

#### **Тема 4. Анатомія мобільного додатку для платформи Android**

Загалом підходи до розробки програмного забезпечення можна розділити на три крупні категорії: нативна, кросплатформна та гібридна розробка. Проте існують і суміжні напрямки: кросплатформна нативна розробка, змішана гібридна розробка тощо.

У межах курсу пропонується нативна розробка мобільних додатків для платформи Android. Під **нативними** додатками розуміють програмне забезпечення, яке було створено для конкретної операційної системи. Для цього використовуються специфічні для даної платформи інструменти розробки та API з метою отримання можливості задіяти будь-яку доступну функціональність ОС у додатку. У випадку операційної системи iOS використовується середовище розробки XCode та мови програмування Swift, Objective-C або C++. Основою розробки нативних додатків для платформи Android є середовище розробки Android Studio та мови програмування Java або Kotlin.

На початку слід зазначити, що схема розробки мобільного додатку для ОС Android з точки зору організації коду дещо відрізняється від підходу, який використовувався у попередніх темах для розгляду мови програмування Kotlin. Тоді точкою входу в додаток вважалась функція `main()`, проте Android-додатки будуються шляхом наслідування базових компонентів з бібліотек Android SDK. Крім цього, створюються метадані, що описують відповідні породжені класи. Платформа Android представляє підхід з багатьма точками входу: програма створює групи компонентів, які можна запустити ззовні неї. Наприклад, компонент, що зчитує QR-коди, постачає окрему функцію, яку багато інших додатків можуть інтегрувати у власний інтерфейс користувача. Замість очікування на те, що користувач запустить додаток напряму, компоненти викликають один одного.

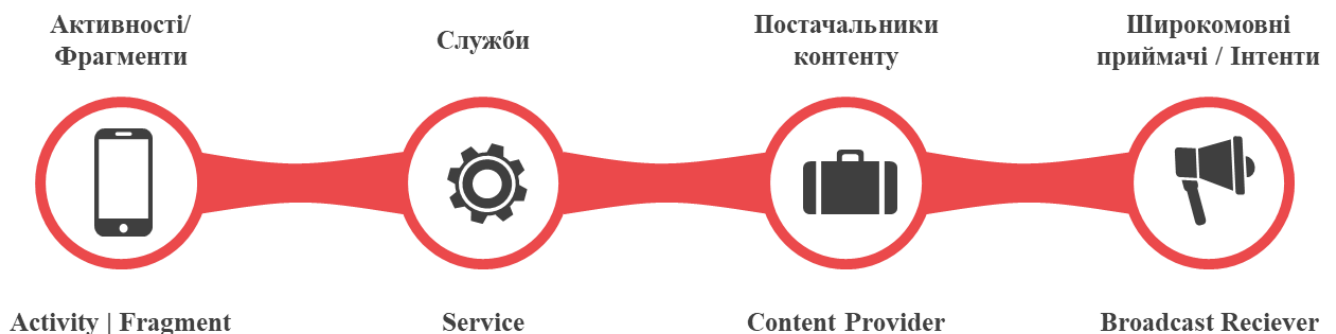
**Доповідь***Мультиплатформа Kotlin.*

#### **Компоненти додатку для платформи Android**

Архітектурною основою стандартного Android-додатку виступають компоненти, зображені на рис. 4.1. **Активність (activity)** є будівельною одиницею інтерфейсу в Android, аналогом класичної веб-сторінки. Частини представленого графічного інтерфейсу зазвичай групують у блоки, які називаються **фрагментами**. Це дозволяє зробити інтерфейс більш модульним, зручніше управляти групами елементів управління (віджетами) та здійснювати адаптацію інтерфейсу до екранів різних розмірів та щільності.

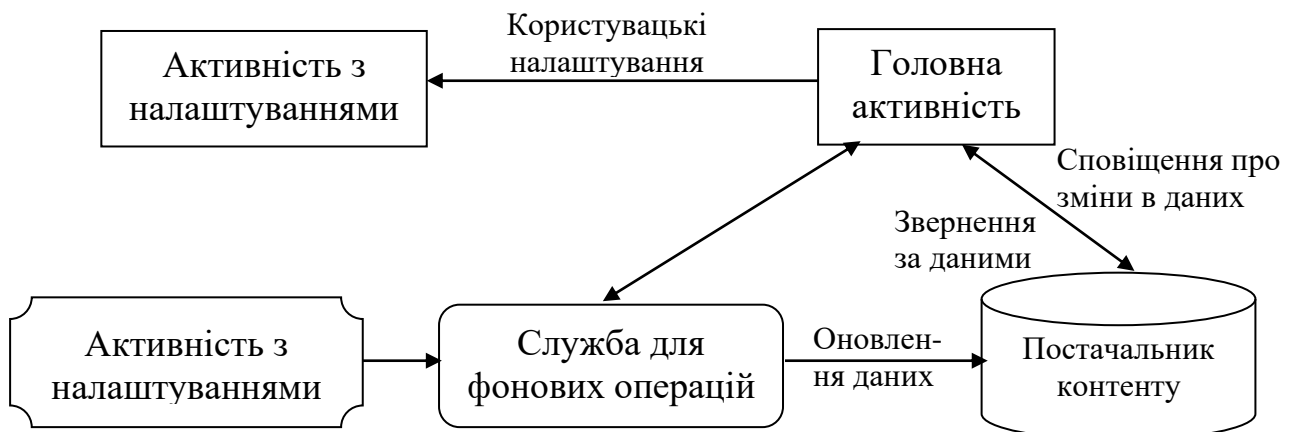
Зазвичай додаток складається з кількох активностей та фрагментів, що передбачає потребу в комунікації між ними. Для цього призначені **інтеннти (intent)**

та **широкомовні приймачі (broadcast receiver)**. За своєю суттю інтенти є зверненнями від одного компонента до іншого всередині програми або назовні. Наприклад, перехід від одного екрану до іншого в межах додатку здійснюється за допомогою **явного інтента**. Аналогічно, при потребі перейти з додатку за посиланням ОС Android викликатиме браузер та передаватиме йому відповідне URL-посилання. Це приклад роботи **неявного інтента**, який буде розглядатись пізніше. На рівні операційної системи пристрою працюють широкомовні приймачі: середовище виконання Android повідомляє про подію, що трапилась, всім зареєстрованим на неї приймачам.



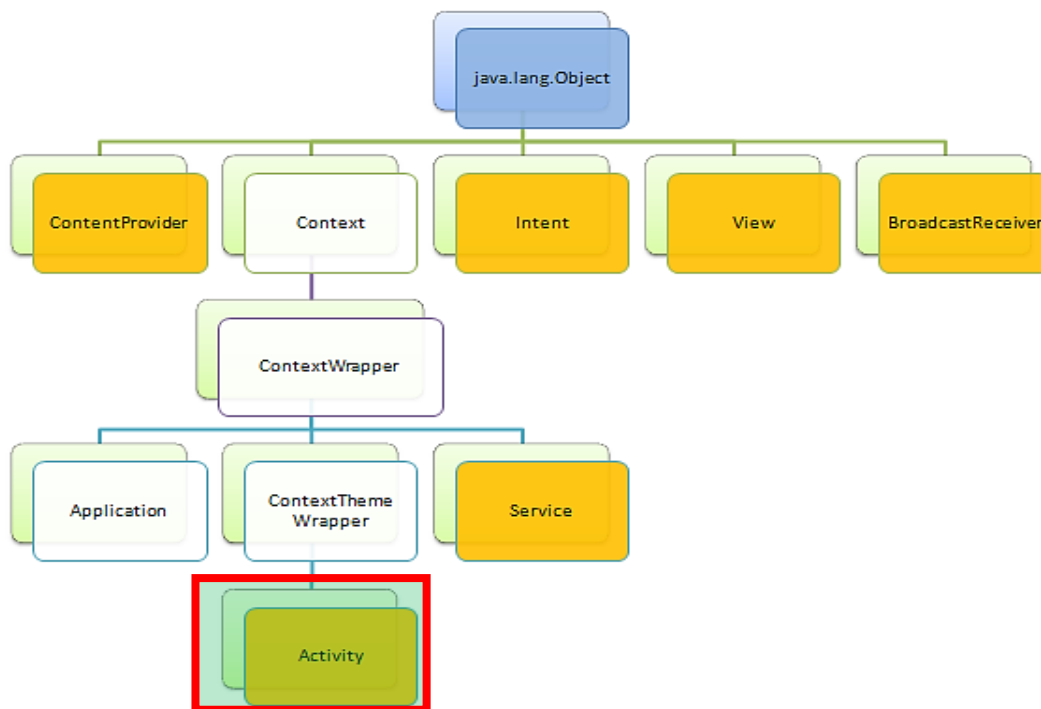
**Рис. 4.1. Компоненти стандартного додатку для ОС Android**

Компонентом для доступу до структурованих наборів даних є **постачальники контенту (content provider)**. Вони допомагають систематизувати роботу з різними джерелами інформації. Крім того, важливим компонентом виступають служби (service), які використовуються для створення фонових задач, що можуть бути активними, але не відображатись. Проста архітектурна комбінація таких компонентів може бути представленою на рис. 4.2. Якщо в планах розробника є реалізація мережових викликів, хорошою ідеєю буде їх виділення в окрему службу. Якщо потрібно повідомляти про різні системні бродкасти, розумно буде задати широкомовний приймач (BroadcastReceiver) та обрати, куди делегувати його події. Якщо додаток потребує зберігати дані не у вигляді стандартних пар «ключ-значення», краще створити постачальник контенту, що відповідатиме за цей сценарій.



