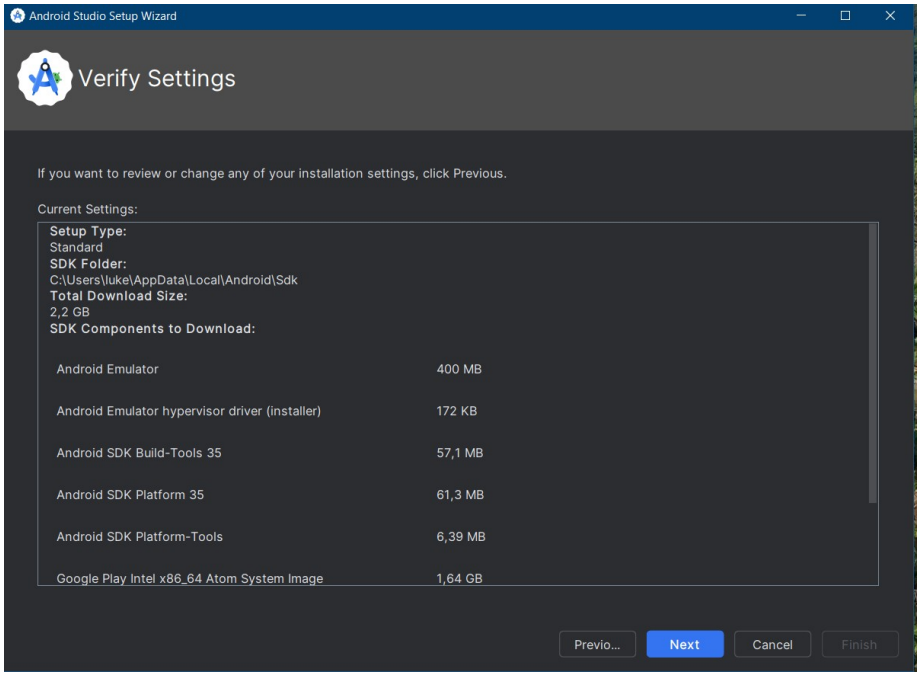
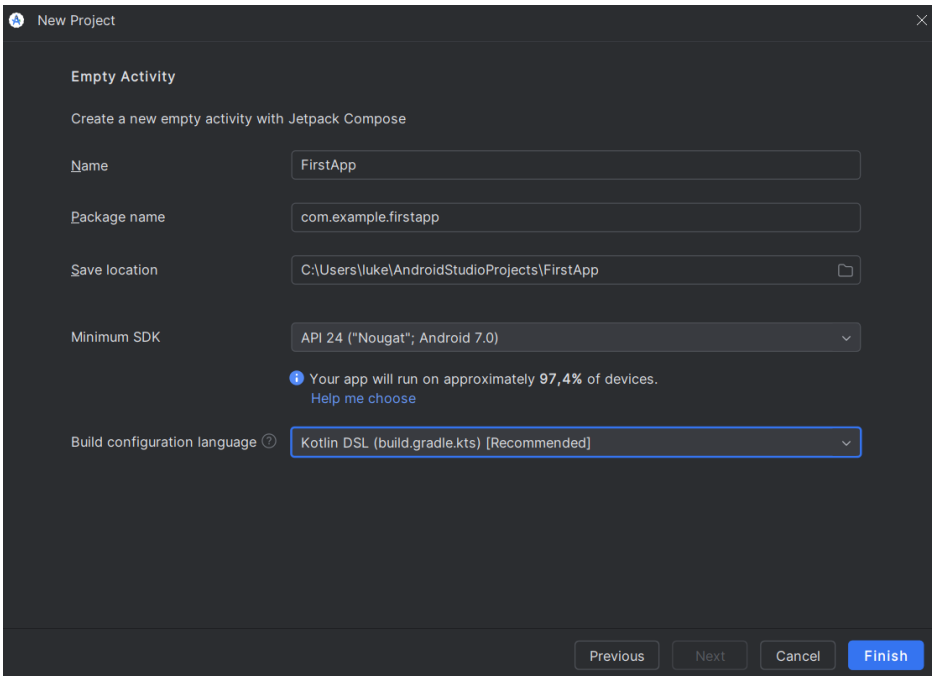
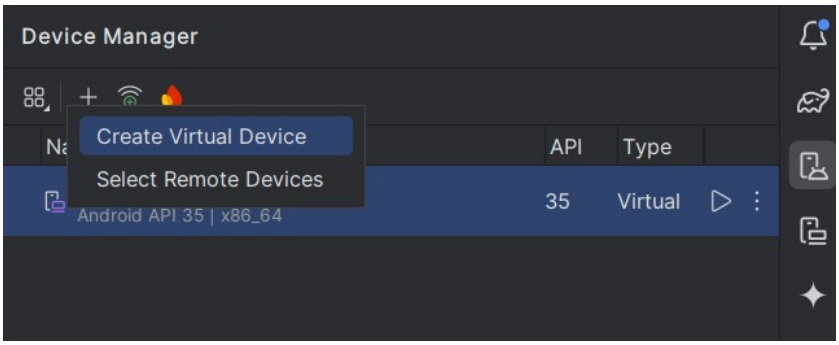
Android Studio pobieramy ze strony:
https://developer.android.com/studio?hl=pl i instalujemy z domyślnymi komponentami:



