

**Usługi i aplikacje Internetu rzeczy (PBL5)**

# **Aplikacje dla systemu Android**

Aleksander Pruszkowski oraz Mykyta Vovk

Instytut Telekomunikacji Politechniki Warszawskiej

# Java minimum wiedzy

## ■ Java – minimum wiedzy

- Co to jest klasa – usystematyzowany zapis definicji nowego typu danych z definicją metod na nich operujących

```
public class Samochod {  
    private int predkosc;  
    private String kolor;  
}
```

### Pola prywatne

(dostęp do nich mają metody tej klasy)

```
    public void ustawPredkosc(int nowaPredkosc) {  
        predkosc = nowaPredkosc;  
    }  
    public void ustawKolor(String nowyKolor) {  
        kolor = nowyKolor; // (4)  
    }  
    public void wypiszInformacje() {  
        System.out.println(  
            "Jestem samochodem! Moj kolor to "  
            + kolor +  
            ", jade z predkoscia " + predkosc);  
    }  
}
```

### Metody

(wywoływać je mogą inne metody w tym innych klas)

Pola i metody mogą być modyfikatorem dostępu czyli słowem:

public – są wtedy dostępne publicznie

private – gdy pola/metody mają być dostępne tylko z tej metody

protected – zabezpieczony jak private ale dostępny dla klas dziedziczących lub nadrzędnych

final – gdy tworzymy obiekt niezmienny (np.: stała PI), próba zmiany jego wartości zakończy się komunikatem o błędzie od kompilatora

# Java minimum wiedzy

## ■ Java – minimum wiedzy

- Konstruktor obiektu – metoda domyślnie uruchamiana podczas tworzenia obiektu danego typu

```
public class Samochod {  
    private int predkosc;  
    private String kolor;  
  
    public void Samochod(){  
        this.predkosc = 0;  
    }  
    public void Samochod(int p){  
        this.predkosc = p;  
    }  
    ...  
}
```

### Różne wersje konstruktora

Uwaga! Tu odwołujemy się do pól tego obiektu za pomocą wskazania „this”

Gdy użyjemy „super” odniesiemy się do tzw. klasy bazowej (po której ta klasa mogłaby dziedziczyć/rozszerzać)

Tutaj konstruktor jest w dwóch wersjach – można zatem tworząc obiekty klasy Samochód na dwa sposoby:

```
Samochod moj = Samochod(180); //użyty będzie konstruktor: public void Samochod(int p);
```

```
Samochod kolegi = Samochod(); //użyty będzie konstruktor: public void Samochod();
```

Uwaga na wieloznaczność – czasami trudno uchwycić która z wersji konstruktora zostanie użyta (to samo dotyczy metod – one też mogą być w różnych wersjach)!

# Java minimum wiedzy

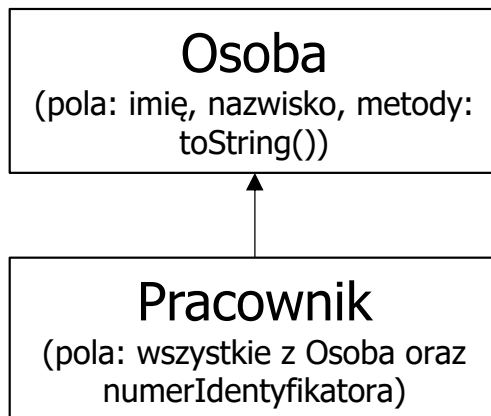
## ■ Java – minimum wiedzy, cd.

### ■ A co to jest dziedziczenie?

- Najprościej ustalenie hierarchi klas mogących wspólnie tworzyć nowe obiekty –  
przykład

#### **Osoba.java**

```
package prostyprzyklad;  
public class Osoba {  
    public String imie;  
    public String nazwisko;  
    public String toString() {return "Osoba " + imie + " " + nazwisko;}  
}
```



Obiekt *Pracownik* może użyć pól publicznych odziedziczonych a zdefiniowanych w *Osoba* ale obiekt *Osoba* nie może odwoływać się do pól *Pracownik*

#### **Pracownik.java**

```
package prostyprzyklad;  
public class Pracownik extends Osoba {  
    public int numerIdentyfikatora;  
    public static void main(String[] args) {  
        Osoba pewnaOsoba = new Osoba();  
        pewnaOsoba.imie = "Jan";  
        pewnaOsoba.nazwisko = "Kowalski";  
        System.out.println(pewnaOsoba);  
        Pracownik pewienPracownik = new Pracownik();  
        pewienPracownik.imie = "Joanna";  
        pewienPracownik.nazwisko = "Sikorska";  
        pewienPracownik.numerIdentyfikatora = 1234;  
        System.out.println(pewienPracownik);  
    }  
}
```

# Java minimum wiedzy

## ■ Java – minimum wiedzy, cd.

### ■ A co to jest polimorfizm?

- Technika która pozwala na traktowanie obiektów pewnej klasy jako obiektów innej klasy, jeżeli jedna z tych klas dziedziczy po drugiej klasie (pośrednio bądź bezpośrednio)

#### **Osoba.java**

```
package prostyprzyklad;
public class Osoba {
    public String imie;
    public String nazwisko;
    public String toString() {
        return "Osoba " + imie + " "
            + nazwisko;
    }
}
```

#### **Pracownik.java**

```
package prostyprzyklad;
public class Pracownik extends Osoba {
    public int numerIdentyfikatora;
    public String toString() {
        return "Pracownik " + imie + " " + nazwisko +
            ", identyfikator: " + numerIdentyfikatora;
    }
    Osoba innaOsoba = new Pracownik();
    innaOsoba.imie = "Adrian";
    innaOsoba.nazwisko = "Sochacki";
    System.out.println(innaOsoba.toString());
}
```

Tutaj definiując obiekt `innaOsoba` na bazie klasy `Pracownik`, wywoła się metoda `toString()` będąca częścią klasy `Pracownik` – czyli program wypisze:

Pracownik Adrian Sochacki, identyfikator: 0

# Java minimum wiedzy

## ■ Java – minimum wiedzy, cd.

- Wyjątki – programowa metoda sprawdzania zbiorczego działania fragmentów kodu

- Przykład

```
public static int podziel(int a, int b) {  
    return a / b;  
}  
  
...  
int x=10;  
int y=0;  
...  
try {  
    //badana sekwencja instrukcji  
    System.out.println("Wynik dzielenia: " + podziel(x, y));  
} catch (ArithmeticException e) { //obsługa wyjątku związanego z arytmetyką  
    System.out.println("Dzielenie przez zero!");  
} catch (IOException e) {  
    //obsługa innych wyjątków (tu I/O od System.out.println())  
    e.printStackTrace();  
    //zrzut błędu (format użyteczny dla programisty)  
} finally {  
    //kod który trzeba wykonać niezależnie od sukcesu lub porażki  
}
```

# Java minimum wiedzy

## ■ Java – minimum wiedzy, cd.

### ■ Zmienne

- boolean – typ logiczny, przyjmuje wartości true lub false

```
boolean wartoscLogiczna = true;
```

- byte – 8 bitowa liczba całkowita, zakres: -128...127

- short – 16 bitowa liczba całkowita, zakres: -32768...32767

- int – 32 bitowa liczba całkowita, zakres: -2147483648...2147483647

```
int liczbaCalkowita = 10;
```

- long – 64 bitowa liczba całkowita, zakres:  $\sim -2^{63} \dots \sim 2^{63} - 1$

```
long liczba64BitowaCalkowita = 1000000000000L; //uwaga na znak ,L'
```

- float – 32 bitowa liczba zmiennie przecinkowa

```
float liczbaRzeczywista = 3.14f; //uwaga na znak ,f'
```

- double – 64 bitowa liczba zmiennie przecinkowa podwójnej precyzji

```
double liczbaRzeczywista = 2.5;
```

- char – typ znakowy 16 bitowy, kodowany w Unicode

```
char znak = 'A';
```

# Java minimum wiedzy

## ■ Java – minimum wiedzy, cd.

- Tablice zmiennych – ciągi obiektów tego samego typu, których liczba po zdefiniowaniu nie może ulec zmianie

- Definiowanie – składania

```
typ[] nazwa_zmiennej; //uwaga! w C/C++ mieliśmy inną składnię
```

- Przykłady

```
int[] calkowite = new int[5]; //tablica na 5 elementow typu int
```

```
int[] tablica_liczb={1, 2, 10, 3, 14}; //tablica wstepnie zainicjowana 5 elementami typu int
```

```
char[20] imie; //tablica znaków
```

```
String nawisko; //ciąg tekstowych typu String (inny typ niż char[20]!)
```

- Określanie liczby elementów tablicy

```
int liczba_elementow=tablica_liczb.length;
```



# Java minimum wiedzy

## ■ Java – minimum wiedzy, cd.

### ■ Zmienne

#### ■ String – przechowują ciągi znaków

```
String sp1 = new String("Szkola");  
String mu2 = "Uczelnia";  
String[] imiona; //tablica ciągów tekstowych String  
imiona = new String[] { "Ala", "Ela", "Ola" }; //tablica String'ow
```

#### ■ Uwaga! String ma własną metodę określania liczby elementów

```
int dlugosc_nazwy=mu2.length();
```

#### ■ Inne metody obiektu String:

##### ■ .charAt(pos) – podaj znak na pozycji 'pos',

```
char last_char = mu2.charAt(mu2.length() - 1);
```

##### ■ .toLowerCase(), .toUpperCase() – zamiana wszystkich znaków na „małe”/”duże” w obiekcie typu String

```
mu2.toUpperCase(); //zmienna mu2 stanie się „UCZELNIA”
```

- .endsWith(ciąg) – czy kończy się ciągiem (zwraca true), .startsWith(ciąg) – zaczyna się ciągiem, .contains(ciąg) – czy zawiera ciąg
- .equals(ciąg) – czy ciąg i łańcuch są identyczne, equalsIgnoreCase(ciąg) – czy ciąg i łańcuch są identyczne bez rozróżniania wielkości znaków
  - UWAGA! Do porównywania ciągów nie powinno używać się „==” – wynik może być błędny
- Inne metody to: .replace(), .substring(), .split()

# Java minimum wiedzy

## ■ Java – minimum wiedzy, cd.

### ■ Typy generyczne

- Szablony dla tworzenia nowych typów obiektów, przydatne dla tworzenia złożonych typów danych

- UWAGA! Wiele błędów wykrywanych jest dopiero w fazie wykonania a nie fazy kompilacji

### ■ Przykład

```
public class CTyp<T> {
    public T element;
    public CTyp(T element){
        this.element = element;
    }
    public T getElement() { return element; }
    public T setElement(T element) { this.element=element; }
}

public class Main {
    public static void main(String[] args) {
        CTyp<String> v1 = new CTyp<String>(new String());
        CTyp<Int> v2 = new CTyp<>(new Int());          //miedzy znakami <> można nie podawać typu
                                                        //kompilator domyśli się że mowa tu o typie Int

        v1.setElement("Test");
        String t=v1.getElement();
    }
}
```

# Java minimum wiedzy

## ■ Java – minimum wiedzy, cd.

### ■ Petle

#### ■ while()

```
int x=10;
while(x >= 0){
    System.out.println(x);
    x--;
}
```

#### ■ do ... while()

```
int x=10;
do{
    System.out.println(x);
    x--;
}while(x >= 0);
```

#### ■ for(...)

```
for(int x=10; x >=0 ; x--){
    System.out.println(x);
}
```

#### ■ For-each

```
int[] wartosci={ 1, 2, 6, 10};
for(int x : wartosci){
    System.out.println(x);
}
```

## JavaScript – tworzenie mobilnych aplikacji MQTT

# Java minimum wiedzy

## ■ JavaScript – tworzenie mobilnych aplikacji MQTT

- JavaScript to odmiana języka Java dla wsparcia dynamicznych stron WWW

- Podstawa to strona WWW, plik index.html:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

```
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
<head>
```

```
...
```

```
</head>
```

```
<body>
```

```
<h1>Mosquitto Websockets</h1>
```

```
<div>
```

```
<div>Subscribed to <input type='text' id='topic' disabled />
```

```
Status: <input type='text' id='status' size="80" disabled /> </div>
```

```
<ul id='ws' style="font-family: 'Courier New', Courier, monospace;"></ul>
```

```
</div>
```

```
</body>
```

```
</html>
```

Tu wpiszemy wsparcie dla MQTT poprzez WebSocket

Pole gdzie będziemy wpisywali status połączenia

Pole gdzie będziemy wpisywać odebrane komunikaty

# Java minimum wiedzy

## ■ JavaScript – tworzenie mobilnych aplikacji MQTT, cd.

- Aby mosquitto potrafiło dostarczać takie pliki konieczne jest dopisanie w pliku konfiguracyjnym brokera:

```
listener 9001
protocol websockets
http_dir /home/student/mqtt_websocket
```

- W katalogu /home/student/mqtt\_websocket umieszczamy plik index.html, oraz skopiowane pliki: jquery.min.js, paho-mqtt.js w których zapisano niezbędne biblioteki
  - Dołączamy je do kodu w index.html wewnątrz sekcji <head></head>

```
<script src="paho-mqtt.js" type="text/javascript"></script>
<script src="jquery.min.js" type="text/javascript"></script>
```

- A przed tymi liniami (ta sama sekcja):

```
<title>Mosquitto Websockets application</title>
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

- Nadal index.html nie wspiera połączenia, kod następnych kroków wpisujemy w sekcji (wewnątrz <head></head>)

```
<script type="text/javascript">
</script>
```

## ■ JavaScript – tworzenie mobilnych aplikacji MQTT, cd.

### ■ Krok 1 – nawiązanie połączenia i dołączenie funkcji usługowych:

```
var host = '10.0.1.73'; //MQTT broker hostname or IP address
var port = 9001; //MQTT WebSocket TCP port
var topic = '#'; //Temat subskrybowanych wiadomości
var reconnectTimeout = 2000;
var mqtt;
```

Deklaracja obiektów (zamiast słowa kluczowego var, można też używać słowo let)

```
function MQTTconnect() {
  if (typeof path == "undefined") {
    path = '/mqtt'; //obiekt zdefiniowany niedługo (bez var czy let)
  }
  mqtt=new Paho.MQTT.Client(host, port, path, "web_" + parseInt(Math.random() * 100, 10) );
  var options = {
    timeout: 3,
    onSuccess: onConnect,
  };
  mqtt.onConnectionLost = onConnectionLost;
  mqtt.onMessageArrived = onMessageArrived;

  mqtt.connect(options); //właściwe łączenie
}
```

Generowanie losowego ID klienta

Utworzenie instancji klienta

Rejestracja funkcji zwrotnych

## ■ JavaScript – tworzenie mobilnych aplikacji MQTT, cd.

### ■ Krok 2 – oprogramowanie funkcji zwrotnych:

```
function onConnect() {  
    $('#status').val('Connected to ' + host + ':' + port + path);  
    mqtt.subscribe(topic, {qos: 0});  
    $('#topic').val(topic);  
}  
  
function onConnectionLost(response) {  
    setTimeout(MQTTconnect, reconnectTimeout);  
    $('#status').val("connection lost: " + response.errorMessage + ". Reconnecting");  
};  
  
function onMessageArrived(message) {  
    var topic = message.destinationName;  
    var payload = message.payloadString;  
    $('#ws').prepend('<li>' + topic + ' = ' + payload + '</li>');  
};
```

Wpisanie informacji o nawiązaniu połączenia

Subskrypcja po zestawieniu połączenia

Przypomnienie co jest subskrybowane

Ponowne połączenie po zadany czasie

Wypisanie komunikatu o ponowieniu łączenia w polu Status

Wypisanie otrzymanej wiadomości w polu „ws”, kod łączy tekst (tu: '<li>', '=' i '</li>' z treści innych zmiennych (tu topic i payload)

### ■ Krok 3 – nawiązanie połączenia gdy dokument zostanie załadowany do przeglądarki

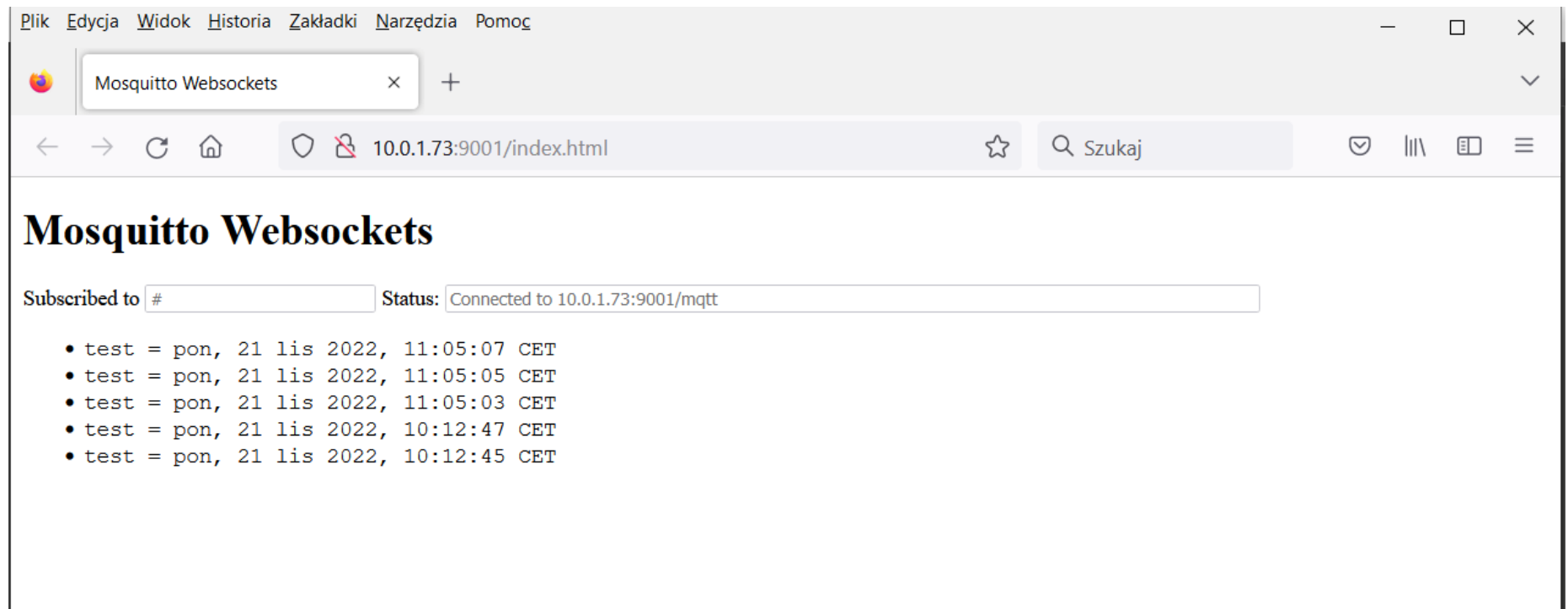
```
$(document).ready(function() {  
    MQTTconnect();  
});
```

Uruchomienie procedury zestawienia połączenia



## ■ JavaScript – tworzenie mobilnych aplikacji MQTT, cd.

- Efekt – do uzyskania poprzez otwarcie w przeglądarce pliku index.html – to wpisujemy adres `http://10.0.1.73:9001/index.html` i otrzymujemy (gdy coś zosatnie opublikowane):



- Opisany przykład pokazuje w uproszczony sposób jak tylko subskrybować – JavaScript daje możliwość wprowadzenia większej interakcji

# Java minimum wiedzy

## ■ JavaScript – tworzenie mobilnych aplikacji MQTT, cd.

- Aby dodać możliwość publikacji na życzenie użytkownika – definiujemy element HTML, zapisując kod w części `<body>...</body>`:

```
<button onclick="myFunction()">Publikuj</button>
```

- Taki wpis sprawi, że gdy użytkownik naciśnie klawisz Publikuj, „aplikacja webowa” wywoła funkcję JS o nazwie `myFunction()`:

```
function myFunction() {  
    mqtt.publish("topic_wiadomosci", "wiadomosc testowa", 0);  
}
```

- Gdybyśmy chcieli publikować cyklicznie – można skorzystać z mechanizmu zdarzeń czasowych – poprzez dopisanie kodu do funkcji obsługi zestawienia połączenia

```
function onConnect() {
```

```
    ...
```

```
    setTimeout(myFunction, 3000);
```

```
}
```

- I zdefiniowaniu `myfunction()`:

```
function myFunction() {  
    mqtt.publish("test_topic", "test", 0);  
    setTimeout(myFunction, 3000);  
}
```

Uruchom `myFunction()` za 3000ms

Ponów uruchomienie `myFunction()` za kolejne 3000ms

Uwaga! można prościej pisać w `onConnet`:

```
setInterval (myFunction, 3000);
```

Podejście takie jednak utrudnia wyłączenie mechanizmu cyklicznego ponawiania wywoływań funkcji gdy zajdą warunki aby zaprzestać tych operacji, np.: gdy otrzyma się określoną wiadomość - w kodzie mógłby być to przypadek:

```
if(payload=="wiadomosc") {  
    //zatrzymaj wywoływanie myFunction()  
}
```

- A podana tu funkcja będzie wywoływać się cyklicznie

### **Zadanie:**

Zbuduj prostą aplikację Webową – jej zadaniem ma być pokazywanie wszystkich wiadomości w oknie przeglądarki oraz gdy treść otrzymanej wiadomości będzie zgodna z wzorcem „start me” ma rozpocząć cykliczne (co 5sek.) publikowanie wiadomości z tekstową reprezentacją liczby opublikowanych dotąd wiadomości, a gdy otrzyma wiadomość „stop me” zakończy proces automatycznego publikowania.

## Java minimum wiedzy

- Dodatek

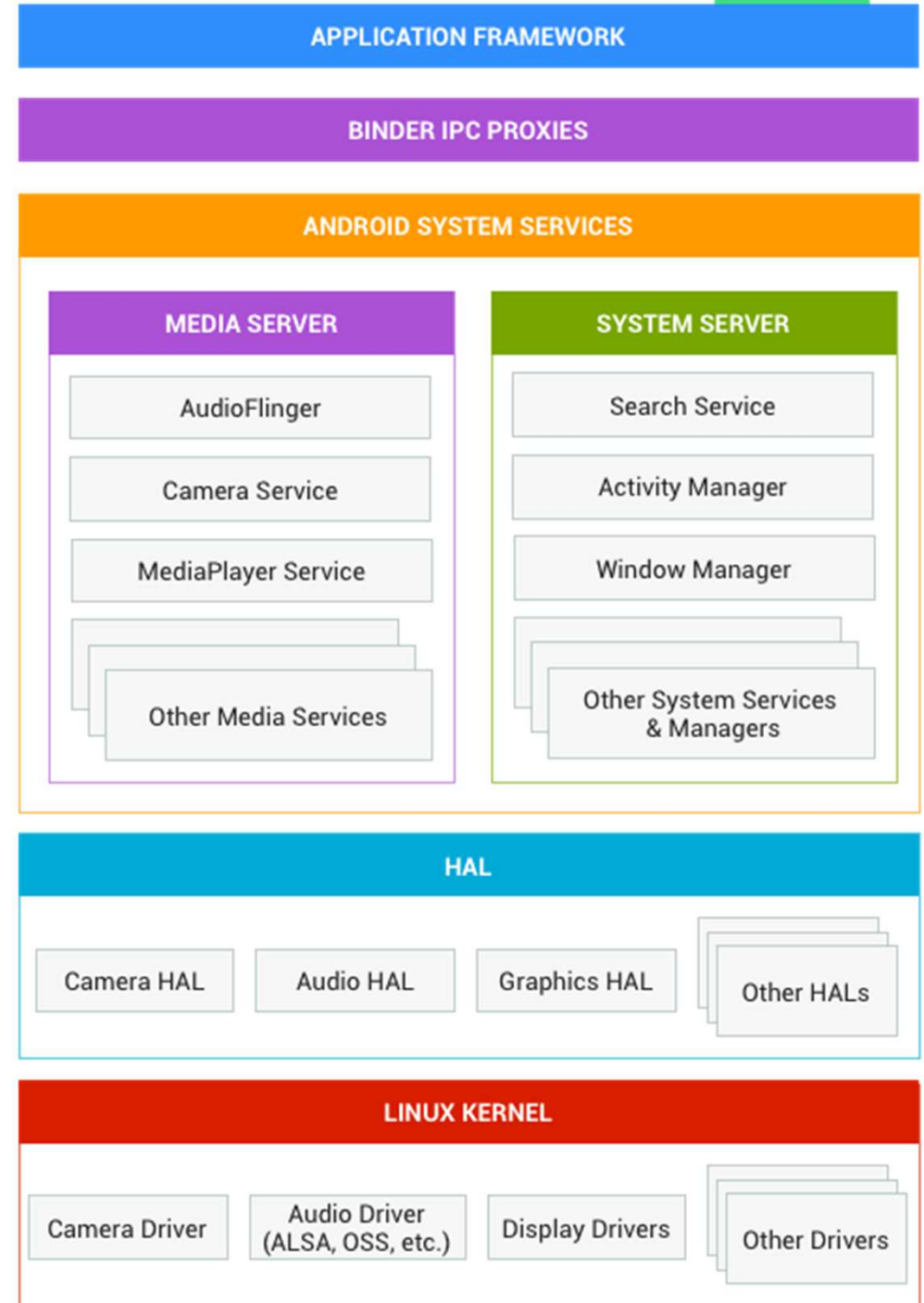
- JavaScript - pomoc opublikowana na stronach  
<https://www.w3schools.com/js/default.asp>

## Android Java – tworzenie mobilnych aplikacji MQTT

# Aplikacje dla systemu Android

## ■ System Android

- Architektura
  - Aplikacje w postaci wirtualnej działają nad systemem Linux
- Aplikacje tworzy się najwygodniej używając Android Studio
  - Do pobrania z <https://developer.android.com/studio>
- Aplikacje dla systemu Android można tworzyć w językach Java lub w Kotlin



## ■ System Android, cd.

### ■ Definicje

- Activity - udostępnia okno, w którym aplikacja rysuje swój interfejs użytkownika.
  - To okno zwykle wypełnia ekran, ale może być mniejsze niż ekran i unosić się nad innymi oknami.
- Intent - w systemie operacyjnym Android to mechanizm programowy, który pozwala użytkownikom koordynować funkcje różnych działań w celu wykonania określonego zadania. Intent to obiekt przesyłania komunikatów, który zapewnia możliwość wykonania wiązania w późnym czasie wykonywania między kodem w różnych aplikacjach w środowisku programistycznym Androida.
  - Jego najbardziej znaczącym zastosowaniem jest uruchamianie Activitis, zapewnia on też system przesyłania wiadomości między aplikacjami, zachęcając je do współpracy i ponownego wykorzystania komponentów.
- Android Service - komponent przeznaczony do wykonywania pewnych czynności bez posiadania interfejsu użytkownika.
  - Service może pobrać plik, odtworzyć muzykę lub zastosować filtr do obrazu. Różne Service mogą być wykorzystywane do komunikacji międzyprocesowej (IPC) między aplikacjami systemu Android.
- Fragment - reprezentuje część interfejsu użytkownika aplikacji (analogia do niezależnego względem głównej aplikacji okienka w systemach PC).
  - Fragment definiuje i zarządza własnym układem, ma własny cykl życia i może obsługiwać własne zdarzenia wejściowe. Fragmenty nie mogą żyć samodzielnie.



# Aplikacje dla systemu Android

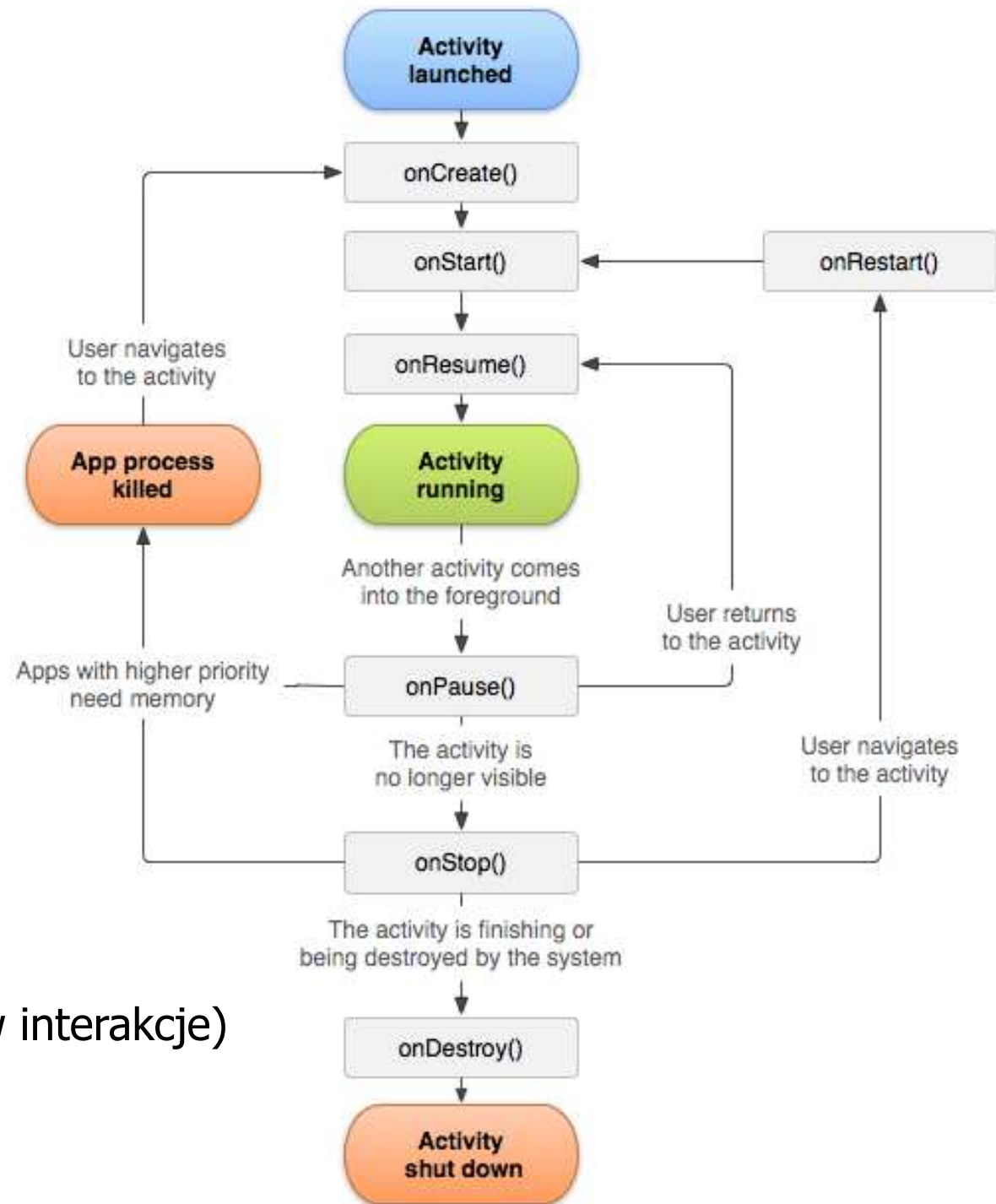
## ■ System Android, cd.

### ■ Życie aktywności

■ Rysunek przedstawia całościowe życie Aktywności

- Prostokąty – metody jakie można zaimplementować aby Aktywność poprawnie mogła reagować na zmianę stanu
- Owale stany w jakich może znaleźć się Aktywność

■ Generalnie w aplikacjach dla Androida, Activity jest tym na czym w danej chwili użytkownik się koncertuje (widzi, wchodzi w interakcje)





# Aplikacje dla systemu Android

## ■ System Android, cd.

### ■ Życie aktywności, cd.

#### ■ Aktywność posiadać może trzy główne pętle

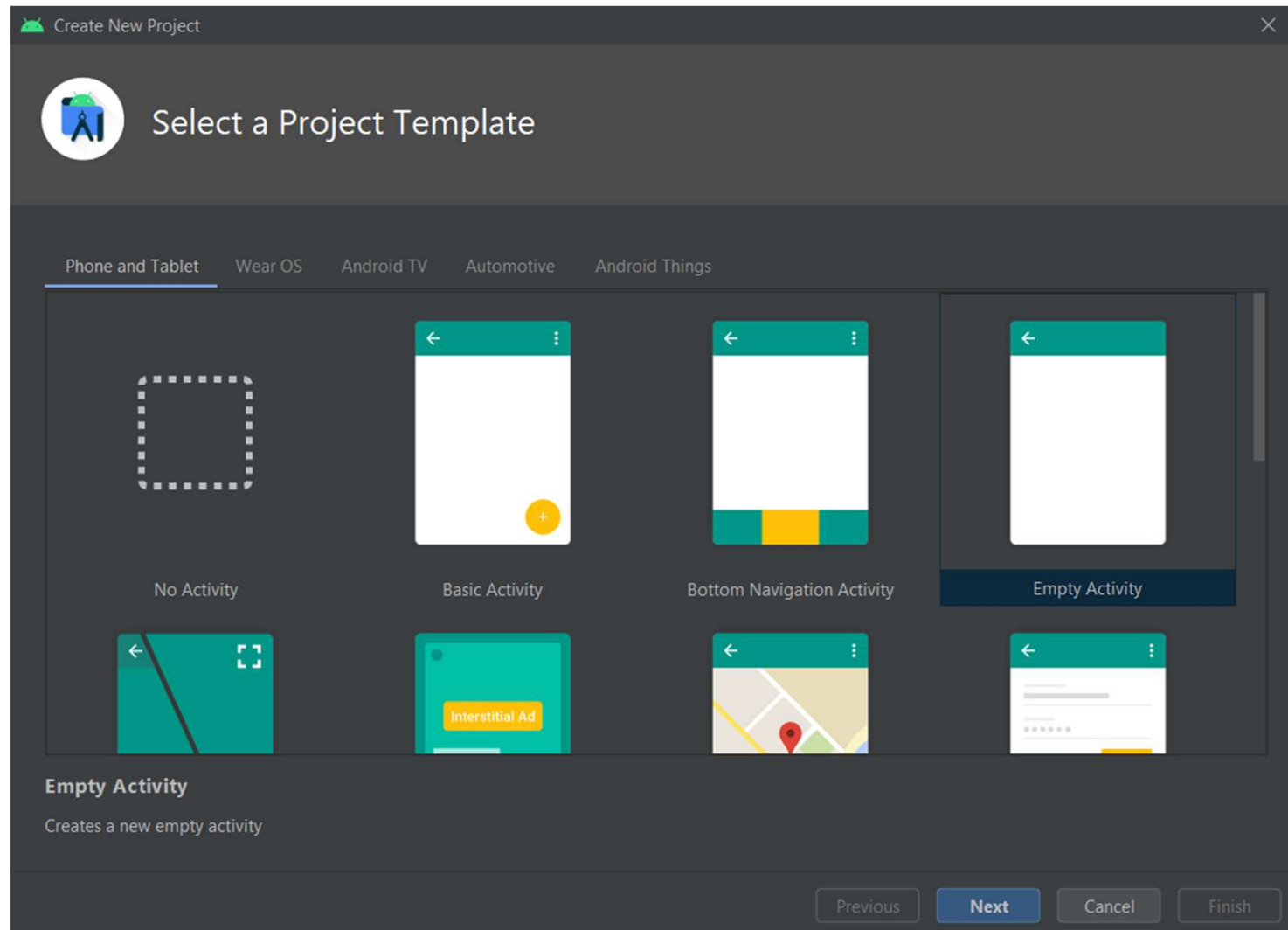
- Całkowite życie Aktywności – to czas między wywołaniem onCreate() a zakończeniem życia Aktywności przez wywołanie onDestroy()
  - onCreate() – ustala np.: „globalne” ustawienia aplikacji
  - onDestroy() – zwalnia wszelkie zasoby, zamyka połączenia
- „Widzialne” życie Aktywności – to czas między wywołaniem onStart() a onStop(), podczas tego czasu użytkownik będzie widział ewentualne zmiany na ekranie
  - Wywołania onStart() i onStop() mogą być realizowane przez system wielokrotnie podczas życia Aktywności – co będzie odpowiadało chwilom gdy Aktywność będzie pojawiać się i znikać przed oczami użytkownika
- Aktywność życząca w tle – czas między wywołaniem onPause() a onResume(), wtedy to Aktywność nie ma kontaktu z użytkownikiem (np.: urządzenie śpi) ale aplikacja nadal pracuje

## ■ Wiemy wszystko – to stworzymy proste projekty (aplikacje)!

### ■ Używamy Android Studio

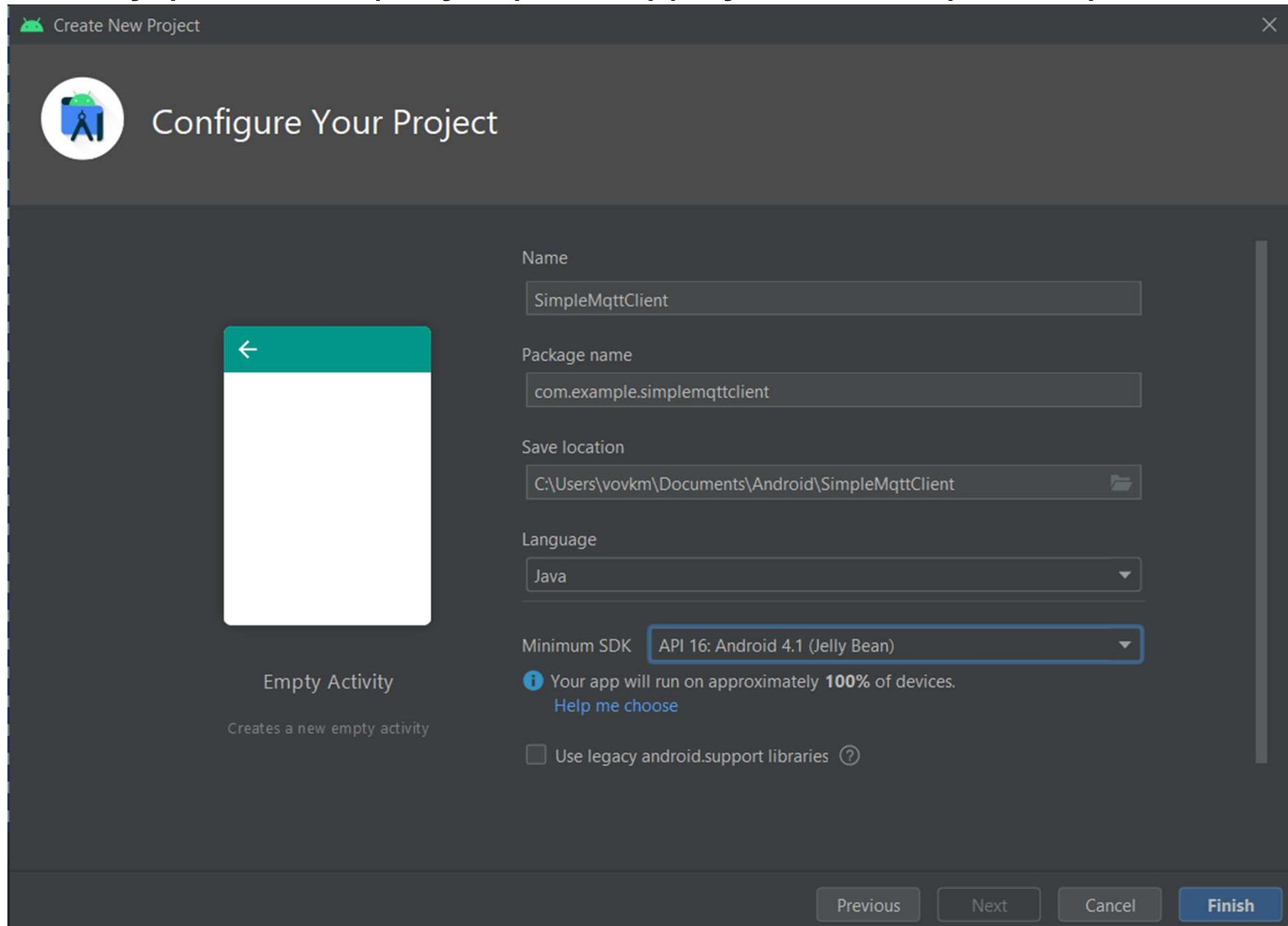
# Aplikacje dla systemu Android

- Android Studio - tworzenie nowego projektu (APP001)
  - ... Wybieramy „Empty Activity” do tworzenia zwykłej aplikacji z pustym ekranem głównym



## Aplikacje dla systemu Android

- Android Studio - tworzenie nowego projektu (APP001), cd.
  - W kolejnym kroku wpisujemy nazwę projektu oraz wybieramy API Android'a



# Aplikacje dla systemu Android

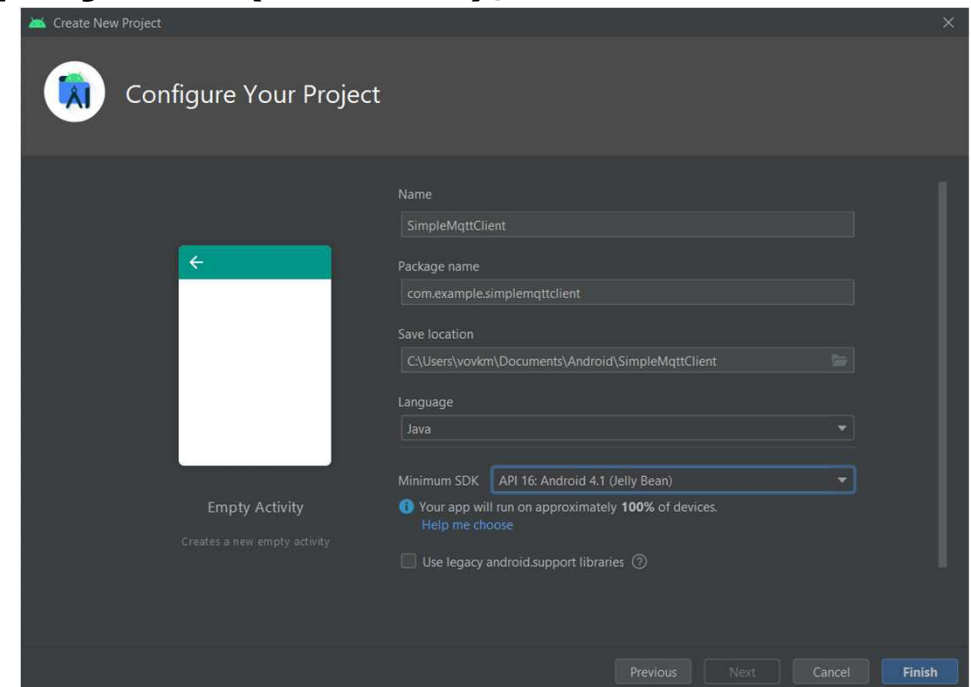
## ■ Android Studio - tworzenie nowego projektu (APP001), cd.

- Pola ustawione są domyślnie – choć to poprawne warto jednak zwrócić uwagę na: Name (nazwa klasy głównej i nazwa aplikacji), Minimum SDK

### ■ Jak wybierać SDK

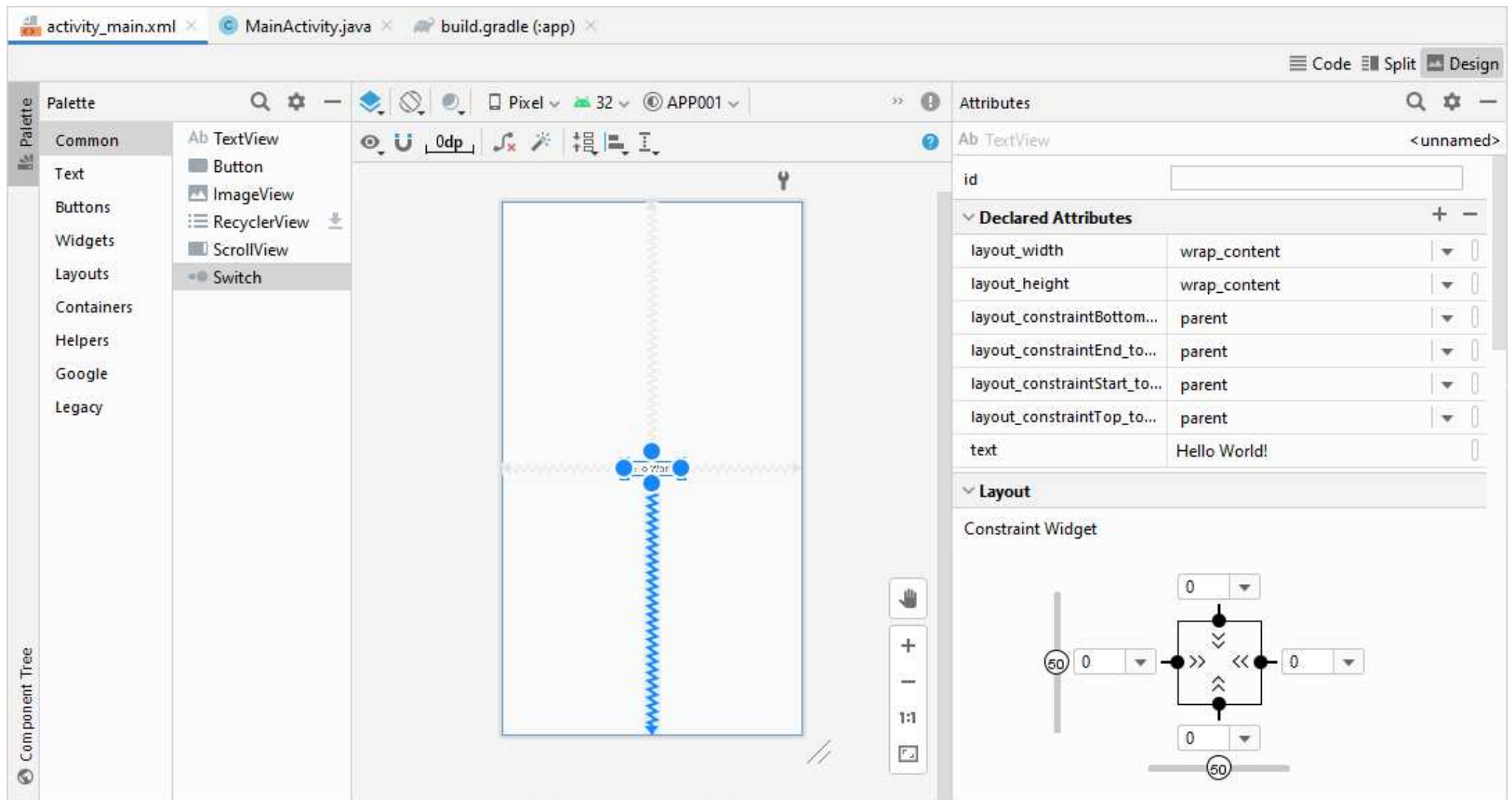
W konfiguracji nowego projektu wybierany był minimalny numer SDK z jakim nasza aplikacja ma działać

- Dla nowego projektu wybieramy stabilną wersję SDK lub/i wersję w której będą wspierane wszystkie pożądane w naszej aplikacji funkcjonalności
- Dla komercyjnych zastosowań - warto zwrócić uwagę na fakt jak wiele platform sprzętowych wspiera wybrane SDK – czyli jak wielu klientów będzie móc korzystać z naszej aplikacji



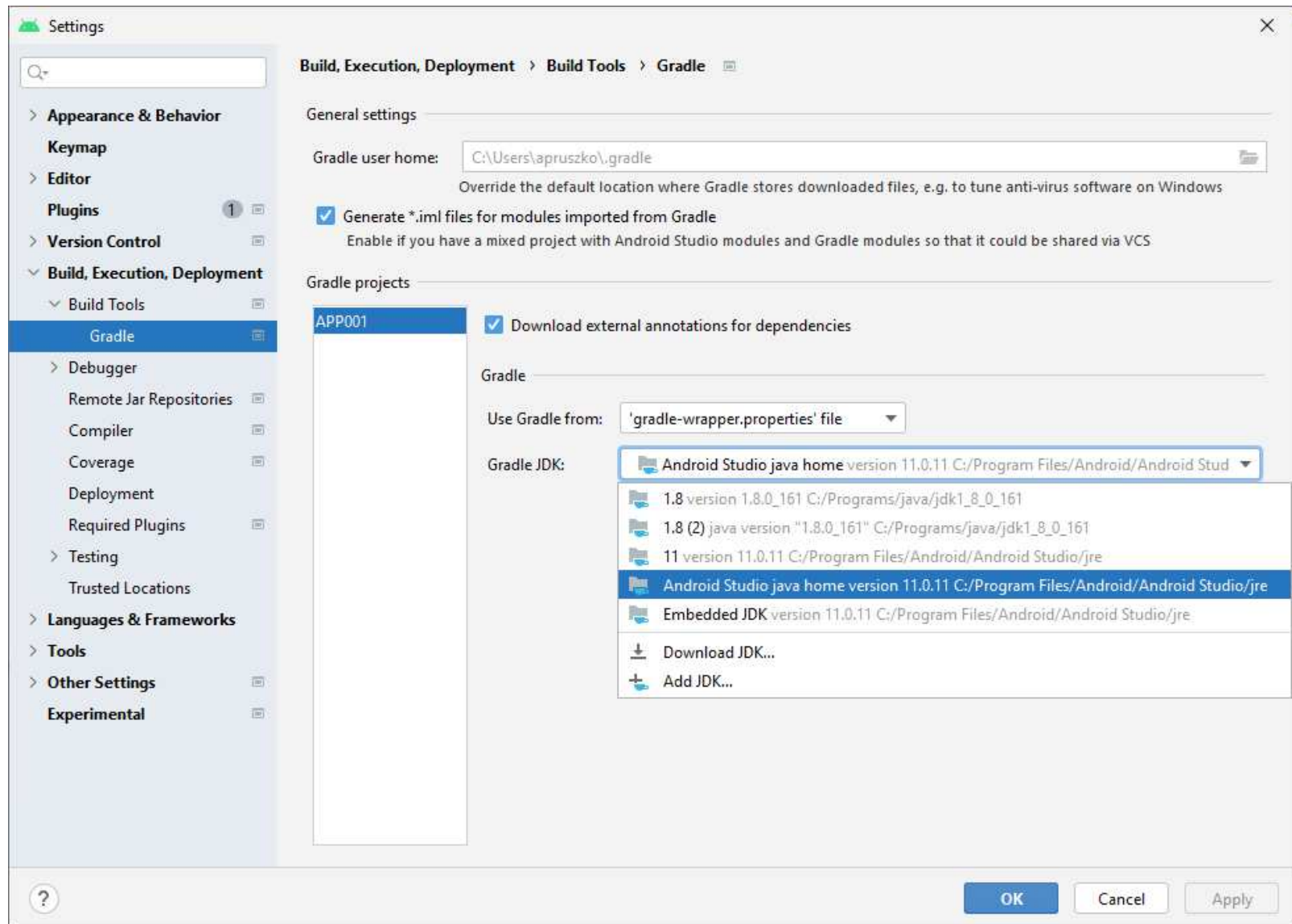
# Aplikacje dla systemu Android

- Android Studio - tworzenie nowego projektu (APP001), cd.
  - Po tych krokach, tworzy się zwykła aplikacja z polem tekstowym „Hello world” na początkowym ekranie (widok z Design Buildera)



# Aplikacje dla systemu Android

- Android Studio - tworzenie nowego projektu (APP001), cd.
  - Problemy z Javą w nie właściwej wersji – możemy użyć JDK przenieszonego z Android Studio



# Aplikacje dla systemu Android

## ■ Android Studio - tworzenie nowego projektu (APP001), cd.

### Kod Aplikacji