**Рис. 4.2. Проста компонентна архітектура Android-додатку**

Представлені компоненти мають чітку відповідність всередині Android SDK (рис. 4.3). Більшість класів, які описують компоненти, знаходяться на одному рівні ієрархії. Виключення становлять служби та активності, які породжені від класу Context (контекст додатку, який можна перемикає). Саме ці класи або їх підкласи стають базовими при описі компонентів додатку розробником.



**Рис. 4.2. Компонентна архітектура класів Android SDK**

### **Налаштування середовища розробки**

Для ОС Android існує багато інструментів розробки для різних операційних систем, основна увага щодо процесу створення програмних продуктів буде сконцентрована на ОС Windows. Розглянемо основні кроки з налаштування програмних інструментів для розробки Android-додатків.

**Крок 1.** Перевірити апаратне забезпечення на сумісність. Для комфортної розробки програмного забезпечення для ОС Android рекомендується завантажити та інсталиувати середовище розробки [Android Studio](#). Рекомендовані вимоги до апаратного забезпечення:

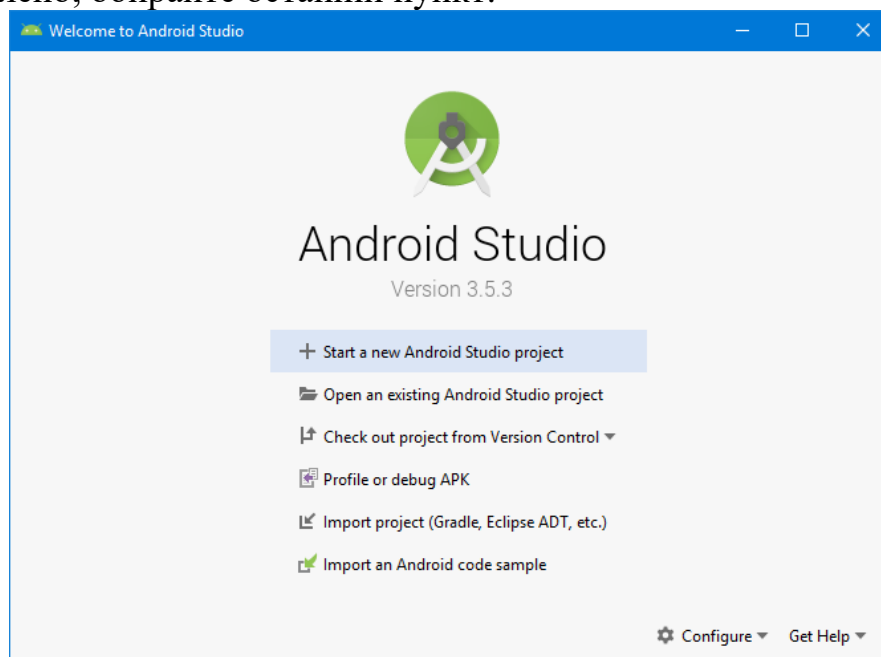
- оперативна пам'ять: 8Гб або більше;
- процесор повинен підтримувати розширення [SSSE3](#) та мати для:
  - Windows: 64-розрядний процесор Intel з підтримкою віртуалізації (VT-x) або AMD (використовуватиметься гіпервізор Windows);
  - Mac: будь-який;
  - Linux: 64-розрядний процесор Intel з підтримкою віртуалізації (VT-x) або AMD з підтримкою AMD-V;
- жорсткий диск, краще SSD, з принаймні 20Гб вільного місця.

Вимоги до процесора зумовлені наявністю емулятора в середовищі розробки, проте налагоджувати та тестувати свій додаток можна і на фізичних Android-пристроях. При роботі під управлінням Windows або Linux необхідно переконатись, що в BIOS комп'ютера ввімкнено підтримку віртуалізації (пункту Virtualization Technology, VT-x тощо встановити значення Enabled).

**Крок 2.** Java Development Kit (JDK). Починаючи з версії Android Studio 2.2, середовище розробки постачається з власною копією OpenJDK, яка використовується за умовчанням. За бажанням можна інсталиувати більш нову версію [JDK](#). При розробці під управлінням ОС Linux перевірте можливість запуску 32-розрядних виконуваних файлів. Наприклад, для Ubuntu 14.10 потрібно запустити команду

```
sudo apt-get install libncurses5:i386 libstdc++6:i386 zlib1g:i386
```

**Крок 3.** Встановлення середовища розробки Android Studio, бібліотек та доповнень. Завантажте інстальатор [Android Studio](#) для відповідної операційної системи та виконайте встановлення середовища розробки. При першому запуску виникає вікно управління імпортом налаштувань. Якщо раніше Android Studio не було встановлено, обирайте останній пункт.

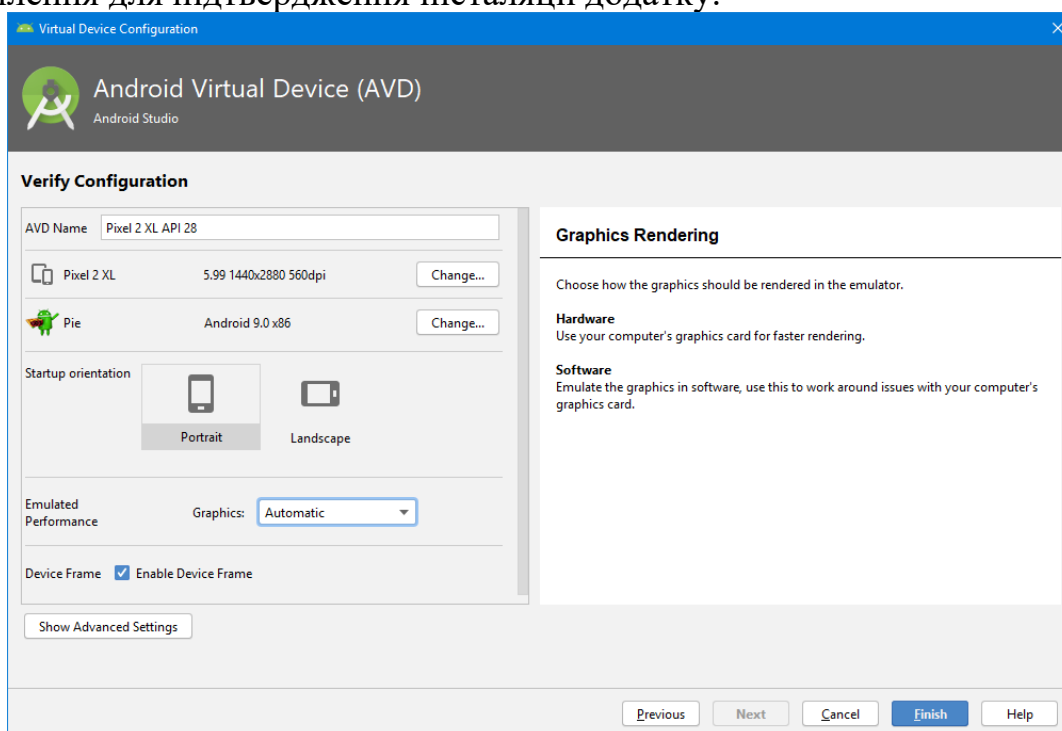


**Рис. 4.3.** Стартове вікно встановленого середовища розробки

**Крок 4.** Налаштування емулятора. На даному кроці можливі наступні проблеми: відсутність потрібного API, вимкнена апаратна підтримки віртуалізації тощо. Для їх вирішення слід завантажити бажану версію платформи та виконати налаштування в BIOS комп'ютера відповідно. Спочатку слід перейти в меню Configure та обрати пункт AVD Manager. При налаштуванні (рис. 4.4) на відносно старих процесорах також може виникнути проблема з підтримкою OpenGL. У

такому випадку в меню Graphics оберіть Software-GLES 2.0. Також можна виконати додаткові налаштування, натиснувши кнопку “Show Advanced Settings”.

