Намерения

`Intent` (*англ.* намерение) — это объект для обмена сообщениями, который можно использовать для запроса действия от других компонентов приложения или системы. Существует три основных варианта их использования:

- запуск активности
- запуск службы, которая будет выполняться в фоновом режиме
- проведение широковещательной рассылки

В этой теме будет разбираться первый вариант использования, другие варианты будут описаны в других темах курса.

Есть два типа намерений:

- **Явные намерения** указывают, какое конкретно приложение или активность требуется запустить, путем предоставления либо имени пакета приложения, либо полного имени класса компонента. Как правило, явные намерения используются для запуска компонента в своем приложении, когда точно известно имя класса активности или службы.
- **Неявные намерения** не указывают конкретный компонент, а вместо этого объявляют общее действие, которое необходимо выполнить. Это позволяет компоненту из другого приложения его обработать: например, открыть страницу в браузере или отобразить местоположение на карте.

В случае неявного намерения система находит подходящий компонент, сравнивая содержимое намерения с *фильтрами намерений*, заявленными в манифесте других приложений на устройстве. Если подходящий компонент только один, то система его запускает и доставляет ему объект `Intent`. Если нашлось несколько подходящих компонентов, то система отобразит диалоговое окно, в котором пользователь может выбрать желаемое приложение.

Фильтр намерений — это запись в манифесте приложения, которая определяет какие намерения приложение хотело бы получать и обрабатывать. Это позволяет другим приложениям запускать компонент вашего приложения с помощью намерения определенного типа. И наоборот, если в манифесте не будет заявлены фильтры намерений, то запуск будет возможен только с явным намерением.

Создание намерения

Объект `Intent` должен содержать информацию, которая необходима система для определения подходящего компонента, и дополнительно может содержать информацию для компонента получателя:

- **Имя запускаемого компонента:** необязательный компонент, если он указан, то намерение становится явным. Как правило, имя компонента указывается в конструкторе `Intent`, однако можно использовать и такие методы как `setComponent()`, `setClassName()` или `setClass()`.

- **Действие:** строка, определяющая какое действие необходимо выполнить. В случае широковещательного намерения — это, напротив, действие, которое произошло и о котором сообщается. Можно указать свои собственные действия для использования в своём приложении, но обычно указываются константы действий из класса `Intent`, например:
    - `ACTION_VIEW` – если требуется отобразить какую-то информацию, например, адрес сайта для открытия в браузере, фотография для просмотра или адрес для отображения на карте.
    - `ACTION_SEND` – если есть данные, которыми нужно поделиться через другое приложение, например, для отправки электронной почты, мессенджер или соцсети.
- **Данные:** объект класса `Uri`, который содержит данные, над которыми требуется выполнить действие, и/или MIME-тип этих данных. Указание MIME-типа часто играет важную роль, потому что позволяет точнее определять какой компонент подходит для данных: хотя аудиофайл и фотография оба относятся к классу «медиа», но маловероятно что аудиопроигрыватель сможет показывать и фотографии. Для установки данных используется функция `setData()`, для задания типа `setType()`, а для того и другого – `setDataAndType()`.
- **Категория:** строка, содержащая дополнительную информацию о типе компонента, который должен обрабатывать намерение. В намерении можно поместить любое количество категорий, но используются они довольно редко.
- **Дополнительная информация:** это пары «ключ-значение», которые могут быть нужны для выполнения запрошенного действия. Добавить их можно с помощью методов `putExtra()`, или путём создания `Bundle`-объекта со всеми данными сразу – тогда используется метод `putExtras()`.

Пример явного намерения: запуск другой активности своего приложения:

```
val intent = Intent(this, BookActivity::class.java)
intent.data = Uri.parse(bookId)
startActivity(intent)
```

Пример неявного намерения: запуск почтового клиента для написания письма:

```
val intent = Intent().apply {
    action = Intent.ACTION_SEND
    putExtra(Intent.EXTRA_TEXT, textMessage)
    type = "text/plain"
}

try {
    startActivity(intent)
}
catch (e: ActivityNotFoundException) {
    // Подходящего приложения может не оказаться!
    // Нужно корректно обрабатывать такие ситуации.
}
```

Чтобы предоставить пользователю возможность выбрать приложение, которое будет использоваться для обработки события, нужно создать намерение с помощью функции

`createChooser()`, и затем передать его в `startActivity` – это приведёт к отображению диалога со списком подходящих приложений:

```
val sendIntent = Intent(Intent.ACTION_SEND)
...
val title = resources.getString(R.string.chooser_title) // "Выберите приложение:"
val chooser = Intent.createChooser(sendIntent, title)
if (sendIntent.resolveActivity(packageManager) != null)
    startActivity(chooser)
```

Неявные намерения могут использоваться для выполнения самых разных действий, таких как установка будильника, добавление или просмотр контакта, проигрывание музыкального файла и т.п.

