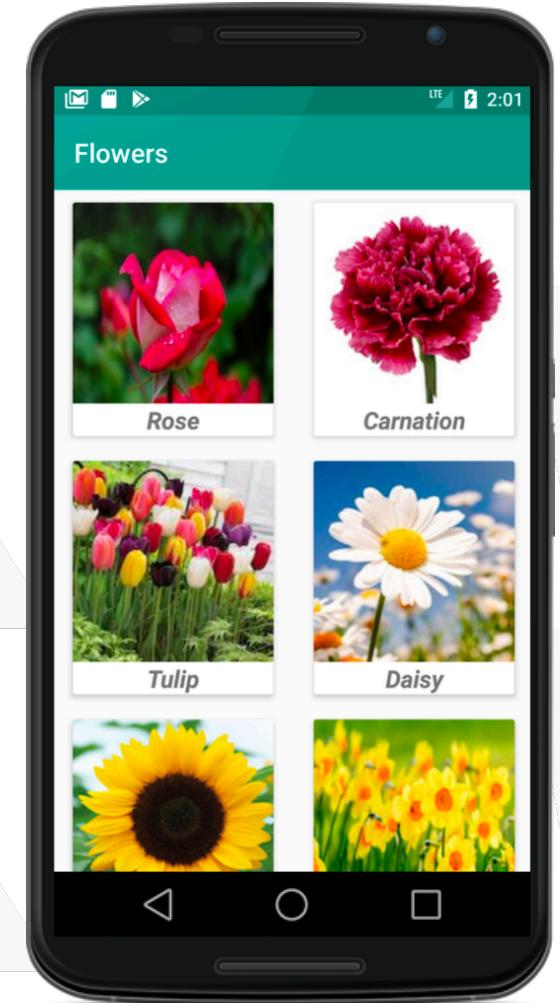
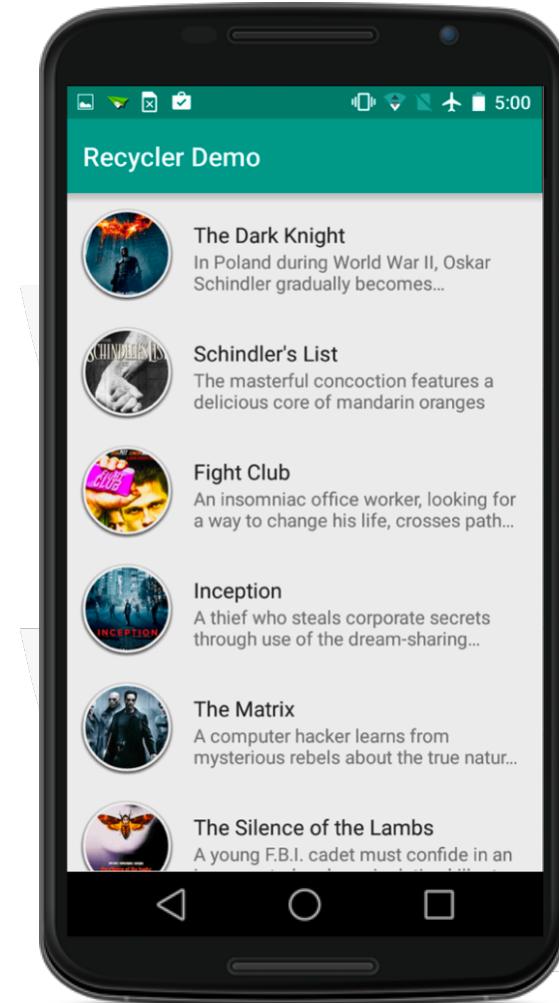
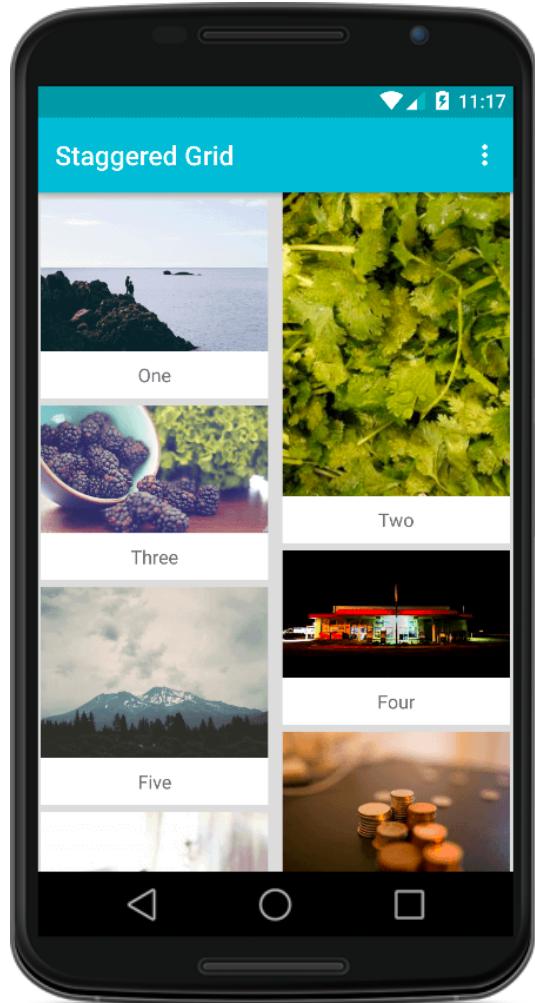


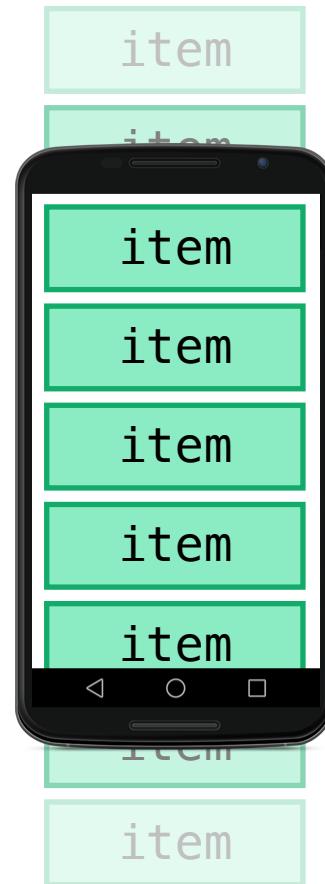
# Списки, RecyclerView и Pagination

Андрей Фоменков  
Одноклассники

# Списки повсюду!



# Попробуем реализовать?



Легко!  
Закинем все эти элементы в  
[LinearLayout](#), который  
лежит внутри [ScrollView](#)



«Например, у нас есть **9** элементов»

# Попробуем реализовать?



Да, уже сложнее, но мы  
всё равно справимся...



