da nastavi servis u slučaju da ga Android ubije. Moguće je postaviti jednu od sledećih konstanti:

- START_NOT_STICKY – ukoliko sistem ubije servis, on se neće ponovo kreirati, osim u slučaju da postoje Intent objekti koji čekaju da budu prosleđeni.
- START_STICKY – ukoliko sistem ubije servis, on će biti ponovo kreiran, metoda onStartCommand() će ponovo biti pozvana sa null Intent objektom ili sa Intent objektima koji čekaju (ukoliko postoje). Ovaj način je pogodan za medija plejere ili slične servise, koji se izvršavaju neodređen vremenski period i čekaju na zadatku.
- START_REDELIVER_INTENT – ukoliko sistem ubije servis, on će biti ponovo kreiran, metoda onStartCommand() će ponovo biti pozvana sa poslednjim Intent objektom koji je bio prosleđen servisu (nakon kog će biti prosleđeni eventualni Intent objekti koji čekaju). Ovaj način je pogodan za servise koji aktivno izvršavaju zadatku koji se mora odmah nastaviti, poput preuzimanja fajla.

Ove vrednosti su relevantne samo u retkim slučajevima da uređaj ostane bez memorije i ubije servis pre nego što on uspe da završi svoj zadatku.

11.3.3. Pokretanje i zaustavljanje servisa

Servis pokreće aktivnost ili neka druga komponenta aplikacije, pozivom metode startService() kojoj prosleđuje Intent objekat. Sistem nakon toga automatski poziva onStartCommand() metodu servisa kojoj prosleđuje dati Intent objekat. Kao što je ranije navedeno, sistem od Android API nivoa 26 uvodi restrikcije na upotrebu pozadinskih servisa ukoliko se sama aplikacija ne nalazi u prvom planu. Ukoliko aplikacija treba da pokrene servis u prvom planu, potrebno je pozvati metodu startForegroundService() umesto metode startService(). Metoda startForegroundService() će pokrenuti pozadinski servis uz najavu sistemu da će se servis promovisati u prvi plan. Nakon kreiranja servisa, mora se pozvati njegova startForeground() metoda u roku od 5 sekundi.

Na primer, servis opisan u prethodnom poglavljju, u klasi pod imenom MojPrimerService, se može pokrenuti u nekoj aktivnosti na sledeći način, upotrebom eksplisitne namere:

```
Intent intent = new Intent(this, MojPrimerService.class);
startService(intent);
```

Metoda startService() se odmah izvršava do kraja i vraća, dok Android pokreće prvo metodu onCreate() (u slučaju da se servis već ne izvršava), a nakon toga i onStartCommand().

Kao što je već više puta navedeno, startovani servis se mora zaustaviti, inače će se izvršavati neodređen vremenski period. Tačnije, sistem će ga uništiti samo u

slučaju da mora da osloboди memoriju. Ukoliko se servis sam ne zaustavlja pozivom metode stopSelf(), neka druga komponenta ga mora zaustaviti pozivom metode stopService(), kao što je prikazano u nastavku:

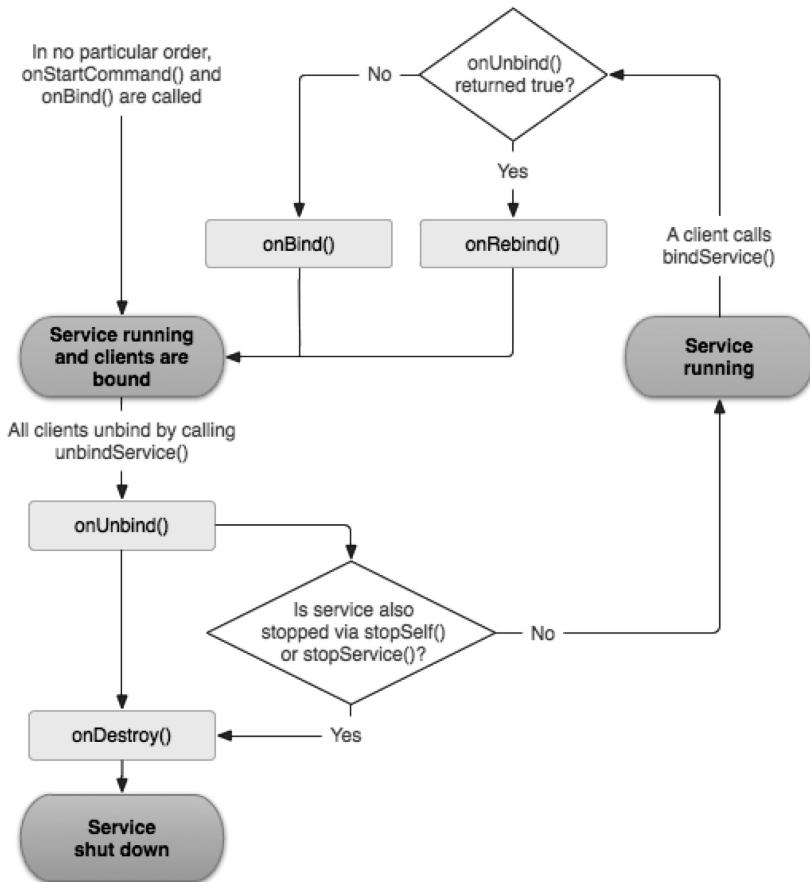
```
Intent intent = new Intent(this, MojPrimerService.class);
stopService(intent);
```

11.4. Vezani servis

Servis daje mogućnost nekoj drugoj komponenti aplikacije da se veže na njega pozivom metode bindService() - u tom slučaju reč je o vezanom servisu. Vezani servisi su odlično rešenje kada je potrebno vršiti interakciju između samog servisa i aktivnosti koja je za njega vezana. Servis u ovom kontekstu ima povezan životni ciklus sa vezanom komponentom koju opslužuje, odnosno nije potrebno zaustavljati ovako definisan servis. Dozvoljeno je da se više komponenti poveže sa servisom u isto vreme. Kada komponenta završi interakciju sa servisom, jednostavno poziva metodu unbindService(), kojom se veza sa servisom ukida. Kada i poslednja povezana komponenta ukine vezu sa servisom, Android sistem ga u najkraćem mogućem roku uništava.

Vezani servis se pravi tako što se implementira metoda onBind(), koja mora da vrati tip IBinder. IBinder je interfejs koji aktivnost može da koristi za komunikaciju sa servisom. Kada aktivnost pozove bindService(), kao rezultat dobija interfejs preko koga može pozivati metode servisa.

Treba napomenuti da je moguće napraviti servis koji je istovremeno i startovan i vezan. To je slučaj kada je servis pokrenut pozivom metode startService(), čime se omogućava da se on izvršava nedefinisano dugo, a druge komponente se mogu vezati za njega pozivom metode bindService(). Ukoliko se servis implementira na ovaj način, sistem neće uništiti servis kada se sve vezane komponente otkače sa njega. Drugim rečima, ukoliko je vezani servis pokrenut sa startService(), mora se eksplicitno zaustaviti pozivom stopSelf() ili stopService() metoda. Ovakav životni ciklus servisa prikazan je na slici 11.3.



Slika 11.3, životni ciklus servisa koji je startovan i vezan u isto vreme.
(slika preuzeta sa: <https://developer.android.com/guide/components/bound-services.html#Lifecycle>)

Za kreiranje servisa koji omogućava vezivanje, mora se obezbititi IBinder – programski interfejs koji će vezane komponente koristiti za interakciju sa servisom. Postoji nekoliko načina definisanja ovog interfejsa:

- Izvođenje iz Binder klase – koristi se u slučaju da se servis koristi isključivo u okviru aplikacije u kojoj je definisan, odnosno izvršava se u istom procesu u kome se izvršava i vezana komponenta. Ovo je preporučeni način programiranja vezanog servisa, a jedini način kada se ne koristi jeste u slučaju da je potrebno da servis bude dostupan za korišćenje iz drugih aplikacija i procesa.
- Upotreba poruka (Messenger) – ovaj način se koristi ukoliko je potrebno da servis mogu da koriste i komponente iz drugih procesa. U ovom

slučaju, servis mora definisati Handler koji odgovara na različite tipove poruka (Message objekti). Handler je osnova Messenger objekta, koji može da podeli IBinder sa klijentom. Klijent onda može da šalje komande servisu putem Message objekata.

U nastavku ove knjige biće opisan prvi način, odnosno servis koji se vezuje za komponente u okviru iste aplikacije, pri čemu se koristi izvođenje iz klase Binder. To je i najčešći oblik upotrebe vezanog servisa. Binder klasa omogućava vezanoj komponenti direktni pristup metodama servisa.

Ceo postupak se može opisati sledećim koracima:

- U okviru servisa se kreira instanca Binder klase koja ispunjava jedan od sledećih uslova:
 - Sadrži javne metode koje klijentska komponenta može da koristi.
 - Vraća kao rezultat instancu Service klase, koja ima javne metode koje klijentska komponenta može da poziva.
 - Vraća instancu neke druge klase koja je ugnježđena u servis, a koja ima javne metode koje klijentska komponenta može da poziva.
- Prethodno definisana instanca Binder klase treba da se vrati u okviru onBind() metode.
- Vezana klijentska komponenta prihvata Binder u okviru callback metode onServiceConnected(), i nakon toga poziva vezani servis kroz dostupne metode.

