

Тема 5. Матеріальний дизайн та стилізація графічного інтерфейсу мобільних додатків

Починаючи з Android 5.0 компанія Google пропонує єдину мову візуального оформлення додатків для платформи – матеріальний дизайн. Уявімо, що інтерфейс побудовано зі шматків цифрового паперу (рис. 5.1), які називають поверхнями (surfaces). Поверхня – це контейнер для контенту. Імітація тривимірності та ієрархічна організація поверхонь відбувається за допомогою тіней. Задавши підйом (атрибут elevation), операційна система Android згенерує тінь.

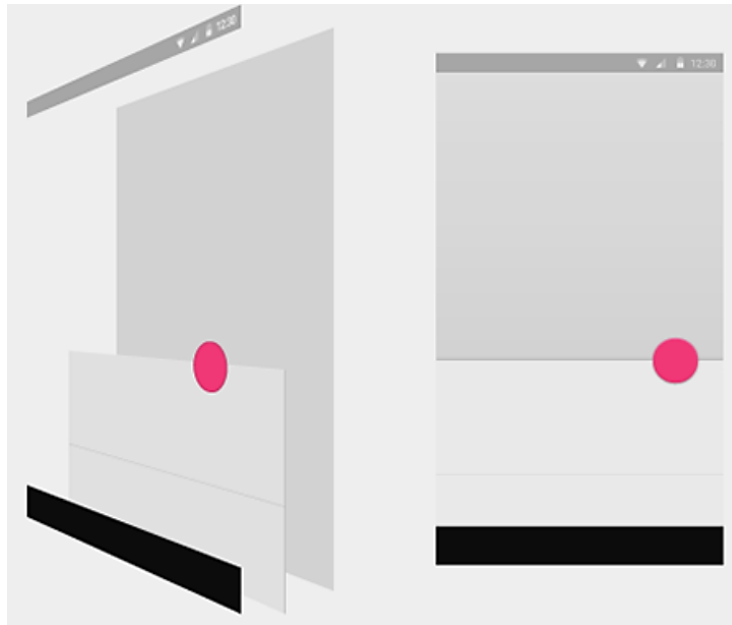


Рис. 5.1. Поверхні в матеріальному дизайні

Матеріальний дизайн включає вимоги до оформлення в контексті шрифтів, кольорів, позиціонування графічних елементів тощо. Також в основу покладено адаптивність дизайну та використання достовірних анімацій. Детальна інформація щодо всіх аспектів матеріального дизайну зібрана на офіційному сайті <https://material.io/>.

Система ресурсів Android-додатків

Ресурси – це статичні біти інформації, що зберігаються окремо від первинного коду (директорія res у проєкті). Ресурси відокремлені від коду, оскільки зберігаються в іншому форматі та використовуються в різних умовах, для яких мають відповідний опис. Наприклад, для інтернаціоналізації матимемо рядки тексту для різних мов. Операційна система Android буде обирати правильний ресурс з усіх кандидатів для конкретного випадку (рядки українською мовою, якщо локалізація – українська).

Директорія res має багато вкладених тек та файлів, які відповідають ресурсам різного виду (рис. 5.2). Посилатись на ресурси можна з різних місць проєкту за допомогою символу “@” з подальшим шляхом до ресурсу. Вигляд такого посилання вже зустрічався, зокрема, у файлі маніфесту:



Рис. 5.2. Класифікація ресурсів

`android:label="@string/app_name"`

Найпростішими ресурсами є рядкові ресурси, які зберігаються в файлах strings.xml у вигляді пар “ключ-значення” всередині елемента <resources>:

`<string name="app_name">ExplicitIntent</string>`

Усі текстові елементи рекомендується зберігати у вигляді відповідних ресурсів. Це буде особливо корисним для інтернаціоналізації додатку, коли будуть створюватись альтернативні файли strings.xml зі спеціальними кваліфікаторами для різних мов.

Також до простих ресурсів відносять масиви рядків, кількісні рядки (plurals), кольори, розміри (dimensions) та стилі. Для масивів рядків передбачений спеціальний елемент <string-array>, всередині якого кожен рядок записується всередині тегів <item> (лістинг 5.1). Для доступу ресурсу спочатку потрібно отримати їх весь доступний набір за допомогою геттера resources або методу getResources(). Далі використовується метод getStringArray() для зчитування масиву відповідно до його ідентифікатора. Виводимо отриманий список у консоль LogCat.



Завдання

Розгляньте роботу з кількісними рядками (plurals) як видом ресурсу, який передбачає адаптацію тексту для слів в однині та множині. Наприклад, маємо 1 копійку, 2 копійки, 5 копійок. Створіть відповідний ресурс, який коректно оброблятиме кількість копійок та виводитиме на екран правильний текст при їх різній кількості.

Лістинг 5.1. Опис та використання масиву рядків

strings.xml

```

1  <string-array name="themes_array">
2      <item>Базовий синтаксис мови програмування
3          Kotlin</item>
4      <item>Вступ до функціонального програмування мовою
5          Kotlin</item>
6      <item>Основи об'єктно-орієнтованого програмування в
7          мові Kotlin</item>
8      <item>Анатомія мобільного додатку для платформи
9          Android</item>
10     <item>Матеріальний дизайн та стилізація графічного
11         інтерфейсу мобільних додатків</item>
12 </string-array>
13
14                                     MainActivity.kt
15 val res = resources
16 val themes = res.getStringArray(R.array.themes_array)
17 for (t in themes) {
18     Log.i("MainActivity", t)
19 }
    
```

При формування ресурсу для кольору відповідне значення записується в XML за допомогою RGB-значення та альфа-каналу. Спеціальний файл colors.xml передбачає доповнення елемента <resources> подібними записами:

```

<color name="opaque_red">#f00</color>
<color name="translucent_red">#80ff0000</color>
    
```

Матеріальний дизайн регламентує використання спеціальних палітр кольорів (palettes, swatches) для оформлення додатку та спеціальних ключів для встановлення кольорів різних частин активності (рис. 5.3):

- основні (primary) кольори – доречні для домінуючих елементів інтерфейсу, наприклад тулбару;
- акцентні кольори (accent, secondary) виділяють додаткову інформацію, вони яскравіші та більш насичені, зазвичай використовуються для виділення кнопок, перемикачів, слайдерів тощо;
- варіанти кольорів (темніші або світліші) для візуального відокремлення сусідніх елементів інтерфейсу.



Завдання

Налаштуйте та підключіть у проєкт власний колірний ресурс, створений за допомогою інструменту для підбору кольорів за адресою <https://material.io/design/color/the-color-system.html#tools-for-picking-colors>. Основний та акцентний кольори обирайте за бажанням або за допомогою сервісу <https://www.materialpalette.com/>

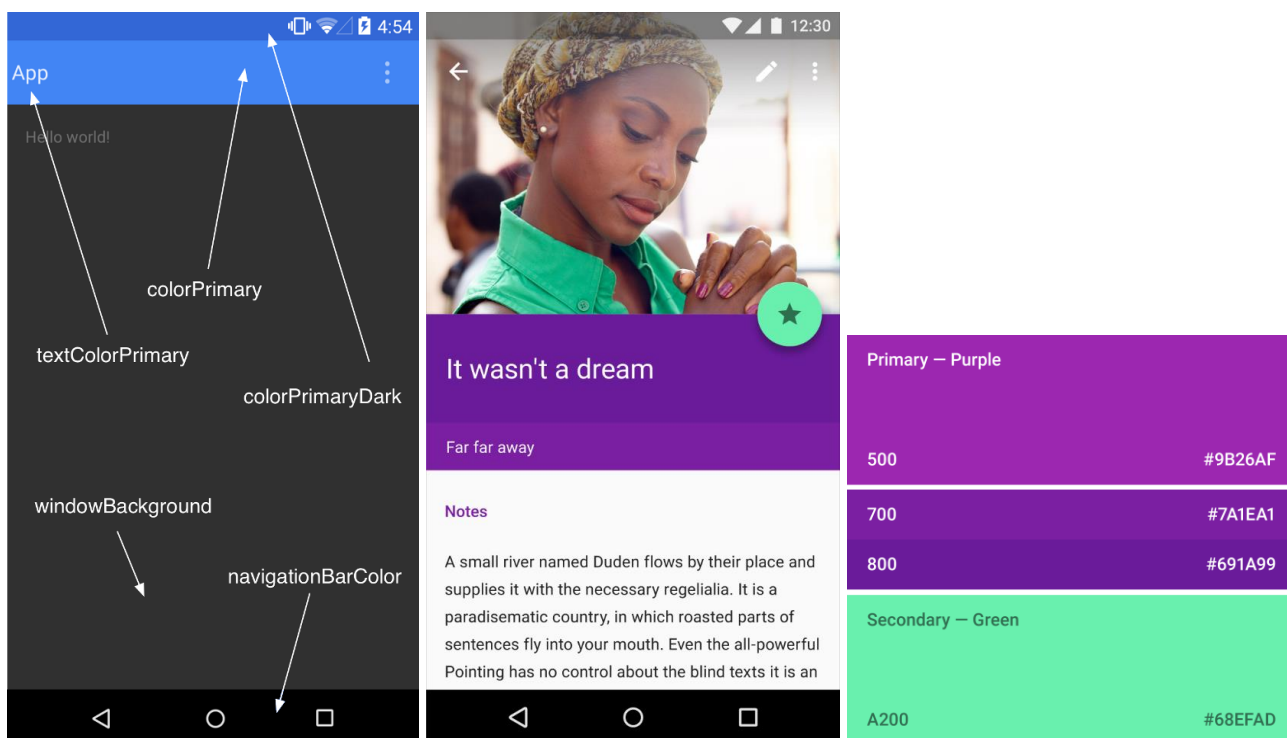


Рис. 5.3. Кольори в матеріальному дизайні

Окремим питанням дизайну завжди є розташування елементів один відносно одного, методи встановлення відступів (spacing methods). Матеріальний дизайн надає чіткі керівництва з приводу рекомендованого розташування елементів інтерфейсу, покриваючи область активності сіткою зі стороною комірки 4dp або кратним значенням. Розміри теж зручно виносити в якості ресурсів. У цілому, натискаючи ПКМ – New – Android Resource File, переходимо в меню створення ресурсів (рис. 5.4). У списку Available qualifiers можна знайти Dimension, а далі ввести роздільну здатність екранів, для яких задаються розміри.

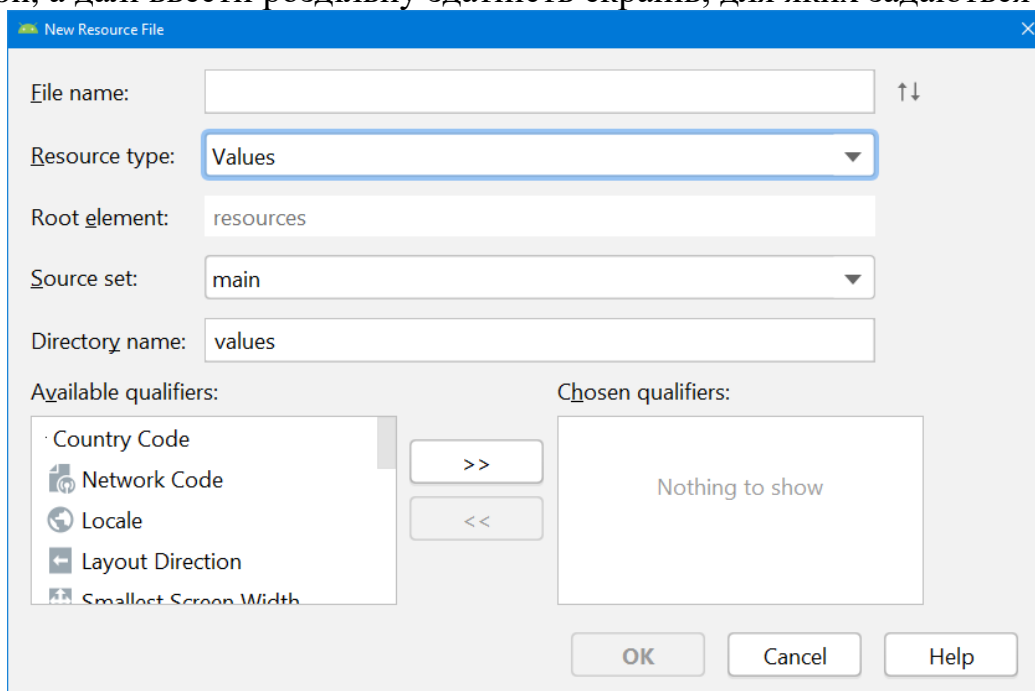




Рис. 5.4. Додавання нового ресурсу

Кваліфікатори дозволяють задавати альтернативні ресурси в одноіменних файлах. Система буде з контексту визначати, який саме файл використовувати. Для розмірностей це будуть різні роздільні здатності екрану, а вибір випадатиме на реальну роздільну здатність для пристрою. Інтернаціоналізація тексту залежатиме від обраної системної мови. Аналогічно можна завантажувати цілі макети залежно від контексту: орієнтації екрану, його розмірів у довжину, ширину тощо.

