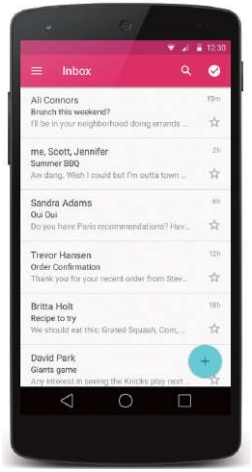
01

01-3. RecyclerView

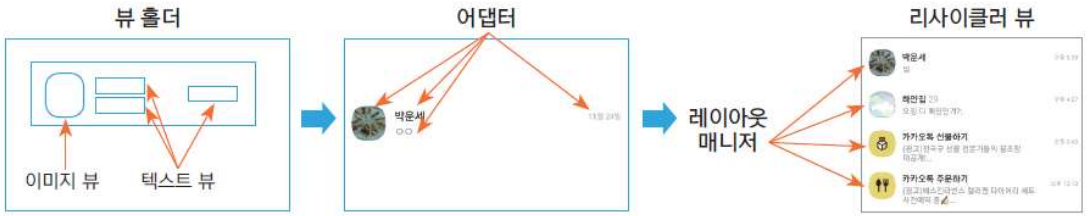
androidx 뷰 활용

11-4 리사이클러 뷰 – 목록 화면 구성

리사이클러 뷰 기초 사용법



- 구성 요소
  - ViewHolder(필수): 항목에 필요한 뷰 객체를 가집니다.
  - Adapter(필수): 항목을 구성합니다.
  - LayoutManager(필수): 항목을 배치합니다.
  - ItemDecoration(옵션): 항목을 꾸밉니다.



11-4 리사이클러 뷰 – 목록 화면 구성

- 뷰 홀더 준비
  - 각 항목에 해당하는 뷰 객체를 가지는 뷰 홀더는 RecyclerView.ViewHolder를 상속받아 작성

• 뷰 홀더 준비

```
class MyViewHolder(val binding: ItemMainBinding): RecyclerView.ViewHolder(binding.root)
```

11-4 리사이클러 뷰 – 목록 화면 구성

- 어댑터 준비
  - 각 항목을 만들어 주는 역할
  - getItemCount(): 항목 개수를 판단하려고 자동으로 호출됩니다.
  - onCreateViewHolder(): 항목의 뷰를 가지는 뷰 홀더를 준비하려고 자동으로 호출됩니다.
  - onBindViewHolder(): 뷰 홀더의 뷰에 데이터를 출력하려고 자동으로 호출됩니다.

• 어댑터 준비

```
class MyAdapter(val datas: MutableList<String>):
    RecyclerView.Adapter<RecyclerView.ViewHolder>() {
    override fun getItemCount(): Int {
        TODO("Not yet implemented")
    }
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
    RecyclerView.ViewHolder {
        TODO("Not yet implemented")
    }
    override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position: Int) {
        TODO("Not yet implemented")
    }
}
```

11-4 리사이클러 뷰 – 목록 화면 구성

• 항목의 개수 구하기

```
override fun getItemCount(): Int = datas.size
```

• 항목 구성에 필요한 뷰 홀더 객체 준비

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): RecyclerView.ViewHolder =
    MyViewHolder(ItemMainBinding.inflate(LayoutInflater.from(parent.context),
        parent, false))
```

11-4 리사이클러 뷰 – 목록 화면 구성

• 뷰에 데이터 출력

```
override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position: Int) {
    Log.d("kkang","onBindViewHolder : $position")
    val binding = (holder as MyViewHolder).binding
    // 뷰에 데이터 출력
    binding.itemData.text = datas[position]
    // 뷰에 이벤트 추가
    binding.itemRoot.setOnClickListener {
        Log.d("kkang", "item root click : $position")
    }
}
```

11-4 리사이클러 뷰 – 목록 화면 구성

- 리사이클러 뷰 출력

• 리사이클러 뷰 출력

```
class RecyclerViewActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val binding = ActivityRecyclerViewBinding.inflate(layoutInflater)
        setContentView(binding.root)
        val datas = mutableListOf<String>()
        for(i in 1..10){
            datas.add("Item $i")
        }
        binding.recyclerView.layoutManager = LinearLayoutManager(this)
        binding.recyclerView.adapter = MyAdapter(datas)
        binding.recyclerView.addItemDecoration(DividerItemDecoration(this,
        LinearLayout Manager.VERTICAL))
    }
}
```

11-4 리사이클러 뷰 – 목록 화면 구성

- 항목을 동적으로 추가 · 제거
  - 항목을 구성하는 데이터에 새로운 데이터를 추가하거나 제거한 후 어댑터의 notifyDataSetChanged() 함수를 호출

• 항목 추가