Primer implementacije vezanog servisa koji koristi Binder klasu dat je u nastavku:

```
public class LokalniVezaniServis extends Service {  
    // Binder objekat koji se daje vezanim komponentama  
    private final IBinder binder = new LokalniBinder();  
    // Generator slučajnih brojeva  
    private final Random mGenerator = new Random();  
  
    /*  
     * Definicija klase koju klijenti koriste kao Binder  
     */  
    public class LokalniBinder extends Binder {  
        LokalniVezaniServis getService() {  
            // Vraca ovu instancu lokalnog servisa, pa  
            klijenti mogu da zovu javne metode  
            return LokalniVezaniServis.this;  
        }  
    }  
}
```

```

@Override
public IBinder onBind(Intent intent) {
    return binder;
}

/* Javno dostupna metoda za klijente, u ovom slucaju
samo generise slucajni broj */
public int getSlucajniBroj() {
    return mGenerator.nextInt(100);
}
}

```

LokalniBinder klasa pruža metodu getService() kojom vezane komponente mogu da dohvate trenutnu instancu klase LokalniVezaniServis. Nakon toga, vezana komponenta može direktno da zove javne metode u okviru servisa, u našem slučaju metodu getSlucajniBroj(). Primer aktivnosti koja se vezuje za LokalniVezaniServis i poziva njegovu metodu getSlucajniBroj() kada se klikne na dugme dat je u nastavku:

```

public class VezanaAktivnost extends Activity {
    LokalniVezaniServis mService; //instanca servisa
    boolean mBound = false; // indikator da li je servis
vezan

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

    @Override
    protected void onStart() {
        super.onStart();
        // povezivanje sa servisom
        Intent intent = new Intent(this,
LokalniVezaniServis.class);
        bindService(intent, connection,
Context.BIND_AUTO_CREATE);
    }

    @Override
    protected void onStop() {
        super.onStop();
        //raskinuti vezu sa vezanim servisom
        unbindService(connection);
        mBound = false;
    }
}

```

```

/* Poziva se nakon klika na dugme definisano u XML
fajlu */
public void onButtonClick(View v) {
    if (mBound) {
        // Pozvati metodu iz lokalnog servisa dostupnog
        // kroz promenljivu mService.
        int slucajniBroj = mService.getSlucajniBroj();
        Toast.makeText(this, "dobijeni broj: " +
        slucajniBroj, Toast.LENGTH_SHORT).show();
    }
}

/* Definicija callback metoda za vezivanje servisa */
private ServiceConnection connection = new
ServiceConnection() {

    @Override
    public void onServiceConnected(ComponentName
        className,
        IBinder service) {
        // Povezani smo sa lokalnim servisom, potrebno
        // je konvertovati IBinder u LokalniBinder i dohvatiti
        // instancu klase LokalniVezaniServis
        LokalniBinder binder = (LokalniBinder) service;
        mService = binder.getService();
        //postaviti boolean indikator na true posto smo
        povezani
        mBound = true;
    }

    @Override
    public void onServiceDisconnected(ComponentName
        arg0) {
        // postaviti boolean indikator na false, posto
        // je veza sa servisom raskinuta
        mBound = false;
    }
}
}

```

Ova aktivnost se povezuje sa servisom pozivom metode bindService(). Android sistem zauzvrat poziva metodu onBind() servisa na koji se aktivnost povezuje, a koja vraća IBinder interfejs koji služi za interakciju sa željenim servisom. Povezivanje se vrši asinhrono, odnosno metoda bindService() se izvršava i vraća odmah, bez vraćanja IBinder objekta pozivajućoj aktivnosti. Da bi aktivnost mogla da prihvati IBinder, mora kreirati instancu ServiceConnection koju će

proslediti metodi bindService(). ServiceConnection je interfejs koji pruža callback metodu kroz koju će sistem proslediti IBinder.

Svi koraci koje aktivnost treba da implementira kako bi se uspešno vezala na servis su sledeći:

- Implementacija ServiceConnection interfejsa. Obavezno je nadjačavanje dve metode:
 - onServiceConnected() – kroz ovu callback metodu Android sistem isporučuje IBinder koji vraća metoda onBind() iz servisa na koji se aktivnost vezuje.
 - onServiceDisconnected() – Android sistem koristi ovu metodu u slučaju da se veza sa servisom izgubi usled ubijanja servisa.
- Poziva se bindService(), pri čemu se kao parametar prosleđuje prethodno opisani implementirani ServiceConnection.
- Kada Android sistem pozove onServiceConnected() callback metodu definisanu u okviru implementacije interfejsa ServiceConnection, pozivajuća aktivnost može početi da poziva metode servisa na način definisan realizacijom IBinder interfejsa.
- Kada je potrebno raskinuti vezu sa servisom, jednostavno se poziva unbindService().

Tipična implementacija ServiceConnection data je u nastavku:

```
private ServiceConnection connection = new
ServiceConnection() {

    @Override
    public void onServiceConnected(ComponentName
className, IBinder service) {
        // Povezani smo sa lokalnim servisom, potrebno
je konvertovati IBinder u LokalniBinder i dohvatiti
instancu klase LokalniVezaniServis
        LokalniBinder binder = (LokalniBinder) service;
        mService = binder.getService();
        //postaviti mBound na true posto smo povezani
        mBound = true;
    }
    @Override
    public void onServiceDisconnected(ComponentName
arg0) {
        // postaviti boolean indikator na false, posto
je veza sa servisom raskinuta
        mBound = false;
    }
};
```

Sa ovako implementiranim interfejsom ServiceConnection, aktivnost se može povezati sa servisom pozivanjem metode bindService()):

```
Intent intent = new Intent(this, LokalniVezaniServis.class);
bindService(intent, connection, Context.BIND_AUTO_CREATE);
```

Parametri poziva bindService() su:

- Intent objekat koji eksplisitno navodi ime servisa na koji se aktivnost povezuje. Obavezno se mora koristiti eksplisitna namera, Android počev od verzije 5.0 izbacuje izuzetak ukoliko se pokuša vezivanje putem implicitne namere.
- Objekat ServiceConnection.
- Indikator koji opisuje dodatne opcije za vezivanje. Tipično treba da bude BIND_AUTO_CREATE, koji kreira servis ukoliko nije već aktivan.

Na kraju, potrebno je napomenuti da komponenta koja se povezuje sa servisom ne mora isključivo biti aktivnost. To može biti i neki drugi servis, ili ContentProvider komponenta. Od četiri osnova tipa komponenti u Android aplikaciji, jedino BroadcastReceiver komponenta ne može da se poveže sa servisom, zbog svog specifičnog životnog ciklusa.

11.5. Pitanja za vežbu

- Koja je uloga servisa u Android aplikacijama?
- Koji tipovi servisa postoje?
- Da li je potrebno prilikom razvoja sopstvene klase servisa proširiti neku Android klasu? Ukoliko da, koju?
- Objasniti callback metode u okviru životnog ciklusa servisa.
- Šta je startovani servis?
- Šta je vezani servis?
- Opisati jasno razliku između startovanog i vezanog servisa.
- Kako se servis definiše u manifestu aplikacije?
- Kako se pokreće servis?
- Kako se može zaustaviti servis?
- Šta je IntentServis?
- Da li IntentServis može da izvrši više zadataka paralelno?
- Kako se može implementirati servis koji omogućava paralelno izvršavanje zadataka?
- Da li je moguće da jedan vezani servis bude povezan sa više klijenata – komponenti?
- Kada se uništava vezani servis?
- Čemu služi IBinder interfejs u vezanom servisu?
- Da li je potrebno zaustaviti vezani servis?
- Čemu služi ServiceConnection interfejs?
- Da li servis može istovremeno da bude i startovan i vezan? U slučaju da može, opisati način funkcionisanja.
- Objasniti razliku između stopSelf() i stopService() metoda. Ko zove koju metodu?
- Koja metoda se poziva kada neka komponenta želi da se poveže sa servisom?
- Koja metoda se poziva kada komponenta želi da raskine vezu sa povezanim servisom?
- Da li je moguće povezati servis sa drugim servisom? Ukoliko ne, objasniti zašto.
- Da li je moguće povezati servis sa BroadcastReceiver komponentom? Ukoliko ne, objasniti zašto.

11.6. Zadaci za vežbu

Primer 1: Realizovati servis koji pušta pozadinsku muziku. Grafički interfejs treba da ima samo dva dugmeta, dugme start koje pokreće servis, i dugme stop koje zaustavlja servis. Servis treba da bude implementiran u obliku startovanog servisa (nije vezan, dakle potrebno ga je zaustaviti kako se ne bi izvršavao neodređeno dugo). Muziku je potrebno pustiti preko MediaPlayer komponente. Kao „muziku“ pustiti podrazumevani ringtone koji će se ponavljati sve dok se servis ne zaustavi (looping postavljen na true).

Rešenje:

Korisnički interfejs je jednostavan, sastoji se od dva dugmeta, koja se mogu postaviti u vertikalni linearni raspored. Sadržaj activity_main.xml fajla dat je sa:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <Button
        android:id="@+id/buttonStart"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Pokreni servis"/>
    <Button
        android:id="@+id/buttonStop"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Zaustavi servis"/>
</LinearLayout>
```

Servis je realizovan klasom PozadinskaMuzikaServis.java. Sadržaj ove klase dat je u nastavku:

```
package com.example.student.servispozadinskamuzika;

import android.app.Service;
import android.content.Intent;
import android.media.MediaPlayer;
import android.os.IBinder;
import android.provider.Settings;
import android.widget.Toast;

public class PozadinskaMuzikaServis extends Service {

    private MediaPlayer player;
```

```

@Override
public IBinder onBind(Intent intent) {
    return null;
}

@Override
public void onCreate() {
    Toast.makeText(this, "Servis je kreiran",
Toast.LENGTH_LONG).show();
}

@Override
public int onStartCommand(Intent intent, int flags, int
startId) {
    player = MediaPlayer.create(this,
Settings.System.DEFAULT_RINGTONE_URI);
    // Rington ce se pustati u kontinuitetu sve dok se
servis ne zaustavi.
    player.setLooping(true);
    // Ova naredba pokreće plejer
    player.start();
    Toast.makeText(this, "Servis je pokrenut",
Toast.LENGTH_LONG).show();
    return START_STICKY;
}

@Override
public void onDestroy() {
    super.onDestroy();
    // Zaustavljamo plejer i oslobođamo sve resurse
    player.stop();
    player.release();
    player = null;
    Toast.makeText(this, "Servis je zaustavljen",
Toast.LENGTH_LONG).show();
}
}

```

U ovom primeru MediaPlayer jednostavno pušta podrazumevani ringtone. Ukoliko bi audio fajl bio postavljen kao lokalni sirovi resurs (u direktorijumu res/raw/), pod identifikatorom audio_1, MediaPlayer bi se mogao inicijalizovati na sledeći način:

```

MediaPlayer player = MediaPlayer.create(context,
R.raw.audio_1);
player.start();

```

Sa druge strane, ukoliko bi bilo potrebno pustiti audio sa udaljenog URL preko HTTP striminga, inicijalizacija MediaPlayer komponente bi mogla biti realizovana na sledeći način:

```
String url = "http://....."; // postaviti URL ovde
MediaPlayer player = new MediaPlayer();
player.setAudioStreamType(AudioManager.STREAM_MUSIC);
player.setDataSource(url);
player.prepare(); // ova naredba moze da potraje dugo!
//(buffering)
player.start();
```

Metoda prepare() može da bude rizična ukoliko zbog baferovanja potraje previše dugo. Zbog toga se može koristiti asinhroni metod prepareAsync().

Glavna aktivnost koja koristi ovaj servis data je klasom MainActivity.java. Na klik na dugme buttonStart, servis se pokreće, dok se zaustavlja klikom na dugme buttonStop. Kod ove aktivnosti dat je u nastavku:

```
package com.example.student.servispozadinskamuzika;

import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;

public class MainActivity extends AppCompatActivity
implements View.OnClickListener {
    private Button buttonStart;
    private Button buttonStop;

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

    private void initComponents () {
        //dohvatanje referenci na komponente interfejsa
        buttonStart = findViewById(R.id.buttonStart);
        buttonStop = findViewById(R.id.buttonStop);
        //postavljanje listenera
        buttonStart.setOnClickListener(this);
        buttonStop.setOnClickListener(this);
    }
    @Override
    public void onClick(View v) {
```

```

        switch(v.getId()){
            case R.id.buttonStart:
                pokreniServis();
                break;
            case R.id.buttonStop:
                zaustaviServis();
                break;
        }
    }
    public void pokreniServis(){
        startService(new Intent(this,
PozadinskaMuzikaServis.class));
    }
    public void zaustaviServis(){
        stopService(new Intent(this,
PozadinskaMuzikaServis.class));
    }
}

```

Sadržaj AndroidManifest.xml fajla, gde se mora deklarisati servis, dat je u nastavku:

```

<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.student.servispozadinskamuzika">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action
                    android:name="android.intent.action.MAIN" />
                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name=".PozadinskaMuzikaServis" />
    </application>
</manifest>

```

Prikaz rada aplikacije dat je na slici 11.4. Klik na dugme pokreni servis pokreće pozadinsku muziku, dok se ista zaustavlja klikom na dugme zaustavi servis.



Slika 11.4, prikaz rada servisa koji pokreće muziku u pozadini.

Primer 2: Realizovati servis koji prihvata poruku koju korisnik unese preko korisničkog interfejsa, i vrati je u obrnutom redosledu karaktera, sa zadrškom od 5 sekundi. Kada korisnik na primer unese poruku „Dobar dan“ i pritisne dugme Pošalji, nakon 5 sekundi treba da dobije odgovor od servisa sledećeg sadržaja: „nad raboD“. Servis realizovati pomoću klase IntentService. Za komunikaciju između servisa i aktivnosti koristiti lokalna obaveštenja (Local Broadcast).

Rešenje: BroadcastReceiver komponente se mogu koristiti za prenos poruka između komponenti aplikacije. Pošto su u pitanju lokalna obaveštenja između aktivnosti i servisa u okviru iste aplikacije, može se koristiti LocalBroadcastManager.

Korisnički interfejs je definisan u activity_main.xml fajlu, čiji je sadržaj dat sa:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
    android:layout_width="match_parent"
        android:layout_height="match_parent">
    <EditText
        android:id="@+id/inputPoruka"
        android:layout_width="match_parent"
```

```

        android:layout_height="wrap_content"
        android:hint="Unesi poruku"/>
    <Button
        android:id="@+id/buttonPosaljiPoruku"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Posalji poruku"/>
    <TextView
        android:id="@+id/labelOdgovor"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="15dp"
        android:lines="20" />
</LinearLayout>
```

Servis koji obrće prosleđeni string sa zadrškom od 5 sekundi realizovan je pomoću IntentService klase. Klasa ServisPoruke.java, koja je izvedena iz klase IntentService, data je sledećim kodom:

```

package com.example.student.primerintentservice;

import android.app.IntentService;
import android.app.Service;
import android.content.Intent;
import android.os.SystemClock;
import androidx.annotation.Nullable;
import androidx.localbroadcastmanager.content.LocalBroadcastManager;

public class ServisPoruke extends IntentService {

    public ServisPoruke() {
        super("ServisPoruke");
    }
    @Override
    protected void onHandleIntent(@Nullable Intent intent)
    {
        String poruka = intent.getStringExtra("poruka");
        intent.setAction(MainActivity.FILTER_ACTION_KLJUC);
        StringBuilder sb = new StringBuilder();
        sb.append(poruka);
        String obrnutaPoruka = sb.reverse().toString();
        SystemClock.sleep(5000);
        LocalBroadcastManager.getInstance(getApplicationContext())
            .sendBroadcast(intent.putExtra("odgovorServisa",
                obrnutaPoruka));
    }
}
```

Aktivnost koja komunicira sa servisom je implementirana u klasi MainActivity.java, čiji sadržaj je dat u nastavku. Ova aktivnost mora da sadrži BroadcastReceiver komponentu, preko koje će prihvatići odgovor koji će servis poslati u obliku lokalnog obaveštenja. To je implementirano pomoću ugnježdene klase LokalniReceiver.

```
package com.example.student.primerintentservice;

import androidx.appcompat.app.AppCompatActivity;
import androidx.localbroadcastmanager.content.LocalBroadcastManager;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity
implements View.OnClickListener{

    private Button buttonPosaljiPoruku;
    private EditText inputPoruka;
    private TextView labelOdgovor;
    private LokalniReceiver receiver;

    //Akcija za lokalni broadcast receiver
    public static final String FILTER_ACTION_KLJUC =
"com.example.student.primerintentservice.akcija";

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

    private void initComponents(){
        buttonPosaljiPoruku =
findViewById(R.id.buttonPosaljiPoruku);
        inputPoruka = findViewByIcon(R.id.inputPoruka);
        labelOdgovor = findViewByIcon(R.id.labelOdgovor);
        buttonPosaljiPoruku.setOnClickListener(this);
    }
}
```

```

@Override
public void onClick(View v) {
    //uzeti uneti tekst i poslati servisu
    String poruka = inputPoruka.getText().toString();
    //proslediti poruku servisu
    Intent intent = new Intent(MainActivity.this,
ServisPoruke.class);
    intent.putExtra("poruka", poruka);
    startService(intent);
}

@Override
protected void onStart() {
    receiver = new LokalniReceiver();
    IntentFilter intentFilter = new IntentFilter();
    intentFilter.addAction(FILTER_ACTION_KLJUC);
    LocalBroadcastManager.getInstance(this).
registerReceiver(receiver, intentFilter);

    super.onStart();
}

@Override
protected void onStop() {
    unregisterReceiver(receiver);
    super.onStop();
}

private class LokalniReceiver extends BroadcastReceiver
{
    @Override
    public void onReceive(Context context, Intent intent) {
        String odgovor =
intent.getStringExtra("odgovorServisa");
        labelOdgovor.setText(odgovor);
    }
}
}

```

Sadržaj AndroidManifest.xml fajla, gde je deklarisan servis:

```

<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.student.primerintentservice">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"

```

```
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity">
        <intent-filter>
            <action
                android:name="android.intent.action.MAIN" />
            <category
                android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <service android:name=".ServisPoruke"/>
</application>
</manifest>
```

Prikaz rada aplikacije prikazan je na slici 11.5. Kada se pošalje poruka, nakon 5 sekundi će stići odgovor od servisa (slika desno).



Slika 11.5, prikaz rada aplikacije sa IntentService

Primer 3: Napraviti servis koji generiše slučajne brojeve. Servis realizovati u obliku vezanog servisa, koji omogućava vezanoj aktivnosti da pristupi njegovoj metodi koja generiše i vraća slučajni broj.

Rešenje: Korisnički interfejs aplikacije može biti jednostavan, dovoljno je jedno dugme na koje kad se klikne treba da se pozove servis da vrati slučajan broj. Interfejs je dat sadržajem fajla activity_main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
    android:layout_width="match_parent"
        android:layout_height="match_parent">
    <Button
        android:id="@+id/buttonGenerisiBroj"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Generisi slučajan broj"/>

    <TextView
        android:id="@+id/labelOdgovor"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="15dp"
        android:lines="2" />
</LinearLayout>
```

Servis se realizuje kao vezani. Koristi se IBinder interfejs da se vrati instanca servisa kroz poziv onBind(). Implementacija servisa data je kodom SlucajniBrojeviServis.java klase:

```
package com.example.student.vezaniservis;

import android.app.Service;
import android.content.Intent;
import android.os.Binder;
import android.os.IBinder;

import java.util.Random;

public class SlucajniBrojeviServis extends Service {

    private Random generatorBrojeva = new Random();
    private final IBinder lokalniBinder = new
LokalniBinder();

    @Override
    public IBinder onBind(Intent intent) {
        return lokalniBinder;
    }
}
```

```

@Override
public boolean onUnbind(Intent intent) {
    return super.onUnbind(intent);
}

@Override
public void onRebind(Intent intent) {
    super.onRebind(intent);
}

@Override
public void onDestroy() {
    super.onDestroy();
}

public int generisiSlucajanBroj(){
    int slucajniBroj = generatorBrojeva.nextInt();
    return slucajniBroj;
}

public class LokalniBinder extends Binder {

    public SlucajniBrojeviServis getService() {
        return SlucajniBrojeviServis.this;
    }
}
}

```

Servis pruža javno dostupnu metodu generisiSlucajanBroj(), koju će pozivati vezana aktivnost. Vezana aktivnost je data klasom MainActivity.java. Aktivnost drži referencu na servis u promenljivoj boundedService, a boolean promenljiva isBound govori da li smo trenutno povezani sa servisom ili ne. Povezivanje sa servisom se radi u okviru onStart() metode, pozivom bindService(). Raskidanje veze sa servisom se radi u okviru metode onStop(). Implementacija klase MainActivity.java data je sledećim kodom:

```

package com.example.student.vezaniservis;

import android.content.ComponentName;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.IBinder;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

```

```

public class MainActivity extends AppCompatActivity
implements View.OnClickListener{

    SlucajniBrojeviServis boundedService;
    boolean isBound = false;

    private Button buttonGenerisiBroj;
    private TextView labelOdgovor;