- Инициировать звонок на указанный номер:

  ```
  val callIntent = Uri.parse("tel:71234567890").let { number ->
      Intent(Intent.ACTION_DIAL, number)
  }
  ```

- Отобразить место с указанными координатами на карте:

  ```
  val intent = Uri.parse("geo:56.465390,84.950164").let { location ->
      Intent(Intent.ACTION_VIEW, location)
  }
  ```

- Открытие страницы в браузере:

  ```
  val intent = Uri.parse("https://www.tpu.ru/").let { webpage ->
      Intent(Intent.ACTION_VIEW, webpage)
  }
  ```

Примеры других действий можно посмотреть в документации (*англ.*).

Запуск другой активности

Часто намерения используются для запуска другой активности в приложении. В простейшем случае для этого достаточно просто указать класс активности, которую требуется запустить:

```
val intent = Intent(this, SecondActivity::class.java)
startActivity(intent)
```

Часто, однако, требуется передать в запускаемую активность какие-либо данные. Для этого используются методы `putExtra`:

```
val intent = Intent(this, SecondActivity::class.java)
intent.putExtra("item", "Книга")
intent.putExtra("price", 137)
startActivity(intent)
```

Вторая активность может прочитать эти данные с помощью объекта `intent`, который содержит информацию о запуске активности:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_second)
    val item = intent.getStringExtra("item") ?: ""
```

```
    val price = intent.getIntExtra("price", 0)
}
```

Если какого-то параметра не оказалось, или он некорректный (например, строка вместо числа), то соответствующая функция либо вернёт значение по умолчанию (переданное вторым параметром), либо `null` и в этом случае нужно использовать один из способов разрешения `null`-значений, например, Элвис-оператор, как в примере выше.

Запуск активности с получением результата

Иногда требуется не просто запустить другую активность, но и дождаться пока она завершится и вернёт какие-то результаты. Для этого есть два подхода: современный и устаревший.

Важно разобраться с обоими подходами – современный подход используется в разных областях, не только для запуска активности, но и для запроса разрешений, получения снимка с камеры и т.д. В то же время устаревший подход по-прежнему широко используется в унаследованном коде, и нужно понимать как он работает.

Современный подход
Для вызова другой активности регистрируется специальный объект-посредник с функцией обратного вызова, который будет вызван после окончания работы функции:

```
var resultLauncher = registerForActivityResult(StartActivityForResult()) { result ->
    if (result.resultCode == Activity.RESULT_OK) {
        val data = result.data
        // Анализ результатов
        ...
    }
}
```

Функция `registerForActivityResult` создаёт объект-посредник. В качестве параметра она принимает класс-контракт – он определяет какие результаты вернутся после завершения вызываемого действия. Так как требуется получить результат из активности, то указан класс `StartActivityForResult`. Кроме него могут использоваться и другие классы, например:

- `PickContact` – диалог выбора контакта
- `RequestPermission` – диалог запроса разрешения
- `TakePicture` / `TakeVideo` – захват фото или видео с камеры
- `OpenDocument` – диалог выбора файла в хранилище

Вызов активности производится с помощью этого объекта-посредника:

```
val intent = Intent(this, SecondActivity::class.java)
resultLauncher.launch(intent)
```

Вторая активность запускается и как-то взаимодействует с пользователем. Когда нужно вернуть результат, она вызывает примерно такой код:

