Практическая работа 1. Recyclerview

Списки данных являются важнейшей частью многих приложений. В этой главе мы покажем, как создать такой список с использованием представлений с переработкой: невероятно гибкого способа построения прокручиваемых списков. Вы научитесь создавать гибкие макеты для ваших списков, включая текстовые представления, флажки и многое другое. Вы узнаете, как создавать адаптеры, которые отображают ваши данные в представлениях с переработкой так, как вам нужно. Вы научитесь использовать карточные представления для оформления данных с имитацией объема. Наконец, мы покажем, как менеджеры макетов могут полностью изменить внешний вид вашего списка всего одной или двумя строками кода. Давайте займемся переработкой.

Как в настоящее время выглядит приложение Tasks

Мы построили приложение `Tasks`, которое предоставляет возможность вводить записи в базе данных Room. Приложение выводит список записей в виде отформатированной строки, которая выглядит так:

Мы решили выводить записи с описаниями задач в отформатированной строке, потому что это относительно простой и быстрый способ просмотра записей, добавленных в базу данных. Тем не менее список выглядит довольно уныло. Нельзя ли как-то украсить его?

Список можно преобразовать в представление с переработкой

Вместо того чтобы отображать список задач в виде отформатированной строки, его можно изменить, чтобы он выглядел примерно так:

Как видите, для данных каждой записи отображается текстовое представление и флажок (вместо простого текста). Элементы на карточках образуют сетку с возможностью прокрутки. Для создания подобных списков используется представление с переработкой. Что это такое?

Для чего нужны представления с переработкой?

Представления с переработкой предоставляют более совершенный и гибкий способ отображения списка данных по сравнению с использованием простой отформатированной строки. Они обладают целым рядом преимуществ:

- Более функциональный пользовательский интерфейс для элементов списка. Каждый элемент отображается в макете, что позволяет использовать для вывода данных такие компоненты, как текстовые представления, графические представления и флажки.
- Гибкие средства позиционирования элементов. Представления с переработкой работают в сочетании с менеджерами макетов, которые позволяют позиционировать представления в вертикальном или горизонтальном списке, сетке или неравномерной сетке, у которой элементы имеют разную высоту.
- Возможность использования для навигации. Для элементов можно предусмотреть обработку щелчков, чтобы по щелчку происходил переход к другому фрагменту.

- Эффективный механизм отображения больших наборов данных. Представления с переработкой используют небольшое количество представлений для создания иллюзии большой группы представлений, выходящих за пределы экрана. Когда каждый элемент выходит за пределы экрана, его представление повторно используется — или перерабатывается — для элементов, которые вошли на экран в результате прокрутки

Представления с переработкой получают свои данные от адаптера

Каждое представление с переработкой, которое вы создаете, использует адаптер для отображения своих данных. Адаптер использует данные из источника данных (такого, как база данных) и связывает их с представлениями в макете элемента. После этого представление с переработкой отображает элементы на экране в виде списка с возможностью прокрутки. Схема взаимодействия источника данных, адаптера и представления с переработкой выглядит так:

Мы добавим представление с переработкой в приложение Tasks. Давайте рассмотрим основные этапы решения этой задачи.

Что мы собираемся сделать

Чтобы добавить представление с переработкой в приложение Tasks, выполните следующие действия:

1. Создание представления с переработкой, в котором выводится список имен задач. Начнем с создания простого представления с переработкой, в котором выводятся только имена задач. С относительно простой первой версией вам будет проще понять, как конструируется каждая часть представления с переработкой и как они взаимодействуют друг с другом.

2. Обновление представления с переработкой для отображения карточек, образующих сетку. После того как простое представление с переработкой заработает, мы изменим его так, чтобы имя каждой задачи и признак ее завершения отображались в карточном представлении, а карточные представления формировали сетку.

Добавление зависимости для представления с переработкой в файл build.gradle приложения

Синхронизируйте это изменение с остальными частями приложения. После того как зависимость будет добавлена, перейдем к построению представления с переработкой. Прежде чем строить представление с переработкой, необходимо добавить зависимость для библиотеки представлений с переработкой в файл `build.gradle` приложения. Откройте приложение Tasks, затем откройте файл `Tasks/app/build.gradle` и добавьте следующую строку в раздел `dependencies`:

```
dependencies {
 ...
 implementation 'androidx.recyclerview:recyclerview:1.2.1'
 ...
}
```

Синхронизируйте это изменение с остальными частями приложения. После того как зависимость будет добавлена, перейдем к построению представления с переработкой.