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

    @Override
    protected void onStart() {
        super.onStart();
        Intent intent = new Intent(this ,
SlucajniBrojeviServis.class);
        //startService(intent); ovo bi kreiralo i
        startovani i vezani servis
        bindService(intent ,
boundServiceConnection,BIND_AUTO_CREATE);
    }

    private void initComponents(){
        buttonGenerisiBroj =
findViewById(R.id.buttonGenerisiBroj);
        labelOdgovor = findViewById(R.id.labelOdgovor);
        buttonGenerisiBroj.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        if(isBound){
            //ukoliko bi ovo bilo dugotrajna operacija
            //moralo bi u odvojenu nit
            int broj =
boundedService.generisiSlucajanBroj();
            labelOdgovor.setText(String.valueOf(broj));
        }
    }

    @Override
    protected void onStop() {

```

```

        super.onStop();
        if(isBound) {
            unbindService(boundServiceConnection);
            isBound = false;
            boundedService = null;
        }
    }

    private ServiceConnection boundServiceConnection = new
ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name,
IBinder service) {
        // Povezani smo sa lokalnim servisom, potrebno
je konvertovati IBinder u LokalniBinder
        // i dohvatiti instancu klase
        SlucajniBrojeviServis
        SlucajniBrojeviServis.LokalniBinder binder =
(SlucajniBrojeviServis.LokalniBinder) service;
        boundedService = binder.getService();
        //postaviti boolean indikator na true posto smo
povezani
        isBound = true;
    }

    @Override
    public void onServiceDisconnected(ComponentName
name) {
        isBound = false;
        boundedService= null;
    }
};

}

```

Sadržaj AndroidManifest.xml fajla dat je sledećim XML kodom:

```

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"

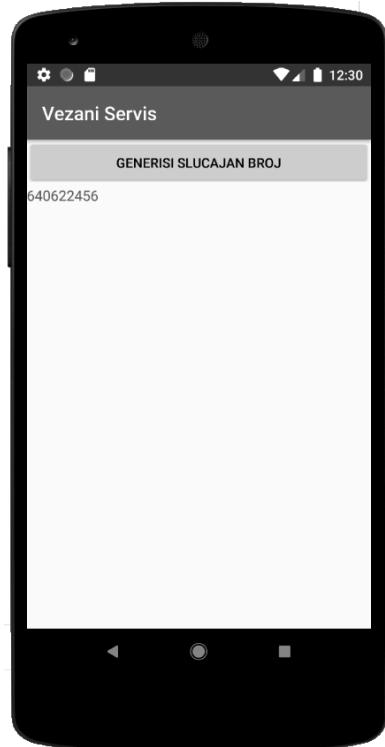
```

```
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity">
        <intent-filter>
            <action
    android:name="android.intent.action.MAIN" />

            <category
    android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <service android:name=".SlucajniBrojeviServis"/>
</application>

</manifest>
```

Prikaz rada aplikacije sa vezanim servisom generatora slučajnih brojeva dat je na slici 11.6. Klikom na dugme, poziva se metoda vezanog servisa (ukoliko je povezan, odnosno isBound nije false), koja vraća generisani slučajan broj. Ovaj broj se zatim postavlja na korisnički interfejs u tekstualno polje sa identifikatorom labelOdgovor.



Slika 11.6, primer izvršavanja vezanog servisa.

Primer 4: Potrebno je implementirati startovani servis koji obavlja dve funkcije. Prvo, nakon startovanja, svake sekunde šalje obaveštenje o broju proteklih sekundi od startovanja (realizovati kroz odvojenu radnu nit). Druga funkcija je da se servisu može proslediti poruka koju on kriptuje jednostavnom Cezarovom šifrom (svaki karakter se menja karakterom udaljenim 3 pozicije, tako da string **abc** postaje **def**) i vraća nazad. Servis se pokreće odgovarajućim dugmadima na korisničkom interfejsu. Na korisničkom interfejsu postoji i polje za unos poruke koju je potrebno kriptovati, kao i dugme kojim se poruka šalje servisu. Na korisničkom ekranu je potrebno prikazati i broj proteklih sekundi od pokretanja servisa. Komunikaciju između aktivnosti i servisa realizovati pomoću BroadcastReceiver komponenti.

Rešenje:

Traženi korisnički interfejs je realizovan u activity_main.xml fajlu, čiji je sadržaj dat u nastavku, a sam izgled korisničkog interfejsa je dat na slici 11.7.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.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"
        tools:context=".MainActivity">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <Button
            android:id="@+id/buttonPokreniServis"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Pokreni servis" />

        <Button
            android:id="@+id/buttonZaustaviServis"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Zaustavi servis" />

        <EditText
            android:id="@+id/inputPoruka"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="poruka..." />
    
```

```

<Button
    android:id="@+id/buttonPosalji"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Posalji poruku servisu" />
<TextView
    android:id="@+id/labelaBrojac"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textStyle="bold"
    android:textSize="26dp"/>
<TextView
    android:id="@+id/labelaPoruka"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:textStyle="bold"
    android:textSize="20dp"/>
</LinearLayout>
</android.support.constraint.ConstraintLayout>

```

Startovani servis je realizovan klasom SifraServis.java, čiji je sadržaj dat u nastavku:

```

package com.example.mzivkovic.myapplication;

import android.app.Service;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.IBinder;
import android.support.annotation.Nullable;
import android.widget.Toast;

public class SifraServis extends Service {
    //od servisa SifraServis ka MainActivity
    final static String KLJUC_INT_OD_SERVISA =
"KLJUC_INT_OD_SERVISA";
    final static String KLJUC_STRING_OD_SERVISA =
"KLJUC_STRING_OD_SERVISA";
    final static String AKCIJA_AZURIRAJ_BROJAC =
"AZURIRAJ_BROJAC";
    final static String AKCIJA_AZURIRAJ_PORUKU =
"AZURIRAJ_PORUKU";

    //od aktivnosti MainActivity ka servisu SifraServis
    final static String KLJUC_PORUKA_SERVISU =

```

```

"KLJUC_PORUKA_SERVISU";
    final static String AKCIJA_PORUKA_SERVISU =
"PORUKA_SERVISU";

    ServisReceiver servisReceiver;
    RadnaNitServisa radnaNitServisa;
    int brojac;

    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    @Override
    public void onCreate() {
        Toast.makeText(getApplicationContext(),
                "servis: onCreate",
        Toast.LENGTH_LONG).show();
        servisReceiver = new ServisReceiver();
        super.onCreate();
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int
startId) {
        Toast.makeText(getApplicationContext(),
                "servis: onStartCommand",
        Toast.LENGTH_LONG).show();

        IntentFilter intentFilter = new IntentFilter();
        intentFilter.addAction(AKCIJA_PORUKA_SERVISU);
        registerReceiver(servisReceiver, intentFilter);

        radnaNitServisa = new RadnaNitServisa();
        radnaNitServisa.start();

        return super.onStartCommand(intent, flags,
startId);
    }

    @Override
    public void onDestroy() {
        Toast.makeText(getApplicationContext(),
                "servis: onDestroy",
        Toast.LENGTH_LONG).show();
        radnaNitServisa.setRunning(false);
        unregisterReceiver(servisReceiver);
    }
}

```

```

        super.onDestroy();
    }

    public class ServisReceiver extends BroadcastReceiver {

        @Override
        public void onReceive(Context context, Intent intent) {

            String action = intent.getAction();
            if(action.equals(AKCIJA_PORUKA_SERVISU)) {
                String poruka =
intent.getStringExtra(KLJUC_PORUKA_SERVISU);
                String sifrovanaPoruka = "";
                for (int i = 0; i < poruka.length(); i++) {
                    char c = poruka.charAt(i);
                    //cezarova sifra, svaki karakter se
menja karakterom udaljenim 3 pozicije
                    int ci = (int)c;
                    ci += 3;
                    c = (char)ci;
                    sifrovanaPoruka += c;
                }
                //posalji nazad ka MainActivity
                Intent i = new Intent();
                i.setAction(AKCIJA_AZURIRAJ_PORUKU);
                i.putExtra(KLJUC_STRING_OD_SERVISA,
sifrovanaPoruka);
                sendBroadcast(i);
            }
        }
    }

    private class RadnaNitServisa extends Thread{

        private boolean running;

        public void setRunning(boolean running) {
            this.running = running;
        }

        @Override
        public void run() {
            brojac = 0;
            running = true;
            while (running) {
                try {
                    Thread.sleep(1000);

```

```
Intent intent = new Intent();

intent.setAction(AKCIJA_AZURIRAJ_BROJAC);
    intent.putExtra(KLJUC_INT_OD_SERVISA,
brojac);
        sendBroadcast(intent);
        brojac++;
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

Aktivnost je implementirana u klasi MainActivity.java, čiji je sadržaj dat sa:

```
package com.example.mzivkovic.myapplication;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity
implements View.OnClickListener {
    private Button buttonPokreniServis;
    private Button buttonZaustaviServis;
    private Button buttonPosaljiPoruku;
    private EditText inputPoruka;
    private TextView labelaBrojac, labelaPoruka;
    Intent mojIntent = null;
    AktivnostReceiver receiver;

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

        initComponents();
    }

    private void initComponents() {
```

```

        buttonPokreniServis =
findViewById(R.id.buttonPokreniServis);
        buttonZaustaviServis =
findViewById(R.id.buttonZaustaviServis);
        buttonPosaljiPoruku =
findViewById(R.id.buttonPosalji);
        inputPoruka = findViewById(R.id.inputPoruka);
        labelaBrojac = findViewById(R.id.labelaBrojac);
        labelaPoruka = findViewById(R.id.labelaPoruka);
        buttonPokreniServis.setOnClickListener(this);
        buttonZaustaviServis.setOnClickListener(this);
        buttonPosaljiPoruku.setOnClickListener(this);
    }

    private void startService(){
        mojIntent = new Intent(MainActivity.this,
SifraServis.class);
        startService(mojIntent);
    }

    private void stopService(){
        if(mojIntent != null){
            stopService(mojIntent);
        }
        mojIntent = null;
    }

    private void posaljiPoruku() {
        String porukaServisu =
inputPoruka.getText().toString();
        Intent intent = new Intent();

intent.setAction(SifraServis.AKCIJA_PORUKA_SERVISU);
        intent.putExtra(SifraServis.KLJUC_PORUKA_SERVISU,
porukaServisu);
        sendBroadcast(intent);
    }

