# Практическая работа 17. Навигация в RecyclerView

Некоторые приложения требуют, чтобы пользователь выбрал элемент из списка. В этой работе вы узнаете, как сделать `RecyclerView` центральной частью структуры вашего приложения, для чего следует организовать обработку щелчков на их элементах. Вы увидите, как реализовать навигацию в `RecyclerView`, чтобы приложение переходило к новому экрану каждый раз, когда пользователь щелкает на записи. Мы покажем, как вывести дополнительную информацию о выбранной записи и обновить ее в базе данных. К концу этой главы у вас будут все средства, необходимые для того, чтобы преобразовать ваши великолепные замыслы в приложение вашей мечты.

## Навигация в `RecyclerView`

Вы узнали, как построить `RecyclerView`, которое выводит список данных с возможностью прокрутки, а также научились использовать `DiffUtil` для повышения его эффективности. Тем не менее это еще не все. `RecyclerView` становятся важнейшим компонентом многих приложений Android, потому что помимо отображения списков данных они могут использоваться для навигации в приложениях. Когда пользователь щелкает на элементе `RecyclerView`, приложение может переходить к новому фрагменту с расширенной информацией об этой записи. Чтобы понять, как работает навигация, мы изменим приложение `Tasks`, чтобы при щелчке на одной из задач в представлении с переработкой происходил переход к новому фрагменту. Этот фрагмент выводит информацию выбранной записи и предоставляет пользователю возможность обновить или удалить ее:

## Структура приложения Tasks в текущей версии

Прежде чем разбираться в том, как нужно изменить приложение `Tasks`, вспомним его текущую структуру. Приложение состоит из одной активности (`MainActivity`), которая отображает фрагмент с именем `TasksFragment`. Этот фрагмент представляет главный экран приложения, а его макет включает `RecyclerView`, которое отображает сетку задач. `RecyclerView` использует адаптер с именем `TaskItemAdapter`, а размещение его элементов определяется файлом макета. `TasksFragment` использует модель представления с именем `TasksViewModel`. Модель представления отвечает за бизнес-логику фрагмента и получает свои данные из базы данных `Room` через интерфейс `TaskDao`. Эти компоненты взаимодействуют по следующей схеме:

## Использование `RecyclerView` для перехода к новому фрагменту

Мы обновим приложение `Tasks`, чтобы по щелчку на задаче в представлении с переработкой отображался фрагмент с именем `EditTaskFragment`. Новый фрагмент будет выглядеть так:

Таким образом, `EditTaskFragment` включает текстовое поле и флажок, при помощи которых пользователь может редактировать задачу. В текстовом поле выводится имя задачи, а флажок содержит признак завершения. Фрагмент также включает кнопку `Update Task`, по которой обновляется запись в базе данных, и кнопку `Delete Task` для удаления записи. По щелчку на любой из этих кнопок приложение возвращается к фрагменту `TasksFragment`, который отображает обновленный список задач в своем представлении с переработкой:

## Что мы собираемся сделать

Построение новой версии приложения состоит из трех фаз:

1. Программирование реакции на щелчки. Мы обновим приложение, чтобы по щелчку на задаче в представлении с переработкой идентификатор этой задачи отображался на панели `Toast`.
2. Переход к `EditTaskFragment` по щелчку на задаче. Мы создадим `EditTaskFragment` и воспользуемся компонентом `Navigation` для перехода к этому фрагменту, когда пользователь щелкает на записи. Открывается новый фрагмент с идентификатором задачи.
3. Запись с описанием задачи отображается в `EditTaskFragment`, чтобы пользователь мог обновить или удалить запись. Для `EditTaskFragment` будет создана модель представления, которая использует интерфейс `TaskDao` для взаимодействия с базой данных.

## Реакция на щелчки

Первое изменение, которое будет внесено в приложение `Tasks`, — вывод панели `Toast` по щелчку на одном из элементов `RecyclerView`. Чтобы каждый элемент мог реагировать на щелчки, мы добавим `OnClickListener` к корневому представлению каждого элемента. Для этого метод `setOnClickListener()` каждого элемента будет вызываться сразу же после добавления данных каждого элемента в его макет. Слушателей `OnClickListener` лучше всего добавлять в методе `bind()` объекта `TaskItemViewHolder`, так как именно здесь переменной связывания данных макета присваивается объект `Task`. Напомним, что метод `bind()` вызывается методом `onBindViewHolder()` объекта `TaskItemAdapter`, который вызывается каждый раз, когда представлению с переработкой потребуется отобразить данные элемента. Ниже приведен код добавления `OnClickListener` в корневое представление макета каждого элемента; мы добавим его в `TaskItemAdapter.kt` через несколько страниц:

```
class TaskItemAdapter : ListAdapter<...>(TaskDiffItemCallback()) {
 ...
 override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
 val item = getItem(position)
 holder.bind(item)
 }
 class TaskItemViewHolder(val binding: TaskItemBinding)
 : ```RecyclerView```.ViewHolder(binding.root) {
 ...
 fun bind(item: Task) {
 binding.task = item
 binding.root.setOnClickListener {
 //Код, выполняемый по щелчку на элементе
 }
 }
 }
}
```

Итак, теперь вы знаете, как добавить слушатель OnClickListener для каждого элемента. Теперь нужно сделать так, чтобы по щелчку на элементе отображалась панель Toast.

## Где создать Toast?

Чтобы панель `Toast` отображалась каждый раз, когда пользователь щелкает на элементе, можно добавить следующий код в метод `setOnClickListener()` держателя представлений:

```
class TaskItemViewHolder(val binding: TaskItemBinding)
 : ```RecyclerView```.ViewHolder(binding.root) {
 ...
 fun bind(item: Task) {
 binding.task = item
 binding.root.setOnClickListener {
 Toast.makeText(binding.root.context, "Clicked task ${item.taskId}",
 Toast.LENGTH_SHORT).show()
 }
 }
}
```

Однако такое решение будет означать, что код, описывающий поведение приложения, будет помещен в код держателя приложения. Последний отвечает за связывание данных с макетом каждого элемента, так что такое размещение вряд ли можно считать естественным. Включение кода `Toast` в держатель представления также снижает гибкость кода держателя представления. Это будет означать, что каждый раз, когда пользователь щелкает на элементе, код будет только отображать панель `Toast` и использовать его в других местах не удастся. Нет ли альтернативного решения?

