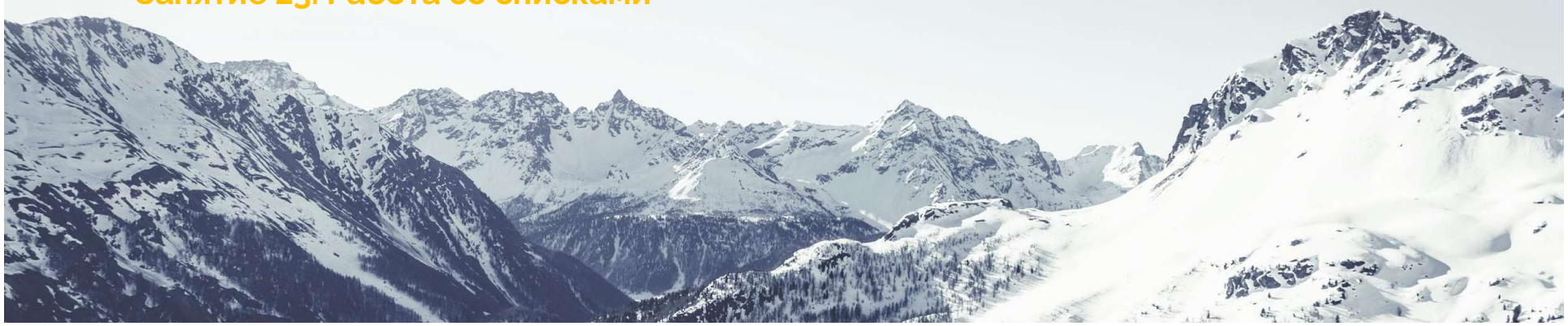

<TeachMeSkills/>

курс

Android разработчик

Занятие 23. Работа со списками



Агенда занятия

Обработка нажатия на элемент RecyclerView

Эффективное обновление списка

RecyclerView Multiple Types

▶ Обработка нажатия на элемент RecyclerView

Эффективное обновление списка

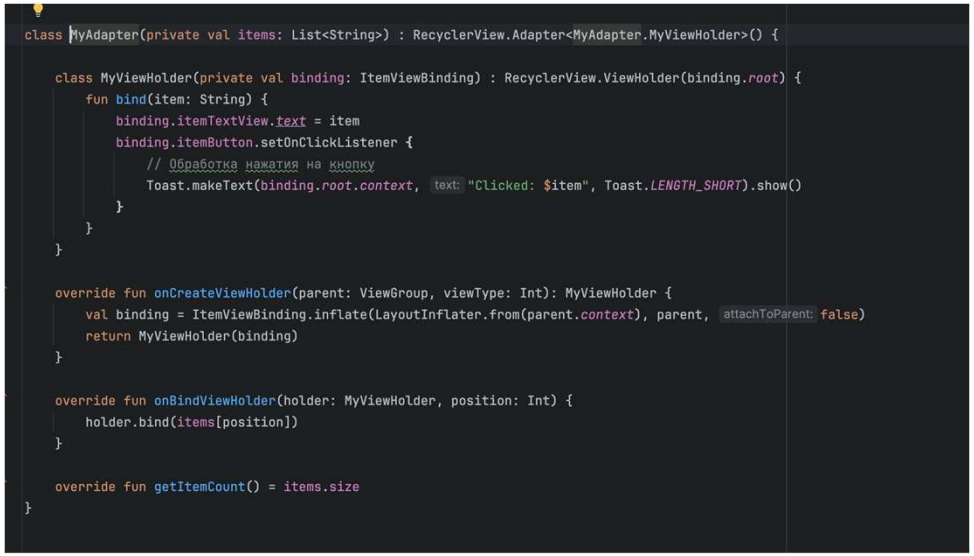
RecyclerView Multiple Types

Обработка нажатия на элемент RecyclerView

Обработку нажатия на элемент RecyclerView лучше всего добавлять в методе onBindViewHolder адаптера. Это связано с тем, что onBindViewHolder вызывается каждый раз, когда RecyclerView привязывает данные к элементу списка, и это позволяет вам настраивать поведение каждого конкретного элемента.

