Диалоги

Диалог – это небольшое окно, предлагающее пользователю принять решение или ввести дополнительную информацию. Диалоговое окно не заполняет экран и обычно используется для модальных событий, требующих от пользователей выполнения действия, прежде чем они смогут продолжить взаимодействие с программой.

Базовым классом для диалогов является класс `Dialog`, но он не используется для непосредственного создания экземпляров. Вместо этого используется один из следующих подклассов:

- `AlertDialog` – может отображать заголовок, до трех кнопок, список выбираемых элементов или пользовательскую разметку.
- `DatePickerDialog` или `TimePickerDialog` – позволяет выбрать дату или время.

Кроме того, обычно используется контейнер `DialogFragment`, который предоставляет все элементы управления для диалогового окна и управления его внешним видом. Он позволяет правильно обрабатывать события жизненного цикла, например, когда пользователь нажимает кнопку «Назад» или поворачивает экран.

Создание диалога

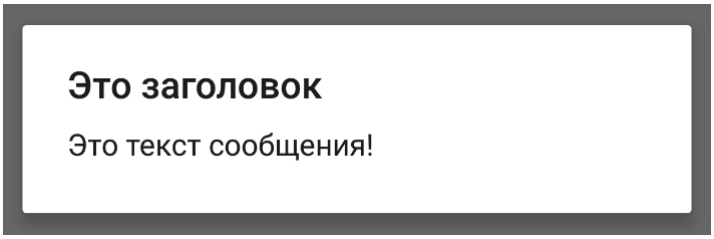
Класс `AlertDialog` позволяет создавать различные варианты оформления диалогов, и в большинстве случаев можно обойтись только им. Диалоговое окно `AlertDialog` состоит из трех областей:

- Заголовок является необязательным и используется только если область содержимого занята подробным сообщением, списком или пользовательским макетом. Если нужно показать сообщение или вопрос, заголовок не нужен.
- Область содержимого отображает сообщение, список вариантов или пользовательскую разметку.
- Кнопки действий, их может быть до трёх штук.

Для создания диалога используется статический метод `AlertDialog.Builder`:

```
val builder = AlertDialog.Builder(context)
builder
    .setMessage("Это текст сообщения!")
    .setTitle("Это заголовок")

val dialog: AlertDialog = builder.create()
dialog.show()
```

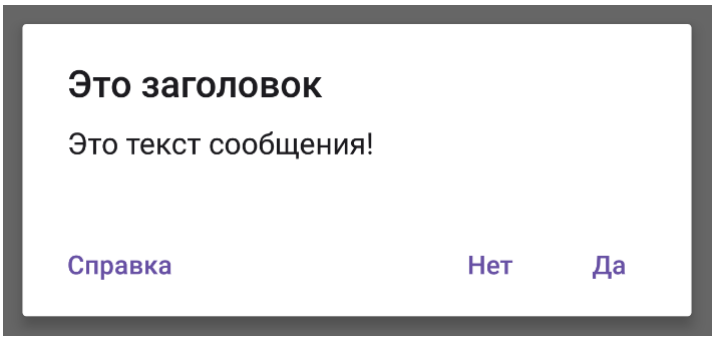
В результате на экране появится такое диалоговое окно:



Для добавления в диалог кнопок действий нужно использовать одну или несколько функций `setXxxButton()`:

```
builder
    .setMessage("Это текст сообщения!")
    .setTitle("Это заголовок")
    .setPositiveButton("Да") { dialog, which ->
        // Что-то делаем
    }
    .setNegativeButton("Нет") { dialog, which ->
        // Делаем что-то другое
    }
    .setNeutralButton("Справка") { dialog, which ->
        // Подсказываем что делать...
    }
```

В результате в диалог будут добавлены кнопки: нейтральная слева, положительные и отрицательные справа. Повлиять на расположение кнопок нельзя, при повторном вызове функции для какого-то типа кнопок – новое действие просто заменит предыдущее:



Функция обратного вызова, которая передаётся в метод при добавлении кнопки, вызывается при её нажатии, и в качестве параметра получает объект `dialog`. С его помощью можно управлять диалогом, например, закрыть его:

```
dialog.dismiss()
```

Диалог со списками

`AlertDialog` позволяет создать список одного из трёх типов:

- Список строк, можно выбрать одну.
- Радиокнопки, можно выбрать одну.
- Чекбоксы, можно выбрать несколько.