Чтобы сообщить представлению с переработкой, как отображать каждый элемен. Прежде всего необходимо сообщить представлению с переработкой, как выводить каждую запись. В первой версии приложения имя каждой задачи должно выводиться в представлении с переработкой, чтобы оно выглядело так:

Для определения того, как должен быть оформлен каждый элемент в представлении с переработкой, используется файл макета. Представление с переработкой использует этот файл макета для отображения каждого элемента. Например, если файл макета состоит из единственного текстового представления, текстовое представление будет отображаться для каждого элемента в списке представления с переработкой. Чтобы создать файл макета, выделите папку `Tasks/app/src/main/res/layout` на панели проекта и выберите команду `File→New→Layout Resource File`. Введите имя файла «task_item» и щелкните на кнопке OK. В первой версии приложения имя каждой задачи представления с переработкой должно отображаться в одном текстовом представлении; добавим текстовое представление в только что созданное текстовое представление. Для этого обновите код `task_item.xml` и приведите его к следующему виду:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:textSize="16sp"
 android:padding="8dp" />
```

И это весь код, необходимый для того, чтобы сообщить представлению с переработкой, как должен отображаться каждый элемент. На следующем этапе мы создадим адаптер представления с переработкой.

Адаптер добавляет данные в представление с переработкой

Как говорилось ранее, когда вы используете представление с переработкой в своем приложении, для него необходимо создать адаптер. Адаптер представления с переработкой выполняет две основные функции: создание каждого из представлений, видимых в представлении с переработкой, и отображение данных в каждом представлении. Для приложения `Tasks` необходимо определить адаптер, который использует `task_item.xml` для создания набора текстовых представлений (по одному для каждой отображаемой записи) и помещает в каждое представление имя задачи. Адаптер будет построен на нескольких следующих страницах. Сначала выделим основные этапы его создания.

1. Определение типа данных, с которыми должен работать адаптер. Наш адаптер должен работать с данными Task, поэтому мы указываем, что он использует `List<Task>`.

2. Определение держателя представления для адаптера. Управляет тем, как должно заполняться каждое представление в макете элемента.

3. Заполнение макета каждого элемента. Когда представлению с переработкой потребуется вывести каждый элемент, мы заполняем экземпляр `task_item.xml` для этого элемента.

4. Отображение данных каждого элемента в макете. Для этого мы добавим значение свойства `taskName` каждого объекта `Task` в текстовое представление макета.

Начнем с создания файла для адаптера.

Создание файла адаптера

Мы создадим адаптер для представления с переработкой `TaskItemAdapter`. Для этого выделите пакет *com.hfad.tasks* в папке *app/src/main/java*, а затем выберите команду `File→New→Kotlin Class/File`. Введите имя файла «TaskItemAdapter» и выберите вариант `Class`. После того как файл будет создан, обновите его код, чтобы он расширял класс `RecyclerView.Adapter`:

```
package com.hfad.tasks
import androidx.recyclerview.widget.RecyclerView
class TaskItemAdapter : RecyclerView.Adapter() {
}
```

Класс преобразуется в адаптер, который может использоваться представлением с переработкой.

Сообщаем адаптеру, с какими данными он должен работать

Когда вы определяете адаптер представления с переработкой, необходимо сообщить ему, какие данные должны добавляться в представление с переработкой. Для этого в адаптер будет добавлено свойство, определяющее тип данных. В приложении `Tasks` представление с переработкой должно отображать список записей задач, поэтому мы добавим в адаптер свойство `List<Task>` с именем `data`. Также будет добавлен специальный метод записи, который вызывает `notifyDataSetChanged()` при обновлении свойства; тем самым он сообщает представлению с переработкой, что данные изменились, чтобы оно могло перерисовать себя. Ниже приведен обновленный код `TaskItemAdapter`; обновите файл `TaskItemAdapter.kt`:

```
class TaskItemAdapter : RecyclerView.Adapter() {
 var data = listOf<Task>()
 set(value) {
 field = value
 notifyDataSetChanged()
 }
}
```

Переопределение метода getItemCount()

Затем необходимо переопределить метод `getItemCount()` адаптера. Тем самым вы сообщаете адаптеру, сколько элементов данных будет отображаться, чтобы эта информация была доступна представлению с переработкой. В коде `TaskItemAdapter` мы используем свойство `List<Task>` с именем data для элементов данных представления с переработкой, что позволяет использовать конструкцию `data.size` для определения количества элементов. Ниже приведен метод `getItemCount()`, который добавляется в `TaskItemAdapter.kt`:

```
class TaskItemAdapter : RecyclerView.Adapter() {
 ...
 override fun getItemCount() = data.size
}
```

Итак, мы определили тип данных, с которыми работает адаптер. Теперь он будет использован для заполнения текстового представления макета. Для этого мы определим держатель представления адаптера.

Определение держателя представления для адаптера

Держатель представления (view holder) содержит информацию о том, как должно отображаться представление в макете элемента, и о его позиции в представлении с переработкой. Его можно рассматривать как держатель корневого представления макета элемента — макета, определяющего, как представление с переработкой должно отображать каждый элемент. В приложении `Tasks` представление с переработкой должно использовать файл макета `task_item.xml` для отображения записей с описаниями задач. Корневым представлением макета является `TextView`, а следовательно, необходимо определить держатель представления, который работает с текстовыми представлениями. Чтобы определить держатель представления, добавьте внутренний класс в класс адаптера, расширяющий `RecyclerView.ViewHolder`. Он включает конструктор, задающий тип корневого представления макета (в данном случае `TextView`). Определение класса адаптера также должно обновляться для определения имени класса адаптера. Ниже приведена новая версия кода `TaskItemAdapter`; обновите файл `TaskItemAdapter.kt`:

