

매니페스트 안드로이드 인터뷰 (한국어 번역판)

안드로이드 GDE의 노하우가 담긴
질문으로 학습하는 안드로이드 기술 면접 가이드



Jaewoong Eum (skydoves)

Manifest Android Interview 한국어

GDE의 노하우가 담긴 질문으로 학습하는 안드로이드 기술 면접
가이드

Jaewoong

이 책은 <https://leanpub.com/manifest-android-interview-kr>에서 구매하실 수 있습니다

이 버전은 2025-10-03에 출판되었습니다 ISBN 979-8285926436



이 책은 [Leanpub](#) 도서입니다. Leanpub은 린 퍼블리싱 프로세스를 통해 저자와 출판사에 힘을 실어줍니다. [린 퍼블리싱](#)은 가벼운 도구와 여러 차례의 반복을 통해 진행 중인 전자책을 출판하여 독자의 피드백을 받고, 올바른 책이 될 때까지 방향을 전환하며, 성공적인 책이 되면 견인력을 확보하는 과정입니다.

© 2025 Jaewoong

차례

서문 (Preface)	1
추천사 (Testimonial)	3
Manuel Vivo (Staff Android Engineer at Bumble, 前 Google Android DevRel)	3
Matt McKenna (Senior Android Engineer at Block, Android GDE)	3
Alejandra Stamato (Lead Android Engineer at HubSpot, 前 Google Android DevRel)	3
Simona Milanovic (Senior Android Developer Relations Engineer, Google Android DevRel)	4
이 책에 대하여	5
To. 면접을 준비하시는 분들께 드리는 말	7
To. 면접관 분들께 드리는 말	8
이슈 제보 및 토론	9
0. 안드로이드 면접 질문	10
카테고리 0: 안드로이드 프레임워크	11
Q) 0. 안드로이드란 무엇인가요?	11
Q) 1. 인텐트(Intent)란 무엇인가요?	15
Q) 2. PendingIntent의 목적은 무엇인가요?	17
Q) 3. Serializable과 Parcelable의 차이점은 무엇인가요?	20
Q) 4. Context란 무엇이며 어떤 유형의 Context가 있나요?	24
Q) 5. Application 클래스란 무엇인가요?	33
Q) 6. AndroidManifest 파일의 목적은 무엇인가요?	36
Q) 7. Activity 생명주기를 설명해주세요	38
Q) 8. Fragment 생명주기를 설명해주세요	45

Q) 9. Service란 무엇인가요?	51
Q) 10. BroadcastReceiver란 무엇인가요?	62
Q) 11. ContentProvider의 목적은 무엇이며, 애플리케이션 간의 안전한 데이터 공유를 어떻게 용이하게 하나요?	65
Q) 12. 구성 변경(configuration changes)을 어떻게 처리하나요?	73
Q) 13. 안드로이드는에서 메모리를 어떻게 효율적으로 관리하며, 메모리 누수(memory leaks)를 어떻게 방지하는지 설명해주세요.	75
Q) 14. ANR 이란 무엇인지, ANR이 발생하는 주요 원인은 무엇이며, 어떻게 예방할 수 있는지 설명해주세요.	77
Q) 15. 딥 링크(deep links)를 어떻게 처리하는지 설명해주세요.	79
Q) 16. 태스크(tasks)와 백 스택(back stack)이란 무엇인가요?	82
Q) 17. Bundle의 사용 목적에 대해서 설명해 주세요.	85
Q) 18. Activity 또는 Fragment 간에 데이터를 어떻게 전달하나요?	87
Q) 19. 화면 회전과 같은 구성 변경이 발생하면 Activity에 어떤 변화가 생기나요?	96
Q) 20. ActivityManager란 무엇인가요?	98
Q) 21. SparseArray를 사용하면 어떤 이점이 있나요?	101
Q) 22. 런타임 권한(runtime permissions)을 어떻게 처리하나요?	103
Q) 23. Looper, Handler, HandlerThread의 역할은 무엇인가요?	108
Q) 24. 예외(exceptions)를 어떻게 추적하나요?	111
Q) 25. 빌드 변형(build variants)과 플레이버(flavors)란 무엇인가요?	114
Q) 26. 접근성(accessibility)을 어떻게 보장하나요?	117
Q) 27. 안드로이드 파일 시스템이란 무엇인가요?	120
Q) 28. 안드로이드 런타임(ART), Dalvik, Dex 컴파일러란 무엇인가요?	122
Q) 29. APK 파일과 AAB 파일의 차이점은 무엇인가요?	125
Q) 30. R8 최적화란 무엇인가요?	127
Q) 31. 애플리케이션 용량을 어떻게 줄이나요?	129
Q) 32. 안드로이드 애플리케이션의 프로세스(process)란 무엇이며, 안드로이드 운영 체제는 이를 어떻게 관리하나요?	134
카테고리 1: 안드로이드 UI - 뷰 (Views)	139
Q) 33. View 생명주기를 설명해주세요	139
Q) 34. View와 ViewGroup의 차이점은 무엇인가요?	145
Q) 35. ViewStub이란 무엇이고, 이를 사용하여 UI 성능을 최적화해 본 경험이 있나요?	148

Q) 36. 커스텀 뷰(custom views)는 어떻게 구현하나요?	152
Q) 37. Canvas란 무엇이며 어떻게 활용하나요?	161
Q) 38. View 시스템의 무효화(invalidation)란 무엇인가요?	164
Q) 39. ConstraintLayout이란 무엇인가요?	166
Q) 40. SurfaceView 대신 TextureView는 언제 사용해야 하나요?	169
Q) 41. RecyclerView는 내부적으로 어떻게 작동하나요?	172
Q) 42. Dp와 Sp의 차이점은 무엇인가요?	182
Q) 43. 나인패치(nine-patch) 이미지의 용도는 무엇인가요?	186
Q) 44. Drawable이란 무엇이며, UI 개발에서 어떻게 사용되나요?	187
Q) 45. 안드로이드의 Bitmap이란 무엇이며, 큰 Bitmap을 효율적으로 처리하는 방법은 무엇인가요?	190
Q) 46. 애니메이션을 어떻게 구현하나요?	203
Q) 47. Window란 무엇인가요?	211
Q) 48. 웹 페이지를 어떻게 렌더링하나요?	218
카테고리 2: Jetpack 라이브러리	225
Q) 49. AppCompat 라이브러리란 무엇인가요?	225
Q) 50. Material Design Components (MDC)란 무엇인가요?	227
Q) 51. ViewBinding를 사용하면 어떤 장점이 있나요?	229
Q) 52. DataBinding의 동작 원리에 대해서 설명해주세요.	232
Q) 53. LiveData에 대해서 설명해 주세요.	239
Q) 54. Jetpack ViewModel에 대해 설명해 주세요.	247
Q) 55. Jetpack Navigation 라이브러리란 무엇인가요?	257
Q) 56. Dagger 2와 Hilt의 동작원리 및 차이점에 대해서 설명해 주세요.	262
Q) 57. Jetpack Paging 라이브러리는 어떤 메커니즘으로 동작하나요?	276
Q) 58. Baseline Profile은 앱의 성능에 어떤 이점을 가져다주나요?	281
카테고리 3: 비즈니스 로직	285
Q) 59. 장기적으로 실행되는 백그라운드 작업을 어떻게 관리하나요?	285
Q) 60. Json 형식을 객체로 어떻게 직렬화(serialize)하나요?	290
Q) 61. 원격 데이터를 가져오기 위해 네트워크 요청을 어떻게 처리하며, 효율성과 신뢰성을 위해 어떤 라이브러리나 기술을 사용하나요?	296
Q) 62. 대규모 데이터 셋을 효율적으로 로드하는 데 왜 페이징 기법이 필요하고, RecyclerView로 구현해 본 경험이 있나요?	309
Q) 63. 네트워크에서 이미지를 어떻게 가져오고 렌더링하나요?	313

Q) 64. 로컬 디바이스에 데이터를 저장하고 복원하는 방법에 대해서 설명해 주세요.	317
Q) 65. 오프라인 우선(offline-first) 아키텍처를 어떻게 설계하실 건가요?	323
Q) 66. 초기 데이터 로딩을 위한 작업을 Compose의 LaunchedEffect와 ViewModel.init() 중 어디에서 하는 것이 가장 이상적인가요?	327
1. Jetpack Compose 면접 질문	333
카테고리 0: Compose Fundamentals	335
Q) 0. Jetpack Compose의 동작 구조는 어떻게 이루어져 있나요?	335
Q) 1. Compose 페이즈(phase)에 대해 설명해 주세요,	340
Q) 2. Jetpack Compose가 선언적(declarative) UI 프레임워크라고 불리는 이유는 무엇인가요?	345
Q) 3. recomposition이란 무엇이며 언제 발생하나요? 또한 앱 성능과 어떤 관련이 있나요?	349
Q) 4. Composable 함수는 내부적으로 어떻게 작동하나요?	353
Q) 5. Jetpack Compose의 안정성(stability)이란 무엇이며, 성능과 어떤 관련이 있나요?	362
Q) 6. 안정성(stabilities) 개선을 통해 Compose 성능을 최적화한 경험이 있나요?	372
Q) 7. 컴포지션(composition)이란 무엇이며 어떻게 생성하나요?	377
Q) 8. XML 기반 프로젝트를 Jetpack Compose로 마이그레이션하는 전략에 대해서 설명해 주세요.	381
Q) 9. Compose 성능 테스트를 항상 릴리스 모드(release mode)에서 해야 하는 이유는 무엇인가요?	386
Q) 10. Jetpack Compose에서 자주 사용하시는 Kotlin 관용구(idioms)에 대해서 말씀해 주세요.	389
카테고리 1: Compose Runtime	395
Q) 11. 상태(State)란 무엇이며 이를 관리하는 데 사용되는 API는 무엇인가요?	395
Q) 12. 상태 호이스팅(state hoisting)으로 어떤 이점을 얻을 수 있나요?	398
Q) 13. remember와 rememberSaveable의 차이점은 무엇인가요?	405
Q) 14. 컴포저블 함수 내에서 안전하게 코루틴 스코프(coroutine scope)를 생성하는 방법은 무엇인가요?	413
Q) 15. 컴포저블 함수 내에서 발생하는 사이드 이펙트를 어떻게 처리하나요?	419
Q) 16. rememberUpdatedState는 왜 사용하고 어떻게 작동하나요?	424

Q) 17. produceState의 목적은 무엇이며 어떻게 작동하나요?	427
Q) 18. snapshotFlow를 사용해 본 경험이 있을까요? 사용 시 주의 사항은 무엇인가요?	432
Q) 19. derivedStateOf가 필요한 시나리오는 무엇이고, recomposition 최적화에 어떻게 도움이 되나요?	436
Q) 20. 컴포저블 함수 또는 컴포지션의 생명주기는 어떻게 되나요?	440
Q) 21. SaveableStateHolder에 대해서 설명해 주세요. 그리고 어떤 시나리오에서 유용한가요?	444
Q) 22. 스냅샷 시스템(snapshot system)이란 무엇이며 어디에 사용하나요?	448
Q) 23. State에 List 유형을 바로 넣어서 사용하면 어떤 문제가 발생하나요? mutableStateListOf와 mutableStateMapOf에 대해서 설명해 주세요.	455
Q) 24. 컴포저블 함수에서 Kotlin의 Flow를 메모리 누수 없이 안전하게 관찰하는 방법은 무엇인가요?	460
Q) 25. CompositionLocals의 역할과 목적에 대해 설명해 주세요.	463
카테고리 2: Compose UI	472
Q) 26. Modifier란 무엇인가요?	472
Q) 27. Layout이란 무엇인가요?	492
Q) 28. Box에 대해서 아는 대로 다 설명해 주세요.	497
Q) 29. Arrangement와 Alignment의 차이점에 대해서 설명해 주세요.	502
Q) 30. Painter에 대해서 설명해 주세요.	505
Q) 31. 네트워크에서 받아오는 이미지는 어떻게 로딩하나요?	508
Q) 32. UI 버벅거림(jank)을 피하면서 리스트에 수백 개의 항목을 효율적으로 렌더링하려면 어떻게 해야 하나요?	512
Q) 33. Lazy List를 활용하여 페이지네이션(pagination) 구현하는 방법을 설명해 주세요.	516
Q) 34. Canvas는 어떤 역할을 하나요?	521
Q) 35. graphicsLayer를 어떻게 활용하나요?	524
Q) 36. Jetpack Compose에서 애니메이션을 어떻게 구현하나요?	531
Q) 37. 화면 간 내비게이션을 어떻게 구현하나요?	538
Q) 38. Compose Preview는 어떻게 작동하고 사용하고 계신 Preview 어노테이션은 어떤 것이 있나요?	545
Q) 39. Compose UI 컴포넌트 또는 스크린 유닛 테스트를 작성해 본 적이 있나요? 어떤 시나리오에서 작성해 봤나요?	554

- Q) 40. 스크린샷 테스트(screenshot testing)란 무엇이며, UI 일관성을 보장하는 데 어떻게 도움이 되나요? 562

Q) 41. Jetpack Compose에서 접근성을 어떻게 보장하나요? 566

서문 (Preface)

이 책의 저자이자 안드로이드, 코틀린, 파이어베이스 분야의 Google Developer Expert(GDE)인 [엄재웅 \(skydoves¹\)](#)입니다. 책의 출판 시점인 2025년도 기준으로 80개 이상의 오픈소스 라이브러리와 프로젝트를 출시하고 운영해 왔으며, 오픈소스 라이브러리는 전 세계 개발자들에 의해 매년 총 2,000만 회 이상 다운로드되고 있습니다. 또한, 안드로이드, Jetpack Compose, 및 코틀린 관련 소식, 최신의 블로그 포스트 등을 공유하고, 각종 팁과 구독자끼리 의견을 나눌 수 있는 구독 기반 저장소인 [Dove Letter²](#)를 운영하고 있습니다.

2014년도부터 안드로이드 개발자로 첫회사를 다니기 시작하여 4개의 국내 회사들을 거쳐, 유럽/미국 기반의 회사에서 약 3년 반 동안 Senior Android Developer Relations 재직하였고, 현재는 미국 샌프란시스코에 위치한 회사에서 Senior Developer Relations 및 Engineer로 재직 중입니다. 지금까지 안드로이드 기술 면접관으로 약 500회 이상 들어간 경험이 있습니다.

저는 소프트웨어 솔루션, 특히 오픈소스 기여 및 기술 콘텐츠를 통해 세상을 더 나은 곳으로 만들자 라는 사명을 가지고, 이 책이 제 여정에서 또 하나의 새로운 출발이라고 믿습니다. 이 책이 완성될 수 있도록 그동안 저를 지지해 준 개발자 커뮤니티, 친구들, 그리고 가족들에게 깊은 감사를 표하고 싶습니다. 책 집필을 완료하는 것은 단순히 제가 성취해야 하는 하나의 이정표일 뿐만 아니라, 안드로이드 생태계의 많은 사람들에게 힘을 실어주기 위한 또 하나의 첫걸음이라 믿습니다.

이 책을 통해 여러분이 새로운 시야를 얻고, 문제 해결 능력을 기르고, 안드로이드 개발과 생태계 전반에 대한 포괄적인 이해를 넓히는데 도움이 되기를 바랍니다. 배움은 항상 계속되는 여정이어야 하며, 여러분이 극복하는 모든 도전은 여러분을 더 나은 엔지니어로 만들어 줄 것이라 믿습니다. 다가오는 기술 면접을 준비하시든, 안드로이드 지식을 학습 하여 더 나은 개발자가 되기 위해 노력하시든, 단순히 구현하는 것 그 이상을 생각하시길 권장합니다. '왜(why)'를 이해하고, 다양한 접근 방식을 실험하며, 호기심과 열정을 가지고 학습을 꾸준히 하신다면 여러분들이 목표하시는 그 이상을 성취하시리라 확신합니다.

언제나 즐거운 코딩 하시길 바라며, 여러분의 안드로이드 여정에 행운이 가득하기를 바랍니다.

¹<https://github.com/skydoves/>

²<https://github.com/doveletter/>

다!

— 엄재웅 ([skydoves](#))³

혹시나 다른 사람으로부터 이 전자책의 무료 사본을 받으셨다면. 걱정 마세요, 여러분을 꾸짖으려는 것이 아닙니다. 대신, 전세계 어딘가에서 저를 마주치게 된다면, 편하게 커피 한 잔 사주시면 좋겠습니다.

³<https://github.com/skydoves/>

추천사 (Testimonial)

Manuel Vivo¹ (Staff Android Engineer at Bumble, 前 Google Android DevRel)

“Manifest Android Interview는 이론 중심의 기술 면접을 준비하는 안드로이드 개발자들에게 필수적인 가이드입니다. 이 책은 깊이 있는 기술적 통찰력, 실용적인 예제, 그리고 세심하게 구성된 ‘Pro Tips for Mastery’ 섹션을 자연스럽게 결합하였습니다. 이 책에 담긴 지식을 통해 안드로이드 면접을 자신 있게 헤쳐나가고 성공적으로 이끄는 데 매우 귀중한 자료가 될 것입니다.”

Matt McKenna² (Senior Android Engineer at Block, Android GDE)

“Manifest Android Interview는 기본기를 다지고, 면접을 준비하며, 모범 사례를 다시 살펴보기에 완벽합니다. 명확한 구조, 쉽게 탐색 가능한 형식, 그리고 심사숙고하여 선별된 질문들은 핵심 안드로이드 개념을 배우고 복습하는 데 있어 최고의 자료입니다.”

Alejandra Stamato³ (Lead Android Engineer at HubSpot, 前 Google Android DevRel)

“세심하게 만들어진 질문, 통찰력 있는 팁, 명확한 코드 샘플을 통해 Manifest Android Interview는 핵심 안드로이드 개념(가령, 매니페스트 파일, 컴포넌트 생명주기, 인텐트, 서비스, 콘텐츠 프로바이더, 브로드캐스트 리시버, 딥링크)을 잘 설명할 뿐만 아니라, ViewModel부터 뷰 시스템, Jetpack Compose 및 다양한 모든 것에 이르기까지 안드로이드 애플리케이션을 구축하는 모든 측면을 탐색하는 데 도움을 줍니다. 여러분의 전문성 수준에 관계없이, 이 책은 모두에게 무언가를 제공합니다. 꿈의 직장을 준비하거나 단순히 우리

¹<https://bsky.app/profile/manuelvicnt.bsky.social>

²<https://bsky.app/profile/mmckenna.me>

³<https://bsky.app/profile/astamato.bsky.social>

모두가 사랑하는 플랫폼에 대한 전문성을 키우려는 경우 등 이 책은 여러분의 여정에서 귀중한 동반자가 될 것입니다.”

Simona Milanovic (Senior Android Developer Relations Engineer, Google Android DevRel)⁴

“이 책의 저자인 재웅님(안드로이드 커뮤니티에서 skydoves로 잘 알려진 분)의 신간 “매니페스트 안드로이드 인터뷰(Manifest Android Interview)”는 면접을 준비하거나 안드로이드 기술을 연마하고 싶은 모든 사람에게 필수입니다.

Compose 런타임과 UI의 기본부터 세부적인 내용까지 모든 것을 담고 있는 광범위하고 상세하며 체계적인 자료입니다. 저는 개인적으로 Jetpack Compose 부분에 특히 집중했는데, 면접 준비에 매우 유용하다고 느꼈습니다.

실제 면접 내용과 매우 유사한 방식으로, 어렵고 실용적인 “이유”와 “방법”에 대한 답을 꾸준히 제시하여 문제 해결 능력을 향상시키고 학습할 수 있도록 도와줍니다.

Compose를 처음 접하는 분이든 면접을 준비하는 분이든, 이 책은 안드로이드 지식과 면접에 대한 자신감을 확실히 높여줄 것입니다.”

참고: 추천사의 순서에는 특별한 의미가 없습니다. 단순히 Random.nextInt(4)를 돌려서 제비뽑기로 나열하였습니다.

⁴<https://bsky.app/profile/anomiss.bsky.social>

이 책에 대하여

 **Manifest Android Interview**에 오신 것을 환영합니다. 이 책은 상세한 답변이 포함된 **108개의 면접 질문, 162개의 추가 실전 질문, 50개 이상의 “Pro Tips for Mastery” 섹션**을 통해 여러분의 안드로이드 개발 전문성을 향상시키기 위해 설계된 포괄적인 가이드북입니다. 면접 질문은 주로 안드로이드 개발(프레임워크, UI, Jetpack 라이브러리, 비즈니스 로직)과 Jetpack Compose(펀더멘탈(fundamental), 런타임, UI)에 중점을 둡니다.

각 질문은 심층적인 설명을 포함하며, 안드로이드와 Jetpack Compose에 대한 체계적인 학습 경로를 안내하고 핵심 개념에 대한 이해를 돋습니다. 모든 질문의 후미에는 실제 면접 상황을 시뮬레이션하도록 설계된 **실전 질문**이 있어, 문제 해결 능력을 다듬고 기술 토론에 효과적으로 대비할 수 있습니다.

이 책에서 제공하는 내용을 넘어, 더 깊이 공부하고자 하는 분들을 위해 관련 자료와 추가적인 참고 문헌이 포함되어 있습니다. 또한, 어려울 수 있는 용어에 대해 가능한 한 각주를 제공하려 노력했으며, 이를 통해 초보자도 어려운 개념을 쉽게 파악하고 이해도를 높일 수 있도록 하였습니다.

“**Pro Tips for Mastery**” 섹션은 고급 주제를 더 깊이 다루며, 특히 Compose 챕터에서는 널리 사용되는 API들의 내부 구조를 파헤치고, 현업에서 실용성 있는 API에 대한 딥다이브를 제공하여 시니어 개발자분들 또한 흥미 있게 읽어보실 수 있습니다. 아울러 중급 개발자들은 해당 섹션을 통해 안드로이드에 대해 조금 더 깊이 이해하고, 기술적 문제에 대한 분석적인 시야를 기르는 데 좋은 자료가 될 것입니다.

이 책은 안드로이드와 Jetpack Compose의 여러 영역에 대해 **포괄적이고 체계적인 탐구 가이드**를 제시하며, 기본적인 개념과 고수준의 API 또는 라이브러리를 포괄적으로 다룹니다. 그럼에도 이 책은 안드로이드 개발의 모든 측면을 다루는 백과사전이 아니며, 하루아침에 안드로이드 개발 전문가로 만들어 준다고 주장하는 “**마스터 북**”도 아닙니다. 대신, 면접을 전략적으로 준비하고 독자가 필요로 하는 니즈와 목표에 맞춰 학습하는 데 도움이 되는 기반을 제공합니다.

또한, 이 책은 카메라 API, 블루투스 또는 저수준의 분야와 같이 특정 API에 고도로 집중된 서드파티 라이브러리나 하드웨어와 같은 주제를 다루지 않고 있습니다. 이러한 특정

분야에 대해 학습이 필요하시다면, 별도의 자료를 통해 학습을 보충하는 것을 권장합니다.
안드로이드 개발자로서의 여정과 커리어 성장 모두에 행운이 있기를 바랍니다! 

To. 면접을 준비하시는 분들께 드리는 말

Manifest Android Interview는 여러분의 안드로이드 개발 전문성을 향상시키고 안드로이드 생태계에 대한 이해를 넓히기 위해 설계된 포괄적인 가이드 북입니다. 다시 한번 강조하지만, 이 책은 하루아침에 전문성을 보장하는 “마스터 북”은 아닙니다. 핵심 주제를 광범위하게 다루지만, 각 회사와 팀과 직무마다 면접자에게 기대하는 사항이 다를 수 있으므로, 면접 과정에서 마주할 수 있는 모든 가능한 면접 질문을 다룰 것으로 기대하시면 안 됩니다.

만약 목표하는 직무가 이 책에서 다루지 않는 특정 분야의 전문성을 요구한다면, 적극적으로 추가 자료를 찾고 개별적으로 학습을 이어 나가셔야 합니다. 채용 공고, 요구되는 기술, 팀의 기대치를 바탕으로 면접 준비를 맞춤화하는 것이 가장 중요합니다. 이 책을 처음부터 끝까지 순서대로 읽기보다는, 먼저 채용 공고를 분석하고 관련 기술을 파악한 다음, 본인에게 필요한 내용을 우선적으로 책을 읽어보시는 것을 적극 권장합니다. 최대한 학습 시간을 효과적으로 사용하셨으면 좋겠습니다.

면접 형식 또한 회사마다, 팀마다, 직무마다 크게 다를 수 있습니다. Leetcode 스타일의 코딩 면접부터, 코딩 과제 또는 시스템 설계 등이 요구될 수 있습니다. 각 면접 과정에서 요구하는 바를 이해하신다면 어떤 영역에 우선순위를 두어야 할지 결정하는 데 도움이 됩니다. 이 책을 최대한 전략적으로 사용하시고, 다양한 면접 스타일과 요구 사항에 맞게 활용하시길 바랍니다.

또한, 많은 기술 면접에서 면접자가 올바르게 이해하고 있는지를 평가하기 위해 후속 질문(꼬리 질문)을 던집니다. 다가오는 면접을 준비하실 때, 이 책에 있는 질문 및 실전 질문에 대해 스스로 후속 질문이나 변형 질문을 만들어 사고하는 방식은 면접 준비에 큰 도움이 될 것입니다. 면접관의 관점에서 이 책에 접근하면 분석적 사고를 기르는데 도움이 되고, 예상치 못한 질문에 대처하는 능력을 어느 정도 기를 수 있으며, 기술 면접에서 발생하는 토론에서 자신감을 가질 수 있습니다.

이 책이 안드로이드와 Jetpack Compose에 대한 지식을 향상시키고, 더 넓은 안드로이드 생태계와 주요 영역에 대한 이해를 심화시키는 포괄적인 가이드가 되기를 바랍니다. 만약 이 책이 여러분의 다음 면접을 성공적으로 이끄는데 데 도움이 된다면, 그것만으로도 이 책의 목적은 이미 달성한 것입니다.

To. 면접관 분들께 드리는 말

만약 여러분이 새로운 면접관이거나 팀에서 만들어 놓은 면접 질문에 대한 정보가 부족하다면, 후보자의 기술적인 역량을 제대로 평가하기 위해 어떤 질문을 던져야 할지 판단이 어려울 수 있습니다. 이 책이 팀에서 적극적으로 사용하는 테크 스택, 특히 모든 팀원이 숙지해야 하는 핵심 시스템이나 자주 사용되는 기술에 맞는 면접 아이디어를 얻어갈 수 있는 유용한 참고서가 되기를 바랍니다.

이 책에 제공된 질문을 면접에서 그대로 사용하실 수도 있지만, 모든 후보자가 책에 제시된 완벽한 답변을 이야기할 것이라고 기대하기보다는, 명확한 채점 기준을 설정하는 것이 중요합니다. 예를 들어, 팀이 Jetpack ViewModel을 프로젝트 전반적으로 사용하고 있다면, 사전에 핵심 포인트나 예상 답변을 정의하는 것이 중요합니다. 이상적으로는 실제 사용 사례와 프로젝트 내에서 ViewModel이 실제로 어떻게 사용되는지를 기반으로 면접 질문을 고민해보셔야 합니다.

예를 들어, 후보자가 해당 면접 질문에 대하여 70% 만큼의 답변을 한다면, 해당 질문에 대해서는 통과한 것으로 채점한다던지와 같은 명확한 기준을 만드셔야 합니다. 또한, 많은 후보자가 면접 중에 긴장할 수 있으므로, 핵심은 같지만 뉘앙스가 다른 형태의 변형 질문을 사전에 준비해놓는다면, 후보자가 올바르게 정답을 이야기하도록 유도하는 데 도움이 될 수 있습니다. 또한, 업무 중에 면접에 들어가야 하기 때문에 귀찮으시겠지만, 최대한 편안한 환경을 조성하신다면 후보자가 자신의 능력을 더 효과적으로 발휘할 수 있습니다.

이는 후보자를 대하는 하나의 접근 방식일 뿐이며, 팀의 필요에 따라 최적화된 방법을 찾아가시는 것이 가장 이상적입니다. 면접관 조차도 모든 주제에 대해 지식을 완벽하게 이해할 수 없기 때문에, 특히 면접 진행 경험이 부족하신 경우 면접 전에 관련 주제를 다시 한번 학습하시거나 검증해 보시는 것이 중요합니다.

훌륭한 면접관이 되는 것은 훌륭한 후보자가 되는 것만큼이나 중요합니다. 미래의 동료를 선택하는 중요한 과정이기 때문입니다. 이 책이 여러분의 면접 방식을 개선하고 팀에 긍정적인 영향을 줄 최고의 후보자를 찾는 데 가치 있는 자료가 되기를 바랍니다.

이슈 제보 및 토론

책에서 내용적인 오류, 오타 또는 오래된 내용을 발견하시면, [GitHub](#)¹에서 이슈를 생성하여 제보하고 기여해 주시면 감사하겠습니다. 여러분의 피드백은 이 책과 더불어 더 넓은 개발자 생태계를 개선하는 데 크게 도움이 됩니다. 이 책은 (특히 한국어) 대부분의 플랫폼에서 전자책으로 출간될 예정이므로, 자주 업데이트될 수 있습니다.

다른 전세계의 개발자들과 소통하고 싶으시다면, 이 책의 저자가 관리하는 글로벌 [Discord 채널](#)²에 자유롭게 참여하여, 실전 면접 질문과 같은 면접 주제에 대해 또 다른 글로벌 독자 및 개발자들과 토론할 수 있습니다. 기술 토론을 하고, 모의 면접을 진행해볼 친구를 찾고, 네트워크를 확장하는데 도움이 되었으면 좋겠습니다.

¹<http://github.com/skydoves/manifest-android-interview>

²<https://discord.gg/emq2zw3YA7>

0. 안드로이드 면접 질문

안드로이드는 2008년 9월 23일 출시된 이후 상당한 발전을 거듭해왔습니다. 수년에 걸쳐 안드로이드 SDK와 생태계에는 안드로이드 아키텍처 컴포넌트(AAC), Jetpack 라이브러리, Jetpack Compose와 같은 새로운 도구와 솔루션이 꾸준히 등장하면서 다양한 변화가 있었습니다. 이러한 기술적인 변화에도 불구하고, Intent, Window, View와 같은 핵심 API와 같은 안드로이드의 기반이 되는 근본적인 시스템들은 대체로 일관성을 유지해왔습니다.

각 회사는 안드로이드 애플리케이션을 구축하기 위해 서로 다른 접근 방식과 기술 스택을 사용합니다. 따라서 채용 공고에 명시된 최소 자격 요건과 해당 조직에서 사용하는 특정 기술에 맞춰 면접을 준비 하는것이 중요합니다. 그들의 기술적 요구 사항을 이해하면 면접을 더 효과적으로 준비하실 수 있습니다.

면접 질문의 난이도는 회사와 면접관에 따라 크게 달라질 수 있습니다. 따라서, 단순히 이 책에서 제시하는 해답을 암기하기보다는 개념을 깊이 이해하고 이를 실전에 적용하는 연습을 하는 것이 좋습니다. 이 책의 질문들은 안드로이드 기술 면접을 준비하는 데 도움이 되는 자료로 사용하기 위함 뿐만 아니라, 안드로이드를 심도있게 학습하고자 하는 모든 분들께 도움이 됩니다. 하지만, 이 세상의 모든 면접 질문을 담은 백과사전이나 교과서로 생각하시면 안 됩니다.

어떤 회사는 안드로이드 심층적인 이해도를 평가하기 위해 저수준의 안드로이드 시스템 및 아키텍처에 초점을 맞출 수 있는 반면, 다른 회사는 이러한 기술을 활용하여 실전에서 얼마나 빨리 코딩할 수 있는지를 판단하기 위해 고수준 API나 라이브러리에 대한 질문에 집중할 수 있습니다. 채점 기준 또한 조직마다 다르므로 모든 질문에 대한 단 하나의 “완벽한” 답은 없습니다. 여기에 제공된 답변은 하나의 가이드라인일 뿐이며, 더 성공적인 면접 준비를 원하신다면 이를 확장하고 더 깊은 지식을 탐구하는 것을 권장합니다.

이 책은 방대한 안드로이드 개발 분야 내에서 나올 수 있는 모든 면접 질문을 다루지 않습니다. 대신, 면접을 효과적으로 준비하고 지원하는 역할의 요구 사항에 맞춰 학습을 효과적으로 하는 데 도움이 됩니다. 또한, 카메라 API, 블루투스 또는 저수준 API와 같이 특정 주제에 초점을 맞추는 서드파티 라이브러리나 하드웨어 기능에 대해서는 깊이 다루지 않는다는 점을 유의하시길 바랍니다. 목표로하는 조직에서 특정 영역에 대한 기술적 이해도를 요구한다면, 별도로 자료를 찾아보시고 스스로 지식을 확장해야 합니다.

카테고리 0: 안드로이드 프레임워크

안드로이드 프레임워크는 안드로이드 애플리케이션을 구축하는 데 필요한 기초 지식입니다. 이는 안드로이드 SDK, 내부 구조, 안드로이드 런타임 시스템 등을 포함합니다. 이러한 구성 요소는 안드로이드의 기본 시스템에 근본적으로 통합되어 있으므로, 안드로이드 개발자가 되기 위한 첫 단계로서 프레임워크를 이해하는 것이 필수적입니다.

이 카테고리에서는 Activity, Fragment, 안드로이드 UI를 포함한 안드로이드 SDK의 핵심 측면과 함께, 최신 안드로이드 개발에 유용한 Jetpack 라이브러리와 같은 추가 지식 영역을 살펴볼 것입니다. 또한 실제 시나리오에서 안드로이드 애플리케이션을 실행하는 데 필수적인 안드로이드 시스템과 비즈니스 로직에 대해 배우게 될 것입니다.

다시 한번 말씀드리지만, 이 책은 전 세계의 모든 가능한 면접 질문을 다루는 것을 목표로 하지 않는다는 점을 명심해야 합니다. 마찬가지로, 제공된 답변이 모든 면접관이 기대하는 완벽한 해답으로서 취급되어서는 안 됩니다. 대신, 통찰력을 얻고 안드로이드 프레임워크에 대한 더 깊은 이해를 향한 학습 여정을 안내하는 참고 자료로 간주하시길 바랍니다.

Q) 0. 안드로이드란 무엇인가요?

안드로이드는 스마트폰과 태블릿과 같은 모바일 기기를 위해 주로 설계된 오픈소스 운영 체제입니다. 구글에 의해 개발 및 유지 관리되며 리눅스 커널에 기반합니다. 안드로이드는 광범위한 하드웨어 구성과 기기를 지원하는 강력하고 유연한 플랫폼을 제공합니다.

안드로이드 OS의 주요 특징

- 오픈 소스 및 커스텀화:** 안드로이드는 오픈 소스([Android Open Source Project^{1\)}](https://source.android.com/)) 이므로 개발자와 제조업체가 필요에 맞게 수정하고 커스텀할 수 있습니다. 이러한 유연성은 웨어러블, TV, IoT 기기를 포함한 다양한 기기에서의 광범위한 채택과 혁신에 기여했습니다.
- SDK를 이용한 애플리케이션 개발:** 안드로이드 앱은 주로 Java 또는 Kotlin과 안드로이드 소프트웨어 개발 키트(SDK)를 사용하여 개발됩니다. 개발자는 Android Studio와 같은 툴을 사용하여 플랫폼용 애플리케이션을 설계, 개발 및 디버그할 수 있습니다.

¹<https://source.android.com/>

3. **풍부한 앱 생태계:** Google Play Store는 안드로이드의 공식 앱 배포 플랫폼으로, 게임부터 생산성 도구까지 다양한 카테고리에 걸쳐 수백만 개의 앱을 제공합니다. 개발자는 서드파티 스토어나 직접 다운로드를 통해 앱을 독립적으로 배포할 수도 있습니다.
4. **멀티태스킹 및 리소스 관리:** 안드로이드는 멀티태스킹을 지원하여 사용자가 여러 앱을 동시에 실행할 수 있게 합니다. 관리형 메모리 시스템과 효율적인 가비지 컬렉션을 사용하여 다양한 기기에서 성능을 최적화합니다.
5. **다양한 하드웨어 지원:** 안드로이드는 저가형 휴대폰부터 프리미엄 플래그십 모델에 이르기까지 광범위한 기기를 구동하며, 다양한 화면 크기, 해상도 및 하드웨어 구성과 광범위한 호환성을 제공합니다.

안드로이드 아키텍처

안드로이드 플랫폼 아키텍처²는 모듈식으로 계층화되어 있으며, 여러 구성 요소로 이루어져 있습니다.

²<https://developer.android.com/guide/platform>

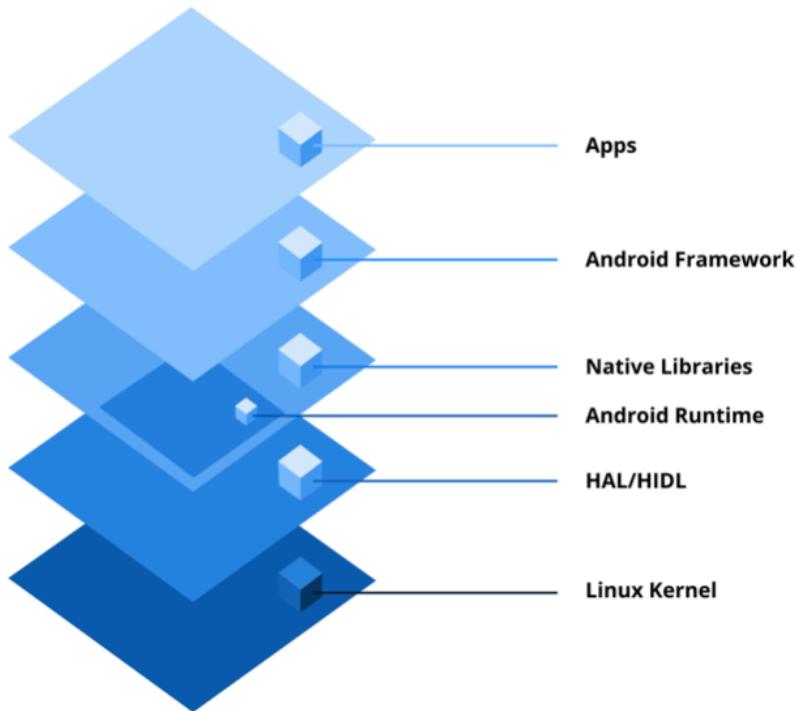


그림 1. android-architecture

- **리눅스 커널 (Linux Kernel):** Linux Kernel은 안드로이드 운영 체제의 기반을 형성합니다. 하드웨어 추상화를 처리하여 소프트웨어와 하드웨어 간의 원활한 상호 작용을 보장합니다. 주요 책임에는 메모리 및 프로세스 관리, 보안 강화, Wi-Fi, Bluetooth, 디스플레이와 같은 하드웨어 구성 요소의 장치 드라이버 관리가 포함됩니다.
- **하드웨어 추상화 계층 (Hardware abstraction layer (HAL)):** Hardware Abstraction Layer (HAL)은 안드로이드의 Java API 프레임워크를 기기 하드웨어에 연결하는 표준 인터페이스를 제공합니다. 이는 라이브러리 모듈로 구성되며, 각 모듈은 카메라나 Bluetooth와 같은 특정 하드웨어 구성 요소에 맞춰져 있습니다. 프레임워크 API가 하드웨어 접근을 요청하면, 안드로이드 시스템은 해당 HAL 모듈을 동적으로 로드하여 요청을 처리합니다.
- **안드로이드 런타임 및 코어 라이브러리 (Android Runtime (ART) 및 Core Libraries):** Android Runtime (ART)은 Kotlin이나 Java에서 컴파일된 바이트코드를 사용하여 애플리케이션을 실행하는 실행 환경입니다. Core Libraries는 기본적인 기능과 서비스를 제공하는 라이브러리 집합입니다.

리케이션을 실행합니다. ART는 최적화된 성능을 위해 Ahead-of-Time (AOT)³ 및 Just-in-Time (JIT)⁴ 컴파일을 지원합니다. 핵심 라이브러리는 데이터 구조, 파일 조작, 스레딩 등을 위한 필수 API를 제공하여 앱 개발을 위한 포괄적인 환경을 제공합니다.

- **네이티브 C/C++ 라이브러리 모음 (Native C/C++ Libraries)**: 안드로이드는 중요한 기능을 지원하기 위해 C 및 C++로 작성된 네이티브 라이브러리 모음을 포함합니다. OpenGL과 같은 라이브러리는 그래픽 렌더링을 관리하고, SQLite는 데이터베이스 작업을 가능하게 하며, WebKit은 웹 콘텐츠 표시를 용이하게 합니다. 이러한 라이브러리는 안드로이드 프레임워크와 애플리케이션에서 성능 집약적인 작업을 위해 직접 사용됩니다.
- **안드로이드 프레임워크 (Android Framework (APIs))**: 애플리케이션 프레임워크 계층은 앱 개발을 위한 고수준 서비스와 API를 제공합니다. 여기에는 개발자가 안드로이드 애플리케이션을 구축할 수 있게 해주는 ActivityManager, NotificationManager, Content Providers 등이 포함됩니다. 이 계층은 개발자가 안드로이드 시스템 기능을 효율적으로 활용할 수 있도록 지원합니다.
- **애플리케이션 (Applications)**: 최상위 계층에는 시스템 앱(예를 들어, 연락처나 설정 앱 등)과 안드로이드 SDK를 사용하여 생성된 서드파티 앱을 포함한 모든 유저 기반의 앱이 포함됩니다. 이러한 앱은 사용자에게 다양한 기능을 제공하고, 각종 시스템들을 원활하게 운용하기 위해 하위 계층과 통신합니다.

요약

안드로이드는 전 세계적으로 가장 널리 사용되는 모바일 운영 체제이며, 여전히 높은 시장 점유율을 차지하고 있습니다. 이는 혁신을 촉진하고 개발자가 전세계 수십억 명의 사용자를 위한 애플리케이션을 만들 수 있는 기회를 제공합니다. 안드로이드의 적응성과 오픈 소스 특성 덕분에 다양한 시장에서 번성하고 스마트폰을 넘어 수많은 기기의 기반을 형성할 수 있었습니다.

실전 질문

Q) 안드로이드 플랫폼 아키텍처는 Linux Kernel, Android Runtime (ART), Hardware Abstraction Layer (HAL) 등 여러 계층으로 구성됩니다. 이 구성 요소들이 애플리케이션

³Ahead-of-Time (AOT) 컴파일은 코드가 런타임 전에 기계 코드로 컴파일되어 실행 중 Just-In-Time (JIT) 컴파일이 필요 없는 프로세스입니다. 해당 접근 방식은 최적화된 사전 컴파일된 바이너리를 생성하여 성능을 향상시키고 런타임 오버헤드를 줄입니다.

⁴Just-In-Time (JIT) 컴파일은 바이트코드가 실행 직전에 동적으로 기계 코드로 변환되는 런타임 프로세스입니다. 이를 통해 런타임 환경은 실제 실행 패턴을 기반으로 코드를 최적화하여 자주 사용되는 코드 경로의 성능을 향상시킬 수 있습니다.

실행과 하드웨어와의 상호 작용을 위해 어떻게 작동하는지 설명해 주세요.

Q) 1. 인텐트(Intent)란 무엇인가요?

Intent⁵는 수행될 작업에 대한 추상적인 설명입니다. 이는 Activity, Service, BroadcastReceiver가 통신할 수 있도록 하는 메시징 객체 역할을 합니다. Intent는 일반적으로 Activity를 시작하거나, 브로드캐스트를 보내거나, Service를 시작하는 데 사용됩니다. 또한 컴포넌트 간에 데이터를 전달할 수 있어 안드로이드의 시스템에서 근본이 되는 요소입니다.

안드로이드에는 명시적(explicit) 및 암시적(implicit) 두 가지 유형의 Intent가 있습니다.

1. 명시적 Intent (Explicit Intent)

- **정의:** 명시적 Intent는 호출할 컴포넌트(Activity 또는 Service)를 직접 이름으로 지정하여 정확히 명시합니다.
- **사용 사례:** 명시적 Intent는 대상 컴포넌트를 알고 있을 때 사용됩니다 (가령, 앱 내의 특정 Activity 시작).
- **시나리오:** 동일한 앱 내에서 한 Activity에서 다른 Activity로 전환하는 경우 명시적 Intent를 사용합니다.

아래 코드처럼 명시적 Intent를 사용할 수 있습니다.

그림 2. Explicit Intent.kt

```
1 val intent = Intent(this, TargetActivity::class.java)
2 startActivity(intent)
```

2. 암시적 Intent (Implicit Intent)

- **정의:** 암시적 Intent는 특정 컴포넌트를 지정하지 않고 수행할 일반적인 작업을 선언합니다. 시스템은 액션(action), 카테고리(category), 데이터(data)를 기반으로 어떤 컴포넌트가 Intent를 처리할 수 있는지 결정합니다.

⁵<https://developer.android.com/reference/android/content/Intent>

- **사용 사례:** 암시적 Intent는 다른 앱이나 시스템 컴포넌트가 처리할 수 있는 작업을 수행하려 할 때 유용합니다 (가령, URL 열기 또는 콘텐츠 공유).
- **시나리오:** 브라우저에서 웹 페이지를 열거나 다른 앱과 콘텐츠를 공유하는 경우 암시적 Intent를 사용합니다. 시스템이 Intent를 처리할 앱을 결정합니다.

아래 예시처럼 암시적 Intent를 사용할 수 있습니다.

그림 3. Implicit Intent.kt

```
1 val intent = Intent(Intent.ACTION_VIEW)  
2 intent.data = Uri.parse("https://www.example.com")  
3 startActivity(intent)
```

요약

명시적 Intent는 대상 컴포넌트가 알려진 내부 앱 내비게이션에 사용됩니다. 반면에 암시적 Intent는 대상을 직접 지정하지 않고 외부 앱이나 다른 컴포넌트가 처리할 수 있는 작업에 사용됩니다. 이는 안드로이드 생태계를 더 유연하게 만들고 앱들이 원활하게 상호 작용할 수 있도록 합니다.

실전 질문

Q) 명시적 인텐트와 암시적 인텐트의 차이점은 무엇이며, 각각 어떤 시나리오에서 사용해야 하나요?

Q) 안드로이드 시스템은 암시적 인텐트를 처리할 앱을 어떻게 결정하며, 적합한 애플리케이션을 찾지 못하면 어떻게 되나요?

💡 Pro Tips for Mastery: 인텐트 필터(Intent Filters)란 무엇인가요?

안드로이드의 **intent filter**⁶는 앱 컴포넌트가 링크 열기나 브로드캐스트 처리와 같은 특정 Intent에 어떻게 응답할 수 있는지를 정의합니다. 이는 Activity, Service 또는 BroadcastReceiver가 처리할 수 있는 Intent 유형을 선언하는 필터 역할을 하며,

⁶<https://developer.android.com/guide/components/intents-filters>

AndroidManifest.xml 파일에 명시됩니다. 각 intent filter는 들어오는 Intent와 정확히 일치시키기 위해 액션, 카테고리 및 데이터 유형을 포함할 수 있습니다. intent filter를 적절하게 정의하면 앱이 다른 앱 및 시스템 컴포넌트와 원활하게 상호 작용하여 기능을 향상시킬 수 있습니다.

암시적 Intent가 전송되면, 안드로이드 시스템은 Intent의 속성을 설치된 앱의 매니페스트 파일에 정의된 intent filter와 비교하여 실행할 적절한 컴포넌트를 결정합니다. 일치하는 항목이 발견되면 시스템은 해당 컴포넌트를 시작하고 Intent 객체를 전달합니다. 여러 컴포넌트가 Intent와 일치하는 경우, 시스템은 사용자에게 선택 대화 상자를 표시하여 작업을 처리하는 데 선호하는 앱을 선택할 수 있도록 합니다.

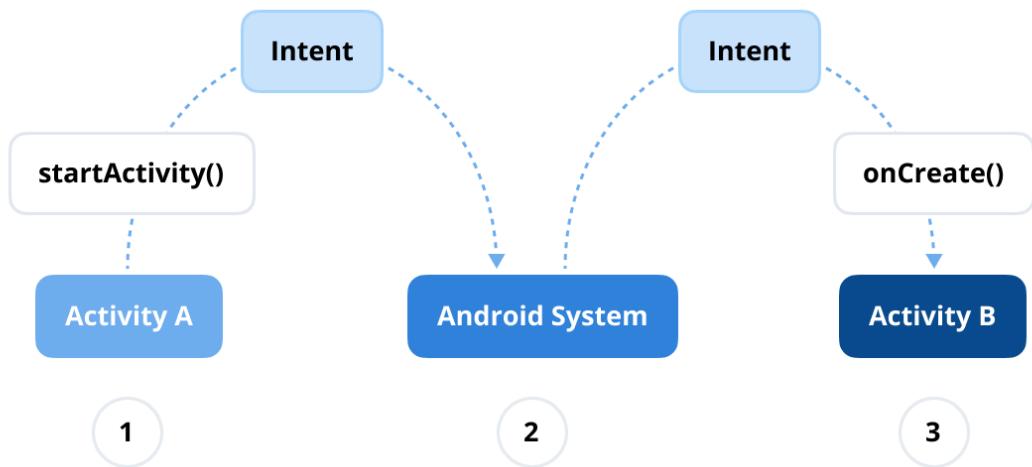


그림 4. intent-filter

Q) 2. PendingIntent의 목적은 무엇인가요?

PendingIntent 는 다른 애플리케이션이나 시스템 컴포넌트가 애플리케이션을 대신하여 미리 정의된 Intent를 나중에 실행할 수 있는 권한을 부여하는 또 다른 종류의 Intent입니다. 이는 알림이나 서비스와의 상호작용과 같이 앱의 수명 주기를 벗어나 트리거되어야 하는

작업에 특히 유용합니다.

PendingIntent의 주요 특징

PendingIntent는 일반 Intent의 래퍼(wrapper) 역할을 하여 앱의 생명주기를 넘어서 지속될 수 있도록 합니다. 이는 여러분의 앱과 동일한 권한으로 다른 앱이나 시스템 서비스에 Intent 실행을 위임합니다. PendingIntent는 Activity, Service 또는 BroadcastReceiver를 위해 생성될 수 있습니다.

PendingIntent는 세 가지 주요 형태로 사용될 수 있습니다.

- **Activity**: Activity를 시작합니다.
- **Service**: Service를 시작합니다.
- **Broadcast**: 브로드캐스트를 보냅니다.

PendingIntent.getActivity(), PendingIntent.getService(), 또는 PendingIntent.getBroadcast()와 같은 팩토리 메서드를 사용하여 PendingIntent를 생성할 수 있습니다.

그림 5. Creating a PendingIntent for a Notification

```
1 val intent = Intent(this, MyActivity::class.java)
2 // FLAG_IMMUTABLE 또는 FLAG_MUTABLE 플래그 지정 필수 (Android 12+)
3 val pendingIntentFlags = if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.S) {
4     PendingIntent.FLAG_UPDATE_CURRENT or PendingIntent.FLAG_IMMUTABLE
5 } else {
6     PendingIntent.FLAG_UPDATE_CURRENT
7 }
8 val pendingIntent = PendingIntent.getActivity(
9     this,
10    0, // requestCode
11    intent,
12    pendingIntentFlags
13 )
14
15 val notification = NotificationCompat.Builder(this, CHANNEL_ID)
```

```
16     .setContentTitle("Title")
17     .setContentText("Content")
18     .setSmallIcon(R.drawable.ic_notification)
19     .setContentIntent(pendingIntent) // 알림을 터쳤을 때 트리거됨
20     .setAutoCancel(true) // 터치하면 알림 자동 삭제
21     .build()
22
23 NotificationManagerCompat.from(this).notify(NOTIFICATION_ID, notification)
```

PendingIntent는 동작 방식과 시스템 또는 다른 컴포넌트와의 상호 작용 방식을 제어하는 다양한 플래그를 지원합니다.

- **FLAG_UPDATE_CURRENT**: 기존 PendingIntent를 새 데이터로 업데이트합니다.
- **FLAG_CANCEL_CURRENT**: 새 PendingIntent를 만들기 전에 기존 PendingIntent를 취소합니다.
- **FLAG_IMMUTABLE**: PendingIntent를 변경 불가능하게 만들어 수신자가 수정하는 것을 방지합니다. (Android 12+에서 중요)
- **FLAG_ONE_SHOT**: PendingIntent가 한 번만 사용될 수 있도록 보장합니다.

사용 사례

1. **노티피케이션 (Notifications)**: 사용자가 노티피케이션을 터쳤을 때 Activity를 여는 것과 같은 작업을 허용합니다.
2. **알람 (Alarms)**: AlarmManager를 사용하여 작업을 예약합니다.
3. **서비스 (Services)**: 백그라운드 작업을 위해 ForegroundService 또는 BroadcastReceiver에 작업을 위임합니다.

보안 고려 사항

악의적인 앱이 기본 Intent를 수정하는 것을 방지하기 위해 PendingIntent에는 항상 FLAG_IMMUTABLE을 설정해야 합니다. 이는 특정 시나리오에서 FLAG_IMMUTABLE가 필수인 안드로이드 12 (API 레벨 31)부터 특히 중요합니다. (FLAG_MUTABLE을 명시적으로 사용해야 하는 경우 주의해서 사용해야 합니다.)

요약

PendingIntent는 앱이 활성 상태가 아닐 때에도 앱과 시스템 컴포넌트 또는 다른 앱 간의 원활한 통신을 가능하게 하는 안드로이드의 핵심 메커니즘입니다. 플래그와 권한을 신중하게 관리함으로써 지연된 작업의 안전하고 효율적인 실행을 보장할 수 있습니다.

실전 질문

Q) PendingIntent란 무엇이며 일반 Intent와 어떻게 다른가요? PendingIntent 사용이 필요한 시나리오를 제시해 줄 수 있나요?

Q) 3. Serializable과 Parcelable의 차이점은 무엇인가요?

안드로이드에서 Serializable과 Parcelable은 모두 다른 컴포넌트(가령, Activity나 Fragment) 간에 데이터를 전달하는 데 사용되는 메커니즘이지만, 성능과 구현 측면에서 다르게 작동합니다. 다음은 이 둘을 비교한 것입니다.

Serializable

- **Java 표준 인터페이스 (Java Standard Interface)**: Serializable은 객체를 바이트 스트림으로 변환하여 Activity 간에 전달하거나 디스크에 쓸 수 있도록 하는 표준 Java 인터페이스입니다.
- **리플렉션 기반 (Reflection-Based)**: Java 리플렉션을 통해 작동합니다. 즉, 시스템이 런타임에 클래스와 필드를 동적으로 검사하여 객체를 직렬화합니다.
- **성능**: Serializable은 리플렉션이 느린 프로세스이기 때문에 Parcelable에 비해 느립니다. 또한 직렬화 중에 많은 임시 객체를 생성하여 메모리 오버헤드를 증가시킵니다.
- **사용 사례**: Serializable은 성능이 중요하지 않거나 안드로이드 특정 코드가 아닌 코드베이스를 다룰 때 유용합니다.

Parcelable

- **안드로이드 기반 인터페이스 (Android-Specific Interface)**: Parcelable은 안드로이드 컴포넌트 내에서 고성능 프로세스 간 통신(IPC⁷)을 위해 특별히 설계된 안드로이드 특정 인터페이스입니다.
- **성능**: Parcelable은 안드로이드에 최적화되어 있고 리플렉션에 의존하지 않기 때문에 Serializable보다 빠릅니다. 많은 임시 객체 생성을 피하여 가비지 컬렉션을 최소화합니다.
- **사용 사례**: Parcelable은 성능이 중요한 안드로이드 데이터 전달, 특히 IPC나 Activity 또는 Service 간 데이터 전달에 선호됩니다.

최신 안드로이드 개발에서는 [kotlin-parcelize plugin⁸](#)이 구현을 자동으로 생성하여 Parcelable 객체를 만드는 과정을 단순화합니다. 이 접근 방식은 이전의 수동 메커니즘에 비해 더 효율적입니다. 클래스에 @Parcelize 어노테이션을 붙이기만 하면 플러그인이 필요한 Parcelable 구현을 생성합니다. 다음은 작동 방식을 보여주는 예시입니다.

그림 6. Parcelable with @Parcelize

```

1 import kotlinx.parcelize.Parcelize
2 import android.os.Parcelable
3
4 @Parcelize
5 class User(val firstName: String, val lastName: String, val age: Int) : Parcelable

```

이 설정을 사용하면 writeToParcel과 같은 메서드를 재정의하거나 CREATOR를 구현할 필요가 없어 보일러 플레이트 코드를 크게 줄이고 가독성을 향상시킵니다.

⁷프로세스 간 통신(Inter-process communication, IPC)은 서로 다른 프로세스가 서로 통신하고 데이터를 공유할 수 있도록 하는 메커니즘으로, 별도의 애플리케이션이나 시스템 서비스 간의 협업을 가능하게 합니다. 안드로이드에서는 **Binder**, **Intents**, **ContentProviders**, **Messenger**와 같은 컴포넌트를 통해 IPC가 이루어지며, 이는 프로세스 간 데이터 교환을 안전하고 효율적으로 가능하게 합니다.

⁸<https://plugins.gradle.org/plugin/org.jetbrains.kotlin.plugin.parcelize>

💡 Additional Tips: @Parcelize로 표기된 클래스가 원시 타입(primitive type)이 아니고, @Parcelize로 표기되지 않은 클래스를 프로퍼티로 포함하고 있다면, 다음 오류가 발생합니다. “**Type is not directly supported by ‘Parcelize’.** Annotate the parameter type with ‘@RawValue’ if you want it to be serialized using ‘writeValue()’.”

이는 Parcelize 컴파일러 플러그인이 직렬화 중에 모든 속성을 평면화하려고 하기 때문에 발생하는 문제이며, 지원되지 않거나 인식되지 않는 유형은 명시적으로 표시해야 합니다. 해당 문제를 해결하려면 모든 프로퍼티의 타입이 원시 타입 또는 @Parcelize 표기된 타입 이거나, 혹은 수동으로 직렬화를 처리하기를 원한다면 프로퍼티에 @RawValue 어노테이션을 추가하고 수동으로 직렬화 로직을 추가해야 합니다.

주요 차이점.

기능	Serializable	Parcelable
유형	표준 Java 인터페이스	안드로이드에 특화된 인터페이스
성능	느림, 리플렉션 사용	빠름, 안드로이드에 최적화됨
가비지 생성	더 많은 가비지 생성 (더 많은 객체)	더 적은 가비지 생성 (효율적)
사용 사례	일반적인 Java 사용에 적합	안드로이드, 특히 IPC에 선호됨

요약

일반적으로 안드로이드 애플리케이션의 경우, 대부분의 사용 사례에서 더 나은 성능 때문에 **Parcelable** 이 권장되는 접근 방식입니다.

- 더 간단한 경우나 성능에 중요하지 않은 작업을 처리할 때, 또는 안드로이드 컴포넌트 간에 데이터 교환 목적 등이 아닌 용도로 사용할 때는 Serializable을 사용할 수 있습니다.

- 성능이 중요한 안드로이드 기반 컴포넌트와 소통할 때는 **Parcelable**을 사용해야 합니다. 안드로이드의 IPC 메커니즘에 훨씬 더 효율적입니다.

실전 질문

Q) 안드로이드에서 **Serializable**과 **Parcelable**의 차이점은 무엇이며, 일반적으로 컴포넌트 간 데이터 전달에 **Parcelable**이 선호되는 이유는 무엇인가요?

Pro Tips for Mastery: **Parcel**과 **Parcelable**이란 무엇인가요?

Parcel은 안드로이드에서 애플리케이션의 다른 컴포넌트(가령, Activity, Service 또는 BroadcastReceiver) 간의 고성능 프로세스 간 통신(IPC)을 가능하게 하는 컨테이너 클래스입니다. 주로 데이터를 마샬링⁹(직렬화, flattening)하고 언마샬링¹⁰(역직렬화, unflattening)하여 안드로이드의 IPC 경계를 넘어서 전달할 수 있도록 사용됩니다.

Parcel은 프로세스 간 통신(IPC) 메커니즘을 통해 직렬화된 데이터와 라이브 **IBinder**¹¹ 객체에 대한 참조를 모두 보내는 데 사용되는 컨테이너입니다. 고성능 IPC 전송을 위해 설계되었으며, 객체(Parcelabe 인터페이스 사용)를 효율적으로 직렬화하고 컴포넌트 간에 전달할 수 있게 합니다. **Parcel**은 범용 직렬화 도구가 아니며 영구 저장에 사용해서는 안 됩니다. 내부 구현이 변경될 수 있어 이전 데이터를 읽을 수 없게 만들 수 있기 때문입니다.

API에는 기본 데이터 유형, 배열 및 **Parcelable** 객체를 읽고 쓰는 다양한 메서드가 포함되어 있어 객체가 자신을 직렬화하고 필요할 때 재구성할 수 있습니다. 또한, 클래스 정보 쓰기를 생략하는 **Parcelable** 작업에 최적화된 메서드가 있어, 효율적인 데이터 처리를 위해 읽는 쪽에서 미리 유형을 알아야 합니다.

Parcelable은 객체를 직렬화하여 **Parcel**을 통해 전달할 수 있도록 하는 안드로이드에 특화된 인터페이스입니다. **Parcelable**을 구현하는 객체는 **Parcel**에 쓰고 복원할 수 있어 안드로이드 컴포넌트 간에 복잡한 데이터를 전달하는 데 적합합니다.

⁹마샬링(Marshaling)은 객체나 데이터 구조를 네트워크를 통해 전송하거나 저장한 후 나중에 재구성할 수 있는 형식으로 변환하는 프로세스입니다. 안드로이드에서는 일반적으로 **Binder**와 같은 메커니즘으로 전송하기 위해 데이터가 직렬화되는 프로세스 간 통신(IPC)에서 사용됩니다.

¹⁰언마샬링(Unmarshaling)은 직렬화된 형식의 데이터나 객체를 원래 형태로 다시 되돌리는 프로세스입니다. 안드로이드에서는 **Binder**와 같은 메커니즘을 통해 전송된 데이터가 수신 프로세스에서 사용하기 위해 역직렬화되는 프로세스 간 통신(IPC)에서 자주 발생합니다.

¹¹**IBinder**는 프로세스 간 통신(IPC)을 위한 핵심 안드로이드 인터페이스입니다. 클라이언트와 서비스와 같은 다른 컴포넌트 간의 저수준 통신 브리지 역할을 하여 원격으로 데이터를 교환하거나 메서드를 호출하여 상호 작용할 수 있도록 합니다.

요약

Parcel은 IPC를 사용하여 컴포넌트 간에 데이터를 전송하기 위한 컨테이너이며 다양한 데이터 유형을 지원합니다. Parcelable은 객체를 효율적인 전송을 위해 Parcel로 직렬화(flatten)할 수 있게 하는 인터페이스입니다. Parcel의 실제 구현 및 작동 메커니즘에 대해 더 자세히 알아보려면 AOSP - `Parcel.java`¹²에서 소스 코드를 탐색할 수 있습니다.

Q) 4. Context란 무엇이며 어떤 유형의 Context가 있나요?

Context는 애플리케이션의 환경 또는 상태를 나타내며 애플리케이션별 리소스 및 클래스에 대한 접근을 제공합니다. 앱과 안드로이드 시스템 간의 브릿지 역할을 하여 컴포넌트가 리소스, 데이터베이스, 시스템 서비스 등에 접근할 수 있도록 합니다. Context는 Activity 실행, 애셋(assets) 접근 또는 레이아웃 인플레이션(inflation)과 같은 작업에 필수적인 컴포넌트입니다.

안드로이드에는 아래와 같이 여러 유형의 Context가 있습니다.

Application Context (애플리케이션 컨텍스트)

Application Context는 애플리케이션의 라이프 사이클과 연결되어 있습니다. 현재 Activity나 Fragment와 독립적인 전역적이고 오래 지속되는 Context가 필요할 때 사용됩니다. 이 Context는 `getApplicationContext()`를 호출하여 획득할 수 있습니다.

Application Context의 사용 사례:

- SharedPreferences나 데이터베이스와 같은 애플리케이션 전체 리소스 접근하는 경우.
- 전체 앱 생명주기 동안 지속되어야 하는 BroadcastReceiver를 등록하는 경우.
- 앱 생명주기 동안 유지되는 라이브러리나 컴포넌트를 초기화하는 경우.

Activity Context (액티비티 컨텍스트)

Activity Context (Activity의 this 인스턴스)는 Activity의 생명주기와 연결되어 있습니다. Activity에 특정한 리소스 접근, 다른 Activity 시작, 레이아웃 인플레이션에 사용됩니다.

Activity Context의 사용 사례:

¹²<https://android.googlesource.com/platform/frameworks/base/+/27f592d/core/java/android/os/Parcel.java>

- UI 컴포넌트를 생성 또는 업데이트하는 경우.
- 다른 Activity 실행하는 경우.
- 현재 Activity 범위에 있는 리소스나 테마에 접근하는 경우.

Service Context (서비스 컨텍스트)

Service Context는 Service의 생명주기와 연결되어 있습니다. 주로 네트워크 작업 수행이나 음악 재생과 같은 백그라운드에서 실행되는 작업에 사용됩니다. Service에 필요한 시스템 수준 서비스에 대한 접근을 제공합니다.

Broadcast Context (브로드캐스트 컨텍스트)

Broadcast Context는 BroadcastReceiver가 호출될 때 제공됩니다. 이는 수명이 짧으며 일반적으로 특정 브로드캐스트에 응답하는 데 사용됩니다. 따라서, Broadcast Context로 장기적인 태스크를 수행하면 안 됩니다.

Context의 일반적인 사용 사례

1. **리소스 접근:** Context는 getString() 또는 getDrawable()과 같은 메서드를 사용하여 문자열, 드로어블(Drawable), 치수(Dimension)와 같은 리소스에 대한 접근을 제공합니다.
2. **레이아웃 인플레이션:** LayoutInflater를 사용하여 XML 레이아웃을 뷰로 인플레이션하는 데 Context를 사용합니다.
3. **액티비티 및 서비스 시작:** Activity(startActivity())와 Service(startService())를 시작하려면 Context가 필요합니다.
4. **시스템 서비스 접근:** Context는 getSystemService()를 통해 ClipboardManager 또는 ConnectivityManager와 같은 시스템 수준 서비스에 대한 접근을 제공합니다.
5. **데이터베이스 및 SharedPreferences 접근:** SQLite 데이터베이스나 SharedPreferences와 같은 영구 저장 메커니즘에 접근하는 데 Context를 사용합니다.

요약

Context는 앱과 시스템 리소스 간의 상호 작용을 가능하게 하는 안드로이드의 핵심 구성 요소입니다. Application Context, Activity Context, Service Context, Broadcast

Context와 같은 다양한 유형의 Context가 존재하며, 각각 다른 목적을 가지고 있습니다. Context를 적절하게 사용하면 효율적인 리소스 관리를 보장하고 메모리 누수나 크래시를 방지할 수 있으므로 올바른 Context를 선택해야하고 불필요하게 유지하지 않는 것이 중요합니다.

실전 질문

Q) 안드로이드 애플리케이션에서 올바른 유형의 Context를 사용하는 것이 왜 중요하며, Activity Context에 대해 오랜 참조를 유지하는 것은 잠재적으로 어떤 문제를 발생시킬 수 있나요?

Pro Tips for Mastery: Context 사용 시 주의할 점은 무엇인가요?

Context는 안드로이드에서 리소스 접근에 편리한 메커니즘이지만, 부적절하게 사용하면 메모리 누수, 크래시 또는 비효율적인 리소스 처리와 같은 심각한 문제를 일으킬 수 있습니다. 이러한 상황을 막기 위해 Context를 효과적으로 사용하기 위한 모범 사례를 이해하는 것이 중요합니다.

가장 흔한 문제 중 하나는 Context, 특히 Activity 또는 Fragment Context에 대한 참조를 해당 생명주기 보다 오래 유지하는 것입니다. 이는 가비지 컬렉터가 Context 또는 관련 리소스에 대한 메모리를 회수할 수 없게 하므로 **메모리 누수**로 이어질 수 있습니다.

가령, 아래 예제 코드는 메모리 누수를 유발합니다.

그림 7. Object Class Example.kt

```

1 object Singleton {
2     var context: Context? = null // 컨텍스트를 유지하여 메모리 누수를 유발합니다.
3 }
```

Context가 필요한 오래 지속되는 객체에는 Application 컨텍스트를 사용하셔야 합니다.

그림 8. Object Class Example.kt

```

1 object Singleton {
2     lateinit var applicationContext: Context
3 }
```

위와 같은 이유로 적절한 유형의 Context를 사용하는 것이 중요합니다. Context의 유형마다 서로 다른 목적을 가지고 있습니다. 잘못된 유형을 사용하면 예기치 않은 동작이 발생할 수 있습니다.

- 레이아웃 인플레이션이나 다이얼로그 표시와 같은 UI 관련 작업에는 Activity Context를 사용하는 것이 적합합니다.
- 라이브러리 초기화와 같이 UI 생명주기와 독립적인 작업에는 Application Context를 사용하는 것이 적합합니다.

아래 예는 Application 컨텍스트를 잘못 사용하는 사례를 보여줍니다.

그림 9. AlertDialog.kt

```
1 // Application Context는 테마에 대한 리소스 정보가 없으므로 AlertDialog 사용에 부적합합니다.  
2 val dialog = AlertDialog.Builder(context.applicationContext) // 잘못됨
```

대신 Activity Context를 사용하여 테마가 올바르게 적용되도록 해야 합니다.

그림 10. AlertDialog.kt

```
1 val dialog = AlertDialog.Builder(activityContext) // 올바름
```

또 다른 고려 사항은 관련 컴포넌트(가령, Activity 또는 Fragment)가 소멸된 후 Context를 사용하지 않는 것입니다. 소멸된 컴포넌트에 연결된 Context에 접근하면 해당 Context에 연결된 리소스가 더 이상 존재하지 않을 수 있으므로 크래시나 예상하지 못한 동작이 발생할 수 있습니다. 아래 예시는 커스텀 뷰가 부적절하게 생성되는 Context의 오용 사례를 보여줍니다.

그림 11. Button.kt

```
1 // 액티비티 참조를 유지하는 버튼 (메모리 누수 가능성)  
2 val button = Button(activity)  
3 activity.finish() // 액티비티는 소멸되었지만 버튼은 참조를 유지합니다.
```

백그라운드 스레드에서 Context 사용 피하기

Context는 메인 스레드용으로 설계되었으며, 특히 리소스에 접근하거나 UI와 상호 작용할 메인 스레드에 의존합니다. 백그라운드 스레드에서 사용하면 예기치 않은 크래시나 스레딩 관련 문제가 발생할 수 있습니다. 백그라운드에서 작업하는 도중에 UI 관련 Context 리소스와 상호 작용이 필요하다면, 반드시 메인 스레드로 다시 전환해야 합니다.

그림 12. Context Example.kt

```
1 viewModelScope.launch {
2     val data = fetchData()
3     // UI 업데이트는 메인 스레드에서 수행해야 합니다.
4     withContext(Dispatchers.Main) {
5         Toast.makeText(context, "Data fetched", Toast.LENGTH_SHORT).show()
6     }
7 }
```

요약

Context를 효과적으로 사용하려면 상황에 따라 적절한 Context 유형을 잘 선택하는 것이 중요합니다. 메모리 누수로 이어질 수 있으므로 Activity 또는 Fragment의 Context를 해당 생명주기를 넘어 참조를 유지하지 않아야 합니다. 항상 특정 작업에 맞는 올바른 유형의 Context를 선택하고, 백그라운드 스레드나 관련 컴포넌트가 소멸된 후에는 사용하지 않아야 합니다. 또한 익명 내부 클래스나 콜백은 의도치 않게 Context에 대한 참조를 유지할 수 있으므로 주의해야 합니다. Context를 적절하게 관리하면 효율적인 리소스 사용을 보장하고 메모리 누수나 애플리케이션 크래시를 방지하는 데 도움이 됩니다.

💡 Pro Tips for Mastery: ContextWrapper란 무엇인가요?

ContextWrapper는 Context를 상속받고 있는 클래스로, Context 객체를 감싸서(wrapping) 래핑된 Context에 대한 호출을 위임하는 기능을 제공합니다. 이는 원본 Context의 동작을 수정하거나 확장하기 위한 중간 계층 역할을 합니다. ContextWrapper를 사용하면 Context와 직접적인 소통을 하지 않고도 특정 기능을 커스텀할 수 있습니다.

ContextWrapper의 목적

ContextWrapper는 기존 Context의 특정 동작을 개선시키거나 재정의해야 할 때 사용됩니다. Context에 대한 호출을 중개하고 추가 기능이나 커스텀 동작을 제공하는 목적으로 많이 사용됩니다.

사용 사례

- 커스텀 컨텍스트**: 앱 전체에 다른 테마를 적용하거나 리소스를 특수한 방식으로 처리하는 등 특정 목적을 위한 커스텀 Context를 생성해야 하는 경우.
- 동적 리소스 처리**: 문자열, 치수(dimension) 또는 스타일(style)과 같은 리소스를 동적으로 제공하거나 수정하기 위해 Context를 래핑하는 경우.
- 의존성 주입**: Dagger나 Hilt와 같은 라이브러리는 의존성 주입을 위해 커스텀 ContextWrapper를 생성하고¹³, 컴포넌트에 해당 ContextWrapper를 Context 타입으로 제공합니다.

ContextWrapper 예제

아래 코드는 ContextWrapper를 사용하여 커스텀 테마를 적용하는 방법을 보여줍니다.

그림 13. CustomThemeContextWrapper.kt

```

1 class CustomThemeContextWrapper(base: Context) : ContextWrapper(base) {
2     private var theme: Resources.Theme? = null
3
4     override fun getTheme(): Resources.Theme {
5         if (theme == null) {
6             theme = super.getTheme()
7             theme?.applyStyle(R.style.CustomTheme, true) // 커스텀 테마 적용
8         }
9         return theme!!
10    }

```

¹³<https://github.com/google/dagger/blob/6b183f85e50c7b0e5e524e57d2f4561786d146cf/java/dagger/hilt/android/internal/managers/FragmentComponentManager.java#L103>

```
11
12     override fun setTheme(themeResId: Int) {
13         // 테마 리소스 ID 설정 시 내부 테마 초기화
14         theme = null
15         super.setTheme(themeResId)
16     }
17 }
```

해당 커스텀 ContextWrapper를 아래와 같이 Activity에서 사용할 수 있습니다.

그림 14. MyActivity.kt

```
1 class MyActivity : AppCompatActivity() {
2     override fun attachBaseContext(newBase: Context) {
3         super.attachBaseContext(CustomThemeContextWrapper(newBase))
4     }
5 }
```

위의 예제에서 CustomThemeContextWrapper는 Activity에 커스텀 테마를 적용하여 액티비티에서 제공하는 컨텍스트의 디폴트 동작을 재정의합니다.

주요 이점

- **재사용성**: 커스텀 로직을 래퍼 클래스에 캡슐화하고 여러 컴포넌트에서 재사용할 수 있습니다.
- **캡슐화**: 원본 Context 구현을 변경하지 않고 동작을 개선시키거나 필요에 맞게 재정의 합니다.
- **호환성**: 이미 존재하던 Context 객체와 원활하게 작동하여 호환성을 유지합니다.

요약

ContextWrapper는 안드로이드에서 Context 동작을 커스텀하기 위한 유연하고, 재사용 가능한 API입니다. 개발자가 원본 Context를 직접 변경하지 않고 호출을 위임하여 Context의 동작을 재정의 할 수 있게 하고, 적응 가능한 애플리케이션을 개발하는데 데 유용합니다.

💡 Pro Tips for Mastery: Activity에서 this와 baseContext 인스턴스의 차이점은 무엇인가요?

Activity에서 this와 baseContext는 모두 Context를 반환한다는 부분에서 비슷하지만, 서로 다른 목적을 가지고 있으며 안드로이드 컨텍스트 계층에서 다른 수준을 나타냅니다. 개발 시 혼선이나 잠재적인 문제를 피하기 위해 각각의 사용 목적에 대해서 이해하는 것이 중요합니다.

Activity에서의 this

Activity에서 this 키워드는 Activity 클래스의 현재 인스턴스를 참조합니다. Activity는 ContextWrapper의 하위 클래스(따라서 간접적으로 Context의 하위 클래스)이므로, this는 생명주기 관리 및 UI와의 상호 작용과 같은 추가 기능을 포함하여 Activity와 상호작용이 가능한 API를 호출할 수 있습니다.

Activity에서 this를 호출하면 Activity의 현재 컨텍스트를 참조하므로, 해당 Activity에서 제공하는 고유한 메서드를 호출할 수 있습니다. 가령, 다른 Activity를 시작하거나, 해당 Activity에 종속된ダイ얼로그를 띄워야 하는 경우 this를 사용할 수 있습니다.

그림 15. Using ‘this’ in an Activity

```
1 val intent = Intent(this, AnotherActivity::class.java)
2 startActivity(intent)
3
4 val dialog = AlertDialog.Builder(this)
5     .setTitle("Example")
6     .setMessage("This dialog is tied to this Activity instance.")
7     .show()
```

Activity에서의 baseContext

baseContext는 Activity가 구축되는 기반 또는 “기본” Context를 나타내며, 이는 Activity가 상속하고 있는 ContextWrapper 클래스의 일부입니다. baseContext는 Context 메서드에 대한 핵심 구현을 제공하는 ContextImpl 인스턴스이기도 합니다.

baseContext는 일반적으로 getBaseContext() 메서드를 통해 획득할 수 있습니다. 보통 직접 사용하는 경우는 드물지만, 커스텀 ContextWrapper을 작업을 하거나 ContextWrapper가 가지고 있는 원본 Context를 참조해야 할 때 사용하는 경우가 있습니다.

그림 16. Accessing baseContext in an Activity

```
1 val systemService = baseContext.getSystemService(Context.LAYOUT_INFLATER_SERVICE)
```

this와 baseContext의 주요 차이점

- 범위(Scope):** this는 현재 Activity 인스턴스와 그 생명주기를 나타내는 반면, baseContext는 Activity가 구축된 저수준의 Context를 참조합니다.
- 사용법(Usage):** this는 일반적으로 다른 Activity를 시작하거나 다이얼로그를 띄우는 작업과 같이 Activity의 생명주기나 UI와 관련된 작업에 사용됩니다. baseContext는 주로 커스텀 ContextWrapper를 구현하는 시나리오에서 Context의 핵심 구현체와 상호 작용할 때 사용됩니다.
- 계층(Hierarchy):** baseContext는 Activity의 기반 Context입니다. baseContext에 접근하면 Activity가 ContextWrapper로서 제공하는 API에 대해 우회적인 접근을 할 수 있습니다.

예제: 커스텀 ContextWrapper

커스텀 ContextWrapper를 작업할 때 this와 baseContext의 구분이 중요합니다. this 키워드는 늘 Activity 자체를 참조하는 반면, baseContext는 원본이 수정되지 않은 Context에 대한 접근을 제공합니다.

그림 17. Custom ContextWrapper and baseContext

```

1 class CustomContextWrapper(base: Context) : ContextWrapper(base) {
2     override fun getSystemService(name: String): Any? {
3         // Example: Modify the LayoutInflater
4         if (name == Context.LAYOUT_INFLATER_SERVICE) {
5             val inflater = super.getSystemService(name) as LayoutInflater
6             // 'this'는 CustomContextWrapper 인스턴스를 참조합니다.
7             return inflater.cloneInContext(this)
8         }
9         // 다른 서비스는 baseContext에서 가져옵니다.
10        return super.getSystemService(name)
11    }
12 }
13
14 override fun attachBaseContext(newBase: Context) {
15     // baseContext를 CustomContextWrapper로 감쌉니다.
16     super.attachBaseContext(CustomContextWrapper(newBase))
17 }
```

요약

Activity에서 this는 현재 Activity 인스턴스를 참조하며 생명주기 및 UI 특정 기능을 갖춘 고수준 Context를 제공합니다. 반면에 baseContext는 Activity가 기반으로 하는 원초적인 Context를 나타내며, 종종 커스텀 ContextWrapper 구현과 같은 고급 시나리오에서 사용됩니다. 통상적인 안드로이드 개발에서는 this가 가장 보편적으로 사용되지만, baseContext를 이해하면 디버깅이나 이식 가능한 & 재사용 가능한 컴포넌트 생성에 사용할 수 있습니다.

Q) 5. Application 클래스란 무엇인가요?

안드로이드의 Application 클래스는 전역 애플리케이션 상태와 생명주기를 유지하기 위한 역할을 합니다. 또한, Activity, Service 또는 BroadcastReceiver와 같은 다른 컴포넌트보다 가장 먼저 초기화되는 앱의 프로세스 진입점 역할을 수행합니다. Application 클래스는

앱의 전체 생명주기 사용 가능한 Context를 제공하므로 앱 전역에 걸쳐 공유되는 리소스 및 인스턴스를 초기화하는 데 이상적입니다.

Application 클래스의 목적

Application 클래스는 전역 상태를 유지하고 애플리케이션 전체 초기화를 수행하도록 설계되었습니다. 개발자는 종종 이 클래스를 상속받아 의존성을 설정하고, 라이브러리를 구성하기도 하고, Activity와 Service 전반에 걸쳐 지속되어야 하는 리소스를 초기화합니다.

기본적으로 모든 안드로이드 애플리케이션은 AndroidManifest.xml 파일에 커스텀 클래스를 지정하지 않는 한 Application 클래스의 기본 구현체를 사용합니다.

Application 클래스의 주요 메서드

1. **onCreate()**: onCreate() 메서드는 앱 프로세스가 생성될 때 호출됩니다. 일반적으로 데이터베이스 인스턴스, 네트워크 라이브러리 또는 Firebase 애널리틱스와 같은 분석 도구와 같은 애플리케이션 전체 의존성을 초기화하는 곳입니다. 애플리케이션 생명주기 동안 단 한 번만 호출됩니다.
2. **onTerminate()**: 이 메서드는 에뮬레이션된 환경에서 애플리케이션이 종료될 때 호출됩니다. 안드로이드가 호출을 보장하지 않으므로 실제 기기의 프로덕션 환경에서는 호출되지 않습니다.
3. **onLowMemory()** 및 **onTrimMemory()**: 이 메서드들은 시스템이 메모리 부족 상태를 감지할 때 트리거됩니다. onLowMemory()는 이전 API 레벨에서 사용되며, onTrimMemory()는 앱의 현재 메모리 상태에 따라 더 세분화된 제어를 제공합니다.

Application 클래스 사용 방법

커스텀 Application 클래스를 만들려면 Application 클래스를 상속받고 AndroidManifest.xml 파일의 <application> 태그에 수동으로 지정해야 합니다.

그림 18. CustomApplication.kt

```
1 class CustomApplication : Application() {  
2  
3     override fun onCreate() {  
4         super.onCreate()  
5         // 전역 의존성 초기화  
6         initializeDatabase()  
7         initializeAnalytics()  
8     }  
9  
10    private fun initializeDatabase() {  
11        // 데이터베이스 인스턴스 설정  
12    }  
13  
14    private fun initializeAnalytics() {  
15        // 분석 추적 구성  
16    }  
17 }
```

그림 19. AndroidManifest.xml

```
1 <application  
2     android:name=".CustomApplication"  
3     ... >  
4     ...  
5 </application>
```

Application 클래스의 사용 사례

1. **전역 리소스 관리**: 데이터베이스, SharedPreferences 또는 네트워크 클라이언트와 같은 리소스를 초기화하고, 애플리케이션 생명주기 전역에 걸쳐서 재사용할 수 있습니다.

2. **컴포넌트 초기화**: Firebase Analytics, Timber 등과 같은 도구는 앱 생명주기 전역에서 사용되는 경우가 다분하기 때문에, 원활한 기능을 보장하기 위해 애플리케이션 시작 중에 적절하게 초기화되어야 합니다.
3. **의존성 주입**: Dagger 또는 Hilt와 같은 프레임워크를 초기화하여 앱 전체에 의존성을 제공할 수 있습니다.

주의 사항

1. 초기에 앱 실행 지연을 방지하기 위해 `onCreate()`에서 무거운 태스크를 실행하지 않는 것이 좋습니다.
2. 관련 없는 로직을 Application 클래스에 마구 넣지 않아야 합니다. 전역 초기화 및 리소스 관리에만 집중하는 것이 좋습니다.
3. 앱 전반에 걸쳐서 사용되어야 하는 공유 리소스에 대해서는 스레드 안정성을 보장해야 합니다.

요약

Application 클래스는 애플리케이션 전역에서 사용해야 하는 리소스를 초기화하고 역할로 많이 사용됩니다. 원활한 전역 초기화를 위해 올바르게 사용하는 것이 중요하고, 안드로이드 SDK에서 제공하는 기본적인 API이지만 명확성을 유지하고 복잡성을 피하기 위해 불필요한 리소스 초기화는 제한하는 것이 좋습니다.

실전 질문

Q) Application 클래스의 목적은 무엇이고, 생명주기 및 리소스 관리 측면에서 Activity 와는 어떻게 다른가요?

Q) 6. AndroidManifest 파일의 목적은 무엇인가요?

AndroidManifest.xml 파일은 안드로이드 운영 체제에 애플리케이션에 대한 필수 정보를 정의하는 안드로이드 프로젝트에서 사실상 아주 중요한 구성 파일입니다. 이는 애플리케이션과 OS 간의 브릿지 역할을 하며, 애플리케이션의 인적사항이라고 할 수 있는 컴포넌트, 권한, 하드웨어 및 소프트웨어 기능 등을 정의하고 있습니다.

AndroidManifest.xml의 주요 기능

1. **애플리케이션 컴포넌트 선언:** Activities, Services, Broadcast Receivers, Content Providers 와 같은 필수 컴포넌트를 등록하여 안드로이드 시스템이 이를 시작하거나 상호 작용하는 방법을 알 수 있도록 합니다.
2. **권한(Permissions):** 앱에 필요한 INTERNET, ACCESS_FINE_LOCATION, 또는 READ_CONTACTS 와 같은 권한을 선언하여 사용자가 앱이 접근할 리소스를 알고 이러한 권한을 부여하거나 거부할 수 있도록 합니다.
3. **하드웨어 및 소프트웨어 요구 사항:** 카메라, GPS 또는 특정 화면 크기와 같이 앱이 의존하는 기능을 명시하여 Play Store가 이러한 요구 사항을 충족하지 않는 기기를 필터링하는 데 도움을 줍니다.
4. **앱 메타 정보(App Metadata):** 앱의 패키지 이름, 버전, 최소 및 대상 API 레벨, 테마, 스타일과 같은 필수 정보를 제공하며, 시스템은 이를 앱 설치 및 실행에 사용합니다.
5. **인텐트 필터(Intent Filters):** 컴포넌트(가령, Activity)에 대한 Intent Filters를 정의하여 링크를 열거나 콘텐츠 공유와 같이 응답할 수 있는 Intent 종류를 명시하고, 다른 앱이 개발자의 앱과 상호 작용할 수 있도록 합니다.
6. **앱 구성 및 세팅(App Configuration and Settings):** 메인 런처 Activity 정의, 백업 동작 구성, 테마 지정과 같은 구성을 포함하며, 이는 앱의 동작 방식과 표시 방식을 제어하는 데 도움이 됩니다.

아래는 AndroidManifest.xml 파일의 예시입니다.

그림 20. AndroidManifest.xml

```
1 <manifest xmlns:android="http://schemas.android.com/apk/res/android">
2
3     <!-- Permissions -->
4     <uses-permission android:name="android.permission.INTERNET" />
5     <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
6
7     <application
8         android:allowBackup="true"
9         android:icon="@mipmap/ic_launcher"
10        android:label="@string/app_name"
11        android:theme="@style/AppTheme">
```

```
12
13     <!-- Main Activity -->
14     <activity android:name=".MainActivity"
15             android:exported="true">
16         <intent-filter>
17             <action android:name="android.intent.action.MAIN" />
18             <category android:name="android.intent.category.LAUNCHER" />
19         </intent-filter>
20     </activity>
21
22     <!-- Additional Components -->
23     <service android:name=".MyService" />
24     <receiver android:name=".MyBroadcastReceiver"
25             android:exported="false" />
26
27     </application>
28 </manifest>
```

요약

AndroidManifest.xml 파일은 안드로이드 앱에서 인적사항과 같은 정보를 들고 있는 중요한 역할을 하며, 안드로이드 OS에 앱의 생명주기, 권한 및 상호 작용을 관리하는 데 필요한 세부 정보를 제공합니다. 이는 본질적으로 앱의 구조와 요구 사항을 정의하는 청사진 역할을 합니다. AndroidManifest.xml의 더 자세한 예시 코드는 [GitHub의 Pokédex project](#)¹⁴에서 살펴보실 수 있습니다.

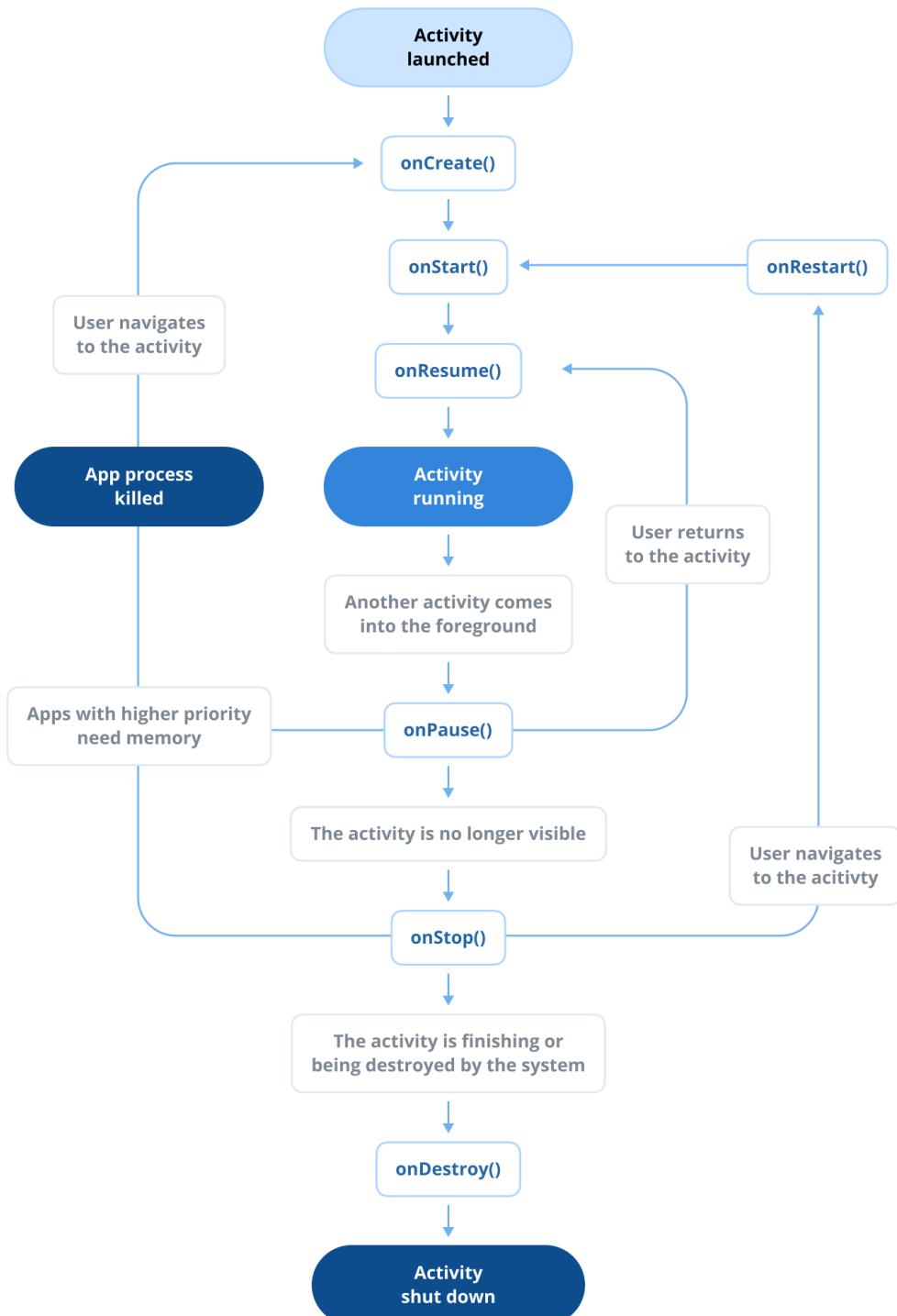
실전 질문

Q) AndroidManifest의 인텐트 필터는 앱 상호 작용을 어떻게 가능하게 하고, 액티비티 클래스가 AndroidManifest에 등록되어있지 않으면 어떻게 되나요?

¹⁴<https://github.com/skydoves/Pokedex/blob/main/app/src/main/AndroidManifest.xml>

Q) 7. Activity 생명주기를 설명해주세요

안드로이드 Activity 생명주기(lifecycle)는 Activity가 생성부터 소멸까지 거치는 다양한 상태를 나타냅니다. 생명주기를 이해하는 것은 리소스를 효과적으로 관리하고, 사용자 입력을 올바르게 처리하며, 원활한 사용자 경험을 보장하는 데 중요합니다. 아래는 Activity 생명주기의 주요 단계입니다.



1. **onCreate()**: Activity가 생성될 때 호출되는 첫 번째 메서드입니다. Activity를 초기화하고, UI 컴포넌트를 설정하며, 저장된 인스턴스 상태를 복원하는 곳입니다. Activity가 소멸되고 재생성되지 않는 한 Activity의 생명주기 동안 단 한 번만 호출됩니다.
2. **onStart()**: Activity가 사용자에게 보이지만 아직 상호 작용할 수는 없습니다. **onCreate()** 이후와 **onResume()** 이전에 호출됩니다.
3. **onRestart()**: Activity가 중지되었다가 다시 시작되는 경우(가령, 사용자가 다시 탐색하는 경우), 이 메서드가 **onStart()** 전에 호출됩니다.
4. **onResume()**: Activity가 포그라운드(foreground)에 있으며 사용자가 상호 작용할 수 있습니다. 일시 중지된 UI 업데이트, 애니메이션 또는 입력 리스너를 재개하는 곳입니다.
5. **onPause()**: 다른 Activity(가령, 뒤로가기)에 의해 Activity가 부분적으로 가려질 때 호출됩니다. Activity는 여전히 보이지만 포커스 중인 상태는 아닙니다. 애니메이션, 센서 업데이트 또는 데이터 저장과 같은 작업을 일시 중지하는 데 자주 사용됩니다.
6. **onStop()**: Activity가 더 이상 사용자에게 보이지 않을 때(가령, 다른 Activity가 포그라운드로 올 때) 호출됩니다. Activity가 중지된 동안 필요하지 않은 리소스(가령, 백그라운드 작업 또는 무거운 객체)를 해제해야 합니다.
7. **onDestroy()**: Activity가 완전히 소멸되고 메모리에서 제거되기 전에 호출됩니다. 남아 있는 모든 리소스를 해제하기 위한 최종 메서드입니다.

요약

Activity는 사용자와의 상호 작용과 안드로이드 시스템의 앱 리소스 관리 상태에게 따라 고유한 생명주기를 거칩니다. 개발자는 Activity에서 제공하는 생명주기 메서드를 통하여 리소스를 효율적으로 관리하고, 필요에 따라 리소스를 해지하기도 하며, 사용자에게 원활한 경험을 제공할 수 있습니다. 더 자세한 내용은 [안드로이드 공식 문서](#)¹⁵를 통해 학습하실 수 있습니다.