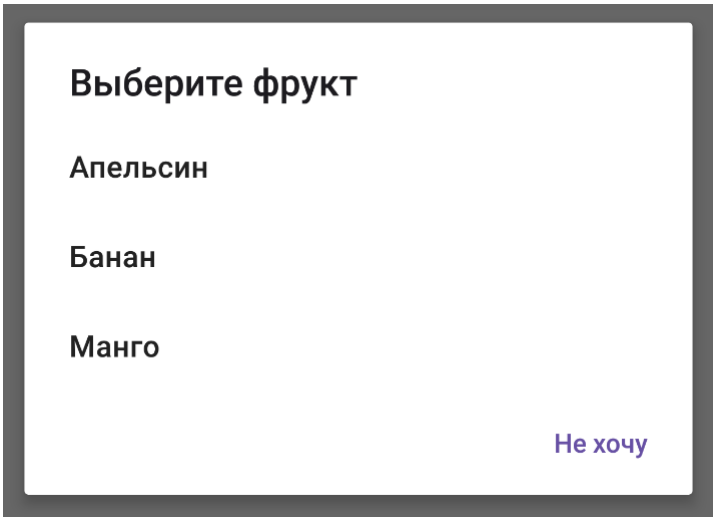
Списки занимают область содержимого, поэтому текст, который в примерах выше задавался с помощью `setMessage()`, выводиться уже не будет! Для пояснения к списку остаётся только область заголовка.

Для создания списка строк используется метод `setItems()`:

```
builder
    .setTitle("Выберите фрукт")
    .setItems(arrayOf("Апельсин", "Банан", "Манго")) { dialog, which ->
        // which содержит номер выбранного элемента из массива
    }
    .setNegativeButton("Не хочу") { dialog, which ->
```

```
        // Делаем что-то другое
    }
```

В результате на экране появится вот такой список:

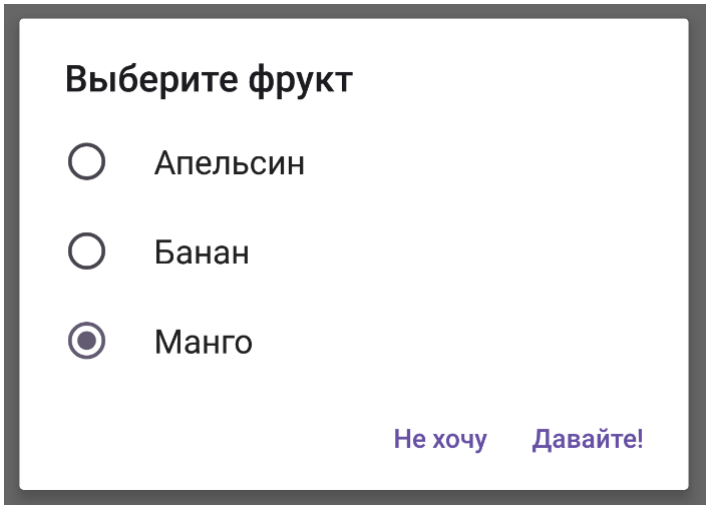


При касании какой-либо строки диалог немедленно закрывается, а функция обратного вызова, установленная методом `setItems()`, получает номер выбранного элемента массива. Поэтому обычно нет смысла добавлять положительную кнопку действия, а вот отрицательную или нейтральную можно добавить для отказа от выбора или других действий.

Для добавления списков с радиокнопками или чекбоксами используется аналогичный подход, только вместо `setItems()` используются методы `setSingleChoiceItems()` или `setMultiChoiceItems()` соответственно.