```
package com.example.app001;
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);
    }
}
```

### Kod pliku activity\_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android=http://schemas.android.com/...
    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" />
</android.support.constraint.ConstraintLayout>
```

Tu mamy faktyczny tekst

- Android Studio - tworzenie nowego projektu (APP001), cd.
  - Jak się ma `R.layout.activity_main` do opisu w pliku XML?
    - Plik XML po kompilacji jest łączony z kodem głównym, dla ułatwienia:
      - R – klasa opisująca wszelkie zasoby
      - layout – klasa dziedziczona z „R”, z wszelkimi zdefiniowanymi „wyglądami”
      - activity\_main – klasa dziedziczona z „layout”, z opisami dotyczącymi głównej aktywności



# Aplikacje dla systemu Android

## ■ Android Studio - tworzenie nowego projektu (APP001), cd.

### ■ A co ten kod wykonuje?

```
package com.example.app001;
```

Nazwa głównego elementu

```
import android.support.v7.app.AppCompatActivity;
```

Importowane klasy

```
import android.os.Bundle;
```

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

Główna klasa tej aplikacji  
– rozszerza  
AppCompatActivity

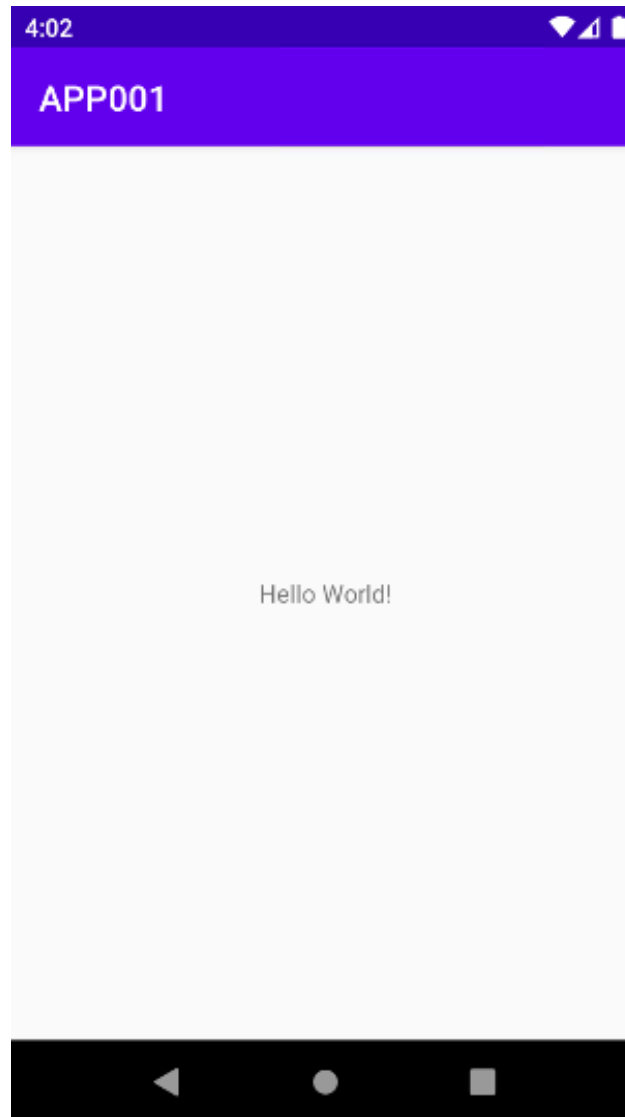
Metoda onCreate() -  
odpowiednik funkcji main()  
w aplikacjach –  
uruchamiana  
podczas tworzenia instancji  
aplikacji

W tej aplikacji wskazujemy, że zasób activity\_main opisuje to co użytkownikowi trzeba pokazać w ramach głównego Activity

Proszę zauważyć, że metoda onCreate() z klasy nadrzędnej AppCompatActivity jest tu jawnie wywoływana

## Aplikacje dla systemu Android

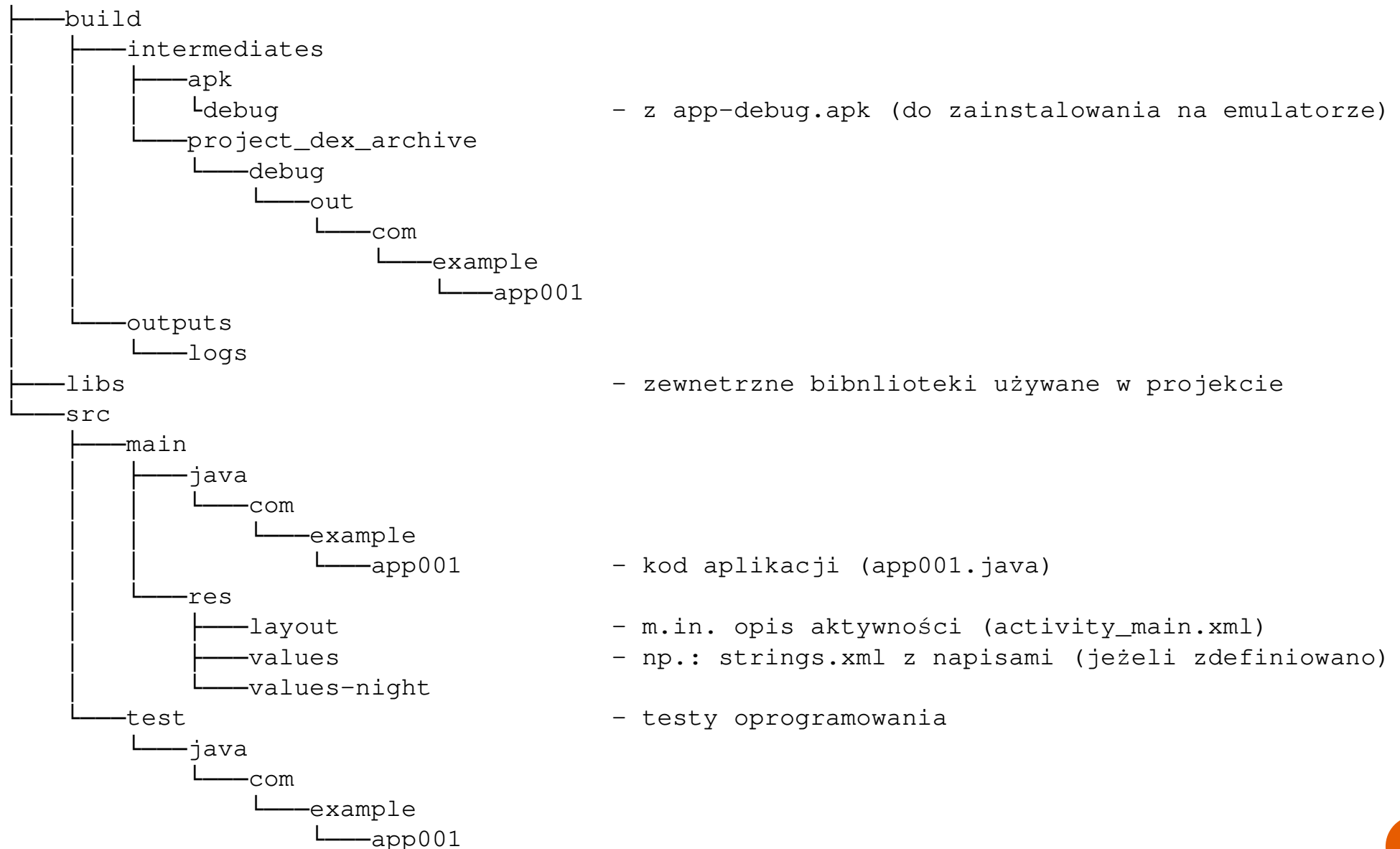
- Android Studio - tworzenie nowego projektu (APP001), cd.
  - Przykład działania aplikacji „Hello world” na emulatorze (Run->Run 'app')



# Aplikacje dla systemu Android

## ■ Android Studio - tworzenie nowego projektu (APP001), cd.

### ■ Struktura projektu (ważniejsze elementy)

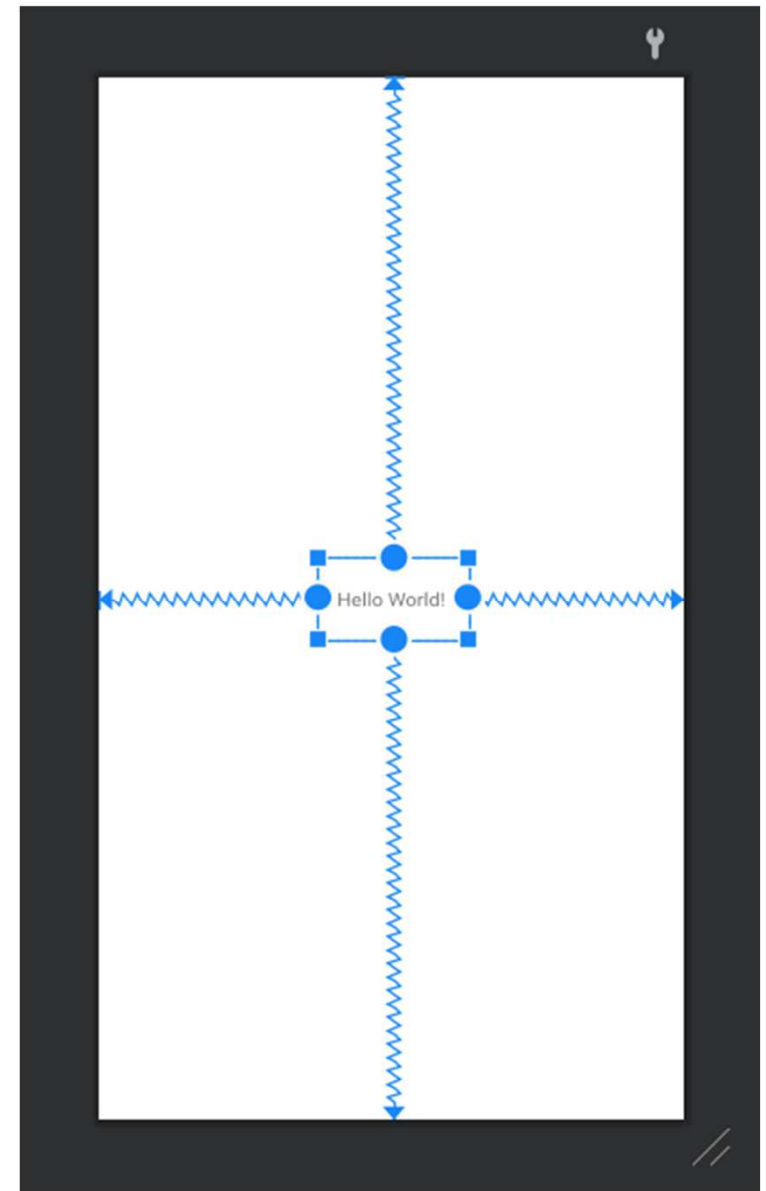
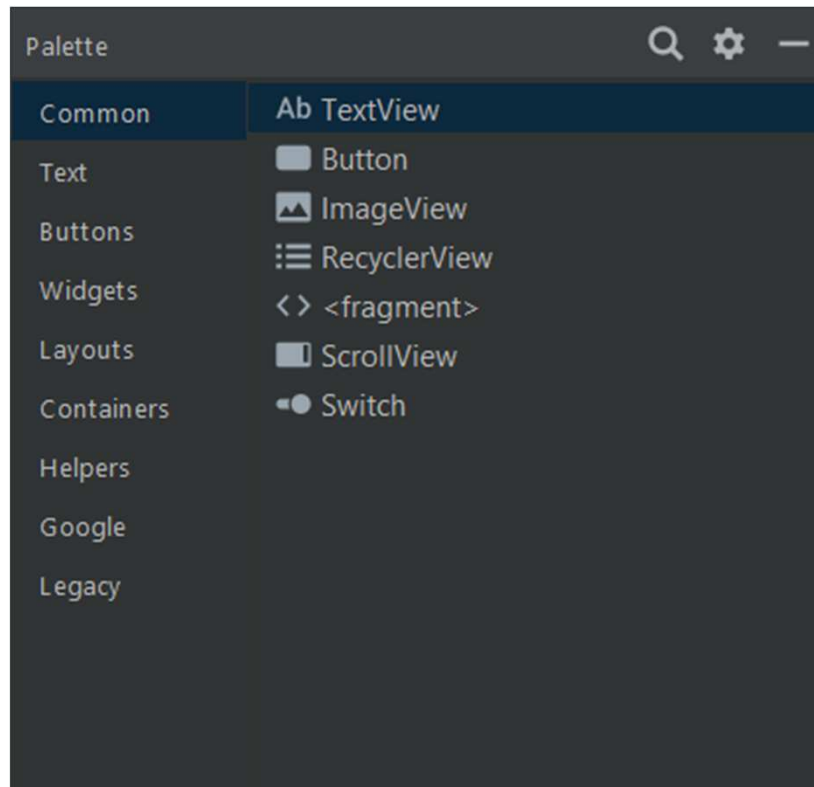


**Zadanie:**

Zbuduj prostą aplikację „Witam!!!” (aplikacja ma zawierać jeden ekran z jednym prostym napisem)

# Aplikacje dla systemu Android

- Tworzenie nowego projektu (APP002) - „Design Builder”
  - Narzędzie wspierające tworzenie interfejsów aplikacji
    - Narzędzie wbudowane w Android Studio
    - Pozwala dodawać elementy do interfejsu z użytkownikiem:
      - Wybieramy element i układamy „go na ekranie telefonu”