    @Override
    protected void onStart() {
        receiver = new AktivnostReceiver();
        IntentFilter intentFilter = new IntentFilter();
intentFilter.addAction(SifraServis.AKCIJA_AZURIRAJ_BROJAC);
intentFilter.addAction(SifraServis.AKCIJA_AZURIRAJ_PORUKU);
        registerReceiver(receiver, intentFilter);
        super.onStart();
    }
}

```

```

@Override
protected void onStop() {
    unregisterReceiver(receiver);
    super.onStop();
}

@Override
protected void onDestroy() {
    super.onDestroy();
    stopService();
}

@Override
public void onClick(View v) {
    switch(v.getId()){
        case R.id.buttonPokreniServis:
            startService();
            break;
        case R.id.buttonZaustaviServis:
            stopService();
            break;
        case R.id.buttonPosalji:
            posaljiPoruku();
            break;
    }
}

private class AktivnostReceiver extends
BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent
intent) {
        String action = intent.getAction();

if(action.equals(SifraServis.AKCIJA_AZURIRAJ_BROJAC)) {
            int int_from_service =
intent.getIntExtra(SifraServis.KLJUC_INT_OD_SERVISA, 0);

labelaBrojac.setText(String.valueOf(int_from_service));
        }else
if(action.equals(SifraServis.AKCIJA_AZURIRAJ_PORUKU)) {
            String string_from_service =
intent.getStringExtra(SifraServis.KLJUC_STRING_OD_SERVISA);
            labelaPoruka.setText(String.valueOf(string_from_service));
        }
    }
}

```

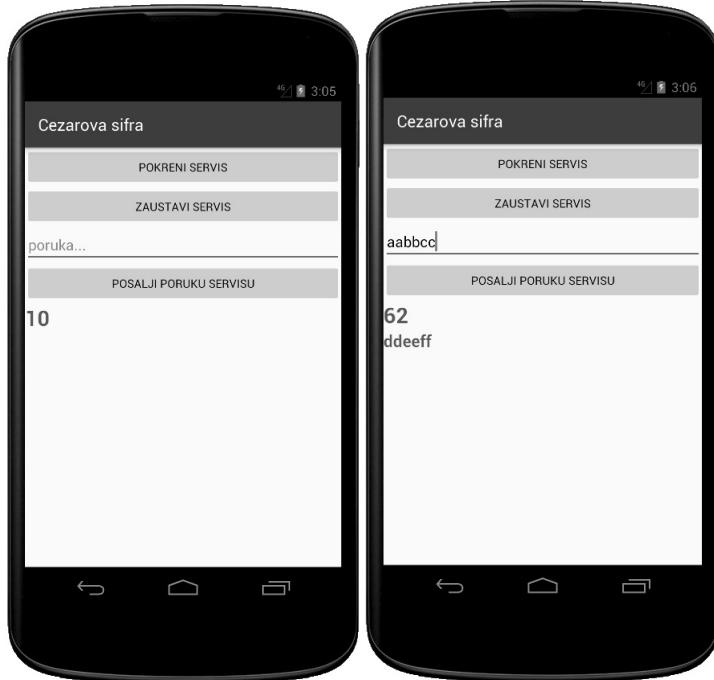
Na kraju, sadržaj AndroidManifest.xml fajla je dat sledećim isečkom XML koda.

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

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

                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name=".SifraServis"
            android:exported="false"/>
    </application>
</manifest>
```

Prikaz rada aplikacije dat je na slici 11.7. Sa leve strane prikazan je pokrenut servis (pri čemu se vidi brojač sekundi od pokretanja), dok je sa desne strane prikazana uneta poruka i odgovor servisa (šifrovana poruka **ddeeef** za unetu vrednost **aabbcc**). Servis se pokreće i zaustavlja odgovarajućim dugmićima pokreni servis i zaustavi servis.



Slika 11.7, prikaz rada servisa koji obavlja Cezarovu šifru.

12. Praktični primeri upotrebe

U ovom poglavlju su pokazani primeri upotrebe različitih šablonu koji se vrlo često koriste u različitim Android aplikacijama. Veoma je važno da programer zna da u kodu pravilno upotrebi kameru, lokaciju uređaja, senzore ili na primer Google Maps API. Postoje vrlo jasni šabloni kako se navedene komponente implementiraju u skladu sa preporukama Android zajednice. Upravo ovim šablonima je posvećen nastavak ovog poglavlja.

12.1. Kamera

Android programerima je dostupan Camera API, koji podržava različite kamere na uređaju i različite funkcionalnosti tih kamera, čime se omogućava da se u okviru aplikacije ubaci mogućnost snimanja fotografija ili video materijala. Polazna tačka svake aplikacije koja treba da omogući upotrebu kamere jeste razmatranje kako će ta aplikacija koristiti kameru. Na primer, moguće je u manifestu deklarisati zahtev da uređaj na kome se aplikacija instalira mora da ima kameru. Druga bitna tačka koju je potrebno razmotriti je način kako aplikacija treba da koristi kameru: da li je dovoljno da se napravi brza fotografija ili video snimak, ili je potrebno implementirati novi prilagođeni način upotrebe.

Pristup gde je potrebno razviti novu, prilagođenu funkcionalnost kamere je znatno obimniji, a implementacija nije baš najjednostavnija. Sa druge strane, moguće je koristiti aplikaciju koja već ima podršku za kameru upotrebom implicitne namere. Ono što je dobro je da je velikoj većini aplikacija sasvim dovoljno da upotrebe već postojeću Android Camera aplikaciju, a implementacija ovog pristupa je krajnje jednostavna i prikazana je u nastavku.

Raspored komponenti korisničkog interfejsa sadrži dugme koje pokreće implicitnu nameru, i ImageView objekat u koji će biti smeštena uhvaćena fotografija. Kod activity_main.xml fajla dat je u nastavku:

```
<?xml version="1.0" encoding="utf-8" ?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/buttonTakePicture"
```

```

        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Take a photo" />

<ImageView
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:id="@+id/capturedImage"/>
</LinearLayout>
```

Kod glavne aktivnosti je dat u klasi MainActivity.java, čiji je sadržaj dat u nastavku:

```

package com.example.student.camerabasic;

import android.content.Intent;
import android.graphics.Bitmap;
import android.provider.MediaStore;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.ImageView;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity
implements View.OnClickListener{

    private Button buttonCapture;
    private ImageView imgCapture;
    private static final int IMAGE_CAPTURE_CODE = 1;

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

        initComponents();
    }

    private void initComponents() {
        buttonCapture =
findViewById(R.id.buttonTakePicture);
        imgCapture = findViewById(R.id.capturedImage);
        buttonCapture.setOnClickListener(this);
    }
}
```

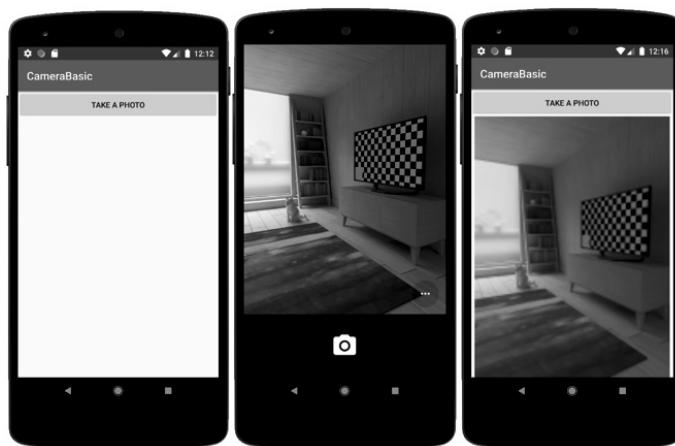
```

@Override
public void onClick(View v) {
    //implicitna namera, samo se opisuje akcija koju je
potrebno izvrsiti
    Intent intent = new
Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    startActivityForResult(intent, IMAGE_CAPTURE_CODE);
}

@Override
protected void onActivityResult(int requestCode, int
resultCode, Intent data) {
    if (requestCode == IMAGE_CAPTURE_CODE) {
        if (resultCode == RESULT_OK) {
            Bitmap bp = (Bitmap)
data.getExtras().get("data");
            imgCapture.setImageBitmap(bp);
        } else if (resultCode == RESULT_CANCELED) {
            Toast.makeText(this, "Cancelled",
Toast.LENGTH_LONG).show();
        }
    }
}

```

Nakon pokretanja, klikom na dugme Take a photo se pokreće Android aplikacija za kameru, koja će nakon snimanja fotografije vratiti sliku kroz callback metodu onActivityResult(), u okviru koje se slika smešta u ImageView polje za prikaz fotografije, kao što je prikazano na slici 12.1.



Slika 12.1, upotreba postojeće aplikacije za kameru upotrebom implicitne namere

12.2. Lokacija

Ukoliko je u aplikaciji potrebno koristiti lokaciju (samo koordinate, bez Google Maps), to se može veoma lako postići implementacijom LocationListener interfejsa. Ovaj interfejs nudi nekoliko metoda za implementaciju, od kojih je neophodno implementirati samo callback metodu onLocationChanged(), koju sistem poziva kada se promeni lokacija uređaja. Sistem kroz poziv ove metode dostavlja Location objekat, koji sadrži sve potrebne informacije o trenutnoj lokaciji. Location objekat sadrži veći broj parametara, od kojih su najbitniji geografska širina i dužina, nadmorska visina i lokacija, koji su ilustrovani u datom primeru. Za dohvatanje lokacije se koristi sistemski servis LocationManager, a u manifestu aplikacije je potrebno dodati odgovarajuće dozvole (lokacija korisnika se smatra za opasnu dozvolu i mora se eksplicitno dodeliti od strane korisnika):