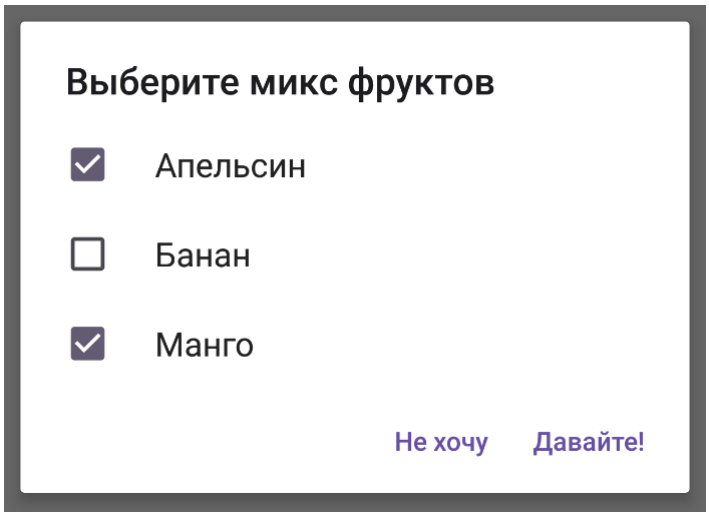
```
builder
    .setTitle("Выберите фрукт")
    .setSingleChoiceItems(arrayOf("Апельсин", "Банан", "Манго"), 2) { dialog, which ->
        Toast.makeText(this, which.toString(), Toast.LENGTH_SHORT).show()
    }
    .setPositiveButton("Давайте!") { dialog, which ->
        // Делаем что-то другое
    }
    .setNegativeButton("Не хочу") { dialog, which ->
        // Делаем что-то другое
    }
```

В списках с радиокнопками и чекбоксами диалог не закрывается при нажатии на элемент, поэтому в этих случаях добавлять кнопку с положительным действием нужно. В момент нажатия на радиокнопку или чекбокс в функцию обратного вызова приходит информация о нажатом элементе, его нужно как-то обрабатывать или запоминать, в то время как диалог продолжает оставаться на экране. Кроме того, в метод добавлен и выбор элемента по умолчанию:



Список с чекбоксами работает аналогично, но вместо номера выбранного элемента передаётся массив логических значений:

```
builder
    .setTitle("Выберите микс фруктов")
    .setMultiChoiceItems(arrayOf("Апельсин", "Банан", "Манго"),
        booleanArrayOf(true, false, true)) { dialog, which, isChecked ->
        Toast.makeText(this, which.toString(), Toast.LENGTH_SHORT).show()
    }
    .setPositiveButton("Давайте!") { dialog, which ->
        // Делаем что-то другое
    }
    .setNegativeButton("Не хочу") { dialog, which ->
        // Делаем что-то другое
    }
```



Диалог с пользовательской разметкой

Если в диалоге требуется более сложная структура, то возможно использование собственной разметки. Создадим разметку в файле *dialog_login.xml* со следующим содержимым:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
```

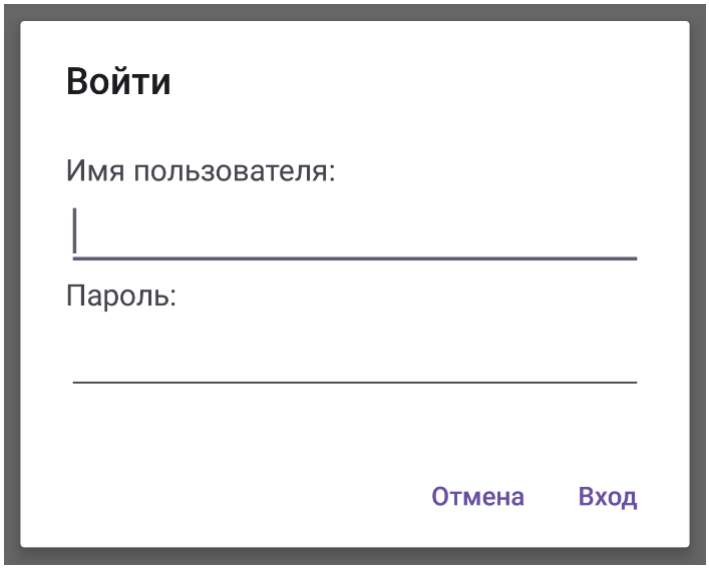
```
    android:layout_height="match_parent"
    android:padding="?dialogPreferredPadding">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="16sp"
        android:text="Имя пользователя:"/>

    <EditText
        android:id="@+id/login"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="16sp"
        android:text="Пароль:"/>