```
package com.hfad.tasks
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView
class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>()
{
 var data = listOf<Task>()
 set(value) {
 field = value
 notifyDataSetChanged()
 }
 override fun getItemCount() = data.size
 class TaskItemViewHolder(val rootView: TextView)
 : RecyclerView.ViewHolder(rootView) {
 }
}
```

После определения держателя представления необходимо указать, какой макет он использует, переопределяя метод `onCreateViewHolder()` адаптера.

Переопределение метода onCreateViewHolder()

Метод `onCreateViewHolder()` адаптера вызывается каждый раз, когда представлению с переработкой потребуется новый держатель представления. Представление с переработкой многократно вызывает метод в момент своего конструирования для построения набора держателей представлений, которые будут отображаться на экране. Метод `onCreateViewHolder()` должен решать две задачи: заполнять макет, используемый для каждого элемента (в данном случае `task_item.xml`), и возвращать держатель представления. Ниже приведен код для адаптера `TaskItemAdapter`; обновите файл `TaskItemAdapter.kt`:

```
import android.view.LayoutInflater
import android.view.ViewGroup
class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>()
{
 ...
 override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
 : TaskItemViewHolder = TaskItemViewHolder.inflateFrom(parent)
 class TaskItemViewHolder(val rootView: TextView)
 : RecyclerView.ViewHolder(rootView) {
 companion object {
 fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
 val layoutInflater = LayoutInflater.from(parent.context)
 val view = layoutInflater
 .inflate(R.layout.task_item, parent, false) as TextView
 return TaskItemViewHolder(view)
 }
 }
 }
}
```

Как видите, мы поместили код заполнения `task_item.xml` в новый метод `inflateFrom()` в `TaskItemViewHolder`, который вызывается методом `onCreateViewHolder()` адаптера:

```
TaskItemViewHolder.inflateFrom(parent)
```

Этот подход возлагает ответственность за макет держателя представления на сам держатель представления (вместо заполнения макета в основном блоке кода адаптера).

Добавление данных в представление макета

Остается добавить в адаптер код, определяющий, как записи должны отображаться в макете держателя представления. Для этого мы переопределим метод `onBindViewHolder()` адаптера, который вызывается каждый раз, когда представлению с переработкой потребуется отобразить данные. Он получает два параметра: держатель представления, с которым связываются данные, и позицию данных в наборе данных. В приложении `Tasks` требуется взять объект Task в определенной позиции свойства data адаптера (`List<Task>`) и отобразить его значение `taskName` в макете держателя представления. Ниже приведен код решения этой задачи для `TaskItemAdapter`; обновите файл `TaskItemAdapter.kt`:

```
class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>()
{
 var data = listOf<Task>()
 ...
 override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
 val item = data[position]
 holder.bind(item)
 }
 class TaskItemViewHolder(val rootView: TextView)
 : RecyclerView.ViewHolder(rootView) {
 ...
 fun bind(item: Task) {
 rootView.text = item.taskName
 }
 }
}
```

Как видите, текстовое представление макета задается в новом методе `bind()`, который добавляется в `TaskItemViewHolder`. Затем метод `onBindViewHolder()` адаптера вызывает его при каждом выполнении.

Мы используем этот подход, потому что с ним держатель представления отвечает за заполнение своего макета (вместо адаптера). И это весь код, необходимый для написания класса `TaskItemAdapter` и его внутреннего класса `TaskItemViewHolder`. Пора рассмотреть полный код.

Полный код TaskItemAdapter.kt

Ниже приведен полный код `TaskItemAdapter`; убедитесь в том, что в файл `TaskItemAdapter.kt` включены все показанные изменения:

```
package com.hfad.tasks
import android.view.LayoutInflater
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView
class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>()
{
 var data = listOf<Task>()
 set(value) {
 field = value
 notifyDataSetChanged()
 }
 override fun getItemCount() = data.size
 override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
 : TaskItemViewHolder = TaskItemViewHolder.inflateFrom(parent)
 override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
 val item = data[position]
 holder.bind(item)
 }
 class TaskItemViewHolder(val rootView: TextView)
```

