АНГЕЛИН ЛАЛЕВ

МОБИЛНО ПРОГРАМИРАНЕ С JAVA

/ЛЕКЦИОНЕН КУРС/

2019

Мобилно програмиране с JAVA /лекционен курс/
Онлайн издание.

ISBN: …….

СЪДЪРЖАНИЕ НА ГЛАВА 4

ГЛАВА 4. ЕЛЕМЕНТИ НА ПОТРЕБИТЕЛСКИЯ ИНТЕРФЕЙС

1. Основни понятия

1.1. Изгледи и изгледни групи

Дискусията на елементите на потребителския интерфейс в Андроид е свързана с две основни понятия - *изглед (view)* и *изгледна група (view group)*.

Изгледите са визуалните елементи на потребителския интерфейс, които служат за директно взаимодействие с потребителя. Доста често те се наричат още контроли, елементи или „джаджи“ (widgets). Примери за такива елементи са полетата за въвеждане, бутоните, изображенията и т.н. Класовете, които представляват тези елементи в програмата, са наследници на класа *android.view.View*.

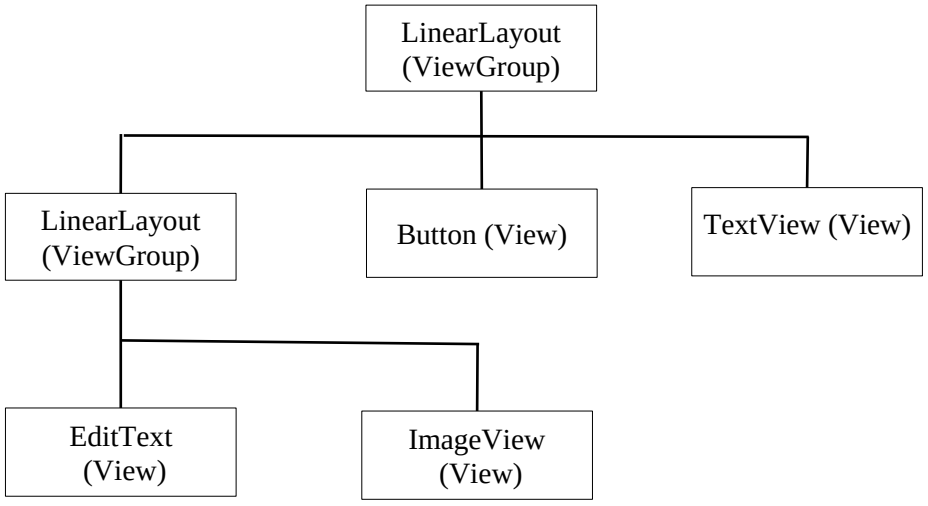
Изгледните групи са специални изгледи, които могат да съдържат в себе си други изгледи. Програмистите могат да мислят за тях като за своеобразни „контейнери“, в които могат да се поставят други визуални елементи. Тези контейнери могат да се поставят един в друг, като тяхната основна роля е да помогнат на процеса на оразмеряване и разполагане на екрана на останалите визуални компоненти.

Менютата и лентите с инструменти са идеален първи пример за изгледна група. Те съдържат в себе си други елементи (например бутони с инструменти), но се третират като един обект при разполагането им върху екрана.

Поради разнородните размери на екраните на мобилните устройства, приложенията за Андроид трябва да разполагат изгледите в потребителския интерфейс относително. Това означава, че не само бутоните и елементите от менютата, но и на практика *всеки* визуален елемент трябва да бъде поставен в изгледна група, за да бъде позициониран правилно на екрана.

Изгледните групи биват няколко вида. Класовете, които представляват тези видове групи в мобилното приложение, са наследници на *android.view.ViewGroup*. Този клас на свой ред е наследник на *android.view.View*, тъй като всеки вид изгледна група е и визуален елемент.

За да помества в прозореца си визуални елементи, всяка дейност на приложение в Андроид трябва да съдържа основна изгледна група. Всички визуални елементи и други изгледни групи се поставят в тази основна група, като по този начин се формира своеобразна йерархия на изгледите.



Фиг. 4.1. Примерна йерархия на изгледите в прозореца на дейност на приложение

Поради основната им роля за разполагане на другите визуални елементи в основния екран на приложението, повечето изгледни групи се наричат лейаути[1].

[1] Т.е. “разположения”, но ние няма да използваме тази дума. Ние предпочитаме „лейаути“ тъй като по този начин създаваме понятие с ясен смисъл, семантично различно от прилагателното „разположение“, което в книгата често се използва в друг контекст.

1.2. Дефиниране на изгледи и изгледни групи

Изгледите и изгледните групи в Андроид могат да се създадат по два начина – чрез зареждане на XML файл или чрез директно създаване в програмния код на приложението.

Най-често цялата йерархия от изгледи групи и изгледи се дефинира в XML файл чрез използване на специално дефинирана за целта схема. Ние ще наричаме такъв файл „лейаут файл“. Подобен файл съдържа подробно описание на всички визуални елементи в един прозорец. Долната разпечатка показва как изглежда такова описание.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout 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:id="@+id/testLayout"
6    android:layout_width="match_parent"
7    android:layout_height="match_parent"
8    android:orientation="vertical"
9    tools:context=".MainActivity">
10
11   <TextView
12     android:id="@+id/label"
13     android:layout_width="match_parent"
14     android:layout_height="wrap_content"
15     android:text="Enter e-mail!" />
16
17   <EditText
18     android:id="@+id/name"
19     android:layout_width="match_parent"
20     android:layout_height="wrap_content"
21     />
22   <Button
23     android:id="@+id/post"
24     android:layout_width="match_parent"
25     android:layout_height="wrap_content"
26     android:text="Post!"/>
27
   </LinearLayout>
```

Главният елемент на XML документа по горе описва изгледна група от тип *LinearLayout*. Неговите дъщерни елементи описват отделните изгледи (*TextView*, *EditText* и *Button*), които се съдържат в изгледната група.

XML елементите имат множество атрибути от именното пространство *android*. Така например атрибутът *android*:*id* се използва, за да се даде уникален идентификатор на всеки компонент. В примера този атрибут е използван на няколко места. Едно от тях например е на ред 12, където атрибутът е използван, за да се даде уникален идентификатор на изгледа от тип *TextView*. Този идентификатор се състои от служебна част *@+id* и уникално име *label*, разделени с наклонена черта. Служебната част указва на системата за компилация и построяване на проекта, че това е нов идентификатор, който трябва да бъде импортиран в служебните класове към приложението.

Процесът, при който програмата изчита лейаут файл и създава съответните екранни елементи, се нарича „надуване“ (inflation). Надуването е доста сложен процес, но за щастие, начинаещите програмисти не трябва да знаят всички детайли за него. Достатъчно е да знаят как такъв лейаут файл се зарежда в прозореца на дейността и как могат да получат референция към обектите, които се създават, за да представляват даден изглед по време на изпълнение на програмата.

Долният пример илюстрира *OnCreate* метод, който зарежда горния лейаут файл в прозореца на дейност, наречена *MainActivity*. В хода на този процес кодът на примера демонстрира и двата вида операции.