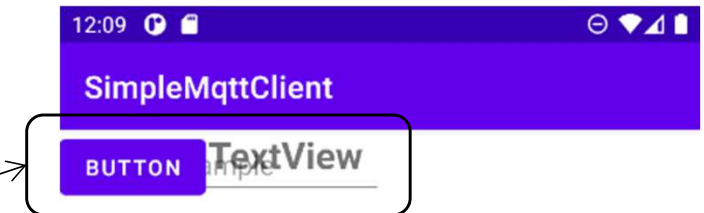


# Aplikacje dla systemu Android

## ■ APP002 - „Design Builder”, cd.

- Najczęściej wykorzystywane są
  - TextView – pole wyświetlania tekstu (można z poziomu aplikacji je modyfikować)
  - PlainText – pole które użytkownik telefonu może edytować
  - Button – przycisk

- Wybierając i przeciągając je na ekran telefonu w Design Bulder otrzymamy:

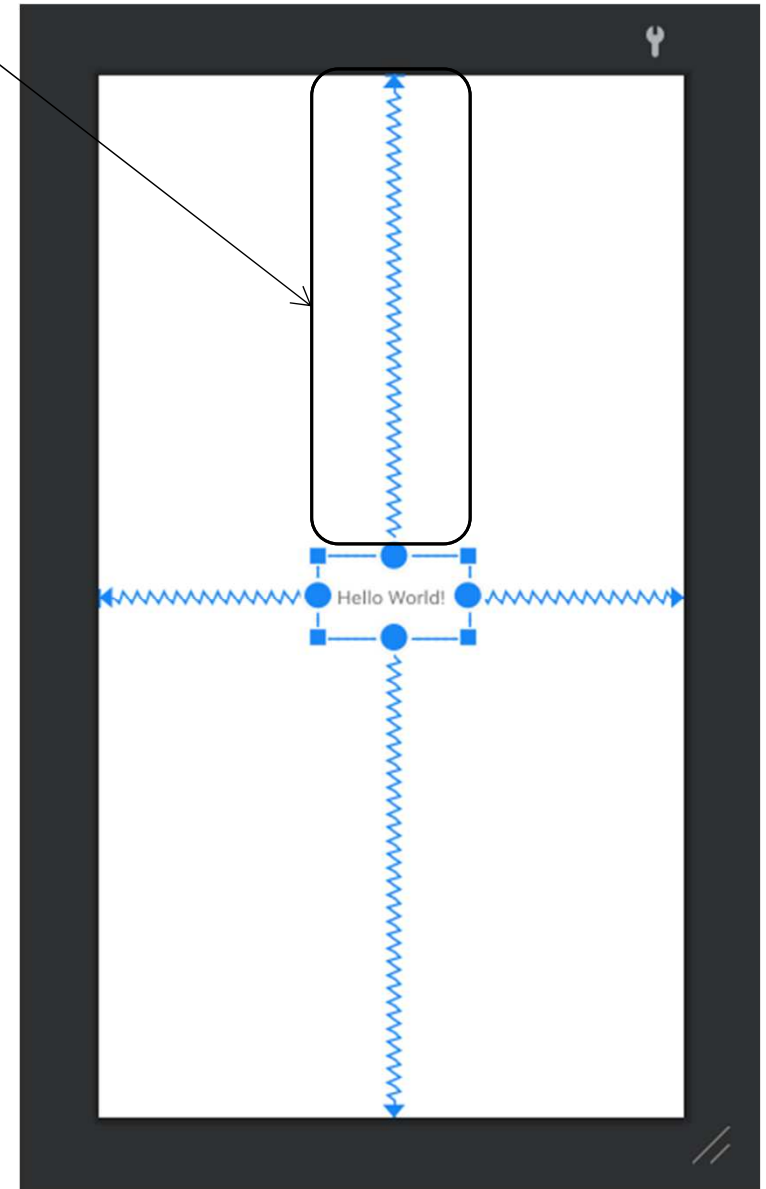
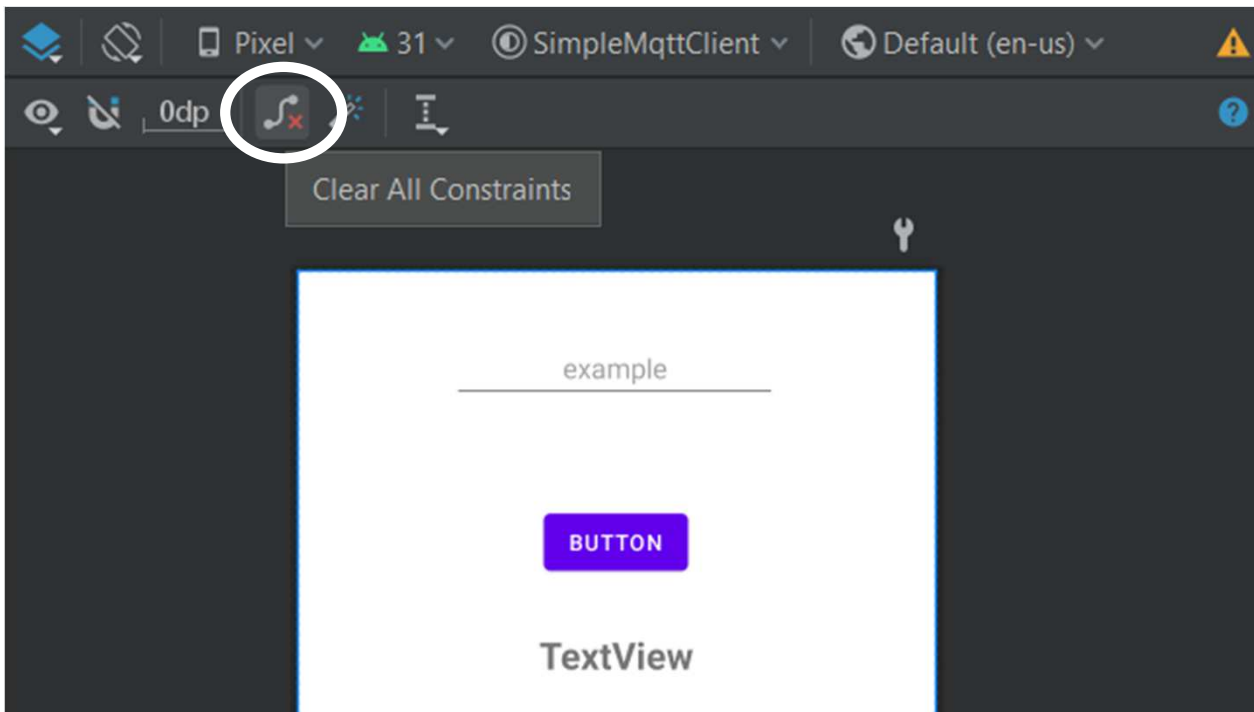


- Beładną zbieraninę elementów
  - Jednak Design Builder pozwala na tworzenie tzw. adaptacyjnego interfejsu z możliwością określenia zależności między elementami (jak daleko elementy mają być od siebie oddalone, ...)
- Proszę pamiętać, że w plikach XML w katalogu res/ znajdziemy finalne definicje interfejsu
  - Tam można znaleźć wiele plików – aplikacja może posiadać wiele „ekranów”!

# Aplikacje dla systemu Android

## ■ APP002 - „Design Builder”, cd.

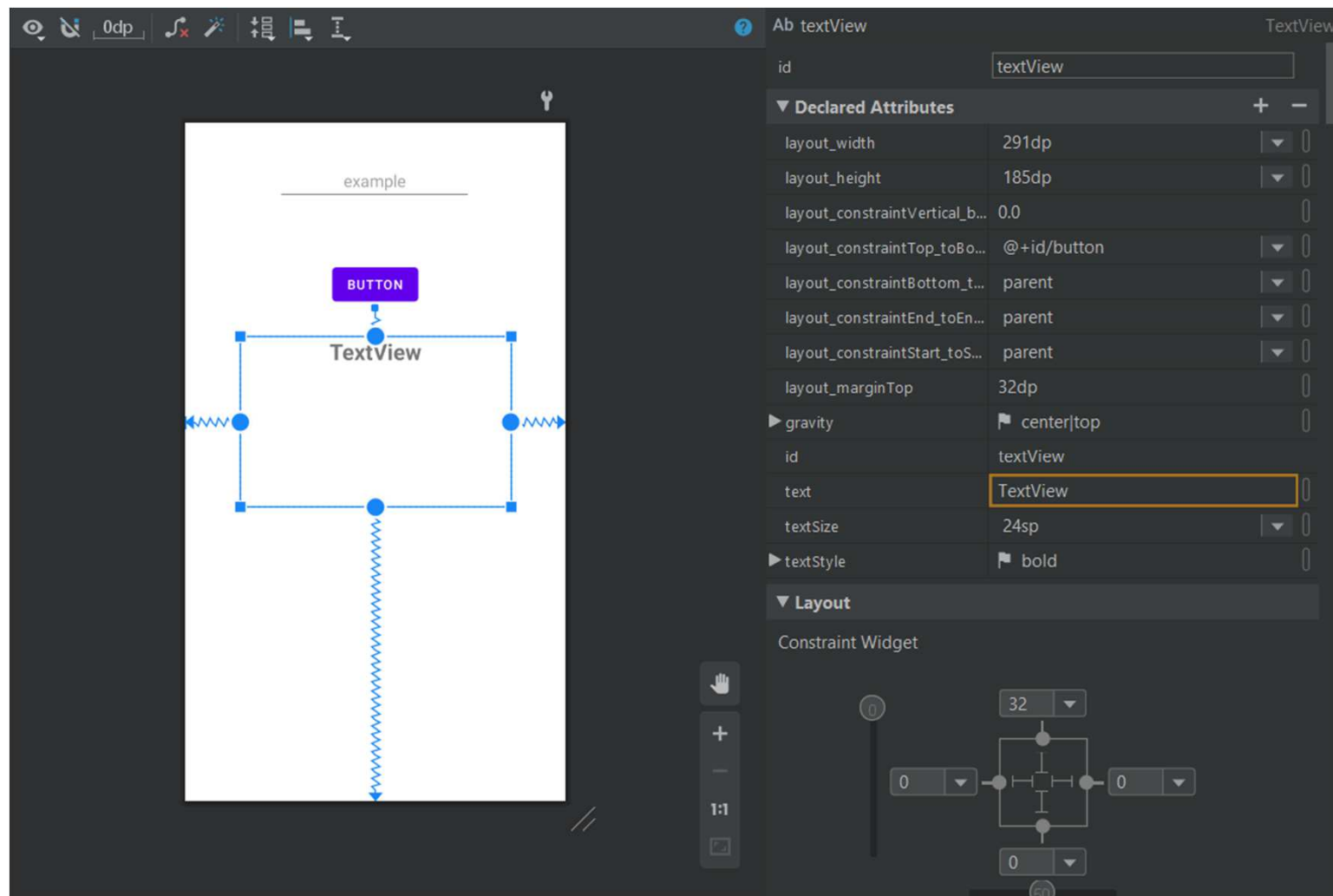
- Narzędzie pozwala zarządzać elementami
  - Ich wielkości, domyślne ustawienia, powiązania
- TIPS: Narzędzie pozwala automatycznie usunąć wszystkie odstępy pomiędzy elementami za pomocą przycisku „Clear All Constraints”
  - proszę pamiętać że ta funkcjonalność może popsuć to co z trudem udało się ustawić ręcznie



# Aplikacje dla systemu Android

## ■ APP002 - „Design Builder”, cd.

- Zmiana atrybutów określonego elementu
  - Np. pole TextView ma wśród swoich atrybutów możliwość wybrania wielkości użytej czcionki, oraz istnieje możliwość wypełnienia jej predefiniowaną treścią



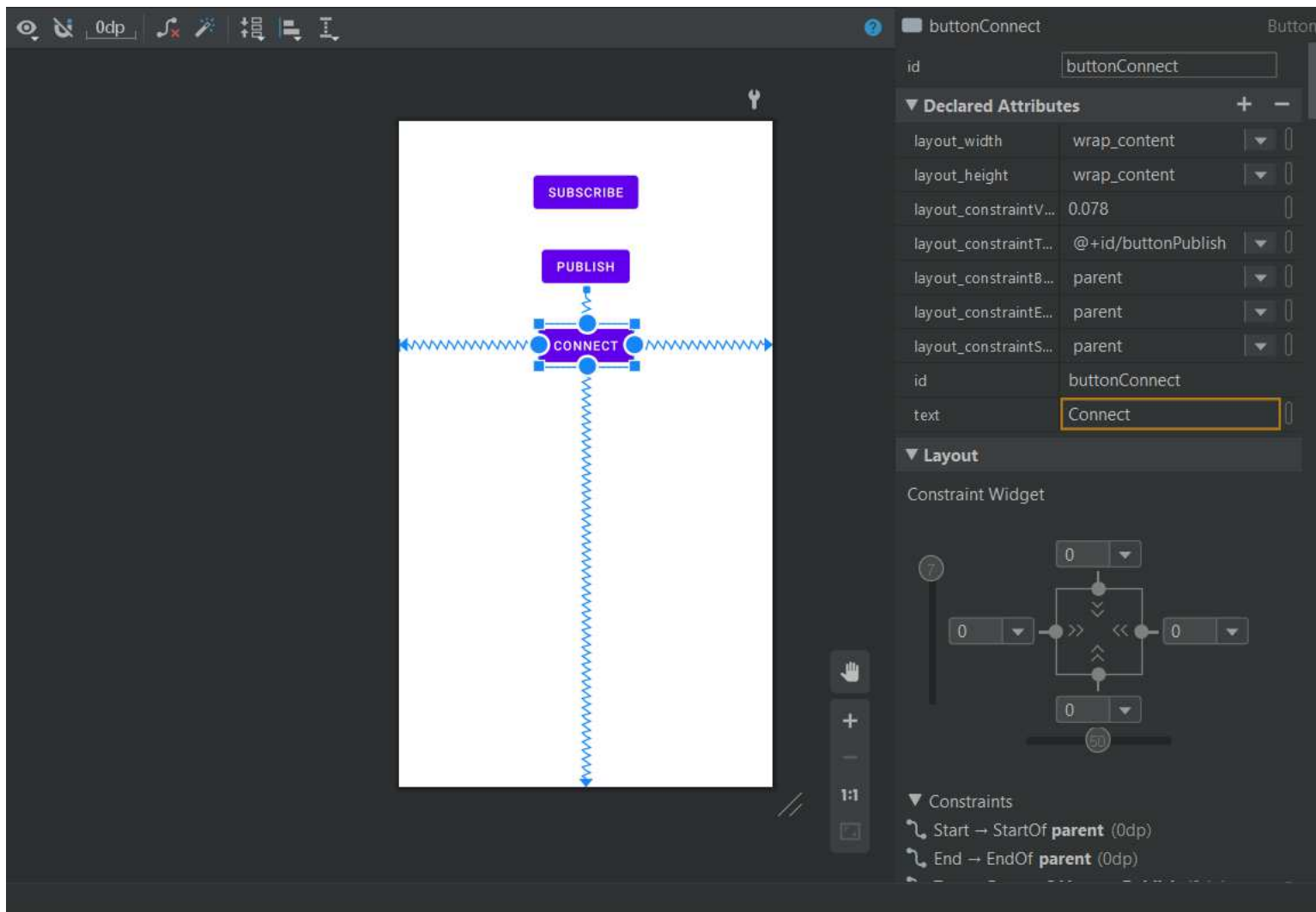


### **Zadanie:**

Zbuduj prostą aplikację: z polem tekstowym wypełnionym stałym krótkim tekstem, klawiszem o nazwie „Connect me” i polem edycyjnym które mogłoby być użytecznym do wpisania tzw. adresu FQDN brokera mosquitto (ta aplikacja nie ma obsługi reakcji na naciśnięcie klawiszy i nie podejmuje się zestawianie połączenia MQTT)

# Aplikacje dla systemu Android

- Interakcja z użytkownikiem - obsługa przycisków (APP003)
  - Dla obsługi przycisków (Buttons) Android Studio za pomocą Design Builder potrafi automatycznie nadać unikatowe w ramach aplikacji identyfikatory (tu: id: buttonConnect) i pozwala nadawać czytelne dla człowieka nazwy (tu: Connect)



## Aplikacje dla systemu Android

- Interakcja z użytkownikiem - obsługa przycisków (APP003), cd.
  - Po zdefiniowaniu wyglądu przycisku taki element trzeba obsłużyć w tworzonym oprogramowaniu:
    - Każdy z dodawanych elementów interfejsu użytkownika przypisujemy do właściwego obiektu klasy Java (tu: Button), np.:

```
private Button buttonConnect;
```