«Теперь у нас есть **9999** элементов»

# Попробуем реализовать?



«В теории список может быть **бесконечным**»

# Идея: ограничить кол-во элементов



В процессе скролла элементы  
*создаются* с одного конца списка, а  
с другого — *уничтожаются*

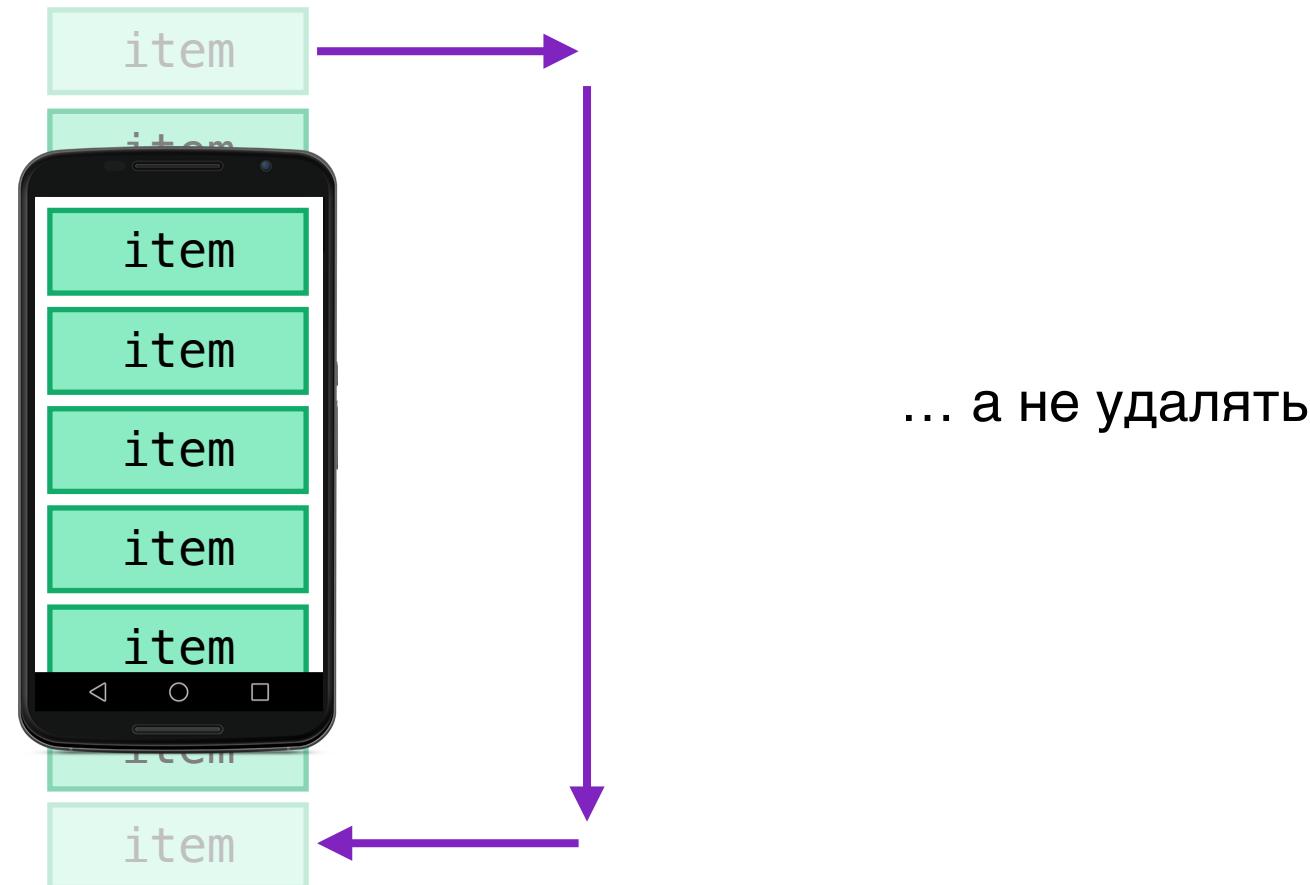
# Переработка отходов (recycling)



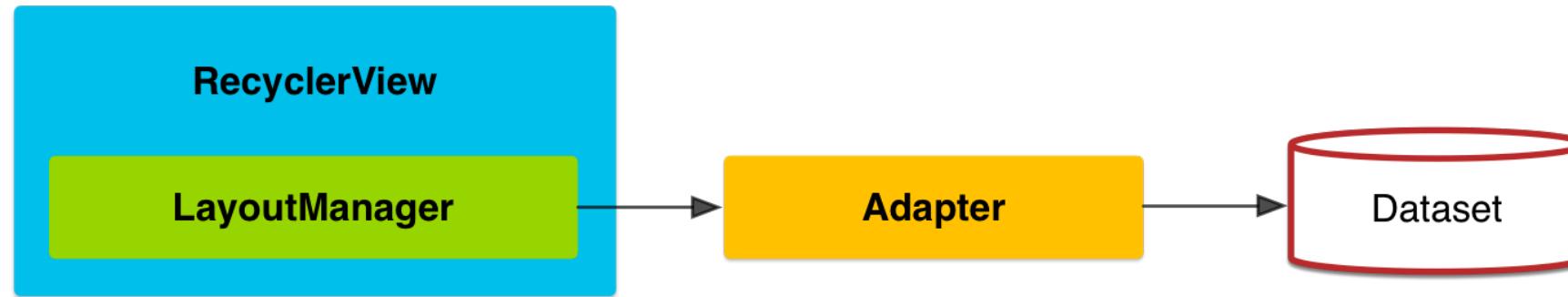
# Переработка отходов (recycling)



# Идея: переиспользовать элементы



# RecyclerView и его друзья



- Это **ViewGroup**, позволяющая отобразить часть элементов некоторого множества данных;
- **Adapter** – класс, связывающий данные с отображаемыми в **RecyclerView** элементами;
- За расположение элементов и направление скролла отвечает **LayoutManager**;
- Каждый отображаемый элемент определяется классом **ViewHolder**.

# ViewHolder

```
data class Item(val firstName: String, val lastName: String, val phoneNumber: String)
```

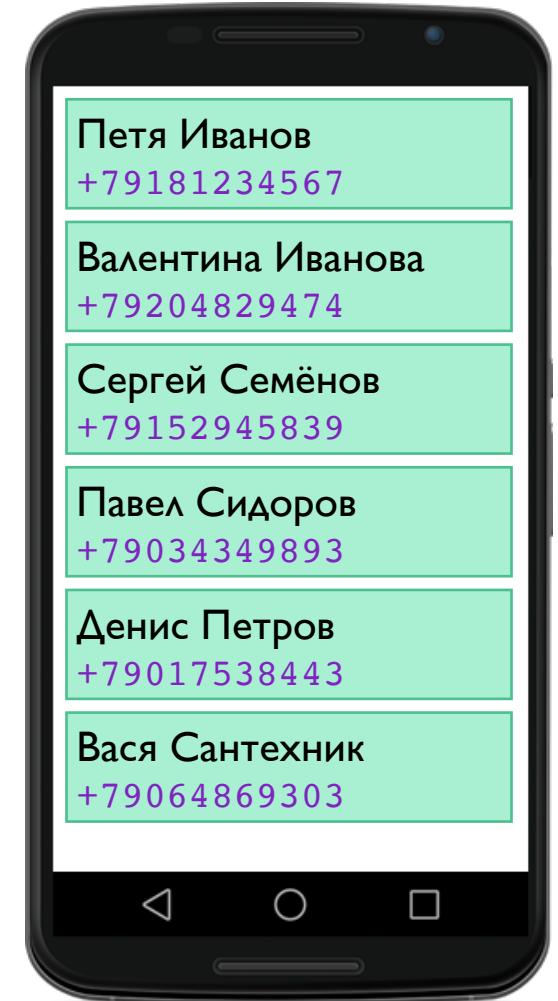
Элемент данных, пришедший от сервера / БД / хранилища и т.д.

```
class SampleViewHolder(val itemView: View) : RecyclerView.ViewHolder(itemView) {
    private val firstNameTextView = itemView.findViewById<TextView>(R.id.first_name)
    private val lastNameTextView = itemView.findViewById<TextView>(R.id.last_name)
    private val phoneNumberTextView = itemView.findViewById<TextView>(R.id.phone_number)

    fun bind(item: Item) {
        firstNameTextView.text = item.firstName
        lastNameTextView.text = item.lastName
        phoneNumberTextView.text = item.phoneNumber
    }
}
```

Родительская View

Связывание элементов View с данными из Item



ViewHolder - место, где Item встречается с View

# Простейший Adapter

```
class SampleAdapter : RecyclerView.Adapter<SampleViewHolder>() {  
    private val itemList = mutableListOf<Item>()  
  
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): SampleViewHolder {  
        val itemView = LayoutInflater.from(parent.context)  
            .inflate(R.layout.item_user_contact, parent, false)  
        return SampleViewHolder(itemView)  
    }  
  
    override fun onBindViewHolder(holder: SampleViewHolder, position: Int) {  
        val item = itemList[position]  
        holder.bind(item)  
    }  
  
    override fun getItemCount(): Int {  
        return itemList.size  
    }  
  
    fun update(list: List<Item>) {  
        itemList.clear()  
        itemList += list  
        notifyDataSetChanged()  
    }  
}
```