실전 질문

Q) `onPause()`와 `onStop()`의 차이점은 무엇인지 설명하고, 리소스 점유율이 높은 작업을 처리하는 경우 해당 메서드들을 어떤 시나리오에서 사용해야 하나요?

¹⁵<https://developer.android.com/reference/android/app/Activity>

💡 Pro Tips for Mastery: 액티비티 간의 생명주기 변화 심층적으로 살펴보기

Activity 생명주기에 대해서 다음과 같은 꼬리 질문을 생각해 볼 수 있습니다. “Activity A를 시작한 다음 Activity B를 시작하고, 다시 Activity A로 순차적으로 돌아올 때 생명주기에 어떤 변화가 일어나는지 각각 설명해 보세요.” 이런 시나리오는 안드로이드 시스템이 여러 Activity 상태를 관리하는 방식에 대하여 얼마나 잘 이해하고 있는지 판단하기 좋은 질문입니다.

두 Activity, 즉 Activity A와 Activity B 사이에서 화면 전환이 이루어질 때 각 Activity에 대한 안드로이드 생명주기 콜백은 특정 순서로 호출됩니다. 해당 시나리오에서 발생 가능한 생명주기의 변화를 단계별로 살펴보겠습니다.

Activity A와 Activity B의 생명주기 흐름 순서

- **Activity A의 초기 실행**

- Activity A: 처음에 실행될 때 onCreate() -> onStart() -> onResume() 순서로 호출 되고, 사용자가 Activity A와 상호 작용할 수 있습니다.

- **Activity A에서 Activity B로 이동**

- Activity A: onPause(), UI를 일시 중지하고 시각적으로 보이는 상태 관련 리소스를 해제합니다.
 - Activity B: onCreate() -> onStart() -> onResume(), 포커스를 가져오고 프로그래운드 Activity가 됩니다.
 - Activity A: onStop(), Activity B가 Activity A를 완전히 오버레이 하는 순간 호출됩니다.

- **Activity B에서 Activity A로 돌아오는 경우**

- Activity B: onPause()
 - Activity A: onRestart() -> onStart() -> onResume(), 포커스를 다시 얻고 프로그래운드로 돌아옵니다.
 - Activity B: onStop() -> onDestroy()

요약

두 Activity를 전환할 때, 포그라운드에 있는 Activity는 백그라운드로 이동하기 전에 반드시 onPause() 함수를 호출합니다. 새로 시작하는 Activity는 onCreate()부터 시작하는 생명주기를 거치고, 화면의 포커스를 가져옵니다. 이전 Activity로 돌아오면 onRestart() 또는 onResume()을 사용하여 일시 중지되었던 상태가 재개되며, 종료하는 Activity는 태스크에 따라 중지되거나 소멸됩니다. 액티비티 간의 전화에서 생명주기에 대한 적절한 이해를 통해 효과적인 리소스 관리와 원활한 사용자 경험을 제공할 수 있습니다.

Pro Tips for Mastery: Activity의 lifecycle 인스턴스란 무엇인가요?

모든 Activity는 **Lifecycle** 인스턴스를 가지고 있으며, 이는 Activity의 생명주기 이벤트를 관찰하고 이에 반응하는 방법을 제공합니다. lifecycle 인스턴스는 [Jetpack Lifecycle library](#)¹⁶의 일부이며, 개발자가 Activity의 생명주기 변화에 대응하여 코드를 깔끔하고 구조화된 방식으로 관리할 수 있도록 합니다.

lifecycle 속성은 ComponentActivity의 하위 클래스에서 노출하는 Lifecycle 클래스의 인스턴스입니다. 이는 Activity의 현재 생명주기 상태를 나타내며 onCreate, onStart, onResume 등과 같은 생명주기 이벤트를 해당 메서드를 직접 재정의하지 않고 관찰하는 방법을 제공합니다. 이를 통해 UI 업데이트, 리소스 해제 또는 LiveData를 구독하는데 유용합니다.

Lifecycle 인스턴스 사용 방법

lifecycle 인스턴스에 생명주기 이벤트를 관찰할 수 있는 LifecycleObserver 또는 DefaultLifecycleObserver 객체를 추가할 수 있습니다. 가령, onStart 및 onStop 상태를 관찰하려면, 해당 콜백을 처리하도록 개발자의 편의에 맞게 커스텀 관찰자를 등록할 수 있습니다.

¹⁶<https://developer.android.com/jetpack/androidx/releases/lifecycle>

그림 22. Observing Lifecycle Events

```
1 class MyObserver : DefaultLifecycleObserver {  
2  
3     override fun onStart(owner: LifecycleOwner) {  
4         super.onStart(owner)  
5         // onStart 시 수행할 작업  
6     }  
7  
8     override fun onStop(owner: LifecycleOwner) {  
9         super.onStop(owner)  
10        // onStop 시 수행할 작업  
11    }  
12 }  
13  
14 class MainActivity : ComponentActivity() {  
15     override fun onCreate(savedInstanceState: Bundle?) {  
16         super.onCreate(savedInstanceState)  
17         lifecycle.addObserver(MyObserver())  
18     }  
19 }
```

위의 예제에서 `MyObserver` 클래스는 `MainActivity`의 생명주기를 관찰합니다. `Activity`가 `STARTED` 또는 `STOPPED` 상태에 들어가면 해당 메서드가 호출됩니다.

Lifecycle 인스턴스 사용 시 이점

- 생명주기 인식:** `lifecycle` 인스턴스를 통해 생명주기 관찰자를 추가하고, `Activity`에서 필요하지 않은 작업을 최소하거나 해제함으로써 불필요한 작업 및 메모리 누수를 사전에 방지할 수 있습니다.
- 관심사 분리:** 생명주기 관찰과 관련된 로직을 `Activity` 클래스 외부로 이동하여 가독성과 유지 관리를 향상시킬 수 있습니다.
- Jetpack 라이브러리와 호환성:** `LiveData` 및 `ViewModel`과 같은 라이브러리는 `lifecycle` 인스턴스와 원활하게 작동하도록 설계되어 반응형 프로그래밍 및 효율적인 리소스 관리를 가능하게 합니다.

요약

Activity의 lifecycle 인스턴스는 안드로이드의 최신 아키텍처의 핵심 구성 요소로, 개발자가 생명주기 관찰을 구조화하고 재사용 가능한 방식으로 처리할 수 있도록 합니다. LifecycleObserver 및 기타 Jetpack 컴포넌트를 활용하여 더 견고하고 유지 관리하기 쉬운 애플리케이션을 만들 수 있습니다.

Q) 8. Fragment 생명주기를 설명해주세요

각 Fragment 인스턴스는 연결된 부모 Activity의 생명주기와 별도로 자체적인 생명주기를 갖습니다. 사용자가 앱과 상호 작용함에 따라 Fragment는 추가되거나, 제거되거나, 화면 안팎으로 이동될 때와 같이 다양한 생명주기 상태를 갖습니다. 생명주기 단계에는 생성되고, 시작되고, 화면에 나타나게 되고, 활성 상태가 되거나, 더 이상 필요하지 않을 때 중지되거나 소멸되는 상태로 전환되는 것 등이 포함됩니다. 이러한 생명주기를 올바르게 이해함으로써 Fragment가 리소스를 효과적으로 처리하고, UI 일관성을 유지하며, 사용자와의 상호작용에서 원활하게 응답할 수 있습니다.

안드로이드의 Fragment 생명주기는 Activity 생명주기와 많이 유사하지만 Fragment만의 고유한 콜백 메서드가 존재합니다.

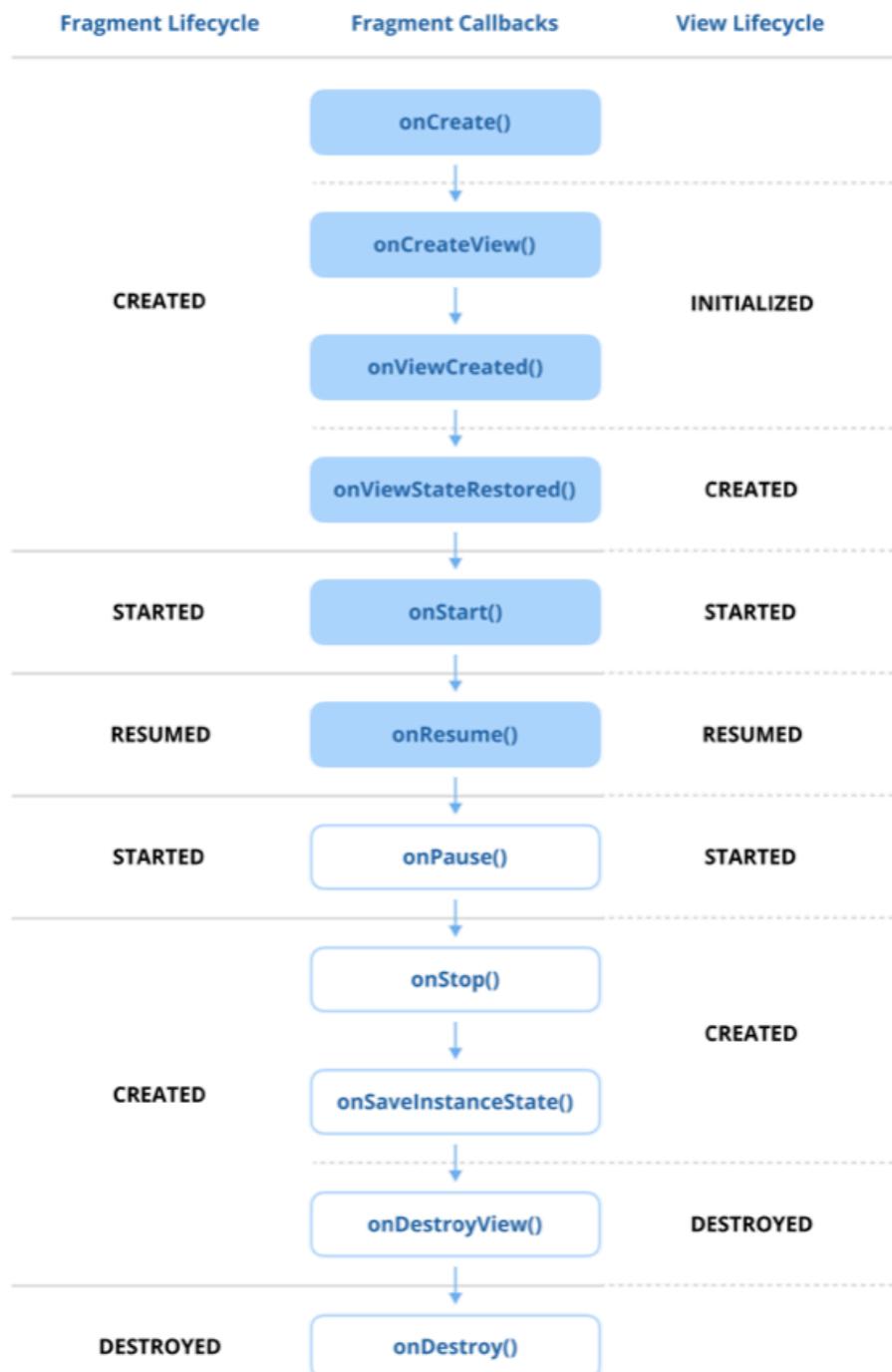


그림 23. fragment-lifecycle

1. **onAttach()**: Fragment가 부모 Activity와 연결될 때 호출되는 첫 번째 콜백입니다. 이제 Fragment가 연결되었으며 Activity 컨텍스트와 상호 작용할 수 있습니다.
2. **onCreate()**: Fragment를 초기화하기 위해 호출됩니다. 이 시점에서 Fragment는 생성되었지만 UI는 아직 생성되지 않았습니다. 일반적으로 필수 컴포넌트를 초기화하거나 저장된 상태를 복원하는 곳입니다.
3. **onCreateView()**: Fragment의 UI가 처음으로 그려질 때 호출됩니다. 이 메서드에서 Fragment 레이아웃의 루트 뷰를 반환합니다. LayoutInflater를 사용하여 Fragment의 레이아웃을 인플레이션하는 곳입니다.
4. **onViewStateRestored()**: Fragment의 뷰 계층이 생성되고 저장된 상태가 뷰에 복원된 후 호출됩니다.
5. **onViewCreated()**: 이 메서드는 Fragment의 뷰가 생성된 후 호출됩니다. 종종 UI 컴포넌트와 사용자 상호 작용 처리에 필요한 로직을 설정하는 데 사용됩니다.
6. **onStart()**: Fragment가 사용자에게 보이게 됩니다. 이는 Activity의 onStart() 콜백과 동일하며, Fragment가 이제 활성 상태이지만 아직 포그라운드에 있지는 않습니다.
7. **onResume()**: Fragment가 이제 완전히 활성 상태이며 포그라운드에서 실행 중이므로 사용자와 상호 작용이 가능합니다. 이 메서드는 Fragment의 UI가 완전히 화면에 표시되고 사용자가 상호 작용할 수 있을 때 호출됩니다.
8. **onPause()**: Fragment가 더 이상 포그라운드에 있지 않지만 여전히 화면에 보이는 경우 호출됩니다. Fragment가 포커스를 잃기 직전이며, Fragment가 포그라운드에 없을 때 지속해서는 안 되는 작업을 일시 중지해야 합니다.
9. **onStop()**: Fragment가 더 이상 보이지 않습니다. Fragment가 화면 밖에 있는 동안 지속해야 할 필요가 없는 작업을 중지하는 곳입니다.
10. **onSaveInstanceState()**: Fragment가 소멸되기 전에 UI 관련 상태 데이터를 저장하여 나중에 복원할 수 있도록 호출됩니다.
11. **onDestroyView()**: Fragment의 뷰 계층이 제거될 때 호출됩니다. 메모리 누수를 방지하기 위해 어댑터를 지우거나 참조를 null로 만드는 등 뷰와 관련된 리소스를 정리해야 합니다.
12. **onDestroy()**: Fragment 자체가 소멸될 때 호출됩니다. 이 시점에서 모든 리소스를 정리해야 하지만 Fragment는 여전히 부모 Activity에 연결되어 있습니다.
13. **onDetach()**: Fragment가 부모 Activity에서 분리되어 더 이상 연결되지 않습니다. 이것이 마지막 콜백이며 Fragment의 생명주기가 완료됩니다.

요약

Fragment 생명주기를 이해하는 것은 안드로이드 앱에서 리소스를 효과적으로 관리하고, 구성 변경(configuration change)을 원활하게 처리하며, 원활한 사용자 경험을 보장하는 데 중요합니다. [안드로이드 공식 문서¹⁷](#)를 통해 추가적으로 학습하실 수 있습니다.

실전 질문

Q) onCreateView()와 onDestroyView()의 목적은 무엇이며, 해당 메서드에서 뷰 관련 리소스를 올바르게 처리하는 것이 왜 중요한가요?

 **Pro Tips for Mastery: fragmentManager과 childFragmentManager의 차이점은 무엇인가요?**

안드로이드에서 fragmentManager과 childFragmentManager는 Fragment를 관리하는 데 필수적 이지만, 서로 다른 목적을 가지고 있으며 별개의 범위 내에서 작동합니다.

fragmentManager

fragmentManager는 FragmentActivity 또는 Fragment와 연결되어 있으며 Activity 수준에서 Fragment를 관리하는 역할을 합니다. 여기에는 부모 Activity에 직접 연결된 Fragment를 추가하거나 교체 또는 제거하는 동작이 포함됩니다.

Activity에서 supportFragmentManager를 호출하면 FragmentActivity에서 관리하는 fragmentManager에 접근할 수 있습니다. fragmentManager에 의해 관리되는 Fragment는 구조적으로 형제 관계이며 동일한 계층 수준에서 작동합니다.

```
1 // 액티비티 수준에서 프래그먼트 관리하기  
2 supportFragmentManager.beginTransaction()  
3     .replace(R.id.container, ExampleFragment())  
4     .commit()
```

이는 일반적으로 Activity에서 주요한 역할을 하는 내비게이션 시스템이나 UI 일부를 담당하는 Fragment를 컨트롤하는데 사용됩니다.

¹⁷<https://developer.android.com/guide/fragments/lifecycle>

childFragmentManager

childFragmentManager은 하나의 Fragment에 속하며 해당 Fragment의 자식 Fragment를 관리합니다. 이를 통해 Fragment가 다른 Fragment를 호스팅하여 중첩된 Fragment 구조를 만들 수 있습니다.

childFragmentManager을 사용하면 부모 Fragment의 생명주기 내에서 Fragment를 정의합니다. 이는 Fragment가 Activity의 Fragment 생명주기와 독립적이며, Fragment를 중첩해서 사용해야 하는 경우 Fragment 내에서 UI와 로직을 캡슐화하는 데 유용합니다.

그림 24. childFragmentManager Example.kt

```
1 // 부모 프래그먼트 내에서 자식 프래그먼트 관리하기
2 childFragmentManager.beginTransaction()
3     .replace(R.id.child_container, ChildFragment())
4     .commit()
```

childFragmentManager에 의해 관리되는 자식 Fragment는 부모 Fragment에 의하여 범위가 지정됩니다. 즉, 생명주기가 부모 Fragment에 연결됩니다. 가령, 부모 Fragment가 소멸되면 자식 Fragment도 소멸됩니다.

주요 차이점

1. 범위(Scope)

- fragmentManager은 Activity 수준에서 작동하며 Activity에 직접 연결된 Fragment를 관리합니다.
- childFragmentManager은 Fragment 내에서 작동하며 부모 Fragment 내에 중첩된 Fragment를 관리합니다.

2. 사용 사례(Use Case)

- Activity의 주요 UI 컴포넌트를 형성하는 Fragment에는 fragmentManager을 사용합니다.
- Fragment가 자체적으로 중첩 Fragment를 관리해야 할 때 childFragmentManager을 사용하여 더 모듈화되고 재사용 가능한 UI 컴포넌트를 만들 수 있습니다.

3. 생명주기(Lifecycle)

- fragmentManager에 의해 관리되는 Fragment는 Activity의 생명주기를 따릅니다.
- childFragmentManager에 의해 관리되는 Fragment는 부모 Fragment의 생명주기를 따릅니다.

요약

fragmentManager와 childFragmentManager 사이의 선택은 UI의 계층 구조에 따라 달라집니다. Activity 수준의 Fragment 관리에는 fragmentManager을 사용해야 합니다. 부모 Fragment 내에 자식 Fragment를 중첩하려면 childFragmentManager를 사용하셔야 합니다. Fragment를 많이 다루는 안드로이드 앱을 더 효율적으로 개발하기 위해서는 이들의 범위와 생명주기 차이를 잘 이해하는 것이 중요합니다.

 **Pro Tips for Mastery:** Fragment의 viewLifecycleOwner 인스턴스가 어떤 역할을 하는지 설명해 주세요.

안드로이드 개발에서 Fragment는 호스팅 Activity에 연결되어 자체적인 생명주기를 가지지만, Fragment의 **뷰 계층(View hierarchy)**은 이와 별도의 생명주기를 갖습니다. Fragment에서 LiveData와 같은 컴포넌트를 관리하거나 생명주기를 인식하는 데이터 소스를 관찰할 때 이 생명주기에 대한 구분이 생각보다 중요합니다. viewLifecycleOwner 인스턴스는 이러한 UI 계층과 관련된 생명주기를 효과적으로 관리하는 데 도움이 됩니다.

viewLifecycleOwner란?

viewLifecycleOwner는 Fragment의 **뷰 계층**과 관련된 **LifecycleOwner**입니다. 이는 Fragment의 onCreateView가 호출될 때 시작되고 onDestroyView가 호출될 때 끝나는 Fragment 뷰의 생명주기를 나타냅니다. 이를 통해 UI 관련 데이터나 리소스를 Fragment의 생명주기가 아닌 Fragment **뷰 계층** 생명주기에 바인딩하여 잠재적인 메모리 누수와 같은 문제를 방지할 수 있습니다.

Fragment의 **뷰 계층** 생명주기는 Fragment 자체의 생명주기보다 짧습니다. 데이터나 생명주기 이벤트를 관찰하기 위해 Fragment의 생명주기(this를 LifecycleOwner를 획득하는 데 사용)를 사용하면 뷰가 소멸된 후도 여전히 뷰에 접근할 위험성이 있습니다. 이는 크래시와 같이 개발자가 의도하지 않은 동작으로 이어질 수 있습니다.

`viewLifecycleOwner`를 사용하면 관찰자나 생명주기 인식 컴포넌트가 **뷰의 생명주기에 연결되어 뷰가 소멸될 때 업데이트를 안전하게 중지할 수 있습니다.**

아래는 잠재적인 메모리 누수를 피하면서 Fragment에서 `LiveData`를 관찰하는 예시입니다.

그림 25. Using `viewLifecycleOwner` with `LiveData`

```
1 class MyFragment : Fragment(R.layout.fragment_example) {  
2  
3     private val viewModel: MyViewModel by viewModels()  
4  
5     override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
6         super.onViewCreated(view, savedInstanceState)  
7  
8         // viewLifecycleOwner를 사용하여 LiveData 관찰  
9         viewModel.data.observe(viewLifecycleOwner) { data ->  
10             // 새 데이터로 UI 업데이트  
11             textView.text = data  
12         }  
13     }  
14 }
```

위의 예제에서 `viewLifecycleOwner`는 Fragment가 아직 살아 있더라도 Fragment의 뷰가 소멸될 때 관찰자가 자동으로 해지되도록 보장합니다.

`lifecycleOwner`와 `viewLifecycleOwner`의 차이점

- **`lifecycleOwner` (Fragment의 생명주기)**: Fragment의 전체 생명주기를 나타내며 상대적으로 생명주기가 더 길고, 호스팅 Activity에 연결됩니다.
- **`viewLifecycleOwner` (Fragment 뷰의 생명주기)**: Fragment 뷰의 생명주기를 나타내며, `onCreateView`에서 시작하여 `onDestroyView`에서 종료됩니다.

요약

`viewLifecycleOwner`는 `LiveData` 관찰이나 뷰에 연결된 리소스 관리와 같이 뷰의 생명주기와 밀접한 관련이 있는 **UI 관련 작업**에 특히 유용합니다.

Q) 9. Service란 무엇인가요?

Service는 앱이 사용자 상호 작용과 독립적으로 장기적으로 작업을 수행할 수 있도록 하는 백그라운드 컴포넌트입니다. Activity와 달리 Service는 사용자 인터페이스가 없으며 앱이 포그라운드에 있지 않을 때도 계속 실행될 수 있습니다. 일반적으로 파일 다운로드, 음악 재생, 데이터 동기화 또는 네트워크 작업 처리와 같은 백그라운드 작업에 사용됩니다.

1. Started Service

Started Service는 앱에서 startService()를 호출할 때 시작됩니다. stopSelf()를 사용하여 스스로 작업을 중지하거나 stopService()를 사용하여 명시적으로 중지될 때까지 백그라운드에서 지속적으로 실행됩니다.

사용 예시

- 백그라운드 음악 재생
- 파일 업로드 또는 다운로드

그림 26. Started Service

```

1 class MyService : Service() {
2     override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
3         // 백그라운드에서 장기 실행 작업 수행
4         return START_STICKY // 서비스가 종료되면 시스템이 재시작 시도
5     }
6
7     override fun onBind(intent: Intent?): IBinder? = null // 바인딩 사용 안 함
8 }
```

2. Bound Service

Bound Service는 안드로이드의 컴포넌트가 bindService()를 사용하여 서비스에 바인딩할 수 있도록 합니다. Service는 바인딩된 클라이언트가 있는 동안 활성 상태를 유지하며 모든 클라이언트의 연결이 끊어지면 자동으로 중지됩니다.

사용 예시

- 원격 서버에서 데이터 가져오기
- 백그라운드 블루투스 연결 관리

그림 27. Bound Service

```

1 class BoundService : Service() {
2     private val binder = LocalBinder()
3
4     inner class LocalBinder : Binder() {
5         fun getService(): BoundService = this@BoundService
6     }
7
8     override fun onBind(intent: Intent?): IBinder = binder // 클라이언트에게 Binder 반환
9 }
```

3. Foreground Service (포그라운드 Service)

Foreground Service는 지속적인 알림을 표시하면서 활성 상태를 유지하는 특별한 유형의 Service입니다. 음악 재생, 내비게이션 또는 위치 추적과 같이 사용자가 계속 인지해야 하는 작업에 사용됩니다.

그림 28. Foreground Service

```

1 class ForegroundService : Service() {
2     override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
3         val notification = createNotification()
4         startForeground(1, notification) // 서비스를 포그라운드 상태로 시작
5         return START_STICKY
6     }
7
8     private fun createNotification(): Notification {
9         // Android 8.0 (API 26) 이상에서는 알림 채널 생성 필수
10        val channelId = "ForegroundServiceChannel"
11        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
```

```

12     val channel = NotificationChannel(
13         channelId,
14         "Foreground Service Channel",
15         NotificationManager.IMPORTANCE_DEFAULT
16     )
17     getSystemService(NotificationManager::class.java)?.
18     ↪ createNotificationChannel(channel)
19 }
20
21     return NotificationCompat.Builder(this, channelId)
22         .setContentTitle("Foreground Service")
23         .setContentText("Running...")
24         .setSmallIcon(R.drawable.ic_notification)
25         .build()
26
27     override fun onBind(intent: Intent?): IBinder? = null
28 }
```

Service 유형 간 차이점

Service 유형	백그라운드 실행	자동 종지	UI 알림 필요
Started Service	✓ 예	✗ 아니요	✗ 아니요
Bound Service	✓ 예	✓ 예 (모든 클라이언트가 언바인드 시)	✗ 아니요
Foreground Service(포그라운 드 서비스)	✓ 예	✗ 아니요	✓ 예

Service 사용 모범 사례

- 즉시 실행이 필요하지 않은 백그라운드 작업에는 Service 대신 **Jetpack WorkManager¹⁸**를 사용합니다.
- 불필요한 리소스 소비를 방지하기 위해 작업이 완료되면 Service를 종지합니다.
- 투명성을 위해 명확한 알림을 제공하여 포그라운드 Service를 책임감 있게 사용합니다.
- 메모리 누수를 방지하기 위해 생명주기 변화(Service 생명주기)을 적절하게 처리합니다.

요약

Service는 사용자 상호 작용 없이 백그라운드 처리를 가능하게 합니다. **Started Service**는 수동으로 종지될 때까지 실행되고, **Bound Service**는 다른 컴포넌트와 상호 작용하며, **Foreground Service**는 지속적인 알림과 함께 활성 상태를 유지합니다. Service를 적절하게 관리하면 효율적인 시스템 리소스 사용과 원활한 사용자 경험을 보장할 수 있습니다.

실전 질문

Q) 안드로이드에서 Started 서비스와 Bound 서비스의 차이점은 무엇이며, 각각 언제 사용해야 하나요?

 **Pro Tips for Mastery:** 포그라운드 서비스(**foreground service**)를 어떻게 처리하나요?

Foreground Service는 사용자가 인지할 수 있는 작업을 수행하는 특별한 유형의 Service입니다. 알림 표시줄에 지속적인 알림을 표시하여 사용자가 해당 작업이 진행 중이라는 사실을 인지하도록 보장합니다. Foreground Service는 미디어 재생, 위치 추적 또는 파일 업로드와 같은 우선순위가 높은 작업에 사용됩니다.

일반 Service와의 차이점은 Foreground Service는 startForeground()를 통해 호출하고, 시작 즉시 알림을 표시해야 한다는 것입니다.

¹⁸<https://developer.android.com/jetpack/androidx/releases/work>

그림 29. ForegroundService.kt

```
1 class ForegroundService : Service() {
2
3     override fun onCreate() {
4         super.onCreate()
5         // 리소스 초기화
6     }
7
8     override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
9         val notification = createNotification()
10        startForeground(1, notification) // 서비스를 포그라운드로 시작
11        // 작업 수행
12        return START_STICKY
13    }
14
15    private fun createNotification(): Notification {
16        val notificationChannelId = "ForegroundServiceChannel"
17        // Android 8.0 (API 26) 이상에서는 알림 채널 생성 필수
18        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
19            val channel = NotificationChannel(
20                notificationChannelId,
21                "Foreground Service",
22                NotificationManager.IMPORTANCE_DEFAULT
23            )
24            getSystemService(NotificationManager::class.java).]
25            → createNotificationChannel(channel)
26        }
27        return NotificationCompat.Builder(this, notificationChannelId)
28            .setContentTitle("Foreground Service")
29            .setContentText("Running in the foreground")
30            .setSmallIcon(R.drawable.ic_notification)
31            .build()
32    }
}
```

```
33     override fun onDestroy() {  
34         super.onDestroy()  
35         // 리소스 정리  
36     }  
37  
38     override fun onBind(intent: Intent?): IBinder? = null  
39 }
```

Service와 Foreground Service의 차이점

- 사용자 인지:** 표준 Service는 눈에 띄지 않게 백그라운드에서 실행될 수 있지만, Foreground Service는 알림이 필요하므로 사용자가 해당 작업이 진행 중이라는 사실을 알 수 있습니다.
- 우선순위:** Foreground Service는 우선순위가 더 높으며, 일반 Service에 비해 메모리 부족 상태에서 시스템에 의해 종료될 가능성이 적습니다.
- 사용 사례:** Service는 가벼운 백그라운드 작업에 이상적이며, Foreground Service는 지속적이고 사용자가 인지할 수 있는 작업에 적합합니다.

Service 모범 사례

- 시스템 리소스를 절약하기 위해 작업이 완료되면 항상 Service를 중지합니다.
- 즉시 실행이 필요하지 않은 백그라운드 작업에는 WorkManager를 사용합니다.
- Foreground Service의 경우, 투명성을 유지하기 위해 알림이 사용자 친화적이고 정보를 제공하는지 확인합니다.

요약

안드로이드 Service는 효율적인 백그라운드 작업을 가능하게 하는 반면, Foreground Service는 사용자 가시성과 함께 지속적인 상호작용이 필요한 경우에 사용됩니다. 둘 다 시스템 리소스를 효율적으로 관리하면서 원활한 사용자 경험을 만드는 데 중요한 역할을 합니다.

 **Pro Tips for Mastery:** 서비스(service)의 생명주기는 어떻게 되나요?

이전에 살펴본 것처럼 Service는 두 가지 모드로 작동할 수 있습니다.

1. **Started Service**: startService()를 사용하여 시작되며 stopSelf() 또는 stopService()를 사용하여 명시적으로 중지될 때까지 계속 실행됩니다.
2. **Bound Service**: bindService()를 사용하여 하나 이상의 컴포넌트에 연결되며 바인딩되어 있는 동안 존재합니다.

생명주기는 onCreate(), onStartCommand(), onBind(), onDestroy()와 같은 메서드를 통해 관리됩니다.

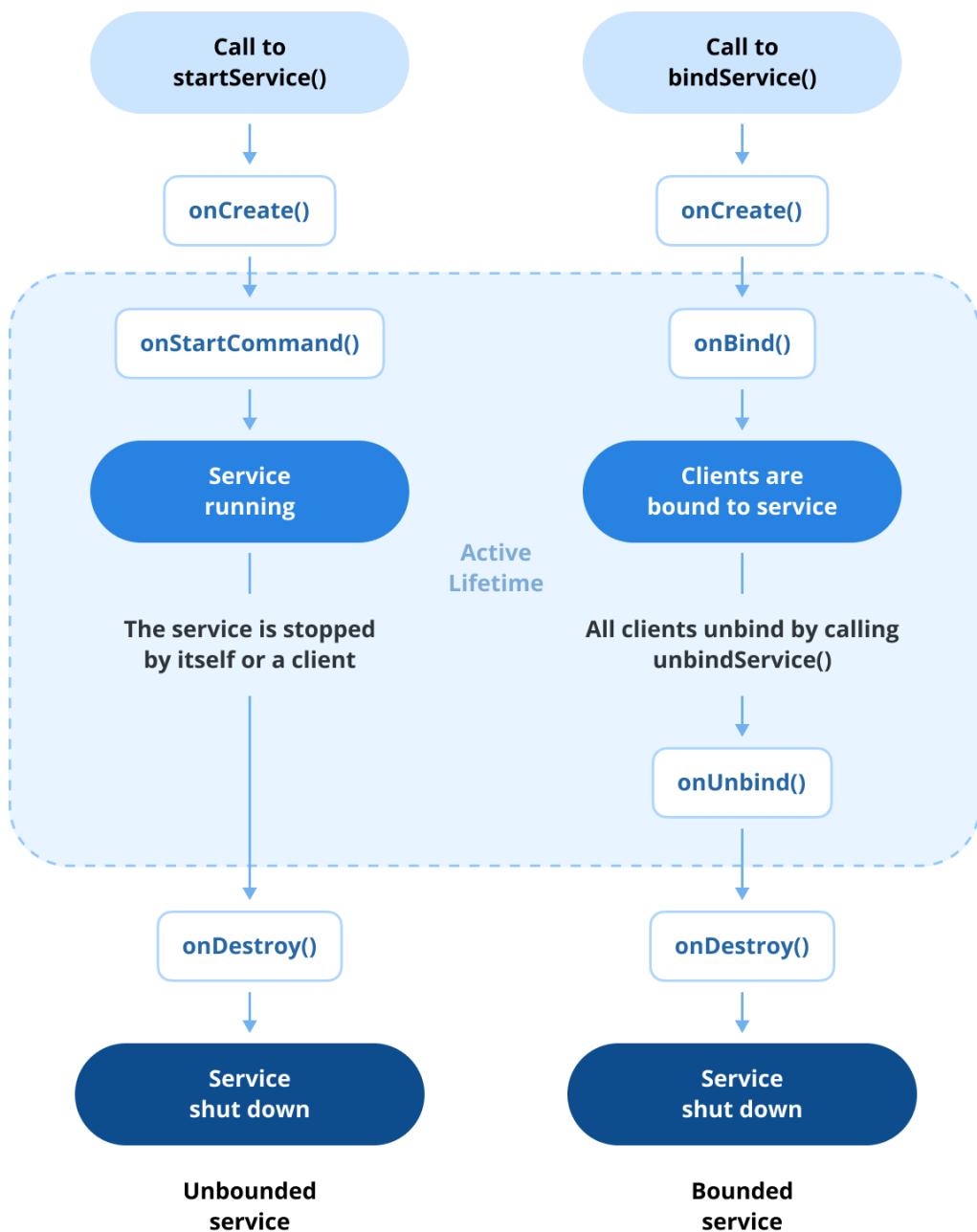


그림 30. service-lifecycle

Started Service의 생명주기

1. **onCreate()**: Service가 처음 생성될 때 호출됩니다. Service에 필요한 리소스를 초기화하는 데 사용됩니다.
2. **onStartCommand()**: startService()로 Service가 시작될 때 트리거됩니다. 이 메서드는 실제 작업 실행을 처리하고 Service가 종료될 경우 반환 값(가령, START_STICKY, START_NOT_STICKY)을 사용하여 재시작 동작을 결정합니다.
3. **onDestroy()**: stopSelf() 또는 stopService()를 사용하여 Service가 중지될 때 호출됩니다. 리소스 해제나 스레드 중지와 같은 정리 작업에 사용됩니다.

그림 31. SimpleStartedService.kt

```
1 class SimpleStartedService : Service() {  
2     override fun onCreate() {  
3         super.onCreate()  
4         // 리소스 초기화  
5     }  
6  
7     override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {  
8         // 장기 실행 작업 수행  
9         return START_STICKY // 서비스가 종료되면 재시작  
10    }  
11  
12    override fun onDestroy() {  
13        super.onDestroy()  
14        // 리소스 정리  
15    }  
16  
17    override fun onBind(intent: Intent?): IBinder? = null // 시작된 서비스에는 사용되지  
18    → 않음  
19 }
```

Bound Service의 생명주기 메서드

1. **onCreate()**: Started Service와 유사하게, Service가 생성될 때 리소스를 초기화합니다.
2. **onBind()**: 컴포넌트가 bindService()를 사용하여 Service에 바인딩될 때 호출됩니다. 이 메서드는 Service에 대한 인터페이스(IBinder)를 제공합니다.
3. **onUnbind()**: 마지막 클라이언트가 Service에서 언바인드될 때 호출됩니다. 바인딩된 클라이언트에 특정한 리소스를 정리하는 곳입니다.
4. **onDestroy()**: Service가 종료될 때 호출됩니다. 리소스 정리 및 진행 중인 작업을 중지합니다.

그림 32. SimpleBoundService.kt

```
1 class SimpleBoundService : Service() {  
2     private val binder = LocalBinder()  
3  
4     override fun onCreate() {  
5         super.onCreate()  
6         // 리소스 초기화  
7     }  
8  
9     override fun onBind(intent: Intent?): IBinder {  
10        return binder // 바인딩된 서비스의 인터페이스 반환  
11    }  
12  
13    override fun onUnbind(intent: Intent?): Boolean {  
14        // 바인딩된 클라이언트가 없을 때 정리  
15        return super.onUnbind(intent) // 기본값은 false, true 반환 시 재바인딩 시  
16        → onRebind 호출  
17    }  
18  
19    override fun onDestroy() {  
20        super.onDestroy()  
21        // 리소스 정리  
22    }
```

```
22  
23     inner class LocalBinder : Binder() {  
24         fun getService(): SimpleBoundService = this@SimpleBoundService  
25     }  
26 }
```

Started Service와 Bound Service 생명주기의 차이점

1. **Started Service**: 어떤 컴포넌트와도 독립적이며 명시적으로 종지될 때까지 실행됩니다.
2. **Bound Service**: 최소 하나 이상의 클라이언트에 바인딩되어 있는 동안에만 존재합니다.

요약

Service 생명주기를 이해하는 것은 효율적이고 신뢰할 수 있는 백그라운드 작업을 구현하는 데 중요합니다. **Started Service**는 독립적인 작업을 수행하고 종지될 때까지 지속되며, **Bound Service**는 클라이언트 상호 작용을 위한 인터페이스를 제공하고 언바인드되면 종료됩니다. 이러한 생명주기를 적절하게 관리하면 최적의 리소스 활용을 보장하고 메모리 누수를 방지할 수 있습니다.

Q) 10. BroadcastReceiver란 무엇인가요?

BroadcastReceiver는 앱이 안드로이드 운영 체제 전체의 브로드캐스트 메시지나 앱 특정 브로드캐스트를 수신하고 응답할 수 있도록 하는 컴포넌트입니다. 이러한 브로드캐스트¹⁹는 일반적으로 시스템이나 다른 애플리케이션에 의해 트리거되어 배터리 상태 변경, 네트워크 연결 업데이트 또는 앱 내에서 전송된 커스텀 Intent와 같은 다양한 이벤트를 알립니다. BroadcastReceiver는 동적인 시스템 또는 앱 수준 이벤트에 반응하는 응답성 있는 애플리케이션을 구축하는 데 유용한 메커니즘입니다.

¹⁹<https://developer.android.com/develop/background-work/background-tasks/broadcasts>

BroadcastReceiver의 목적

BroadcastReceiver는 Activity나 Service의 생명주기에 직접적으로 연결되지 않을 수 있는 이벤트를 처리하는 데 사용됩니다. 이는 앱이 백그라운드에서 계속 실행되지 않고도 변경 사항에 반응할 수 있도록 하는 메시징 시스템 역할을 하여 리소스를 절약합니다.

브로드캐스트 유형

1. **시스템 브로드캐스트(System Broadcasts)**: 안드로이드 운영 체제에서 배터리 잔량 변경, 시간대 업데이트 또는 네트워크 연결 변경과 같은 시스템 이벤트를 앱에 알리기 위해 보냅니다.
2. **커스텀(Custom Broadcasts)**: 애플리케이션에서 앱 내부 또는 앱 간에 특정 정보나 이벤트를 전달하기 위해 보냅니다.

커스텀 BroadcastReceiver 선언하기

BroadcastReceiver를 생성하려면 BroadcastReceiver 클래스를 상속받고 브로드캐스트 처리 로직을 정의하는 onReceive 메서드를 재정의해야 합니다.

그림 33. MyBroadcastReceiver.kt

```
1 class MyBroadcastReceiver : BroadcastReceiver() {
2     override fun onReceive(context: Context, intent: Intent) {
3         val action = intent.action
4         if (action == Intent.ACTION_BATTERY_LOW) {
5             // 배터리 부족 이벤트 처리
6             Log.d("MyBroadcastReceiver", "Battery is low!")
7         }
8     }
9 }
```

커스텀 BroadcastReceiver 등록하기

BroadcastReceiver는 두 가지 방법으로 등록할 수 있습니다.

1. **매니페스트 파일을 통한 정적 등록**: 앱이 실행 중이지 않을 때도 처리해야 하는 이벤트에 사용합니다. `AndroidManifest.xml` 파일에 `<intent-filter>`를 추가합니다.

그림 34. `AndroidManifest.xml`

```
1 <!-- Android 8.0 (API 26) 이상에서는 대부분의 암시적 브로드캐스트 수신 제한 -->
2 <receiver android:name=".MyBroadcastReceiver"
3         android:exported="false">
4     <intent-filter>
5         <action android:name="android.intent.action.BATTERY_LOW" />
6     </intent-filter>
7 </receiver>
```

2. **코드를 통한 동적 등록**: 앱이 활성 상태이거나 특정 상태일 때만 처리해야 하는 이벤트에 사용합니다.

그림 35. Dynamic Registration in Activity

```
1 val receiver = MyBroadcastReceiver()
2 val intentFilter = IntentFilter(Intent.ACTION_BATTERY_LOW)
3 // Android Tiramisu (API 33) 이상에서는 RECEIVER_EXPORTED 또는 RECEIVER_NOT_EXPORTED
4 // → 플래그 필요
5 val flags = if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
6     Context.RECEIVER_NOT_EXPORTED
7 } else {
8     0 // 플래그 없음
9 }
10 registerReceiver(receiver, intentFilter, flags)
11 // Activity나 컴포넌트가 소멸될 때 등록 해제 필수
12 override fun onDestroy() {
13     unregisterReceiver(receiver)
14     super.onDestroy()
15 }
```

유의 사항

- **생명주기 관리**²⁰: 동적 등록을 사용하는 경우 메모리 누수를 방지하기 위해 `unregisterReceiver`를 사용하여 리시버를 등록 해제해야 합니다.
- **백그라운드 실행 제한**²¹: 안드로이드 8.0 (API 레벨 26)부터 백그라운드 앱은 `Implicit broadcast exceptions`²²를 제외하고 브로드캐스트 수신에 제한을 받습니다. 이러한 경우를 처리하려면 `Context.registerReceiver` 또는 `JobScheduler`를 사용해야 합니다.
- **보안**²³: 민감한 정보가 포함된 브로드캐스트는 무단 접근을 방지하기 위해 권한으로 보호해야 합니다.

BroadcastReceiver 사용 사례

- 네트워크 연결 변경 모니터링.
- SMS 또는 통화 이벤트에 응답.
- 충전 상태와 같은 시스템 이벤트에 대한 UI 업데이트.
- 커스텀 브로드캐스트로 작업 또는 알람 예약.

요약

`BroadcastReceiver`는 OS 시스템과 상호 작용하는 반응형 애플리케이션을 구축하는 데 필수적인 컴포넌트입니다. 앱이 시스템 또는 앱 이벤트를 효율적으로 수신하고 응답할 수 있도록 합니다. 특히 올바른 생명주기 관리 및 최신 안드로이드 운영 체제의 제한 사항을 준수하는 등 적절하게 사용하면 앱을 견고하고 효율적으로 리소스를 운용할 수 있습니다.

실전 질문

Q) 브로드캐스트의 유형에는 어떤 것이 있으며, 기능 및 사용 측면에서 시스템 브로드캐스트와 커스텀 브로드캐스트는 어떤 차이가 있나요?

²⁰<https://developer.android.com/develop/background-work/background-tasks/broadcasts#unregister-broadcast>

²¹<https://developer.android.com/about/versions/oreo/background>

²²<https://developer.android.com/develop/background-work/background-tasks/broadcasts/broadcast-exceptions>

²³<https://developer.android.com/develop/background-work/background-tasks/broadcasts#security-considerations>

Q) 11. ContentProvider의 목적은 무엇이며, 애플리케이션 간의 안전한 데이터 공유를 어떻게 용이하게 하나요?

ContentProvider²⁴는 구조화된 데이터에 대한 접근을 관리하고 애플리케이션 간 데이터 공유를 위한 표준화된 인터페이스를 제공하는 컴포넌트입니다. 다른 앱이나 컴포넌트가 데이터를 쿼리, 삽입, 업데이트 또는 삭제하는 데 사용할 수 있는 중앙 저장소 역할을 하여 앱 간의 안전하고 일관된 데이터 공유를 보장합니다.

ContentProvider는 여러 앱이 동일한 데이터에 접근해야 하거나, 데이터베이스 또는 내부 저장소 구조를 노출하지 않고 다른 앱에 데이터를 제공하려는 경우 특히 유용합니다.

ContentProvider의 목적

ContentProvider의 주요 목적은 데이터 접근 로직을 캡슐화하여 앱 간 데이터 공유를 더 쉽고 안전하게 만드는 것입니다. 이는 SQLite 데이터베이스, 파일 시스템 또는 네트워크 기반 데이터일 수 있는 기본 데이터 소스를 추상화하고 데이터와 상호 작용하기 위한 통합 인터페이스를 제공합니다.

ContentProvider의 주요 구성 요소

ContentProvider는 데이터 접근 주소로 **URI (Uniform Resource Identifier)**를 사용합니다. URI는 다음으로 구성됩니다.

1. **권한(Authority)**: ContentProvider를 식별합니다 (예를 들어, com.example.myapp.provider).
2. **Path**: 데이터 유형을 지정합니다 (가령, /users 또는 /products).
3. **ID (optional)**: 데이터 셋 내의 특정 항목을 참조합니다.

ContentProvider 구현하기

ContentProvider를 생성하려면 ContentProvider를 상속받고 다음 메서드를 구현해야 합니다.

- `onCreate()`: ContentProvider를 초기화합니다.

²⁴<https://developer.android.com/guide/topics/providers/content-provider-basics>

- `query()`: 데이터를 검색합니다.
- `insert()`: 새 데이터를 추가합니다.
- `update()`: 기존 데이터를 수정합니다.
- `delete()`: 데이터를 제거합니다.
- `getType()`: 데이터의 MIME 유형을 반환합니다.

그림 36. MyContentProvider.kt

```
1 class MyContentProvider : ContentProvider() {  
2  
3     private lateinit var database: SQLiteDatabase  
4  
5     override fun onCreate(): Boolean {  
6         database = MyDatabaseHelper(context!!).writableDatabase  
7         return true  
8     }  
9  
10    override fun query(  
11        uri: Uri,  
12        projection: Array<String>?,  
13        selection: String?,  
14        selectionArgs: Array<String>?,  
15        sortOrder: String?  
16    ): Cursor? {  
17        // URI 파싱 및 테이블/조건 결정 필요  
18        return database.query("users", projection, selection, selectionArgs, null,  
19            null, sortOrder)  
20    }  
21  
22    override fun insert(uri: Uri, values: ContentValues?): Uri? {  
23        val id = database.insert("users", null, values)  
24        // context?.contentResolver?.notifyChange(uri, null) // 데이터 변경 알림 (필요시)  
25        return ContentUris.withAppendedId(uri, id)  
26    }  
27}
```

```
26  
27     override fun update(uri: Uri, values: ContentValues?, selection: String?,  
28         selectionArgs: Array<String>?): Int {  
29         val count = database.update("users", values, selection, selectionArgs)  
30         // if (count > 0) context?.contentResolver?.notifyChange(uri, null) // 데이터  
31         // 변경 알림 (필요시)  
32         return count  
33     }  
34  
35     override fun delete(uri: Uri, selection: String?, selectionArgs: Array<String>?):  
36         Int {  
37         val count = database.delete("users", selection, selectionArgs)  
38         // if (count > 0) context?.contentResolver?.notifyChange(uri, null) // 데이터  
39         // 변경 알림 (필요시)  
40         return count  
41     }  
42  
43 }
```

ContentProvider 등록하기

다른 앱에서 ContentProvider에 접근할 수 있도록 하려면 `AndroidManifest.xml` 파일에 선언해야 합니다. `authority` 속성은 ContentProvider를 고유하게 식별합니다.

그림 37. AndroidManifest.xml

```
1 <provider
2     android:name=".MyContentProvider"
3     android:authorities="com.example.myapp.provider"
4     android:exported="true"
5     android:grantUriPermissions="true" />
```

ContentProvider에서 데이터 접근하기

다른 앱에서 ContentProvider와 상호 작용하려면 ContentResolver 클래스를 사용해야 합니다. ContentResolver는 데이터를 쿼리, 삽입, 업데이트 또는 삭제하는 메서드를 제공합니다.

그림 38. Accessing Data from a ContentProvider

```
1 val contentResolver = context.contentResolver
2
3 // 데이터 쿼리
4 val cursor = contentResolver.query(
5     Uri.parse("content://com.example.myapp.provider/users"),
6     null,
7     null,
8     null,
9     null
10 )
11
12 // 데이터 삽입
13 val values = ContentValues().apply {
14     put("name", "John Doe")
15     put("email", "johndoe@example.com")
16 }
17 contentResolver.insert(Uri.parse("content://com.example.myapp.provider/users"),
18     values)
```

ContentProvider 사용 사례

- 다른 애플리케이션 간 데이터 공유.
- 앱 시작 시 컴포넌트 또는 리소스 초기화.
- 연락처, 미디어 파일 또는 앱별 데이터와 같은 구조화된 데이터에 대한 접근 제공.
- 연락처 앱이나 파일 선택기와 같은 안드로이드 시스템 기능과의 통합 활성화.
- 세분화된 보안 제어를 통한 데이터 접근 허용.

요약

ContentProvider는 앱 간에 구조화된 데이터를 안전하고 효율적으로 공유하기 위한 중요한 컴포넌트입니다. 내장되어 있는 데이터 저장 메커니즘을 추상화 함으로써, 외부에서 안전하게 데이터에 접근하기 위한 표준화된 인터페이스를 제공합니다. 이를 적절하게 활용한다면 데이터 무결성, 보안 및 안드로이드 시스템 기능과의 호환성 등을 보장합니다.

실전 질문

Q) ContentProvider URI의 주요 구성 요소는 무엇이며, ContentResolver는 데이터를 쿼리하거나 수정하기 위해 ContentProvider와 어떻게 상호 작용하나요?

 **Pro Tips for Mastery:** 앱 시작 시 리소스나 초기 셋업을 위해 ContentProvider를 사용하는 사용 사례

ContentProvider의 또 다른 사용 사례는 앱 시작 시 리소스나 초기 셋업을 구성하는 것입니다. 일반적으로 리소스나 라이브러리 초기화는 Application 클래스에서 하지만, 관심사를 더 잘 분리하기 위해 초기화 관련 로직을 별도의 ContentProvider에 캡슐화할 수 있습니다. 커스텀 ContentProvider를 생성하고 AndroidManifest.xml에 등록하면 초기화 작업을 효율적으로 위임할 수 있습니다.

ContentProvider의 onCreate() 메서드는 Application.onCreate() 메서드보다 먼저 호출되므로 초기화를 위한 훌륭한 진입점 역할을 합니다. 가령, [Firebase Android SDK](#)²⁵는 커스텀 ContentProvider를 사용하여 Firebase SDK의 의존성을 추가하는 것만으로도

²⁵<https://github.com/firebase/firebase-android-sdk/blob/6a03d4ca8ab6ae86968cded8e04e8802d5393882.firebaseio-common/src/main/java/com/google/firebase/provider/FirebaseInitProvider.java#L34>

자동으로 초기화합니다. 이러한 방식은 Application 클래스에서 개발자가 수동으로 FirebaseApp.initializeApp(this)를 호출할 수고를 덜어줍니다. 단 한 줄이라지만 수백 만의 프로젝트에서 한 줄씩 줄여준다면 그 역할을 충분히 잘하고 있다고 볼 수 있습니다.

다음은 Firebase에서 FirebaseInitProvider라는 커스텀 ContentProvider를 구현하는 실제 코드입니다.

```
1 public class FirebaseInitProvider extends ContentProvider {
2     /** Called before {@link Application#onCreate()}. */
3     @Override
4     public boolean onCreate() {
5         try {
6             currentlyInitializing.set(true);
7             if (FirebaseApp.initializeApp(getContext()) == null) {
8                 Log.i(TAG, "FirebaseApp initialization unsuccessful");
9             } else {
10                 Log.i(TAG, "FirebaseApp initialization successful");
11             }
12             return false; // 초기화 성공 여부와 관계없이 false 반환 (Provider를 유지할 필요가
13             // 없기 때문에)
14         } finally {
15             currentlyInitializing.set(false);
16         }
17         // 다른 ContentProvider 메서드들은 null 또는 예외를 반환/던집니다.
18     }
```

FirebaseInitProvider는 아래와 같이 같이 Firebase SDK 내부에 존재하는 AndroidManifest 파일에 등록되어, 개발자들이 SDK 의존성을 추가하는 것만으로도 해당 ContentProvider가 개발자들의 프로젝트에 자동으로 통합됩니다.

```
1 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
2   xmlns:tools="http://schemas.android.com/tools">
3     <!--Although the *SdkVersion is captured in gradle build files, this is required
4       for non gradle builds-->
5     <!--<uses-sdk android:minSdkVersion="21"/>-->
6     <application>
7       <provider
8         android:name="com.google.firebaseio.provider.FirebaseInitProvider"
9         android:authorities="${applicationId}.firebaseinitprovider"
10        android:directBootAware="true"
11        android:exported="false"
12        android:initOrder="100" /> <!-- initOrder로 초기화 순서 제어 -->
13      </application>
14    </manifest>
```

이 패턴은 필수 리소스나 라이브러리가 앱 프로세스 생명주기의 초반에 자동으로 초기화되도록 보장하여 더 깔끔하고 모듈화된 설계를 제공합니다. ContentProvider의 또 다른 사용 사례는 애플리케이션 시작 시 컴포넌트를 간단하고 효율적으로 초기화 할 수 있는 방법을 제공하는 Jetpack App Startup²⁶ 라이브러리입니다. 내부 구현은 InitializationProvider라는 클래스를 사용하고, 이는 ContentProvider를 활용하여 Initializer 인터페이스를 구현하는 모든 사전 정의된 클래스 정보를 가져와 초기화합니다. 이는 Application.onCreate() 메서드가 호출되기 전에 초기화 로직이 처리되도록 보장합니다.

다음은 App Startup 라이브러리에서 중추적인 역할을 하는 InitializationProvider의 내부 구현을 그대로 가져온 모습입니다.

²⁶<https://developer.android.com/topic/libraries/app-startup>

```
1 /**
2  * The {@link ContentProvider} which discovers {@link Initializer}s in an application
3  * and
4  * initializes them before {@link Application#onCreate()}.
5 */
6
7     @Override
8     public final boolean onCreate() {
9         Context context = getContext();
10        if (context != null) {
11            Context applicationContext = context.getApplicationContext();
12            if (applicationContext != null) {
13                // Initialize all registered Initializer classes.
14                AppInitializer.getInstance(context).discoverAndInitialize(getClass());
15            } else {
16                StartupLogger.w("Deferring initialization because `applicationContext`"
17                " is null.");
18            }
19        } else {
20            throw new StartupException("Context cannot be null");
21        }
22        return true; // Provider 생성 성공 표시
23    }
24    // 다른 ContentProvider 메서드들은 null 또는 예외를 반환/던집니다.
25 }
```

위 코드에서 `onCreate()` 메서드는 `AppInitializer.getInstance(context).discoverAndInitialize(getClass())` 호출하며, 이는 `Application` 생명주기가 시작되기 전에 등록된 모든 `Initializer` 구현을 자동으로 검색하고 초기화합니다. 이를 통해 `Application.onCreate()` 메서드를 난잡하게 만들지 않고 앱 컴포넌트를 모듈식으로 분리하여 보다 깔끔하고 효율적으로 초기화할 수 있습니다.

Q) 12. 구성 변경(configuration changes)을 어떻게 처리하나요?

안드로이드에서 구성 변경을 올바르게 처리하는 것은 화면 회전, 언어 변경, 다크/라이트 모드 전환, 글꼴 크기 또는 두께 조정과 같은 앱 설정에 변경되었을 때 원활한 사용자 경험을 유지하는 데 중요한 역할을 합니다. 기본적으로 안드로이드 시스템은 구성 변경이 발생할 때 Activity를 다시 시작하며, 이로 인해 일시적으로 UI의 상태가 손실될 수 있습니다. 따라서 구성 변경에 적절하게 대응하려면 아래와 같은 전략을 고려해 볼 수 있습니다.

1. **UI 상태 저장 및 복원**: `onSaveInstanceState()` 및 `onRestoreInstanceState()` 메서드를 구현하여 Activity 재생성 중 UI 상태를 보존하고 복원합니다. 이를 통해 사용자는 구성 변경 후에도 동일한 상태로 돌아갈 수 있습니다.
2. **Jetpack ViewModel**: `ViewModel` 클래스를 활용하여 구성 변경에도 유지되어야 하는 UI 관련 데이터를 저장합니다. `ViewModel` 객체는 Activity 재시작 범위를 넘어서 존재하도록 설계되었으므로, 구성 변경이 발생했을 때 데이터를 보존하고 다시 복원하는 데 이상적입니다.
3. **구성 변경 수동으로 처리하기**: 애플리케이션이 특정 구성 중에 리소스를 업데이트 할 필요가 없고 Activity 재시작을 피하고 싶다면, `AndroidManifest.xml`에서 Activity가 처리할 구성 변경 사항을 `android:configChanges` 속성을 사용하여 선언할 수 있습니다. 그 후, `onConfigurationChanged()` 메서드를 재정의하여 해당 변경 사항이 발생했을 때 수동으로 관리합니다.
4. **Jetpack Compose에서 rememberSaveable 활용**: Jetpack Compose에서는 `rememberSaveable`를 사용하여 구성 변경이 발생했을 때 UI 상태를 보존할 수 있습니다. 이는 `onSaveInstanceState()`와 유사하게 작동하지만 Compose에 특화되어 있으며 Composable 상태를 일관되게 유지하는 데 도움이 됩니다. 이에 대해서는 [Chapter 1: Jetpack Compose Interview Questions](#)에서 더 자세히 살펴볼 예정입니다.

추가 팁

- **내비게이션 및 백 스택 보존**: Navigation 컴포넌트를 사용하면 구성 변경 시 내비게이션 백 스택이 보존됩니다.
- **앱 구성에 의존적인 데이터 피하기**: 가능하면 앱 구성(Configuration, 앱의 설정)에 의존적인 값을 UI 레이아웃에 직접 저장하지 않는 것이 좋습니다. 꼭 해야 한다면, 구성 변경 시 데이터를 보존하도록 설계된 `ViewModel`과 같은 대안을 고려하실 수 있습니다.

요약

구성 변경을 올바르게 처리하는 것은 사용자 데이터가 예상치 않게 손실되는 상황을 방지하여 사용자 경험을 향상시키는 데 중요한 역할을 합니다. 구성 변경 처리에 대한 포괄적인 가이드는 [공식 안드로이드 문서](#)²⁷를 참고하실 수 있습니다.

실전 질문

- Q) 구성 변경(Configuration Change)을 처리하기 위한 전략에는 무엇이 있으며, ViewModel은 구성 변경으로부터 손실될 수 있는 UI 관련 데이터를 어떻게 보존하나요?
- Q) AndroidManifest 파일에서 android:configChanges 속성은 Activity 생명주기와 동작에 어떤 영향을 미치며, Activity 재시작에 의존하는 것이 아니라 onConfigurationChanged() 메서드를 사용해야 하는 시나리오의 예시를 들어주세요.

Q) 13. 안드로이드는에서 메모리를 어떻게 효율적으로 관리하며, 메모리 누수(memory leaks)를 어떻게 방지하는지 설명해주세요.

안드로이드는 사용되지 않는 메모리를 자동으로 회수하여 활성 중인 애플리케이션 및 서비스에게 효율적인 메모리 할당을 보장하는 **가비지 컬렉션(garbage collection)** 메커니즘을 통해 메모리를 관리합니다. 이는 관리형 메모리 환경에 의존하므로, 개발자는 C++과 같은 언어에서처럼 메모리를 수동으로 할당하고 해제할 필요가 없습니다. Dalvik 또는 ART 런타임(이 장 뒷부분에서 살펴볼 예정입니다)은 메모리 사용량을 모니터링하고, 더 이상 참조되지 않는 객체를 정리하며, 과도한 메모리 소비를 방지합니다.

안드로이드는 시스템 메모리가 부족할 때 포그라운드 애플리케이션의 원활한 작동을 우선시하며 백그라운드 프로세스를 종료하기 위해 **low-memory killer**를 사용합니다. 따라서 개발자는 시스템 성능에 미치는 영향을 최소화하기 위해 앱이 리소스를 효율적으로 사용하도록 해야 합니다.

안드로이드에서 메모리 누수의 원인

메모리 누수는 애플리케이션이 더 이상 필요하지 않은 객체에 대한 참조를 유지하여 가비지 컬렉터가 메모리를 회수하지 못하게 할 때 발생합니다. 일반적인 원인으로는 부적절한 생명주기 관리, 정적 참조 또는 Context에 대한 장기 참조 유지 등이 원인입니다.

²⁷<https://developer.android.com/guide/topics/resources/runtime-changes>

메모리 누수를 피하기 위한 모범 사례

1. **생명주기를 인지하는 컴포넌트 사용:** ViewModel, `collectAsStateWithLifecycle`²⁸과 함께 사용하는 Flow 또는 LiveData와 같은 생명주기를 인지하는 컴포넌트를 활용하면 관련 생명주기가 끝날 때 리소스가 적절하게 해제됩니다. 이러한 컴포넌트는 연관된 생명주기가 더 이상 활성 상태가 아니거나 특정 상태로 전환될 때 자동으로 객체를 정리하여 메모리 누수를 방지합니다.
2. **Context에 대한 오랜 참조 피하기:** 정적 필드나 싱글톤과 같은 오래 지속되는 객체에서 Activity 또는 Context에 대한 참조를 유지하지 않아야 합니다. 만약 Context에 대해서 장기적인 참조가 필요하다면, Activity나 Fragment의 생명주기와는 독립적인 ApplicationContext를 사용하는 것이 좋습니다.
3. **리스너 및 콜백 등록 올바르게 해제하기:** 항상 적절한 생명주기 메서드에서 리스너, 관찰자 또는 콜백을 등록 올바르게 해제해야 합니다. 가령, onPause() 또는 onStop()에서 BroadcastReceiver를 등록을 해제해야 앱이 백그라운드에 있을 때 참조를 유지하지 않습니다.
4. **중요하지 않은 객체는 WeakReference 사용하기:** 강력한 참조가 필요하거나, 장기적으로 참조가 보장되어야 하는 객체가 아니라면 WeakReference를 사용하는 것이 좋습니다. WeakReference로 참조하는 객체는 메모리가 필요할 때 가비지 컬렉터가 해당 객체를 언제든지 회수할 수 있습니다.
5. **누수 감지 툴 사용:** 개발 중에 메모리 누수를 탐지하고 문제를 수정하기 위해 `LeakCanary`²⁹와 같은 툴을 활용할 수 있습니다. 이 라이브러리는 어떤 객체가 메모리 누수를 일으키는지, 어떻게 해결하는지에 대한 방법을 제공합니다. 또한, Android Studio의 `Memory Profiler`³⁰를 사용하여 버벅거림, 멈춤 현상, 심지어 앱 크래시로 이어질 수 있는 메모리 누수 및 메모리 변동(churn)을 식별할 수 있습니다.
6. **View에 대한 정적 참조 피하기:** View는 Activity 컨텍스트에 대한 참조를 유지하여 메모리 누수를 유발할 수 있으므로 정적 필드에 저장해서는 안 됩니다.
7. **리소스 닫기:** 파일 스트림, Cursor 또는 데이터베이스 연결과 같은 리소스는 더 이상 필요하지 않을 때 항상 명시적으로 해제합니다. 가령, 데이터베이스 쿼리 후 Cursor를 닫습니다.
8. **Fragment와 Activity 현명하게 사용하기:** Fragment를 과도하게 사용하거나 부적절하게 참조를 유지하지 않습니다. onDestroyView() 또는 onDetach()에서 Fragment 참조를

²⁸<https://developer.android.com/reference/kotlin/androidx/lifecycle/compose/package-summary#extension-functions>

²⁹<https://square.github.io/leakcanary/>

³⁰<https://developer.android.com/studio/profile/memory-profiler>

정리합니다.

요약

안드로이드의 메모리 관리는 효율적이지만 메모리 누수를 방지하기 위해 개발자가 모범 사례를 따라야 합니다. 생명주기 인식 컴포넌트를 사용하고, Context나 View에 대한 정적 참조를 피하며, LeakCanary와 같은 도구를 활용하면 누수 가능성을 크게 줄일 수 있습니다. 적절한 생명주기 이벤트 중 리소스를 관리하고 정리하면 앱 성능과 사용자 경험이 향상됩니다.

실전 질문

Q) 애플리케이션에서 메모리 누수의 일반적인 원인은 무엇이며, 이를 사전에 방지하기 위한 방법에는 어떤 것들이 있나요?

Q) 안드로이드의 가비지 컬렉션 메커니즘은 어떻게 작동하며, 개발자는 앱에서 메모리 누수를 감지하고 수정하기 위해 어떤 방법을 사용할 수 있나요?

Q) 14. ANR 이란 무엇인지, ANR이 발생하는 주요 원인은 무엇이며, 어떻게 예방할 수 있는지 설명해주세요.

ANR (Application Not Responding) 은 앱의 메인 스레드(UI 스레드)가 너무 오랫동안, 통상적으로 5초 이상 차단(blocking)될 때 발생하는 안드로이드의 시스템 오류입니다. ANR이 발생하면 안드로이드는 사용자에게 앱을 닫거나 응답을 기다리도록 안내합니다. ANR이 자주 발생한다면 사용자 경험을 저하시키는데, 이는 다음과 같은 다양한 요인에 의해 발생할 수 있습니다.

- 메인 스레드에서 5초 이상 걸리는 무거운 작업
- 장시간 실행되는 네트워크 또는 데이터베이스 등의 I/O 작업
- UI 스레드 차단 작업 (가령, UI 스레드에서의 동기 작업 등)

ANR 예방 방법

ANR을 예방하려면 무겁거나 시간이 많이 걸리는 작업을 오프로드하여 메인 스레드의 응답성을 유지하는 것이 중요합니다. 몇 가지 모범 사례를 살펴보겠습니다.

1. **무거운 작업을 메인 스레드 밖으로 이동**: 파일 I/O, 네트워크 요청 또는 데이터베이스 쿼리와 같은 무거운 작업을 처리하기 위해서는 백그라운드 스레드(e.g., AsyncTask, Executors 또는 Thread)를 사용합니다. 더 모던하고 안전한 접근 방식으로 백그라운드 작업을 처리하고 싶다면 Kotlin Coroutines에서 제공하는 Dispatchers.IO를 활용할 수 있습니다.
2. **WorkManager 사용하기**: 데이터 동기화와 같이 백그라운드에서 실행되어야 하는 장기적인 작업에는 WorkManager³¹를 사용합니다 (이에 대해서는 **카테고리 3: 비즈니스 로직**에서 더 자세히 살펴볼 예정입니다). WorkManager는 개발자가 필요한 작업을 사전에 스케줄링하고, 메인 스레드 외부에서 실행되도록 보장합니다.
3. **데이터 불러오기 최적화**: 대규모 데이터 셋을 효율적으로 처리하기 위해 Paging을 구현하여 데이터를 작고 관리 가능한 청크로 가져와 UI 과부하를 방지하고 성능을 향상시킬 수 있습니다.
4. **구성 변경 시 UI 작업 최소화**: ViewModel을 활용하여 UI 관련 데이터를 유지하고 화면 회전과 같은 구성 변경 중에 불필요한 UI 렌더링을 피합니다.
5. **Android Studio로 모니터링 및 프로파일링**: Android Studio Profiler 도구를 활용하여 CPU, 메모리 및 네트워크 사용량을 모니터링합니다. ANR을 유발할 수 있는 성능 병목 현상을 식별하고 해결하는 데 도움이 됩니다.
6. **블로킹(blocking) 호출 피하기**: 원활한 앱 성능을 보장하기 위해 메인 스레드에서 긴 루프, sleep 호출 또는 네트워크 요청을 동기로 수행하여 UI 스레드를 블로킹하는 작업을 수행하지 않습니다.
7. **가벼운 지연 작업에 Handler 사용**: 응답성 있는 앱 사용자 경험을 위해 Thread.sleep() 대신 Handler.postDelayed()를 사용하여 메인 스레드를 차단하지 않고 가벼운 지연 작업을 처리할 수 있습니다.

요약

ANR (Application Not Responding)은 앱의 메인 스레드(UI 스레드)가 일반적으로 5초 이상 차단될 때 발생하는 안드로이드 오류이며, 사용자 경험을 저하시키고 현재 UI 상태를 모두

³¹<https://developer.android.com/topic/libraries/architecture/workmanager>

잃게 합니다. ANR을 방지하려면 네트워크로부터 데이터 요청, 데이터베이스 쿼리, 무거운 계산 작업 수행과 같은 과부하 작업을 백그라운드 스레드로 이동하여 메인 스레드를 가볍게 유지해야 합니다. 또한 페이징(Paging) 등을 활용하여 데이터 핸들링 작업을 최적화하고 [Android Studio Profiler](#)³²를 사용하여 앱을 프로파일링할 수 있습니다. 자세한 내용은 [ANR에 대한 공식 안드로이드 문서](#)³³를 참조하실 수 있습니다.

실전 질문

Q) ANR을 진단하고 앱 성능과 유저 경험을 개선해보신 경험이 있으신가요?

Q) 15. 딥 링크(deep links)를 어떻게 처리하는지 설명해주세요.

딥 링크(Deep links)³⁴는 사용자가 URL이나 알림과 같은 외부 소스에서 앱 내의 특정 화면이나 기능으로 직접 이동할 수 있도록 합니다. 딥 링크를 처리하려면 `AndroidManifest.xml`에서 이를 정의하고 해당 Activity나 Fragment에서 들어오는 Intent를 처리해야 합니다.

1단계: 매니페스트에서 딥 링크 정의하기

딥 링킹을 활성화하려면 딥 링크를 처리해야 하는 Activity에 대해 `AndroidManifest.xml`에서 `intent filter`를 선언합니다. `intent filter`는 앱이 응답하는 URL 구조 또는 스키마(scheme)를 지정합니다.

그림 39. `AndroidManifest.xml`

```
1 <activity android:name=".MyDeepLinkActivity"
2     android:exported="true">
3     <intent-filter>
4         <action android:name="android.intent.action.VIEW" />
5         <category android:name="android.intent.category.DEFAULT" />
6         <category android:name="android.intent.category.BROWSABLE" />
7         <!-- 딥 링크 URI 정의 -->
8         <data
```

³²<https://developer.android.com/studio/profile>

³³<https://developer.android.com/topic/performance/vitals/anr>

³⁴<https://developer.android.com/training/app-links/deep-linking>

```
9      android:scheme="https"
10     android:host="example.com"
11     android:pathPrefix="/deepLink" />
12   </intent-filter>
13 </activity>
```

- android:scheme: URL 스키마(가령, https)를 지정합니다.
- android:host: 도메인(가령, example.com)을 지정합니다.
- android:pathPrefix: URL의 경로(가령, /deepLink)를 정의합니다.

이 설정을 통해 `https://example.com/deepLink`와 같은 URL이 `MyDeepLinkActivity`를 열도록 허용합니다.

2단계: Activity에서 딥 링크 처리하기

Activity 내부에서 들어오는 Intent 데이터를 검색하고 처리하여 적절한 화면으로 이동하거나 작업을 수행합니다.

그림 40. `MyDeepLinkActivity.kt`

```
1 class MyDeepLinkActivity : AppCompatActivity() {
2
3     override fun onCreate(savedInstanceState: Bundle?) {
4         super.onCreate(savedInstanceState)
5         setContentView(R.layout.activity_my_deep_link)
6
7         // Intent 데이터 가져오기
8         val intentData: Uri? = intent?.data
9         if (intentData != null) {
10             // 쿼리 파라미터 검색 예시
11             val id = intentData.getQueryParameter("id")
12             navigateToFeature(id)
13         }
14     }
}
```

```

15
16  private fun navigateToFeature(id: String?) {
17      // 딥 링크 데이터를 기반으로 특정 화면으로 이동
18      if (id != null) {
19          Toast.makeText(this, "Navigating to item: $id", Toast.LENGTH_SHORT).show()
20
21          // 실제 내비게이션 또는 작업 수행
22          // navigate(..) or doSomething(..)
23      }
24  }
25 }
```

3단계: 딥 링크 테스트하기

딥 링크를 테스트하려면 아래의 adb 명령어를 사용할 수 있습니다.

```

1 adb shell am start -a android.intent.action.VIEW \
2 -d "https://example.com/deepLink?id=123" \
3 com.example.myapp # 앱 패키지 이름
```

위의 명령어는 딥 링크 동작을 시뮬레이션 하기 위해 앱을 실행합니다.

추가 고려 사항

- 커스텀 스키마(Custom Schemes)**: 앱 내부적으로 실행하는 딥링크에 대해서는 커스텀 스키마(예를들어, myapp://)를 사용할 수 있지만, 경우에 따라 더 넓은 호환성을 위해 HTTP(S) URL을 선호합니다.
- 내비게이션(Navigation)**: 딥 링크 데이터를 기반으로 앱 내의 다른 Activity나 Fragment로 이동하기 위해 Intent를 사용합니다.
- 폴백 처리(Fallback Handling)**: 앱이 딥 링크 데이터가 유효하지 않거나 불완전한 경우를 처리하여 더 나은 사용자 경험을 제공할 수 있습니다.
- App Links**: HTTP(S) 딥 링크가 브라우저 대신 앱에서 직접 열리도록 하려면 App Links³⁵를 설정해야 합니다.

³⁵<https://developer.android.com/studio/write/app-link-indexing>

요약

딥 링크 처리는 `AndroidManifest.xml`에서 URL 패턴을 정의하고 대상 `Activity`에서 데이터를 처리하는 것을 포함합니다. 딥 링크 데이터를 추출하고 그에 맞는 동작을 수행함으로써 사용자를 앱의 특정 기능이나 콘텐츠로 네비게이팅하여 사용자 경험과 참여를 향상시키는 등 다양한 전략으로 사용될 수 있습니다.

실전 질문

Q) 안드로이드에서 딥 링크를 어떻게 테스트하고, 다양한 기기와 시나리오에서 올바르게 작동하는지 확인하기 위해 사용하는 디버깅 기법이 있다면 설명해 주세요.

Q) 16. 태스크(tasks)와 백 스택(back stack)이란 무엇인가요?

태스크(Task)³⁶는 사용자가 특정 목표를 달성하기 위해 상호 작용하는 `Activity`의 집합입니다. 태스크는 백 스택(back stack)으로 구성되며, 이는 `Activity`가 시작될 때 추가되고 사용자가 뒤로 이동하거나 시스템이 리소스를 회수할 때 제거되는 후입선출(LIFO) 구조로 이루어져 있습니다.

태스크 (Tasks)

태스크는 일반적으로 런처나 `Intent`를 통해 `Activity`가 실행될 때 시작됩니다. 태스크는 `Intent` 및 `Activity` 런치 모드(launch modes)가 어떻게 구성되었는지에 따라 여러 애플리케이션과 해당 `Activity`에 속해 있을 수 있습니다. 가령, 이메일 앱에서 링크를 클릭하면 동일한 태스크의 일부로 브라우저가 열릴 수 있습니다. 태스크는 연관된 `Activity`가 소멸될 때까지 활성 상태를 유지합니다.

백 스택 (Back Stack)

백 스택은 태스크 내 `Activity`의 기록을 유지합니다. 사용자가 새 `Activity`로 이동하면 현재 `Activity`가 스택에 푸시(push)됩니다. 뒤로 가기 버튼을 누르면 스택의 맨 위 `Activity`가 팝(pop)되어 그 아래에 있던 `Activity`가 재개됩니다. 이 메커니즘은 직관적인 탐색과 사용자 워크플로우의 연속성을 보장합니다.

³⁶<https://developer.android.com/guide/components/activities/tasks-and-back-stack>

태스크와 백 스택은 Activity 런치 모드와 인텐트 플래그(intent flags)의 영향을 받습니다. 런치 모드와 Intent 플래그는 태스크 및 백 스택 내 Activity의 동작을 제어하는 데 사용되는 메커니즘입니다. 이러한 구성을 통해 개발자는 Activity가 어떻게 시작되고 다른 Activity와 어떻게 상호 작용하는지 정의할 수 있습니다.

런치 모드 (Launch Modes)

Note: 점심 모드 아닙니다.

런치 모드는 Activity가 어떻게 인스턴스화되고 백 스택에서 처리되는지를 결정합니다. 안드로이드에는 네 가지의 기본적인 런치 모드가 있습니다.

1. **standard**: 기본 런치 모드입니다. 인스턴스가 이미 존재하더라도 Activity가 시작될 때마다 새 인스턴스가 생성되어 백 스택에 추가됩니다.
2. **singleTop**: Activity의 인스턴스가 이미 백 스택의 맨 위에 있는 경우 새 인스턴스가 생성되지 않습니다. 대신 기존 인스턴스가 `onNewIntent()`에서 Intent를 처리합니다.
3. **singleTask**: 태스크 내에 Activity의 인스턴스가 하나만 존재합니다. 인스턴스가 이미 존재하는 경우 맨 앞으로 가져오고 `onNewIntent()`가 호출됩니다. 이는 앱의 진입점 역할을 하는 Activity에 유용합니다.
4. **singleInstance**: `singleTask`과 유사하지만, Activity가 다른 Activity와 분리된 고유한 태스크에 배치됩니다. 이는 동일한 태스크에 다른 Activity가 포함될 수 없도록 보장합니다.

인텐트 플래그 (Intent Flags)

인텐트 플래그는 Intent가 전송될 때 Activity가 시작되는 방식이나 백 스택의 동작을 수정하는 데 사용됩니다. 일반적으로 사용되는 몇 가지 플래그는 다음과 같습니다.

- **FLAG_ACTIVITY_NEW_TASK**: 새 태스크에서 Activity를 시작하거나, 태스크가 이미 존재하는 경우 해당 태스크를 맨 앞으로 가져옵니다.
- **FLAG_ACTIVITY_CLEAR_TOP**: Activity가 이미 백 스택에 있는 경우, 그 위에 있는 모든 Activity가 날아가고 기존 인스턴스가 Intent를 처리합니다.
- **FLAG_ACTIVITY_SINGLE_TOP**: Activity가 백 스택의 맨 위에 있는 경우 새 인스턴스가 생성되지 않도록 보장합니다. 이는 종종 다른 플래그와 함께 사용됩니다.

- **FLAG_ACTIVITY_NO_HISTORY**: Activity가 백 스택에 추가되는 것을 방지하여 종료 후에도 유지되지 않도록 합니다.

사용 사례

- **런치 모드(Launch Modes)**는 주로 AndroidManifest.xml 파일의 <activity> 태그 아래에 선언되어 개발자가 Activity의 기본 동작을 설정할 수 있도록 합니다.
- **인텐트 플래그 (Intent Flags)**는 Intent를 생성할 때 개발자가 유동적으로 플래그를 설정할 수 있는 방식으로 작동하여, 특정 시나리오에 대해 더 많은 유연성을 제공합니다.

예제

그림 41. Intent with Flags

```

1 val intent = Intent(this, SecondActivity::class.java).apply {
2     flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TOP
3 }
4 startActivity(intent)

```

위의 예제에서 백 스택에 SecondActivity가 존재하지 않는 경우, 새 태스크에서 시작됩니다. 백 스택에 이미 SecondActivity가 존재하는 경우 그 위에 있는 모든 액티비티가 제거되고, 기존 SecondActivity 인스턴스가 Intent를 처리합니다.

요약

태스크와 백 스택은 Activity의 생명주기와 탐색 기록을 관리하여 사용자 친화적인 워크플로우를 가능하게 하는 안드로이드 내비게이션 모델의 핵심입니다. 런치 모드는 Activity가 태스크 내에서 어떻게 시작되고 관리되는지에 대한 기본 동작을 정의하는 반면, 인텐트 플래그는 유사한 동작에 대해 런타임에 제어할 수 있습니다. 이 두 가지를 함께 사용하면 Activity 생명주기와 백 스택 탐색을 정밀하게 관리할 수 있습니다. 자세한 내용은 [Tasks and the back stack³⁷](#) 문서를 통해 추가적으로 학습하실 수 있습니다.

³⁷<https://developer.android.com/guide/components/activities/tasks-and-back-stack>

실전 질문

Q) `singleTask`와 `singleInstance`의 차이점은 무엇이며, 각각 어떤 시나리오에서 사용해야 하나요?

Q) Activity 런치 모드에는 각각 어떤 타입이 존재하며, 태스크 및 백 스택 동작에 어떤 영향을 미치나요?

Q) 17. Bundle의 사용 목적에 대해서 설명해 주세요.

Bundle은 Activity, Fragment, Service와 같은 컴포넌트 간에 데이터를 전달하는 데 사용되는 키-값 쌍 데이터 구조입니다. 일반적으로 앱 내에서 작은 용량의 데이터를 효율적으로 전송하는 데 사용됩니다. Bundle은 가볍고 안드로이드 운영 체제가 쉽게 관리하고 전송할 수 있는 형식으로 데이터를 직렬화하도록 설계되었습니다.

Bundle의 일반적인 사용 사례

1. **Activity 간 데이터 전달**: 새 Activity를 시작할 때 Intent에 Bundle을 담아서 대상 Activity에 데이터를 전달할 수 있습니다.
2. **Fragment 간 데이터 전달**: Fragment 트랜잭션에서 Bundle은 getArguments() 및 getArguments()와 함께 전달되어 Fragment 간에 데이터를 보냅니다.
3. **인스턴스 상태 저장 및 복원**: Bundle은 onSaveInstanceState() 및 onRestoreInstanceState()와 같은 생명주기 메서드에서 구성 변경 중에 임시 UI 상태를 저장하고 복원하는 데 사용됩니다.
4. **Service에 데이터 전달**: Service를 시작하거나 바인딩된 Service에 데이터를 전달할 때 Bundle을 통해 데이터를 운반할 수 있습니다.

Bundle 작동 방식

Bundle은 데이터를 키-값 구조로 직렬화하여 작동합니다. 키는 문자열이며 값은 기본 유형, Serializable, Parcelable 객체 또는 다른 Bundle일 수 있습니다. 이를 통해 데이터를 효율적으로 저장하고 전송할 수 있습니다.

예제: Activity 간 데이터 전달

그림 42. Passing Data Between Activities

```

1 // Sending data from Activity A
2 val intent = Intent(this, ActivityB::class.java).apply {
3     putExtra("user_name", "John Doe")
4     putExtra("user_age", 25)
5 }
6 startActivity(intent)
7
8 // Receiving data in Activity B
9 val name = intent.getStringExtra("user_name")
10 val age = intent.getIntExtra("user_age", -1)

```

위의 예제에서 데이터는 Intent.putExtra()를 통해 내부적으로 Bundle에 패키징됩니다.

예제: Fragment 간 데이터 전달**그림 43. Passing Data Between Fragments**

```

1 // Sending data to Fragment
2 val fragment = MyFragment().apply {
3     arguments = Bundle().apply {
4         putString("user_name", "Jane Doe")
5        .putInt("user_age", 30)
6     }
7 }
8
9 // Retrieving data in Fragment
10 val name = arguments?.getString("user_name")
11 val age = arguments?.getInt("user_age")

```

위의 예제에서 데이터는 Fragment의 arguments를 통해 내부적으로 Bundle에 패키징됩니다.

예제: 상태 저장 및 복원

그림 44. Saving and Restoring Instance State

```
1 override fun onSaveInstanceState(outState: Bundle) {  
2     super.onSaveInstanceState(outState)  
3     outState.putString("user_input", editText.text.toString())  
4 }  
5  
6 override fun onRestoreInstanceState(savedInstanceState: Bundle) {  
7     super.onRestoreInstanceState(savedInstanceState)  
8     val userInput = savedInstanceState.getString("user_input")  
9     editText.setText(userInput)  
10 }
```

위의 예시에서 `Bundle`은 화면 회전과 같은 구성 변경으로부터 사용자가 입력했던 값이 보존되도록 합니다.

요약

`Bundle`은 컴포넌트 및 생명주기 이벤트 간에 데이터를 효율적으로 전달하고 보존하기 위한 안드로이드의 중요한 구성 요소입니다. 가볍고 유연한 구조로 인해 애플리케이션 상태 및 데이터 전송 관리에 필수적인 도구입니다.

실전 질문

Q) 구성 변경 중 `onSaveInstanceState()`는 UI 상태를 보존하기 위해 `Bundle`을 어떻게 활용하며, `Bundle`에 어떤 유형의 데이터를 담을 수 있나요?

Q) 18. Activity 또는 Fragment 간에 데이터를 어떻게 전달하나요?

Activity 또는 **Fragment** 간의 데이터 전송은 상호 작용적이고 동적인 화면을 구성하는 데 중요한 역할을 합니다. 안드로이드는 앱 아키텍처를 준수하면서 원활한 통신을 보장하기 위해 이를 위한 다양한 메커니즘을 제공합니다.

Activity 간 데이터 전달

한 Activity에서 다른 Activity로 데이터를 전달하는 가장 일반적인 메커니즘은 Intent입니다. 데이터는 키-값 쌍(putExtra())의 형태로 Intent에 추가되고, 수신하는 Activity는 getIntent()을 사용하여 해당 값을 가져옵니다.

그림 45. Passing Data Between Activities

```
1 // 데이터를 전달하는 Activity
2 val intent = Intent(this, SecondActivity::class.java).apply {
3     putExtra("USER_NAME", "John Doe")
4     putExtra("USER AGE", 25)
5 }
6 startActivity(intent)
7
8 // 데이터를 수신하는 Activity
9 class SecondActivity : AppCompatActivity() {
10     override fun onCreate(savedInstanceState: Bundle?) {
11         super.onCreate(savedInstanceState)
12         setContentView(R.layout.activity_second)
13
14         val userName = intent.getStringExtra("USER_NAME")
15         val userAge = intent.getIntExtra("USER AGE", 0)
16         Log.d("SecondActivity", "User Name: $userName, Age: $userAge")
17     }
18 }
```

Fragment 간 데이터 전달

Fragment 간 데이터 전달에도 Bundle을 사용할 수 있습니다. 보내는 Fragment는 키-값 쌍의 형태로 Bundle을 생성하고 arguments를 통해 수신하는 Fragment로 전달합니다.

그림 46. Passing Data Between Fragments

```
1 // 데이터를 전달하는 Fragment
2 val fragment = SecondFragment().apply {
3     arguments = Bundle().apply {
4         putString("USER_NAME", "John Doe")
5         putInt("USER AGE", 25)
6     }
7 }
8 parentFragmentManager.beginTransaction()
9     .replace(R.id.fragment_container, fragment)
10    .commit()
11
12 // 데이터를 수신하는 Fragment
13 class SecondFragment : Fragment() {
14     override fun onCreateView(
15         inflater: LayoutInflater, container: ViewGroup?,
16         savedInstanceState: Bundle?
17     ): View? {
18         val view = inflater.inflate(R.layout.fragment_second, container, false)
19
20         val userName = arguments?.getString("USER_NAME")
21         val userAge = arguments?.getInt("USER AGE")
22         Log.d("SecondFragment", "User Name: $userName, Age: $userAge")
23
24         return view
25     }
26 }
```

Jetpack Navigation 라이브러리로 Fragment 간 데이터 전달하기

Jetpack Navigation³⁸ 라이브러리가 지원하는 Safe Args³⁹ 플러그인을 사용하면 대상 간 타입-세이프(type-safe) 내비게이션과 인수 전달을 가능하게 하는 direction 및 argument 클래스가 컴파일 타임에 자동적으로 생성되어, 런타임 시 값을 안전하게 가져올 수 있습니다.

1. 내비게이션 그래프에서 인수(argument) 정의하기

아래의 nav_graph.xml 예시에서 username라는 String 타입의 인수를 정의하고 있습니다.

```

1 <fragment
2     android:id="@+id/secondFragment"
3     android:name="com.example.SecondFragment">
4     <argument
5         android:name="username"
6         app:argType="string" />
7 </fragment>
```

2. 소스 프래그먼트에서 데이터 전달하기

Safe Args 플러그인은 컴파일 타임에 대상 객체와 빌더 클래스를 생성하여 아래 예시와 같이 인수를 안전하고 명시적으로 전달할 수 있도록 합니다.

그림 47. FirstFragment.kt

```

1 val action = FirstFragmentDirections
2     .actionFirstFragmentToSecondFragment(username = "skydoves")
3 findNavController().navigate(action)
```

3. 대상 프래그먼트에서 데이터 검색하기

마지막으로, 아래 코드와 같이 전달된 인수를 통해 데이터를 가져올 수 있습니다.

³⁸<https://developer.android.com/guide/navigation>

³⁹<https://developer.android.com/guide/navigation/use-graph/safe-args>

그림 48. SecondFragment.kt

```
1 val username = arguments?.let {  
2     SecondFragmentArgs.fromBundle(it).username  
3 }
```

Safe Args를 사용하여 정의된 인수를 컴파일 타임에 검사하여 정적인 코드로 만들고, 런타임에 해당 인수 값을 안전하게 가져옴으로써 런타임 오류를 줄일 수 있습니다. 또한, 해당 플러그인에서 규정하는 컨벤션을 통하여 클래스 및 메서드를 생성해 주기 때문에, Fragment 간의 데이터 핸들링 관련 코드의 가독성을 향상시킬 수 있습니다.

Shared ViewModel 사용하기

동일한 Activity 내에서 Fragment끼리 통신해야 하는 경우 **Shared ViewModel**을 고려해 볼 수 있습니다. **Shared ViewModel**은 동일한 Activity 내의 여러 Fragment 간에 공유되는 ViewModel 인스턴스를 의미합니다. 이는 Jetpack의 androidx.fragment:fragment-ktx 패키지에서 제공하는 activityViewModels() 메서드를 사용하여 구현할 수 있습니다. 해당 메서드는 ViewModel의 범위를 Activity로 지정하여 Fragment가 동일한 ViewModel 인스턴스에 접근하고 공유할 수 있도록 합니다. 이 방법을 통해 Fragment끼리 의존성이 생기는 것을 피하고, **Shared ViewModel**을 통해 반응형으로 상호작용할 때 각 Fragment의 생명주기에 따라 안전하게 데이터를 수신할 수 있습니다. 아래의 예시를 통해서 어떤 의미인지 자세히 살펴보실 수 있습니다.

그림 49. Using a Shared ViewModel

```
1 // Shared ViewModel  
2 class SharedViewModel: ViewModel() {  
3  
4     private val _userData = MutableStateFlow<User?>(null)  
5     val userData : StateFlow<User?>  
6  
7         fun setUserData(user: User) {  
8             _userData.value = user  
9         }  
10    }
```

```
11
12 // Fragment A (데이터를 전송하는 Fragment)
13 class FirstFragment : Fragment() {
14     private val sharedViewModel: SharedViewModel by activityViewModels()
15
16     fun updateUser(user: User) {
17         sharedViewModel.setUserData(user)
18     }
19 }
20
21 // Fragment B (데이터를 수신하는 Fragment)
22 class SecondFragment : Fragment() {
23     private val sharedViewModel: SharedViewModel by activityViewModels()
24
25     override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
26         lifecycleScope.launch {
27             viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.RESUMED) {
28                 sharedViewModel.userData.collectLatest { user ->
29                     ..
30                 }
31             }
32         }
33     }
34 }
35
36 // Activity (Activity에서 데이터 수신하기)
37 class MainActivity : ComponentActivity() {
38     private val sharedViewModel: SharedViewModel by viewModels()
39
40     override fun onCreate(savedInstanceState: Bundle?) {
41         lifecycleScope.launch {
42             lifecycle.repeatOnLifecycle(Lifecycle.State.RESUMED) {
43                 sharedViewModel.userData.collectLatest { user ->
44                     ..
```

```
45         }
46     }
47 }
48 }
49 }
```

요약

1. Intent는 putExtra()와 getIntent()를 사용하여 Activity 간 데이터를 전달하는 데 사용할 수 있습니다.
2. Bundle은 arguments 속성을 통해 Fragment 간 데이터를 전달하는 데 사용할 수 있습니다.
3. Jetpack Navigation과 Safe Args 플러그인을 사용하면 컴파일 타임에 생성된 direction 및 argument 클래스를 통해 Fragment 간 태입 세이프하게 인수 전달이 가능합니다.
4. 동일한 Activity 내 Fragment 간 데이터 공유에는 **Shared ViewModel**이 유용하고, [안드로이드의 공식문서에서 권장](#)⁴⁰하는 방식 중에 하나입니다. 상황에 따라 활용되는 방법이 다르고, 기능의 요구 사항에 따라서 적절하게 활용하면 가장 효과적인 방법이 될 수도 있습니다.

실전 질문

Q) 동일한 Activity 내에서 Fragment 간 데이터를 주고받을 때 어떤 방법이 효과적인지 설명해 주시고, ViewModel을 활용한다면 Bundle이나 직접적인 Fragment 트랜잭션을 사용하는 것과 비교했을 때 어떤 이점이 있나요?

Pro Tips for Mastery: Fragment Result API

상황에 따라 Fragment에서 다른 Fragment 혹은 Activity 간에 일회성으로 값을 전달해야 합니다. 가령, QR 코드 스캔을 수행하는 Fragment가 스캔된 데이터를 이전의 Fragment로 다시 보내야 하는 상황이 생길 수 있습니다.

⁴⁰<https://developer.android.com/guide/fragments/communicate#fragments>

Fragment⁴¹ 버전 1.3.0 이상에서는 각 FragmentManager가 FragmentResultOwner를 구현하여, Fragment가 서로 직접 참조할 필요 없이 결과 리스너를 통해 통신할 수 있습니다. 이는 데이터 전달을 단순화하면서 느슨한 결합(**loose coupling**)을 유지합니다.

Fragment B(송신자)에서 **Fragment A(수신자)**로 데이터를 전달하기 위해 아래와 같은 방법을 사용할 수 있습니다.

1. **Fragment A(결과를 받는 프래그먼트)**에서 결과 리스너 설정하기.
2. **Fragment B**에서 동일한 requestKey를 사용하여 결과 전달하기.

Fragment A에서 결과 리스너 설정하기

Fragment A는 setFragmentResultListener()를 사용하여 리스너를 등록해야 하며, STARTED 상태가 될 때 결과를 받도록 보장합니다.

그림 50. FragmentA.kt

```

1 class FragmentA : Fragment() {
2
3     override fun onCreate(savedInstanceState: Bundle?) {
4         super.onCreate(savedInstanceState)
5
6         // 데이터를 수신하기 위해 리스너 등록
7         parentFragmentManager.setFragmentResultListener("requestKey", this) {
8             requestKey, bundle ->
9                 val result = bundle.getString("bundleKey")
10                // 수신한 결과 값 처리
11            }
12        }
13    }

```

setFragmentResultListener("requestKey", ...)는 고유한 키 값을 사용하여 리스너를 등록합니다. 이를 통해 여러 개의 리스너가 등록된 경우 리스너끼리 구분이 용이해집니다. 모든 콜백들은 Fragment가 **STARTED** 상태에 들어갈 때 실행됩니다.