Обработка нажатия на элемент RecyclerView

Происходит внутри ViewHolder так же, как происходила в activity



```
class MyAdapter(private val items: List<String>) : RecyclerView.Adapter<MyAdapter.MyViewHolder>() {

    class MyViewHolder(private val binding: ItemViewBinding) : RecyclerView.ViewHolder(binding.root) {
        fun bind(item: String) {
            binding.itemTextView.text = item
            binding.itemButton.setOnClickListener {
                // Обработка нажатия на кнопку
                Toast.makeText(binding.root.context,  text: "Clicked: $item", Toast.LENGTH_SHORT).show()
            }
        }
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MyViewHolder {
        val binding = ItemViewBinding.inflate(LayoutInflater.from(parent.context), parent,  attachToParent: false)
        return MyViewHolder(binding)
    }

    override fun onBindViewHolder(holder: MyViewHolder, position: Int) {
        holder.bind(items[position])
    }

    override fun getItemCount() = items.size
}
```

Обработка нажатия на элемент RecyclerView

Эффективное обновление списка

RecyclerView Multiple Types

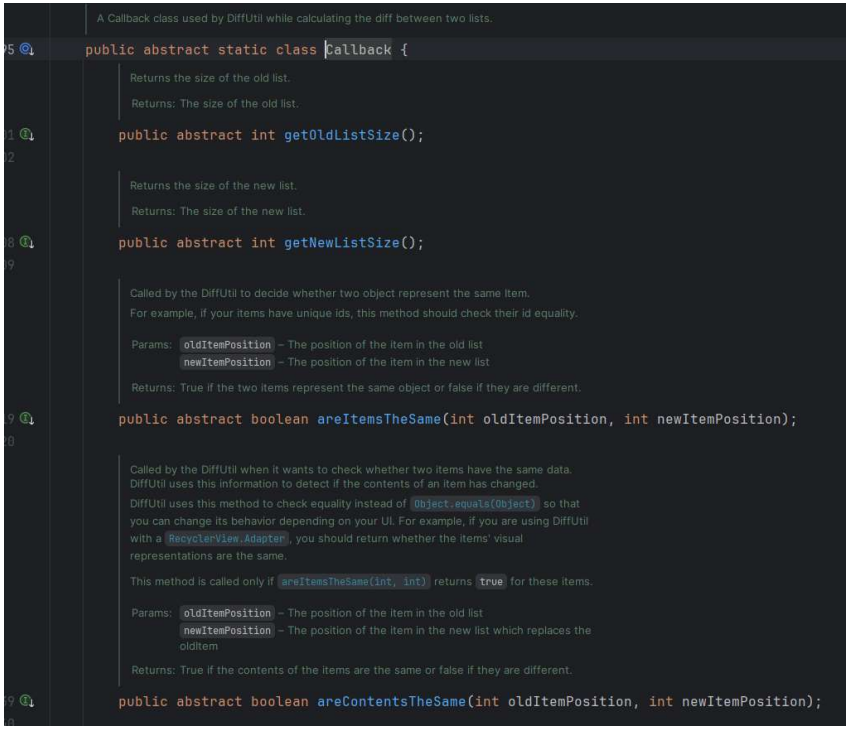
Эффективное обновление списка

DiffUtil — это утилита, которая вычисляет разницу между двумя списками и генерирует набор команд обновления для адаптера.
Под командами подразумеваются все из списка справа и *notifyDataSetChanged()*

- `notifyItemChanged`
- `notifyItemInserted`
- `notifyItemRemoved`
- `notifyItemRangeChanged`
- `notifyItemRangeInserted`
- `notifyItemRangeRemoved`

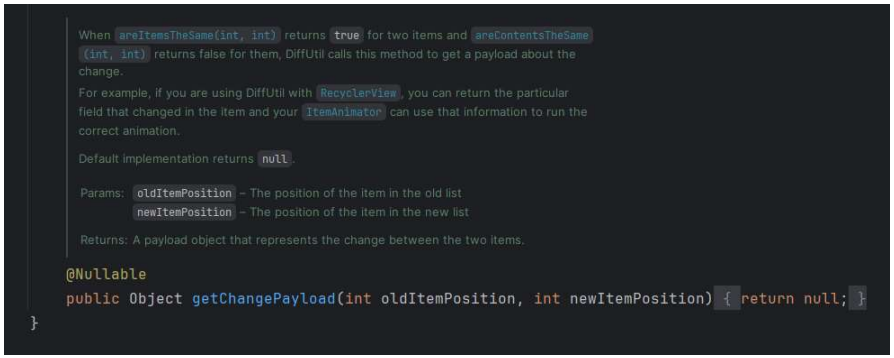
Методы класса DiffUtil

1. getOldListSize() - размер старого списка данных.
2. getNewListSize() - размер нового списка данных.
3. areItemsTheSame(int oldItemPosition, int newItemPosition) - для проверки, представляют ли два элемента один и тот же объект. Например, вы можете сравнить уникальные идентификаторы объектов.
4. areContentsTheSame(int oldItemPosition, int newItemPosition) - для проверки, имеют ли элементы одно и то же содержимое. Он вызывается только в том случае, если areItemsTheSame возвращает true.



Методы класса DiffUtil часть 2

5. getChangePayload(int oldItemPosition, int newItemPosition) [опциональный] - позволяет передать специфические данные о различиях между элементами, чтобы адаптер мог обновить только измененные части элемента. Это более оптимизированный подход, чем перерисовка всего элемента.