 <p>Завдання</p>	<p><i>Дослідіть процес додавання нового ресурсу та створіть альтернативні ресурси для написів (strings.xml) українською мовою. Стандартний файл strings.xml працює для англійської мови та решти неспецифікованих мов.</i></p>
--	--

 <p>Завдання</p>	<p><i>Розгляньте альтернативний макет для різних орієнтацій екрану пристрою. Нехай у портретній орієнтації дві кнопки на екрані розташовуються одна під одною, а в ландшафтній – поряд в одному рядку.</i></p>
--	--

Типографіка та іконографіка матеріального дизайну

Уявлення про матеріал як шматок цифрового паперу вимагає розвиненої системи для виводу друкованої інформації, тому вибір та комбінування шрифтів є дуже важливим завданням для цілісного представлення інтерфейсу. Спеціально для матеріального дизайну компанія Google розробила шрифт Roboto, який за умовчанням є основним шрифтом операційної системи Android. Крім того, сервіс Google Fonts пропонує широкий набір шрифтів, які можна завантажити, впровадити в проєкт та мінімально редагувати.

Стандартні шрифти з Android SDK можна стилізувати за допомогою кількох XML-атрибутів:

- ***android:fontFamily*** використовується для заміни шрифту за умовчанням для додатку. У комплекті йде sans-serif шрифт. ***android:fontFamily*** вимагає API 16+.
- ***android:typeface*** використовується замість ***fontFamily*** до API 15. За умовчанням цього атрибуту ***normal*** або ***sans***.
- ***android:textStyle*** може набувати значень ***bold***, ***italic***, ***normal*** та комбінувати їх, наприклад, ***android:textStyle="bold|italic"***.

Компанія Google пропонує для завантаження та застосування шрифти з сервісу [Google Fonts](https://fonts.google.com/). Для того, щоб використати деякі з них у своєму проєкті, можете знайти атрибут ***android:fontFamily*** відповідного віджета в режимі дизайну та обрати пункт More Fonts... У вікні, що відкрилось, натисніть на кнопку «+» та оберіть шрифт з переліку (рис. 5.5).

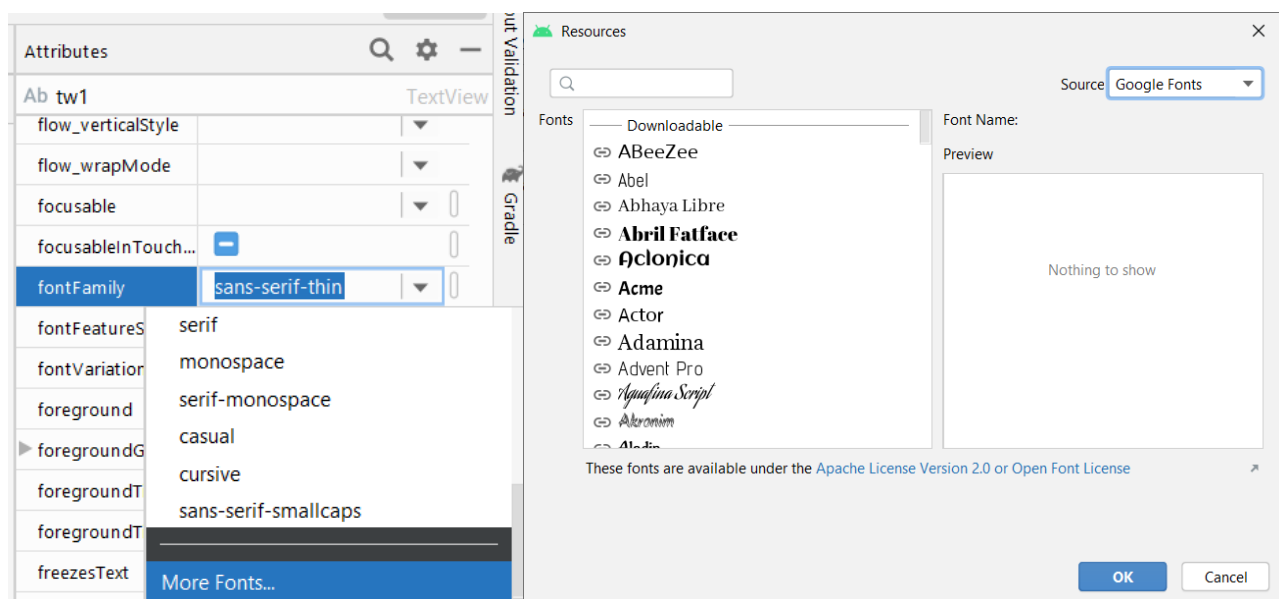


Рис. 5.5. Використання шрифтів Google Fonts

Для завантаження в проєкт інших шрифтів слід представити їх у вигляді ресурсів. Операційна система Android підтримує формати .otf (OpenType) та .ttf (TrueType). Маючи файл із шрифтом, його потрібно зберегти в проєкті. Для коректного підключення шрифтів назви відповідних файлів повинні складатись з латинських літер та знаків підкреслення (рис. 5.6).

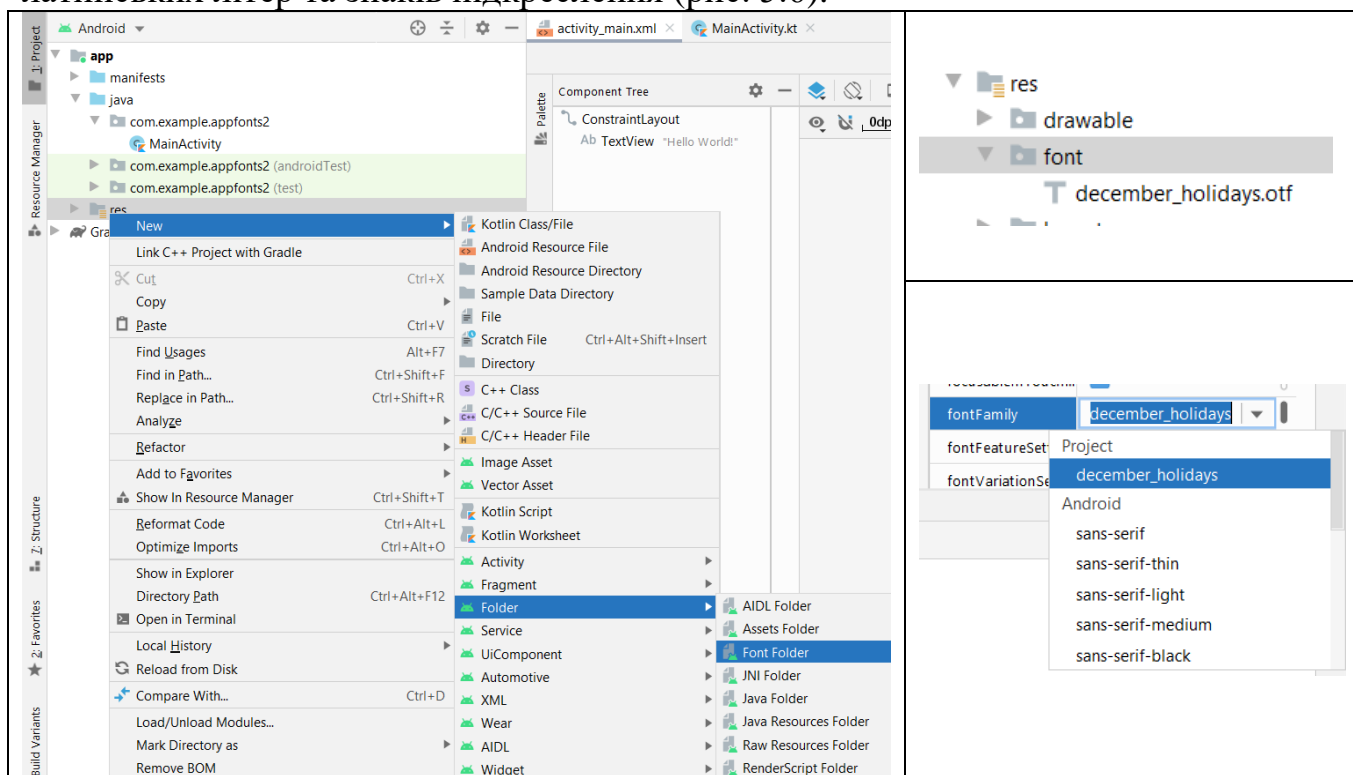


Рис. 5.6. Додавання власного шрифта в проєкт

Допускається й програмне встановлення шрифту з Kotlin-коду. Нехай маємо віджет TextView з ідентифікатором tw1. Тоді слід застосувати метод `getFont()` з класу `ResourcesCompat`, якому слід передати контекст та ідентифікатор шрифта (`R.font.назва_шрифта`):


```
val tw1 = findViewById<TextView>(R.id.tw1) as TextView
tw1.typeface = ResourcesCompat.getFont(this,
                                     R.font.december_holidays)
tw1.setTextSize(TypedValue.COMPLEX_UNIT_SP, 50.toFloat())
```

Також встановимо розмір для тексту в масштабованих пікселях за допомогою методу `setTextSize()`. У результаті отримаємо активність, представлену на рис. 5.7.




Hello World!



Рис. 5.7. Програмно встановлений шрифт

Для розробки іконки додатку вже існує багато сервісів. Наприклад, будемо використовувати інструментарій від компанії Google (розробник – Роман Нурік), що називається [Android Asset Studio](#). До того, як розпочати формування набору іконок, краще мати зображення іконки з роздільною здатністю 512x512 пікселів, бажано в форматі .png. Ви можете завантажити такі зображення з Інтернету або створити в графічному редакторі. Перейшовши в категорію «Launcher icon generator» на вебсторінці сервісу та натиснувши на кнопку Image, можна завантажити зображення з іконкою, результат чого показано на рис. 5.8.

Виконавець може обрати колір фону (Background color) іконки, її форму (Shape), ефекти редагування. Натисніть кнопку  для завантаження готових ресурсів. В отриманому архіві буде міститись набір ресурсів, які слід перенести в проект, замінивши стандартні варіанти. Це можна зробити за допомогою Провідника: розпакуйте архів у відповідну папку res.

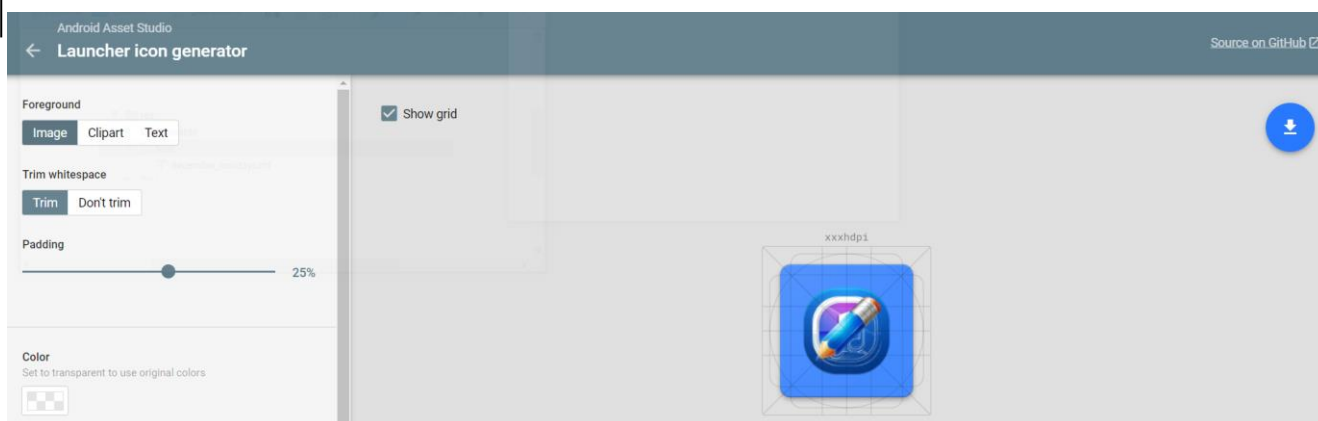


Рис. 5.8. Сервіс Android Asset Studio

Стилізація графічного інтерфейсу відповідно до матеріального дизайну

Стиль — це колекція атрибутів, яка задає вигляд одного представлення (view), серед яких можуть бути колір шрифту, розмір шрифту, колір заливки тощо. Задача стилів — інкапсулювати набір значень атрибутів, що будуть використовуватись повторно, умовно або іншим чином. Вони реалізують бажання відокремити стилізацію від основних макетів інтерфейсу. Атрибут style може застосовуватись до віджета/контейнера, щоб діяти відповідно лише на цей віджет/контейнер, проте стиль автоматично не застосовується до дочірніх представлень. Стиль впливає на контейнер, але не на його вміст. Непрацюючі стилі ігноруються.