⁴¹<https://developer.android.com/jetpack/androidx/releases/fragment>

Fragment B에서 결과 보내기

Fragment B는 setFragmentResult()를 사용하여 결과를 전달할 수 있으며, Fragment A가 활성화될 때 데이터를 받아볼 수 있도록 보장합니다.

그림 51. FragmentB.kt

```
1 class FragmentB : Fragment() {  
2  
3     private lateinit var button: Button  
4  
5     override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
6         super.onViewCreated(view, savedInstanceState)  
7  
8         button = view.findViewById(R.id.button)  
9         button.setOnClickListener {  
10            val result = "result"  
11            // 값을 Fragment A로 전달  
12            parentFragmentManager.setFragmentResult("requestKey", bundleOf("bundleKey"  
13                → to result))  
14        }  
15    }  
}
```

setFragmentResult("requestKey", bundleOf("bundleKey" to result))는 고유한 키 값을 사용하여 FragmentManager에 전달할 값을 저장합니다. **Fragment A가 활성 상태가 아니라면**, Fragment A가 재개되고 리스너를 등록할 때까지 같은 계속 저장되어 있습니다.

프래그먼트 결과의 동작

- **키-리스너의 1:1 관계**: 각 키는 한 번에 **하나의 리스너와 하나의 결과만** 가질 수 있습니다.
- **보류 중인 결과는 덮어쓰여짐**: 리스너가 활성화되기 전에 여러 결과가 설정되면 **최신 결과만** 저장됩니다.
- **결과는 소비 후 삭제됨**: Fragment가 결과를 수신하고 처리하면 결과는 FragmentManager에서 제거됩니다.

- **백 스택의 프래그먼트는 결과를 받지 못함:** Fragment는 결과를 받으려면 백 스택에서 팝(pop)되고 STARTED 상태여야 합니다.
- **STARTED 상태의 리스너는 즉시 트리거됨:** Fragment B가 결과를 설정할 때 Fragment A가 이미 활성 상태이면 리스너는 즉시 실행됩니다.

요약

Fragment Result API는 직접 참조 없이 Fragment 간에 일회성으로 값을 전달할 수 있는 메커니즘을 제공합니다. FragmentManager를 활용하여 결과를 수신하는 Fragment가 활성화될 때까지 결과 값이 안전하게 보존되므로, Fragment 간에 의존성을 분리하고 생명주기를 인지하여 동작하는 통신 메커니즘을 보장합니다. 해당 접근 방식은 QR 코드 스캔, 사용자 입력 디아일로그 또는 양식 제출과 같은 다양한 시나리오에서 (A->B->A 형태의 화면 이동) 유용하며, Fragment 기반 내비게이션을 더 효율적으로 만들어줍니다.

Q) 19. 화면 회전과 같은 구성 변경이 발생하면 Activity에 어떤 변화가 생기나요?

안드로이드에서 구성 변경(가령, 화면 회전, 테마 변경, 글꼴 크기 조정 또는 언어 업데이트)이 발생하면 시스템은 새 구성을 적용하기 위해 현재 Activity를 종료하고 다시 실행하게 됩니다. 이러한 동작은 앱의 리소스가 변경된 구성을 새롭게 반영하고 앱이 다시 로드되도록 보장합니다.

구성 변경 중 기본 동작

1. **Activity 종료 및 재시작:** 구성 변경이 발생하면 Activity가 종료된 다음 다시 재시작됩니다. 이 과정에는 아래의 과정이 포함됩니다.
 - 시스템은 현재 실행 중인 Activity의 onPause(), onStop(), onDestroy() 메서드를 순차적으로 호출합니다.
 - 구성을 변경하면 Activity는 다시 시작되고, onCreate() 메서드가 호출됩니다.
2. **리소스 다시 로드하기:** 시스템은 새 구성에 따라 리소스(가령, 레이아웃, 드로어블 또는 문자열)를 다시 로드하여 앱이 화면 방향, 테마 또는 언어와 같은 변경 사항이 반영될 수 있도록 합니다.

3. 데이터 손실 방지: 개발자는 재생성 중 데이터 손실을 방지하기 위해 `onSaveInstanceState()` 및 `onRestoreInstanceState()` 메서드를 사용하거나 `ViewModel`을 활용하여 인스턴스 상태를 저장하고 복원할 수 있습니다.

그림 52. Handling Configuration Changes with `SavedInstanceState`

```

1 override fun onSaveInstanceState(outState: Bundle) {
2     super.onSaveInstanceState(outState)
3     outState.putString("user_input", editText.text.toString())
4 }
5
6 override fun onCreate(savedInstanceState: Bundle?) {
7     super.onCreate(savedInstanceState)
8     setContentView(R.layout.activity_main)
9
10    val restoredInput = savedInstanceState?.getString("user_input")
11    editText.setText(restoredInput)
12 }
```

재생성을 유발하는 구성 변경

- 화면 회전:** 화면 방향을 세로와 가로 간에 변경하여 새 크기에 맞게 레이아웃을 다시 로드합니다.
- 다크/라이트 테마 변경:** 사용자가 다크 모드와 라이트 모드 간에 전환하면 앱은 테마별 리소스(가령, 색상 및 스타일)를 다시 로드합니다.
- 글꼴 크기 변경:** 기기의 글꼴 크기 설정 조정은 새 배율을 반영하도록 텍스트 리소스를 다시 로드합니다.
- 언어 변경:** 시스템 언어 업데이트는 현지화된 리소스(가령, 다른 언어의 문자열)를 다시 로드합니다.

Activity 재생성 피하기

Activity를 다시 시작하지 않고 구성 변경을 처리하려면 매니페스트 파일에서 `android:configChanges` 속성을 추가하면 됩니다. 이 방식은 변경 사항을 개발자가 수동적으로 처리하는 형태로 책임을 개발자에게 위임합니다.

그림 53. Avoiding Activity Recreation Using android:configChanges

```
1 <activity
2     android:name=".MainActivity"
3     android:configChanges="orientation|screenSize|keyboardHidden" />
```

이렇게 되면 시스템은 Activity를 소멸시키고 다시 생성하지 않습니다. 대신 onConfigurationChanged() 메서드가 호출되어 개발자가 변경 사항을 수동으로 처리할 수 있게 됩니다.

그림 54. Handling Configuration Changes Manually

```
1 override fun onConfigurationChanged(newConfig: Configuration) {
2     super.onConfigurationChanged(newConfig)
3
4     if (newConfig.orientation == Configuration.ORIENTATION_LANDSCAPE) {
5         // 화면이 가로로 변경된 경우 로직 처리
6     } else if (newConfig.orientation == Configuration.ORIENTATION_PORTRAIT) {
7         // 화면이 세로로 변경된 경우 로직 처리
8     }
9 }
```

요약

구성 변경이 발생하면 기본 동작은 Activity를 소멸시키고 다시 생성하여 새 구성에 맞게 리소스를 다시 로드하는 것입니다. 개발자는 onSaveInstanceState()나 ViewModel을 사용하여 일시적으로 UI 상태를 보존할 수 있습니다. 매니페스트 파일에서 android:configChanges 속성을 추가하면, 소멸-재생성 과정을 생략하고 수동적으로 구성 변경에 대한 대응을 할 수 있습니다.

실전 질문

Q) 구성 변경으로 인한 Activity 재생성 중 발생하는 데이터 손실을 어떻게 복원 및 보존할 수 있으며, UI 상태를 어떻게 복구시키는지 설명해 주세요.

Q) 20. ActivityManager란 무엇인가요?

ActivityManager는 기기에서 실행 중인 Activity, 태스크(Task), 프로세스(Process)에 대한 정보를 제공하고 관리하는 안드로이드 시스템 서비스입니다. 이는 안드로이드 프레임워크의 일부로, 개발자가 앱 생명주기, 메모리 사용량 및 태스크 관리 측면과 상호 작용하고 제어할 수 있도록 합니다. 다음은 ActivityManager의 주요 기능입니다.

- 태스크 및 Activity 정보:** ActivityManager는 실행 중인 태스크, Activity 및 해당 스택 상태에 대한 세부 정보를 추적할 수 있습니다. 이는 개발자가 앱 동작 및 시스템 리소스 사용량을 모니터링하는 데 도움이 됩니다.
- 메모리 관리:** 앱의 메모리 소비 및 시스템 전체 메모리 상태를 포함하여 시스템 전체의 메모리 사용량에 대한 정보를 제공합니다. 개발자는 이를 사용하여 앱 성능을 최적화하고 메모리 부족 상태를 처리할 수 있습니다.
- 앱 프로세스 관리:** ActivityManager는 실행 중인 앱 프로세스 및 Service에 대한 세부 정보를 쿼리할 수 있습니다. 개발자는 이 정보를 사용하여 앱 상태를 감지하거나 프로세스 수준의 변화에 응답할 수 있습니다.
- 디버깅 및 진단:** 힙 덤프 생성 또는 앱 프로파일링과 같이 디버깅을 위한 도구를 제공하여 성능 병목 현상이나 메모리 누수를 식별하는 데 도움이 될 수 있습니다.

ActivityManager에서 제공하는 메서드

- getRunningAppProcesses():** 기기에서 현재 실행 중인 프로세스 목록을 반환합니다.
- getMemoryInfo(ActivityManager.MemoryInfo memoryInfo):** 사용 가능한 메모리, 임계 메모리, 기기가 메모리 부족 상태인지 여부 등 시스템에 대한 자세한 메모리 정보를 검색합니다. 이는 메모리 부족 상태 동안 앱 동작을 최적화하는 데 유용합니다.
- killBackgroundProcesses(String packageName):** 시스템 리소스를 확보하기 위해 지정된 앱의 백그라운드 프로세스를 종료합니다. 이는 리소스 집약적인 앱을 테스트하거나 관리하는 데 유용합니다.
- isLowRamDevice():** 기기가 저사양 RAM으로 분류되는지 확인하여 앱이 저메모리 기기에 대한 리소스 사용량을 최적화하는 데 도움을 줍니다.
- appNotResponding(String message):** 테스트 목적으로 ANR(App Not Responding) 이벤트를 시뮬레이션합니다. 디버깅 중에 앱이 ANR 상황에서 어떻게 동작하거나 응답하는지 이해하는 데 사용할 수 있습니다.

- **clearApplicationUserData()**: 파일, 데이터베이스 및 Shared Preferences를 포함하여 애플리케이션과 관련된 모든 사용자별 데이터를 지웁니다. 공장 초기화나 앱을 기본 상태로 재설정하는 경우에 종종 사용됩니다.

사용 예제

아래 코드는 ActivityManager를 사용하여 메모리 정보를 가져오는 방법을 보여줍니다.

그림 55. Retrieve Memory Information

```

1 val activityManager = getSystemService(Context.ACTIVITY_SERVICE) as ActivityManager
2 val memoryInfo = ActivityManager.MemoryInfo()
3 activityManager.getMemoryInfo(memoryInfo)
4
5 Log.d(TAG, "Low memory state: ${memoryInfo.lowMemory}")
6 Log.d(TAG, "Available memory: ${memoryInfo.availMem / (1024 * 1024)} MB")
7 Log.d(TAG, "Threshold memory: ${memoryInfo.threshold / (1024 * 1024)} MB")
8
9 val processes = activityManager.runningAppProcesses
10 if (processes != null && processes.isNotEmpty()) {
11     Log.d(TAG, "Process name: ${processes.first().processName}")
12 }
13
14 // 테스트 목적으로 ANR(App Not Responding) 이벤트를 시뮬레이션합니다. (주의해서 사용)
15 // activityManager.appNotResponding("Pokedex is not responding")
16
17 // 애플리케이션이 디스크에서 할당 중인 데이터를 지우도록 허용 (주의해서 사용)
18 // activityManager.clearApplicationUserData()

```

LeakCanary에서의 ActivityManager 활용 사례

LeakCanary⁴²는 Block이라는 기업에서 관리하는 안드로이드 애플리케이션용 오픈 소스 메모리 누수 감지 라이브러리입니다. 개발 중에 앱의 메모리 누수를 자동으로 모니터링하고

⁴²<https://square.github.io/leakcanary/>

감지하여, 개발자가 메모리 누수를 효율적으로 고치는 데 도움이 될만한 분석을 제공합니다. LeakCanary는 내부적으로 **메모리 상태 및 정보 추적을 위해 ActivityManager를 활용합니다**⁴³.

요약

ActivityManager는 시스템 수준 관리, 성능 튜닝 및 앱 동작 모니터링을 위한 것입니다. 최신 안드로이드에서는 해당 기능들이 부분적으로 더 고도화된 API로 대체되었지만, 안드로이드 애플리케이션에서 리소스 사용을 관리하고 최적화하기 위한 도구로 여전히 활용할 수 있습니다. 적절한 상황에서 효율적으로 사용하면 개발자는 의도하지 않은 시스템 성능 저하를 피하기 위한 목적으로 활용할 수 있습니다.

실전 질문

Q) `ActivityManager.getMemoryInfo()`를 어떻게 앱 성능 최적화에 활용할 수 있으며, 시스템이 메모리 부족 상태에 들어가면 개발자는 어떤 조치를 취해야 하나요?

Q) 21. SparseArray를 사용하면 어떤 이점이 있나요?

SparseArray (`android.util` 패키지)는 `HashMap`과 유사하게 정수 키를 객체 값에 매핑하는 안드로이드에 최적화된 데이터 구조입니다. 정수인 키와 함께 사용하도록 최적화되어 있어 정수 기반 키를 사용할 때 일반적인 Java/Kotlin의 Map이나 `HashMap`보다 메모리 관리 측면에서 효율이 좋고 상황에 따라 성능이 더 좋습니다.

SparseArray의 주요 특징

- 메모리 효율성:** 키-값 매핑을 위해 `HashTable`을 사용하는 `HashMap`과 달리 `SparseArray`은 오토박싱(기본 `int`를 `Integer`로 변환)을 피하고 `Entry` 객체와 같은 추가 데이터 구조에 의존하지 않습니다. 이로 인해 훨씬 적은 메모리를 소비합니다.
- 성능:** 매우 큰 데이터 셋의 경우 `HashMap`만큼 빠르지는 않지만, `SparseArray`은 메모리 최적화 덕분에 중간 크기의 데이터 셋에서 더 나은 성능을 제공합니다.
- Null 키 값 사용 불가:** `SparseArray`은 키 값으로 기본 정수를 사용하므로 키 값에 `null` 사용을 허용하지 않습니다.

⁴³<https://github.com/square/leakcanary/blob/02d0d8b6ebfe8de55c109b904d7b526063f3f852/leakcanary/leakcanary-android-process/src/main/java/leakcanary/LeakCanaryProcess.kt#L75>

SparseArray의 사용법은 간단하며 HashMap과 유사한 형태로 사용할 수 있습니다.

그림 56. Using SparseArray

```
1 import android.util.SparseArray
2
3 val sparseArray = SparseArray<String>()
4 sparseArray.put(1, "One")
5 sparseArray.put(2, "Two")
6
7 // 요소 접근
8 val value = sparseArray[1] // "One"
9
10 // 요소 제거
11 sparseArray.remove(2)
12
13 // 요소 순회
14 for (i in 0 until sparseArray.size()) {
15     val key = sparseArray.keyAt(i)
16     val value = sparseArray.valueAt(i)
17     println("Key: $key, Value: $value")
18 }
```

SparseArray 사용의 이점

- 오토박싱 방지:** HashMap<Integer, Object>에서는 키가 Integer 객체로 저장되어 박싱 및 언박싱 작업으로 인한 오버헤드가 발생합니다. SparseArray는 int 키로 직접 작동하여 메모리와 계산 작업을 절약합니다.
- 메모리 절약:** SparseArray는 키와 값을 저장하기 위해 내부적으로 기본 배열을 사용하여, Entry와 같은 여러 객체를 생성하는 HashMap 구현에 비해 메모리 차지 공간을 줄입니다.
- 컴팩트한 데이터 저장에 효율적:** 적은 수의 키-값 쌍이 있는 밀도가 낮은(sparse) 데이터 셋이나 키가 넓은 정수 범위에 걸쳐 드문드문 분포된 데이터 셋에 적합합니다.

4. **안드로이드 특화**: 제한된 리소스 시나리오를 처리하기 위해 안드로이드에 특화된 구조로 설계되어, 안드로이드 UI 컴포넌트에서 View ID를 객체에 맵핑하는 등의 시나리오에 특히 효과적입니다.

SparseArray의 한계

SparseArray는 메모리 효율적이지만 모든 사용 사례에 항상 적합한 선택은 아닙니다.

1. **성능 트레이드오프**: SparseArray의 요소 접근은 키 조회를 위해 이진 탐색을 (binary search) 사용하기 때문에 매우 큰 데이터 셋의 경우 HashMap보다 느립니다.
2. **정수 키만 사용 가능**: 정수 키로 제한되어 다른 유형의 키가 필요한 사용 사례에는 적합하지 않습니다.

요약

SparseArray는 안드로이드에서 메모리 효율성을 위해 최적화된, 정수 키를 객체 값에 맵핑하는 특수한 자료 구조입니다. 오토박싱을 피하고 메모리 사용량을 줄이는 측면에서 HashMap보다 상당한 이점을 제공하며, 특히 정수 키를 사용하는 데이터 셋에 적합합니다. 메모리 절약을 위해 일부 성능을 절충할 수 있지만, 안드로이드 애플리케이션과 같이 리소스가 제한적인 사례에 유용합니다.

실전 질문

Q) HashMap 대신 SparseArray를 사용하는 것이 어떤 시나리오에서 더 효율적이며, 성능 및 사용성 측면에서 트레이드오프는 무엇인가요?

Q) 22. 런타임 권한(runtime permissions)을 어떻게 처리하나요?

안드로이드에서 런타임 권한을 처리하는 것은 원활한 사용자 경험을 보장하면서 사용자 데이터에 접근하는 데 필수적입니다. 안드로이드 6.0 (API 레벨 23)부터 앱은 설치 시 자동으로 권한을 획득하는 대신 런타임에 위험 권한(dangerous permissions)을 명시적으로 요청해야 합니다. 이 접근 방식은 사용자가 필요할 때만 권한을 부여하도록 허용하여 사용자 개인 정보 보호를 강화합니다.

권한 선언 및 확인

권한을 요청하기 전에 앱은 `AndroidManifest.xml` 파일에 해당 권한을 선언해야 합니다. 런타임 시에는 사용자가 해당 권한이 필요한 기능과 상호 작용할 때만 권한을 요청해야 합니다. 사용자에게 요청하기 전에 `ContextCompat.checkSelfPermission()`을 사용하여 권한이 이미 부여되었는지 확인하는 것이 중요합니다. 권한이 부여된 경우 해당 동작은 계속 진행할 수 있으며, 그렇지 않은 경우 앱은 사용자 권한을 올바르게 요청해야 합니다.

그림 57. CheckPermissions

```
1 when {
2     ContextCompat.checkSelfPermission(this, Manifest.permission.CAMERA)
3         == PackageManager.PERMISSION_GRANTED -> {
4         // 권한 부여됨, 동작 이어서 진행
5     }
6     ActivityCompat.shouldShowRequestPermissionRationale(
7         this, Manifest.permission.CAMERA
8     ) -> {
9         // 사용자가 권한 요청을 거부한 경우
10        showPermissionRationale()
11    }
12 else -> {
13     // 사용자 권한 요청
14     requestPermissionLauncher.launch(Manifest.permission.CAMERA)
15 }
16 }
```

권한 요청하기

권한 요청에 권장되는 방법은 권한 처리를 단순화하는 `ActivityResultLauncher` API를 사용하는 것입니다. 그러면 시스템은 사용자에게 권한 요청을 허용하거나 거부하도록 안내합니다.

그림 58. RequestPermissionLauncher

```
1 val requestPermissionLauncher =  
2     registerForActivityResult(ActivityResultContracts.RequestPermission()) { isGranted  
3         ->  
4             if (isGranted) {  
5                 // 권한 부여됨, 동작 이어서 진행  
6             } else {  
7                 // 권한 거부됨  
8             }  
9 }
```

시스템은 권한 요청을 관리하고, 사용자에게ダイ얼로그를 표시하며, 사용자는 권한을 동의하거나 거절할 수 있습니다.

권한 요청 근거(Rationale) 제공하기

경우에 따라 시스템은 `shouldShowRequestPermissionRationale()`을 사용하여 권한을 요청하기 전에 해당 기능을 사용하기 위해 권한이 필요한 근거(rationale)를 표시할 것을 권장합니다. `true`인 경우 UI는 해당 권한이 필요한 이유를 설명해야 합니다. 이는 사용자 경험을 개선하고 권한 획득 가능성을 높입니다.

`shouldShowRequestPermissionRationale()`은 사용자가 “다시 묻지 않음”을 선택하고 권한을 거부했을 때뿐만 아니라, 앱을 처음 설치하고 아직 권한 요청이 한 번도 이루어지지 않은 초기 상태에서도 `false`를 반환한다는 점을 유의해야 합니다. 즉, 해당 메서드는 기본적으로 `false`를 반환합니다.

그림 59. PermissionRationale

```
1 fun showPermissionRationale() {  
2     AlertDialog.Builder(this)  
3         .setTitle("권한 필요")  
4         .setMessage("이 기능이 제대로 작동하기 위해 카메라 접근 권한이 필요합니다.")  
5         .setPositiveButton("확인") { _, _ ->  
6             requestPermissionLauncher.launch(Manifest.permission.CAMERA)  
7         }  
8         .setNegativeButton("취소", null)  
9         .show()  
10}
```

권한 거부 처리하기

사용자가 권한을 여러 번 거부하면 안드로이드는 이를 영구적인 거부로 받아들여 앱이 해당 권한을 다시는 요청할 수 없게 합니다. 이 경우 앱은 사용자에게 기능이 제한된 이유에 대해 고지하고, 필요한 경우 시스템 설정으로 안내하여 직접 해당 권한을 처리할 수 있도록 해야 합니다. 유의해야 할 점은 권한이 요청이 한 번도 이루어지지 않은 경우에도 `shouldShowRequestPermissionRationale()`는 `false`를 반환하기 때문에 권한 요청이 단 한 번이라도 이루어졌는지 확인이 필요합니다.

그림 60. PermissionDenial

```
1 if (!ActivityCompat.shouldShowRequestPermissionRationale(this,  
2     → Manifest.permission.CAMERA) &&  
3     ContextCompat.checkSelfPermission(this, Manifest.permission.CAMERA) !=  
4     → PackageManager.PERMISSION_GRANTED) {  
5     // 권한 요청이 한 번도 이루어지지 않은 경우는 권한 요청  
6     requestPermissions()  
7  
8     // 권한 요청이 한 번이라도 이루어진 적이 있다면 설정 화면으로 이동  
9     showSettingsDialog()  
10}
```

```
9  
10 fun showSettingsDialog() {  
11     val intent = Intent(Settings.ACTION_APPLICATION_DETAILS_SETTINGS).apply {  
12         data = Uri.parse("package:$packageName")  
13     }  
14     startActivity(intent)  
15 }
```

위치 권한 처리하기

위치 권한은 **포그라운드(foreground)** 및 **백그라운드(background)** 접근으로 분류됩니다. 포그라운드 위치 접근에는 ACCESS_FINE_LOCATION 또는 ACCESS_COARSE_LOCATION이 필요하며, 백그라운드 접근에는 **추가적으로** ACCESS_BACKGROUND_LOCATION 권한이 필요합니다.

```
1 <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />  
2 <uses-permission android:name="android.permission.ACCESS_BACKGROUND_LOCATION" />
```

안드로이드 10 (API 레벨 29)부터 백그라운드 위치를 요청하는 앱은 먼저 포그라운드 접근 권한을 요청한 다음 별도로 백그라운드 권한을 요청해야 합니다.

일회성 권한 (One-Time Permissions)

안드로이드 11 (API 레벨 30)은 위치, 카메라, 마이크에 대해 **일회성 권한**⁴⁴을 도입했습니다. 사용자는 임시적으로 접근 권한을 부여할 수 있으며, 앱을 종료하면 해당 권한은 사라집니다.

요약

런타임 권한을 적절하게 처리하면 보안, 규정 준수 및 사용자 신뢰를 보장할 수 있습니다. 권한 상태 확인, 권한을 요청하는 이유 및 근거 제공, 상황에 맞는 권한 요청, 거부 시 정중한 처리와 같은 모범 사례를 따르면 원활하고 개인 정보를 존중하는 사용자 경험을 만들 수 있습니다.

⁴⁴<https://developer.android.com/training/permissions/requesting#one-time>

실전 질문

Q) 안드로이드의 런타임 권한 시스템은 사용자 개인 정보를 어떻게 보호하며, 민감한 권한을 요청하기 전에 앱은 어떤 시나리오를 고려해야 하나요?

Q) 23. Looper, Handler, HandlerThread의 역할은 무엇인가요?

Looper, Handler, HandlerThread는 스레드를 관리하고 비동기 통신을 처리하기 위해 함께 작동하는 컴포넌트입니다. 이들은 백그라운드 스레드에서 작업을 수행하면서 UI 업데이트를 위해 메인 스레드와 상호 작용하기 위한 필수적인 컴포넌트입니다.

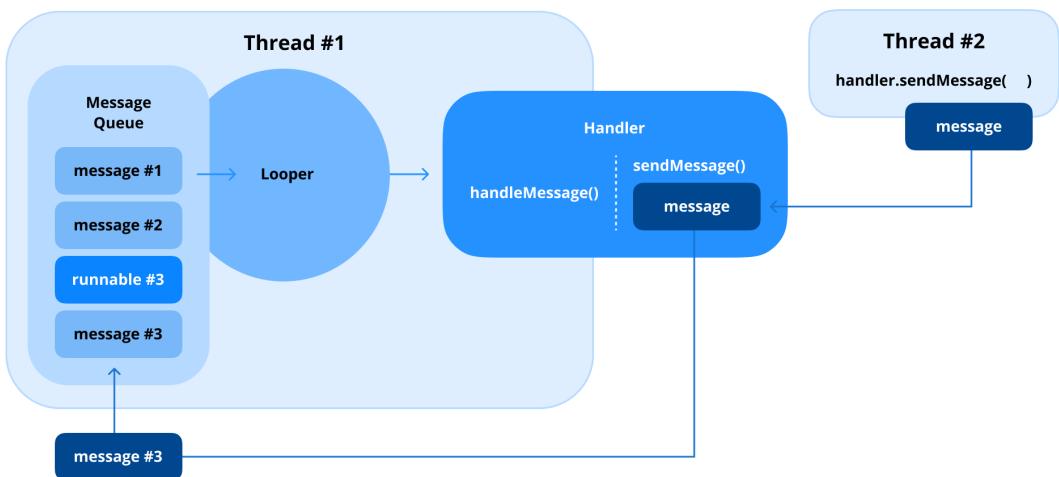


그림 61. looper-handler

Looper

Looper는 스레드를 살아있게 유지하여 메시지 또는 작업 큐를 순차적으로 처리하는 안드로이드 스레딩 모델의 일부입니다. 이는 안드로이드의 메인 스레드(UI 스레드) 및 다른 워커 스레드에서 중심적인 역할을 합니다.

- 목적: 메시지 큐를 지속적으로 모니터링하고 메시지나 작업을 적절한 핸들러에 가져와 디스패치합니다.

- 사용법:** 메시지를 처리하는 모든 스레드에는 Looper가 필요합니다. 메인 스레드에는 자동으로 Looper가 있지만, 워커 스레드의 경우 명시적으로 준비해야 합니다.
- 초기화:** Looper.prepare()를 사용하여 스레드에 Looper를 연결하고 Looper.loop()를 사용하여 루프를 시작합니다.

그림 62. Using Looper in a Thread

```

1 val thread = Thread {
2     Looper.prepare() // 스레드에 Looper 연결
3     val handler = Handler(Looper.myLooper()!!) // Looper를 사용하여 Handler 생성
4     Looper.loop() // 메시지 루프 시작
5 }
6 thread.start()

```

Handler

Handler는 스레드의 메시지 큐 내에서 메시지나 작업을 보내고 처리하는 데 사용됩니다. Looper와 함께 작동합니다.

- 목적:** 한 스레드에서 다른 스레드로 작업이나 메시지를 전달합니다 (가령, 백그라운드 스레드에서 UI 업데이트 하기).
- 동작:** Handler가 생성될 때, 생성된 스레드 및 해당 스레드의 Looper에 연결됩니다. Handler로 전송된 작업은 해당 스레드에서 처리됩니다.

그림 63. Posting Tasks with Handler

```

1 val handler = Handler(Looper.getMainLooper()) // 메인 스레드에서 실행됨
2
3 handler.post {
4     // UI 업데이트 코드
5     textView.text = "Updated from background thread"
6 }

```

HandlerThread

HandlerThread는 내장된 Looper를 가진 특수한 Thread입니다. 작업 또는 메시지 큐를 처리할 수 있는 백그라운드 스레드를 생성하는 과정을 단순화합니다.

- 목적:** 자체 Looper를 가진 워커 스레드를 생성하여 해당 스레드에서 작업을 순차적으로 처리할 수 있도록 합니다.
- 생명주기:** start()로 HandlerThread를 시작한 다음 getLooper()를 사용하여 Looper를 얻습니다. 리소스를 해제하려면 항상 quit() 또는 quitSafely()를 사용하여 Looper를 종료해야 합니다.

그림 64. Using HandlerThread

```

1 val handlerThread = HandlerThread("WorkerThread")
2 handlerThread.start() // 스레드 시작
3
4 val workerHandler = Handler(handlerThread.looper) // 해당 Looper를 사용하여 작업 처리
5
6 workerHandler.post {
7     // 백그라운드 작업 수행
8     Thread.sleep(1000)
9     Log.d("HandlerThread", "Task completed")
10 }
11
12 // 스레드 종료
13 handlerThread.quitSafely() // 처리 중인 메시지 완료 후 안전하게 종료

```

주요 차이점 및 관계

- Looper:** 메시지 처리의 중추이며, 스레드를 살아있게 유지하고 메시지 큐를 처리합니다.
- Handler:** Looper와 상호 작용하여 메시지와 작업을 큐에 넣거나 처리합니다.
- HandlerThread:** 자동 Looper 설정으로 백그라운드 스레드 생성을 단순화합니다.

사용 사례

- **Looper**: 메인 스레드 또는 워커 스레드에서 연속적인 메시지 큐를 관리하는 데 사용됩니다.
- **Handler**: 스레드 간 통신(가령, 백그라운드 스레드에서 UI 업데이트)에 이상적입니다.
- **HandlerThread**: 데이터 처리나 네트워크 요청과 같이 전용 스레드가 필요한 백그라운드 작업에 적합합니다.

요약

Looper, Handler, HandlerThread는 안드로이드에서 스레드와 메시지 큐를 관리하기 위한 기반을 제공합니다. Looper는 스레드가 작업을 지속적으로 처리할 수 있도록 보장하고, Handler는 작업 통신을 위한 인터페이스를 제공하며, HandlerThread는 내장 메시지 루프를 갖춘 워커 스레드를 편리하게 관리하는 방법을 제공합니다.

실전 질문

Q) Handler는 Looper와 어떻게 작동하여 스레드 간 통신을 용이하게 하며, Handler의 일반적인 사용 사례를 말씀해주세요.

Q) HandlerThread란 무엇이며, Looper.prepare()를 사용하여 수동으로 스레드를 생성하는 것과 비교하여 백그라운드 스레드 관리를 어떻게 단순화하나요?

Q) 24. 예외(exceptions)를 어떻게 추적하나요?

안드로이드에서 예외를 추적하는 것은 앱에서 발생하는 문제를 효과적으로 진단하고 해결하는데 중요합니다. 안드로이드는 문제를 식별하고 디버깅하는 데 도움이 되는 여러 도구와 기술을 제공합니다.

Logcat을 이용한 예외 로깅

Android Studio에서 사용 가능한 [Logcat⁴⁵](#)은 로그를 보고 예외를 추적하는 아주 기본적인 도구입니다. 예외가 발생하면 시스템은 예외 유형, 메시지 및 예외가 발생한 코드 줄을 포함하여 자세한 스택 트레이스를 Logcat에 기록합니다. E/AndroidRuntime과 같은 키워드를 사용하여 Logcat 로그를 필터링하여 예외에 집중할 수 있습니다.

⁴⁵<https://developer.android.com/studio/debug/logcat>

try-catch를 이용한 예외 처리

try-catch 블록을 사용하면 예외를 제어된 방식으로 처리하고 코드의 중요한 부분에서 앱 크래시를 방지할 수 있습니다.

그림 65. try-catch.kt

```
1 try {  
2     val result = performRiskyOperation()  
3 } catch (e: Exception) {  
4     Log.e("Error", "Exception occurred: ${e.message}", e) // 스택 트레이스 로깅  
5 }
```

위의 예시에서는 예외가 Logcat에 기록되어 추적 및 해결이 용이해집니다.

전역 예외 핸들러 사용하기

Thread.setDefaultUncaughtExceptionHandler를 사용하여 전역 예외 핸들러를 설정하면 앱 전체에서 처리되지 않은 예외를 포착하는 데 도움이 됩니다. 이는 중앙 집중식 오류 보고 또는 로깅에 특히 유용합니다.

그림 66. Global Exception Handler.kt

```
1 class MyApplication : Application() {  
2     override fun onCreate() {  
3         super.onCreate()  
4         val defaultHandler = Thread.getDefaultUncaughtExceptionHandler()  
5         Thread.setDefaultUncaughtExceptionHandler { thread, exception ->  
6             Log.e("GlobalHandler", "Uncaught exception in thread ${thread.name}:  
7                 ${exception.message}", exception)  
8             // 예외 세부 정보 저장 또는 서드 파티 솔루션으로 전송 (Crashlytics 등)  
9             // FirebaseCrashlytics.getInstance().recordException(exception)  
10            // 기존 핸들러 호출 (선택 사항, 시스템 기본 크래시 동작 유지)  
11            defaultHandler?.uncaughtException(thread, exception)  
12     }
```

```
12     }
13 }
```

이 접근 방식은 애플리케이션 전체의 런타임 문제를 디버깅하고 모니터링하는 데 매우 효과적입니다. 또한 디버그 또는 QA 빌드에서만 전역 예외 핸들러를 구현할 수 있습니다. 이를 통해 QA 전문가가 효율적으로 예외를 추적하고 개발팀에 자세한 보고서를 전달하여 디버깅 및 문제 해결 프로세스를 간소화할 수 있습니다. 구현 사례에 대해 더 자세히 알아보고 싶다면 GitHub의 오픈 소스 프로젝트인 [snitcher](#)⁴⁶를 확인해 볼 수 있습니다.

Firebase Crashlytics 사용하기

[Firebase Crashlytics](#)⁴⁷는 프로덕션 환경에서 예외를 추적하는 훌륭한 도구입니다. 처리되지 않은 예외를 자동으로 기록하고 스택 트레이스, 기기 상태 및 사용자 정보와 함께 자세한 크래시 보고서를 제공합니다. 중요하지 않은 문제에 대해 커스텀 예외를 기록할 수도 있습니다.

그림 67. Firebase Crashlytics.kt

```
1 try {
2     val data = fetchData()
3 } catch (e: IOException) {
4     FirebaseCrashlytics.getInstance().recordException(e)
5 }
```

Crashlytics는 Firebase와 통합되어 크래시 분석 및 해결 추적을 용이하게 합니다.

브레이크포인트(Breakpoints)를 이용한 디버깅

Android Studio에서 브레이크포인트를 찍으면 코드 실행을 일시 중지하고 앱 상태를 점진적으로 검사할 수 있습니다. 이는 개발 중에 예외의 원인을 식별하는 데 특히 유용합니다. 디버그 모드를 활성화하고 브레이크포인트에 도달했을 때 변수, 메서드 호출 및 예외 스택 트레이스를 상세하게 탐색할 수 있습니다.

⁴⁶<https://github.com/skydoves/snitcher>

⁴⁷<https://firebase.google.com/docs/crashlytics>

버그 리포트(Bug Report) 캡처하기

안드로이드에서 버그 리포트를 캡처하면 기기 로그, 스택 트레이스 및 시스템 정보를 수집하여 문제를 진단하고 수정하는 데 도움이 됩니다. 기본적으로 ADB는 서드파티 솔루션 없이 버그 리포트를 캡처하고 생성하는 훌륭한 방법을 제공합니다. 세 단계로 버그 리포트를 생성할 수 있습니다.

1. **개발자 옵션**: 개발자 옵션을 활성화하고, 설정 > 개발자 옵션 > 버그 신고로 이동하여 버그 신고 유형을 선택하고 생성된 보고서를 공유합니다.
2. **Android Emulator에서 버그 신고 캡처**: 확장 컨트롤을 열고 버그 신고를 선택한 다음 관련 세부 정보와 함께 보고서를 저장합니다.
3. **ADB (Android Debug Bridge)를 사용하여 버그 신고 캡처**: 터미널에서 adb bugreport /path/to/save/bugreport를 실행하거나, adb -s <device_serial_number> bugreport로 특정 기기를 지정합니다.

생성된 **ZIP 파일**에는 디버깅에 필수적인 dumpsys, dumpstate, logcat과 같은 로그가 포함되어 있습니다. 버그 리포트는 접근될 때까지 저장되며 성능 및 크래시 진단을 위해 분석에 사용될 수 있습니다. 자세한 내용은 [공식 문서](#)⁴⁸에서 확인할 수 있습니다.

요약

예외 추적은 로컬 도구와 프로덕션 모니터링 등 다양한 방법을 포함합니다. Logcat은 자세한 런타임 로그를 제공하며, try-catch 블록과 전역 예외 핸들러는 예외가 효과적으로 기록되고 관리되도록 보장합니다. Firebase Crashlytics는 프로덕션 환경에서의 크래시 보고 및 디버깅을 위한 강력한 도구입니다. Android Studio의 브레이크포인트는 개발 중 점진적이고 세밀하게 디버깅 경험을 가능하게 합니다. 소개된 방법들을 모두 함께 사용하면 포괄적인 예외 관리가 가능하고, 일반적으로 예외는 크래시로 직결되는 경우가 많기 때문에 예외 추적 및 관리만 잘해도 더 나은 유저 경험을 제공할 수 있습니다.

실전 질문

Q) Logcat을 사용하여 개발 환경에서 예외를 디버깅하는 것과 Firebase Crashlytics와 같은 도구를 사용하여 프로덕션 환경에서 예외를 처리하는 것의 차이점은 무엇인가요? 또한, Logcat과 같은 로컬 환경에서 추적된 예외랑 프로덕션에서 추적된 예외를 각각 어떻게 해결하시나요?

⁴⁸<https://developer.android.com/studio/debug/bug-report>

Q) 25. 빌드 변형(build variants)과 플레이버(flavors)란 무엇인가요?

빌드 변형(build variants)과 플레이버(flavors)⁴⁹는 단일 코드베이스에서 애플리케이션의 다양한 버전을 생성하는 유연한 방법을 제공합니다. 이 시스템을 통해 개발자는 동일한 프로젝트 내에서 개발 및 프로덕션 빌드 또는 무료 및 유료 버전과 같은 여러 구성을 효율적으로 관리할 수 있습니다.

빌드 변형 (Build Variants)

빌드 변형은 특정 빌드 타입(build type)과 제품 플레이버(product flavor)(플레이버가 정의된 경우)를 결합한 결과입니다. 안드로이드 Gradle 플러그인은 각 조합에 대해 빌드 변형을 생성하여 다양한 사용 사례에 맞는 APK 또는 번들을 생성할 수 있도록 합니다.

빌드 타입은 애플리케이션이 어떻게 빌드되는지를 나타내며, 일반적으로 아래 타입을 포함합니다.

- **디버그(Debug)**: 개발 중에 사용되는 빌드 구성입니다. 종종 디버그 도구, 로그 및 테스트용 디버그 툴을 활용하여 개발자가 개발 향상성을 높이도록 합니다.
- **릴리스(Release)**: 배포에 최적화된 구성으로, 종종 리소스 최적화 및 최소화(minification), 난독화(obfuscation)가 적용되고 스토어 게시를 위해 별도의 릴리스 키로 서명되어야 합니다.

기본적으로 모든 안드로이드 프로젝트에는 debug 및 release 빌드 타입이 포함됩니다. 개발자는 특정 요구 사항에 맞게 커스텀 빌드 타입을 추가할 수 있습니다.

제품 플레이버 (Product Flavors)

제품 플레이버를 통해 개발자는 무료 및 유료 버전이나 us 및 eu와 같은 지역별 버전과 같이 앱의 다양한 변형을 정의할 수 있습니다. 각 플레이버는 애플리케이션 ID, 버전 이름 또는 리소스와 같은 고유한 구성을 가질 수 있습니다. 이를 통해 코드를 복제하지 않고 맞춤형 빌드를 쉽게 만들 수 있습니다.

아래는 build.gradle.kts에서 제품 플레이버를 구성하는 예시입니다.

⁴⁹<https://developer.android.com/build/build-variants>

그림 68. Build Variants.kt

```
1 // Kotlin DSL (build.gradle.kts)
2 android {
3     ...
4     flavorDimensions += "version" // 플레이버 차원(dimension) 정의
5
6     productFlavors {
7         create("free") {
8             dimension = "version"
9             applicationIdSuffix = ".free"
10            versionNameSuffix = "-free"
11        }
12
13        create("paid") {
14            dimension = "version"
15            applicationIdSuffix = ".paid"
16            versionNameSuffix = "-paid"
17        }
18    }
19 }
```

이와 같이 설정하면 안드로이드 Gradle 플러그인은 freeDebug, freeRelease, paidDebug, paidRelease 총 4가지의 조합으로 제품 변형에 따른 빌드를 수행할 수 있습니다.

빌드 타입과 플레이버 결합하기

빌드 변형 시스템은 **빌드 타입과 제품 플레이버**를 결합하여 가능한 빌드의 매트릭스를 만듭니다.

- freeDebug: 디버깅용 무료 버전.
- paidRelease: 릴리스에 최적화된 유료 버전.

각 조합은 변형 조건에 따른 설정, 리소스를 가지거나 코드를 다르게 동작시킬 수 있습니다. 가령, 무료 버전에서는 광고를 표시하지만 유료 버전에서는 비활성화할 수 있습니다. 빌드

타입 및 플레이버에 따라 서로 다른 리소스 디렉토리 및 Java/Kotlin 코드로 빌드 함으로써 다르게 동작하는 앱을 빌드할 수 있습니다.

빌드 변형 및 플레이버 사용의 이점

- 효율적인 구성:** 프로젝트를 통째로 복제할 필요가 없으므로, 중복 코드를 줄이고 단일 코드베이스에서 여러 빌드를 처리할 수 있습니다.
- 커스텀 동작:** 유료 버전에서 프리미엄 기능을 활성화하거나 디버그와 릴리스 빌드에서 각각 다른 API를 사용하는 등 앱 동작을 맞춤 설정할 수 있습니다.
- 자동화:** Gradle은 빌드 변형에 따라 APK 서명, 최적화 및 난독화와 같은 작업을 자동화합니다.

요약

안드로이드의 빌드 변형은 빌드 타입과 제품 플레이버를 결합하여 맞춤형 앱을 빌드할 수 있도록 합니다. 빌드 타입은 앱 빌드 방법(e.g., debug vs. release)에 대한 구성을 정의하고, 제품 플레이버는 앱의 변형(e.g., free vs. paid)을 정의합니다. 두 가지 개념을 함께 사용하면 단일 코드베이스로 여러 빌드를 처리하여 앱 배포의 효율성과 확장성 또한 보장합니다.

실전 질문

Q) 빌드 타입과 제품 플레이버의 차이점은 무엇이며, 빌드 변형을 생성하기 위해 그 두 가지가 어떤 식으로 함께 작동하나요?

Q) 26. 접근성(accessibility)을 어떻게 보장하나요?

접근성은 시각, 청각 또는 신체 장애가 있는 사람들을 포함하여 모든 사람이 애플리케이션을 사용할 수 있도록 보장하는 것입니다. 접근성 기능을 구현하면 사용자 경험이 향상되고 WCAG(Web Content Accessibility Guidelines)와 같은 글로벌 접근성 표준 준수를 보장할 수 있습니다.

콘텐츠 설명(Content Descriptions) 활용하기

콘텐츠 설명은 UI 컴포넌트에 텍스트 레이블을 제공하여 TalkBack⁵⁰과 같은 스크린 리더가 시각 장애가 있는 사용자에게 해당 컴포넌트를 알릴 수 있도록 합니다. 버튼, 이미지, 아이콘과 같이 상호 작용하거나 정보를 제공하는 컴포넌트에는 android:contentDescription 속성을 사용합니다. 컴포넌트의 목적이 단순히 앱을 꾸미는 장식용이라서 스크린 리더가 무시해야 하는 경우는 android:contentDescription을 null로 설정하거나 View.IMPORTANT_FOR_ACCESSIBILITY_NO를 사용합니다.

```
1 <ImageView  
2     android:contentDescription="사용자 프로필 사진"  
3     android:src="@drawable/profile_image" />
```

동적 글꼴 크기 지원하기

앱이 기기 설정에서 사용자가 설정한 글꼴 크기 환경 설정을 존중하도록 보장합니다. 접근성 설정에 따라 자동으로 크기가 조정되도록 텍스트 크기에는 sp 단위를 사용합니다.

```
1 <TextView  
2     android:textSize="16sp"  
3     android:text="샘플 텍스트" />
```

포커스 관리 및 탐색

특히 커스텀 뷰, 다이얼로그 및 양식의 경우 포커스 동작을 적절하게 관리합니다. 키보드 및 D-패드 사용자를 위한 논리적인 탐색 경로를 정의하려면 android:nextFocusDown, android:nextFocusUp 및 관련 속성을 사용합니다. 또한 스크린 리더로 앱을 테스트하여 요소 간 포커스 이동이 자연스러운지 확인합니다.

색상 대비 및 시각적 접근성

저시력 또는 색맹 사용자의 가독성을 향상시키기 위해 텍스트와 배경색 간에 충분한 대비를 제공합니다. Android Studio의 Accessibility Scanner와 같은 도구는 앱의 색상 대비를 평가하고 최적화하는 데 도움이 될 수 있습니다.

⁵⁰<https://support.google.com/accessibility/android/answer/6283677>

커스텀 뷰 및 접근성

커스텀 뷰를 만들 때 AccessibilityDelegate를 구현하여 스크린 리더가 커스텀 UI 컴포넌트와 상호 작용하는 방식을 정의할 수 있습니다. 커스텀 컴포넌트에 의미 있는 설명을 제공하려면 onInitializeAccessibilityNodeInfo() 메서드를 재정의합니다.

그림 69. Custom View.kt

```
1 class CustomView(context: Context, attrs: AttributeSet? = null) : View(context, attrs)
2     → {
3         init {
4             importantForAccessibility = IMPORTANT_FOR_ACCESSIBILITY_YES
5                 // 클릭 가능하게 설정 (예시)
6             isClickable = true
7             setAccessibilityDelegate(object : AccessibilityDelegate() {
8                 override fun onInitializeAccessibilityNodeInfo(host: View, info:
9                     AccessibilityNodeInfo) {
10                     super.onInitializeAccessibilityNodeInfo(host, info)
11                     // 역할 및 상태 정보 설정
12                     info.className = Button::class.java.name // 예: 버튼 역할 부여
13                     info.text = "커스텀 구성 요소 설명" // 접근성 텍스트
14                     // info.contentDescription = "추가 설명"
15                     // info.isCheckable = true
16                     // info.isChecked = ...
17                     info.addAction(AccessibilityNodeInfo.AccessibilityAction.ACTION_CLICK)
18                 }
19             })
20         } // 다른 필요한 접근성 콜백 재정의
21     } // ... (뷰 나머지 구현)
```

접근성 테스트하기

Android Studio의 **Accessibility Scanner** 및 **Layout Inspector**와 같은 도구를 사용하여 접근성 문제를 식별하고 수정합니다. 이와 같은 도구는 앱이 보조 기술에 의존하는 사용자에게 앱을 사용 가능하도록 보장하는 데 많은 도움이 됩니다.

요약

안드로이드 애플리케이션의 접근성을 보장하려면 콘텐츠 설명 제공, sp 단위를 사용한 동적 글꼴 크기 지원, 탐색을 위한 포커스 관리, 적절한 색상 대비 보장, 커스텀 뷰에 대한 접근성 지원 추가하기 등이 포함됩니다. 또한, 안드로이드에서 제공하는 접근성 관련 도구를 활용하고 철저히 테스트함으로써 모든 사용자에게 포괄적이고 접근 가능한 애플리케이션을 구축할 수 있습니다. 접근성에 대한 자세한 내용은 아래 공식 문서를 확인하세요.

- 앱 접근성 향상⁵¹
- 앱 접근성 개선 원칙⁵²
- 앱 접근성 테스트⁵³

실전 질문

Q) 동적인 글꼴 사이즈를 지원하기 위한 모범 사례는 무엇이고, 텍스트 크기에 dp 단위보다 sp 단위를 사용하는 것이 선호되는 이유는 무엇인가요?

Q) 개발자는 보조 기술에 의존하는 사용자를 위해 적절한 포커스 관리 및 탐색을 어떻게 보장할 수 있으며, 접근성 문제를 테스트하는 데 도움이 되는 안드로이드 도구에는 어떤 것이 있나요?

Q) 27. 안드로이드 파일 시스템이란 무엇인가요?

안드로이드 파일 시스템은 안드로이드 기기에서 데이터 저장을 관리하고 구성하는 구조화된 환경입니다. 애플리케이션과 사용자가 파일을 효율적으로 저장, 검색 및 관리할 수 있도록 합니다. 파일 시스템은 리눅스의 파일 시스템 아키텍처 위에 구축되어 엄격한 보안 및 권한 모델을 준수하면서 애플리케이션을 위한 비공개 및 공유 저장 공간을 모두 제공합니다.

⁵¹<https://developer.android.com/guide/topics/ui/accessibility/apps>

⁵²<https://developer.android.com/guide/topics/ui/accessibility/principles>

⁵³<https://developer.android.com/guide/topics/ui/accessibility/testing>

안드로이드 파일 시스템의 주요 구성 요소

안드로이드 파일 시스템은 각각 고유한 목적을 가진 다양한 디렉토리와 파티션으로 구성됩니다.

- **System Partition (/system)**: 안드로이드 프레임워크 라이브러리, 시스템 앱 및 구성 파일을 포함한 핵심 운영 체제 파일이 들어 있습니다. 이 파티션은 우발적이거나 악의적인 수정을 방지하기 위해 일반 사용자 및 앱에 대해 읽기 전용입니다.
- **Data Partition (/data)**: 데이터베이스, Shared Preferences 및 사용자가 생성한 파일을 포함한 앱별 데이터가 저장됩니다. 각 앱은 /data/data 내에 해당 앱만 접근할 수 있는 비공개 디렉토리를 가지므로 데이터 보안이 보장됩니다.
- **Cache Partition (/cache)**: 시스템 업데이트나 기기 재시작 시 유지할 필요가 없는 캐시된 파일과 같은 임시 데이터 저장에 사용됩니다.
- **External Storage (/sdcard 또는 /storage)**: 여러 앱에서 접근할 수 있는 공유 저장 공간을 제공하며, 이미지, 비디오, 문서와 같은 미디어 파일에 자주 사용됩니다. 이는 내부 또는 이동식 SD 카드일 수 있습니다.
- **Temporary Files (/tmp)**: 앱 실행 중 임시 파일을 저장하는 위치입니다. 이러한 파일은 일반적으로 앱이나 시스템이 재시작될 때 지워집니다.

안드로이드에서 파일 접근하기

애플리케이션은 안드로이드 프레임워크에서 제공하는 API를 사용하여 파일 시스템과 상호 작용합니다. 필요한 파일 가시성 및 수명에 따라 앱은 파일을 다른 위치에 저장할 수 있습니다.

- **내부 저장소(Internal Storage)**: [애플리케이션 샌드박스](#)⁵⁴ 내의 비공개 저장 공간으로, 해당 앱만 접근할 수 있습니다. 민감하거나 앱별 데이터 저장에 이상적입니다.
- **외부 저장소(External Storage)**: 여러 앱에서 접근할 수 있는 공유 저장 공간으로, 사용자가 앱 외부에서 접근할 것으로 예상하는 사용자 생성 콘텐츠 또는 미디어를 저장하는 데 사용됩니다.

⁵⁴<https://source.android.com/docs/security/app-sandbox>

파일 권한 및 보안

안드로이드 파일 시스템은 엄격한 권한 모델을 적용합니다.

- **비공개 앱 데이터**: 앱의 내부 저장소에 저장된 파일은 비공개이며 해당 앱만 접근할 수 있습니다.
- **공유 파일**: 앱 간에 파일을 공유하려면 개발자는 외부 저장소나 적절한 권한이 있는 ContentProvider를 사용할 수 있습니다.
- **범위 지정 저장소(Scoped Storage)**: 안드로이드 10에서 도입되었으며, 공유 저장소에 대한 직접 접근을 제한하여 앱이 MediaStore 또는 SAF(Storage Access Framework) API를 사용하도록 요구합니다.

요약

안드로이드 파일 시스템은 애플리케이션과 사용자를 위한 데이터 저장을 구성하고 관리하는 강력하고 안전한 환경입니다. 시스템 파일, 앱별 데이터 및 공유 콘텐츠를 위한 전용 공간을 제공하는 동시에 엄격한 보안 및 권한 제어를 준수합니다. 개발자는 다양한 API를 통해 이 시스템과 상호 작용하여 앱의 요구 사항에 맞는 효율적이고 안전한 파일 관리를 가능하게 합니다.

실전 질문

Q) 안드로이드는 파일 시스템에서 보안 및 권한을 어떻게 관리하며, 앱이 서로의 비공개 데이터에 접근할 수 없도록 보장하는 메커니즘은 무엇인가요?

Q) 28. 안드로이드 런타임(ART), Dalvik, Dex 컴파일러란 무엇인가요?

안드로이드 애플리케이션은 기기에서 실행되기 위해 고유한 런타임 환경과 컴파일 프로세스에 의존합니다. **안드로이드 런타임(ART)**, **Dalvik**, **Dex 컴파일러**⁵⁵는 이 프로세스에서 중요한 역할을 하며, 앱이 성능, 메모리 효율성 및 안드로이드 기기와의 호환성을 위해 최적화되도록 보장합니다.

⁵⁵<https://source.android.com/docs/core/runtime>

안드로이드 런타임 (ART)

안드로이드 런타임(ART) 은 안드로이드 4.4 (KitKat)에서 도입되어 안드로이드 5.0 (Lollipop)부터 디폴트로 사용되는 관리형 런타임 환경입니다. 이는 안드로이드 앱 실행을 위한 런타임으로 Dalvik을 대체하며 여러 사항들을 개선하였습니다.

ART는 **Ahead-of-Time (AOT)**⁵⁶ 컴파일을 사용하여 애플리케이션을 컴파일하며, 앱 설치 중에 바이트코드를 기계 코드로 변환합니다. 이는 런타임 시 Just-in-Time (JIT)⁵⁷ 컴파일의 필요성을 없애 앱 시작 시간을 단축하고 실행 중 CPU 사용량을 줄입니다.

ART의 주요 특징은 다음과 같습니다.

- **개선된 성능:** AOT 컴파일은 최적화된 기계 코드를 생성하여 런타임 오버헤드를 줄입니다.
- **가비지 컬렉션:** ART는 더 나은 메모리 관리를 위해 개선된 가비지 컬렉션 기술을 도입했습니다.
- **디버깅 및 프로파일링 지원:** ART는 개발자를 위해 상세한 스택 트레이스 및 메모리 사용량 분석과 같은 향상된 도구를 제공합니다.

Dalvik

Dalvik은 ART 이전에 안드로이드에서 사용된 런타임입니다. 가상 머신 환경에서 애플리케이션을 실행하도록 설계되었으며 제한된 메모리와 처리 능력을 위해 최적화되었습니다.

Dalvik은 **Just-in-Time (JIT)** 컴파일을 사용하여 런타임에 바이트코드를 기계 코드로 변환합니다. 이 접근 방식은 앱 설치에 필요한 시간을 줄이지만, 즉석 컴파일로 인해 런타임 오버헤드가 증가합니다.

Dalvik의 주요 특징은 다음과 같습니다.

- **컴팩트한 바이트코드:** Dalvik은 낮은 메모리 사용량과 빠른 실행을 위해 최적화된 .dex (Dalvik Executable) 파일을 사용합니다.

⁵⁶Ahead-of-Time (AOT) 컴파일은 코드가 런타임 전에 기계 코드로 컴파일되어 실행 중 Just-In-Time (JIT) 컴파일이 필요 없는 프로세스입니다. 해당 접근 방식은 최적화된 사전 컴파일된 바이너리를 생성하여 성능을 향상시키고 런타임 오버헤드를 줄입니다.

⁵⁷Just-In-Time (JIT) 컴파일은 바이트코드가 실행 직전에 동적으로 기계 코드로 변환되는 런타임 프로세스입니다. 이를 통해 런타임 환경은 실제 실행 패턴을 기반으로 코드를 최적화하여 자주 사용되는 코드 경로의 성능을 향상시킬 수 있습니다.

- **레지스터 기반 VM:** Dalvik은 스택 기반(Java Virtual Machine과 같음)이 아닌 레지스터 기반이므로 명령어 효율성이 향상됩니다.

Dalvik의 느린 앱 시작 시간과 높은 CPU 사용량과 같은 한계로 인해 최신 안드로이드 버전에서는 ART로 대체되었습니다.

Dex 컴파일러

Dex 컴파일러는 Java/Kotlin 컴파일러에서 생성된 Java 바이트코드를 .dex (Dalvik Executable) 파일로 변환합니다. 이러한 .dex 파일은 컴팩트하며 Dalvik 및 ART 런타임 환경에 최적화되어 있습니다.

Dex 컴파일러는 안드로이드 애플리케이션이 기기에서 효율적으로 실행되도록 하는 데 중요한 역할을 합니다. Dex 컴파일러의 주요 측면은 다음과 같습니다.

- **멀티dex(Multi-dex) 지원:** 64K 메서드 제한을 초과하는 애플리케이션의 경우 Dex 컴파일러는 바이트코드를 여러 .dex 파일로 분할하는 것을 지원합니다.
- **바이트코드 최적화:** 컴파일러는 안드로이드 기기에서 더 나은 메모리 사용량과 실행 성능을 위해 바이트코드를 최적화합니다.

Dex 컴파일 프로세스는 안드로이드 빌드 시스템에 통합되어 있으며, 앱 개발의 빌드 단계 중에 발생합니다.

Dalvik에서 ART로의 전환

Dalvik에서 ART로의 전환은 안드로이드 런타임 환경을 상당히 개선시켰습니다. ART의 AOT 컴파일, 개선된 가비지 컬렉션 및 프로파일링 기능은 더 나은 개발자 및 사용자 경험을 제공합니다. Dalvik용으로 설계된 앱은 .dex 파일을 사용하기 때문에 ART와 완벽하게 호환되어 개발자에게 원활한 마이그레이션을 보장합니다.

요약

안드로이드 런타임(ART), Dalvik 및 Dex 컴파일러는 안드로이드 앱 실행의 기반을 형성합니다. AOT 컴파일과 향상된 성능을 갖춘 ART는 JIT 컴파일에 의존했던 Dalvik을 대체했습니다. Dex 컴파일러는 Java 바이트코드를 두 런타임 환경 모두에 최적화된 .dex 파일로 변환하여 그 간극을 메웁니다. 이러한 구성 요소들은 안드로이드 기기에서 효율적이고 빠르며 안정적인 앱 실행을 보장합니다.

실전 질문

Q) ART의 Ahead-of-Time (AOT) 컴파일은 Dalvik의 Just-in-Time (JIT) 컴파일과 어떻게 다르며, 앱 시작 시간과 CPU 사용량에 어떤 영향을 미치나요?

Q) 29. APK 파일과 AAB 파일의 차이점은 무엇인가요?

안드로이드 애플리케이션은 두 가지 기본 포맷인 **APK(Android Package)**와 **AAB(Android App Bundle)**⁵⁸를 사용하여 배포 및 설치할 수 있습니다. 두 포맷 모두 안드로이드 앱을 패키징하는 역할을 하지만, 목적, 구조 및 설치 중 리소스 처리 방식에서 차이가 있습니다.

APK (Android Package)

APK 파일은 안드로이드 애플리케이션을 배포하고 설치하는 전통적인 포맷입니다. 이는 앱이 기기에서 작동하는 데 필요한 모든 리소스, 코드 및 메타데이터를 포함하는 완전하고 즉시 설치 가능한 패키지입니다.

APK 파일은 과할 정도로 완전하다고 볼 수 있습니다. 즉, 모든 기기 구성(가령, 화면 밀도, CPU 아키텍처, 언어)에 대한 모든 리소스를 포함합니다. 이로 인해 사용자의 기기와 관련 없는 리소스를 포함하게 되어 파일 크기가 커질 수 있습니다.

APK 파일은 기기에 직접 설치되며 공식 앱 스토어 외부에서 공유하거나 사이드로딩 (sideloading)할 수 있습니다. 그러나 다양한 기기에 대한 구성을 관리하는 것은 개발자의 책임이며, APK에는 특정 기기에 불필요한 리소스가 종종 포함됩니다.

AAB (Android App Bundle)

Google에서 도입한 AAB 형식은 APK와 같은 설치 가능한 형식이 아닌 게시 포맷입니다. 개발자는 AAB를 Google Play에 업로드하면, Google Play는 이를 개별 기기에 맞게 최적화된 APK로 처리합니다.

AAB 파일은 모듈식으로, 다른 구성에 대한 리소스와 코드를 별개의 번들로 분리합니다. Google Play는 이 모듈식 구조를 사용하여 다운로드 시점에 기기별 APK를 생성합니다. 가령, 특정 화면 크기, CPU 아키텍처 및 언어에 필요한 리소스와 코드만 전달하여 사용자 기기의 앱 크기를 줄입니다.

⁵⁸<https://developer.android.com/guide/app-bundle>

AAB 파일은 Google Play의 서버 측에서 처리되므로 직접 설치할 수 없습니다. 따라서, 개발자는 bundletool 유ти리티와 같은 추가 도구 없이 AAB 파일을 사용자에게 직접 배포하여 설치하도록 사이드로딩을 유도할 수 없습니다.

APK와 AAB의 주요 차이점

1. 목적 및 구조

- APK: 모든 구성에 대한 모든 리소스와 코드를 포함하는 완전한 패키지입니다.
- AAB: 기기별 APK를 생성하는 모듈식 게시 형식입니다.

2. 파일 크기

- APK: 모든 기기에 대한 리소스를 포함하여 크기가 더 큽니다.
- AAB: 더 작고 최적화된 APK를 생성할 수 있게 하여 사용자에게 전달되는 앱 크기를 줄입니다.

3. 배포

- APK: 기기에 직접 공유하고 사이드로딩할 수 있습니다.
- AAB: Google Play에 업로드되며, Google Play가 최종 사용자를 위해 최적화된 APK를 생성합니다.

4. 관리

- APK: 개발자가 리소스와 구성을 수동으로 관리해야 합니다.
- AAB: 구성 관리를 Google Play에 위임하여 프로세스를 자동화합니다.

5. 도구 및 호환성

- APK: 모든 안드로이드 기기 및 앱 스토어에서 지원됩니다.
- AAB: 설치 가능한 APK를 생성하기 위해 Google Play 또는 bundletool이 필요하며, 기본적으로 Google 외 앱 스토어와 호환되지 않습니다.

요약

APK 파일은 독립적이고 직접 설치할 수 있는 반면, AAB 파일은 최신 배포를 위해 설계되어 더 작고 기기별 맞춤화 된 APK를 설치 가능하게 합니다. 개발자는 AAB를 통해 앱 크기 감소 및 자동화된 구성 관리의 이점을 얻을 수 있으며, APK는 사이드로딩 및 Google Play 외 배포가 필요하다면 여전히 필수적입니다. 이와 같은 차이점을 통해 개발자는 앱의 배포 전략에 따라 올바른 포맷을 선택할 수 있습니다.

실전 질문

Q) AAB 포맷은 어떻게 다양한 기기 구성에 대해서 앱을 최적화하며, 이는 기존 APK 포맷에 비해 어떤 장점이 있나요?

Q) 30. R8 최적화란 무엇인가요?

R8⁵⁹은 안드로이드 빌드 프로세스에서 APK 또는 AAB의 크기를 줄이고 런타임 성능을 향상시키기 위해 사용되는 코드 축소 및 최적화 도구입니다. 안드로이드 빌드 시스템에 통합되어 이전의 **프로가드 (ProGuard)** 도구를 대체하고, 코드 축소, 최적화, 난독화 및 리소스 관리를 위한 향상된 기능을 제공합니다.

R8 작동 방식

R8은 빌드 단계 중에 애플리케이션 코드를 수정하여 아래의 목적들을 달성합니다.

- **코드 축소(Shrinking)**: 애플리케이션 코드베이스에서 사용되지 않는 클래스, 메서드, 필드 및 속성을 제거하여 최종적인 APK 또는 AAB 크기를 줄입니다.
- **최적화(Optimization)**: 런타임 성능을 향상시키기 위해 코드를 단순화하고 재구성합니다. 여기에는 메서드 인라이닝(inlining), 중복 코드 제거, 동일한 코드 블록 병합 등이 포함됩니다.
- **난독화(Obfuscation)**: 클래스, 메서드 및 필드의 이름을 변경하여 원래 이름을 모호하게 만들어 리버스 엔지니어링을 더 어렵게 만듭니다.
- **리소스 최적화**: 사용되지 않는 리소스(가령, 레이아웃, 드로어블(Drawable), 문자열)를 제거하여 앱 사이즈를 더욱 최소화합니다.

⁵⁹<https://developer.android.com/build/shrink-code#configuration-files>

R8 최적화의 주요 특징

- **죽은 코드 제거(Dead Code Removal)**: R8은 코드베이스를 분석하여 앱에서 도달할 수 없거나 사용되지 않는 코드를 식별하고 제거합니다.
- **인라이닝(Inlining)**: 짧은 메서드나 함수를 호출자쪽에 직접 인라인하여 메서드 호출 오버헤드를 줄이고 런타임 성능을 향상시킵니다.
- **클래스 병합(Class Merging)**: 유사한 클래스나 인터페이스를 하나로 결합하여 메모리 공간을 줄이고 효율성을 향상시킵니다.
- **도달 불가능한 코드 제거(Unreachable Code Elimination)**: 실행되지 않는 코드 경로를 완전히 제거합니다.
- **상수 폴딩 및 전파(Constant Folding and Propagation)**: 표현식을 단순화하고 변수를 가능한 경우 해당 상수 값으로 대체합니다.
- **난독화(Obfuscation)**: R8은 코드의 의미 있는 이름을 더 짧고 덜 설명적인 이름으로 대체하여 앱 크기를 줄이고 리버스 엔지니어링을 더 어렵게 만듭니다.

R8 구성

R8은 앱 빌드 구성을 위해 **ProGuard 규칙**을 동일하게 사용합니다. 코드의 어떤 부분을 축소, 난독화 또는 최적화에서 제외할지 지정할 수 있습니다. 이에 대한 사용 사례는 다음과 같습니다.

- **리플렉션을 위한 코드 보존**: 리플렉션을 통해 접근하는 클래스나 메서드는 런타임에 해당 패키지 이름을 알아야하기 때문에 ProGuard 규칙에 명시적으로 유지(keep)하여 난독화 하지 않도록 합니다.
- **서드파티 라이브러리 제외**: 일부 라이브러리는 내부적으로 리플렉션을 사용할 수도 있고, 정상적인 기능 동작을 위해 특정 규칙이 필요할 수 있습니다.

아래는 클래스를 난독화 하지 않고 보존하기 위한 ProGuard 규칙입니다.

```
1 -keep class com.example.myapp.MyClass { *; }
```

R8의 장점

- **긴밀한 통합**: R8은 안드로이드 빌드 시스템에 내장되어 있어 일반적인 ProGuard 규칙 외에 추가 설정이 필요하지 않습니다.
- **향상된 효율성**: 축소, 최적화 및 난독화를 단일 패스로 결합하여 ProGuard보다 빠르고 효율적입니다.
- **앱 크기 감소**: 사용되지 않는 코드와 리소스를 제거하여 최종 APK 또는 AAB 크기를 크게 줄입니다.
- **향상된 보안**: 난독화는 해커 및 크래커가 앱을 리버스 엔지니어링하기 어렵게 만들어 보안을 강화합니다.

R8의 한계

- **과도한 축소 위험**: 제대로 구성하지 않으면 R8이 간접적으로 참조되는 코드나 리소스를 제거하여 런타임 오류를 유발할 수 있습니다.
- **복잡한 구성**: 복잡한 프로젝트, 특히 리플렉션이나 동적 클래스 로딩을 사용하는 프로젝트의 경우 ProGuard 규칙 작성이 복잡하고 까다로울 수 있습니다.
- **디버깅 어려움**: 난독화는 스택 트레이스에 난독화된 이름이 로깅되기 때문에 디버깅을 더 어렵게 만들 수 있습니다.

요약

R8은 최신 안드로이드 개발의 필수 도구로, 포괄적인 코드 사이즈 감량, 최적화 및 난독화 기능을 제공합니다. 앱 크기를 줄이고 런타임 성능을 향상시키며 보안을 강화함으로써 R8은 개발자가 효율적이고 컴팩트한 애플리케이션을 빌드하는 데 도움을 줍니다. 필요한 코드가 의도치 않게 제거되는 것을 방지하고 원활한 앱 동작을 보장하려면 ProGuard 규칙을 사용한 바른 구성이 중요합니다. R8에 대한 심화학습은 Jake Wharton의 글 [R8 Optimization: Staticization](#)⁶⁰을 읽어보실 수 있습니다.

실전 질문

- Q) R8 최적화는 앱 성능을 어떻게 개선하고, APK/AAB 용량을 어떻게 줄이나요?
Q) R8은 ProGuard와 어떻게 다르며, 어떤 추가적인 장점을 제공하나요?

⁶⁰<https://jakewharton.com/r8-optimization-staticization/>

Q) 31. 애플리케이션 용량을 어떻게 줄이나요?

안드로이드 애플리케이션 용량을 최적화하는 것은 특히 저장 공간이 제한적이거나 인터넷 연결이 느린 사용자의 사용자 경험을 개선하는 데 필수적입니다. 애플리케이션의 기능과 동작을 손상시키지 않으면서 용량을 줄이기 위해 여러 전략을 사용할 수 있습니다.

사용하지 않는 리소스 제거하기

이미지, 레이아웃 또는 문자열과 같이 사용되지 않는 리소스는 불필요하게 APK 또는 AAB 크기를 증가시킵니다. Android Studio의 **Lint**와 같은 도구는 이러한 리소스를 식별하는데 도움이 될 수 있습니다. 사용하지 않는 리소스를 제거한 후, `build.gradle` 파일에서 `shrinkResources`를 활성화하여 빌드 프로세스 중에 사용되지 않는 리소스를 자동으로 제거하도록 합니다.

```
1 android {  
2     buildTypes {  
3         release {  
4             minifyEnabled true  
5             shrinkResources true  
6         }  
7     }  
8 }
```

R8로 코드 축소 활성화하기

안드로이드의 기본 코드 축소기 및 최적화 도구인 R8은 사용되지 않는 클래스와 메서드를 제거하며, 코드를 난독화하여 더 컴팩트하게 만듭니다. ProGuard를 적절하게 사용하면 중요한 코드나 리플렉션 기반 라이브러리는 난독화를 생략하고 오동작하지 않도록 보장합니다.

```
1 android {  
2     buildTypes {  
3         release {  
4             minifyEnabled true  
5                 proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),  
6                 → 'proguard-rules.pro'  
7         }  
8     }  
9 }
```

리소스 최적화 사용하기

이미지 및 XML 파일과 같은 리소스를 최적화하면 앱 용량을 크게 줄일 수 있습니다.

- **벡터 드로어블(Vector Drawables)**: 확장 가능한 그래픽을 위해 래스터 이미지(가령, PNG, JPEG) 대신에 용량을 덜 차지하는 벡터 드로어블로 대체합니다.
- **이미지 압축**: TinyPNG 또는 ImageMagick과 같은 도구를 사용하여 눈에 띠는 품질 손상 없이 래스터 이미지를 압축합니다.
- **WebP 형식**: 이미지를 PNG 또는 JPEG보다 압축률이 좋은 WebP 형식으로 변환합니다.

```
1 android {  
2     defaultConfig {  
3         vectorDrawables.useSupportLibrary = true  
4     }  
5 }
```

Android App Bundles (AAB) 사용하기

Android App Bundle (AAB) 형식으로 전환하면 Google Play가 개별 기기에 맞는 최적화된 APK를 제공할 수 있습니다. 이는 특정 구성(가령, 화면 밀도, CPU 아키텍처 또는 언어)에 필요한 리소스와 코드만 포함하여 앱 용량을 줄입니다.

```
1 android {  
2     bundle {  
3         density {  
4             enableSplit = true  
5         }  
6         abi {  
7             enableSplit = true  
8         }  
9         language {  
10            enableSplit = true  
11        }  
12    }  
13 }
```

불필요한 의존성 제거하기

프로젝트의 의존성을 검토하고 사용되지 않거나 중복되는 라이브러리를 제거합니다. Android Studio의 [Gradle Dependency Analyzer](#)⁶¹를 사용하여 무거운 라이브러리 및 전이 의존성(transitive dependencies)을 식별할 수 있습니다.

네이티브 라이브러리 최적화하기

앱에 네이티브 라이브러리가 포함된 경우 다음 전략을 사용하여 앱 용량을 줄일 수 있습니다.

- **사용하지 않는 아키텍처 제외**: build.gradle 파일의 abiFilters 옵션을 사용하여 필요한 ABI만 포함합니다.
- **디버그 심볼 제거**: stripDebugSymbols를 사용하여 네이티브 라이브러리에서 디버깅 심볼을 제거합니다.

⁶¹https://www.jetbrains.com/help/idea/work-with-gradle-dependency-diagram.html#dependency_analyzer

```

1 android {
2     defaultConfig {
3         ndk {
4             abiFilters "armeabi-v7a", "arm64-v8a" // 필요한 ABI만 포함
5         }
6     }
7     packagingOptions {
8         exclude "**/lib/**/*.so.debug"
9     }
10 }

```

Proguard 규칙을 구성하여 디버그 정보 줄이기

디버깅 메타데이터는 최종 APK 또는 AAB에 불필요한 무게를 더합니다. proguard-rules.pro 파일을 구성하여 이러한 정보를 제거합니다.

```

1 -dontwarn com.example.unusedlibrary.**
2 -keep class com.example.important.** { *; }

```

동적 기능(Dynamic Features) 사용하기

동적 기능 모듈을 사용하면 자주 사용되지 않는 기능을 주문형 모듈(해당 기능이 필요한 경우 설치)으로 분리하여 앱을 모듈화할 수 있습니다. 이는 초기 다운로드 용량을 줄이는 데 도움이 됩니다.

```

1 // app/build.gradle (Groovy DSL)
2 android {
3     dynamicFeatures = [":feature1", ":feature2"]
4 }

```

앱 내 대용량 애셋 피하기

- 비디오나 고해상도 이미지와 같은 대용량 애셋은 콘텐츠 전송 네트워크(CDN)에 호스팅하고 런타임에 동적으로 로드합니다.
- 미디어 콘텐츠는 앱과 함께 번들링하는 대신 스트리밍을 사용합니다.

요약

안드로이드 애플리케이션 크기를 줄이는 데는 사용되지 않는 리소스 제거, 코드 축소를 위한 R8 활성화, 리소스 최적화, App Bundle과 같은 최신 형식 활용 등 다양한 전략이 있습니다. 또한 의존성 검토, 네이티브 라이브러리 최적화, 기능 모듈화는 앱 용량을 최소화하는데 도움을 줍니다. 이러한 전략들을 통해 가볍고 성능 좋은 애플리케이션을 빌드하여 우수한 사용자 경험을 제공할 수 있습니다.

실전 질문

Q) 앱에 APK/AAB 크기를 크게 증가시키는 고해상도 이미지가 포함되어 있습니다. 시각적 품질을 유지하면서 이미지 리소스를 어떻게 최적화하고, 최대 효율성을 위해 어떤 이미지 포맷을 사용할 수 있을까요?

Q) 애플리케이션은 보통 여러 기능들이 포함되어 있지만 그중 일부는 사용자가 자주 사용하지 않습니다. 해당 기능을 필요할 때부터 사용할 수 있도록 하여 초기 앱 용량을 줄이는 방법에는 무엇이 있을까요?

Q) 32. 안드로이드 애플리케이션의 프로세스(process)란 무엇이며, 안드로이드 운영 체제는 이를 어떻게 관리하나요?

안드로이드에서 **프로세스(process)** 는 애플리케이션이 실행되는 환경입니다. 각 안드로이드 앱은 다른 앱과 격리된 자체 프로세스에서 단일 실행 스레드로 작동하여 시스템 보안, 메모리 관리 및 내결함성(fault tolerance)을 보장합니다. 안드로이드 프로세스는 리눅스 커널을 사용하여 운영 체제에 의해 관리되며 엄격한 생명주기 규칙을 따릅니다. 기본적으로 동일한 애플리케이션의 모든 컴포넌트는 메인 스레드라고 하는 동일한 프로세스 및 스레드에서 실행됩니다.

안드로이드에서 프로세스 작동 방식

안드로이드 애플리케이션이 시작되면 운영 체제는 리눅스 `fork()` 시스템 함수를 호출하여 해당 앱을 위한 새 프로세스를 생성합니다. 각 프로세스는 Dalvik 또는 ART(안드로이드 런타임) 가상 머신의 고유 인스턴스에서 실행되어 안전하고 독립적인 실행을 보장합니다. 안드로이드는 각 프로세스에 고유한 리눅스 사용자 ID(UID)를 할당하여 권한 제어 및 파일 시스템 경리를 포함한 엄격한 보안 경계를 적용합니다.

애플리케이션 컴포넌트와 프로세스 연결

기본적으로 안드로이드 애플리케이션의 모든 컴포넌트는 동일한 프로세스 내에서 실행되며, 대부분의 애플리케이션은 이 표준을 따릅니다. 하지만 개발자는 `AndroidManifest.xml` 파일의 `android:process` 속성을 사용하여 프로세스 할당을 커스텀할 수 있습니다. 해당 속성은 `<activity>`, `<service>`, `<receiver>`, `<provider>`와 같은 컴포넌트에 적용될 수 있어 컴포넌트가 별도의 프로세스에서 실행되거나 선택적으로 프로세스를 공유할 수 있도록 합니다. `<application>` 요소도 이 속성을 지원하여 모든 컴포넌트의 기본 프로세스를 정의합니다.

그림 70. `ProcessConfigurationExample.xml`

```
1 <service  
2     android:name=".MyService"  
3     android:process=":remote" />
```

이 예제에서 `MyService` 서비스는 `:remote`라는 별도의 프로세스에서 실행되어 독립적인 작동과 향상된 내결함성을 가능하게 합니다.

또한, 다른 애플리케이션의 컴포넌트가 동일한 리눅스 사용자 ID를 가지고 동일한 인증서로 서명된 경우 동일한 프로세스를 공유할 수 있습니다. 안드로이드는 시스템 리소스 요구에 따라 프로세스를 동적으로 관리하며, 필요할 때 우선순위가 낮은 프로세스를 종료합니다. 더 이상 보이지 않는 `Activity`를 호스팅하는 프로세스는 보이는 컴포넌트를 호스팅하는 프로세스보다 종료될 가능성이 더 높습니다. 안드로이드 시스템은 연관된 컴포넌트가 작업을 수행해야 할 때 프로세스를 다시 시작하여 최적의 시스템 성능과 사용자 경험을 보장합니다.

프로세스와 앱 생명주기

안드로이드는 시스템 메모리와 앱의 현재 상태를 기반으로 [프로세스 및 앱 생명주기](#)⁶²를 관리하며, 엄격한 우선순위 계층을 따릅니다.

- 포그라운드 프로세스(Foreground Process)**: 사용자와 활발하게 상호 작용하며 실행 중인 프로세스입니다. 가장 높은 우선순위의 프로세스이며 거의 종료되지 않습니다.
- 보이는 프로세스(Visible Process)**: 사용자에게 보이지만 활발하게 상호 작용하지 않는 프로세스입니다(가령, 다이얼로그 뒤의 `Activity`).

⁶²<https://developer.android.com/guide/components/activities/process-lifecycle>

3. **서비스 프로세스(Service Process)**: 데이터 동기화나 음악 재생과 같은 작업을 수행하는 백그라운드 Service를 실행하는 프로세스입니다.
4. **캐시된 프로세스(Cached Process)**: 더 빠른 재실행을 위해 메모리에 유지되는 유형 프로세스입니다. 캐시된 프로세스는 우선순위가 가장 낮으며 메모리가 부족할 때 가장 먼저 종료됩니다.

안드로이드 시스템은 메모리를 확보하고 시스템 안정성을 유지하기 위해 우선순위가 낮은 프로세스를 자동으로 종료합니다.

보안 및 권한

각 안드로이드 프로세스는 리눅스 보안 모델을 사용하여 샌드박스 처리되어 엄격한 권한 기반 접근 제어를 시행합니다. 이러한 격리는 안드로이드 권한 시스템을 통해 명시적으로 권한이 부여되지 않는 한 애플리케이션이 다른 프로세스의 데이터에 접근할 수 없도록 보장합니다. 이 보안 모델은 안드로이드의 멀티태스킹 환경의 기본이며 시스템 안정성과 데이터 개인 정보 보호를 모두 지원합니다.

요약

안드로이드의 프로세스는 애플리케이션 컴포넌트를 위한 실행 환경 역할을 하여 격리되고 안전하며 효율적인 앱 작동을 보장합니다. 안드로이드 시스템에 의해 관리되는 프로세스는 메모리 제약 조건, 사용자 활동 및 애플리케이션 우선순위에 따라 생성, 예약 및 종료됩니다. 개발자는 매니페스트 파일의 구성과 안드로이드 권한 관리 시스템을 통해 프로세스 동작을 추가로 제어하여 견고하고 확장 가능한 애플리케이션 개발을 가능하게 할 수 있습니다.

실전 질문

- Q) 서로 다른 안드로이드 컴포넌트들을 별도의 프로세스에서 실행해야 하는 애플리케이션을 개발 중이라고 해봅시다. `AndroidManifest`에서 이를 어떻게 구성하며, 여러 프로세스를 사용할 때의 잠재적인 단점은 무엇인가요?
- Q) 안드로이드는 메모리가 부족할 때 어떤 프로세스를 종료할지 결정하기 위해 우선순위 기반 프로세스 관리 시스템을 사용합니다. 시스템이 프로세스 우선순위를 어떻게 정하는지, 그리고 중요한 프로세스가 종료되는 것을 방지하기 위해 개발자가 따라야 할 전략은 무엇인지 설명해 주세요.

💡 Pro Tips for Mastery: Activities, Services, Broadcast Receivers, Content Providers가 안드로이드의 4대 주요 컴포넌트라고 불리는 이유는 무엇인가요?

Activity, Service, BroadcastReceiver, ContentProvider는 안드로이드 애플리케이션이 시스템 및 다른 애플리케이션과 상호 작용할 수 있도록 하는 필수 구성 요소이기 때문에 안드로이드의 4대 주요 컴포넌트로 불립니다. 이러한 컴포넌트는 앱의 생명주기를 관리하고, 동작을 정의하며, 프로세스 간 통신을 가능하게 하여 안드로이드의 프로세스 및 애플리케이션 생명주기 모델과 밀접하게 연결됩니다. 하지만 해당 컴포넌트들이 4대 주요 컴포넌트로 불리는데는 프로세스와 밀접한 관련이 있기 때문입니다.

각 컴포넌트가 안드로이드 프로세스와 어떤 관련이 있는가

- Activities:** Activity는 사용자 인터페이스가 있는 단일 화면을 나타냅니다. 사용자 상호 작용의 진입점이며 안드로이드 프로세스 생명주기와 밀접하게 연결되어 있습니다. 사용자가 앱을 열면 시스템은 앱의 프로세스에서 Activity를 시작합니다. 프로세스가 종료되면 Activity는 소멸되고, 앱을 다시 시작하면 새 프로세스가 생성됩니다.
- Services:** Service는 사용자 인터페이스 없이 백그라운드 작업을 수행합니다. 애플리케이션이 보이지 않을 때도 실행될 수 있어 음악 재생이나 파일 다운로드와 같은 작업을 허용합니다. Service는 앱의 매니페스트에 지정된 `android:process` 속성에 따라 앱과 동일한 프로세스 또는 별도의 프로세스에서 실행될 수 있습니다.
- Broadcast Receivers:** BroadcastReceiver를 사용하면 애플리케이션이 네트워크 변경이나 배터리 상태 업데이트와 같은 시스템 전체 브로드캐스트 메시지를 수신하고 응답할 수 있습니다. 앱이 실행 중이 아니더라도 트리거되어 필요한 경우 안드로이드 시스템이 해당 프로세스를 시작하도록 합니다.
- Content Providers:** ContentProvider는 공유 애플리케이션 데이터를 관리하여 앱이 중앙 집중식 데이터베이스에서 읽거나 쓸 수 있도록 합니다. 이는 프로세스 간 통신을 허용하므로 다른 애플리케이션 간에 데이터를 공유하는 데 사용될 수 있으며, 안드로이드 시스템이 프로세스를 안전하고 효율적으로 관리하도록 요구합니다.

안드로이드 프로세스와의 연결

이러한 컴포넌트는 안드로이드 시스템이 앱 사용량, 메모리 가용성 및 작업 우선순위에 따라 프로세스를 관리하기 때문에 안드로이드 프로세스와 직접적으로 연결됩니다. 컴포넌트가 트리거되면(가령, Activity 열기 또는 브로드캐스트 수신), 안드로이드 시스템은 아직 실행

중이 아닌 앱의 프로세스를 새롭게 시작할 수 있습니다. 각 컴포넌트는 매니페스트 파일의 `android:process` 속성을 사용하여 자체 프로세스를 할당받을 수도 있어 리소스 집약적인 작업에 더 많은 유연성을 제공합니다.

이는 네 가지 주요 안드로이드 컴포넌트—**Activity**, **Service**, **BroadcastReceiver**, **ContentProvider**—각각이 안드로이드 OS에서 자체 전용 프로세스를 가질 수 있음을 의미합니다. 이 네 가지 컴포넌트는 전용 프로세스에서 실행되도록 구성될 수 있으므로 시스템 수준 기능을 얻게 되어 다른 안드로이드 컴포넌트에 비해 더 강력하고 독립적입니다. 이 설계는 백그라운드 실행, IPC 및 시스템 수준 상호 작용을 가능하게 하여 안드로이드 앱이 복잡한 다중 프로세스 작업을 효율적으로 처리할 수 있도록 보장합니다.

 **면접관을 위한 팁:** 각각의 컴포넌트가 안드로이드의 4대 주요 컴포넌트라고 많은 자료에서 언급하고 있지만, 일반적으로 개발자들이 그 핵심적인 이유 보다는 암기를 통해 알고 있기 때문에 단순히 “안드로이드의 4대 컴포넌트가 무엇인가요?”라는 질문 보다 지원자의 이해도를 심층적으로 평가할 수 있는 좋은 예시입니다.

요약

`Activity`, `Service`, `BroadcastReceiver`, `ContentProvider`는 필수적인 애플리케이션 기능, 사용자 상호 작용 및 앱 간 통신을 가능하게 하기 때문에 안드로이드의 4대 주요 컴포넌트라고 불립니다. 안드로이드 프로세스 모델과의 긴밀한 관계를 통해 효율적인 프로세스 관리, 최적의 리소스 활용 및 시스템 수준 작업 조정을 보장하여 안드로이드 앱 개발의 근간을 형성합니다.

카테고리 1: 안드로이드 UI - 뷰 (Views)

안드로이드 UI는 화면, 레이아웃, 위젯 및 애플리케이션의 구조와 상호 작용을 형성하는 다양한 컴포넌트를 디자인하기 위한 안드로이드 개발의 핵심입니다. 개발의 다른 측면도 중요하지만, UI 시스템은 사용자가 앱에 대한 첫인상을 결정하게 되고 의미 있는 사용자 상호 작용을 촉진하므로 중요한 역할을 합니다. 시각적으로 매력적이고 반응성이 뛰어난 애플리케이션을 만들기 위해서는 안드로이드 UI에 대한 깊은 이해가 필수적입니다.

최근에는 [Jetpack Compose](#)⁶³ 생태계가 빠르게 성장하여 UI를 구축하는 데 프로덕션 수준에서 표준처럼 채택되고 있습니다. Jetpack Compose가 안드로이드 UI 개발의 미래를 대표한다고 해도 과언이 아닙니다. 안드로이드를 처음 시작하는 새로운 개발자라면 기존 View 시스템을 먼저 배울 필요 없이 [Chapter 1: Jetpack Compose Interview Questions](#)으로 바로 넘어가셔도 됩니다.

하지만, 일부 대기업에서는 Jetpack Compose로 마이그레이션하는 것이 결코 쉽지 않을 수 있고 단기 전략과 맞지 않기 때문에 여전히 많은 회사들이 안드로이드 View 시스템에 더러 의존하고 있습니다. 이러한 회사의 기술 면접을 준비하는 경우 기존 View 시스템에 대한 탄탄한 이해가 여전히 필수적일 수 있습니다.

안드로이드 View에서는 모든 UI 요소가 기본적으로 메인 스레드에서 실행되므로 고성능 애플리케이션을 구축하려면 View 생명주기와 일반적으로 사용되는 UI 컴포넌트에 대한 확실한 이해가 필수적입니다. 또한 Window나 텍스트 단위와 같은 안드로이드 UI 시스템의 핵심 개념을 이해하면 개발자가 애플리케이션을 섬세하고 올바르게 구축하기 위한 결정을 스스로 내리는 데 많은 도움이 됩니다.

많은 경우 디자인 팀의 복잡한 디자인 스펙을 충족하기 위해 커스텀 뷰를 만들어야 합니다. 따라서 안드로이드 UI 시스템을 깊이 이해하는 것은 성능에 있어서 효율적으로 개발하고 안드로이드 프레임워크를 학습하는 것에 이어 한 단계 더 좋은 안드로이드 개발자가 되기 위한 필수적인 단계입니다.

Q) 33. View 생명주기를 설명해주세요

안드로이드에서 **View 생명주기(Lifecycle)**는 View(가령, TextView 또는 Button 등)가 생성되고, Activity나 Fragment에 연결되고, 화면에 표시되고, 최종적으로 소멸되거나 분리되는

⁶³<https://developer.android.com/compose>

동안 거치는 생명주기 이벤트를 나타냅니다. View 생명주기를 이해하면 개발자가 View의 초기화, 렌더링 및 소멸을 관리하고, 사용자 작업 또는 시스템 이벤트에 응답하여 View 생명주기에 따라 커스텀 뷰를 구현하며, 적절한 시점에 리소스를 폐기하는 데 도움이 됩니다.

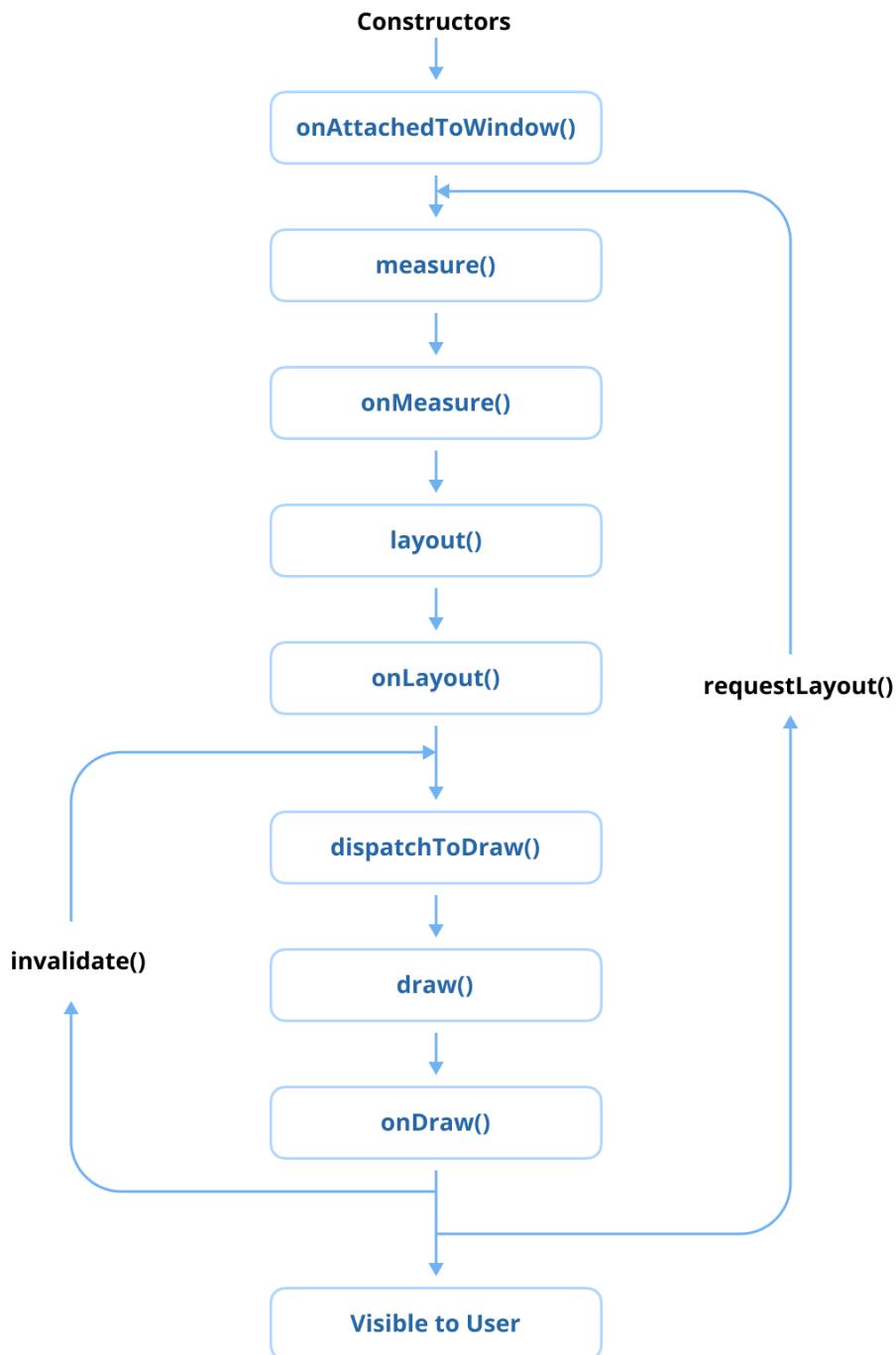


그림 71. view-lifecycle

1. **View 생성 (onAttachedToWindow)**: View가 하드 코딩 방식으로 인스턴스화되거나 XML 레이아웃에서 인플레이션되는 단계입니다. 리스너 설정 및 데이터 바인딩과 같은 초기 설정 작업이 여기서 수행됩니다. onAttachedToWindow() 메서드는 View가 부모 뷰에 추가되고 화면 렌더링을 할 준비를 마쳤을 때 트리거됩니다.
2. **Layout 단계 (onMeasure, onLayout)**: 이 단계에서는 View의 크기와 위치를 측정합니다. onMeasure() 메서드는 레이아웃 매개변수와 부모 제약 조건에 따라 View의 너비와 높이를 결정합니다. 측정된 후 onLayout() 메서드는 View를 부모 내에 배치하여 화면에 표시될 위치를 최종 결정합니다.
3. **Drawing 단계 (onDraw)**: 크기와 위치가 최종 결정된 후 onDraw() 메서드는 텍스트나 이미지와 같은 View의 내용을 [Canvas](#)⁶⁴에 렌더링 합니다. 커스텀 View는 해당 메서드를 재정의하여 커스텀 드로잉 로직을 구현할 수 있습니다.
4. **Event 처리 (onTouchEvent, onClick)**: 상호 작용하는 View는 이 단계에서 터치 이벤트, 클릭, 제스처와 같은 사용자 상호 작용을 처리합니다. onTouchEvent() 및 onClick()과 같은 메서드는 이러한 이벤트를 처리하고 사용자 입력에 대한 View의 응답을 정의하는 데 사용됩니다.
5. **View 분리 (onDetachedFromWindow)**: View가 화면과 부모 ViewGroup에서 제거될 때(가령, Activity 또는 Fragment가 소멸 중일 때), onDetachedFromWindow() 메서드가 호출됩니다. 이 단계는 리소스를 정리하거나 리스너를 분리하는 데 이상적입니다.
6. **View 소멸**: View가 더 이상 사용되지 않으면 가비지 컬렉션됩니다. 개발자는 메모리 누수를 방지하고 성능을 최적화하기 위해 이벤트 리스너나 백그라운드 작업과 같은 모든 리소스가 적절하게 해제되었는지 확인해야 합니다.