**Крок 5.** Налаштування фізичного пристрою. Для можливості інсталяції та запуску додатку, який розробляється, на своєму мобільному пристрої необхідно активувати режим “Для розробників” та включити пункти меню налаштувань “Налагодження по USB”, “Встановлення по USB” або подібні пункти (залежно від графічної оболонки). Також потрібен встановлений режим передачі файлів. Під час компіляції на екрані мобільного пристрою можуть виникати діалогові повідомлення для підтвердження інсталяції додатку.



**Рис. 4.4. Вікно конфігурації емулятора**



### **Завдання**

*Налаштуйте емулятор пристрою для свого персонального комп'ютера та перевірте його працездатність. За наявності мобільного пристрою з операційною системою Android*

### **Створення першого проєкту**

Для створення нового проєкту слід обрати пункт “Start a new Android Studio project” на стартовому екрані середовища розробки Android Studio (рис. 4.3). Наступне вікно дозволяє обрати шаблон додатку – для початку краще обирати шаблон “Empty Activity”. Подальші налаштування стандартні: назва проєкту, пакету, мінімальної версії платформи (рекомендується Android 5.0), вибір мови програмування (Kotlin) та ін.

Вигляд середовища розробки з першим проєктом показано на рис. 4.5. Для відображення структури проєкту краще використовувати режим “Project Files”. На

даному етапі найбільшу цікавість викликають файли MainActivity.kt, activity\_main.xml та AndroidManifest.xml. Перші два файли описують зовнішній вигляд (xml) та логіку роботи (Kotlin) створеної активності. Файл маніфесту конфігурує метадані для подальшого запуску проєкту (лістинг 4.1).

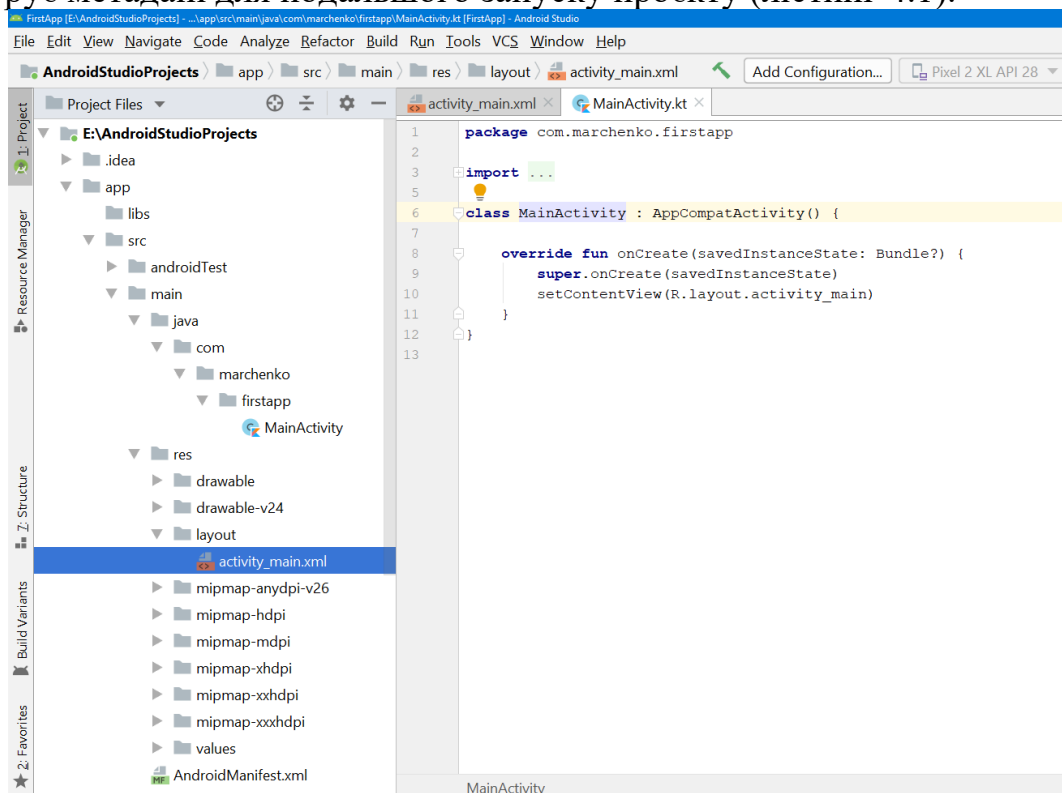


Рис. 4.5. Вигляд середовища розробки Android Studio

Лістинг 4.1. Маніфест найпростішого проєкту

```

1  <?xml version="1.0" encoding="utf-8" ?>
2  <manifest
3  xmlns:android="http://schemas.android.com/apk/res/android"
4    package="com.marchenko.firstapp">
5    <application
6      android:allowBackup="true"
7      android:icon="@mipmap/ic_launcher"
8      android:label="@string/app_name"
9      android:roundIcon="@mipmap/ic_launcher_round"
10     android:supportsRtl="true"
11     android:theme="@style/AppTheme">
12     <activity android:name=".MainActivity">
13       <intent-filter>
14         <action android:name=
15           "android.intent.action.MAIN" />
16         <category android:name=
17           "android.intent.category.LAUNCHER" />
18       </intent-filter>
19     </activity>
20   </application>
21 </manifest>

```

В елементі <application> стандартного файлу маніфесту присутній атрибут android:allowBackup, який за умовчанням набуває значення true. Він вказує, що додаток має працювати в межах системи резервного копіювання платформи Android. Це не завжди вдалий вибір, тому можна замінити значення на false. Маніфест наповнюється шляхом реєстрації компонентів. Наприклад, у лістингу 4.1 зареєстрована активність, побудована на основі класу MainActivity, а елемент <intent-filter> визначає, що активність буде запускаючою (рядки 12-18). Вище записані налаштування візуального оформлення: іконка, назва додатку, тема оформлення та ін. За межами елементу <application> можуть бути присутніми інші налаштування:

- <uses-permission /> .
- <permission />
- <permission-tree />
- <permission-group />
- <instrumentation />
- <uses-sdk />
- <uses-configuration />
- <uses-feature />
- <supports-screens />
- <compatible-screens />
- <supports-gl-texture />

Структурна організація файлів проєкту передбачає наявність окремих директорій для проєкту, який розробляється, та тестового проєкту (папка test), який визначатиме в подальшому можливість автоматизованого тестування. До важливих файлів також слід віднести build.gradle. Це файл конфігурації для збирання проєкту, в якому, зокрема, записані початкові налаштування проєкту. Важливим блоком налаштувань є dependencies, який описує бібліотеки для підключення в проєкті. Також зверніть увагу на пункт buildToolsVersion з блоку android. Це версія збирача, який постійно оновлюється, тому потрібно слідкувати за поточною версією інструментів збірки в Android Studio. У випадку неузгодженості версій в меню File – Project Structure можна звіритись з доступними версіями (рис. 4.6).

**Завдання**

*Запустіть згенерований проєкт на емуляторі або фізичному пристрої. Замініть текст “Hello, World” на “Виконано студентом групи \_\_\_\_ ПІБ” та перезберіть проєкт. Для цього перейдіть у xml-файл з розміткою інтерфейсу та скоригуйте його.*



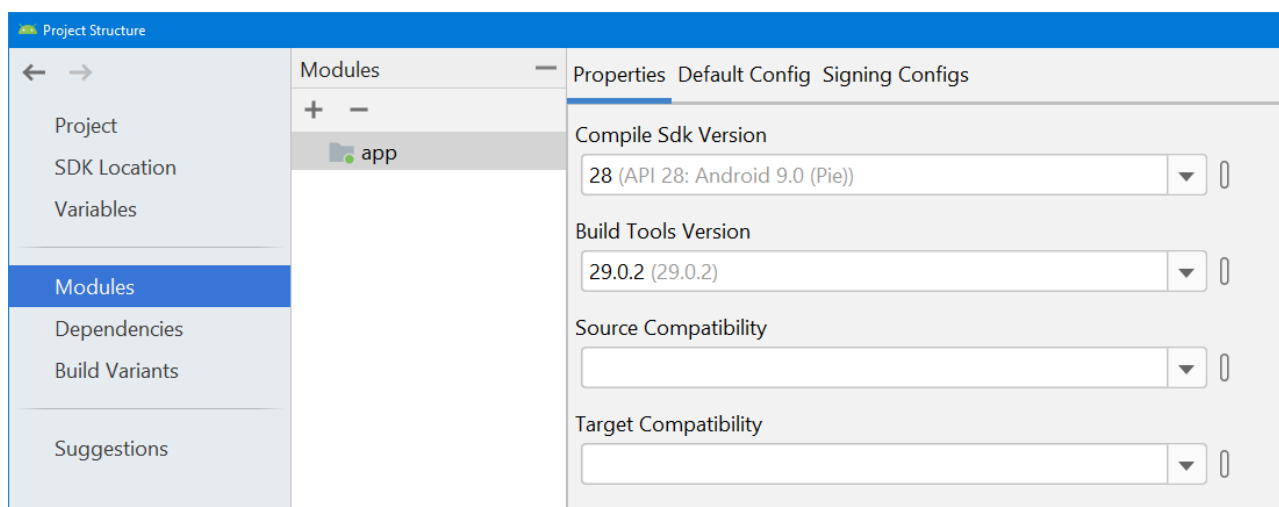


Рис. 4.6. Візуальне налаштування конфігурації збирання проєкту



#### Завдання

Для подальшого управління проєктом створіть відповідний репозиторій на платформах GitHub або BitBucket та виконайте один комміт. Вбудований плагін для роботи з сервісом GitHub вже присутній в Android Studio. Для підтримки сервісу BitBucket доведеться дозавантажити окремий плагін.