```
datas.add("new data")
adapter.notifyDataSetChanged()
```

11-4 리사이클러 뷰 – 목록 화면 구성

레이아웃 매니저

- 레이아웃 매니저는 어댑터로 만든 항목을 리사이클러 뷰에 배치
  - LinearLayoutManager: 항목을 가로나 세로 방향으로 배치합니다.
  - GridLayoutManager: 항목을 그리드로 배치합니다.
  - StaggeredGridLayoutManager: 항목을 높이가 불규칙한 그리드로 배치합니다.

- 항목을 가로·세로 방향으로 배치
  - LinearLayoutManager를 사용

• 항목을 세로로 배치

```
binding.recyclerView.layoutManager =
    LinearLayoutManager(this)
```

▶ 실행 결과

11-4 리사이클러 뷰 – 목록 화면 구성

- LinearLayoutManager의 orientation값을 Linear LayoutManager.HORIZONTAL로 지정

• 항목을 가로로 배치

```
val layoutManager = LinearLayoutManager(this)
layoutManager.orientation =
    LinearLayoutManager.HORIZONTAL
binding.recyclerView.layoutManager = layoutManager
```

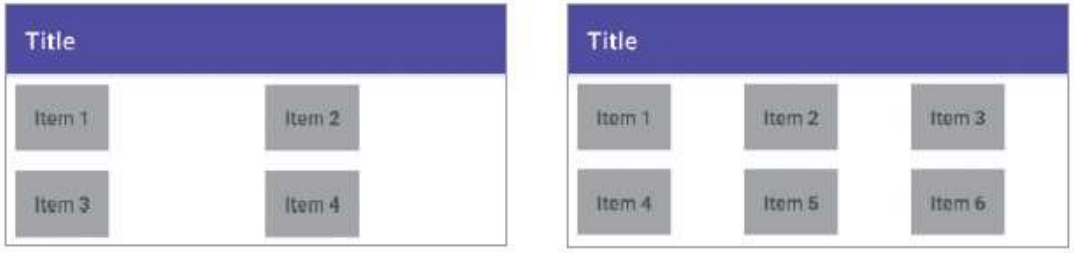
▶ 실행 결과

11-4 리사이클러 뷰 – 목록 화면 구성

- 그리드로 배치하기
  - GridLayoutManager를 이용
  - 생성자의 숫자는 그리드에서 열의 개수를 뜻합니다
  - 방향을 설정할 수 있습니다.

• 항목을 그리드로 배치

```
val layoutManager = GridLayoutManager(this, 2)
binding.recyclerView.layoutManager = layoutManager
```



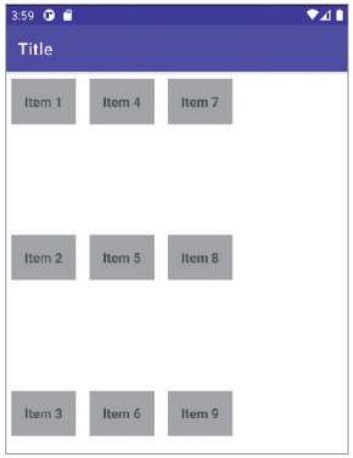
11-4 리사이클러 뷰 – 목록 화면 구성

- 그리드로 배치하기
  - 가로로 설정하려면 생성자에 GridLayoutManager.HORIZONTAL을 지정

• 그리드에서 항목을 가로로 배치

```
val layoutManager = GridLayoutManager(this, 3,
GridLayoutManager.HORIZONTAL, false)
binding.recyclerView.layoutManager = layoutManager
```

▶ 실행 결과



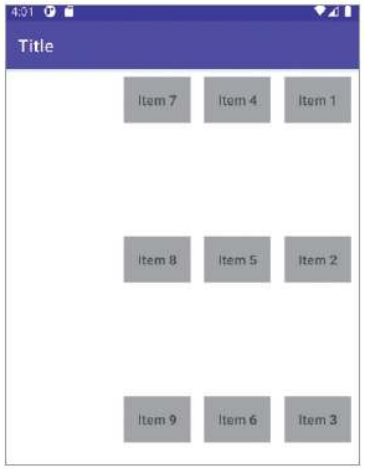
11-4 리사이클러 뷰 – 목록 화면 구성

- GridLayoutManager 생성자의 네 번째 매개변수에 Boolean값을 설정

• 그리드에서 항목을 오른쪽부터 배치

