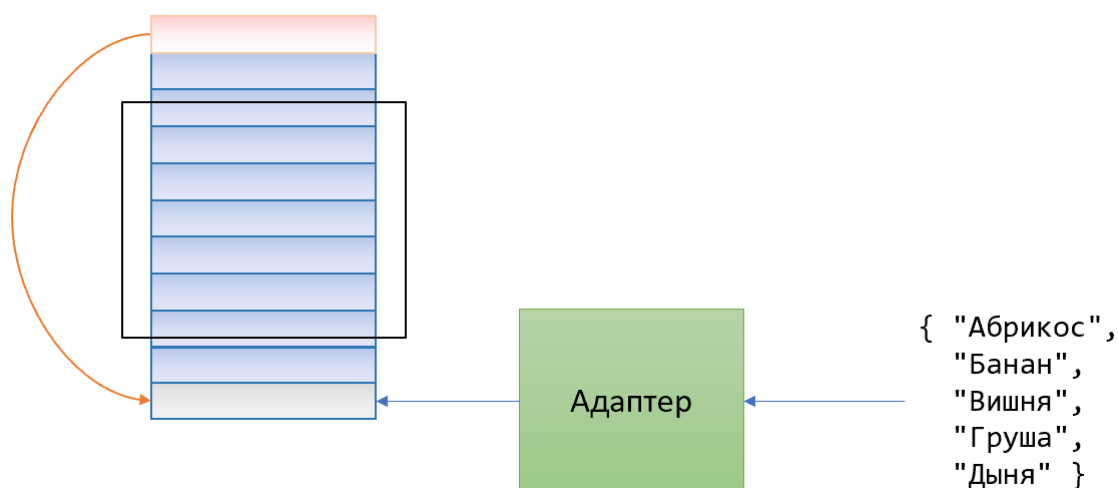


# Элемент управления RecyclerView

[RecyclerView](#) позволяет эффективно отображать большие наборы данных. Вы предоставляете данные и определяете, как выглядит каждый элемент, а библиотека [RecyclerView](#) динамически создает элементы, когда они необходимы. Когда элемент прокручивается за пределы экрана, [RecyclerView](#) не уничтожает его представление, а вместо этого повторно использует его для новых элементов, которые появляются на экране. Это позволяет снизить расход памяти и повысить производительность приложения.



Внутри [RecyclerView](#) сделан как представление на основе [ViewGroup](#), то есть по сути в некотором смысле является контейнером, так же как [LinearLayout](#), [GridLayout](#) и т.п. Разница в том, что в другие контейнеры дочерние элементы добавляются вручную, а [RecyclerView](#) автоматизирует этот процесс с помощью адаптера.

Каждый отдельный элемент в списке представлен особым *держателем представления* (англ. view holder). Изначально, когда держатель представления только создан, с ним ещё не связаны никакие данные. Когда возникает необходимость, [RecyclerView](#) заполняет его новыми данными, а впоследствии может делать это ещё не раз. Держатель представления всегда базируется на классе [RecyclerView.ViewHolder](#).

[RecyclerView](#) создаёт держатели и по мере необходимости связывает их с данными с помощью специального *адаптера* на базе класса [RecyclerView.Adapter](#).