```
1  package uk.co.lalev.layouttest;
2  …
3  import android.widget.Button;
4  import android.widget.EditText;
5  import android.widget.TextView;
6  …
7
8  public class MainActivity extends AppCompatActivity {
9
10   @Override
11   protected void onCreate(Bundle savedInstanceState) {
12     super.onCreate(savedInstanceState);
13     setContentView(R.layout.activity_main);
14
15     LinearLayout l = (LinearLayout) findViewById(R.id.testLayout);
16     TextView label = (TextView) findViewById(R.id.label);
17     EditText name = (EditText) findViewById(R.id.email);
```

```
18      Button post = (Button) findViewById(R.id.post);
19
20      name.setHint("Enter email address");
21    }
22  }
```

На ред 13 в примера, програмата извиква метода *setContentView* с параметър *R.layout.activity_main*, което води до надуване на лейаут файла с име *activity_main.xml* и зареждането му в прозореца на текущата дейност (*MainActivity*).

R е изключително важен клас, който се създава при компилация на програмата. Системата за компилация и построяване на приложения поставя в този клас всички идентификатори на ресурси под формата на константи. Тези константи могат да се използват по време на изпълнение, за да се получи достъп до съответните ресурси.

XML лейаут файловете са именно вид ресурси, заедно с изображенията, таблиците със символни низове и други не-програмни елементи на приложението. Поради това техните идентификатори се намират в подклас *R.layout*. *R.layout.activity_main* е точно константата, съдържаща уникалния идентификатор на файла *activity_main.xml*.

След като *setContentView* зареди лейаут файла в прозореца на дейността, обектите, представляващи отделните изгледи, стават достъпни за програмата. За да получи референция към тях, програмистът трябва да използва метода *findViewById*.

findViewById връща обект, представляващ съответния изглед. Методът взема един параметър - уникалният идентификатор на изгледа, зададен в XML файла.

За разлика от лейаут фаловете, които автоматично получават уникален идентификатор на база името на файла, за да има даден визуален компонент

уникален идентификатор в класа *R*, той трябва на първо място да е дефиниран с разгледания по-горе атрибут *anroid:id*. Това е така, тъй като може да се окаже, че програмистът не желае да манипулира програмно даден компонент. Пример за такива компоненти, които рядко се модифицират по време на изпълнение на програмата, и следователно доста често не се нуждаят от уникални идентификатори, са текстовите етикети *TextView*. Те съдържат описателен текст, който остава непроменен през цялото време на изпълнение.

В нашия пример, програмистът е дефинирал *android:id* за всички елементи в XML файла, поради което съответните константи са налични и могат да се използват за получаване на референция към съответните обекти. На редове 15 до 18, програмистът е използвал *findViewById*, за да получи съответните референции.

Читателите трябва да забележат три важни момента, свързани с тези редове. На първо място, *LinearLayout* е изгледна група, докато *TextView*, *Button* и *EditText* са изгледи. Но тъй като изгледните групи са просто специални изгледи, *findViewById* може да се използва за получаване на референции и към двата вида обекти. Освен това, читателите трябва да забележат, че имената на константите в подкласа *R.id* се формират от стойността на атрибута *android:id*, като в името на константата не влиза служебната част *@+id*. Така например *EditText* елементът в XML файла имаше атрибут *android:id="@+id/email"* поради което неговият идентификатор е представен в програмата като *R.id.email*. И на последно място, читателите трябва да обърнат внимание, че самите стойности на константите, представляващи уникални идентификатори на ресурсите, всъщност са числа, но тези числа никога не се използват директно. Класът *R* служи именно за избягване на нуждата да се работи директно с числени идентификатори.

Последен детайл, който касае метода на създаване на визуалните елементи от XML файл, е това, че XML файловете могат да се редактират или ръчно, или чрез графичния редактор на Android Studio. Вторият начин обикновено е силно предпочитан от начинаещи програмисти.

Освен създаването на изгледи чрез надуване, в редки случаи програмите за Андроид създават визуалните елементи директно в програмния код. Долният пример демонстрира създаването на същата йерархия от изгледи като горе, но този път без използването на XML файл.