## Код Toast будет передаваться TasksFragment в лямбда-выражении

Альтернативное решение — определить код, необходимый каждому элементу для выполнения в `TasksFragment`, и передать его `TaskItemViewHolder` — через `TaskItemAdapter` — в лямбда-выражении. Такой подход означает, что тем, что происходит по щелчку, будет управлять фрагмент, а не держатель представления. Прежде чем рассматривать код, разберемся в том, как он работает.

## Как работает код

При выполнении кода происходят следующие события:

1. `TasksFragment` передает лямбда-выражение конструктору `TaskItemAdapter`. Лямбда-выражение включает код для отображения панели `Toast`.
2. При вызове метода `onBindViewHolder()` объекта `TaskItemAdapter` он вызывает метод `bind()` объекта `TaskItemViewHolder` и передает ему лямбда-выражение.
3. `TaskItemViewHolder` добавляет лямбда-выражение в слушатель `OnClickListener` каждого элемента. Когда пользователь щелкает на каждом элементе (`CardView`), выполняется лямбда-выражение и отображается панель `Toast`.

Чтобы реализовать эту схему, необходимо обновить код `TasksFragment`, `TaskItemAdapter` и `TaskItemViewHolder`. Начнем с обновления `TaskItemAdapter` и `TaskItemViewHolder`, чтобы адаптер мог получить лямбда и передать его держателю представления. Код для решения этой задачи приведен на следующей странице.

# Полный код TaskItemAdapter.kt

Ниже приведен код `TaskItemAdapter` и `TaskItemViewHolder`; обновите файл `TaskItemAdapter.kt`:

```
package com.hfad.tasks
import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.```RecyclerView```.widget.ListAdapter
import androidx.```RecyclerView```.widget.```RecyclerView```
import com.hfad.tasks.databinding.TaskItemBinding
class TaskItemAdapter(val clickListener: (taskId: Long) -> Unit)
 : ListAdapter<Task, TaskItemAdapter.TaskItemViewHolder>(TaskDiffItemCallback()) {
 override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
 : TaskItemViewHolder = TaskItemViewHolder.inflateFrom(parent)
 override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
 val item = getItem(position)
 holder.bind(item, clickListener)
 }
 class TaskItemViewHolder(val binding: TaskItemBinding)
 : ```RecyclerView```.ViewHolder(binding.root) {
 companion object {
 fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
 val layoutInflater = LayoutInflater.from(parent.context)
 val binding = TaskItemBinding.inflate(layoutInflater, parent, false)
 return TaskItemViewHolder(binding)
 }
 }
 fun bind(item: Task, clickListener: (taskId: Long) -> Unit) {
 binding.task = item
 binding.root.setOnClickListener { clickListener(item.taskId) }
 }
 }
}
```

## Передача лямбда-выражения TaskItemAdapter

Итак, когда конструктор `TaskItemAdapter` получает параметр с лямбда-выражением, мы должны передать его в коде `TasksFragment` при создании фрагмента. Как вы помните, `TasksFragment` создает объект `TaskItemAdapter` в методе `onCreateView()`, который затем назначается представлению с переработкой следующим образом:

```
override fun onCreateView(
 inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
 ...
 val adapter = TaskItemAdapter()
 binding.tasksList.adapter = adapter
```

```
 ...
}
```

Так как при каждом щелчке на представлении с переработкой должна отображаться панель `Toast`, код можно обновить, чтобы конструктору `TaskItemAdapter` передавалось следующее лямбда-выражение:

```
override fun onCreateView(
 inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
 ...
 val adapter = TaskItemAdapter { taskId ->
 Toast.makeText(context, "Clicked task $taskId", Toast.LENGTH_SHORT).show()
 }
 binding.tasksList.adapter = adapter
 ...
}
```

Адаптер передает лямбда-выражение методу `bind()` объекта `TaskItemViewHolder`, который использует лямбда-выражение в коде `OnClickListener`, назначаемом в корневом представлении каждого элемента. Когда пользователь щелкает на элементе в представлении с переработкой, выполняется лямбда-выражение. Теперь вы знаете все необходимое для того, чтобы по щелчку на элементе `RecyclerView` отображалась панель `Toast`. Посмотрим, как выглядит полный код `TasksFragment`.

# Полный код TasksFragment.kt

Ниже приведен обновленный код `TasksFragment`; убедитесь в том, что код `TasksFragment.kt` включает все изменения:

```
package com.hfad.tasks
import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Toast
import androidx.lifecycle.Observer
import androidx.lifecycle.ViewModelProvider
import com.hfad.tasks.databinding.FragmentTasksBinding
class TasksFragment : Fragment() {
 private var _binding: FragmentTasksBinding? = null
 private val binding get() = _binding!!
 override fun onCreateView(
 inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
 ): View? {
 _binding = FragmentTasksBinding.inflate(inflater, container, false)
 val view = binding.root
```

```
 val application = requireNotNull(this.activity).application
 val dao = TaskDatabase.getInstance(application).taskDao
 val viewModelFactory = TasksViewModelFactory(dao)
 val viewModel = ViewModelProvider(
 this, viewModelFactory).get(TasksViewModel::class.java)
 binding.viewModel = viewModel
 binding.lifecycleOwner = viewLifecycleOwner

val adapter = TaskItemAdapter { taskId ->
 Toast.makeText(context, "Clicked task $taskId", Toast.LENGTH_SHORT).show()
 }
 binding.tasksList.adapter = adapter
 viewModel.tasks.observe(viewLifecycleOwner, Observer {
 it?.let {
 adapter.submitList(it)
 }
 })
 return view
 }
 override fun onDestroyView() {
 super.onDestroyView()
 _binding = null
 }
}
```

## Что происходит при выполнении кода

При выполнении приложения происходят следующие события:

1. `TasksFragment` создает объект `TaskItemAdapter` и назначает его адаптером для `RecyclerView`. Фрагмент передает лямбда-выражение (с именем `clickListener`) адаптеру, чтобы тот отображал панель `Toast` при выполнении.