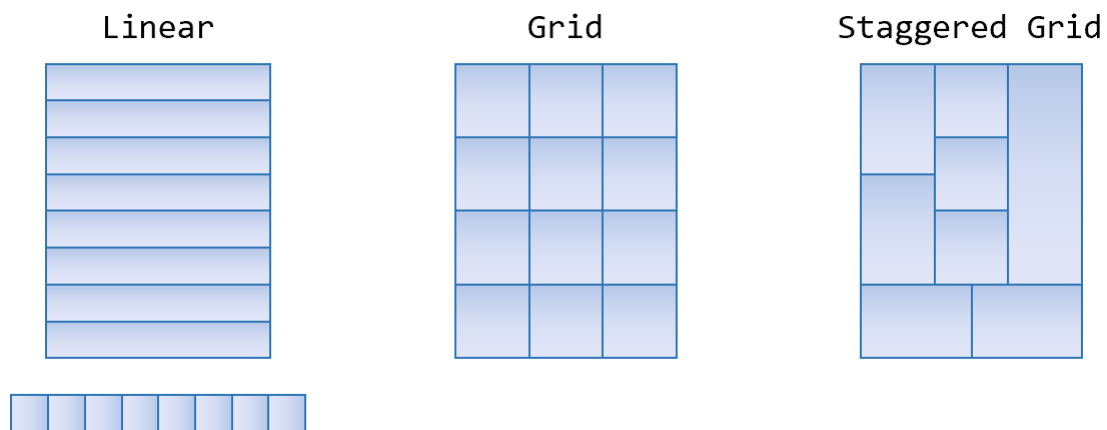
Ещё одним важным элементом [RecyclerView](#) является *менеджер разметки*, который упорядочивает элементы в списке, определяет как будет выглядеть сам список. Чаще всего используется один из готовых менеджеров разметки, предоставляемых библиотекой [RecyclerView](#), но можно определить и свой собственный. Все менеджеры разметки основаны на абстрактном классе [LayoutManager](#).

## Реализация списка

Прежде всего, следует определить как будет выглядеть список. Обычно используется один из трёх стандартных менеджеров разметки библиотеки [RecyclerView](#):

- [LinearLayoutManager](#) упорядочивает элементы в одномерный список.

- `GridLayoutManager` упорядочивает элементы в двумерной сетке:
  - Если сетка расположена **вертикально**, `GridLayoutManager` пытается сделать так, чтобы все элементы в каждой строке имели одинаковую ширину и высоту, но разные строки могут иметь разную высоту.
  - Если сетка расположена **горизонтально**, `GridLayoutManager` пытается сделать так, чтобы все элементы в каждом столбце имели одинаковую ширину и высоту, но разные столбцы могут иметь разную ширину.
- `StaggeredGridLayoutManager` аналогичен `GridLayoutManager`, но не требует, чтобы элементы в строке имели одинаковую высоту (для вертикальных сеток) или элементы в одном столбце имели одинаковую ширину (для горизонтальных сеток). В результате элементы в строке или столбце могут оказаться смещенными друг относительно друга.



Далее следует подготовить разметку элемента – XML-файл, который будет определять вид каждого элемента в списке. Внутри такой разметки могут быть самые разные элементы управления: `TextView`, `ImageView`, `RatingBar` и т.д. Это позволяет создавать комплексные и стильные элементы.

На основе разметки нужно создать класс держателя, унаследовав его от `ViewHolder`. Как правило, в этом классе создаются поля для тех элементов разметки, которые будут заполняться данными.

Наконец, нужно создать класс-адаптер. Именно он будет брать данные, которые следует отобразить, и наполнять ими держатели – этот процесс называется *привязкой*. Адаптер наследуется от класса `RecyclerView.Adapter` и должен переопределять следующие методы:

- `onCreateViewHolder()`: этот метод вызывается каждый раз, когда необходимо создать новый держатель `ViewHolder`. В этом методе создаётся и инициализируется объект `ViewHolder` и связанная с ним разметка, но содержимым они не заполняются: на этом этапе держатель еще не привязан к конкретным данным.
- `onBindViewHolder()`: этот метод вызывается когда нужно связать существующий держатель с данными. Метод извлекает нужные данные и использует их для заполнения разметки держателя. Например, если отображается список контактов, то метод может найти очередной контакт, и элементы `TextView` в держателе именем абонента и номером его телефона.
- `getItemCount()`: этот вызывается для получения размера данных. Например, в приложении адресной книги это может быть общее количество адресов. `RecyclerView`

использует этот размер для определения количества элементов, которые следует отобразить.

