АНГЕЛИН ЛАЛЕВ

МОБИЛНО ПРОГРАМИРАНЕ С JAVA

/ЛЕКЦИОНЕН КУРС/

2019

Мобилно програмиране с JAVA /лекционен курс/
Онлайн издание.

ISBN: …….

СЪДЪРЖАНИЕ НА ГЛАВА 3

ГЛАВА 5. НАМЕРЕНИЯ (INTENTS)

1. Основни концепции при работа с намерения

Намеренията представляват механизъм за предаване на съобщения между приложенията. Най-честият случай на употреба на намеренията, който гарантира, че те имат важно място във всеки курс за начинаещи, е стартирането на дейности от същото или външно приложение..

Намеренията са представени от класа *Intent*. Когато една дейност иска да подаде сигнал или да стартира друга дейност, тя създава нов обект от този клас и го подава на операционната система за изпълнение, използвайки предназначени за целта методи на класа *Activity*.

Intent обектите могат да бъдат попълвани с няколко вида информация. Те са:

а/ име на компонент, който да бъде стартиран;

б/ действие;

в/ категория/категории;

г/ местоположение на данните

д/ тип на данните;

е/ допълнителна информация;

Когато *Intent* обектите подават точното име на компонент, който да бъде стартиран, подаването на действие, категории, местоположение на данните и тип на данните обикновено не е необходимо. Такива намерения се наричат *експлицитни* и обикновено се използват за стартиране на дейности от същото приложение.

Тъй като различните потребители използват различен софтуер, извикването на конкретен компонент от конкретно *външно* приложение, за да се изпълни дадена задача, не е особено разумна идея. Така например

извикването на конкретен клиент за електронна поща, за да се изпрати съобщение, може и да не сработи. Дори да приемем, че става въпрос за Gmail, който е инсталиран на повечето Android системи, може да се окаже, че потребителят не е конфигурирал Gmail, а използва друго приложение.

Поради тази причина, когато се извиква външно приложение обикновено не се подава конкретен компонент, а се описва действието, което ще се извърши. На база това описание, което се съдържа в полетата действие, категории, местоположение и тип на данните, операционната система и потребителят ще решат коя дейност или услуга[1] ще бъдат изпълнени за осъществяване на намерението.

Намерения, които подават комбинация от горните 4 вида информация, но не и името на конкретен компонент, се наричат *имплицитни*. Имплицитните намерения преминават през процес на *резолюция* при който операционната система сравнява описанието на извикваната дейност в намерението с описанията на дейностите, предлагани от различни приложения на системата. При наличие на съвпадение, намерението бива изпълнено. В хода на резолюцията може да се окаже, че дадено имплицитно намерение може да бъде изпълнено от повече от една дейност при което потребителят ще бъде запитан да избере кое действие да бъде стартирано.

[1] Услугите са компоненти на приложенията, които работят на заден фон и нямат прозорци и графичен потребителски интерфейс. Услугите ще бъдат разгледани в следващите глави.

2. Попълване на *Intent* обекти

2.1. Действия

Действието е текстов низ. Това поле най-често се попълва при имплицитни намерения, които описват какво трябва да се свърши, но оставят операционната система и потребителя да изберат коя дейност или услуга да се стартира, за да се осъществи заявеното действие.

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 стартиране на дейност, която показва всички аларми.

Горният пример е част от приложение, което реагира на потребителски бутон, като стартира действието за преглед на алармите в системата.

На ред 9 потребителят създава нов *Intent* обект, а на ред 10 той задава действието с метода *setAction*. Константата за действието на ред 10 е дефинирана в пакета *android.provider.AlarmClock,* поради което той е импортиран на ред 3 в разпечатката.

Подаваното намерение е имплицитно, тъй като не указва конкретен клас на дейност, която да се изпълни. Когато се подава имплицитно намерение, може да се окаже, че на системата няма инсталирано приложение с дейност, която да изпълни намерението. Приложенията трябва да проверяват за възникването на тази ситуация, тъй като опит за изпълнение на намерение, което не може да бъде изпълнено, ще доведе до забиване на приложението, което е подало намерението. В примера, тази проверка се извършва на ред 11 чрез метода *resolveActivity* на класа *Intent*. Този метод взема за параметър обекта, представляващ пакетния мениджър, който на свой ред е достъпен чрез метода *GetPackageManager*, който текущият клас наследява от класа *Activity*. Ако *resolveActivity* върне *null*, системата не може да изпълни намерението и програмата би следвало да се откаже от опитите за изпълнението му.

