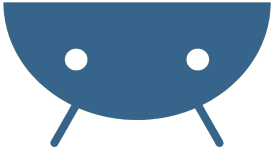


Izrada mobilnih aplikacija pomoću Android Studia

Seminarski Rad,

Joško Križanović,

Split, studeni 2022.

Sadržaj

Poglavlje 1

Uvod

Mobitel je danas jedan od najzastupljenijih uređaja kojeg ljudi nose svugdje. To nije čudno jer umjesto da sa sobom nosimo sat, fotoapart, kalendar, budilicu, kućni telefon, štopericu, kartu, kalkulator,... možemo samo ponijet mobitel! U svijetu gdje operacijski sustav Android obuhvaća više od 2.5 bilijuna mobitela i tableta zgodno i korisno je znati raditi aplikacije za njega.

U ovom seminaru nastojim vas čitatelje naučiti kako programirati jednostavne aplikacije u Android Studiu.
Od vas očekujem da imate neko jednostavno predznanje o programiranju te da znate programirati bar u jednom jeziku tj. u bilo kojem jeziku. To znači da znate neke općenite pojmove kao što su deklaracije, zadavanje vrijednosti, funkcije itd. Da sam morao detaljno objasniti svaki programerski pojam zajedno sa objašnjenjem sintakse programskog jezika Kotlina ovaj seminar bi bio moj magistarski rad.

1.1 Što je Android?

Android je operacijski sustav razvijen za mobitele i tablete, a poslije opremljen za rad s pametnim televizijama (Android TV), autima (Android Auto) i pametnim satovima (Wear OS).

Android je razvila Američka kompanija *Android Inc.* osnovana u listopadu 2003. kojoj je namjera stvaranja Androida bila da stvori napredni operacijski sustav digitalnih kamera. Vidjevši da im je tržište za digitalne kamera premaleno preusmjerili su razvoj Androida kao operacijskog sustava za mobitele. 2005. godine *Google* kupuje *Android Inc.* te 2007. godine predstavlja javnosti operacijski sustav Android koji će biti instaliran na prve mobitele u rujnu 2008. godine.

Napisan u *C/C++* jeziku, a aplikacije za njega su napisane ili u *Javi* ili u novijem programskom jeziku *Kotlin*. U ovom seminaru ću obraditi razvoj aplikacija s *Kotlinom* jer *Google* preporučava razvoj aplikacija s njim.

1.2 Što je Android Studio i kako ga instalirati?

Android Studio je integrirano razvojno okurženje *(engl. Integrated Development Environment)*stvoreno za razvoj aplikacija za Android sučelje.

Da možete pratit zajedno sa mnom razvoj jednostavnih aplikacija trebat ćete ga instalirati. Na poveznici https://developer.android.com/studio naći ćete sve potrebne datoteke za instalirati Android Studio. Za Windows dobit ćete datoteku sa nastavkom *.exe*, pokrenite je i pratite upute za instalaciju koje vam ona daje. Ako se niste snašli ili imate drugi operacijski sustav na poveznici https://developer.android.com/studio/install naći ćete dodatne upute za instalaciju Android Studija.

Poglavlje 2

Kotlin

Kotlin je statički pisan, višeplatformski programski jezik opće namjene. *Kotlin* je dizajniran tako da je međuoperativan s *Javom* te može koristit biblioteke i kod napisan u*Javi*. Napravljen je 2011. a proglašen kao preferirani programski jezik za razvijanje Android aplikacija 2019. Mi ćemo ga iz tog razloga korisiti za razvoj naših aplikacija, te ću vas uvesti u njegovu sintaksu. Ne morate pratiti sa mnom na računalu jer ću samo objasniti sintaksu i prikazati jednostavne primjere.

2.1 Ulazna točka Kotlin programa

Ulazna točka *Kotlin* programa je funkcija `main()`:

```
fun main(){
    println("Pozdrav svijete!")
}
```

`main()` može primat argumente od korisnika putem niza stringova:

```
/* hello.kt */
fun main(args: Array<String>){
    for( string in args )
        println("Pozdrav ${string}")
}
```

```
$ kotlinc hello.kt -include-runtime -d hello.jar
$ java -jar hello.jar Josko Valentino
Pozdrav Josko
Pozdrav Valentino
```

Android aplikacije ne koriste `main()` kao ulaznu točku nego `MainActivity` koja nije funkcija nego klasa `Activity` koja će biti bolje objašnjena u sljedećem poglavlju.

2.2 Varijable

Dvije ključne riječi za definiranje varijabli su `val` i `var`:
`val` definira varijablu koja se ne može mijenjati nakon što se prvi put inicijalizira:

```
/* kotlin.kt */
fun main(){
    /* Definiramo varijablu string tipa String koja
    je samo za čitanje i incijaliziramo je s vrijednošću
    "Kotlin" */
    val string = "Kotlin"
    /* Sljedeća linija koda nije moguća te će pri
    kompilaciji biti izbačena pogreška */
    string = "Java"
}
```

```
$ kotlinc kotlin.kt -include-runtime -d kotlin.jar
kotlin.kt:3:2: error: val cannot be reassigned
        string = "Java"
 ^
```

`var` definira varijablu kojoj se može mijenjati vrijednost:

```
/* Sada istu varijablu definiramo uz
nastavak 'var' */
var string = "Kotlin"
/* Možemo mijenjati vrijednost string u "Java" */
string = "Java"
```

Za razliku od *Jave*, *Kotlinu* nije potrebno eksplicitno navesti tip varijable, on svoj tip prepozna pomoću vrijednosti koju mu zadamo:

```
/* Kotlin automatski prepozna da se radi o tipu varijable
String. */
val string = "Kotlin"
```

Da eksplicitno navedemo njegov tip moramo staviti ”:” ispred imena varijable i navest njegov tip:

```
/* Eksplicitno kažemo Kotlinu da se radi o varijabli tipa
String */
val string: String = "Kotlin"
```

S ključnom riječ `lateinit` možemo incijalizirati varijablu nakon što smo je definirali:

```
/* Definiramo varijablu imena "string" tipa String koja
je samo za čitanje, pošto je ne incijaliziramo moramo
eksplicitno napisati tip i modifikator lateinit */
lateinit val string: String
/* Incijaliziramo je s vrijednošću "Java" */
string = "Java"
```

Obični tipovi u *Kotlinu* su `Integer` za cijelobrojne brojeve, `Float` i `Double` za racionalne brojeve, `Char` za spremiti jedno slovo u jednom bajtu, `Boolean` za spremanje jednobitne vrijednosti *True* ili *False*, `String` za spremanje teksta i `Array` za spremanje nizova drugih tipova.

```
val broj: Integer = 100
// Double za spremanje decimalnih brojeva
val dec: Double = 54.2
// Char za spremanje jednog slova
val newline: Char = "\n"
// Boolean za spremanje vrijednosti true ili false
val jeKotlin: Boolean = True
// String za spremanje teksta
val grad: String = "Makarska"
// Niz stringova "Kotlin", "Java" i "C"
val strNiz: Array = Array(3) {"Kotlin", "Java", "C"}
```

2.3 Funkcije

Sintaksa funkcije u *Kotlin* je veoma slična kao u drugim programskim jezicima.