Коллекция данных

Создание экземпляра SampleViewHolder

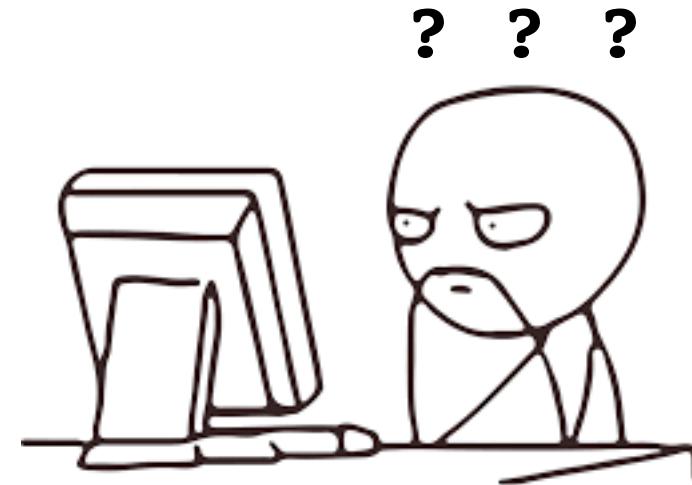
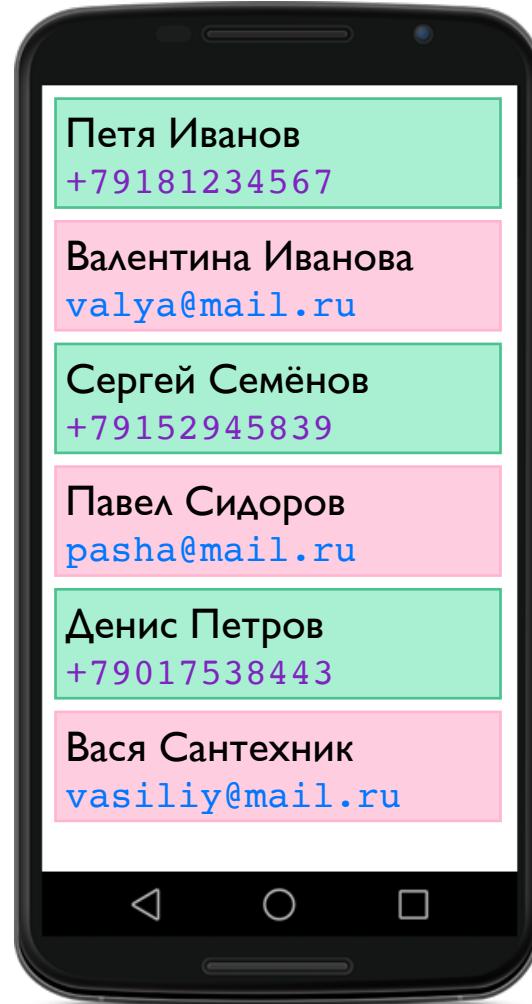
XML-вёрстка для View

Связывание с данными

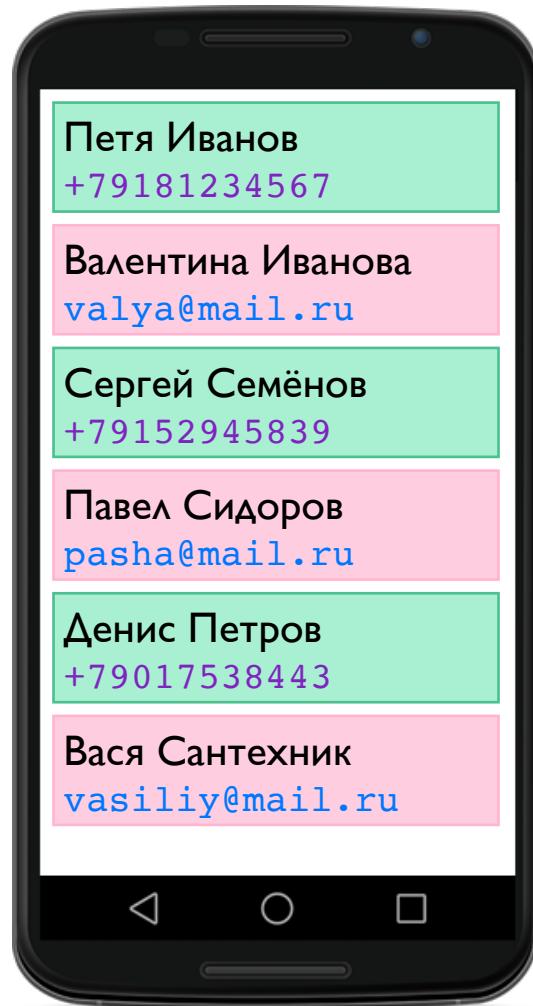
Размер коллекции

Обновление списка

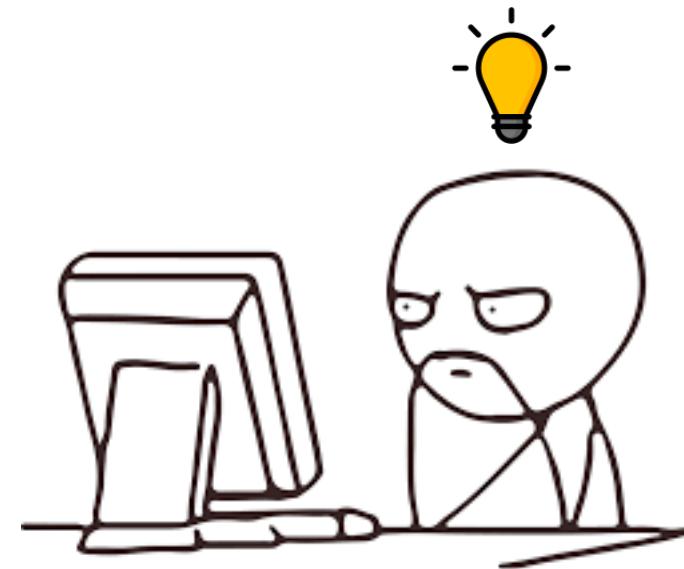
# Adapter с различными видами ViewHolder



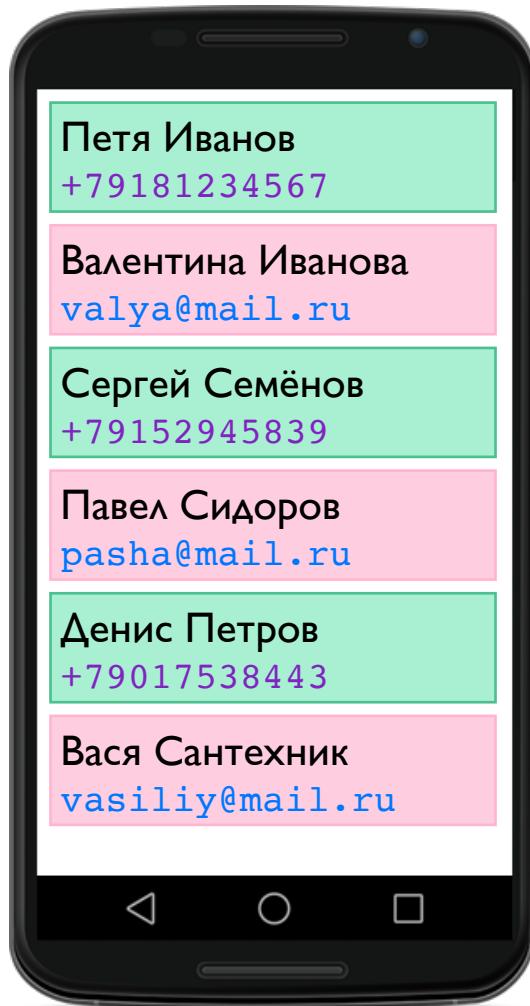
# Adapter с различными видами ViewHolder



```
sealed class Item {  
  
    data class UserPhone(  
        val firstName: String,  
        val lastName: String,  
        val phoneNumber: String,  
    ) : Item()  
  
    data class UserEmail(  
        val firstName: String,  
        val lastName: String,  
        val email: String,  
    ) : Item()  
}
```