- Mechanizmy wewnętrzne systemu Android wspierają automatyzację nadawania ID dla obiektów UI i pomagają w oprogramowaniu wykorzystać te ID poprzez metodę `findViewById()` :

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

- W kolejnym kroku należy po uruchomieniu aplikacji (tu w metodzie: `onCreate()`) obsłużyć akcję związaną z klawiszem:

```
buttonConnect.setOnClickListener(  
    new View.OnClickListener() {  
        @Override  
        public void onClick(View v) {  
            //...  
        }  
    }  
);
```

## ■ Interakcja z użytkownikiem - obsługa przycisków (APP003), cd.

- Podczas pracy aplikacji niektóre atrybuty przycisków można modyfikować – tutaj np.: po naciśnięciu klawisza zmienić napis z nim związany

```
new View.OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        if (isConnected) {  
            isConnected = false;  
            buttonConnect.setText("DISCONNECT");  
        } else {  
            isConnected = true;  
            buttonConnect.setText("CONNECT");  
        }  
    }  
};
```

W realnych aplikacjach ta flaga musi być odpowiednio zdefiniowana i odpowiednio zgodnie ze stanem połączenia ustawiana/kasowana – tutaj pozwala wyłącznie zorientować się aplikacji w swoim działaniu

- Tutaj oprócz zmiany napisu zapisujemy jaki stan odpowiada aktualnemu działaniu aplikacji – w przypadku zastosowania łączności np.: MQTT tutaj także trzeba byłoby wykonać akcję związaną z nawiązaniem połączenia lub jego zerwaniem
  - Proszę pamiętać że w zaawansowanych aplikacjach takie operację mogą być zlecane tzw. wątkom których działanie odciąża główny kod aplikacji

## Obsługa połączenia MQTT (APP004)

# Aplikacje dla systemu Android

## ■ Interakcja z użytkownikiem - obsługa połączenia MQTT (APP004)

- Aplikacja pod system Android dla łączności z brokrem MQTT wymaga dołączenia do jej kodu biblioteki Paho – poprzez:

- w pliku build.gradle (opis projektu) w sekcji repositories{...}

```
maven {  
    url "https://repo.eclipse.org/content/repositories/paho-snapshots/"  
}
```

- w sekcji dependencies{...}

```
implementation 'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.0.2'
```

- oraz w pliku AndroidManifest.xml definiującym wymagane uprawnienia aplikacji:

```
<uses-permission android:name="android.permission.INTERNET" />
```

- pozwala aplikacji na otwieranie tzw. sieciowych gniazd (network sockets)

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

- pozwala aplikacji na dostęp do informacji o sieciach (np.: czy mamy jakąś łączność)

```
<uses-permission android:name="android.permission.WAKE_LOCK" />
```

- pozwala tzw. zarządcy energii (PowerManager) - dla tworzonej aplikacji - blokować mechanizmy oszczędzania energii (usypianie CPU, przyciemnianie ekranu, itp.)

- Więcej informacji na <https://developer.android.com/reference/android/Manifest.permission>

### ■ Interakcja z użytkownikiem - obsługa połączenia MQTT (APP004), cd.

- Kod aplikacji najlepiej tworzyć poprzez wykreowanie interfejsu MqttCallback (plik MqttCallbackImpl.java)

```
public class MqttCallbackImpl implements MqttCallback { ... }
```

- A w nim metody obowiązkowe
  - Odpowiedzialna za obsługę otrzymanych wiadomości

```
@Override
```

```
public void messageArrived(String t, MqttMessage message) throws Exception {  
    //... - dla testów to może pozostać puste  
}
```

- Przekazująca sterowanie aplikacji gdy zerwane zostanie połączenie z brokerem

```
@Override
```

```
public void connectionLost(Throwable cause) {  
    //... - dla testów to może pozostać puste  
}
```

- Informująca o dostarczeniu wiadomości do brokera

```
@Override
```

```
public void deliveryComplete(IMqttDeliveryToken token) {  
    //... - dla testów to może pozostać puste  
}
```

- Interakcja z użytkownikiem - obsługa połączenia MQTT (APP004), cd.
  - Warto dodać kod obsługujący proces nawiązywania połączenia i jego zamykania o implementacji:

```
MqttClient client; //pole prywatne klasy MqttCallbackImpl
public boolean disconnect() {
    try {
        client.disconnect(); return true;
    } catch (MqttException e) { return false;
    }
}

public boolean connect(String domain, int port, MqttConnectOptions option) { ...
    try {
        client = new MqttClient("tcp://" + domain + ":" + port,
                                MqttClient.generateClientId(),
                                new MemoryPersistence() );
        client.setCallback(this);
        client.connect(mqttConnectOptions);
        return true;
    } catch (MqttException e) {
        return false;
    }
}
```

Wskazanie gdzie będzie obiekt obsługujący wiadomości do wysłania lub te odebrane, gdy ich dłuższe przechowywanie wymagane będzie przez QoS protokołu MQTT



### ■ Interakcja z użytkownikiem - obsługa połączenia MQTT (APP004), cd.

- Aby kod w języku Java mógł poddać się kompilacji - należy dodać odpowiednie biblioteki do implementacji w pliku `MqttCallbackImpl.java`:

```
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;  
import org.eclipse.paho.client.mqttv3.MqttAsyncClient;  
import org.eclipse.paho.client.mqttv3.MqttCallback;  
import org.eclipse.paho.client.mqttv3.MqttClient;  
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;  
import org.eclipse.paho.client.mqttv3.MqttException;  
import org.eclipse.paho.client.mqttv3.MqttMessage;  
import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;
```

- Jak widać wszystkie zaimplementowane są w bibliotece Paho

### ■ Interakcja z użytkownikiem - obsługa połączenia MQTT (APP004), cd.

- Aby pracować z połączeniem MQTT w kodzie głównej aktywności - trzeba jeszcze dodać wsparcie dla opcji związanych z połączeniem – import biblioteki

```
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
```

- Oraz zmodyfikować główna aktywność – dodając obiekt klasy MqttCallbackImpl

```
MqttCallbackImpl mqttCallback;
```

- A w metodzie onClick() związaną z klawiszem „Connect” dodać właściwą reakcję

```
boolean isConnected = false;
```

```
public void onClick(View view) {
```

```
    if (!isConnected) {
```

```
        mqttCallback = new MqttCallbackImpl();
```

```
        MqttConnectOptions mqttConnectOptions = new MqttConnectOptions();
```

```
        mqttConnectOptions.setUserName("rw");
```

```
        mqttConnectOptions.setPassword("readwrite".toCharArray());
```

```
        if (mqttCallback.connect("test.mosquitto.org", 1884, mqttConnectOptions)) {
```

```
            isConnected = true; //połączenie udało się zestawić
```

```
            //...
```

```
        }
```

```
    } else {
```

```
        if(mqttCallback.disconnect()) { isConnected=false; //...
```

```
    }}}
```

### ■ Interakcja z użytkownikiem - obsługa połączenia MQTT (APP004), cd.

- Opcje połączenia – w podanym przykładzie korzystamy z testowego brokera
  - Dostępny jest on publicznie, łączymy się z nim poprzez wywołanie:

```
if (mqttCallback.connect("test.mosquitto.org", 1884, mqttConnectOptions)) {  
    ... }
```

- Proszę pamiętać o obsłudze sytuacji gdy system odmówi połączenie (np.: adres mógłby być błędny, broker nie przyjmuje określonego użytkownika, ...)
- Broker publiczny pozwala dokonywać prób z oprogramowaniem, ale łącząc się na porcie 1884 TCP system wymaga od nas uwierzytelnienia się - co w aplikacji realizujemy przez klasę `MqttConnectOptions` zadeklarowaną jako:

```
MqttConnectOptions mqttConnectOptions = new MqttConnectOptions();
```

- Login ustawienia się na nazwę „rw” (takiego użytkownika wpuści powyższy broker) za pomocą:  
`mqttConnectOptions.setUsername("rw");`

- Hasło na „readwrite” poprzez wywołanie:

```
mqttConnectOptions.setPassword("readwrite".toCharArray());
```

- Metoda `toCharArray()` dokonuje konwersji z `String` do ciągu tekstowego

**Zadanie:**

Zbuduj prostą aplikację MQTT łączącą się z użyciem uwierzytelnień ze zbudowanym przez Ciebie brokerem Mosquitto (na tym etapie bez wymiany wiadomości, a napis na klawiszu „Connect” po pomyślnym zestawieniu połączenia zmieni się na „Disconnect”)

## ■ Interakcja z użytkownikiem – interakcja z MQTT (APP005)

- W poprzednio tworzonej aplikacji komunikowała się ona z użytkownikiem wyłącznie zmieniając napis na klawiszu (CONNECT zamieniano na DISCONNECT)
  - Podejście to jest mało użyteczne użytkownikowi
- Pełna integracja biblioteki Paho-MQTT z interfejsem użytkownika (UI) jest bardziej skomplikowana
  - Wynika to z faktu że wątki (a na nich bazuje biblioteka) nie powinny bezpośrednio „używać” interfejsu użytkownika
- Pomocna tu może stać się technologia LiveData
  - Aby tworzona aplikacja mogła ją używać należy w pliku build.gradle dodać:

```
implementation 'android.arch.lifecycle:livedata:1.1.0'
```

```
implementation 'android.arch.lifecycle:viewmodel:1.1.0'
```

```
implementation 'androidx.appcompat:appcompat:1.1.1'
```

- A w kodzie Java zaimportować biblioteki:

```
import androidx.lifecycle.Observer;
```

```
import androidx.lifecycle.ViewModelProvider;
```

### ■ Interakcja z użytkownikiem – interakcja z MQTT (APP005), cd.

- W klasie głównej aktywności dodajemy prywatny obiekt:

```
private MainViewModel mainViewModel;
```

- Implementacja tej klasy – patrz następny slajd(!)

- A podczas startu aktywności (onCreate()) utworzy właściwy obiekt:

```
mainViewModel=new ViewModelProvider(this,  
    new ViewModelProvider.NewInstanceFactory()).get(MainViewModel.class);
```

- Obiekt ten połączymy z MqttCallbackImpl ale z nieco zmienionym konstruktorem:

```
mqttCallback=new MqttCallbackImpl(MainActivity.this, mainViewModel.getmText() );
```

- Dzięki tym zabiegom będzie możliwe połączenie wątku biblioteki Paho z główną aktywnością poprzez tzw. zmienną dzieloną

## ■ Interakcja z użytkownikiem – interakcja z MQTT (APP005), cd.

- Klasa `MainViewModel` ma implementację:

```
package com.example.app005;

import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.ViewModel;

public class MainViewModel extends ViewModel {
    private MutableLiveData<String> mText;           //"zmienna dzielona"

    public MainViewModel() {
        mText = new MutableLiveData<>();
        mText.setValue("");
    }

    public MutableLiveData<String> getmText() { //pozwala na modyfikację pola mText
        return mText;
    }

    public LiveData<String> getText() {           //pozwala na tworzenie obserwatora
        return mText;
    }
}
```

- Interakcja z użytkownikiem – interakcja z MQTT (APP005), cd.
  - Zmiana konstruktora implementacja klasy `MqttCallbackImpl` sprowadza się do zapamiętania w konstruktorze dwóch dodatkowych argumentów:

```
public MqttCallbackImpl(MainActivity mainActivity,  
                           MutableLiveData<String> mutableLiveData) {  
    this.handler=mainActivity;           //zapamiętanie kontekstu głównej aktywności  
    this.mutableLiveData=mutableLiveData; //zapamiętanie „zmiennej dzielonej”  
}
```

- Oraz dodania (na jej początku) do klasy `MqttCallbackImpl` pól:

```
public MainActivity handler;  
  
MutableLiveData<String> mutableLiveData; //”zmienna dzielona” to ciąg tekstowy
```



### ■ Interakcja z użytkownikiem – interakcja z MQTT (APP005), cd.

- Teraz wewnątrz metody `connect()` klasy `MqttCallbackImpl` można zmienić implementację na:

```
public boolean connect(String domain, int port,
                      MqttConnectOptions mqttConnectOptions) {
    try {
        client = new MqttClient( . . . );
        client.setCallback(this);
        client.connect(mqttConnectOptions);
        mutableLiveData.setValue("Has been connected");
        return true;
    } catch (MqttException e) {
        mutableLiveData.setValue("Connection failure");
        return false;
    }
}
```

- W powyższej implementacji wywołania metody `mutableLiveData.setValue( ... )` umożliwią właściwe przekazywanie danych do głównej aktywności

## ■ Interakcja z użytkownikiem – interakcja z MQTT (APP005), cd.

- Następny krok to sprawienie aby główna aktywność mogła zareagować w określonym momencie – zatem w ramach metody onCreate() musimy zarejestrować kod który ma być wywoływany gdy zmienna dzielona ulegnie modyfikacji:

```
mainViewModel.getText().observe(MainActivity.this, new Observer<String>() {  
    @Override public void onChanged(@Nullable String s) {  
        Toast.makeText(getApplicationContext(), s, Toast.LENGTH_LONG).show();  
    }  
});
```

- Metoda getApplicationContext() jest częścią klasy Activity po której dziedziczy obiekt MainActivity, rozszerzający AppCompatActivity – stąd jest on dostępny bez konieczności podawania klasy macierzystej
- Toast.LENGTH\_LONG to czas trwania przez jaki komunikat jest widoczny na ekranie (można użyć także Toast.LENGTH\_SHORT)
- Efektem działania powyższych zmian będzie obserwowanie komunikatów na ekranie urządzenia: „Has been connected” lub „Connection failure”
  - Za tą wizualizację odpowiedzialne są tzw. elementy Toast
    - wyglądające analogicznie jak komiksowe dymki

### ■ Interakcja z użytkownikiem – interakcja z MQTT – dodatek o subskrybowaniu

- W bibliotece Paho, subskrypcje można zrealizować dopiero gdy zestawione zostanie połączenie – kontrolować stan połączenia i subskrypcji można za pomocą flag zdefiniowanych w głównej aktywności:

```
boolean isConnected = false;
```

```
boolean isSubscribed = false;
```

- Następnie możemy napisać metodę odpowiedzialną za obsługę naciśnięcia określonego klawisza – którego naciśnięcie wykona właściwą subskrypcję i zmianę tych flag

```
public void onClick(View view){  
    if (!isConnected){  
        if (!isSubscribed){  
            if (mqttCallback.subscribe("app/test")){  
                isSubscribed = true;  
            }else{  
                mutableLiveData.setValue("Subscription failure");  
            }  
        }  
    }  
}
```

### **Zadanie:**

Zbuduj prostą aplikację MQTT łączącą się poprzez port 1883 z użyciem uwierzytelnień, ze zbudowanym przez Ciebie brokerem Mosquitto.