Настройка DiffUtil с помощью Callback

Создайте класс, наследующий DiffUtil.Callback, и реализуйте необходимые методы.

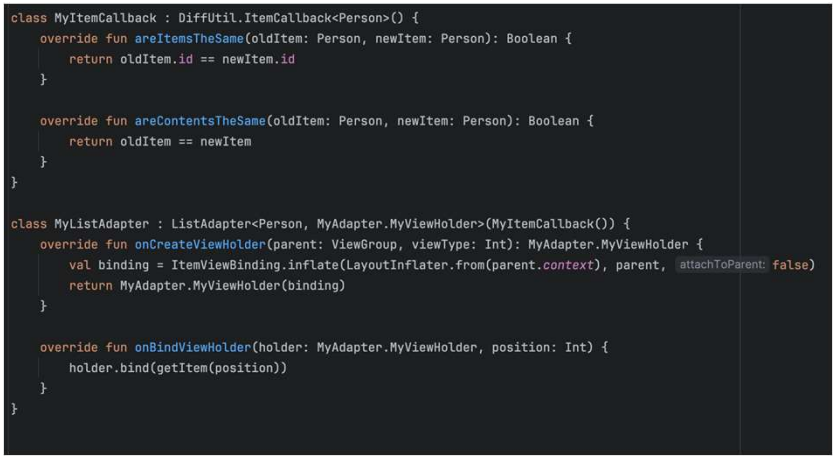
```
class MyDiffUtilCallback(private val oldList: List<Person>, private val newList: List<Person>) :
    DiffUtil.Callback() {

    override fun getOldListSize() = oldList.size

    override fun getNewListSize() = newList.size

    override fun areItemsTheSame(oldItemPosition: Int, newItemPosition: Int): Boolean {
        return oldList[oldItemPosition].id == newList[newItemPosition].id
    }

    override fun areContentsTheSame(oldItemPosition: Int, newItemPosition: Int): Boolean {
        return oldList[oldItemPosition] == newList[newItemPosition]
    }
}
```

```
fun updateList(newList: List<Person>) {
    val diffResult = DiffUtil.calculateDiff(
        MyDiffUtilCallback(
            oldList = items,
            newList = newList
        )
    )
    items.clear()
    items.addAll(newList)
    diffResult.dispatchUpdatesTo( adapter: this)
}
```

Настройка DiffUtil с помощью ItemCallback

Создайте класс, наследующий DiffUtil.ItemCallback, и реализуйте необходимые методы.

Создайте адаптер, наследующий ListAdapter.



```
class MyItemCallback : DiffUtil.ItemCallback<Person>() {
    override fun areItemsTheSame(oldItem: Person, newItem: Person): Boolean {
        return oldItem.id == newItem.id
    }

    override fun areContentsTheSame(oldItem: Person, newItem: Person): Boolean {
        return oldItem == newItem
    }
}

class MyListAdapter : ListAdapter<Person, MyAdapter.MyViewHolder>(MyItemCallback()) {
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MyAdapter.MyViewHolder {
        val binding = ItemViewBinding.inflate(LayoutInflater.from(parent.context), parent, attachToParent: false)
        return MyAdapter.MyViewHolder(binding)
    }

    override fun onBindViewHolder(holder: MyAdapter.MyViewHolder, position: Int) {
        holder.bind(getItem(position))
    }
}
```

Применение DiffUtil

```
// Настройка RecyclerView
val adapter = MyAdapter(items)
binding.recyclerView.layoutManager = LinearLayoutManager( context: this
binding.recyclerView.adapter = adapter

binding.goBack.setOnClickListener {
    adapter.updateList(items)
    MyListAdapter().submitList(items)
}
binding.update.setOnClickListener {
    adapter.updateList(updatedPeople)
    MyListAdapter().submitList(updatedPeople)
}
```

Пример работы

Смотрим проект. Git

Сравнение подхода Callback и ItemCallback

DiffUtil.Callback

- Необходимость реализации вручную: Вы должны создать свой класс, наследующий DiffUtil.Callback, и реализовать все его абстрактные методы.
- Полный контроль: Вы получаете полный контроль над тем, как сравниваются элементы и их содержимое.
- Использование с кастомными адаптерами: Чаще используется, когда вы создаете собственные адаптеры и хотите полностью контролировать обновления.

DiffUtil.ItemCallback

- Проще в использовании: Требует реализации только двух методов.
- Используется с ListAdapter: ItemCallback предназначен для использования с ListAdapter, который берет на себя выполнение вычислений DiffUtil и обновление списка.
- Меньше кода: Не нужно управлять списками вручную, так как ListAdapter делает это за вас.