# Adapter с различными видами ViewHolder



```
sealed class Item {

    data class UserPhone(
        val firstName: String,
        val lastName: String,
        val phoneNumber: String,
    ) : Item()

    data class UserEmail(
        val firstName: String,
        val lastName: String,
        val email: String,
    ) : Item()

}
```

```
sealed class ItemViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {

    class UserPhone(itemView: View) : ItemViewHolder(itemView) {

        fun bind(item: Item.UserPhone) { ... }

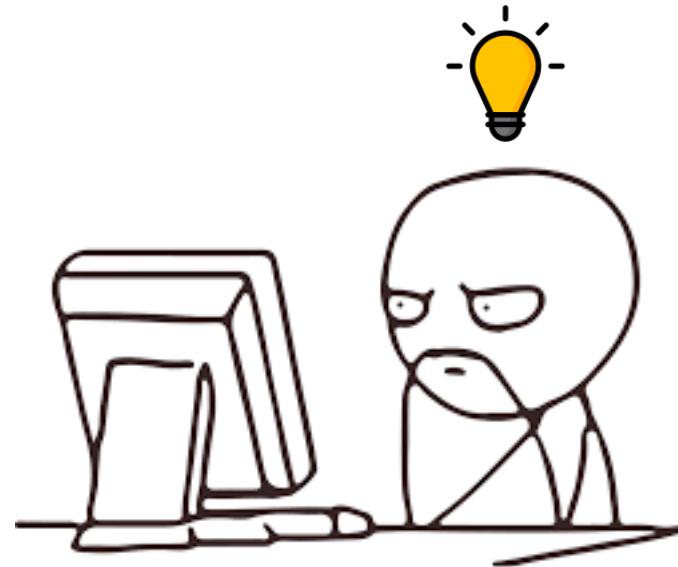
    }

    class UserEmail(itemView: View) : ItemViewHolder(itemView) {

        fun bind(item: Item.UserEmail) { ... }

    }

}
```



# Adapter с различными видами ViewHolder

```
class SampleAdapter : RecyclerView.Adapter<ItemViewHolder>() {  
  
    private fun inflate(parent: ViewGroup, @LayoutRes layoutResId: Int): View {  
        return LayoutInflater.from(parent.context).inflate(layoutResId, parent, false)  
    }  
  
    private companion object {  
        const val VIEW_TYPE_USER_PHONE = 0  
        const val VIEW_TYPE_USER_EMAIL = 1  
    }  
  
    private val itemList = mutableListOf<Item>()  
  
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ItemViewHolder {  
        return when (viewType) {  
            VIEW_TYPE_USER_PHONE -> {  
                val itemView = inflate(parent, R.layout.item_user_phone)  
                ItemViewHolder.UserPhone(itemView)  
            }  
            VIEW_TYPE_USER_EMAIL -> {  
                val itemView = inflate(parent, R.layout.item_user_email)  
                ItemViewHolder.UserEmail(itemView)  
            }  
            else -> throw IllegalArgumentException("Unknown view type: $viewType")  
        }  
    }  
}
```

# Adapter с различными видами ViewHolder

```
override fun onBindViewHolder(holder: ItemViewHolder, position: Int) {
    val item = itemList[position]

    when (holder) {
        is ItemViewHolder.UserPhone -> {
            holder.bind(item as Item.UserPhone)
        }
        is ItemViewHolder.UserEmail -> {
            holder.bind(item as Item.UserEmail)
        }
    }
}

override fun getItemCount(): Int {
    return itemList.size
}

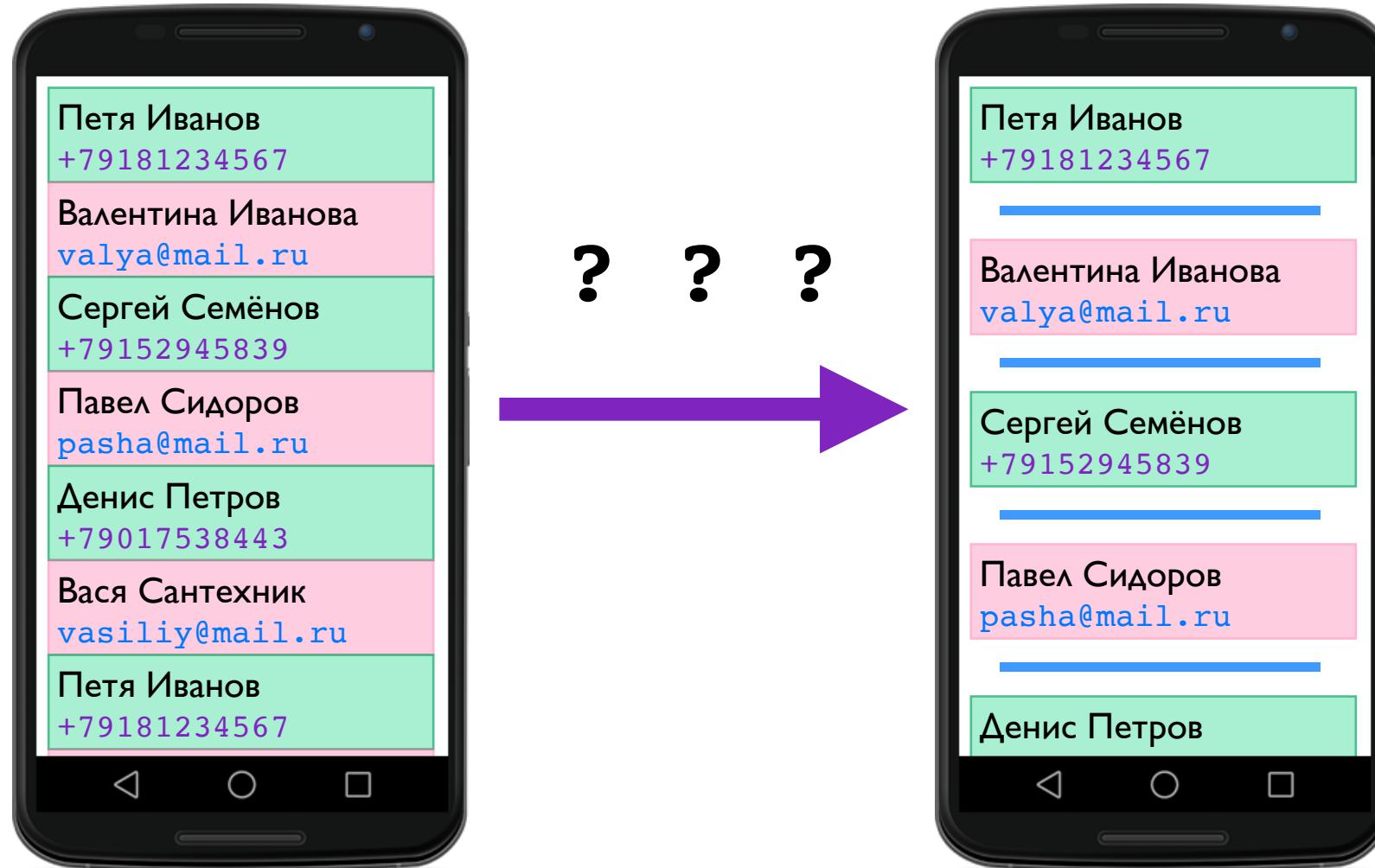
override fun getItemViewType(position: Int): Int {
    return when (itemList[position]) {
        is Item.UserPhone -> VIEW_TYPE_USER_PHONE
        is Item.UserEmail -> VIEW_TYPE_USER_EMAIL
    }
}

fun update(list: List<Item>) { ... }
```