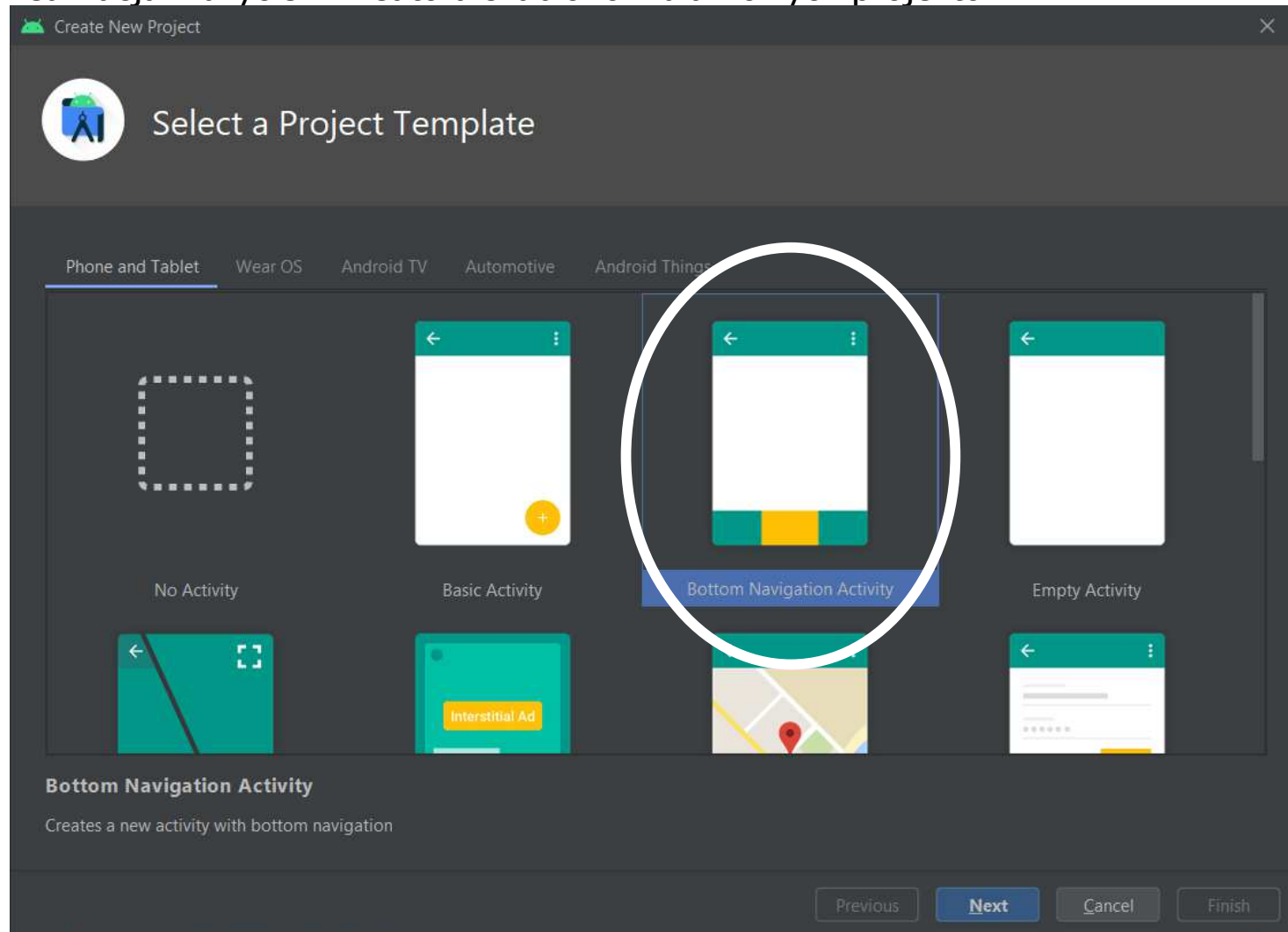
Niech Twoja aplikacja umożliwia subskrypcje danych, które po otrzymaniu w aplikacji systemu Android będą wyświetlane za pomocą elementów typu Toast na ekranie.

Nie zapomnij o „pobudzaniu brokera” wiadomościami testowymi i obserwacji co jest faktycznie publikowane.

- Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006)
  - Oprócz elementów typu Toast w systemie Android dla potrzeb aplikacji przewidziano używanie wielu ciekawych mechanizmów współpracy z interfejsem użytkownika, wśród nich przydatnymi mogą być Klawisze Nawigacyjne i Fragmenty
    - Klawisze nawigacyjne (Button Navigation) – mechanizm wspierający aplikacje składające się z wielu okien (formalnie nazywanych fragmentami)
      - Są one wykreowane przez kreatora aplikacji
      - Domyślnie klawisze nawigacyjne będą umieszczone na dole ekranu
      - Ich dotknięcie przez użytkownika przełączy widok na tzw. fragmenty czyli osobne okna w ramach tej samej aplikacji
      - Generalnie fragmenty to inne części aktywności ze swoim własnym cyklem życia

# Aplikacje dla systemu Android

- Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006), cd.
  - Klawisze nawigacyjne
    - Realizacja z użyciem kreatora szablonów dla nowych projektów



# Aplikacje dla systemu Android

## ■ Interakcja z użytkownikiem – zawansowana interakcja z MQTT (APP006), cd.

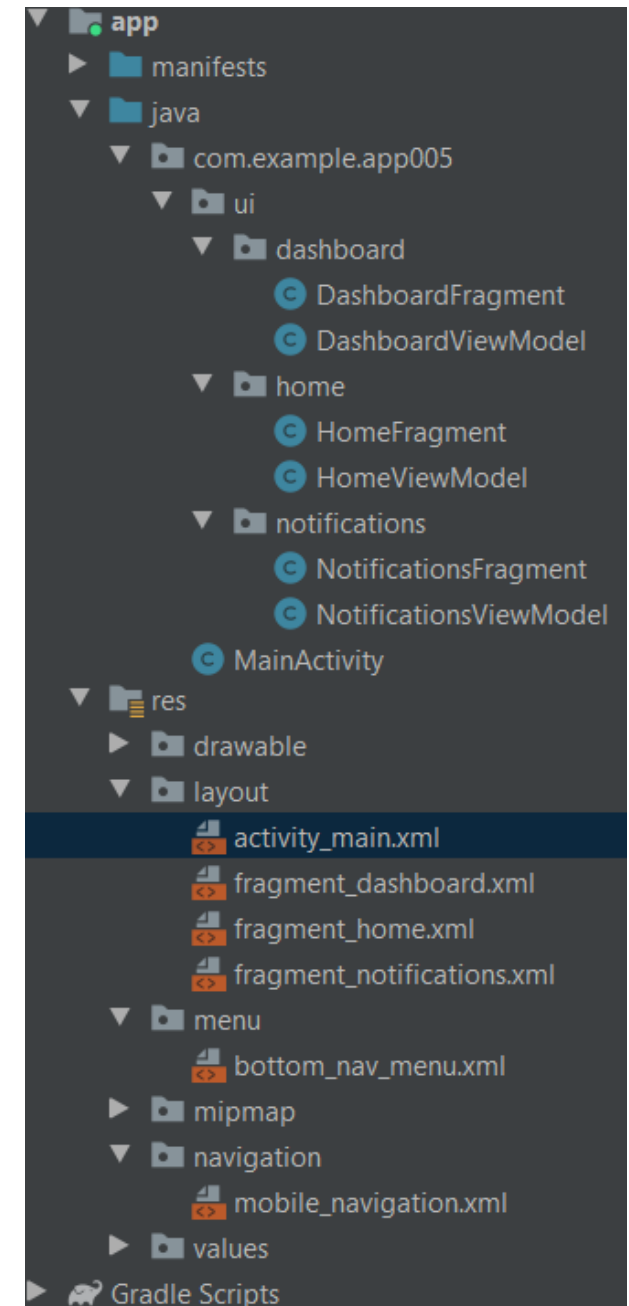
### ■ Klawisze nawigacyjne, cd.

#### ■ Kreator wytworzy projekt z wstępnie ustalonymi implementacjami

- Uwaga! podczas tworzenia warto zaznaczyć opcję „Use legacy android.support libraries” – rozwiązuje problemy z niegodnościami bibliotek

#### ■ Rysunek obok przedstawia widok drzewa wykreowanej aplikacji

- Mamy zatem oprócz implementacji głównej aktywności drzewo UI z obsługą każdego fragmentu
  - Każdy z nich będzie w efekcie definiował nowe okno
  - Podobnie będzie wyglądał opis XML – tu oprócz activity\_main.xml otrzymamy: fragment\_dashboard.xml, fragment\_home.xml, fragment\_notifications.xml
- Dodatkowo kreator utworzył mobile\_navigation.xml który m.in. odpowiada za to które z okien zostanie pokazane jako pierwsze po uruchomieniu aplikacji



## ■ Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006), cd.

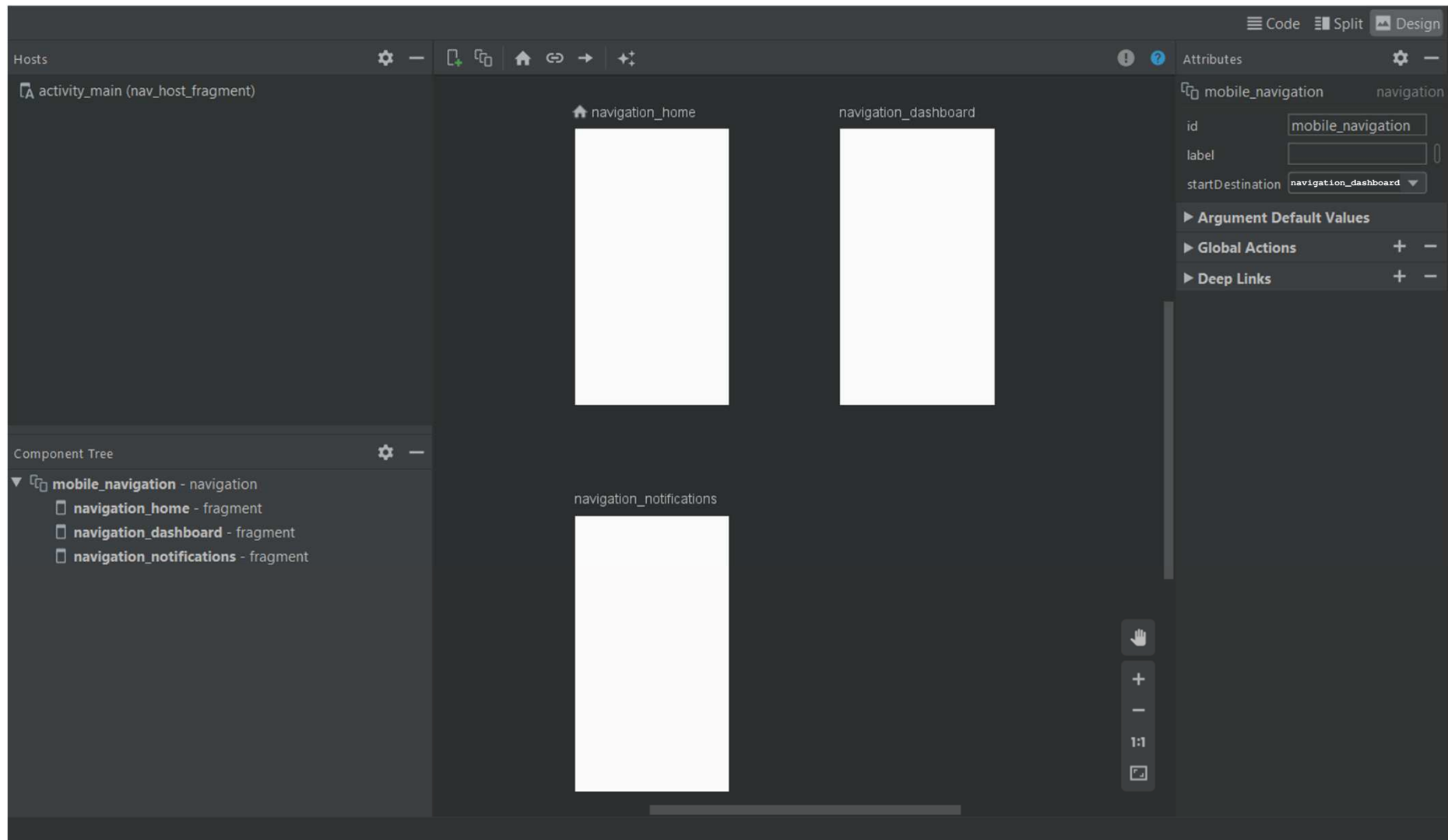
### ■ Klawisze nawigacyjne - opis w mobile\_navigation.xml

```
<?xml version="1.0" encoding="utf-8"?>
  xmlns:app=http://schemas.android.com/apk/res-auto
  xmlns:tools="http://schemas.android.com/tools"
  <navigation xmlns:android=http://schemas.android.com/apk/res/android
    android:id="@+id/mobile_navigation"
    app:startDestination="@+id/navigation_dashboard">
    <fragment
      android:id="@+id/navigation_home"
      android:name="com.example.app006.ui.home.HomeFragment"
      android:label="@string/title_subscribe"
      tools:layout="@layout/fragment_home" />
    <fragment
      android:id="@+id/navigation_dashboard"
      android:name="com.example.app006.ui.dashboard.DashboardFragment"
      android:label="@string/title_connect"
      tools:layout="@layout/fragment_dashboard" />
    <fragment
      android:id="@+id/navigation_notifications"
      android:name="com.example.app006.ui.notifications.NotificationsFragment"
      android:label="@string/title_publish"
      tools:layout="@layout/fragment_notifications" />
    </navigation>
```



# Aplikacje dla systemu Android

- Interakcja z użytkownikiem – zawansowana interakcja z MQTT (APP006), cd.
  - Klawisze nawigacyjne - opis w mobile\_navigation jest także edytowalny poprzez Design Buildera



### ■ Interakcja z użytkownikiem – zawansowana interakcja z MQTT (APP006), cd.

- Klawisze nawigacyjne, cd.
  - Opis nazw każdego z fragmentów (i nazw widniejących w każdym z okien) wpisano do pliku res/values/strings.xml

```
<resources>

    <string name="app_name">APP006</string>

    <string name="title_subscribe">SUBSCRIBE</string>

    <string name="title_connect">CONNECT</string>

    <string name="title_publish">PUBLISH</string>

</resources>
```

- Przydatne gdy chcemy szybko zmieniać język w nowej wersji aplikacji
  - Jedno miejsce definiuje elementy opisowe interfejsu użytkownika a wiele części oprogramowania bazuje na takich definicjach

### ■ Interakcja z użytkownikiem – zawansowana interakcja z MQTT (APP006), cd.

- Klawisze nawigacyjne, cd.
  - Fragmenty mogą w momentach otrzymywania danych być nie dostępne (nie wyświetlane) a powinny w tym czasie przechowywać swój stan – stąd także i one muszą korzystać z techniki LiveData, co wspiera kreator
  - W przypadku „wyklikanej” aplikacji powstał kod z trzema „fragmentami” - dla jednego z nich o nazwie „Dashboard” mamy pliki Java z implementacjami
    - Np.: DashboardViewModel – tworzy obiekt przechowujący stan fragmentu (LiveData), którego klasa powstaje przez rozszerzenie klasy ViewModel:

```
public class DashboardViewModel extends ViewModel {  
    private MutableLiveData<String> mText;           // zmienna dzielona  
    // Dodanie gettera zwracającego obiekt typu MutableLiveData<String>  
    public MutableLiveData<String> getmText() {  
        return mText;} //metoda wspierająca rejestrację obiektu MqttCallbackImpl  
    public HomeViewModel() {  
        mText = new MutableLiveData<>();  
        mText.setValue("This is home fragment");  
    }  
    public LiveData<String> getText() {return mText; }  
}
```

### ■ Interakcja z użytkownikiem – zawansowana interakcja z MQTT (APP006), cd.

- Kod MainActivity musi importować biblioteki (to kreator utworzył)

```
import android.os.Bundle;
import android.support.design.widget.BottomNavigationView;
import android.support.v7.app.AppCompatActivity;
import androidx.navigation.NavController;
import androidx.navigation.Navigation;
import androidx.navigation.ui.AppBarConfiguration;
import androidx.navigation.ui.NavigationUI;
```

## ■ Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006), cd.

- Wygenerowany automatycznie kod MainActivity wygląda tak:

```
public class MainActivity extends AppCompatActivity {  
    @Override protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        BottomNavigationView navView = findViewById(R.id.nav_view);  
        AppBarConfiguration appBarConfiguration = new  
            AppBarConfiguration.Builder(R.id.navigation_home,  
                                       R.id.navigation_dashboard,  
                                       R.id.navigation_notifications).build();  
        NavController navController = Navigation.findNavController(this,  
                                                                R.id.nav_host_fragment);  
        NavigationUI.setupActionBarWithNavController(this, navController,  
                                                    appBarConfiguration);  
        NavigationUI.setupWithNavController(navView, navController);  
    }  
}
```

### ■ Interakcja z użytkownikiem – zawansowana interakcja z MQTT (APP006), cd.

- Jak widać implementacja nie zawiera nic związanego łącznością MQTT – tę funkcjonalność można zaimplementować tam gdzie użytkownik wprowadzi dane związane z połączeniem (adres, port, ...) np.: w DashboardFragment
  - Proszę zauważyć że narzędzie wygeneruje wzorcową implementację

```
...    //importowanie niezbędnych bibliotek

public class DashboardFragment extends Fragment {
    private FragmentDashboardBinding binding;
...    //Tu wstawimy deklaracje pól/obiektów (część A)
    public View onCreateView(@NonNull LayoutInflater inflater,
                               ViewGroup container, Bundle savedInstanceState) {
        dashboardViewModel = new ViewModelProvider(this,
                                                    new ViewModelProvider.NewInstanceFactory())
                                                    .get(DashboardViewModel.class);
        binding = FragmentDashboardBinding.inflate(inflater, container, false);
        View root = binding.getRoot();
        ...//Tu wstawimy własny kod (część B)
        return root;
    }
}
```

### ■ Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006) - implementacja DashboardFragment część A

- Definiujemy (używając Design Builder) elementy: Button, EditText i TextView, a następnie w kodzie definiujemy

```
private Button connectButton;  
private EditText host, port, userName, password;  
private TextView connected;
```