На ред 12, в случай на успешна *резолюция*, програмата предава намерението на операционната система за изпълнение чрез метода *startActivity*.

Много важен детайл, който трябва да се спомене, е това, че понякога системата и потребителите ограничават приложенията по отношение на това какви намерения могат да осъществяват. Програма като горната, която възнамерява да манипулира алармите на системата, трябва да заяви това в манифеста на приложението и да мине през процес на одобрение, при който потребителят евентуално ще ѝ разреши да изпълнява подобни действия. За

целта в манифеста трябва да бъде поставен елемент *uses-permission*, както е показано на по-долния пример:

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

  …

  <application
      …
  </application>

  …

  <uses-permission android:name="com.android.alarm.permission.SET_ALARM" />

</manifest>
```

Друг важен детайл, който трябва да се спомене, е това, че ако дадена потребителска програма реши да дефинира собствени действия, тя трябва да ги префиксира с името на собствения си пакет, което ще гарантира уникалност на името, избрано за даденото действие. Така например действието ACTION_CURE_CANCER би следвало да се дефинира като текстова константа “*com.wonderworks.medical.CURE_CANCER*”, ако то е част от пакета *com.wonderworks.medical*.

Избраният по-горе пример не е особено представителен за процеса на подаване на намерения, тъй като много малко дейности могат да бъдат открити от операционната система само по името на действието. На практика доста често, наред със самото действие, трябва да се подаде доста сложна комбинация категория и/или местоположение и тип на данните.

Също така много малко дейности не изискват подаване на допълнителна информация от рода на текст на изпращаното съобщение, съдържание на създаваната бележка за напомняне и т.н.

В примера по-долу е демонстрирано извикване на намерение с подаване на допълнителни данни.

```
…
```

```
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_SET_ALARM)
11     .putExtra(AlarmClock.EXTRA_MESSAGE, "MyCustomAlarm")
12     .putExtra(AlarmClock.EXTRA_HOUR, 23)
13     .putExtra(AlarmClock.EXTRA_MINUTES, 10);
14   if (i.resolveActivity(getPackageManager()) != null) {
15     startActivity(i);
16   }
17 }
18 …
```

Този път се добавя нова аларма - действие, което изисква да се укаже допълнителна информация. Тази информация се указва чрез *putExtra* методи. Андроид дефинира такива методи за всички елементарни типове от данни. Тези методи вземат като първи параметър текст – идентификатор на допълнителната стойност, която ще бъде предавана, а на второ място се предава самата стойност.

В горния пример, първото извикване на *putExtra* метод записва в намерението двойка идентификатор – стойност, като текстът на идентификатора е под формата на предоставена от Андроид константа – EXTRA_MESSAGE. Следващите две извиквания записват цели числа, които, както подсказват идентификаторите, представляват съответно час и минута.

Допълнителната информация към различните действия може да бъде най-разнообразна. Важно е да се направи разграничението, че за разлика от информацията, която се съдържа в полетата за категория, тип на данните и местоположение на данните, допълнителна информация не се използва за намиране на дейност, която да бъде изпълнена, а по-скоро съдържа информация, която е нужна на дейността, за да приключи успешно, веднъж щом операционната система я намери и стартира.

2.2. Категории

Задаване на категории вероятно е най-объркващият детайл, който притеснява начинаещите програмисти при попълване на намерения. Категориите са просто логически групи, към които може да се причисли всяко действие. Объркването идва от това, всяка дейност, която трябва да бъде извиквана от друго приложение, трябва да обяви в своя филтър за намерения (тези филтри ще бъдат разгледани по-нататък) една или повече категории. Като минимум, подобна дейност трябва да обяви категорията "*android.intent.category.DEFAULT*", която е плейсхолдър, използван да укаже, че действието няма специална категория.

Когато дадено намерение извиква дадена дейност, не е необходимо то да посочва, че извикваното действие трябва да има категорията „*android.intent.category.DEFAULT“*. Това е ясно по подразбиране. Ако обаче към действието в намерението се добавят една или повече други категории, те трябва да отговарят на категориите, посочени във филтъра за намерения на извикваната дейност, за да бъде стартирана тя от операционната система. Сравнението е еднопосочно, т.е. ако намерението посочи две конкретни категории, тези категории трябва да бъдат във филтъра за намерения на извикваната дейност, като няма значение дали в този филтър освен тях има посочени и други категории.

Обикновено категории, различни от *android.intent.category.DEFAULT*, се поставят в намеренията тогава, когато извикваната дейност не работи пряко с някакви данни и файлове. В такъв случай полетата за местоположение и тип на данните в намерението ще останат празни и операционната система няма да може да определи по тях коя дейност да стартира.

Долният пример показва как се стартират главната дейност на приложението карти и главната дейност на клиента за ел. поща. И двете дейности не работят директно с данни.