2. `TasksFragment` передает `TaskItemAdapter` список `List<Task>`. `List<Task>` содержит актуальный список записей из базы данных.

3. Метод `onCreateViewHolder()` объекта `TaskItemAdapter` вызывается для каждого элемента, который должен отображаться в представлении с переработкой. В результате создается набор объектов `TaskItemViewHolder`.

4. Метод `onBindViewHolder()` объекта `TaskItemAdapter` вызывается для каждого `TaskItemViewHolder`. В результате вызывается метод `bind()` объекта `TaskItemViewHolder`, которому передается элемент, на котором щелкнул пользователь, и лямбдавыражение `clickListener`.

5. Метод `bind()` объекта `TaskItemViewHolder` добавляет слушателя `OnClickListener` к корневому представлению макета каждого держателя представления. В данном примере корневым представлением является `CardView`.

6. Когда пользователь щелкает на элементе в представлении с переработкой, слушатель `OnClickListener` регистрирует щелчок. Он выполняет лямбда-выражение `clickListener`,

которое отображает панель `Toast`.

При запуске приложения `TasksFragment` отображает в представлении с переработкой сетку карточек, как и прежде. Если щелкнуть на одной из задач, представление выводит ее идентификатор на панели `Toast`.

Вы узнали, как сделать так, чтобы элементы представления с переработкой реагировали на события щелчков. Мы воспользуемся новыми знаниями, чтобы приложение переходило к новому фрагменту по щелчку на элементе. Но сначала проверьте свои силы на следующем упражнении.

## `RecyclerView` должно использоваться для перехода к новому фрагменту

К настоящему моменту вы узнали, как заставить `RecyclerView` реагировать на события щелчков. Например, когда пользователь щелкает на задаче в представлении с переработкой приложения `Tasks`, на экране появляется сообщение. На следующем этапе мы изменим это поведение, чтобы по щелчку на элементе приложение переходило к новому фрагменту (который мы создадим) и выводило идентификатор задачи. Новая версия приложения должна выглядеть так:

Чтобы эта схема заработала, мы воспользуемся компонентом `Navigation` для перехода к новому фрагменту и плагином `Safe Args` для передачи фрагменту идентификатора задачи. Это означает, что для включения этих компонентов придется обновить проект и файлы `build.gradle` приложения.

## Сначала обновляется файл build.gradle проекта

Начнем с обновления файла `build.gradle` проекта. В нем должна содержаться информация о том, какая версия компонента `Navigation` используется в приложении, а также путь к классам для плагина `Safe Args`. Для этого откройте файл `Tasks/build.gradle` и добавьте следующие строки в соответствующие разделы:

```
buildscript {
 ext.nav_version = "2.3.5"
 ...
 dependencies {
 ...
 classpath "androidx.navigation:navigation-safe-args-gradle-plug-in:$nav_version"
 }
}
```

а затем файл `build.gradle`

Также необходимо добавить плагин `Safe Args` в файл `build.gradle` приложения вместе с зависимостью для компонента `Navigation`. Откройте файл `Tasks/app/build.gradle` и добавьте следующие строки в соответствующие разделы:

```
plugins {
 ...
 id 'androidx.navigation.safeargs.kotlin'
```

```
}
...
dependencies {
 ...
 implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
 ...
}
```

После внесения изменений щелкните на ссылке `Sync Now`, чтобы синхронизировать их с остальными частями проекта. После добавления компонента `Navigation` и плагина `Safe Args` необходимо создать новый фрагмент, к которому будет переходить приложение.

## Создание EditTaskFragment

Мы создадим новый фрагмент с именем `EditTaskFragment`, к которому будет переходить приложение по щелчку на элементе `RecyclerView`. Выделите пакет `com.hfad.tasks` в папке `app/src/main/java` и выберите команду `File→New→Fragment→Fragment (Blank)`. Введите имя фрагмента «EditTaskFragment», затем имя его макета «fragment_edit_task» и убедитесь в том, что выбран язык Kotlin. Мы займемся обновлением кода `EditTaskFragment` и его макета через несколько страниц. А пока создадим граф навигации, который сообщает, как должны происходить переходы между фрагментами приложения.

Граф навигации добавляется в проект по той же схеме, как в других, созданных ранее приложениях. Выделите папку `Tasks/app/src/main/res` на панели проекта, затем выберите команду `File→New→Android Resource File`. Введите имя файла «nav_graph», выберите тип ресурса «Navigation» и щелкните на кнопке OK. IDE создает граф навигации с именем `nav_graph.xml`. Граф навигации должен описывать, как пользователь переходит между `TasksFragment` и `EditTaskFragment`. Схема навигации в нашем приложении выглядит так:

1. Приложение отображает `TasksFragment`. Это первый фрагмент, который видит пользователь, поэтому он становится стартовой целью в графе навигации.

2. Когда пользователь щелкает на элементе в представлении с переработкой `TasksFragment`, приложение переходит к `EditTaskFragment`. `TasksFragment` передает `EditTaskFragment` параметр типа Long, содержащий идентификатор задачи для элемента, на котором щелкнул пользователь.

3. Когда пользователь щелкает на кнопке в `EditTaskFragment` (мы добавим ее во фрагмент позднее в этой главе), приложение возвращается к `TasksFragment`.