Тема — це колекція атрибутів, яка визначає вигляд всього додатку, активності чи ієрархії представлень, тобто працює більш глобально, ніж стиль. При застосуванні теми кожному представленню в додатку чи активності встановлюються визначені значення відповідних атрибутів. Крім того, теми застосовуються і для елементів інтерфейсу, які не є представленнями: status bar, window background та ін.

Стилі та теми зазвичай оголошуються у ресурсному файлі styles.xml, який розміщено в директорії res/values/. У сучасних версіях Android Studio також автоматично створюються файли themes.xml для відокремленої кастомізації тем. Як теми, так і стилі використовують <style>-елементи, проте вони мають різне спрямування. Можна розглядати такі елементи як пари «ключ-значення», в яких ключами є атрибути, а значеннями — ресурси. Атрибути тем відрізняються від атрибутів представлень тим, що вони не застосовуються до окремого типу представлень. Скоріше, це семантично іменовані вказівники на значення, які більш широко застосовуються в додатку. Тема надає конкретні значення цих іменованих ресурсів.

Важливе питання: коли використовувати стилі? Перша рекомендація — застосування кількох семантично однакових представлень. Наприклад, при створенні калькулятора кнопки мають виглядати в єдиному стильовому оформленні. Аналогічно, якщо маєте кілька екранів з одним стильовим виконанням тексту. Загальна ідея: існують представлення (views), які не просто

використовують спільні атрибути, а й відіграють однакову роль у додатку. Вигляд роботи зі стилем представлено в лістингу 5.2.

Лістинг 5.2. Уривки коду для стилізації окремого віджета

```

1                                     styles.xml
2 <style name="Widget.Plaid.Button.InlineAction" parent="...">
3     <item name="android:gravity">center_horizontal</item>
4     <item name="android:textAppearance">
5         @style/TextAppearance.CommentAuthor</item>
6     <item name="android:drawablePadding">
7         @dimen/spacing_micro</item>
8 </style>
9
10                                     activity_main.xml (застосування стилю до атрибута)
11 <Button ...
12     android:gravity="center_horizontal"
13     android:textAppearance="@style/TextAppearance.CommentAuthor"
14     android:drawablePadding="@dimen/spacing_micro"/>
15
16                                     activity_main.xml (застосування стилю до віджета)
17 <Button style="@style/Widget.Plaid.Button.InlineAction"
18     .../>

```

Теми застосовуються для налаштування стандартних стилів у всьому додатку чи ієрархії представлень. Конфігурування окремих елементів тем зав'язане на батьківському стилі, який змінюється залежно від версії операційної системи. До прикладу на рис. 5.9 показано відмінності стилізації стандартних віджетів EditText у різних версіях ОС Android. Демонстрація уривків коду для роботи з темами наведена в лістингу 5.3.

Лістинг 5.3. Уривки коду для стилізації за допомогою тем

```

1                                     AndroidManifest.xml (застосування теми до всього додатку)
2 <application
3     ...
4     android:theme="@style/Theme.AppFonts2">
5
6
7                                     AndroidManifest.xml (застосування теми до активності)
8 <activity android:name=".MainActivity"
9     android:theme="@style/Theme.AppFonts2">
10
11
12                                     activity_main.xml (застосування теми до ієрархії представлень)
13 <androidx.constraintlayout.widget.ConstraintLayout
14     ...
15     android:theme="@style/Theme.AppFonts2">

```

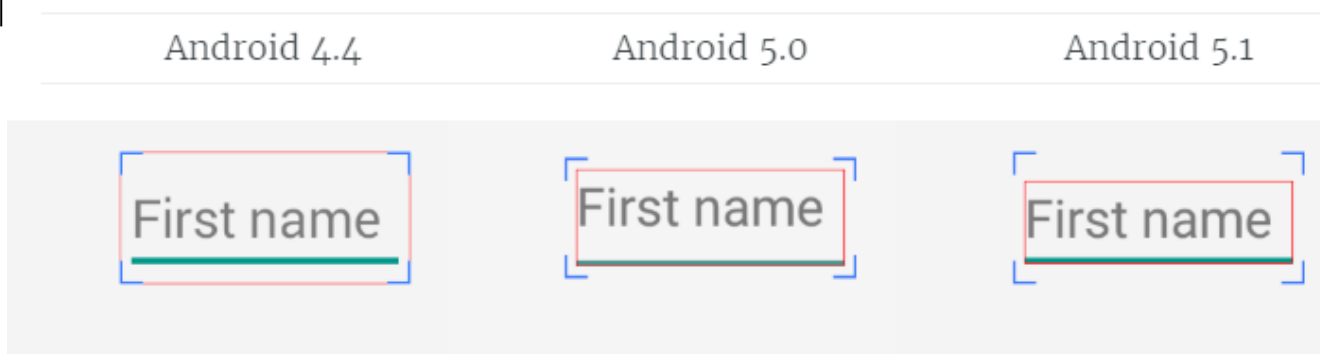


Рис. 5.9. Стилiзацiя вiджета EditText залежно вiд версiї ОС Android

Тему також можна задати програмно в кодi, огортаючи iснуючий контекст в об'єкті класу ContextThemeWrapper.

<https://ataulm.com/2019/11/20/using-context-theme-wrapper.html>

Далi обрану стилiзацiю можна застосувати для рендерингу iнтерфейсу за допомогою LayoutInflator-об'єкта.

Поширенi атрибути для тем налаштовують кольори, розмiри, графiчнi об'єкти (drawables), вигляд тексту, форму об'єкта, стилi кнопок тощо. Визначення кольору здiйснюється такими атрибутами:

- ?attr/colorPrimary – основний колiр брендингу додатка;
- ?attr/colorSecondary – вторинний (акцентний) колiр брендингу додатку, зазвичай яскравий та контрастний вiдносно основного кольору;
- ?attr/colorOn[Primary, Secondary, Surface та iн.] – колiр, контрастний до заданого;
- ?attr/color[Primary, Secondary]Variant – альтернативний вiдтiнок заданого кольору;
- ?attr/colorSurface – колiр поверхонь компонентiв, зокрема карток, меню та iн.;
- ?android:attr/colorBackground – колiр заливки екрану.
- ?attr/colorPrimarySurface – перемикає кольори colorPrimary у Light-темах, colorSurface у Dark-темах;
- ?attr/colorError – колiр для вiдображення помилок;
- ?attr/colorControlNormal – колiр, що застосовується до iконок/елементiв управлiння у їх нормальному станi;
- ?attr/colorControlActivated – колiр, який застосовується до iконок/елементiв управлiння у їх активованому (activated, checked) станi;
- ?attr/colorControlHighlight – колiр, застосований для пiдсвiчування елементiв управлiння (ripple-ефекти, селектори спискiв тощо);
- ?android:attr/textColorPrimary – найбільш помiтний колiр тексту;
- ?android:attr/textColorSecondary – акцентний колiр тексту;

Популярними атрибутами для встановлення розмiрiв у темах є:

- ?attr/listPreferredItemHeight – стандартна (мінімальна) висота елементів списку;
- ?attr/actionBarSize – висота панелі дій (toolbar).

Для роботи з графічними об'єктами часто використовуються атрибути:

- ?attr/selectableItemBackground – ripple-ефект/підсвічування інтерактивних елементів (також зручний для об'єктів переднього плану!);
- ?attr/selectableItemBackgroundBorderless – **unbounded** ripple-ефекти;
- ?attr/dividerVertical – графічний об'єкт, який може використовуватись як вертикальний роздільник між візуальними елементами;
- ?attr/dividerHorizontal – графічний об'єкт, який може використовуватись як горизонтальний роздільник між візуальними елементами.

Масштабування шрифтів управляються різновидами textAppearance-атрибутів. Також ці атрибути відрізняються насиченістю (вагою, weight) шрифтів (рис. 5.10):

- ?attr/textAppearanceHeadline1 – визначає за умовчанням текст зі світлою насиченістю з розміром 96sp;
- ?attr/textAppearanceHeadline2 – визначає за умовчанням текст зі світлою насиченістю та розміром 60sp;
- ?attr/textAppearanceHeadline3 – визначає за умовчанням текст зі звичайною насиченістю та розміром 48sp;
- ?attr/textAppearanceHeadline4 – визначає за умовчанням текст зі звичайною насиченістю та розміром 34sp;
- ?attr/textAppearanceHeadline5 – визначає за умовчанням текст зі звичайною насиченістю та розміром 24sp;
- ?attr/textAppearanceHeadline6 – визначає за умовчанням текст з середньою насиченістю з розміром 20sp;
- ?attr/textAppearanceSubtitle1 – визначає за умовчанням текст зі звичайною насиченістю та розміром 16sp;
- ?attr/textAppearanceSubtitle2 – визначає за умовчанням текст з середньою насиченістю розміром 14sp;
- ?attr/textAppearanceBody1 – визначає за умовчанням текст зі звичайною насиченістю та розміром 16sp;
- ?attr/textAppearanceBody2 – визначає за умовчанням текст зі звичайною насиченістю та розміром 14sp;
- ?attr/textAppearanceCaption – визначає за умовчанням текст зі звичайною насиченістю та розміром 12sp;

- ?attr/textAppearanceButton – визначає за умовчанням текст з середньою насиченістю з усіма великими літерами розміром 14sp;
- ?attr/textAppearanceOverline – визначає за умовчанням текст зі звичайною насиченістю та з усіма великими літерами розміром 10sp.



Рис. 5.10. Вигляд тексту залежно від значення атрибуту textAppearance

Застосована тема може налаштовувати вигляд матеріальних поверхонь додатку, змінюючи їх форму (shaping, рис. 5.11). Візуальні компоненти можна категоріювати за розміром: малі (small), середні (medium) та великі (large). Звідси, маємо популярні атрибути для конфігурування:

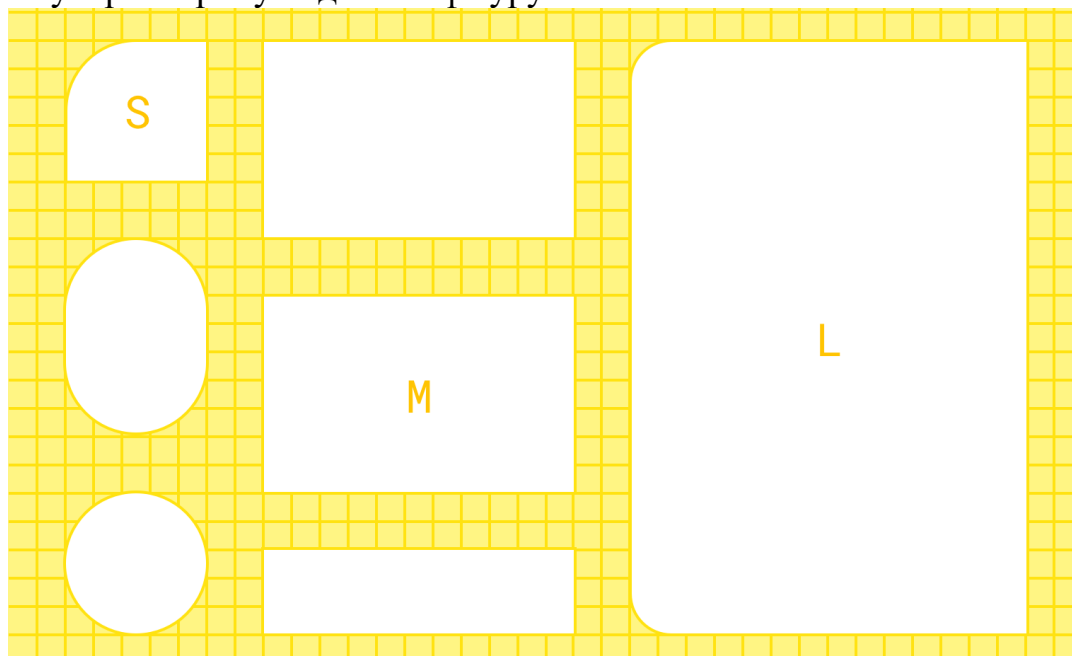


Рис. 5.11. Налаштування форми матеріальних поверхонь

- ?attr/shapeAppearanceSmallComponent використовується для кнопок, Chip-віджетів, текстових полів тощо. За умовчанням має значення скошених кутів (rounded corners) з радіусом 4dp;

- `?attr/shapeAppearanceMediumComponent` застосовується для карток, діалогових екранів, діалогів вибору дати й часу (Date Pickers) та ін. За умовчанням має значення скошених кутів з радіусом 4dp;
- `?attr/shapeAppearanceLargeComponent` використовується для віджетів на зразок Bottom Sheet (спливаюча інформаційна панель). За умовчанням має значення скошених кутів з радіусом 0dp, тобто без заокруглення.