요약

View의 생명주기는 생성, 측정, 레이아웃, 드로잉, 이벤트 처리 및 최종 분리를 포함하며, 안드로이드 애플리케이션 내에서 표시되고 사용되는 동안 거치는 단계를 반영합니다. 자세한 내용은 [안드로이드 공식 문서](#)⁶⁵를 확인하세요.

실전 질문

Q) 이미지 로딩이나 애니메이션 설정과 같이 비용이 많이 드는 기능을 포함하는 커스텀 View를 만든다고 가정해 봅시다. View 생명주기의 어느 시점에서 이러한 리소스 및 기능을

⁶⁴<https://developer.android.com/reference/android/graphics/Canvas>

⁶⁵<https://developer.android.com/develop/ui/views/layout/custom-views/custom-components>

초기화해야 하며, 메모리 누수를 방지하기 어떻게 방지할 수 있나요?

Q) 애플리케이션에 성능 문제가 발생하는 동적으로 생성된 View를 포함하는 복잡한 레이아웃이 있습니다. 적절한 응답성을 유지하면서 렌더링 효율성을 향상시키기 위해 onMeasure() 및 onLayout() 메서드를 어떻게 최적화할 수 있을까요?

 **Pro Tips for Mastery:** View의 `findViewTreeLifecycleOwner()` 함수는 어떤 역할을 하나요?

`findViewTreeLifecycleOwner()` 함수는 View 클래스의 확장 함수입니다. View 트리 계층을 거슬러 올라가 View 트리에 연결된 가장 가까운 LifecycleOwner를 찾아 반환합니다. LifecycleOwner는 일반적으로 **Activity**, **Fragment** 또는 LifecycleOwner를 구현하는 커스텀 컴포넌트와 같은 호스팅 컴포넌트의 생명주기 범위를 나타냅니다. LifecycleOwner를 찾지 못하면 함수는 null을 반환합니다.

`findViewTreeLifecycleOwner()`를 사용하는 이유

이 함수는 LiveData, ViewModel 또는 LifecycleObserver와 같은 생명주기 인식 요소와 상호 작용해야 하는 커스텀 View나 서드파티 컴포넌트로 작업할 때 특히 유용합니다. 이를 통해 View는 호스팅 Activity나 Fragment에 대한 명시적인 의존성 없이 연관된 생명주기에 접근할 수 있습니다.

`findViewTreeLifecycleOwner()`를 사용하면 다음을 보장할 수 있습니다.

- 생명주기를 인식하는 컴포넌트(LiveData 등)가 올바른 생명주기에 제대로 바인딩됩니다.
- 생명주기가 끝나면 관찰자가 정리되도록 하여 메모리 누수를 방지합니다.

LifecycleObserver 인스턴스를 바인딩해야 하는 커스텀 View를 떠올려 봅시다. `findViewTreeLifecycleOwner()`를 사용하면 관찰을 올바른 생명주기에 시작할 수 있습니다.

그림 72. Using findViewTreeLifecycleOwner with LiveData

```
1 class CustomLifecycleAwareView @JvmOverloads constructor(
2     context: Context,
3     attrs: AttributeSet? = null
4 ) : LinearLayout(context, attrs) {
5
6     private var lifecycleObserver: LifecycleObserver? = null
7
8     fun bindObserver(observer: LifecycleObserver) {
9         lifecycleObserver?.let { removeObserver(it) }
10
11     val lifecycleOwner = findViewTreeLifecycleOwner()
12
13     lifecycleOwner?.lifecycle?.addObserver(observer) ?: run {
14         Log.e("CustomView", "No LifecycleOwner found for the View")
15     }
16     lifecycleObserver = observer
17 }
18
19 override fun onDetachedFromWindow() {
20     super.onDetachedFromWindow()
21     lifecycleObserver?.let { removeObserver(it) }
22 }
23
24 private fun removeObserver(observer: LifecycleObserver) {
25     findViewTreeLifecycleOwner()?.lifecycle?.removeObserver(observer)
26     lifecycleObserver = null
27 }
28 }
```

여기서 커스텀 CustomLifecycleAwareView는 View 트리의 가장 가까운 LifecycleOwner에 동적으로 바인딩되어 LifecycleObserver 관찰이 적절한 생명주기에 연결되도록 보장합니다.

주요 사용 사례

- 커스텀 뷰(Custom Views)**: 커스텀 View 내의 생명주기 인식(lifecycle-aware, 생명주기와 연결되어 생명주기의 변화에 따라 동작이 결정되는) 컴포넌트가 LifecycleObserver, LiveData와 같은 생명주기 관찰자를 관찰하거나 리소스를 관리할 수 있도록 합니다.
- 서드파티 라이브러리(Third-Party Libraries)**: 재사용 가능한 UI 컴포넌트가 명시적인 생명주기 관리 없이 생명주기 인식 리소스와 상호 작용할 수 있도록 합니다.
- 로직 결합도 분리(Decoupling Logic)**: View가 View 트리에서 자체 LifecycleOwner를 독립적으로 획득하도록 하여 부모 클래스와의 결합도를 줄이는 데 도움이 됩니다.

한계

`findViewTreeLifecycleOwner()`는 유용한 기능을 제공하지만, View 트리에 LifecycleOwner가 존재해야 합니다. LifecycleOwner를 제공하는 소유자가 없으면 함수는 `null`을 반환하므로 크래시나 예상치 않은 동작이 발생할 수 있기 때문에, 신중하게 사용해야 합니다.

요약

View의 `findViewTreeLifecycleOwner()` 함수는 View 트리에서 가장 가까운 LifecycleOwner를 획득하는 데 유용한 기능을 제공합니다. 커스텀 View나 서드파티 라이브러리에서 생명주기 인식 컴포넌트 작업을 단순화하여 적절한 생명주기 관리를 보장하고 View와 호스팅 컴포넌트 간의 결합도를 줄입니다.

Q) 34. View와 ViewGroup의 차이점은 무엇인가요?

View와 ViewGroup은 UI 컴포넌트를 구현하기 위한 기본적인 개념입니다. 둘 다 `android.view` 패키지의 일부이지만 UI 계층 구조에서 다른 목적을 가집니다.

View란?

View는 화면에 표시되는 직사각형 형태의 UI 구현하는 최소 단위의 단일 컴포넌트입니다. 이는 Button, TextView, ImageView, EditText와 같은 모든 UI 컴포넌트의 기본 클래스가 됩니다. 각 View는 화면에 렌더링되면 터치 또는 키 이벤트와 같이 애플리케이션 사용자와의 상호 작용을 처리합니다.

그림 73. Example of a View

```
1 val textView = TextView(context).apply {  
2     text = "Hello, World!"  
3     textSize = 16f // sp 단위 사용 권장  
4     setTextColor(Color.BLACK)  
5 }
```

 **Additional Tips:** View 시스템은 전체 UI 프레임워크의 중추 역할을 하는 안드로이드 개발의 핵심 기반 중 하나입니다. 렌더링, UI 컴포넌트 업데이트 및 사용자 상호 작용을 가능하게 하는 콜백 시스템 관리를 담당합니다. 기본 버튼부터 복잡한 레이아웃까지 모든 UI 요소는 View 클래스 위에 구축됩니다.

AOSP의 [View.java](#)⁶⁶ 내부 구현을 살펴보면 34,000줄 이상의 코드로 구현되어 있습니다. 이것이 의미하는 바는 View 인스턴스를 생성하고 관리하는 것이 의외로 상당한 오버헤드가 수반됨을 뜻합니다. 결과적으로 불필요한 View 생성은 애플리케이션 성능에 영향을 미쳐 메모리 사용량을 늘리고 전체 레이아웃의 렌더링 속도를 늦출 수 있습니다.

성능을 최적화하려면 가능한 한 불필요한 View 인스턴스를 피해야 합니다. 또한 여러번 중첩된 뷰는 측정, 레이아웃 및 렌더링 시간을 증가시킬 수 있으므로 레이아웃 트리의 계층 구조를 단순화하는 것이 중요합니다. UI 계층 구조를 얹고 효율적으로 유지하면 더 부드러운 성능, 더 나은 응답성 및 낮은 리소스 소비를 보장할 수 있습니다.

ViewGroup이란?

ViewGroup은 여러 View 또는 다른 ViewGroup 요소를 포함하는 일종의 컨테이너입니다. 이는 LinearLayout, RelativeLayout, ConstraintLayout, FrameLayout과 같은 레이아웃의 기본이 되는 클래스입니다. ViewGroup은 자식 뷰의 레이아웃과 위치를 관리하며, 화면에서 자식 View들의 사이즈가 측정되고 그려지는 방식을 정의합니다.

⁶⁶<https://cs.android.com/android/platform/superproject/main/+/main/frameworks/base/core/java/android/view/View.java>

그림 74. Example of a ViewGroup

```
1 val linearLayout = LinearLayout(context).apply {  
2     orientation = LinearLayout.VERTICAL  
3     addView(TextView(context).apply { text = "Child 1" })  
4     addView(Button(context).apply { text = "Child 2" })  
5 }
```

💡 Additional Tips: ViewGroup 클래스는 View를 확장하고 ViewParent 및 ViewManager 인터페이스를 모두 구현합니다. ViewGroup은 다른 View 객체를 위한 컨테이너 역할을 하므로 독립형 View보다 본질적으로 더 복잡하고 리소스 집약적입니다. LinearLayout, RelativeLayout, ConstraintLayout과 같은 레이아웃은 모두 ViewGroup 구현의 예이며, ViewGroup 인스턴스의 과도한 중첩은 렌더링 성능에 부정적인 영향을 미칠 수 있습니다.

ViewParent 인터페이스는 View 객체의 부모 역할을 담당하며, 레이아웃 측정, 터치 이벤트 처리 및 렌더링 순서를 관리합니다. 반면에 ViewManager 인터페이스는 ViewGroup 계층 내에서 자식 뷰를 동적으로 추가하고 제거하는 메서드를 제공합니다. ViewGroup은 추가적인 레이아웃 계산을 수행하고 여러 자식 뷰를 관리해야 하므로, 불필요한 중첩을 줄이는 것이 성능을 최적화하고 원활한 UI 렌더링을 보장합니다.

View와 ViewGroup의 주요 차이점**1. 목적**

- View는 콘텐츠를 표시하거나 사용자와 상호 작용하도록 설계된 단일 UI 요소입니다.
- ViewGroup은 여러 자식 뷰를 구성하고 관리하기 위한 컨테이너입니다.

2. 계층

- View는 UI 계층 구조의 리프 노드(leaf node)입니다. 따라서 다른 뷰를 포함할 수 없습니다.
- ViewGroup은 여러 자식 뷰 또는 다른 ViewGroup 요소를 포함할 수 있는 브랜치 노드(branch node)입니다.

3. 레이아웃 동작

- View는 레이아웃 매개변수에 의해 정의된 자체적인 크기와 위치를 가집니다.
- ViewGroup은 LinearLayout 또는 ConstraintLayout과 같이 정의된 레이아웃 규칙을 사용하여 자식 뷰의 크기와 위치를 결정합니다.

4. 상호작용 핸들링

- View는 터치 및 키 이벤트를 처리할 수 있습니다.
- ViewGroup은 onInterceptTouchEvent와 같은 메서드를 사용하여 자식의 이벤트를 가로채고 관리할 수 있습니다.

5. 성능

- ViewGroup은 계층 구조로 인해 렌더링에 복잡성을 더합니다. 중첩된 ViewGroup을 과도하게 사용하면 렌더링 시간 증가 및 UI 업데이트 지연과 같은 성능 문제가 발생할 수 있습니다.

요약

View는 모든 UI 요소의 근간이 되며, ViewGroup은 여러 View 객체를 구성하고 관리하기 위한 컨테이너 역할을 합니다. 이들은 함께 복잡한 안드로이드 사용자 인터페이스를 구축하기 위해 다양한 형태로 활용됩니다. 각각의 역할과 차이점을 이해하는 것은 레이아웃을 최적화하고 반응성 있는 사용자 경험을 보장할 수 있습니다.

실전 질문

- Q) View 생명주기에서 requestLayout(), invalidate(), postInvalidate()가 어떻게 작동하는지 설명하고 각각 언제 사용해야 하나요?
- Q) View 생명주기는 Activity 생명주기와 어떻게 다르며, 효율적인 UI 렌더링을 위해 둘다 이해하는 것이 왜 중요한가요?

Q) 35. **ViewStub**이란 무엇이고, 이를 사용하여 UI 성능을 최적화해 본 경험이 있나요?

ViewStub은 명시적으로 필요할 때까지 레이아웃의 인플레이션을 지연시키는 데 사용되는 가볍고 보이지 않는 플레이스홀더 뷰입니다. 앱 생명주기 동안 지금 당장 필요하지 않은 뷰를 필요한 시기에 적절하게 인플레이션하여 오버헤드를 피함으로써 성능을 개선하는 데 사용됩니다.

ViewStub의 주요 특징

- 가벼움**: **ViewStub**은 인플레이션될 때까지 레이아웃 공간을 차지하거나 리소스를 소비하지 않으므로 메모리 공간이 최소화된 매우 가벼운 뷰입니다.
- 인플레이션 지연**: **ViewStub**에 지정된 실제 레이아웃은 `inflate()` 메서드가 호출되거나 **ViewStub**이 보이게 될 때만 인플레이션됩니다.
- 일회성**: 한번 인플레이션되면 **ViewStub**은 뷰 계층 구조에서 인플레이션된 레이아웃으로 대체되며 재사용할 수 없습니다.

ViewStub의 일반적인 사용 사례

- 조건부 레이아웃**: **ViewStub**은 오류 메시지, 진행률 표시줄 또는 선택적 UI 요소와 같이 조건부로 표시되는 레이아웃에 이상적입니다.
- 초기 렌더링 시간 절감**: 복잡하거나 리소스 집약적인 뷰의 인플레이션을 지연시킴으로써 **ViewStub**은 `Activity` 또는 `Fragment`의 초기 렌더링 시간을 개선하는 데 도움이 됩니다.
- 동적 UI**: 필요할 때만 화면에 동적으로 콘텐츠를 렌더링하는 데 사용될 수 있어 메모리 사용량을 최적화합니다.

ViewStub 사용 방법

ViewStub은 인플레이션할 레이아웃 속성과 함께 XML 레이아웃에 정의할 수 있습니다.

그림 75. activity_main.xml

```
1 <LinearLayout  
2     xmlns:android="http://schemas.android.com/apk/res/android"  
3     android:layout_width="match_parent"  
4     android:layout_height="match_parent"  
5     android:orientation="vertical">  
6  
7     <!-- Regular Views -->  
8     <TextView  
9         android:id="@+id/title"  
10        android:layout_width="wrap_content"  
11        android:layout_height="wrap_content"  
12        android:text="Main Content" />  
13  
14     <!-- Placeholder ViewStub -->  
15     <ViewStub  
16         android:id="@+id/viewStub"  
17         android:layout_width="match_parent"  
18         android:layout_height="wrap_content"  
19         android:layout="@layout/optional_content" />  
20 </LinearLayout>
```

그림 76. Inflating a ViewStub

```
1 class MainActivity : AppCompatActivity() {  
2     override fun onCreate(savedInstanceState: Bundle?) {  
3         super.onCreate(savedInstanceState)  
4         setContentView(R.layout.activity_main)  
5  
6         val viewStub = findViewById<ViewStub>(R.id.viewStub)  
7  
8         // 필요할 때 레이아웃 인플레이션  
9         val inflatedView: View? = try {
```

```
10         viewStub.inflate() // 성공 시 inflatedView 반환, 실패 시 null 또는 예외
11     } catch (e: IllegalStateException) {
12         // 이미 인플레이트된 경우 findViewById 사용
13         findViewById(viewStub.inflatedId)
14     }
15
16     // 인플레이션된 뷰에 접근 (null 체크 필요)
17     inflatedView?.let {
18         val optionalTextView = it.findViewById<TextView>(R.id.optionalText)
19         optionalTextView.text = "Inflated Content"
20     }
21 }
22 }
```

ViewStub의 장점

- 최적화된 성능**: 늦게 초기화 해도 되는 뷰 생성을 지연시켜 메모리 사용량을 줄이고 초기 렌더링 성능을 개선합니다.
- 쉬운 뷰 관리**: 뷰를 수동으로 추가하거나 제거하지 않고도 UI 요소를 선택적으로 렌더링함으로써 쉽게 관리할 수 있습니다.
- 쉬운 사용성**: API 사용이 간단하고 XML 통합이 쉬워 개발자가 쉽게 활용할 수 있습니다.

ViewStub의 한계

- 일회성**: 일단 인플레이션되면 ViewStub은 뷰 계층 구조에서 제거되며 재사용할 수 없습니다.
- 제한된 컨트롤**: 플레이스홀더이므로 인플레이션될 때까지 사용자 상호 작용을 처리하거나 복잡한 작업을 수행할 수 없습니다.

요약

ViewStub은 필요할 때까지 뷰를 인플레이션을 지연시켜 성능을 최적화하는 유용한 UI 컴포넌트입니다. 조건부 뷰이나 지금 당장 렌더링이 필요하지 않은 뷰에 특히 유용하

며, 메모리 사용량을 줄이고 앱 응답성을 개선하는 데 도움이 됩니다. ViewStub을 적절하게 사용하면 더 효율적이고 개선된 사용자 경험을 얻을 수 있습니다.

실전 질문

Q) ViewStub이 인플레이션될 때 어떤 일이 발생하며, 레이아웃 성능 및 메모리 사용량 측면에서 뷰 계층 구조에 어떤 영향을 미치나요?

Q) 36. 커스텀 뷰(custom views)는 어떻게 구현하나요?

커스텀 뷰 구현은 여러 화면에서 재사용해야 하는 특정 스펙과 동작을 가진 UI 컴포넌트를 사용자 정의해야 할 때 필수적입니다. 커스텀 뷰를 사용하면 개발자는 애플리케이션 전체에서 일관성과 유지 관리성을 보장하면서 시각적 표현과 상호 작용 로직을 모두 맞춤 설정할 수 있습니다. 커스텀 뷰를 생성하면 복잡한 UI 로직을 캡슐화하고 재사용성을 높이며 프로젝트 내 다른 레이어의 구조를 단순화할 수 있습니다.

또한, 애플리케이션이 안드로이드에서 기본적으로 제공하는 표준적인 UI 컴포넌트로 달성할 수 없는 디자인 요청사항을 구현해야 하는 경우 커스텀 뷰 개발이 필수적입니다. 안드로이드 개발에서는 다음 단계에 따라 XML을 사용하여 커스텀 뷰를 생성할 수 있습니다.

1. 커스텀 View 클래스 생성하기

먼저, 기본 뷰 클래스(View, ImageView, TextView 등)를 확장하는 새 클래스를 정의합니다. 그런 다음 구현하려는 커스텀 동작에 따라 `onDraw()`, `onMeasure()`, `onLayout()`과 같은 필요한 생성자 및 메서드를 재정의합니다. 아래 예시는 `onDraw()` 메서드를 오버라이드하여 캔버스에 적접 빨간색 원을 그리는 커스텀 뷰 입니다.

그림 77. Custom CircleView.kt

```
1 class CustomCircleView @JvmOverloads constructor(
2     context: Context,
3     attrs: AttributeSet? = null,
4     defStyle: Int = 0
5 ): View(context, attrs, defStyle) {
6
7     private val paint = Paint(Paint.ANTI_ALIAS_FLAG).apply { // Anti-aliasing 추가
8         color = Color.RED
9         style = Paint.Style.FILL
10    }
11
12    override fun onDraw(canvas: Canvas) {
13        super.onDraw(canvas)
14        // 중앙에 빨간색 원 그리기
15        canvas.drawCircle(width / 2f, height / 2f, min(width, height) / 4f, paint) // ← 반지름 설정
16    }
17 }
```

2. XML 레이아웃에서 커스텀 View 사용하기

커스텀 뷰 클래스를 생성한 후 XML 레이아웃 파일에서 직접 참조할 수 있습니다. 커스텀 클래스의 전체 패키지 이름(아래 예시의 경우 com.example.myapp.ui.CustomCircleView)을 정확하게 사용해야 합니다. 아래 예시와 같이 XML에서 정의할 수 있는 커스텀 속성(layout_width 등)을 커스텀 뷰에 전달할 수도 있습니다.

그림 78. Layout.xml

```
1 <com.example.myapp.ui.CustomButtonView
2     android:id="@+id/customCircleView"
3     android:layout_width="100dp"
4     android:layout_height="100dp"
5     android:layout_gravity="center"/>
```

3. 커스텀 속성 추가하기 (선택 사항)

res/values 폴더에 새 attrs.xml 파일을 만들어 커스텀 속성을 정의할 수 있습니다. 이를 통해 XML 레이아웃에서 뷰의 속성을 커스텀할 수 있습니다. 가령, 원을 그리는 커스텀 뷰에서 원의 색상이나 반지름을 사용자 정의할 수 있도록 하여, 커스텀 뷰의 재사용성을 확장시킬 수 있습니다.

그림 79. attrs.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3     <declare-styleable name="CustomCircleView">
4         <attr name="circleColor" format="color"/>
5         <attr name="circleRadius" format="dimension"/>
6     </declare-styleable>
7 </resources>
```

커스텀 뷰 클래스에서 context.obtainStyledAttributes()를 사용하여 커스텀 속성 값을 가져올 수 있습니다.

그림 80. Custom CircleView.kt

```
1 class CustomCircleView @JvmOverloads constructor(
2     context: Context,
3     attrs: AttributeSet? = null,
4     defStyleAttr: Int = 0 // defStyleAttr로 이름 변경 권장
5 ): View(context, attrs, defStyleAttr) {
6
7     var circleColor: Int = Color.RED
8     var circleRadius: Float = 50f
9
10    private val paint = Paint(Paint.ANTI_ALIAS_FLAG).apply {
11        style = Paint.Style.FILL
12    }
13
14    init {
15        attrs?.let {
16            getAttrs(it, defStyleAttr)
17        }
18        paint.color = circleColor // 페인트 색상 초기화
19    }
20
21    private fun getAttrs(attrs: AttributeSet, defStyleAttr: Int) {
22        val typedArray = context.obtainStyledAttributes(
23            attrs, R.styleable.CustomButtonView, defStyleAttr, 0 // 기본 스타일 리소스 0
24        )
25        try {
26            setTypeArray(typedArray)
27        } finally {
28            typedArray.recycle()
29        }
30    }
31
32    private fun setTypeArray(typedArray: TypedArray) {
```

```
33     circleColor = typedArray.getColor(R.styleable.CustomButtonCircleView_circleColor,
34         → Color.RED)
35     circleRadius =
36         → typedArray.getDimension(R.styleable.CustomButtonCircleView_circleRadius, 50f)
37 }
38
39
40
41
42
43 // 필요하다면 onMeasure 재정의
44 override fun onMeasure(widthMeasureSpec: Int, heightMeasureSpec: Int) {
45     val desiredWidth = (circleRadius * 2 + paddingLeft + paddingRight).toInt()
46     val desiredHeight = (circleRadius * 2 + paddingTop + paddingBottom).toInt()
47
48     val width = resolveSize(desiredWidth, widthMeasureSpec)
49     val height = resolveSize(desiredHeight, heightMeasureSpec)
50     setMeasuredDimension(width, height)
51 }
52
53 // circleColor 또는 circleRadius 변경 시 뷰를 다시 그리도록 하는 메서드
54 fun setCircleProperties(color: Int, radius: Float) {
55     this.circleColor = color
56     this.circleRadius = radius
57     paint.color = color
58     requestLayout() // 뷰의 크기 변경 가능성이 있으므로 requestLayout 호출
59     invalidate() // 다시 그리기 요청
60 }
61 }
```

이제 XML 파일에서 아래와 같이 커스텀 속성을 사용할 수 있습니다.

그림 81. Custom Attributes.xml

```
1 <com.example.myapp.ui.CustomButtonView
2     xmlns:app="http://schemas.android.com/apk/res-auto"
3     android:id="@+id/customCircleView"
4     android:layout_width="100dp"
5     android:layout_height="100dp"
6     app:circleColor="@color/blue"
7     app:circleRadius="30dp"/>
```

4. 레이아웃 측정 처리하기 (선택 사항)

커스텀 뷰가 크기를 측정하는 방식을 수동적으로 처리하고 싶고, 특히 표준적인 뷰와 다르게 동작해야 하는 경우 `onMeasure()` 메서드를 재정의하여 구현할 수 있습니다.

그림 82. onMeasure.kt

```
1 override fun onMeasure(widthMeasureSpec: Int, heightMeasureSpec: Int) {
2     // 원하는 기본 크기 설정
3     val desiredWidth = (paddingLeft + paddingRight +
4         suggestedMinimumWidth).coerceAtLeast(200) // 예시 크기
5
6     val desiredHeight = (paddingTop + paddingBottom +
7         suggestedMinimumHeight).coerceAtLeast(200) // 예시 크기
8
9     // MeasureSpec 모드와 크기를 기반으로 최종 크기 결정
10    val width = resolveSize(desiredWidth, widthMeasureSpec)
11    val height = resolveSize(desiredHeight, heightMeasureSpec)
12
13    // 최종 측정된 크기 설정
14    setMeasuredDimension(width, height)
15 }
```

 **Additional Tips:** 애플리케이션 요구 사항이나 특수한 디자인 스펙에 맞는 재사용 가능하고 특화된 컴포넌트가 필요할 때는 커스텀 뷰 개발이 필수적입니다. 커스텀 뷰는 애니메이션, 콜백, 커스텀 속성 및 기타 고급 기능을 통합하여 기능과 사용자 경험을 향상시킬 수 있습니다. 커스텀 뷰 구축에 대한 이해를 높이려면 GitHub에서 다양한 예제와 실전 코드를 살펴보며 학습하실 수 있습니다. 이미 많은 애플리케이션에서 사용되고 있는 검증된 오픈 소스 라이브러리의 사례를 관찰하는 것이 좋은 통찰력과 영감을 얻는 가장 빠른 길입니다. 이미 많은 안드로이드 프로젝트에서 사용되고 있는 오픈 소스 라이브러리인 [ElasticViews⁶⁷](#) 및 [ProgressView⁶⁸](#) 등을 살펴보시면 커스텀 뷰 구현에 대한 실질적인 사례를 살펴보실 수 있습니다. 해당 프로젝트들은 재사용 가능하고 동적인 커스텀 뷰를 개발하는 실용적인 접근 방식을 보여주고 있습니다. 또는 [CircleImageView⁶⁹](#)를 살펴보실 수도 있습니다.

요약

안드로이드에서 커스텀 뷰를 잘 구현하면 UI 디자인에 유연성을 부여할 수 있습니다. 또한, 커스텀 뷰 시스템과 [Canvas⁷⁰](#)를 사용하여 다양한 커스텀 위젯 만들고 사용자 경험을 풍부하게 할 수 있습니다. [커스텀 뷰에 관한 안드로이드 개발자 문서⁷¹](#)를 통해 추가적인 학습을 하실 수 있습니다.

실전 질문

Q) XML 레이아웃에서 이전 버전과의 호환성을 보장하면서 커스텀 뷰에 커스텀 속성을 효율적으로 적용하려면 어떻게 해야 하나요?

 **Pro Tips for Mastery:** 커스텀 뷰의 기본 생성자에서 `@JvmOverloads`를 사용할 때 왜 주의해야 하나요?

Kotlin의 `@JvmOverloads` 어노테이션은 Kotlin 함수 또는 클래스에 대해 여러 오버로드된 메서드 또는 생성자를 자동으로 생성하여 Kotlin과 Java 간의 상호 운용성을 단순화하는

⁶⁷<https://github.com/skydoves/ElasticViews>

⁶⁸<https://github.com/skydoves/ProgressView>

⁶⁹<https://github.com/hdodenhof/CircleImageView>

⁷⁰<https://developer.android.com/reference/android/graphics/Canvas>

⁷¹<https://developer.android.com/guide/topics/ui/custom-components>

기능입니다. 이는 Java가 기본 인수를 기본적으로 지원하지 않기 때문에 Kotlin의 기본 인수가 관련된 경우 특히 유용합니다.

`@JvmOverloads`를 사용하면 Kotlin 컴파일러는 컴파일된 바이트코드에서 기본값을 가진 매개변수의 모든 가능한 조합을 나타내기 위해 여러 메서드 또는 생성자 시그니처를 컴파일 타임에 자동적으로 생성합니다.

하지만 커스텀 뷰를 구현할 때 `@JvmOverloads`를 신중하게 사용하지 않으면 의도치 않게 기본 뷰 스타일을 재정의하여 커스텀 뷰의 의도된 스타일링이 손실될 수 있습니다. 이는 `Button`이나 `TextView`와 같이 미리 정의된 스타일이 있는 안드로이드 뷰를 확장하는 커스텀 뷰를 만들 때 특히 문제가 됩니다.

예를 들어, 커스텀 `TextInputEditText`를 구현할 때 다음과 같이 정의할 수 있습니다.

```

1 class ElasticTextInputEditText @JvmOverloads constructor(
2     context: Context,
3     attrs: AttributeSet? = null,
4     defStyleAttr: Int = 0 // 여기서 defStyleAttr 대신 defStyle을 사용하고 기본값을 0으로 설정
5 ) : TextInputEditText(context, attrs, defStyle) {
6     ..
7 }
```

이 예제에서 `ElasticTextInputEditText`를 일반 `TextInputEditText`처럼 사용하면 예기치 않은 동작이나 스타일링에 잠재적인 손상이 발생할 수 있습니다. 이는 위 코드에서 볼 수 있듯이 `defStyle` 값이 0으로 재정의되어 커스텀 뷰가 의도한 스타일링을 잃을 수 있기 때문입니다.

`View`가 XML 파일에서 인플레이션될 때 두 매개변수 생성자(`Context` 및 `AttributeSet`)가 호출되고, 이는 다시 세 매개변수 생성자를 호출합니다.

```

1 // @JvmOverloads가 생성한 두 매개변수 생성자 (내부적으로)
2 public ElasticTextInputEditText(Context context, @Nullable AttributeSet attrs) {
3     this(context, attrs, 0); // defStyle에 0 전달
4 }
```

세 번째 매개변수인 `defStyleAttr`은 커스텀 구현에서 종종 0으로 설정하는 경우가 많습니다. 그러나 이 세 매개변수 생성자의 목적은 [안드로이드 문서](#)⁷²에 다음과 같이 설명되어 있습니다.

⁷²[https://developer.android.com/reference/android/view/View#View\(android.context.Context,%20android.util.AttributeSet,%20int\)](https://developer.android.com/reference/android/view/View#View(android.context.Context,%20android.util.AttributeSet,%20int))

XML에서 인플레이션을 수행하고 테마 속성에서 클래스별 기본 스타일을 적용합니다. 이 View 생성자를 사용하면 하위 클래스가 인플레이션될 때 자체 기본 스타일을 사용할 수 있습니다. 예를 들어, Button 클래스의 생성자는 이 버전의 상위 클래스 생성자를 호출하고 defStyleAttr에 R.attr.buttonStyle을 제공합니다. 이를 통해 테마의 버튼 스타일이 모든 기본 뷰 속성(특히 배경)뿐만 아니라 Button 클래스의 속성도 수정할 수 있습니다.

적절한 defStyleAttr 값(가령, ElasticTextInputEditText의 경우 R.attr.editTextStyle)을 생략하면 커스텀 뷰가 상속된 스타일 구성을 잃어 XML 인플레이션 중에 일관성 없거나 깨진 동작이 발생할 수 있습니다.

TextInputEditText의 내부 구현을 살펴보면 아래 코드에서 볼 수 있듯이 내부적으로 R.attr.editTextStyle을 defStyleAttr로 사용한다는 것을 알 수 있습니다.

```
1 public class TextInputEditText extends AppCompatEditText {  
2  
3     // ... 필드 정의 ...  
4  
5     public TextInputEditText(@NonNull Context context) {  
6         this(context, null);  
7     }  
8  
9     public TextInputEditText(@NonNull Context context, @Nullable AttributeSet attrs) {  
10        // 기본 스타일 속성으로 R.attr.editTextStyle 사용  
11        this(context, attrs, R.attr.editTextStyle);  
12    }  
13  
14    public TextInputEditText(  
15        @NonNull Context context, @Nullable AttributeSet attrs, int defStyleAttr) {  
16        super(context, attrs, defStyleAttr);  
17        // ... 나머지 생성자 로직 ...  
18    }  
19}
```

TextInputEditText의 구현에서 세 번째 매개변수(defStyleAttr)로 androidx.appcompat.R.attr.editText 전달하는 것을 어떤 방법(AOSP를 서치하든, IDE에서 클래스를 타고 올라가든)으로든 알아야합니다. 이는 테마 스타일을 제공하는 AppCompat 라이브러리나 Android SDK에서 기본적으로 제공하는 여러 커스텀 뷰마다 완전히 다른 스타일 값을 사용하고 있습니다. ElasticTextInputEditText 예시의 경우는 올바른 스타일링을 보장하기 위해 커스텀 뷰의 생성자에서 defStyleAttr 매개변수의 기본값으로 R.attr.editTextStyle을 설정하여 수정할 수 있습니다.

그림 83. ElasticTextInputEditText.kt

```
1 class ElasticTextInputEditText @JvmOverloads constructor(
2     context: Context,
3     attrs: AttributeSet? = null,
4     // 기본값으로 올바른 스타일 속성 참조
5     defStyleAttr: Int = androidx.appcompat.R.attr.editTextStyle
6 ) : TextInputEditText(context, attrs, defStyleAttr) {
7     // Custom implementation
8 }
```

androidx.appcompat.R.attr.editTextStyle을 기본값으로 명시적으로 할당함으로써 커스텀 뷰가 XML 인플레이션 중에 예상되는 기본 스타일을 상속하여 원래 TextInputEditText의 동작과 일관성을 유지하도록 보장합니다.

Q) 37. Canvas란 무엇이며 어떻게 활용하나요?

Canvas는 커스텀 드로잉을 위한 핵심 구성 요소입니다. 화면이나 Bitmap과 같은 다른 드로잉이 가능한 표면에 직접 그래픽을 렌더링하기 위한 인터페이스를 제공합니다. Canvas는 개발자에게 드로잉 프로세스에 대한 완전한 제어를 제공하여 커스텀 뷰, 애니메이션 및 시각 효과를 만드는 데 다방면으로 활용됩니다.

Canvas 작동 방식

Canvas 클래스는 도형, 텍스트, 이미지 및 기타 콘텐츠를 그릴 수 있는 2D 드로잉 표면을 나타냅니다. 그려진 콘텐츠가 어떻게 보여야 하는지(색상, 스타일, 스트로크 너비 등)

정의하는 Paint 클래스와 긴밀하게 상호 작용합니다. 커스텀 View의 onDraw() 메서드를 재정의하면 Canvas 객체가 전달되어 무엇을 그릴지 정의할 수 있습니다.

다음은 View에서 기본적인 커스텀 드로잉 예시입니다.

그림 84. Custom View.kt

```
1 class CustomView(context: Context) : View(context) {
2     private val paint = Paint().apply {
3         color = Color.BLUE
4         style = Paint.Style.FILL
5     }
6
7     override fun onDraw(canvas: Canvas) {
8         super.onDraw(canvas)
9         // View 중앙에 파란색 원 그리기
10        canvas.drawCircle(width / 2f, height / 2f, 100f, paint)
11    }
12 }
```

이 예제에서 onDraw() 메서드는 Canvas 객체를 사용하여 커스텀 뷰의 중앙에 파란색 원을 그립니다.

Canvas의 일반적인 작업

Canvas는 다음과 같은 다양한 드로잉 작업을 허용합니다.

- **도형(Shapes)**: drawCircle(), drawRect(), drawLine()과 같은 메서드를 사용하여 원, 사각형, 선과 같은 도형을 그릴 수 있습니다.
- **텍스트(Text)**: drawText() 메서드는 지정된 좌표와 모양으로 텍스트를 렌더링합니다.
- **이미지(Images)**: drawBitmap()을 사용하여 이미지를 렌더링합니다.
- **커스텀 패스(Custom Paths)**: Path 객체와 drawPath()를 결합하여 복잡한 모양을 그릴 수 있습니다.

변환 (Transformations)

Canvas는 크기 조절(scaling), 회전(rotating), 이동(translating)과 같은 변환을 지원합니다. 이러한 작업은 캔버스의 좌표계를 수정하여 복잡한 장면을 더 쉽게 그릴 수 있도록 합니다.

- **이동(Translation)**: `canvas.translate(dx, dy)`를 사용하여 캔버스 원점을 새 위치로 이동합니다.
- **크기 조절(Scaling)**: `canvas.scale(sx, sy)`를 사용하여 드로잉 크기를 배율로 조절합니다.
- **회전(Rotation)**: `canvas.rotate(degrees)`를 사용하여 지정된 각도로 캔버스를 회전합니다.

이러한 변환은 메서드를 호출 할 때마다 누적되어 적용되며, 이후의 모든 드로잉 작업에 영향을 미칩니다.

사용 사례

Canvas는 다음과 같이 고급 커스텀 그래픽이 필요한 시나리오에서 특히 유용합니다.

1. **Custom Views**: 표준적으로 제공되는 컴포넌트로 달성할 수 없는 고유한 UI 컴포넌트 만들기.
2. **Games**: 정밀한 제어로 게임 그래픽을 렌더링하는 경우.
3. **Charts and Diagrams**: 커스텀 형식으로 데이터를 시각화 해야하는 경우.
4. **Image Processing**: 프로그래밍 방식으로 이미지 수정 또는 결합이 필요한 경우.

요약

Canvas는 화면에 커스텀 그래픽을 렌더링하는 유연하고 유용한 방법을 제공합니다. 도형, 텍스트, 이미지 그리기 메서드와 변환을 활용하여 개발자는 풍부한 시각적 및 커스텀 경험을 만들 수 있습니다. 고급 그래픽 기능이 필요한 커스텀 뷰를 만드는 데 널리 사용됩니다.

실전 질문

Q) AndroidX 라이브러리에서 지원하지 않는 복잡한 모양이나 UI 요소를 렌더링 하는 커스텀 뷰를 어떻게 만들 수 있을까요? 예를 들어, 화면에 로딩 중 상태를 표현하는 커스텀 스피너를 직접 그린다면, 어떤 Canvas 메서드와 API를 활용할 수 있을까요?

Q) 38. View 시스템의 무효화(invalidation)란 무엇인가요?

무효화(Invalidation)는 View를 다시 그려야 함을 표시하는 프로세스를 의미합니다. 이는 변경 사항이 발생할 때 UI를 업데이트하기 위해 안드로이드 View 시스템에서 사용되는 기본 메커니즘입니다. View가 무효화되면 시스템은 다음 드로잉 주기 동안 화면의 해당 부분을 새로 고쳐야 함을 인지하고 새롭게 그립니다.

무효화 작동 방식

View에서 invalidate() 또는 postInvalidate()와 같은 메서드를 호출하면 무효화 프로세스가 트리거됩니다. 시스템은 View를 “더티(dirty)”로 플래그 지정하며, 이는 다시 그려야 함을 의미합니다. 다음 프레임 동안 시스템은 무효화된 View를 드로잉 패스에 포함시켜 시각적 표현을 업데이트합니다.

예를 들어, 위치, 크기 또는 모양과 같은 View의 속성이 변경되면 무효화 프로세스를 통해 사용자가 업데이트된 상태를 볼 수 있도록 보장합니다.



Additional Tips: 무효화(invalidation) 프로세스가 필요한 이유는 View에 변화가 필요할 때마다, 새롭게 View를 다시 렌더링 하는 것이 아니라, 개발자나 시스템에 의해 시기적절한 순간에 업데이트되도록 제한함으로써 더 나은 성능을 만들 수 있습니다. 그렇지 않고 자동적으로 View가 업데이트된다면, 무분별한 렌더링으로 앱의 전반적인 성능이 크게 떨어질 수 있습니다. Jetpack Compose에서는 이러한 무효화 프로세스가 없고 상태에 따라 자동적으로 UI가 업데이트되기 때문에 성능에 더 각별한 주의가 필요합니다.

무효화를 위한 주요 메서드

1. **invalidate()**: 이 메서드는 단일의 View를 무효화하는 데 사용됩니다. View를 더티로 표시하여 시스템이 다음 레이아웃 패스 중에 다시 그리도록 신호를 보냅니다. View를 즉시 다시 그리는 것이 아니라 다음 프레임을 위해 예약합니다.
2. **invalidate(Rect dirty)**: 이는 invalidate()의 오버로드된 버전으로, 다시 그려야 하는 View 내의 특정 직사각형 영역을 지정할 수 있습니다. 더 작은 부분으로 다시 그리기를 제한하여 성능을 최적화합니다.
3. **postInvalidate()**: 이 메서드는 UI 스레드가 아닌 스레드에서 View를 무효화하는 데 사용됩니다. 무효화 요청을 메인 스레드에 게시하여 스레드 안전성을 보장합니다.

invalidate()를 사용하여 커스텀 View 업데이트하기

아래는 상태가 변경될 때 UI를 다시 그리기 위해 invalidate() 메서드가 사용되는 커스텀 View의 예시입니다.

그림 85. CustomView.kt

```
1 class CustomView(context: Context) : View(context) {
2     private var circleRadius = 50f
3     private val paint = Paint().apply { color = Color.RED } // Paint 객체 미리 생성
4
5     override fun onDraw(canvas: Canvas) {
6         super.onDraw(canvas)
7         // 현재 반지름으로 원 그리기
8         canvas.drawCircle(width / 2f, height / 2f, circleRadius, paint)
9     }
10
11    fun increaseRadius() {
12        circleRadius += 20f
13        invalidate() // View를 다시 그려야 함으로 표시
14    }
15 }
```

무효화 모범 사례

- View의 특정 영역만 다시 그려야 할 때 부분 업데이트를 위해 `invalidate(Rect dirty)`를 사용합니다. 이는 변경되지 않은 영역의 불필요한 다시 그리기를 피하여 성능을 향상 시킵니다.
- 특히 애니메이션이나 복잡한 레이아웃에서 성능 병목 현상을 방지하기 위해 `invalidate()`를 자주 또는 불필요하게 호출하지 않습니다.
- 백그라운드 스레드에서의 무효화 요청에는 `postInvalidate()`를 사용하여 업데이트가 메인 스레드에서 안전하게 발생하도록 보장합니다.

요약

무효화는 UI 업데이트가 시각적으로 반영되도록 보장하는 안드로이드 렌더링 파이프라인의 중요한 개념입니다. 개발자는 `invalidate()` 또는 `postInvalidate()`와 같은 메서드를 사용하여 부드러운 성능을 유지하면서 뷰를 효율적으로 새로 고칠 수 있습니다. 무효화를 적절하게 사용하면 불필요한 다시 그리기를 최소화하여 최적화되고 반응성이 뛰어난 애플리케이션을 만들 수 있습니다.

실전 질문

Q) `invalidate()` 메서드는 어떻게 작동하며 `postInvalidate()`와 어떻게 다른가요? 각각이 적합한 실제 사용 사례를 제시해주세요.

Q) 백그라운드 스레드에서 UI 요소를 업데이트해야 하는 경우, 다시 그리기 작업이 메인 스레드에서 안전하게 수행되도록 어떻게 보장할 수 있나요?

Q) 39. ConstraintLayout이란 무엇인가요?

`ConstraintLayout`⁷³은 여러 레이아웃을 중첩하지 않고 복잡하고 반응성이 뛰어난 사용자 인터페이스를 만들기 위해 안드로이드에서 도입된 유연하고 여러모로 유용한 레이아웃입니다. 다른 뷰나 부모 컨테이너에 상대적인 **제약 조건(constraints)**을 사용하여 뷰의 위치와 크기를 정의할 수 있습니다. 이를 통해 깊게 중첩된 뷰 계층 구조가 필요 없어 성능과 코드 가독성이 향상됩니다.

⁷³<https://developer.android.com/reference/android/support/constraint/ConstraintLayout#developer-guide>

ConstraintLayout의 주요 특징

1. **제약 조건을 이용한 위치 지정**: 뷰는 정렬, 중앙 배치 및 앵커링을 위한 제약 조건을 사용하여 형제 뷰 또는 부모 레이아웃에 상대적으로 위치 지정될 수 있습니다.
2. **유연한 크기 제어**: `match_constraint`, `wrap_content` 및 고정 크기와 같은 옵션을 제공하여 반응형 레이아웃을 쉽게 디자인할 수 있습니다.
3. **체인(Chain) 및 가이드라인(Guideline) 지원**: 체인을 사용하면 뷰를 동일한 간격으로 가로 또는 세로로 그룹화할 수 있으며, 가이드라인을 사용하면 고정 또는 백분율 기반 위치에 정렬할 수 있습니다.
4. **배리어(Barrier) 및 그룹핑(Grouping)**: 배리어는 참조된 뷰의 크기에 따라 동적으로 조정되며, 그룹핑은 여러 뷰의 가시성 변경을 단순화합니다.
5. **성능 향상**: 여러 중첩 레이아웃의 필요성을 줄여 레이아웃 렌더링 속도를 높이고 앱 성능을 향상시킵니다.

ConstraintLayout 예제

아래 코드는 `TextView`와 `Button`이 있는 간단한 레이아웃을 보여줍니다. `Button`은 `TextView` 아래에 위치하고 가로로 중앙에 배치됩니다.

그림 86. `activity_main.xml`

```
1 <androidx.constraintlayout.widget.ConstraintLayout
2     xmlns:android="http://schemas.android.com/apk/res/android"
3     xmlns:app="http://schemas.android.com/apk/res-auto" <!-- app 네임스페이스 추가 -->
4     android:layout_width="match_parent"
5     android:layout_height="match_parent">
6
7     <TextView
8         android:id="@+id/title"
9         android:layout_width="wrap_content"
10        android:layout_height="wrap_content"
11        android:text="Hello, World!"
12        android:layout_marginTop="16dp"
13        app:layout_constraintTop_toTopOf="parent"
14        app:layout_constraintStart_toStartOf="parent"
```

```
15      app:layout_constraintEnd_toEndOf="parent" />
16
17  <Button
18      android:id="@+id/button"
19      android:layout_width="wrap_content"
20      android:layout_height="wrap_content"
21      android:text="Click Me"
22      app:layout_constraintTop_toBottomOf="@+id/title"
23      app:layout_constraintStart_toStartOf="parent"
24      app:layout_constraintEnd_toEndOf="parent"
25      android:layout_marginTop="8dp" /> <!-- 상단 마진 추가 (선택 사항) -->
26  </androidx.constraintlayout.widget.ConstraintLayout>
```

ConstraintLayout의 장점

- 플랫 뷰 계층 구조:** 중첩된 LinearLayout이나 RelativeLayout과 달리 ConstraintLayout은 플랫 계층 구조를 가능하게 하여 렌더링 성능을 향상시키고 레이아웃 관리를 단순화합니다.
- 반응형 디자인:** 백분율 기반 제약 조건 및 배리어와 같은 도구를 제공하여 다양한 화면 크기 및 방향에 맞게 레이아웃을 조정합니다.
- 내장 도구:** Android Studio의 Layout Editor는 시각적 디자인 인터페이스로 ConstraintLayout을 지원하여 제약 조건을 쉽게 만들고 조정할 수 있습니다.
- 고급 기능:** 체인, 가이드라인 및 배리어는 추가 코드나 중첩 레이아웃 없이 복잡한 UI 디자인을 단순화합니다.

ConstraintLayout의 한계

- 단순 레이아웃에 대한 복잡성:** LinearLayout이나 FrameLayout으로 충분할 수 있는 단순한 레이아웃에 사용하기에는 과할 수 있습니다.
- 학습 곡선:** 제약 조건 및 고급 기능에 대한 이해가 필요하며, 처음 접하는 분들은 학습 곡선이 조금 가파를 수 있습니다.

ConstraintLayout 사용 사례

1. **반응형 UI**: 다양한 화면 크기에서 정밀한 정렬과 적응성이 필요한 디자인에 이상적입니다.
2. **복잡한 레이아웃**: 여러 겹치는 요소나 복잡한 위치 지정 요구 사항이 있는 UI에 적합합니다.
3. **성능 최적화**: 중첩된 뷰 계층 구조를 단일 플랫 구조로 대체하여 레이아웃을 최적화하는 데 도움이 됩니다.

요약

ConstraintLayout은 안드로이드 UI를 디자인하기 위한 유용하고 효율적인 레이아웃입니다. 중첩 레이아웃의 필요성을 없애고 위치 지정 및 정렬을 위한 고급 도구를 제공하며 성능을 향상시킵니다. 학습 곡선이 있을 수 있지만 ConstraintLayout을 잘 활용하신다면 반응성이 뛰어나고 시각적으로 매력적인 레이아웃을 만들 수 있습니다.

실전 질문

- Q) ConstraintLayout은 중첩된 LinearLayout 및 RelativeLayout과 비교하여 성능을 어떻게 향상시키나요? ConstraintLayout 사용이 더 효율적인 시나리오를 말씀해주세요.
- Q) ConstraintLayout에서 match_constraint (0dp) 동작이 어떻게 작동하는지 설명해주세요. wrap_content 및 match_parent와 어떻게 다르며, 어떤 상황에서 사용해야 하나요?

Q) 40. SurfaceView 대신 TextureView는 언제 사용해야 하나요?

SurfaceView는 별도의 스레드에서 렌더링이 처리되는 시나리오를 위해 설계된 특수한 View로, 전용 드로잉 표면을 제공합니다. 이는 일반적으로 성능이 중요한 비디오 재생, 커스텀 그래픽 렌더링 또는 게임과 같은 작업에 사용됩니다. SurfaceView의 주요 특징은 메인 UI 스레드 외부에 별도의 표면을 생성하여 다른 UI 작업을 차단하지 않고 효율적인 렌더링을 가능하게 한다는 것입니다.

표면은 SurfaceHolder 콜백 메서드를 통해 생성 및 관리되며, 필요에 따라 렌더링을 시작하고 중지할 수 있습니다. 가령, 저수준 API를 사용하여 비디오를 재생하거나 게임 루프에서 그래픽을 계속해서 그리는 데 SurfaceView를 사용할 수 있습니다.

그림 87. Basic SurfaceView usage

```
1 class CustomSurfaceView(context: Context) : SurfaceView(context),
2     → SurfaceHolder.Callback {
3     init {
4         holder.addCallback(this)
5     }
6
6     override fun surfaceCreated(holder: SurfaceHolder) {
7         // 여기서 렌더링 또는 드로잉 시작
8     }
9
10    override fun surfaceChanged(holder: SurfaceHolder, format: Int, width: Int, height: Int) {
11        // 표면 변경 처리
12    }
13
14    override fun surfaceDestroyed(holder: SurfaceHolder) {
15        // 여기서 렌더링 중지 또는 리소스 해제
16    }
17 }
```

SurfaceView는 연속적인 렌더링(비디오 재생 등)에 효율적이라 크기 조절이나 회전과 같은 변환에는 제한이 있어 고성능 사용 사례에는 적합하지만, 동적인 상호 작용이 요구되는 UI에는 덜 유연하고 적합하지 않습니다.

반면에 TextureView는 콘텐츠를 오프스크린으로 렌더링하는 또 다른 방법을 제공하면서도 SurfaceView와 달리 UI 계층 구조에 원활하게 통합됩니다. 즉, TextureView는 회전, 크기 조절, 알파 블렌дин과 같은 기능을 허용하여 변환하거나 애니메이션화할 수 있습니다. 라이브 카메라 피드를 표시하거나 커스텀 변환으로 비디오를 렌더링하는 등의 작업에 자주 사용됩니다.

SurfaceView와 달리 TextureView는 메인 스레드에서 작동합니다. 이로 인해 연속 렌더링에는 성능적인 측면에서 덜 효율적이지만, 다른 UI 컴포넌트와의 더 나은 상호작용을 가능하게 하고 실시간 변환을 지원합니다. 따라서 상황에 맞는 것을 적절하게 선택하는 것 또한 중요합니다.

그림 88. Basic TextureView usage

```
1 class CustomTextureView(context: Context, attrs: AttributeSet? = null) :  
2     TextureView(context, attrs), TextureView.SurfaceTextureListener {  
3     init {  
4         surfaceTextureListener = this  
5     }  
6  
6     override fun onSurfaceTextureAvailable(surface: SurfaceTexture, width: Int, height:  
7         Int) {  
8         // 렌더링 시작 또는 SurfaceTexture 사용  
9     }  
10  
10    override fun onSurfaceTextureSizeChanged(surface: SurfaceTexture, width: Int,  
11        height: Int) {  
12        // 표면 크기 변경 처리  
13    }  
14  
14    override fun onSurfaceTextureDestroyed(surface: SurfaceTexture): Boolean {  
15        // 리소스 해제 또는 렌더링 중지  
16        return true // SurfaceTexture가 앱 프로세스에 의해 해제되었음을 나타냄  
17    }  
18  
19    override fun onSurfaceTextureUpdated(surface: SurfaceTexture) {  
20        // 표면 텍스처 업데이트 처리 (프레임 업데이트 등)  
21    }  
22 }
```

TextureView는 비디오 스트림 애니메이션이나 UI 내에서 콘텐츠를 동적으로 블렌딩하는 등 시각적 변환이 필요한 사용 사례에 특히 유용합니다.

SurfaceView와 TextureView의 차이점

주요 차이점은 각 컴포넌트가 렌더링 및 UI 통합을 처리하는 방식에 있습니다. SurfaceView는 별도의 스레드에서 작동하여 비디오 재생이나 게임과 같은 연속 렌더링 작업에 효율적입니다.

또한 렌더링을 위한 별도의 Window를 생성하여 성능을 보장하지만, 변환되거나 애니메이션화되는 능력은 제한됩니다. 반면 TextureView는 다른 UI 컴포넌트와 동일한 Window를 공유하여 크기 조절, 회전 또는 애니메이션이 가능하므로 UI 관련 사용 사례에 더 유연합니다. 그러나 메인 스레드에서 작동하므로 고성능 렌더링이 필요한 작업에는 효율적이지 않을 수 있습니다.

요약

SurfaceView는 게임이나 연속적인 비디오 렌더링과 같이 성능이 가장 중요한 시나리오에 가장 적합합니다. 반면에 TextureView는 비디오 애니메이션이나 라이브 카메라 피드 표시와 같이 원활한 UI 통합 및 시각적 변환이 필요한 사용 사례에 더 적합합니다. 이들 간의 선택은 애플리케이션이 성능을 우선시하는지 UI 상호작용과 같은 유연성을 우선시하는지에 따라 달라집니다.

실전 질문

- Q) 효율적인 리소스 관리 및 메모리 누수 방지를 위해 SurfaceView의 생명주기를 어떻게 적절하게 관리해야 하나요?
- Q) 카메라 미리보기를 회전 및 크기 조절과 같은 UI 변환과 함께 표시해야 하는 요구 사항이 주어졌을 때, SurfaceView와 TextureView 중 어떤 컴포넌트를 선택하는 것이 적합한가요?

Q) 41. RecyclerView는 내부적으로 어떻게 작동하나요?

RecyclerView는 새로운 아이템 뷰를 반복적으로 인플레이션하는 대신 재활용하여 대규모 데이터 셋을 효율적으로 표시하도록 설계된 유용하고 유연한 안드로이드 컴포넌트입니다. **ViewHolder 패턴**으로 알려진 뷰 관리를 위한 **객체 풀(object pool)**과 유사한 **메커니즘**을 사용하여 이러한 효율성을 달성합니다.

RecyclerView 내부 메커니즘의 핵심 개념

- 뷰 재활용 (Recycling Views):** RecyclerView는 데이터 셋의 모든 항목에 대해 새 뷰를 생성하는 대신 기존 뷰를 재사용합니다. 뷰가 보이는 영역 밖으로 스크롤되면 소멸되는 대신 뷰 풀(RecyclerView.RecycledViewPool)에 추가됩니다. 새 항목이 뷰에 들어오면 RecyclerView는 가능한 이 풀에서 기존 뷰를 검색하여 인플레이션 오버헤드를 피합니다.

2. **ViewHolder 패턴**: RecyclerView는 항목 레이아웃 내 뷰에 대한 참조를 저장하기 위해 ViewHolder를 사용합니다. 이는 바인딩 중 반복적인 findViewById() 호출을 방지하여 레이아웃 순회 및 뷰 조회를 줄여 성능을 향상시킵니다.
3. **Adapter의 역할**: RecyclerView.Adapter는 데이터 소스와 RecyclerView를 연결하는 다리 역할을 합니다. 어댑터의 onBindViewHolder() 메서드는 뷰가 재사용될 때 데이터를 뷰에 바인딩하여 보이는 항목만 업데이트되도록 보장합니다.
4. **RecycledViewPool**: RecycledViewPool은 사용되지 않는 뷰가 저장되는 객체 풀 역할을 합니다. 이를 통해 RecyclerView는 유사한 뷰 유형을 가진 여러 목록 또는 섹션에서 뷰를 재사용하여 메모리 사용량을 더욱 최적화할 수 있습니다.

재활용 메커니즘

1. **스크롤 및 항목 가시성**: 사용자가 스크롤하면 뷰 밖으로 나가는 항목은 RecyclerView에서 분리되지만 소멸되지 않습니다. 대신 RecycledViewPool에 추가됩니다.
2. **재활용된 뷰에 데이터 리바인딩**: 새 항목이 뷰에 들어오면 RecyclerView는 먼저 RecycledViewPool에서 필요한 유형의 사용 가능한 뷰를 확인합니다. 일치하는 항목이 발견되면 onBindViewHolder()를 사용하여 새 데이터로 리바인딩하여 뷰를 재사용합니다.
3. **사용 가능한 뷰가 없는 경우 인플레이션**: 풀에 적합한 뷰가 없는 경우 RecyclerView는 onCreateViewHolder()를 사용하여 새 뷰를 인플레이션합니다.
4. **효율적인 메모리 사용**: 뷰를 재활용함으로써 RecyclerView는 메모리 할당 및 가비지 컬렉션을 최소화하여 대규모 데이터 셋이나 빈번한 스크롤이 포함된 시나리오에서 발생할 수 있는 성능 문제를 줄입니다.

다음은 기본적인 RecyclerView 구현 예시입니다.

그림 89. RecyclerView Adapter Example

```
1 class MyAdapter(private val dataList: List<String>) :  
2     → RecyclerView.Adapter<MyAdapter.MyViewHolder>() {  
3  
4     // ViewHolder 클래스: 아이템 뷰의 참조를 저장  
5     class MyViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {  
6         val textView: TextView = itemView.findViewById(R.id.textView)  
7     }  
8  
9     // ViewHolder 생성: 새 뷰 인플레이션  
10    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MyViewHolder {  
11        val view = LayoutInflater.from(parent.context).inflate(R.layout.item_layout,  
12            → parent, false)  
13        return MyViewHolder(view)  
14    }  
15  
16    // ViewHolder에 데이터 바인딩  
17    override fun onBindViewHolder(holder: MyViewHolder, position: Int) {  
18        holder.textView.text = dataList[position]  
19    }  
20  
21    // 데이터 셋 크기 반환  
22    override fun getItemCount(): Int = dataList.size  
23 }
```

RecyclerView의 객체 풀 접근 방식의 장점

1. **향상된 성능**: 뷰를 재사용하면 새 레이아웃 인플레이션 오버헤드가 줄어들어 더 부드러운 스크롤과 더 나은 성능을 제공합니다.
2. **효율적인 메모리 관리**: 객체 풀은 뷰를 재활용하여 메모리 할당을 최소화하고 빈번한 가비지 컬렉션을 방지합니다.
3. **커스텀**: RecycledViewPool은 각 유형에 대한 최대 뷰 수를 관리하도록 커스텀할 수 있어 개발자가 특정 사용 사례에 대한 동작을 최적화할 수 있습니다.

요약

RecyclerView는 사용되지 않는 뷰가 RecycledViewPool에 저장되고 필요할 때 재사용되는 효율적인 객체 풀 메커니즘을 사용합니다. ViewHolder 패턴과 결합된 이 설계는 메모리 사용량을 최소화하고 레이아웃 인플레이션 오버헤드를 줄이며 성능을 향상시켜 RecyclerView를 안드로이드 애플리케이션에서 대규모 데이터 셋을 표시하는 필수 도구로 만듭니다.

실전 질문

- Q) RecyclerView의 ViewHolder 패턴은 ListView와 비교하여 성능 향상 측면에서 어떤 이점이 있나요?
- Q) RecyclerView에서 ViewHolder의 생성부터 재활용까지의 생명주기를 설명해주세요.
- Q) RecycledViewPool이란 무엇이며, 뷰 아이템 렌더링을 최적화하는 데 어떻게 사용할 수 있나요?

 **Pro Tips for Mastery:** 동일한 RecyclerView에서 다른 유형의 아이템을 어떻게 구현하나요?

RecyclerView는 동일한 목록에서 여러 아이템 유형을 지원합니다. 이를 구현하기 위해 커스텀 어댑터, 아이템 뷰 유형 및 적절한 레이아웃을 활용 할 수 있습니다. 핵심은 아이템 유형을 구별하고 올바르게 바인딩하는 것입니다.

여러 아이템 유형 구현 단계

1. **아이템 유형 정의:** 각 아이템 유형은 고유 식별자(일반적으로 정수 상수)로 표시됩니다. 이러한 식별자를 통해 어댑터는 뷰 생성 및 바인딩 중에 아이템 유형을 구별할 수 있습니다.
2. **getItemViewType() 재정의:** 어댑터에서 getItemViewType() 메서드를 재정의하여 데이터 셋의 각 아이템에 대해 적절한 유형을 반환합니다. 이 메서드는 RecyclerView가 인플레이션할 레이아웃 유형을 결정하는 데 도움이 됩니다.
3. **여러 ViewHolder 처리:** 각 아이템 유형에 대해 별도의 ViewHolder 클래스를 생성합니다. 각 ViewHolder는 해당 레이아웃에 데이터를 바인딩하는 역할을 합니다.
4. **뷰 유형에 따라 레이아웃 인플레이션:** onCreateViewHolder() 메서드에서 getItemViewType()에서 반환된 뷰 유형에 따라 적절한 레이아웃을 인플레이션합니다.

5. 적절하게 데이터 바인딩: onBindViewHolder() 메서드에서 아이템 유형을 확인하고 해당 ViewHolder를 사용하여 데이터를 바인딩합니다.

예제: 여러 아이템 유형 구현하기

그림 90. Adapter for Multiple Item Types in RecyclerView

```
1 class MultiTypeAdapter(private val items: List<ListItem>) :  
2     RecyclerView.Adapter<RecyclerView.ViewHolder>() {  
3  
4     // 아이템 유형 상수 정의  
5     companion object {  
6         const val TYPE_HEADER = 0  
7         const val TYPE_CONTENT = 1  
8     }  
9  
10    // 위치에 따른 아이템 유형 반환  
11    override fun getItemViewType(position: Int): Int {  
12        return when (items[position]) {  
13            is ListItem.Header -> TYPE_HEADER  
14            is ListItem.Content -> TYPE_CONTENT  
15        }  
16    }  
17  
18    // 뷰 유형에 따라 ViewHolder 생성  
19    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):  
20        RecyclerView.ViewHolder {  
21        return when (viewType) {  
22            TYPE_HEADER -> {  
23                val view =  
24                    LayoutInflater.from(parent.context).inflate(R.layout.item_header,  
25                        parent, false)  
26                HeaderViewHolder(view)  
27            }  
28        }  
29    }
```

```
24         TYPE_CONTENT -> {
25             val view =
26                 LayoutInflator.from(parent.context).inflate(R.layout.item_content,
27                 parent, false)
28             ContentViewHolder(view)
29         }
30     }
31
32     // ViewHolder에 데이터 바인딩
33     override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position: Int) {
34         when (holder) {
35             is HeaderViewHolder -> holder.bind(items[position] as ListItem.Header)
36             is ContentViewHolder -> holder.bind(items[position] as ListItem.Content)
37         }
38     }
39
40     override fun getItemCount(): Int = items.size
41
42     // 헤더 유형 ViewHolder
43     class HeaderViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
44         private val title: TextView = itemView.findViewById(R.id.headerTitle)
45
46         fun bind(item: ListItem.Header) {
47             title.text = item.title
48         }
49     }
50
51     // 콘텐츠 유형 ViewHolder
52     class ContentViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
53         private val content: TextView = itemView.findViewById(R.id.contentText)
54
55         fun bind(item: ListItem.Content) {
```

```
56         content.text = item.text  
57     }  
58 }  
59 }
```

그림 91. Data Classes Representing Different Item Types

```
1 // 다양한 아이템 유형을 나타내는 데이터 클래스 (Sealed Class 사용)  
2 sealed class ListItem {  
3     data class Header(val title: String) : ListItem()  
4     data class Content(val text: String) : ListItem()  
5 }
```

주요 참고 사항

- 효율성:** RecyclerView는 ViewHolder를 재사용하며 여러 아이템 유형을 처리해도 성능이 저하되지 않습니다. 각 아이템 유형은 getItemViewType() 메서드와 적절한 바인딩을 통해 효율적으로 관리됩니다.
- 명확한 분리:** 각 아이템 유형에는 자체 레이아웃과 ViewHolder가 있어 관심사 분리를 보장하고 코드를 더 깔끔하게 만듭니다.
- 확장성:** 새 아이템 유형을 추가하기 위해서는 최소한의 변경만 해도 됩니다. 새 레이아웃, ViewHolder를 정의하고 getItemViewType() 및 onCreateViewHolder()의 로직을 조정하기만 하면 됩니다.

요약

RecyclerView에서 여러 아이템 유형을 구현하려면 각 아이템의 유형을 결정하는 getItemViewType(), 바인딩을 위한 별도의 ViewHolder, 각 유형에 대한 고유한 레이아웃을 조합하여 사용합니다. 이 접근 방식은 RecyclerView가 통합된 목록에서 다양한 콘텐츠를 지원하면서 효율적이고 확장 가능하게 유지되도록 보장합니다.

💡 Pro Tips for Mastery: RecyclerView의 성능을 어떻게 향상시키나요?

ListAdapter⁷⁴와 **DiffUtil**⁷⁵을 활용하여 RecyclerView의 성능을 향상시킬 수 있습니다. DiffUtil은 두 목록 간의 차이를 계산하고 RecyclerView 어댑터를 효율적으로 업데이트하는 안드로이드 유ти리티 클래스입니다. DiffUtil을 활용하면 목록의 모든 항목을 비효율적으로 다시 렌더링할 수 있는 notifyDataSetChanged()를 불필요하게 호출하는 것을 피할 수 있습니다. 대신 DiffUtil은 세분화된 수준에서 변경 사항을 식별하고 영향을 받는 항목만 업데이트합니다.

RecyclerView의 기본 동작은 대부분의 항목이 변경되지 않은 경우에도 데이터가 변경될 때 보이는 모든 항목을 리바인딩하고 다시 그립니다. 이는 특히 대규모 데이터 셋에서 성능을 저하시킬 수 있습니다. DiffUtil은 필요한 최소 업데이트 세트(삽입, 삭제 및 수정)를 계산하고 이를 어댑터에 직접 적용하여 이를 최적화합니다.

DiffUtil 사용 단계

- DiffUtil 콜백 생성:** 커스텀 DiffUtil.ItemCallback을 구현하거나 DiffUtil.Callback을 상속받습니다. 이 클래스는 이전 목록과 새 목록 간의 차이 계산을 어떻게 할지 정의합니다.
- 어댑터에 목록 업데이트 제공:** 새 데이터가 도착하면 어댑터에 전달하고 DiffUtil을 사용하여 차이를 계산합니다. ListAdapter를 사용하는 경우 submitList() 메서드를 사용하거나 커스텀 어댑터의 경우 notifyItemChanged()와 같은 메서드를 사용하여 이러한 변경 사항을 어댑터에 적용합니다.
- RecyclerView 어댑터와 DiffUtil 바인딩:** DiffUtil을 어댑터에 통합하여 업데이트를 자동으로 처리합니다.

예제: RecyclerView와 DiffUtil 구현하기

⁷⁴[https://developer.android.com/reference/androidx/recyclerview/widgetListAdapter](https://developer.android.com/reference/androidx/recyclerview/widget/ListAdapter)

⁷⁵<https://developer.android.com/reference/androidx/recyclerview/widget/DiffUtil>

그림 92. DiffUtil Callback Implementation

```
1 class MyDiffUtilCallback : DiffUtil.ItemCallback<MyItem>() {
2     override fun areItemsTheSame(oldItem: MyItem, newItem: MyItem): Boolean {
3         // 아이템이 동일한 데이터를 나타내는지 확인
4         return oldItem.id == newItem.id
5     }
6
7     override fun areContentsTheSame(oldItem: MyItem, newItem: MyItem): Boolean {
8         // 아이템의 내용이 동일한지 확인
9         return oldItem == newItem
10    }
11 }
```

그림 93. Adapter Using ListAdapter with DiffUtil

```
1 class MyAdapter : ListAdapter<MyItem, MyViewHolder>(MyDiffUtilCallback()) {
2
3     override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MyViewHolder {
4         val view = LayoutInflater.from(parent.context).inflate(R.layout.item_layout,
5             parent, false)
6         return MyViewHolder(view)
7     }
8
9     override fun onBindViewHolder(holder: MyViewHolder, position: Int) {
10        holder.bind(getItem(position))
11    }
12
13 class MyViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
14     private val textView: TextView = itemView.findViewById(R.id.textView)
15
16     fun bind(item: MyItem) {
17         textView.text = item.name
18     }
19 }
```

```
18     }  
19 }
```

그림 94. Submitting Data to the Adapter

```
1 val adapter = MyAdapter()  
2 recyclerView.adapter = adapter  
3  
4 val oldList = listOf(MyItem(1, "Old Item"), MyItem(2, "Another Item"))  
5 val newList = listOf(MyItem(1, "Updated Item"), MyItem(3, "New Item"))  
6  
7 // 자동으로 아이템의 다른 부분을 계산하고 필요한 부분만 RecyclerView에서 업데이트  
8 adapter.submitList(newList)
```

DiffUtil의 주요 이점

1. **향상된 성능**: 전체 목록을 새로 고치는 대신 수정된 항목만 업데이트하여 렌더링 오버헤드를 줄입니다.
2. **세분화된 업데이트**: 항목 삽입, 삭제 및 수정을 개별적으로 처리하여 애니메이션을 더 부드럽고 자연스럽게 만듭니다.
3. **ListAdapter와의 원활한 통합**: ListAdapter는 DiffUtil을 내부적으로 구현하고 있는 안드로이드 Jetpack 라이브러리의 어댑터로, 보일러 플레이트 코드를 줄입니다.

고려 사항

1. **대규모 목록에 대한 오버헤드**: DiffUtil은 효율적이지만 매우 큰 두 목록 간에 차이점이 많은 경우 오히려 더 많은 오버헤드를 만들 수 있습니다. 따라서 상황에 맞게 신중하게 사용해야 합니다.
2. **불변 데이터**: 데이터 모델이 불변(immutable)인지 확인합니다. 가변 데이터는 DiffUtil이 변경 사항을 계산하려고 할 때 불일치를 유발할 수 있습니다.

요약

RecyclerView와 함께 DiffUtil을 사용하면 전체 목록을 다시 렌더링 하는 대신 필요한 업데이트만 적용하여 성능을 향상시킵니다. DiffUtil.ItemCallback 및 ListAdapter를 사용하면 업데이트를 효율적으로 관리하고 더 부드러운 애니메이션을 만들며 UI의 전반적인 응답성을 향상시킬 수 있습니다. 이 접근 방식은 자주 변경되는 데이터 셋을 처리하는 경우 특히 유용합니다.

Q) 42. Dp와 Sp의 차이점은 무엇인가요?

안드로이드 사용자 인터페이스를 디자인할 때 UI 컴포넌트가 다양한 화면 크기와 해상도에 어떻게 적응하는지 고려해야 합니다. 이를 위해 사용되는 필수적인 개념에는 **Dp (Density-independent Pixels)** 와 **Sp (Scale-independent Pixels)** 가 있습니다. 둘 다 기기 간 일관성을 보장하는 데 도움이 되지만 다른 목적을 가지고 있습니다.

Dp란 무엇인가?

Dp (Density-independent Pixels) 는 패딩, 마진, 너비와 같은 UI 요소의 측정 단위입니다. 다양한 화면 밀도를 가진 기기에서 UI 컴포넌트의 일관된 물리적 크기를 제공하도록 설계되었습니다. 1 Dp는 160 DPI(인치당 도트 수) 화면의 물리적 픽셀 1개와 같으며, 안드로이드는 기기 밀도에 맞게 Dp를 자동으로 조절합니다.

예를 들어, Button 너비를 100dp로 지정하면 저밀도 및 고밀도 화면 모두에서 대략 동일한 크기로 표시되지만 이를 렌더링하는 데 필요한 픽셀 수는 다릅니다.

Sp란 무엇인가?

Sp (Scale-independent Pixels) 는 텍스트 크기에만 사용됩니다. Dp와 유사하게 작동하지만 사용자의 글꼴 크기 환경 설정을 추가로 고려합니다. 즉, Sp는 화면 밀도와 기기의 접근성 설정 모두를 기반으로 텍스트 크기를 조절하므로 읽기 쉽고 접근 가능한 텍스트를 보장하는데 이상적입니다.

예를 들어, TextView를 16sp로 설정하면 화면 밀도에 맞게 적절하게 크기가 조절되고 사용자가 시스템 글꼴 크기를 늘린 경우에도 조정됩니다.

Dp와 Sp의 주요 차이점

주요 차이점은 크기 조절 동작입니다.

- 목적:** Dp는 크기(예: 버튼 크기, 패딩)에 사용하고 Sp는 텍스트 크기에 사용합니다.
- 사용자 정의 환경:** Sp는 사용자가 정의한 글꼴 크기 환경 설정을 존중하지만 Dp는 그렇지 않습니다.
- 밀도 보장성:** 둘 다 화면 밀도에 따라 크기가 조절되지만 Sp는 모든 사용자가 텍스트를 접근 가능하고 읽을 수 있도록 보장합니다.

요약

기기 간 일관된 레이아웃을 유지하려면 View 크기, 마진, 패딩과 같은 UI 컴포넌트에 Dp를 사용해야 합니다. 텍스트의 경우, 시각적 일관성을 유지하면서 사용자 환경 설정을 존중하기 위해 Sp를 사용합니다. 특히 텍스트 개발에서, 각 차이점을 올바르게 이해하고 사용함으로써 시각적으로 안정성이 보장되고 접근성을 올바르게 지원하는 안드로이드 앱을 만들 수 있습니다.

Pro Tips for Mastery: Sp 단위를 사용할 때 화면 깨짐을 어떻게 처리하나요?

Sp (Scale-independent Pixels) 를 사용하는 것은 화면 밀도와 사용자 글꼴 환경 설정에 따라 크기가 조절되므로 안드로이드에서 텍스트 접근성을 보장하는 데 중요합니다. 그러나 사용자가 정의한 큰 글꼴 크기로 인한 과도한 크기 조절은 UI 컴포넌트가 겹치거나 화면을 벗어나는 레이아웃 깨짐 문제로 이어질 수 있습니다. 사용자 친화적인 경험을 유지하려면 이러한 시나리오를 적절하게 처리하는 것이 필수적입니다.

화면 깨짐 방지 전략

사용자가 시스템 글꼴 크기를 크게 늘리면 Sp 단위로 인해 텍스트 요소가 의도한 경계를 넘어 커질 수 있습니다. 이는 특히 버튼, 레이블 또는 컴팩트한 화면과 같이 제한된 공간에서 레이아웃을 깨뜨릴 수 있습니다.

1. 콘텐츠를 적절하게 감싸기 (Wrap Content Properly)

TextView나 Button과 같은 텍스트 기반 컴포넌트의 크기가 wrap_content로 설정되었는지 확인합니다. 이렇게 하면 컨테이너가 텍스트 크기에 따라 동적으로 확장되어 텍스트 잘림이나 오버플로우를 피할 수 있습니다.

```
1 <TextView  
2     android:layout_width="wrap_content"  
3     android:layout_height="wrap_content"  
4     android:textSize="16sp"  
5     android:text="샘플 텍스트" />
```

2. TextView에 minLines 또는 maxLines 사용하기

텍스트 확장 동작을 제어하려면 `minLines` 및 `maxLines` 속성을 사용하여 레이아웃을 방해하지 않고 텍스트가 읽기 쉽게 유지되도록 합니다. 오버플로우를 올바르게 처리하려면 `ellipsize`와 결합합니다.

```
1 <TextView  
2     android:layout_width="match_parent"  
3     android:layout_height="wrap_content"  
4     android:textSize="16sp"  
5     android:maxLines="2"  
6     android:ellipsize="end"  
7     android:text="이것은 제대로 처리되지 않으면 레이아웃을 깨뜨릴 수 있는 긴 샘플  
→ 텍스트입니다." />
```

3. 중요한 UI 컴포넌트에 고정 크기 사용하기

일관된 크기가 필수적인 경우 버튼과 같은 중요한 컴포넌트에 `Dp` 사용을 고려합니다. 이렇게 하면 사용자 환경 설정에 의한 텍스트 크기 조절에 관계없이 컴포넌트 크기가 안정적으로 유지됩니다. 또는, 컴포넌트 깨짐을 최소화하기 위해 중요한 컴포넌트 내의 텍스트 크기에는 `Sp`를 신중하게 사용하는 것이 안전합니다.

```
1 <Button  
2     android:layout_width="100dp"  
3     android:layout_height="50dp"  
4     android:textSize="14sp"  
5     android:text="버튼" />
```

4. 극단적인 글꼴 크기로 테스트하기

항상 기기 설정에서 사용할 수 있는 가장 큰 시스템 글꼴 크기로 앱을 테스트합니다. 깨지거나 겹치는 UI 컴포넌트를 식별하고 더 큰 텍스트를 올바르게 반영할 수 있도록 레이아웃을 다듬습니다.

5. 제약 조건을 사용한 동적 크기 조절 고려하기

ConstraintLayout을 사용하여 컴포넌트 위치 지정 및 크기 조절에 유연성을 더합니다. 텍스트가 확장될 때도 다른 UI 요소와 겹치지 않도록 텍스트 요소에 대한 제약 조건을 정의합니다.

```
1 <androidx.constraintlayout.widget.ConstraintLayout  
2     xmlns:android="http://schemas.android.com/apk/res/android"  
3     xmlns:app="http://schemas.android.com/apk/res-auto"  
4     android:layout_width="match_parent"  
5     android:layout_height="wrap_content">  
6  
7     <TextView  
8         android:id="@+id/sampleText"  
9         android:layout_width="0dp"  
10        android:layout_height="wrap_content"  
11        android:textSize="16sp"  
12        app:layout_constraintStart_toStartOf="parent"  
13        app:layout_constraintEnd_toEndOf="parent"  
14        app:layout_constraintTop_toTopOf="parent"  
15        android:text="동적 레이아웃 예제" />  
16 </androidx.constraintlayout.widget.ConstraintLayout>
```

6. Sp 대신 Dp 크기 사용하기

이는 팀의 접근 방식에 따라 호불호가 갈릴 수 있습니다. 일부 회사는 사용자 조정 글꼴 크기로 인한 레이아웃 문제를 방지하기 위해 텍스트 크기에 **Sp** 대신 **Dp**를 사용하는 경우도 더러 있습니다. 그러나 **Sp**는 접근성을 지원하도록 따로 설계되어 사용자의 가독성 환경 설정에 따라 텍스트가 조정되도록 보장하기 때문에, **Sp** 대신 **Dp**를 사용하는 접근 방식은 사용자 경험을 저해할 수 있습니다.

요약

Sp로 인한 화면 깨짐을 처리하려면 `wrap_content` 사용, 제약 조건 설정, 텍스트 확장 제한 정의와 같은 모범 사례를 결합할 수 있습니다. 극단적인 글꼴 크기로 테스트하고 레이아웃을 동적으로 관리하면 앱이 시각적으로 일관되고 모든 사용자에게 접근 가능하도록 보장할 수 있습니다.

실전 질문

Q) 텍스트 크기에 **Sp**를 사용할 때 발생할 수 있는 잠재적인 레이아웃 깨짐 문제를 경험해 보거나 해결해 본 적이 있나요?

Q) 43. 나인패치(nine-patch) 이미지의 용도는 무엇인가요?

나인패치(Nine-Patch) 이미지는 시각적 품질을 잃지 않고 늘리거나 크기를 조절할 수 있는 특수 형식의 PNG 이미지로, 안드로이드에서 유연하고 적응 가능한 UI 컴포넌트를 만드는 데 필수적입니다. 주로 버튼, 배경, 컨테이너와 같이 다양한 화면 크기 및 콘텐츠 크기에 맞게 동적으로 크기를 조절해야 하는 요소에 사용됩니다.

나인패치 이미지의 주요 특징

1. **늘어나는 영역(Stretchable Areas)**: 나인패치 이미지는 이미지의 나머지 부분의 무결성을 유지하면서 늘릴 수 있는 영역을 정의합니다. 이는 이미지의 가장 바깥쪽 1픽셀 테두리에 있는 검은색 선(가이드)을 사용하여 정의할 수 있습니다.
2. **콘텐츠 영역 정의(Content Area Definition)**: 검은색 선은 이미지 내부에서 늘어날 수 있는 콘텐츠 영역을 지정하여 이미지 내 텍스트 또는 기타 UI 요소의 적절한 정렬을 보장합니다.

3. **동적 크기 조절(Dynamic Resizing)**: 비례적으로 크기가 조절되어 다양한 화면 크기를 가진 기기에서도 UI 컴포넌트가 깨지지 않고 모양을 유지하도록 보장합니다.

XML에서의 사용 예제

아래 코드는 나인패치 이미지를 버튼의 배경으로 사용하는 방법을 보여줍니다.

그림 95. button_layout.xml

```
1 <Button  
2     android:layout_width="wrap_content"  
3     android:layout_height="wrap_content"  
4     android:background="@drawable/button_background.9.png" <!-- 파일 이름에 .9 포함 -->  
5     android:text="Click Me" />
```

나인패치 이미지의 한계

- 수동 생성**: 적절한 크기 조절 및 정렬을 보장하기 위해 수동적으로 이미지 리소스를 생성해야하고 실제로 잘 동작하는지 테스트가 필요합니다.
- 제한된 사용 사례**: 직사각형 또는 정사각형 형태의 요소(가령, 채팅 말풍선 등)에 가장 적합하며, 복잡하거나 불규칙한 모양에는 덜 효과적입니다.

요약

나인패치 이미지는 안드로이드에서 확장 가능하고 시각적으로 일관된 UI 컴포넌트를 만들기 위한 유연하고 효율적인 솔루션입니다. 늘어나는 영역과 콘텐츠 영역을 정의함으로써 버튼 및 배경과 같은 요소가 세련된 모양을 유지하면서 다양한 화면 크기 및 동적 콘텐츠에 원활하게 적응하도록 보장합니다.

실전 질문

Q) 나인패치 이미지는 일반 PNG 이미지와 어떻게 다르며, 어떤 시나리오에서 나인패치 이미지를 사용해야 하나요?

Q) 44. Drawable이란 무엇이며, UI 개발에서 어떻게 사용되나요?

Drawable은 화면에 그릴 수 있는 모든 것에 대한 추상화 개념입니다. 이미지, 벡터 그래픽, 특정 모양 기반의 요소와 같은 다양한 유형의 그래픽 콘텐츠의 기본 클래스 역할을 합니다. Drawable은 배경, 버튼, 아이콘, 커스텀 뷰를 포함한 UI 컴포넌트에서 널리 사용됩니다.

안드로이드는 특정 사용 사례에 맞게 설계된 다양한 유형의 Drawable 객체를 제공합니다.

1. BitmapDrawable (래스터 이미지)

BitmapDrawable은 PNG, JPG 또는 GIF와 같은 래스터 이미지를 표시하는 데 사용됩니다. 비트맵 이미지의 크기 조절, 타일링 및 필터링을 허용합니다.

그림 96. bitmap_drawable.xml

```
1 <bitmap xmlns:android="http://schemas.android.com/apk/res/android"
2     android:src="@drawable/sample_image"
3     android:tileMode="repeat"/>
```

이는 일반적으로 ImageView 컴포넌트에서 이미지 표시 또는 배경으로 사용됩니다.

2. VectorDrawable (확장 가능한 벡터 그래픽)

VectorDrawable은 XML 경로를 사용하여 확장 가능한 벡터 그래픽(SVG 유사)을 나타냅니다. 비트맵과 달리 벡터는 어떤 해상도에서도 품질을 유지합니다.

그림 97. vector_drawable.xml

```
1 <vector xmlns:android="http://schemas.android.com/apk/res/android"
2     android:width="24dp"
3     android:height="24dp"
4     android:viewportWidth="24"
5     android:viewportHeight="24">
6     <path
7         android:fillColor="#FF0000"
8         android:pathData="M12,2L15,8H9L12,2Z"/> <!-- 예시 삼각형 경로 -->
9 </vector>
```

VectorDrawable은 아이콘, 로고 및 확장 가능한 UI 요소에 이상적이며 다양한 화면 밀도에서 픽셀화 문제를 방지합니다.

3. NinePatchDrawable (패딩이 있는 크기 조절 가능 이미지)

NinePatchDrawable은 모서리나 패딩과 같은 특정 영역을 보존하면서 크기를 조절할 수 있는 특수 유형의 비트맵입니다. 채팅 말풍선 및 버튼과 같이 특정 영역에 있어서 늘어나는 UI 컴포넌트를 만드는 데 유용합니다.

나인패치 이미지(.9.png)에는 늘어나는 영역과 고정 영역을 정의하는 추가 1픽셀 테두리가 포함됩니다.

그림 98. Example of NinePatchDrawable

```
1 <nine-patch xmlns:android="http://schemas.android.com/apk/res/android"
2     android:src="@drawable/chat_bubble.9.png"/>
```

.9.png를 만들려면 Android Studio의 나인패치 도구를 사용하여 늘어나는 영역을 정의할 수 있습니다.

4. ShapeDrawable (커스텀 모양)

ShapeDrawable은 XML에서 정의되며 이미지를 사용하지 않고 등근 사각형, 타원 또는 기타 단순한 모양을 만드는 데 사용할 수 있습니다.

그림 99. shape_drawable.xml

```
1 <shape xmlns:android="http://schemas.android.com/apk/res/android"
2     android:shape="rectangle">
3     <solid android:color="#FF5733"/>
4     <corners android:radius="8dp"/>
5 </shape>
```

ShapeDrawable은 버튼, 배경 및 커스텀 UI 컴포넌트에 유용합니다.

5. LayerDrawable (여러 Drawable 쌓기)

LayerDrawable은 여러 Drawable을 단일 계층 구조로 결합하는 데 사용되며 복잡한 UI 배경에 유용합니다.

그림 100. layer_drawable.xml

```
1 <layer-list xmlns:android="http://schemas.android.com/apk/res/android">
2   <item>
3     <shape android:shape="rectangle">
4       <solid android:color="#000000"/>
5     </shape>
6   </item>
7   <item android:drawable="@drawable/icon" android:top="10dp" android:left="10dp"/>
8     ← <!-- 위치 조정 가능 -->
</layer-list>
```

이는 오버레이 효과 및 쌓인 시각 효과를 만드는 데 유용합니다.

요약

Drawable 클래스는 안드로이드에서 다양한 유형의 그래픽을 처리하는 유연한 방법을 제공합니다. 올바른 Drawable을 선택하는 것은 디자인 요구 사항, 확장성 및 UI 복잡성과 같은 사용 사례에 따라 달라집니다. 이러한 다양한 Drawable 유형을 활용하여 개발자는 안드로이드 애플리케이션에서 최적화되고 시각적으로 매력적인 UI 컴포넌트를 만들 수 있습니다.

실전 질문

Q) Drawable만 사용하여 사용자 상호 작용에 따라 모양과 색상이 변경되는 동적 배경을 가진 버튼을 어떻게 만들 수 있나요?

Q) 45. 안드로이드의 Bitmap이란 무엇이며, 큰 Bitmap을 효율적으로 처리하는 방법은 무엇인가요?

Bitmap은 메모리 내 이미지 표현입니다. 픽셀 데이터를 저장하며 리소스, 파일 또는 원격 소스에서 가져온 이미지를 화면에 렌더링하는 데 자주 사용됩니다. 비트맵 객체는 특히 고해상도 이미지의 경우 대량의 픽셀 데이터를 보유하므로 부적절하게 처리하면 메모리 고갈 및 OutOfMemoryError가 쉽게 발생할 수 있습니다.

큰 Bitmap의 문제점

카메라나 인터넷에서 다운로드한 이미지와 같은 많은 이미지는 이를 표시하는 UI 컴포넌트에서 요구하는 크기보다 훨씬 큽니다. 이러한 전체 해상도 이미지를 불필요하게 로드하면 다음과 같은 문제가 발생합니다.

- 과도한 메모리를 소비합니다.
- 성능 오버헤드를 유발합니다.
- 메모리 압박으로 인해 크래시가 발생할 위험이 있습니다.

메모리를 할당하지 않고 Bitmap 크기 읽기

비트맵을 로드하기 전에 전체 로드가 정당한지 결정하기 위해 **크기를 검사**하는 것이 중요합니다. `BitmapFactory.Options` 클래스를 사용하면 `inJustDecodeBounds = true`로 설정하여 픽셀 데이터에 대한 메모리 할당을 피하면서 이미지 메타데이터를 디코딩할 수 있습니다.

그림 101. `ReadBitmapDimensions.kt`

```
1 val options = BitmapFactory.Options().apply {
2     inJustDecodeBounds = true
3 }
4 BitmapFactory.decodeResource(resources, R.drawable.myimage, options) // R.id ->
→ R.drawable 설정
5
6 val imageWidth = options.outWidth
7 val imageHeight = options.outHeight
8 val imageType = options.outMimeType
```

이 단계는 이미지 크기가 표시 요구 사항과 일치하는지 평가하여 불필요한 메모리 할당을 방지하는 데 도움이 됩니다.

샘플링을 사용하여 축소된 Bitmap 로드하기

크기를 알게 되면 `inSampleSize` 옵션을 사용하여 비트맵을 대상 크기에 맞게 축소할 수 있습니다. 이는 이미지를 2, 4 등의 배수로 서브샘플링하여 메모리 사용량을 줄입니다. 예를 들어, `inSampleSize = 4`로 로드된 2048×1536 이미지는 512×384 비트맵이 됩니다.

그림 102. CalculateInSampleSize.kt

```
1 fun calculateInSampleSize(options: BitmapFactory.Options, reqWidth: Int, reqHeight: Int): Int {
2     // 이미지의 원본 높이와 너비
3     val (height, width) = options.run { outHeight to outWidth }
4     var inSampleSize = 1
5
6     if (height > reqHeight || width > reqWidth) {
7         val halfHeight = height / 2
8         val halfWidth = width / 2
9
10        // 요청된 너비와 높이보다 크거나 같을 때까지 inSampleSize를 2배씩 늘립니다.
11        while (halfHeight / inSampleSize >= reqHeight && halfWidth / inSampleSize >=
12             reqWidth) {
13            inSampleSize *= 2
14        }
15    }
16    return inSampleSize
}
```

이는 이미지 품질과 메모리 효율성의 균형을 맞추는 데 도움이 됩니다.

서브샘플링을 사용한 전체 디코딩 프로세스

calculateInSampleSize를 사용하여 두 단계로 비트맵을 디코딩할 수 있습니다.

1. 경계(bounds)만 디코딩합니다.
2. 계산된 inSampleSize를 설정하고 축소된 비트맵을 디코딩합니다.

그림 103. DecodeSampledBitmap.kt

```
1 fun decodeSampledBitmapFromResource(
2     res: Resources,
3     resourceId: Int,
4     reqWidth: Int,
5     reqHeight: Int
6 ): Bitmap? { // Bitmap? 반환 타입 변경 (디코딩 실패 가능성)
7     // 먼저 inJustDecodeBounds=true로 크기 확인
8     return BitmapFactory.Options().run {
9         inJustDecodeBounds = true
10        BitmapFactory.decodeResource(res, resourceId, this)
11
12        // inSampleSize 계산
13        inSampleSize = calculateInSampleSize(this, reqWidth, reqHeight)
14
15        // inSampleSize를 설정하고 비트맵 디코딩
16        inJustDecodeBounds = false
17        BitmapFactory.decodeResource(res, resourceId, this)
18    }
19 }
```

이를 ImageView와 함께 사용하려면 간단히 호출합니다.

그림 104. SetImageBitmap.kt

```
1 // ImageView 크기에 맞게 reqWidth, reqHeight 설정
2 val bitmap = decodeSampledBitmapFromResource(resources, R.drawable.myimage, 100, 100)
3 imageView.setImageBitmap(bitmap)
```

이 접근 방식은 UI가 최적화된 메모리 사용량으로 적절한 크기의 이미지를 얻도록 보장합니다.

요약

Bitmap은 안드로이드에서 메모리를 많이 사용하는 이미지 표현입니다. 성능 문제와 크래시를 피하려면 먼저 `inJustDecodeBounds`를 사용하여 이미지 크기를 검사한 다음, `inSampleSize`로 큰 비트맵을 서브샘플링하여 필요한 만큼만 로드하고, 마지막으로 크기 조정을 위한 2단계 전략을 사용하여 효율적으로 디코딩하는 것이 중요합니다. 해당 프로세스는 메모리가 제한된 기기에서 이미지 집약적인 애플리케이션을 견고하게 구축하는 데 많은 도움이 됩니다.

실전 질문

Q) 큰 Bitmap을 메모리에 로드하는 것은 어떤 위험성이 있으며, 어떻게 효율적으로 처리할 수 있나요?

 **Pro Tips for Mastery:** 커스텀 이미지 로딩 시스템에서 큰 비트맵 캐싱을 어떻게 구현하나요?

특히 이미지 목록, 그리드 또는 캐러셀을 처리할 때 원활하고 메모리 안전한 안드로이드 애플리케이션을 구축하려면 큰 비트맵을 효율적으로 관리하는 것이 필수적입니다. 안드로이드는 두 가지 효과적인 전략을 제공합니다. `LruCache`를 사용한 **메모리 내 캐싱**과 `DiskLruCache`⁷⁶를 사용한 **디스크 기반 캐싱**이 존재합니다.

LruCache를 사용한 메모리 내 캐싱

`LruCache`는 Bitmap을 위한 선호되는 메모리 내 캐싱 솔루션입니다. 최근에 사용된 항목에 대한 강력한 참조를 유지하고 메모리가 부족할 때 가장 최근에 사용되지 않은 항목을 자동으로 제거합니다. 작동 방식은 다음과 같습니다.

⁷⁶<https://github.com/JakeWharton/DiskLruCache>

그림 105. MemoryCacheInit.kt