S ključnom riječi `fun` definiramo funkciju s izabranim imenom. Ne smijemo imenovati funkciju s imenom nekih ključnih riječi koje *Kotlin* koristi, to vrijedi i za sve stvari koje se moraju definirati s nekim imenom. Između zagrada ne trebamo ali možemo definirati jedan ili više argumenata koje moramo dati funkciji kada je pozovemo. Nakon zagrada ako stavimo ”:” možemo definirati tip vrijednosti koji će funkcija vratiti. Ako to polje ostavimo prazno, funkcija neće vraćati vrijednost. S ključnom riječi `return` vraćamo vrijednost tipa kojeg smo zadali da će funkcija vratiti i vraćamo se u funkciju iz kojeg je naša funkcija pozvana. Ako nemamo tip kojeg moramo vratiti `return` nas samo vraća u funkciju iz kojeg je naša funkcija pozvana.

```
/* kotlin.kt */
fun zbroj(a: Int, b: Int ): Int { /*
~~~ ~~~~~  ~~~~~~   ~~~~~  ~~~~~
 ^     ^       ^       ^      ^
Ključna|       |       |      |
riječ  |       |       |      |
fun    |       |       |      |
  Ime funkcije |       |      |
               |       |      |
Prva varijabla tipa    |      |
Int                    |      |
    Druga varijabla tipa      |
    Int                       |
            Tip koji će funkcija vratiti
*/
    // Funkcija će vratiti zbroj brojeva a i b
    return a + b
}

fun main() {
    val a = 2
    val b = 2
    val sum = zbroj( a, b )
    println(${a} + ${b} = ${sum})
}
```

```
$ kotlinc kotlin.kt -include-runtime -d kotlin.jar
$ java -jar kotlin.jar
2 + 2 = 4
```

2.4 Klase

Za definiciju klase koristimo ključnu riječ `class`- Klase se definiraju slično kao i funkcije, jedina razlika je što možemo definirati funkcije i unutarnje klase unutar klase. Modifikatori `private` i `public` nam mogu sakriti varijable, funkcije ili klase koje ne želimo da korisnik koda dira:

```
/* kotlin.kt */
import java.time.LocalDateTime

class Pravac(var p0: Double, var p: Double) { /*
~~~~~ ~~~~~~ ~~~~~~~~~~~~~~~  ~~~~~~~~~~~~~
 ^        ^              ^                ^
ključna   |              |                |
riječ     |              |                |
        ime klase        |                |
        prvi argument klase         drugi argument klase */
    private val vrjemeKreacije = LocalDateTime.now()
    public var lp = Math.abs(p-p0)
    public fun printKoord() {
        println("p0,p(${p0},␣${p})")
    }
    public fun printVrijemeKreacije(){
        println("${vrijemeKreacije}s")
    }
}

fun main() {
    val pravac = Pravac( -2.0, 4.0)
    pravac.printKoord()
    println("Duzina␣pravca␣je␣${pravac.lp}")
    pravac,printVrijemeKreacije()
}
```

```
$ kotlinc kotlin.kt -include-runtime -d kotlin.jar
$ java -jar kotlin.jar
p0,p(-2.0, 4.0)
Duzina pravca je 6.0
2022-11-10T07:26:24.276910s
```

Klase mogu naslijediti svojstva od drugih klasa, te ćemo s klasom `Pravac` preko nasljedstva napraviti klasu `Vektor`. Modifikator `open` nam omogućuje da pomoću modifikatora `override` možemo napisati našu vrijednost, proceduru itd. za izabranu varijablu, funkciju itd. s `open` modifikatorom. Nasljedstvo pišemo tako da nakon zagrada napišemo ":" i dajemo ime klase koju ćemo naslijediti. Argumenti nasljedne klase će biti dani putem argumenata naše nove klase.

```
/* kotlin.kt */
import java.time.LocalDateTime
/* Dodali smo open modifikator da je možemo naslijediti */
open class Pravac(open var p0: Double, open var p: Double) {
    private val vrjemeKreacije = LocalDateTime.now()
    public var lp = Math.abs(p-p0)
    open public fun printKoord() {
        println("p0,p(${p0},␣${p})")
    }
}
/* klasa Vektor koja nasljeđuje svojstva od klase Pravac */
class Vektor(var x0: Double, var y0: Double,
             var x:  Double, var y:  Double): Pravac(x0, x){

    public var lx = lp
    public var ly = Math.abs(y-y0)
    public var lxy =
    Math.sqrt(Math.pow(lx,2.0)+Math.pow(ly,2.0))
    override fun printKoord() { // nova definicija printKoord()
        println("x0,y0(${x0},␣${y0})")
        println("x,y(${x},␣${y})")
    }
}

fun main() {
    val vektor = Vektor( 0.0, 0.0, 1.0, 1.0)
    vektor.printKoord()
    println("Duzina␣vektora␣je␣${vektor.lxy}")
}
```

```
$ kotlinc kotlin.kt -include-runtime -d kotlin.jar
$ java -jar kotlin.jar
x0,y0(0.0, 0.0)
x,y(1.0, 1.0)
Duzina vektora je 1.4142135623730951
```

2.5 Komentari

Postoje dvije vrste komentara: *jednolinijski* i *višelinijski* komentari.

```
// Jednolinijski komentar

/*
    Višelinijski
    komentar
    vooah
*/
```

2.6 Grananje

Za grananje koristimo ključne riječi `if`, `else` i `else if`. Uvjet se stavlja između zagrada te se između vitičastih zagrada stavlja blok koda koji će se izvršiti ako je uvjet ispunjen.

```
fun main(argumenti: Array<String>) {

    // Ako nismo dobili dovoljno argumenata od korisnika
    // završavamo program
    if( argumenti.size < 1 ) {
                return
    }
    // Ovisno o tekstu koji smo dobili granamo
    // u blok koda kojem je uvjet prvi ispunjen.
    if( argumenti.get(0) == "Mjau" ){

        println("Macka!")

    } else if( argumenti.get(0) == "Woof") {

        println("Pas!")

    } else {

        println("Sta?")

    }

}
```

Ako unutar vitičatih zagrada `if`, `else` ili `else if` izjava se nalazi samo jedna linija koda, vitčaste zagrade se mogu ukloniti bez da se *Kotlin* pobuni.

```
/* kotlin.kt */
fun main(argumenti: Array<String>) {

    if( argumenti.size < 1 )
                return

    if( argumenti.get(0) == "Mjau" )
        println("Macka!")
    else if( argumenti.get(0) == "Woof")
        println("Pas!")
    else
        println("Sta?")

}
```

```
$ kotlinc kotlin.kt -include-runtime -d kotlin.jar
$ java -jar kotlin.jar Mjau
Macka!
$ java -jar kotlin.jar Woof
Pas!
$ java -jar kotlin.jar Mjof
Sta?
```

Grana `if`, `else` i `else if` iz prijašnjeg koda se može zamijeniti s jednim pozivom izjave `when`.

```
when(argumenti.get(0)){
    "Mjau" -> {
        println("Macka!")
    }
    "Woof" -> {
        println("Pas!")
    }
    else -> {
        println("Sta?")
    }
}
```

Kao i u `if`, `else` i `else if` izjavama možemo izbaciti vitičaste zagrade ako imamo samo jednu liniju koda unutar vitičastih zagrada.

```
when(argumenti.get(0)){
    "Mjau" -> println("Macka!")
    "Woof" -> println("Pas!")
    else   -> println("Sta?")
}
```

2.7 Operatori

Ako ste ikad koristili programski jezik C, već ste upoznali skoro sve operatore koje *Kotlin* koristi.