Також слід звернути увагу на налаштування стилізації кнопок у матеріальних темах, оскільки це дуже поширений віджет у додатках. Матеріальний дизайн визначає три види базової стилізації кнопок: Contained, Text та Outlined (рис. 5.12). Теми матеріальних компонентів пропонують відповідні атрибути для стилізації кнопок `MaterialButton`:

- `?attr/materialButtonStyle` передбачає за умовчанням contained-стилізацію;
- `?attr/borderlessButtonStyle` визначає текстову стилізацію кнопок;
- `?attr/materialButtonOutlinedStyle` застосовується для outlined-стилізації.



Рис. 5.12. Стилiзованi кнопки в матерiальних темах

Додатково передбачені атрибути для налаштування прозорості віджетів:

- `?android:attr/disabledAlpha` за умовчанням відключає альфа-канал віджетів;
- `?android:attr/primaryContentAlpha` застосовує альфа-канал до елементів на передньому плані (foreground elements);
- `?android:attr/secondaryContentAlpha` задає альфа-канал для супутніх (secondary) елементів.

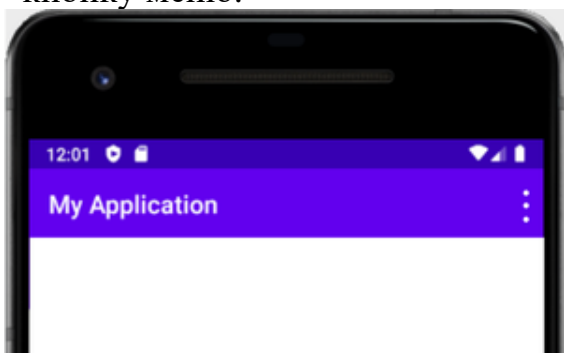
За відсутності доречного атрибуту теми можна створити власний. З цією метою створюється ресурсний файл `attrs.xml`, у якому оголошуються як окремі атрибути, так і теми (елемент `<declare-styleable>`) з наборами власних атрибутів (елемент `<attr>`). Приклад подібного файлу можете побачити [тут](#).

Панель інструментів та використання меню

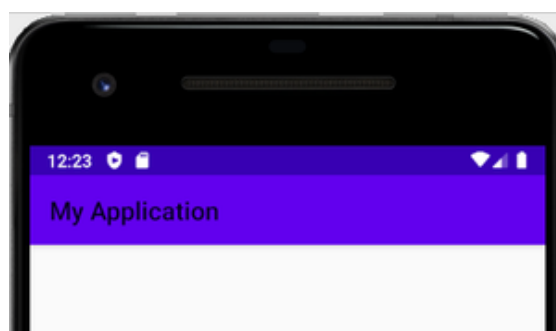
Панель вгорі Android-додатку із заголовком, меню, пошуком та навігацією має довгу еволюційну історію. Вперше вона з'явилась ще до Android 3.0 (API 11) у вигляді панелі додатку (AppBar, рис. 5.13а) та ідейна мала надати більшої унікальності та узгодженості інтерфейсу додатку. На панелі відображався заголовок для поточної активності та кнопка меню (:).

Подальший розвиток панелі було представлено в 2013 році в формі панелі ActionBar для Android 3.0. Візуальний елемент AppBar мав досить обмежену функціональність, тому пропонувались теми, в яких стандартною панеллю був ActionBar (рис. 5.13б), наприклад, Theme.AppCompat.Light.DarkActionBar. У свою чергу, новий візуальний елемент пропонував наступні можливості:

- кнопку для навігації чи бокової панелі (drawer);
- іконку додатку;
- заголовок та підзаголовок;
- кнопку дій (action button, зокрема кнопку пошуку);
- кнопку меню.



(a)



(б)

Рис. 5.13. Старіші форми тулбару

Актуальна форма панелі – Toolbar – з'явилась разом з концепцією матеріального дизайну в Android 5.0 (API 21). Візуальний елемент Toolbar є різновидом ViewGroup-об'єкта, який можна розмістити в XML-макетах та розширює можливості свого попередника ActionBar. Панель Toolbar більш гнучка в контексті зовнішнього вигляду та функціональності. На відміну від ActionBar, її позиція не є строго зафіксованою вгорі активності. Розташовувати Toolbar можливо в будь-якому місці активності як звичайне представлення. Дана панель має зворотну сумісність аж до API 7 (Android 2.1) та має цілий набір компонентів для додавання на неї:

- кнопку для навігації, бокової панелі чи **Up button**;
- брендове лого чи іконку додатку;
- заголовок та підзаголовок;
- елементи ActionMenu;
- багато власних представлень, на зразок TextView, ImageView та ін.

Перша особливість програмної реалізації додатку з панеллю Toolbar полягає в тому, що стандартна тема (файл res/values/themes.xml) породжена від Theme.MaterialComponents.DayNight.DarkActionBar. Тому спершу слід змінити тему на таку, що містить у назві «NoActionBar» замість «DarkActionBar» (рис. 5.14). Після цього потрібно додати Toolbar-елемент у розмітку активності (лістинг 5.4). Від цього моменту можна стилізувати панель інструментів, як це зроблено в лістингу 5.5

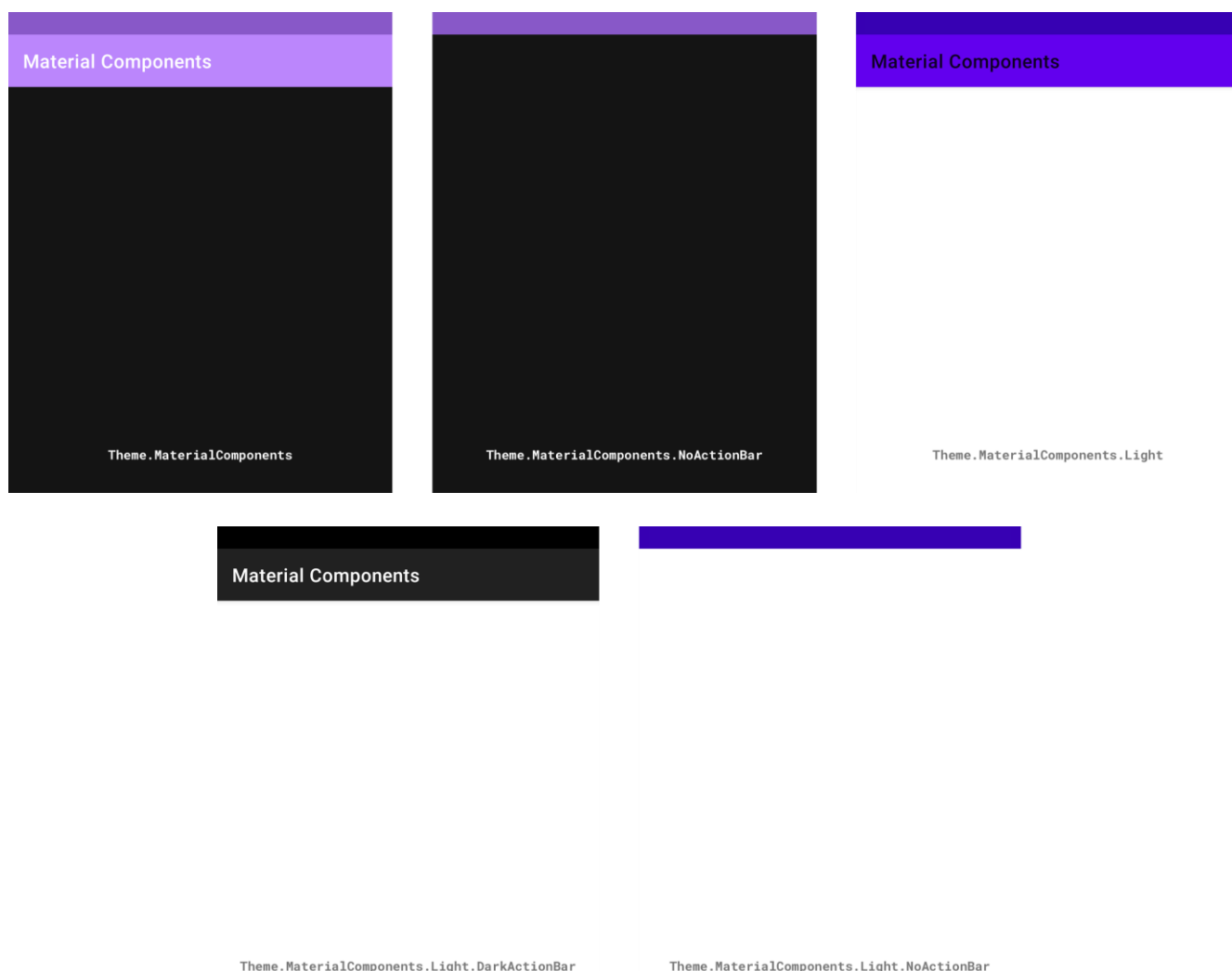


Рис. 5.14. Базові теми з ActionBar та без нього з MDC

Додатково завантажимо іконки для стрілки назад, яка буде присутня на тулбарі. Для спрощення компоновання візьмемо за базовий диспетчер LinearLayout. Оскільки тулбар поводить себе, як звичайний віджет, робимо його дочірнім відносно диспетчера компоновань. Лістинг 5.4 демонструє кілька можливостей тулбару, зокрема використання кнопки навігації, логотипу, заголовкового та підзаголовкового тексту. У якості лого було обрано системну іконку. Результат запуску представлено на рис. 5.15.

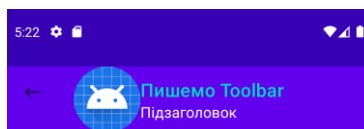
Лістинг 5.4. Базова реалізація тулбару
activity_main.xml

```

1  <LinearLayout
2  xmlns:android="http://schemas.android.com/apk/res/android"
3  xmlns:app="http://schemas.android.com/apk/res-auto"
4  android:orientation="vertical"
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
10
11  <androidx.appcompat.widget.Toolbar

```

```
12         android:id="@+id/toolbar"
13         android:layout_width="match_parent"
14         android:layout_height="wrap_content"
15         android:background="?attr/colorPrimary"
16         app:title="Пишемо Toolbar"
17         app:subtitle="Підзаголовок"
18         app:logo="@android:drawable/sym_def_app_icon"
19         app:titleTextColor="@color/teal_200"
20         app:subtitleTextColor="@color/white" />
21
22 </LinearLayout>
23
24                                     MainActivity.kt
25 override fun onCreate(savedInstanceState: Bundle?) {
26     super.onCreate(savedInstanceState)
27     setContentView(R.layout.activity_main)
28
29     val toolbar = findViewById<Toolbar>(R.id.toolbar)
30     setSupportActionBar(toolbar)
31     toolbar?.navigationIcon = ContextCompat.getDrawable(this,
32         R.drawable.baseline_keyboard_backspace_24)
33     toolbar?.setNavigationOnClickListener {
34         Toast.makeText(applicationContext, "Натиснуто на іконку
35             навігації!", Toast.LENGTH_SHORT).show()
36     }
37 }
```



Натиснуто на іконку навігації!



Рис. 5.15. Вигляд активності після натиснення на кнопку навігації

Додамо до представленого інтерфейсу кнопку меню. Спочатку слід визначити структуру меню у вигляді XML-ресурсу. Додамо в проєкт відповідну директорію та ресурс, як показано на рис. 5.16.