```
1 object LruCacheManager {
2     // 사용 가능한 최대 메모리 (KB)
3     val maxMemory = (Runtime.getRuntime().maxMemory() / 1024).toInt()
4     // 캐시 크기를 최대 메모리의 1/8로 설정
5     val cacheSize = maxMemory / 8
6
7     val memoryCache = object : LruCache<String, Bitmap>(cacheSize) {
8         // 캐시 항목 크기를 KB 단위로 정의
9         override fun sizeOf(key: String, bitmap: Bitmap): Int {
10             // 비트맵의 바이트 수를 KB로 변환
11             return bitmap.byteCount / 1024
12         }
13     }
14 }
```

메모리 오버플로우 위험을 방지하기 위해 사용 가능한 메모리의 약 1/8을 할당합니다. 이 설정은 이미지에 빠르게 접근하고 중복 디코딩을 피합니다. 사용 방법은 다음과 같습니다.

그림 106. MemoryCacheUsage.kt

```
1 fun loadBitmap(imageId: Int, imageView: ImageView) {
2     val key = imageId.toString()
3     // 메모리 캐시에서 먼저 확인
4     LruCacheManager.memoryCache.get(key)?.let {
5         imageView.setImageBitmap(it)
6     } ?: run {
7         // 캐시에 없으면 플레이스홀더 설정 및 백그라운드 작업 요청
8         imageView.setImageResource(R.drawable.image_placeholder)
9
10        val workRequest = OneTimeWorkRequestBuilder<BitmapDecodeWorker>()
11            .setInputData(workDataOf("imageId" to imageId))
12            .build()
13        WorkManager.getInstance(context).enqueue(workRequest)
14 }
```

```
14     }
15 }
```

그리고 Jetpack의 커스텀 Worker에서는 아래와 같이 처리할 수 있습니다. (Jetpack의 WorkManager는 이 챕터의 후반부에 다룰 예정입니다)

그림 107. BitmapWorkerTask.kt

```
1 class BitmapDecodeWorker(
2     context: Context,
3     workerParams: WorkerParameters
4 ) : CoroutineWorker(context, workerParams) {
5
6     override suspend fun doWork(): Result {
7         val imageId = inputData.getInt("imageId", -1)
8         if (imageId == -1) return Result.failure()
9
10        // 리소스에서 샘플링된 비트맵 디코딩
11        val bitmap = decodeSampledBitmapFromResource(
12            res = applicationContext.resources,
13            resId = imageId,
14            reqWidth = 100, // 대상 너비
15            reqHeight = 100 // 대상 높이
16        )
17
18        // 비트맵을 메모리 캐시에 추가
19        bitmap?.let {
20            LruCacheManager.memoryCache.put(imageId.toString(), it)
21            return Result.success()
22        }
23
24        return Result.failure()
25    }
26 }
```

여기서 메모리 핸들링 로직에 SoftReference 또는 WeakReference를 사용하지 않았는데, 이는 공격적인 가비지 컬렉션으로 인해 캐싱값 자체를 더 이상 신뢰할 수 없는 상황이 발생할 수 있기 때문입니다.

DiskLruCache를 사용한 디스크 캐싱

안드로이드에서 메모리는 제한적이고 휘발성이라는 특성을 갖습니다. 비트맵이 앱 세션 간에 지속되고 재계산을 피하도록 보장하려면 [DiskLruCache](#)⁷⁷ 라이브러리를 사용하여 비트맵을 디스크에 저장할 수 있습니다. 이는 특히 리소스 집약적인 이미지나 스크롤 가능한 이미지 목록을 처리할 때 유용합니다.

먼저, 디코딩된 비트맵을 유지하기 위해 안전한 해싱 및 I/O 로직으로 DiskLruCache를 래핑하는 DiskCacheManager를 작성할 수 있습니다.

그림 108. DiskCacheManager.kt

```
1 import android.content.Context
2 import android.graphics.Bitmap
3 import android.graphics.BitmapFactory
4 import com.jakewharton.disklrucache.DiskLruCache
5 import java.io.File
6 import java.io.IOException
7 import java.io.OutputStream
8 import java.security.MessageDigest
9 import java.security.NoSuchAlgorithmException
10
11 class DiskCacheManager(context: Context, cacheDirName: String = "images", cacheSize:
12     → Long = 10 * 1024 * 1024) {
13
14     private var diskLruCache: DiskLruCache? = null
15     private val lock = Any()
16
17     init {
18         val cacheDir = getDiskCacheDir(context, cacheDirName)
19         if (!cacheDir.exists()) {
```

⁷⁷<https://github.com/JakeWharton/DiskLruCache>

```
19         cacheDir.mkdirs()
20     }
21     try {
22         // DiskLruCache 열기 (앱 버전, 값 개수, 최대 크기)
23         diskLruCache = DiskLruCache.open(cacheDir, 1, 1, cacheSize)
24     } catch (e: IOException) {
25         e.printStackTrace()
26     }
27 }
28
29 private fun getDiskCacheDir(context: Context, uniqueName: String): File {
30     val cachePath = context.cacheDir.path
31     return File(cachePath + File.separator + uniqueName)
32 }
33
34 // 키를 안전한 파일 이름으로 변환 (SHA-1 사용)
35 private fun filenameForKey(key: String): String {
36     return try {
37         val messageDigest = MessageDigest.getInstance("SHA-1")
38         messageDigest.update(key.toByteArray())
39         bytesToHexString(messageDigest.digest())
40     } catch (e: NoSuchAlgorithmException) {
41         key.hashCode().toString() // fallback
42     }
43 }
44
45 private fun bytesToHexString(bytes: ByteArray): String {
46     val sb = StringBuilder()
47     for (b in bytes) {
48         val hex = Integer.toHexString(0xFF and b.toInt())
49         if (hex.length == 1) {
50             sb.append('0')
51         }
52         sb.append(hex)
```

```
53         }
54     return sb.toString()
55 }
56
57 // 디스크 캐시에서 비트맵 가져오기
58 fun get(key: String): Bitmap? {
59     synchronized(lock) {
60         val safeKey = filenameForKey(key)
61         var snapshot: DiskLruCache.Snapshot? = null
62         return try {
63             snapshot = diskLruCache?.get(safeKey)
64             snapshot?.InputStream(0)?.use { inputStream ->
65                 BitmapFactory.decodeStream(inputStream)
66             }
67         } catch (e: IOException) {
68             e.printStackTrace()
69             null
70         } finally {
71             snapshot?.close()
72         }
73     }
74 }
75
76 // 디스크 캐시에 비트맵 설정하기
77 fun set(key: String, bitmap: Bitmap) {
78     synchronized(lock) {
79         val safeKey = filenameForKey(key)
80         var editor: DiskLruCache.Editor? = null
81         try {
82             editor = diskLruCache?.edit(safeKey)
83             if (editor != null) {
84                 editor.newOutputStream(0).use { outputStream ->
85                     bitmap.compress(Bitmap.CompressFormat.JPEG, 100, outputStream)
86                     editor.commit()
87                 }
88             }
89         } finally {
90             editor?.close()
91         }
92     }
93 }
```

```
87             }
88         } else {
89             diskLruCache?.flush() // editor가 null이면 flush 시도
90         }
91     } catch (e: IOException) {
92         e.printStackTrace()
93         try {
94             editor?.abort()
95         } catch (ignored: IOException) {}
96     }
97 }
98 }
99 }
```

이 클래스는 다음을 보장합니다.

- 디스크 안전한 SHA-1 기반 파일 이름 생성
- 안전한 I/O 작업
- 디스크 캐시에 중복으로 데이터를 쓰는 행위 방지

다음으로, Jetpack WorkManager의 CoroutineWorker를 사용하여 메인 스레드 외부에서 디스크 캐싱을 수행하고 메모리 및 디스크 전략을 안전하게 결합합니다.

그림 109. BitmapDiskWorker.kt

```
1 import androidx.work.*
2 import android.content.Context
3 import android.graphics.Bitmap
4 // DiskCacheManager 및 LruCacheManager 필요
5
6 class BitmapWorker(
7     private val context: Context,
8     workerParams: WorkerParameters
9 ) : CoroutineWorker(context, workerParams) {
```

```
10 // DiskCacheManager 인스턴스는 싱글톤 또는 의존성 주입으로 관리해야 합니다.
11 private val diskCacheManager = DiskCacheManager(context)
12
13 override suspend fun doWork(): Result {
14     val key = inputData.getString("imageKey") ?: return Result.failure()
15     val resId = inputData.getInt("resId", -1)
16     if (resId == -1) return Result.failure()
17
18     // 메모리 캐시 먼저 확인
19     LruCacheManager.memoryCache.get(key)?.let {
20         // 메모리 캐시에 이미 있으면 성공
21         return Result.success()
22     }
23
24     // 디스크 캐시 확인
25     diskCacheManager.get(key)?.let { bitmapFromDisk ->
26         // 디스크 캐시에 있으면 메모리 캐시에 추가하고 성공
27         LruCacheManager.memoryCache.put(key, bitmapFromDisk)
28         return Result.success()
29     }
30
31     // 캐시에 없으면 디코딩 및 캐싱
32     val bitmap = decodeSampledBitmapFromResource(
33         applicationContext.resources,
34         resId,
35         reqWidth = 100, // 필요한 크기 지정
36         reqHeight = 100 // 필요한 크기 지정
37     )
38
39     // 비트맵을 메모리 및 디스크 캐시에 추가
40     return try {
41         bitmap?.let {
42             addBitmapToCache(diskCacheManager, key, it)
43         }
44     } catch (e: Exception) {
45         e.printStackTrace()
46         return Result.failure()
47     }
48 }
```

```
44             Result.success()
45         } ?: Result.failure() // 디코딩 실패
46     } catch (e: Exception) {
47         Result.failure()
48     }
49 }
50
51 private fun addBitmapToCache(diskCacheManager: DiskCacheManager, key: String,
52     → bitmap: Bitmap) {
53     // 메모리 캐시에 추가 (존재하지 않는 경우)
54     if (LruCacheManager.memoryCache.get(key) == null) {
55         LruCacheManager.memoryCache.put(key, bitmap)
56     }
57
58     // 디스크 캐시에 추가 (존재하지 않는 경우)
59     // DiskCacheManager.set 메서드 내부에서 중복 확인하므로 여기서는 바로 호출
60     diskCacheManager.set(key, bitmap)
61 }
62
63 // decodeSampledBitmapFromResource 함수 구현 필요 (Q45 예제 참조)
64 private fun decodeSampledBitmapFromResource(res: Resources, resId: Int, reqWidth:
65     → Int, reqHeight: Int): Bitmap? {
66     // ... (Q45의 함수 구현) ...
67     return null // 실제 구현 필요
68 }
69 }
```

위의 WorkManager는 다음을 수행합니다.

- 가능하면 디스크 캐시에서 읽습니다.
- 디코딩으로 대체합니다.
- 결과를 메모리 및 디스크 캐시 모두에 저장합니다.
- 코루틴을 활용하여 메인 스레드 외부에서 안전하게 실행합니다.

요약

안드로이드에서 큰 Bitmap을 효율적으로 캐시하려면 빠른 최근에 접근했던 메모리 캐싱을 위해 LruCache를 사용하고, 앱 세션을 넘어 Bitmap을 유지하기 위해 DiskLruCache를 사용합니다. 두 전략을 하이브리드 형태로 결합하고 구성 변경 시 메모리 캐시를 유지하여 원활한 경험을 제공합니다. WorkManager를 사용한 적절한 초기화 및 백그라운드 작업을 통해 큰 비트맵으로 작업할 때 앱 성능과 사용자 경험을 향상시킬 수 있습니다.

Q) 46. 애니메이션을 어떻게 구현하나요?

애니메이션은 부드러운 전환을 만들고, UI 변화에 대해서 유저들의 이목을 집중시키며, 시각적 피드백을 제공하여 사용자 경험을 향상시킬 수도 있습니다. 안드로이드는 기본 속성 변경부터 정교한 레이아웃 애니메이션까지 애니메이션을 구현하기 위한 여러 메커니즘을 제공합니다. 아래는 전통적인 뷰 기반의 안드로이드 개발에서의 애니메이션 방법을 전반적으로 살펴봅니다.

View Property Animations

API 레벨 11에서 도입된 View Property Animations는 alpha, translationX, translationY, rotation, scaleX와 같은 View 객체의 속성을 애니메이션화할 수 있습니다. 이 애니메이션은 간단한 뷰에 간단한 변화를 줄 때 이상적입니다.

그림 110. View.animate.kt

```
1 val view: View = findViewById(R.id.my_view)
2 view.animate()
3     .alpha(0.5f)
4     .translationX(100f)
5     .setDuration(500) // 지속 시간 설정
6     .setInterpolator(AccelerateDecelerateInterpolator()) // 인터polator 설정 (선택 사항)
7     .start()
```

ObjectAnimator

ObjectAnimator는 View 객체뿐만 아니라 setter 메서드가 있는 모든 객체의 속성을 애니메이션화할 수 있습니다. 커스텀 속성을 애니메이션화하는 데 더 큰 유연성을 제공합니다.

그림 111. ObjectAnimator.kt

```
1 val animator = ObjectAnimator.ofFloat(view, "translationY", 0f, 300f)
2 animator.duration = 500
3 animator.interpolator = OvershootInterpolator() // 인터플레이터 설정 (선택 사항)
4 animator.start()
```

AnimatorSet

AnimatorSet은 여러 애니메이션을 순차적으로 또는 동시에 실행하도록 결합하여 복잡한 애니메이션을 조정하는 데 적합합니다.

그림 112. AnimatorSet.kt

```
1 val fadeAnimator = ObjectAnimator.ofFloat(view, "alpha", 1f, 0f)
2 val moveAnimator = ObjectAnimator.ofFloat(view, "translationX", 0f, 200f)
3
4 val animatorSet = AnimatorSet()
5 // 순차적으로 재생: fadeAnimator 실행 후 moveAnimator 실행
6 animatorSet.playSequentially(fadeAnimator, moveAnimator)
7 // 동시에 재생: animatorSet.playTogether(fadeAnimator, moveAnimator)
8 // 복잡한 순서: animatorSet.play(fadeAnimator).before(moveAnimator) 등
9 animatorSet.duration = 1000 // 전체 세트 지속 시간 (개별 설정도 가능)
10 animatorSet.start()
```

ValueAnimator

ValueAnimator는 임의의 값 사이를 애니메이션화하는 방법을 제공하여 커스텀 가능하고 유연한 애니메이션을 구현할 수 있습니다. 인터플레이터로 애니메이션 진행을 제어함으로써 너비, 높이, 알파 또는 기타 속성을 애니메이션화하는 등 광범위한 사용 사례에 적용할 수 있습니다. 이는 특정 요구 사항에 맞는 정밀하고 동적인 애니메이션을 구현하는 데 적합합니다.

그림 113. ValueAnimator.kt

```
1 val valueAnimator = ValueAnimator.ofInt(0, 100) // 0에서 100까지 정수 값 변경
2 valueAnimator.duration = 500
3 valueAnimator.addUpdateListener { animation ->
4     val animatedValue = animation.animatedValue as Int
5     // animatedValue를 사용하여 UI 업데이트 (예: ProgressBar 너비 조정)
6     val params = binding.progressbar.layoutParams
7     params.width = ((screenSize / 100f) * animatedValue).toInt() // 실수 나눗셈 주의
8     binding.progressbar.layoutParams = params
9 }
10 valueAnimator.start()
```

XML 기반 View 애니메이션

XML 기반 애니메이션은 단순성과 재사용성을 위해 리소스 파일에서 애니메이션 동작을 정의합니다. 이러한 애니메이션은 위치, 크기 조절, 회전 또는 투명도를 조절하는 데 사용할 수 있습니다.

XML 예제: res/anim/slide_in.xml**그림 114. XML-Based View Animations.xml**

```
1 <translate xmlns:android="http://schemas.android.com/apk/res/android"
2     android:fromXDelta="-100%"
3     android:toXDelta="0%"
4     android:duration="500"
5     android:interpolator="@android:anim/accelerate_decelerate_interpolator" /> <!--
   ↳ 인터플레이터 추가 -->
```

사용법

그림 115. AnimationUtils.kt

```
1 val animation = AnimationUtils.loadAnimation(this, R.anim.slide_in)
2 view.startAnimation(animation)
```

MotionLayout

MotionLayout은 복잡한 모션 및 레이아웃 애니메이션을 만들기 위한 안드로이드 전용 컴포넌트입니다. ConstraintLayout 위에 구축되었으며 XML을 사용하여 상태 간의 애니메이션 및 전환을 정의할 수 있습니다.

XML 예제: res/xml/motion_scene.xml (경로 변경: res/xml/)

```
1 <MotionScene xmlns:android="http://schemas.android.com/apk/res/android"
2   xmlns:app="http://schemas.android.com/apk/res-auto">
3
4   <ConstraintSet android:id="@+id/start">
5     <!-- 시작 상태 제약 조건 정의 -->
6     <Constraint android:id="@+id/box">
7       <Layout
8         android:layout_width="100dp"
9         android:layout_height="100dp"
10        app:layout_constraintTop_toTopOf="parent"
11        app:layout_constraintStart_toStartOf="parent" />
12     </Constraint>
13   </ConstraintSet>
14
15   <ConstraintSet android:id="@+id/end">
16     <!-- 종료 상태 제약 조건 정의 -->
17     <Constraint android:id="@+id/box">
18       <Layout
19         android:layout_width="100dp"
20         android:layout_height="100dp"
21         app:layout_constraintBottom_toBottomOf="parent"
```

```
22             app:layout_constraintEnd_toEndOf="parent" />
23         </Constraint>
24     </ConstraintSet>
25
26     <Transition
27         app:constraintSetStart="@+id/start"
28         app:constraintSetEnd="@+id/end"
29         app:duration="500">
30         <!-- 스와이프 또는 클릭 트리거 추가 -->
31         <OnSwipe
32             app:touchAnchorId="@+id/box"
33             app:touchAnchorSide="top"
34             app:dragDirection="dragDown" />
35     </Transition>
36
37 </MotionScene>
```

MotionLayout 사용 예시

```
1 <androidx.constraintlayout.motion.widget.MotionLayout
2     android:layout_width="match_parent"
3     android:layout_height="match_parent"
4     app:layoutDescription="@xml/motion_scene" > <!-- 모션 씬 파일 참조 -->
5
6     <!-- 애니메이션될 뷰 -->
7     <View
8         android:id="@+id/box"
9         android:layout_width="100dp"
10        android:layout_height="100dp"
11        android:background="@color/blue" />
12
13 </androidx.constraintlayout.motion.widget.MotionLayout>
```

MotionLayout은 전환 및 상태에 대한 정밀한 제어로 정교한 애니메이션을 만드는 데 적합합니다.

Drawable 애니메이션

Drawable 애니메이션은 AnimationDrawable을 사용하여 프레임별 전환을 포함하며, 로딩 스피너와 같은 간단한 애니메이션을 만드는 데 적합합니다.

XML 예제: res/drawable/animation_list.xml

```
1 <animation-list xmlns:android="http://schemas.android.com/apk/res/android"
2   android:oneshot="false">
3     <item android:drawable="@drawable/frame1" android:duration="100" />
4     <item android:drawable="@drawable/frame2" android:duration="100" />
5     <!-- 더 많은 프레임 추가 -->
6   </animation-list>
```

사용법

그림 116. AnimationDrawable.kt

```
1 // ImageView의 배경으로 설정했다고 가정
2 val imageView: ImageView = findViewById(R.id.animated_image)
3 imageView.setBackgroundResource(R.drawable.animation_list) // 또는 setImageResource
4 val animationDrawable = imageView.background as AnimationDrawable // 또는
5   imageView.drawable
6 animationDrawable.start()
```

물리 기반 Animations

물리 기반 애니메이션은 실제 역학을 시뮬레이션합니다. 안드로이드는 자연스럽고 동적인 모션 효과를 만들기 위해 SpringAnimation 및 FlingAnimation API를 제공합니다.

그림 117. SpringAnimation.kt

```

1 val springAnimation = SpringAnimation(view, DynamicAnimation.TRANSLATION_Y, of) // 최종
   ↳ 위치 설정
2 springAnimation.spring = SpringForce() // SpringForce 객체 생성
3     .setFinalPosition(of) // 최종 위치 설정 (중복될 수 있음)
4     .setStiffness(SpringForce.STIFFNESS_LOW) // 강성 설정
5     .setDampingRatio(SpringForce.DAMPING_RATIO_HIGH_BOUNCY) // 감쇠 비율 설정
6 springAnimation.start()

```

요약

안드로이드는 간단한 속성 변경부터 복잡한 상태 기반 전환까지 애니메이션을 구현하기 위한 다양한 API를 제공합니다. 크기 조절이나 이동과 같은 간단한 변환에는 **View Property Animations** 및 **ObjectAnimator**가 효과적입니다. 더 복잡한 시나리오의 경우, **AnimatorSet**을 통해 여러 애니메이션을 조정할 수 있으며, **ValueAnimator**는 임의의 값을 애니메이션화하는 유연한 방법을 제공합니다.

실전 질문

- Q) 클릭 시 확장 및 축소되는 버튼에 부드러운 애니메이션을 추가하려고 합니다. 어떻게 구현해 볼 수 있을까요?
- Q) 전통적인 View 애니메이션 대신 MotionLayout을 사용하는 사례는 무엇이고, 그 장점은 무엇인가요?

 **Pro Tips for Mastery:** 인터플레이터(interpolator)는 애니메이션과 어떻게 작동하나요?

Interpolator는 애니메이션 값 변경 속도를 수정하여 애니메이션이 시간 경과에 따라 어떻게 진행되는지 정의합니다. 애니메이션의 가속, 감속 또는 등속 운동을 제어하여 더 자연스럽거나 시각적으로 매력적으로 보이게 만듭니다.

Interpolator는 시작 값과 끝 값 사이의 애니메이션 동작을 정의하는 데 사용됩니다. 예를 들어, 애니메이션이 느리게 시작하여 속도를 높인 다음 멈추기 전에 느려지도록 만들 수

있습니다. 이는 선형 진행을 넘어 애니메이션 실행 방식을 유연하게 제어할 수 있도록 합니다.

안드로이드는 원하는 효과에 따라 사용할 수 있는 여러 인터플레이터 API를 제공합니다.

1. **LinearInterpolator**: 가속이나 감속 없이 일정한 속도로 애니메이션합니다.
2. **AccelerateInterpolator**: 느리게 시작하여 점진적으로 속도를 높입니다.
3. **DecelerateInterpolator**: 빠르게 시작하여 끝으로 갈수록 느려집니다.
4. **AccelerateDecelerateInterpolator**: 부드러운 효과를 위해 가속과 감속을 모두 결합합니다.
5. **BounceInterpolator**: 스프링 애니메이션을 모방하여 애니메이션이 튕기는 것처럼 보이게 합니다.
6. **OvershootInterpolator**: 최종 값을 초과하여 애니메이션한 후 다시 정착합니다.

ObjectAnimator, ValueAnimator 또는 ViewPropertyAnimator와 같은 모든 애니메이션 객체에 인터플레이터를 적용할 수 있습니다. 인터플레이터는 setInterpolator() 메서드를 사용하여 설정됩니다.

다음은 ObjectAnimator에 인터플레이터를 적용하는 예시입니다.

그림 118. InterpolatorExample.kt

```
1 val animator = ObjectAnimator.ofFloat(view, "translationY", 0f, 500f)
2 animator.duration = 1000
3 animator.interpolator = OvershootInterpolator() // OvershootInterpolator 적용
4 animator.start()
```

이 예제에서 OvershootInterpolator는 뷰가 최종 위치에 도달하기 전에 목표 위치를 넘어 애니메이션하도록 만들어 동적인 효과를 생성합니다.

Interpolator 인터페이스를 확장하고 getInterpolation() 메서드를 재정의하여 커스텀 인터플레이터를 만들 수도 있습니다. 이렇게 하면 애니메이션 타이밍을 완전히 커스텀할 수 있습니다.

그림 119. CustomInterpolator.kt

```
1 class CustomInterpolator : Interpolator {
2     override fun getInterpolation(input: Float): Float {
3         // 애니메이션 타이밍을 위한 커스텀 로직 (예: 제곱 가속)
4         return input * input
5     }
6 }
7
8 // 커스텀 인터플레이터 사용
9 animator.interpolator = CustomInterpolator()
```

이 커스텀 인터플레이터는 애니메이션이 느리게 시작하여 시간이 지남에 따라 가속되도록 이차적으로 진행시킵니다.

요약

Interpolator는 타이밍과 진행을 제어하여 안드로이드 애니메이션을 향상시키고 더 시각적인 효과를 표현하도록 합니다. 안드로이드는 다양한 사용 사례를 위해 여러 인터플레이터 API를 제공하며, 상황에 따라 고유한 동작을 위해 커스텀 인터플레이터를 정의할 수 있습니다. 인터플레이터를 효과적으로 활용하면 부드럽고 자연스러운 애니메이션으로 사용자 경험을 크게 향상시킬 수도 있습니다. 그래프 및 수식 등을 통해 각 인터플레이터를 시각적으로 이해하고 싶다면 [Understanding Interpolators in Android⁷⁸](#)를 읽어보셔도 좋습니다.

Q) 47. Window란 무엇인가요?

Window는 화면에 표시되는 Activity 또는 다른 UI 컴포넌트의 모든 뷰를 담는 컨테이너를 나타냅니다. **View** 계층 구조의 최상위 요소이며 애플리케이션 UI와 디스플레이 간의 다리 역할을 합니다.

모든 Activity, Dialog, 또는 Toast는 **Window** 객체에 연결되어 있으며, 이는 포함된 뷰의 레이아웃 매개변수, 애니메이션 및 전환 기능 등을 제공합니다.

⁷⁸<https://medium.com/mindorks/understanding-interpolators-in-android-ce4e8d1d71cd>

Window의 주요 특징

Window 클래스는 다음과 같은 여러 주요 기능을 제공합니다.

1. **DecorView**: Window는 계층 구조의 루트 뷰 역할을 하는 DecorView를 포함합니다. 일반적으로 상태 표시줄(status bar), 내비게이션 바(navigation bar) 및 앱의 콘텐츠 영역을 포함합니다.
2. **레이아웃 매개변수**: Window는 크기, 위치, 가시성과 같은 레이아웃 매개변수를 사용하여 뷰가 어떻게 정렬되고 표시되는지 정의합니다. 이는 프로그래밍 방식으로 커스텀할 수 있습니다.
3. **입력 처리**: Window는 입력 이벤트(가령, 터치 제스처, 키 누름)를 처리하고 이를 적절한 뷰로 전달합니다.
4. **애니메이션 및 전환**: Window는 화면 열기, 닫기 또는 화면 간 전환을 위한 애니메이션을 지원합니다.
5. **시스템 UI 처리**: Window는 상태 표시줄 및 내비게이션 바와 같은 시스템 UI 요소를 표시하거나 숨길 수 있습니다.

Window 관리

Window는 윈도우 추가, 제거 또는 업데이트를 담당하는 시스템 서비스인 WindowManager에 의해 관리됩니다. 이를 통해 다양한 윈도우(가령, 앱 윈도우, 시스템 다이얼로그, 알림)가 기기에서 올바르게 공존하고 상호 작용할 수 있습니다.

Window의 사용 사례

1. **Activity Window 커스텀**: getWindow() 메서드를 사용하여 Activity의 윈도우 동작을 수정할 수 있습니다. 가령, 상태 표시줄을 숨기거나 또는 배경색상을 변경할 수 있습니다.

그림 120. systemUiVisibility.kt

```
1 // 전체 화면 모드 설정 (상태 표시줄 숨김)  
2 window.decorView.systemUiVisibility = View.SYSTEM_UI_FLAG_FULLSCREEN  
3 // 배경을 검은색으로 설정  
4 window.setBackgroundDrawable(ColorDrawable(Color.BLACK))
```

1. **Dialog 생성**: Dialog는 새로운 윈도우를 사용하여 그 위에 구현되므로 다른 UI 요소 위에 떠 있을 수 있습니다.
2. **오버레이 사용**: TYPE_APPLICATION_OVERLAY를 통해 시스템 수준 기능이나 헤드업 알림과 같은 오버레이를 만드는 데 Window를 사용할 수 있습니다.
3. **멀티 윈도우 모드 처리**: 안드로이드는 분할 화면이나 PIP(Picture-in-Picture) 모드와 같은 기능을 활성화하기 위해 멀티 윈도우를 지원합니다.



Additional Tips: Window 클래스는 2006년도 Android SDK의 API 레벨 1 때부터 존재했던 가장 고전적이고, 안드로이드 시스템의 근간이 되는 기능입니다.

요약

Window 클래스는 앱의 뷰와 UI 요소를 위한 최상위 컨테이너를 제공하는 안드로이드의 필수 구성 요소입니다. 앱이 화면, 시스템 UI 및 사용자 입력과 상호 작용하는 방식을 커스텀할 수 있게 하며, WindowManager를 통해 Window를 관리하고 안드로이드 환경과의 원활한 통합을 보장합니다.

실전 질문

Q) 단순한 레이아웃을 가진 Activity가 화면에 표시될 때 몇 개의 Window가 존재하며, 어느 부분에 필요한가요?

💡 Pro Tips for Mastery: WindowManager란 무엇인가요?

WindowManager는 화면에서 윈도우의 배치, 크기 및 모양을 관리하는 안드로이드 시스템 서비스입니다. 애플리케이션과 안드로이드 시스템 간의 윈도우 관리 인터페이스 역할을 하며, 앱이 윈도우를 생성, 수정 또는 제거할 수 있도록 합니다. 안드로이드의 윈도우는 전체 화면 Activity부터 플로팅 오버레이까지 무엇이든 될 수 있습니다.

WindowManager의 주요 책임

WindowManager는 시스템의 윈도우 계층 구조를 관리하는 역할을 합니다. 윈도우가 z-순서 (레이어링) 및 다른 시스템 윈도우와의 상호 작용에 따라 올바르게 표시되도록 보장합니다. 예를 들어, 윈도우의 포커스 변경, 터치 이벤트 및 애니메이션을 처리합니다.

일반적인 사용 사례

- **커스텀 View 추가:** 개발자는 WindowManager를 사용하여 플로팅 위젯이나 시스템 오버레이와 같이 앱의 표준 Activity 외부에 커스텀 뷰를 표시할 수 있습니다.
- **기존 Window 수정:** 애플리케이션은 크기 조절, 위치 변경 또는 투명도 변경과 같이 기존 윈도우의 속성을 업데이트할 수 있습니다.
- **Window 제거:** removeView() 메서드를 사용하여 프로그래밍 방식으로 윈도우를 제거할 수 있습니다.

WindowManager 작업하기

WindowManager 서비스는 Context.getSystemService(Context.WINDOW_SERVICE)를 통해 접근합니다.

아래는 WindowManager를 사용하여 화면에 플로팅 뷰를 추가하는 예시입니다.

그림 121. Adding a floating view using WindowManager

```
1 val windowManager = context.getSystemService(Context.WINDOW_SERVICE) as WindowManager
2
3 val floatingView = LayoutInflater.from(context).inflate(R.layout.floating_view, null)
4
5 val params = WindowManager.LayoutParams(
6     WindowManager.LayoutParams.WRAP_CONTENT, // 너비
7     WindowManager.LayoutParams.WRAP_CONTENT, // 높이
8     // Android 8.0 (API 26) 이상에서는 TYPE_APPLICATION_OVERLAY, 이전 버전에서는
9     // → TYPE_PHONE 등 사용
10    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
11        WindowManager.LayoutParams.TYPE_APPLICATION_OVERLAY
12    } else {
13        WindowManager.LayoutParams.TYPE_PHONE
14    },
15    WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE, // 윈도우 플래그 (포커스 안 받음)
16    PixelFormat.TRANSLUCENT // 픽셀 형식 (투명 배경 허용)
17 )
18
19 // 초기 위치 설정 (예: 화면 좌상단)
20 params.gravity = Gravity.TOP or Gravity.START
21 params.x = 0
22 params.y = 0
23
24 // 윈도우에 뷰 추가
25 windowManager.addView(floatingView, params)
26
27 // ... 추후에 플로팅 뷰가 더 사용되지 않을 때 제거 ...
28 // windowManager.removeView(floatingView)
```

위의 예제에서 아래와 같은 `WindowManager.LayoutParams` 속성들이 사용되었습니다.

- `TYPE_APPLICATION_OVERLAY`는 뷰가 다른 앱 위에 표시되도록 허용합니다. (권한 필요)

- FLAG_NOT_FOCUSABLE과 같은 플래그는 달리 지정하지 않는 한 윈도우가 사용자 입력과 상호 작용하지 않도록 합니다.

제한 사항 및 권한

시스템 오버레이와 같은 특정 유형의 윈도우에는 SYSTEM_ALERT_WINDOW와 같은 특별한 사용자 권한이 필요합니다. 안드로이드 8.0 (API 레벨 26)부터 시스템은 보안상의 이유로 오버레이 윈도우에 대해 더 엄격한 제한을 부과합니다.

요약

WindowManager는 안드로이드에서 윈도우를 관리하기 위한 기본적인 API입니다. 개발자가 표준 Activity 생명주기 외부에서 코드로 직접 뷰를 추가, 업데이트 및 제거할 수 있도록 합니다. WindowManager를 적절하게 사용하면 플로팅 위젯이나 오버레이와 같은 고급 기능을 활성화할 수 있지만, 권한 및 사용자 경험에 대한 신중한 고려가 필요합니다.

Pro Tips for Mastery: PopupWindow란 무엇인가요?

PopupWindow는 기존 레이아웃 위에 떠 있는 팝업 뷰를 표시하는 데 사용되는 UI 컴포넌트입니다. 일반적으로 화면을 덮고 해제하기 위해 사용자 상호 작용이 필요한 Dialog와 달리, PopupWindow는 더 유연하며 화면의 특정 위치에 배치될 수 있으며, 메뉴나 툴팁과 같은 임시 또는 상황별 UI 요소에 자주 사용됩니다. 다음은 PopupWindow의 특징입니다.

- 레이아웃 위에 존재하는 어떠한 뷰든 해당 뷰 위에 콘텐츠를 표시할 수 있습니다.
- 팝업 외 영역 화면을 어둡게 하거나 사용자 인터렉션을 차단할 필요가 없는 시나리오에서 팝업 뒤의 다른 UI 컴포넌트와 상호 작용할 수 있습니다.
- 커스텀 레이아웃, 애니메이션 및 해제 동작을 구현할 수 있습니다.
- 원활한 사용자 경험을 위해 터치 기반 해제 및 포커스 제어를 지원합니다.

아래 예제 코드와 같이 PopupWindow를 생성하고 화면에 노출할 수 있습니다.

그림 122. PopupWindowExample.kt

```
1 class MainActivity : AppCompatActivity() {
2     override fun onCreate(savedInstanceState: Bundle?) {
3         super.onCreate(savedInstanceState)
4         setContentView(R.layout.activity_main)
5
6         // 팝업을 위한 커스텀 레이아웃 인플레이트
7         val popupView = layoutInflater.inflate(R.layout.popup_layout, null)
8
9         // PopupWindow 생성
10        val popupWindow = PopupWindow(
11            popupView,
12            ViewGroup.LayoutParams.WRAP_CONTENT, // 너비
13            ViewGroup.LayoutParams.WRAP_CONTENT, // 높이
14            true // 포커스 가능하게 설정 (외부 클릭 시 닫기 위해 필요)
15        )
16
17        // 팝업 배경 설정 (외부 클릭 감지를 위해 필요)
18        popupWindow.setBackgroundDrawable(ColorDrawable(Color.TRANSPARENT))
19        // 팝업 외부 터치 가능하게 설정
20        popupWindow.isOutsideTouchable = true
21
22        // 팝업 외부 클릭 시 닫기 (Focusable이 true일 때 작동)
23        // popupView.setOnTouchListener는 팝업 내부 터치에만 해당됩니다.
24        // 외부 클릭 처리는 PopupWindow 속성으로 합니다.
25
26        val button = findViewById<Button>(R.id.button)
27        button.setOnClickListener { anchorView -> // 클릭된 뷰(버튼)를 anchorView로 사용
28            // 버튼 아래에 PopupWindow 표시
29            popupWindow.showAsDropDown(anchorView)
30        }
31    }
32 }
```

 **Additional Tips:** PopupWindow는 Window를 상속하고 있을 것 같지만, PopupWindow라는 독립적인 클래스로서 존재하고, 내부에서는 WindowManager를 통해 Window를 화면에 추가하고 제거합니다.

요약

PopupWindow는 플로팅 UI 요소를 표시하기 위한 안드로이드의 고전적이고 유용한 API입니다. 커스텀 레이아웃을 사용하고 유연한 위치 지정을 지원하는 기능 덕분에 주 앱 흐름을 방해하지 않고 사용자 상호 작용을 향상시키는 상황별 메뉴, 툴팁 및 임시 팝업을 만드는데 이상적입니다.

Q) 48. 웹 페이지를 어떻게 렌더링하나요?

WebView는 앱 내에서 직접 웹 콘텐츠를 표시하고 상호 작용할 수 있는 유용한 안드로이드 컴포넌트입니다. 애플리케이션에 내장된 미니 브라우저 역할을 하여 개발자가 웹 페이지를 렌더링하고, HTML 콘텐츠를 로드하거나, JavaScript를 직접적으로 실행할 수 있도록 합니다.

앱이 실행 중인 기기에서 최신 **WebView** 기능을 안전하게 활용하려면 [AndroidX Webkit](#)⁷⁹ 라이브러리를 사용해야 합니다. 이 라이브러리는 이전 버전과 호환되는 API를 제공하여 기기의 안드로이드 버전에 관계없이 최신 기능에 접근할 수 있도록 보장합니다.

WebView 초기화하기

WebView를 사용하려면 레이아웃 파일에 포함하거나 코드로 직접 생성할 수 있습니다. 아래는 XML 레이아웃에 WebView를 추가하는 예시입니다.

⁷⁹<https://developer.android.com/reference/androidx/webkit/package-summary>

```
1 <!-- activity_main.xml -->
2 <WebView
3     android:id="@+id/webView"
4     android:layout_width="match_parent"
5     android:layout_height="match_parent" />
```

필요한 경우 코드를 통해 직접 생성할 수도 있습니다.

그림 123. WebView.kt

```
1 val webView = WebView(this)
2 setContentView(webView)
```

웹 페이지 로드하기

웹 페이지를 로드하려면 WebView 인스턴스에서 loadUrl() 메서드를 사용합니다. 페이지가 인터넷 접근을 요구하는 경우 안드로이드 매니페스트에서 필요한 권한을 활성화해야 합니다.

그림 124. WebView.kt

```
1 val webView: WebView = findViewById(R.id.webView)
2 webView.loadUrl("https://www.example.com")
```

인터넷 접근을 허용하기 위해 AndroidManifest.xml에 다음 권한을 추가해야 합니다.

```
1 <uses-permission android:name="android.permission.INTERNET" />
```

JavaScript 활성화하기

웹 콘텐츠가 JavaScript를 요구하는 경우 WebSettings를 수정하여 활성화합니다.

그림 125. WebView.kt

```
1 val webSettings = webView.settings  
2 webSettings.javaScriptEnabled = true
```

WebView 동작 커스텀하기

WebView는 이벤트를 처리하고 동작을 커스텀하는 메서드를 제공합니다.

- **페이지 내비게이션 가로채기:** 외부 브라우저에서 열지 않고 WebView 내에서 페이지 내비게이션을 처리하려면 WebViewClient를 사용해야 합니다.

그림 126. Navigation.kt

```
1 webView.webViewClient = object : WebViewClient() {  
2     // API 24 미만 버전 호환성  
3     override fun shouldOverrideUrlLoading(view: WebView?, url: String?): Boolean {  
4         url?.let { view?.loadUrl(it) }  
5         return true // WebView가 URL 로딩을 처리함을 나타냄  
6     }  
7  
8     // API 24 이상 버전  
9     override fun shouldOverrideUrlLoading(view: WebView?, request:  
10        WebResourceRequest?): Boolean {  
11         request?.url?.let { view?.loadUrl(it.toString()) }  
12         return true  
13     }  
14 }
```

- **다운로드 처리하기:** WebView를 통해 다운로드되는 파일을 관리하려면 DownloadListener를 활용할 수 있습니다.

그림 127. DownloadListener.kt

```

1 webView.setDownloadListener { url, userAgent, contentDisposition, mimeType,
2     → contentLength ->
3         // 여기서 파일 다운로드 처리 (예: DownloadManager 사용)
4         val request = DownloadManager.Request(Uri.parse(url))
5         // ... (DownloadManager 설정) ...
6         val downloadManager = getSystemService(Context.DOWNLOAD_SERVICE) as DownloadManager
7         downloadManager.enqueue(request)
}

```

- **WebView에서 JavaScript 실행하기**: evaluateJavascript 또는 loadUrl("javascript:'...')을 사용하여 JavaScript 코드를 주입합니다.

그림 128. JavaScript.kt

```

1 // API 19 이상 권장 방식
2 webView.evaluateJavascript("document.body.style.backgroundColor = 'red';") { result ->
3     Log.d("WebView", "JavaScript 실행 결과: $result")
4 }
5 // 이전 방식 (결과 받기 어려움)
6 // webView.loadUrl("javascript:document.body.style.backgroundColor = 'blue';")

```

JavaScript를 안드로이드 코드에 바인딩하기 위한 포괄적인 가이드

JavaScript와 안드로이드 코드를 통합하면 클라이언트 측 스크립트와 안드로이드 네이티브 기능 간의 원활한 상호 작용을 허용하여 하이브리드 웹 애플리케이션을 향상시킬 수 있습니다. 이는 특히 안드로이드 앱 내 WebView에서 실행되는 웹 애플리케이션에 유용하며, JavaScript가 안드로이드 특정 기능을 활용할 수 있도록 합니다. 예를 들어, JavaScript의 alert() 함수에 의존하는 대신 네이티브 안드로이드 디바이스로그나 토스트 메시지를 트리거할 수 있습니다.

이 상호 작용을 구현하려면 addJavascriptInterface() 메서드를 사용해야 합니다. 해당 메서드는 Java 객체를 WebView의 JavaScript 컨텍스트에 바인딩하여 인터페이스 이름을 지정함으로써 JavaScript를 통해 해당 메서드에 접근할 수 있도록 합니다.

JavaScript를 안드로이드에 바인딩하는 예제를 통해서 살펴볼 수 있습니다.

그림 129. WebAppInterface.kt

```
1 // WebView와 통신할 인터페이스 클래스 정의
2 class WebAppInterface(private val context: Context) {
3
4     // JavaScript에서 호출할 수 있도록 @JavascriptInterface 어노테이션 사용
5     @JavascriptInterface
6     fun showToast(message: String) {
7         Toast.makeText(context, message, Toast.LENGTH_SHORT).show()
8     }
9 }
10
11 // WebView 설정 및 인터페이스 바인딩
12 val webView: WebView = findViewById(R.id.webview)
13 webView.settings.javaScriptEnabled = true // JavaScript 활성화 필수
14 // "Android"라는 이름으로 인터페이스 객체 등록
15 webView.addJavascriptInterface(WebAppInterface(this), "Android")
```

해당 예제에서 WebAppInterface 클래스는 @JavascriptInterface 어노테이션이 달린 showToast 메서드를 노출합니다. addJavascriptInterface() 메서드는 이 인터페이스를 “Android”라는 이름으로 WebView에 바인딩합니다. 이제 WebView에서 실행되는 JavaScript가 이 메서드를 직접적으로 호출할 수 있습니다.

아래 예제에서는 JavaScript에서 showToast 메서드를 호출하는 방법을 보여줍니다.

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>WebView Example</title>
5 </head>
6 <body>
7     <button onclick="callAndroidFunction()">Click Me</button>
8     <script type="text/javascript">
9         function callAndroidFunction() {
10             // "Android" 인터페이스의 showToast 메서드 호출
11             Android.showToast("Hello from JavaScript!");
12         }
13     </script>
14 </body>
15 </html>
```

버튼을 클릭하면 JavaScript 함수 `callAndroidFunction()`이 안드로이드 인터페이스의 `showToast` 메서드를 호출하여 네이티브 토스트 메시지를 표시합니다.

`addJavascriptInterface()`는 유용하지만 한편으로 상당한 보안 위험이 따릅니다. `WebView`가 신뢰할 수 없거나 동적인 HTML 콘텐츠를 로드하는 경우 공격자가 악의적인 JavaScript를 주입하여 노출된 인터페이스를 악용하고 의도하지 않은 안드로이드 코드를 잠재적으로 실행할 수 있습니다.

보안 고려 사항

- 보안 위험에 앱을 노출시킬 수 있으므로 필요하지 않은 한 JavaScript를 활성화하지 않는 것이 좋습니다.
- 파일에 대한 무단 접근을 방지하기 위해 `setAllowFileAccess()` 및 `setAllowFileAccessFromFileURLs()`를 신중하게 사용하는 것이 좋습니다.
- **교차 사이트 스크립팅(XSS)** 또는 **URL 스피핑**과 같은 공격을 방지하기 위해 항상 사용자 입력을 검증하고 URL을 정제해야 합니다.
- `@JavascriptInterface`를 통해 노출된 메서드가 보안 취약점을 유발하지 않는지 확인해야 합니다.

요약

WebView는 안드로이드 애플리케이션에서 웹 콘텐츠를 렌더링하기 위한 기본적인 컴포넌트입니다. WebViewClient를 사용하여 동작을 커스텀하고 필요할 때 JavaScript와 같은 기능을 활성화하여 사용자에게 원활한 경험을 제공할 수 있습니다. 하지만 앱에 웹 콘텐츠를 통합할 때는 항상 보안 및 성능 영향을 고려해야 합니다. 자세한 내용은 [WebView에서 웹 앱 빌드](#)⁸⁰ 공식문서를 참조하세요.

실전 질문

Q) 외부 링크를 클릭할 때 사용자가 앱을 벗어나는 것을 방지하기 위해 WebView 내비게이션을 효과적으로 처리하는 방법에는 무엇이 있는지 설명해 주세요

⁸⁰<https://developer.android.com/develop/ui/views/layout/webapps/webview#kotlin>

카테고리 2: Jetpack 라이브러리

Jetpack⁸¹은 안드로이드 개발자가 애플리케이션을 더 효율적이고 유지 관리 가능하게 구축하는 데 도움이 되도록 Google에서 제공하는 라이브러리 및 툴 모음입니다. Jetpack 라이브러리는 생명주기 관리, UI 내비게이션, 백그라운드 작업 및 데이터 저장소를 포함하여 일반적으로 개발자들이 안드로이드 앱 개발 시 마주할 수 있는 다양한 문제를 해결하기 위한 솔루션을 제시합니다. Jetpack 라이브러리는 최신 안드로이드 개발 관행과 원활하게 통합되도록 설계되어 모범 사례를 준수하면서 앱 개발 시 여러 측면을 간소화합니다.

Jetpack은 모듈식, 즉 라이브러리 형태로 제공되므로 개발자는 프로젝트에 필요한 특정 라이브러리만 포함하도록 선택할 수 있습니다. 이러한 유연성을 통해 팀은 ViewModel을 상태 관리에, Navigation을 화면 전환 처리에, Room을 로컬 데이터베이스 관리에 통합하는 등의 기능을 유연하게 필요와 상황에 따라서 통합할 수 있습니다.

Jetpack은 앱 개발 시 마주할 수 있는 일반적이고 복잡한 문제들을 대부분 해결함으로써 안드로이드의 핵심 기능을 보완하는 역할을 수행합니다. 그러나 개발자가 Jetpack 라이브러리를 반드시 사용해야 하는 것은 아니며, 특정 상황에서는 Jetpack이 아닌 대체 솔루션이나 매뉴얼 하게 구현하는 것이 여전히 선호될 수 있습니다. Jetpack 라이브러리가 실제 안드로이드 생태계에서 어떻게 활용되는지 올바르게 이해한다면, 개발자가 안드로이드 앱 개발 시 어떤 Jetpack 라이브러리 및 솔루션을 적재적소에 채택해야 할지 올바른 결정을 내릴 수 있습니다.

이번 카테고리는 모든 Jetpack 라이브러리를 다루지 않습니다. 대신 안드로이드 애플리케이션 구축에 널리, 일반적으로 사용되는 라이브러리에 중점을 둡니다. 이 책에서 다루지 않는 다른 Jetpack 라이브러리를 학습하고 싶으시다면 [공식 안드로이드 문서](#)⁸²에서 더 포괄적인 리소스를 참조할 수 있습니다.

Q) 49. AppCompat 라이브러리란 무엇인가요?

AppCompat 라이브러리⁸³는 개발자가 하위 버전의 안드로이드와의 호환성을 유지하는 데 도움이 되도록 설계된 Android Jetpack 제품군의 일부입니다. 하위 안드로이드 버전과의

⁸¹<https://developer.android.com/jetpack>

⁸²<https://developer.android.com/jetpack>

⁸³<https://developer.android.com/jetpack/androidx/releases/appcompat>

하위 호환성을 보장하면서 최신 기능을 앱에 사용할 수 있습니다. 해당 라이브러리는 다양한 안드로이드 버전을 가진 광범위한 기기를 대상으로 하는 개발자에게 특히 유용합니다.

AppCompat 라이브러리는 앱 기능과 디자인 일관성을 향상시키는 다양한 하위 호환 컴포넌트 및 유ти리티를 제공합니다. 아래는 주요 기능입니다.

- **UI 컴포넌트 하위 호환성:** AppCompat 라이브러리는 FragmentActivity를 확장하고 하위 버전의 안드로이드와의 호환성을 보장하는 AppCompatActivity와 같은 최신 UI 컴포넌트를 제공합니다. 이를 통해 개발자는 하위 안드로이드 버전을 실행하는 기기에서 액션 바와 같은 기능을 사용할 수 있습니다.
- **Material Design 지원:** AppCompat을 사용하면 개발자는 하위 안드로이드 버전을 실행하는 기기에 Material Design 원칙을 통합할 수 있습니다. 여기에는 AppCompatButton, AppCompatTextView 등 기기의 API 레벨에 따라 모양과 동작을 자동으로 조정하는 위젯이 제공됩니다.
- **테마 및 스타일링 지원:** AppCompat을 사용하면 Theme.AppCompat과 같은 테마를 사용하여 모든 API 레벨에서 일관된 UI를 보장할 수 있습니다. 이러한 테마는 벡터 드로어블 지원과 같은 최신 스타일링 기능을 하위 안드로이드 버전에 제공합니다.
- **동적 기능 지원:** AppCompat 라이브러리는 동적 리소스 로딩 및 벡터 드로어블 지원을 제공하여 하위 호환성을 유지하면서 최신 디자인 요소를 효율적으로 구현하기 쉽게 만듭니다.

AppCompat을 사용하는 이유

AppCompat 라이브러리를 사용하는 주된 이유는 최신 안드로이드 기능과 UI 컴포넌트가 지원되는 모든 API 레벨에서 일관되게 작동하도록 보장하는 것입니다. 개발자가 최신 기능이 풍부한 앱을 구축하는 데 집중할 수 있도록 하면서, 하위 버전의 안드로이드 기기에서 호환성을 유지하기 위한 복잡성을 줄여줍니다.

AppCompatActivity 사용 예제

아래는 안드로이드 앱에서 AppCompatActivity를 사용하는 간단한 예제입니다.

그림 130. AppCompatActivityExampleActivity.kt

```
1 import androidx.appcompat.app.AppCompatActivity
2 import android.os.Bundle
3
4 class MainActivity : AppCompatActivity() {
5     override fun onCreate(savedInstanceState: Bundle?) {
6         super.onCreate(savedInstanceState)
7         setContentView(R.layout.activity_main)
8     }
9 }
```

이 예제에서 AppCompatActivity는 Activity가 하위 안드로이드 버전을 실행하는 기기에서 액션 바와 같은 기능을 사용할 수 있도록 보장합니다.

요약

AppCompat 라이브러리는 광범위한 기기 및 API 레벨과 호환되는 안드로이드 애플리케이션을 구축하는 데 유용합니다. 하위 호환 컴포넌트, Material Design 지원 및 일관된 테마를 제공함으로써 하위 버전 기기에서의 사용자 경험을 향상시키면서 개발을 단순화합니다.

실전 질문

Q) AppCompat 라이브러리는 하위 안드로이드 버전에서 Material Design 지원을 어떻게 가능하게 하며, 이와 같은 동작을 기반으로 하는 주요 UI 컴포넌트에는 무엇이 있나요?

Q) 50. Material Design Components (MDC)란 무엇인가요?

Material Design Components (MDC)⁸⁴는 Google의 Material Design 가이드라인⁸⁵을 기반으로 하는 커스텀 가능한 UI 위젯 및 컴포넌트 집합입니다. 해당 컴포넌트는 개발자가 앱의 브랜딩 및 디자인 요구 사항에 맞게 모양과 동작을 커스텀할 수 있도록 하면서 일관되고 사용자 친화적인 인터페이스를 제공하도록 설계되었습니다.

⁸⁴<https://developer.android.com/design/ui/mobile/guides/components/material-overview>

⁸⁵<https://m2.material.io/design/guidelines-overview>

Material Design Components는 안드로이드 프로젝트에 원활하게 통합되고 최신 디자인 원칙이 효과적으로 구현되도록 보장하는 [Material Components for Android \(MDC-Android\)](#) 라이브러리⁸⁶를 제공합니다.

Material Design Components의 주요 특징

1. **Material Theming**: MDC는 [Material Theming](#)⁸⁷을 통해 테마 설정을 지원하여 개발자가 타이포그래피, 모양 및 색상을 전역적으로 또는 컴포넌트 수준에서 커스텀할 수 있도록 합니다. 이를 통해 앱 전체에서 일관성을 유지하면서 UI를 브랜드 아이덴티티에 쉽게 맞출 수 있습니다.
2. **미리 빌드된 UI 컴포넌트**: MDC는 버튼, 카드, 앱 바, 내비게이션 드로어, 칩 등과 같이 즉시 사용할 수 있는 광범위한 UI 컴포넌트를 제공합니다. 이러한 컴포넌트는 접근성, 성능 및 반응성에 최적화되어 있습니다.
3. **애니메이션 지원**: Material Design은 [모션 및 전환](#)⁸⁸을 강조합니다. MDC에는 공유 요소 전환 (shared element transition), 리플 효과 및 시각적 피드백과 같은 애니메이션에 대한 내장 지원이 포함되어 사용자 상호 작용을 향상시킵니다.
4. **다크 모드 지원**: 해당 라이브러리에는 다크 모드를 쉽게 구현할 수 있는 API가 포함되어 있어 개발자가 시각적 일관성을 보장하면서 라이트 및 다크 모드에 대한 테마를 정의할 수 있습니다.
5. **접근성**: MDC는 더 큰 터치 대상, 접근성을 위한 semantic 레이블 및 적절한 포커스 관리와 같은 기능을 제공하여 접근성 표준을 준수하며 모든 사용자를 위한 포괄적인 UI를 보장합니다.

Material Button 사용 예제

아래는 MDC 라이브러리의 Material Button을 사용하는 방법의 예시입니다.

⁸⁶<https://github.com/material-components/material-components-android>

⁸⁷<https://m2.material.io/design/material-theming/overview.html#material-theming>

⁸⁸<https://m2.material.io/design/motion/understanding-motion.html#principles>

그림 131. MaterialButtonExample.xml

```
1 <com.google.android.material.button.MaterialButton  
2     xmlns:app="http://schemas.android.com/apk/res-auto"  
3     android:layout_width="wrap_content"  
4     android:layout_height="wrap_content"  
5     android:text="Click Me"  
6     app:cornerRadius="8dp"  
7     app:icon="@drawable/ic_example"  
8     app:iconGravity="start"  
9     app:iconPadding="8dp" />
```

여기서 `MaterialButton` 위젯은 `Material Design` 원칙에 맞게 둥근 모서리, 아이콘 및 패딩으로 커스텀됩니다.

요약

`Material Design Components`를 사용하여 개발자는 Google의 `Material Design` 가이드라인을 준수하는 현대적이고 일관된 사용자 인터페이스를 만들 수 있습니다. 테마 설정, 미리 빌드된 위젯, 애니메이션 지원 및 접근성 도구와 같은 기능을 통해 MDC는 다양한 기기 및 화면 크기에서 적응성과 반응성을 보장하면서 고품질 UI 구현 프로세스를 단순화합니다.

실전 질문

Q) MDC의 `Material Theming`은 앱 전체에서 디자인 일관성을 유지하는 데 어떻게 도움이 되나요?

Q) 51. `ViewBinding`을 사용하면 어떤 장점이 있나요?

`ViewBinding`⁸⁹은 레이아웃의 뷰와 상호 작용하는 프로세스를 단순화하기 위해 안드로이드에서 도입된 기능입니다. 수동으로 `findViewById()`를 호출하지 않아도 되고, 뷰에 접근하는 타입-세이프(type-safe) 방식을 제공하여 보일러 플레이트 코드를 줄이고 잠재적인 런타임 오류를 최소화합니다.

⁸⁹<https://developer.android.com/topic/libraries/view-binding>

ViewBinding 작동 방식

프로젝트에서 ViewBinding을 활성화하면 안드로이드는 각 XML 레이아웃 파일에 대한 바인딩 클래스를 생성합니다. 생성된 바인딩 클래스의 이름은 레이아웃 파일 이름에서 파생되며, 각 밑줄(_)은 카멜 케이스(camel case)로 변환되고 이름 끝에 Binding이 추가됩니다. 예를 들어, 레이아웃 파일 이름이 activity_main.xml이면 생성된 바인딩 클래스는 ActivityMainBinding이 됩니다.

바인딩 클래스에는 레이아웃의 모든 뷰에 대한 참조가 포함되어 있어 캐스팅하거나 findViewById()를 호출할 필요 없이 직접 접근할 수 있습니다.

그림 132. ViewBinding.kt

```
1 // Activity에서의 사용 예제
2 class MainActivity : AppCompatActivity() {
3     // 바인딩 클래스 인스턴스 선언
4     private lateinit var binding: ActivityMainBinding
5
6     override fun onCreate(savedInstanceState: Bundle?) {
7         super.onCreate(savedInstanceState)
8         // 레이아웃 인플레이트 및 바인딩 클래스 초기화
9         binding = ActivityMainBinding.inflate(layoutInflater)
10        // 루트 뷰를 액티비티의 컨텐츠 뷰로 설정
11        setContentView(binding.root)
12
13        // 바인딩 객체를 통해 뷰에 직접 접근
14        binding.textView.text = "Hello, ViewBinding!"
15        binding.button.setOnClickListener { /* 클릭 리스너 로직 */ }
16    }
17 }
```

위 예제에서 ActivityMainBinding은 activity_main.xml 레이아웃 파일에 대해 생성된 클래스입니다. inflate() 메서드는 바인딩 클래스의 인스턴스를 생성하는 데 사용되고, binding.root는 레이아웃을 설정하기 위해 setContentView()에 전달됩니다.

ViewBinding의 장점

- **타입 안전성(Type Safety)**: 캐스팅할 필요 없이 뷰에 직접 접근하여 타입 불일치로 인한 런타임 오류를 제거합니다.
- **더 깔끔한 코드**: findViewById()를 호출할 필요가 없어지고 보일러 플레이트 코드가 줄어듭니다.
- **Null 안전성(Null Safety)**: nullable 타입의 뷰를 자동으로 처리하여 선택적 UI 컴포넌트와 상호 작용할 때 더 안전한 코드를 보장합니다.
- **성능**: DataBinding과 달리 ViewBinding은 바인딩 표현식이나 추가 XML 파싱을 사용하지 않으므로 런타임 오버헤드가 최소화됩니다.

DataBinding과의 비교

DataBinding은 바인딩 표현식 및 양방향 데이터 바인딩과 같은 더 많은 기능을 제공하지만 더 복잡하고 런타임 오버헤드를 유발합니다. 반면에 ViewBinding은 순수하게 뷰 상호 작용 단순화에 중점을 두며 성능 면에서 더 가볍습니다. LiveData나 Flow 등을 바인딩하여 데이터를 직접적으로 결합하는 등과 같은 기능이 필요하지 않은 경우 이상적입니다.

ViewBinding 활성화하기

프로젝트에서 ViewBinding을 활성화하려면 build.gradle 파일에 다음을 추가합니다.

```
1 // build.gradle (app 수준)
2 android {
3     ...
4     buildFeatures {
5         viewBinding = true
6     }
7 }
```

활성화되면 프로젝트의 모든 XML 레이아웃에 대해 바인딩 클래스가 자동으로 생성됩니다.

💡 Additional Tips: ViewBinding이 구글에 의해서 공식적으로 지원되기 전에는, findViewById를 사용하지 않고 annotation processing을 활용한 의존성 주입 (dependency injection)을 통해 필드에 View 인스턴스를 주입하여 타입 안정성을 보장하는 형태의 라이브러리 ButterKnife⁹⁰가 사용되었습니다. 현재의 JakeWharton을 존재하게 만들었다고 해도 과언이 아닌 ButterKnife는 한때 안드로이드 생태계에서 모르는 사람이 없을 정도로 많이 사용되었으며, 안드로이드 생태계에 많은 변화와 영감을 준 오픈 소스 라이브러리입니다. 현재는 ViewBinding이 구글에 의해 공식 지원되면서 ButterKnife는 deprecated 되었지만, 당시는 상당히 창의적인 아이디어였기에 많은 인기를 끌었고, 의존성 주입에 대해 견고한 학습을 원하시는 분들은 여전히 소스 코드를 탐색하며 학습해 볼 만한 가치가 있습니다.

요약

ViewBinding은 안드로이드 앱에서 뷰와 상호 작용하는 가볍고 타입-세이프한 방법으로, 보일러 플레이트 코드를 줄이고 코드 안전성을 향상시킵니다. findViewById()의 간단한 대안이며 DataBinding의 고급 기능이 필요하지 않은 경우 훌륭한 선택입니다. ViewBinding을 활성화하면 더 나은 유지 관리성과 런타임 안전성을 보장하면서 UI 상호 작용을 간소화할 수 있습니다.

실전 질문

Q) ViewBinding은 findViewById()와 비교하여 타입 안전성과 null 안전성을 어떻게 개선 하며, 해당 접근 방식의 이점은 무엇인가요?

Q) 52. DataBinding의 동작 원리에 대해서 설명해주세요.

DataBinding⁹¹은 XML 레이아웃의 UI 컴포넌트를 앱의 데이터 소스에 직접 바인딩할 수 있는 안드로이드 라이브러리입니다. findViewById()와 같은 보일러 플레이트 코드를 줄이고 UI와 데이터 모델 간의 실시간 업데이트를 허용하여, UI 디자인에 선언적 프로그래밍을 부분적으로 가능하게 합니다. 또한 이 개념은 UI 로직과 비즈니스 로직 분리를 위한 디자인

⁹⁰<https://github.com/JakeWharton/butterknife>

⁹¹<https://developer.android.com/topic/libraries/data-binding>

패턴으로 Microsoft에서 시작된⁹² MVVM (Model-View-ViewModel) 아키텍처에서 중심적인 역할을 합니다.

DataBinding 활성화하기

DataBinding을 활성화하려면 build.gradle 파일에 다음을 추가합니다.

```
1 // build.gradle (app 수준)
2 android {
3     ...
4     buildFeatures {
5         dataBinding = true
6     }
7 }
```

DataBinding 작동 방식

DataBinding은 <layout> 태그를 사용하는 각 XML 레이아웃에 대한 바인딩 클래스를 생성합니다. 이 클래스는 뷰에 대한 직접적인 접근을 제공하고 표현식을 사용하여 XML에서 직접 데이터를 바인딩할 수 있습니다.

DataBinding XML 레이아웃 예제

⁹²<https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm>

그림 133. activity_main.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <layout xmlns:android="http://schemas.android.com/apk/res/android"
3   xmlns:app="http://schemas.android.com/apk/res-auto">
4
5   <data>
6     <variable
7       name="vm"
8       type="com.example.myapp.ui.UserViewModel" /> <!-- ViewModel 클래스 경로 -->
9   </data>
10
11  <LinearLayout
12    android:layout_width="match_parent"
13    android:layout_height="match_parent"
14    android:orientation="vertical"
15    android:padding="16dp">
16
17    <TextView
18      android:layout_width="wrap_content"
19      android:layout_height="wrap_content"
20      android:text="@{vm.user.name}" /> <!-- 단방향 바인딩 -->
21
22    <TextView
23      android:layout_width="wrap_content"
24      android:layout_height="wrap_content"
25      android:text="@{String.valueOf(vm.user.age)}" /> <!-- 정수를 문자열로 변환 -->
26
27    <EditText
28      android:layout_width="match_parent"
29      android:layout_height="wrap_content"
30      android:hint="Enter name"
31      android:text="@{viewModel.input}" /> <!-- 양방향 바인딩 -->
32
33    <Button
```

```
34         android:layout_width="wrap_content"
35         android:layout_height="wrap_content"
36         android:text="Update User"
37         android:onClick="@{() -> viewModel.updateUser(vm.user)}" /> <!-- 이벤트
38             ↳ 바인딩 -->
39
40     </LinearLayout>
41 </layout>
```

이 예제에서 User 객체는 XML 레이아웃에 바인딩됩니다. `vm.user.name` 및 `vm.user.age` 값은 `TextView` 컴포넌트에 동적으로 표시됩니다. 또한 `EditText`는 `ViewModel`의 `userName`과 양방향으로 바인딩되고, 버튼 클릭 시 `ViewModel`의 메서드를 호출합니다.

코드에서 데이터 바인딩하기

그림 134. `MainActivity.kt`

```
1 class MainActivity : AppCompatActivity() {
2     // ViewModel 인스턴스 (예: Hilt 또는 ViewModelProvider 사용)
3     private val viewModel: UserViewModel by viewModels()
4
5     override fun onCreate(savedInstanceState: Bundle?) {
6         super.onCreate(savedInstanceState)
7         // DataBindingUtil을 사용하여 레이아웃 설정 및 바인딩 객체 가져오기
8         val binding: ActivityMainBinding = DataBindingUtil.setContentView(this,
9             ↳ R.layout.activity_main)
10
11         // 바인딩 변수에 ViewModel 및 데이터 모델 설정
12         binding.viewModel = viewModel
13
14         // LifecycleOwner 설정 (LiveData 바인딩 등에 필요)
15         binding.lifecycleOwner = this
16     }
}
```

```
17  
18 // 예시 User 및 ViewModel  
19 data class User(val name: String, val age: Int)  
20  
21 class UserViewModel : ViewModel() {  
    private val _user = MutableLiveData<User>(User("Alice", 25)) // 사용자 데이터용  
    val user: LiveData<User> = _user  
24  
25     // input MutableLiveData가 EditText와 양방향 바인딩되어 있으므로  
26     // EditText 내용이 변경되면 input 값이 자동으로 업데이트됩니다.  
27     val input = MutableLiveData<String>() // 양방향 바인딩용  
28  
29     fun updateUser(user: User?) {  
        // 사용자 업데이트 로직 (예시)  
        Log.d("DataBinding", "Update user clicked for: ${user?.name}")  
        _user.value?.let { currentUser ->  
            val updatedUser = currentUser.copy(name = user?.name.orEmpty())  
            _user.value = updatedUser  
        }  
    }  
37 }
```

여기서 user 객체는 레이아웃의 데이터 소스로 설정되고, 데이터가 변경되면 UI가 자동으로 업데이트됩니다. 이는 데이터 바인딩 자체적으로 LiveData 또는 StateFlow를 XML에서 사용하면 주어진 lifecycle에 따라 값을 구독하고, 값이 업데이트될 때마다 자동적으로 갱신시켜 주기 때문입니다.

DataBinding의 특징

1. **양방향 데이터 바인딩(Two-Way Data Binding)**: UI와 기본 데이터 모델 간의 데이터 자동 동기화를 가능하게 합니다. 해당 방식은 입력 필드 값을 업데이트 하는 등에 특히 유용합니다.

```

1 <EditText
2     android:layout_width="match_parent"
3     android:layout_height="wrap_content"
4     android:text="@{vm.input}" /> <!-- @= 기호 사용 -->

```

2. **바인딩 표현식(Binding Expressions)**: 문자열 연결 또는 조건문과 같은 간단한 로직을 XML에서 직접 사용할 수 있습니다.

```

1 <TextView
2     android:layout_width="wrap_content"
3     android:layout_height="wrap_content"
4     android:text="@{user.age > 18 ? `성인` : `미성년자`}"
5     android:visibility="@{user.isAdmin ? View.VISIBLE : View.GONE}" />

```

3. **생명주기 인식(Lifecycle Awareness)**: 생명주기가 적절한 상태일 때만(가령, Activity 또는 Fragment가 활성 상태일 때) UI를 자동으로 업데이트합니다. (LiveData 또는 StateFlow와 함께 사용할 때).

DataBinding의 장점

- 보일러 플레이트 코드 감소**: findViewById() 및 명시적인 UI 업데이트가 필요 없어집니다.
- 실시간 UI 업데이트**: 데이터 변경 사항을 UI에 자동으로 반영합니다.
- 선언적 UI**: 로직을 XML로 이동하여 잘 사용하면 복잡한 레이아웃을 단순화할 수 있습니다.
- 테스트 용이성 향상**: UI와 코드를 분리하여 둘 다 독립적으로 테스트하기 쉽게 만듭니다.

DataBinding의 단점

- 성능 오버헤드**: ViewBinding과 같은 더 가벼운 솔루션에 비해 더 많은 런타임 오버헤드가 발생합니다.
- 복잡성**: 작거나 간단한 프로젝트에는 불필요한 복잡성을 유발할 수 있습니다.
- 학습 곡선**: 바인딩 표현식 및 생명주기 관리에 대한 러닝 커브가 요구됩니다.

요약

DataBinding을 사용하면 UI 요소를 XML 레이아웃의 데이터 소스에 직접 바인딩하여 보일러플레이트 코드를 줄이고 선언적 UI 프로그래밍을 일부 가능하게 합니다. 양방향 데이터 바인딩 및 바인딩 표현식과 같은 고급 기능을 지원하여 동적으로 UI를 업데이트할 수 있습니다. 반면, 초기 학습 곡선을 포함하여 복잡성과 성능 오버헤드를 더할 수 있지만, 레이아웃에 데이터를 동적으로 바인딩하고 관찰 및 업데이트를 하도록 하여 선언형 UI를 부분적으로 구현할 수 있다는 점과, 복잡한 레이아웃 코드를 XML에 단순화할 수 있다는 장점이 있습니다.

실전 질문

Q) DataBinding과 ViewBinding의 주요 차이점은 무엇이며, 어떤 시나리오에서 각각을 선택하는 것이 좋을까요?

Q) MVVM 아키텍처에서 DataBinding은 어떤 역할을 하며, 안드로이드 개발에서 UI 로직과 비즈니스 로직을 분리하는 데 어떻게 활용할 수 있나요?

Pro Tips for Mastery: ViewBinding과 DataBinding의 차이점은 무엇인가요?

ViewBinding과 **DataBinding**은 모두 앱에서 뷰 작업을 할 때 null 안정성 및 효율적으로 레이아웃 요소에 접근하기 위해 안드로이드에서 제공하는 라이브러리입니다. 두 라이브러리는 유사하지만 서로 다른 목적을 가지고 있으며 각자 고유한 기능을 제공합니다.

ViewBinding은 `findViewById` 없이 레이아웃의 뷰에 접근하는 것을 단순화하기 위한 목적을 가지고 있습니다. 각 XML 레이아웃 파일에 대해 바인딩 클래스를 생성하며, 이 클래스에는 `id`가 있는 모든 뷰에 대한 직접 참조를 제공합니다. 따라서, 타입 안전성을 향상시키고 보일러 플레이트 코드를 줄입니다.

ViewBinding의 주요 특징은 아래와 같습니다.

- XML 레이아웃 파일에 대한 바인딩 클래스를 생성합니다.
- 레이아웃의 뷰에 대한 직접 참조를 제공하여 `findViewById`를 사용하지 않고 안전하게 레이아웃 요소에 접근할 수 있습니다.
- `nullable` 및 뷰 유형에 대한 컴파일 타임 검사를 제공하여 타입 안전성을 보장합니다.
- 바인딩 표현식이나 데이터 기반 업데이트와 같은 고급 기능은 지원하지 않습니다.

반면에 **DataBinding**은 UI 컴포넌트를 데이터 소스에 직접 바인딩할 수 있는 더 복잡하고 유연한 라이브러리입니다. 바인딩 표현식, 관찰 가능한 데이터 및 양방향 데이터 바인딩을 지원하여 MVVM 아키텍처 구현 등에 적합합니다.

DataBinding의 주요 특징은 아래와 같습니다.

- XML에서 UI 요소를 데이터 소스에 바인딩할 수 있습니다.
- UI 컴포넌트를 동적으로 업데이트하기 위한 바인딩 표현식을 지원합니다.
- UI와 데이터 간의 실시간 동기화를 위한 양방향 데이터 바인딩을 제공합니다.
- LiveData 및 StateFlow와 같이 생명주기에 따라 관찰 가능한 타입의 데이터 통합을 제공합니다.

주요 차이점

1. **목적:** ViewBinding은 뷰 접근을 단순화하는 반면, DataBinding은 고급 데이터 기반 UI 바인딩을 가능하게 합니다.
2. **컴파일 타임 클래스 생성:** ViewBinding은 뷰에 대한 직접 참조를 생성합니다. DataBinding은 뷰에 대한 직접 참조뿐만 아니라 내장 데이터 바인딩 기능이 있는 추가 클래스도 생성합니다.
3. **표현식:** ViewBinding은 XML에서 표현식을 지원하지 않지만, DataBinding은 바인딩 표현식과 동적 데이터 바인딩을 지원합니다.
4. **양방향 바인딩:** DataBinding만 양방향 바인딩을 지원합니다.
5. **성능:** ViewBinding은 데이터 바인딩 로직을 처리하지 않으므로 더 빠르고 오버헤드가 적습니다.

요약

특히 간단한 프로젝트에서 findViewById 없이 간단한 뷰 참조가 필요할 때는 **ViewBinding**을 사용합니다. 복잡한 데이터 기반 UI 또는 MVVM 아키텍처에서 작업할 때는 **DataBinding**이 더 효율적인 상황이 많습니다. 동적 바인딩 기능을 제공하고 LiveData, StateFlow 및 @Bindable 어노테이션을 통해 관찰 가능한 속성 또는 메서드를 커스텀하여 구현할 수 있기 때문입니다. 따라서, DataBinding은 더 다양한 기능들을 제공하지만 추가 오버헤드가 발생할 수 있으므로, ViewBinding 사용만으로도 충분한 간단한 프로젝트에는 필요하지 않을 수 있습니다.

Q) 53. LiveData에 대해서 설명해 주세요.

LiveData⁹³는 안드로이드 Jetpack 아키텍처 컴포넌트에서 제공하는 관찰 가능한 데이터 헌터 클래스입니다. 이는 생명주기를 인식하므로 Activity, Fragment, View와 같이 연관된 안드로이드 컴포넌트의 생명주기에 따라 동작이 달라집니다. 따라서, LiveData는 연관된 컴포넌트가 활성 생명주기 상태(가령, started 또는 resumed)일 때만 데이터를 관찰하고 UI를 업데이트하도록 보장합니다.

LiveData의 주요 목적은 UI 컴포넌트가 데이터 변경 사항을 관찰하고 해당 데이터가 변경될 때마다 UI를 반응형으로 업데이트할 수 있도록 하는 것입니다. 이는 안드로이드 애플리케이션에서 반응형 UI 패턴을 구현하는 데 좋은 역할을 합니다.

LiveData는 아래와 같은 이점을 제공합니다.

- 생명주기 인식(Lifecycle Awareness)**: LiveData는 컴포넌트의 생명주기를 관찰하고 컴포넌트가 활성 상태일 때만 데이터를 업데이트하여 크래시 및 메모리 누수 위험을 줄입니다.
- 자동 정리(Automatic Cleanup)**: 컴포넌트에 연결된 관찰자는 주어진 생명주기가 소멸될 때 자동으로 제거되고 정리됩니다.
- 관찰자 패턴(Observer Pattern)**: UI 컴포넌트는 관찰자를 활용하여 LiveData의 데이터가 변경될 때 자동으로 업데이트됩니다.
- 스레드 안전성(Thread Safety)**: LiveData는 스레드 안전하도록 설계되어 백그라운드 스레드에서 업데이트할 수 있습니다.

다음 예제는 UI 관련 데이터를 관리하기 위해 ViewModel에서 LiveData를 사용하는 방법을 보여줍니다.

⁹³<https://developer.android.com/topic/libraries/architecture/livedata>

그림 135. LiveData Example

```
1 // ViewModel
2 class MyViewModel : ViewModel() {
3     // 내부 수정을 위한 MutableLiveData
4     private val _data = MutableLiveData<String>()
5     // 외부 수정을 방지하기 위해 LiveData로 노출
6     val data: LiveData<String> get() = _data
7
8     fun updateData(newValue: String) {
9         // LiveData 값 업데이트
10        _data.value = newValue
11    }
12 }
13
14 // Fragment 또는 Activity
15 class MyFragment : Fragment() {
16     private val viewModel: MyViewModel by viewModels()
17
18     override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
19         super.onViewCreated(view, savedInstanceState)
20
21         // LiveData 관찰
22         viewModel.data.observe(viewLifecycleOwner) { updatedData ->
23             // 새 데이터로 UI 업데이트
24             textView.text = updatedData
25         }
26     }
27 }
```

이 예제에서 MyViewModel은 데이터를 보유하고 Fragment는 LiveData 객체를 관찰합니다. updateData 함수가 호출될 때마다 UI가 자동으로 업데이트됩니다.

MutableLiveData와 LiveData의 차이점

- **MutableLiveData**: setValue() 또는 postValue()를 통해 데이터 수정을 허용합니다. 일반적으로 외부에서의 직접적인 수정을 방지하기 위해 ViewModel 내에서 비공개로 유지됩니다.
- **LiveData**: 외부 컴포넌트가 데이터를 수정하는 것을 방지하는 읽기 전용 버전의 LiveData로, 더 나은 캡슐화를 보장합니다.

LiveData 사용 사례

LiveData는 일반적으로 아래 상황들에서 주로 사용됩니다.

1. **UI 상태 관리**: LiveData는 네트워크 응답이나 데이터베이스와 같은 소스의 데이터를 담는 컨테이너 역할을 하여 UI 컴포넌트에 원활하게 바인딩될 수 있도록 합니다. 이를 통해 기본 데이터가 변경될 때마다 UI가 자동으로 업데이트되어 인터페이스가 앱 상태와 동기화되도록 보장합니다.
2. **관찰자 패턴 구현**: LiveData는 발행자(publisher) 역할을 하고 Observer 인터페이스 구현이 구독자(subscriber) 역할을 하는 관찰자 패턴을 따릅니다. 이 디자인 패턴은 LiveData 값이 변경될 때마다 구독자에게 실시간 업데이트를 용이하게 하여 동적 UI 업데이트나 데이터 기반 상호 작용과 같은 시나리오에 매우 적합합니다.
3. **일회성 이벤트**: 토스트를 노출시키거나 다른 화면으로 이동하는 one-off 일회성 이벤트에도 사용될 수 있습니다. 단, 이러한 경우는 [SingleLiveEvent⁹⁴](#) 또는 유사한 구현으로 커스텀하여 처리해야 합니다.



Additional Tips: LiveData는 Kotlin의 StateFlow/SharedFlow와 자주 비교되곤 하며, 많은 개발자들이 이미 Flow로 마이그레이션을 진행했거나 도입을 고려 중인 것으로 보여집니다. StateFlow의 등장으로 인해 LiveData는 deprecated 되어야 하는 API라고 주장하는 개발자들이 더러 있는데, 이는 명백하게 잘못된 의견입니다. Flow는 안드로이드와 완전히 독립적인 API이고 안드로이드 컴포넌트에 대한 수명주기를 전혀 알지 못합니다.

따라서 Flow는 구독 (collect) 및 구독 해지를 안드로이드 라이프 사이클에 따라 올바르게

⁹⁴<https://gist.github.com/skydoves/60f83bf678803e3b65742d541aba935f#file-singleliveevent-kt>

하지 않으면, 화면이 활성 상태가 아닌 상태에도 데이터를 지속적으로 수집하는 등 메모리 누수로 이어질 가능성이 높습니다. 반면, LiveData는 구독 시점부터 애초에 lifecycle 인스턴스를 결합하도록 하여, 개발자가 적절한 시기에 구독해지를 해주지 않아도 생명주기에 따라 안전하게 데이터를 수집하기 때문에 상황에 따라서 LiveData를 활용하는 편이 더 편하고 올바른 선택이 될 수도 있습니다.

한편, “안드로이드의 의존성을 제거하기 위해 LiveData에서 Flow로 마이그레이션 한다”라는 의견 또한 특별히 설득력이 없는 주장입니다. 일반적으로 LiveData 타입이 네트워크 모듈의 응답 타입으로 정의되어 ViewModel까지 이어지는데, Hilt를 비롯하여 모든 Jetpack 관련 라이브러리를 모두 제거하면서까지 JVM 타입의 멀티 모듈 설계 (network, data, domain 레이어에 한하여) 통하여 얻어갈 수 있는 성능적 & 구조적인 이점에 대해서는 아직 통계적으로 명확히 밝혀진 것이 없기 때문입니다.

따라서, 다수의 선택을 단순히 따라가기 보다는, 데이터에 대해 복잡한 가공처리가 필요하거나 Flow를 레버리지 할 수 있는 명백한 상황에서 Flow를 채택하는 것이 올바른 선택입니다. 안드로이드에서 Flow를 안전하게 구독하는 방법에 관해서는 Manuel Vivo의 [A safer way to collect flows from Android UIs⁹⁵](#)를 참고하시길 바랍니다.

요약

LiveData는 안드로이드에서 반응형 및 생명주기를 인식하는 UI 상태를 구축하는데 효율적입니다. 생명주기를 고려하는 방식으로 데이터 변경 사항을 안전하게 관리하고 관찰함으로써 크래시 또는 메모리 누수 가능성을 줄입니다. 이로 인해 LiveData는 특히 MVVM 아키텍처에서 최신 안드로이드 개발에서 자주 사용되고 있습니다.

실전 질문

Q) LiveData는 생명주기 인식을 어떻게 보장하며, RxJava 또는 EventBus와 같은 전통적인 observable한 객체와 비교하여 어떤 이점을 제공하나요?

Q) LiveData에서 setValue()와 postValue()의 차이점은 무엇이며, 각각 언제 사용해야 하나요?

⁹⁵<https://medium.com/androiddevelopers/a-safer-way-to-collect-flows-from-android-uils-23080b1f8bda>

Q) LiveData의 한계는 무엇이며, 구성 변경 시 다시 트리거되지 않고 내비게이션 또는 토스트 메시지 표시와 같은 여러 UI 이벤트를 관찰해야 하는 경우에 어떻게 처리해야 하나요?

 **Pro Tips for Mastery:** LiveData에서 setValue()와 postValue() 메서드의 차이점은 무엇인가요?

LiveData에서 setValue()와 postValue() 메서드는 LiveData 객체가 보유한 데이터를 업데이트하는 데 사용되지만, 특히 스레딩 및 동기화 측면에서 다른 사용 사례와 동작을 가집니다.

1. setValue()

setValue() 메서드는 데이터를 동기적으로 업데이트하며 **메인 스레드**(UI 스레드)에서만 호출할 수 있습니다. 값을 즉시 업데이트하고 변경 사항이 동일한 프레임 동안 관찰자에게 반영되도록 해야 할 때 사용됩니다.

그림 136. Example of ‘setValue()’

```

1 val liveData = MutableLiveData<String>()
2
3 fun updateOnMainThread() {
4     if (Looper.myLooper() == Looper.getMainLooper()) {
5         liveData.setValue("Updated Value") // 메인 스레드에서만 작동
6     } else {
7         // 메인 스레드가 아니면 예외 발생 가능성
8         Log.e("LiveData", "Cannot call setValue on a background thread")
9     }
10 }
```

해당 메서드는 UI 이벤트나 안드로이드 생명주기 컴포넌트와 상호 작용할 때와 같이 이미 메인 스레드에서 작업 중인 경우에 적합합니다. 백그라운드 스레드에서 setValue()를 호출하려고 하면 예외가 발생합니다.

2. postValue()

postValue() 메서드는 데이터를 비동기적으로 업데이트하는 데 사용되므로 백그라운드 스레드에서 UI를 업데이트 해야하는 경우에 적합합니다. 호출되면 메인 스레드에서 업데이트가 발생하도록 예약하여 현재 스레드를 차단하지 않고 스레드 안전성을 보장합니다.

그림 137. Example of ‘postValue()’

```

1 val liveData = MutableLiveData<String>()
2
3 fun updateInBackground() {
4     Thread {
5         // 백그라운드 스레드에서 값 전달
6         liveData.postValue("Updated Value") // 모든 스레드에서 호출 가능
7     }.start()
8 }
```

`postValue()`는 네트워크 요청이나 데이터베이스 쿼리와 같은 백그라운드 작업과 관련된 시나리오에서 특히 유용합니다. 메인 스레드로 명시적으로 전환할 필요가 없기 때문입니다.

`postValue()` 메서드의 내부 구현을 살펴보면 백그라운드 실행자를 활용하여 값을 메인 스레드에 전달합니다.

그림 138. postValue internals

```

1 protected void postValue(T value) {
2     boolean postTask;
3     synchronized (mDataLock) {
4         postTask = mPendingData == NOT_SET; // 이전에 전달된 작업이 없는지 확인
5         mPendingData = value; // 보류 중인 데이터 업데이트
6     }
7     if (!postTask) {
8         return; // 이미 전달된 작업이 있으면 반환
9     }
10    // 메인 스레드 실행자에 Runnable 실행
11    ArchTaskExecutor.getInstance().postToMainThread(mPostValueRunnable);
12 }
```

이 메서드는 먼저 `mDataLock`에서 동기화하고 `mPendingData`를 업데이트하여 작업을 실행해야 하는지 확인합니다. 값이 이미 보류 중이면 중복 실행을 피합니다. 그렇지 않으면 `mPostValueRunnable`을 `ArchTaskExecutor`를 통해 메인 스레드에서 실행되도록 예약하여 스레

드 안전 업데이트를 보장합니다. 이를 통해 백그라운드 스레드에서 안전하게 업데이트를 게시하면서 스레드 안전성을 유지할 수 있습니다.

반면에 다음 코드가 메인 스레드에서 실행되면 어떤 상황이 발생할까요?

```
1 liveData.postValue("a")
2 liveData.setValue("b")
```

값 "b"가 즉시 반영되고, 나중에 메인 스레드가 전달 받은 작업을 처리할 때 "b"를 "a"로 덮어씁니다. 이 동작은 `postValue()`가 업데이트를 비동기적으로 예약하는 반면 `setValue()`는 메인 스레드에서 동기적으로 값을 업데이트하기 때문에 발생합니다. 또한 `postValue()`는 메인 스레드가 게시된 작업을 실행하기 전에 `postValue()`가 여러 번 호출되면 `mPendingData`에 가장 최근 값만 유지하므로 마지막 값만 전달됩니다.

주요 차이점

Aspect	<code>setValue()</code>	<code>postValue()</code>
Thread	메인 스레드에서 호출해야 함	모든 스레드에서 호출 가능
Synchronous / Asynchronous	값을 즉시 동기적으로 업데이트함	메인 스레드에서 업데이트를 비동기적으로 예약함
Use Case	UI 업데이트 또는 메인 스레드에서 시작된 변경	백그라운드 스레드 업데이트 또는 비동기 작업
Observer	동일한 프레임 동안	메인 스레드 처리 후
Notification	관찰자를 즉시 트리거함	다음 프레임에서 관찰자를 트리거함

일반적인 사용 패턴

- **setValue() 사용:** 사용자 상호 작용이나 생명주기 기반 이벤트와 같이 업데이트가 메인 스레드에서 직접 트리거될 때 적합합니다.

- **postValue() 사용:** 데이터베이스에서 데이터를 쿼리하거나, 네트워크 호출 수행 또는 그 외 장기적으로 실행되는 작업과 같이 백그라운드 스레드로 작업 중이면서 값을 업데이트 해야할 때 적합합니다.

요약

setValue()와 postValue()는 모두 LiveData 객체의 값을 업데이트하는 역할을 하지만 스레딩 동작이 다릅니다. setValue()는 동기적이며 메인 스레드로 제한되는 반면, postValue()는 비동기적이며 모든 스레드에서 호출할 수 있습니다. 적절한 메서드를 선택하면 안드로이드 애플리케이션에서 스레드 안전성과 원활한 데이터 업데이트를 보장할 수 있습니다.

Q) 54. Jetpack ViewModel에 대해 설명해 주세요.

Jetpack ViewModel은 생명주기를 인식하는 방식으로 UI 관련 데이터를 저장하고 관리하도록 설계된 안드로이드 아키텍처 컴포넌트의 핵심 구성 요소입니다. 화면 회전과 같은 구성 변경 시에도 데이터가 유지되도록 보장하면서 UI 로직과 비즈니스 로직을 분리하여 개발자가 견고하고 유지 가능한 앱을 만드는 데 도움을 줍니다.

ViewModel의 주요 목표는 구성 변경 중에 UI 관련 데이터를 보존하는 것입니다. 예를 들어, 사용자가 기기를 회전하면 Activity 또는 Fragment가 소멸되고 다시 생성되지만 ViewModel은 파괴되지 않아 데이터가 그대로 유지되도록 보장합니다.

그림 139. ViewModel Example

```
1 data class DiceUiState(  
2     val firstDieValue: Int? = null,  
3     val secondDieValue: Int? = null,  
4     val numberofRolls: Int = 0,  
5 )  
6  
7 class DiceRollViewModel : ViewModel() {  
8  
9     // 화면 UI 상태 노출  
10    private val _uiState = MutableStateFlow(DiceUiState())  
11    val uiState: StateFlow<DiceUiState> = _uiState.asStateFlow()
```

```
12
13     // 비즈니스 로직 처리
14     fun rollDice() {
15         _uiState.update { currentState ->
16             currentState.copy(
17                 firstDieValue = Random.nextInt(from = 1, until = 7),
18                 secondDieValue = Random.nextInt(from = 1, until = 7),
19                 numberOfRolls = currentState.numberOfRolls + 1,
20             )
21         }
22     }
23 }
```

이 예제에서 ViewModel의 상태 값은 Activity가 구성 변경으로 인해 다시 생성되더라도 유지됩니다.

ViewModel의 특징

1. **생명주기 인식(Lifecycle Awareness)**: ViewModel은 Activity 또는 Fragment의 생명주기에 범위가 지정됩니다. 사용자가 화면에서 벗어나는 등 연관된 UI 컴포넌트가 더 이상 사용되지 않을 때 자동으로 소멸됩니다.
2. **구성 변경 간 지속성(Persistence Across Configuration Changes)**: 구성 변경 중에 소멸되고 다시 생성되는 Activity 또는 Fragment와 달리 ViewModel은 상태를 유지하여 데이터 손실을 방지하고 데이터의 반복적인 재로드를 피합니다.
3. **관심사 분리(Separation of Concerns)**: ViewModel은 UI 관련 로직과 비즈니스 로직을 분리하여 더 깔끔하고 유지 관리하기 쉬운 코드를 설계하는 데 도움이 됩니다. UI 레이어는 ViewModel에서 업데이트를 관찰하므로 반응형 프로그래밍 원칙을 구현하기가 더 쉬워집니다.

ViewModel 생성 및 사용

Jetpack activity-ktx 라이브러리⁹⁶에서 제공하는 ComponentActivity의 확장함수인 `viewModels()` 텔리게이트를 사용하여 ViewModel을 보다 쉽게 생성할 수 있습니다. 해

⁹⁶<https://developer.android.com/jetpack/androidx/releases/activity>

당 확장 함수는 ViewModel 생성을 단순화합니다.

그림 140. Using ViewModel in an Activity

```
1 class DiceRollActivity : AppCompatActivity() {  
2  
3     // activity-ktx 아티팩트의 'viewModels()' 델리게이트 함수 사용  
4     private val viewModel: DiceRollViewModel by viewModels()  
5  
6     override fun onCreate(savedInstanceState: Bundle?) {  
7         super.onCreate(savedInstanceState) // super.onCreate 호출 추가  
8  
9         // 시스템이 액티비티의 onCreate() 메서드를 처음 호출할 때 ViewModel 생성.  
10        // 다시 생성된 액티비티는 첫 번째 액티비티에서 생성된 동일한 DiceRollViewModel  
11        // 인스턴스를 받습니다.  
12  
13        lifecycleScope.launch {  
14            repeatOnLifecycle(Lifecycle.State.STARTED) {  
15                viewModel.uiState.collect { uiState -> // 수집된 상태 사용  
16                    // UI 요소 업데이트 (예: 텍스트뷰 업데이트)  
17                    // updateUi(uiState)  
18                }  
19            }  
20        }  
21    }
```

ViewModel 인스턴스는 ViewModel 인스턴스의 생명주기를 관리하기 위한 메커니즘 역할을 하는 ViewModelStoreOwner에 스코프가 지정됩니다. ViewModelStoreOwner는 Activity, Fragment, Navigation 그래프, Navigation 그래프 내 대상 또는 개발자가 정의한 커스텀 ViewModelStoreOwner가 될 수도 있습니다. Jetpack 라이브러리는 다양한 사용 사례를 충족하기 위해 ViewModel 스코프를 지정하는 다양한 옵션을 제공합니다. 포괄적인 개요는 아래 일러스트에서 제공하는 [ViewModel API 치트 시트](#)⁹⁷를 참조할 수 있습니다.

⁹⁷<https://developer.android.com/topic/libraries/architecture/viewmodel/viewmodel-cheatsheet>

ViewModel APIs cheat sheet

v1.0 by @AndroidDev

This cheat sheet lists the ViewModel APIs in Jetpack. It includes the artifact where they can be found, the scope of the returned ViewModel instance, and an example of how to use them.

You can scope ViewModel instances to a `ViewModelStoreOwner`.
The owner can be an Activity, Fragment, Navigation graph, a destination of a Navigation graph, or your own custom owner.

For more use cases and to learn about ViewModel factories:
[goo.gl/architecture-viewmodel](https://developer.android.com/architecture/viewmodel)

Get an instance in Jetpack Compose

Compose - lifecycle.lifecycle-viewmodel-compose

```
= viewModel()           closest ViewModelStoreOwner
@Composable
fun MyScreen(vm: ViewModel = viewModel()) { ... }

= viewModel(vmStoreOwner)      any ViewModelStoreOwner
// Example of getting a ParentViewModel instance scoped to the Navigation graph
composable("myScreen") { backStackEntry:BackStackEntry ->
    val parentEntry = remember(backStackEntry) {
        navController.getBackStackEntry("parentNavigationRoute")
    }
    val parentViewModel = viewModel(parentEntry)
    ...
}
```

Compose + Navigation + Hilt - hilt.hilt-navigation-compose

When using Hilt and Navigation together, use this API instead of `viewModel()`

```
= hiltViewModel()          closest ViewModelStoreOwner
@Composable
fun MyScreen(vm: ViewModel = hiltViewModel()) { ... }
```

Core ViewModel API

ViewModel - lifecycle:lifecycle-viewmodel-ktx

```
ViewModelProvider      any ViewModelStoreOwner
val vm = ViewModelProvider(anyViewModelStoreOwner)[MyViewModel::class.java]
```

The APIs and artifacts listed in this page follow this template

Library - artifact

ViewModel API

Scope of the returned ViewModel instance

Example code

Get an instance in an Activity

Activity - activity.activity-ktx

```
by viewModels()           Activity (ViewModelStoreOwner)
val vm: MyViewModel by viewModels()
```

Get an instance in a Fragment

Fragment - fragment.fragment-ktx

```
by viewModels()           Fragment (ViewModelStoreOwner)
val vm: MyViewModel by viewModels()
```

```
by viewModels(ownerProducer)      any ViewModelStoreOwner
// Example of getting a SharedViewModel instance scoped to the parent Fragment
val vm: SharedViewModel by viewModels(ownerProducer = { requireParentFragment() })
```

```
by activityViewModels()          Activity (ViewModelStoreOwner)
val vm: SharedViewModel by activityViewModels()
```

Fragment + Navigation - navigation.navigation-fragment

```
by navGraphViewModels(graphId)      NavGraph (ViewModelStoreOwner)
val vm: SharedViewModel by navGraphViewModels(R.id.nav_graph)
```

Fragment + Navigation + Hilt - hilt.hilt-navigation-fragment

When using Hilt and Navigation together, use this API instead of `navGraphViewModels()`. To get an instance not scoped to a NavGraph, you can keep using `viewModels()` or `activityViewModels()`.