Связываем ViewHolder с данными

Определяем viewType по позиции элемента

# Разделители между элементами



# Разделители между элементами

Петя Иванов  
+79181234567

«Добавить разделитель  
в XML элемента»

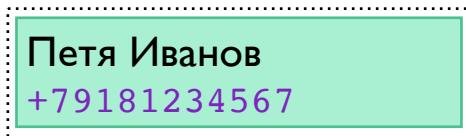
Петя Иванов  
+79181234567



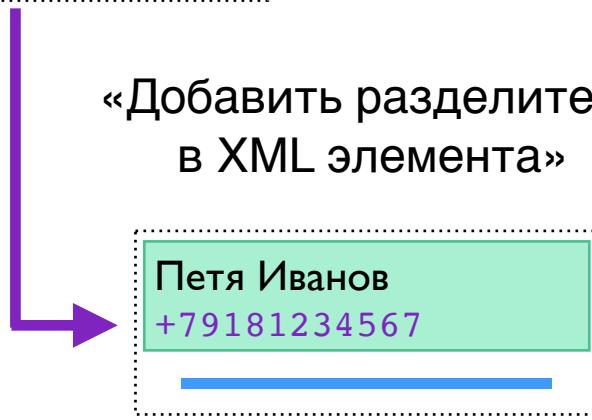
Плохое решение



# Разделители между элементами



«Добавить разделитель в XML элемента»



Плохое решение

«Использовать `ItemDecoration`»

```
val decoration = DividerItemDecoration(this, LinearLayout.VERTICAL)
val divider = ContextCompat.getDrawable(this, R.drawable.divider)
val recyclerView = findViewById<RecyclerView>(R.id.recycler_view)

decoration.setDrawable(divider)

recyclerView.layoutManager = LinearLayoutManager(this)
recyclerView.adapter = adapter
recyclerView.addItemDecoration(decoration)
```



Хорошее решение

# Обновление списка

```
fun update(list: List<Item>) {  
    itemList.clear()  
    itemList += list  
    notifyDataSetChanged()  
}
```

It will always be more efficient to use more specific change events if you can. Rely on notifyDataSetChanged as a last resort.

- Метод `notifyDataSetChanged()` уведомляет об изменении данных;
- Использовать его можно, но не рекомендуется;
- Происходит обновление всех видимых элементов, даже если изменение касается только одного.

# Обновление списка

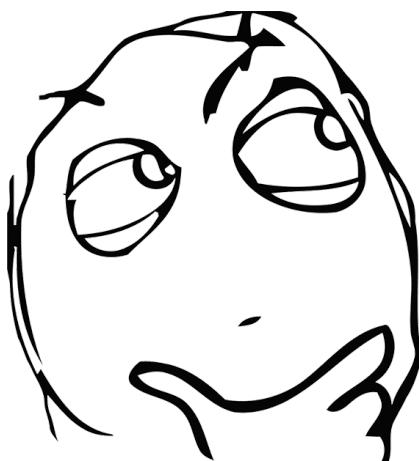
```
fun update(list: List<Item>) {  
    itemList.clear()  
    itemList += list  
    notifyDataSetChanged()  
}
```

It will always be more efficient to use more specific change events if you can. Rely on notifyDataSetChanged as a last resort.

- Метод `notifyDataSetChanged()` уведомляет об изменении данных;
- Использовать его можно, но не рекомендуется;
- Происходит обновление всех видимых элементов, даже если изменение касается только одного;
- Есть другие методы для частичного обновления списка:

```
...  
notifyItemChanged(3) // Changed at position = 3  
notifyItemInserted(4) // Inserted at position = 4  
notifyItemMoved(5, 6) // Moved from position = 5 to = 6  
notifyItemRemoved(7) // Removed at position = 7  
notifyItemRangeChanged(5, 10) // Range changed from position = 5, count = 10  
...
```

# Обновление списка



старый список

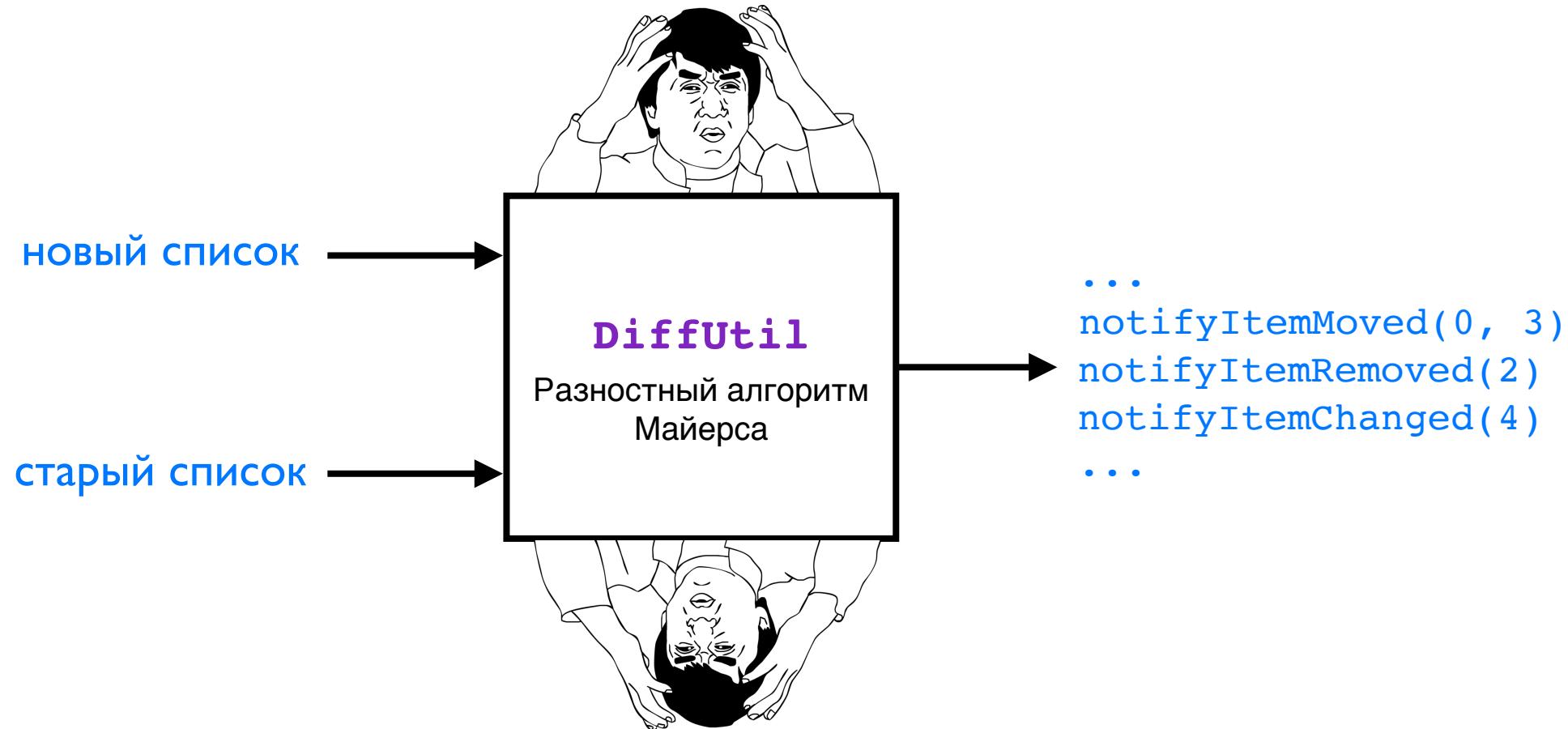
Петя Иванов +79181234567
Валентина Иванова valya@mail.ru
Сергей Семёнов +79152945839
Павел Сидоров pasha@mail.ru
Денис Петров +79017538443
Вася Сантехник vasiliy@mail.ru

новый список

Валентина Иванова valya@mail.ru
Сергей Семёнов 02
Денис Петров +79017538443
Вася Сантехник vasiliy@mail.ru
Петя Иванов +79181234567

«Получается, самому нужно  
как-то сравнивать два списка?»