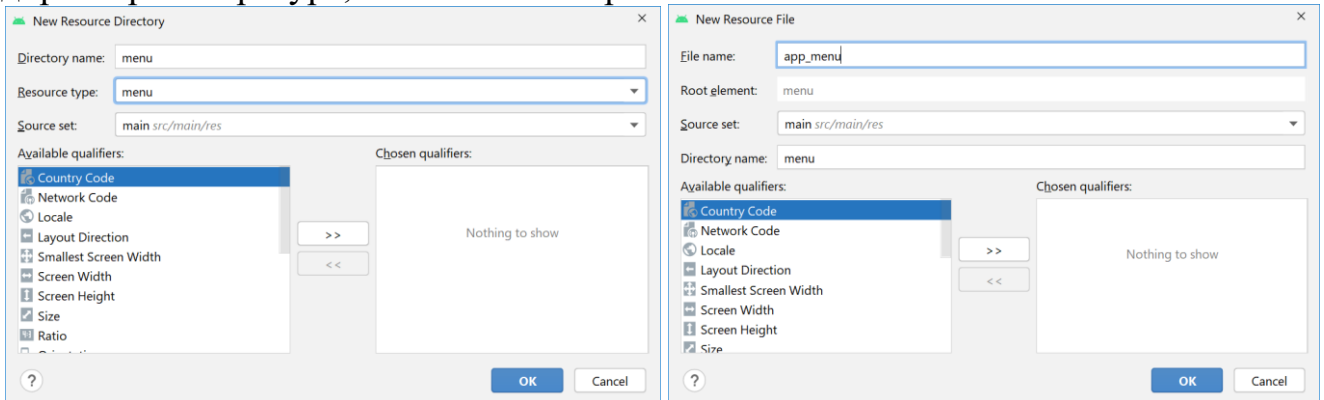


Рис. 5.16. Додавання директорії для ресурсів меню

Детальну довідку щодо атрибутів для окремих пунктів меню (<item>) можна знайти [за посиланням](#). Найбільш поширені з них – це id, title, icon, showAsAction. Останній атрибут визначає режим показу пункту меню:

- never – не показувати пункт меню на тулбарі (розміщувати в контекстному меню :);
- ifRoom – показувати пункт на тулбарі, якщо є місце;
- always – показувати завжди на тулбарі;
- withText – елемент буде показуватись лише з його заголовком;
- collapseActionView – елемент може згортатись у кнопку або розгортатись на всю ширину тулбару при натисненні.

Загалом ви можете кастомізувати тулбар, додаючи в його піддерево елементів інші диспетчери компоновань та віджети.

Диспетчер компоновань ConstraintLayout

Для реалізації фронтенду мобільних додатків під операційну систему Android нині рекомендують використовувати бібліотеку MDC (Material Design Components). Вона постачає стилізовані версії стандартних віджетів, таких як кнопки, тулбари перемикачі та ін. Також бібліотека має інструменти для забезпечення оформлення віджетів відповідно до темної теми, реалізує анімовані міжекранні переходи, розширяє стандартні диспетчери компоновань та ін.

<https://medium.com/androiddevelopers/we-recommend-material-design-components-81e6d165c2dd>

Диспетчер компонувань *ConstraintLayout* є основою макетування інтерфейсу сучасних Android-додатків та був представлений у 2016 році. Його можна розглядати як потужнішу версію диспетчера компонувань *RelativeLayout* з багатьма покращеннями та новими властивостями. Проте *ConstraintLayout* є одним з найбільш ресурсозатратних диспетчерів компонувань, тому слід ретельно підбирати кореневий елемент для макетування. Також, на відміну від *LinearLayout* чи *FrameLayout*, дуже не рекомендується вкладати *ConstraintLayout* при проєктуванні інтерфейсу.

Одна з причин використання *ConstraintLayout* – створення респонсивного, адаптивного користувацького інтерфейсу. Також перевагою даного диспетчера є спрощення ієрархії представлень при побудові насиченого віджетами інтерфейсу користувача. Розмітка, що використовує *ConstraintLayout*, більш читабельна та зменшує кількість шаблонного (boilerplate) коду.

Диспетчер *ConstraintLayout* концентрується на відносному позиціонуванні, як і *RelativeLayout*. Обмеження (constraints) можуть застосовуватись як відносно іншого представлення, так і напряду стосовно батьківського елемента інтерфейсу. Приклад реалізації відносного позиціонування представлено в лістингу 5.5 та на рис. 5.15.

Лістинг 5.5. Демонстрація обмежень *ConstraintLayout*

```
1 <androidx.constraintlayout.widget.ConstraintLayout
2 xmlns:android="http://schemas.android.com/apk/res/android"
3   xmlns:app="http://schemas.android.com/apk/res-auto"
4   android:layout_width="match_parent"
5   android:layout_height="match_parent">
6
7   <Button
8       android:id="@+id/button_center"
9       android:layout_width="wrap_content"
10      android:layout_height="wrap_content"
11      android:text="Centered"
12      app:layout_constraintBottom_toBottomOf="parent"
13      app:layout_constraintEnd_toEndOf="parent"
14      app:layout_constraintStart_toStartOf="parent"
15      app:layout_constraintTop_toTopOf="parent" />
16
17   <Button
18       android:id="@+id/button_right"
19       android:layout_width="wrap_content"
20       android:layout_height="wrap_content"
21       android:layout_marginStart="32dp"
22       android:text="Right"
23       app:layout_constraintBottom_toBottomOf="parent"
24       app:layout_constraintStart_toEndOf="@id/button_center"
25       app:layout_constraintTop_toTopOf="parent" />
26
27   <Button
28       android:id="@+id/button_left"
```

```

29         android:layout_width="wrap_content"
30         android:layout_height="wrap_content"
31         android:layout_marginEnd="32dp"
32         android:text="Left"
33         app:layout_constraintBottom_toBottomOf="parent"
34         app:layout_constraintEnd_toStartOf="@id/button_center"
35         app:layout_constraintTop_toTopOf="parent" />
36
37     <Button
38         android:id="@+id/button_above"
39         android:layout_width="wrap_content"
40         android:layout_height="wrap_content"
41         android:layout_marginBottom="32dp"
42         android:text="Above"
43         app:layout_constraintBottom_toTopOf="@id/button_center"
44         app:layout_constraintEnd_toEndOf="parent"
45         app:layout_constraintStart_toStartOf="parent" />
46
47     <Button
48         android:id="@+id/button_below"
49         android:layout_width="wrap_content"
50         android:layout_height="wrap_content"
51         android:layout_marginTop="32dp"
52         android:text="Below"
53         app:layout_constraintEnd_toEndOf="parent"
54         app:layout_constraintStart_toStartOf="parent"
55         app:layout_constraintTop_toBottomOf="@id/button_center"
56 />
57
58     <Button
59         android:id="@+id/button_bottom"
60         android:layout_width="0dp"
61         android:layout_height="wrap_content"
62         android:text="Bottom"
63         app:layout_constraintBottom_toBottomOf="parent"
64         app:layout_constraintEnd_toEndOf="parent"
65         app:layout_constraintStart_toStartOf="parent" />
66
67     <Button
68         android:id="@+id/button_top"
69         android:layout_width="0dp"
70         android:layout_height="wrap_content"
71         android:text="Top"
72         app:layout_constraintEnd_toEndOf="parent"
73         app:layout_constraintStart_toStartOf="parent"
74         app:layout_constraintTop_toTopOf="parent" />
75 </androidx.constraintlayout.widget.ConstraintLayout>

```

За допомогою властивостей-обмежень (constraint properties) можна вирівнювати елементи один відносно одного без використання вкладених диспетчерів компоновки. Також можна не переживати через розмір екрану:

інтерфейс автоматично адаптуватиметься під наявний екран. Зверніть увагу, що для коректної роботи margin-атрибутів у певному напрямку необхідно задати обмеження в цьому напрямку! Інакше поле (margin) не спрацює.

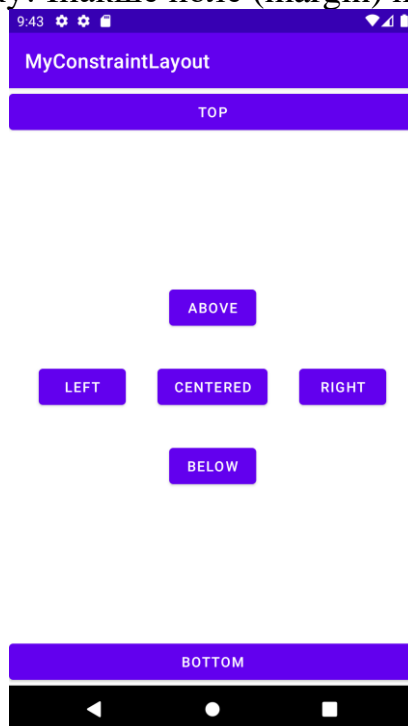


Рис. 5.15. Вигляд користувацького інтерфейсу з лістингу 5.5

Працювати з ConstraintLayout дуже зручно в режимі дизайну Android Studio. При перетягуванні елементу інтерфейса з панелі Palette можна в візуальному режимі протягувати обмеження відносно інших елементів за допомогою якірних точок (anchor points), як це показано на рис. 5.16. Тим не менше, встановлені обмеження реально спрацюють лише тоді, коли будуть присутні принаймні одне вертикальне та одне горизонтальне обмеження. Перевірити дану вимогу можна в XML-коді. Якщо інтерфейс буде незбалансовано по вертикалі чи горизонталі, у розмітці будуть присутні обмеження абсолютного позиціонування (`tools:layout_editor_absolute_X`, `tools:layout_editor_absolute_Y`) замість обмежень відносного позиціонування. Також налаштування макету досить просто здійснюється з бокової панелі Attributes разом з визначенням полів (margin) стосовно обраного елементу інтерфейсу.

Розташування елементів інтерфейсу один відносно одного супроводжується match-обмеженнями (match constraints, 0dp). Це означає, що відповідна розмірність елементу буде обчислюватись у відповідності до розмірності прив'язаних до неї інших елементів. Продемонструємо дану особливість на прикладі ширини кнопки (властивості `layout_width`), яка прив'язана до двох інших кнопок, як показано на рис. 5.17. Також можна задавати атрибути `app:layout_constraintWidth_min`, `app:layout_constraintWidth_max`, щоб визначати мінімальну та максимальну ширину елемента при match-обмеженнях. Аналогічні обмеження стосуватимуться і висоти.

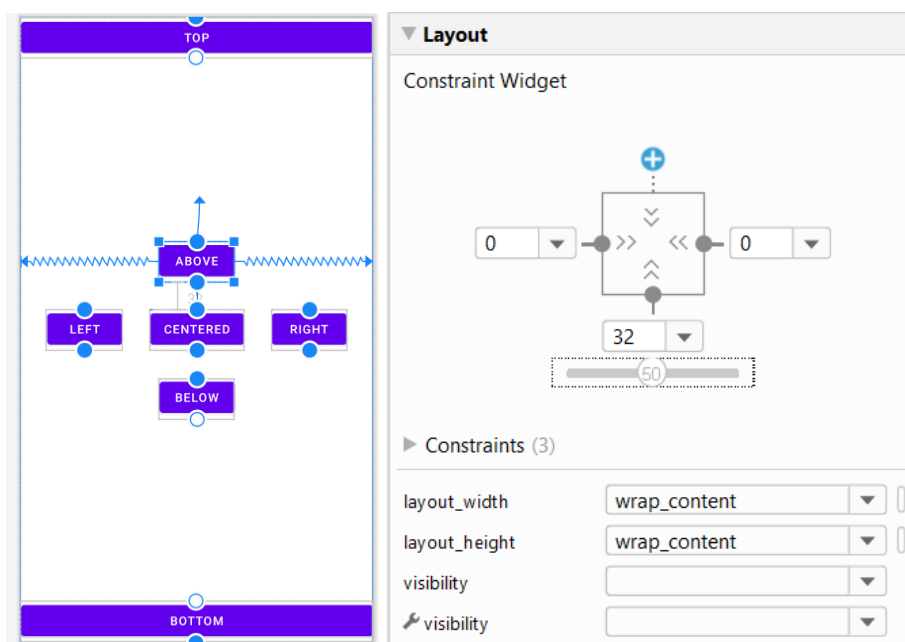
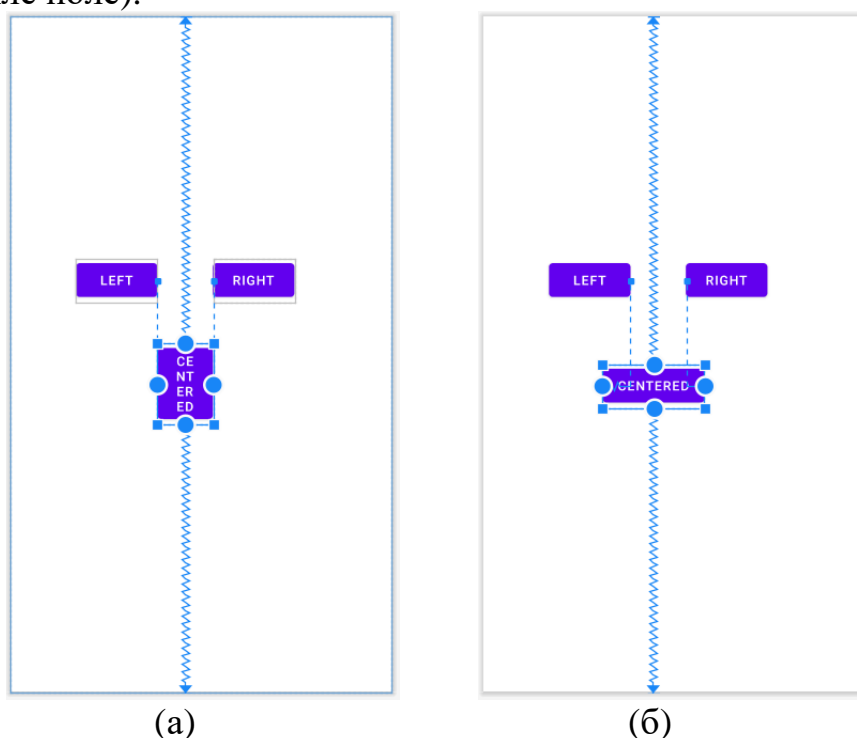