```
val intent = Intent()
intent.putExtra("result", ...)
intent.putExtra("somedata", ...)
setResult(Activity.RESULT_OK, intent)
finish()
```

Да, результаты тоже возвращаются через `Intent`! А функция `finish()` прекращает работу активности.

После закрытия второй активности её результат поступает в объект-посредник, и может быть каким-то образом проанализирован.

Устаревший подход
До введения объекта-посредника для получения результатов из другой активности использовалась функция `startActivityForResult()`, которая просто запускала другую активность и завершала свою работу, а когда вторая активность возвращала управление, то система вызывала функцию обратного вызова `onActivityResult()`, в которой анализировалось какая именно активность завершилась, и получились результаты:

1. Первая активность запускает вторую:

   ```
   val intent = Intent(this, ItemActivity::class.java)
   startActivityForResult(intent, MY_CODE)
   ```

   Здесь `MY_CODE` – это просто числовая константа, чтобы когда придут результаты работы второй активности, их можно было отличить от результатов работы других активностей, ведь все они возвращаются в одну функцию `onActivityResult`.

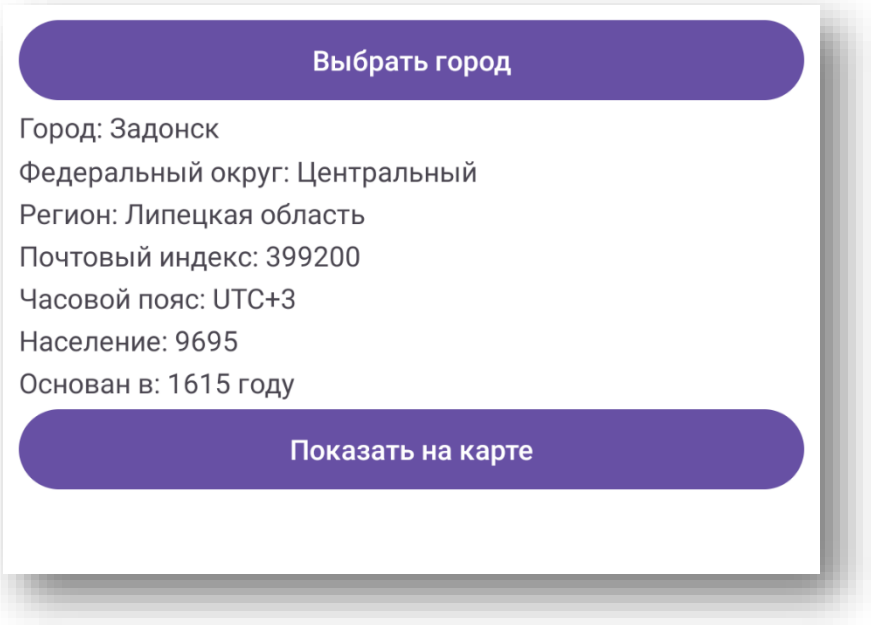
2. Вторая активность возвращает код обычным образом, который был разобран выше.
3. В первой активности система ищет и вызывает функцию `onActivityResult`, в которой производится анализ результатов:

   ```
   override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
       super.onActivityResult(requestCode, resultCode, data)
       if (requestCode == MY_CODE && resultCode == Activity.RESULT_OK) {
           val result = data?.getIntExtra("index", -1)
           val somedata = data?.getStringExtra("item") ?: ""
           ...
       }
   }
   ```