- Dla wsparcia zmiennych dzielonych definiujemy:

```
private DashboardViewModel dashboardViewModel;
```

- Dla połączenia z brokerem definiujemy

```
MqttCallbackImpl mqttCallback;
```

### ■ Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006) - implementacja DashboardFragment część A

- Obsługa interfejsu z użytkownikiem wymaga połączenia artefaktów ekranowych z obiektami

```
connectButton = root.findViewById(R.id.buttonConnect);  
host = root.findViewById(R.id.editTextHost);  
port = root.findViewById(R.id.editTextPort);  
userName = root.findViewById(R.id.editTextUserName);  
password = root.findViewById(R.id.editTextPassword);  
connected = root.findViewById(R.id.isConnected);
```

- Wytworzenia obiektu komunikacji MQTT ze wskazaniem właściwej zmiennej dzielonej tego fragmentu

```
mqttCallback = new MqttCallbackImpl(dashboardViewModel.getMText());
```

- Inicjujemy elementy ekranu

```
host.setText(Settings.host);  
port.setText(Settings.port);  
userName.setText(Settings.userName);  
password.setText(Settings.password);
```



### ■ Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006) - implementacja Dashboard część B

- Obsługa klawisza zestawiającego połączenie (connectButton)

```
connectButton.setOnClickListener(new View.OnClickListener() {  
    @Override public void onClick(View view) {  
        String userNameTxt, passwdTxt, hostTxt; int portNum;  
        hostTxt = host.getText().toString();  
        if(!hostTxt.equals("")){  
            ...//operacje i testy na pozostałych polach (passwdTxt, hostTxt, portNum)  
            if (!MqttCallbackImpl.isConnected) {  
                MqttConnectOptions mqttConnectOptions = new MqttConnectOptions();  
                mqttCallback.connect(hostTxt, portNum, mqttConnectOptions);  
            }else{  
                mqttCallback.disconnect();  
            }  
        }else{  
            dashboardViewModel.getMText().setValue("Please, enter the host name");  
        }  
    }  
});
```

### ■ Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006) - implementacja DashboardFragment część B, cd.

- Obsługa konwersji napisu do liczby – wymagane przy podawaniu numeru portu

```
portNum = Integer.parseInt(port.getText().toString());
```

- Taki kod zadziała poprawnie gdy wprowadzone dane będą „podatne” na konwersję – błąd przy wprowadzaniu liczby przez użytkownika może zakończyć się zakończeniem pracy przez aplikację, zapobiec temu można przez sekcję „try”

```
try {  
    portNum = Integer.parseInt(port.getText().toString());  
} catch (NumberFormatException e) {  
    portNum = 1883;  
}
```

- Nie jest to rozwiązanie idealne ale rozwiązujące problem zakończenia życia aplikacji
  - Lepszym rozwiązaniem byłoby poinformowanie użytkownika o błędzie (Toast) i wyjście z kodu opisywanej tu funkcji

### ■ Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006), - implementacja DashboardFragment część B, cd.

- Implementacja MqttCallbackImpl wymaga także modyfikacji – implementacja mechanizmu obserwowania połączenia:

```
dashboardViewModel.getText().observe(getViewLifecycleOwner(),
                                     new Observer<String>() {
    @Override public void onChanged(@Nullable String s) {
        if (MqttCallbackImpl.isConnected) {
            connectButton.setText("DISCONNECT");
            host.setEnabled(false); port.setEnabled(false);
            userName.setEnabled(false); password.setEnabled(false);
        } else {
            connectButton.setText("CONNECT");
            host.setEnabled(true); port.setEnabled(true);
            userName.setEnabled(true); password.setEnabled(true);
        }
        connected.setText(s);
    }
});
```

### ■ Interakcja z użytkownikiem – zawansowana interakcja z MQTT (APP006), cd.

- Implementacja `MqttCallbackImpl` wymaga także modyfikacji – implementacja pól:

```
public static MqttClient client;  
MutableLiveData<String> mutableLiveData;    //kopia zmiennej dzielonej  
public static boolean isConnected;          //flaga statusu połączenia  
public static ArrayList<String> subscribedTopics = new ArrayList<>();
```

- Implementacja `MqttCallbackImpl` wymaga także modyfikacji – implementacja związana z konstruktorem:

```
public MqttCallbackImpl(MutableLiveData<String> mutableLiveData) {  
    this.mutableLiveData = mutableLiveData;  
    isConnected = false;  
}  
  
public MqttCallbackImpl(){//druga wersja konstruktora (intencjonalnie pusta)  
}
```

- Oraz metody związane z połączeniem – następne slajdy

### ■ Interakcja z użytkownikiem – zawansowana interakcja z MQTT (APP006) - implementacja MqttCallbackImpl, cd.

#### ■ Odbiór wiadomości:

```
public void messageArrived(String topic, MqttMessage message) throws Exception {  
    String payload = new String(message.getPayload()); //przetworzenie wiadomości  
    HomeViewModel.addLog(topic + ": " + payload);  
}
```

- Jak widać tutaj messageArrived() korzysta z metody addLog() udostępnianej przez HomeViewModel

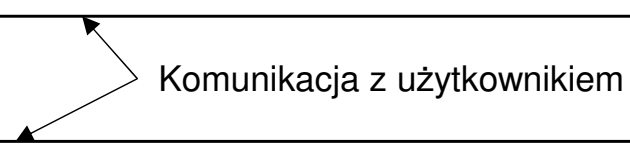
#### ■ Podstawowe metody

```
@Override public void connectionLost(Throwable cause) {  
    HomeViewModel.addLog("Connection was lost");  
}  
  
@Override public void deliveryComplete(IMqttDeliveryToken token) {  
    //...tu implementacja może pozostać pusta  
}
```

### ■ Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006) - implementacja MqttCallbackImpl, cd.

#### ■ Zestawienie połączenia:

```
public boolean connect(String domain, int port, MqttConnectOptions options) {  
    try {  
        client = new MqttClient("tcp://" + domain + ":" + port ,  
                                MqttClient.generateClientId(), new MemoryPersistence());  
        client.setCallback(this);  
        client.connect(options);  
        isConnected = true;  
        HomeViewModel.addLog("Has been connected to "+domain+": "+port);  
        return true;  
    } catch (MqttException e) {  
        HomeViewModel.addLog("Can`t connect to "+domain+": "+port);  
        return false;  
    }  
}
```



Komunikacja z użytkownikiem

- Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006) - implementacja MqttCallbackImpl, cd.

- Rozłączenie połączenia

```
public boolean disconnect() {  
    try {  
        client.disconnect();  
        isConnected = false;  
        HomeViewModel.addLog("Has been disconnected from broker");  
        return true;  
    } catch (MqttException e) {  
        HomeViewModel.addLog("Can't disconnect from broker.");  
        return false;  
    }  
}
```

Wywołanie metody bibliotecznej MQTT

Komunikacja z użytkownikiem

- Oraz nowe metody związane z publikacją i subskrypcją – następne slajdy

### ■ Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006) - implementacja MqttCallbackImpl, cd.

#### ■ Subskrypcja wiadomości

```
public boolean subscribe (String topic){  
    try {  
        client.subscribe(topic);  
        subscribedTopics.add(topic);  
        HomeViewModel.addLog(topic+": Has been subscribed!!!");  
        return true;  
    } catch (MqttException e) {  
        HomeViewModel.addLog(topic+": Can't subscribe.");  
        return false;  
    }  
}
```

//Wywołanie metody bibliotecznej MQTT  
//Zapisanie na przyszłość co było zasubskrybowane

Komunikacja z użytkownikiem

#### ■ Metodę tą używa HomeFragment



- Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006) - implementacja `MqttCallbackImpl`, pozostałe metody

- Subskrypcja wiadomości

```
public boolean unsubscribe (String topic){  
    try {  
        client.unsubscribe (topic);  
        subscribedTopics.remove (topic);  
        HomeViewModel.addLog (topic+": Has been unsubscribed!!!");  
        return true;  
    } catch (MqttException e) {  
        HomeViewModel.addLog (topic+": Can't unsubscribe.");  
        return false;  
    }  
}
```

//Wywołanie metody bibliotecznej MQTT  
//Usunięcie informacji co było zasubskrybowane

Komunikacja z użytkownikiem

- Metodę tą używa `HomeFragment`

## ■ Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006) - HomeFagment

- HomeFagment – tworzy obiekt „rysujący” w nowym oknie, jego klasa powstaje poprzez rozszerzenie klasy Fragments

```
public class HomeFagment extends Fragment {  
    private HomeViewModel homeViewModel;  
    private TextView logs;private Button buttonSubscribe;private EditText topic;  
    MqttCallbackImpl mqttCallback;  
  
    public View onCreateView(@NonNull LayoutInflater inflater,  
                               ViewGroup container, Bundle savedInstanceState){  
        homeViewModel = HomeViewModel new ViewModelProvider(this,  
            new ViewModelProvider.NewInstanceFactory()).get(HomeViewModel.class);  
        logs = root.findViewById(R.id.logs);  
        buttonSubscribe = root.findViewById(R.id.buttonSubscribe);  
        topic = root.findViewById(R.id.topic);  
        mqttCallback = new MqttCallbackImpl();  
        if (checkTopics(topic.getText().toString())){  
            buttonSubscribe.setText("UNSUBSCRIBE");  
        } else { buttonSubscribe.setText("SUBSCRIBE");}  
        ... //Części: H1, H2 i H3 - omówione na następnych slajdach  
        return root;    }    }
```

## ■ Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006) - HomeFagment, cd.

- HomeFagment – część H1: sprawdzenie czy ,topic' jest subskrybowany, po każdej zmianie obiektu interfejsu użytkownika typu EditText (zakłada się że taki element zdefiniowano via Design Builder)

```
topic.addTextChangedListener(new TextWatcher() {  
    @Override public void beforeTextChanged(CharSequence charSequence, int i, int i1, int i2){  
    }  
    @Override public void onTextChanged(CharSequence charSequence, int i, int i1, int i2){  
    }  
    @Override public void afterTextChanged(Editable editable) {  
        String t;  
        t=editable.toString();  
        if (checkTopics(t)){ buttonSubscribe.setText("UNSUBSCRIBE");  
        } else { buttonSubscribe.setText("SUBSCRIBE"); }  
    } });  
private boolean checkTopics (String text){  
    for (String s: MqttCallbackImpl.subscribedTopics)  
        if (text.equals(s)) return true;  
    return false;  
}
```

### ■ Interakcja z użytkownikiem – zawansowana interakcja z MQTT (APP006) - HomeFagment, cd.

- HomeFagment – część H2: obsługa naciśnięcia klawisza subskrypcji

```
buttonSubscribe.setOnClickListener(new View.OnClickListener() {  
    @Override public void onClick(View view) {  
        if (MqttCallbackImpl.isConnected){  
            if (buttonSubscribe.getText().toString().equals("SUBSCRIBE")){  
                mqttCallback.subscribe(topic.getText().toString());  
                buttonSubscribe.setText("UNSUBSCRIBE");  
            } else {  
                mqttCallback.unsubscribe(topic.getText().toString());  
                buttonSubscribe.setText("SUBSCRIBE");  
            }  
        } else {  
            Toast.makeText(HomeFagment.this.getContext(),  
                "Please conect the broker first!", Toast.LENGTH_LONG).show();  
        }  
    }  
});
```

### ■ Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006) - HomeFagment, cd.

- HomeFagment – część H3: odświeżanie ekranu

```
homeViewModel.getText().observe(getViewLifecycleOwner(),
                                new Observer<ArrayList<String>>()
{
    @Override public void onChanged(@Nullable ArrayList<String> s) {
        logs.setText(""); //kasowanie poprzedniej zawartości
        for (String ss: s){
            logs.setText(ss + "\n" + logs.getText());
        }
    }
});
```

- Za odpowiednie przedstawianie treści odpowiada HomeViewModel – następny slajd

### ■ Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006) – HomeViewModel:

```
public class HomeViewModel extends ViewModel {  
    public static MutableLiveData<ArrayList<String>> mText = new MutableLiveData<>();  
    public HomeViewModel() {  
        if (mText.getValue() != null) {           // ustawienia wartości początkowej  
            mText.setValue(new ArrayList<>(mText.getValue()));  
        } else{ mText.setValue(new ArrayList<>()); }  
    }  
    public static void addLog (String logTxt){  
        if (mText.getValue().size() < 10){ //czy mamy więcej niż 10 komunikatów?  
            mText.getValue().add("> " + logTxt);  
            mText.postValue(new ArrayList<>(mText.getValue()));  
        } else {  
            mText.getValue().remove(0);  
            mText.getValue().add("> " + logTxt);  
            mText.postValue(new ArrayList<>(mText.getValue()));  
        }  
    }  
    public LiveData<ArrayList<String>> getText(){return mText;} // tzw. getter  
}
```

- Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006) - implementacja MqttCallbackImpl, pozostałe metody

- Publikacja wiadomości

```
public void clientPublish(String mqttTopic, String message) {  
    try {  
        client.publish(mqttTopic, new MqttMessage(message.getBytes()));  
        HomeViewModel.addLog(mqttTopic+" (o): Message sent OK.");  
    } catch (MqttException e) {  
        HomeViewModel.addLog(mqttTopic+": Send failure.");  
    }  
}
```

Wywołanie metody bibliotecznej MQTT

Komunikacja z użytkownikiem

- Metodę tą używa NotificationsFragment

### ■ Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006) – NotificationsFragment:

```
public class NotificationsFragment extends Fragment {  
    private NotificationsViewModel notificationsViewModel;  
    private Button buttonPublish;  
    private EditText topicPublish, message;  
    MqttCallbackImpl mqttCallback;  
    public View onCreateView(@NonNull LayoutInflater inflater,  
                             ViewGroup container, Bundle savedInstanceState) {  
        notificationsViewModel = new ViewModelProvider(this,  
                                                         new ViewModelProvider.NewInstanceFactory())  
                                                         .get(NotificationsViewModel.class);  
        View root = inflater.inflate(R.layout.fragment_notifications, container, false);  
        buttonPublish = root.findViewById(R.id.buttonPublish);  
        topicPublish = root.findViewById(R.id.publishTopic);  
        message = root.findViewById(R.id.message);  
        mqttCallback = new MqttCallbackImpl(notificationsViewModel.getmText());  
        ... //Części: N1, N2 - omówione na następnych slajdach  
        return root;  
    }  
}
```



### ■ Interakcja z użytkownikiem – zawansowana interakcja z MQTT (APP006) – NotificationsFragment, cd.

#### ■ NotificationsFragment N1:

```
buttonPublish.setOnClickListener(new View.OnClickListener() {  
    @Override public void onClick(View view) {  
        if (MqttCallbackImpl.isConnected){  
            mqttCallback.clientPublish(  
                topicPublish.getText().toString(),message.getText().toString());  
        } else {  
            Toast.makeText(NotificationsFragment.this.getContext(),  
                "Please conect the broker first!", Toast.LENGTH_LONG).show();  
        }  
    }  
});
```

### ■ Interakcja z użytkownikiem – zaawansowana interakcja z MQTT (APP006) – NotificationsFragment, cd.

#### ■ NotificationsFragment N2:

```
notificationsViewModel.getText().observe(getViewLifecycleOwner(),  
                                         new Observer<String>() {  
    @Override public void onChanged(@Nullable String s) {  
        Toast.makeText(NotificationsFragment.this.getContext(), s,  
                        Toast.LENGTH_LONG).show();  
    }  
});
```

#### ■ Implementacja NotificationsViewModel jest dość prosta, podobna do tych z innych fragmentów:

```
public class NotificationsViewModel extends ViewModel {  
    private MutableLiveData<String> mText;  
    public NotificationsViewModel() {  
        mText = new MutableLiveData<>();  
    }  
    public MutableLiveData<String> getmText() { return mText;}  
    public LiveData<String> getText() {  
        return mText; //tzw. getter  
    }  
}
```

### **Zadanie:**

Zbuduj aplikację MQTT z pełną obsługą interfejsu użytkownika, umożliwiającą: podawanie informacji niezbędnych do nawiązania połączenia, publikacji dowolnej tekstowej wiadomości pod wybranym tematem, subskrypcji wiadomości w wybranym temacie. Treści tematów powinny być wprowadzane przez interfejs użytkownika, podobnie jak treść publikowanej wiadomości. Nie zapomnij o „pobudzaniu brokera” wiadomościami testowymi i obserwacji co jest faktycznie publikowane.

**Dziękuję za uwagę**