```
    : RecyclerView.ViewHolder(rootView) {
    companion object {
    fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
    val layoutInflater = LayoutInflater.from(parent.context)
    val view = layoutInflater
    .inflate(R.layout.task_item, parent, false) as TextView
    return TaskItemViewHolder(view)
    }
    }
    fun bind(item: Task) {
    rootView.text = item.taskName
    }
    }
}
```

Код адаптера готов Мы написали весь код, необходимый для `TaskItemAdapter`. Этот код:

1. Сообщает, что работает с данными `Task` Для этого мы определили свойство `List<Task>` с именем `data`.
2. Использует держатель представлений с именем `TaskItemViewHolder`. Класс `TaskItemViewHolder` был включен в `TaskItemAdapter` как внутренний класс.
3. Заполняет макет каждого элемента. Это происходит при вызове метода `onCreateViewHolder()`.
4. Выводит данные каждого элемента в макете. Для этого используется метод `onBindViewHolder()`.

Как говорилось ранее, адаптер соединяет источник данных и представление с переработкой. Источник данных, адаптер и представление с переработкой взаимодействуют по следующей схеме:

Мы завершили работу над кодом адаптера. Пора переходить к другой части этой модели — представления с переработкой.

В приложении должно отображаться представление с переработкой

Следующее, что необходимо сделать, отобразить представление с переработкой в `TasksFragment` (главном экране приложения `Tasks`) и заставить его использовать только что созданный нами адаптер. Кратко напомним, как должно выглядеть представление с переработкой:

Добавление представления с переработкой в макет

Чтобы в приложении отображалось представление с переработкой, добавьте элемент `<androidx.recyclerview.widget.RecyclerView>` в файл макета фрагмента. Код должен выглядеть так:

```
<androidx.recyclerview.widget.RecyclerView
 android:id="@+id/tasks_list"
 android:layout_width="match_parent"
```

```
 android:layout_height="match_parent"
 app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager" />
```

Строка:

```
app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"
```

указывает, что в представлении с переработкой используется менеджер макета, который определяет, как представление с переработкой позиционирует свои элементы. В данном случае используется менеджер линейного макета; это означает, что элементы представления с переработкой будут отображаться в вертикальном списке со строками полной длины. И это весь код, который необходим для добавления представления с переработкой в макет `TasksFragment`. Давайте посмотрим, как он выглядит в целом.

Полный код fragment_tasks.xml

Ниже приведен полный код `fragment_tasks.xml` (макет `TasksFragment`). Как видите, мы заменили текстовое представление представлением с переработкой; обновите код `fragment_tasks.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<layout
 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"
 tools:context=".TasksFragment">
 <data>
 <variable
 name="viewModel"
 type="com.hfad.tasks.TasksViewModel" />
 </data>
 <LinearLayout
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:orientation="vertical">
 <EditText
 android:id="@+id/task_name"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:inputType="text"
 android:hint="Enter a task name"
 android:text="@={viewModel.newTaskName}" />
 <Button
 android:id="@+id/save_button"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_gravity="center"
 android:text="Save Task"
 android:onClick="@{() -> viewModel.addTask()}" />
```

```
<androidx.recyclerview.widget.RecyclerView
 android:id="@+id/tasks_list"
 android:layout_width="match_parent"
 android:layout_height="0dp"
 android:layout_weight="1"
 android:gravity="top"
 app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager" />
 </LinearLayout>
</layout>
```

И это весь код, необходимый для добавления представления с переработкой в макет `TasksFragment`. Затем нужно отдать команду представлению с переработкой использовать созданный нами адаптер.

Использование адаптера представлением с переработкой

Чтобы отдать команду представлению с переработкой использовать адаптер, следует создать экземпляр адаптера и присоединить его к представлению с переработкой. Это делается в коде Kotlin фрагмента. В нашем случае нужно отдать команду представлению с переработкой использовать `TaskItemAdapter`. Для этого следует включить в метод `onCreateView()` объекта `TasksFragment` следующую строку:

```
override fun onCreateView(...): View? {
 ...
 val adapter = TaskItemAdapter()
 binding.tasksList.adapter = adapter
 ...
}
```

Этот код будет добавлен в `TasksFragment` на следующей странице.

Обновленный код TasksFragment.kt

Ниже приведен код `TasksFragment`; обновите файл `TasksFragment.kt`:

```
package com.hfad.tasks
...
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()
    binding.tasksList.adapter = adapter
    return view
    }
    override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
    }
   }
```

Представление с переработкой добавляется в макет `TasksFragment`

Мы написали весь код, необходимый для отображения представления с переработкой в макете `TasksFragment`, и отдали ему команду использовать адаптер `TaskItemAdapter`. Остается сделать последний шаг: соединить адаптер с источником данных.

Как вы узнали ранее, адаптер использует данные из источника данных (например, базы данных) и связывает их с представлениями в макете элемента. Тогда представление с переработкой отображает элементы на экране устройства. Источник данных, адаптер и представление с переработкой взаимодействуют по следующей схеме:

Таким образом, чтобы отображать данные в представлении с переработкой приложения `Tasks`, необходимо сообщить объекту `TaskItemAdapter`, какие данные он должен использовать.