## Обновление графа навигации

Ниже приведен полный код графа навигации; обновите файл `nav_graph.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation 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/nav_graph"
 app:startDestination="@id/tasksFragment">
```

```
    <fragment
    android:id="@+id/tasksFragment"
    android:name="com.hfad.tasks.TasksFragment"
    android:label="fragment_tasks"
    tools:layout="@layout/fragment_tasks" >
    <action
    android:id="@+id/action_tasksFragment_to_editTaskFragment"
    app:destination="@id/editTaskFragment" />
    </fragment>
    <fragment
    android:id="@+id/editTaskFragment"
    android:name="com.hfad.tasks.EditTaskFragment"
    android:label="fragment_edit_task"
    tools:layout="@layout/fragment_edit_task" >
    <argument
    android:name="taskId"
    app:argType="long" />
    <action
    android:id="@+id/action_editTaskFragment_to_tasksFragment"
    app:destination="@id/tasksFragment"
    app:popUpTo="@id/tasksFragment"
    app:popUpToInclusive="true" />
    </fragment>
  </navigation>
```

На следующем шаге нужно связать граф навигации с `MainActivity`, чтобы активность отображала каждый фрагмент при переходе к нему

## Добавление `NavHostFragment` в макет `MainActivity`

Чтобы связать только что созданный граф навигации с `MainActivity`, необходимо добавить хост навигации в макет и отдать ему команду использовать nav_graph.xml в качестве графа навигации. Это позволит `MainActivity` отображать правильные фрагменты при перемещении пользователя по приложению. Хост навигации добавляется в макет тем же способом, который применялся в предыдущих главах: добавлением `NavHostFragment` в представление `FragmentContainerView` файла `activity_main.xml`. Обновите файл `activity_main.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
 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/nav_host_fragment"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:padding="16dp"

 android:name="androidx.navigation.fragment.NavHostFragment"
```

```
    app:navGraph="@navigation/nav_graph"
    app:defaultNavHost="true"
    tools:context=".MainActivity" />
```

И это весь код, который необходимо изменить в макете `MainActivity`. Теперь нужно сделать так, чтобы фрагмент `TasksFragment` переходил к `EditTaskFragment`, когда пользователь щелкает на одном из элементов в его представлении с переработкой.

## Переход от TasksFragment к EditTaskFragment

Каждый раз, когда пользователь щелкает на элементе `RecyclerView` фрагмента `TasksFragment`, приложение должно перейти к `EditTaskFragment` и передать идентификатор задачи, на которой щелкнул пользователь. Одно из возможных решений — обновление лямбда-выражения, которое `TasksFragment` передает своему адаптеру `TaskItemAdapter`, чтобы оно включало весь необходимый код навигации:

```
override fun onCreateView(
 inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
 ...
 val adapter = TaskItemAdapter { taskId ->
 val action = TasksFragmentDirections
 .actionTasksFragmentToEditTaskFragment(taskId)
 this.findNavController().navigate(action)
 }
 binding.tasksList.adapter = adapter
 ...
}
```

Как видите, класс `TasksFragmentDirections` (сгенерированный плагином `Safe Args`) используется для передачи идентификатора задачи элемента фрагменту `EditTaskFragment` и перехода к этому фрагменту. Однако такой подход означает, что бизнес-логика (принятие решения о том, когда фрагмент `TasksFragment` должен перейти к `EditTaskFragment`) размещается в коде фрагмента — вместо добавления ее к `TasksViewModel`. Как вы узнали, подобные решения должны приниматься в коде модели представления, а не в коде фрагмента. Для решения этой проблемы мы будем использовать подход, аналогичный тому, который использовался в приложении `Guessing Game`. Мы добавим в `TasksViewModel` новое свойство `Live Data`. В нем будет храниться идентификатор задачи, на которой щелкнул пользователь. Когда значение этого свойства изменяется, `TasksFragment` реагирует переходом к `EditTaskFragment` с передачей идентификатора.

## Добавление нового свойства в TasksViewModel

Начнем с добавления в `TasksViewModel` нового свойства `Live Data` для идентификатора задачи, который должен передаваться из `TasksFragment` фрагменту `EditTaskFragment`. Назовем это свойство `navigateToTask`; оно будет определяться следующим кодом (он будет добавлен в `TasksViewModel.kt` на следующей странице):

```
class TasksViewModel(val dao: TaskDao) : ViewModel() {
 ...
 private val _navigateToTask = MutableLiveData<Long?>()
 val navigateToTask: LiveData<Long?>
 get() = _navigateToTask
 ...
}
```

Как видно из листинга, свойство `navigateToTask` использует изменяемое резервное свойство с модификатором `private`; это означает, что значение этого свойства может задать только `TasksViewModel`. Тем самым свойство защищается от нежелательных обновлений со стороны других классов.

## Добавление методов для обновления нового свойства

Каждый раз, когда пользователь щелкает на задаче в представлении с переработкой, фрагмент `TasksFragment` должен перейти к фрагменту `EditTaskFragment` и передать ему идентификатор задачи. Для этого мы добавим в `TasksViewModel` два метода — `onTaskClicked()` и `onTaskNavigated()`, — которые будут использоваться для присваивания значения резервного свойства `navigateToTask`. Метод `onTaskClicked()` присваивает свойству идентификатор задачи, а метод `onTaskNavigated()` возвращает ему значение null.

Код этих двух методов:

```
fun onTaskClicked(taskId: Long) {
 _navigateToTask.value = taskId
}
fun onTaskNavigated() {
 _navigateToTask.value = null
}
```

И это все изменения, которые необходимо внести в `TasksViewModel`. Полный код будет приведен на следующей странице.