```
1  package uk.co.lalev.layouttest;
2  …
3  import android.widget.Button;
4  import android.widget.EditText;
5  import android.widget.LinearLayout;
6  import android.widget.TextView;
7  …
8  public class MainActivity extends AppCompatActivity {
9
10   @Override
11   protected void onCreate(Bundle savedInstanceState) {
12     super.onCreate(savedInstanceState);
13
14     LinearLayout testLayout = new LinearLayout(this);
15     testLayout.setOrientation(LinearLayout.VERTICAL);
16     TextView label = new TextView(this);
17     label.setText("Enter email!");
18     testLayout.addView(label);
19     EditText email = new EditText(this);
20     email.setHint("Enter email address");
21     testLayout.addView(email);
22     Button post = new Button(this);
23     post.setText("Post!");
24     testLayout.addView(post);
25
26     setContentView(testLayout);
27   }
28 }
```

2. Основни визуални елементи

2.1. TextView

TextView елементът показва обикновен текст, който се извежда на екрана на мобилното устройство. Доста често *TextView* елементи се използват като пояснителни „етикети“ към други визуални елементи. Основен атрибут на *TextView* елемента е атрибутът *android:text*, който съдържа текста, който се извежда на екрана на мобилното устройство. Долната таблица описва някои от основните атрибути на *TextView* елемента:

Атрибут	**Тип**	**Описание**
`android:text`	Текст	Текст на бутона

`android:drawable_left`
`android:drawable_right`
`android:drawable_top`
`android:drawable_bottom`
	Идентификатор на ресурс	Изображение – икона, която се разполага съответно от лявата, дясната, горната или долната страна на текста.

По време на изпълнението на програмата, програмистът може да манипулира свойствата, представлявани от тези атрибути със следните методи:

Метод	**Описание**
`void setText (CharSequence text)`	Задава текст на елемента
`CharSequence getText()`	Получава текста на елемента
`void setCompoundDrawablesRelativeWithIntrinsicBounds (int start, int top, int end, int bottom)`	Задава изображения, които да се покажат съответно в началото на текста, над него, в края на текста или под него. Текстът при повечето (но не всички) езици се разполага отляво надясно. За тези езици началото на текста е вляво, а краят на текста – вдясно. Одтелните параметри трябва да бъдат идентификатори на ресурси,
`void setCompoundDrawablesRelativeWithIntrinsicBounds (Drawable start, Drawable top, Drawable end, Drawable bottom)`	Аналогичен на предния, с тази разлика, че изображенията при този метод трябва да са обекти от клас *Drawable*.

Класът *TextView* е родител на всички визуални елементи, които извеждат текст, поради което методите и атрибутите на *TextView* за работа с текст могат да се използват непроменени при всички такива визуални елементи.

Следващият пример демонстрира програмното създаване на TextView елемент и вмъкването му в LinearLayout група.

2.2. Button

Бутоните (вж. фиг. 4.2) са традиционни визуални елементи, които се използват от потребителя за иницииране на някакво действие.

Фиг. 4.2. Бутони в Андроид

Класът *Button* е наследник на класа *TextView*, поради което методите му за задаване на текст и изображение са на практика идентични с тези на *TextView*. Бутоните реагират и на няколко събития, най-често използваното от което е *onClick*, което настъпва, когато потребител щракне върху бутона. Събитията са разгледани по-нататък в изложението.

2.3. EditText

EditText представлява многофункционално поле за редактиране на текст. *EditText* е наследник на *TextView*, поради което той има същите атрибути и методи, свързани със задаването и получаването на съдържанието на текста в полето.

EditText притежава и редица други атрибути, свързани с редактирането на текст:

Атрибут	**Тип**	**Описание**
`android:inputType`	Текст	Указание за типа на данните, които ще се въвеждат в полето.
`android:imeOptions`	Текст	Допълнителни указания за редактора на входния метод.

Стойността на *android:inputType* се интерпретира от метода за въвеждане (екранна клавиатура и т.н.) на мобилното устройство, за да представи най-подходящата клавиатура за въвждане в полето. Възможните стойности са целочислени константи - флагове, които в XML файла се задават чрез текстов низ. Текстовият низ позволява комбиниране на флагове с побитово *или*. Така например *android:inputType=”text | textAutoCorrect”* е комбинация от два флага. Флагът *text* (стойност 1) указва, че това е обикновено текстово поле, т.е. в него могат да се въвеждат всякакви символи. Флагът *textAutoCorrect* (стойност 8001) указва, че методът за въвеждане следва да предложи автоматична корекция на текста, въвеждан от потребителя.

Флаговете, които могат да бъдат посочени в *android:inputType,* са документирани в референтното ръководство за разработчици на Андроид[2]. По-важните от тях са:

date	Текстът в полето ще представлява дата.
dateTime	Текстът в полето ще представлява дата и време.
number	Полето ще съдържа само цифри
numberDecimal	Комбиниран с *number*, този флаг позволява въвеждането на десетичен разделител.
numberSigned	Комбиниран с *number*, този флаг позволява въвеждането на число със знак.
numberPassword	Въвежданата стойност ще представлява числова парола.
phone	Въвежданата стойност ще представлява телефон.

[2] Вж. https://developer.android.com/reference/android/widget/TextView.html#attr_android:inputType

text	Въвежданата стойност ще представлява произволен текстов низ.
textEmailAddress	Полето ще съдържа адрес за електронна поща.
textMultiLine	Полето ще съдържа многоредов текст.
textPassword	Полето ще съдържа произволен текстов низ, който ще се използва за парола.

Допълнителните указания за редактора на входния метод (IME) поволяват, наред с други възможности, замяна на бутона *Enter* на клавиатурата с бутон за действие. Тези указания също са под формата на флагове.

В долната таблица са показани някои от тези флагове:

actionNext	Бутонът за действие извършва операция “next”, което води до преминаване към следващото поле за въвеждане.
actionPrevious	Бутонът за действие извършва операция “previous”, което води до преминаване към предишното поле за въвеждане.
actionGo	Бутонът за действие извършва операция “go”. Типично тази операция се асоциира с въвеждане на URL адреси в Интернет и непосредственото им отваряне след като се въведат в полето.
actionSearch	Бутонът за действие извършва операция “search”. Типично тази операция се асоциира с търсене на елементи, въведени в полето.
actionPost	Бутонът за действие извършва операция „post”. Тази операция типично се асоциира с изпращане на съобщение (ел. поща и т.н.)

За да прихване натискането на бутона за действия на виртуалната клавиатура, програмата трябва да зададе обработчик на събитието *onEditorActionListener*. Примери за това са дадени по-нататък в главата.

2.4. ImageView

ImageView контролите се използват за показване на изображения в прозореца на дейността. Класът *ImageView* се наследява от всички други визуални компоненти, които показват основно изображения, поради което описаните в тази точка методи и атрибути се отнасят по същия начин и за тях.

Основните атрибути на *ImageView* са свързани именно с изображението, което трябва да бъде показано в контрола.

Атрибут	Тип	Описание
`app:srcCompat`	Идентификатор на ресурс	Векторно изображение – икона, която се разполага в бутона.

Важно е да се отбележи, че първите версии на Андроид използваха растерни изображения, които не се мащабират добре на различни по-големина екрани. Новите версии преминаха към векторни изображения. За запазване на съвместимостта, Андроид и до днес поддържа стария атрибут *app:src*, който обаче не поддържа векторна графика и следва да се избягва във всички нови програми.

2.3. ImageButton

ImageButton представлява бутон, който вместо текст съдържа изображение. *ImageButton* показва своето изображение центрирано в бутона, което не е възможно за постигане чрез добавяне на изображение към *Button* и премахване на текста.

Долният пример илюстрира най-елементарна употреба на *Intent* обект с действие.