### Робота з активностями

Детальніше розглянемо програмний код, який стосується активності. Розмітка з файлу `activity_main.xml` (лістинг 4.2) вказує на деревоподібну структуру інтерфейсу, побудовану на базі кореневого елемента `ConstraintLayout`. Це так званий **диспетчер компоновань**, який визначає принципи макетування інтерфейсу та організацію його елементів всередині себе. У даному випадку, він будується на основі прив'язок (constraints) елементів управління (віджетів) до країв диспетчера. Таким чином, віджет для представлення тексту – `TextView` – центрується відносно усіх чотирьох сторін у рядках 13-16. Диспетчер компоновань має власні налаштування у вигляді атрибутів: `android:layout_width` (ширина, `match_parent` відповідає батьківському елементу управління – ширині екрану), `android:layout_height` (висота), а також `tools:context` – додаткове налаштування для відображення інтерфейсу всередині візуального редактора Android Studio. Ідентифікатори перед двокрапками (`android`, `tools` та інші) є просторами імен, доступними для використання. Вони підключаються за допомогою атрибутів `xmlns` (XML NameSpace), даного набору просторів зазвичай достатньо для розробки графічного інтерфейсу. Атрибути налаштування текстового віджету представлені в рядках 10-12.

#### Лістинг 4.2. XML-розмітка простої активності

```

1  <?xml version="1.0" encoding="utf-8" ?>
2  <androidx.constraintlayout.widget.ConstraintLayout
3  xmlns:android="http://schemas.android.com/apk/res/android"
4      xmlns:app="http://schemas.android.com/apk/res-auto"
5      xmlns:tools="http://schemas.android.com/tools"

```



```
6      android:layout_width="match_parent"
7      android:layout_height="match_parent"
8      tools:context=".MainActivity">
9      <TextView
10         android:layout_width="wrap_content"
11         android:layout_height="wrap_content"
12         android:text="Hello World!"
13         app:layout_constraintBottom_toBottomOf="parent"
14         app:layout_constraintLeft_toLeftOf="parent"
15         app:layout_constraintRight_toRightOf="parent"
16         app:layout_constraintTop_toTopOf="parent" />
17 </androidx.constraintlayout.widget.ConstraintLayout>
```

Розбір записаної розмітки та рендеринг графічного інтерфейсу в подальшому покладається на операційну систему. Лістинг 4.3 показує метод життєвого циклу onCreate(), всередині якого викликається метод setContentView(), якому передається ідентифікатор файлу з розміткою. Такий системний виклик здійснить парсинг XML-розмітки з подальшим рендерингом. Власне метод onCreate() є заміщеною версією відповідного методу з класу AppCompatActivity з Android SDK (рядок 6). Цей клас описує шаблон стандартної порожньої активності, проте з підтримкою зворотної сумісності для застарілих версій операційної системи Android.

#### *Лістинг 4.3. Логіка роботи активності*

```
1 package com.marchenko.firstapp
2
3 import androidx.appcompat.app.AppCompatActivity
4 import android.os.Bundle
5
6 class MainActivity : AppCompatActivity() {
7     override fun onCreate(savedInstanceState: Bundle?) {
8         super.onCreate(savedInstanceState)
9         setContentView(R.layout.activity_main)
10    }
11 }
```

Все, що представлено на екрані емулятора або пристрою між панеллю статусу (status bar) згори та навігаційною панеллю (navigation bar) знизу, є областю активності (рис. 4.7). Для зміни її наповнення передбачені два основні файли, які розглядають активність з точки зору дизайну та логіки.