Kotlin sadrži matematičke operatore '+' (zbrajanja), '-' (oduzimanja), '*" (množenja), "/" (dijeljenje), "%" (cjelobrojni ostatak dijeljenja):

```
/* kotlin.kt */
fun main() {

    val a = 10
    val b = 5
    println("zbrajanje  a + b = ${a+b}")
    println("oduzimanje a - b = ${a-b}")
    println("mnozenje   a * b = ${a*b}")
    println("djeljenje  a / b = ${a/b}")
    println("modul      a % b = ${a%b}")

}
```

```
$ kotlinc kotlin.kt -include-runtime -d kotlin.jar
$ java -jar kotlin.jar
zbrajanje  a + b = 15
oduzimanje a - b = 5
mnozenje   a * b = 50
djeljenje  a / b = 2
modul      a % b = 0
```

Za dodjeljivanje vrijednosti varijablama koristimo operator "=". Kod dojeljivanja vrijednosti s desne strane se upisuje u varijablu s lijeve strane operatora. Za dodjeljivanje vrijednosti možemo još koristiti augmentirane operatore dodjeljivanja. Augmentirani operatori dodjelivanja uzmu varijablu s lijeve strane operatora te ovisno o tipu augmentiranog operatora nad varijablom s vrši matematička operacija sa vrijednošću s desne strane operatora i rezultat operacije se zapisuje natrag u uzetu varijablu. Tipovi augmentiranih operatora zadavanja su: "+=", "-=", "*=", "/=" i "%=".

```
/* kotlin.kt */
fun main() {

    val a = 10
    val b = 5
    var c0 = 0

    // Ovo
    c0 = a
    c0 = c0 * b
    println("c0 = c0 * b   -> ${c0}")
    // i ovo su iste operacije ali drugačije zapisane
    c0 = a
    c0 *= b
    println("c0 *= b       -> ${c0}")
}
```

```
$ kotlinc kotlin.kt -include-runtime -d kotlin.jar
$ java -jar kotlin.jar
c0 = c0 * b   -> 50
c0 *= b       -> 50
```

Operatori "++" i "--" povećavaju ili smanjuju primitivne tipove za 1, koristimo ih tako da ih postavimo pored varijable neovisno o strani, a najviše se koristi za korake u petljama.

```
var i = 0
while( i < 10 ) {
    /* Neki kod */
    i++ // Idemo na sljedeći korak petlje
        // sve dok nam se ne ispuni uvjet
}
```

Operatori odnosa nam provjeravaju odnos između dvije varijable. To su operatori ”<”, ”>”, ”<=” i ”>=”. Također imamo operator jednakosti ”==” i ”!=” te operatore jednakosti za pokazivače ”===” i ”!==”.

```
/* kotlin.kt */
fun main( strings: Array<String>) {

    if( strings.size < 1 )
        return

    var i:Int? = strings[0].toIntOrNull()
    if( i == null ) {
        println("loš unos!")
        return
        }

    if( i < 10 )
        println("i je manji od 10!")
    else if( i > 10 )
        println("i je veći od 10!")
    else if( i == 10 )
        println("i je jednak 10!")

}
```

```
$ kotlinc kotlin.kt -include-runtime -d kotlin.jar
$ java -jar kotlin.jar
loš unos!
$ java -jar kotlin.jar 9
i je manji od 10!
$ java -jar kotlin.jar 11
i je veći od 10!
$ java -jar kotlin.jar 10
i je jednak 10!
```

Da bi mogli provjeriti više mogućnosti u jednom uvjetu možemo koristiti ključne riječi "`&&`" za logički i, "`||`" za logički ili i "`!`" za logički ne.

```
val i = 13
if( i < 10 && i > 15 )
    println("I ispunjava i logički uvjet!")
if( i > 10 || i < 15 ) {
    println("I ispunjava ili logički uvjet!")
}
if( !i ) {
    println("i je jednako 0!")
}
```

2.8 Petlje

U *Kotlinu* postoje dvije vrsta petlja, `for` i `while` petlj.

`for` petlje će se ponavljati za svaki element neke varijable koja se sastoji od više vrijednosti kao npr. nizovi.

```
for( objekt in kolekciji ) {
    /* Radimo nešto sa varijablom objekt */
}
```

`for` petlji možemo izbrisati vitičaste zagrade ako izvršava samo jednu liniju koda.

```
for( string in strings ) println("$string")
```

Da ponavljamo `for` petlju od nekog početnog broj do krajnjeg broja koristimo raspone.

```
for( i in 1..10 )
    println("$i")
```

`while` petlje koriste uvjete za ponavljanje te će se ponavljati dok im se uvjet zapisan između zagrada ne ispuni.

```
var i = 0
while( i < 25 ) {
    i++
}
```

`do ... while` petlja nam omogućuje da se kod unutar petlje izvrši bar jedanput prije nego li se uvjet u zagradama provjeri.

```
var i = 0
do {
    println("U petlji")
    i++
} while( i <= 0 )
```

Naredba `continue` nas vraća na početak gdje gledamo je li ispunjen naš uvjet te preskačemo izvršavanje svih linija koda u petlji ispod izjave `continue`. Naredba `break` nas izbacuje iz trenutačne petlje te sve linje koda u petlji ispod izjave `break` ostaju neizvršene.

```
/* kotlin.kt */
fun main( strings: Array<String> ) {

    if( strings.size < 1 )
        return
    var i = 0
    while( i <= 1 ) {

        println("i = $i")
        i++
        when( strings[0] ) {
            "break"->break
            "continue"->continue
        }
        println("iza when!")
    }
}
```

```
$ kotlinc kotlin.kt -include-runtime -d kotlin.jar
$ java -jar kotlin.jar
i = 0
iza when!
i = 1
iza when!
$ java -jar kotlin.jar continue
i = 0
i = 1
$ java -jar kotlin.jar break
i = 0
```

2.9 Rasponi

Rasponi *(engl. range)* koriste se da se provjeri je li broj sadržan u nekom rasponu kod `if`, `else if`, `while` i `when` izjavama. U `for` petlji raspon nam kontrolira raspon u kojem će se petlja ponavljati. Ključna riječ raspona je `in` te za karakteristke raspona koriste se ključne riječi "`..`", `downTo`, `until` i `step`.

Kod `if`, `else if`, `while` i `when` izjavama raspon se prvo saziva s ključnom riječ `in` te se zapisuje naš raspon pomoću dvije varijable ili vrijednosti između kojih se nalazi ključna riječ "`..`".

```
/* kotlin.kt */
fun main() {

    val i = 10
    val j = 11

    /* i in 1..10 je ekvivalentno izjavi
     * i <= 1 && i >= 10
     */
    if( i in 1..10 )
        println("i je u rasponu!")
    else
        println("i nije u rasponu!")

    if( j in 1..10 )
        println("j je u rasponu!")
    else
        println("j nije u rasponu!")

}
```

```
$ kotlinc kotlin.kt -include-runtime -d kotlin.jar
$ java -jar kotlin.jar
i je u rasponu!
j nije u rasponu!
```

U `for` petljama, pošto trebamo više kontrole nad smjerom i korakom petlje, možemo koristiti ključne riječi `downTo` i `step`. Također se može koristiti naredba `until` umjesto ".." te će kao i ".." ponavljati od a do b ali ne uključujući b.

```
/* Kreira se varijabla 'i' u rasponu for petlje koja će
početi iz vrijednosti 1 i svaki korak petlje će se on
povećati za 1 dok ne dođe do vrijednosti 11. */
for( i in 1..10 )
    println("$i")

/* Radit će isto kao i prijašnja petlja, ali u
suprotnome smijeru i svakim korakom se 'i' poveća za 2. */
for( i in 10 downTo 1 step 2 )
    println("$i")

/* Radit će isto kao prva petlja ali će se zaustaviti
kad 'i' dođe na vrijednost 10. */
for( i in 1 until 10 step 1 )
    println("$i")
```

Kada imamo kolekciju objekata možemo iteretirati preko njegovih sadržanih objekata.

```
for( objekt in kolekciji ) {
    println("$objekt")
}
```

U `if`, `else if`, `while` i `when` izjavama možemo provjeriti je li neki objekt sadržan u kolekciji pomoću `in` izjave.

```
when {
    "pas" in zivotinjama -> println("Imamo psa!")
    "macka" in zivotinjama -> println("Imamo macku!")
}
```

2.10 Varijable koje mogu biti "null"

Ako želimo da vrijednost može imati vrijednost "`null`", moramo eksplicitno postaviti da nam varijabla može imati vrijednost "`null`" tako da stavimo znak "`?`" pored njegovog tipa.

```
var vrijednost : String? = null
```

2.11 Provjera tipa i generični tip

Možemo provjeriti tip varijable generične vrijendosti pomoću 'is' izjave. Generičan tip se poziva s ključnom riječ Any.

```
fun jeliInt( objekt: Any ) : Boolean {

    return if( objekt is Int )
        true
    else
        false

}
```

2.12 Funkcije višeg reda i lambda funkcije

Funkcije u *Kotlinu* se mogu spremiti u varijablu, te zbog toga funkcije u *Kotlinu* mogu biti višeg reda. U programiranju Android aplikacija se funkcije višeg reda i lambda funkcije puno koriste te je važno da ih znamo.