```
1  …
2  import android.content.Intent;
3  import android.provider.AlarmClock;
4  …
5  //Важно! Кодът по-долу няма да работи, ако потребителят не е
6  // разрешил на приложението да манипулира времето и алармите.
7
8  public void onClick(View v) {
9    Intent i = new Intent();
10   i.setAction(AlarmClock.ACTION_SHOW_ALARMS);
11   if (i.resolveActivity(getPackageManager()) != null) {
12     startActivity(i);
13   }
14 }
15 …
```

Програмите са свободни да дефинират собствени текстови низове за действията, които извършват, но на практика най-често използват някои от стандартизираните действия, предлагани от Андроид. Операционната система дефинира голям брой текстови константи за стандартни действия, които се изпълняват от повечето програми. Така например константата *ACTION_PICK*, която е равна на текстовия низ *„android.intent.action.PICK“,* обозначава действие по избиране на един от множество елементи. По същия начин *ACTION_SHOW_ALARMS* на ред 10 в примера е текстова константа за

действие, което се обработва от приложението за часовник на системата, и води до

:

`Intent(String action)`
Създава *Intent* обект с посоченото действие *action*.
`Intent(String action, Uri uri)`
Създава *Intent* обект с посоченото действие *action* и URI идентификатор на данните *uri*.
`Intent(Context packageContext, Class<?> cls)`
Създава *Intent* обект за експлицитно намерение, който извиква дейност с име на класа *cls*.

В зависимост от ситуацията обаче, този *Intent* обект по принцип може да съдържа всякаква друга информация, дискутирана до момента, включително стойности, поставени с *putExtra*. Долната таблица представя методите, които се използват за получаване на данните в *Intent* обект.

`String getAction()`
Получава действието в дадено намерение.
`boolean hasCategory(String category)`
Проверява дали даденото намерение има категория *category*.
`Set<String> getCategories()`
Връща *Set* обект с категориите в намерението.
`Uri getData()`
Връща Uri обект, съдържащ URI идентификатор на данни.
`String getType()`
Връща текстов низ, представляващ MIME тип на данните.
`boolean hasExtra(String name)`
Проверява дали даденото намерение има допълнителни данни с етикет *name*.
`Bundle getExtras()`
Връща *Bundle* обект с допълнителните данни към намерението.
`byte getByteExtra(String name, byte defaultValue)` `int getIntExtra(String name, int defaultValue)`

`float getFloatExtra(String name, float defaultValue)`
`double getDoubleExtra(String name, double defaultValue)`
`boolean getBooleanExtra(String name, boolean defaultValue)`
`String getStringExtra(String name)`
`Bundle getBundleExtra(String name)`

Връща допълнителна стойност от указания тип и етикет *name*. Ако такава липсва, връща указаната стойност по подразбиране *defaultValue*.

`byte[] getByteArrayExtra(String name)`
`int[] getIntArrayExtra(String name)`
`float[] getFloatArrayExtra(String name)`
`double[] getDoubleArrayExtra(String name)`
`boolean[] getBooleanArrayExtra(String name)`
`String[] getStringArrayExtra(String name)`

Връща масив от указания тип, съхранен в намерението под даден етикет *name*.

Системата използва тези данни на редове 41-44, за да попълни нов *Intent* обект, който отново извиква приложението за контакти, този път с дейност, която редактира избрания по-рано контакт.

4. Създаване на филтри за намерения на дейност

Освен да извиква дейности от други приложения, доста често дадено приложение трябва да предоставя дейности за извикване от страна на други приложения. За целта приложението трябва да приема имплицитни намерения, подадени от операционната система.

Филтрите за намерения определят това какви намерения ще приеме дадено приложение. Тези филтри са XML елементи, които се поставят в манифеста на приложението.

Основният XML елемент, който описва филтър за намерения, е елементът *intent-filter*. Този елемент се поставя в *activity* елемента, описващ дадена дейност, като е напълно приемливо дадена дейност да няма такъв елемент (в случай, че не е предвидено тя да се извиква имплицитно) или

дейността да има повече от един такъв елемент (в случай че дейността може да се извика по няколко различни начина).

Долният пример показва филтър за намерения на хипотетична дейност, която може да отваря Youtube видеа.

```
<activity android:name=".YoutubePlayer">
  <intent-filter tools:ignore="AppLinkUrlError">
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    <data android:scheme="http"  android:host="www.youtube.com"
      android:pathPrefix="/watch" />
    <data android:scheme="https" android:host="www.youtube.com"
      android:pathPrefix="/watch" />
  </intent-filter>
</activity>
```

Когато операционната система обработва намерение и търси коя дейност да бъде стартирана, тя сравнява информацията, попълнена в *Intent* обекта с тази, декларирана в *intent-filter* елементите от манифеста на всяко едно инсталирано приложение. Този процес първо сравнява действията, след което сравнява категориите, след което сравнява данните и техния тип. Дейност, която премине успешно сравнението, се включва списък с кандидати. Ако този списък има само една дейност, тя се изпълнява директно. Ако се има повече кандидати, потребителят ще бъде запитан за това кой кандидат да бъде изпълнен, като има вариант операционната система да запомни избора му.

В *intent-filter* елемента програмистът следва да дефинира едно или повече действия. Когато операционната система сравнява намерение с филтъра, тя проверява дали попълненото в намерението действие отговаря на някое от дефинираните във филтъра. В противен случай операционната система ще счете, че подаденото намерение не отговаря на дадения филтър.

Дефинирането на действие става със създаването на нов *action* елемент. Този елемент има един задължителен атрибут - *android:name*, който е текстов

низ с името на действието. Попълнен *action* елемент може да бъде видян на ред 3 в предния пример.

Ако сравнението за действие премине успешно, операционната система ще сравни категориите в *Intent* обекта с категориите във филтъра за намерението. Ако всяка категория в *Intent* обекта фигурира и във филтъра, операционната система ще продължи със сравняването на данните. В противен случай тя ще счете, че текущото действие не отговаря на намерението.

Както бе споменато в предните точки, за да има съвпадение, не е задължително във филтъра да има само тези категории, които са посочени в *Intent* обекта, но е задължително всички категории от този обект да бъдат посочени и във филтъра. Всеки *Intent* обект по подразбиране се попълва с категория *android.intent.category.DEFAULT* при извикването на *StartActivity* и *StartActivityForResult,* поради което всеки филтър за намерение на дейност, която се извиква по този начин, също трябва да съдържа тази категория.

Задаването на нова категория във филтъра за намерения става с добавянето на елемент *category*. Този елемент има един задължителен атрибут - *android:name*, който се попълва с името на категорията. Попълнени *category* елементи могат да бъдат видени на редове 4 и 5 в горния пример

При съвпадение на категориите, операционната система продължава със сравнението на данните. Всеки филтър за намерения може да дефинира един или повече *data* елементи. Тези елементи могат да имат следните атрибути:

android:scheme	Символен низ, който показва схемата на URI адреса. Тази схема може да бъде „content“, „http“, „https”, “file” и т.н.
android:host	Това е име на хост, като в случаите, когато схемата не е http или https, тук може да се запише името на пакета на приложението.
android:port	Порт

android:path	Пълен път до ресурса
android:pathPrefix	Частичен път до ресурса
android:pathPattern	Шаблон за път до ресурса
android:mimeType	MIME тип

Важно е да се отбележи, че атрибутите на всички *data* елементи формират единен филтър. *data* елементите в двата примера по-долу са напълно еквивалентни по отношение филтрирането на намерения:

```
1 …
2 <data android:scheme="kitten" android:host="com.lalev.io" />
3 …
```

```
1 …
2 <data android:scheme="kitten" />
3 <data android:host="com.lalev.io" />
4 …
```

По същия начин следващите две комбинации от *data* елементи дават напълно еквивалентни резултати при филтриране на намерения.

```
1 …
2 <data android:scheme="kitten" android:host="com.lalev.io" />
3 <data android:scheme="octopus" android:host="com.lalev.iop" />
4 …
```

```
1 …
2 <data android:scheme="octopus" />
3 <data android:scheme="kitten" />
4 <data android:host="com.lalev.io" />
5 <data android:host="com.lalev.iop" />
6 …
```

Когато комбинациите от *data* елементи зададат две и повече стойности на един и същ атрибут, тези стойности се интерпретират като алтернативни възможности за съответната част на URL адреса.

data елементите от последния пример ще бъдат удовлетворени и от четирите URI адреса, посочени по-долу:

- kitten://com.lalev.io
- kitten://com.lalev.iop
- octopus://com.lalev.io

- octopus://com.lalev.iop

Важно е да се отбележи, че ако даден филтър не дефинира атрибут *android:scheme*, неговите *android:host* атрибути ще бъдат игнорирани от операционната система при сравнение на филтъра с намерения. По същия начин, ако даден филтър не дефинира *android:host*, то неговите *android:port*, *android:path*, *android:pathPrefix* и *android:pathPattern* атрибути ще бъдат игнорирани от операционната система.

Атрибутът *android:mimeType* е независим от останалите атрибути и често е единствен атрибут на *data* елементите в много филтри за намерения.

Долният пример дефинира филтър за намерения, който обработва URI адреси, указващи папки на специфичен FTP сървър.

```
1 …
2 <intent-filter>
3     <action android:name="android.intent.action.VIEW" />
4     <category android:name="android.intent.category.DEFAULT" />
5     <category android:name="android.intent.category.BROWSABLE" />
6     <data android:scheme="ftp" />
7     <data android:host="ftp.lalev.com" />
8     <data android:pathPattern="/.*/download/version.*/" />
9 </intent-filter>
10 …
```

В този пример може да се наблюдава и употребата на атрибута *pathPattern*, който позволява използването на уайлдкард шаблон за указване на пътища. Шаблонът **.*** (точка и звезда) указва, че на това място може да се намира произволен символен низ (включително и такъв с нулева дължина). Следните URI адреси удовлетворяват зададения в шаблона филтър за намерения:

- ftp://ftp.lalev.com/test/download/version1/
- ftp://ftp.lalev.com/production/download/version-demo/
- ftp://ftp.lalev.com/test/download/version/

Последният пример към тази точка демонстрира използването на *android:mimeType*. Приложеният по-долу филтър описва дейност, която може да показва съдържанието различни графични файлове.

```
…
<intent-filter>
  <action android:name="android.intent.action.VIEW" />
  <category android:name="android.intent.category.DEFAULT" />
  <data android:scheme="file" />
  <data android:mimeType="image/png" />
  <data android:mimeType="image/tiff" />
  <data android:mimeType="image/jpeg" />
</intent-filter>…
```

5. Примерен проект

Примерният проект към настоящата глава представлява елементарно приложение, което реализира фото-рамка. Приложението позволява на потребителя да избере определен брой снимки от външната памет на мобилното устройство, които да бъдат показвани една след друга на екрана.

По-долу са изложени по-важните файлове на проекта:

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.example.pictureframe">

  <application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity
      android:name=".PlayActivity"
      android:configChanges="orientation|keyboardHidden|screenSize"
      android:label="@string/title_activity_play"
      android:theme="@style/FullscreenTheme">
    </activity>
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
  <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
</manifest>
```

activity_main.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/ConstraintLayout"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".MainActivity" >

  <com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/fbPlay"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="16dp"
    android:clickable="true"
    android:onClick="OnPlayClick"
    app:layout_constraintBottom_toTopOf="@+id/fbAdd"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.88"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="1.0"
    app:srcCompat="@android:drawable/ic_media_play" />

  <com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/fbAdd"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:clickable="true"
    android:onClick="OnAddClick"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.88"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.93"
    app:srcCompat="@android:drawable/ic_input_add" />

  <ListView
    android:id="@+id/ImageList"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:layout_editor_absoluteX="0dp"
    tools:layout_editor_absoluteY="85dp">
  </ListView>
</androidx.constraintlayout.widget.ConstraintLayout>
```

MainActivity.java

```java
package com.example.pictureframe;

import androidx.appcompat.app.AppCompatActivity;

import android.content.Intent;
import android.os.Bundle;
import android.provider.MediaStore;
```