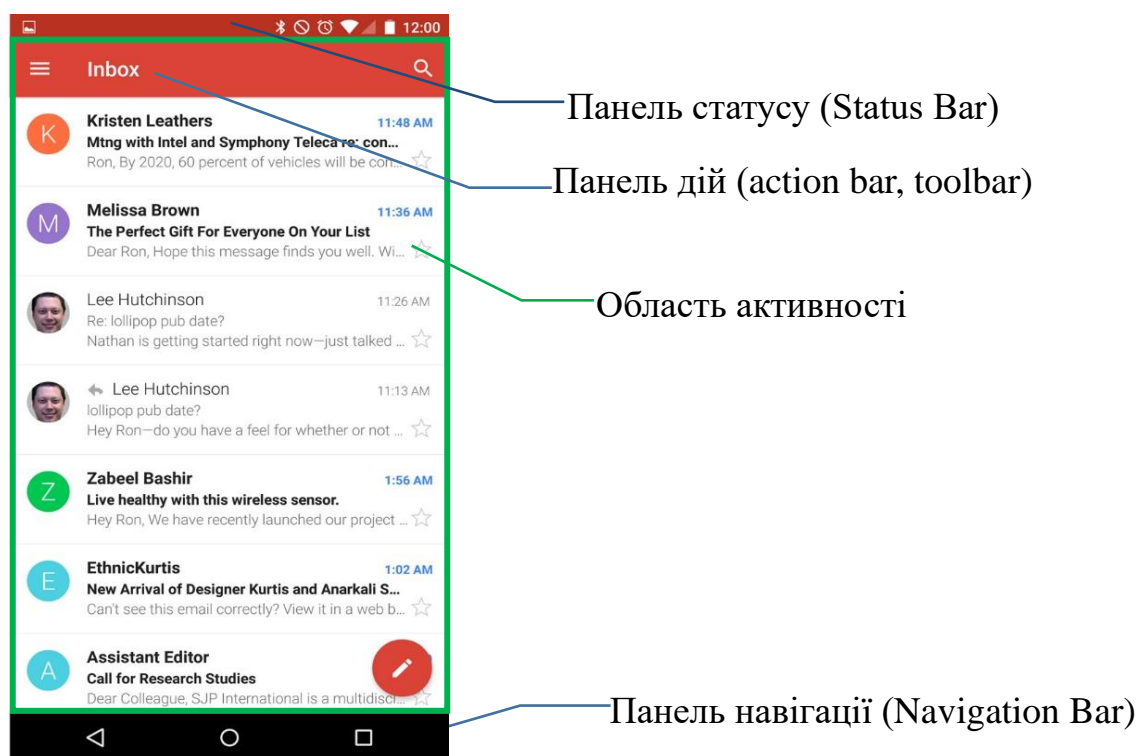


Рис. 4.7. Область активності

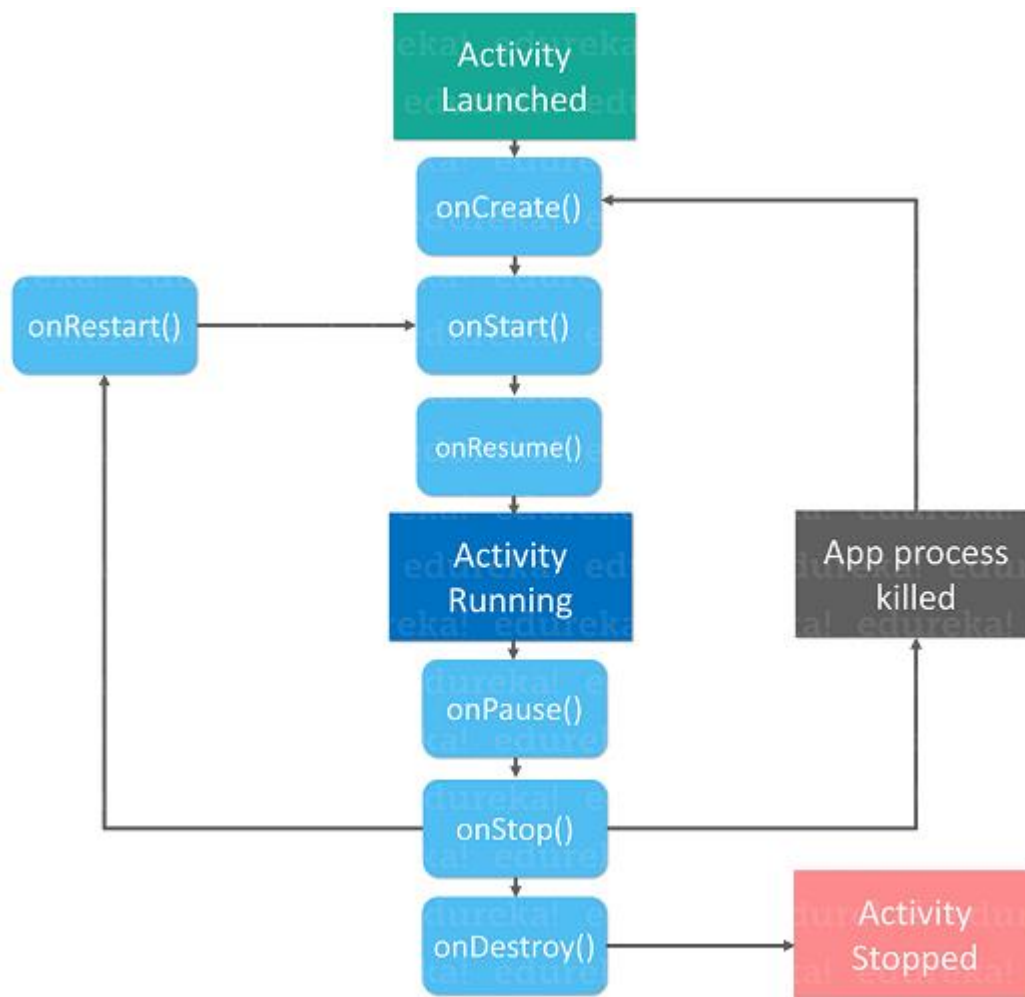
Зазвичай на екрані в кожний момент часу відображається 1-2 активності, що змушує решту запущених додатків змінювати стан своїх активностей:

- **активний стан.** Запущена активність відображається на екрані, в цей момент з нею можна взаємодіяти;
- **призупинений (paused) стан.** Призупинена активність відображається на екрані, проте не має фокусу. Наприклад, коли з'являється діалогове вікно, яке частково перекриває активність та не дає з нею взаємодіяти, поки не зникне;
- **зупинений (stopped) стан.** Зупинена активність не відображається на екрані, а знаходиться у фоні та, ймовірно, буде перервана системою, якщо пам'яті не вистачатиме. Активність зупиняється, якщо інша виходить на передній план (foreground) та стає активною. Наприклад, коли при дзвінку запущений до цього додаток зупиняється;
- **знищений (destroyed) стан.** Активність припиняє існування та вивантажується з пам'яті.

Перехід між цими станами відбувається у відповідь на деякі події на системному рівні. Операційна система Android пропонує для активності 6 методів зворотного виклику (callback methods), які дозволяють запрограмувати реакцію на такі переходи (рис. 4.8):

- **метод onCreate().** Повинен бути реалізованим у будь-якому випадку. Зазвичай всередині нього відбувається виклик методу setContentView(). Тут же можуть ініціалізуватись макет інтерфейсу та різного роду змінні. В рамках даного методу можливий запуск фонового потоку (background thread);

- метод *onStart()*. Викликається, коли активність стає видимою. Зазвичай в ньому відбувається реєстрація широкомовних приймачів, реініціалізація станів;
- метод *onResume()*. Активність видима, і користувач може з нею взаємодіяти. Типові дії: реєстрація широкомовних приймачів, відновлення (restore) змінних, перезапуск анімацій;
- метод *onPause()*. Фіксуються (commit) зміни, зберігаються змінні та налаштування, відбуваються зупинка анімацій, скасування реєстрації (unregistering) широкомовних приймачів;
- метод *onRestart()*. Встановлення з'єднань з сервером, виділення ресурсів;
- метод *onStop()*. Розрив з'єднань з сервером, вивільнення ресурсів, скасування реєстрації широкомовних приймачів;
- метод *onDestroy()*. Активність знищується та вивантажується з пам'яті.

**Рис. 4.8. Методи життєвого циклу активності**

Продемонструємо роботу методів життєвого циклу активності в лістингу 4.4. Доповнимо стандартну реалізацію даних методів діагностичними повідомленнями про викликаний метод.

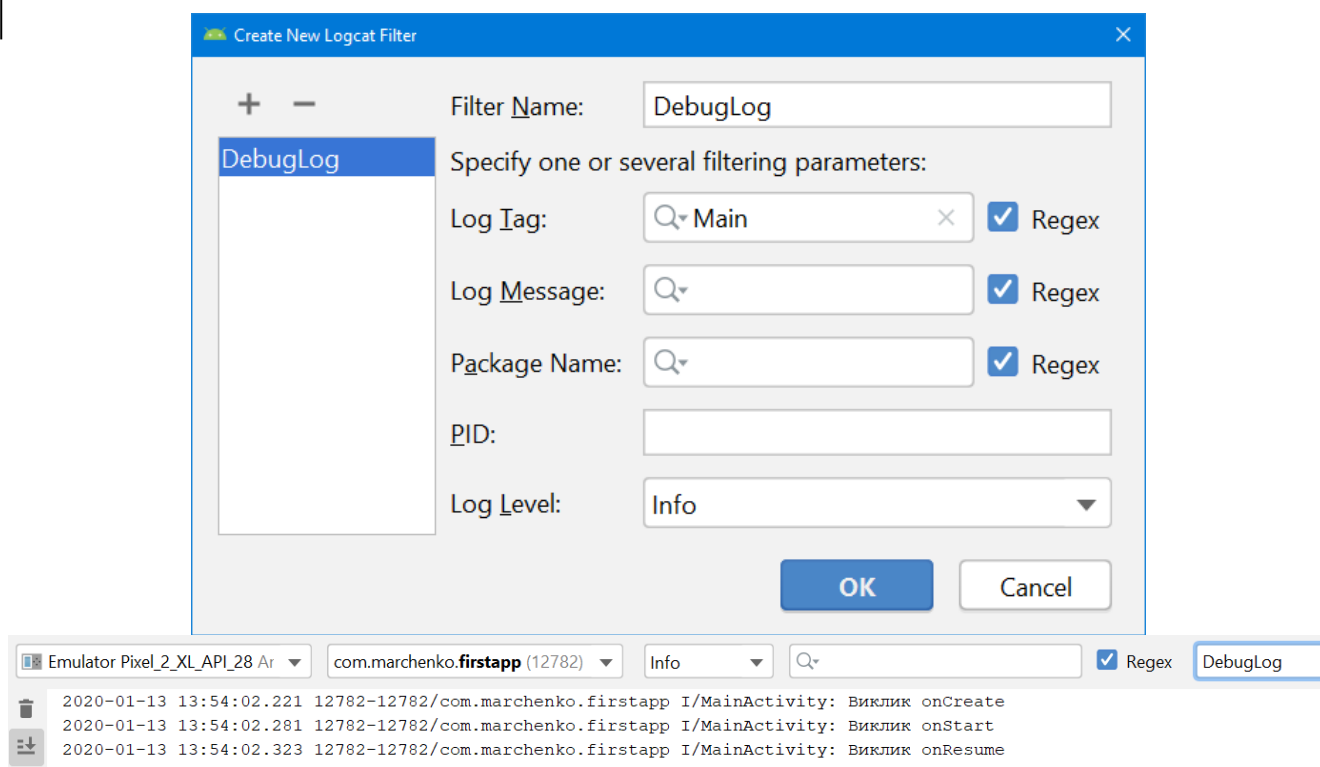
**Лістинг 4.4. Послідовність викликів методів життєвого циклу**


```

1  // імпорти
2  class MainActivity : AppCompatActivity() {
3      val Any.TAG: String
4      get() {
5          val tag = javaClass.simpleName
6          return if (tag.length <= 23) tag
7              else tag.substring(0, 23)
8      }
9      override fun onCreate(savedInstanceState: Bundle?) {
10         super.onCreate(savedInstanceState)
11         setContentView(R.layout.activity_main)
12         Log.i(TAG, "Виклик onCreate")
13     }
14     override fun onStart() {
15         super.onStart()
16         Log.i(TAG, "Виклик onStart")
17     }
18     override fun onResume() {
19         super.onResume()
20         Log.i(TAG, "Виклик onResume")
21     }
22     override fun onStop() {
23         super.onStop()
24         Log.i(TAG, "Виклик onStop")
25     }
26     override fun onPause() {
27         super.onPause()
28         Log.i(TAG, "Виклик onPause")
29     }
30     override fun onRestart() {
31         super.onRestart()
32         Log.i(TAG, "Виклик onRestart")
33     }
34     override fun onDestroy() {
35         super.onDestroy()
36         Log.i(TAG, "Виклик onDestroy")
37     }
38 }

```

Якщо середовище розробки не виконує автоматичне імпортування потрібних модулів, слід налаштувати його: перейти в меню *File – Settings – Editor – General – Auto Import* та обрати All у випадяючому списку “Insert imports on paste”. Відберемо потрібні діагностичні повідомлення шляхом їх фільтрування для консолі LogCat та виведемо виклики методів життєвого циклу після запуску додатку (рис. 4.9). Таким чином, на початку роботи відбувається виклик трьох методів.