## Полный код TasksViewModel.kt

Ниже приведен обновленный код `TasksViewModel`; убедитесь в том, что код `TasksViewModel.kt` содержит все изменения:

```
package com.hfad.tasks
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch
class TasksViewModel(val dao: TaskDao) : ViewModel() {
 var newTaskName = ""
```

```
    val tasks = dao.getAll()
    private val _navigateToTask = MutableLiveData<Long?>()
    val navigateToTask: LiveData<Long?>
    get() = _navigateToTask
    fun addTask() {
    viewModelScope.launch {
    val task = Task()
    task.taskName = newTaskName
    dao.insert(task)
    }
    }
    fun onTaskClicked(taskId: Long) {
    _navigateToTask.value = taskId
    }
    fun onTaskNavigated() {
    _navigateToTask.value = null
    }
   }
```

Код `TasksViewModel` обновлен. Давайте посмотрим, какие изменения необходимо внести в код `TasksFragment`.

## Переход от TasksFragment к EditTaskFragment

Необходимо обновить код `TasksFragment`, чтобы по щелчку на задаче происходил переход к `EditTaskFragment` с передачей идентификатора задачи. Для этого мы вызовем метод `onTaskClicked()` объекта `TasksViewModel`, когда пользователь щелкает на задаче, и выполним переход к `EditTaskFragment`, когда свойство `navigateToTask` обновляется новым идентификатором задачи.

## Вызов onTaskClicked() по щелчку на задаче

Чтобы вызвать метод `onTaskClicked()`, мы передадим приведенное ниже лямбда-выражение конструктору `TaskItemAdapter`:

```
val adapter = TaskItemAdapter { taskId ->
 viewModel.onTaskClicked(taskId)
}
binding.tasksList.adapter = adapter
```

Каждый раз, когда пользователь щелкает на задаче, выполняется лямбда-выражение; оно вызывает метод `onTaskClicked()` объекта `TasksViewModel`, присваивая `navigateToTask` идентификатор задачи.

## Переход к EditTaskFragment при обновлении navigateToTask

Чтобы фрагмент `TasksFragment` переходил к `EditTaskFragment`, мы отдадим ему команду наблюдать за свойством `navigateToTask` объекта `TaskViewModel`. Когда этому свойству присваивается идентификатор задачи (Long), фрагмент переходит к `EditTaskFragment` и передает ему идентификатор.

После этого свойству navigateToTask снова возвращается значение null вызовом метода `onTaskNavigated()` модели представления. Для этого используется следующий код:

```
viewModel.navigateToTask.observe(viewLifecycleOwner, Observer { taskId ->
 taskId?.let {
 val action = TasksFragmentDirections
 .actionTasksFragmentToEditTaskFragment(taskId)
 this.findNavController().navigate(action)
 viewModel.onTaskNavigated()
 }
})
```

## Полный код TasksFragment.kt

Ниже приведен обновленный код `TasksFragment`; убедитесь в том, что он включает все изменения:

```
package com.hfad.tasks
import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

import androidx.lifecycle.Observer
import androidx.lifecycle.ViewModelProvider
import androidx.navigation.fragment.findNavController
import com.hfad.tasks.databinding.FragmentTasksBinding
class TasksFragment : Fragment() {
 private var _binding: FragmentTasksBinding? = null
 private val binding get() = _binding!!
 override fun onCreateView(
 inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
 ): View? {
 _binding = FragmentTasksBinding.inflate(inflater, container, false)
 val view = binding.root
 val application = requireNotNull(this.activity).application
 val dao = TaskDatabase.getInstance(application).taskDao
 val viewModelFactory = TasksViewModelFactory(dao)
 val viewModel = ViewModelProvider(
 this, viewModelFactory).get(TasksViewModel::class.java)
 binding.viewModel = viewModel
 binding.lifecycleOwner = viewLifecycleOwner

 val adapter = TaskItemAdapter { taskId ->

 viewModel.onTaskClicked(taskId)
 }
 binding.tasksList.adapter = adapter
 viewModel.tasks.observe(viewLifecycleOwner, Observer {
 it?.let {
```

```
 adapter.submitList(it)
 }
 })
 viewModel.navigateToTask.observe(viewLifecycleOwner, Observer { taskId ->
 taskId?.let {
 val action = TasksFragmentDirections
 .actionTasksFragmentToEditTaskFragment(taskId)
 this.findNavController().navigate(action)
 viewModel.onTaskNavigated()
 }
 })
 return view
 }
 override fun onDestroyView() {
 super.onDestroyView()
 _binding = null
 }
}
```

Мы обновили код `TasksViewModel` и `TasksFragment`, чтобы по щелчку на задаче в представлении с переработкой фрагмент `TasksFragment` переходил к `EditTaskFragment` и передавал ему идентификатор задачи. Следующее, что необходимо сделать, — вывести идентификатор задачи в макете `EditTaskFragment`.

## Вывод идентификатора задачи в EditTaskFragment

Чтобы в `EditTaskFragment` выводился идентификатор задачи, необходимо обновить макет фрагмента и код Kotlin. Мы добавим в макет текстовое представление, а затем воспользуемся кодом Kotlin для получения идентификатора задачи и включения его в текстовое представление. Начнем с включения текстового представления в макет фрагмента; обновите файл `fragment_edit_task.xml` и приведите его к следующему виду:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:orientation="vertical"
 android:padding="16dp"
 tools:context=".EditTaskFragment">
 <TextView
 android:id="@+id/task_id"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:textSize="16sp" />
</LinearLayout>
```

Также необходимо обновить `EditTaskFragment.kt` После того как текстовое представление будет добавлено в макет `EditTaskFragment`, необходимо заполнить его идентификатором задачи. Для этого в метод `onCreateView()` добавляется следующий код:

```
val textView = view.findViewById<TextView>(R.id.task_id)
val taskId = EditTaskFragmentArgs.fromBundle(requireArguments()).taskId
textView.text = taskId.toString()
```

Как видите, в этом решении используется класс `EditTaskFragmentArgs` (сгенерированный плагином `Safe Args`) для получения значения аргумента `taskId`, переданный `EditTaskFragment`. Затем полученное значение используется для задания текста текстового представления. Посмотрим, как выглядит полный код `EditTaskFragment`.

## Полный код EditTaskFragment.kt

Ниже приведен полный код `EditTaskFragment`; внесите изменения в файл `EditTaskFragment.kt` и приведите его к следующему виду:

```
package com.hfad.tasks
import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
class EditTaskFragment : Fragment() {
 override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
 savedInstanceState: Bundle?): View? {
 val view = inflater.inflate(R.layout.fragment_edit_task, container, false)
 val textView = view.findViewById<TextView>(R.id.task_id)
 val taskId = EditTaskFragmentArgs.fromBundle(requireArguments()).taskId
 textView.text = taskId.toString()
 return view
 }
}
```

## Что происходит при выполнении кода

При выполнении кода происходят следующие события:

1. `TasksFragment` создает объект `TaskItemAdapter` и назначает его адаптером `RecyclerView`. Фрагмент передает адаптеру лямбда-выражение `clickListener`, которое вызывает метод `onTaskClicked()` объекта `TasksViewModel` при его выполнении.

2. `TasksFragment` передает `List<Task>` объекту `TaskItemAdapter`. `List<Task>` содержит актуальный список записей из базы данных.

3. `TaskItemAdapter` создает набор объектов `TaskItemViewHolder`, и для корневого представления каждого держателя представления назначается `OnClickListener`. В данном примере корневым представлением является `CardView`.

4. Когда пользователь щелкает на задаче в представлении с переработкой, `OnClickListener` регистрирует щелчок и выполняет лямбда-выражение. Он вызывает метод `onTaskClicked()` объекта `TasksViewModel`, который присваивает свойству `_navigateToTask` идентификатор задачи, на которой был сделан щелчок.

5. `TasksFragment` оповещается об обновлении свойства `navigateToTask` объекта `TasksViewModel`, которое использует `_navigateToTask` в качестве резервного свойства. Фрагмент переходит к `EditTaskFragment` и передает ему значение свойства `navigateToTask`.

6. Фрагмент `EditTaskFragment` получает переданное ему значение `taskId` и выводит его в макете.

При запуске приложения `TasksFragment` отображает сетку карточек в представлении с переработкой, как и в предыдущей версии. Если щелкнуть на одной из задач, приложение переходит к фрагменту `EditTaskFragment`, который отображает идентификатор задачи.

Вы научились использовать `RecyclerView` для перехода к новому фрагменту и передачи идентификатора задачи, на которой был сделан щелчок. Далее мы обновим приложение `Tasks`, чтобы по щелчку на элементе фрагмент `EditTaskFragment` отображал полную информацию о задаче и предоставлял возможность обновления или удаления записи из базы данных.

## Использование EditTaskFragment для обновления записей

Мы обновили приложение `Tasks`, чтобы когда пользователь щелкает на задаче в представлении с переработкой, оно переходило к фрагменту `EditTaskFragment`, в котором выводится идентификатор задачи. Но на самом деле в `EditTaskFragment` должна выводиться полная запись задачи, а пользователь должен иметь возможность обновления или удаления задачи из базы данных. Для этого мы обновим код `EditTaskFragment`, который должен выглядеть так:

1. Когда пользователь щелкает на задаче в представлении с переработкой, `EditTaskFragment` выводит расширенную информацию о ней. Для этого фрагмент читает запись задачи из базы данных, выводит имя задачи и признак ее завершения.

2. Когда пользователь обновляет описание задачи и щелкает на кнопке `Update Task`, изменения сохраняются в базе. Информация обновляется в базе данных, после чего происходит возврат к `TasksFragment`.

3. Если пользователь щелкнет на кнопке `Delete Task`, задача удаляется. Запись задачи удаляется из базы данных, после чего происходит переход к `TasksFragment`.

Для каждого из этих действий фрагмент должен взаимодействовать с базой данных `Room`; это означает, что он должен использовать интерфейс `TaskDao`. Но прежде чем обновлять приложение, стоит припомнить, что делает интерфейс `TaskDao`.

## Использование TaskDao для взаимодействия с записями базы данных

Как вы узнали, взаимодействие с записями, хранящимися в базе данных Room, осуществляется через интерфейс DAO. Например, приложение Tasks включает объект DAO с именем TaskDao, обеспечивающий взаимодействие с записями задач. Напомним, как выглядит код TaskDao:

```
@Dao
interface TaskDao {
 @Insert
 suspend fun insert(task: Task)
 @Update
 suspend fun update(task: Task)
 @Delete
 suspend fun delete(task: Task)
 @Query("SELECT * FROM task_table WHERE taskId = :key")
 fun get(key: Long): LiveData<Task>
 @Query("SELECT * FROM task_table ORDER BY taskId DESC")
 fun getAll(): LiveData<List<Task>>
}
```

Как видите, интерфейс включает методы для загрузки одной или нескольких записей из базы данных и сопрограммы для вставки, обновления и удаления записей в фоновом потоке.

## Создание модели представления для обращения к методам TaskDao

Мы воспользуемся методами TaskDao, чтобы фрагмент EditTaskFragment мог прочитать запись задачи из базы данных, обновить ее описание или удалить ее. Вместо того чтобы добавлять этот код в EditTaskFragment, мы создадим новую модель представления (с именем EditTaskViewModel), которая управляет бизнес-логикой и данными фрагмента. EditTaskViewModel будет вызывать методы TaskDao и передавать результаты EditTaskFragment. Перейдем к созданию EditTaskViewModel.

## Создание EditTaskViewModel

Чтобы создать класс EditTaskViewModel, выделите пакет com.hfad.tasks в папке app/src/main/java, затем выберите команду File→New→Kotlin Class/File. Введите имя файла «EditTaskViewModel» и выберите вариант Class.

Прежде всего объект EditTaskViewModel должен прочитать запись задачи из базы данных приложения, чтобы отобразить ее в макете EditTaskFragment. Для этого конструктору модели представления будут передаваться два значения: идентификатор задачи, который указывает, какую задачу нужно получить, и объект TaskDao, который будет использоваться для взаимодействия с базой данных. Также в модель представления будет добавлено свойство LiveData<Task> (с именем task), значение которого будет задаваться методом get() объекта TaskDao. Свойству присваивается запись задачи, которую хочет просмотреть пользователь. Ниже приведен код, который делает все это; полный код EditTaskViewModel будет приведен через пару страниц:

```
class EditTaskViewModel(taskId: Long, val dao: TaskDao) : ViewModel() {
 val task = dao.get(taskId)
}
```

Мы также добавим в `EditTaskViewModel` методы `updateTask()` и `deleteTask()`, которые будут использоваться `EditTaskFragment` для обновления или удаления записей задач. Эти методы будут вызывать сопрограммы `update()` и `delete()` объекта `TaskDao`:

```
fun updateTask() {
 viewModelScope.launch {
 dao.update(task.value!!)
 }
}
fun deleteTask() {
 viewModelScope.launch {
 dao.delete(task.value!!)
 }
}
```

Прежде чем добавлять этот код в `EditTaskViewModel`, посмотрим, что еще должен делать код модели представления.

## EditTaskViewModel сообщает EditTaskFragment, когда выполнить переход

Последнее, что должен сделать объект `EditTaskViewModel`, — сообщить фрагменту `EditTaskFragment`, когда ему следует вернуться к `TasksFragment`. Для этого мы добавим в модель представления новое свойство `LiveData<Boolean>` (с именем `navigateToList`) наряду с резервным свойством с именем `_navigateToList`. `EditTaskFragment` будет наблюдать за свойством `navigateToList`, и когда свойство принимает значение `true`, происходит переход к `TasksFragment`.

Значение `true` будет присваиваться `_navigateToList` в методах `updateTask()` и `deleteTask()` модели представления. Это означает, что сразу же после обновления или удаления записи задачи приложение переходит к `TasksFragment`. Ниже приведен обновленный код этих методов:

```
fun updateTask() {
 viewModelScope.launch {
 dao.update(task.value!!)
 _navigateToList.value = true
 }
}
fun deleteTask() {
 viewModelScope.launch {
 dao.delete(task.value!!)
 _navigateToList.value = true
 }
}
```

Также в `EditTaskViewModel` будет добавлен метод с именем `onNavigatedToList()`, который возвращает `_navigateToList` значение `false`. Код этого метода выглядит так:

```
fun onNavigatedToList() {
 _navigateToList.value = false
}
```

Полный код `EditTaskViewModel` приведен на следующей странице.

# Полный код EditTaskViewModel.kt

Ниже приведен полный код `EditTaskViewModel`; убедитесь в том, что код `EditTaskViewModel.kt` содержит все изменения:

```
package com.hfad.tasks
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch
class EditTaskViewModel(taskId: Long, val dao: TaskDao) : ViewModel() {
 val task = dao.get(taskId)
 private val _navigateToList = MutableLiveData<Boolean>(false)
 val navigateToList: LiveData<Boolean>
 get() = _navigateToList
 fun updateTask() {
 viewModelScope.launch {
 dao.update(task.value!!)
 _navigateToList.value = true
 }
 }
 fun deleteTask() {
 viewModelScope.launch {
 dao.delete(task.value!!)
 _navigateToList.value = true
 }
 }
 fun onNavigatedToList() {
 _navigateToList.value = false
 }
}
```

## EditTaskViewModel необходима фабрика модели представления

Следующее, что необходимо сделать,— определить фабрику модели представления с именем `EditTaskViewModelFactory`, которая используется `EditTaskFragment` для создания экземпляра `EditTaskViewModel`. Как вы узнали, фабрика модели представления необходима для всех моделей представлений, которые, как и `EditTaskViewModel`, не содержат конструктор без аргументов.

## Создание EditTaskViewModelFactory

Чтобы создать фабрику, выделите пакет `com.hfad.tasks` в папке `app/src/main/java` и выберите комаду `File→New→Kotlin Class/File`. Введите имя файла «EditTaskViewModelFactory» и выберите вариант `Class`. Когда файл будет создан, обновите код в файле `EditTaskViewModelFactory.kt`, чтобы он выглядел так:

```
package com.hfad.tasks
import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider
class EditTaskViewModelFactory(private val taskId: Long,
 private val dao: TaskDao)
 : ViewModelProvider.Factory {
 override fun <T : ViewModel?> create(modelClass: Class<T>): T {
 if (modelClass.isAssignableFrom(EditTaskViewModel::class.java)) {
 return EditTaskViewModel(taskId, dao) as T
 }
 throw IllegalArgumentException("Unknown ViewModel")
 }
}
```

После написания кода класса `EditTaskViewModel` и его фабрики нужно обновить код фрагмента `EditTaskFragment` и его макета. Начнем с макета.

## В fragment_edit_task.xml должны отображаться данные задачи

Обновим файл `fragment_edit_task.xml`, чтобы он включал текстовое поле и флажок, в котором будут отображаться данные `Task`. Мы воспользуемся механизмом связывания данных для связывания этих представлений со свойствами `taskName` и `taskDone` в задаче `EditTaskViewModel`. Также в макет будут добавлены две кнопки, которые будут вызывать методы `deleteTask()` и `updateTask()` объекта `EditTaskViewModel` и позволят пользователю обновить или удалить запись задачи.

Ниже приведена новая версия кода макета; обновите файл `fragment_edit_task.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<layout
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"
 tools:context=".EditTaskFragment">
 <data>
 <variable
 name="viewModel"
 type="com.hfad.tasks.EditTaskViewModel" />
 </data>

<LinearLayout

 xmlns:tools="http://schemas.android.com/tools"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:orientation="vertical"
```

```
    android:padding="16dp"

    <EditText

    android:id="@+id/task_name"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:inputType="text"
    android:text="@={viewModel.task.taskName}" />

    <CheckBox
    android:id="@+id/task_done"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:checked="@={viewModel.task.taskDone}" />
    <Button
    android:id="@+id/update_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="Update Task"
    android:onClick="@{() -> viewModel.updateTask()}" />
    <Button
    android:id="@+id/delete_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="Delete Task"
    android:onClick="@{() -> viewModel.deleteTask()}" />
    </LinearLayout>
  </layout>
```