```
by hiltNavGraphViewModels(graphId)      NavGraph (ViewModelStoreOwner)
val vm: SharedViewModel by hiltNavGraphViewModels(R.id.nav_graph)
```

그림 141. viewmodel-apis

요약

Jetpack ViewModel은 구성 변경 시에도 원활하게 유지되도록 보장하면서 UI 상태 관련 데이터를 저장하고 관리하도록 설계된 Jetpack의 핵심 구성 요소입니다. 안드로이드 컴포넌트의 생명주기를 인식하고 MVVM 아키텍처 패턴과 효과적으로 통합되어 화면 회전과 같은 이벤트 중에 데이터를 유지함으로써 상태 관리를 단순화하고 전반적인 개발 경험을 향상시킵니다.

실전 질문

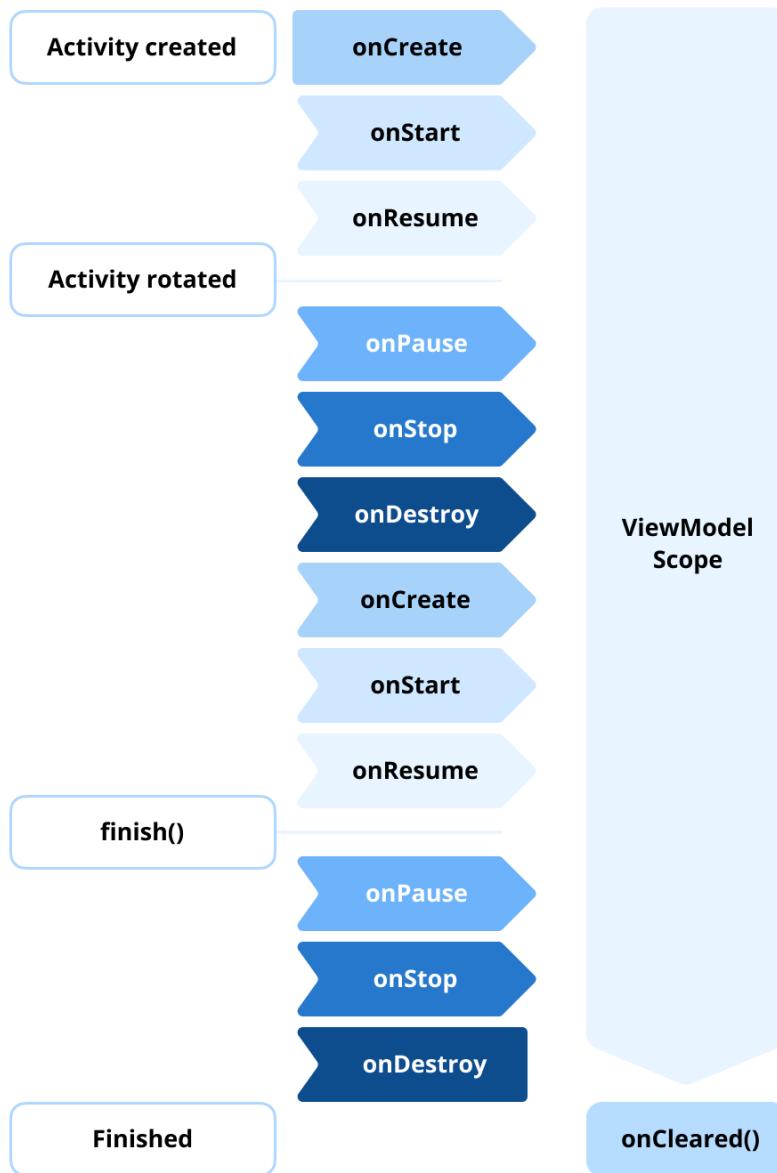
- Q) ViewModel은 구성 변경 시 데이터를 어떻게 유지하며, `onSaveInstanceState()`를 사용하여 상태를 저장하는 것과 어떻게 다른가요?
- Q) `ViewModelStoreOwner`의 목적은 무엇이며, 동일한 Activity 내의 여러 Fragment 간에 ViewModel을 어떻게 공유할 수 있나요?
- Q) UI 상태 관리를 위해 ViewModel 내에서 StateFlow 또는 LiveData를 사용하는 것의 장점과 잠재적인 단점은 무엇인가요?

Pro Tips for Mastery: ViewModel의 생명주기는 어떻게 되나요?

ViewModel의 생명주기는 `ViewModelStoreOwner`(Activity, Fragment 또는 기타 생명주기 인식 컴포넌트일 수 있음)에 연결됩니다. ViewModel은 `ViewModelStoreOwner`의 범위 내에서 존재하며, 화면 회전과 같은 구성 변경 시에도 데이터와 상태가 유지되도록 보장합니다.

이러한 생명주기에 기반한 설계는 ViewModel을 UI 관련 데이터를 효율적으로 관리하고, 변경 사항 전반에 걸쳐 데이터를 보존하는 데 중요한 역할을 합니다. 예를 들어, Activity의 경우, ViewModel은 Activity가 완전히 파괴되고 메모리에서 제거될 때까지 유지됩니다.

아래 그림은 Activity 생명주기와 관련하여 ViewModel의 라이프 사이클을 간략하게 보여줍니다. ViewModel 인스턴스가 어떻게 생성되고 Activity 생명주기 이벤트를 통해 유지되며, Activity가 일시적으로 소멸되고 다시 생성되어도 ViewModel이 여전히 살아있는 것을 보여줍니다. 해당 예제에서는 Activity를 사용하지만 동일하게 `ViewModelStoreOwner` 역할을 하는 Fragment 및 기타 생명주기와 밀접하게 관련된 컴포넌트(가령, 내비게이션 호스트 등)에서도 마찬가지로 비슷하게 작동합니다.

그림 142. `viewmodel-lifecycle`

ViewModelStoreOwner가 처음 생성될 때 ViewModel 인스턴스가 초기화됩니다. 소유자가 메모리에 남아 있는 한 동일한 ViewModel 인스턴스가 유지됩니다. 기기 회전과 같은 구성 변경이 발생하면 소유자는 다시 생성되지만 기존 ViewModel 인스턴스가 재사용되어 데이터를 다시

로드하거나 다시 초기화할 필요가 없습니다. ViewModel의 이러한 재사용은 더 나은 성능과 더 부드러운 사용자 경험을 보장합니다.

마지막으로 ViewModel은 ViewModelStoreOwner가 영구적으로 소멸될 때 비로소 제거됩니다. 예를 들어, Activity가 완료되거나 Fragment가 부모에서 제거되고 돌아올 것으로 예상되지 않으면 ViewModel의 onCleared() 메서드가 호출됩니다. 이는 코루틴 작업의 취소나 데이터 베이스 연결을 끊는 등과 같이 리소스를 정리하여 메모리 누수를 방지하기 위해 유용합니다. ViewModel의 생명주기는 안드로이드 애플리케이션에 대한 효율적인 리소스 관리 및 상태 지속성을 보장합니다.

자세한 내용은 [공식 문서](#)⁹⁸를 참조하세요.

Pro Tips for Mastery: 구성 변경 후에도 ViewModel이 어떻게 유지될 수 있나요?

안드로이드에서 Jetpack ViewModel은 화면 회전이나 기기 언어 업데이트와 같은 구성 변경 시에도 살아남도록 설계되었습니다. ViewModel이 UI 컴포넌트(Activity 또는 Fragment와 같음)를 위해 생성될 때 컴포넌트의 **생명주기 소유자(lifecycle owner)**에 연결됩니다. Activity의 경우 이 생명주기는 ComponentActivity 클래스에 포함되어 있으며, Fragment의 경우 Fragment 클래스에 포함되어 있습니다. 핵심 요소는 구성 변경 시 유지되어 ViewModel이 다시 생성되지 않고 데이터를 유지할 수 있도록 하는 ViewModelStore입니다.

Jetpack 라이브러리의 내부 구현을 살펴보면 [androidx.activity.ComponentActivity](#)⁹⁹와 [androidx.fragment.app.Fragment](#)¹⁰⁰가 모두 ViewModelStoreOwner¹⁰¹ 인터페이스를 구현한다는 것을 알 수 있습니다. 해당 인터페이스를 통해 Activity와 Fragment는 자체적인 ViewModelStore를 가질 수 있으며, 이는 ViewModel 인스턴스가 구성 변경 시에도 유지되도록 보장합니다. ViewModelStore는 아래 코드에서 볼 수 있듯이 String과 ViewModel이 1:1 쌍을 이루도록 Map 형태로서 ViewModel 인스턴스를 관리합니다.

⁹⁸<https://developer.android.com/topic/libraries/architecture/viewmodel#lifecycle>

⁹⁹<https://cs.android.com/androidx/platform/frameworks/support/+/androidx-main:activity/activity/src/main/java/androidx/activity/ComponentActivity.kt;l=111?q=ComponentActivity>

¹⁰⁰<https://cs.android.com/androidx/platform/frameworks/support/+/androidx-main:fragment/fragment/src/main/java/androidx/fragment/app/Fragment.java;l=126?q=androidx.fragment.app.Fragment&sq=>

¹⁰¹<https://developer.android.com/reference/androidx/lifecycle/ViewModelStoreOwner>

그림 143. ViewModelStore.kt

```
1 public open class ViewModelStore {
2
3     private val map = mutableMapOf<String, ViewModel>()
4
5     @RestrictTo(RestrictTo.Scope.LIBRARY_GROUP)
6     public fun put(key: String, viewModel: ViewModel) {
7         val oldViewModel = map.put(key, viewModel)
8         oldViewModel?.clear() // 이전 ViewModel 정리
9     }
10
11    @RestrictTo(RestrictTo.Scope.LIBRARY_GROUP)
12    public operator fun get(key: String): ViewModel? {
13        return map[key]
14    }
15
16    @RestrictTo(RestrictTo.Scope.LIBRARY_GROUP)
17    public fun keys(): Set<String> {
18        return HashSet(map.keys)
19    }
20
21    /**
22     * ViewModelStore에서 모든 ViewModel을 지웁니다.
23     * 이후에 저장된 ViewModel은 clear()가 호출될 때까지 유지됩니다.
24     */
25    public fun clear() {
26        for (vm in map.values) {
27            vm.clear() // 각 ViewModel의 onCleared() 호출
28        }
29        map.clear()
30    }
31 }
```

ComponentActivity의 또 다른 주목할 만한 부분은 자체 생명주기 상태를 관찰하는 능력입니다. Activity가 생성될 때, `onDestroy()`가 구성 변경으로 인해 트리거 되지 않은 경우 생명주기 상태가 `ON_DESTROY`로 전환되면 유지된 모든 ViewModel 인스턴스를 지우는 메커니즘으로 동작합니다. 이는 아래 코드에서 볼 수 있듯이 발생할 수 있는 생명주기 이벤트로부터 ViewModel 인스턴스를 보존하면서 효율적으로 리소스를 정리할 수 있도록 보장합니다.

그림 144. ComponentActivity ViewModelStore.kt

```
1 getLifecycle().addObserver(new LifecycleEventObserver() {
2     @Override
3     public void onStateChanged(@NonNull LifecycleOwner source,
4         @NonNull Lifecycle.Event event) {
5         if (event == Lifecycle.Event.ON_DESTROY) {
6             // 사용 가능한 컨텍스트 지우기
7             mContextAwareHelper.clearAvailableContext();
8             // 구성 변경 중이 아니라면 ViewModelStore 지우기
9             if (!isChangingConfigurations()) {
10                 getViewModelStore().clear();
11             }
12             mReportFullyDrawnExecutor.activityDestroyed();
13         }
14     }
15 });
```

요약

ViewModel은 Activity 및 Fragment와 같이 ViewModel 생명주기를 관리하도록 설계된 안드로이드 컴포넌트의 ViewModelStore에 저장되어 구성 변경 시에도 유지됩니다. 마찬가지로 Compose Navigation 라이브러리¹⁰²에서는 ViewModelStore가 내비게이션 경로에 스코프가 지정되어 특정 화면에 대해 ViewModel이 적절하게 유지되도록 보장합니다. 더 다양하고 특수한 사용 사례를 위해 ViewModelStore를 수동으로 관리하는 것도 가능하지만, 구성 변경 중 ViewModel 인스턴스를 복구하는 과정을 올바르게 구현하는 것이 생각보다 까다로워, 해당 방식은 일반적으로 권장되지는 않습니다.

¹⁰²<https://developer.android.com/develop/ui/compose/navigation>

💡 Pro Tips for Mastery: Jetpack ViewModel과 Microsoft에서 제시한 MVVM 아키텍처 ViewModel의 차이점에 대해서 설명해 주세요.

공식 문서¹⁰³에 따르면 Jetpack **ViewModel**은 **비즈니스 로직 또는 UI 상태 홀더** 역할을 하도록 설계된 생명주기를 인식하는 컴포넌트입니다. 관련 비즈니스 로직을 캡슐화하면서 상태를 관리하고 UI에 제공합니다. 장점은 상태를 보유하고 화면 회전이나 Activity 재생성과 같은 구성 변경 시에도 유지하는 기능에 있습니다. 이는 UI가 불필요하게 데이터를 다시 요청할 필요가 없도록 보장하여 효율성과 응답성을 향상시킵니다. 본질적으로 Jetpack ViewModel은 안드로이드 애플리케이션 내에서 생명주기를 인식하고, UI 상태를 관리하는 데 최적화되어 있습니다.

반면, 원래 Microsoft에서 도입한 **MVVM (Model-View-ViewModel) 아키텍처**는 **ViewModel**을 View와 Model 간의 다리 역할을 한다고 설명합니다. MVVM ViewModel은 View가 데이터 바인딩할 수 있는 속성과 명령을 구현하여 UI에 필요한 기능을 제공합니다. 또한 변경 사항에 대한 알림 이벤트를 통해 View에 상태 변경을 알립니다. MVVM의 ViewModel은 View와 Model 간의 상호 작용을 조정하고 UI에 필요한 비즈니스 로직을 추상화하거나 호출을 중개하는 역할을 합니다. 상태 관리 및 안드로이드 컴포넌트 생명주기를 인지하는 것에 중점을 둔 Jetpack ViewModel과 달리, MVVM ViewModel은 View가 상태 변경에 반응할 수 있도록 하는 바인딩 메커니즘(Data Binding)을 강조하여 더 선언적이고 모듈화된 설계를 용이하게 합니다.

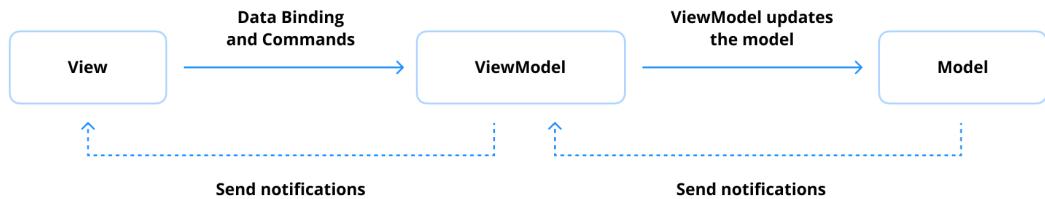


그림 145. mvvm-pattern

¹⁰³<https://developer.android.com/topic/architecture/ui-layer/stateholders>

Jetpack ViewModel과 MVVM ViewModel의 차이점

기본적으로 **Jetpack ViewModel**과 **MVVM ViewModel**은 다른 목적을 가지고 설계되었으며, 별개의 패러다임 하에서 작동합니다. Jetpack ViewModel은 주로 구성 변경 시 UI 관련 상태를 유지하는 데 의의를 두어, 생명주기를 인식하는 UI 상태 관리에 초점을 둡니다. 반면에 MVVM 아키텍처에서 정의된 MVVM ViewModel은 View와 Model 간의 중개자 역할을 하며 바인딩을 처리하고 UI가 ViewModel에서 제공하는 데이터를 표시하는 데만 관련되도록 보장합니다.

Jetpack ViewModel의 이름은 MVVM ViewModel과의 연관성을 시사할 수 있지만, 이것을 단지 사용하는 것만으로는 MVVM 아키텍처의 요구 사항을 완전히 충족하지 못합니다. MVVM의 원래 의도를 충족하려면 개발자는 UI가 ViewModel에서 제공하는 데이터에 수동적으로 반응하도록 보장하는 추가 바인딩 메커니즘을 구현해야 합니다. 여기에는 UI 레이어를 기본 비즈니스 로직에서 분리하여 관심사 분리를 명확히 하고 테스트 용이성을 향상시키는 견고한 데이터 바인딩 시스템 구축이 포함됩니다.

요약

Jetpack ViewModel과 MVVM ViewModel은 범위와 구현에서 다릅니다. Jetpack ViewModel은 안드로이드 애플리케이션에서 생명주기를 인식하는 UI 상태 관리에 중점을 두는 반면, MVVM ViewModel은 View와 Model 간의 데이터 및 명령 바인딩을 강조하여 더 선언적이고 수동적인 UI를 가능하게 합니다. Compose의 경우 ViewModel에서 직접 데이터를 관찰하면 구현이 단순화되지만, XML 기반 UI는 MVVM의 목표를 달성하기 위해 DataBinding이 필요합니다. 따라서, 개발자는 작업 중인 아키텍처 및 UI 프레임워크에 따라 접근 방식을 신중하게 선택할 필요가 있습니다.

Q) 55. Jetpack Navigation 라이브러리란 무엇인가요?

Jetpack Navigation 라이브러리¹⁰⁴는 앱 내 내비게이션을 단순화하고 표준화하기 위해 안드로이드에서 제공하는 프레임워크입니다. 개발자가 선언적으로 다양한 앱 화면 간의 내비게이션 경로와 전환을 정의할 수 있도록 하여 보일러 플레이트 코드를 줄이고 전반적인 사용자 경험을 향상시킵니다.

해당 라이브러리는 Activity, Fragment, Composable에 대한 내비게이션을 관리하는 API를 제공하며, 딥 링크, 백 스택 관리 및 애니메이션과 같은 추가적인 기능도 지원합니다.

¹⁰⁴<https://developer.android.com/guide/navigation>

Jetpack Navigation 라이브러리는 내비게이션을 원활하게 처리하기 위해 함께 작동하는 몇 가지 필수 구성 요소로 구성됩니다.

내비게이션 그래프 (Navigation Graph)

내비게이션 그래프는 앱 대상(화면) 간의 내비게이션 흐름과 관계를 정의하는 XML 리소스입니다. 각 대상은 Fragment, Activity 또는 커스텀 뷰와 같은 화면을 나타나대고 동작의 목적지 등을 정의합니다.

그림 146. nav_graph.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <navigation xmlns:android="http://schemas.android.com/apk/res/android"
3     xmlns:app="http://schemas.android.com/apk/res-auto" <!-- app 네임스페이스 추가 -->
4     xmlns:tools="http://schemas.android.com/tools" <!-- tools 네임스페이스 추가 (선택
   ↳ 사항) -->
5     android:id="@+id/nav_graph"
6     app:startDestination="@+id/homeFragment">
7
8     <fragment
9         android:id="@+id/homeFragment"
10        android:name="com.example.app.HomeFragment"
11        android:label="Home"
12        tools:layout="@layout/fragment_home"> <!-- 레이아웃 미리보기 (선택 사항) -->
13        <action
14            android:id="@+id/action_home_to_details"
15            app:destination="@+id/detailsFragment" />
16    </fragment>
17
18    <fragment
19        android:id="@+id/detailsFragment"
20        android:name="com.example.app.DetailsFragment"
21        android:label="Details"
22        tools:layout="@layout/fragment_details">
23        <!-- 인수 정의 (예시) -->
```

```
24     <argument>
25         android:name="itemId"
26         app:argType="integer" />
27     </fragment>
28 </navigation>
```

NavController

NavController는 내비게이션 그래프의 컨테이너 역할을 하여 대상을 호스팅하고 대상 간의 내비게이션을 관리합니다. 사용자가 탐색할 때 컨테이너 내에서 Fragment를 동적으로 교체합니다.

그림 147. activity_main.xml

```
1 <androidx.fragment.app.FragmentContainerView
2     android:id="@+id/nav_host_fragment"
3     android:name="androidx.navigation.fragment.NavHostFragment"
4     android:layout_width="match_parent"
5     android:layout_height="match_parent"
6     app:defaultNavHost="true" <!-- 뒤로 가기 버튼 처리 -->
7     app:navGraph="@navigation/nav_graph" /> <!-- 내비게이션 그래프 참조 -->
```

NavController

NavController는 내비게이션 작업을 처리하고 백 스택을 관리하는 역할을 합니다. 이를 사용하여 직접 코드로 목적지 간에 이동하거나 전반적인 내비게이션 흐름을 컨트롤할 수 있습니다. 이름에서 알 수 있듯이, 말 그대로 내비게이션의 컨트롤러입니다.

그림 148. MainActivity.kt

```
1 class MainActivity : AppCompatActivity() {  
2  
3     override fun onCreate(savedInstanceState: Bundle?) {  
4         super.onCreate(savedInstanceState)  
5         setContentView(R.layout.activity_main)  
6  
7         // NavHostFragment에서 NavController 찾기  
8         val navHostFragment = supportFragmentManager  
9             .findFragmentById(R.id.nav_host_fragment) as NavHostFragment  
10        val navController = navHostFragment.navController  
11  
12        // 버튼 클릭 시 내비게이션 수행  
13        findViewById<Button>(R.id.navigateButton).setOnClickListener {  
14            // 액션 ID를 사용하여 이동  
15            navController.navigate(R.id.action_home_to_details)  
16        }  
17    }  
18 }
```

Safe Args

Safe Args는 타입-세이프(type-safe) 내비게이션 및 인수 전달 코드를 생성하는 Gradle 플러그인입니다. 대상 간 데이터를 전달할 때 수동으로 번들을 만들 필요가 없습니다.

그림 149. Passing Data with Safe Args

```
1 // HomeFragmentDirections 클래스는 Safe Args 플러그인이 생성  
2 val action = HomeFragmentDirections.actionHomeToDetails(itemId = 42)  
3 findNavController().navigate(action)
```

Deep Linking

이 라이브러리는 딥 링크를 지원하여 사용자가 URL이나 알림과 같은 외부 소스에서 특정 화면으로 직접 이동할 수 있도록 합니다.

그림 150. Deep Link in Navigation Graph

```
1 <fragment
2     android:id="@+id/detailsFragment"
3     android:name="com.example.app.DetailsFragment">
4     <deepLink
5         app:uri="https://example.com/details/{itemId}" /> <!-- itemId 인수를 받는 딥
6             ↳ 링크 -->
7     </fragment>
```

Jetpack Navigation 라이브러리의 이점

1. **중앙 집중식 내비게이션**: 명확하고 유지 관리 가능한 구조를 위해 모든 내비게이션 흐름을 하나의 XML 파일에서 관리합니다.
2. **타입-세이프 인수**: 생성된 Safe Args 클래스를 사용하여 대상 간에 데이터를 안전하게 전달합니다.
3. **백 스택 관리**: 일관된 내비게이션을 위해 백 스택 동작을 자동으로 처리합니다.
4. **딥 링크 지원**: 외부 내비게이션 요청을 원활하게 처리하여 사용자 경험을 향상시킵니다.
5. **Jetpack 컴포넌트와의 통합**: Fragment, ViewModel, LiveData와 잘 작동하여 생명주기를 고려한 내비게이션을 보장합니다.

요약

Jetpack Navigation 라이브러리는 내비게이션 경로, 전환 및 인수를 관리하기 위한 선언적이고 중앙 집중적인 접근 방식을 제공하여 안드로이드 애플리케이션 내 내비게이션을 단순화합니다. 다른 Jetpack 컴포넌트와 원활하게 통합되고, 딥 링킹을 지원하며, 타입-세이프 인수 전달을 위한 Safe Args와 같은 도구를 제공합니다. 이 라이브러리는 보일러 플레이트 코드를 줄이고 일관된 내비게이션 패턴을 보장하여 개발자 경험을 향상시킵니다.

실전 질문

Q) Jetpack Navigation 라이브러리는 백 스택을 어떻게 처리하고, NavController로 어떻게 백스택을 조작할 수 있나요?

Q) Safe Arguments란 무엇이며, Jetpack Navigation Component에서 목적 내비게이션 간 데이터를 전달할 때 타입 안전성을 어떻게 보장하나요?

Q) 56: Dagger 2와 Hilt의 동작원리 및 차이점에 대해서 설명해 주세요.

안드로이드 개발을 위한 의존성 주입 라이브러리에는 다양한 선택지가 있으며, 그중 Dagger 2와 Hilt가 가장 대표적인 옵션으로 꼽힙니다. 두 라이브러리 모두 Google에서 개발하고 공식적으로 지원할 뿐만 아니라, 대규모 프로젝트에서 사용성이 많이 검증되었기 때문에 어느 정도 신뢰할 수 있는 선택입니다.

Dagger 2란?

Dagger 2¹⁰⁵는 안드로이드 및 JVM 환경을 위한 정적 컴파일 타임 기반의 의존성 주입(DI) 라이브러리입니다. 객체 생성을 관리하고 의존성을 자동으로 제공하여 모듈성을 개선하고 애플리케이션 테스트를 용이하게 하도록 설계되었습니다. Dagger 2는 컴파일 타임에 코드를 생성하여, 리플렉션에 기반한 DI 프레임워크에 비해 더 나은 성능을 보장합니다.

Dagger 2는 @Module, @Provides, @Inject와 같은 어노테이션을 사용하여 의존성을 선언하고 요청합니다. 개발자는 컴포넌트와 모듈을 통해 의존성 그래프를 생성하며, Dagger 2는 런타임에 이를 자동으로 해결합니다.

¹⁰⁵<https://dagger.dev/>

그림 151. Dagger 2 Example

```
1  @Module
2  class NetworkModule {
3      @Provides
4      fun provideRetrofit(): Retrofit {
5          return Retrofit.Builder()
6              .baseUrl("https://example.com")
7              .build()
8      }
9  }
10
11 @Component(modules = [NetworkModule::class])
12 interface AppComponent {
13     // MainActivity에 의존성 주입
14     fun inject(activity: MainActivity)
15 }
16
17 class MainActivity : AppCompatActivity() {
18     // Retrofit 의존성 주입 요청
19     @Inject
20     lateinit var retrofit: Retrofit
21
22     override fun onCreate(savedInstanceState: Bundle?) {
23         super.onCreate(savedInstanceState)
24         // Dagger 컴포넌트 생성 및 주입 실행
25         DaggerAppComponent.create().inject(this)
26         // 이제 retrofit 인스턴스 사용 가능
27     }
28 }
```

Hilt란 무엇인가?

Hilt¹⁰⁶는 Dagger 2 위에 구축된 안드로이드 전용 의존성 주입 라이브러리입니다. Activity, Fragment, ViewModel과 같이 안드로이드 생명주기와 밀접한 관련이 있는 클래스에 스코프가 지정된 사전 정의된 컴포넌트를 제공하여, Dagger를 안드로이드 프로젝트에 통합하는 프로세스를 전체적으로 단순화합니다.

Hilt는 @HiltAndroidApp 및 @AndroidEntryPoint와 같은 어노테이션을 제공하여 DI 설정을 간소화함으로써 Dagger 2에 필요한 많은 보일러 플레이트 코드를 제거합니다. 또한 @Singleton 및 @ActivityScoped와 같은 범위를 정의하여 의존성 생명주기를 관리합니다. 그렇기에 Hilt로 구현이 가능한 것은 Dagger 2로도 모두 구현이 가능합니다.

그림 152. Hilt Example

```
1  @HiltAndroidApp // Hilt 사용을 위한 Application 클래스 어노테이션
2  class MyApplication : Application()
3
4  @AndroidEntryPoint // Hilt가 의존성을 주입할 Activity
5  class MainActivity : AppCompatActivity() {
6      @Inject // Retrofit 의존성 주입 요청
7      lateinit var retrofit: Retrofit
8
9      override fun onCreate(savedInstanceState: Bundle?) {
10          super.onCreate(savedInstanceState)
11          setContentView(R.layout.activity_main)
12          // Hilt가 자동으로 의존성 주입 처리
13      }
14  }
15
16  @Module
17  @InstallIn(SingletonComponent::class) // 모듈이 설치될 컴포넌트 지정 (앱 전체 범위)
18  object NetworkModule {
19      @Provides
20      fun provideRetrofit(): Retrofit {
21          return Retrofit.Builder()
```

¹⁰⁶<https://dagger.dev/hilt/>

```
22     .baseUrl("https://example.com")
23     .build()
24 }
25 }
```

Dagger 2와 Hilt의 주요 차이점

- 통합 프로세스:** Dagger 2는 개발자가 컴포넌트와 인젝터를 수동으로 정의해야 하므로 장황한 보일러 플레이트 코드를 작성해야 할 수 있습니다. Hilt는 사전 정의된 컴포넌트와 생명주기에 스코핑 된 어노테이션을 제공하여 이를 단순화합니다.
- 안드로이드 생명주기 통합:** Hilt는 안드로이드에 특화되어 있으며 Activity, Fragment, ViewModel과 같은 안드로이드 컴포넌트에 대한 내장 지원을 제공합니다. Dagger 2는 더 범용적이며 생명주기 인식 컴포넌트에 대해 수동적인 설정이 필요합니다.
- 스코핑(Scoping):** Hilt는 @Singleton, @ActivityScoped, @FragmentScoped와 같이 안드로이드 생명주기 클래스와 밀접하게 통합된 사전 정의된 범위를 제공합니다. Dagger 2에서는 스코핑에 수동 설정 및 커스텀 어노테이션이 필요합니다.
- 코드 단순성:** Hilt는 많은 보일러 플레이트 코드를 추상화하여 DI 설정의 복잡성을 줄여 초보급가 DI 라이브러리 사용을 더 쉽게 접근할 수 있도록 합니다. Dagger 2는 유연하고 복잡한 구조에서 더 세밀하게 사용할 수 있지만, 개발자가 모든 컴포넌트와 관계를 수동으로 정의해야 합니다.
- 사용 사례:** Dagger 2는 복잡하고 커스텀된 의존성 주입 그래프가 필요한 프로젝트에 적합합니다. Hilt는 안드로이드 프로젝트를 위해 맞춤 설계되었으며 사용 편의성과 안드로이드 컴포넌트와의 통합에 중점을 둡니다.

Hilt 및 Dagger 2에서 제공하는 어노테이션

Hilt가 Dagger 위에 구축되었기 때문에 Hilt와 Dagger 2는 동작 형태가 유사하고, 여러 동일한 어노테이션을 사용합니다. 아래는 Hilt 및 Dagger에서 동일하게 사용되거나, Hilt에 특화된 어노테이션에 대한 설명입니다.

Dagger 2 기반의 어노테이션 (Dagger에서 제공하고 Hilt에서도 사용)

- @Inject¹⁰⁷:** 의존성 주입을 위해 생성자, 필드 또는 메서드에 표시합니다. 의존성 주입을 실질적으로 요청하는 데 사용됩니다.

¹⁰⁷<https://docs.oracle.com/javaee/6/api/javax/inject/Inject.html>

2. **@Provides**¹⁰⁸: @Module 내에서 의존성 생성 메서드를 정의합니다. Hilt와 Dagger 모두 이 어노테이션을 사용하여 객체를 제공합니다.
3. **@Module**¹⁰⁹: 클래스를 의존성 제공자 컨테이너로 선언합니다. 모듈은 관련된 의존성 생성 로직을 그룹화합니다.
4. **@Binds**¹¹⁰: @Module 내에서 인터페이스를 구현에 맵핑하는 데 사용되어 의존성 정의 시 보일러 플레이트 코드를 줄입니다.
5. **@Qualifier**¹¹¹: 커스텀 어노테이션을 사용하여 동일한 타입에 대해 여러 의존성 바인딩을 구별합니다.
6. **@Scope**¹¹²: 특정 의존성의 생명주기를 제어하기 위해 커스텀 스코핑 어노테이션을 정의할 수 있습니다.
7. **@Singleton**¹¹³: 의존성이 해당 범위(일반적으로 앱 생명주기) 내내 단일 공유 인스턴스를 가져야 함을 지정합니다.
8. **@Component**¹¹⁴: 의존성 그래프의 인터페이스를 정의합니다. @Component는 모듈을 주입 대상에 연결하고 의존성 생명주기를 제어합니다.
9. **@Subcomponent**¹¹⁵: 지정한 범위 내에서 의존성을 관리하기 위한 케이스를 위해 @Component 내에 더 작은 의존성 그래프를 생성합니다. 종종 자체 생명주기를 가진 자식 컴포넌트를 만드는 데 사용됩니다.

흥미로운 점은 @Inject, @Qualifier, @Scope, @Singleton 어노테이션의 경우 Dagger 라이브러리에서 자체적으로 제공하는 것이 아닌, Java 스펙(javax.inject 패키지)에서 제공되는 것을 활용하고 있다는 것입니다.

Hilt에 특화된 어노테이션

1. **@HiltAndroidApp**¹¹⁶: Hilt를 부트스트랩하고 전체 앱에 대한 의존성 그래프를 생성하기 위해 사용합니다. Application 클래스에 사용할 수 있고, Hilt를 초기화 하기 위한 필수 어노테이션입니다.

¹⁰⁸<https://dagger.dev/api/latest/dagger/Provides.html>

¹⁰⁹<https://dagger.dev/api/latest/dagger/Module.html>

¹¹⁰<https://dagger.dev/api/latest/dagger/Binds.html>

¹¹¹<https://docs.oracle.com/javaee/7/api/javax/inject/Qualifier.html>

¹¹²<https://docs.oracle.com/javaee/6/api/javax/inject/Scope.html>

¹¹³<https://docs.oracle.com/javaee/7/api/javax/inject/Singleton.html>

¹¹⁴<https://dagger.dev/api/latest/dagger/Component.html>

¹¹⁵<https://dagger.dev/api/latest/dagger/Subcomponent.html>

¹¹⁶<https://dagger.dev/hilt/application>

2. **@AndroidEntryPoint¹¹⁷**: 안드로이드 컴포넌트(가령, Activity, Fragment, Service)를 주입 대상으로 마크합니다. 해당 어노테이션을 사용하는 것만으로도 커스텀 Dagger 컴포넌트를 정의할 필요가 없어집니다.
3. **@InstallIn¹¹⁸**: @Module이 설치되어야 하는 컴포넌트(가령, SingletonComponent, ActivityComponent)를 지정합니다.
4. **@EntryPoint¹¹⁹**: Hilt에서 관리하는 안드로이드 컴포넌트가 아닌 외부에서 의존성에 접근하기 위한 진입점을 정의하는 데 사용됩니다.
5. **@HiltViewModel¹²⁰**: Jetpack ViewModel을 Hilt와 통합하기 위한 특수 어노테이션입니다. ViewModel이 생명주기를 인식하면서 Hilt의 의존성 주입을 사용할 수 있도록 보장합니다. @HiltViewModel 어노테이션은 생성자에 @Inject와 함께 사용해야 합니다.
6. **Scope Annotations¹²¹ (@ActivityRetainedScoped, @ViewModelScoped, @ActivityScoped, @FragmentScoped, @ViewScoped, @ServiceScoped)**: 사용자가 컴포넌트를 수동으로 정의하고 인스턴스화하는 순수 Dagger와 달리, Hilt는 사전 정의된 컴포넌트를 제공하여 특정 라이프 사이클에 의존성을 바인딩하는 프로세스를 단순화합니다. Hilt에는 Hilt에 특화된 컴포넌트 뿐만 아니라, 스코프 지정 어노테이션들이 포함되어 있어 의존성 주입을 더 간소화하고 안드로이드 생명주기에 따라 의존성을 관리하기 쉽도록 합니다.

Hilt는 @HiltAndroidApp, @AndroidEntryPoint, @HiltViewModel과 같은 어노테이션을 도입하여 안드로이드 애플리케이션의 의존성 주입을 단순화합니다. 이러한 추상화는 Dagger 컴포넌트를 수동으로 정의하는 것과 관련하여 많은 보일러 플레이트 코드를 제거합니다.

포괄적인 개요는 아래 일러스트가 제공하는 [Hilt 및 Dagger 어노테이션 치트 시트¹²²](#)를 통해 학습하실 수 있습니다.

¹¹⁷<https://dagger.dev/hilt/android-entry-point>

¹¹⁸<https://dagger.dev/hilt/modules>

¹¹⁹<https://dagger.dev/hilt/entry-points>

¹²⁰<https://dagger.dev/hilt/view-model>

¹²¹<https://dagger.dev/hilt/components>

¹²²<https://developer.android.com/training/dependency-injection/hilt-cheatsheet>

Hilt & Dagger Annotations

v2.33 - @AndroidDev

Annotation	Usage	Code Sample
<code>@HiltAndroidApp</code>	Kicks off Hilt code generation. Must annotate the Application class.	<pre><code>@HiltAndroidApp class MyApplication : Application() { ... }</code></pre>
<code>@AndroidEntryPoint</code>	Adds a DI container to the Android class annotated with it. This requires using Hilt's Gradle Plugin.	<pre><code>@AndroidEntryPoint class MyActivity : AppCompatActivity() { ... }</code></pre>
<code>@Inject</code>	Constructor Injection. Tells which constructor to use to provide instances and which dependencies the type has.	<pre><code>class AnalyticsAdapter @Inject constructor(private val service: AnalyticsService) { ... }</code></pre>
	Field injection. Populates fields in <code>@AndroidEntryPoint</code> annotated classes. Fields cannot be private.	<pre><code>@AndroidEntryPoint class MyActivity : AppCompatActivity() { @Inject lateinit var adapter: AnalyticsAdapter ... }</code></pre>
<code>@HiltViewModel</code>	Tells Hilt how to provide instances of an Architecture Component ViewModel.	<pre><code>@HiltViewModel class MyViewModel @Inject constructor(private val adapter: AnalyticsAdapter, private val state: SavedStateHandle) : ViewModel() { ... }</code></pre>
<code>@Module</code>	Class in which you can add bindings for types that cannot be constructor injected.	<pre><code>@InstallIn(SingletonComponent::class) @Module class AnalyticsModule { ... }</code></pre>
<code>@InstallIn</code>	Indicates in which Hilt-generated DI containers (<code>SingletonComponent</code> in the code) module bindings must be available.	<pre><code>@InstallIn(SingletonComponent::class) @Module class AnalyticsModule { ... }</code></pre>
<code>@Provides</code>	Adds a binding for a type that cannot be constructor injected: <ul style="list-style-type: none"> - Return type is the binding type. - Parameters are dependencies. - Every time an instance is needed, the function body is executed if the type is not scoped. 	<pre><code>@InstallIn(SingletonComponent::class) @Module class AnalyticsModule { @Provides fun providesAnalyticsService(converterFactory: GsonConverterFactory): AnalyticsService { return Retrofit.Builder() .baseUrl("https://example.com") .addConverterFactory(converterFactory) .build() .create(AnalyticsService::class.java) } }</code></pre>

그림 153. hilt-cheatsheet

@Binds	<p>Shorthand for binding an interface type:</p> <ul style="list-style-type: none"> - Methods must be in a module. - @Binds annotated methods must be abstract. - Return type is the binding type. - Parameter is the implementation type. 	<pre>@InstallIn(SingletonComponent::class) @Module abstract class AnalyticsModule { @Binds abstract fun bindsAnalyticsService(analyticsServiceImpl: AnalyticsServiceImpl): AnalyticsService }</pre>
Scope Annotations: @Singleton @ActivityScoped ...	<p>Scoping object to a container.</p> <p>The same instance of a type will be provided by a container when using that type as a dependency, for field injection, or when needed by containers below in the hierarchy.</p>	<pre>@Singleton class AnalyticsAdapter @Inject constructor(private val service: AnalyticsService) { ... }</pre>
Qualifiers for predefined Bindings: @ApplicationContext @ActivityContext	<p>Predefined bindings you can use as dependencies in the corresponding container.</p> <p>These are qualifier annotations.</p>	<pre>@Singleton class AnalyticsAdapter @Inject constructor(@ApplicationContext val context: Context private val service: AnalyticsService) { ... }</pre>
@EntryPoint	<p>Obtain dependencies in classes that are either not supported directly by Hilt or cannot use Hilt.</p> <p>The interface annotated with @EntryPoint must also be annotated with @InstallIn passing in the Hilt predefined component from which that dependency is taken from.</p> <p>Access the bindings using the appropriate static method from EntryPointAccessors passing in an instance of the class with the DI container (which matches the value in the @InstallIn annotation) and the entry</p>	<pre>class MyContentProvider(): ContentProvider { @InstallIn(SingletonComponent::class) @EntryPoint interface MyContentProviderEntryPoint { fun analyticsService(): AnalyticsService } override fun query(...): Cursor { val appContext = context?.applicationContext ?: throw IllegalStateException() val entryPoint = EntryPointAccessors.fromApplication(appContext, MyContentProviderEntryPoint::class.java) val analyticsService = entryPoint.analyticsService() ... } }</pre>

For more information about DI, Hilt and Dagger: <https://d.android.com/dependency-injection>

그림 154. hilt-cheatsheet

요약

Dagger 2와 Hilt는 모두 객체 생성 및 관리를 간소화하는 의존성 주입 라이브러리입니다. Dagger 2는 더 유연하고 순수 Java 또는 안드로이드 프로젝트에서 사용할 수 있지만 더 많은 수동적이고 복잡한 설정이 요구됩니다. 반면에 Hilt는 Dagger 2 위에 구축되었지만

생명주기를 인식하는 컴포넌트와 쉽게 통합하고 보일러 플레이트 코드를 줄여 안드로이드에 특화된 DI 솔루션을 제공합니다. 대부분의 경우 Hilt를 선택하는 것이 개발적인 측면에서 훨씬 편하고, Dagger 2는 DI 아키텍처를 고도로 커스텀하여 설계해야 하는 경우에 더 적합합니다.

실전 질문

- Q) Dagger 2와 비교하여 의존성 Hilt는 주입을 어떻게 단순화하고, 안드로이드 애플리케이션에서 Hilt를 사용하는 것의 장점은 무엇인가요?
- Q) Dagger와 Hilt에서 @Provides와 @Binds의 차이점은 무엇이고, 각각 언제 사용해야 하나요?
- Q) Hilt에서 @Singleton, @ActivityScoped, @ViewModelScoped를 사용하면 내부적으로 스코핑이 어떻게 작동하는지 매커니즘을 설명하고, 해당 스코프가 사용되면 애플리케이션 내 의존성의 수명을 어떻게 관리하는지 설명해 주세요.

💡 Pro Tips for Mastery: 수동으로 의존성 주입을 구현해 본 적이 있나요?

DI 프레임워크에 의존하지 않고 수동으로 의존성 주입을 구현할 수 있으며, 이는 구현체에 대한 완전하고 세밀한 제어가 요구되는 경우 유용할 수 있습니다. 그러나 이러한 접근 방식은 이미 존재하는 DI 프레임워크 및 라이브러리 사용하는 것과 비교하여 장단점이 있습니다.

수동 의존성 주입(런타임 기반이라고 가정)은 객체의 전체 생명주기(스코핑, 그룹화 및 메모리 누수 방지를 위한 적절한 리소스 정리 포함)를 개발자가 수동적으로 관리해야 하므로 상당한 비용이 요구됩니다. 종종 광범위한 보일러 플레이트 코드를 작성해야 하고 **서비스 로케이터 패턴(service locator pattern)**¹²³으로 전락하거나, 전역적인 싱글턴 패턴에 의존하게 될 위험이 있습니다. 반면 Dagger 2 및 Hilt와 같은 라이브러리는 어노테이션에 기반하여 컴파일 타임에 주입 관련 코드를 생성하기 때문에 유지 관리를 단순화하고, 컴파일 타임에 유효성 검사를 보장하기 때문에 의존성 주입 순환참조와 같은 현상이 발생하는 것을 사전에 방지합니다.

한편으로, Dagger 2 및 Hilt와 같은 DI 라이브러리(컴파일 타임 기반)는 컴파일 중 어노테이션 프로세싱¹²⁴ 및 코드 생성으로 인해 빌드 시간을 증가시킵니다. 런타임 초기화에

¹²³https://en.wikipedia.org/wiki/Service_locator_pattern

¹²⁴어노테이션 프로세싱(Annotation processing)은 Java 및 Kotlin의 컴파일 타임 메커니즘으로, APT(Annotation Processing Tool) 또는 KAPT(Kotlin Annotation Processing Tool)와 같은 도구를 통해 어노테이션을 사용하여 추가 코드를 생성하거나 기존 코드를 검증합니다.

의존하는 수동 DI는 빌드 시간을 줄일 수 있지만, 컴파일 타임에 최적화된 프레임워크 및 라이브러리에 비해 런타임 성능이 떨어질 수 있습니다. 궁극적으로 수동 의존성 주입은 일반적으로 중소 규모 프로젝트에 더 적합하며, 더 크고 복잡한 애플리케이션은 DI 프레임워크 및 라이브러리에서 제공하는 확장성과 효율성의 이점을 누리는 것이 더 현명합니다. 하지만, 언제나 그렇듯 정답은 없습니다. 어떤 솔루션 해야 할지는 프로젝트의 크기, 복잡성 및 특정 요구 사항에 따라 달라집니다.

Manifest Kotlin Interview for Android Developers¹²⁵ 책에서 이 주제에 대해 더 깊이 다룰 예정입니다. 이에 대해 더 자세한 내용을 학습하고 싶으시다면, 공식 안드로이드 문서의 **수동 의존성 주입**¹²⁶ 가이드를 참고하시길 바랍니다. 또한 공식 Compose 예제의 일부인 **Jetcaster 프로젝트**¹²⁷는 수동 의존성 주입을 구현한 사례를 보여줍니다.

💡 Pro Tips for Mastery: Dagger 2 및 Hilt 이외에 알고 있는 DI 라이브러리가 있나요?

Dagger 2와 Hilt는 구글에서 권장되기 때문에 안드로이드에서 가장 보편적으로 사용되는 의존성 주입(DI) 라이브러리이지만, **Koin** 및 **Anvil**과 같이 DI 구현을 다른 접근 방식으로 제공하는 라이브러리가 있습니다.

Koin: 가볍고 사용하기 쉬운 DI 라이브러리

Koin¹²⁸은 단순성을 염두에 두고 설계된 경량 의존성 주입 라이브러리입니다. 어노테이션, 컴파일 타임의 코드 생성 또는 무거운 보일러 플레이트 코드의 필요성을 없애고, Kotlin DSL을 사용하여 의존성 모듈을 정의하는 데 중점을 둡니다.

Koin의 주요 특징은 아래와 같습니다.

- **어노테이션 처리 없음:** 의존성은 Kotlin 코드로 정의되어 어노테이션 처리를 피하고 빌드 시간을 단축합니다.
- **Kotlin 우선 접근 방식:** Kotlin DSL을 사용하여 DI 구성을 매우 읽기 쉽고 직관적으로 만듭니다.
- **사용 편의성:** 설정이 빠르고 소규모 프로젝트 또는 빠른 DI 환경 세팅을 추구하는 개발자에게 이상적입니다.

¹²⁵<https://www.kotlin.skydoves.me/>

¹²⁶<https://developer.android.com/training/dependency-injection/manual>

¹²⁷<https://github.com/android/compose-samples/blob/84dc6417dfbd6408f498d71afcc1272087c6b3d7/Jetcaster/app/src/main/java/com/example/jetcaster/Graph.kt>

¹²⁸<https://github.com/InsertKoinIO/koin>

- **동적 해결**: 런타임 시 의존성이 결정되는 시나리오에서 유용한 동적 의존성 솔루션을 지원합니다.

예제는 아래와 같습니다.

그림 155. Koin.kt

```
1 // 모듈 정의
2 val appModule = module {
3     single { Repository() } // 싱글톤 정의
4     factory { ViewModel(get()) } // 팩토리(매번 새 인스턴스) 정의, get()으로 의존성 주입
5 }
6
7 // Koin 시작 (Application 클래스 등에서)
8 startKoin {
9     androidContext(this@MyApplication) // 안드로이드 컨텍스트 제공
10    modules(appModule) // 모듈 등록
11 }
12
13 // 의존성 주입 (Activity, Fragment 등에서)
14 class MyActivity : AppCompatActivity() {
15     // by inject() 델리게이트 사용
16     val viewModel: ViewModel by inject()
17     // 또는 get() 직접 사용
18     // val repository: Repository = get()
19 }
```

Koin은 컴파일 타임 의존성 관련 코드를 전혀 생성하지 않고 모두 런타임에 처리하기 때문에, 소규모 프로젝트나 빌드 성능이 우선시 되는 시나리오에 적합합니다. 또한 Dagger 2와 Hilt 가 현재 KMP를 지원하지 않는 점을 감안할 때 Kotlin Multiplatform(KMP) 프로젝트에서는 가장 적합한 선택입니다. 최근 [Koin 4.0 Stable 버전¹²⁹](#)의 출시로 Koin은 더 복잡한 사용 사례에서 신뢰할 수 있는 DI 라이브러리로서의 입지를 다지고 있습니다.

¹²⁹<https://blog.insert-koin.io/koin-4-0-official-release-f4827bbcfce3>

 **Fun Facts:** Koin은 Arnaud라는 개발자가 1인 오픈 소스 프로젝트로 운영하다가, 2022년 프랑스에 [Kotzilla](https://kotzillaproject.org/)¹³⁰라는 기업을 세우고 현재는 무료 오픈 소스 라이브러리이자 Kotzilla의 메인 제품으로서 성장하고 있습니다.

Dagger 2 vs. Koin?

[Reddit](#)¹³¹이라는 해외 익명 커뮤니티를 보면 의존성 주입 라이브러리인 Dagger 2와 Koin을 비교하는 토론을 확인해 보실 수 있습니다. Dagger 초기 버전 개발자인 Jake Wharton은 다음과 같은 의견을 남겼습니다.

Jake Wharton: Koin은 의존성 주입기 솔루션이 아니라 수동으로 의존성 주입을 수행하기 위해 reified 트릭을 동반한 서비스 로케이터 패턴이므로, 사용하면 사용할 수록 보일러 플레이트 코드가 불균형하게 확장될 것입니다. Dagger(및 Guice 등)에는 일정량의 고정 오버헤드가 있지만, 바인딩이 주입된 타입 전체에 자동으로 전파되기 때문에 거의 그래프 형태를 변경할 필요가 없습니다. 수동 의존성 주입을 사용하면 주입된 타입 전체에 수동으로 바인딩을 전파해야 합니다.

Koin은 작은 토이 프로젝트에 사용하는 거라면 큰 상관이 없습니다. 사실 라이브러리를 아예 쓰지 않아도 될 정도죠. 하지만 수백 개의 바인딩과 수많은 주입 타입, 깊은 의존성 그래프를 가진 진지한 규모의 앱을 개발하려는 거라면, Koin으로 수작업으로 작성할 코드를 대신 자동 생성해주는 정식 DI(Dependency Injection) 라이브러리를 사용하는 것이 훨씬 낫습니다.

Koin은 서비스 로케이터 패턴인가요?

[Reddit](#)¹³²에서 의존성 주입을 위해 Dagger 2와 Koin의 차이점을 논하는 토론이 있었습니다. Dagger 초기 버전 개발에 중추적인 역할을 한 Jake Wharton이 해당 주제에 대해 본인의

¹³⁰<https://kotzillaproject.org/>

¹³¹https://www.reddit.com/r/androiddev/comments/8ch4cg/dagger2_vs_koin_for_dependency_injection/

¹³²https://www.reddit.com/r/androiddev/comments/8ch4cg/dagger2_vs_koin_for_dependency_injection/

의견을 밝히면서 초기에 많은 의견 대립이 발생하곤 했습니다.

Koin 팀: Koin은 DI와 서비스 로케이터 패턴을 모두 지원하여 개발자에게 유연성을 제공합니다. 그러나 의존성이 생성자 매개변수로 전달되는 DI, 특히 생성자 주입 사용을 강력히 권장합니다. 해당 접근 방식은 더 나은 테스트 용이성을 촉진하고 코드를 더 쉽게 이해할 수 있도록 합니다.

Koin의 디자인 철학은 필요할 때는 복잡한 구성을 허용하면서, 일반적인 상황에서는 단순성과 DI 환경 셋업의 용이성에 중점을 둡니다. Koin을 사용하면 개발자는 효과적으로 의존성을 관리할 수 있는데, DI는 개발 대부분의 시나리오에서 권장되고 선호되는 접근 방식입니다.

Anvil: Dagger 2에 기반한 컴파일러 플러그인

Anvil¹³³은 Block(전 Square)에서 개발한 Dagger 컴파일러 플러그인입니다. 특히 Dagger를 많이 사용하는 프로젝트의 경우 모듈 생성을 위한 보일러 플레이트 코드를 단순화하여 Dagger의 사용성을 향상시킵니다. 내부적으로는 Hilt와 매우 유사하게 작동합니다. Anvil은 팩토리 및 모듈 코드 생성을 간소화하여 초기 Dagger의 셋업 비용을 줄입니다.

Anvil의 주요 특징은 다음과 같습니다.

- **손쉬운 Dagger 셋업:** 어노테이션이 붙은 클래스에 대해 Dagger 컴포넌트 및 팩토리를 자동으로 생성합니다.
- **보일러 플레이트 코드 감소:** 수동 구성은 최소화하여 대규모 프로젝트에서 Dagger를 더 쉽게 사용할 수 있도록 합니다.
- **Dagger와의 통합:** 기존 DI 설정을 크게 변경할 필요 없이 Dagger와 원활하게 작동합니다.
- **빌드 성능:** 특정 컴포넌트에 대한 어노테이션 프로세싱 오버헤드를 줄여 빌드 시간을 개선하는 데 크게 도움이 됩니다.

아래는 Anvil 사용 예시입니다.

¹³³<https://github.com/square/anvil>

그림 156. Dagger2.kt

```
1 // Anvil 어노테이션 사용 예시
2 import com.squareup.anvil.annotations.ContributesBinding
3 import com.squareup.anvil.annotations.ContributesTo
4 import javax.inject.Inject
5
6 interface Repository
7 interface AppScope // 커스텀 스코프
8
9 // Repository 구현 클래스에 ContributesBinding 적용
10 @ContributesBinding(AppScope::class)
11 class MyRepository @Inject constructor() : Repository
12
13 // 모듈 인터페이스에 ContributesTo 적용
14 @ContributesTo(AppScope::class)
15 interface AppModule {
16     // Provides 메서드 정의 (Anvil이 구현 생성)
17     fun provideSomeDependency(): SomeDependency
18 }
```

Anvil은 내부적으로 Dagger를 사용하고 있으며, 보일러 플레이트 코드를 줄이고 개발자 생산성을 향상시키려는 프로젝트에 이상적입니다. Anvil은 문서에서 다음과 같이 소개하고 있습니다.

Hilt는 Google이 권장하는 Android용 의존성 주입(DI) 가이드입니다. @InstallIn을 통한 엔트리 포인트 및 모듈 정의 방식은 Anvil과 유사한 기능을 제공합니다. 만약 Hilt를 사용한다면, Anvil을 따로 사용할 필요는 없습니다.

다만 Hilt는 다양한 기능을 제공하는 동시에 몇 가지 제약도 함께 따릅니다. 우리의 경우 수천 개의 모듈과 여러 개의 Dagger 컴포넌트가 존재하는 코드베이스를 Hilt로 전환하는 것은 현실적으로 불가능했습니다. 우리가 필요했던 기능은 모듈과 컴포넌트 인터페이스를 자동 병합하는 기능 하나뿐이었고, 또한 Dagger의 annotation processor를 특정 모듈에만

제한적으로 사용하여 빌드 성능을 최적화하고 있었기 때문에, Hilt를 사용하면 이런 제약을 더 이상 적용할 수 없었습니다. Anvil은 Hilt가 공개되기 한참 이전부터 개발이 시작되었고, 내부적으로는 이미 프로덕션 환경에서 사용되고 있었습니다.

올바른 라이브러리 선택하기

- **Koin**은 소규모 프로젝트나 DI를 빠르게 셋업 하여 프로젝트에 적용하고 싶은 경우, 또는 어노테이션 처리 없이 간단하게 DI를 사용하고자 하는 경우 좋은 선택입니다.
- **Anvil**은 이미 대규모 프로젝트에서 Dagger를 사용하고 있으며, 기존 Dagger 컴포넌트 와의 호환성을 유지하면서 워크플로를 단순화하려는 상황에 가장 적합합니다.

Q) 57. Jetpack Paging 라이브러리는 어떤 메커니즘으로 동작하나요?

Jetpack Paging 라이브러리¹³⁴는 대규모 데이터 셋을 청크 또는 “페이지” 단위로 로드하고 표시하는 프로세스를 돋도록 설계된 안드로이드 아키텍처 컴포넌트입니다. 데이터베이스나 API와 같은 소스에서 데이터를 효율적으로 가져와야 하는 애플리케이션에 특히 유용하며, 메모리 사용량을 최소화하고 RecyclerView 기반 UI의 전반적인 성능을 향상시킵니다.

Paging 라이브러리는 데이터를 점진적으로 로드하기 위한 구조화된 접근 방식을 제공합니다. 데이터 캐싱, 재시도 메커니즘, 효율적인 메모리 사용과 같은 주요 측면을 기본적으로 처리합니다. 해당 라이브러리는 로컬 데이터 소스(가령, Room 데이터베이스)와 원격 소스(가령, 네트워크 API) 또는 이 둘의 조합을 모두 지원합니다.

Paging 라이브러리의 구성 요소

1. **PagingData**: 점진적으로 로드되는 데이터 스트림을 나타냅니다. RecyclerView와 같은 UI 컴포넌트에 의해 관찰되고 사용될 수 있습니다.
2. **PagingSource**: 데이터 소스에서 데이터가 로드되는 방식을 정의하는 역할을 합니다. 위치 또는 ID와 같은 키값을 기반으로 데이터 페이지를 로드하는 메서드를 제공합니다.

¹³⁴<https://developer.android.com/topic/libraries/architecture/paging/v3-overview>

3. **Pager**: PagingSource와 PagingData 간의 중개자 역할을 합니다. PagingData 스트림의 생명주기를 관리합니다.
4. **RemoteMediator**: 로컬 캐싱과 원격 API 데이터를 결합할 때 경계 조건을 구현하는 데 사용됩니다.

Paging 라이브러리 작동 방식

Paging 라이브러리는 데이터를 페이지로 분할하여 효율적인 데이터 로딩을 가능하게 합니다. 사용자가 RecyclerView를 스크롤하면 라이브러리는 필요에 따라 새 데이터 페이지를 가져와 최소한의 메모리 사용량을 보장합니다. 이 라이브러리는 Flow 또는 LiveData를 기본적으로 지원하여 데이터 변경 사항을 관찰하고 그에 따라 UI를 업데이트할 수 있도록 합니다.

다음은 일반적인 워크플로우입니다.

1. **PagingSource**를 정의하여 데이터 가져오는 방법을 지정합니다.
2. **Pager**를 사용하여 PagingData의 Flow를 생성합니다.
3. ViewModel에서 PagingData를 관찰하고 RecyclerView에서 렌더링하기 위해 PagingDataAdapter에 전달합니다.

Jetpack Paging 구현 예시

먼저, 네트워크에서 데이터를 가져오는 **PagingSource**를 다음과 같이 구현합니다.

그림 157. ExamplePagingSource.kt

```
1 class ExamplePagingSource(  
2     private val apiService: ApiService  
3 ) : PagingSource<Int, ExampleData>() { // Int: 페이지 키 타입, ExampleData: 로드할 데이터  
4     // 타입  
5     override suspend fun load(params: LoadParams<Int>): LoadResult<Int, ExampleData> {  
6         // 현재 페이지 키 가져오기 (null이면 첫 페이지)  
7         val page = params.key ?: 1  
8         return try {
```

```
9         // API 호출하여 데이터 가져오기
10        val response = apiService.getData(page, params.loadSize)
11        // 로드 결과 반환 (성공 시 Page, 실패 시 Error)
12        LoadResult.Page(
13            data = response.items,
14            prevKey = if (page == 1) null else page - 1, // 이전 페이지 키
15            nextKey = if (response.items.isEmpty()) null else page + 1 // 다음 페이지
16            ← 키
17        )
18    } catch (e: IOException) { // 네트워크 오류 처리
19        LoadResult.Error(e)
20    } catch (e:HttpException) { // HTTP 오류 처리
21        LoadResult.Error(e)
22    }
23
24    // 페이지 키를 정의하는 로직 (선택 사항, Paging 3에서는 load()에서 키 처리)
25    override fun getRefreshKey(state: PagingState<Int, ExampleData>): Int? {
26        // 가장 최근 접근한 위치(anchorPosition)를 기반으로 키 반환 시도
27        return state.anchorPosition?.let { anchorPosition ->
28            state.closestPageToPosition(anchorPosition)?.prevKey?.plus(1)
29            ?: state.closestPageToPosition(anchorPosition)?.nextKey?.minus(1)
30        }
31    }
32 }
```

다음으로, PagingSource와 PagingData 간의 중개를 위해 리포지토리에서 Pager를 생성합니다.

그림 158. ExampleRepository.kt

```
1 class ExampleRepository(private val apiService: ApiService) {
2     fun getExampleData(): Flow<PagingData<ExampleData>> {
3         return Pager(
4             // Paging 구성 설정 (페이지 크기 등)
5             config = PagingConfig(
6                 pageSize = 20, // 각 페이지에 로드할 항목 수
7                 enablePlaceholders = false // 플레이스홀더 사용 여부
8                 // prefetchDistance = 5 // 미리 로드할 거리 (선택 사항)
9                 // initialLoadSize = 40 // 초기 로드 크기 (선택 사항)
10            ),
11            // PagingSource 인스턴스를 제공하는 팩토리
12            pagingSourceFactory = { ExamplePagingSource(apiService) }
13        ).flow // PagingData 스트림 반환
14    }
15 }
```

다음으로, ViewModel에서 PagingData를 관찰할 수 있습니다.

그림 159. ExampleViewModel.kt

```
1 class ExampleViewModel(private val repository: ExampleRepository) : ViewModel() {
2     val exampleData: Flow<PagingData<ExampleData>> = repository.getExampleData()
3         // viewModelScope 내에서 스트림 캐싱 (구성 변경 시 데이터 유지)
4         .cachedIn(viewModelScope)
5 }
```

마지막으로, 아래 예제와 같이 PagingDataAdapter를 상속받는 커스텀 RecyclerView.Adapter를 생성하여 RecyclerView에 데이터를 전달하고 랜더링할 수 있습니다.

그림 160. ExampleAdapter.kt

```
1 class ExampleAdapter : PagingDataAdapter<ExampleData,
2     ↳ ExampleViewHolder>(DIFF_CALLBACK) {
3
4     override fun onBindViewHolder(holder: ExampleViewHolder, position: Int) {
5         val item = getItem(position) // PagingDataAdapter에서 제공하는 getItem 사용
6         // ViewHolder에 데이터 바인딩 (item이 null일 수 있음에 유의)
7         item?.let { holder.bind(it) }
8     }
9
10    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
11        ↳ ExampleViewHolder {
12        val view = LayoutInflater.from(parent.context).inflate(R.layout.example_item,
13            ↳ parent, false)
14        return ExampleViewHolder(view)
15    }
16
17    // ViewHolder 클래스 (예시)
18    class ExampleViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
19        // 뷰 바인딩 또는 findViewById 등
20        fun bind(item: ExampleData) {
21            // 아이템 데이터로 뷰 업데이트
22        }
23
24    companion object {
25        // DiffUtil 콜백 정의 (ListAdapter와 유사)
26        private val DIFF_CALLBACK = object : DiffUtil.ItemCallback<ExampleData>() {
27            override fun areItemsTheSame(oldItem: ExampleData, newItem: ExampleData):
28                ↳ Boolean {
29                return oldItem.id == newItem.id // 고유 ID 비교
30            }
31        }
32    }
33}
```

```

29     override fun areContentsTheSame(oldItem: ExampleData, newItem:
30         ExampleData): Boolean {
31         return oldItem == newItem // 데이터 내용 비교
32     }
33 }
34 }
```

요약

Jetpack Paging 라이브러리는 점진적 데이터 로딩 구현을 제공합니다. PagingSource, Pager, PagingDataAdapter를 포함한 핵심 구성 요소는 대규모 데이터 셋 처리를 돕습니다. 무한 스크롤, 페이지네이션된 API 또는 대규모 데이터베이스를 처리하는 애플리케이션에 특히 유용하며, 개발자가 데이터 쿼리 및 UI 업데이트를 업데이트하는 부분까지 라이브러리에 맡기고 그 외의 개발에 더 집중할 수 있도록 합니다.

실전 질문

Q) Paging 라이브러리는 데이터 로딩 중 오류를 어떻게 처리하며, 페이지네이션된 데이터 흐름에서 오류 처리 및 재시도 메커니즘을 구현하기 위해 사용해 본 전략은 무엇인가요?

Q) 58. Baseline Profile은 앱의 성능에 어떤 이점을 가져다주나요?

Baseline Profiles¹³⁵는 앱 시작 시간과 런타임 실행을 최적화하기 위한 안드로이드 앱의 성능 향상을 위한 플러그인입니다. Baseline Profiles는 미리 컴파일된 코드 정보를 제공함으로써 코드 해석 및 **just-in-time (JIT)**¹³⁶¹³⁷ 컴파일 단계를 우회하고 더 빠른 앱 실행을 가능하게 합니다. 앱의 첫 실행에 대해서 20-30%의 속도 향상을 기대할 수 있고, 결국 더 부드럽고 효율적인 사용자 경험을 제공할 수 있습니다. Android Runtime (ART)은 이러한

¹³⁵<https://developer.android.com/topic/performance/baselineprofiles/overview>

¹³⁶<https://source.android.com/docs/core/runtime/jit-compiler#flow>

¹³⁷Just-In-Time (JIT) 컴파일은 바이트코드가 실행 직전에 동적으로 기계 코드로 변환되는 런타임 프로세스입니다. 이를 통해 런타임 환경은 실제 실행 패턴을 기반으로 코드를 최적화하여 자주 사용되는 코드 경로의 성능을 향상시킬 수 있습니다.

프로파일을 사용하여 앱 설치 중에 중요한 코드 경로를 식별하고 미리 컴파일하여 응답성을 개선하고 앱 시작 지연 시간을 줄입니다.

Baseline Profiles는 제공된 프로파일에 정확한 코드 경로를 정의하기 위해 [Ahead-of-Time \(AOT\)](#)¹³⁸¹³⁹ 컴파일을 활용합니다. 생성된 프로파일에는 ART가 설치 단계 중에 컴파일하는 클래스 및 메서드 정보가 포함됩니다. 라이브러리 작성자의 경우 Baseline Profiles를 사용하면 라이브러리의 성능을 최적화할 수 있고, 결국 해당 라이브러리를 사용하는 애플리케이션에서도 이 효과를 볼 수 있습니다.

Baseline Profiles 작동 방식

- 중요한 코드 경로 정의:** 개발자는 주요 실행 경로를 프로파일링하거나 애플리케이션 시작 순간부터 가장 일반적인 사용자의 앱 사용 패턴을 기반으로 성능에 중요한 메서드 및 클래스를 미리 정의할 수 있습니다.
- 프로파일 생성:** 프로파일은 Jetpack Macrobenchmark 라이브러리와 같은 도구를 사용하여 생성됩니다. 이를 통해 앱 동작을 기록하고 테스트하여 중요한 코드 경로를 식별할 수 있습니다.
- 프로파일 전파:** 생성된 Baseline Profile은 APK 또는 AAB와 함께 번들로 제공되어 최종 사용자에게 전파되어 배포됩니다.
- 설치 중 최적화:** 앱이 사용자 기기에 설치될 때 ART는 프로파일을 사용하여 미리 정의해 두었던 메서드 및 클래스를 네이티브 코드로 미리 컴파일합니다.

AGP 8.0 이상부터는 [Baseline Profile Gradle 플러그인](#)¹⁴⁰을 활용할 수 있습니다. 해당 플러그인은 Baseline Profiles 생성을 간소화하고 패키지 필터링 옵션을 제공하며 플레이어 컨트롤을 포함한 더 편리한 기능을 제공합니다. [Baseline Profiles 생성](#)¹⁴¹ 가이드라인에 따라 Baseline Profiles를 생성하면 각 모듈 내 /src/main/generated/baselineProfiles 디렉토리 아래에 baseline-prof.txt라는 이름의 생성된 Baseline Profiles를 발견할 수 있습니다. 텍스트 파일을 클릭하면 아래와 같이 앱 최초 실행 시 필요한 클래스 및 메서드의 사전 정의된 패키지 정보를 확인할 수 있습니다.

¹³⁸https://source.android.com/docs/core/runtime#AOT_compilation

¹³⁹Ahead-of-Time (AOT) 컴파일은 코드가 런타임 전에 기계 코드로 컴파일되어 실행 중 Just-In-Time (JIT) 컴파일이 필요 없는 프로세스입니다. 해당 접근 방식은 최적화된 사전 컴파일된 바이너리를 생성하여 성능을 향상시키고 런타임 오버헤드를 줄입니다.

¹⁴⁰<https://mvnrepository.com/artifact/androidx.benchmark/benchmark-baseline-profile-gradle-plugin?repo=google>

¹⁴¹<https://developer.android.com/topic/performance/baselineprofiles/create-baselineprofile>

```

Lcom/skydoves/landscapist/DataSource;
HSPLcom/skydoves/landscapist/DataSource;-->$values() [Lcom/skydoves/landscapist/DataSource;
HSPLcom/skydoves/landscapist/DataSource;--><clinit>()V
HSPLcom/skydoves/landscapist/DataSource;--><init>(Ljava/lang/String;I)V
Lcom/skydoves/landscapist/DrawablePainter;
HSPLcom/skydoves/landscapist/DrawablePainter;--><init>(Landroid/graphics/drawable/Drawable;)V
HSPLcom/skydoves/landscapist/DrawablePainter;-->access$getDrawInvalidateTick(Lcom/skydoves/
landscapist/DrawablePainter;)I
HSPLcom/skydoves/landscapist/DrawablePainter;-->access$setDrawInvalidateTick(Lcom/skydoves/
landscapist/DrawablePainter;I)V
HSPLcom/skydoves/landscapist/DrawablePainter;-->access$setDrawableIntrinsicSize-uvyYCjk(Lcom/skydoves/
landscapist/DrawablePainter;J)V
HSPLcom/skydoves/landscapist/DrawablePainter;-->getCallback()Landroid/graphics/drawable/
Drawable$Callback;
HSPLcom/skydoves/landscapist/DrawablePainter;-->getDrawInvalidateTick()I
HSPLcom/skydoves/landscapist/DrawablePainter;-->getDrawable()Landroid/graphics/drawable/Drawable;
HSPLcom/skydoves/landscapist/DrawablePainter;-->getDrawableIntrinsicSize-NH-jbRc()J
HSPLcom/skydoves/landscapist/DrawablePainter;-->getIntrinsicSize-NH-jbRc()J
HPLcom/skydoves/landscapist/DrawablePainter;-->onDraw(Landroidx/compose/ui/graphics/_drawscope_
DrawScope;)V
PLcom/skydoves/landscapist/DrawablePainter;-->onForgotten()V
HSPLcom/skydoves/landscapist/DrawablePainter;-->onRemembered()V
HSPLcom/skydoves/landscapist/DrawablePainter;-->setDrawInvalidateTick(I)V
HSPLcom/skydoves/landscapist/DrawablePainter;-->setDrawableIntrinsicSize-uvyYCjk(J)V
Lcom/skydoves/landscapist/DrawablePainter$callback$2;
HSPLcom/skydoves/landscapist/DrawablePainter$callback$2;--><init>(Lcom/skydoves/landscapist/
DrawablePainter;)V
HSPLcom/skydoves/landscapist/DrawablePainter$callback$2;-->invoke()Lcom/skydoves/landscapist/
DrawablePainter$callback$2$1;
HSPLcom/skydoves/landscapist/DrawablePainter$callback$2;-->invoke()Ljava/lang/Object;

```

그림 161. baseline-profiles

Google 팀에서 직접 개발하고 관리하는 GitHub 저장소 [Now-in-Android](#)¹⁴²에서 구현 세부 정보를 자세히 살펴볼 수 있습니다. 해당 프로젝트는 각 화면에 대해 Baseline Profiles를 생성하기 위한 포괄적인 가이드를 제공하며, 다양한 유저 저니 (User Journey)에 따라 Baseline Profiles를 생성하고, 성능을 효과적으로 최적화하는 방법을 적용하였습니다.

요약

Baseline Profiles는 중요한 코드 경로 및 패키지 정보를 미리 컴파일하여 앱 시작 시간을 줄이고 원활한 런타임 실행을 보장함으로써 앱 성능을 최적화하는 훌륭한 도구입니다. Jetpack Macrobenchmark 라이브러리와 같은 도구를 활용하면 개발자가 이러한 중요한 경로를 식별하고 정의하여 사용자가 다양한 기기에서 더 빠르고 반응성이 뛰어난 앱을 경험할 수 있도록 보장합니다. Baseline Profiles에 대한 더 깊이 이해하고, 앱 뿐만 아니라 라이브러리에서의 활용법 및 다양한 실전 팁에 대해 학습하고자 하시는 경우, [Improve Your](#)

¹⁴²<https://github.com/android/nowinandroid/tree/main/benchmarks/src/main/kotlin/com/google/samples/apps/nowinandroid>

[Android App Performance With Baseline Profiles](#)¹⁴³ 포스트를 읽어보시면 학습에 많은 도움이 됩니다.

실전 질문

Q) Baseline Profiles를 활용하면 Android Runtime (ART)이 앱 성능을 어떻게 개선시키고, 해당 접근 방식이 기존 Just-In-Time (JIT) 컴파일과 비교하여 가지는 성능적인 이점은 무엇인가요?

¹⁴³<https://medium.com/proandroiddev/improve-your-android-app-performance-with-baseline-profiles-297f388082e6>

카테고리 3: 비즈니스 로직

비즈니스 로직은 일반적으로 데이터 계층(Data layer)¹⁴⁴ 및 도메인 계층 (Domain Layer)¹⁴⁵ 내에 존재하며, UI와 관련성이 적은 작업에 중점을 둡니다. 즉, 네트워크로부터 간단한 데이터 요청, 데이터베이스 쿼리, 데이터 전/후 처리, 로컬 저장소에 값 읽고 쓰기, 그 외 소켓 등 백엔드 서버와의 통신 등이 포함됩니다. 현시점에 모바일 앱 개발을 한다라고 했을 때, 이와 같은 비즈니스 논리와 관련된 작업은 대부분 선택이 아닌 필수인 경우가 많습니다. 따라서 해당 로직들을 올바르게 이해하고, 효율적으로 사용하는 것은 매우 중요합니다.

비즈니스 로직을 학습하는 가장 좋은 방법은 실제로 문제를 해결해 보고, 솔루션을 반복적으로 개선하는 등 실무 기반의 경험을 하는 것입니다. 실제 프로젝트에서의 실무 경험은 다양한 전략과 솔루션의 목적과 필요성에 대한 명확성을 제공하여 견고하고 효율적인 시스템을 설계할 수 있도록 합니다. 널리 채택된 솔루션을 단순히 암기하고 사용하는 대신 그 이면에 있는 편더멘탈을 이해하는 것이 중요합니다. 단순히 API 혹은 라이브러리의 사용법을 익히는 것에서 그치는 게 아니라, 누구나 사용하기 쉽게 설계된 메서드 클 하나가 내부적으로 얼마나 복잡한 과정을 통해 구현되었는지 이해하는 것이 잠재적으로 어려운 문제를 해결하기 위한 통찰력을 길러줍니다. 따라서, 이러한 이해 없이는 학습 자체가 피상적인 방향으로 흐를 수 있고, 장기적으로 기술적인 성장을 더디게 만들 수도 있습니다.

비즈니스 로직을 파악하는 가장 효과적인 방법은 먼저 이 책에서 제시하는 다양한 아이디어나 청사진을 이해하는 것부터 시작해 볼 수 있습니다. 그 후에 이전 경험과 새롭게 터득한 개념을 연결 짓거나, 현재 프로젝트에 적극적으로 적용해 보면서 실무 경험을 발전시킬 수 있습니다. 또한, 실제 시나리오에 따라 동일한 솔루션이라도 완전히 다른 방식으로 구현될 수 있음을 명심해야 합니다. 단순히 답을 암기하는 대신 열린 마음으로 문제 대해 접근해야 하고 다양한 가능성을 분석하며 가장 효율적이고 실용적인 솔루션을 찾으려고 노력해야 합니다. 즉, 최선의 로직은 존재하지만 완벽한 로직은 존재하지 않습니다. 커뮤니티에서 비효율적이라고 악명 높은 솔루션도 직접 사용해 보고 직접 경험해보지 못하면, 알지 못하는 것입니다. 해당 솔루션이 왜 비효율적이고, 어떤 상황에서는 효율적일 수 있는지 직접 경험해보는 것만이 넓은 통찰력을 기르고 빠르게 성장하는 지름길입니다.

¹⁴⁴<https://developer.android.com/topic/architecture/data-layer>

¹⁴⁵<https://developer.android.com/topic/architecture/domain-layer>

Q) 59. 장기적으로 실행되는 백그라운드 작업을 어떻게 관리하나요?

안드로이드는 최적의 리소스 사용과 최신 OS 제한 사항 준수를 보장하면서 장기적으로 실행 가능한 백그라운드 작업을 처리하기 위한 여러 메커니즘을 제공합니다. 각 솔루션은 작업의 성격, 콜 사이트와 상호작용을 해야 하는지 혹은 앱 생명주기와의 상호 작용 여부에 따라 달라집니다.

조건부 작업에 적합한 WorkManager

앱이 닫히거나 기기 재부팅 후에도 실행되어야 하는 작업의 경우 `WorkManager`¹⁴⁶가 공식적으로 권장되는 솔루션입니다. 백그라운드 작업을 관리하고 네트워크 가용성 또는 충전 상태와 같은 제약 조건 하에서 작업이 실행되도록 보장합니다. 예를 들어, 로그 업로드, 데이터 동기화, 비디오와 같이 영상 파일 업로드 등이 일반적인 사용 사례입니다.

그림 162. `WorkManager.kt`

```
1 class UploadWorker(appContext: Context, workerParams: WorkerParameters) :  
2     Worker(appContext, workerParams) {  
3         override fun doWork(): Result {  
4             // 여기서 백그라운드 작업 수행  
5             uploadData()  
6             return Result.success()  
7         }  
8     }  
9     // 작업 예약  
10    val constraints = Constraints.Builder()  
11        .setRequiredNetworkType(NetworkType.CONNECTED) // 네트워크 연결 필요 제약 조건  
12        .build()  
13  
14    val workRequest = OneTimeWorkRequestBuilder<UploadWorker>()  
15        .setConstraints(constraints)  
16        .build()
```

¹⁴⁶<https://developer.android.com/topic/libraries/architecture/workmanager>

17
18 WorkManager.getInstance(context).enqueue(workRequest)

오랜 작업에 적합한 Service

음악 재생이나 위치 추적과 같이 지속적이고 오랜 실행이 필요한 작업에는 **서비스(Service)**가 이상적입니다. 서비스는 UI와 독립적으로 실행되며 앱이 백그라운드에 있을 때도 계속 실행될 수 있습니다. 작업이 노티피케이션과 함께 사용자가 인지할 수 있는 상황에서 실행되어야 하는 경우 **Foreground Services**를 사용합니다.

그림 163. MyForegroundService.kt

```
1 class MyForegroundService : Service() {  
2     override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {  
3         // 장기 실행 작업 수행  
4         startForeground(NOTIFICATION_ID, createNotification())  
5         // START_STICKY, START_REDELIVER_INTENT 등 반환 값 고려  
6         return START_NOT_STICKY  
7     }  
8     // 알림 생성 로직 (Notification Channel 포함)  
9     private fun createNotification(): Notification {  
10        // ...  
11        return NotificationCompat.Builder(this, CHANNEL_ID).build()  
12    }  
13  
14    override fun onBind(intent: Intent?): IBinder? = null  
15 }  
16
```

Kotlin Coroutines 및 Dispatchers 사용하기

앱 생명주기에 연결된 작업의 경우 **Kotlin Coroutines**¹⁴⁷는 Kotlin 언어 수준에서 깔끔하고 구조화된 접근 방식을 제공합니다. 무거운 작업을 오프로드하려면 Dispatchers.IO를 사용하고

¹⁴⁷<https://kotlinlang.org/docs/coroutines-overview.html>

CPU 집약적인 계산에는 Dispatchers.Default를 사용합니다. 해당 접근 방식은 앱이 닫힌 후에도 유지될 필요가 없는 작업에 이상적입니다.

그림 164. `CoroutinesDispatchers.kt`

```
1 class MyViewModel : ViewModel() {  
2     fun fetchData() {  
3         viewModelScope.launch(Dispatchers.IO) { // IO 디스패처에서 네트워크 작업 실행  
4             val data = fetchFromNetwork()  
5             // 결과를 메인 스레드로 전환하여 UI 업데이트  
6             withContext(Dispatchers.Main) {  
7                 updateUI(data)  
8             }  
9         }  
10    }  
11 }
```

 **Additional Tips:** CPU 집약적 (CPU-intensive)란 무엇을 의미할까요? 대체로 'CPU 집약적'이라는 단어 자체에서 오는 느낌 때문에 I/O 작업보다 굉장히 무겁고 CPU를 많이 사용해야 할 것 같은 작업이라고 생각하는 개발자분들이 계시지만, 실제로는 그렇지 않습니다. CPU 집약적인 작업은 연산 능력을 요구하며 사용 가능한 CPU 코어 수에 의해 제약을 받습니다. 즉, 디바이스에서 사용 가능한 코어의 수만큼만 쓰레드를 생성하여 작업합니다. 적합한 작업에는 암호화, 이미지 처리, 비디오 인코딩 또는 데이터 분석과 같은 작업이 포함됩니다. 이러한 작업은 전적으로 CPU에 의존하기 때문에 사용 가능한 코어 수보다 많은 스레드를 추가하면 성능 향상보다는 스레드 경합이 발생하는 경우가 많습니다.

반면, I/O와 관련한 작업은 CPU 코어 수보다 많은 스레드를 사용함으로써 여러 I/O 작업이 CPU 경합을 일으키지 않고 동시에 실행될 수 있도록 합니다. 예를 들어, 8개의 코어를 가진 시스템은 수십 개 또는 수백 개의 I/O 스레드를 효율적으로 관리할 수 있습니다. Kotlin 코루틴과 같은 비동기 프로그래밍 모델은 더 적은 수의 스레드로 수많은 I/O 바운드 작업을 처리할 수 있도록 하여 리소스 활용을 더욱 최적화합니다. 이러한 접근 방식은 리소스 오버헤드를 크게 줄이는 동시에 확장성과 효율성을 향상시킵니다.

이와 관련하여 더 자세히 학습하고 싶으시다면 [Understanding Coroutines Dispatchers¹⁴⁸](#) 포스트를 참고하시길 바랍니다.

시스템 수준 작업에 적합한 JobScheduler

작업이 기기 전체 작업과 관련되고 특정 조건(가령, 충전 중에만 실행)이 필요한 경우 **JobScheduler**를 사용할 수 있습니다. 즉시 실행이 필요하지 않은 작업에 적합합니다. (**WorkManager**는 어차피 내부적으로 **JobScheduler** 등을 활용하므로 일반적인 상황에서는 **WorkManager** 사용을 권장합니다)

그림 165. JobScheduler.kt

```

1 val jobScheduler = context.getSystemService(JobScheduler::class.java)
2 val jobInfo = JobInfo.Builder(JOB_ID, ComponentName(context, MyJobService::class.java))
3     .setRequiredNetworkType(JobInfo.NETWORK_TYPE_UNMETERED) // 네트워크에 연결 중인
4     ↳ 상황에만 실행
5     .setRequiresCharging(true) // 충전 중인 상황에만 실행
6     .setPeriodic(TimeUnit.MINUTES.toMillis(15)) // 작업 주기 설정
7     .build()
8 jobScheduler.schedule(jobInfo)

```

요약

어떤 솔루션을 선택해야 하느냐는 작업의 성격에 따라 달라집니다. **WorkManager**는 신뢰할 수 있는 조건부 영구 작업에 가장 적합하며, **Services**는 미디어 재생이나 위치 추적과 같은 연속적이고 오랜 작업에 적합합니다. **Kotlin Coroutines**는 생명주기에 바인딩된 가벼운 작업에 적합하며, **JobScheduler**는 시스템 수준 작업에 적합합니다. 각 솔루션이 필요한 상황이나 스펙이 다르기 때문에, 올바르게 선택하고 관리하면 안드로이드에서 효율적인 백그라운드 작업 처리를 보장할 수 있습니다.

¹⁴⁸<https://medium.com/proandroiddev/understanding-coroutines-dispatchers-2578dc577851>

실전 질문

Q) 안드로이드 앱에서 백엔드 서버로부터 대용량 파일(수백 MB)을 다운로드하는 기능을 구현해야 합니다. 다운로드는 앱이 닫혀도 계속되어야 하며, 성능 및 네트워크 조건 측면에서 효율적이어야 합니다. WorkManager, Foreground Service, JobScheduler 중 어떤 것을 선택하여 구현하시겠나요?

Q) 60. Json 형식을 객체로 어떻게 직렬화(serialize)하나요?

최신 안드로이드 앱은 원격 서버(REST API, GraphQL 엔드포인트, Firebase 등)와 자주 상호 작용하기 때문에 JSON을 객체로 직렬화하는 것은 안드로이드 개발에서 일반적인 작업입니다. 일반적으로 백엔드에서 JSON 형식으로 데이터를 자주 교환하는데, 이는 가볍고 가독성이 좋으며 플랫폼으로부터 독립적이기 때문입니다.

지속적으로 변동성이 있는 사용자 데이터, 환경 설정 또는 콘텐츠(가령, 뉴스 기사, 메시지)를 리모트 서버(백엔드)에서 가져오기 위해 앱은 Kotlin 객체를 JSON으로 직렬화하여 데이터를 백엔드로 보내 필요한 정보를 요청하고, 서버의 JSON 응답을 받으면 다시 객체로 역직렬화해야 합니다.

직렬화(Serialization)와 역직렬화(Deserialization)란?

- **직렬화(Serialization)**: 객체나 데이터 구조를 나중에 쉽게 저장, 전송 또는 재구성할 수 있는 형식으로 변환하는 프로세스입니다. 안드로이드와 백엔드 통신에서는 종종 객체를 JSON 문자열이나 유사한 구조화된 형식으로 변환하는 것을 의미합니다.
- **역직렬화(Deserialization)**: 직렬화된 데이터(JSON 문자열 등)를 가져와 애플리케이션에서 작업할 수 있는 메모리 내 객체로 다시 재구성하는 역 프로세스입니다.

수동 직렬화 및 역직렬화

외부의 솔루션을 사용하지 않고도 직접 수동적으로 문자열 조작 및 파싱 기술을 통해 객체를 JSON 문자열로 변환하고, 그 반대로 변환하여 수동으로 JSON을 직렬화하고 역직렬화할 수도 있습니다.

수동 직렬화는 객체의 속성을 JSON 문자열로 바꾸는 것을 동반합니다.

그림 166. ManualSerialization.kt

```
1 data class User(val name: String, val age: Int)
2
3 fun serializeUser(user: User): String {
4     // 각 속성을 JSON 형식에 맞게 문자열로 변환
5     return """{
6         "name": "${user.name}",
7         "age": ${user.age}
8     """".trimIndent() // 들여쓰기 제거
9 }
10
11 // 사용 예제
12 val user = User("John", 30)
13 val jsonString = serializeUser(user)
14 // 출력: {"name":"John", "age":30}
```

반면에 수동 역직렬화는 JSON 문자열을 파싱하고 값을 추출하여 객체를 재구성하는 것을 동반합니다.

그림 167. ManualDeserialization.kt

```
1 fun deserializeUser(json: String): User {
2     // 정규 표현식을 사용하여 값 추출 (간단한 예시)
3     val nameRegex = """name"\s*:\s*([^\"]*)""".toRegex()
4     val ageRegex = """age"\s*:\s*(\d+)""".toRegex()
5
6     val name = nameRegex.find(json)?.groups?.get(1)?.value ?: ""
7     val age = ageRegex.find(json)?.groups?.get(1)?.value?.toIntOrNull() ?: 0
8
9     return User(name, age)
10 }
11
12 // 사용 예제
13 val jsonString = """{"name":"John", "age":30}"""
```

```
14 val user = deserializeUser(jsonString)
15 // 출력: User(name="John", age=30)
```

이 방법은 유지 관리, 안전성 및 확장성 문제로 인해 프로덕션 앱에는 권장되는 방법은 아니지만, 학습 목적이나 의존성을 최소화해야 하는 가벼운 경우에는 유용할 수 있습니다. 수동적으로 직/역직렬화를 하는 대신, **kotlinx.serialization**, **Moshi**, **Gson**과 같은 라이브러리를 통해 해당 프로세스를 훨씬 더 효과적으로 구현하여, JSON 문자열을 Kotlin 또는 Java 객체로 또는 그 반대로 역변환할 수 있습니다.

kotlinx.serialization

JetBrains에서 개발한 **kotlinx.serialization**¹⁴⁹은 Kotlin과 직접적으로 통합되어 Kotlin의 언어적 기능을 활용하도록 설계되었습니다. 어노테이션을 사용하여 직렬화 동작을 정의하고 JSON뿐만 아니라 ProtoBuf와 같은 다른 형식과도 원활하게 작동합니다.

그림 168. Using **kotlinx.serialization** for JSON Serialization

```
1 import kotlinx.serialization.*
2 import kotlinx.serialization.json.*
3
4 @Serializable // 직렬화 가능 클래스임을 명시
5 data class User(val name: String, val age: Int)
6
7 val json = """{"name": "John", "age": 30}"""
8 // JSON 문자열을 User 객체로 역직렬화
9 val user: User = Json.decodeFromString<User>(json)
10 // User 객체를 JSON 문자열로 직렬화
11 val serializedJson: String = Json.encodeToString(user)
```

kotlinx.serialization은 Kotlin 컴파일러 플러그인을 사용하여 Kotlin 객체를 JSON 문자열로 변환(직렬화)하고 JSON 문자열을 다시 Kotlin 객체로 파싱(역직렬화)하는 타입 안정성을 보장하며, 내부적으로 리플렉션을 사용하지 않는 메커니즘을 제공하므로 최신 안드로이드 및 Kotlin 개발에서 가장 선호되는 방법 중 하나입니다.

¹⁴⁹<https://github.com/Kotlin/kotlinx.serialization>

내부적으로 Kotlin은 Moshi의 리플렉션 모드 및 Gson과 달리 무거운 런타임 리플렉션 없이 효율적인 직렬화를 수행하기 위해 컴파일러가 생성하는 코드에 의존합니다.

Moshi

Square에서 개발한 [Moshi](#)¹⁵⁰는 타입 안전성과 Kotlin 지원을 강조하는 최신 JSON 라이브러리입니다. Gson과 달리 Moshi는 **Kotlin의 null 가능성 및 기본 매개변수**를 기본적으로 지원합니다. 이로 인해 Moshi는 Kotlin을 사용하는 프로젝트에 아주 적합합니다.

그림 169. Using Moshi for JSON Serialization

```

1 data class User(val name: String, val age: Int)
2
3 // Moshi 인스턴스 생성 (Kotlin 지원 추가 필요)
4 val moshi = Moshi.Builder()
5     .add(KotlinJsonAdapterFactory()) // Kotlin 지원 추가
6     .build()
7 // User 클래스에 대한 어댑터 가져오기
8 val adapter: JsonAdapter<User> = moshi.adapter(User::class.java)
9 val json = """{"name": "John", "age": 30}"""
10
11 // JSON을 객체로 역직렬화
12 val user: User? = adapter.fromJson(json)
13 // 객체를 JSON으로 직렬화
14 val serializedJson: String? = user?.let { adapter.toJson(it) }

```

Moshi에서는 JSON 직렬화 및 역직렬화를 처리하는 두 가지 주요 접근 방식이 있는데, 리플렉션에 기반한 동작 방식과 및 컴파일 타임에 코드를 생성(codegen)하는 방식 두 가지입니다. 둘 다 JSON을 Kotlin/Java 객체로 또는 그 반대로 역변환할 수 있지만 성능, 런타임 동작 및 내부 작동 방식에서 상당한 차이가 있습니다.

- **리플렉션 기반 Moshi(Reflection-Based Moshi)**: 리플렉션 기반 Moshi¹⁵¹는 Java 리플렉션을 사용하여 런타임에 동적으로 JSON 어댑터를 생성하므로 추가 설정 필요 없이 사용법이 간단하다는 장점이 있지만, 반드시 런타임 오버헤드가 발생합니다.

¹⁵⁰<https://github.com/square/moshi>

¹⁵¹<https://github.com/square/moshi?tab=readme-ov-file#reflection>

- **코드 생성 기반 Moshi(Codegen-Based Moshi)**: 코드 생성 기반 Moshi¹⁵²는 어노테이션 프로세스(Kotlin SymbolProcessor¹⁵³) 기법을 통해 컴파일 타임에 JSON 어댑터를 생성하여, 더 빠른 런타임 성능과 컴파일 타임 오류 검사를 제공합니다.

Moshi의 리플렉션 기반 어댑터는 빠른 셋업 환경을 제공하지만, 런타임에 성능 오버헤드가 발생하기 때문에 권장되는 솔루션은 아닙니다. 반면 Moshi의 코드 생성 기반의 플러그인은 컴파일 타임에 최적화된 어댑터를 생성하여 더 나은 성능을 제공하므로 일반적인 상황에서 더 선호됩니다.



Additional Tips: 개발자 커뮤니티에는 Kotlin IR(Intermediate Representation) 컴파일러 플러그인을 사용하여 구축된 MoshiX¹⁵⁴라는 또 다른 라이브러리가 있습니다. 컴파일 타임에 고도로 최적화된 코드를 생성하여 KSP 기반 또는 리플렉션 기반 워크플로에 비해 더 나은 성능을 제공합니다.