```
1  …
2  import android.content.Intent;
3  import android.provider.AlarmClock;
4  …
5  public void onClick(View v) {
6    Intent i = new Intent();
7    i.setAction(Intent.ACTION_MAIN)
8      .addCategory(Intent.CATEGORY_APP_EMAIL);
9    if (i.resolveActivity(getPackageManager()) != null) {
10      startActivity(i);
11   }
12 }
13
14 public void onClick1(View v) {
15   Intent i = new Intent();
16   i.setAction(Intent.ACTION_MAIN)
17     .addCategory(Intent.CATEGORY_APP_MAPS);
18   if (i.resolveActivity(getPackageManager()) != null) {
19     startActivity(i);
20   }
21 }
22 …
```

При извикване на *startActivity* е възможно да се окаже, че на системата са открити повече дейности с описаните характеристики. В такъв случай системата ще изведе запитване към потребителя за това кое приложение да бъде стартирано. Възможно е също потребителят да е съхранил предпочитанията си при предишно запитване. В такъв случай системата направо ще изпълни предпочитаното приложение.

2.3. Местоположение на данните

Полето за местоположение представлява URI идентификатор, който показва местоположението на даден ресурс, който програмата иска да манипулира (т.е. покаже, изтрие, редактира и пр.). URI идентификаторите са

генерализация на URL адресите в уеб, поради което всеки http адрес в уеб е валиден URI. За разлика от URL, URI адресите могат да сочат както към ресурси, които се намират в уеб, така и към ресурси, които са съхранени в мобилното устройство.

Много често URI идентификаторът към едно действие е достатъчен, за да бъде определена дейността, която ще го обработи. Така например URI *http://www.google.com* ще предизвика отваряне на дейност, способна да браузва Интернет. URI от рода на *content://com.android.contacts/contacts/lookup/…* ще сочи към елемент от списъка с контактите на мобилното устройство и ще бъде обработен от програмата за управление на контакти на Андроид. По същия начин URI *file:///mnt/sdcard/myPicture.jpg* ще сочи към файл, разположен на мобилното устройство, който ще бъде обработен от подходяща дейност.

По долните примери демонстрират попълването на полето за местоположение на данните, чрез метода *setData*:

```
1  …
2  import android.content.Intent;
3  import android.net.Uri;
4  …
5    public void onClick(View v) {
6      Intent i = new Intent();
7      i.setAction(Intent.ACTION_VIEW)
8        .setData(Uri.parse("https://www.google.com"));
9      if (i.resolveActivity(getPackageManager())!=null) {
10        startActivity(i);
11     }
12   }
```