# За нас это сделает DiffUtil



# Реализация DiffUtil.Callback

```
private companion object {

    class SampleDiffUtilCallback(
        private val oldList: List<Item>,
        private val newList: List<Item>, + Старый и новый список
    ) : DiffUtil.Callback() {

        override fun getOldListSize(): Int { ●———— Размер старого списка
            return oldList.size
        }

        override fun getNewListSize(): Int { ●———— Размер нового списка
            return newList.size
        }

        override fun areItemsTheSame(oldItemPosition: Int, newItemPosition: Int): Boolean {
            return oldList[oldItemPosition].id == newList[newItemPosition].id ●———— Более «легковесное»
        }                                              сравнение элементов.
                                                       Например, по id

        override fun areContentsTheSame(oldItemPosition: Int, newItemPosition: Int): Boolean
            // Compare item contents if areItemsTheSame() == true ●———— Глубокое сравнение
        }                                              элементов через
                                                       контент

        override fun getChangePayload(oldItemPosition: Int, newItemPosition: Int): Any? {
            // Return changed payload if areItemsTheSame() == true && areContentsTheSame() == false
        }

    }
}
```

↑ Вывести конкретные поля, где произошло изменение ●———— Например, для анимаций

# DiffUtil в действии

```
class SampleAdapter : RecyclerView.Adapter<ItemViewHolder>() {  
    private val itemList = mutableListOf<Item>() ●———— Уже заполненный старый список  
    ...  
    fun update(list: List<Item>) {  
        val callback = SampleDiffUtilCallback(oldList = itemList, newList = list)  
        val result = DiffUtil.calculateDiff(callback) ●———— Расчёт разницы - diff  
        itemList.clear()  
        itemList += list  
  
        result.dispatchUpdatesTo(this) ●———— Обновляем Adapter  
    }  
    ...
```

# DiffUtil в действии

```
class SampleAdapter : RecyclerView.Adapter<ItemViewHolder>() {  
  
    private val itemList = mutableListOf<Item>() ●———— Уже заполненный старый список  
    ...  
    ↓———— Новый список  
    fun update(list: List<Item>) {  
        val callback = SampleDiffUtilCallback(oldList = itemList, newList = list)  
        val result = DiffUtil.calculateDiff(callback) ●———— Расчёт разницы - diff  
  
        itemList.clear()  
        itemList += list  
  
        result.dispatchUpdatesTo(this) ●———— Обновляем Adapter  
    }  
    ...  
}
```



Расчёт происходит  
на UI потоке

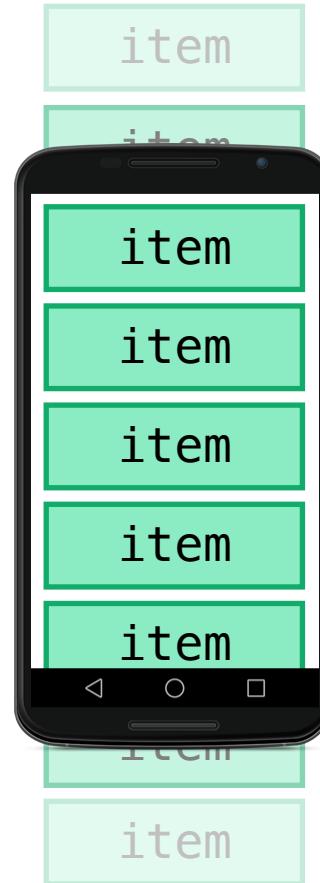


# Пару слов об оптимизациях...

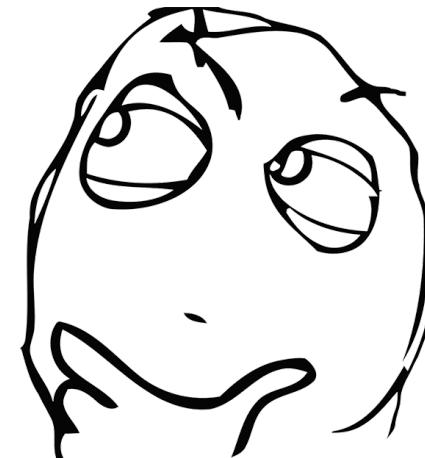
- `ListAdapter` - тот же `Adapter`, но со встроенной функцией расчёта *diff* в фоновом потоке;
- `AsyncLayoutInflater` - класс для создания View в фоновом потоке;
- Тяжёлые операции (*parsing, вычисления, загрузка и т.д.*) нужно исключить из логики `Adapter`;
- Вызов `RecyclerView.setHasFixedSize(true)` подсказывает, что размер не будет меняться  
=> `RecyclerView` может применить внутренние оптимизации и избежать лишних шагов layout;
- Регулирование размера внутреннего кэша `ViewHolder`'ов;
- Тормозит скролл списка? Используйте Profiler для анализа.

# Pagination

# Как загружать почти бесконечную ленту?

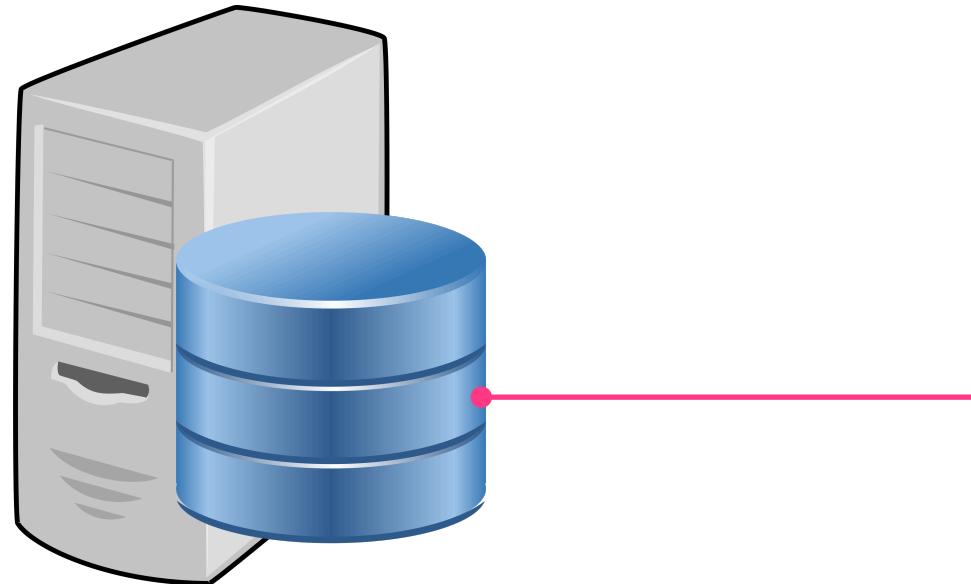


- По частям, видимо?
- Насколько большими должны быть части?
- Откуда начинать загрузку?
- Как узнать, что мы дошли до конца ленты?