Gson

Google에서 개발한 Gson¹⁵⁵은 널리 사용되는 JSON 라이브러리입니다. Java 객체를 JSON으로 직렬화하고 JSON을 다시 Java 객체로 역직렬화할 수 있습니다. 간단한 API와 통합 용이성 덕분에 인기 있는 라이브러리입니다.

그림 170. Using Gson for JSON Serialization

```

1 data class User(val name: String, val age: Int)
2
3 val gson = Gson()
4 val json = """{"name": "John", "age": 30}"""
5 val user = gson.fromJson(json, User::class.java) // JSON을 객체로 역직렬화
6 val serializedJson = gson.toJson(user) // 객체를 JSON으로 직렬화

```

¹⁵²<https://github.com/square/moshi?tab=readme-ov-file#codegen>

¹⁵³<https://kotlinlang.org/docs/ksp-overview.html>

¹⁵⁴<https://github.com/ZacSweers/MoshiX>

¹⁵⁵<https://github.com/google/gson>

그러나 **Gson**보다 **kotlinx.serialization** 또는 **Moshi**를 선택해야 하는 명백한 이유가 있습니다. 그 이유는 아래와 같습니다.

1. **더 나은 Kotlin 지원**: Gson은 Java용으로 설계되었으며 Kotlin 기능(가령, 기본 매개변수, val/var 차이, null 허용성)을 **kotlinx.serialization** 또는 **Moshi**만큼 자연스럽게 처리하지 못합니다.
2. **성능 및 효율성**: **kotlinx.serialization** 및 **Moshi**(특히 코드 생성 라이브러리 사용 시)는 런타임에 리플렉션에 크게 의존하는 Gson보다 빠르고 메모리 효율적입니다.
3. **멀티플랫폼 호환성**: **kotlinx.serialization**은 Kotlin Multiplatform(KMP)을 완벽하게 지원하는 반면, Moshi/Gson은 JVM 위에서 동작하도록 설계되어 있어 크로스 플랫폼 앱에는 적합하지 않습니다.
4. **컴파일 타임 안전성(Compile-Time Safety)**: 컴파일 타임에 필요한 코드를 생성하는 **kotlinx.serialization** 및 **Moshi**는 컴파일 타임에 대부분의 오류를 감지하여 런타임 크래시를 줄이는 반면, Gson은 종종 런타임 오류로 크래시를 발생하도록 합니다.

다음은 Gson과 Moshi의 핵심 기여자인 Jake Wharton의 코멘트입니다.

Jake Wharton ([reddit:r/androiddev¹⁵⁶](https://www.reddit.com/r/androiddev/comments/dhjdk2/moshi_vs_kotlinx_serialization_with_retrofit/)): Moshi는 이름만 빼면 모든 면에서 Gson v3이라고 불려도 되기 때문에 마이그레이션이 쉬울 것입니다. Kotlin 지원은 약간의 셋업이 필요하지만, 사용하기 편리하기를 바랍니다. 가장 큰 단점은 의외로 의존성인 `kotlin-reflect`를 사용하거나 빌드 성능에 영향을 미치는 코드 생성 기법을 사용해야 한다는 것입니다.

`kotlinx.serialization`도 훌륭하지만, 기능이 적습니다(Moshi는 Gson보다 기능이 적도록 설계되었기 때문에 당연한 이야기입니다). 하지만 멀티플랫폼과 같은 기능들을 사용할 수 있습니다. 유일한 단점은 현재 스트리밍 기능이 없다는 것입니다. 따라서 응답 본문이 방대하면 메모리 부담이 커집니다.

저는 모델을 JS와 공유하기 때문에 멀티플랫폼이 필요하므로 `kotlinx.serialization`을 사용합니다.

¹⁵⁶https://www.reddit.com/r/androiddev/comments/dhjdk2/moshi_vs_kotlinx_serialization_with_retrofit/

요약

JSON을 객체로 직렬화하기 위해 Gson, Moshi, kotlinx.serialization과 같은 라이브러리는 JSON 문자열을 Kotlin 객체에 매핑하는 효율적인 API를 제공합니다. **kotlinx.serialization**은 Kotlin 우선 및 Multiplatform 프로젝트에 가장 적합한 선택으로, 컴파일 타임 안전성, 경량 런타임 및 완전한 네이티브 Kotlin 지원을 제공합니다. **Moshi**는 빠른 성능과 더 안전한 JSON 처리가 필요한 안드로이드 중심 앱(특히 코드 생성 사용 시)에 이상적이며, **Gson**은 JVM 기반의 프로젝트에 통합하기 쉽지만 성능, Kotlin 지원 및 현대 앱 개발의 모범 사례에서는 뒤처집니다. 더 많은 히스토리를 살펴보고 싶으시다면 Reddit에서 제기된 토론 [Why use Moshi over Gson?](#)¹⁵⁷을 살펴보시길 바랍니다.

실전 질문

Q) API의 JSON 응답이 주어졌을 때, 이를 Kotlin 데이터 클래스로 어떻게 역직렬화하시겠습니까? Kotlin을 사용하는 프로젝트에서 어떤 라이브러리를 선택할 것이고, 그 이유는 무엇인가요?

Q) Kotlin 데이터 클래스에 정의되지 않은 누락되거나 추가적인 필드가 있는 JSON 문자열을 역직렬화해야 하는 경우 어떻게 처리하실 건가요?

Q) 61. 원격 데이터를 가져오기 위해 네트워크 요청을 어떻게 처리하며, 효율성과 신뢰성을 위해 어떤 라이브러리나 기술을 사용하나요?

일반적으로 [Retrofit](#)¹⁵⁸과 [OkHttp](#)¹⁵⁹는 안드로이드 개발에서 백엔드로부터 네트워크 요청을 하는 데 흔히 사용되는 라이브러리입니다. Retrofit은 선언적 인터페이스를 제공하여 API 상호 작용을 단순화하는 반면, OkHttp는 기본적인 HTTP 클라이언트 역할을 하여 연결 풀링, 캐싱 및 효율적인 통신을 제공합니다.

Retrofit을 사용한 네트워크 요청

Retrofit은 HTTP 요청을 깔끔하고 타입 세이프한 API 인터페이스로 추상화합니다. JSON 응답을 Kotlin 또는 Java 객체로 변환하기 위해 Gson 또는 Moshi와 같은 직렬화 라이브러

¹⁵⁷https://www.reddit.com/r/androiddev/comments/684flw/why_use_moshi_over_gson/

¹⁵⁸<https://square.github.io/retrofit/>

¹⁵⁹<https://square.github.io/okhttp/>

리와 원활하게 작동합니다.

Retrofit을 사용하여 백엔드 데이터를 가져오려면 다음 단계를 따릅니다.

1. **API 인터페이스 정의**: 어노테이션을 사용하여 API 엔드포인트와 HTTP 메서드를 선언합니다.

그림 171. API Interface with Retrofit

```
1 interface ApiService {  
2     @GET("data")  
3     // 코루틴 지원을 위한 suspend 함수 또는 Call<DataModel> 반환  
4     suspend fun fetchData(): Response<DataModel>  
5 }
```

2. **Retrofit 인스턴스 설정**: 베이스 URL과 JSON 직렬화를 위해ConverterFactory를 아래와 같이 하여 Retrofit을 구성합니다.

그림 172. Retrofit Instance Setup

```
1 // Kotlinx Serialization Converter Factory 예시  
2 val retrofit = Retrofit.Builder()  
3     .baseUrl("https://api.example.com/")  
4     .addConverterFactory(Json{ ignoreUnknownKeys = true } // 알 수 없는 키 무시 설정 추가  
5         .asConverterFactory("application/json".toMediaType()))  
6     .build()  
7  
8 // API 서비스 인스턴스 생성  
9 val apiService = retrofit.create(ApiService::class.java)
```

3. **네트워크 호출하기**: 코루틴을 사용하여 API를 비동기적으로 호출합니다.

그림 173. Making a Network Request with Retrofit

```
1 viewModelScope.launch { // ViewModel 스코프 또는 다른 코루틴 스코프 사용
2     try {
3         val response = apiService.fetchData()
4         if (response.isSuccessful) {
5             // 성공적인 응답 처리
6             val data = response.body()
7             if (data != null) { // Nullable일 수 있으므로 확인 필요
8                 Log.d(TAG, "Data fetched: $it")
9                 // UI 업데이트 또는 데이터 처리
10            } else {
11                Log.e(TAG, "Response body is null")
12            }
13        } else {
14            // 오류 응답 처리
15            Log.e(TAG, "Error: ${response.code()} - ${response.message()}")
16            // 오류 본문 파싱 (response.errorBody()) 등
17        }
18    } catch (e: Exception) { // 네트워크 오류 또는 기타 예외 처리
19        Log.e(TAG, "Network request failed", e)
20        // 사용자에게 오류 표시 등
21    }
22 }
```

OkHttp를 사용한 커스텀 HTTP 요청

OkHttp는 헤더, 캐싱 등을 세밀하게 제어할 수 있도록 HTTP 요청을 관리하기 위해 더 직접적인 접근 방식을 보여줍니다.

그림 174. Basic GET Request with OkHttp

```
1 val client = OkHttpClient()
2
3 val request = Request.Builder()
4     .url("https://api.example.com/data")
5     .build()
6
7 // 비동기 요청 실행
8 client.newCall(request).enqueue(object : Callback {
9     override fun onFailure(call: Call, e: IOException) {
10         // 네트워크 오류 처리
11         e.printStackTrace()
12     }
13
14     override fun onResponse(call: Call, response: Response) {
15         // 응답 스레드는 백그라운드 스레드일 수 있으므로 UI 업데이트 시 주의
16         response.use { // response.body().close() 자동 호출
17             if (response.isSuccessful) {
18                 val responseBody = response.body?.string() // 응답 본문 읽기 (한 번만 가능)
19                 Log.d(TAG, "Response: $responseBody")
20                 // UI 업데이트는 메인 스레드에서 수행
21                 // runOnUiThread { /* UI 업데이트 */ }
22             } else {
23                 Log.e(TAG, "Error: ${response.code}")
24             }
25         }
26     }
27 })
```

OkHttp와 Retrofit 통합하기

Retrofit은 내부적으로 OkHttp를 HTTP 클라이언트로 사용합니다. 인터셉터를 추가하여 로깅, 인증 또는 캐싱과 같은 OkHttp의 동작을 커스텀할 수 있습니다.

그림 175. Custom OkHttpClient with Retrofit

```
1 val okHttpClient = OkHttpClient.Builder()
2     // 로깅 인터셉터 추가 (예시)
3     .addInterceptor(HttpLoggingInterceptor().apply { level =
4         → HttpLoggingInterceptor.Level.BODY })
5     // 인증 헤더 추가 인터셉터
6     .addInterceptor { chain ->
7         val originalRequest = chain.request()
8         val newRequest = originalRequest.newBuilder()
9             .header("Authorization", "Bearer your_token") // 실제 토큰 사용
10            .build()
11         chain.proceed(newRequest)
12     }
13     .connectTimeout(30, TimeUnit.SECONDS) // 타임아웃 설정 (선택 사항)
14     .readTimeout(30, TimeUnit.SECONDS)
15     .build()
16
17 val retrofit = Retrofit.Builder()
18     .baseUrl("https://api.example.com/")
19     .client(okHttpClient) // 커스텀 OkHttpClient 설정
20     .addConverterFactory(GsonConverterFactory.create()) // 또는 다른 Converter Factory
21     .build()
```

해당 설정은 인증 토큰 첨부 또는 디버깅 목적의 상세 로깅 활성화와 같은 기능을 추가할 수 있도록 합니다.

요약

Retrofit과 OkHttp는 안드로이드에서 네트워크 요청을 처리하기 위한 견고한 솔루션을 제공합니다. Retrofit은 내부적으로 굉장히 복잡한 처리를 하고 있음에도 API 표면은 개발자들이 사용하기 쉽게 설계되었으며, HTTP 호출 생성 및 실행을 단순화하는 반면, OkHttp는 네트워크 동작을 사용자가 쉽게 커스텀할 수 있는 유연성을 제공합니다. 두 라이브러리를 함께 활용하면 안드로이드 애플리케이션에서 쉽고 효율적인 네트워킹 시스템을 구현할 수 있습니다.

실전 질문

Q) 앱에서 동시에 여러 API 요청을 수행하고 UI를 업데이트하기 전에 결과를 결합해야 한다고 가정해 봅시다. Retrofit과 코루틴을 사용하여 이를 효율적으로 구현하려면 어떻게 해야 하나요?

Q) API 응답 실패 시 어떻게 처리하고, 재시도 메커니즘은 어떻게 구현하나요?

Pro Tips for Mastery: OkHttp Authenticator 및 Interceptor를 사용하여 OAuth 토큰 갱신하기

OAuth로 보호되는 API로 작업할 때 토큰 만료 및 갱신 시나리오를 처리하는 것이 일반적입니다. OkHttp는 토큰을 가로채고 새로 고치는 두 가지 주요 메커니즘이 **Authenticator**¹⁶⁰와 **Interceptor**¹⁶¹를 제공합니다. 둘 다 다른 목적을 가지고 있으며 애플리케이션의 특정 요구 사항에 따라 선택적으로 사용할 수 있습니다.

OkHttp Authenticator

OkHttp의 Authenticator 인터페이스는 토큰 만료와 같은 인증 문제를 처리하도록 따로 설계되었습니다. 서버가 401 Unauthorized 상태 코드를 응답하면 Authenticator가 호출되어 업데이트된 인증 자격 증명이 포함된 새 요청을 제공합니다.

다음은 토큰을 새로 고치기 위해 Authenticator를 구현하는 방법입니다.

그림 176. TokenAuthenticator.kt

```

1 class TokenAuthenticator(
2     private val tokenProvider: TokenProvider // 토큰 제공 및 새로고침 로직 캡슐화
3 ) : Authenticator {
4
5     override fun authenticate(route: Route?, response: Response): Request? {
6         // 이전 요청이 이미 인증 헤더를 가지고 있었는지 확인 (무한 루프 방지)
7         val previousToken = response.request.header("Authorization")
8
9         // 토큰 동기화 및 새로고침 (한 번만 시도하도록 제어)

```

¹⁶⁰<https://square.github.io/okhttp/3.x/okhttp/okhttp3/Authenticator.html>

¹⁶¹<https://square.github.io/okhttp/features/interceptors/>

```
10     synchronized(this) {
11         // 현재 토큰과 이전 요청의 토큰 비교
12         val currentToken = tokenProvider.getToken() // 현재 저장된 토큰 가져오기
13         // 토큰이 변경되지 않았거나 새로고침 실패 시 null 반환 (더 이상 재시도 안 함)
14         if (previousToken != null && previousToken == "Bearer $currentToken") {
15             val newToken = tokenProvider.refreshToken() // 동기적으로 토큰 새로고침
16             ← 시도
17             if (newToken == null) {
18                 // 새로고침 실패 시 처리 (예: 로그아웃)
19                 tokenProvider.clearToken() // 토큰 제거
20                 return null // 인증 실패
21             }
22
23             // 새로고침된 토큰 또는 현재 토큰으로 새 요청 생성
24             val refreshedToken = tokenProvider.getToken()
25             if (refreshedToken != null) {
26                 return response.request.newBuilder()
27                     .header("Authorization", "Bearer $refreshedToken")
28                     .build()
29             }
30         }
31         return null // 토큰 없으면 인증 불가
32     }
33 }
34
35 // TokenProvider 인터페이스/클래스 (예시)
36 interface TokenProvider {
37     fun getToken(): String?
38     fun refreshToken(): String? // 동기 방식으로 구현 필요
39     fun clearToken()
40     // ... (토큰 저장/관리 로직)
41 }
```

TokenProvider는 일반적으로 새로 고침 엔드포인트에 동기 네트워크 호출을 하여 토큰을 새로 고치는 역할을 하는 커스텀 클래스입니다.

Authenticator를 사용하려면 OkHttpClient를 생성할 때 아래와 같이 설정이 필요합니다.

그림 177. OkHttpClientSetup.kt

```
1 val okHttpClient = OkHttpClient.Builder()
2     .authenticator(TokenAuthenticator(tokenProvider))
3     // ... 다른 설정 ...
4     .build()
```

OkHttp Interceptor 사용하기

Interceptor는 토큰 추가 및 갱신 로직을 처리할 수 있는 더 유연한 접근 방식입니다. Authenticator와 달리 Interceptor는 요청 또는 응답이 처리되기 전에 가로채서 수정할 수 있습니다.

일반적인 구현에는 401 상태 코드에 대한 응답을 확인하고 토큰을 인라인으로 갱신하는 방법이 있습니다.

그림 178. TokenInterceptor.kt

```
1 class TokenInterceptor(
2     private val tokenProvider: TokenProvider
3 ) : Interceptor {
4     override fun intercept(chain: Interceptor.Chain): Response {
5         // 현재 토큰 가져오기
6         val token = tokenProvider.getToken()
7         // 요청에 토큰 추가 (토큰이 있는 경우)
8         val originalRequest = chain.request()
9         val requestWithToken = if (token != null) {
10             originalRequest.newBuilder()
11                 .header("Authorization", "Bearer $token")
12                 .build()
13         } else {
```

```
14         originalRequest // 토큰 없으면 원본 request 사용
15     }
16
17     // 토큰 요청
18     var response = chain.proceed(requestWithToken)
19
20     // 토큰 만료 확인 (401 응답)
21     if (response.code == 401) {
22         // 동기화 블록으로 토큰 새로고침 중복 방지
23         synchronized(this) {
24             val newToken = tokenProvider.refreshToken() // 동기적으로 토큰 새로고침
25             if (newToken != null) {
26                 // 새 토큰으로 request 다시 만들기
27                 val newRequest = requestWithToken.newBuilder()
28                     .header("Authorization", "Bearer $newToken")
29                     .build()
30                 // 이전 응답 닫기 (리소스 누수 방지)
31                 response.close()
32                 // 새롭게 만들어진 request로 재시도
33                 response = chain.proceed(newRequest)
34             } else {
35                 // 간신히 실패 시 처리 (로그아웃 등)
36                 tokenProvider.clearToken()
37             }
38         }
39     }
40     return response
41 }
42 }
```

마찬가지로, OkHttpClient를 생성할 때 아래와 같이 인터셉터를 설정해줘야 합니다.

그림 179. OkHttpClientSetup.kt

```
1 val okHttpClient = OkHttpClient.Builder()
2     .addInterceptor(TokenInterceptor(tokenProvider))
3     // ... 다른 설정 ...
4     .build()
```

Authenticator와 Interceptor의 주요 차이점

- 목적:** Authenticator는 일반적으로 401 응답에 의해 트리거되는 인증 문제를 처리하도록 설계되었습니다. 반면 Interceptor는 요청 및 응답 처리 모두에 대해 더 세분화된 제어를 처리할 수 있습니다.
- 자동적으로 트리거:** Authenticator는 401 응답에 대해 자동으로 호출되는 반면, Interceptor는 특정 시나리오를 감지하고 처리하기 위한 수동 로직이 필요합니다.
- 사례:** 간단한 인증 문제에는 Authenticator를 사용하고 복잡한 요청/응답 처리가 필요한 시나리오에는 Interceptor를 사용하는 것이 권장됩니다.

요약

간단히 OAuth 토큰을 갱신하는 것이 목적이라면 Authenticator를 통해 401 서버 응답을 처리하는 것이 적합하며, Interceptor는 커스텀 토큰 관리 등을 위한 더 세부적인 유연함을 제공합니다. 둘 사이의 선택은 애플리케이션의 특정 요구 사항과 복잡성에 따라 달라집니다. 두 접근 방식 모두 API 호출을 중단하지 않고 투명하게 토큰을 갱신시켜 원활한 사용자 경험을 보장합니다.

💡 Pro Tips for Mastery: Retrofit CallAdapter란?

Retrofit의 **CallAdapter**¹⁶²는 개발자가 Retrofit API 메서드의 반환 타입을 사용자 정의 할 수 있도록 하는 추상화 API입니다. 기본적으로 Retrofit API 메서드는 동기식 또는 비동기식으로 실행될 수 있는 HTTP 요청을 나타내는 `Call<T>` 객체를 반환합니다. 하지만, **CallAdapter**를 사용하면 이 기본 반환 타입을 `LiveData`, `Flow`, `RxJava` 또는 커스텀 타입과 같은 다른 타입으로 변환할 수 있습니다.

¹⁶²<https://square.github.io/retrofit/2.x/retrofit/retrofit2/CallAdapter.html>

CallAdapter는 반응형 프로그래밍 또는 코루틴 기반 접근 방식과 같이 안드로이드 개발에서 사용되는 다양한 프로그래밍 패러다임 또는 라이브러리에 Retrofit을 적용할 수 있도록 하는 데 중요한 역할을 합니다.

CallAdapter 작동 방식

Retrofit은 CallAdapter 인스턴스를 생성하기 위해 CallAdapter.Factory를 사용합니다. CallAdapter는 런타임에 Call<T> 객체를 원하는 타입으로 변환하는 역할을 합니다. 이 프로세스는 Retrofit이 API 인터페이스에 대한 프록시를 생성할 때 수행됩니다.

Retrofit의 기본 CallAdapter

기본적으로 Retrofit에는 Call<T> 객체를 직접 반환하는 CallAdapter가 포함되어 있습니다. 다른 타입을 원하면 적절한 라이브러리를 사용하거나, 커스텀 CallAdapter를 작성해야 합니다.

Retrofit과 Coroutine CallAdapter 사용하기

Kotlin에서 **Kotlin Coroutines Adapter**를 (Retrofit에 내장되어 있기 때문에 추가적인 설정은 필요 없음) 사용하면 Retrofit API 인터페이스에서 suspend 함수를 사용할 수 있습니다. Call<T> 타입을 반환하는 대신 어댑터는 해당 메서드를 실행하면서 실제 네트워크 호출 결과를 반환하거나 오류에 대한 예외를 던지도록 합니다.

다음은 Retrofit과 함께 suspend 수정자를 사용하는 방법입니다.

그림 180. ExampleApi.kt

```
1 interface ExampleApi {
2     @GET("users")
3     suspend fun getUsers(): List<User> // suspend 함수로 직접 결과 반환
4 }
```

이 경우 Retrofit은 내부적으로 **Kotlin Coroutines Adapter**를 사용하여 원래 기본 타입인 Call<List<User>>에서 List<User>를 반환하는 suspend 함수로 변환합니다.

커스텀 CallAdapter

LiveData를 Retrofit과 통합하려면 다음과 같이 커스텀 CallAdapter를 만들 수 있습니다.

그림 181. LiveDataCallAdapter.kt

```
1 import androidx.lifecycle.LiveData
2 import retrofit2.*
3 import java.lang.reflect.ParameterizedType
4 import java.lang.reflect.Type
5
6 // LiveData를 반환하는 CallAdapter 구현
7 class LiveDataCallAdapter<R>(private val responseType: Type) : CallAdapter<R,
8     LiveData<R> {
9
10    override fun responseType(): Type = responseType
11
12    override fun adapt(call: Call<R>): LiveData<R> {
13        return object : LiveData<R>() {
14            private var started = AtomicBoolean(false)
15            override fun onActive() {
16                super.onActive()
17                if (started.compareAndSet(false, true)) { // 한 번만 실행되도록 보장
18                    call.enqueue(object : Callback<R> {
19                        override fun onResponse(call: Call<R>, response: Response<R>) {
20                           .postValue(response.body()) //.postValue 사용 (백그라운드
21                                ↪ 스레드 실행 가능성)
22                        }
23
24                        override fun onFailure(call: Call<R>, t: Throwable) {
25                            // 오류 처리: null 또는 특정 오류 상태 전달
26                           .postValue(null)
27                        }
28                    })
29                }
30            }
31        }
32    }
33}
```

```
32
33 // LiveDataCallAdapter 인스턴스를 생성하는 Factory
34 class LiveDataCallAdapterFactory : CallAdapter.Factory() {
35     override fun get(
36         returnType: Type,
37         annotations: Array<Annotation>,
38         retrofit: Retrofit
39     ): CallAdapter<*, *?> {
40         // 반환 타입이 LiveData인지 확인
41         if (getRawType(returnType) != LiveData::class.java) {
42             return null
43         }
44         // LiveData의 제네릭 타입 추출 (LiveData<List<User>>에서 List<User> 추출)
45         val observableType = getParameterUpperBound(0, returnType as ParameterizedType)
46         val rawObservableType = getRawType(observableType)
47         // 제네릭 타입이 Response인 경우 처리 (선택 사항)
48         if (rawObservableType == Response::class.java) {
49             if (observableType !is ParameterizedType) {
50                 throw IllegalArgumentException("Response must be parameterized as
51                     ↪ Response<Foo> or Response<? extends Foo>")
52             }
53             val bodyType = getParameterUpperBound(0, observableType)
54             return LiveDataCallAdapter<Any>(bodyType) // Response<T>의 T 타입 사용
55         }
56         // 일반 LiveData<T> 반환 타입 처리
57         return LiveDataCallAdapter<Any>(observableType)
58     }
}
```

커스텀 어댑터를 사용하려면 Retrofit를 생성할 때 아래와 같이 직접 개발한 LiveDataCallAdapterFactory 팩토리를 추가합니다.

그림 182. RetrofitSetup.kt

```

1 val retrofit = Retrofit.Builder()
2     .baseUrl("https://example.com/")
3     .addConverterFactory(GsonConverterFactory.create()) // Converter Factory 먼저 추가
4     .addCallAdapterFactory(LiveDataCallAdapterFactory()) // CallAdapter Factory 추가
5     .build()

```

이제 Retrofit의 API 메서드가 LiveData 타입을 반환할 수 있게 되었습니다.

그림 183. ExampleApi.kt

```

1 interface ExampleApi {
2     @GET("users")
3     fun getUsers(): LiveData<List<User>> // LiveData 반환 타입
4 }

```

요약

Retrofit의 **CallAdapter**는 기본적인 `Call<T>` 타입 유형을 `LiveData`, `Flow`, `RxJava`의 `Observable` 또는 커스텀 유형을 포함한 다양한 타입으로 변환할 수 있는 유연성을 제공합니다. 이를 통해 선호하는 아키텍처 또는 라이브러리와 Retrofit을 원활하게 통합할 수 있습니다. Retrofit에는 `Call<T>` 및 코루틴에 대한 기본 `CallAdapter`가 포함되어 있지만, 커스텀 어댑터를 직접 구현하여 특정 요구 사항에 맞는 세밀한 사용 사례를 구현할 수 있습니다. 커스텀 **CallAdapter**에 대한 자세한 내용은 [Modeling Retrofit Responses With Sealed Classes and Coroutines](#)¹⁶³를 통해 상세하게 학습하실 수 있습니다.

Q) 62. 대규모 데이터 셋을 효율적으로 로드하는 데 왜 페이징 기법이 필요하고, RecyclerView로 구현해 본 경험이 있나요?

페이징 시스템은 대규모 데이터 셋을 처리할 때 데이터 로드 및 화면에 렌더링 하는 방식을 최적화합니다. 데이터를 더 작고 쉽게 관리 가능한 청크로 가져옴으로써 원활한 앱 성능과 더 나은 사용자 경험을 보장합니다.

¹⁶³<https://medium.com/proandroiddev/modeling-retrofit-responses-with-sealed-classes-and-coroutines-9d6302077dfe>

데이터를 더 작은 페이지로 로드하면 메모리 사용량이 크게 줄어들어 잠재적인 메모리 부족 문제를 방지할 수 있습니다. 또한 한 번에 많은 데이터를 로드하는 것이 아니라, 현재 화면에 보여지는데 필요한 데이터만 가져와 렌더링 하므로 초기 로드 시간이 단축됩니다. 더 필요한 데이터는 필요할 때만 요청되므로, 네트워크 사용량이 최소화되어 특히 대역폭이 제한된 시나리오에서 리소스를 효율적으로 사용할 수 있습니다.

사용자 경험 관점에서 페이징 시스템은 사용자가 스크롤을 내려 정보를 탐색할 때 데이터를 동적으로 로드하여 리스트 또는 그리드에서 부드러운 스크롤을 가능하게 합니다. 이러한 접근 방식은 기기나 네트워크에 과부하를 주지 않고 원활하고 반응성이 뛰어난 인터페이스를 제공하므로 무한 스크롤 또는 대규모 데이터 소스가 요구되는 애플리케이션에 특히 유용합니다.

다음과 같이 직접적으로 페이징 시스템을 구현할 수 있습니다.

페이징 시스템 직접 구현

1. **RecyclerView.Adapter 및 ViewHolder 생성:** 첫 번째 단계는 해당 ViewHolder와 함께 RecyclerView.Adapter 또는 ListAdapter를 생성하는 것입니다. 해당 구성 요소는 데이터 셋을 RecyclerView에 관리하고 바인딩하기 위해 필수입니다. Adapter는 데이터 소스를 처리하고 ViewHolder는 데이터 셋의 아이템이 개별적으로 렌더링 되는 방식을 정의합니다.

그림 184. ListAdapter Example.kt

```
1 class PokedexAdapter : ListAdapter<Pokemon, PokedexAdapter.PokedexViewHolder>(diffUtil)
2     {
3
4         override fun onCreateViewHolder(
5             parent: ViewGroup,
6             viewType: Int
7         ): PokedexViewHolder {
8             val binding = ItemPokemonBinding.inflate(LayoutInflater.from(parent.context),
9                 parent, false) // parent 전달
10            return PokedexViewHolder(binding)
11        }
12    }
```

```
11  override fun onBindViewHolder(holder: PokedexViewHolder, position: Int) {
12      // getItem()으로 안전하게 아이템 가져오기
13      holder.bind(getItem(position))
14  }
15
16  inner class PokedexViewHolder(
17      private val binding: ItemPokemonBinding,
18  ) : RecyclerView.ViewHolder(binding.root) {
19      fun bind(pokemon: Pokemon) {
20          // 데이터 바인딩 로직
21      }
22  }
23
24  companion object {
25      private val diffUtil = object : DiffUtil.ItemCallback<Pokemon>() {
26          override fun areItemsTheSame(oldItem: Pokemon, newItem: Pokemon): Boolean =
27              oldItem.name == newItem.name
28
29          override fun areContentsTheSame(oldItem: Pokemon, newItem: Pokemon): Boolean =
30              oldItem == newItem
31      }
32  }
33 }
```

-
2. **RecyclerView**에 **addOnScrollListener** 추가: 다음으로, **RecyclerView**에 **addOnScrollListener**를 구현하여 스크롤 상태를 트래킹 합니다. 이를 통해 사용자가 현재 화면에서 마지막으로 보이는 항목까지 스크롤하는 경우를 감지할 수 있습니다. 화면에서 마지막으로 보이는 항목이 데이터 셋 끝(전체 사이즈)에 가까워지면 네트워크 또는 데이터베이스에서 다음 데이터 셋에 대한 로드를 트리거합니다. 더 부드러운 로딩을 보장하려면 사용자가 끝에 도달하기 전에 데이터를 미리 가져오기 위해 임계값(threshold)을 사용하여 스크롤의 지연이나 중단을 방지합니다.

그림 185. addOnScrollListener Example.kt

```

1 recyclerView.addOnScrollListener(object : RecyclerView.OnScrollListener() {
2     override fun onScrolled(recyclerView: RecyclerView, dx: Int, dy: Int) {
3         super.onScrolled(recyclerView, dx, dy)
4         // 레이아웃 매니저 가져오기 (LinearLayoutManager 또는 GridLayoutManager)
5         val layoutManager = recyclerView.layoutManager ?: return // Null 체크
6
7         val lastVisiblePosition = when (layoutManager) {
8             is LinearLayoutManager -> layoutManager.findLastVisibleItemPosition()
9             is GridLayoutManager -> layoutManager.findLastVisibleItemPosition()
10            else -> return // 다른 레이아웃 매니저는 처리하지 않음
11        }
12
13        val totalItemCount = layoutManager.itemCount
14        val threshold = 4 // 미리 로드할 아이템 개수 (임계값)
15
16        // 마지막 아이템에 가까워지고 로딩 중이 아닐 때 다음 페이지 미리 로드
17        if (lastVisiblePosition + threshold >= totalItemCount && totalItemCount > 0 &&
18            !viewModel.isLoading.value) {
19            viewModel.loadNextPage()
20        }
21    })

```

3. **RecyclerView.Adapter에 새 데이터 셋 추가:** viewModel.loadNextPage()가 성공적으로 트리거 되면 새로 검색된 데이터 셋을 RecyclerView.Adapter의 기존 데이터에 추가합니다. 결과적으로 리스트에 새 항목이 추가되고, 레이아웃은 이에 따라 원활하게 업데이트됩니다. (ListAdapter의 경우 submitList() 사용)

요약

데이터를 페이지로 분할하고, 스크롤 이벤트를 관찰하고, 어댑터를 동적으로 업데이트하여 직접 페이징 시스템을 만들 수 있습니다. 이를 구현하는 다양한 방법이 있지만 Jetpack

Paging 라이브러리¹⁶⁴를 활용하면 또 다른 접근 방식으로 페이징 시스템을 구현할 수 있습니다. 또는 오픈 소스 라이브러리의 [RecyclerViewPaginator](#)¹⁶⁵와 같은 커스텀 접근 방식을 살펴보는 것도 학습 자료로 유용합니다.

실전 질문

Q) 앱이 1000개의 데이터 셋을 가져와 리스트에 표시해야 한다고 가정해 봅시다. 원활한 스크롤을 보장하고 메모리 사용량을 줄이기 위해 효율적인 페이징 시스템을 어떻게 구현하시겠습니까?

Q) RecyclerView로 직접 페이징 시스템을 구현할 때 발생할 수 있는 문제는 무엇이며, 원활한 사용자 경험을 제공하기 위해 어떻게 해결할 수 있나요?

Q) 63. 네트워크에서 이미지를 어떻게 가져오고 렌더링하나요?

이미지 로딩은 사용자 프로필이나 네트워크에서 가져온 콘텐츠를 표시하는 작업과 같이 최신 애플리케이션 개발에 있어서 흔히 요구되는 기능입니다. 개발자가 직접 이미지 로딩 시스템을 만드는 것도 가능하지만, 네트워크 요청, 이미지 크기 조절, 캐싱, 렌더링 및 효율적인 메모리 관리와 같은 복잡한 기능을 구현해야 합니다.

직접 이미지 로딩 솔루션을 구축하는 대신, 이미 대규모 프로젝트에서 사용되어 안정성이 입증되었으며, 풍부한 API를 제공하는 [Glide](#), [Coil](#), [Fresco](#)와 같은 라이브러리를 활용할 수 있습니다. 해당 라이브러리들은 이미지 다운로드, 캐싱 및 렌더링과 같은 작업을 원활하게 처리하여 개발자가 애플리케이션의 다른 중요한 부분에 집중할 수 있도록 합니다.

Glide

[Glide](#)¹⁶⁶는 안드로이드 개발 생태계에서 오랫동안 인기를 끌어 온 빠르고 효율적인 이미지 로딩 라이브러리입니다. 캐싱, 플레이스홀더 이미지 및 이미지 변환과 같은 복잡한 시나리오를 처리하는 데 이상적입니다. Google의 공식 오픈 소스 프로젝트를 포함하여 많은 글로벌 제품 및 오픈 소스 프로젝트에서 사용되고 있습니다.

¹⁶⁴<https://developer.android.com/topic/libraries/architecture/paging/v3-overview>

¹⁶⁵<https://github.com/skydoves/BaseRecyclerViewAdapter?tab=readme-ov-file#recyclerviewpaginator>

¹⁶⁶<https://github.com/bumptech/glide>

그림 186. Glide Example.kt

```
1 Glide.with(context)
2     .load("https://example.com/image.jpg") // 이미지 URL
3     .placeholder(R.drawable.placeholder) // 로딩 중 표시할 이미지
4     .error(R.drawable.error_image) // 오류 시 표시할 이미지
5     .circleCrop() // 원형 자르기 변환 (선택 사항)
6     .transition(DrawableTransitionOptions.withCrossFade()) // 페이드 효과 (선택 사항)
7     .into(imageView) // 대상 ImageView
```

Glide는 자동으로 이미지를 캐시하여 네트워크 호출을 최적화하고 성능을 향상시킵니다. 애니메이션 GIF 지원, [플레이스홀더](#)¹⁶⁷, [변환](#)¹⁶⁸, [캐싱](#)¹⁶⁹, [리소스 재사용](#)¹⁷⁰과 같은 유용한 기능을 제공합니다.

Coil

[Coil](#)¹⁷¹은 Kotlin Multiplatform으로 설계된 100% Kotlin 기반의 이미지 로딩 라이브러리입니다. 내부적으로 [Coroutines](#)를 활용하고 Jetpack Compose와 같은 최신 기능을 지원합니다. 특히 Coil은 내부적으로 [OkHttp](#)¹⁷² 및 [Coroutines](#)¹⁷³와 같이 안드로이드 프로젝트에서 이미 널리 사용되는 라이브러리를 사용하기 때문에 기존 프로젝트에 원활하게 통합될 수 있습니다.

¹⁶⁷<https://bumptech.github.io/glide/doc/placeholders.html>

¹⁶⁸<https://bumptech.github.io/glide/doc/transformations.html>

¹⁶⁹<https://bumptech.github.io/glide/doc/caching.html>

¹⁷⁰<https://bumptech.github.io/glide/doc/resourcereuse.html>

¹⁷¹<https://github.comcoil-ktcoil>

¹⁷²<https://square.github.iookhttp/>

¹⁷³<https://kotlinlang.orgdocs/coroutines-overview.html>

그림 187. Coil Example.kt

```
1 import coil.load
2 import coil.transform.CircleCropTransformation
3
4 imageView.load("https://example.com/image.jpg") {
5     crossfade(true) // 크로스페이드 효과
6     placeholder(R.drawable.placeholder)
7     error(R.drawable.error_image)
8     transformations(CircleCropTransformation()) // 원형 자르기 변환
9     // size(Size.ORIGINAL) // 원본 크기 로드 (선택 사항)
10    // memoryCachePolicy(CachePolicy.ENABLED) // 메모리 캐시 정책 설정
11    // diskCachePolicy(CachePolicy.ENABLED) // 디스크 캐시 정책 설정
12 }
```

Coil은 Kotlin 및 Jetpack Compose와 원활하게 통합되며, 이미지 변환¹⁷⁴, 애니메이션 GIF 지원¹⁷⁵, SVG 렌더링¹⁷⁶, 비디오 프레임 추출¹⁷⁷과 같은 유용한 기능을 제공합니다. 가벼운 특성 덕분에 최신 안드로이드 프로젝트에 훌륭한 선택입니다.

이미지 로딩 라이브러리 중에서 Coil은 현재 가장 활발하게 관리되고 있으며 Jetpack Compose, Kotlin Multiplatform 및 기타 최신 안드로이드 솔루션에 대한 다양한 지원을 제공합니다. Kotlin 우선 접근 방식과 지속적인 유지보수 덕분에 오늘날 개발자 커뮤니티에서 가장 선호되는 솔루션이 되었습니다.

Fresco

Fresco¹⁷⁸는 Meta¹⁷⁹에서 개발한 이미지 로딩 라이브러리로, 조금 더 복잡한 사용 사례를 위해 설계되었습니다. 이미지를 디코딩하고 표시하기 위해 자체 파이프라인을 사용하는 접근 방식을 사용합니다.

¹⁷⁴<https://coil-kt.github.io/coil/transformations/>

¹⁷⁵<https://coil-kt.github.io/coil/gifs/>

¹⁷⁶<https://coil-kt.github.io/coil/svg/>

¹⁷⁷<https://coil-kt.github.io/coil/videos/>

¹⁷⁸<https://github.com/facebook/fresco>

¹⁷⁹<https://github.com/facebook>

그림 188. Fresco Example.kt

```
1 // Fresco 초기화 (Application 클래스에서)
2 // Fresco.initialize(this);
3
4 // 레이아웃에서 SimpleDraweeView 사용
5 val draweeView: SimpleDraweeView = findViewById(R.id.drawee_view)
6 val uri = Uri.parse("https://example.com/image.jpg")
7 draweeView.setImageURI(uri)
8
9 // 또는 Controller를 사용하여 더 세밀하게 제어
10 // val controller = Fresco.newDraweeControllerBuilder()
11 //     .setUri(uri)
12 //     .setOldController(draweeView.controller)
13 //     .build()
14 // draweeView.controller = controller
```

Fresco는 자체적으로 SimpleDraweeView라는 커스텀 View를 제공합니다.

```
1 <com.facebook.drawee.view.SimpleDraweeView
2     xmlns:fresco="http://schemas.android.com/apk/res-auto" <!-- 네임스페이스 추가 -->
3     android:id="@+id/drawee_view"
4     android:layout_width="wrap_content"
5     android:layout_height="wrap_content"
6     fresco:placeholderImage="@drawable/placeholder" <!-- 플레이스홀더 설정 -->
7     fresco:failureImage="@drawable/error_image" <!-- 오류 이미지 설정 -->
8     fresco:progressBarImage="@drawable/loading_spinner" <!-- 로딩 스피너 설정 -->
9 />
```

Fresco는 큰 이미지 처리, 점진적 렌더링 및 고급 캐싱 전략에 매우 효율적이어서 메모리 제약이 있는 애플리케이션에 특히 유용합니다. 안드로이드 4.x 이하에서 이미지를 특수 메모리 영역에 할당하여 성능을 비약적으로 높였지만, 사실 4.x 이하의 디바이스는 현대 사회에서 거의 찾아보기 힘들 정도로 사용률이 감소했기 때문에 해당 이점이 거의 사라졌습니다.

그래도 Fresco는 이미지 파이프라인, 드로위(Drawees), 최적화된 메모리 관리, 고급 로딩 메커니즘, 스트리밍 및 애니메이션과 같은 괜찮은 기능을 제공합니다. 프로젝트에 복잡한 이미지 처리 시나리오가 포함된 경우 여전히 Fresco를 활용하는 사례(Twitter 안드로이드 앱 등)가 있습니다. 자세한 내용은 [공식 문서](#)¹⁸⁰를 확인하세요.

요약

각 라이브러리는 고유한 이점을 제공합니다.

- **Glide**: 유연하고 네트워크 이미지를 쉽게 처리하는 데 널리 사용됩니다. Jetpack Compose를 지원하지만 몇 년 동안 베타 상태로 남아 있습니다. 그 이유는 다음 챕터인 Jetpack Compose Interview Question에서 자세히 다를 예정입니다.
- **Coil**: Kotlin 중심적이고 가벼우며 최신 안드로이드 프로젝트에 원활하게 통합됩니다. Compose 및 Kotlin Multiplatform을 지원합니다.
- **Fresco**: 메모리 집약적인 시나리오에 적합하며 점진적 이미지 로딩, 이미지 파이프라인 및 더 복잡한 작업과 같은 고급 기능을 처리하는데 효율적입니다.

프로젝트 요구 사항에 따라 라이브러리를 선택하되, 원활한 사용자 경험을 제공하기 위해 항상 적절한 캐싱, 오류 처리 및 리소스 관리를 보장해야 합니다.

실전 질문

Q) 앱이 백엔드 서버에서 고해상도 이미지를 로드하여 RecyclerView에서 부드러운 스크롤을 제공해야 합니다. 어떤 이미지 로딩 라이브러리를 선택할 것이고, UI 지연을 방지하기 위해 성능을 어떻게 최적화하시겠습니까?

Q) 64. 로컬 디바이스에 데이터를 저장하고 복원하는 방법에 대해서 설명해 주세요.

안드로이드는 경량화된 키-값 기반의 데이터 저장, 구조화된 데이터베이스 구축 및 쿼리 또는 로컬 파일 처리 등 각 시나리오에 적합한 데이터 저장 메커니즘을 제공합니다.

¹⁸⁰<https://frescolib.org/>

SharedPreferences

`SharedPreferences`¹⁸¹는 앱 내 설정이나 사용자 환경 설정과 같은 가벼운 값 가장 적합한 키-값 쌍 형태의 데이터 저장 메커니즘입니다. Boolean, Int, String, Float와 같은 원시타입 (primitive 타입)을 저장하고 앱 재시작 시에도 유지할 수 있습니다. `SharedPreferences`는 동기적으로 작동하여 메인 스레드를 차단하는 문제가 있고, 최근 비동기 처리 함수 등을 제공하는 `DataStore`의 등장으로 인해 최신 애플리케이션에서는 덜 선호되고 있습니다.

그림 189. `SharedPreferences Example.kt`

```

1 import androidx.core.content.edit // KTX 확장 함수 사용
2
3 val sharedpreferences = context.getSharedPreferences("app_prefs", Context.MODE_PRIVATE)
4 // KTX 확장 함수를 사용하여 간단하게 데이터 수정 후 반영
5 sharedpreferences.edit {
6     putString("user_name", "skydoves")
7    .putInt("user_score", 100)
8     // apply()는 비동기, commit()은 동기
9     // KTX edit {} 블록은 자동으로 apply() 호출
10 }
11
12 // 값 읽기
13 val userName = sharedpreferences.getString("user_name", null) // 기본값 지정

```

DataStore

`Jetpack DataStore`¹⁸²는 `SharedPreferences`를 대체하는 더 모던하고 효율적인 방법입니다. 키-값 저장을 위한 `PreferencesDataStore`와 구조화된 객체 값 데이터를 위한 `ProtoDataStore`의 두 가지 유형을 제공합니다. `SharedPreferences`와 달리 `DataStore`는 비동기식이므로 메인 스레드를 차단하는 잠재적인 문제를 방지합니다.

¹⁸¹<https://developer.android.com/training/data-storage/shared-preferences>

¹⁸²<https://developer.android.com/topic/libraries/architecture/datastore>

그림 190. DataStore Example.kt

```
1 import androidx.datastore.preferences.core.edit
2 import androidx.datastore.preferences.core.stringPreferencesKey
3 import androidx.datastore.preferences.preferencesDataStore
4 import kotlinx.coroutines.flow.first
5 import kotlinx.coroutines.flow.map
6 import kotlinx.coroutines.runBlocking
7
8 // Context의 확장 함수로 DataStore 인스턴스 생성 (싱글톤 권장)
9 val Context.dataStore: DataStore<Preferences> by preferencesDataStore(name =
    "settings")
10
11 // 데이터 저장을 위한 키 정의
12 val USER_NAME_KEY = stringPreferencesKey("user_name")
13
14 // 데이터 쓰기 (코루틴 내에서 수행)
15 suspend fun saveUserName(context: Context, name: String) {
16     context.dataStore.edit { settings ->
17         settings[USER_NAME_KEY] = name
18     }
19 }
20
21 // 데이터 읽기 (Flow 사용)
22 fun getUserIdFlow(context: Context): Flow<String?> {
23     return context.dataStore.data
24         .map { preferences ->
25             preferences[USER_NAME_KEY] // 키에 해당하는 값 반환 (없으면 null)
26         }
27 }
28
29 // 값 저장 및 복원 예시
30 fun exampleUsage(context: Context) {
31     viewModelScope.launch {
32         saveUserName(context, "John Doe")
```

```

33     val name = getUserNameFlow(context).first() // Flow에서 첫 번째 값 가져오기
34     Log.d("DataStore", "User name: $name")
35 }
36 }
```

Room Database

Room Database¹⁸³는 구조화되고 관계형 데이터를 처리하도록 설계된 SQLite를 수준 높게 추상화한 솔루션입니다. 어노테이션, 컴파일 타임 검사, 반응형 프로그래밍을 위한 LiveData 또는 Flow 지원을 통해 데이터베이스 관리를 매우 단순화합니다. 개발자가 데이터베이스 쿼리를 위해서 쿼리문을 공부해야 할 필요가 전혀 없습니다. Room은 복잡한 쿼리나 대량의 구조화된 데이터 저장이 필요한 앱에 이상적입니다.

그림 191. Room Database Example.kt

```

1 // 데이터 엔티티 정의
2 @Entity(tableName = "users") // 테이블 이름 지정 (선택 사항)
3 data class User(
4     @PrimaryKey val id: Int,
5     @ColumnInfo(name = "user_name") val name: String // 컬럼 이름 지정 (선택 사항)
6 )
7
8 // 데이터 접근 객체(DAO) 정의
9 @Dao
10 interface UserDao {
11     // suspend 함수로 비동기 처리
12     @Insert(onConflict = OnConflictStrategy.REPLACE) // 크래시 발생 시 대체 전략
13     suspend fun insertUser(user: User)
14
15     @Query("SELECT * FROM users WHERE id = :userId") // 명확한 파라미터 이름 사용
16     suspend fun getUserId(userId: Int): User? // Nullable 반환 타입
17
18     @Query("SELECT * FROM users ORDER BY user_name ASC")
```

¹⁸³<https://developer.android.com/training/data-storage/room>

```
19     fun getAllUsers(): Flow<List<User>> // Flow를 사용하여 데이터 변경 관찰
20 }
21
22 // 데이터베이스 클래스 정의
23 @Database(entities = [User::class], version = 1, exportSchema = false) // 스키마
→ 내보내기 비활성화 (선택 사항)
24 abstract class AppDatabase : RoomDatabase() {
25     abstract fun userDao(): UserDao
26
27     // 싱글톤 인스턴스 제공 (권장)
28     companion object {
29         @Volatile
30         private var INSTANCE: AppDatabase? = null
31
32         fun getInstance(context: Context): AppDatabase {
33             return INSTANCE ?: synchronized(this) {
34                 val instance = Room.databaseBuilder(
35                     context.applicationContext,
36                     AppDatabase::class.java,
37                     "app_database" // 데이터베이스 파일 이름
38                 )
39                 // .addMigrations(MIGRATION_1_2) // 마이그레이션 추가 (필요시)
40                 .build()
41                 INSTANCE = instance
42                 instance
43             }
44         }
45     }
46 }
```

File Storage

바이너리 또는 커스텀 데이터의 경우 안드로이드는 내부 또는 외부 저장소에 파일을 저장할 수 있도록 합니다. 내부 저장소는 앱 간 접근할 수 없으며, 외부 저장소는 다른 앱과 공유할

수 있습니다. 파일 I/O 작업은 이미지, 비디오 또는 커스텀 직렬화된 데이터 저장과 같은 작업에 사용할 수 있습니다.

그림 192. File Storage Example.kt

```
1 // 내부 저장소의 앱별 디렉토리에 파일 생성
2 val file = File(context.filesDir, "user_data.txt")
3 try {
4     FileOutputStream(file).use { fos ->
5         fos.write("Sample user data".toByteArray())
6     }
7     // 또는 BufferedWriter 사용
8     // file.bufferedWriter().use { out ->
9     //     out.write("Sample user data")
10    // }
11 } catch (e: IOException) {
12     e.printStackTrace()
13     // 오류 처리
14 }
15
16 // 파일 읽기 (예시)
17 try {
18     val content = file.readText()
19     Log.d("FileStorage", "File content: $content")
20 } catch (e: IOException) {
21     e.printStackTrace()
22 }
```

요약

안드로이드의 저장 메커니즘 선택은 데이터의 유형과 복잡성에 따라 달라집니다. **SharedPreferences** 또는 **DataStore**는 사용자 설정이나 기능 플래그와 같은 가벼운 키-값 데이터를 저장하는 데 이상적입니다. **Room**은 구조화된 쿼리를 사용하여 복잡한 관계형 데이터를 관리하는 데 적합하며, **File Storage**는 바이너리 파일이나 대규모 커스텀 데이터

셋을 처리하는 데 가장 적합합니다. 각 방법은 애플리케이션 요구 사항에 따라 효율적이고 신뢰할 수 있는 데이터 저장을 보장 위해 각자만의 이점을 제공합니다.

실전 질문

Q) 오프라인 접근을 위해 네트워크 API에서 받은 대용량 JSON 응답을 저장해야 하는 시나리오에서 어떤 로컬 저장 메커니즘을 사용할 것이고, 그 이유는 무엇인가요?

Q) 65. 오프라인 우선(**offline-first**) 아키텍처를 어떻게 설계하실 건가요?

오프라인 우선 디자인은 애플리케이션이 로컬에 캐시되거나 저장된 데이터에 의존하여 활성 네트워크 연결 없이도 앱이 원활하게 작동하도록 보장합니다. 이러한 접근 방식은 특히 인터넷 연결이 좋지 않거나 간헐적인으로 인터넷이 끊기는 시나리오에서 사용자 경험을 향상 시킵니다. 데이터를 로컬에 캐시하거나 저장하고 연결이 복원되면 원격 서버와 동기화하여 매끄러운 경험을 제공합니다. 오프라인 우선 앱 설계에 대한 모범 사례는 [오프라인 우선에 대한 안드로이드 공식 문서](#)¹⁸⁴에서 제공합니다.

오프라인 우선 아키텍처의 핵심 개념

1. **로컬 데이터 지속성(Local Data Persistence)**: 신뢰할 수 있는 오프라인 우선 전략은 로컬 데이터 저장소에서 시작됩니다. Jetpack의 일부인 **Room Database**는 구조화된 로컬 데이터를 관리하기 위한 권장되는 솔루션입니다. 앱이 오프라인 상태에서도 데이터에 접근하고 업데이트할 수 있도록 보장하기 때문입니다. Room은 Kotlin 코루틴, Flow, LiveData와 원활하게 작동하여 UI에 반응형 업데이트를 제공합니다.
2. **데이터 동기화(Data Synchronization)**: 로컬 데이터와 원격 데이터 간의 동기화는 일관성을 보장합니다. **WorkManager**는 이를 위한 훌륭한 선택 중 하나로, 네트워크 연결과 같은 조건이 충족될 때 지연된 동기화 작업이 실행되도록 합니다. WorkManager는 실패한 작업을 자동으로 재시도하여 데이터 무결성을 보장합니다.
3. **캐싱 및 가져오기 정책(Cache and Fetch Policies)**: 아래와 같이 데이터 캐싱(쓰기) 및 가져오기(읽기)에 대한 명확한 정책을 정의해야 합니다.
 - **캐싱 데이터 읽기**: 앱이 먼저 로컬 저장소에서 데이터를 가져오고 필요할 때만 네트워크에 새로운 데이터를 요청합니다.

¹⁸⁴<https://developer.android.com/topic/architecture/data-layer/offline-first>

- **캐싱 데이터 쓰기**: 업데이트가 로컬에 기록되고 백그라운드에서 서버와 동기화됩니다.

4. **충돌 해결(Conflict Resolution)**: 로컬 소스와 원격 소스 간에 데이터를 동기화할 때 충돌 해결 전략을 구현해야 합니다.

- **최신 데이터 우선**: 가장 최근 변경 사항을 우선시합니다.
- **사용자 정의**: 사용자가 수동으로 충돌을 해결하거나 도메인별 규칙을 적용하도록 허용합니다.

실제 구현

아래는 Room과 WorkManager를 사용하여 오프라인 우선 기능을 구현하는 예시입니다.

그림 193. OfflineFirstFeature.kt

```
1 // 데이터 엔티티 (동기화 상태 플래그 포함)
2 @Entity
3 data class Article(
4     @PrimaryKey val id: Int,
5     val title: String,
6     val content: String,
7     val isSynced: Boolean = false // 서버와 동기화되었는지 여부
8 )
9
10 // DAO 인터페이스
11 @Dao
12 interface ArticleDao {
13     @Query("SELECT * FROM Article")
14     fun getAllArticles(): Flow<List<Article>> // Flow로 변경 사항 관찰
15
16     // 로컬에만 있는 미동기화된 Article 조회
17     @Query("SELECT * FROM Article WHERE isSynced = 0")
18     suspend fun getUnsyncedArticles(): List<Article>
19 }
```

```
20     @Insert(onConflict = OnConflictStrategy.REPLACE)
21     suspend fun insertArticle(article: Article)
22
23     // 동기화 완료 후 상태 업데이트
24     @Query("UPDATE Article SET isSynced = 1 WHERE id = :articleId")
25     suspend fun markArticleAsSynced(articleId: Int)
26 }
27
28 // 동기화 Worker
29 class SyncWorker(appContext: Context, params: WorkerParameters) :
30     CoroutineWorker(appContext, params) {
31     override suspend fun doWork(): Result {
32         // 데이터베이스 및 DAO 인스턴스 가져오기 (의존성 주입 권장)
33         val articleDao = AppDatabase.getInstance(applicationContext).articleDao()
34
35         return try {
36             // 미동기화된 데이터 가져오기
37             val unsyncedArticles = articleDao.getUnsyncedArticles()
38
39             if (unsyncedArticles.isNotEmpty()) {
40                 // 서버와 동기화 시도
41                 if (syncToServer(unsyncedArticles)) {
42                     // 성공 시 로컬 상태 업데이트
43                     unsyncedArticles.forEach {
44                         articleDao.markArticleAsSynced(it.id)
45                     }
46                     Log.d("SyncWorker", "Sync successful for ${unsyncedArticles.size}")
47                     → articles.)
48                 } else {
49                     Log.e("SyncWorker", "Sync failed.")
50                     // 재시도 로직은 WorkManager가 처리하거나 여기서 명시적으로
51                     → Result.retry() 반환 가능
52                     return Result.retry()
53                 }
54             }
55         } catch (e: Exception) {
56             Log.e("SyncWorker", "Sync failed due to ${e.message}")
57             return Result.failure()
58         }
59     }
60 }
```

```

51         } else {
52             Log.d("SyncWorker", "No articles to sync.")
53         }
54         Result.success()
55     } catch (e: Exception) {
56         Log.e("SyncWorker", "Sync failed with error", e)
57         Result.failure() // 실패로 간주
58     }
59 }
60
61 // 서버와 동기화하는 실제 로직 (네트워크 호출 등)
62 private suspend fun syncToServer(articles: List<Article>): Boolean {
63     // 실제 네트워크 요청 및 결과 처리 로직 구현
64     Log.d("SyncWorker", "Attempting to sync ${articles.size} articles...")
65     // 아래 예시에서는 가짜로 자연시키고 성공을 반환하고 있는데,
66     // 실제 구현에서는 네트워크 요청 로직이 들어감
67     kotlinx.coroutines.delay(2000)
68     return true // 실제 구현에서는 API 호출 결과에 따라 반환
69 }
70 }
```

이제 동기화 전략에 따라 SyncWorker를 실행할 수 있습니다. 예를 들어, 모든 타임라인 데이터를 동기화해야 하는 경우 Jetpack의 App Startup을 활용하여 사용자가 앱을 시작할 때 SyncWorker를 한 번만 트리거할 수 있습니다. 실제 구현 예시는 GitHub의 [SyncWorker.kt¹⁸⁵](#) 및 [SyncInitializer.kt¹⁸⁶](#)를 참조할 수 있습니다.

요약

1. 백그라운드 동기화 관리를 위해 **WorkManager**을 사용합니다.

¹⁸⁵<https://github.com/android/nowinandroid/blob/d42262c9391ccd1d59a0c92476c2b349a5acc3af.sync/work/src/main/kotlin/com/google/samples/apps/nowinandroid/sync/workers/SyncWorker.kt#L51>

¹⁸⁶<https://github.com/android/nowinandroid/blob/d42262c9391ccd1d59a0c92476c2b349a5acc3af.sync/work/src/main/kotlin/com/google/samples/apps/nowinandroid/sync/initializers/SyncInitializer.kt#L23>

2. 로컬 데이터 저장을 위해 **Room**을 활용합니다.
3. 효율적인 데이터 업데이트를 위해 명확한 캐시 정책을 정의합니다.
4. 데이터 일관성을 보장하기 위해 충돌 해결 메커니즘을 구현합니다.

요약

안드로이드의 오프라인 우선 접근 방식은 사용자가 인터넷 연결 상태에 관계없이 원활한 기능을 보장할 수 있도록 합니다. Room, WorkManager 및 적절한 캐싱 전략과 같은 도구를 활용하여 일관된 사용자 경험을 유지할 수 있습니다. 오프라인 우선 기능을 효과적으로 구현하기 위한 가이드는 [공식 문서¹⁸⁷](#)를 통해서 더 학습하실 수 있습니다.

실전 질문

- Q) 네트워크가 자주 끊기는 국가 및 지역에서도 원활한 사용자 경험을 보장하기 위해 오프라인 우선 기능을 어떻게 설계하시겠습니까?
- Q) 로컬 Room 데이터베이스 변경 사항을 백엔드 서버의 최신 데이터와 동기화하는 데 어떤 전략을 사용할 것이고, 로컬 및 백엔드 데이터가 모두 변경되었을 때 충돌을 어떻게 해결하시겠습니까?

Q) 66. 초기 데이터 로딩을 위한 작업을 Compose의 LaunchedEffect와 ViewModel.init() 중 어디에서 하는 것이 가장 이상적인가요?

 **Note:** 이 질문은 Kotlin Coroutines 및 Jetpack Compose와 관련된 기본 개념을 포함합니다. 해당 주제에 익숙하지 않다면 먼저 [Kotlin](#) 및 [Chapter 1: Jetpack Compose Interview Questions](#)를 학습하시고, 나중에 이 질문을 다시 읽어보시면 더 명확하게 이해하시는 데 도움이 될 것입니다.

안드로이드 개발에서 자주 논의되는 주제는 초기 데이터를 Composable의 LaunchedEffect에서 로드할지 아니면 ViewModel의 init() 블록 내에서 로드할지 여부입니다. 공식 [안드로이드](#)

¹⁸⁷<https://developer.android.com/topic/architecture/data-layer/offline-first>

문서¹⁸⁸ 및 architecture-samples GitHub¹⁸⁹의 예제는 일반적으로 구성 변경 시 더 나은 생명주기 관리 및 데이터 지속성을 위해 ViewModel.init() 내에서 데이터를 로드할 것을 예시로 보여주고 있습니다. 이 책의 저자가 일반적으로 초기 데이터를 로드하기 위해 안드로이드 커뮤니티에서는 어떤 방식을 선호하는지 아래와 같은 설문조사를 시행한 적이 있습니다.

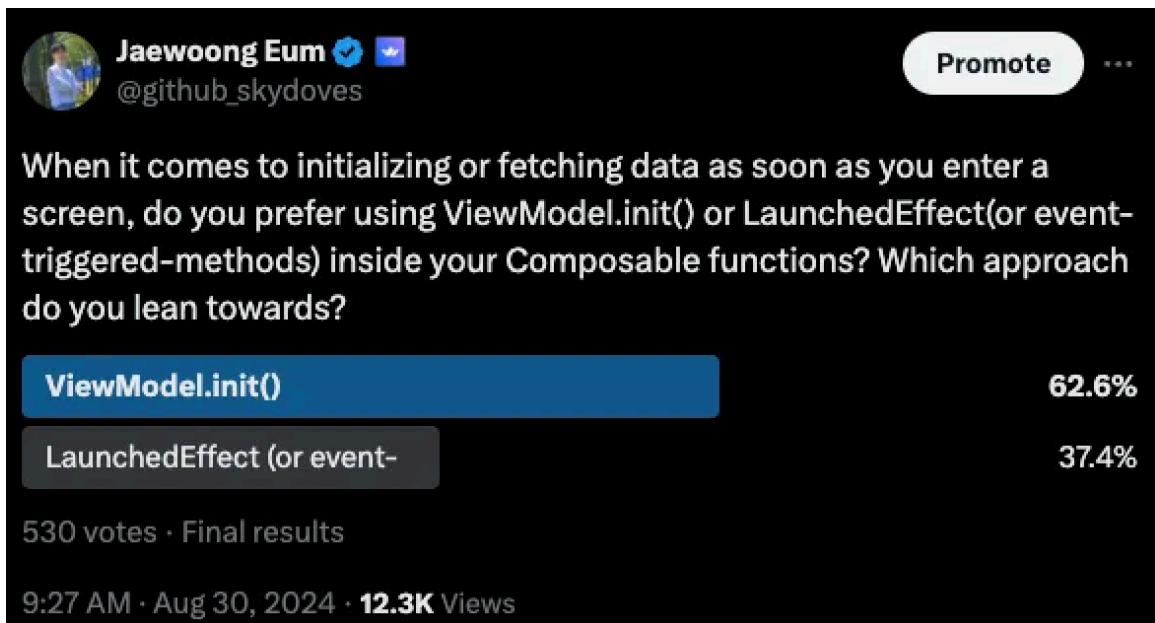


그림 194. initial-data-poll

결과는 보시는 바와 같이 대부분의 개발자는 ViewModel.init() 내에서 초기 데이터를 로드하는 것을 선호합니다. 특히 어떤 개발자는 생명주기 안정성과 상태 관리를 강조하면서 LaunchedEffect와 같은 Composable 함수를 사용하는 것보다 ViewModel.init()가 더 나은 선택인 이유를 아래와 같이 설명했습니다.

안드로이드 개발자 A: “Jetpack Compose UI를 애플리케이션의 상태 또는 데이터의 시작적 표현으로 간주한다면, 앱에 무엇을 해야 할지 지시하기 위해 UI에 의존한다는 것은

¹⁸⁸<https://developer.android.com/kotlin/flow/stateflow-and-sharedflow#stateflow>

¹⁸⁹<https://github.com/android/architecture-samples/blob/130f5dbebd0c7b5ba195cc08f25802ed9f0237e5/app/src/main/java/com/example/android/architecture/blueprints/todoapp/addeeditTask/AddEditTaskViewModel.kt#L64>

사실 설계 결함으로 볼 수 있습니다. 이러한 관점에서 `ViewModel.init()`를 사용하는 것이 `LaunchedEffect` 내에서 직접 데이터를 가져오는 것보다 더 나은 관심사 분리를 제공하여 비즈니스 로직과 UI 상태 관리가 구분되도록 보장합니다.”

반대로 다른 개발자는 아래와 같은 관점을 공유했습니다.

안드로이드 개발자 B: “`ViewModel.init()`에만 의존하면 특정 동작이 트리거 되는 시점에 대한 제어가 어려워지고 유닛 테스트가 복잡해질 수 있다는 것을 명심해야 합니다. 대신 `ViewModel` 내에서 이벤트 기반 흐름을 관찰하여 트리거되고 자연 초기화될 수 있는 독립적인 함수를 정의하는 것을 선호합니다. 해당 접근 방식은 더 큰 유연성과 제어권을 제공하여 `LaunchedEffect` 또는 이벤트 트리거 작업과 같은 메서드가 데이터 로딩을 더 효과적으로 관리할 수 있도록 합니다. 추가적인 이점은 `ViewModel.init` 블록에 전적으로 의존하는 대신 사용자 상호 작용이나 특정 이벤트를 기반으로 데이터를 다시 가져올 때 효율성이 향상된다는 것입니다. 이는 UI가 비즈니스 로직과 동적이고 지속적인 작업이 요구될 때 특히 유용해집니다.”

둘 다 안티패턴. 자연 관찰(Lazy Observation) 사용하기

두 솔루션 모두 주목할 만한 단점이 있습니다. 흥미롭게도 Google의 Android Toolkit 팀의 Ian Lake는 두 접근 방식 모두 안드로이드 개발에서 안티패턴으로 간주된다고 지적하며 초기 데이터 로딩 관리에 다른 대안이 필요함을 시사했습니다.

Ian Lake
@ianhlake

Both are anti-patterns: object creation shouldn't have side-effects and `LaunchedEffect` is going to re-run on each first composition of screen.

Instead, prefer cold flows that kick off their work when you start collecting them via `collectAsStateWithLifecycle()`.

12:59 PM · Aug 30, 2024 · 4,194 Views

그림 195. initial-data-comment

`ViewModel.init()`에서 초기 데이터를 로드하면 `ViewModel` 생성 중에 의도하지 않은 사이드 이펙트가 발생하고, UI 상태 관리라는 의도된 역할에서 벗어나 생명주기 처리를 복잡하게 만들 수 있습니다.

마찬가지로 Jetpack Compose의 `LaunchedEffect` 내에서 데이터를 초기화하면 매 첫 컴포지션마다 반복적으로 트리거 될 위험이 있습니다. 이는 `ViewModel`의 생명주기가 일반적으로 `Composable` 함수의 생명주기 보다 길기 때문에, `Composable` 함수가 새로 컴포지션에 진입할 때마다 동일한 `ViewModel` 인스턴스에 대해서 동일한 비즈니스 로직을 반복적으로 트리거할 가능성이 있습니다. 따라서, 생명주기 불일치로 예기치 않은 동작이 발생하고 의도된 데이터 흐름을 방해할 수 있습니다.

이러한 우려를 해결하기 위해 Ian Lake는 자연 초기화를 위해 **cold flows**¹⁹⁰ 사용을 권장합니다. 이 접근 방식에서 `Flow`는 수집되기 시작할 때만 네트워크 요청이나 데이터베이스 쿼리와 같은 비즈니스 로직을 실행합니다. UI 레이어에서 구독자가 있을 때까지 `Flow`는 비활성 상태로 유지되어 불필요한 작업이 수행되지 않도록 보장합니다.

자연 관찰 모범 사례

해당 접근 방식의 예시를 GitHub의 `Pokedex-Compose`¹⁹¹ 프로젝트에서 일부 가져와 아래에서 보여주고 있습니다.

그림 196. Lazy Observation Example.kt

```

1 val pokemon: StateFlow<Pokemon?> = savedStateHandle.getStateFlow("pokemon", null)
2
3 // pokemon Flow가 변경될 때마다 flatMapLatest를 사용하여 새로운 pokemonInfo Flow 생성
4 val pokemonInfo: StateFlow<PokemonInfo?> = pokemon.filterNotNull().flatMapLatest {
5     → pokemon ->
6         // 실제 데이터 로딩 로직 (네트워크 요청 등)
7         detailsRepository.fetchPokemonInfo(
8             name = pokemon.name,
9             onComplete = { isLoading = false }, // 로딩 상태 업데이트
10            onError = { errorMessage = it } // 오류 메시지 업데이트

```

¹⁹⁰콜드 플로우(**cold flow**)는 수집될 때까지 비활성 상태로 유지되는 플로우입니다. 실행 로직은 수집기(collector)가 구독할 때만 시작되어 활성 구독자가 없는 경우 불필요한 작업이 수행되지 않도록 보장합니다.

¹⁹¹<https://github.com/skydoves/pokedex-compose>

```
10      )
11 } .stateIn(
12     scope = viewModelScope, // ViewModel 스코프 사용
13     // 구독자가 있을 때만 Flow 구독 시작, 구독자가 없다면 5초 후 데이터 발행 중지
14     started = SharingStarted.WhileSubscribed(5_000),
15     initialValue = null // 초기값 설정
16 )
```

위 코드에서 업스트림 호출(`detailsRepository.fetchPokemonInfo()`)은 첫 번째 구독자가 Flow 수집을 시작할 때만 트리거 됩니다. 그런 다음 결과는 캐시 되어 `stateIn` 메서드를 사용하여 상태로 변환되어 효율적인 데이터 관리와 중복 작업 최소화를 보장합니다. `stateIn` 메서드는 콜드 플로우를 핫 플로우인 `StateFlow`로 변환하며, 일부 상태 값에 대한 업데이트를 제공하고, 생성 또는 유지 관리가 비싼 편이지만 가장 최근 값을 수집해야 하고 여러 구독자가 생길 수 있는 상황에서 유용합니다.

궁극적으로 `pokemonInfo` 속성이 핫 플로우¹⁹²로 정의되더라도 가장 최근에 방출된 값은 업스트림 플로우의 단일 실행 인스턴스에서 나옵니다. 해당 인스턴스는 여러 다운스트림 구독자 간에 공유되며 지역 초기화되어 효율적인 데이터 관리와 중복 실행을 방지합니다.

Manifest Kotlin Interview for Android Developers¹⁹³ 책에서 Coroutines 및 Flow에 대해 더 자세히 학습하실 수 있습니다. 지금 당장 모든 개념을 이해하기 어렵다면 **Chapter 1: Jetpack Compose Interview Questions**을 학습한 후 이 카테고리를 다시 읽어보시면 더 명확하게 이해하실 수 있을 것입니다.

요약

안드로이드에서 초기 데이터를 로드하는 방법에는 Jetpack Compose의 `LaunchedEffect` 사용하거나, `ViewModel`의 `init()` 메서드 사용, 콜드 플로우를 통한 지역 관찰 등 여러 가지가 있습니다. 이 책의 저자가 커뮤니티에서 시작한 해당 디스커션은 효율성을 높이고 부작용을 피하기 위해 콜드 플로우 활용을 제안하는 것으로 끝났습니다. 그러나 늘 그렇듯 모든 상황에서 완벽한 해결책이라는 것은 없습니다. 각 프로젝트에는 고유한 요구 사항이 있습니다.

¹⁹²핫 플로우(**hot flow**)는 활성 수집기가 있는지 여부에 관계없이 값을 적극적으로 방출하는 플로우입니다. 상태를 유지하고 데이터를 계속 생성하므로 `StateFlow` 및 `SharedFlow`와 같이 공유되고 지속적으로 실행되는 데이터 스트림에 적합합니다.

¹⁹³<https://www.kotlin.skydoves.me/>

앱의 특정 요구 사항을 이해하는 것은 가장 적합한 접근 방식을 선택하는 데 중요합니다. 해당 디스커션은 애플리케이션 컨텍스트에 효과적으로 적용할 수 있는 실용적인 전략을 논하였습니다. 관련하여 더 학습하고 싶으시다면 [Loading Initial Data in LaunchedEffect vs. ViewModel](#)¹⁹⁴ 포스트에서 자세히 읽어볼 수 있습니다.

실전 질문

Q) ViewModel.init() 또는 Jetpack Compose의 LaunchedEffect에서 초기 데이터를 로드하는 것의 장단점은 무엇이며, 언제 어떤 접근 방식을 사용하시나요? 만약 다른 접근법을 선호한다면 어떤 것이 있나요?

Q) ViewModel.init() 또는 LaunchedEffect와 비교할 때 콜드 플로우를 사용한 자연 관찰은 초기 데이터를 로드할 때 효율성을 어떻게 개선하나요? 해당 접근 방식이 유익한 시나리오 예를 들어주세요.

¹⁹⁴<https://proandroiddev.com/loading-initial-data-in-launchedeffect-vs-viewmodel-f1747c20ce62>

1. Jetpack Compose 면접 질문

구글의 안드로이드 팀에서 [Jetpack Compose 1.0 stable 발표¹](#) 이후, Jetpack Compose는 프로덕션 환경에서의 채택이 가속화되었습니다. Google은 2023년까지 Google Play Store에 올라간 앱 중 125,000개 이상의 앱이 Jetpack Compose로 개발되었다고 밝혔습니다. 오늘날 Jetpack Compose는 안드로이드 개발뿐만 아니라, 멀티 플랫폼 지원으로 다양한 애플리케이션 개발에 널리 사용되는 UI 툴킷이 되었으며, 개발자가 더 높은 생산성과 효율성을 낼 수 있는 선언형 프로그래밍을 할 수 있도록 합니다.

일부 회사는 여전히 XML 기반 레이아웃에 의존하고 있지만, 한편으로 큰 프로젝트를 하루 아침에 XML에서 Compose로 전환할 수 있는 것이 아닙니다. 초기에 안드로이드 개발에 Kotlin이 채택되었을 때 Kotlin과 Java 코드가 공존했던 것처럼, 많은 개발자는 여전히 XML 기반 레이아웃과 Compose 접근 방식에 모두 익숙해져야 하는 과도기적인 시대에 있습니다. 그러나 최근 몇 년 동안 Jetpack Compose 관련 생태계가 크게 성장했고, 대부분의 일반적으로 마주할 수 있는 문제는 커뮤니티나 공식 솔루션을 통해 해결할 수 있으므로, 최근 개발되는 프로젝트들은 대체로 Jetpack Compose를 채택하는 경향이 있습니다.

Jetpack Compose는 향상된 코드 재사용성 및 Lifecycle, Navigation, Hilt와 같은 기존 안드로이드 프레임워크와의 원활한 호환성을 지원하는 등 다양한 이점을 제공합니다. 또한 코루틴이나 각종 커뮤니티에서 개발된 솔루션을 공식 솔루션으로 채택하고 확장 함수를 지원하는 등 유연한 확장성을 보여주고 있습니다. 이러한 장점들에도 불구하고 개발자는 성능을 최적화하고 UI 렌더링의 비효율성을 방지하기 위해 recomposition과 같은 Jetpack Compose의 내부 메커니즘에 대해 깊이 이해해야 합니다. 이는 선언적 UI의 장점이자 단점에 해당합니다.

이 챕터는 **Compose Fundamentals**, **Compose Runtime**, **Compose UI**의 메인 카테고리로 구성되어 있으며, 각 카테고리는 Jetpack Compose의 다양한 동작과 그 내부 원리를 파악하는데 중점을 둡니다. 기호에 따라 순서에 상관없이 읽으셔도 되지만, 구조화된 학습 경험을 위해 순차적으로 읽으셔도 문제없도록 내용이 설계되었습니다.

이 책은 가능한 모든 Jetpack Compose 관련 면접 질문을 다루기보다는, Compose을 실전에

¹<https://android-developers.googleblog.com/2021/07/jetpack-compose-announcement.html>

사용함에 있어 기본기를 다지면서 한편으로는 내부 작동 API나 그 메커니즘을 이해하여 Compose에 대한 전문성을 기르는 것을 목표로 합니다. 이 책의 킥이라고 할 수 있는 “**Pro Tips for Mastery**” 섹션은 주요한 Compose API의 내부 동작 구현을 자세히 살펴보며 Compose가 내부적으로 어떻게 동작하는지 실용성 있는 API를 기반으로 더 깊은 이해를 돋습니다. 일부 개념이 처음에는 복잡해 보일 수 있는데, 처음에는 누구나 그렇기 때문에 전혀 겁먹을 필요 없습니다. 개념이 머릿속에서 명확하게 그려질 때까지 계속 읽고 반복하시는 것만이 해답입니다.

두 번째 챕터에 오신 것을 환영합니다! 잠시 시간을 내어 커피 한 잔 가져오셔서 편안하게 본인의 속도에 맞춰 책 내용들을 천천히 탐색해 보시길 바랍니다. 이 책에서 다루는 질문들은 Jetpack Compose에 대한 이해를 높이고 기술면접을 자신 있게 대비하는 데 도움이 되도록 설계되었습니다. 그럼 즐거운 시간 되실 바랍니다! ☕✨

카테고리 0: Compose Fundamentals

Jetpack Compose는 시스템적으로 **Compose Compiler**, **Compose Runtime**, **Compose UI**의 세 가지 계층 구조로 구성되어 있습니다. `remember`, `LaunchedEffect`, `Box`, `Column`, `Row`와 같이 UI 화면을 만들기 위해 개발자들이 사용하는 API는 대부분 **Runtime** 및 **UI** 계층에서 제공됩니다. Jetpack Compose는 기존 프로젝트에 라이브러리 의존성을 추가하는 것만으로도 원활하게 작동되지만, 실제로 내부적인 동작 구조는 훨씬 복잡한 구조로 이루어져 있습니다.

일반적으로 애플리케이션 개발을 위해 Jetpack Compose를 활용한다면, 내부 구조에 대한 깊은 이해가 필수적으로 요구되는 것은 아닙니다. 그러나, 전반적인 아키텍처를 파악하면 Compose의 렌더링 단계 및 선언적 UI 개발의 특성 같은 개념을 포함하여 앱의 전반적인 성능을 높이는 등 다양한 역할을 이해하는 데 크게 도움이 될 수 있습니다.

이번 카테고리에서는 Jetpack Compose의 내부적인 동작 원리, Stability와 같은 안정성, 성능 개선 등 심화적인 내용을 다루고 있으며, 한편으로 굉장히 실용적인 내용들을 위주로 다룹니다. 이런 내용들을 처음 접하시는 분들은 조금 어렵게 느껴지실 수 있는데, 이미 Compose를 어느 정도 사용해 보신 분들께서는 보다 쉽게 이해하실 수 있는 **카테고리 1: Compose Runtime** 또는 **카테고리 2: Compose UI**부터 읽어보시는 것도 좋습니다.

Q) 0. Jetpack Compose의 동작 구조는 어떻게 이루어져 있나요?

Jetpack Compose는 선언적 접근 방식을 사용하여 네이티브 안드로이드 애플리케이션 (JetBrains에서 구축한 Compose Multiplatform은 크로스 플랫폼 지원)을 구축하기 위한 가장 최신의 UI 툴킷입니다. Compose의 구조는 **Compose Compiler**, **Compose Runtime**, **Compose UI**의 세 가지 주요 계층으로 이루어져 있습니다. 각 계층은 UI 코드를 상호 작용 가능한 애플리케이션으로 변환하는 데 중요한 역할을 합니다.



그림 197. compose-structure

Compose Compiler

Compose Compiler는 Kotlin으로 작성된 선언적 UI 코드를 Jetpack Compose가 실행할 수 있는 최적화된 코드로 변환하는 역할을 합니다. 컴파일 타임에 @Composable 함수를 처리하고, 필요한 UI 업데이트 및 recomposition 로직을 생성합니다. 컴파일러는 Kotlin 컴파일러로 구현되어 효율적인 코드 생성을 보장하고 상태 관리, 코드 최적화, 더 나은 성능을 위한 람다 리프팅(lambda lifting)과 같은 기능을 지원합니다.

KAPT² 및 KSP³와 같은 기존 어노테이션 처리 도구와 달리 Compose Compiler 플러그인은 FIR (Frontend Intermediate Representation⁴)에서 직접 작동합니다. 이러한 특성으로 컴파일러는 컴파일 타임에 정적 코드에 대해 더 자세하게 접근할 수 있으며, 개발자가 작성한 Kotlin 소스 코드를 동적으로 변환하고 최적화된 Java 바이트코드를 생성할 수 있습니다. @Composable과 같은 Compose 라이브러리의 어노테이션은 코드 생성, recomposition 관리 및 성능 최적화와 같은 작업을 조율하는 Compose Compiler의 내부 메커니즘을 통해 작동합니다. 이러한 고유한 접근 방식은 Kotlin 컴파일러 파이프라인과의 밀접한 통합을 보장하고, 개발 효율성과 런타임 성능을 모두를 향상시킵니다.

²<https://kotlinlang.org/docs/kapt.html>

³<https://github.com/google/ksp>

⁴중간 표현(Intermediate Representation, IR)은 컴파일러가 컴파일 프로세스 중에 소스 코드에 대한 설명을 생성하는데 데 사용되는 추상적인 코드 구조입니다. 소스 코드와 대상으로 하는 기계 코드 간의 다리 역할을 하여 플랫폼 독립적인 최적화, 코드 분석 및 효율적인 코드 생성을 가능하게 합니다.

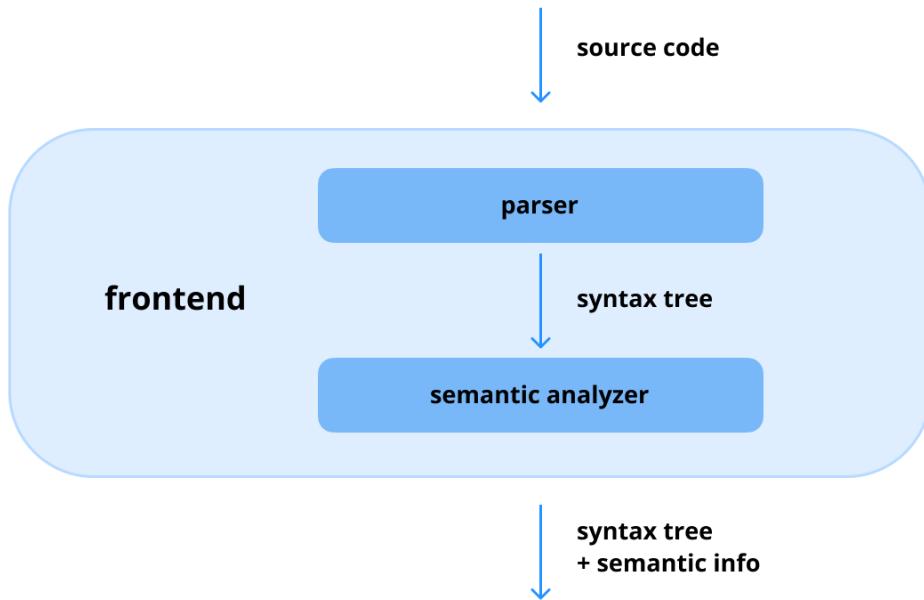


그림 198. compose-compiler

Compose Runtime

Compose Runtime은 recomposition 및 상태 관리를 지원하는 데 필요한 핵심 기능을 제공합니다. 변경 가능한 상태(mutable states)를 처리하고, 스냅샷(snapshots)을 관리하며, 애플리케이션 상태가 변경될 때마다 UI 업데이트를 트리거합니다. 또한, Compose의 반응형 UI 시스템을 구동하는 핵심 역할을 하여, 상태 변경에 따라 올바른 UI 컴포넌트가 동적으로 업데이트되도록 보장합니다.

Compose Runtime은 갭 버퍼(gap buffer)⁵ 자료 구조에서 영감을 받은 슬롯 테이블(slot table)⁶을 사용하여 컴포지션 상태를 메모이징(memoizing)하는 방식으로 작동합니다. 내부

⁵갭 버퍼(Gap buffer)는 텍스트 편집기에서 문자의 동적 시퀀스를 효율적으로 관리하는데 사용되는 자료 구조입니다. 필요할 때만 요소를 이동하여 빠른 삽입 및 삭제를 허용하는 “갭(gap)”이 있는 연속적인 메모리 블록을 유지하여 빈번한 수정 오버헤드를 줄입니다.

⁶슬롯 테이블(Slot table)은 컴포지션 단계 중 UI 컴포넌트의 상태를 저장하고 관리하기 위해 Jetpack Compose에서 사용되는 자료 구조입니다. UI 컴포넌트, 해당 관계 및 연관된 상태를 효율적으로 추적하여 상태 변경에 영향을 받는 요소만 업데이트함으로써 최적화된 recomposition을 가능하게 합니다.

직으로는 반응형 UI 구축에 필수적인 몇 가지 중요한 작업을 수행합니다. 여기에는 사이드 이펙트(side-effects)⁷ 관리, remember를 사용한 상태 보존, 상태 변경 시 recomposition 트리거, CompositionLocal을 사용한 컨텍스트별 데이터 저장, UI 계층 구조를 효율적으로 생성하기 위한 Compose 레이아웃 노드 구축 등이 포함됩니다. 이러한 동적 관리 시스템은 Jetpack Compose가 선언형 UI로서 작동할 수 있도록 하는 핵심적인 역할을 합니다.

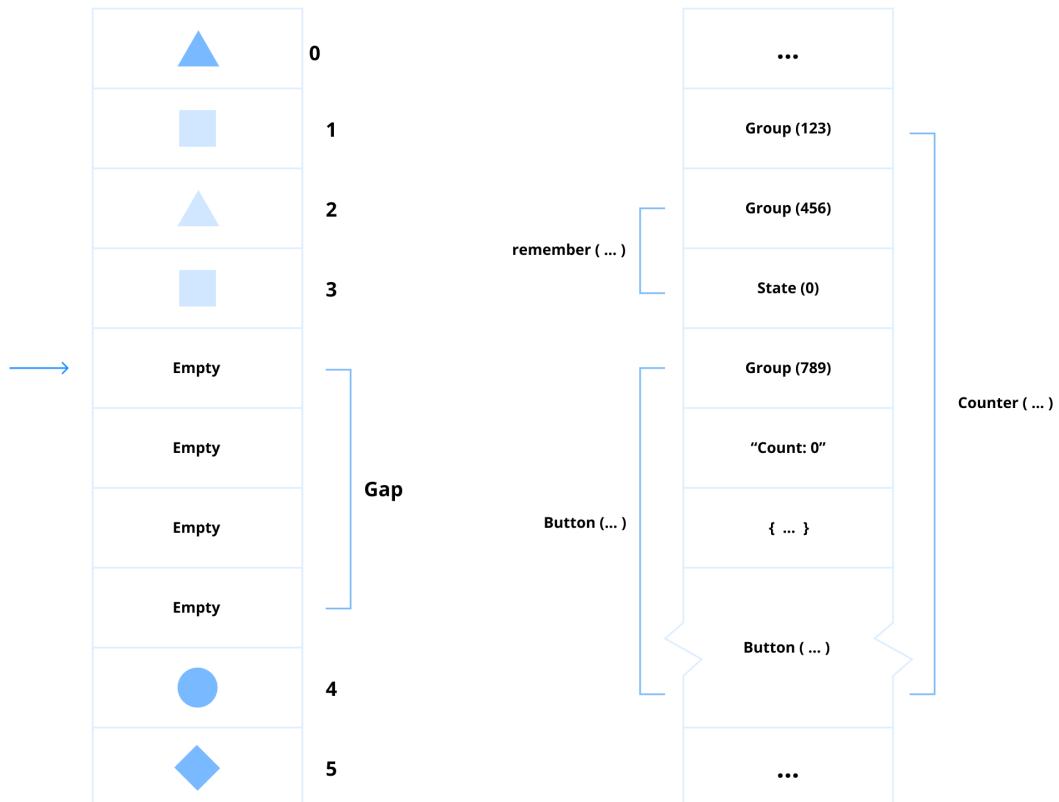


그림 199. compose-runtime

⁷사이드 이펙트(Side-effects)는 값을 반환하는 것 외에 프로그램 상태에 영향을 미치거나 외부 환경과 상호 작용하는 프로그래밍 작업입니다. 변수 수정, 네트워크 요청 수행 또는 UI 업데이트와 같은 예가 있으며, 이는 함수의 범위를 벗어난 프로그램 동작에 영향을 미칠 수 있습니다.

Pro Tips for Mastery: 갭 버퍼에서 링크 테이블로 마이그레이션

AOSP 코드⁸를 살펴보면, 안드로이드 팀은 갭 버퍼에서 링크 테이블(Link Table) 자료 구조로 마이그레이션하고 있습니다. 링크 테이블은 연결된 노드를 사용하여 데이터를 구성하므로 요소의 효율적인 삽입, 삭제 및 재배열이 가능합니다. 해당 변경 사항은 SlotTable의 편집 성능을 향상시키면서, 동시에 현재 구현된 동작 방식의 효율성을 유지하는 것을 목표로 합니다.

Compose UI

Compose UI 계층은 애플리케이션 구축을 위한 고수준 컴포넌트 및 UI 위젯을 제공합니다. 텍스트, 버튼, 레이아웃 컨테이너와 같은 기본 요소와 커스텀 UI 컴포넌트 구축을 위한 더 상위호환의 API를 포함합니다. 내부적으로 들어가면 Compose UI 모듈은 근본적으로 안드로이드의 전통적인 UI 시스템 위에서 동작하는 한편으로, 그 위에서 동작하는 Compose의 메커니즘이 따로 있기 때문에 생각보다 복잡한 관계를 이루고 있으며, 기존의 안드로이드 UI 시스템과 완전히 분리된 개념이라고 생각하기는 어렵습니다.

Compose UI 라이브러리는 Compose Runtime에 의해 처리되는 Compose 레이아웃 트리 구축을 단순화하도록 설계된 광범위한 컴포넌트를 제공합니다. [Kotlin Multiplatform⁹](#) 지원을 통해 JetBrains는 최근 [Compose Multiplatform¹⁰](#)를 적극적으로 발전시켜, 개발자가 안드로이드, iOS, 데스크톱 및 WebAssembly를 포함한 여러 플랫폼에서 동일한 Compose UI 라이브러리를 사용하여 일관된 UI를 만들 수 있도록 개발자 생태계를 만들고 있습니다. 이러한 크로스 플랫폼 호환성은 Flutter나 React Native와 유사하게 멀티 플랫폼 간의 개발을 간소화하고 일관성있는 사용자 경험을 보장합니다.

⁸<https://android-review.googlesource.com/c/platform/frameworks/support/+/2644758>

⁹<https://kotlinlang.org/docs/multiplatform.html>

¹⁰<https://www.jetbrains.com/compose-multiplatform/>

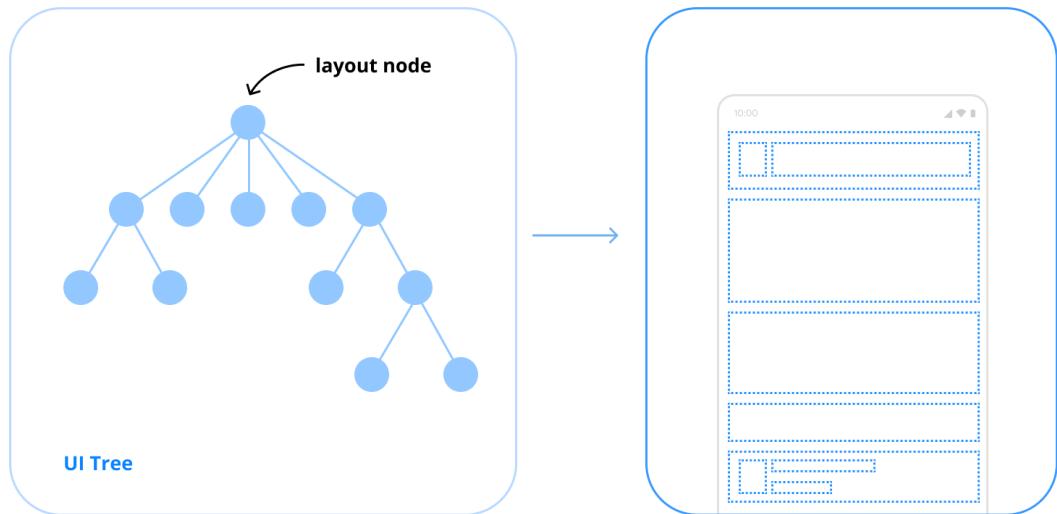


그림 200. layout

요약

Jetpack Compose의 구조는 관심사를 효과적으로 분리하도록 설계되었습니다. Compose Compiler는 UI 코드를 실행 가능한 컴포넌트로 변환하고, Compose Runtime은 상태 및 recomposition을 관리하며, Compose UI 계층은 UI 렌더링에 즉시 사용할 수 있는 위젯 및 UI 컴포넌트를 제공합니다. 이러한 계층화된 아키텍처는 전부 모듈식으로 구현되어 있어 서로 독립적인 관계를 이루고 있으면서도, 효율적이며 유지 관리 가능한 안드로이드 애플리케이션 개발을 보장합니다.

실전 질문

- Q) Compose Compiler의 역할은 무엇이고, KAPT 또는 KSP와 같은 전통적인 어노테이션 프로세서와 어떻게 다른가요?
- Q) Compose Runtime은 recomposition과 상태를 어떻게 관리하며, 내부적으로 어떤 자료 구조를 사용하나요?

Q) 1. Compose 페이즈(phase)에 대해 설명해 주세요,

Jetpack Compose는 UI를 화면에 그릴 때 **Composition**, **Layout**, **Drawing**의 세 가지 주요 단계로 나누어진 렌더링 파이프라인을 따릅니다. UI를 효율적으로 구축하고, 정렬하거나, 올바르게 렌더링하기 위해 해당 단계들을 순차적으로 수행합니다.