Рис. 5.16. Візуальне конструювання обмежень для елементів інтерфейсу

Одним із важливих завдань макетування є забезпечення видимості окремих елементів інтерфейсу. Нехай на макеті присутні багато елементів інтерфейсу, проте, згідно з вимогами, показувати потрібно лише деякі з них. Проблемою стає розташування окремих представлень, коли їх батьківський елемент перестає відображатись. Для цього у ConstraintLayout представлено таке поняття, як «Gone Margin» (зникле поле).



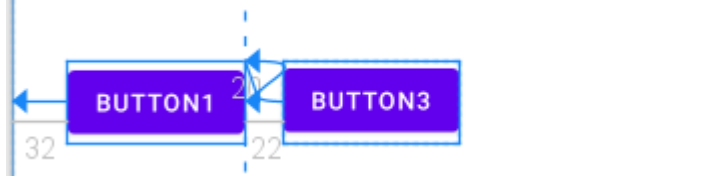
**Рис. 5.17. Адаптивність ширини кнопки при значеннях
(а) match_constraint (0dp), (б) wrap_content**

Уявімо, що користувацький інтерфейс організовано так, як це показано на рис. 5.18. Потрібно, щоб елемент А перестав відображатись, а елемент В

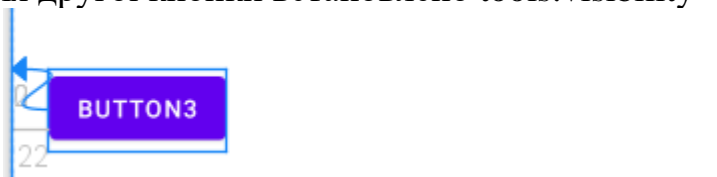
автоматично змістився на колишню позицію елемента А. Для цього зазвичай використовують атрибут `tools:visibility` із значенням `"gone"`. Інколи корисно зберігати поле певного розміру відносно краю екрану, як показано на рис. 5.18е. Перша кнопка має `margin` розміром 32dp, а друга – 8dp. При зникненні першої кнопки відступ від лівого краю для другої кнопки не зберігається (рис. 5.18д). За потреби, розмір такого поля можна зберегти, записавши для решти елементів ланцюга атрибут `app:layout_goneMarginStart` з однаковим значенням. Доступні інші подібні атрибути: `layout_goneMarginEnd`, `layout_goneMarginLeft`, `layout_goneMarginTop`, `layout_goneMarginRight` та `layout_goneMarginBottom`.



(а) кнопки пов'язані в ланцюг та всі видимі



(б) для другої кнопки встановлено `tools:visibility="gone"`



(в) для першої та другої кнопки встановлено `tools:visibility="gone"`



(г) дві кнопки з полями зліва 32dp і 8dp відповідно









(д) перша кнопка зникає, для другої кнопки використовується стандартний `margin`



(е) перша кнопка зникає, для другої задано `app:layout_goneMarginStart="32dp"`

Рис. 5.18. Робота з полями при використанні `ConstraintLayout`

При виборі елемента управління в режимі дизайну на панелі інструментів візуального редактора Android Studio будуть присутні кілька кнопок (рис. 5.19):

-  View Options дозволяє вивести задані обмеження в різних режимах, з нашаруванням елементів системного інтерфейсу, розмітки полів, візуальних підказок чи приховуванням невибраних представлень та ін.;
-  Enable Autoconnection to Parent – при ввімкненні обмеження будуть автоматично налаштовуватись при перетягуванні представлень в область попереднього перегляду;
-  Default Margins – стандартне значення полів (відступів). Можна встановлювати окремо для кожного елементу;
-  Clear All Constraints – видаляє всі обмеження з макету;
-  Infer Constraints – автоматично створює обмеження; спрацьовує тільки при натисненні кнопки. Функціональність Autoconnect спрацьовує при кожному додаванні нового компоненту на макет;
-  GuideLines – має дві опції: Add Vertical GuideLine та Add Horizontal GuideLine. Передбачає невидимі лінії, які можуть створюватись горизонтально та вертикально та застосовуватись, щоб накладати обмеження на представлення (рис. 5.19). Наприклад, можна розділити екран на кілька еквівалентних частин або задати start та end поля. Це зменшує кількість шаблонного коду. Приклад доповнення розмітки гайдлайнами наведено в лістингу 5.6.

Лістинг 5.6. Уривок коду впровадження гайдлайнів у ConstraintLayout

```

1 <androidx.constraintlayout.widget.Guideline
2     android:id="@+id/guideline_vertical_start"
3     android:layout_width="wrap_content"
4     android:layout_height="wrap_content"
5     android:orientation="vertical"
6     app:layout_constraintGuide_begin="16dp" />
7
8 <androidx.constraintlayout.widget.Guideline
9     android:id="@+id/guideline_vertical_end"
10    android:layout_width="wrap_content"
11    android:layout_height="wrap_content"
12    android:orientation="vertical"
13    app:layout_constraintGuide_end="16dp" />
14
15 <androidx.constraintlayout.widget.Guideline
16    android:id="@+id/guideline_horizontal1"
17    android:layout_width="wrap_content"
18    android:layout_height="wrap_content"
19    android:orientation="horizontal"
20    app:layout_constraintGuide_percent="0.25" />
21
22 <androidx.constraintlayout.widget.Guideline
23    android:id="@+id/guideline_horizontal2"
24    android:layout_width="wrap_content"

```

```
25     android:layout_height="wrap_content"
26     android:orientation="horizontal"
27     app:layout_constraintGuide_percent="0.50" />
28
29 <androidx.constraintlayout.widget.Guideline
30     android:id="@+id/guideline_horizontal3"
31     android:layout_width="wrap_content"
32     android:layout_height="wrap_content"
33     android:orientation="horizontal"
34     app:layout_constraintGuide_percent="0.75" />
```

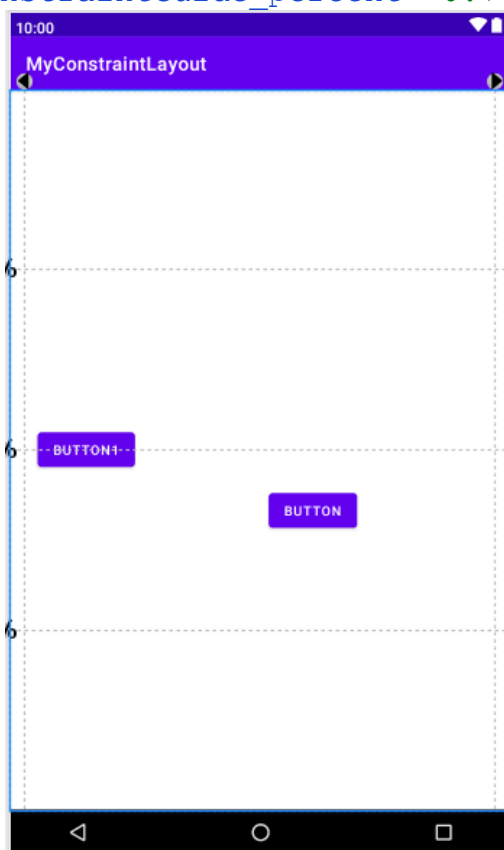


Рис. 5.19. Результат впровадження гайдлайнів з лістингу 5.6

Ще однією корисною можливістю наряду з гайдлайнами є бар'єри. Їх основна мета полягає в створенні невидимого бар'єра під час виконання додатку, щоб забезпечити неможливість його перетинання відповідними компонентами інтерфейсу незалежно від розмірних змін.

Інша можливість, спільна з `LinearLayout`, – це ланцюги (chains) пов'язаних компонентів, які дозволяють вишикувати їх у рядок чи стовпчик. Розташування таких елементів визначається атрибутами `layout_constraintHorizontal_chainStyle` чи `layout_constraintVertical_chainStyle` для першого елементу ланцюга. Поведінка ланцюга буде регламентуватись відповідним стилем:

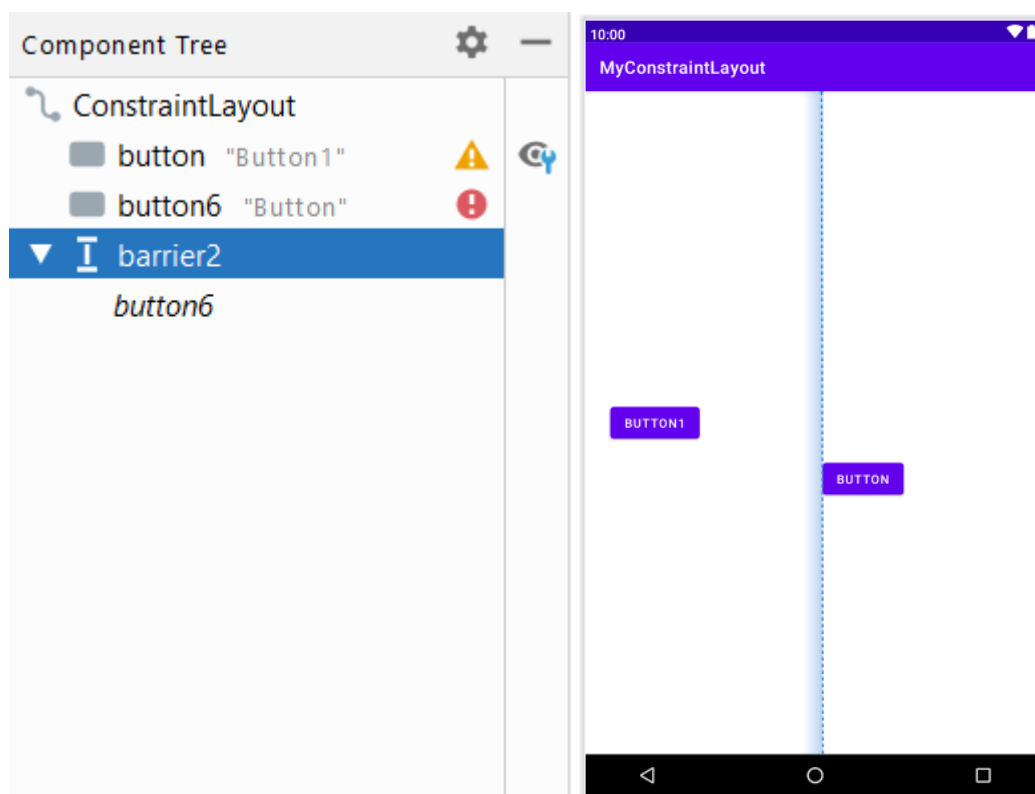


Рис. 5.20. Застосування бар'єру до компоненту інтерфейсу button6

- CHAIN_SPREAD: поведінка за умовчанням. Рівномірно розташовує всі представлення в ланцюгу;
- CHAIN_SPREAD_INSIDE: позиції першого та останнього елементів притягуються до країв, решта елементів розташовуються рівномірно;
- CHAIN_PACKED: елементи ланцюга будуть «запаковані» разом. Атрибут горизонтального чи вертикального відхилення (bias) для дочірнього елемента вплине на розташування «запакованих» елементів.