```
1  …
2  import android.content.Intent;
3  import android.net.Uri;
4  …
5    public void onClick(View v) {
6      Intent i = new Intent();
7      i.setAction(Intent.ACTION_VIEW)
8        .setData(Uri.parse("geo:43.620987,25.345067"));
9      if (i.resolveActivity(getPackageManager())!=null) {
10       startActivity(i);
11     }
12   }
```

Горните два примера показват съответно стартирането на дейности за браузване на Интернет и отваряне на карти. За разлика от по-ранния пример, който демонстрира само стартиране на приложение – браузър и приложение за карти, настоящият пример стартира браузър с конкретен адрес за отваряне. По същия начин, приложението за карти се стартира с дейността, която показва конкретно местоположение.

От примера е видно, че името „данни“ или „местоположение на данните“ вероятно не е съвсем точно. Данните могат да бъдат само URI идентификатор и всеки възможен URI идентификатор може да бъде поставен в полето данни, стига да има инсталирана дейност, в чийто манифест е описано, че обработва дадената URI схема на адресиране (*https*, *geo*, *file*, *content* и т.н).

Работата с URI адреси в Android е опосредствана от класа Uri. Този клас притежава статични методи[2] като метода *parse*, които създават Uri обекти на база текстова информация. *parse* просто взема текстов низ, представляващ валиден URI адрес, и на негова база инстанцира обект, който да представлява този URI в системата.

2.4. Тип на данните

Много действия, като например създаването на нов документ, не могат да посочат URI, тъй като документът/обектът тепърва предстои да бъде създаден от извикваната дейност. Такива действия обикновено посочват тип на документа. Това става чрез подаване на текстов низ, който представлява MIME тип.

MIME типът уникално идентифицира вида на документа. Такива типове например могат да бъдат например *application/pdf* (PDF документ), или

[2] За читателите, запознати с шаблоните за дизайн, точното име е метод-фабрика (factory method).

audio/mpeg (MP3 файл). Разширявайки MIME стандарта, който се използва и в уеб технологиите, Android регистрира MIME типове и за всеки елемент, който не е файл, но може да се създава, изтрива и т.н. Така например типът *vnd.android.cursor.item/contact* указва елемент от списъка на контактите.

Долният пример демонстрира създаването на нов контакт в адресната книга на устройството:

```
1  …
2  package com.example.datatest;
3  …
4  import android.content.Intent;
5  import android.provider.ContactsContract;
6  …
7    public void onClick(View v) {
8      Intent i = new Intent();
9      i.setAction(Intent.ACTION_INSERT)
10       .setType(ContactsContract.Contacts.CONTENT_TYPE);
11     if (i.resolveActivity(getPackageManager())!=null) {
12       startActivity(i);
13     }
14   }
```

Както е видно от примера, типът на данните отново се подава чрез предварително дефинирана константа - *CONTENT_TYPE*, която е равна на текста *vnd.android.cursor.dir/contact*.

2.5. Конкретен клас

Както бе споменато по-горе, подаването на точен клас определя точната дейност, която трябва да бъде изпълнена. Това прави намерението експлицитно. По-долният пример демонстрира как се подава такова намерение за изпълнение на дейност от същото приложение:

```
1  …
2  import android.content.Intent;
3  …
4
5    public void onClick1(View v) {
6      Intent i = new Intent();
7      i.setClassName(getPackageName(), NewActivity.class.getName());
8      startActivity(i);
```