```
 8 import android.view.View;
 9 import android.widget.ArrayAdapter;
10 import android.widget.ListView;
11
12 import java.util.ArrayList;
13
14 public class MainActivity extends AppCompatActivity {
15
16   public final static int PICK_RESULT = 1;
17
18   public final static String EXTRA_IMAGES =
19       "com.example.pictureframe.EXTRA_IMAGES";
20
21   private ArrayList<String> pictures = new ArrayList<>();
22   private ArrayAdapter adapter;
23
24   @Override
25   protected void onCreate(Bundle savedInstanceState) {
26     super.onCreate(savedInstanceState);
27     setContentView(R.layout.activity_main);
28
29     ListView imageList = findViewById(R.id.ImageList);
30     adapter = new ArrayAdapter(this, android.R.layout.simple_list_item_1,
31         pictures);
32     imageList.setAdapter(adapter);
33   }
34
35   public void OnAddClick(View v) {
36     Intent i = new Intent();
37     i.setAction(Intent.ACTION_PICK);
38     i.setData(MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
39     if (i.resolveActivity(getPackageManager()) != null) {
40       startActivityForResult(i, PICK_RESULT);
41     }
42   }
43
44   public void OnPlayClick(View v) {
45     Intent i = new Intent();
46     i.setClass(this, PlayActivity.class);
47     String[] pics = pictures.toArray(new String[pictures.size()]);
48     i.putExtra(EXTRA_IMAGES, pics);
49     startActivity(i);
50   }
51
52   @Override
53   protected void onActivityResult(int requestCode, int resultCode, Intent data)
54 {
55     if ((requestCode == PICK_RESULT) && (resultCode == RESULT_OK)) {
56       adapter.add(data.getDataString());
57     }
58   }
   }
```

PlayActivity.java

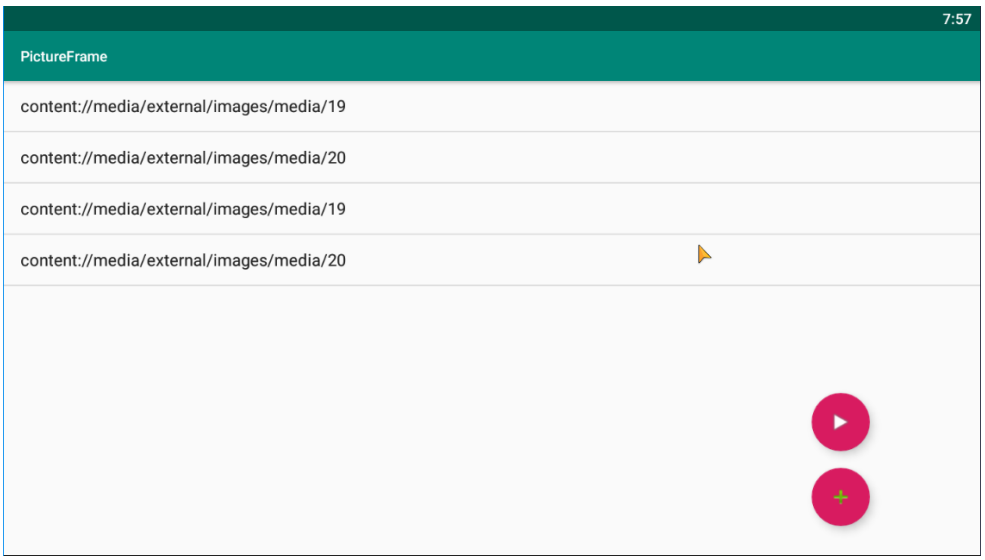
```
 1 package com.example.pictureframe;
 2
 3 import androidx.appcompat.app.ActionBar;
 4 import androidx.appcompat.app.AppCompatActivity;
```

```
 5 import androidx.constraintlayout.widget.ConstraintLayout;
 6 import androidx.core.content.ContextCompat;
 7
 8 import android.Manifest;
 9 import android.content.pm.PackageManager;
10 import android.net.Uri;
11 import android.os.Build;
12 import android.os.Bundle;
13 import android.view.View;
14 import android.widget.FrameLayout;
15 import android.widget.ImageView;
16
17 import java.util.Timer;
18 import java.util.TimerTask;
19
20 public class PlayActivity extends AppCompatActivity {
21
22   private final static int REQUEST_EXTERNAL_STORAGE = 1;
23
24   private String[] images;
25   private int imgCount;
26   private ImageView iView;
27
28   public void showNextPicture() {
29     if (++imgCount > images.length - 1) {
30       imgCount = 0;
31     }
32     iView.setImageURI(Uri.parse(images[imgCount]));
33   }
34
35   @Override
36   protected void onCreate(Bundle savedInstanceState) {
37     super.onCreate(savedInstanceState);
38
39     getWindow().getDecorView()
40         .setSystemUiVisibility(View.SYSTEM_UI_FLAG_FULLSCREEN |
41             View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
42         );
43
44     ActionBar actionBar = getSupportActionBar();
45     if (actionBar != null) {
46       actionBar.hide();
47     }
48
49     images = getIntent().getStringArrayExtra(MainActivity.EXTRA_IMAGES);
50
51     iView = new ImageView(this);
52     iView.setBackgroundColor(0xFF000000);
53     FrameLayout frame = new FrameLayout(this);
54     frame.addView(iView);
55     setContentView(frame);
56
57     if (ContextCompat.checkSelfPermission(this,
58         Manifest.permission.READ_EXTERNAL_STORAGE) !=
59         PackageManager.PERMISSION_GRANTED) {
60       if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
61         requestPermissions(
62             new String[]{Manifest.permission.READ_EXTERNAL_STORAGE},
```