`TasksFragment` будет поставлять данные задач адаптеру `TaskItemAdapter` Чтобы сообщить `TaskItemAdapter`, какие данные задач следует использовать, мы заставим `TasksFragment` обновлять его свойство `data` с `List<Task>`. `TasksFragment` получает этот список из свойства `tasks` объекта `TasksViewModel`:

Для этого необходимо сначала предоставить `TasksFragment` доступ к свойству `tasks` объекта `TasksViewModel`.

Обновление кода TasksViewModel.kt

Как вы помните, свойство tasks объекта `TasksViewModel` в настоящее время помечено модификатором `private`. Этот модификатор необходимо удалить, чтобы код `TasksFragment` мог получить значение свойства. Также будет удален весь код, добавленный в предыдущей главе для преобразования данных задач в отформатированную строку: этот код стал лишним, так как мы используем представление с переработкой для отображения `List<Task>`. Ниже приведена новая версия кода `TasksViewModel`; обновите файл `TasksViewModel.kt`:

```
package com.hfad.tasks

import androidx.lifecycle.ViewModel
```

```
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch

class TasksViewModel(val dao: TaskDao) : ViewModel() {
 var newTaskName = ""
 val tasks = dao.getAll()

fun addTask() {
 viewModelScope.launch {
 val task = Task()
 task.taskName = newTaskName
 dao.insert(task)
 }
 }
 }
```

Фрагмент `TasksFragment` получил доступ к свойству tasks, теперь он должен передавать объект `List<Task>` из свойства адаптеру `TaskItemAdapter`.

Фрагмент TasksFragment должен обновлять свойство данных TaskItemAdapter

Как вы уже знаете, свойство tasks объекта `TasksViewModel` содержит список задач в виде данных `Live Data`, которые читаются из базы данных с помощью кода:

```
val tasks = dao.getAll()
```

Так как это свойство использует механизм `Live Data`, можно отдать команду `TasksFragment` наблюдать за ним, чтобы при каждом изменении его значения фрагмент получал оповещение. Тогда `TasksFragment` может присвоить новейшую версию списка свойству `data` адаптера, гарантируя, что отображаемые в представлении с переработкой данные всегда остаются актуальными. Код наблюдения за свойствами `Live Data` вам уже знаком, поэтому мы просто приведем код, который необходимо добавить в `TasksFragment`:

```
class TasksFragment : Fragment() {
 ...
 override fun onCreateView(
 inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
 ): View? {
 ...
 val adapter = TaskItemAdapter()
 binding.tasksList.adapter = adapter
 viewModel.tasks.observe(viewLifecycleOwner, Observer {
 it?.let {
 adapter.data = it
 }
 })
 ...
```

```
    }
    ...
}
```

Обновим файл `TasksFragment.kt` и включим в него это изменение.

Полный код TasksFragment.kt

Ниже приведена новая версия `TasksFragment`; обновите файл `TasksFragment.kt`:

```
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 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()
 binding.tasksList.adapter = adapter
 viewModel.tasks.observe(viewLifecycleOwner, Observer {
 it?.let {
 adapter.data = it
 }
 })
 return view
 }
 override fun onDestroyView() {
 super.onDestroyView()
 _binding = null
 }
}
```

Работа над кодом представления с переработкой завершена.

Это заняло некоторое время, но мы написали весь код, необходимый для отображения имен задач в представлении с переработкой. Основные этапы этой работы:

1. Создание адаптера с именем TaskItemAdapter. Адаптер соединяет представление с переработкой с его источником данных. В приложении Tasks источником данных является база данных Room, содержащая записи с задачами.

2. Присоединение TaskItemAdapter в представлении с переработкой. Мы добавили представление с переработкой в макет TasksFragment и отдали ему команду использовать TaskItemAdapter в коде Kotlin.

3. Передача актуального списка `List<Task>` адаптеру TaskItemAdapter. Для этого TasksFragment задает значение свойства data объекта TaskItemAdapter при каждом обновлении списка задач Live Data из объекта TasksViewModel.

Прежде чем мы проведем тест-драйв приложения и посмотрим, как выглядит представление с переработкой, разберемся, что же происходит при выполнении кода.

Что происходит при выполнении кода

При выполнении кода происходят следующие события:

1. При запуске приложения `MainActivity` отображает `TasksFragment`. `TasksFragment` использует `TasksViewModel` как свою модель представления.

2. `TasksFragment` создает объект `TaskItemAdapter` и присваивает его представлению с переработкой в качестве адаптера.

3. `TasksFragment` наблюдает за свойством `tasks` объекта `TasksViewModel`. Это свойство имеет тип `LiveData<List<Task>>` и содержит актуальный список записей из базы данных.