**Рис. 4.9. Програмний вивід до лістингу 4.4**

 <b>Завдання</b>	<p><i>Протестуйте роботу додатку при окремих натисненнях на кнопки ◀ (“Back”) та ○ (“Home”). У чому полягають відмінності даних дій? Як відреагує додаток на зміну орієнтації пристрою?</i></p>
--	---

Макетування інтерфейсу передбачає візуальну організацію віджетів в області активності. Вбудовані елементи управління породжені від класу View, який також присутній на рис. 4.2. Розглянемо поширені атрибути для опису окремих віджетів.

Розмір елементів управління може диктуватись розміром батьківського контейнеру (match\_parent), внутрішнім наповненням (wrap\_content) або задаватись у певних одиницях вимірювання: px (пікселі), in (дюйми), mm (міліметри), pt (пункти), dp (density independent pixels), sp (scale independent pixels). Більшість одиниць вимірювання вже не використовуються, оскільки не дозволяють адаптувати розміри для різних екранів. Зазвичай застосовуються dp (незалежні від щільності пікселі) та sp (масштабовані dp, в основному для роботи з текстом). У таблиці 4.1 порівнюється кількість dp для різних екранів відповідно до прийнятої для платформи Android класифікації їх роздільної здатності. У подальшому вона дозволить ретельно розробляти систему ресурсів для додатку.

Таблиця 4.1. Кількість dp на екранах різної щільності

Роздільна здатність екрану	Щільність, dpi	Піксельна пропорційність	Розмір зображень, пікселів
xxxhdpi	640	4.0	400 x 400
xxhdpi	480	3.0	300 x 300
xhdpi	320	2.0	200 x 200
hdpi	240	1.5	150 x 150
mdpi	160	1.0	100 x 100

До поширених атрибутів також відносяться:

- *android:layout\_margin* – відступи між віджетами та іншими суміжними сутностями (іншими віджетами, краями екрану тощо);
  - *android:padding* – відступ від країв віджета до його вмісту (контенту).
- Наприклад, для стандартної кнопки контентом буде напис на ній. Застосування цих атрибутів показано на рис. 4.10.

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="4dp"
    android:padding="8dp"
    android:text="Button 1"/>
```

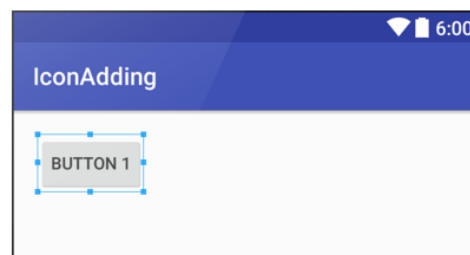


Рис. 4.10. Поширені атрибути для звичайної кнопки

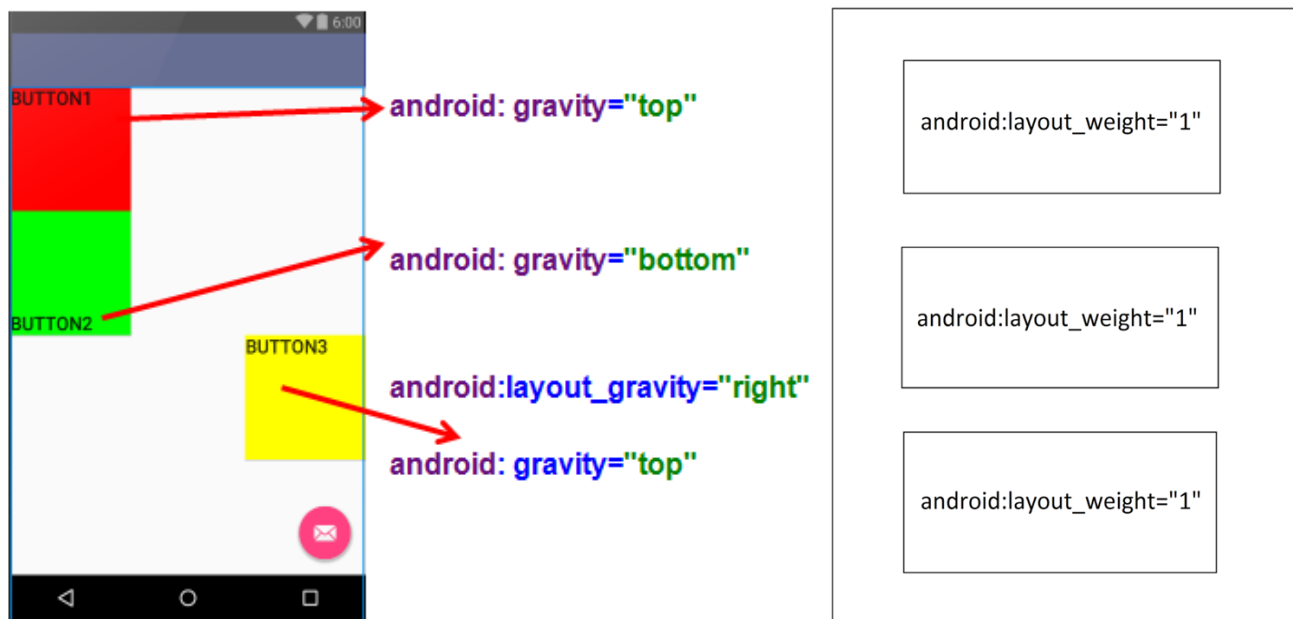
Найпростішими диспетчерами компоновки є *LinearLayout* (елементи вибудовуються в рядок або стовпчик), *RelativeLayout* (відносне позиціонування віджетів) та *FrameLayout* (допускає накладання (overlay) одних віджетів поверх інших). Для *LinearLayout* поширеними атрибутами є:

- *android:orientation* – вирівнювання рядком (значення “horizontal”) або устовпчик (значення “vertical”);
- *android:gravity* – вирівнювання вмісту віджета відносно країв (top, bottom, right та ін.);
- *android:layout\_gravity* – вирівнювання віджета відносно країв батьківського контейнера (top, bottom, right та ін.);
- *android:layout\_weight* – пропорційність розмірів для елементів управління всередині диспетчера компоновки. Візуальна демонстрація атрибутів наведена на рис. 4.11.

При відносному позиціонуванні (за ідентифікатором) для диспетчера компоновки *RelativeLayout* можуть застосовуватись атрибути *android:layout\_above*, *android:layout\_below* (розташовують елемент вище/нижче вказаного елементу); *android:layout\_toLeftOf*, *android:layout\_toRightOf* (встановлює елемент зліва/справа від вказаного елементу); *android:layout\_alignBaseline* (вирівнює базову лінію елементу з базовою лінією вказаного елементу); *android:layout\_alignBottom*, *android:layout\_alignLeft*,



android:layout\_alignRight, android:layout\_alignTop (вирівнює низ / лівий край / правий край / верх елемента з відповідною частиною вказаного елемента).



**Рис. 4.11. Візуалізація поширених атрибутів для LinearLayout**

Диспетчер компоновки FrameLayout має тільки один дочірній елемент, від якого йде решта елементів. Видимість елементів можна управляти за допомогою атрибута android:visibility. У межах матеріального дизайну Android SDK пропонує більш потужні диспетчери компоновки на зразок CoordinatorLayout, ConstraintLayout та ін.

Управляти виглядом віджету можна не лише в XML-розмітці, а й у відповідному Kotlin-файлі. Проте звернутись до віджету з коду можна через його ідентифікатор або тег, що змушує додати до його XML-опису атрибут:

**android:id="@+id/tv"**

Значення атрибута виступає статичним ресурсом (R.id.tv), символ "@" означає звернення до цього ресурсу, а символ "+" – автоматичне додавання ресурсу до загального переліку ресурсів додатку. Детальніше обговоримо систему ресурсів у наступній темі. Відповідно до введеного ідентифікатора можна створити Kotlin-об'єкт, який дозволить маніпулювати віджетом. Для цього всередині onCreate() після виклику setContentView() потрібен метод findViewById<тип\_віджета>():