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

Nakon dodavanja dozvola u manifestu, aktivnost koja treba da dohvata obaveštenja o promeni lokacije se implementira na sledeći način:

```
package com.example.mzivkovic.myapplication;  
  
import android.Manifest;  
import android.annotation.TargetApi;  
import android.app.Activity;  
import android.content.Context;  
import android.content.pm.PackageManager;  
import android.location.Location;  
import android.location.LocationListener;  
import android.location.LocationManager;  
import android.os.Build;  
import android.support.v4.app.ActivityCompat;  
import android.support.v4.content.ContextCompat;  
import android.support.v7.app.AppCompatActivity;  
import android.os.Bundle;  
import android.widget.TextView;  
  
public class MainActivity extends AppCompatActivity  
implements LocationListener {  
  
    private TextView labelInfo;
```

```

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

private void initComponents() {
    labelInfo = findViewById(R.id.labelInfo);
}

private void initGPS() {
    LocationManager lm = (LocationManager)
getSystemService(Context.LOCATION_SERVICE);
    if (ActivityCompat.checkSelfPermission(this,
            Manifest.permission.ACCESS_FINE_LOCATION)
!= PackageManager.PERMISSION_GRANTED &&
            ActivityCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_COARSE_LOCATION)
        !=
PackageManager.PERMISSION_GRANTED) {
        // TODO: Consider calling
        //      ActivityCompat#requestPermissions
        // here to request the missing permissions, and
        // then overriding
        //      public void onRequestPermissionsResult(int
requestCode, String[] permissions,
        //
int[] grantResults)
        // to handle the case where the user grants the
permission. See the documentation
        // for ActivityCompat#requestPermissions for
more details.
        return;
    }

    lm.requestLocationUpdates(LocationManager.GPS_PROVIDER,
5000, 10, this);
}

//ono sto nas zanima je locationChanged dogadjaj
//moramo prvo da aktiviramo servis za lokaciju,
// i tek onda da on obavestava povremeno nasu klasu
koja implementira dati metod
@Override
public void onLocationChanged(Location location) {
    double lat = location.getLatitude();
}

```

```

        double lon = location.getLongitude();
        double alt = location.getAltitude();
        double speed = location.getSpeed();

        labelInfo.setText("Lat = " + lat +
                          "\nLon = " + lon +
                          "\nAlt = " + alt +
                          "\nSpeed = " + speed);
    }

    @Override
    public void onStatusChanged(String provider, int
status, Bundle extras) {     }

    @Override
    public void onProviderEnabled(String provider) {     }

    @Override
    public void onProviderDisabled(String provider) {     }

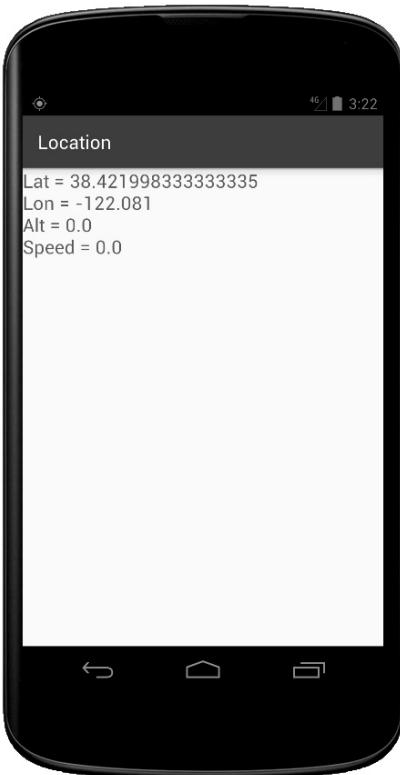
}

```

Nakon pokretanja aplikacije, ova aktivnost će prihvati obaveštenja o promeni lokacije na način koji je definisano u sledećem pozivu:

```
lm.requestLocationUpdates(LocationManager.GPS_PROVIDER,
5000, 10, this);
```

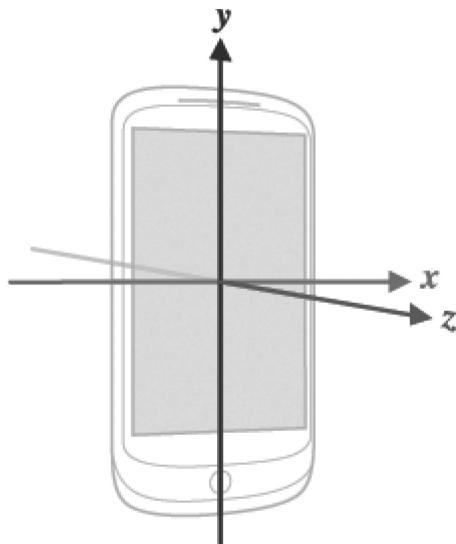
Drugi parametar ove metode označava minimalni interval ažuriranja lokacije, dok treći parametar označava minimalnu distancu koja se smatra za promenu lokacije (10m u navedenom primeru). Nakon pokretanja, aplikacija će početi da redovno prikazuje lokaciju, kao što je prikazano na slici 12.2.



Slika 12.2, primer dohvatanja lokacije uređaja

12.3. Senzori

Još jedna od specifičnih osobina Android platforme jeste dostupnost velikog broja senzora u okviru uređaja. Veliki broj uređaja ima dostupne senzore za pokret, orijentaciju, kao i različite uslove iz okruženja. Očitavanja sa ovih senzora moguće je koristiti i u okviru aplikacije. Programeri koriste senzore u različitim prilikama, na primer, mobilna igra može da inkorporira pokrete korisnika, rotaciju i slično. Sa druge strane, aplikacija za vremensku prognozu može da očitava temperaturu ili senzor za vlažnost. Svi ovi senzori su veoma precizni, a vraćaju sirova očitavanja u obliku SensorEvent objekta koji se sastoji iz niza vrednosti. Od tipa senzora zavisi koje vrednosti treba pročitati iz ovog niza. Na primer, senzor rotacije vraća tri vrednosti koje se mogu uzeti iz niza. Koordinatni sistem koji senzori pokreta i pozicije koriste je prikazan na slici 12.3, a važno je razumeti da se ose nikada ne pomeraju (x osa je uvek postavljena po širini uređaja, y osa po visini, a z osa izlazi normalno iz ekrana uređaja).



Slika 12.3, Koordinatni sistem koji koriste senzori pokreta na Android uređaju
(Slika preuzeta sa:

https://developer.android.com/guide/topics/sensors/sensors_overview.html)

Primer aplikacije koja koristi senzor rotacije dat je u nastavku:

```
package com.example.mzivkovic.myapplication;

import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity
implements SensorEventListener{
    private TextView labelInfo;
    // ziroskop, akcelerometar, vektor rotacije
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        initComponents();
        initSensor();
```

```

}

private void initComponents() {
    labelInfo = findViewById(R.id.labelInfo);
}

private void initSensor() {
    SensorManager sm = (SensorManager)
getSystemService(Context.SENSOR_SERVICE);
    //nakon referenciranja, pravimo konkretni senzor
    Sensor s =
sm.getDefaultSensor(Sensor.TYPE_ROTATION_VECTOR);
    sm.registerListener(this, s, 1000);
}

//ovaj koristimo, to je univerzalni event za sve
senzore, mi moramo da poznajemo koji senzor je u pitanju
kako bismo izvukli vrednosti
@Override
public void onSensorChanged(SensorEvent event) {

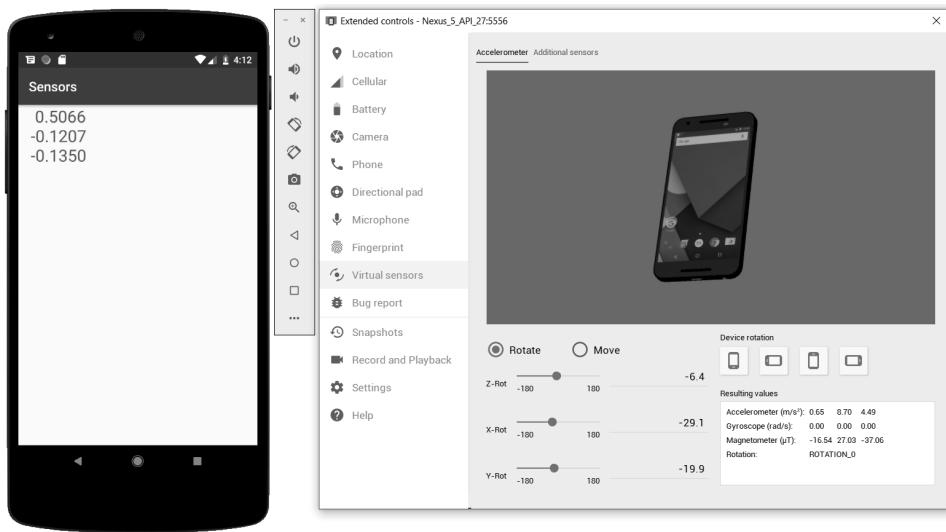
labelInfo.setText(String.format("%10.4f\n%10.4f\n%10.4f\n",
event.values[0], event.values[1], event.values[2]));
}

@Override
public void onAccuracyChanged(Sensor sensor, int
accuracy) {

}
}

```

Prikaz rada aplikacije dat je na slici 12.4. Emulator nudi mogućnost simuliranja pokreta i rotacije uređaja kroz dodatna podešavanja emulatora, što olakšava testiranje aplikacija koje rade sa senzorima pokreta.

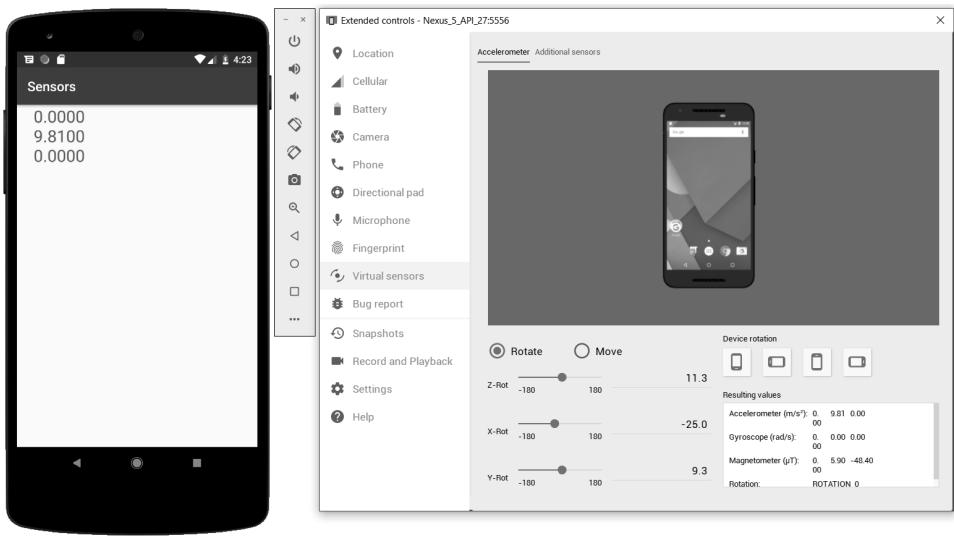


Slika 12.4, senzor rotacije uređaja

Kod za dohvatanje očitavanja akcelerometra je praktično identičan. Jedina razlika je što se dohvata drugi podrazumevani senzor kroz initSensor() metodu, kao što je prikazano u nastavku:

```
private void initSensor(){
    SensorManager sm = (SensorManager)
getSystemService(Context.SENSOR_SERVICE);
    //nakon referenciranja, pravimo konkretni senzor
    Sensor s =
sm.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
    sm.registerListener(this, s, 1000);
}
```

Primer rada aplikacije ukoliko se koristi akcelerometar umesto senzora rotacije, prikazan je na slici 12.5. Pošto je telefon u vertikalnom položaju, imamo samo ubrzanje gravitacije po y osi.