Lambda funkcije su funkcije bez imena, njih možemo spemiti u neku varijablu ili izravno dati kao argument nekoj funkciji.

```
/* Funkcija s imenom "zbroj" koji će zbrojit dva broja */
fun zbroj( a: Int, b: Int ): Int {
    return a + b
}
/* Lambda verzija funkcije "zbroj" */
val lambdaZbroj: ( Int, Int ) -> Int = { x, y -> /*
    ~~~~~~~~~~~  ~~~~~~~~~~~  ~~~~~~     ~~~~
         ^             ^         ^          ^
    Ime varijable      |         |          |
                  Broj i tip     |          |
                  parametra      |          |
                            Tip kojeg       |
                            funkcija        |
                            vraća           |
                                      Ime parametara */
    /* Blok koda naše funkcije.
    Vraćamo rezulat zbroja x + y */
    x + y
}
```

Funkcije višeg reda su funkcije koje u svojim parametrima mogu primiti funkciju i/ili vratiti funkciju.

```
/* kotlin.kt */
/* Deklariramo funkciju "baciKockicu" koja će nam primiti 3
   parametra: raspon brojeva, koliko puta bacamo i funkcija
   koja će se izvršiti svaki put kada nam se baci kockica */
fun baciKockicu(
    raspon: IntRange,
    brojPonavljanja: Int,
    funkcija: ( broj: Int ) -> Unit ) {
/*  ~~~~~~~~  ~~~~~~~~~~~~~~ ~~~~~~~
        ^             ^          ^
    Ime parametra     |          |
    funkcije          |          |
    "baciKockicu"     |          |
    čiji tip prima    |          |
    funkciju.         |          |
                      |          |
    Neki broj parametara koje    |
    funkcija mora imati, te ime  |
    i tipovi tih parametara      |
                                 |
    Tip koji nam funkcija koju dajemo kao
    parametar funkciji "baciKocku" mora vratiti.
    "Unit" znači da nam ne mora ništa vraćati. */

    for( i in 0 until brojPonavljanja ) {
        val a = raspon.random() /* Dobivamo naumični
            broj iz našeg raspona */
        funkcija(a) /* Nasumični broj dajemo funkciji
        koja je dana putem parametara te ona dalje
        obrađuje broj.*/
    }

}
```

```
fun main() {
    baciKockicu( 1..12, 3,
    /* Deklariramo ime parametara naše lambda funkcije,
    broj parametara i tip parametara nam je poznat
    prema onome što parametar "funkcija" iz funkcije
    "baciKocku" prima. */
    { broj ->

        println("Dobili smo broj $broj!")

    })
}
```

```
$ kotlinc kotlin.kt -include-runtime -d kotlin.jar
$ java -jar kotlin.jar
Dobili smo broj 3!
Dobili smo broj 2!
Dobili smo broj 9!
```

Ako nam je parametar koji prima funkciju zadnji parametar u našoj funkciji, možemo lambda funkciju zapisati van zagrada.

```
fun main() {
    baciKockicu( 1..12, 3 ){ broj ->

        println("Dobili smo broj $broj!")

    }
}
```

Poglavlje 3

Razvoj aplikacija za Android sučelje

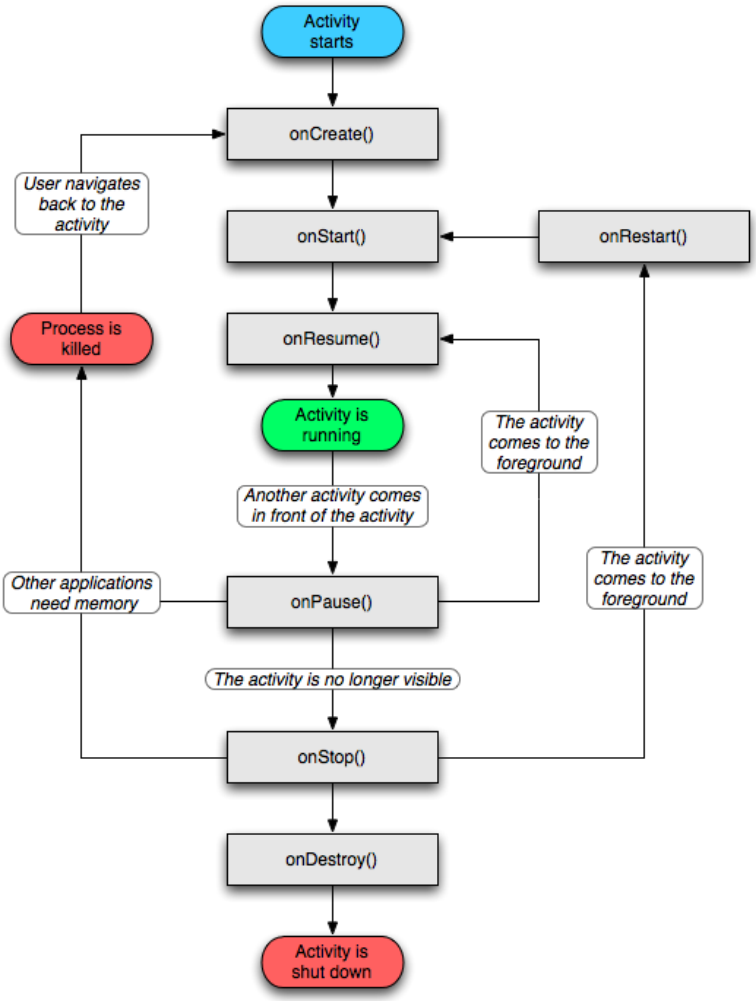
Napokon smo došli do dijela gdje ćemo učiti pravo programiranje za Android platformu. Uvest ću vas u Android programiranje sa jednostavnom aplikacijom i aplikacijom koju predajem uz svoj rad. Ali prije toga moramo naučiti par bitnih pojomova.

3.1 Bitne komponente Android aplikacije

Za znati programirati aplikacije za Android, moramo znati par bitnih komponenata Android aplikacije:

3.1.1 Aktivnosti

`Aktivnost` *(engl. Activity)* je glavni konstrukcijski blok naše aplikacije. Aktivnost nam pruža glavni prozor naše aplikacije u kojem se prikazuju naši elementi. Kao ulazna točka Android programa koristi se `MainActivity`. Osim aktivnosti `MainActivity` možemo imati i druge aktivnosti, primjerice u aplikaciji za rukovanje e-pošte, možemo imati aktivnost za biranje pošte, čitanje pošte i pisanje pošte.