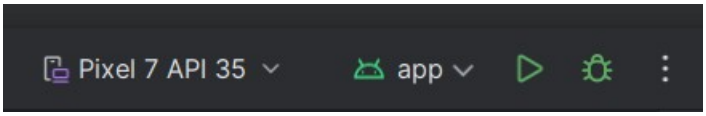
Tworzymy nowy projekt z pustą aktywnością:



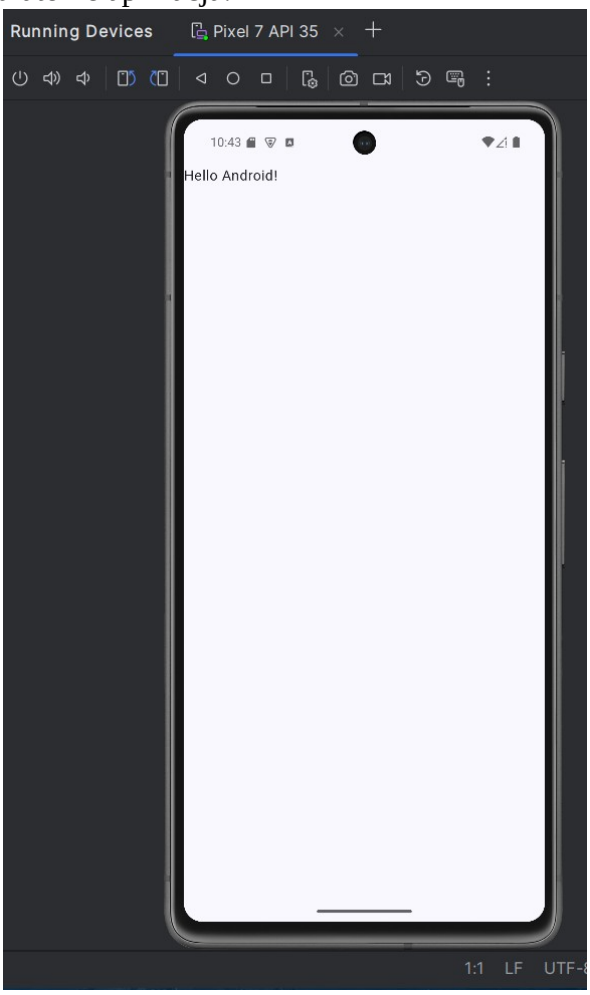
Dodajemy nowe wirtualne urządzenie (emulator androida) np. Pixel7:



Wybieramy wersję API androida (wybieram domyślnie pobraną 35)
Możemy uruchomić emulator lub od razu zainstalować i uruchomić na nim naszą aplikację:



Uruchomiona na emulatorze aplikacja:



Importy androidowe naszej aplikacji:

```
package com.example.firstapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import com.example.firstapp.ui.theme.FirstAppTheme
```

W nowoczesnym programowaniu aplikacji mobilnych na systemy android do budowania UI używa się Jetpack Compose, czyli zestawu narzędzi zastępujących tradycyjne podejście oparte na plikach XML. Jetpack Compose pozwala na budowanie UI w sposób deklaratywny, bez potrzeby używania XML do definiowania layoutów.