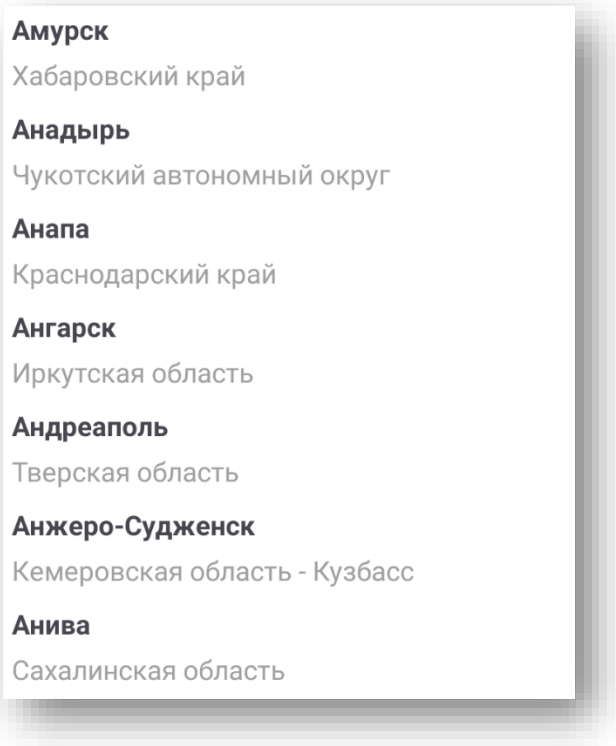
Хотя такой метод по-прежнему работает, он объявлен устаревшим и больше не рекомендуется к использованию.

Задание

Создайте приложение, отображающее информацию о выбранном городе, а также показывающее его на карте. Внешний вид приложения должен быть примерно следующий:

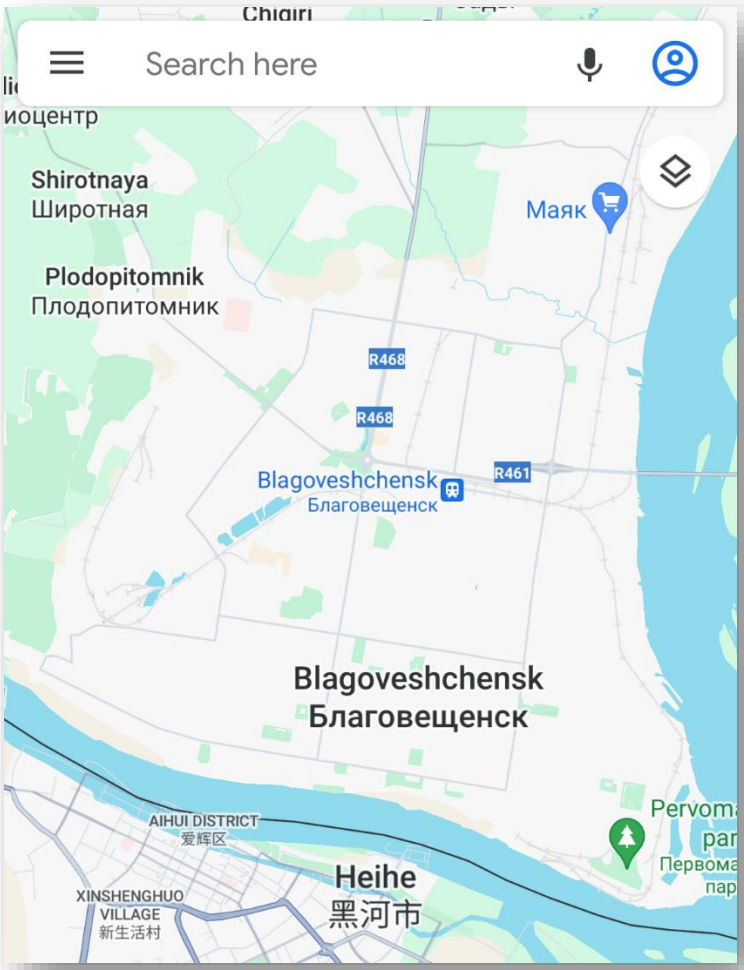


При нажатии кнопки «Выбрать город» запускается вторая активность, в которой отображается список `RecyclerView` с названиями городов и их регионов (ведь могут быть города с одинаковыми названиями, которые находятся в разных частях страны!). Вторая активность может выглядеть примерно так:



При нажатии на элемент списка активность закрывается, и информация о том какой город был выбран передаётся в основную активность, которая, в свою очередь, тут же отображает информацию об этом городе.

Кнопка «Показать на карте» запускает неявное намерение с координатами, система найдёт подходящее приложение для отображения карт и запустит его, показав нужный город:

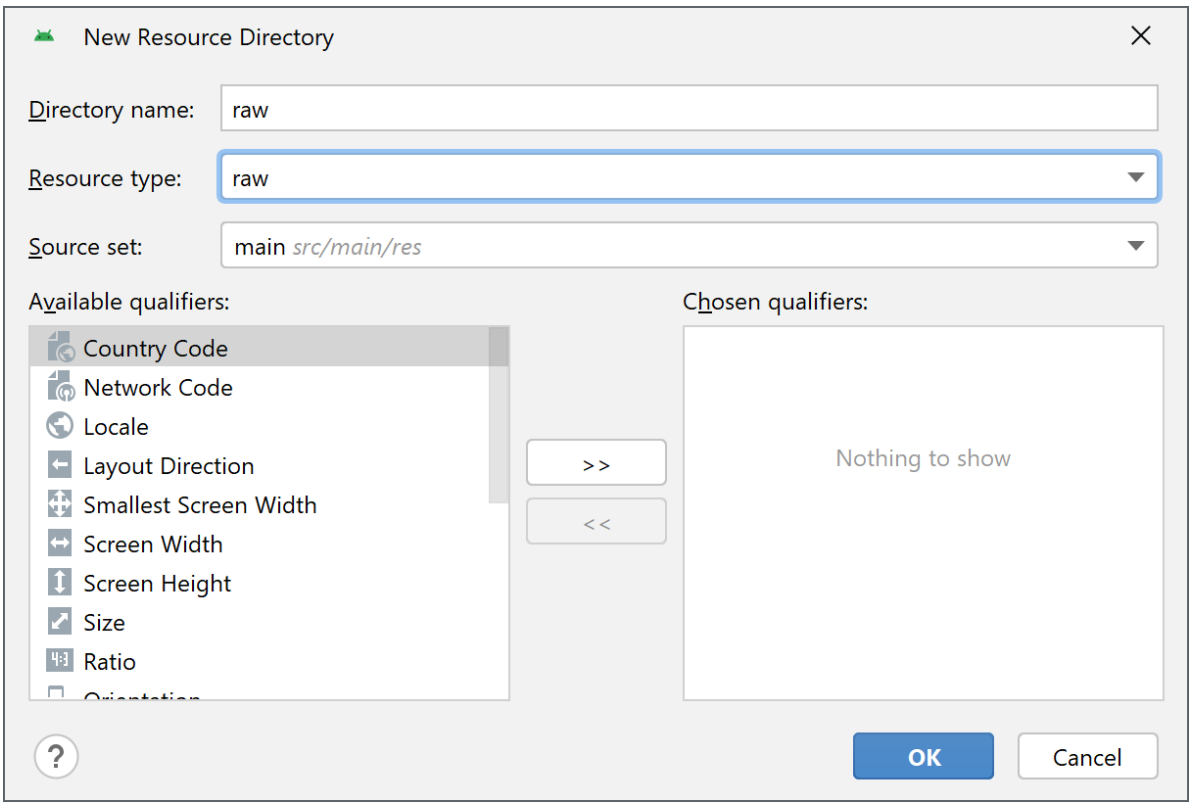


Список городов с информацией о них можно найти в интернете, однако процесс её сбора может быть достаточно трудоёмким, поэтому можно воспользоваться готовым файлом, приложенным к этой странице. Файл содержит данные в формате CSV, когда каждая строчка содержит одну запись, а поля в ней разделены точкой с запятой или каким-то другим разделителем. В общем случае для работы с таким файлом крайне желательно использовать какую-либо библиотеку, поскольку часто данные могут содержать кавычки и их правильное разбиение может оказаться нетривиальным процессом. Однако в нашем файле данные довольно простые, поэтому используем встроенные функции для их чтения. Прежде всего создадим класс `City` для хранения информации о городе:

```
data class City(
    val title: String,
    val region: String,
    val district: String,
    val postalCode: String,
    val timezone: String,
    val population: String,
    val founded: String,
    val lat: Float,
    val lon: Float
)
```

Сам файл *cities.csv* нужно положить в ресурсы, в папку `raw`, которая содержит файлы в произвольном формате. Изначально этой папки может не быть, поэтому её нужно создать:

щёлкнуть правой кнопкой на папке *res*, и выбрать *New → Android Resource Directory*, и в поле *Resource type* указать *raw*:



А затем уже в эту папку можно положить файл *cities.csv*. Доступ к таким raw-ресурсам осуществляется с помощью функции `resources.openRawResource()`, которая открывает указанный файл для чтения и возвращает стандартный класс `InputStream`, из которого можно читать данные побайтово или целыми блоками. А можно, как в примере ниже, вызвать метод `bufferedReader()` и получить объект для чтения текстовых данных построчно.

Поскольку доступ к данным потребуется как из основной активности (для информации о городе), так и из второй активности (для списка городов), то вынесем данные о городах в *синглтон* – особый объект, который описывается как класс, но с ключевым словом `object`:

```
object Common {
    val cities = mutableListOf()

    fun initCities(ctx: Context) {
        if (cities.isEmpty()) {
            val lines = ctx.resources.openRawResource(R.raw.cities)
                        .bufferedReader().lines().toArray()
            for (i in 1 until lines.count()) {
                val parts = lines[i].toString().split(";")
                val city = City(
                    parts[3], // city
                    parts[2], // region
                    parts[1], // federal_district
                    parts[0], // postal_code
                    parts[4], // timezone
                    parts[7], // population
                    parts[8], // founded
                    parts[5].toFloat(), // lat
                    parts[6].toFloat()  // lon
                )
```

```
                cities.add(city)
            }
            cities.sortBy { it.title }
        }
    }
}
```

В этом объекте создан список городов `cities`, и определена функция, которая читает CSV-файл из ресурсов, разбирает его на составляющие, и формирует список городов. И ещё сортирует его по полю `title`! Обращаться к такому объекту можно как к обычному статическому классу:

```
Common.initCities(this)
...
val title = Common.cities[0].title
...
```

В качестве результата лабораторной работы как обычно загрузите отчёт и zip-архив с проектом.