# Пример: загрузка сообщений с сервера

Сервер + БД с сообщениями



```
data class Message(  
    val id: Long?,  
    val text: String,  
    // Other fields  
)
```

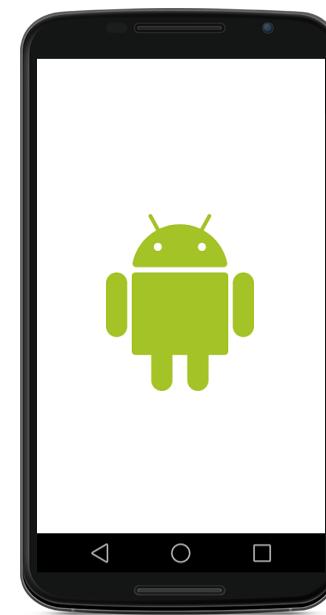
Формат одного сообщения

# Пример: загрузка сообщений с сервера

Сервер + БД с сообщениями



Клиент



«Выдайте мне **5** сообщений,  
начиная с **id = 125**»

\* Request \*

\* Response \*

«Пожалуйста, вот они»

```
id = 125, text = «Good morning ...»  
id = 126, text = «Morning. Nice to ...»  
id = 127, text = «Pleasure to meet ...»  
id = 128, text = «Dr. Sen is from ...»  
id = 129, text = «I also belong to ...»
```

Запросили **5** сообщений – пришло **5** сообщений

# Пример: загрузка сообщений с сервера

Сервер + БД с сообщениями



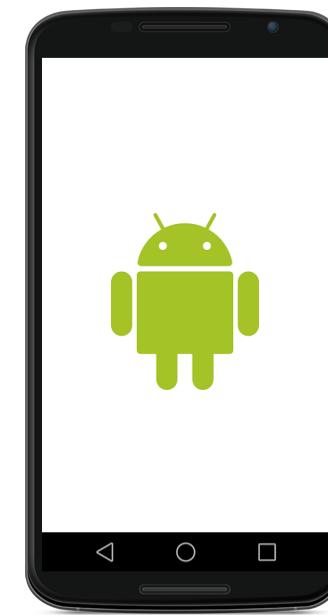
«Выдайте мне ещё 5 сообщений,  
начиная с id = 130»

\* Request \*

\* Response \*

«Пожалуйста, вот они»

Клиент

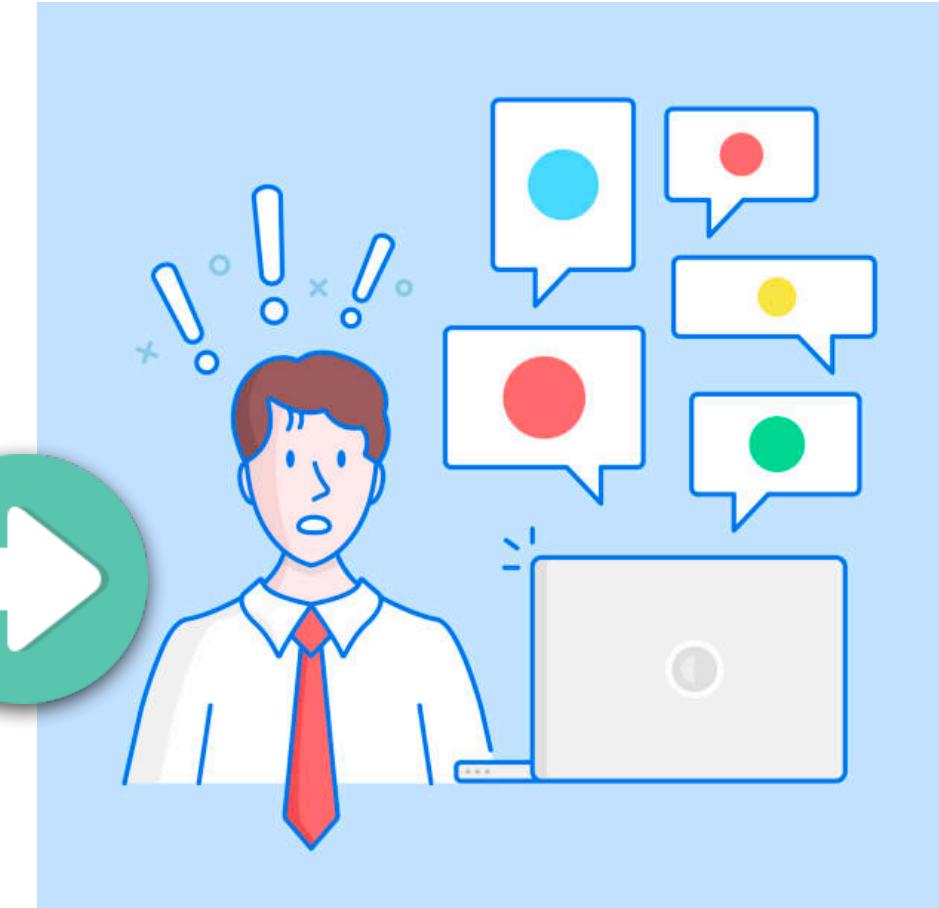


Запросили ещё 5 сообщений,  
но пришло только 2

Значит мы дошли до  
конца списка

id = 125, text = «Good morning ...»  
id = 126, text = «Morning. Nice to ...»  
id = 127, text = «Pleasure to meet ...»  
id = 128, text = «Dr. Sen is from ...»  
id = 129, text = «I also belong to ...»  
id = 130, text = «If I'm not wrong ...»  
id = 131, text = «Yes you're right ...»

# Забываем на время про сообщения

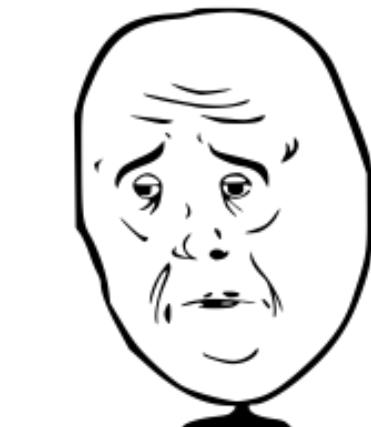


# Пока нас не было...



```
id = 125, text = ...
id = 126, text = ...
id = 127, text = ...
id = 128, text = ...
id = 129, text = ...
id = 130, text = ...
id = 131, text = ...
```

«Теперь нужно загрузить сообщения  
с **id = 132...631**  
к уже имеющимся?»



# Плохое и хорошее решение

```
id = 125, text = ...  
...  
id = 131, text = ...
```

```
id = 132, text = ...  
...  
...  
...  
...  
...  
...  
...  
...  
...  
...  
...  
id = 631, text = ...
```