    <EditText
        android:id="@+id/password"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="textPassword"/>

</LinearLayout>
```

Обратите внимание на атрибут `padding` со значением `?dialogPreferredPadding` – это стандартное значение, которое соответствует отступам от края диалога. Использование такого атрибута позволит пользовательской разметке более аккуратно «вписаться» в габариты диалога.

В коде подключение разметки осуществляется методом `setView()`. В этот метод можно передать либо идентификатор ресурса разметки (например, `R.layout.dialog_login`), либо ссылку на созданную в программе разметку. Второй вариант часто предпочтительнее, потому что в этом случае проще получить доступ к элементам управления в диалоге.

```
builder
  .setTitle("Войти")
  // Создание разметки из ресурса
  val view = LayoutInflater.from(requireContext()).inflate(R.layout.dialog_login, null)
  val etLogin = view.findViewById<EditText>(R.id.login)
  val etPassword = view.findViewById<EditText>(R.id.password)
  .setView(view)
  .setPositiveButton("Вход") { dialog, which ->
      // Вход с систему
  }
  .setNegativeButton("Отмена") { dialog, which ->
      // Отмена авторизации
  }
```

На экране появится диалог с пользовательской разметкой:



Для доступа к значениям диалога в этом примере будут использоваться объекты `etLogin` и `etPassword`, полученные из созданной разметки `view` с помощью метода `findViewById()`.

Использование DialogFragment

`AlertDialog` удобен и прост, однако если повернуть экран, то созданный с его помощью диалог исчезнет, да и нажатие кнопки «Назад» закроет диалог без какой-либо реакции. Чтобы полноценно использовать диалоги – нужно задействовать класс `DialogFragment`:

```
class MyDialog : DialogFragment() {

    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
        val builder = AlertDialog.Builder(this.requireContext())
        builder.setMessage("Это диалог!")
        builder.setPositiveButton("OK", null)
        return builder.create()
    }
}
```

Теперь можно в нужном месте программы вызвать это окно:

```
val dlg = MyDialog()
dlg.show(supportFragmentManager, "my_dialog")
```

Вроде бы изменилось не так много – внутри всё тот же `AlertDialog`, но если повернуть экран, то диалог уже никуда не исчезает! Класс `DialogFragment` берёт на себя «закадровую» работу по обработке подобных технических моментов.

Если требуется передать из диалога какие-либо данные в вызывающую активность, то вариант с передачей функции с помощью `setOnItemClickListener`, к сожалению, не подойдёт. Ведь если сменится конфигурация (например, будет изменена ориентация телефона), то вызывающая активность будет пересоздана заново, и привязка функции пропадёт. Рекомендуемый способ в таком случае: объявить и реализовать в вызывающей активности какой-либо интерфейс, а в диалоге в функции `onAttach` сохранять информацию об активности. Эта функция будет вызываться каждый раз при (пере)создании диалога, в т. ч. смене конфигурации:

```
class MyDialog : DialogFragment() {

    // Интерфейс с одной функцией - чтобы передать нужные параметры в активность
    interface DataListener {
        fun onDialogData(username: String, password: String)
    }

    // Слушатель интерфейса
    lateinit var listener: DataListener


    builder.setPositiveButton("Вход") { dialog, which ->
        // Вызов функции из интерфейса - передача данных перед закрытием диалога
        listener.onDialogData(tvUsername.text.toString(), tvPassword.text.toString())
    }

    // (Пере)привязка к активности
    override fun onAttach(context: Context) {
        super.onAttach(context)
        listener = context as DataListener
    }
}
```

При таком подходе в активности должен быть реализован этот интерфейс, и тогда даже если активность будет пересоздана заново во время отображения диалога – функция для приема данных всё равно будет в ней присутствовать:

```
class MainActivity : AppCompatActivity(), MyDialog.DataListener {

    override fun onDialogData(username: String, password: String) {
        Toast.makeText(this, "Вход как: $username", Toast.LENGTH_SHORT).show()
    }