4. `TasksFragment` присваивает значение свойства `data` объекта `TaskItemAdapter` списку `List<Task>`.

5. Метод `onCreateViewHolder()` объекта `TaskItemAdapter` вызывается для каждого элемента, который должен отображаться в представлении с переработкой. При этом для каждого элемента создается объект `TaskItemViewHolder` (держатель представления). Для каждого держателя представления заполняется макет (определяемый файлом `task_item.xml`).

6. Метод `onBindViewHolder()` объекта `TaskItemAdapter` вызывается для каждого объекта `TaskItemViewHolder`. Данные связываются с текстовым представлением в макете каждого держателя представления.

7. При каждом обновлении свойства tasks объекта `TasksViewModel` фрагмент `TasksFragment` передает обновленный список `List<Task>` объекту `TaskItemAdapter`. Этапы 5 и 6 повторяются, чтобы представление с переработкой оставалось актуальным.

При запуске приложения `TasksFragment` отображает имена всех задач в представлении с переработкой. Если ввести новую задачу, она будет добавлена в список представления с переработкой. Приложение работает так, как было задумано.

Представления с переработкой чрезвычайно гибки

К настоящему моменту вы узнали, как построить простое представление с переработкой, которое отображает список имен задач. Для этого мы создали макет, который используется каждым элементом списка, определили адаптер для заполнения его данными и добавили представление с переработкой в макет `TaskFragment`.

В этом приложении мы создадим представление с переработкой для вывода простого списка имен задач, но это только начало. Представления с переработкой также могут использоваться для других целей, в том числе:

- Вывода списков изображений с включением графических представлений в макет элемента.
- Использования других менеджеров макетов для отображения элементов в виде сетки (вместо вертикального списка).
- Программирования реакции на щелчки, чтобы представление могло использоваться для навигации.

Чтобы показать, насколько гибкими могут быть представления с переработкой, мы изменим только что созданное представление с переработкой, чтобы в нем выводилась более подробная информация о каждой задаче. Посмотрим, как будет выглядеть новая версия представления с переработкой.

Представление с переработкой 2.0

Мы обновим представление с переработкой, чтобы имя каждой задачи отображалось в текстовом представлении, рядом с которым находится флажок — признак ее завершения. Данные записи каждой задачи отображаются на отдельной карточке, которая кажется слегка приподнятой. Карточки образуют сетку. Новая версия представления с переработкой должна выглядеть так:

Чтобы создать эту версию представления с переработкой, мы изменим файл `task_item.xml` (макет, используемый элементами представления с переработкой), чтобы в нем использовалось карточное представление. Это разновидность фреймового макета с закругленными углами и тенями, из-за которых он кажется слегка приподнятым над фоном.

Добавление зависимости для карточного представления в файл build.gradle приложения

Чтобы использовать карточное представление, прежде всего необходимо добавить зависимость для его библиотеки в файл `build.gradle` приложения. Откройте файл `Tasks/app/build.gradle` и добавьте следующую строку в раздел `dependencies`:

```
dependencies {
 ...
 implementation 'androidx.cardview:cardview:1.0.0'
 ...
}
```

Не забудьте синхронизировать это изменение с остальными частями приложения.

Создание карточного представления

Мы используем карточное представление из файла `task_item.xml`, которое включает текстовое представление и флажок.

Чтобы создать карточное представление, добавьте элемент `<androidx.cardview.widget.CardView>` в код макета. Код типичного карточного представления выглядит так:

```
<androidx.cardview.widget.CardView
 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="wrap_content"
 android:layout_margin="8dp"
 app:cardElevation="4dp"
 app:cardCornerRadius="4dp" >
 ...
</androidx.cardview.widget.CardView>
```

Как видно из этого кода, в него включается дополнительное пространство имен:

```
xmlns:app="http://schemas.android.com/apk/res-auto"
```

Это позволяет добавить атрибуты, с которыми карточка имеет закругленные углы и эффект тени, из-за которого она кажется приподнятой над окружающим фоном. Закругленные углы создаются атрибутом `app:cardCornerRadius`, а атрибут `app:cardElevation` создает эффект объема и добавляет отбрасываемые тени:

После того как вы определите карточное представление, в него можно добавить любые представления, которые в нем должны присутствовать. В приложении `Tasks` в карточное представление следует добавить текстовое представление для имени задачи и флажок для признака ее завершения. Давайте посмотрим, как это будет выглядеть в коде.