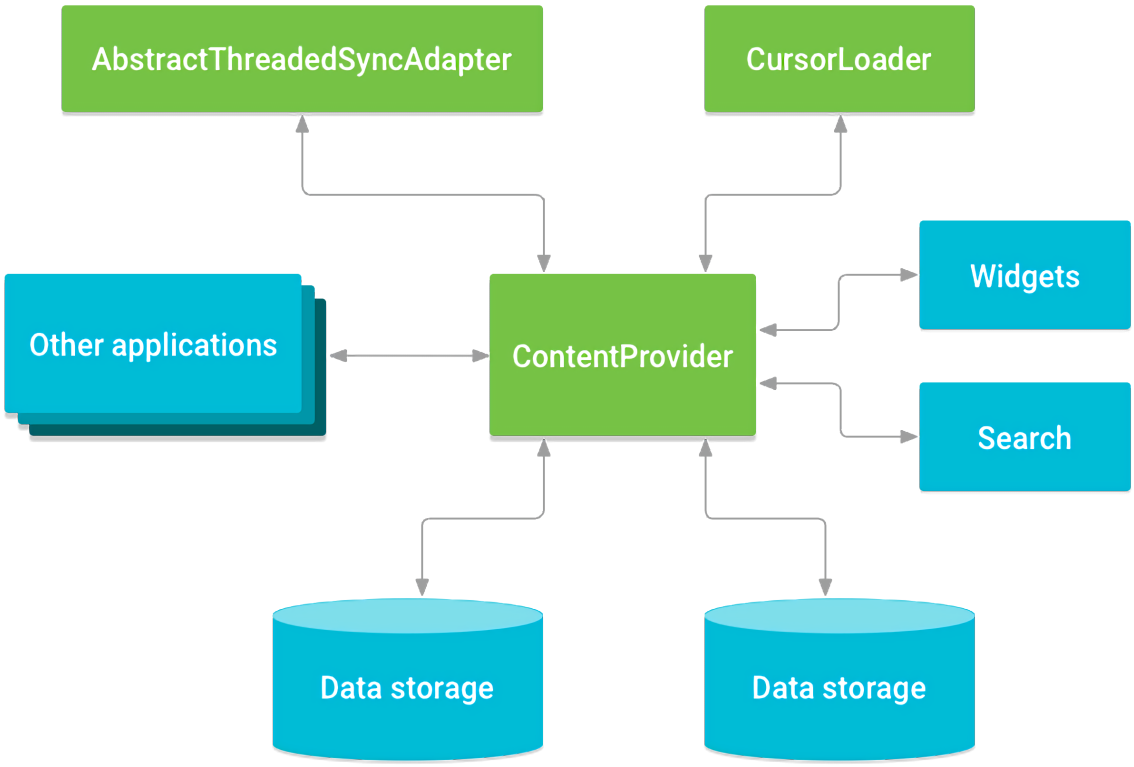
Slika 3.1 – Životni vijek aktivnosti, za svaki dio životnog vijeka može se napisati kod koji će se izvršit kada životni vijek dođe do tog dijela.

3.1.2 Usluge

`Usluga` *(engl. Service)* je komponenta Android aplikacije koja može dugotrajno raditi u pozadini. Primjerice kada korisnik upali nekakvu aplikaciju, nakon što izađe iz aplikacije ili pauzira aplikaciju u pozadini može još uvijek biti pokrenuta usluge te aplikacije neovisno o životnom vijeku aplikacije. Primjer bi bio nekakva aplikacija za primanje i slanje poruka koja će u pozadini s nekakvom uslugom vidjeti jeli itko slao poruku korisniku dok je aplikacija ugašena te nam može ispostaviti notifikaciju ako je.

3.1.3 Pružatelji sadržaja

`Pružatlej sadržaja` *(engl. Content Provider)* je apstraktni sloj za pristupljanje pohrani podataka sustava ili drugih aplikacija. Zbog toga Androidov model razvoja programa potiče programere da svoje podatke naprave pristupne drugim aplikacijama.



Slika 3.2 – Grafički prikaz pružatelja sadržaja, preko njega svaka komponenta može pristupiti pohrani drugih komponentenata iz slike.

3.1.4 Prijemnici obavijesti

`Prijemnik obavijesti` *(engl. Broadcast Reciver)* je prijemnik obavijesti sustava. Sustav šalje obavijesti o zbivanja u sustavu, primjerice: kada je baterija pri kraju, kada se spojimo ili odspojimo sa Wi-Fi mreže itd. Prijemnik obavijesti aplikacije sluša obavijesti sustava te obavještava aplikaciju tako da aplikacija može pristupit u skladu s obavijesti.

3.2 Bitni pojmovi Android programiranja

3.2.1 Widgeti

`Widget` *(eng. Widget)* je mali dio našeg UI sučelja. Widgeti su nam tekst, gumbovi, liste, slike itd. Svaki widget ima svoj identifikacijski broj te pomoću njega možemo zvati funkcije vezane uz taj widget.

3.2.2 Kontejner/Upravitelj Rasporeda

Za organiziranje widgeta na ekranu, koristimo kontejnersku klasu "`Upravitelj Rasporeda`" *(engl. Layout Manager)*. Pomoću njega možemo na kompleksniji i precizniji način rasporediti naše widgete po ekranu. Odnos između widgeta i kontejnera se zapisuje u resursnu datoteku upravitelja izgleda od koje će se prikazat UI.

3.2.3 Resursi

`Resursi` *(engl. resources)* su stvari kao slike, tekst i druge stvari koje aplikacija koristi tokom svojeg vremena izvođenja. Tipične Android aplikacije koriste mnogo resursnih datoteka.

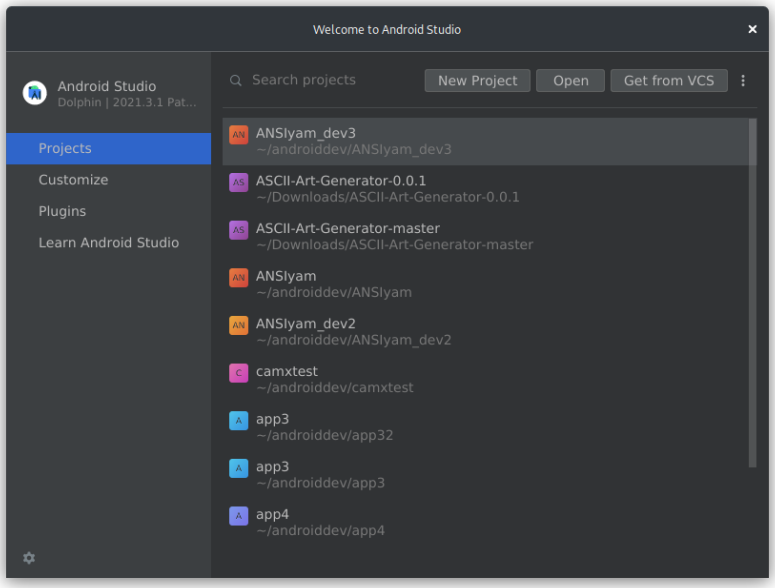
3.2.4 Fragmenti

`Fragmenti` *(engl. Fragment)* su korisni za prilagođavanje izgleda aplikacije kada koristimo mobitel ili tablet. Primjerice za neku aplikaciju koja prima e-poštu. Ako želimo kad koristimo tablet da se istovremeno može birat i čitat pošta, a za mobtel da možemo samo jednu od tih stvari moramo koristiti fragmente.

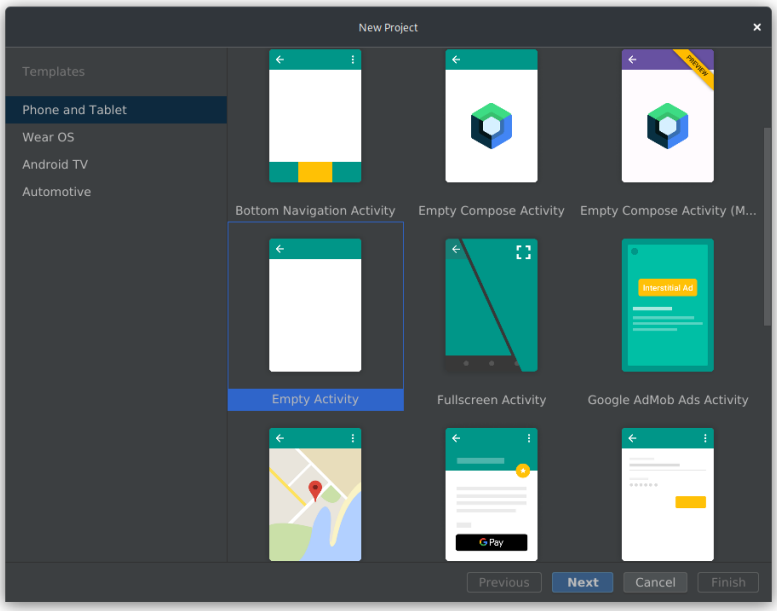
3.3 Razvoj jednostavne Android aplikacije