```
63                REQUEST_EXTERNAL_STORAGE);
64          }
65        }
66        imgCount = 0;
67        iView.setImageURI(Uri.parse(images[imgCount]));
68
69        Timer t = new Timer();
70        t.schedule(new TimerTask() {
71          @Override
72          public void run() {
73            runOnUiThread(new Runnable() {
74              @Override
75              public void run() {
76                showNextPicture();
77              }
78            });
79          }
80        }, 5000, 5000);
81      }
82    }
```

Приложението се състои от две дейности – *MainActivity* и *PlayActivity*. *MainActivity* е главна дейност на приложението. Нейна отговорност е показването на списък с графични изображения, които по-късно ще бъдат показани от рамката (Вж. фиг. 5.2).

MainActivity зарежда съдържанието си от лейаут файл, който съдържа *ListView* контрол и два плаващи бутона (*FloatingActionButton*), организирани в *ConstraintLayout*. Плаващите бутони са наименувани съответно *fbAdd* и *fbPlay*.



Фиг. 5.2. *MainActivity* позволява на потребителя да управлява списък с изображения, които ще бъдат показани от рамката

Щракването върху *fbAdd* стартира метода *onAddClick* (редове 35-42 в *MainActivity.java*). Този метод попълва имплицитно намерение с действието ACTION_PICK и URI *MediaStore.Images.Media.EXTERNAL_CONTENT_URI*. Резолюцията на това намерение води до стартиране на дейност за избиране на изображение от галерията на мобилното устройство. Тази дейност връща резултат, поради което извикването ѝ става със *StartActivityForResult*. Както бе обяснено по-горе, тази функция изисква втори параметър освен *Intent* обект. Този втори параметър е уникален идентификатор, който да послужи за обработката на отговора, който ще бъде получен в *OnActivityResult*. В случая за уникален идентификатор служи константата *PICK_RESULT*, която е дефинирана на ред 16 в *MainActivity.java*.

При получаване на отговор, *OnActivityResult* изчита URI на данните в *Intent* обекта *data*. Този обект носи информация за отговора, който в случая е URI на избраното изображение. *OnActivityResult* използва *getDataString* за да получи това URI като символен низ. След това този низ се добавя към *ListView* контрола чрез неговия адаптер *adapter* (ред 55).

Използването на *ListView* с адаптер бе демонстрирано в предните глави. В случая приложението използва най-елементарния подход, като за адаптер е използван *ArrayAdapter*. *ArrayAdapter* позволява информацията от ListView контрола да се съхранява в масив или *ArrayList*. Приложението използва втория вариант, поради по-голямата му гъвкавост – *ArrayList* позволява добавянето на неограничен брой елементи. Инициализацията на *ListView* контрола се извършва в *OnCreate* метода на *MainActivity* (редове 25-33 в *MainActivity.java*).

След като потребителят е добавил определен брой изображения към списъка (които от съображения за краткост на примера се показват в *ListView* елемента само с тяхното URI), той следва да натисне бутона *fbPlay*.

fbPlay стартира втората дейност на приложението – *PlayActivity* – чрез експлицитно намерение. Интересният момент тук е това, че методът *onPlayClick* предава масив със символни низове като допълнителна информация. За целта на ред 18 е дефинирана нова EXTRA_ константа, която е уникална за нашето приложение.

На ред 47 в *MainActivity.java* е дефиниран нов масив *pics*, който се попълва с елементите от *ArrayList* обекта *pictures*. Методът *toArray* изисква подаването на масив с нужната големина, поради което такъв масив първо се създава с *new String[pictures.size()]*. *toArray* връща референция към попълнения масив, която се присвоява на *pics*. На следващия ред масивът се поставя в *Intent* обекта с помощта на *putExtra* метод.

Дейността *PlayActivity* е по-особена с това, че тя създава своя лейаут динамично. Тоест, вместо да се зарежда от ресурсен файл със *setContentView*, лейаутът се изгражда изцяло в *OnCreate* метода на дейността. Този метод извършва още дейности, свързани със скриването на контролните елементи на прозореца.

На ред 39 в *OnCreate* метода на *PlayActivity*, програмата използва *getWindow* за да получи *Window* обект, представляващ прозореца, в който се изпълнява дейността. Програмата незабавно извиква метода *getDecorView* на този обект, за да получи изглед, който представлява стандартната декорация на прозореца – т.е. системни ленти и пр.

Методът *setSystemUVisibility* на този изглед позволява задаването на различни флагове, които засягат кои части от прозореца ще бъдат видни. Подаването на флаг *View.SYSTEM_UI_FLAG_FULLSCREEN* и флаг

View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION води до скриването съответно на лентата за статуса и лентата за навигация.

На редове 44-47 от *OnCreate*, програмата скрива и лентата на дейността При това положение дейността има на разположение целя екран на мобилното устройство.

На ред 49 чрез *getExtra*, дейността *PlayActivity* получава изпратения от *MainActivity* масив с URI адреси на изображения, които предстои да бъдат показани на пълен екран. Нов момент тук е използването на метода *getIntent* на класа *Activity* за получаване на *Intent* обекта, с който е стартирана дадената дейност.

На следващите редове (51-55) *OnCreate* създава лейаута и визуалните елементи на дейността. На ред 49 се създава нов *ImageView* елемент, а на следващия ред същия елемент получава черен заден фон. Изборът на непрозрачен фон е важна стъпка, тъй като предотвратява наслагването на изображения, когато в *ImageView* евентуално се зареждат последователно изображения с различни размери.

На ред 53 се създава *FrameLayout* обект. Този обект представлява Frame Layout, който е предназначен за показване на един-единствен визуален елемент на цялата площ на лейаута, която в случая съвпада с площта на целия екран. На следващия ред *ImageView* елемента се вмъква в лейаута с помощта на метода *addView*, а на ред 55 *FrameLayout* обектът се задава като лейаут на дейността със *setContentView*.

Редове 57 до 65 са особено интересни и засягат темата за сигурността на Андроид приложенията, която ще бъде разгледана подробно в следващите глави. Докато получаването на URI на изображение, избрано от потребителя, не изисква специални привилегии, прочитането на съдържанието на файла, който се намира на това URI, изисква изрични разрешения.

Мястото за деклариране на факта, че програмата изисква такива разрешения, е *AndroidManifest.xml*. В примера, на ред 25 от този файл, чрез елемента *uses-permission* програмата декларира, че се нуждае от разрешението *android.permission.READ_EXTERNAL_STORAGE*. Това разрешение гарантира, че програмата ще може да използва URI адрес и да получи съдържанието на обекта, който се намира на този адрес.

Според това колко критични са исканите разрешения, операционната система реагира различно. Ако разрешенията са с малък потенциал за експлоатация от страна на злонамерени програми, тези разрешения се дават веднага от операционната система, без тя да се консултира с потребителя (такива бяха например разрешенията за задаване на аларма, използвани в предишните примери).