W tradycyjnym podejściu opartym na XML interfejs użytkownika tworzony jest imperatywnie – najpierw definiuje się layout w pliku XML, a następnie w kodzie manipuluje jego elementami (findViewById).

W nowym podejściu UI jest reaktywne, co oznacza, że automatycznie dostosowuje się do zmian w stanie aplikacji. Gdy stan (np. tekst, liczba, wartość) ulega zmianie, Jetpack Compose automatycznie aktualizuje interfejs użytkownika bez potrzeby ręcznej aktualizacji widoków.

Interfejs tworzony jest za pomocą funkcji kompozycyjnych (@Composable), które opisują jakie elementy mają być wyświetlone, w jakim układzie, z jakimi stylami, i jak powinny reagować na interakcje użytkownika.

Aplikacja na androida składa się z aktywności **Activity**. Każda z aktywności może być traktowana jak interaktywny ekran z interfesjem dzięki któremu użytkownik może wyświetlać i wprowadzać dane.

MainActivity jest uruchamiana jako pierwsza, gdy użytkownik otwiera aplikację.
Kod aplikacji może wyglądać następująco:

```
class MainActivity : ComponentActivity() {
    @OptIn(ExperimentalMaterial3Api::class)
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
//        enableEdgeToEdge()
        setContent {
                        FirstAppTheme {
                                Scaffold( topBar = {
                                        TopAppBar(title = {  Text("Pierwsza Aplikacja") })
            },modifier = Modifier.fillMaxSize()) {  innerPadding ->
                                        Greeting(
                    name = "Android",
                    modifier = Modifier.padding(innerPadding)
                )
                //ToastDrawer(modifier = Modifier.padding(innerPadding))
            }
                        }
                }
        }
}

@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}
```

Główna aktywność jest tworzona za pomocą funkcji **onCreate:**

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

//        enableEdgeToEdge() -- w API35 rozszerza okno aplikacji od krawędzi do krawędzi

setContent {
```

Metoda, która pozwala na zdefiniowanie całego interfejsu użytkownika za pomocą funkcji kompozycyjnych w Kotlinie.

Funkcje kompozycyjne są wywoływane z nawiasem klamrowym, gdyż ich ostatnim parametrem jest lambda, w tym wypadku:

```
content: @Composable () -> Unit
```

Unit w Kotlinie totyp, który jest automatycznie używany w miejscach, gdzie funkcje nie zwracają żadnej wartości.
analogicznie dla

```
FirstAppTheme{
```

Funkcja ta znajduje się w com.example.firstapp.ui.theme używa niestandardowych ustawień Material Design takich jak kolory, czy wygląd czcionki deklarowanych w ui.theme

Następnie mamy **Scaffold**, który działa jako szablon dla standardowego układu ekranu mogący posiadać elementy takie jak:

topBar: pasek aplikacji na górze ekranu.
bottomBar: pasek aplikacji u dołu ekranu.
floatingActionButton: przycisk znajdujący się w prawym dolnym rogu strony ekranu

W Scaffold Lambda definiuje główną zawartość ekranu, a jej parametrem jest innerPadding, które jest dynamicznie przekazywane przez Scaffold, aby treść nie zasłaniała się z innymi elementami UI (np. top bar).