Полный код task_item.xml

Ниже приведена новая версия `task_item.xml`; обновите содержимое файла:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView
 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="wrap_content"
 android:layout_margin="8dp"
 app:cardElevation="4dp"
 app:cardCornerRadius="4dp" >
 <LinearLayout
```

```
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:orientation="vertical" >
 <TextView
 android:id="@+id/task_name"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:textSize="16sp"
 android:padding="8dp" />
 <CheckBox
 android:id="@+id/task_done"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:textSize="16sp"
 android:padding="8dp"
 android:clickable="false"
 android:text="Done?" />
 </LinearLayout>
</androidx.cardview.widget.CardView>
```

Затем необходимо обновить держатель представления адаптера, чтобы он работал с новым макетом, и заполнить представления карточки.

Держатель представления адаптера должен работать с новым кодом макета

При определении `TaskItemAdapter` (адаптера представления с переработкой) был включен внутренний класс `TaskItemViewHolder`. Мы использовали его для заполнения макета (текстового представления), который ассоциировался с каждым элементом представления с переработкой, и для заполнения имени задачи. Напомним, как выглядел код исходного внутреннего класса:

```
class TaskItemViewHolder(val rootView: TextView)
 : RecyclerView.ViewHolder(rootView) {
 companion object {
 fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
 val layoutInflater = LayoutInflater.from(parent.context)
 val view = layoutInflater
 .inflate(R.layout.task_item, parent, false) as TextView
 return TaskItemViewHolder(view)
 }
 }
 fun bind(item: Task) {
 rootView.text = item.taskName
 }
}
```

После изменения `task_item.xml` следует обновить класс `TaskItemViewHolder`, чтобы он работал с новым макетом. Для этого необходимо внести три изменения:

1. Обновить конструктор держателя представления, чтобы он использовал CardView вместо TextView.
2. Изменить метод inflateFrom(), чтобы он заполнял макет каждого элемента с CardView.
3. Обновить метод bind(), чтобы он заполнял текстовое представление и флажок в макете значениями свойств taskName и taskDone.

Код, необходимый для внесения этих изменений, вам уже знаком, поэтому мы просто приведем обновленный код `TaskItemAdapter` — и внутреннего класса `TaskItemViewHolder` — на следующей странице.

Полный код TaskItemAdapter.kt

Ниже приведен обновленный код `TaskItemAdapter`, который работает с новым кодом макета; обновите файл `TaskItemAdapter.kt`:

```
package com.hfad.tasks
import android.view.LayoutInflater
import android.view.ViewGroup
import android.widget.CheckBox
import android.widget.TextView
import androidx.cardview.widget.CardView
import androidx.recyclerview.widget.RecyclerView
class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>()
{
 var data = listOf<Task>()
 set(value) {
 field = value
 notifyDataSetChanged()
 }
 override fun getItemCount() = data.size
 override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
 : TaskItemViewHolder = TaskItemViewHolder.inflateFrom(parent)
 override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
 val item = data[position]
 holder.bind(item)
 }

class TaskItemViewHolder(val rootView: TextView CardView) :
RecyclerView.ViewHolder(rootView) {
 val taskName = rootView.findViewById<TextView>(R.id.task_name)
 val taskDone = rootView.findViewById<CheckBox>(R.id.task_done)
 companion object {
 fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
 val layoutInflater = LayoutInflater.from(parent.context)
 val view = layoutInflater
 .inflate(R.layout.task_item, parent, false) as TextView CardView
 return TaskItemViewHolder(view)
 }
 }
```

```
fun bind(item: Task) {
 taskName.text = item.taskName
 taskDone.isChecked = item.taskDone
 }
 }
}
```

Как представление с переработкой выглядит сейчас

Если запустить приложение после обновления кода `task_item.xml` и `TaskItemAdapter.kt`, вы увидите представление с переработкой, которое выводит данные задач в вертикальном списке карточных представлений:

Представление с переработкой размещает карточки именно так, потому что в файле `fragment_tasks.xml` указано, что в нем должен использоваться менеджер линейного макета:

```
<androidx.recyclerview.widget.RecyclerView
 ...
 app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager" />
```

По умолчанию этот менеджер макета размещает элементы в вертикальном списке с записями, распространяющимися на всю ширину. Однако вы можете включить дополнительные настройки — или выбрать другую разновидность менеджера макета,— чтобы изменить способ отображения элементов. Рассмотрим некоторые возможные варианты.

Галерея менеджеров макетов

Ниже описаны некоторые альтернативные возможности размещения элементов в представлениях с переработкой, а также способы их создания.

Размещение элементов по горизонтали

Менеджер линейного макета по умолчанию отображает элементы в вертикальном списке. Однако элементы также можно разместить по горизонтали, для этого достаточно переключиться на горизонтальную ориентацию:

```
<androidx.recyclerview.widget.RecyclerView ...
 app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"
 android:orientation="horizontal" />
```

Использование `GridLayoutManager` для размещения элементов в сетке

Если вы хотите выстроить элементы в сетку, попробуйте воспользоваться классом `GridLayoutManager`. Атрибут `app:spanCount` определяет количество столбцов в сетке:

```
<androidx.recyclerview.widget.RecyclerView ...
 app:layoutManager="androidx.recyclerview.widget.GridLayoutManager"
 app:spanCount="2" />
```

Размещение элементов в неравномерной сетке Если ваши элементы имеют разные размеры, можно воспользоваться классом `StaggeredGridLayoutManager`:

```
<androidx.recyclerview.widget.RecyclerView ...
 app:layoutManager="androidx.recyclerview.widget.StaggeredGridLayoutManager"
 app:spanCount="2" />
```

Применим один из этих стилей к представлению с переработкой в приложении `Tasks` и посмотрим, что происходит при запуске приложения.

Обновление fragment_tasks.xml для размещения элементов в сетке

Мы обновим представление с переработкой, чтобы элементы размещались в сетке из двух столбцов. Ниже приведена новая версия кода макета; обновите файл `fragment_tasks.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<layout
 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"
 tools:context=".TasksFragment">
 <data>
 <variable
 name="viewModel"
 type="com.hfad.tasks.TasksViewModel" />
 </data>
 <LinearLayout
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:orientation="vertical">
 <EditText
 ... />
 <Button
 ... />
 <androidx.recyclerview.widget.RecyclerView
 android:id="@+id/tasks_list"
 android:layout_width="match_parent"
 android:layout_height="0dp"
 android:layout_weight="1"
 android:gravity="top"
 app:layoutManager="androidx.recyclerview.widget.GridLayoutManager"
 app:spanCount="2" />
```

```
 </LinearLayout>
</layout>
```

Что происходит при выполнении кода

При выполнении приложения происходят следующие события:

1. `TasksFragment` создает объект `TaskItemAdapter` и назначает его адаптером представления с переработкой.
2. `TasksFragment` присваивает свойству data объекта `TaskItemAdapter` список `List<Task>`. `TasksFragment` получает `List`Task>`, наблюдая за свойством tasks объекта `TasksViewModel`.
3. Метод `onCreateViewHolder()` объекта `TaskItemAdapter` вызывается для каждого элемента, который должен отображаться в представлении с переработкой. В результате для каждого элемента создается объект `TaskItemViewHolder`. Макет (определенный в файле `task_item.xml`) заполняется для каждого держателя представления.
4. Метод `onBindViewHolder()` объекта `TaskItemAdapter` вызывается для каждого `TaskItemViewHolder`. Вызов связывает данные с представлениями в макете каждого держателя представления.
5. Представление с переработкой использует менеджер макета для размещения своих элементов. Так как представление с переработкой использует `GridLayoutManager` со значением `spanCount`, равным 2, элементы выстраиваются в сетку с двумя столбцами.
6. При каждом обновлении свойства tasks объекта `TasksViewModel`, `TasksFragment` передает обновленный список `List<Task>` объекту `TaskItemAdapter`. Этапы с 3-го по 5-й повторяются, чтобы представление с переработкой оставалось актуальным.

При запуске приложения представление с переработкой фрагмента `TasksFragment` отображает сетку карточек; в каждой карточке выводится имя задачи и признак ее завершения. Когда вы вводите новую задачу, она появляется в представлении с переработкой сразу же после того, как вы щелкнете на кнопке `Save Task`.

Поздравляем! Вы научились управлять внешним видом представлений с переработкой при помощи менеджеров макетов, а также отображать данные в сетке карт с поддержкой прокрутки. В следующей главе мы расширим новые знания и внесем дальнейшие улучшения в представление с переработкой.

Резюме

- Представление с переработкой — гибкое средство для вывода списка данных с возможностью прокрутки.
- Зависимость для библиотеки представлений с переработкой добавляется в файл `build.gradle` приложения.
- Для каждого представления с переработкой вы определяете макет для его элементов и адаптер, который помещает данные в представления каждого элемента. Затем представление с переработкой отображает эти элементы.

- Адаптер должен расширять класс `RecyclerView.Adapter`.
- Адаптер использует держатель представления, который расширяет `RecyclerView.ViewHolder`. Он содержит информацию о том, как должен отображаться макет каждого элемента, и обычно определяется как внутренний класс адаптера.
- Метод `onCreateViewHolder()` адаптера заполняет макет каждого элемента и создает держателей представлений.
- Метод `onBindViewHolder()` адаптера связывает данные с представлениями в макете каждого элемента.
- Для добавления представления с переработкой используется элемент `<androidx.recyclerview.widget.RecyclerView>`.
- Карточное представление — разновидность фреймового макета с закругленными углами и имитацией рельефа.
- Зависимость для библиотеки карточных представлений добавляется в файл `build.gradle` приложения.
- Менеджер макета управляет отображением элементов.
- `LinearLayoutManager` размещает элементы по вертикали или по горизонтали.
- `GridLayoutManager` размещает элементы в сетке.
- `StaggeredGridLayoutManager` напоминает `GridLayoutManager`, но поддерживает элементы с различающимися размерами.