```
val textView = findViewById<TextView>(R.id.tv)
textView.setText("Демонстрація TextView!")
```

Надалі працюємо з текстовим полем через об'єкт textView. Тут змінюємо текст напису за допомогою методу setText(), в який передаємо новий текст або ресурс, у якому цей текст зберігається.



### Завдання

Ознайомтесь з каталогом стандартних віджетів Android Studio. Детально розгляньте та продемонструйте в xml-розмітці основні атрибути для віджета TextView: `android:textSize`, `android:textStyle`, `android:textColor` та ін.

Використання методу `findViewById()` досить просте, проте має декілька недоліків:

- низька продуктивність результуючого коду, пов'язана з потребою застосовувати цей метод багато разів для роботи з кожним окремим елементом інтерфейсу. Це ж веде і до формування великої кількості шаблонного (boilerplate) коду;
- відсутність null-безпеки, оскільки пошук в ієрархії представлень не представленого в макеті елементу згенерує `NullPointerException`;
- до Android API 26 метод повертав екземпляр батьківського класу – `View`, тому також було необхідно вручну зводити результат до типу елементу інтерфейса. Помилка вибору цього типу призводила до викидання `ClassCastException` під час виконання програми.

Проблему шаблонного коду намагалась вирішити бібліотека Butter Knife, проте нині вона вважається застарілою. У 2017 році було представлено плагін Android Kotlin Extensions для Gradle, який містив *синтетичні властивості* (Kotlin Synthetics). Вони дозволяли для кожного layout-файлу автоматично генерувати клас з представленнями та кешувати результати виклику `findViewById()`. Даний підхід не підтвердив свою універсальність:

- була наявна лише часткова null-безпека. Зокрема, при існуванні багатьох layout-файлів для різних конфігурацій пристрою деякі з елементів інтерфейсу могли не бути присутніми в усіх них. За таких умов доводилось вручну перевіряти результати на рівність null;
- забруднювався простір імен. Маючи елемент інтерфейсу з однаковим ідентифікатором у різних layout-файлах, можна було випадково імпортувати цей елемент не з того файлу, що в подальшому призводило до викидання `NullPointerException`;
- синтетичні властивості використовуються лише в Kotlin, тому для Java вони не підтримувались.

У зв'язку з переліченими недоліками наприкінці 2020 року синтетики були офіційно визнаними застарілими. Тепер рекомендованими для роботи з елементами інтерфейсу в Kotlin-кодi є підходи на основі *прив'язування даних* (data binding). На даному етапі будемо користуватись методикою прив'язки представлень (view binding) – спрощеною формою прив'язки даних. Замість роботи з окремими елементами інтерфейсу цей підхід пропонує формувати єдиний binding-клас



Нині пропонується кілька аналогічних способів

<https://medium.com/better-programming/why-are-kotlin-synthetics-deprecated-and-what-are-the-alternatives-5c2b087dda1c#:~:text=Recently%20Android%20has%20announced%20that,calls%20to%20findViewById%20with%20kotlinx.>

### **Комунікація всередині та між додатками**

Одиницею комунікації між компонентами є інтент (клас Intent). За своєю суттю це абстрактний опис операції, яку одна активність хоче виконати з іншою, наприклад, зняти фото чи подзвонити. Найбільш розповсюджений сценарій – запуск іншої активності в своєму додатку. Також інтенти використовуються для оголошення про запуск активності або служби, націлених на виконання певних дій (як правило, з певною частиною даних); передачі повідомлень про те, що відбулась певна дія або подія. Будь-який додаток може зареєструвати широкомовний приймач та відстежувати інтенти, реагувати на них. Android транслює інтенти для оголошення про системні події, наприклад, про зміни у стані підключення до мережі чи в рівні заряду батареї.

Код, який реалізує одну активність, напрямку не викликає код, що реалізує іншу. Додатки описують інтент, який вони бажають виконати, та просять систему знайти доречного виконавця завдання. Для прикладу створимо та використаємо явний інтент: додамо в додаток другу активність та перейдемо на неї з основної

активності при натисненні на кнопку (лістинги 4.5). Доповнювати проєкт компонентами вручну небажано, середовище розробки дозволяє додавати готові шаблонні компоненти, зокрема й активності (рис. 4.12).

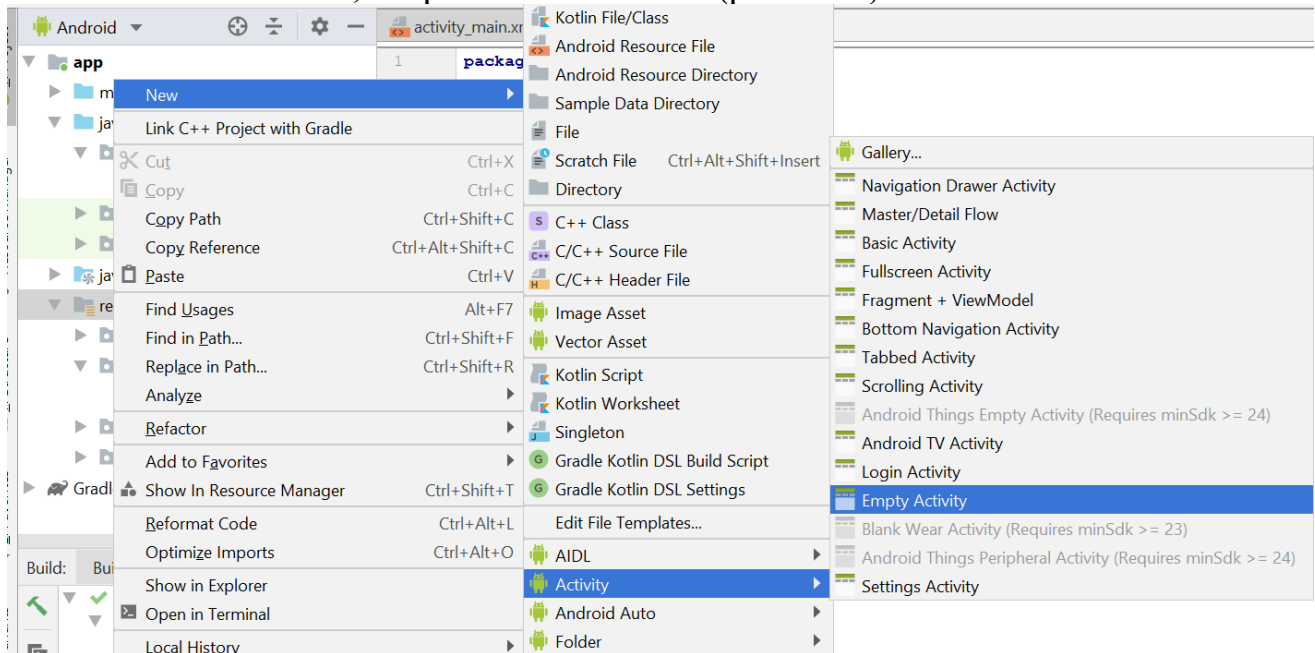


Рис. 4.12. Додавання нової активності (ПКМ на директорії res)



#### Завдання

Перегляньте маніфест проєкту до додавання нової активності та після цього. Які зміни відбулись у ньому та проєкті в цілому?

У лістингу 4.5 показано зміни в головній активності: заміна текстового поля на кнопку в XML-розмітці та доповнення Kotlin-коду обробником події натиснення на кнопку. Інтент називається явним, оскільки існує чітке джерело запиту на виконання операції та чіткий відповідач на цей запит (друга активність). Це підтверджується рядком 15 з лістингу 4.5. Метод `startActivity()` “надсилає” інтент від відправника до отримувача. Додаток у дії зображено на рис. 4.13.

**Лістинг 4.5. Уривки коду з реалізацією переходу до нової активності**  
**activity\_main.xml**

```

1
2 <Button
3     android:layout_width="wrap_content"
4     android:layout_height="wrap_content"
5     android:text="Відправити інтент"
6     android:textSize="24sp"
7     android:onClick="sendIntent"
8     app:layout_constraintBottom_toBottomOf="parent"
9     app:layout_constraintLeft_toLeftOf="parent"
10    app:layout_constraintRight_toRightOf="parent"
11    app:layout_constraintTop_toTopOf="parent" />
12

```

```

13 |                                                                 MainActivity.kt
14 | fun sendIntent(view: View) {
15 |     val intent= Intent(this,SecondActivity::class.java)
16 |     startActivity(intent)
17 | }
    