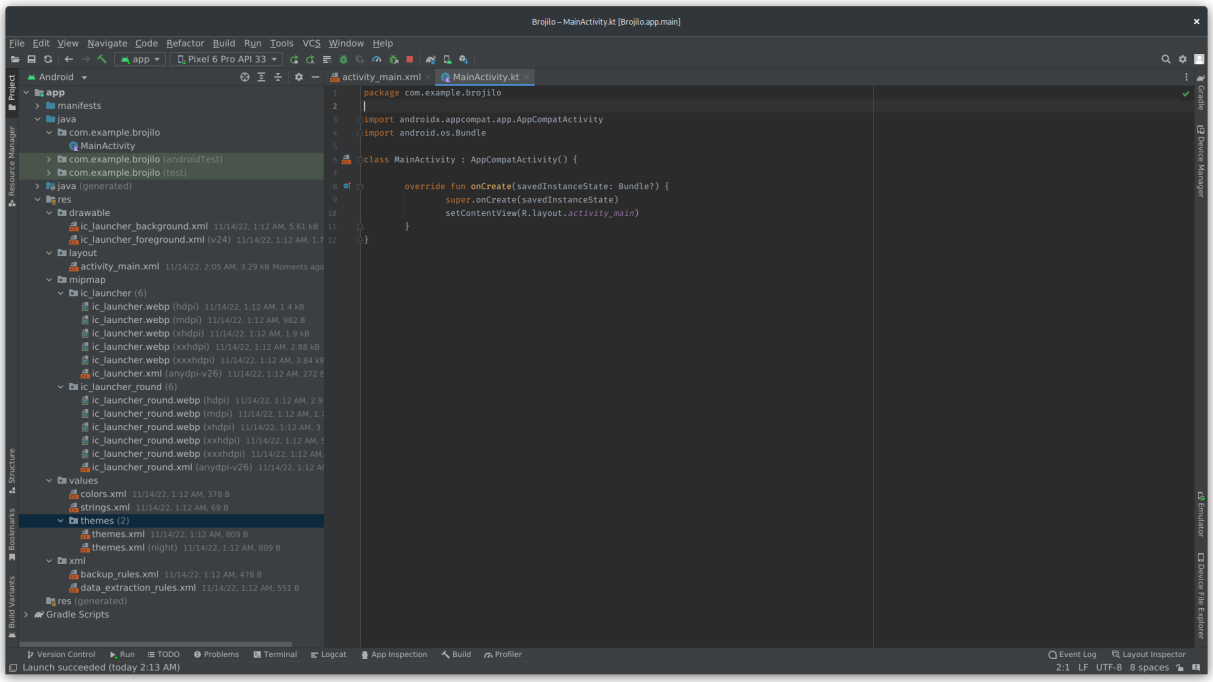
3.3.1 Kreiranje novog projekta

Napokon smo došli do djela gdje ćemo naučiti kako programirat Android aplikacije! Za početak ćemo otvoriti Android Studio i započeti novi projekt.



Kada kliknemo *"Novi projekt"* uzet ćemo unaprijed definirani predložak *"Prazna Aktivnost"* te ćemo uči u Android Studio.





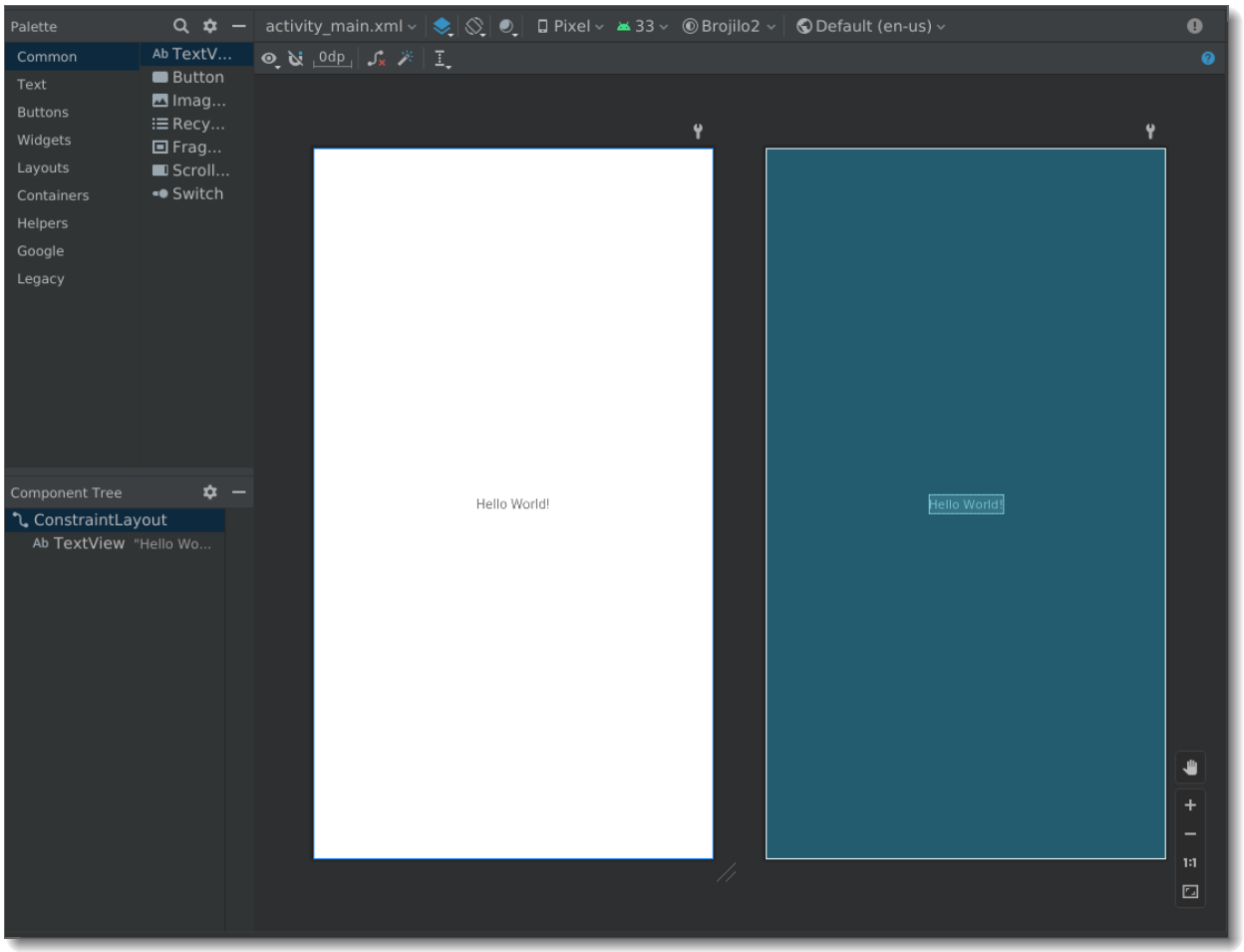
Ovako će nam izgledati Android Studio kada stvorimo novi projekt. S lijeve strane nam se nalazi struktura datoteki i mapi našeg projekta.

Trenutačno je otvoren naš `MainActivity` spremljen u datoteci *MainActivity.kt*, kada otvorimo novi projekt u njemu će biti zapisan kod:

```
/* Ime projekta, moj se zove brojilo */
package com.example.brojilo

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

/* Naša klasa zvana MainActivity koja nasljeđuje od klase
AppCompatActivity*/
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }

}
```

Trenutačni izgled naše aplikacije je spremljen u datoteci *activity_main.xml*. Svaka aktivnost i svaki fragment ima svoj *".xml"* datoteku tj. *upravitelj izgleda* koja mu kontrolira koje widgete i kontejnere sadrži, te njihove odnose i vrijednosti.