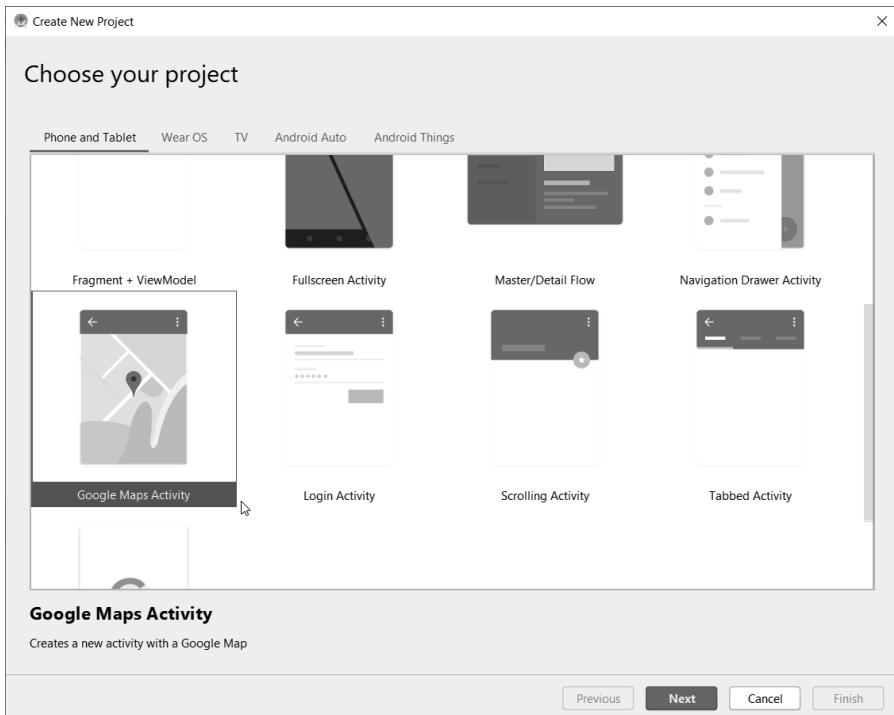
Slika 12.5, primer rada akcelerometra

12.4. Rad sa Google mapama

Maps SDK dozvoljava programerima da u aplikaciju dodaju mape bazirane na Google mapama. API direktno rukuje pristupom Google Maps serverima, prikazom mapa, kao i pokretima koje korisnik aplikacije vrši nad mapom. API omogućava postavljanje markera na mapu, iscrtavanje linijskih segmenta i poligona, kao i preklapanje grafičkih elemenata preko mape.

Android Studio omogućava veoma jednostavan način kreiranja Google Maps aktivnosti, koja je ponuđena kao šablon pri kreiranju nove aktivnosti (slika 12.6). Odabirom Google Maps aktivnosti projekat automatski nudi jednostavni primer koda, uz uputstva kako se aktivira Google Maps API. Generisani ključ se kopira i stavlja u google_maps_api.xml fajl. Detaljna uputstva o aktivaciji Google Maps API i generisanju ključa čitalac može pronaći na adresi:

<https://developers.google.com/maps/documentation/android-sdk/start>



Slika 12.6, kreiranje Google Maps aktivnosti

Aplikacija koja koristi mape mora imati dozvolu za pristup lokaciji korisnika i Internetu, odnosno u manifestu aplikacije se moraju zatražiti dozvole za android.permission.INTERNET (omogućava pristup Internetu) i android.permission.ACCESS_FINE_LOCATION (pristup lokaciji). Aktivnost koja koristi mape treba da implementira nekoliko interfejsa kako bi se omogućila puna funkcionalnost API-ja. Primer upotrebe Google Maps API-ja ilustrovan je dohvatanjem trenutne lokacije korisnika i postavljanjem markera sa informacijama o trenutnoj lokaciji na mapi:

```
public class MapsActivity extends FragmentActivity
implements OnMapReadyCallback,
        GoogleApiClient.ConnectionCallbacks,
        GoogleApiClient.OnConnectionFailedListener,
        LocationListener {

    public static final int ZAHTEV_DOZVOLA_LOKACIJA = 50;
    GoogleApiClient apiClient;
    Location poslednjaLokacija;
    Marker markerTrenutneLokacije;
    LocationRequest locationRequest;
    private GoogleMap mMap;
```

```

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

    //ukoliko je Android verzija >= Marshmallow mora se
    proveriti
    //da li aplikacija ima potrebnu dozvolu dinamicki
    if (android.os.Build.VERSION.SDK_INT >=
Build.VERSION_CODES.M) {
        checkLocationPermission();
    }
    SupportMapFragment mapFragment =
(SupportMapFragment)
        getSupportFragmentManager()
            .findFragmentById(R.id.map);

    mapFragment.getMapAsync(this);
}

//OnMapReadyCallback: ova metoda se poziva kada je mapa
sprema za upotrebu
@Override
public void onMapReady(GoogleMap googleMap) {
    mMap = googleMap;
    mMap.setMapType(GoogleMap.MAP_TYPE_NORMAL);
    //Postoje i druge opcije za prikaz mape, poput
    satelita,
    //terena, ili hibridnog prikaza
    //mMap.setMapType(GoogleMap.MAP_TYPE_SATELLITE);
    //mMap.setMapType(GoogleMap.MAP_TYPE_TERRAIN);
    //mMap.setMapType(GoogleMap.MAP_TYPE_HYBRID);
    //aktivacija dodatnih kontrola
    mMap.getUiSettings().setZoomControlsEnabled(true);
    mMap.getUiSettings().setZoomGesturesEnabled(true);
    mMap.getUiSettings().setCompassEnabled(true);
    //Inicijalizacija Google Play Servisa
    if (android.os.Build.VERSION.SDK_INT >=
Build.VERSION_CODES.M) {
        if (ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION)
            == PackageManager.PERMISSION_GRANTED) {
            initializeGoogleApiClient();
            mMap.setMyLocationEnabled(true);
        }
    } else {
        initializeGoogleApiClient();
    }
}

```

```

        mMap.setMyLocationEnabled(true);
    }
}

//ova metoda sluzi za inicializaciju Google Play
Servisa
protected synchronized void initializeGoogleApiClient()
{
    // Builder se koristi kao pomoc u konstrukciji
    GoogleApiClient objekta
    // poziv metode addApi () specificira koji API-ji
    su potrebni aplikaciji
    apiClient = new GoogleApiClient.Builder(this)
        .addConnectionCallbacks(this)
        .addOnConnectionFailedListener(this)
        .addApi(LocationServices.API)
        .build();
    //Klijent mora biti konektovan pre izvrsavanja bilo
    koje operacije
    apiClient.connect();
}
//GoogleApiClient.ConnectionCallbacks: ova callback
// metoda se poziva svaki put kada se uredjaj poveze
ili izgubi konekciju
@Override
public void onConnected(Bundle bundle) {
    locationRequest = new LocationRequest();
    locationRequest.setInterval(1000);
    locationRequest.setFastestInterval(1000);
    locationRequest.setPriority(LocationRequest.
PRIORITY_BALANCED_POWER_ACCURACY);
    if (ContextCompat.checkSelfPermission(this,
        Manifest.permission.ACCESS_FINE_LOCATION)
        == PackageManager.PERMISSION_GRANTED) {

LocationServices.FusedLocationApi.requestLocationUpdates(
apiClient, locationRequest, this);
    }
}
@Override
public void onConnectionSuspended(int i) {
}
//Ova callback metoda se poziva kada se promeni
lokacija uredjaja
@Override
public void onLocationChanged(Location location) {
    poslednjaLokacija = location;
    if (markerTrenutneLokacije != null) {

```

```

        markerTrenutneLokacije.remove();
    }
    //Prikaz markera trenutne lokacije na mapi
    LatLng latLng = new LatLng(location.getLatitude(),
location.getLongitude());
    MarkerOptions markerOptions = new MarkerOptions();
    markerOptions.position(latLng);
    LocationManager locationManager = (LocationManager)
        getSystemService(Context.LOCATION_SERVICE);
    String provider =
locationManager.getBestProvider(new Criteria(), true);
    if (ActivityCompat.checkSelfPermission(this,
        Manifest.permission.ACCESS_FINE_LOCATION)
!= PackageManager.PERMISSION_GRANTED &&
        ActivityCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_COARSE_LOCATION)
        !=
PackageManager.PERMISSION_GRANTED) {
        return;
    }
    Location locations =
locationManager.getLastKnownLocation(provider);
    List<String> providerList =
locationManager.getAllProviders();
    if (locations != null && providerList != null &&
providerList.size() > 0) {
        double longitude = locations.getLongitude();
        double latitude = locations.getLatitude();
        Geocoder geocoder = new
Geocoder(getApplicationContext(),
            Locale.getDefault());
        try {
            List<Address> listAddresses =
geocoder.getFromLocation(latitude,
                longitude, 1);
            if (null != listAddresses &&
listAddresses.size() > 0) {
                //dohvatanje informacija o lokaciji i
njihovo postavljanje na marker
                String drzava =
listAddresses.get(0).getCountryName();
                String grad =
listAddresses.get(0).getLocality();
                String adresa =
listAddresses.get(0).getAddressLine();
                markerOptions.title("'" + latLng + "','" +
adresa + "," + grad + "," + drzava);
            }
        }
    }
}