Розміщення елементів у ланцюгу відповідно до заданої поведінки представлено на рис. 5.21.

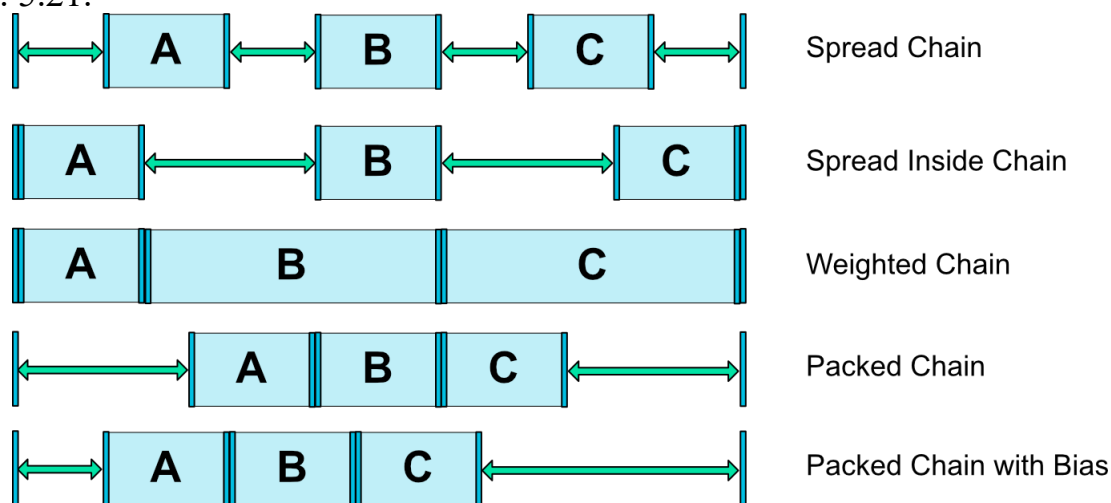


Рис. 5.21. Стилі поведінки ланцюга

Додатково можна ввести групи представлень. Це особливо корисно, коли потрібно здійснити деяку операцію відразу над кількома компонентами

інтерфейсу, зокрема управляти видимістю. Один зі способів це зробити – внести їх у спільний диспетчер компоновувань та керувати його видимістю відповідно. Проте тепер повертаємось до проблеми надмірної вкладеності компонентів інтерфейсу, а також ситуацій, коли немає можливості розмістити згруповані елементи в одному диспетчері компоновувань через їх розташування (між ними можуть бути інші елементи, які не відносяться до групи). Звідси, зручно додати представлення в групу або в XML, або в редакторі, вибравши пункт меню Add Group (рис. 5.22).

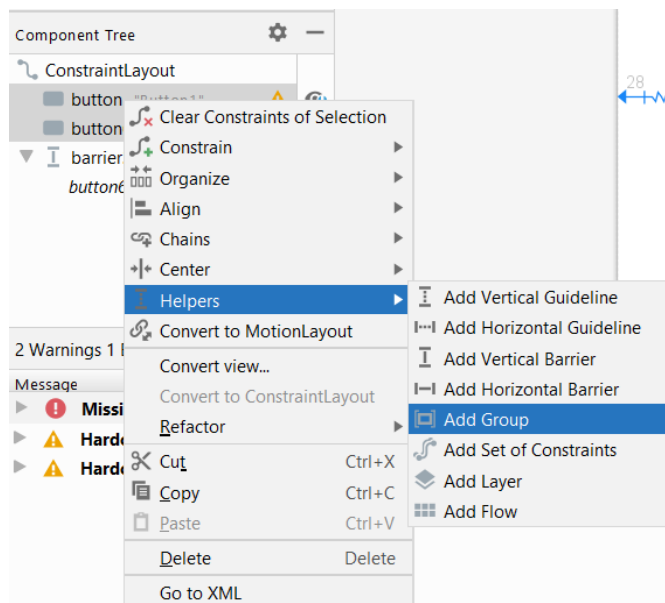


Рис. 5.22. Додавання представлень у групу

<https://medium.com/swlh/constraint-layout-8b13ace936df>

Спискові елементи управління

Основним компонентом інтерфейсу для представлення списків та сіток елементів нині є RecyclerView. Застарілий віджет ListView був досить обмежений у кастомізації вигляду своїх елементів, переважно використовуючись для представлення рядкових списків. Сіткова організація спискових елементів раніше покладалась на компонент GridView. Разом з матеріальним дизайном також було представлено і більш універсальний списковий компонент RecyclerView.

Нагадаємо загальну процедуру роботи зі списковими елементами управління. З цією метою використовуються адаптери, які займаються відображенням даних із джерела на візуальні елементи списку. Адаптер займається заповненням (inflation) визначеної області екрану відповідно до XML-шаблону розмітки елементів списку. Кожне представлення такої розмітки приходить з унікальним ідентифікатором, що застосовується для додавання логіки роботи представлення та кастомізації зовнішнього вигляду. Сформований адаптер передається відповідним ListView- або GridView-компонентам для рендерингу на екрані.

Одна з ключових проблем – відносно неефективне використання пам'яті згаданими візуальними компонентами. Вони можуть перевикористати (recycle)

макет елемента списку, проте не його дочірні компоненти. Таким чином, доводилось викликати `findViewById()` для звернення до окремих віджетів окремих елементів списку при кожному виклику методу `getView()` адаптера. Насичені інтерфейси можуть спричиняти суттєве навантаження на пам'ять та процесор пристрою, зробити інтерфейс «підлагуючим».

Початкове вирішення даної проблеми – використання паттерну View Holder для підвищення плавності прокручування списків. Ідея полягала у створенні ін-методу-посилання на всі представлення, потрібні для заповнення макету списку. Ви визначаєте ці посилання один раз і повторно використовуєте їх без втрат продуктивності через повторювані виклики `findViewById()`. Зауважте, що для `ListView` або `GridView` це рішення є необов'язковим, тому без знання цієї особливості дані компоненти будуть повільно працювати в реальній роботі.

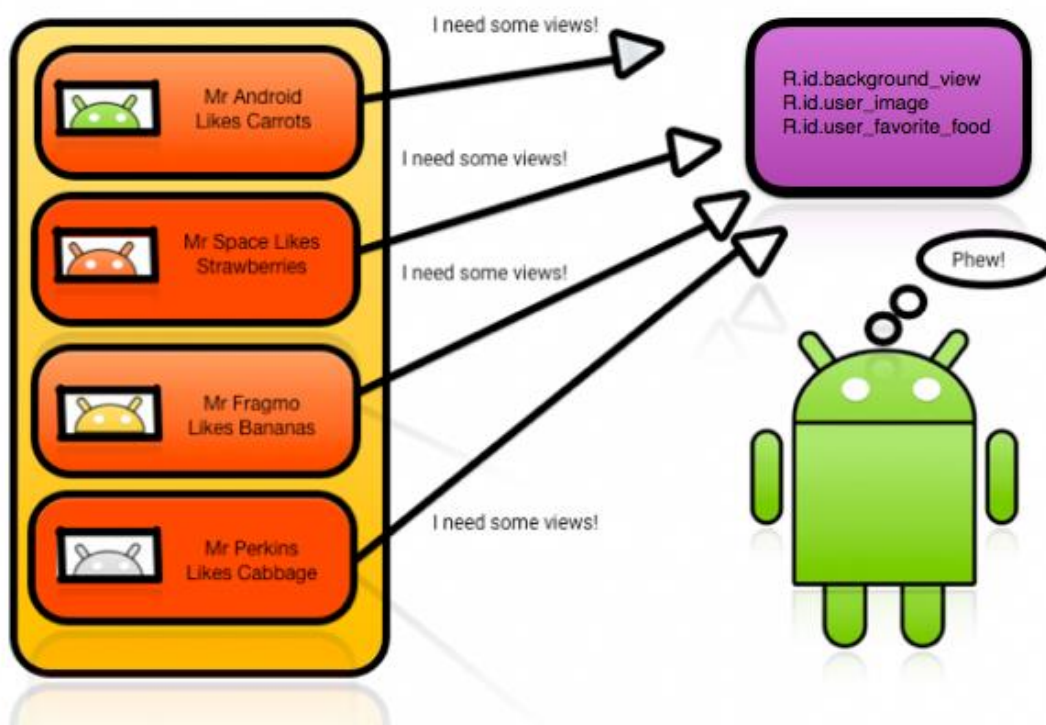


Рис 5.23. Ідеологія використання паттерну ViewHolder

Поява компоненту `RecyclerView` все змінила. Структурно основними частинами компоненту `RecyclerView` є наступні:

- `RecyclerView.Adapter`;
- `RecyclerView.LayoutManager`;
- `RecyclerView.ItemAnimator`;
- `RecyclerView.ViewHolder`.

Він все ще використовує адаптер для джерела даних, проте також зобов'язує створювати `ViewHolder`-и для зберігання посилань у пам'яті. Для постачання нового представлення `RecyclerView` буде або створювати новий `ViewHolder`-об'єкт для заповнення макету та утримування посилань, або перевикористає (`recycle`) певний вже існуючий `ViewHolder`-а (рис. 5.24).

Додатковою перевагою RecyclerView є використання заготовлених анімацій, хоч розробник також може задавати власні.

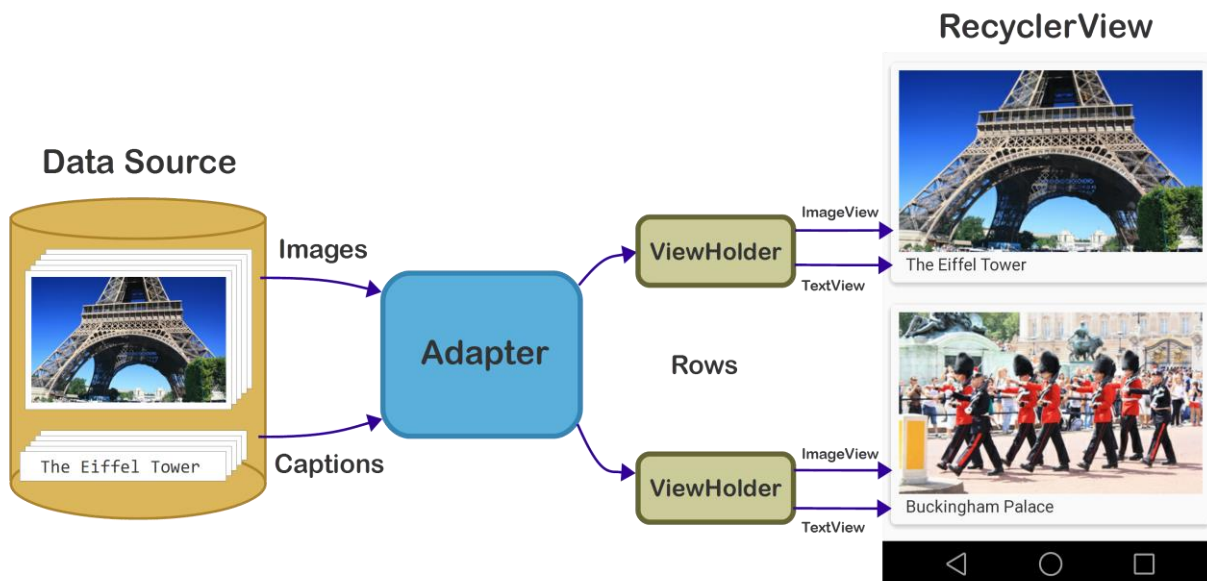


Рис. 5.24. Принцип роботи компонента RecyclerView

Найбільш цікавою частиною RecyclerView є його менеджер макету. Об'єкт класу `LayoutManager` розташовує елементи RecyclerView та вказує, коли можна повторно звернутись до елементів списку, які вийшли за межі області екрану. Елемент управління RecyclerView впорядковує дану функціональність так, щоб дозволити різне макетування: вертикальні списки, горизонтальні списки, сітки (grid), асиметричні (staggered) сітки або ваш власний макет (рис. 5.25). У той же час для ListView характерне макетування лише на основі вертикального списку. Менеджеру макетів за умовчанням не передбачено, тому за такої потреби цю дію можна прописати в коді. Дуже часто базовим макетом стає `LinearLayoutManager`, тому є можливість задати його як з XML-розмітки:

```
<androidx.recyclerview.widget.RecyclerView ...  
    app:layoutManager="LinearLayoutManager" />
```