```
val layoutManager = GridLayoutManager(this, 3,
GridLayoutManager.HORIZONTAL, true)
binding.recyclerView.layoutManager = layoutManager
```

▶ 실행 결과



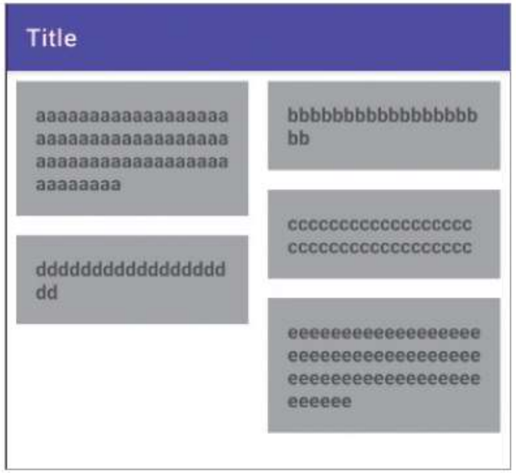
11-4 리사이클러 뷰 – 목록 화면 구성

- 높이가 불규칙한 그리드로 배치하기
  - StaggeredGridLayoutManager는 뷰의 크기가 다르면 지그재그 형태로 배치

• 지그재그 그리드 형태로 배치

```
val layoutManager = StaggeredGridLayout
Manager(2, StaggeredGridLayoutManager.
VERTICAL)
binding.recyclerView.layoutManager =
layoutManager
```

▶ 실행 결과



11-4 리사이클러 뷰 – 목록 화면 구성

아이템 데커레이션

- 아이템 데커레이션은 리사이클러 뷰를 다양하게 꾸밀 때 사용
- 항목의 구분선을 출력해 주는 DividerItem Decoration
- ItemDecoration을 상속받는 개발자 클래스를 만들고 이 클래스에서 다양한 꾸미기 작업을 합니다.
  - onDraw(): 항목이 배치되기 전에 호출됩니다.
  - onDrawOver(): 항목이 모두 배치된 후 호출됩니다.
  - getItemOffsets(): 개별 항목을 꾸밀 때 호출됩니다.

11-4 리사이클러 뷰 – 목록 화면 구성

• 아이템 데커레이션 구현

```
class MyDecoration(val context: Context): RecyclerView.ItemDecoration() {
    override fun onDraw(c: Canvas, parent: RecyclerView, state: RecyclerView.State) {
        super.onDraw(c, parent, state)
    }
    override fun onDrawOver(c: Canvas, parent: RecyclerView, state: RecyclerView.State) {
        super.onDrawOver(c, parent, state)
    }
    override fun getItemOffsets(
        outRect: Rect,
        view: View,
        parent: RecyclerView,
        state: RecyclerView.State
    ) {
        super.getItemOffsets(outRect, view, parent, state)
    }
}
```

11-4 리사이클러 뷰 – 목록 화면 구성

• 항목이 배치되기 전에 호출되는 onDraw() 함수

```
override fun onDraw(c: Canvas, parent: RecyclerView, state: RecyclerView.State) {
    super.onDraw(c, parent, state)
    c.drawBitmap(BitmapFactory.decodeResource(context.getResources(), R.drawable.stadium),
        0f, 0f, null)
```

11-4 리사이클러 뷰 – 목록 화면 구성

• 모든 항목이 배치된 후 호출되는 onDrawOver() 함수

```
override fun onDrawOver(c: Canvas, parent: RecyclerView, state: RecyclerView.State) {
    super.onDrawOver(c, parent, state)
    // 뷰 크기 계산
    val width = parent.width
    val height = parent.height
    // 이미지 크기 계산
    val dr: Drawable? = ResourcesCompat.getDrawable(context.getResources(),
        R.drawable.kbo, null)
    val drWidth = dr?.intrinsicWidth
    val drHeight = dr?.intrinsicHeight
    // 이미지가 그려질 위치 계산
    val left = width / 2 - drWidth?.div(2) as Int
    val top = height / 2 - drHeight?.div(2) as Int
    c.drawBitmap(
        BitmapFactory.decodeResource(context.getResources(), R.drawable.kbo),
        left.toFloat(),
        top.toFloat(),
        null
    )
}
```

11-4 리사이클러 뷰 – 목록 화면 구성

• 개별 항목을 꾸미는 getItemOffsets() 함수

```
override fun getItemOffsets(
    outRect: Rect,
    view: View,
    parent: RecyclerView,
    state: RecyclerView.State
) {
    super.getItemOffsets(outRect, view, parent, state)
    val index = parent.getChildAdapterPosition(view) + 1
    if (index % 3 == 0)
        outRect.set(10, 10, 10, 60) // left, top, right, bottom
    else
        outRect.set(10, 10, 10, 0)
    view.setBackgroundColor(Color.LTGRAY)
    ViewCompat.setElevation(view, 20.0f)
}
```

11-4 리사이클러 뷰 – 목록 화면 구성

- 아이템 데커레이션 객체를 리사이클러 뷰에 적용할 때는 addItemDecoration() 함수를 이용

• 리사이클러 뷰에 아이템 데커레이션 적용