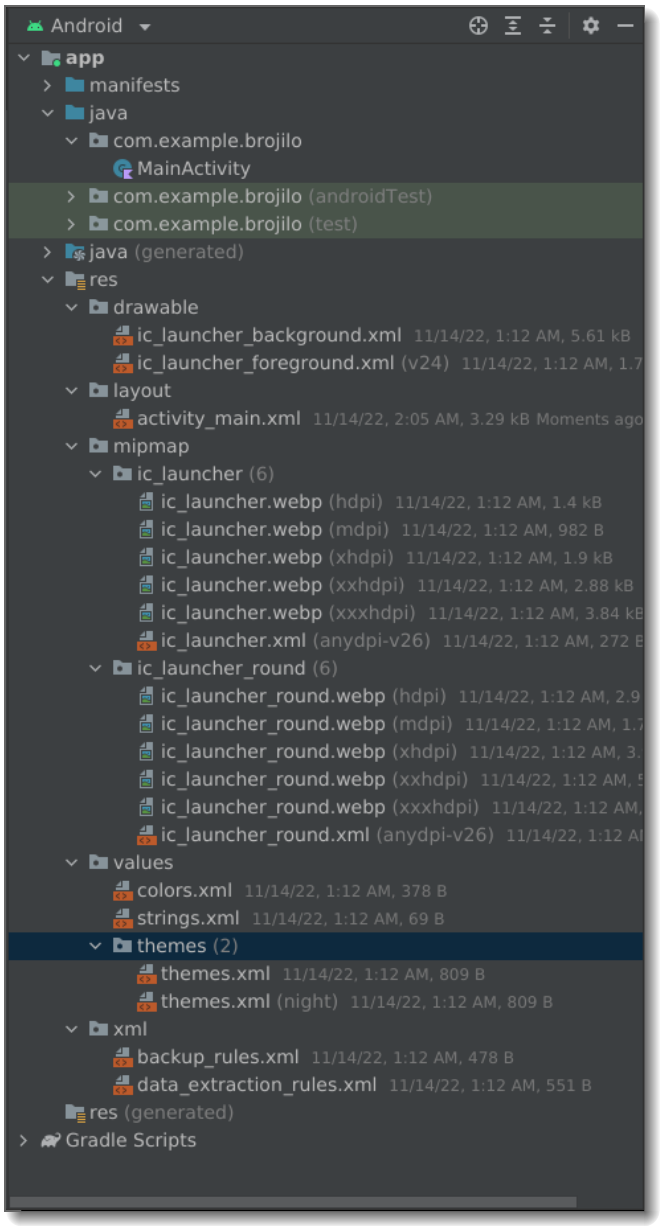
Kada prvi put otvorimo u novom projektu *activity_main.xml* ili bilo koju novu kreiranu *".xml"* datoteku koja je *upravitelj izgleda* neke aktivnosti ili fragmenta ona će izgledat ovako:



Slika 3.3 – Zadani oblik *main_activity.xml* pri kreiranju novog projekta, praktički *Hello, World!* za Android programiranje.

3.3.2 Kompozicija mape Android programa

U našem novom projektu možemo vidjeti još neke druge stvari osim *main_layout.xml* u našoj *res* mapi.
res datoteka koristi se za spremanje sljedećih vrijednosti:
tekst kojeg možemo koristit bilogdje u aplikaciji spremamo u datoteku *values/string.xml*,
boje spremamo u datoteku *values/colors.xml*,
temu aplikacije možemo prilagoditi našim ukusima pomoću datoteke *values/themes/themes.xml*,
u mapi *layout* čuvamo naše **upravitelje izgleda** aktivnosti i fragmenata,
u mapi *mipmap* čuvamo **slike sa rastorskom grafikom**,
dok u folderu *drawables* čuvamo **slike sa vektorskom grafikom**.
Naša *res* mapa može imati puno drugih resursnih vrijednosti koje programeri mogu dodat, ovo su samo obične resursne datoeke koje dobijemo pri stvaranju novog projekta.
Mapa *Gradle scripts* sadrži naš *Gradle* kompajler. Pomoću njega možemo kontrolirati naše kompilacijske opcije kao eksterne biblioteke, minimalni SDK (minimalna verzija Androida) itd.
Imamo jos drugih datoteka u našoj glavnoj mapi kao *manifest/Android_Manifest.xml* koji nam informacije o aplikaciji:
ime, glavnu aktivnost, temu itd.
U na kraju u mapi *java* nam je sadržan sav kod ove aplikacije.



Slika 3.4 – Strukutra glavne mape našeg projekta

3.3.3 Dodavanje widgeta i pozicioniranje

Za početnu aplikaciju napraviti ćemo jednostavni brojač koji će brojati broj puta kojeg smo kliknuli jedan gumb. Također ćemo dodat jedan *switch* koji će nam odobriti ili onemogućiti brojanje.

Kao prvo, trebamo izbrisat *TextView* widget sa tekstom *"Hello World!"* a umjesto njega ćemo dodat druge widgete.

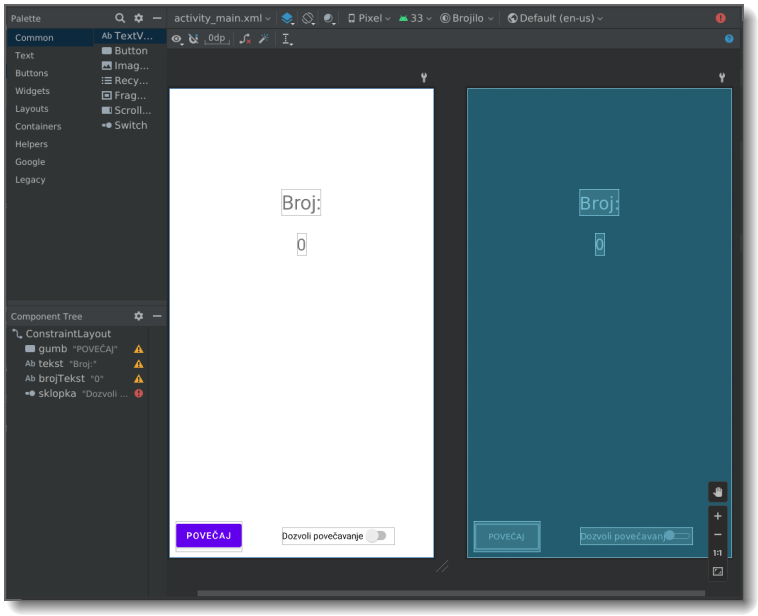
Trebat će nam dva *TextView* widgeta, jedan ćemo nazvat *tekst*, a drugi ćemo nazvat *brojTekst*. Za *tekst* widget postavit ćemo da njegov tekst čita *"broj"*, a za drugog ćemo postaviti da čita *"0"*.

Kako se nalazimo u *ConstraintView* kontejnjeru možemo kliknut na jedan od naših widgeta te će nam se pojaviti četri kugle na četiri strane tog widgeta. Te četri kugle reprezentiraju na koji je element limitiran naš widget, te ovisno o tim limitacijama naš widget će se onda dinamički micat ovisno o veličini i omjeru ekrana. Mi ćemo za widget *tekst* spojiti lijevu, desnu i donju kuglu sa lijevom, desnom i gornjom kuglom našeg drugog widgeta *brojTekst*. Za widget *brojTekst*, njegove kugle ćemo spojiti sa istoimenom stranom ekerana kao kugla koju spajamo te ćemo je centrirat kod budemo gotovi.

Trebat će nam jedan *Button* widget kojeg ćemo nazvat *gumb*, postavit ćemo mu tekst da čita *"Povećaj"*, a njegovu donju i lijevu kuglu ćemo spojit sa donjom i lijevom stranom ekrana.

Treba će nam jedan *Switch* widget kojeg ćemo nazvat *sklopka*. Njegovu desnu, gornju i donju kuglu ćemo spojiti sa lijevom, gornjom i donjom kuglom *gumb* widgeta.

Kada završimo to bi sve trebalo izgledat ovako:



Slika 3.5 – Izgled našeih dodani elemanta u upravitelju izgleda za naš `MainActivity`

3.3.4 Dobivanje i postavljanje vrijednosti widgeta

Sada kad smo kreirali i postavili sve widgete možemo prosljediti sa kodiranjem njihovih *događaja (engl. Events)*.