+500 сообщений  
к списку

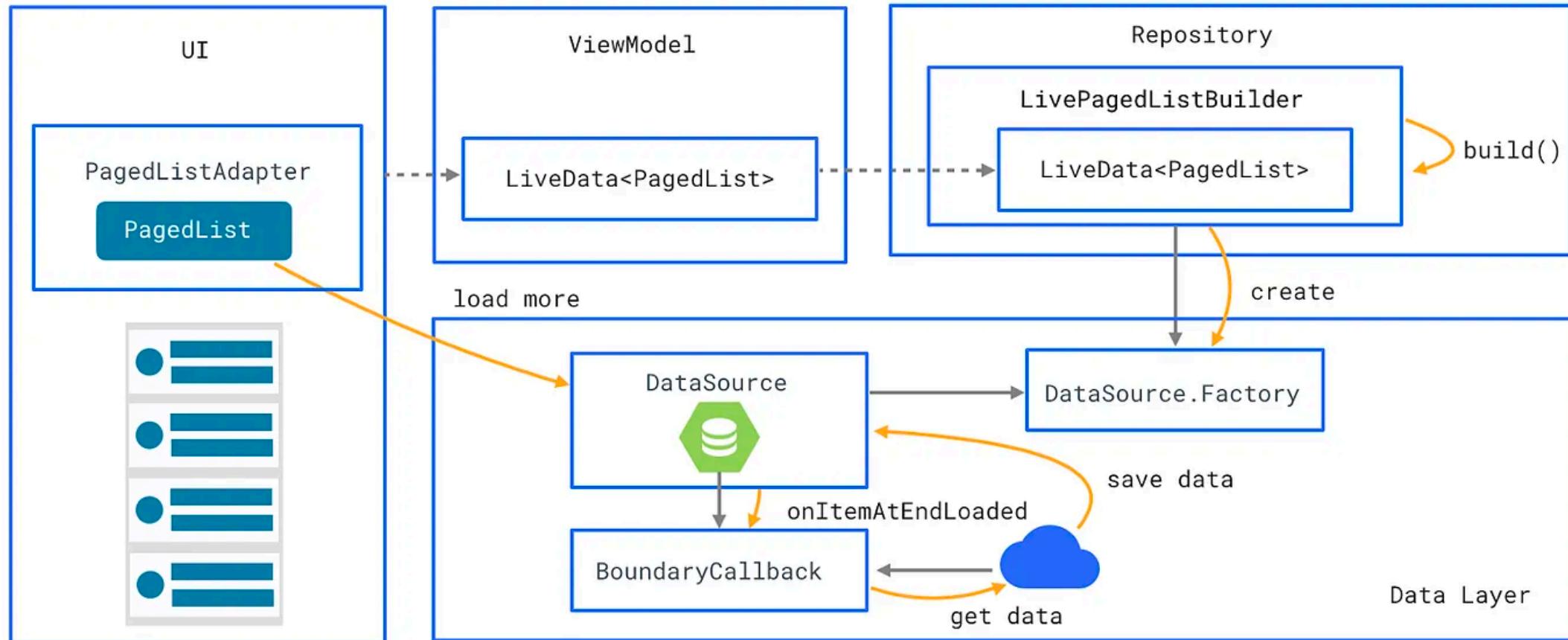
загрузить несколько,  
чтобы влезало в экран

```
id = 125, text = ...  
...  
id = 131, text = ...
```

```
id = 621, text = ...  
...  
id = 631, text = ...
```

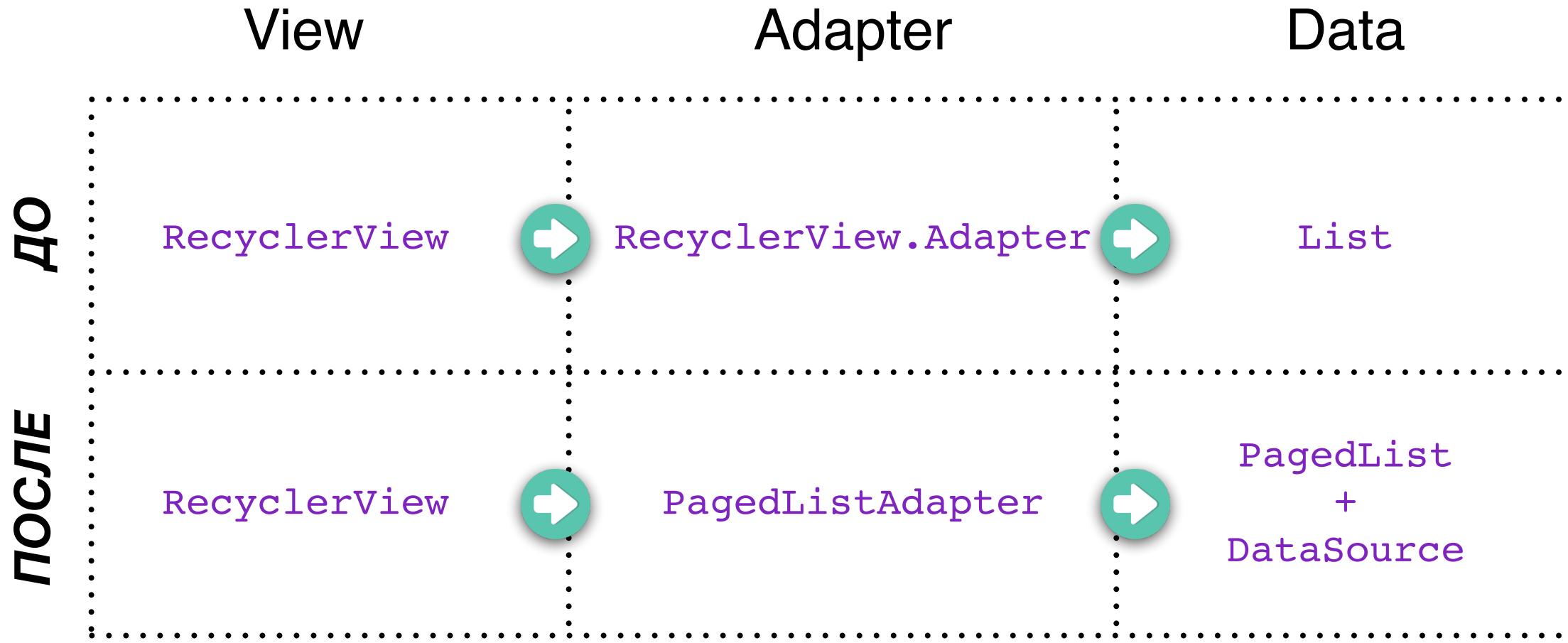


# Android Paging Library



Но пойдём по порядку...

# Реализация до и после



# DataSource и PagedList

- `DataSource` может обращаться к кэшу, БД или серверу через сетевой клиент;
- `PagedList` запрашивает данные у `DataSource`;
- Если данные меняются, то связку `PagedList` + `DataSource` нужно пересоздать;
- `PagedList.Config` определяет когда и каким образом загружать данные.

```
val config = PagedList.Config.Builder()
    .setEnablePlaceholders(false) // null-placeholder'ы не нужны
    .setInitialLoadSizeHint(20) // Кол-во элементов для загрузки при старте
    .setPageSize(10) // Кол-во элементов для последующих загрузок
    .setPrefetchDistance(5) // Сколько элементов до конца списка должно быть, чтобы запустить очередной fetch
    .build()

return PagedList.Builder(dataSource, config)
    .setFetchExecutor(Executors.newSingleThreadExecutor()) // Указываем Executor, на котором происходит fetch
    .build()
```

# Виды DataSource

- **PositionalDataSource** — запрашивает данные по **позиции**.  
Пример: данные из файла, указываем с какой строки и сколько строк грузить.
- **PageKeyedDataSource** — вместе с очередной порцией данных передает нам **ключ** для получения следующей порции данных.  
Пример: загрузка ленты с сервера.
- **ItemKeyedDataSource** — то же, что и **PageKeyedDataSource**, но мы получаем **данные**. Из данных берём **ключ**.  
Пример: выводим из БД данные, отсортированные по какому-либо ключу.

# Соберём всё вместе

```
// DataSource
MyPositionalDataSource dataSource = new MyPositionalDataSource(new EmployeeStorage());

// PagedList
PagedList.Config config = new PagedList.Config.Builder()
    .setEnablePlaceholders(false)
    .setPageSize(10)
    .build();

PagedList<Employee> pagedList = new PagedList.Builder<>(dataSource, config)
    .setMainThreadExecutor(new MainThreadExecutor())
    .setBackgroundThreadExecutor(Executors.newSingleThreadExecutor())
    .build();

// Adapter
adapter = new EmployeeAdapter(diffUtilCallback);
adapter.submitList(pagedList);

// RecyclerView
recyclerView.setAdapter(adapter);
```

# Что почитать и посмотреть?

**Create dynamic lists with RecyclerView**

<https://developer.android.com/develop/ui/views/layout/recyclerview>

**Paging library overview**

<https://developer.android.com/topic/libraries/architecture/paging/v3-overview>

**Александр Сорокин — Как происходит рендеринг экрана сообщений ВКонтакте**

[https://www.youtube.com/watch?v=GZkTwgetUWI&t=1301s&ab\\_channel=Mobius](https://www.youtube.com/watch?v=GZkTwgetUWI&t=1301s&ab_channel=Mobius)

**Ускоряем работу RecyclerView. Лучшие практики оптимизации**

<https://www.youtube.com/watch?v=o8rzzQPOo2U>

# Спасибо за внимание!