```

```

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

markerOptions.icon(BitmapDescriptorFactory.defaultMarker
(BitmapDescriptorFactory.HUE_BLUE));
    markerTrenutneLokacije =
mMap.addMarker(markerOptions);

mMap.moveCamera(CameraUpdateFactory.newLatLng(latLng));
    mMap.animateCamera(CameraUpdateFactory.zoomTo(11));
    if (apiClient != null) {
        LocationServices.FusedLocationApi.
removeLocationUpdates(apiClient, this);
    }
}

@Override
public void onConnectionFailed(ConnectionString
connectionResult) {
}
public boolean checkLocationPermission() {
    if (ContextCompat.checkSelfPermission(this,
            Manifest.permission.ACCESS_FINE_LOCATION)
        != PackageManager.PERMISSION_GRANTED) {
        if
(ActivityCompat.shouldShowRequestPermissionRationale(this,
Manifest.permission.ACCESS_FINE_LOCATION)) {
            ActivityCompat.requestPermissions(this,
                new
String[] {Manifest.permission.ACCESS_FINE_LOCATION},
                ZAHTEV_DOZVOLA_LOKACIJA);
        } else {
            ActivityCompat.requestPermissions(this,
                new
String[] {Manifest.permission.ACCESS_FINE_LOCATION},
                ZAHTEV_DOZVOLA_LOKACIJA);
        }
        return false;
    } else {
        return true;
    }
}
@Override
public void onRequestPermissionsResult(int requestCode,
    String permissions[], int[]

```

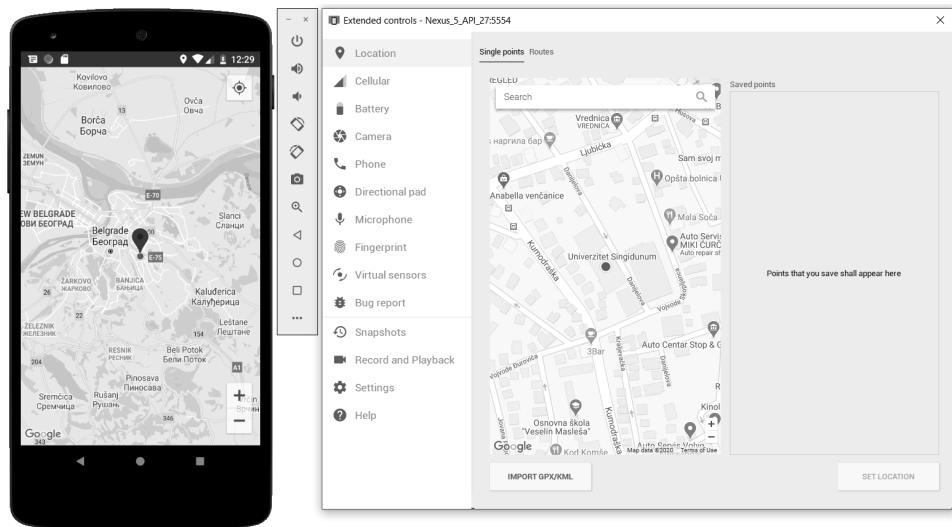
```

grantResults) {
    switch (requestCode) {
        case ZAHTEV_DOZVOLA_LOKACIJA: {
            if (grantResults.length > 0
                && grantResults[0] ==
PackageManager.PERMISSION_GRANTED) {
                if
(ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION)
==

PackageManager.PERMISSION_GRANTED) {
                    if (apiClient == null) {
                        initializeGoogleApiClient();
                    }
                    mMap.setMyLocationEnabled(true);
                }
            } else {
                Toast.makeText(this, "permission
denied",
Toast.LENGTH_LONG).show();
            }
            return;
        }
    }
}

```

Nakon pokretanja date aplikacije, prikazaće se plavi marker sa trenutnom lokacijom korisnika na mapi, kao što je prikazano na slici 12.7. Kroz dodatna podešavanja emulatora moguće je postaviti proizvoljnu lokaciju, a čak i simulirati uslove vožnje ili kretanja.



Slika 12.7, aplikacija koja postavlja marker trenutne lokacije na mapi

Kada se klikne na prikazani marker, pojavljuju se i dodatne informacije o lokaciji, kao što je prikazano na slici 12.8.



Slika 12.8, prikaz markera na mapi sa dodatnim informacijama o lokaciji

Literatura

1. Horton J., *Android Programming for Beginners*, Second Edition, Packt Publishing, 2018.
2. Smyth N. *Android Studio 3.5 Development Essentials - Java Edition: Developing Android 10 (Q) Apps Using Android Studio 3.5, Java and Android Jetpack*, Payload Media, 2019.
3. Darwin I. F., *Android Cookbook: Problems and Solutions for Android Developers*, Second Edition, O'Reilly Media, Inc., 2017.
4. Griffiths Dawn, Griffiths David, *Head First Android Development: A Brain – Friendly Guide*, Second Edition, O'Reilly Media, Inc., 2017.
5. Phillips B., Stewart C., Marsicano K., *Android Programming: The Big Nerd Ranch Guide*, Third Edition, Big Nerd Ranch Guides, 2017.
6. Boyer R., *Android 9 Development Cookbook: Over 100 recipes and solutions to solve the most common problems faced by Android developers*, Third Edition, Packt Publishing Limited, 2018.
7. Shuller M., *Android Programming: Mastering Course for Beginners - Quick Start to Develop Your Own App (Android studio, Android Development, App Development. Updated to Android 6 Platform Book 1)*, Kindle Edition.
8. Zapata B. C., *Android Studio Essentials*, First Edition, Packt Publishing, 2015.
9. Zapata B. C., *Android Studio Application Development*, First Edition, Packt Publishing, 2013.
10. Aliferi C., *Android Programming Cookbook, Kick-start your Android projects*, JCG, Exelixis Media P.C., 2016.
11. Kothari P., *Android Application Development (With Kitkat Support)*, Black Book, Dreamtech Press, 2014.
12. Burd B., *Android Application Development All-In-One for Dummies*, Second Edition, Wiley, 2015.
13. Živković D., *Osnove Java programiranja*, 8. izdanje, Univerzitet Singidunum, 2017.
14. Živković D., *Java programiranje*, 2. izdanje, Univerzitet Singidunum, 2013.
15. Cornez T., Cornez R., *Android Programming Concepts*, Jones & Bartlett Publishers, 2015.
16. Burnette E., *Hello Android: Introducing Google's Mobile Development Platform*, Fourth Edition, Shroff/Pragmatic Bookshelf, 2015.
17. Android developers: <https://developer.android.com/>
18. Android developers blog: <https://android-developers.googleblog.com/>
19. Mobile App Download and Usage Statistics (2019): <https://buildfire.com/app-statistics/>
20. Statista: <https://www.statista.com/>

21. Friesen J., Android Studio for beginners, JavaWorld (2020):
<https://www.javaworld.com/article/3095406/android-studio-for-beginners-part-1-installation-and-setup.html>
22. Android Room with a View – Java:
<https://codelabs.developers.google.com/codelabs/android-room-with-a-view/index.html#0>
23. Codelabs for Android Developer Fundamentals:
<https://developer.android.com/courses/fundamentals-training/toc-v2>
24. Vogella tutorials: <https://www.vogella.com/tutorials/android.html>
25. JournalDev tutorials: <https://www.journaldev.com/android>
26. TutorialsPoint: <https://www.tutorialspoint.com/android/index.htm>
27. Tair M., video materijali:
<http://zadatak.singidunum.ac.rs/videos/android/index.php>

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.451.9 ANDROID

004.42:004.451

ЖИВКОВИЋ, Миодраг, 1982-

Razvoj mobilnih aplikacija / Miodrag Živković. - 1. izd. - Beograd : Univerzitet

Singidunum, 2020 (Beograd : Caligraph). - XIV, 370 str. : ilustr. ; 24 cm

Tiraž 700. - Bibliografija: str. 369-370.

ISBN 978-86-7912-719-8

a) Оперативни систем "Android" -- Програмирање

COBISS.SR-ID 283224076

© 2020.

Sva prava zadržana. Nijedan deo ove publikacije ne može biti reproducovan u bilo kom vidu i putem bilo kog medija, u delovima ili celini bez prethodne pismene saglasnosti izdavača.



Miodrag Živković

RAZVOJ MOBILNIH APLIKACIJA

ANDROID JAVA PROGRAMIRANJE

Programi napisani u Javi se mogu pronaći svuda, od Android mobilnih uređaja, preko klasičnih desktop aplikacija, pa sve do Internet strana i web servisa. Zbog sve veće popularnosti mobilnih aplikacija i dostupnosti mobilnih uređaja, jedno od najtraženijih zanimanja danas je upravo razvoj mobilnih aplikacija za Android platformu, što predstavlja fokus ovog udžbenika. Mobilni operativni sistemi poput Androida su danas dostupni ne samo za mobilne telefone, već i za tablete, kao i čitav niz drugih uređaja, poput pametnih satova i televizora. Samim tim je i tržiste na koje se mogu plasirati Android aplikacije značajno povećano. Mobilne aplikacije imaju veći broj prednosti u odnosu na tradicionalne desktop aplikacije. Ovakav tip aplikacija je uvek uz korisnika, pošto mobilni telefon nosimo uvek sa sobom. Mobilni telefon nudi još dodatne mogućnosti u obliku pristupa Internetu, velikog broja senzora (poput akcelerometra ili kompasa), pristupa mapama, mogućnosti za pozicioniranje preko GPS i kamere, a veliki broj dostupnih aplikacija je besplatan. Programiranje za Android zahteva dobro poznavanje koncepta nasleđivanja, apstraktnih klasa i interfejsa, jer je većina standardnih Android komponenti realizovana na ovaj način, odnosno od programera se zahteva da proširi odgovarajuće standardne Android klase i da implementira već postojeće interfejsе. Dodatno, potrebno je imati u vidu i ograničenja na koja ranije nije obraćano previše pažnje. Na primer, veličina ekrana za prikaz je veoma ograničena i drastično je manja od veličine standardnih desktop monitora. Pošto je telefon mobilni uređaj, mora se obratiti pažnja i na ograničen kapacitet baterije i na ograničenu procesorsku moć, koja je višestruko manja od modernih računara.