## Пример: телефонная книга

Разберём пример работы `RecyclerView` реализовав простую телефонную книгу. Для начала создадим класс, который будет хранить информацию о контактах – имя и номер телефона:

```
data class Contact(  
    val name: String,  
    val number: String  
)
```

Создадим файл разметки с именем `list_item.xml`, который будет содержать два элемента `TextView` друг над другом:

```
<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:orientation="vertical"  
    android:padding="4dp">  
  
    <TextView  
        android:id="@+id/name"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:layout_marginBottom="4dp"  
        android:textStyle="bold"/>  
  
    <TextView  
        android:id="@+id/number"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:textColor="#808080"/>  
  
</LinearLayout>
```

Каждый из элементов получил свой идентификатор, чтобы к ним можно было обращаться из кода.

Теперь спроектируем объект `ViewHolder`, который будет хранить данное представление. В качестве параметра конструктор получает уже готовое представление, и для удобства из него извлекаются ссылки на сами элементы с именем и номером:

```
class ContactHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {  
    var tvName: TextView  
    var tvNumber: TextView  
  
    init {  
        tvName = itemView.findViewById(R.id.name)  
        tvNumber = itemView.findViewById(R.id.number)  
    }  
}
```

Класс `ContactHolder` обычно вложен внутрь адаптера, поэтому создадим сразу и его:

```

class ContactsAdapter(val contacts: List<Contact>) :
    RecyclerView.Adapter<ContactsAdapter.ContactHolder>() {

    class ContactHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        var tvName: TextView
        var tvNumber: TextView

        init {
            tvName = itemView.findViewById(R.id.name)
            tvNumber = itemView.findViewById(R.id.number)
        }
    }

    // Создание нового держателя, и надувание для него представления из разметки
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ContactHolder {
        val inflater = LayoutInflater.from(parent.context)
        val view = inflater.inflate(R.layout.list_item, parent, false)

        return ContactHolder(view)
    }

    // Привязка данных к держателю
    override fun onBindViewHolder(holder: ContactHolder, position: Int) {
        holder.tvName.text = contacts[position].name
        holder.tvNumber.text = contacts[position].number
    }

    // Количество элементов в списке
    override fun getItemCount(): Int {
        return contacts.size
    }
}

```

Функция `onBindViewHolder` получает в качестве параметра держатель, который нужно (заново или в первый раз) наполнить данными, а также позицию элемента в списке. Позиция обычно совпадает с позицией данных в массиве.

Осталось разместить элемент `RecyclerView` в активности, не забыв указать какой менеджер разметки нужно использовать:

```

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

```

И теперь можно связать всё воедино при запуске приложения: заполнить данные, создать адаптер, и назначить его списку:

```

val contacts = listOf(
    Contact("Буратино", "+71237420001"),
    Contact("Папа Карло", "+71237420002"),
    Contact("Джузеппе Сизый Нос", "+71237420003"),
    Contact("Мальвина", "+71237420004"),
    Contact("Артемон", "+71237420005"),
    Contact("Пьеро", "+71237420006"),
    Contact("Арлекин", "+71237420007"),

```

```

        Contact("Черепаша Тортила", "+71237420008"),
        Contact("Карабас-Барабас", "+71237420009"),
        Contact("Дуремар", "+71237420010"),
        Contact("Лиса Алиса", "+71237420011"),
        Contact("Кот Базилио", "+71237420012")
    )

```

```
val adapter = ContactsAdapter(contacts)
```

```
val list = findViewById<RecyclerView>(R.id.list)
list.adapter = adapter
```

В результате после запуска программы получится примерно такая адресная книга:

```

    Буратино
    +71237420001

    Папа Карло
    +71237420002

    Джузеппе Сизый Нос
    +71237420003

    Мальвина
    +71237420004

    Артемон
    +71237420005

    Пьеро
    +71237420006

    Арлекин
    +71237420007