}
```

Диалоги выбора даты и времени

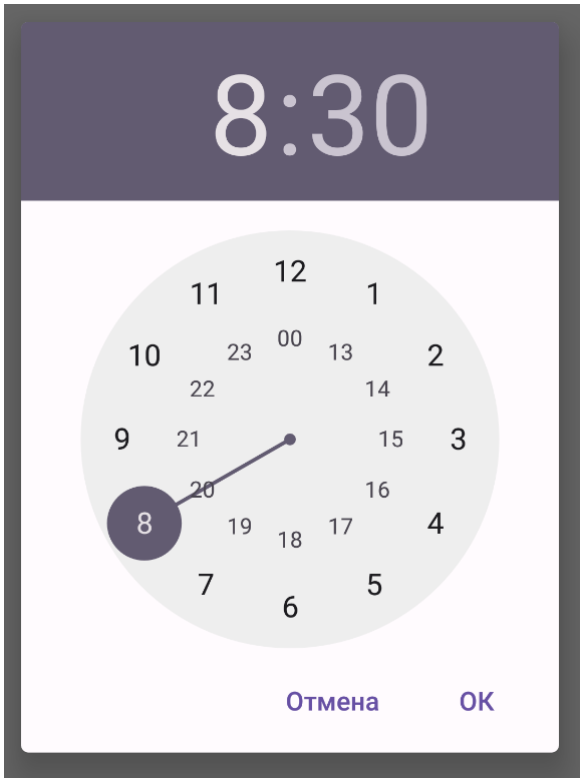
Для выбора даты используется встроенный диалог `DatePickerDialog`: на вход ему нужно передать начальные год, месяц и день:

```
override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
    return DatePickerDialog(requireContext(), {
            view, year, month, day ->
    }, 2023, 11, 15)
}
```

Следует обратить внимание на номер месяца: согласно стандартной практике Java нумерация начинается с нуля, поэтому январь – это 0, а декабрь – 11! На экране появится примерно такой диалог:

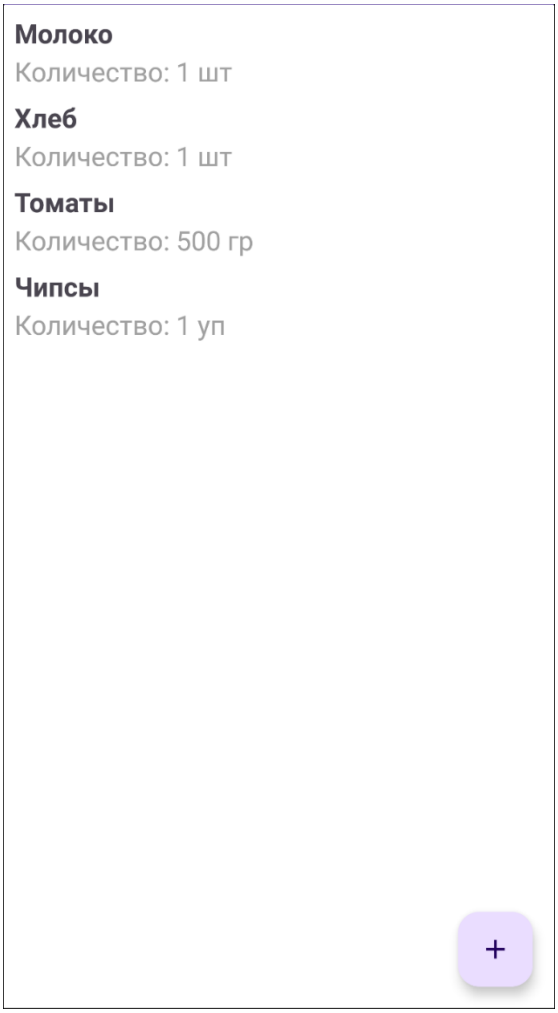
Аналогично, для выбора времени используется диалог `TimePickerDialog`:

```
override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
    return TimePickerDialog(activity, {
            view, hour, minute ->
    }, 8, 30, true)
}
```

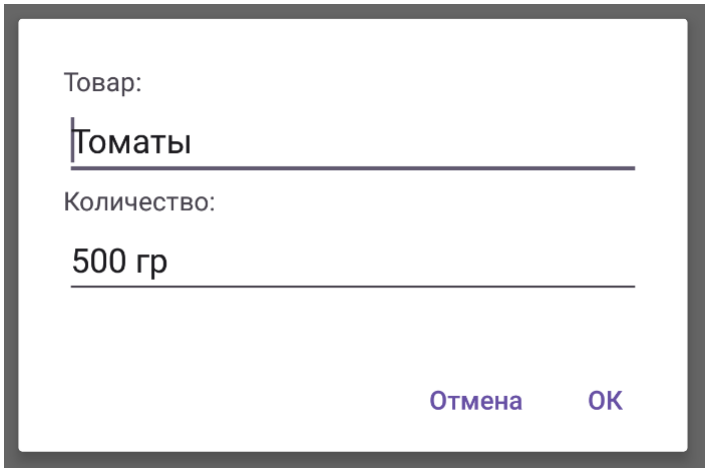


Задание

Создайте приложение для ведения списка покупок. В основе должен лежать `RecyclerView`, который отображает список примерно в следующем виде:



При нажатии плавающей кнопки должен появляться диалог с пустыми полями, а при его положительном завершении – в список должна добавляться новая покупка:



При касании записи она должна открываться на редактирование: появляется такой же диалог, но уже с заполненными полями, а при положительном завершении – этот элемент в списке обновляется.

При жесте смахивания на какой-либо записи (проведении по ней пальцем влево или вправо) она должна удаляться из списка.

Плавающая кнопка

Для добавления плавающей кнопки используется элемент управления `FloatingActionButton`. По сути это обычная кнопка с тенью и картинкой, проблема же

заключается в том как разместить её поверх списка. Для этого идеально подходит контейнер `FrameLayout`, который «складирует» элементы один над другим, а «растаскивать» их по экрану нужно с помощью атрибута `layout_gravity`. Для данной лабораторной можно использовать примерно такую разметку:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/list"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="fill"
        app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"/>

    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/fab_add"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/add"
        android:layout_margin="16dp"
        android:layout_gravity="bottom|end"/>

</FrameLayout>
```

Жест смахивания

Для реализации жеста смахивания элемента из списка используется класс `ItemTouchHelper`. Он автоматически отслеживает смахивания и перетаскивание элементов вверх или вниз, а от программы требуется лишь реагировать на уведомления об этом. Для этого следует создать слушатель этих уведомлений – они идут в комплекте друг с другом, поэтому требуется добавить обе функции, даже если нужна только одна из них:

```
val swipeCallback = object : ItemTouchHelper.SimpleCallback(0,
    ItemTouchHelper.LEFT + ItemTouchHelper.RIGHT) {

    // Перетскивание не требуется, поэтому возвращаем false
    override fun onMove(recyclerView: RecyclerView,
                        viewHolder: RecyclerView.ViewHolder,
                        target: RecyclerView.ViewHolder): Boolean {
        return false
    }

    // Смахивание - удаляем элемент и оповещаем об этом адаптер
    override fun onSwiped(viewHolder: RecyclerView.ViewHolder, direction: Int) {
        val pos = viewHolder.adapterPosition
        items.removeAt(pos)
        adapter.notifyItemRemoved(pos)
    }
}
```

В конструкторе указывается направление, в котором допустим жест смахивания – сейчас там стоит `LEFT` и `RIGHT`, но можно оставить только одну из констант.

Функция `onSwipe` получает в качестве параметра объект-держатель того элемента, который подвергается смахиванию. Как правило, требуется лишь узнать позицию этого элемента в списке, а затем удалить его, и – обязательно! – известить об изменениях адаптер. Для извещения адаптера об изменениях предусмотрено несколько функций:

- `notifyItemInserted` – добавлен элемент с указанным индексом
- `notifyItemChanged` – изменён элемент с указанным индексом
- `notifyItemMoved` – элемент с указанным индексом перемещён
- `notifyItemRemoved` – удалён элемент с указанным индексом
- `notifyDataSetChanged` – вообще всё поменялось: данную функцию использовать не рекомендуется, она плохо влияет на производительность, нужно использовать функции из списка выше

После того как слушатель готов, его следует подключить к списку:

```
val swipeHelper = ItemTouchHelper(swipeCallback)
swipeHelper.attachToRecyclerView(myRecyclerView)
```

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