Сравнение подхода Callback и ItemCallback

DiffUtil.Callback

Используйте, если вам нужен полный контроль над процессом обновления списка и вы работаете с кастомным адаптером. Это полезно, если вы хотите использовать дополнительные методы или логику обработки изменений. Используется в 90% случаев, т.к. зачастую без кастомного гибкого адаптера не реализовать задачу.

DiffUtil.ItemCallback

Используйте, если вы хотите упростить реализацию обновлений и используете ListAdapter. Это легче и более удобно для стандартных случаев, где не требуется дополнительной логики обработки изменений списка.

Сравнение подхода DiffUtil и notifyDataSetChanged()

notifyDataSetChanged()

- Вызывает полную перерисовку всех элементов в RecyclerView, независимо от того, какие элементы действительно изменились. Это приводит к вызову onBindViewHolder для каждого элемента в списке.
- Легко использовать, так как не требует сложной логики для определения изменений в данных.
- Полезно в ситуациях, когда изменений мало, или в случае быстрого прототипирования.

DiffUtil.ItemCallback

Решает все проблемы notifyDataSetChanged(), но усложняет реализацию.

Смотрим на проект и логи.

Обработка нажатия на элемент RecyclerView

Эффективное обновление списка

▶ **RecyclerView Multiple Types**

RecyclerView Multiple Types

getItemViewType()

Это основной и наиболее гибкий способ отображения разных типов элементов в RecyclerView.

Как это работает:

Переопределение getItemViewType(int position) у адаптера.

Создание разных ViewHolder для каждого типа

В onBindViewHolder, используйте holder и позицию, чтобы правильно привязать данные к каждому типу элемента.

RecyclerView Multiple Types

[Смотрим проект. Git]

```
class MultiTypeAdapter(private val data: List<Any>) :

    companion object {
        private const val TYPE_HEADER = 0
        private const val TYPE_ITEM = 1
        private const val TYPE_FOOTER = 2
    }

    override fun getItemViewType(position: Int): Int {
        return when (data[position]) {
            is Header -> TYPE_HEADER
            is Item -> TYPE_ITEM
            is Footer -> TYPE_FOOTER
            else -> throw IllegalArgumentException("Unknown type")
        }
    }
```

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): RecyclerView.ViewHolder {
    val layoutInflater = LayoutInflater.from(parent.context)
    return when (viewType) {
        TYPE_HEADER -> {
            val binding = HeaderViewBinding.inflate(layoutInflater, parent, attachToParent: false)
            HeaderViewHolder(binding)
        }

        TYPE_ITEM -> {
            val binding = ItemViewMultiBinding.inflate(layoutInflater, parent, attachToParent: false)
            ItemViewHolder(binding)
        }

        TYPE_FOOTER -> {
            val binding = FooterViewBinding.inflate(layoutInflater, parent, attachToParent: false)
            FooterViewHolder(binding)
        }

        else -> throw IllegalArgumentException("Invalid view type")
    }
}

override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position: Int) {
    when (holder) {
        is HeaderViewHolder -> holder.bind(data[position] as Header)
        is ItemViewHolder -> holder.bind(data[position] as Item)
        is FooterViewHolder -> holder.bind(data[position] as Footer)
    }
}
```

RecyclerView Multiple Types

Какие есть недостатки у этого способа?

Как можно оптимизировать этот метод и уменьшить вероятность ошибки?

Знакомство

Открытые вопросы

Опрос по теории

▶ **Практика**

Задачи

Задача 1: Нажатие на элемент списка

Реализовать список, отображающий имена студентов и их рейтинг. По нажатию на рейтинг, он должен увеличиваться на 0.01.

Задачи

Задача 2: Обновление списка с помощью notifyDataSetChanged()

Реализовать список, отображающий имена студентов.
Добавить две кнопки.
Первая преобразует имена студентов добавляя случайную цифру к имени.
Вторая возвращает списку изначальный вид.
Реализовать логирование метода bind для каждого элемента списка.

Задачи

Задача 3: Создание DiffUtil Callback

Реализация задачи 2 с помощью DiffUtil

Задачи

Задача 4: Создание DiffUtil ItemCallback

Реализация задачи 2 с помощью DiffUtil ItemCallback

Задачи

Задача 5: Список с разными элементами

Создать список способный отобразить:

- Заголовок
- Простой текст
- Кнопку

Q&A

Домашнее задание

Задачи

Задача 1: Экран со списоком различных видов постов

Экран представляет из себя список из различных элементов.
Возможные элементы:
Имя Автора и Текст под ним
Картинка и текст под ней
Текст и кнопка под ней
Должна быть возможность обновить список по кнопке
Использовать diffUtil для расчета обновлений

*использовать SwipeRefreshLayout при желании усложнить

Задачи

Задача 2*: Кастомизация списка

Для списка реализованного в первой задаче добавить отступы между эллементами 20dp
При этом последний и первый элементы должны быть без лишних отступов.

Q&A

Ваши вопросы

Спасибо