```

Выглядит адресная книга хорошо, однако имеет существенный недостаток: при нажатии на элемент списка ничего не происходит. Такое поведение является нормальным для [RecyclerView](#), ведь он по сути просто контейнер для отображения других элементов, и ничего не знает о том для чего они предназначены. Организация взаимодействия ложится на плечи разработчика: если требуется реакция на нажатие на элемент (или какую-то его часть), то нужно добавить соответствующий обработчик к представлению. Как правило, это делается при создании объекта-держателя. Доработаем адаптер чтобы он обрабатывал нажатия на элемент и уведомлял об этом создателя:

```

class ContactsAdapter(val contacts: List)

    ...

    private var onItemClickListener: ((Int) -> Unit)? = null

    fun setOnItemClickListener(f: (Int) -> Unit) { onItemClickListener = f }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ContactHolder {
        val inflater = LayoutInflater.from(parent.context)
        val view = inflater.inflate(R.layout.list_item, parent, false)

        val holder = ContactHolder(view)
    }

```

```

        view.setOnClickListener {
            val pos = holder.adapterPosition
            if (pos != NO_POSITION)
                onItemClick?.invoke(pos)
        }

        return holder
    }
    ...

```

В адаптер добавлена ссылка на функцию, а также метод `setOnItemClickListener`, который принимает такую функцию и присваивает её ссылке. А при нажатии на элемент определяется позиция элемента и вызывается функция (если, конечно, она была установлена). В главной активности следует установить обработчик нажатия у адаптера:

```

val adapter = ContactsAdapter(contacts)
adapter.setOnItemClickListener {
    Toast.makeText(this, contacts[it].name, Toast.LENGTH_SHORT).show()
}















val list = findViewById<RecyclerView>(R.id.list)
list.adapter = adapter

```

Теперь при нажатии на элемент списка появится всплывающее сообщение с именем контакта.

## Задание

Создайте приложение, показывающее пользователю витрину онлайн-магазина. Приложение должно выглядеть примерно следующим образом:

	<b>Арахисовая паста</b> Цена: 179.99 ₺	
	<b>Пицца</b> Цена: 499.99 ₺	
	<b>Рис</b> Цена: 119.99 ₺	
	<b>Брокколи</b> Цена: 339.99 ₺	
	<b>Сыр</b> Цена: 899.99 ₺	
	<b>Каша</b> Цена: 49.99 ₺	
	<b>Молоко</b> Цена: 89.99 ₺	

Список должен быть организован с помощью элемента [RecyclerView](#).