Prije negoli počnemo pisat ikakav funkcionalni kod, moramo dobiti *pogled (engl. View)* naših widgeta i spremiti ih u varijablu. To ćemo raditi pomoću funkcije *findViewById*. Ona zahtjeva dva parametara, između "<" i ">" zagradama ćemo napisat tip widgeta kojeg tražimo. Nakon toga u normalnim zagradama joj dajemo identifikacijki broj našeg widgeta. Svaki identifikacijski kod počne sa *R.id.* te nakon toga zapisujemo *id* widgeta.

```
val gumb = findViewById<Button>(R.id.gumb)
```

Tako da prije negoli počnemo pisat moramo dobiti takozvani `pogled` naših widgeta.

```
/* Ime projekta, moj se zove brojilo */
package com.example.brojilo

import android.widget.Button
import android.widget.Switch
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

/* Naša klasa zvana MainActivity koja nasljeđuje od klase
AppCompatActivity*/
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val sklopka = findViewById<Switch>(R.id.sklopka)
        val tekstBroj = findViewById<TextView>(R.id.brojTekst)
        val gumb = findViewById<Button>(R.id.gumb)

    }

}
```

Kada smo dobili *poglede* naših widgeta. Sada ćemo napisat kod za naš brojač.

```
/* Ime projekta, moj se zove brojilo */
package com.example.brojilo

import android.widget.Button
import android.widget.Switch
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

/* Naša klasa zvana MainActivity koja nasljeđuje od klase
AppCompatActivity*/
class MainActivity : AppCompatActivity() {

    var broj = 0

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val sklopka = findViewById<Switch>(R.id.sklopka)
        val tekstBroj = findViewById<TextView>(R.id.brojTekst)
        val gumb = findViewById<Button>(R.id.gumb)

        /* gumb ima funkciju setOnClickListener
        koja prima lambda funkciju koja
        će se izvršit svaki put kad kliknemo gumb. */
        gumb.setOnClickListener{
            /* Jedino ćemo povećat broj ako nam je sklopka
            isključena */
            if( sklopka.isChecked() != true ) {
                /* Povećaj broj i prikaži promjene */
                broj++
                tekstBroj.text = broj.toString()

            }

        }

    }

}
```

3.3.5 Emulator

Slika 3.6 – Slika aplikacije u radu iz virtualnog mobitela

Da vidimo kako aplikacija radi moramo napraviti virtualni mobitel tj. emulator Android operacijskog sustava. Pored zelene strjelice na gornjoj traci, kliknemo na kučicu u kojoj možemo izabrati naš emulator. Pošto mi nemamo emulator jer nam je ovo prvi put da koristimo Android Studio, u opciji *"Device Manager"* kliknit ćemo gumb *"Create device"* i stvoriti novi virtualni mobitel. Izaberite bilo koji ponuđeni mobitel, ja sam izabrao *Pixel 6 Pro*. Kada ste gotovi sa kreacijom Android Studio će automatski preuzet sve potrebne datoteke da kreira uređaj. Koliko je još potrebno možete vidjeti u donjem lijevom kutu.
Kad se sve instalira možete kliknut zeleni gumb da kompjalirate i pokrenete vašu aplikaciju da možemo vidjet kako radi.

3.4 Seminarska aplikacija

Za svoju seminarsku aplikaciju bilo je potrebno da koristim *CameraX* API. Pomoću njega sam mogao dobiti slike sa kamere.

CameraX je API koji nam olakšava korištenje kamere, on ima podršku za najkorištenije slučajeve uporabe: `pretpregled` *(engl. preview)*, `analiza slike` *(engl. image analysis)*, `slikavanje` *(engl. image capture)* i `snimanje` *(engl. camera recording)*. Ja sam uz dodatnu bibloteku com.neberox.library:asciicreator:0.0.1 kreirao aplikaciju koja pretvara sliku dobivenu od kamere u sliku sastavljenu od različith slova. Za moju aplikaciju bilo je potrebno da koristim tri od ćetiri slučaja koje *CameraX* pokirva:

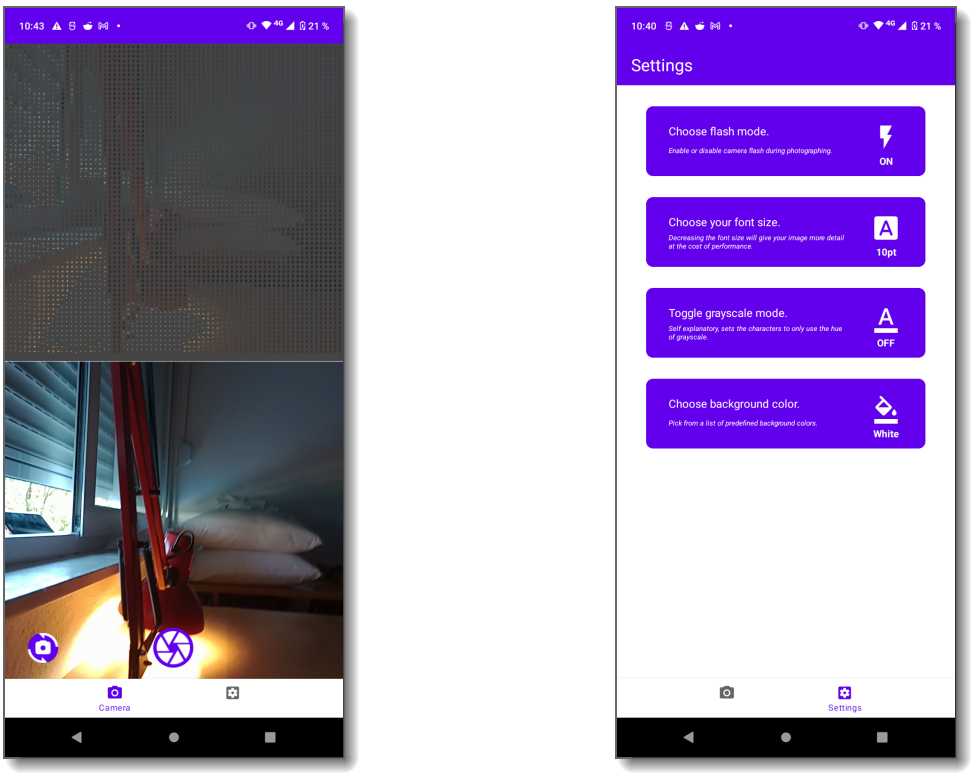
Pretpregled mi je potreban za vidjeti pravu sliku uz sliku sastavljenu od slova.

Analiza slike mi je potrebna da uz pretpregled prikazujem sliku sastavljenu od slova tako da koristnik ne slikaje "slijepo".

I zadnje *slikavanje* mi je potrebno da slikamo te poslje obradimo slikanu sliku i pretvormo je u sliku od slovima.

Dodatno sam kreirao postavke da kontrilramo vrijednosti vezane uz sliku te također i gumb za okrenuti kameru sa zadnje u prednju i obrnuto.

Kod za moju aplikaciju je prevelik da stane ovdje pa ću umjesto koda prikazat aplikaciju par slika aplika



Slika 3.7 – Slike moje aplikacije za seminarski rad.

Zaključak

U ovom seminaru mi je zadan zadatak za kojeg sam morao naučit nešto kompletno novo jer nisam imao ikakvog predznanja o temi koja mi je zadana. Ja sam se odlučio za mobilno programiranje, specifično programiranje za Android platformu. Možda to nebi bila najbolja opcija za neke jer sigurno bilo puno lakših stvari obraditi umjesto programiranja za Android platformu, za mene je ovo odlično jer mogu naučiti neke stvari koje će mi služiti cijeli život. Ja se nadam da sam vas čitatelju također nešto novo naučio i da sam vam pobudio interes da sami naučite više o programiranju za Android jer je znanje alat koji će vas služiti cijeli život!