```

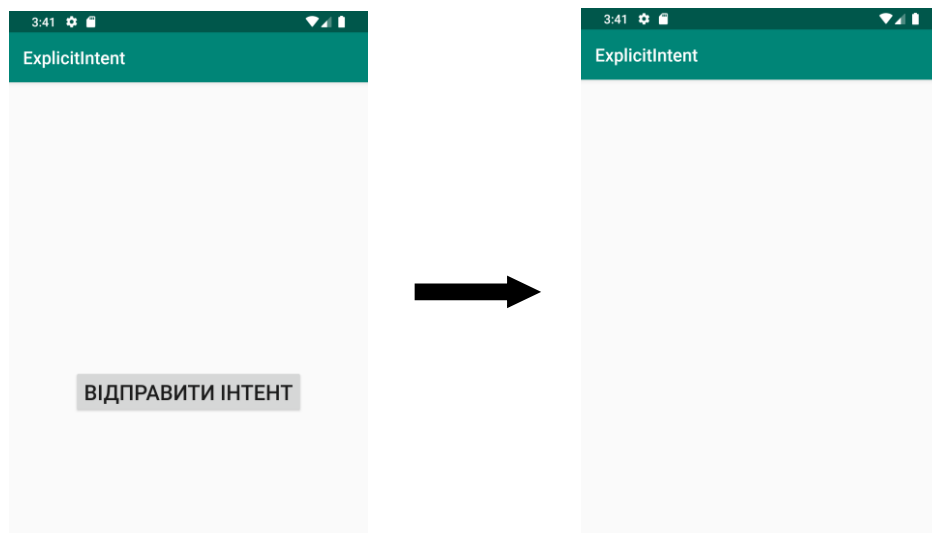


Рис. 4.13. Вигляд додатку в дії

Інтенти можна уявляти як посилки, в які дозволено вкладати додаткову інформацію у вигляді пар “ключ-значення”, причому ключ повинен бути типу String, а значення можуть бути як примітивного (int, double тощо), так і посилкового типу (CharSequence, Bundle, Parcelable, Serializable):

```

intent.putExtra("name", "Станіслав");
intent.putExtra("age", 25);
    
```

«Розпаковку» інтента в другій активності можна виконати так:

```

val extras = intent.extras
if (extras != null) {
    val name = extras.getString("name")
    val age = extras.getInt("age")
}
    
```



Завдання

Розгляньте метод `startActivityForResult()` та поверніть результат успішності переходу на нову активність (подивіться метод `setResult()`). Спробуйте також повернути певні дані з другої активності в головну.

Неявні інтенти мають дещо інший вигляд: вони визначають потрібну дію, для якої операційна система підбере виконавця. Наприклад, існує потреба

переглянути веб-сторінку за деякою адресою. Тоді подібний інтент може виглядати так:

```
val intent = Intent(Intent.ACTION_VIEW,  
                    Uri.parse("http://google.com"))
```

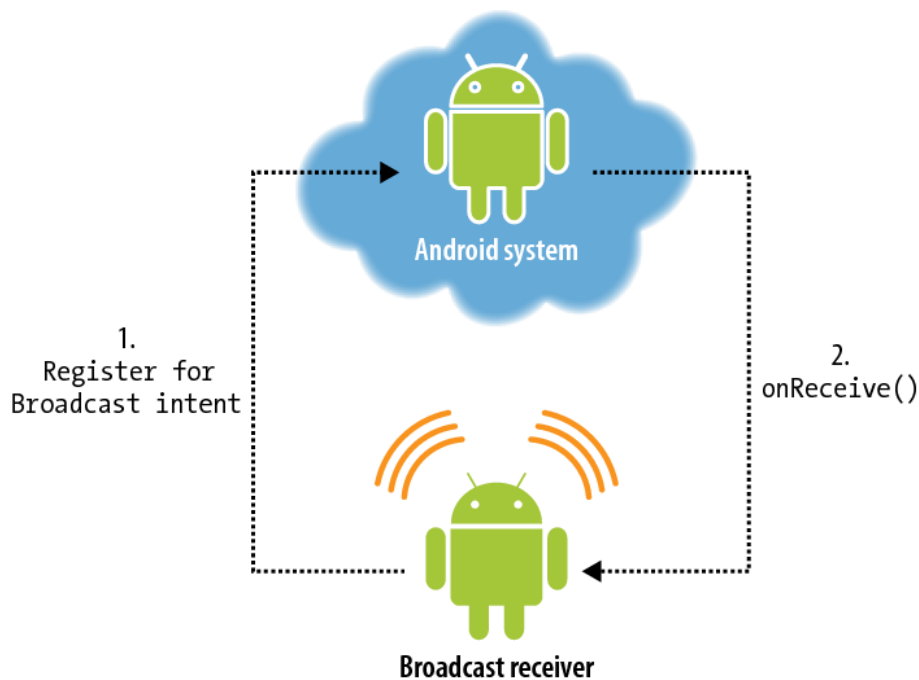
У даному випадку тип Uri зіставляється з доступними сумісними додатками операційної системи, тому може з'явитись перелік браузерів, здатних виконати перехід. Підбір такого переліку система виконує на основі MIME типу, який також можна задати для інтенту за допомогою методу setType().

Система або додатки час від часу розсилають бродкасти: від сповіщення про низький заряд до повідомлення про вимкнення екрану чи зміни стану підключення до Wi-Fi. Клас BroadcastReceiver реалізує варіант високорівневої міжпроцесної взаємодії в Android за допомогою Intent-об'єктів. Широкомовні приймачі мають простий життєвий цикл (рис. 4.14). Середовище виконання Android повідомляє про подію, що трапилась, усім зареєстрованим на цю подію приймачам.

Існує декілька видів бродкастів залежно від синхронності їх обробки та доступності в системі:

- **звичайний бродкаст (Normal Broadcast)** – асинхронний бродкаст, що використовує sendBroadcast(). Будь-який широкомовний приймач отримує бродкаст без визначеного порядку;
- **впорядкований бродкаст (Ordered Broadcast)** – синхронний бродкаст, що використовує sendOrderedBroadcast(). Широкомовний приймач отримує бродкаст за пріоритетом. Такий бродкаст можна скасувати (abort);
- **локальний бродкаст (Local Broadcast)** використовується лише всередині додатку;
- **липкий бродкаст (Sticky Broadcast)** – використовує метод sendStickyBroadcast(Intent). Звичайний бродкаст після відправки та обробки системою стає недоступним. Липкий (sticky) інтент означає, що після виконання бродкасту він залишається в системі. Тому решта компонентів може швидко отримати дані за допомогою registerReceiver(BroadcastReceiver, IntentFilter).





**Рис. 4.14. Життєвий цикл широкомовного приймача**

Основні системні події, на які можна підписати широкомовний приймач, наступні:

- `Intent.ACTION_BOOT_COMPLETED`;
- `Intent.ACTION_POWER_CONNECTED`;
- `Intent.ACTION_POWER_DISCONNECTED`;
- `Intent.ACTION_BATTERY_LOW`;
- `Intent.ACTION_BATTERY_OKAY` та ін.

Дуже часто

### **Процесна модель операційної системи Android**

У більшості випадків кожний Android-додаток працює всередині власного Linux-процесу. Такий процес створюється при запуску коду додатку та залишається в оперативній пам'яті, поки він буде порібний. Як тільки посилань на код додатку не буде, а система вимагатиме пам'яті, процес буде вивантажено.

Особливістю процесної моделі операційної системи Android є те, що життєвий цикл процесу напряду не управляється цим процесом (додатком). Замість цього життєвий цикл визначає операційна система відповідно до комбінації компонентів додатку, про запуск та роботу яких система знає. Крім того, вплив чинитимуть важливість цих компонентів для користувача, кількість доступної пам'яті в системі та ін. Тому некоректне розуміння життєвих циклів компонентів може призвести до переривання роботи додатку при виконанні важливої роботи.

Поширена помилка життєвого циклу процесу стосується широкомовних приймачів, про які йшлося у попередньому питанні. Клас `BroadcastReceiver` передбачає запуск потоку (thread) при реагуванні на інтеніт у методі життєвого циклу `onReceive()`. Після завершення роботи методу система вважатиме широкомовний інтеніт неактивним, тому його процес розглядатиметься як непотрібний (якщо інші компоненти додатку не працюють в ньому в цей час). У такому випадку при потребі в пам'яті система може припинити (kill) процес, а з ним і запущений потік. Типове вирішення цієї проблеми – планування роботи широкомовного приймача за допомогою класу `JobService` (`Job Scheduler API`). Таким чином, система знатиме, що робота широкомовного приймача триває.



Доповідь

*Технології планування задач в операційній системі Android.  
Alarm Manager, Job Scheduler, GCM Network Manager, Firebase Job Dispatcher,  
Sync Adapter*

## Практичні завдання

1. .