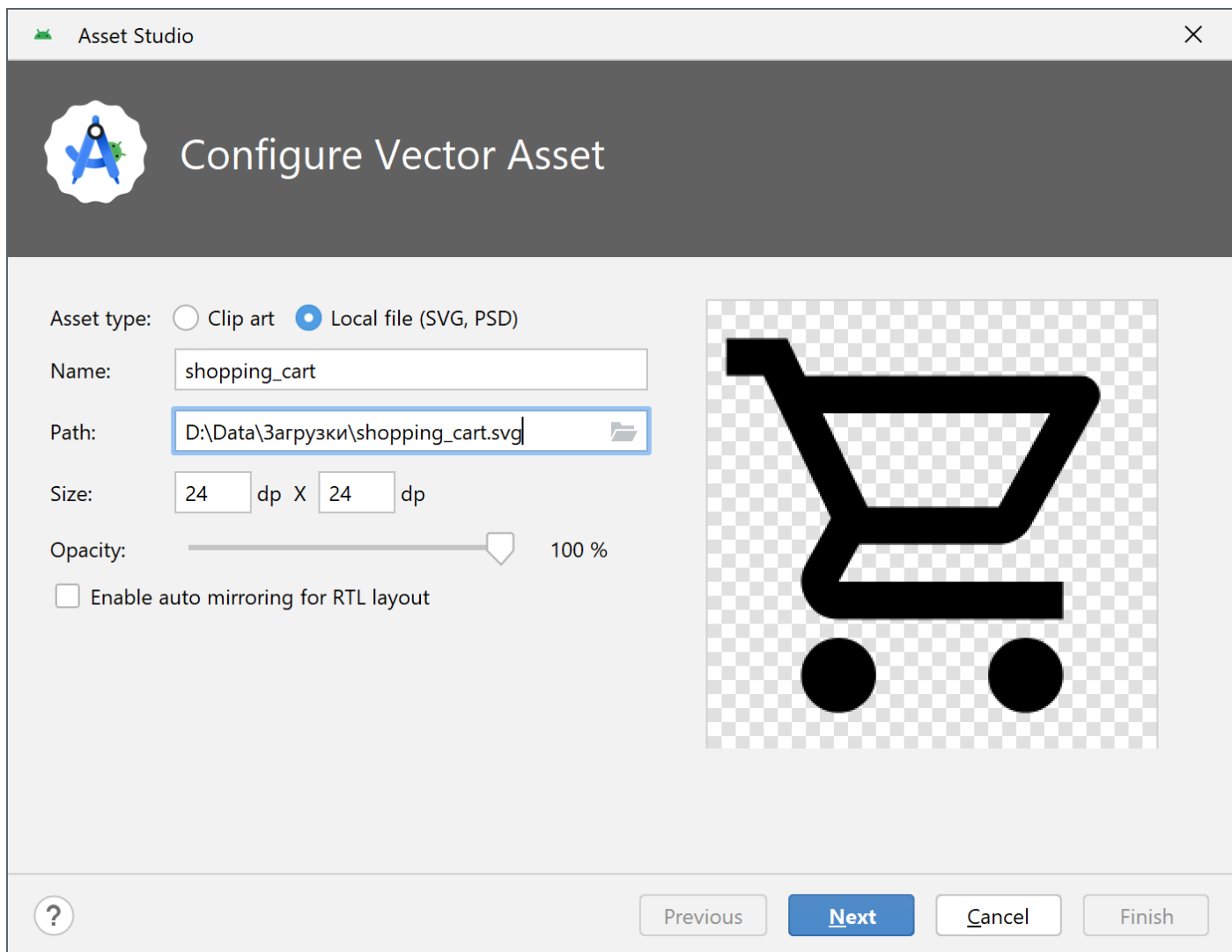
Разметка элемента списка должна быть сделана на основе [CardView](#). Он представляет собой контейнер на базе [FrameLayout](#), только добавляет рамку с возможностью скругления углов, тень и т.д. Внутрь можно поместить другой контейнер, ведь [FrameLayout](#) и его наследники – не самые удобные в плане размещения элементов.

Каждый элемент должен содержать изображение товара, название, стоимость, а также кнопку добавления товара в тележку. В примере выше изображения товаров взяты из лабораторной работы 8 (игра «Съедобно-несъедобно»), но можно взять любые другие товары.

При нажатии кнопки со значком добавления товара в тележку информация об этом должна передаваться в основную активность:

- Если товар ещё **не был** добавлен, то появляется всплывающее сообщение зеленого цвета с текстом «Добавлено: *<название товара>*», а значок меняется на тележку без знака «+».
- Если товар уже **был** добавлен, то появляется всплывающее сообщение красного цвета с текстом «Удалено: *<название товара>*», а значок меняется обратно на тележку со знаком «+».

Значки тележки со знаком «+» и без него приложены в архиве, однако можно использовать другие аналогичные по смыслу значки. Чтобы добавить векторное изображение (SVG-файл со значком) в ресурсы, нужно щёлкнуть правой кнопкой по папке *res* → *drawable*, и в меню выбрать *New* → *Vector Asset*, а затем в появившемся окне выбрать SVG-файл и добавить его:



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