```
binding.recyclerView.addItemDecoration(MyDecoration
(this))
```

▶ 실행 결과

리사이클러 뷰 – 변경사항 적용

- 특정 항목이 변경되거나 삭제, 혹은 특정 위치에 새로운 항목이 추가되기도 한다.
- 이 경우에 데이터 변경을 한후 Adapter 에게 변경사항을 적용할 것을 알려줘야 한다.
- 변경사항 적용을 알려주는 방법은 notifyXXX() 함수를 이용하는 방법과 DiffUtil 을 이용하는 방법이 있다.

- notifyDataSetChanged()
  - 전체 데이터셋이 변경되었음을 알림
  - 모든 아이템을 다시 바인딩하고 레이아웃을 재구성
  - 성능 비용이 크므로 가능하면 다른 특정 notify 함수 사용 권장
- notifyItemChanged(int position)
  - 특정 위치의 아이템이 변경되었음을 알림
  - 해당 위치의 아이템만 다시 바인딩
- notifyItemInserted(int position)
  - 특정 위치에 새 아이템이 삽입되었음을 알림
  - 삽입 애니메이션 표시

리사이클러 뷰 – 변경사항 적용

- notifyItemRemoved(int position)
  - 특정 위치의 아이템이 제거되었음을 알림
  - 제거 애니메이션 표시
- notifyItemRangeChanged(int positionStart, int itemCount)
  - 연속된 여러 아이템이 변경되었음을 알림
  - 시작 위치부터 지정된 개수만큼의 아이템 업데이트
- notifyItemRangeInserted(int positionStart, int itemCount)
  - 연속된 여러 아이템이 삽입되었음을 알림
- notifyItemRangeRemoved(int positionStart, int itemCount)
  - 연속된 여러 아이템이 제거되었음을 알림
- notifyItemMoved(int fromPosition, int toPosition)
  - 아이템이 한 위치에서 다른 위치로 이동했음을 알림

리사이클러 뷰 – 변경사항 적용

- notifyXXX 을 사용하는 경우
  - 간단한 업데이트에 직접적이고 빠르게 적용 가능한 장점이 있다.
  - 특정 아이템의 변경 위치를 정확히 알고 있을 때, 소량의 데이터만 변경될 때 유용
  - 하지만 여러 아이템이 변경되거나 복잡한 변경 패턴(삽입, 삭제, 이동 등의 조합)이 있을 때 모든 변경사항을 수동으로 추적하고 적절한 notify 메서드를 호출해야 하는 단점이 있다.

- DiffUtil
  - 복잡한 데이터 변경 사항을 자동으로 계산하여 필요한 최소한의 업데이트만 수행이 가능하다.
  - 리스트 전체가 교체되거나, 다수의 변경 사항이 있거나, 변경 패턴이 복잡할 때 유용

리사이클러 뷰 – 변경사항 적용

- DiffUtil.Callback
  - DiffUtil.Callback 을 상속받은 Callback 클래스를 작성
  - 이 클래스내에서 각 항목이 변경된 것을 판단하기 위한 함수를 오버라이드 받아 작성해야 한다.

```
class MyDiffCallback(
    private val oldList: MutableList<String>,
    private val newList: MutableList<String>
) : DiffUtil.Callback() {

    override fun getOldListSize(): Int = oldList.size
    override fun getNewListSize(): Int = newList.size

    override fun areItemsTheSame(oldItemPosition: Int, newItemPosition: Int): Boolean {

    }

    override fun areContentsTheSame(oldItemPosition: Int, newItemPosition: Int): Boolean {

    }

}
```

리사이클러 뷰 – 변경사항 적용

- 변경사항 적용
  - 변경사항 판단을 할 수 있는 Callback 을 등록하고 데이터 변경후 dispatchUpdatesTo() 함수를 호출하여 변경된 항목에 대한 작업을 지시한다.

```
val diffCallback = MyDiffCallback(data, newNumbers)
val diffResult = DiffUtil.calculateDiff(diffCallback)

// 데이터  업데이트


// 변경사항  적용
diffResult.dispatchUpdatesTo(this)
```

감사합니다

단단히 마음먹고 떠난 사람은
산꼭대기에 도착할 수 있다.
산은 올라가는 사람에게만 정복된다.

윌리엄 셰익스피어
William Shakespeare