```
Scaffold(modifier = Modifier.fillMaxSize()) {  innerPadding ->
        Greeting(
        name = "Android",
        modifier = Modifier.padding(innerPadding)
    )
}
```

Na koniec własna funkcja kompozycyjna z tekstem Android:

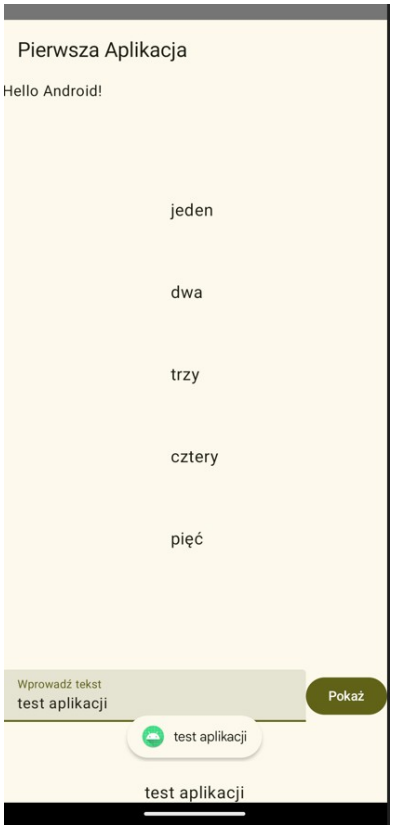
```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}
```

Modifier w Composable działa jak narzędzie do budowania "łańcuchów" modyfikacji.
W przykładzie w Greeting dodaje tylko wewnętrzny marginesy.
W Scaffold rozszerza go na całe okno. Scaffold można potraktować jako kontener na kolejne elementy UI.

Aby wykonać zadanie z zajęć potrzebujemy jeszcze kilku narzędzi.
Zakładamy, że poza przywitaniem chcemy napisać aplikację z listą 5 elementów na górze ekranu,

inputem do wpisywania tekstu oraz przyciskiem który wywoła Toast oraz wyświetli wpisany tekst na ekran.
Utworzymy własną funkcję analogicznie jak Greating z przekazaniem modifier. A w niej kolejne komponenty.

```
@Composable
fun ToastDrawer(modifier: Modifier = Modifier) { //…
}
```



Do rozmieszczenia komponentów należałoby użyć funkcji **Column** oraz **Row** z odpowiednią aranżacją. Dla rozmieszczenia pionowego **verticalArrangement** kolumny mamy np:

```
verticalArrangement = Arrangement.SpaceBetween
```

Arrangement.Top: Wszystkie elementy są rozmieszczone na górze kolumny.
Arrangement.Bottom: Wszystkie elementy są rozmieszczone na dole kolumny.
Arrangement.Center: Wszystkie elementy są rozmieszczone na środku kolumny.
Arrangement.SpaceBetween: Pierwszy element znajduje się na górze, ostatni na dole, a pozostałe są rozłożone równo pomiędzy nimi.
Arrangement.SpaceAround: Elementy są równomiernie rozmieszczone, z dodatkowym miejscem na górze i dole kolumny.
Arrangement.SpaceEvenly: Wszystkie elementy są rozmieszczone równomiernie, z równą przestrzenią między nimi, jak i nad i pod nimi.

Lista będzie na górze, a przycisk(Button) z inputem(TextField) i odpowiedzią(Text) pod spodem.

W wypadku rozmieszczenia poziomego:

```
horizontalAlignment = Alignment.CenterHorizontally
```

Alignment.Start: Elementy są wyrównane do lewego brzegu.
Alignment.CenterHorizontally: Elementy są wyrównane do środka w poziomie.
Alignment.End: Elementy są wyrównane do prawego brzegu.

Wiersze Row analogicznie mają te same właściwości.

W kolumnach jak i wierszach, które są funkcjami kompozycyjnymi możemy umieszczać komponenty, oraz kolejne kolumny i wiersze a w nich poszczególne komponenty.

np.:

```
Column(
    modifier = modifier,
    verticalArrangement = Arrangement.SpaceBetween,
    horizontalAlignment = Alignment.CenterHorizontally
) { //tutaj np. kolejna kolumna z listą, i druga z wierszem oraz tekstem.
}
```

Wyświetlenie zawartości listy w kolumnie może odbyć się za pomocą zwykłej pętli:

```
Column(
    modifier = modifier,
    horizontalAlignment = Alignment.Start
) {
        val texts = listOf("jeden", "dwa", "trzy", "cztery", "pięć")
    for (item in texts) {
        Text(text = item, modifier = modifier)
    }
}
```

Wiersz w kolejnej kolumnie może wyglądać następująco:

```
Row(
    modifier = modifier,
    verticalAlignment = Alignment.CenterVertically
) { //tutaj np. TextField oraz Button
  }
```

Jeżeli chcemy umieścić TextField w wierszu mógłby wyglądać tak:

```
TextField(
    value = inputText,
    onValueChange = {  inputText = it  },
    label = {  Text("Wprowadź tekst") },
    modifier = Modifier.weight(1f)
)
```

```
value = inputText
```

Wartość ze zmiennej inputText. Należy ją utworzyć w funkcji ToastDrawer.
Jednak nie jest to zwykła zmienna. Jej stan musi zostać zapamiętany oraz zmieniany w razie potrzeby – w tym wypadku na każdą zmianę tekstu w TextFieldzie

```
onValueChange = {  inputText = it  }
```

Gdy użytkownik wpisuje tekst: onValueChange aktualizuje stan inputText.
Compose przebudowuje UI: Po aktualizacji stanu, Compose automatycznie odświeża interfejs, pokazując nowy tekst w polu tekstowym.

Jak prawidłowo utworzyć taką zmienną?

```
var inputText  by remember {   mutableStateOf("") }
```

Jetpack Compose używa operatora **by** do delegowania zarządzania stanem w funkcji remember
remember Zapamiętuje stan, aby nie był tracony podczas ponownego renderowania UI.
mutableStateOf("") tworzy stan, który początkowo zawiera pusty ciąg znaków
MutableState<String>.
Kiedy wartość inputText się zmienia (np. użytkownik wpisuje tekst w pole tekstowe), interfejs użytkownika jest automatycznie odświeżany, aby pokazać nową wartość.

```
modifier = Modifier.weight(1f)
```

waga zajmowanego komponentu w tym wypadku w wierszu. Jeżeli inne komponenty nie będą miały przypisanej wagi – zajmie większość dostępnego miejsca.

Przycisk do wywoływania Toast'a (krótkiego powiadomienia):
context należy ustawić w funkcji kompozycyjnej Composable (ewentualnie przekazać do tej funkcji przy tworzeniu aktywności).

```
val context = LocalContext.current
```

W przycisku

```
Button(
   onClick = {
                Toast.makeText(context
, inputText, Toast.LENGTH_SHORT).show()
       displayedText = inputText
   }
) {
       Text("Show text")
}
```

```
onClick
```

Co się dzieje po kliknięciu – pokazujemy powiadomienie **Toast**,
Powiadomienie potrzebuje kontekstu aplikacji (aktywności)
Context to obiekt, który zapewnia dostęp do zasobów i funkcji systemowych Androida.

przypisujemy wartość do kolejnej zmiennej.
Wykorzystamy zmienną displayedText w kolejnym komponencie Text(poza Buttonem).

```
Text(text = displayedText,modifier=modifier)
```

Aby Text mógł się automatycznie zmienić, displayedText również musi być zapamiętany i obsługiwany przez stan:

```
var displayedText by remember {   mutableStateOf("") }
```

Ważne:
remember przechowuje stan tylko w czasie trwania jednej kompozycji. Po zniszczeniu aktywności (np. przez obrót ekranu) stan jest tracony.
RememberSaveable zapisuje i przywraca stan między zmianami konfiguracji, takimi jak obrót ekranu, używając mechanizmów zapisywania stanu w Androidzie.

Zadanie1:
W prostej aplikacji androidowej utwórz 5 tekstów (Text), dane pobrane w pętli z listy stringów.
Na dole aplikacji utwórz TextField oraz Button obok siebie.
Pod spodem umieść Text do wyświetlenia zawartości TextField.

Na przyciśnięcie przycisku powinien pojawiać się Toast o treści z zawartości TextFielda oraz powinien aktualizować się Text na samym dole aplikacji.