Ако обаче разрешенията позволяват изпълнението на опасни за сигурността операции, потребителят ще бъде запитан от операционната система дали иска да разреши на приложението съответния достъп. Случаят с *READ_EXTERNAL_STORAGE* е точно такъв, тъй като това разрешение дава на програмата право да чете потребителски файлове.

Във версиите на Android до 5.1 (API 22) включително, операционната система ще поиска от потребителя това и евентуални други разрешения още по време на инсталация на приложението, като исканите разрешения ще бъдат напълно определени от съдържанието на *uses-permission* елементите в манифеста. При отказ от страна на потребителя да даде дори едно от исканите разрешения, операционната система ще откаже да инсталира приложението.

Андроид 6.0 и следващите версии преминават към друг модел на искане на разрешения, при които приложението ще се инсталира без запитване до потребителя, но „опасните“ разрешения ще бъдат поискани непосредствено преди извършването на операциите, които се нуждаят от тях. Това означава, че

освен да ги дефинира в манифеста, програмата ще трябва да предвиди и програмен код, който да поиска разрешение от потребителя в подходящия момент.

В примера, този програмен код се намира на ред 58. На този ред разрешението се иска чрез метода *requestPermissions*, който взема като първи аргумент масив със символни низове, представляващи съответните разрешения. Като втори аргумент на метода се предава код за идентификация, който има същото предназначение като кода, използван при *StartActivityForResult*. Тъй като при даването на разрешения операционната система извиква специална колбак функция (при условие че тази функция е дефинирана от дейността), този код ще помогне при обработването на върнатия резултат. Нашият пример не използва подобна колбак функция, поради което подаденият код не се използва никъде.

Използването на *requestPermissions* крие проблеми, ако програмистът иска да стартира приложението и на системи, които са по-стари от Android 6.0. Тези системи нямат метод *requestPermissions*, тъй като, както беше споменато, потребителите одобряват разрешенията, поискани от програмата, още при инсталацията ѝ. За да се избегне тази ситуация, кодът на ред 58 е поставен в *if* оператор, който проверява дали версията на Android API е равна или по-голяма от 23 - Android 6.0. Точната процедура касае сравняването *Build.VERSION.SDK_INT*, която съдържа версията на Android API на текущото устройство, с константата *Build.VERSION_CODES.M,* която идва от Marshmallow и е равна на 23.

На редове 66 и 67 програмата подготвя фоторамката за работа, като инициализира брояча на фоторамката *imgCount* и зарежда първото изображение в *ImageView* контрола.

На ред 69 кодът инициализира стандартен Java таймер обект, който евентуално ще послужи за смяна на изображението на всеки 5 секунди. Задаването на код, който да се извиква от таймера, както и задаването на интервал, на който да става това, се задава с метода *schedule* на класа *Timer*. Този метод има няколко вариации:

`void schedule(TimerTask task, Date time)`
Задава за изпълнение задача, указана от *task*. Изпълнението е еднократно и се в извършва на дата и време, посочени от *time*.
`void schedule(TimerTask task, long delay)`
Задава за изпълнение задача, указана от *task*. Изпълнението е еднократно и се извършва след изчакване от приблизително *delay* милисекунди.
`void schedule(TimerTask task, Date firstTime, long period)`
Задава за изпълнение задача, указана от *task*. Изпълнението е многократно и започва в момент, посочен от *firstTime*. След това задачата се повтаря на период от *period* милисекунди.
`void schedule(TimerTask task, long delay, long period)`
Задава за изпълнение задача, указана от *task*. Изпълнението е многократно и започва след *delay* милисекунди, след което се повтаря на *period* милисекунди.

Всички вариации на метода приемат за първи параметър *TimerTask* обект, който описва задачата, която ще се изпълни. *TimerTask* е абстрактен клас с един абстрактен метод – методът *run*. Този метод съдържа именно кода, който ще бъде изпълнен от таймера при настъпване на определения момент.

На редове 70 до 80 в примера кодът създава и инстанцира анонимен клас, наследник на *TimerTask*, който реализира метод *run*. Този клас бива предаден като първи параметър на *schedule*. На ред 80 се намират и втория и третия параметър на метода, които задават съответно забавяне от 5000 милисекунди преди първото извикване на задачата на таймера и 5000 милисекунди между последващите извиквания.

Изключително важен факт, свързан с таймерите и потребителския интерфейс на Андроид, е това, че таймерите се изпълняват в отделна нишка, различна от тази на дейността, която е стартирала таймера. В същото време Андроид ограничава кои нишки могат да манипулират елементите на

графичния потребителски интерфейс. Само нишката, която е инициализирала тези елементи, може програмно да манипулира техните свойства.

За програмата в примера това представлява проблем, тъй като на практика означава, че кодът в метода *run* не може да манипулира потребителския интерфейс на дейността *PlayActivity* и следователно не може да смени текущата снимка със следващата в списъка.

За да помогне в подобна ситуация, класът *Activity* предлага метод *runOnUiThread*. Този метод изпълнява код в нишката, инициализирала дейността, и приема един параметър – клас, който реализира интерфейса *Runnable*. Този интерфейс има един метод, наречен *run*. На практика това е много подобно на ситуацията с *TimerTask* и дори, ако трябва да сме максимално прецизни, *TimerTask* е дефиниран като абстрактен клас, който също реализира *Runnable*.

На редове 73-77 програмата дефинира нов анонимен клас с нужните свойства, като предава този клас на метода *runUiOnThread*. Методът *run* на този нов клас извиква *showNextPicture* в нишката на графичния интерфейс. *showNextPicture* на свой ред съдържа код за зареждането на ново изображение в *ImageView* контрола на *PlayActivity*.

Така очертаното приложение е доста грубовато. Така например *PlayActivity* не предвижда никакъв начин за връщане към главната дейност освен натискането на бутона *Back* на мобилното устройство. По-сериозно приложение евентуално би могло да добави обработчик на *OnClick* събитието на *ImageView* контрола, така че при докосване, *PlayActivity* да се затвори и изпълнението на приложението да се върне към *MainActivity*.

По същия начин, *MainActivity* би могла да показва *thumbnail* изображения, подредени в *GridView*, а анонимните класове можеха да бъдат заменени от ламбда изрази. Тъй като настоящото приложение има за цел да

акцентува върху попълването и подаването на намерения, подобни разширения са изпуснати. Читателите могат да проверят своите знания и умения, разширявайки програмния код на примера в очертаните насоки.