```
 9   }
10
11   public void onClick2(View v) {
12     Intent i = new Intent();
13     i.setClass(this, NewActivity.class);
14     startActivity(i);
15   }
16
17   public void onClick3(View v) {
18     Intent i = new Intent();
19     i.setClass(getApplicationContext(), NewActivity.class);
20     startActivity(i);
21   }
```

Трите метода в горния пример илюстрират алтернативни начини за подаване на клас. Методът *setClassName* (редове 7-8) приема за параметри два символни низа. Първият от тях е име на пакета на приложението. Вторият от тях е име на клас на дейността, която ще бъде стартирана. Системата слепя двете имена за да формира напълно квалифицираното име на клас за дейността, която ще бъде извикана.

Името на пакета на приложението се получава чрез извикване на метода *getPackageName* на *Activity* обекта, който представлява текущата дейност. Този метод е дефиниран в абстрактния клас *Context*, който се наследява от класовете *Activity*, *Service* и *Application*. Това означава, че същият метод може да се извика и от контекста на приложението, при условие че този контекст е получен по-рано с помощта *getApplicationContext*. Това уточнение има значение за следващите два метода, които демонстрират по-популярен и чист начин за задаване на дейност от текущото приложение.

Вместо *setClassName*, на ред 14 в *onClick1* е използван методът *setClass*. Той също взема два параметъра. Първият е *Context* обект, докато вторият е класовият литерал на търсения клас.

Класовият литерал е просто комбинация от името на класа и текста *.class*. Той е общ начин в Java да се получи или подаде информация за даден клас, която включва и името на този клас.

Context обектът се използва само за да се получи името на пакета на приложението и всъщност на мястото на този параметър може да се подаде или контекста на приложението, или референция към текущата дейност или услуга, която инициира намерението. И двата класа дефинират нужния метод *getPackageName*, дискутиран по-горе.

OnClick2 използва *this* (референция към Activity класа на текущата дейност), докато *OnClick3* използва *GetApplicationContext*() за да получи контекст обект, който да подаде на *setClass*. Резултатът и в двата случая е напълно еднакъв.

Последни детайли, които трябва да бъдат споменати по отношение на попълването на намерения, касаят това, че класът *Intent* предлага множество конструктори, които комбинират създаването на *Intent* обект с попълването му.

Долната таблица показва някои от тези конструктори:

`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*.

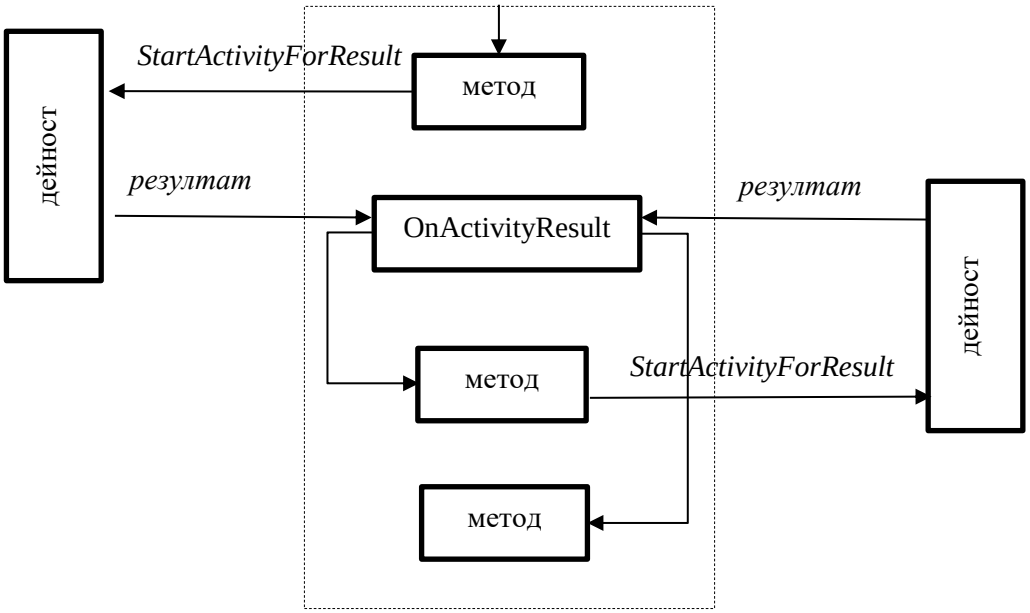
Информация как се попълват най-често задаваните намерения може да се намери в ръководството за програмиране на Android, предоставяно от Google[3].

[3] Вж. https://developer.android.com/guide/components/intents-common

3. Получаване на резултат при извикване на дейност

Много често дейността, която подава намерение, очаква отговор от извикваната дейност. Този отговор може да е код за грешка или пък някакви данни. Когато е необходимо да бъде получен такъв отговор, трябва да се използва методът *startActivityForResult*. Използването на този метод вместо *startActivity*, ще доведе до това, че когато извикваната дейност приключи, операционната система ще извика *OnActivityResult* на извикващата дейност и ще предаде резултат.

Независимо колко дейности се стартират, операционната система винаги ще извиква *OnActivityResult*. Тоест програмата може да стартира по ред Google карти, друга дейност от същото приложение и например, клиент за електронна поща. Независимо коя от извиканите дейности приключва, извикваният метод, който ще получи резултата, ще е само един – *OnActivityResult* (Вж. фиг. 5.1).



Фиг. 5.1. *OnActivityResult* е обща входна точка за получаване на резултатите от всички дейности

За да работи този подход, всяка дейност, която се стартира трябва да има собствен уникален идентификатор. Този уникален идентификатор се подава на *StartActivityForResult* и се получава в *OnActivityResult*, за да укаже точно коя дейност е приключила и връща резултат.

Следващият пример демонстрира как работи това:

```
1  …
2  import android.content.Intent;
3  import android.net.Uri;
4  import android.provider.ContactsContract;
5  import android.util.Log;
6  …
7  public class MainActivity extends AppCompatActivity {
8  …
9    private static final int RESLT_CONTACTS_PICK=1;
10 …
11   public void OnClick(View v) {
12     Intent i = new Intent(Intent.ACTION_PICK);
13     i.setType(ContactsContract.Contacts.CONTENT_TYPE);
14     if (i.resolveActivity(getPackageManager())!=null) {
15       startActivityForResult(i, RESLT_CONTACTS_PICK);
16     } else {
17       Log.d("OnClick", "Cannot resolve activity!");
18     }
19   }
20
21   @Override
22   protected void onActivityResult(int requestCode, int resultCode, Intent data)
23 {
34
35     super.onActivityResult(resultCode, requestCode, data);
36
37     switch (requestCode) {
38       case RESLT_CONTACTS_PICK:
39         // При сериозни програми е добра практика
40         // обработката да се изнесе в отделен метод.
41         if (resultCode==RESULT_OK) {
42           Uri item = data.getData();
43           Intent i = new Intent();
44           i.setAction(Intent.ACTION_EDIT);
45           i.setData(item);
46           if (i.resolveActivity(getPackageManager()) != null) {
47             startActivity(i);
48           } else {
49             Log.d("onActivityResult", "Cannot resolve activity!");
50           }
51         }
52         break;
53     }
54   }
   …
```

На редове 12 и 13 от горната разпечатка програмата създава *Intent* обект с действие *ACTION_PICK* и тип *ContactsContract.Contacts.CONTENT_TYPE*. Тази комбинация води до отваряне на приложението с контакти. Това става на ред 15. където *startActivityForResult* е извикана с двата си параметъра – *Intent* обект и уникален код. В случая уникалният код е единица и е дефиниран на ред 9 в константата *RESLT_CONTACTS_PICK*.

Когато потребителят избере контакт, стартираната дейност от приложението за контакти приключва успешно и операционната система извиква *onActivityResult*. Първият с три параметъра – *requestCode* – е подаденият по-рано уникален код. Вторият и третият параметър са съответно код за грешка и *Intent* обект, който носи резултатните данни. В случая резултатните данни са URI на избрания от потребителя контакт.

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.

```
…
<intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    <data android:scheme="ftp" />
    <data android:host="ftp.lalev.com" />
    <data android:pathPattern="/.*/download/version.*/" />
</intent-filter>
…
```

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.

```
1 …
2 <intent-filter>
3   <action android:name="android.intent.action.VIEW" />
4   <category android:name="android.intent.category.DEFAULT" />
5   <data android:scheme="file" />
6   <data android:mimeType="image/png" />
7   <data android:mimeType="image/tiff" />
8   <data android:mimeType="image/jpeg" />
9 </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

```
 1 <?xml version="1.0" encoding="utf-8"?>
 2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
 3   package="com.example.pictureframe">
 4
 5   <application
 6     android:allowBackup="true"
 7     android:icon="@mipmap/ic_launcher"
 8     android:label="@string/app_name"
 9     android:roundIcon="@mipmap/ic_launcher_round"
10     android:supportsRtl="true"
11     android:theme="@style/AppTheme">
12     <activity
13       android:name=".PlayActivity"
14
```

```
15        android:configChanges="orientation|keyboardHidden|screenSize"
16        android:label="@string/title_activity_play"
17        android:theme="@style/FullscreenTheme">
18      </activity>
19      <activity android:name=".MainActivity">
20        <intent-filter>
21          <action android:name="android.intent.action.MAIN" />
22          <category android:name="android.intent.category.LAUNCHER" />
23        </intent-filter>
24      </activity>
25    </application>
26    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
  </manifest>
```