так і в Kotlin-коді:

```
if (recyclerView.layoutManager == null)  
    recyclerView.layoutManager = LinearLayoutManager(this)
```

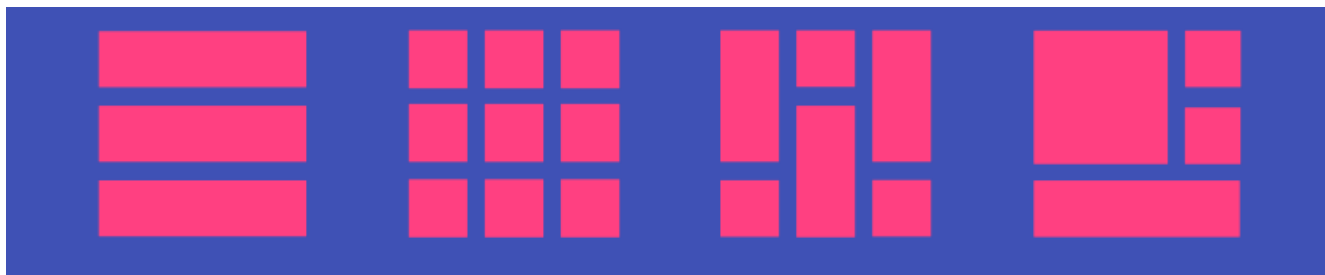


Рис. 5.25. Макети LayoutManager зліва направо: вертикальний список (LinearLayoutManager), сітка (GridLayoutManager), асиметрична сітка (StaggeredGridLayoutManager), змішаний (mixed) макет

Для початку роботи додамо новий layout-ресурс (рис. 5.26), щоб сформуванати макет окремого елемента списку. Він складатиметься із зображення, тексту з датою зйомки цього зображення та описом до фотографії (лістинг 5.7).

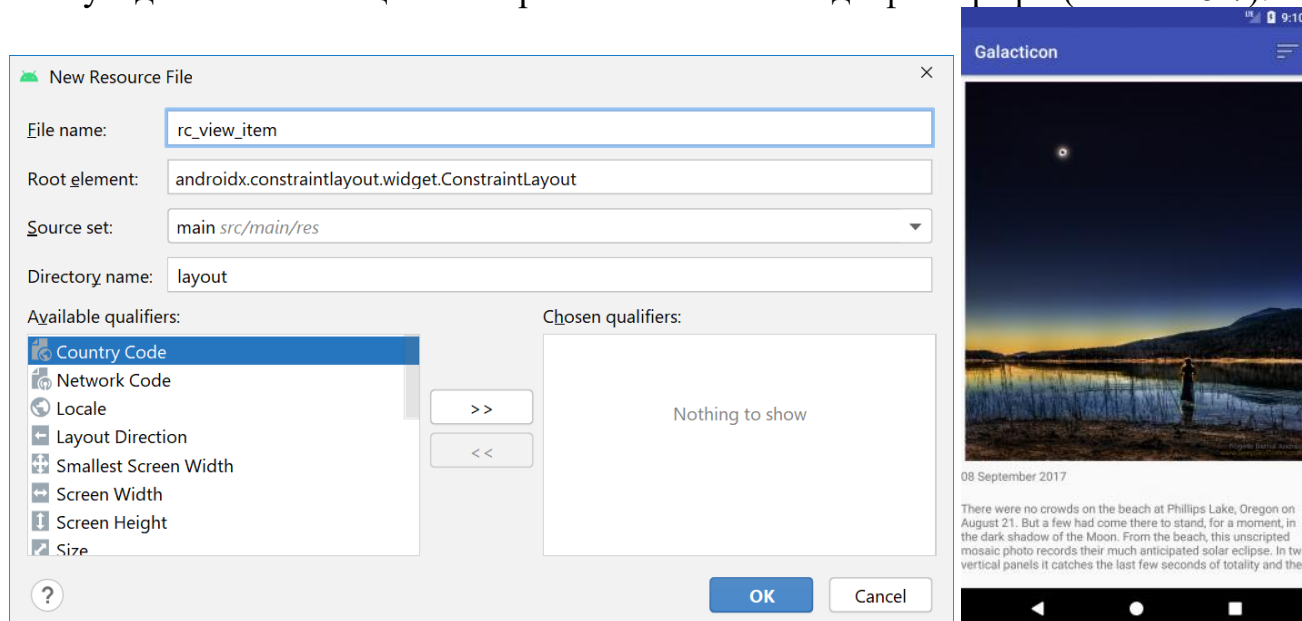


Рис. 5.26. Додавання нового layout-ресурсу

Лістинг 5.7. Розмітка макету для елемента списку

```

1  <androidx.constraintlayout.widget.ConstraintLayout
2      xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:app="http://schemas.android.com/apk/res-auto"
4      xmlns:tools="http://schemas.android.com/tools"
5      android:padding="8dp"
6      android:layout_width="match_parent"
7      android:layout_height="wrap_content">
8      <ImageView
9          android:id="@+id/itemImage"
10         android:layout_width="0dp"
11         android:layout_height="wrap_content"
12         android:layout_marginTop="8dp"
13         android:adjustViewBounds="true"
14         app:layout_constraintBottom_toTopOf="@+id/itemDate"
15         app:layout_constraintEnd_toEndOf="parent"
16         app:layout_constraintHorizontal_bias="0.5"
17         app:layout_constraintStart_toStartOf="parent"
18         app:layout_constraintTop_toTopOf="parent"
19         app:layout_constraintVertical_bias="0.74" />
20
21     <TextView
22         android:id="@+id/itemDate"
23         android:layout_width="0dp"
24         android:layout_height="wrap_content"
25         android:layout_gravity="top|start"
26         android:layout_marginTop="8dp"
27         android:layout_weight="1"
28         app:layout_constraintBottom_toTopOf="@+id/itemDescription"

```

```
29         app:layout_constraintEnd_toEndOf="parent"
30         app:layout_constraintHorizontal_bias="0.5"
31         app:layout_constraintStart_toStartOf="parent"
32         app:layout_constraintTop_toBottomOf="@+id/itemImage"
33         tools:text="Some date" />
34
35     <TextView
36         android:id="@+id/itemDescription"
37         android:layout_width="wrap_content"
38         android:layout_height="wrap_content"
39         android:layout_gravity="center|start"
40         android:layout_weight="1"
41         android:ellipsize="end"
42         android:maxLines="5"
43         app:layout_constraintBottom_toBottomOf="parent"
44         app:layout_constraintEnd_toEndOf="parent"
45         app:layout_constraintHorizontal_bias="0.5"
46         app:layout_constraintStart_toStartOf="parent"
47         app:layout_constraintTop_toBottomOf="@+id/itemDate" />
48
49 </androidx.constraintlayout.widget.ConstraintLayout>
```

Разом з цим додаємо компонент RecyclerView на макет активності та дописуємо базовий код для роботи з ним (лістинг 5.8). Підключаємо логіку роботи компонента RecyclerView та встановлюємо базовим менеджером макету LinearLayoutManager.

Лістинг 5.7. Розмітка макету для елемента списку
activity_main.xml

```
1
2 ...
3 <androidx.recyclerview.widget.RecyclerView
4     android:id="@+id/recyclerView"
5     android:layout_width="match_parent"
6     android:layout_height="match_parent"
7     android:scrollbars="vertical"/>
8 ...
9
10                                     MainActivity.kt
11 private lateinit var layoutManager: LinearLayoutManager
12
13 override fun onCreate(savedInstanceState: Bundle?) {
14     ...
15     val recyclerView =
16         findViewById<RecyclerView>(R.id.recyclerView)
17     layoutManager = LinearLayoutManager(this)
18     recyclerView.layoutManager = layoutManager
19 }
```

Створюємо адаптер для компонента RecyclerView в новому Kotlin-класі на базі абстрактного класу RecyclerView.Adapter<RecyclerView.ViewHolder>, де

PhotoHolder – це майбутній клас для опису даних (рис. 5.27). Загалом до обов’язків адаптера відносяться такі дії:

- створення ViewHolder-ів;
- заповнення ViewHolder-ів інформацією;
- сповіщення RecyclerView щодо того, які елементи змінилися;
- обробка дотиків;
- часткове оновлення даних;
- управління кількістю ViewType-ів;
- інформація щодо перевикористання ViewHolder-а.

Для реалізації пропонується три базових методи адаптера RecyclerView:

- onCreateViewHolder(parent: ViewGroup, viewType: Int): RecyclerView.Adapter.PhotoHolder – створює новий ViewHolder-об’єкт, коли він потрібний RecyclerView. Тут передбачено **Это тот момент, когда создаётся layout строки списка, передается объекту ViewHolder, и каждый дочерний view-компонент может быть найден и сохранен.**
- onBindViewHolder(holder: RecyclerView.Adapter.PhotoHolder, position: Int) – приймає об’єкт ViewHolder і встановлює необхідні дані для відповідного візуалізованого елемента списку;
- getItemCount(): Int – повертає загальну кількість елементів списку. Значення списку передаються за допомогою конструктора.

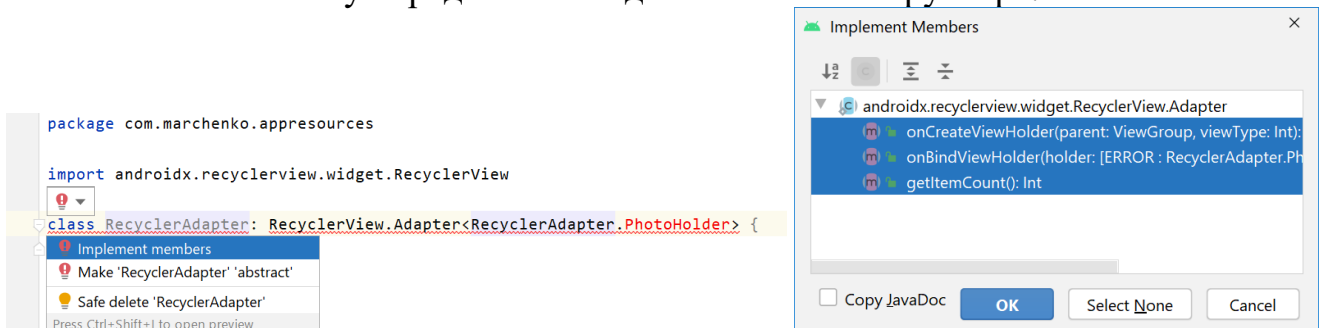


Рис. 5.27. Перші кроки зі створення адаптера

Додамо в клас адаптера основний конструктор, який прийматиме списковий масив фотографій:

```
class RecyclerView.Adapter(private val photos: ArrayList<Photo>):  
    RecyclerView.Adapter<RecyclerView.Adapter.PhotoHolder>
```

Тепер можемо реалізувати метод getItemCount(), який просто повертатиме розмір даного масиву. Надалі слід реалізувати паттерн ViewHolder у відповідному класі PhotoHolder (лістинг 5.8). Він передбачатиме наступні дії:

- створення класу, який буде успадковуватись від RecyclerView.ViewHolder, щоб дозволити адаптеру використовувати цей клас у якості ViewHolder-а, як показано в рядку 1 лістингу;

- додавання посилання на представлення you've inflated to allow the ViewHolder to access the ImageView and TextView as an extension property. Kotlin Android Extensions plugin adds hidden caching functions and fields to prevent the constant querying of views.
- Initialize the View.OnClickListener.
- Implement the required method for View.OnClickListener since ViewHolders are responsible for their own event handling.
- Add a key for easy reference to the item launching the RecyclerView.

Лістинг 5.8. Клас PhotoHolder

```
1 class PhotoHolder(v: View) : RecyclerView.ViewHolder(v),
2   View.OnClickListener {
3     private var view: View = v
4     private var photo: Photo? = null
5
6     init {
7         v.setOnClickListener(this)
8     }
9
10    override fun onClick(v: View) {
11        Log.d("RecyclerView", "CLICK!")
12    }
13
14    companion object {
15        private val PHOTO_KEY = "PHOTO"
16    }
17 }
```