## Обновление EditTaskFragment.kt

Осталось внести последнее изменение в приложение `Tasks` — обновить код Kotlin объекта `EditTaskFragment`. Этот код должен:

1. Задать значение переменной связывания данных `viewModel` макета. Ей необходимо присвоить экземпляр `EditTaskViewModel`, который будет создаваться фрагментом.

2. Назначить владельца жизненного цикла макета. Это необходимо для того, чтобы макет мог взаимодействовать со свойствами `Live Data`.

3. Наблюдать за свойством `navigateToList` модели представления. Когда свойство принимает значение `true`, фрагмент переходит к `TasksFragment` и вызывает метод `onNavigatedToList()` объекта `EditTaskViewModel`.

Код, который все это делает, вам уже знаком, поэтому мы приведем обновленный код `EditTaskFragment` на следующей странице.

## Полный код EditTaskFragment.kt

Ниже приведен полный код `EditTaskFragment`: убедитесь в том, что файл `EditTaskFragment.kt` содержит все изменения:

```
package com.hfad.tasks
import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

import androidx.lifecycle.Observer
import androidx.lifecycle.ViewModelProvider
import androidx.navigation.findNavController
import com.hfad.tasks.databinding.FragmentEditTaskBinding
class EditTaskFragment : Fragment() {
 private var _binding: FragmentEditTaskBinding? = null
 private val binding get() = _binding!!
 override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
 savedInstanceState: Bundle?): View? {

 _binding = FragmentEditTaskBinding.inflate(inflater, container, false)
 val view = binding.root
 val taskId = EditTaskFragmentArgs.fromBundle(requireArguments()).taskId

 val application = requireNotNull(this.activity).application
 val dao = TaskDatabase.getInstance(application).taskDao
 val viewModelFactory = EditTaskViewModelFactory(taskId, dao)
 val viewModel = ViewModelProvider(this, viewModelFactory)
 .get(EditTaskViewModel::class.java)
 binding.viewModel = viewModel
 binding.lifecycleOwner = viewLifecycleOwner
 viewModel.navigateToList.observe(viewLifecycleOwner, Observer { navigate ->
 if (navigate) {
 view.findNavController()
 .navigate(R.id.action_editTaskFragment_to_tasksFragment)
 viewModel.onNavigatedToList()
 }
 })
 return view
 }
 override fun onDestroyView() {
 super.onDestroyView()
 _binding = null
 }
}
```

И это все, что необходимо для отображения данных `Task` в макете, с возможностью обновления или удаления записи. Давайте разберемся, что происходит в коде во время его выполнения.

## Что происходит при выполнении кода

При выполнении приложения происходят следующие события:

1. Когда пользователь щелкает на задаче, `TasksFragment` переходит к `EditTaskFragment` и передает идентификатор задачи.
2. `EditTaskFragment` получает ссылку на объект `EditTaskViewModel`.
3. `EditTaskViewModel` вызывает метод `get()` объекта `TaskDao`, передавая ему идентификатор задачи.
4. Метод `get()` объекта `TaskDao` возвращает значение `LiveData<Task>`, которое присваивается свойству `task` объекта `EditTaskViewModel`.

5a. Когда пользователь щелкает на кнопке `Update Task`, кнопка вызывает метод `updateTask()` объекта `EditTaskViewModel`. Этот метод использует метод `update()` объекта `TaskDao` для обновления записи в базе данных.

5b. Когда пользователь щелкает на кнопке `Delete Task`, вызывается метод `deleteTask()` объекта `EditTaskViewModel`. Этот метод использует метод `delete()` объекта `TaskDao` для удаления записи.

6. Приложение переходит к `TasksFragment`. Любые изменения, внесенные в запись задачи, отражаются в представлении с переработкой.

При запуске приложения `TasksFragment` отображает сетку карточек в представлении с переработкой, как и прежде. Если щелкнуть на одной из задач, приложение переходит к фрагменту `EditTaskFragment`, который отображает данные записи. Если внести изменения в задачу и щелкнуть на кнопке `Update Task`, изменения сохраняются в базе данных и отображаются в представлении с переработкой `TasksFragment`.

Когда вы щелкаете на задаче, приложение переходит к `EditTaskFragment` и отображает описание задачи, как и прежде. Если щелкнуть на кнопке `Delete Task`, запись удаляется из базы данных. Когда приложение переходит к `TaskFragment`, запись исчезает из списка.

Поздравляем! Вы научились строить приложения, использующие `RecyclerView`, для перехода между записями, с возможностью обновления и удаления записей. Это позволяет вам предоставить пользователю мощный и гибкий механизм работы с данными в приложении.

# Резюме

- Элементы представлений с переработкой могут реагировать на щелчки. В частности, эта возможность может использоваться для перехода к другому фрагменту с расширенной информацией об элементе, на котором был сделан щелчок.
- Чтобы `RecyclerView` реагировало на щелчки, следует добавить слушатель `OnClickListener` для корневого представления макета, используемого его элементами.
- Чтобы сообщить слушателю `OnClickListener`, как следует реагировать на щелчки, передайте ему лямбда-выражение. Это лямбда-выражение будет выполняться каждый раз, когда `OnClickListener` регистрирует щелчок.

- Лямбда-выражение может определяться в коде фрагмента и передаваться `OnClickListener` через конструктор адаптера.