activity_main.xml

```
 1  <?xml version="1.0" encoding="utf-8"?>
 2  <androidx.constraintlayout.widget.ConstraintLayout
 3    xmlns:android="http://schemas.android.com/apk/res/android"
 4    xmlns:app="http://schemas.android.com/apk/res-auto"
 5    xmlns:tools="http://schemas.android.com/tools"
 6    android:id="@+id/ConstraintLayout"
 7    android:layout_width="match_parent"
 8    android:layout_height="match_parent"
 9    tools:context=".MainActivity" >
10
11    <com.google.android.material.floatingactionbutton.FloatingActionButton
12      android:id="@+id/fbPlay"
13      android:layout_width="wrap_content"
14      android:layout_height="wrap_content"
15      android:layout_marginBottom="16dp"
16      android:clickable="true"
17      android:onClick="OnPlayClick"
18      app:layout_constraintBottom_toTopOf="@+id/fbAdd"
19      app:layout_constraintEnd_toEndOf="parent"
20      app:layout_constraintHorizontal_bias="0.88"
21      app:layout_constraintStart_toStartOf="parent"
22      app:layout_constraintTop_toTopOf="parent"
23      app:layout_constraintVertical_bias="1.0"
24      app:srcCompat="@android:drawable/ic_media_play" />
25
26    <com.google.android.material.floatingactionbutton.FloatingActionButton
27      android:id="@+id/fbAdd"
28      android:layout_width="wrap_content"
29      android:layout_height="wrap_content"
30      android:clickable="true"
31      android:onClick="OnAddClick"
32      app:layout_constraintBottom_toBottomOf="parent"
33      app:layout_constraintEnd_toEndOf="parent"
34      app:layout_constraintHorizontal_bias="0.88"
35      app:layout_constraintStart_toStartOf="parent"
36      app:layout_constraintTop_toTopOf="parent"
37      app:layout_constraintVertical_bias="0.93"
38      app:srcCompat="@android:drawable/ic_input_add" />
39
40    <ListView
41      android:id="@+id/ImageList"
42      android:layout_width="match_parent"
```

```
43      android:layout_height="match_parent"
44      tools:layout_editor_absoluteX="0dp"
45      tools:layout_editor_absoluteY="85dp">
46    </ListView>
47  </androidx.constraintlayout.widget.ConstraintLayout>
```

MainActivity.java

```
 1  package com.example.pictureframe;
 2
 3  import androidx.appcompat.app.AppCompatActivity;
 4
 5  import android.content.Intent;
 6  import android.os.Bundle;
 7  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    }
    }
```

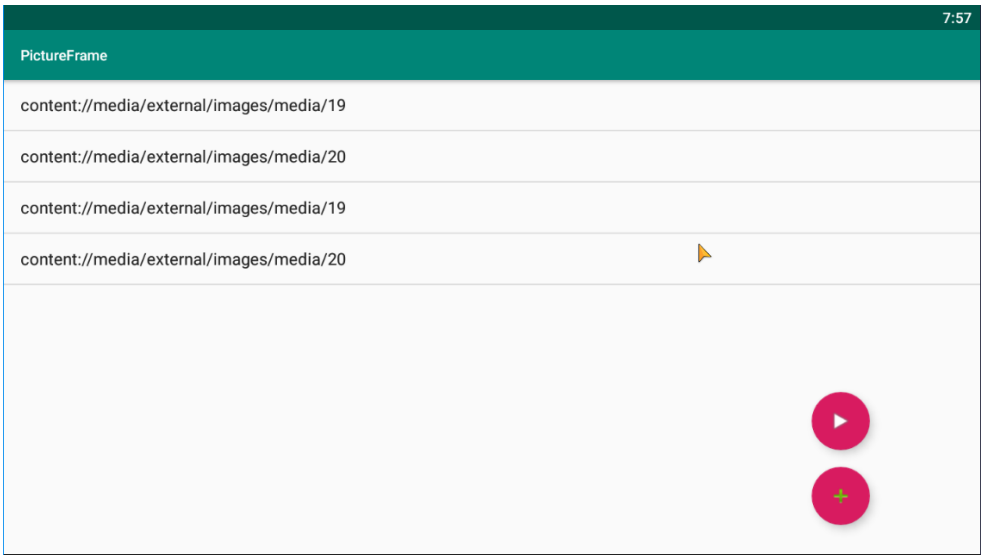
PlayActivity.java

```
 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.