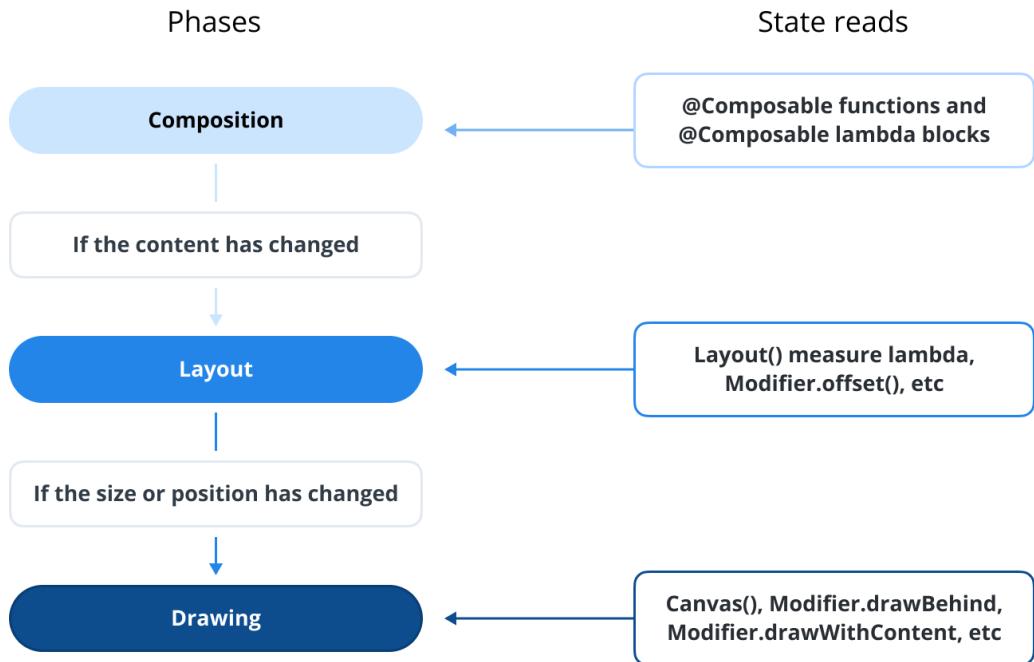


그림 201. compose-phase

Composition

Composition 단계는 `@Composable` 함수를 실행하고 UI 트리를 구축하여 컴포저블 함수에 대한 설명을 생성하는 역할을 합니다. 이 단계에서 Compose는 초기 UI 구조를 구축하고

Slot Table¹¹이라는 데이터 구조에 컴포저블 간의 관계를 기록합니다. 상태 변경이 발생하면 Composition 단계는 영향을 받는 UI에 대해 다시 계산하고, 필요한 경우 recomposition을 트리거합니다.

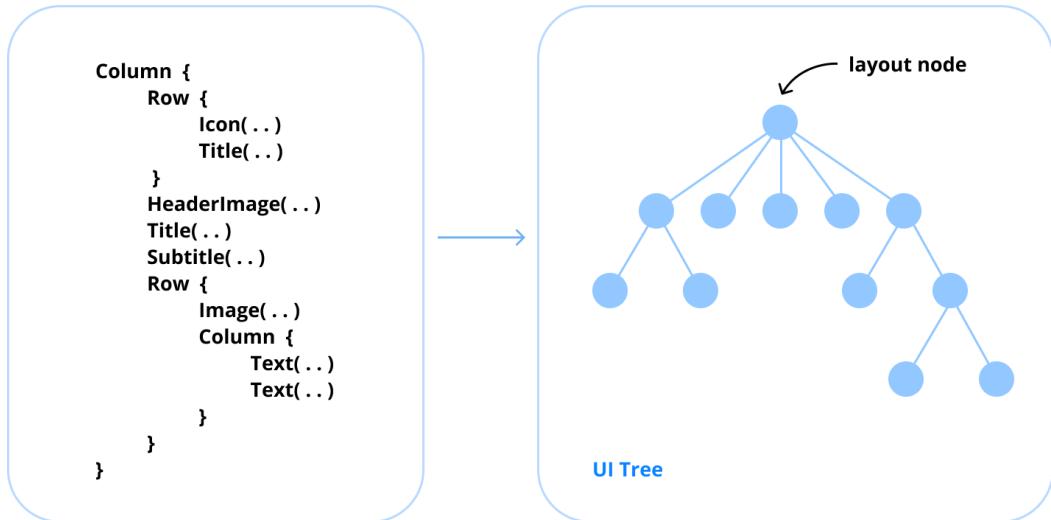


그림 202. composition

Composition의 주요 작업

- @Composable 함수 실행
- UI 트리 생성 및 업데이트
- recomposition을 위한 변경 사항 추적

Layout

Layout 단계는 Composition 단계 바로 직후에 수행됩니다. 제공된 제약 조건에 따라 각 UI 컴포넌트의 크기와 위치를 결정합니다. 각 컴포저블은 자식 요소를 측정(크기 및 위치

¹¹슬롯 테이블(Slot table)은 컴포지션 단계 중 UI 컴포넌트의 상태를 저장하고 관리하기 위해 Jetpack Compose에서 사용되는 자료 구조입니다. UI 컴포넌트, 해당 관계 및 연관된 상태를 효율적으로 추적하여 상태 변경에 영향을 받는 요소만 업데이트함으로써 최적화된 recomposition을 가능하게 합니다.

등)하고, 크기를 결정하며, 부모에 대한 상대적 위치를 비치합니다.

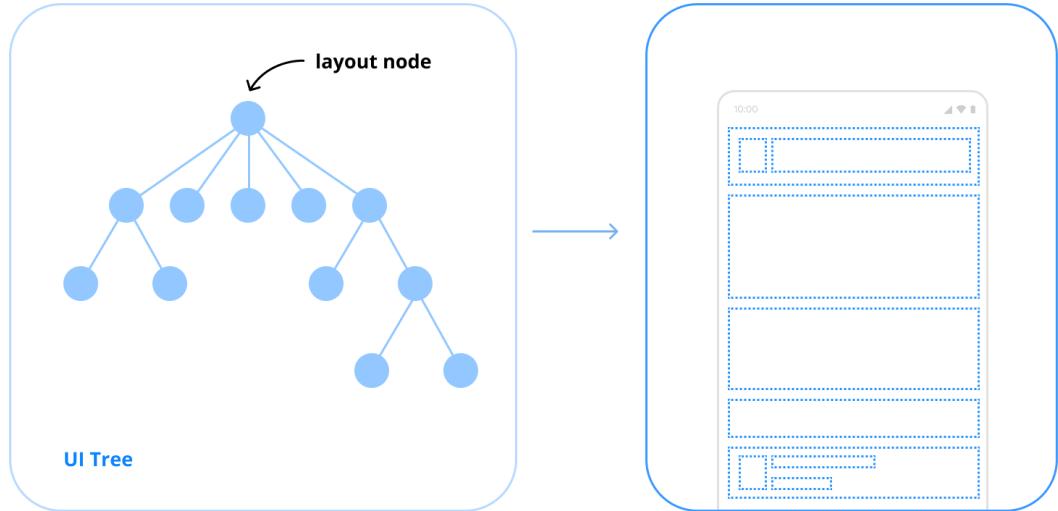


그림 203. layout

Layout의 주요 작업

- UI 컴포넌트 측정
- 너비, 높이 및 위치 정의
- 부모 컨테이너 내 자식 배치

Drawing

Drawing 단계는 앞선 Composition과 Layout 단계를 마친 UI 컴포넌트가 화면에 렌더링되는 절차입니다. Compose는 안드로이드에서 해당 UI들을 렌더링하기 위해 Skia 그래픽 엔진을 사용하여 부드럽고 하드웨어 가속 기반의 렌더링을 제공합니다. 커스텀 드로잉 로직은 Compose의 Canvas API를 사용하여 구현할 수 있습니다.

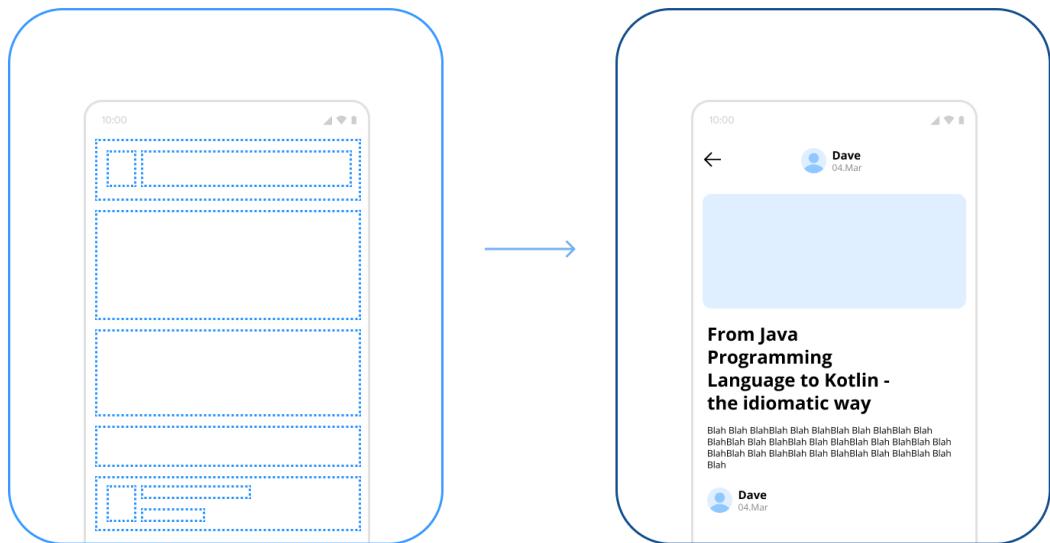


그림 204. drawing

Drawing의 주요 작업

- 시각적 요소 렌더링
- 화면에 UI 컴포넌트 그리기
- 커스텀 드로잉 작업 적용

요약

Jetpack Compose의 3단계 렌더링 모델은 효율적이며 확장 가능한 UI 구축 프로세스를 보장합니다. Composition 단계는 UI 트리를 구축하고, Layout 단계는 컴포넌트를 정렬하며, Drawing 단계는 모든 것을 시각적으로 렌더링 합니다. Compose UI 렌더링 프로세스에 관련하여 더 심층적으로 학습하고 싶으시다면 [Jetpack Compose Phases¹²](#) 공식 안드로이드 문서를 참조하실 수 있습니다.

¹²<https://developer.android.com/develop/ui/compose/phases>

실전 질문

- Q) Composition 단계에서는 어떤 일이 발생하며, recomposition과 어떤 관련이 있나요?
Q) Compose Phase에서 Layout 단계는 어떻게 작동하나요?

Q) 2. Jetpack Compose가 선언적(declarative) UI 프레임워크라고 불리는 이유는 무엇인가요?

Jetpack Compose는 개발자가 상태 변경 시 UI를 어떻게 업데이트할지를 나타내는 것이 아닌, 특정 상태에서 UI가 어떻게 보여야 하는지를 설명하는 선언적 UI 프레임워크의 특성을 가지고 있습니다. 이는 개발자가 뷰를 업데이트하고 UI 일관성을 유지하기 위해 수동으로 UI를 업데이트하는 전통적인 명령형(imperative) UI 접근 방식과 비교됩니다.

Jetpack Compose에서 선언적 UI의 주요 특징

상태 주도 UI(State-Driven UI): 선언적 UI 프레임워크에서는 상태 관리 시스템이 라이브러리 자체에 내장되어 있습니다. 시스템은 각 컴포넌트의 상태를 추적하고 상태가 변경될 때 UI를 자동으로 업데이트합니다. 개발자는 특정 상태에 대해 UI가 어떻게 보여야 하는지만 정의하면 되고, 프레임워크가 렌더링 업데이트를 처리합니다. Jetpack Compose에서 UI는 전적으로 상태에 의해 구동됩니다. 상태가 변경될 때마다 프레임워크는 recomposition을 트리거하여 영향을 받는 UI 컴포넌트만 업데이트하고 최신 데이터를 반영하여 뷰를 수동적으로 관리할 필요가 없도록 합니다.

컴포넌트를 함수 또는 클래스로 정의: 선언적 UI 프레임워크의 특성 중 하나로는 UI 컴포넌트를 함수 또는 클래스로 표현되는 모듈식 컴포넌트로 정의하도록 합니다. 이러한 컴포넌트는 UI 레이아웃과 동작을 모두 설명하여 XML과 같은 마크업 언어와 Kotlin 또는 Java와 같은 네이티브 프로그래밍 언어 간의 간극을 줄입니다. Jetpack Compose에서는 @Composable 어노테이션을 함수에 붙임으로써 재사용 가능한 UI 컴포넌트를 정의합니다. 각 함수는 현재 상태를 기반으로 UI를 설명하고 다른 함수와 결합하여 모듈식이며 확장 가능한 구조를 만들 수 있습니다.

직접적인 데이터 바인딩(Direct Data Binding): 선언적 UI 프레임워크를 사용하면 개발자가 모델 데이터를 UI 컴포넌트에 직접 바인딩하여 수동적으로 데이터를 동기화해야 하는 수고를 덜 수 있습니다. 해당 접근 방식은 더 깔끔하고 유지 관리하기 쉬운 코드를 만듭니다. Jetpack Compose에서 개발자는 함수 매개변수를 통해 데이터를 바인딩하여 기존의 XML처럼

추가적으로 요구되는 데이터 바인딩 계층이나 복잡한 어댑터 패턴을 사용하지 않고도 UI 개발을 크게 단순화합니다.

컴포넌트 멱등성(Component Idempotence): 선언적 프레임워크의 핵심적인 특징은 멱등성입니다. 즉, 컴포넌트는 호출 횟수에 관계없이 동일한 입력에 대해 동일한 출력을 생성합니다. 해당 속성은 컴포넌트의 일관된 동작과 재사용성을 보장합니다. Jetpack Compose에서 모든 @Composable 함수는 본질적으로 멱등성의 성질을 가지므로, 동일한 입력 매개변수에 대하여 반드시 동일한 UI 결과를 생성하여 예측 가능하고 안정적인 UI 렌더링을 가능하도록 합니다.

Jetpack Compose vs. XML

Jetpack Compose는 선언적 UI 접근 방식을 채택하였고, 개발자가 상태 조건을 Kotlin 내에 직접 포함시켜 논리적으로 UI 코드를 작성할 수 있도록 합니다. 이러한 접근 방식은 상태 변경에 대해 UI가 자동으로 업데이트되도록 보장하고 상태 관리와 코드 가독성 모두를 단순화합니다. 해당 접근법에 대한 이해를 돋기 위해 버튼을 누를 때마다 눌린 횟수를 보여주는 아래 예시를 살펴볼 수 있습니다.

그림 205. Jetpack Compose.kt

```
1  @Composable
2  fun Main() {
3      var count by remember { mutableStateOf(0) }
4      CounterButton(count)
5      count++
6  }
7 }
8
9 @Composable
10 fun CounterButton(count: Int, onClick: () -> Unit) {
11     Button(onClick = onClick) {
12         Text("Clicked: $count")
13     }
14 }
```

Jetpack Compose가 주요 원칙에 따라 선언적 UI 프레임워크로 분류되는 이유는 다음과 같습니다.

1. **함수로 UI 정의**: @Composable 어노테이션이 달린 함수는 Compose Compiler에 의해 해석 및 변환되어 선언적 UI 생성을 가능하게 합니다. 이는 선언적 UI의 첫 번째 원칙인 함수 또는 클래스를 통해 UI 컴포넌트를 정의하는 것과 일치합니다.
2. **상태 관리**: Compose Runtime에서 제공하는 remember와 같은 함수는 컴포저블의 상태와 생명주기를 효율적으로 관리합니다. 이는 두 번째 선언적 UI 특성인 컴포넌트 내 자동 상태 관리를 충족합니다.
3. **직접 데이터 바인딩**: CounterButton 컴포저블 함수의 count 매개변수는 UI에 직접 바인딩되어 데이터가 UI 컴포넌트에 원활하게 연결될 수 있음을 보여줍니다. 이는 선언적 UI의 세 번째 핵심 원칙인 직접적인 데이터 바인딩 조건을 충족합니다.
4. **컴포넌트 역동성**: CounterButton 컴포저블은 동일한 입력 값에 대해 일관되게 동일한 UI 출력을 생성하여 예측 가능한 동작을 보장합니다. 이는 선언적 UI의 네 번째 원칙인 신뢰할 수 있고 재사용 가능한 컴포넌트를 위한 역동성 보장을 지원합니다.

XML과 Jetpack Compose의 동작 방식을 코드를 통해 비교해 보고, Compose가 왜 선언형 UI인지 이해를 더해보도록 하겠습니다.

그림 206. XML.xml

```
1 <RelativeLayout  
2     android:layout_width="match_parent"  
3     android:layout_height="match_parent"  
4     android:gravity="center"  
5     android:orientation="horizontal"  
6     android:padding="4dp">  
7  
8     <Button  
9         android:id="@+id/button"  
10        android:layout_width="wrap_content"  
11        android:layout_height="wrap_content"  
12        android:layout_centerInParent="true"  
13        android:text="Clicked: 0" />  
14  
15 </RelativeLayout>
```

언뜻 보기에도 XML 레이아웃은 XML 자체가 본질적으로 선언적이기 때문에 선언적 UI 접근 방식과 유사해 보일 수 있습니다. 안드로이드의 전통적인 레이아웃 시스템에서 개발자는 구조와 속성을 설명하여 UI가 어떻게 보여야 하는지를 정의하고, 기본적인 렌더링 프로세스는 프레임워크에 맡깁니다. 이는 선언적 프로그래밍의 핵심 원칙인 UI가 **어떻게** 렌더링되어야 하는지가 아니라 **무엇**을 렌더링 할지 정의해야 하는 것과 일치합니다.

가장 주요 차이점은 상태 및 로직 처리에 있습니다. XML 기반 개발에서는 UI 구조와 속성이 XML에 정의되는 반면, 상태 관리 및 UI 업데이트는 Java 또는 Kotlin을 사용하여 명령형 코드에서 별도로 구현됩니다. 이것이 가장 큰 차이점이라고 볼 수 있는데, XML과 같이 UI 로직에 대한 언어적 분리는 더 복잡한 워크플로우로 이어지며, UI와 애플리케이션 로직 간의 수동적인 동기화가 요구됩니다.

그림 207. Imperative.kt

```
1 var counter = 0
2
3 binding.button.setOnClickListener {
4     counter++
5     binding.button.text = counter.toString()
6 }
```

반면, Jetpack Compose와 같은 선언적 프레임워크는 상태 관리와 UI 정의를 밀접하게 바인딩하여 상태가 변경될 때 자동적으로 UI 업데이트를 가능하게 할 뿐만 아니라, 상태 기반 업데이트를 Kotlin 언어 자체 내에서 원활하게 통합합니다. 즉, UI와 상태 변경에 대한 응답 방식을 모두 같은 위치에서 정의할 수 있기 때문에, 코드가 더 응집력 있고 별도의 명령형 핸들러가 필요하지 않습니다.

요약

Jetpack Compose는 개발자가 앱 상태를 기반으로 UI가 **무엇**을 표시해야 하는지 표현할 수 있기 때문에 선언적 UI라고 할 수 있습니다. **recomposition**¹³을 통해 UI가 **어떻게** 업데이트되는지에 대해서는 Compose Runtime이 자동으로 처리하여 개발자는 UI 업데이트에 대해서는 전혀 신경 쓰지 않아도 되고, 잠재적으로 더 깔끔하고 유지 보수가 쉬운 코드를 만들립니다. 이러한 선언적 UI 개발은 상태에 따른 동적인 UI 업데이트가 가능하도록 하여 코드를 더 직관적으로 만들고, 잘 활용한다면 안드로이드 앱 개발의 복잡성을 크게 줄입니다.

¹³<https://developer.android.com/develop/ui/compose/mental-model#recomposition>

실전 질문

Q) Jetpack Compose의 선언적 UI의 특성은 전통적인 명령형 XML UI 개발과 어떻게 다르며, 어떤 이점을 제공하나요? 또한, 기존 XML UI 개발에 비해서 유의해야 할 사항은 무엇인가요?

Q) Jetpack Compose는 어떻게 컴포저블 함수에 대해 멱등성을 보장하며, 이것이 선언적 UI 시스템에서 중요한 이유가 무엇인가요?

Q) 3. **recomposition**이란 무엇이며 언제 발생하나요? 또한 앱 성능과 어떤 관련이 있나요?

Jetpack Composes는 이미 렌더링된 UI 레이아웃을 업데이트하기 위해 세 가지 주요 단계 (Composition, Layout, Drawing)를 통하여 상태 변경이 발생할 때마다 UI를 다시 그리는 메커니즘을 사용합니다. 이러한 프로세스를 **리컴포지션(Recomposition)** 이라고 합니다. **recomposition**이 발생하면 Compose는 **Composition** 단계부터 새롭게 시작하며, 여기서 컴포저블 노드는 UI 변경 사항을 Compose Runtime에 알리고, 업데이트된 UI가 최신 상태를 반영하도록 보장합니다.

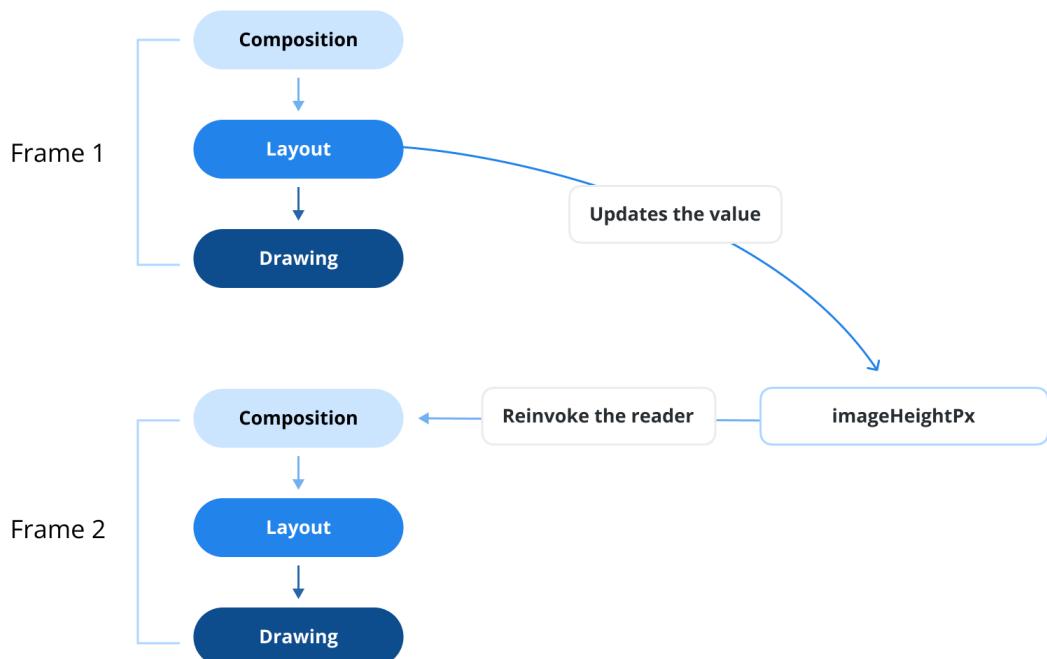


그림 208. recomposition

Recomposition이 발생하는 조건

대부분의 모바일 애플리케이션에서는 앱 내 데이터 모델을 메모리에 담아 표현하는 **상태 (state)**를 유지합니다. 상태 변경됨에 따라 UI도 함께 동기화되도록 하기 위해 Jetpack Compose는 두 가지 메인 메커니즘을 통해 recomposition을 트리거합니다.

- 매개변수에 변경이 발생했을 때(Input Changes):** 컴포저블 함수는 입력 매개변수가 변경될 때 recomposition을 트리거합니다. Compose 런타임은 equals() 함수를 사용하여 새 매개변수 값을 이전 매개변수 값과 비교합니다. 비교 결과가 false이면 런타임은 변경 사항을 인지하고 recomposition을 트리거하여 동기화가 필요한 부분에 한해서만 UI를 업데이트합니다.
- 상태 변경이 관찰되었을 때(Observing State Changes):** Jetpack Compose는 일반적으로 remember 함수와 **State API**를 함께 사용하여 상태 변경을 모니터링합니다. 이러한 접근 방식은 상태 객체를 메모리에 보존하고 recomposition이 발생했을 때에도 메모리에 저장된 값을 복원하여 UI에 최신 상태를 일관되게 반영하도록 보장합니다.

Recomposition과 성능

Jetpack Compose에서 `recomposition`은 UI가 상태 변화를 자동으로 감지하고 동기화 될 수 있도록 하는 반응형 특성을 담당하는 핵심 기능이지만, 과도하거나 불필요한 `recomposition`은 앱 성능을 저하시킬 수 있습니다. 따라서, `recomposition` 작동 방식과 이를 효과적으로 추적하는 방법을 이해하는 것은 Compose 애플리케이션 최적화에 필수적입니다. `recomposition`은 이 장 뒷부분의 “`stability`란 무엇인가요?” 질문에서 더 자세히 살펴볼 안정성(`stability`)과 같은 다양한 요인의 영향을 받습니다. `recomposition`을 최적화하고 앱 성능을 향상시키려면 [Layout Inspector](#)¹⁴를 사용하여 실제 앱 내에서 발생하는 `recomposition` 횟수를 추적함으로써 불필요한 `recomposition`을 식별하는 것이 효과적입니다.

`Layout Inspector`를 사용하면 에뮬레이터 또는 실제 기기에서 실행 중인 앱의 Compose 레이아웃을 트래킹 할 수 있습니다. 컴포저블이 얼마나 자주 `recompose` 되거나 `recomposition`을 건너뛰는지 모니터링하여 잠재적인 성능 문제를 식별하는 데 도움이 됩니다. 예를 들어, 코딩 실수로 인해 과도한 UI `recomposition`이 발생하여 성능이 저하되거나, UI가 `recompose`되지 않아 업데이트돼야 할 UI가 제대로 업데이트되지 않는 경우가 발생할 때도 있는데, 이런 문제들을 추적하기에 유용합니다. 특히 컴포저블 함수 특성상 최상위 노드 컴포저블 아래에 수많은 자식 노드가 존재할 수 있기 때문에, 어떤 컴포저블에서 문제가 발생했는지 개발자가 감으로만 식별하기 어려운 경우가 많습니다. 이런 경우 `Layout Inspector`가 굉장히 도움됩니다.

¹⁴<https://developer.android.com/studio/debug/layout-inspector>

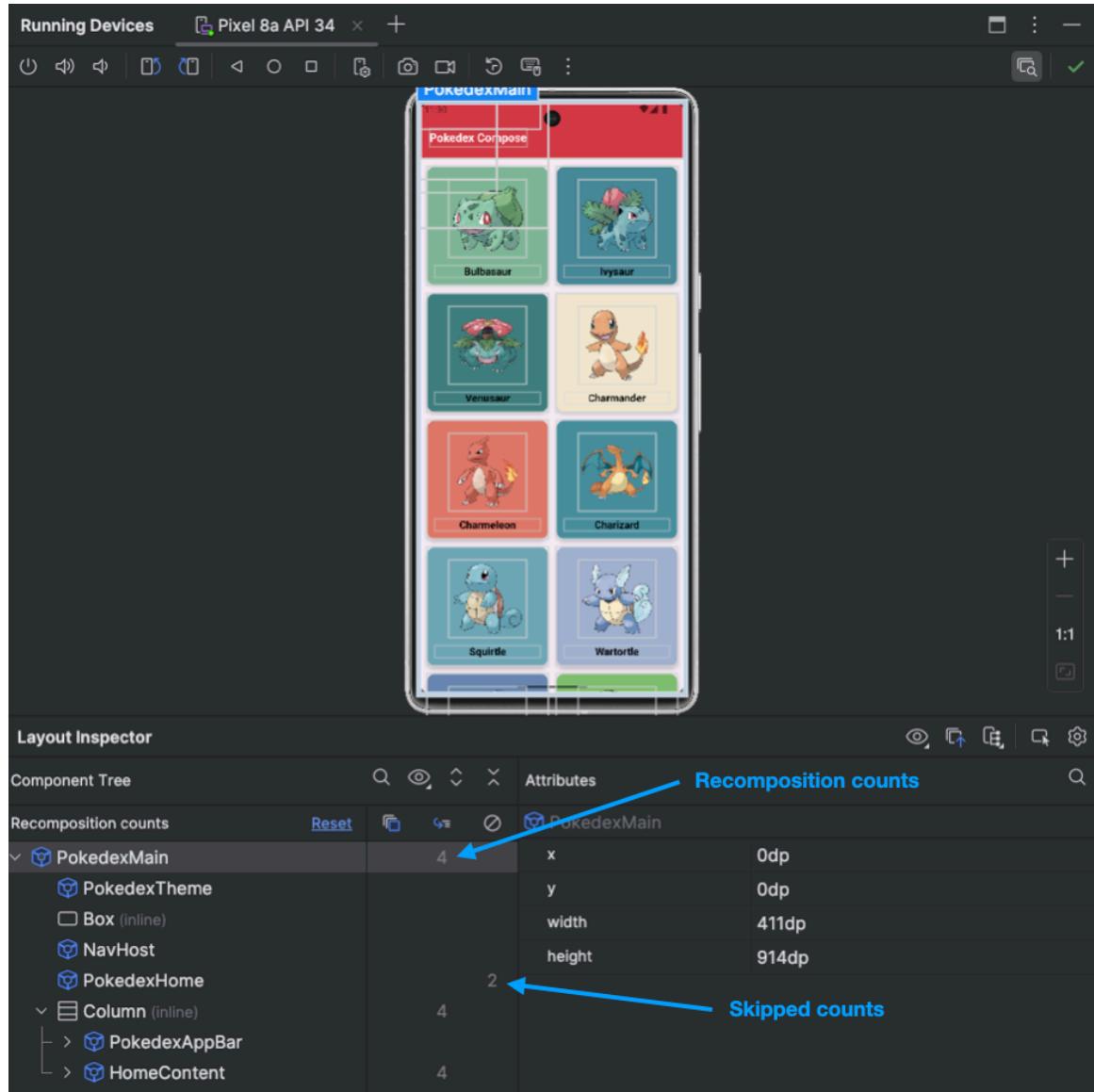


그림 209. layout-inspector

Android Studio에서 Layout Inspector는 recomposition이 발생할 때 UI의 어떤 부분이 recomposition을 트리거하는지 쉽게 찾아낼 수 있도록 도와줍니다. 따라서, recomposition 횟수를 추적하고 UI 업데이트를 최적화하는 데 활용하여 Compose 앱이 최적의 성능을 유지하도록 보장하는 데 유용합니다.

프로젝트에서 recomposition 추적을 시험해 볼 수 있는 또 다른 도구가 있는데, 바로 **Composition Tracing**¹⁵입니다. 이 도구는 성능 문제를 진단하는 데 유용한 도구로, 시스템 추적을 통해 오버헤드가 낮은 측정을 제공하고 메서드 추적을 통해 세부적인 함수 호출 추적을 제공하지만, 앱 성능에는 직접적으로 영향을 미치지 않습니다. 시스템 추적은 일반적으로 개별적인 컴포저블 함수를 포함하지는 않는다는 점이 조금 아쉽지만, 구글팀에서 지속적으로 개선시키고 있고, 시스템 추적의 효율성과 메서드 추적의 세분성을 결합하여 Jetpack Compose를 사용하는 프로젝트에서 나름대로 성능 추적을 가시화시키려는 노력이 엿보이고 있습니다.

요약

Recomposition은 Jetpack Compose에서 반응형 UI를 만드는 데 필수적이지만, Compose Phase라는 세 가지 주요 단계(Composition, Layout, Drawing)를 완전히 처음부터 다시 시작해야 하고 그에 따른 성능 비용이 발생합니다. 앱 성능을 최적화에 대해서 더 심화적인 학습을 원하시는 분께서는 [6 Jetpack Compose Guidelines to Optimize Your App Performance](#)¹⁶ 또는 [Optimize App Performance By Mastering Stability in Jetpack Compose](#)¹⁷ 블로그 포스트를 읽어보시는 것을 적극 권장합니다. Jetpack Compose에서 불필요한 recomposition을 최소화하면서 앱의 UI 성능을 최적화하는 모범 사례와 여러 가지 꿀팁들을 제공합니다.

실전 질문

Q) 불필요한 recomposition을 줄이고 앱 성능을 최적화해보신 경험이 있나요? 앱 성능을 향상 시키기 위해 어떤 전략을 사용해 볼 수 있을까요?

Q) 4. Composable 함수는 내부적으로 어떻게 작동하나요?

Jetpack Compose는 일반적인 함수를 선언적 UI로 동작시키기 위해 @Composable 어노테이션을 사용합니다. 해당 어노테이션이 붙은 함수는 컴파일 타임 시 **Compose Compiler Plugin**에 의해 개발자가 작성한 Kotlin 함수를 Jetpack Compose의 내부 메커니즘에 맞는 상태 기반 UI 코드로 변환됩니다.

¹⁵<https://developer.android.com/develop/ui/compose/tooling/tracing>

¹⁶<https://medium.com/proandroiddev/6-jetpack-compose-guidelines-to-optimize-your-app-performance-be18533721f9>

¹⁷<https://medium.com/proandroiddev/optimize-app-performance-by-mastering-stability-in-jetpack-compose-69f40a8c785d>

컴파일러 변환 (Compiler Transformation)

함수에 `@Composable` 어노테이션을 추가하면 **Compose Compiler Plugin**이 Kotlin 컴파일 프로세스를 가로챕니다. 컴파일러는 컴포저블 함수를 표준적인 Kotlin 함수로 취급하지 않고, Compose의 반응형 시스템을 추가하기 위해 기존의 함수에 추가적인 매개변수와 각종 로직을 주입합니다. 가장 중요한 매개변수 중 하나는 컴포지션 상태를 추적하고 UI 상태가 변경될 때 `recomposition`을 처리하는 Composer 매개변수입니다. Composer 매개변수는 컴파일 타임에 컴파일러가 주입하는 객체이기 때문에 개발자는 해당 존재의 유무 자체를 알지 못해도 되고, 최종적으로 사용되는 API의 형태는 굉장히 심플합니다.

아래와 같이 간단하게 텍스트를 보여주는 컴포저블 함수가 있다고 합시다.

그림 210. MyComposable.kt

```

1 @Composable
2 fun MyComposable() {
3     Text("Hello, Compose!")
4 }
```

내부적으로 이는 Composer 객체와 상태 관리를 위한 기타 메타데이터를 주입하여 완전히 새로운 버전의 함수로 변형합니다. 위의 간단한 코드를 JavaByte코드로 디컴파일하면 아래와 같은 결과가 나옵니다.

```

1 private static final void MyComposable(Composer $composer, int $changed) {
2     $composer = $composer.startRestartGroup(1017251926);
3     ComposerKt.sourceInformation($composer,
4             "C(MyComposable)119@3919L23:MainActivity.kt#nzuxg3");
5     if ($changed == 0 && $composer.getSkipping()) {
6         $composer.skipToGroupEnd();
7     } else {
8         if (ComposerKt.isTraceInProgress()) {
9             ComposerKt.traceEventStart(1017251926, $changed, -1,
10                    "com.xperiventure.vlip.ui.MyComposable (MainActivity.kt:118)");
11     }
12 }
```

```

11     TextKt.Text-Nvy7gAk("Hello, Compose!", (Modifier)null, 0L, (TextAutoSize)null,
12         → 0L, (FontStyle)null, (FontWeight)null, (FontFamily)null, 0L,
13         → (TextDecoration)null, (TextAlign)null, 0L, 0, false, 0, 0, (Function1)null,
14         → (TextStyle)null, $composer, 6, 0, 262142);
15     if (ComposerKt.isTraceInProgress()) {
16         ComposerKt.traceEventEnd();
17     }
18
19     ScopeUpdateScope var10000 = $composer.endRestartGroup();
20     if (var10000 != null) {
21         var10000.updateScope(MainActivityKt::MyComposable$lambda$0);
22     }

```

컴포지션 및 리컴포지션 (Composition and Recomposition)

Compose Runtime은 `@Composable` 함수의 생명주기를 관리합니다. 앞서 다루었던 Compose Phases의 3단계 중 첫 번째 단계인 **Composition 단계**에서 런타임은 컴포저블 함수를 실행하고 UI 트리를 구축합니다. 해당 트리는 **Slot Table**이라는 데이터 구조에 저장되어 Compose가 UI를 효율적으로 관리하고 업데이트하는 역할을 담당합니다.

만약, 상태가 변경되면 **Recomposition**이 트리거 됩니다. 업데이트해야 하는 UI 트리를 다시 빌드하는 대신 Compose는 슬롯 테이블을 사용하여 UI의 어떤 부분을 업데이트해야 하는지 결정하고, 해당 컴포저블 함수만 선택적으로 다시 실행합니다. 이는 추후에 살펴볼 **Smart Recomposition**이라는 동작과 관련이 있습니다.

Remember 및 상태 관리 (Remember and State Management)

상태를 관리하기 위해 Compose는 `remember` 및 `State`와 같은 API를 제공합니다. 이러한 메커니즘은 런타임 및 컴파일러와 긴밀하게 작용하여, `recomposition`으로부터 상태를 보존하여 UI가 앱의 데이터 모델과 일관성을 유지하도록 보장합니다.

아래는 버튼을 누르면 버튼 누른 횟수를 보여주는 간단한 함수입니다.

그림 211. State.kt

```
1 @Composable
2 fun Counter() {
3     var count by remember { mutableStateOf(0) }
4     Button(onClick = { count++ }) {
5         Text("Count: $count")
6     }
7 }
```

여기서 `remember`는 `count` 상태가 `recomposition`이 발생하더라도 값이 메모리에 유지되도록 보장하고, `mutableStateOf`는 상태가 변경될 때 Compose에 이 사실을 알리고 `recomposition`을 트리거합니다.

요약

내부적으로 컴포저블 함수는 Compose Compiler Plugin을 활용하여 Kotlin 코드를 Compose Runtime에서 관리하는 반응형 UI 컴포넌트로 변환합니다. `remember`와 같은 상태 관리 도구와 함께 동작하는 이러한 상태 기반 시스템은 상태 변경을 효율적으로 트래킹하고, 렌더링 및 동적 UI 업데이트를 보장합니다. 이러한 아키텍처가 기반이 되어 Jetpack Compose를 선언적이고 효율적인 UI 프레임워크로 만드는 것입니다. 이와 관련하여 추가적인 학습을 원하신다면 [Android Developers: Under the hood of Jetpack Compose¹⁸](#)를 살펴보시는 것을 권장합니다.

실전 질문

Q) 함수에 `@Composable` 어노테이션을 추가하면 컴파일 시 내부적으로 어떻게 동작하고, 이것이 Jetpack Compose를 선언형 UI로 작동하도록 하는 데 어떤 역할을 하나요?

Pro Tips for Mastery: Compose Compiler와 Composer

Compose Compiler는 빌드 프로세스 중에 `@Composable` 함수의 중간 표현(IR)을 최적화된 코드로 변환합니다. 이 변환은 UI 렌더링 및 상태를 효율적으로 관리하여 Compose에서

¹⁸<https://medium.com/androiddevelopers/under-the-hood-of-jetpack-compose-part-2-of-2-37b2c20c6cdd>

선언적 UI 패러다임을 가능하게 하는 데 필수적인 역할을 합니다. 해당 프로세스의 핵심 구성 요소 중 하나는 컴포저를 함수의 상태를 추적하고 관리하는 데 중추적인 역할을 하는 Composer입니다.

Composable 함수 변환

일반적인 Kotlin 함수에 @Composable 어노테이션을 달면 Compose 컴파일러는 해당 함수에 대해 Composer라는 암시적 매개변수를 추가하여 함수 자체를 바꿉니다. 이 매개변수는 컴포저를 함수와 Compose Runtime 간의 다리 역할을 하여 런타임이 UI 상태, recomposition 및 기타 핵심 기능을 효율적으로 관리할 수 있도록 합니다.

가령, 다음과 같이 간단한 컴포저를 함수가 있습니다.

그림 212. Greeting.kt

```

1 @Composable
2 fun Greeting(name: String) {
3     Text("Hello, $name!")
4 }
```

위 코드는 Compose 컴파일러에 의해 컴파일 타임에 다음과 같은 함수로 변환됩니다.

그림 213. Composer.kt

```

1 fun Greeting(name: String, composer: Composer, key: Int) {
2     composer.startRestartGroup(key) // 리컴포지션 그룹 시작
3     val nameArgChanged = composer.changed(name) // name 매개변수 변경 감지
4     // name이 변경되었거나 그룹이 처음 생성된 경우 Text 실행
5     if (nameArgChanged || composer.inserting) {
6         composer.startReplaceableGroup(..) // 대체 가능 그룹 시작
7         Text("Hello, $name!", composer = composer, changed = 0) // Text 호출 시
8             ↪ Composer 전달
9         composer.endReplaceableGroup() // 대체 가능 그룹 종료
10    } else {
11        composer.skipToGroupEnd() // 변경 없으면 스킵
12    }
13 }
```

```
12     composer.endRestartGroup()?.updateScope { // 리컴포지션 그룹 종료 및 스코프 업데이트
13         Greeting(name, composer, key or 1) // 리컴포지션 시 호출될 람다 설정
14     }
15 }
```

이와 같은 코드 변환은 런타임이 입력 매개변수(위의 예시에서는 name 매개변수)의 변경 사항을 추적하고 recomposition의 필요 여부를 결정하기 위한 후크(hook) 기법이 사용된 것입니다. 이러한 후크를 활용하여 Compose는 변경되지 않은 UI에 대해 recomposition을 건너뛰고 성능을 최적화할 수 있습니다.

Composer의 역할

Composer¹⁹는 중앙 상태 관리자 역할을 하는 Compose 런타임의 저수준 API입니다. Compose의 Kotlin 컴파일러 플러그인을 통해 활용되고, code generation helpers(코드 생성 헬퍼)라는 컴파일 타임 코드 생성 시스템에서 사용되는 인터페이스입니다.

Composer가 담당하는 역할은 다음과 같습니다.

1. **상태 관리(State Management)**: 상태를 추적하고 매개변수 입력값이 변경될 때 컴포저블 함수가 올바르게 recompose 되도록 보장합니다.
2. **UI 계층 구조 구성(UI Hierarchy Construction)**: UI의 트리 구조를 유지하고 상태가 변경됨에 따라 노드를 효율적으로 업데이트합니다.
3. **최적화(Optimization)**: 입력 매개변수의 변경 사항을 분석하여 UI의 어떤 부분을 recompose해야 하는지 결정합니다.
4. **리컴포지션 제어(Recomposition Control)**: 특정 컴포저블에 대한 recomposition 시작, 건너뛰기 또는 종료를 포함한 recomposition의 생명주기를 조정합니다.

Composer는 recomposition이 점진적으로 수행될 수 있도록 보장합니다. 즉, UI 트리의 필요한 부분만 업데이트하여 불필요한 연산을 줄이고 성능을 향상시킵니다.

¹⁹<https://cs.android.com/androidx/platform/frameworks/support/+/androidx-main:compose/runtime/runtime/src/commonMain/kotlin/androidx/compose/runtime/Composer.kt;l=475?q=composer>

요약

Compose 컴파일러는 `@Composable` 함수를 변환하여 Composer 및 필요한 매개변수를 주입하고, Compose 런타임이 UI 상태 및 recomposition을 효율적으로 관리할 수 있도록 합니다. Composer는 상태 추적, recomposition 및 UI 업데이트를 처리하는 Compose 런타임의 중추적인 역할을 합니다. 이러한 컴파일러와 런타임의 상호작용 시스템으로 인해 Jetpack Compose가 선언적 UI 패러다임대로 동작할 수 있도록 합니다.

Pro Tips for Mastery: 컴포저블 함수 내에서 직접 비즈니스 로직 호출을 피해야 하는 이유

Jetpack Compose는 처음부터 멀티스레드 안전성을 고려하여 미래 지향적인 사고방식으로 설계되었습니다. 현재 컴포저블 함수는 멀티스레드 환경에서 실행되지 않지만(당장은 Compose가 메인 스레드에서 작동합니다), 구글의 **Compose 팀 소속 엔지니어는 컴포저블 함수가 멀티스레드 환경에서 돌아갈 가능성을 염두하고 코드를 작성하는 것이 좋다**라고 비공식 커뮤니티에서 의견을 공유했습니다. 공식적인 커뮤니티에서는 아직 잘 알려지지는 않았지만, Compose가 여러 스레드 코어를 활용하여 컴포저블을 병렬로 실행할 수 있도록 (아마도 Jetpack Compose 2.0과 같이 미래의 버전에서) 잠재적인 최적화를 고려 중인 것으로 유추해 볼 수 있습니다.

Jetpack Compose의 향후 버전은 백그라운드 스레드 풀에서 컴포저블 함수를 병렬로 실행하여 recomposition을 최적화할 수 있을 것으로 예상됩니다. 따라서, 언젠가 컴포저블 함수가 멀티 스레드 환경을 고려하지 않은 채로 작성되었다면 이로 인한 수많은 문제가 발생할 수도 있습니다. 예를 들어, 컴포저블이 ViewModel과 상호 작용하는 경우 여러 스레드에서 동시에 호출하면 개발자가 전혀 의도하지 않은 동작이 발생할 수 있습니다.

멀티스레드 환경에서 컴포저블 함수가 원활하게 작동하도록 하려면 다음 사항을 준수해야 합니다.

- **사이드 이펙트 방지(Avoid Side Effects):** 컴포저블 함수는 사이드 이펙트로부터 자유로워야 합니다. 외부 상태나 해당 범위 내 변수를 컴포저블 함수 내에서 직접적으로 수정하거나 무언가를 요청하지 않아야 합니다. 만약 사이드 이펙트를 발생시키지 않고 싶다면 사이드 이펙트 핸들러 API (`SideEffect`, `LaunchedEffect` 등)을 사용해야 합니다.

- **콜백에서 사이드 이펙트 트리거(Trigger Side Effects in Callbacks)**: 사이드 이펙트를 발생시킬 수 있는 모든 UI 관련 상호작용 작업에는 onClick과 같은 콜백을 사용해야 합니다. 이러한 콜백은 항상 UI 스레드에서 실행되어 일관성을 보장합니다.
- **가변적인 상태 피하기(Avoid Mutable Shared State)**: 상태를 나타내는 가변적인 변수(Kotlin의 var 등을 사용하여 정의된 변수)를 통해 UI를 업데이트하는 행위는 멀티 스레드 환경에서 안전하지 않기 때문에 컴포저블 함수 내에서 이와 같은 공유 변수를 사용하여 UI를 업데이트하면 안 됩니다. 상태를 올바르게 사용하기 위해서는 Compose 런타임에서 제공하는 상태 관리를 위한 전용 API인 remember와 State를 올바르게 사용해서 상태값을 관리해야 합니다.

사이드 이펙트가 없는 컴포저블 vs. 사이드 이펙트가 발생하는 컴포저블

아래의 예시는 사이드 이펙트가 없는 컴포저블 함수입니다.

그림 214. SideEffectFreeComposable.kt

```
1 @Composable
2 fun SideEffectFree(text: String) {
3     Card {
4         Text(text = "This is a side-effect-free composable: $text")
5     }
6 }
```

위의 함수는 입력값에만 의존하고 외부 상태나 내부 변수를 수정하지 않으므로 멀티 스레드 환경에서 안전합니다.

반대로 다음 예시는 사이드 이펙트가 발생합니다.

그림 215. SideEffectComposable.kt

```
1 @Composable
2 fun SideEffect() {
3     var items = 0 // 상태 정의를 위해 로컬 변수 사용
4
5     Card {
6         Text(text = "$items") // 리컴포지션이 발생할 때마다 items 값 증가
7         items++ // 이런 식으로 상태를 관리하면 안 됩니다.
8     }
9 }
```

위의 컴포저블 함수는 로컬 변수 `items`를 함수 내에서 직접적으로 수정하는데, 이렇게 되면 `recomposition`마다 수정됩니다. 따라서 값을 쓰는 행위 자체만 해도 멀티 스레드 환경에서 안전하지 않고, 개발자가 의도하지 않은 대로 동작할 가능성이 높습니다.

이미 Jetpack Compose를 어느 정도 사용해 본 개발자라면 “당연한 거 아니야? `remember`랑 `mutableStateOf`를 사용해야지!”라고 생각할 수 있는데, 근본적으로 왜 문제가 되는지 고민해보지 않으면 막상 위의 코드가 왜 잘 못된 코드인지 갑작스레 질문받는 경우 명확하게 설명하기 어렵습니다.

“컴포저블 함수 내에서 로컬 변수를 사용하여 상태 관리를 하면 안 되는 이유가 무엇인가요”라는 질문에 대해 단순히 “`remember`랑 `mutableStateOf`를 사용해야 하기 때문입니다”, 혹은 “컴포저블 함수가 여러 번 실행될 수 있으니까요” 보다 조금 더 근본적인 답변을 원하는 면접관분들이 분명히 존재한다는 것도 명심하셔야 합니다.

어떤 순서로든 실행될 수 있는 Composable 함수

Jetpack Compose는 주로 메인 스레드에서 작동하지만 멀티스레딩을 염두에 두고 설계되어 향후 최적화를 위한 여지를 남겨둡니다. 즉, 컴포저블 함수는 여러 스레드에서 실행될 수 있는 것처럼 작성되어 멀티 스레드 환경에서 안전하고 실행 순서로부터 독립적으로 유지되어야 합니다.

즉, 작성된 대로 컴포저블 함수가 순차적으로 실행되는 것처럼 보일 수 있지만 Compose는 절대 순서 그대로의 동작을 보장하지 않습니다. 컴포저블 함수가 여러 자식 컴포저블을

호출하는 경우 해당 자식은 어떤 순서로든 실행될 수 있습니다. Compose는 컨텍스트에 따라 특정 UI 컴포넌트 렌더링을 우선시하여 UI 렌더링 프로세스를 최적화할 수 있습니다.

가령, Row안에 3개의 자식 컴포저블이 있는 다음과 같은 코드를 상상해 보시길 바랍니다.

그림 216. Composable Order.kt

```
1 @Composable
2 fun TabLayout() {
3     Row {
4         FirstComposable { /* ... */ }
5         SecondComposable { /* ... */ }
6         ThirdComposable { /* ... */ }
7     }
8 }
```

위의 예제에서 FirstComposable, SecondComposable, ThirdComposable는 순차적으로 실행될 것 같지 않나요? 하지만 위의 컴포저블 함수들은 완전 무작위 랜덤으로 실행될 수 있습니다. 따라서, 사이드 이펙트를 완전히 피하기 위해서는 컴포저블이 공통적으로 가변성을 가진 전역 변수 등에 의존하면 안 됩니다. 예를 들어, FirstComposable 함수는 SecondComposable에서 값을 변경하는 전역 변수의 값에 의존한다고 가정하고 코딩하면 안 됩니다. 이렇게 동작하는 이유는 컴포저블이 예측 가능하고, 재사용 가능하며, Compose의 멀티스레드 실행 모델을 위해 향후 최적화와 호환되도록 보장하기 위함입니다.

요약

Composable 함수는 현재 단일 스레드에서 실행되더라도 향후 멀티스레드 환경에서 실행될 가능성을 염두에 두고 설계되었습니다. Compose가 발전함에 따라 정확성과 스레드 안전성을 보장하려면 사이드 이펙트 없는 컴포저블을 작성하는 것이 중요합니다. 가변적인 로컬 변수에 기반한 상태를 피하고 사이드 이펙트를 피하기 위해 UI 상호작용에는 최대한 콜백을 사용함으로써, 견고하고 재사용 가능한 UI 컴포넌트를 만들 수 있습니다. 이러한 메커니즘은 성능과 안정성을 향상시킬 뿐만 아니라, Compose가 추후 멀티 스레드 환경에서 병렬로 컴포저블을 실행할 수 있는 잠재적인 최적화를 대비하기 위한 동작으로 이해하실 수 있습니다.

Q) 5. Jetpack Compose의 안정성(stability)이란 무엇이며, 성능과 어떤 관련이 있나요?

Jetpack Compose의 안정성(Stability)은 클래스나 타입이 동일한 매개변수 입력값에 대해 일관된 결과를 생성하도록 보장하는 속성을 나타냅니다. 안정적인 클래스나 함수는 recomposition이 여러 번 발생하더라도 개발자의 의도와는 다르게 동작이 변경되지 않도록 보장합니다. 이러한 특성은 Jetpack Compose가 중복된 recomposition 없이 UI 업데이트를 효율적으로 처리하도록 하는 데 중요합니다.

Recomposition은 이미 렌더링된 UI를 업데이트하기 위해 다양한 메커니즘을 통해 트리거됩니다. 이 중 컴포저블 함수의 매개변수 안정성은 Compose 런타임과 컴파일러가 recomposition이 필요한 시기를 결정하는 방식에 있어서 중추적인 역할을 합니다.

Compose 컴파일러는 컴포저블 함수의 매개변수를 검사하고 **stable** 또는 **unstable**로 분류합니다. 이러한 분류는 Compose 런타임이 매개변수의 입력값 변경에 따라 컴포저블 함수를 다시 렌더링해야 하는지 여부를 결정하는 데 사용되므로 효율적으로 recomposition을 관리하고 앱 성능을 향상시키는 데 매우 중요합니다.

Stable vs. Unstable

매개변수가 어떻게 stable 또는 unstable로 분류되는지 궁금증을 가지실 수 있습니다. 판단의 기준은 컴포저블 함수의 매개변수 타입을 검사하는데, Compose 컴파일러는 다음과 같은 기준으로 안전성을 분류합니다.

- String을 포함한 **원시 타입(Primitive types)** 이 읽기 전용인 val로 정의 된다면, 고유한 값은 변경되지 않으므로 stable입니다.
- 값을 캡처하지 않는 람다 표현식(예: (Int) -> String)과 같은 **함수 타입(Function types)** 도 동작이 예측 가능하므로 stable로 간주됩니다.
- **클래스(Classes)**, 특히 읽기 전용이거나(val) 변경 불가능하고(immutable) stable 한 public 속성을 가진 데이터 클래스는 stable로 간주됩니다. 또한 @Stable 또는 @Immutable과 같은 안정성 어노테이션이 명시적으로 붙어있는 클래스는 stable로 간주합니다. 해당 어노테이션과 내부적인 작동 원리에 대해서는 추후에 살펴보도록 하겠습니다.

아래와 같은 데이터 클래스를 상상해 봅시다.

그림 217. Stable.kt

```

1 data class User(
2     val id: Int,
3     val name: String,
4 )

```

읽기 전용이며(val), 원시 타입의 프로퍼티들로 구성된 User 데이터 클래스는 Compose 컴파일러에 의해 stable로 간주됩니다.

반면에 Compose 컴파일러는 다음 기준에 따라 특정 매개변수 타입을 unstable로 간주합니다.

- **인터페이스 및 추상 클래스(Interfaces and Abstract Classes)**: List, Map 또는 Any와 같은 인터페이스는 컴파일 시점에 구현체의 종류를 보장할 수 없기 때문에 반드시 unstable로 간주됩니다. 이렇게 간주하는 이유는 추후에 더 자세히 살펴볼 예정입니다.
- **가변적인 프로퍼티를 가진 클래스(Clases with Mutable Properties)**: 하나 이상의 가변적이거나 본질적으로 불안정하다고 간주되는 타입의 public 프로퍼티를 단 하나라도 포함하는 클래스는 unstable로 분류됩니다.

예를 들어, 아래와 같은 데이터 클래스를 상상해 볼 수 있습니다.

그림 218. Unstable.kt

```

1 data class MutableUser(
2     val id: Int,
3     var name: String // 가변적인 프로퍼티 인해 이 클래스는 불안정하다고 간주됩니다.
4 )

```

위의 예제에서 MutableUser 클래스는 가변적인 프로퍼티(name)의 존재로 인해 unstable로 간주됩니다.

User 데이터 클래스가 모두 원시 타입의 프로퍼티들로 구성되어 있더라도 가변성을 가지는 name 프로퍼티의 존재로 인해 Compose 컴파일러는 이를 unstable로 간주합니다. 이러한 판단의 기준은 클래스 안정성이 모든 프로퍼티의 안정성을 종합적으로 평가하여 결정된다는 것을 보여줍니다. 결과적으로 단 하나의 가변적인 프로퍼티만으로도 클래스를 통째로 불안정한 타입으로 간주합니다.

Composable 함수 추론하기

Compose 컴파일러가 컴포저블 함수의 유형을 추론하고 최적화하는 원리를 잘 이해하는 것도 중요합니다. Compose 컴파일러는 Kotlin 컴파일러 플러그인 기반으로, 컴파일 타임에 개발자가 작성한 소스 코드를 분석합니다. 분석 외에도 효율적인 실행을 위해 컴포저블 함수의 고유한 요구 사항에 맞게 개발자가 작성한 원본 소스 코드를 수정합니다.

성능을 최적화하기 위해 컴파일러는 컴포저블 함수를 **Restartable**, **Skippable**, **Moveable**, **Replaceable**과 같은 유형으로 분류합니다. 이 중 **Restartable** 및 **Skippable** 유형은 recomposition을 실행시키는 논리에서 결정적인 역할을 합니다. 각 역할에 대해서 자세히 살펴보겠습니다.

- **재실행 가능(Restartable)**: Compose 컴파일러에 의해 결정되는 컴포저블 유형으로, Restartable로 분류된 함수는 recomposition 프로세스의 기반을 형성합니다. 즉, 매개변수 입력값 또는 상태가 변경되면 Compose 런타임은 UI를 업데이트하기 위해 recomposition을 위해 함수를 재호출 합니다. 대부분의 컴포저블 함수는 기본적으로 restartable로 간주되므로 런타임이 필요할 때마다 recomposition을 트리거할 수 있습니다.
- **생략 가능(Skippable)**: Skippable 함수는 스마트 recomposition에 의해 활성화된 특정 조건 하에서 recomposition을 건너뛸 수 있습니다. 이러한 recomposition 최적화는 복잡한 컴포저블 계층 구조의 최상위 노드에 있는 루트 컴포저블의 성능 향상에 중요한데, recomposition을 건너뛰면 하위 함수를 재호출 하지 않아도 되기 때문입니다. 특히 컴포저블 함수는 restartable이면서 동시에 skippable일 수 있습니다. 이는 필요할 때 recomposition을 수행할 수 있지만, 조건이 허용되면 건너뛸 수 있음을 의미합니다.

요약

Jetpack Compose의 안정성은 앱 성능에 직접적인 영향을 미치는 아주 중요한 메커니즘입니다. 안정적인 유형을 사용하고 recompositon을 최대한 생략하도록 컴포저블 함수를 설계함으로써, 더 부드럽고 최적화된 UI 사용자 경험을 제공할 수 있습니다.

실전 질문

Q) Compose 컴파일러는 매개변수가 stable인지 unstable인지 어떻게 결정하며, 이것이 recomposition 동작에 어떤 영향을 미치나요?

Q) Jetpack Compose의 `@Stable` 및 `@Immutable` 어노테이션은 무엇이며 어디에, 어떻게 사용해야 하나요?

Pro Tips for Mastery: Smart Recomposition이란?

안정성 원칙과 Compose 컴파일러가 stable 타입과 unstable 타입을 분류하는 방법을 살펴 봤는데요, 이러한 동작이 recomposition에 미치는 영향을 본질적으로 이해하는 것이 중요합니다. Compose 컴파일러는 컴포저블 함수에 전달된 매개변수의 안정성을 평가하여 Compose 런타임이 recomposition을 효율적으로 트리거 할 수 있도록 해당 정보들을 제공합니다.

클래스의 안정성이 충분히 결정되면 Compose 런타임은 **Smart recomposition** 이라고 알려진 메커니즘을 통해 해당 정보를 활용합니다. Smart recomposition은 컴파일러에서 제공하는 안정성 데이터에 의존하여 불필요한 업데이트를 선택적으로 건너뛰어 UI 성능과 응답성을 최적화합니다.

Smart Recomposition 작동 방식

새로운 매개변수 입력값이 컴포저블 함수에 전달될 때마다 Compose는 클래스의 `equals()` 메서드를 사용하여 이전 값과 비교합니다.

- **Stable 매개변수, 입력값 변화 없음:** 매개변수가 stable하고 값이 변경되지 않은 경우(`equals()`가 `true` 반환), Compose는 해당 UI 컴포넌트의 recomposition을 건너뜁니다.
- **Stable 매개변수, 입력값 바뀜:** stable 매개변수의 값이 변경된 경우(`equals()`가 `false` 반환), 런타임은 recomposition을 트리거하여 UI가 업데이트된 상태를 반영하도록 보장합니다.
- **Unstable 매개변수:** 매개변수가 unstable인 경우 Compose는 입력값 변경 여부에 관계없이 항상 recomposition을 트리거합니다.

불필요한 Recomposition을 피하는 것의 중요성

중복된 recomposition을 건너뛰면 함수를 다시 실행하고 UI 컴포넌트를 다시 그리는 데 필요한 산술적인 오버헤드가 줄어들어 UI 성능이 향상됩니다. 특히 불필요한 recomposition은 상태 관계가 복잡한 컴포넌트들이 큰 UI 계층 구조를 이루고 있을수록 더 심하게 성능을 저하시킬 수 있습니다.

요약

Jetpack Compose는 본질적으로 Smart recomposition을 통해 성능 최적화를 지원하지만, 개발자는 가능한 한 stable 클래스를 설계하고 recomposition을 최소화하는 것이 좋습니다. 안정성 원칙을 올바르게 이해하고 적용함으로써 개발자는 더 효율적이고 성능 친화적인 UI를 구현할 수 있습니다.

Pro Tips for Mastery: 안정성 어노테이션(stability annotations)의 종류와 각 차이점

안정성 어노테이션을 사용하면 개발자가 클래스의 안정성을 명시적으로 결정하여 Compose 컴파일러가 recomposition을 최적화하는 데 도움이 됩니다. [compose-runtime 라이브러리](#)²⁰에서 제공하는 두 가지 기본 어노테이션은 `@Immutable`과 `@Stable`가 있으며, 클래스의 특성에 따라 각 어노테이션은 서로 다른 목적을 가지고 사용됩니다.

`@Immutable`

`@Immutable` 어노테이션은 클래스를 **완전히 변경 불가능(fully immutable)**으로 표시하여 단순히 `val`과 같은 읽기 전용 제약 조건을 사용하는 것보다 Compose 컴파일러와의 더 강력한 약속을 체결합니다. 즉, **클래스 내의 모든 속성이** 컴파일러에 의해 변경 불가능한 것으로 처리되도록 보장합니다. 클래스에 `@Immutable` 어노테이션이 달리면 Compose 컴파일러는 해당 값이 절대 변경되지 않는다고 가정하여 해당 클래스를 매개변수로 받는 함수에 대해 recomposition을 안전하게 건너뛰어 UI 성능을 향상시킬 수 있습니다.

`@Immutable`의 주요 특징은 아래와 같습니다.

- 클래스의 모든 속성을 변경 불가능한 것으로 간주합니다.
- 일반적으로 가변적인 프로퍼티가 없는(읽기 전용 또는 `immutable`한) 클래스에 사용됩니다.
- 클래스 내 상태 변경이 없음을 보장하여 Compose 컴파일러의 부담을 덜고 UI 성능 최적화를 단순화합니다.

아래와 같은 예시에서 유용합니다.

²⁰<https://developer.android.com/jetpack/androidx/releases/compose-runtime>

그림 219. Immutable Example.kt

```

1 @Immutable
2 data class User(val id: Int, val items: List<String>)

```

위 예제에서 `id` 프로퍼티는 읽기 전용이며 원시 타입이므로 `stable`하며, `items` 프로퍼티는 `List`이므로 `unstable` 유형으로 간주됩니다. 결과적으로 전체적인 `User` 클래스는 `unstable` 하다고 간주됩니다. 하지만, 개발자는 해당 클래스가 런타임 시에 로지컬하게 변경되지 않을 것임을 확신할 수 있기 때문에, 클래스에 `@Immutable`을 마킹하고 명시적으로 `stable` 유형으로 만들 수 있습니다.

@Stable

`@Stable` 어노테이션은 속성이 `immutable`하거나 `recomposition`에 영향을 미치지 않는 제어된 가변성을 가진 클래스에 사용됩니다. 읽기 전용보다는 강력하지만 `@Immutable` 어노테이션보다는 조금 덜 엄격한 약속을 의미합니다. 즉, 클래스 자체는 `stable`로 간주되지만 해당 속성은 여전히 변경될 수 있습니다. 단, Compose 런타임에 의하여 안전한 제어되고 예측 가능한 방식으로 작동하게 됩니다.

따라서 `@Stable` 어노테이션은 `public` 속성은 읽기 전용이거나 `immutable` 하지만, 클래스 자체가 완전히 `stable`로 간주될 수 없는 클래스에 가장 적합합니다. 예를 들어, Jetpack Compose에서 `State`²¹ 인터페이스는 읽기 전용인 `value` 속성을 외부로 노출하고 있지만, 해당 프로퍼티는 `State`를 상속받는 `MutableState`²² 인스턴스를 생성할 경우, `setValue` 함수를 통해 값이 여전히 수정될 수 있습니다.

`@Stable`의 주요 특징은 아래와 같습니다.

- 일부 가변 속성을 허용하지만 클래스의 전반적인 안정성을 보장합니다.
- 특정 가변성이 내부적으로 잘 관리되는 클래스에 적합합니다.
- 완전한 불변성을 보장할 수 없는 경우 유연하게 사용할 수 있습니다.

아래와 같은 예시를 살펴볼 수 있습니다.

²¹<https://developer.android.com/reference/kotlin/androidx/compose/runtime/State>

²²<https://developer.android.com/reference/kotlin/androidx/compose/runtime/MutableState>

그림 220. StableExample.kt

```
1 @Stable
2 interface State<out T> {
3     val value: T
4 }
5
6 @Stable
7 interface MutableState<T> : State<T> {
8     override var value: T
9     operator fun component1(): T
10    operator fun component2(): (T) -> Unit
11 }
```

State 및 MutableState에서 볼 수 있듯이 MutableState에 의해 생성된 State 인스턴스는 setValue (state.value = newValue)를 사용하여 값을 바꿀수 있습니다. 이는 State 인터페이스가 보유한 value은 읽기 전용이지만, 타 구현체에 의해서 결국 값이 언제든지 바뀔 수 있음을 의미합니다.

그러나, setValue를 사용하여 동일한 값을 넣는다면, 언제나 일관된 동일한 결과가 나옵니다. 따라서, 이는 개발자가 동일한 입력값에 대해서 늘 동일한 결과가 나온다는 면등성 원리를 준수하였고, Compose는 이를 가변성이 존재하기에 완전 불변은 보장하지는 못 하지만, 해당 가변성이 내부적으로 잘 관리되며 예측이 가능하다고 판단합니다. 따라서, 위 예시에서 State 및 MutableState 인터페이스는 모두 @Stable 어노테이션으로 표기될 수 있습니다.

경우에 따라 아래 예제와 같이 @Stable로 표기된 함수를 볼 수 있습니다. 이는 함수의 반환 값이 Compose 컴파일러에 의해 stable로 간주되어 효율적인 recomposition을 보장함을 나타냅니다.

그림 221. clipScrollableContainer.kt

```

1 @Stable
2 fun Modifier.clipScrollableContainer(orientation: Orientation) =
3     then(
4         if (orientation == Orientation.Vertical) {
5             Modifier.clip(VerticalScrollableClipShape)
6         } else {
7             Modifier.clip(HorizontalScrollableClipShape)
8         }
9     )

```

@Immutable과 @Stable의 차이점

@Immutable과 @Stable 어노테이션의 구분이 처음에는 혼란스러울 수 있지만 원리만 이해하면 의외로 간단합니다. @Immutable 어노테이션은 클래스의 모든 public 속성이 완전히 변경 불가능해야 함을 보장합니다. 즉, 객체가 생성된 후 상태가 변경될 수 없습니다. 반대로 @Stable 어노테이션은 동일한 입력에 대해 일관되고 예측 가능한 결과를 생성(멱등성의 성질)하는 한 가변 속성이 있는 객체에 적합합니다. 이러한 차이점을 통해 개발자는 클래스의 안정성 및 가변성에 따라 적절한 어노테이션을 선택할 수 있습니다.

특성	@Immutable	@Stable
불변성 요구 사항	모든 프로퍼티를 불변으로 간주해야 함.	일부 프로퍼티에 대해 제어가 가능한 가변성 허용.
유즈 케이스	변경이 완전히 불가능해야 하는 데이터 모델 클래스.	UI 상태와 같이 제어할 수 있는 가변적인 클래스.
Recomposition 동작	동일한 매개변수를 가진 종속 Composable에 대해 recomposition을 완전히 건너뜀.	가변 프로퍼티가 변경되면 언제든지 recomposition이 트리거 될 수 있음.

@Immutable 어노테이션은 특히 Kotlin 데이터 클래스를 사용할 때 **I/O 작업**(가령, 네트워크 응답 또는 데이터베이스 엔티티)의 응답 모델과 같이 내부 프로퍼티들이 모두 읽기 전용인 경우에 적합합니다. 응답 모델이 인터페이스나 불안정한 클래스를 포함하는 경우 **unstable**로 간주될 수 있는데, @Immutable로 표기하면 **Compose** 컴파일러는 클래스를 불변 유형으로 처리하여 안정성을 보장하고 recomposition을 최적화합니다.

가령, 아래와 같은 예시를 살펴볼 수 있습니다.

그림 222. Immutable Example.kt

```
1 @Immutable
2 public data class User(
3     public val id: String,
4     public val nickname: String,
5     public val profileImages: List<String>,
6 )
```

위의 예시에서 List라는 unstable 유형 때문에 User 클래스 자체가 unstable로 간주될 수 있는 상황에서, 개발자는 profileImages 프로퍼티의 내부 아이템들이 바뀌지 않을 것을 로지컬하게 알고 있기 때문에 @Immutable로 표기하고 User 클래스를 stable 유형으로 만들 수 있습니다.

반대로 @Stable 어노테이션은 다양한 구현을 허용하고 내부에서 가변 상태를 포함할 수 있는 인터페이스에 적합합니다. 아래 예시가 @Stable 어노테이션을 효과적으로 사용하는 방법을 보여줍니다.

그림 223. Stable Example.kt

```
1 @Stable
2 interface ViewState<T : Result<T>> {
3     val value: T?
4     val exception: Throwable?
5
6     val hasSuccess: Boolean
7         get() = exception == null
8 }
```

@Stable 어노테이션을 사용하면 `UiState` 클래스가 `stable`로 표시되어 Smart recomposition을 통해 값이 변경되지 않은 경우는 recomposition을 생략할 수 있습니다. 이는 불필요한 recomposition을 최소화하여 UI 업데이트 효율성을 향상시킵니다.

요약

안정성 어노테이션을 적절하게 사용하면 Jetpack Compose의 UI 성능을 최적화할 수 있습니다. recomposition을 최소화하기 위해 완전히 불변 타입이라고 확신할 수 있는 클래스에는 `@Immutable`을 사용하고, 동작이 예측 가능하며 가변성을 제어할 수 있는 클래스에는 `@Stable`을 사용합니다. 이와 같이 안정성 어노테이션을 올바르게 이해하고 사용하면 UI 성능과 앱 성능을 향상시킬 수 있습니다.

Pro Tips for Mastery: `@Immutable` 대신 `@Stable`를 잘못 사용한다면?

Jetpack Compose에서 `@Stable`과 `@Immutable`이 서로 다른 목적을 가지고 있지만, 현재 Compose Compiler가 처리하는 방식에는 기능적인 차이가 없습니다. 즉, 특정 클래스에 `@Immutable` 대신 `@Stable`을 (또는 그 반대로) 실수로 사용하더라도 즉각적인 문제는 발생하지 않습니다. 그러나 애초에 이러한 구분이 따로 존재하는 이유는 무엇일까요?

한 가지 추측할 수 있는 이유는 구글의 Compose 팀이 향후 최적화나 동작 변경에 대한 가능성을 열어두고 의도적으로 목적을 구분했다는 것입니다. 지금 당장은 두 어노테이션이 큰 차이 없이 작동할 수 있지만, Compose 메커니즘이 점차 발전함에 따라 내부 처리 방식이 달라질 수 있습니다. 의도된 사용법을 명시적으로 이해하고 올바르게 사용하면, 추후에 해당 동작에 실제로 차이가 생기더라도 코드가 새 버전에서 호환될 것이고, 결국 미래에 컴파일러 동작에 대한 잠재적인 마이그레이션 문제를 최소화할 수 있습니다.

Q) 6. 안정성(stabilities) 개선을 통해 Compose 성능을 최적화한 경험이 있나요?

Jetpack Compose의 성능 최적화는 컴포저블 함수를 안정화하고 불필요한 recomposition을 최소화하는 데 달려 있습니다. 안정성은 Compose가 recomposition 중에 어떤 함수를 건너뛸지 효과적으로 결정할 수 있도록 하여 UI 렌더링 효율성을 향상시킵니다. 안정성을 보장하고 고급 컴파일러 기능을 활용하여 성능을 향상 시키기 위한 전략에 대해서 살펴보겠습니다.

Immutable Collections

List 또는 Map과 같은 읽기 전용 컬렉션은 구현 과정에서 참조값이 아닌 내부의 아이템에 변경이 발생할 수 있기 때문에, Compose 컴파일러에 의해 unstable로 처리됩니다. 가령, 아래와 같은 예시를 떠올려 볼 수 있습니다.

그림 224. MutableList Example.kt

```
1 internal var mutableUserList: MutableList<User> = mutableListOf()
2 public val userList: List<User> = mutableUserList
```

위 예제에서 userList는 본질적으로 읽기 전용인 List로 선언됩니다. 하지만, MutableList를 통해서 인스턴스화될 수 있으므로 기본 구현체인 mutableUserList의 아이템 항목들은 언제든지 변경이 가능합니다. Compose 컴파일러는 아이템 항목들이 변경 불가능한지 가능한지에 대한 여부를 결정할 수 없으므로 recomposition의 정확성을 보장하기 위해 해당 인스턴스를 unstable로 처리합니다. 만약 내부 아이템 항목에 변화가 발생했고, 정상적으로 recomposition이 발생해야 하는데 발생하지 않으면, 개발자의 의도와는 달리 UI가 제 때 업데이트 되지 않는 문제가 발생할 수 있습니다.

안정성을 보장하기 위해 본질적으로 stable한 [kotlinx.collections.immutable](#)²³ 또는 Guava의 [immutable](#) 컬렉션²⁴을 사용할 수 있습니다. 해당 라이브러리는 ImmutableList 및 ImmutableSet과 같은 불가변성을 특징으로 하여 컬렉션을 읽기 전용으로 제한하고, 효율적으로 recomposition을 수행할 수 있도록 합니다.

 Pro Tips for Mastery: ImmutableList²⁵ 코드를 살펴보면 ImmutableList로 결국 인터페이스로 정의되어 있는데 왜 ImmutableList 는 stable이고 List는 unstable 일까요? 이러한 고민을 하면서 책을 읽으셨다면 너무 잘하고 계신 겁니다. 더 나은 개발자가 되기 위해서는 당연한 것에 늘 의문을 가져야 합니다. ImmutableList가 stable로 처리되는 이유는 Compose 컴파일러 내부에서 해당 패키지를 stable로 처리하도록 하드코딩 되어있기 때문입니다. 뿐만 아니라 Dagger의 Lazy 등 또한 stable로 처리되는데, 이에 대해서 살펴보고 싶다면 Compose 컴파일러 라이브러리의 [KnownStableConstructs.kt](#)²⁶ 파일을 살펴보실 수 있습니다.. 코드에서 볼 수 있듯이 Compose 컴파일러는 stable로 처리해야 하는 클래스의

²³<https://github.com/Kotlin/kotlinx.collections.immutable>

²⁴<https://github.com/google/guava/wiki/ImmutableCollectionsExplained>

패키지 이름 목록을 명시적으로 하드 코딩하여 관리합니다.

Lambda Stability

람다 표현식을 매개변수로 받은 컴포저블 함수는 어떨까요? 컴포즈 컴파일러는 람다 함수를 매개변수로 받는 경우 반드시 stable로 처리합니다. 하지만 내부적으로 람다가 람다식 스코프의 외부 변수를 참조하는지 여부에 따라 다르게 처리합니다.

- **값을 캡처하지 않는 람다(Non-Capturing Lambdas)**: 외부 변수에 의존하지 않는 람다(값을 캡처하지 않는다고 표현합니다)는 싱글톤으로 최적화되어 stable로 간주되고, recomposition을 트리거 하지 않습니다.
- **값을 캡처하는 람다(Capturing Lambdas)**: 외부 변수에 의존하는 람다(값을 캡처한다고 표현합니다)는 동적으로 변경 사항에 응답하기 위해 remember를 사용하여 메모이제이션됩니다. 변경 사항을 관찰하기 위해 외부 변수가 remember API의 key 매개변수로 전달되고, 이는 결국 동일한 key 값에 대해서 동일한 리턴값이 나오는 것을 의미하는 것이기 때문에 멱등성의 원리를 준수하여 stable로 간주됩니다. 따라서 recomposition의 안정성과 일관성을 모두를 보장합니다.

클로저 내에서 값을 캡처한다는 것은 람다 표현식이 클로저 외부 범위의 변수에 의존한다는 의미입니다. 람다가 외부 변수에 의존하지 않으면 아래 예제와 같이 **캡처하지 않는 (non-capturing)** 것으로 간주됩니다.

그림 225. Non-Capturing Lambda.kt

```
1 modifier.clickable {
2     Log.d("Log", "This Lambda doesn't capture any values")
3 }
```

²⁵<https://github.com/Kotlin/kotlinx.collections.immutable/blob/2d31e2fcc30136a05db7f4ddf0a34bf6c2ebac51/core/commonMain/src/ImmutableList.kt#L20>

²⁶<https://github.com/JetBrains/kotlin/blob/eadcce82e781d7be850118e82333893ab7cf10a9/plugins/compose/compiler-hosted/src/main/java/androidx/compose/compiler/plugins/kotlin/analysis/KnownStableConstructs.kt#L48>

람다 매개변수가 값을 캡처하지 않으면 Kotlin은 람다를 **싱글톤**으로 취급하여 최적화하고 stable로 간주합니다. 만약 람다가 클로저 범위 외부의 변수에 의존하는 경우 아래 예제와 같이 **캡처하는(capturing)** 것으로 판단하고, Compose 컴파일러가 컴파일 타임에 코드를 변형하여 remember 등을 삽입하고 해당 람다 매개변수가 stable하도록 만듭니다.

그림 226. Capturing Lambda.kt

```

1 var sum = 0
2 ints.filter { it > 0 }.forEach {
3     sum += it // 외부 변수 'sum'을 캡처
4 }
```

래퍼 클래스 (Wrapper Classes)

제어할 수 없는 불안정한 클래스(가령, 서드파티 라이브러리에서 제공되는 클래스)의 경우 아래 예제와 같이 안정성 어노테이션으로 마크한 래퍼 클래스를 만들 수 있습니다.

그림 227. Wrapper Classes.kt

```

1 @Immutable
2 data class ImmutableList(
3     val user: List<User>
4 )
```

위의 예시에서 List는 여전히 unstable이지만, Wrapper(ImmutableUserList)는 immutable로 간주되므로 ImmutableList 클래스 자체는 stable하게 간주됩니다. 그런 다음 아래 예제와 같이 해당 래퍼 클래스를 컴포저블 함수의 매개변수 유형으로 사용할 수 있습니다.

그림 228. Wrapper Classes.kt

```

1 @Composable
2 fun UserAvatars(
3     modifier: Modifier,
4     userList: ImmutableList, // 안정적인 Wrapper 타입 사용
5 )
```

안정성 구성 파일 (Stability Configuration File)

Compose Compiler 1.5.5에 공개된 [안정성 구성 파일](#)²⁷을 사용하면 stable로 간주하고 싶은 클래스를 개발자가 임의로 지정할 수 있습니다.

- stable로 처리할 클래스의 패키지 명을 나열하는 compose_compiler_config.conf 파일을 만듭니다.
- Compose 컴파일러가 해당 파일에 정의된 패키지 목록에 대해 recomposition을 건너뛸 수 있도록 build.gradle.kts에서 추가적으로 설정이 필요합니다.
- 해당 기능은 특히 서드파티 클래스나 커스텀 타입을 stable하게 만드는 데 유용합니다. 이를 활성화하면 래퍼 클래스를 수동으로 만들 필요 없이 프로젝트 내에서 전역적으로 특정 클래스를 **stable**로 지정할 수 있습니다.

Strong Skipping Mode

Compose Compiler 1.5.4에서 소개된 [Strong Skipping Mode](#)²⁸는 불안정한 매개변수를 포함하더라도, restartable로 분류된 컴포저블 함수에 대해 recomposition 생략을 활성화 합니다.

- stable 매개변수가 있는 컴포저블 함수는 객체 동등성을 사용하여 값을 비교하는 반면, unstable 매개변수는 인스턴스 동등성을 사용하여 비교됩니다.
- 일단 해당 기능이 활성화되면 모든 컴포저블 함수에 대해서 적용되기 때문에, 특정한 함수를 제외하려면 @NonSkippableComposable 어노테이션을 사용하여 열외 시킬 수 있습니다.

요약

매개변수 안정화, immutable 컬렉션 사용, 람다 최적화, 래퍼 클래스 사용, Strong Skipping Mode와 같은 기능들을 적극적으로 활용하여 Jetpack Compose UI의 성능을 크게 향상시킬

²⁷<https://developer.android.com/develop/ui/compose/performance/stability/fix#configuration-file>

²⁸<https://developer.android.com/develop/ui/compose/performance/stability/strongskipping>

수 있습니다. 이러한 전략은 불필요한 recomposition을 줄이고 UI 응답성을 개선하며 더 부드러운 상호 작용을 보장합니다. Compose UI 및 앱 성능 최적화와 관련하여 추가적인 학습을 원하신다면, [Optimize App Performance by Mastering Stability in Jetpack Compose²⁹](#) 및 [GitHub: compose-performance³⁰](#) 리포지토리를 참고하세요.

실전 질문

Q) List를 매개변수로 받는 컴포저블 함수에서 불필요하게 발생하는 recomposition을 어떻게 최적화하시겠습니까?

Q) 앱에서 recomposition 효율성을 개선하기 위해 시도해 본 전략 혹은 Compose 컴파일러 기능 중에 어떤 게 있나요?

Q) 7. 컴포지션(composition)이란 무엇이며 어떻게 생성하나요?

컴포지션(Composition)³¹은 컴포즈 컴파일러가 해석한 UI에 대한 정보를 실질적인 UI로 구체화하는 하나의 단계입니다. UI를 컴포저블의 트리 구조로 구성하며, **Composer**를 활용하여 트리를 동적으로 생성하고 관리합니다. 컴포지션은 상태를 기록하고 UI를 효율적으로 업데이트하기 위해 노드 트리에 필요한 변경 사항을 적용하며, 변경 사항을 적용하고 UI를 업데이트 하는 과정을 **recomposition**이라고 합니다. 본질적으로 컴포지션은 Jetpack Compose의 중추적인 역할을 하여 런타임에 UI 구조와 컴포저블 함수의 상태를 모두 관리합니다.

컴포지션 생성하기

컴포지션은 컴포저블 함수를 화면에 렌더링할 수 있게 UI 계층 구조로 변환하는 프로세스입니다. 이러한 과정은 Jetpack Compose가 작동하는 방식의 핵심으로, 런타임에서 상태의 변경 사항을 추적하고 UI를 효율적으로 업데이트할 수 있도록 합니다. 컴포지션을 생성하고 관리하는 방법은 다음과 같습니다.

²⁹<https://proandroiddev.com/optimize-app-performance-by-mastering-stability-in-jetpack-compose-69f40a8c785d>

³⁰<https://github.com/skydoves/compose-performance>

³¹<https://developer.android.com/develop/ui/compose/lifecycle#composition-anatomy>

ComponentActivity.setContent() 함수

컴포지션을 생성하는 가장 일반적인 방법은 ComponentActivity 또는 ComposeView에서 제공하는 setContent 함수를 사용하는 것입니다. 이 함수는 컴포지션을 초기화하고 그 안에 표시될 콘텐츠를 정의합니다.

그림 229. setContent.kt

```
1 import androidx.activity.ComponentActivity
2 import androidx.activity.compose.setContent // setContent 임포트 경로 수정
3 import androidx.compose.runtime.Composable
4
5 class MainActivity : ComponentActivity() {
6     override fun onCreate(savedInstanceState: Bundle?) {
7         super.onCreate(savedInstanceState)
8
9         setContent { // Activity의 content를 Compose UI로 설정
10             MyComposableContent()
11         }
12     }
13 }
14
15 @Composable
16 fun MyComposableContent() {
17     // Compose UI 컴포넌트로 레이아웃 구현
18 }
```

위 예제에서 볼 수 있듯이 setContent는 컴포저블 함수를 렌더링하고 **컴포지션**을 시작하는 역할을 하며, Compose UI의 **진입점(entry point)** 이 됩니다.

ComponentActivity.setContent를 더 자세히 살펴보자면, 아래와 같이 내부적으로 ComposeView에 의존하며 ComposeView.setContent를 호출하여 **컴포지션**을 초기화하고 생성한다는 것을 알 수 있습니다.

그림 230. ComponentActivity.setContent.kt

```

1 public fun ComponentActivity.setContent(
2     parent: CompositionContext? = null,
3     content: @Composable () -> Unit
4 ) {
5     // 이미 윈도우에 연결된 ComposeView 찾기
6     val existingComposeView = window.decorView
7         .findViewById<ViewGroup>(android.R.id.content)
8         .getChildAt(0) as? ComposeView
9
10    if (existingComposeView != null) with(existingComposeView) {
11        // 기존 View 재사용
12        setParentCompositionContext(parent)
13        setContent(content)
14    } else ComposeView(this).apply {
15        // 새 ComposeView 생성 및 설정
16        setParentCompositionContext(parent)
17        setContent(content)
18        // ViewTree 관련 Owner 설정
19        setOwners()
20        // Activity의 ContentView로 설정
21        setContentView(this, DefaultActivityContentLayoutParams)
22    }
23 }
```

단계별로 살펴보면, 함수는 먼저 Activity의 뷰 계층 구조에서 기존 ComposeView를 검색하고, 없으면 새 인스턴스를 생성합니다.

```

1 val existingComposeView =
2     window.decorView.findViewById<ViewGroup>(android.R.id.content).getChildAt(0) as?
3         ComposeView
```

이 단계는 Activity에 ComposeView가 이미 있는 경우 불필요하게 새 인스턴스를 생성하는 대신

재사용할 수 있도록 보장합니다. 기존 ComposeView가 발견되면 새 부모 CompositionContext(제공된 경우)와 새 컴포저를 콘텐츠로 업데이트됩니다.

```
1 if (existingComposeView != null)
2     with(existingComposeView) {
3         setParentCompositionContext(parent)
4         setContent(content)
5     }
```

ComposeView가 없으면 함수는 새 인스턴스를 생성하고 Activity에 레이아웃을 추가하기 전에 구성 작업을 완료합니다.

```
1 ComposeView(this).apply {
2     setParentCompositionContext(parent)
3     setContent(content)
4     setOwners()
5     setContentView(this, DefaultActivityContentLayoutParams)
6 }
```

위 코드에서 볼 수 있듯이 Compose UI는 본질적으로 전통적인 **View** 시스템, 특히 ComposeView 내에서 렌더링 됩니다. 이는 Jetpack Compose와 안드로이드 SDK에서 제공하는 렌더링 시스템 간의 **다리(bridge)** 역할을 한다고 볼 수 있습니다. 따라서, Jetpack Compose는 기존의 전통적인 뷰 시스템과 '완전히 관련 없는 새로운 프레임워크다'라고 생각하면 곤란합니다. Compose도 보시는 것처럼 결국은 기존의 View 시스템 위에서 동작하는 것을 기반으로 합니다. 그렇기 때문에 Compose UI와 관련된 라이브러리 의존성만 추가하여도 기존의 안드로이드 프로젝트에서 원활하게 통합되어 동작할 수 있고, 다양한 기존 View와의 마이그레이션이 가능한 것입니다.

XML 레이아웃에 Compose 포함시키기

금방 살펴보았듯이, Compose를 전통적인 안드로이드 View 계층 구조에 통합하려면 ComposeView를 사용할 수 있습니다. 이를 통해 XML에 정의된 레이아웃 내에서도 **컴포지션**을 생성할 수 있으며, setContent API가 내부적으로 작동하는 원리와 유사하게 작동합니다.

그림 231. ComposeView.kt

```
1 import androidx.compose.ui.platform.ComposeView
2
3 // XML이 아닌 Kotlin 코드로 직접 ComposeView 생성 및 설정
4 val composeView = ComposeView(context).apply {
5     setContent {
6         MyComposableContent()
7     }
8 }
9 // 기존 ViewGroup에 추가
10 // viewGroup.addView(composeView)
```

ComposeView를 기존 ViewGroup에 추가하거나 <androidx.compose.ui.platform.ComposeView> 태그를 사용하여 XML 레이아웃에서 직접 사용할 수도 있습니다.

요약

컴포지션은 컴포저블 함수를 렌더링하여 Compose UI의 계층 구조를 구축하고 관리하는 프로세스입니다. 컴포지션을 생성하려면 컴포저블 함수로 UI를 정의하고, ComponentActivity.setContent 또는 ComposeView 기반의 ComposeView.setContent와 같은 메커니즘을 사용하여 컴포지션을 초기화해야 합니다.

실전 질문

Q) ComposeView는 전통적인 View와 Compose UI 시스템을 어떻게 연결하며, 언제 사용되나요? 그리고, ComposeView와 컴포지션의 관계는 어떻게 되나요?

Q) 8. XML 기반 프로젝트를 Jetpack Compose로 마이그레이션하는 전략에 대해서 설명해 주세요.

구글의 Android 팀에서는 XML 기반 UI에서 Jetpack Compose로 마이그레이션 하기 위한 다양한 전략들을 제공해 주고 있습니다. 하지만, 마이그레이션 프로세스는 작업에 들어가는 비용과 기존 코드와의 충돌을 최소화하기 위해 신중한 계획이 필요합니다. 먼저, 모범 사례 및 공식 가이드라인을 기반으로 한 주요 전략에 대해서 살펴보겠습니다.

1. 점진적 마이그레이션 (Incremental Migration)

점진적 마이그레이션은 기존의 XML 기반의 코드를 점진적으로 Compose 기반으로 바꾸어 동일한 프로젝트에서 XML과 Compose가 공존하다가 최종적으로 모두 Compose 기반의 코드로 바꾸는 것입니다.

- **XML에 Compose를 함께 사용하기:** 기존 XML 레이아웃에 Compose 콘텐츠를 포함하려면 ComposeView를 사용하면 됩니다. 이러한 접근 방식은 XML 기반 화면을 부분적으로 Compose로 마이그레이션 하려는 경우 유용합니다.

그림 232. Embedding Compose in XML Layout

```
1 <androidx.compose.ui.platform.ComposeView  
2     android:id="@+id/compose_view"  
3     android:layout_width="match_parent"  
4     android:layout_height="wrap_content" />
```

그림 233. Setting Content for ComposeView

```
1 findViewById<ComposeView>(R.id.compose_view).setContent {  
2     Greeting("Hello Compose!")  
3 }
```

- **Compose에 XML 포함하기:** 반대로, 컴포저블 함수 내에 XML 기반 뷰를 포함하려면 AndroidView를 사용하면 됩니다. 이는 마이그레이션 중에 레거시 컴포넌트의 기능을 유지하는 데 유용합니다. 가령, ExoPlayer나 YouTubePlayerView와 같이 아직 Jetpack Compose UI를 공식적으로 지원하지 않는 컴포넌트를 Jetpack Compose에서 사용하는 경우가 있습니다.

그림 234. Embedding XML in Compose

```
1 @Composable
2 fun LegacyViewComposable() {
3     AndroidView(factory = { context ->
4         // LayoutInflater를 사용하여 XML 레이아웃 인플레이트
5         LayoutInflater.from(context).inflate(R.layout.legacy_view, null)
6     })
7 }
```

이와 같은 상호호환 전략은 유연한 마이그레이션을 허용하고 기존 기능이 예상하지 못하게 잘못 작동하는 위험을 최소화합니다.

2. 화면별 마이그레이션 (Screen-by-Screen Migration)

가장 실용적인 전략은 화면 구성이 간단하여 Compose 마이그레이션이 쉬운 화면부터 시작하여, 한 번에 한 화면씩 마이그레이션 하는 것입니다.

1. Compose로 마이그레이션 하기 쉽거나, 마이그레이션 하면 즉각적인 이점이 생길만한 화면을 선별합니다.
2. 해당 화면의 레이아웃 구조를 Compose를 사용하는 형태로 완전히 다시 설계합니다.
3. Compose의 상태 기반 아키텍처를 활용하여 XML 레이아웃을 컴포저블 함수로 교체합니다.

이 전략을 통해 개발자는 점진적으로 Compose를 학습하면서 중요한 기능 개발에 집중할 수 있습니다. 화면을 마이그레이션 할 때, 팀은 화면 단위로 Jetpack Compose를 마이그레이션 한다라는 명확한 목표를 가지므로 전체적인 프로세스를 관리하기가 더 쉬워집니다.

3. 점진적 컴포넌트 마이그레이션 (Gradual Component Migration)

화면 단위로 마이그레이션 하는 것 역시 복잡한 화면의 경우는 한 번에 Compose로 교체하기 까다로울 수 있습니다. 이런 경우 텍스트, 버튼 또는 커스텀 컴포넌트와 같은 개별적인 재사용 가능한 컴포넌트, 또는 전체 디자인 시스템부터 마이그레이션 하는 전략을 선택해 볼 수 있습니다. 아래와 같이 단계별로 전략을 짜보는 것도 좋습니다.

1. 자주 사용되는 View 기반의 UI 컴포넌트 또는 디자인 시스템의 일부를 선별합니다.
2. 선별한 컴포넌트를 컴포저를 함수로 다시 만듭니다.
3. 해당 컴포넌트를 사용 중인 XML 레이아웃에서 기존 컴포넌트를 새 컴포저를 컴포넌트로 교체합니다.

이와 같은 전략으로 컴포넌트를 개별적으로 View에서 Compose로 전환하거나, 디자인 시스템을 Compose로 다시 구현하여 화면을 **부분적으로 마이그레이션**할 수 있습니다. 결과적으로, 프로젝트의 여러 화면에서 재사용되고 있는 UI 컴포넌트에 대한 일관성을 보장하고 마이그레이션 비용을 줄일 수 있습니다.

4. 전체 재작성 (Full Rewrite)

광범위한 레거시 코드가 있거나 오래된 UI 라이브러리에 복잡한 의존관계를 가지고 있는 경우, 그냥 처음부터 프로젝트 전체를 새로 Jetpack Compose로 작성하는 것이 최선의 선택이 될 수도 있습니다.

1. Compose에서 테마, 레이아웃 및 커스텀 컴포넌트, 혹은 디자인 시스템 전체를 다시 개발합니다.
2. 전체 UI 스택에 Compose를 활용하여 XML로 구현했던 앱과 동작 형태를 똑같이 가져갑니다.
3. 하는 김에 Compose 활용에 최적화된 MVVI 또는 MVVM과 같은 최신 디자인 패턴을 채택하여 앱 전체 아키텍처를 재정의 해볼 수도 있습니다.

사실 마이그레이션 전략이라고 하기에는 단순히 프로젝트를 완전히 처음부터 새로 개발하는 경우라서, 개발팀은 **상당한 시간과 자원이** 요구되며 애플리케이션은 QA팀과 함께 철저한 **테스트 및 검증**을 거쳐야 합니다. 오히려 동일한 기능을 다른 테크 스펙을 통해 새롭게 개발해야 하는 상황을 비즈니스 적으로 프로덕트 매니저나, QA 팀에 납득시키는 것이 가장 어려울 수 있습니다. 특히나, 팀이 아직 Jetpack Compose에 익숙하지 않은 경우 몇 번 난관에 부딪치다 보면 다시 View 기반의 프로젝트로 돌아갈 가능성도 있고, 개발에 들어가는 시간이 기대했던 것보다 더 요구될 수 있으므로 신중한 계획이 더욱 중요합니다.

5. 라이브러리에 대한 상호 운용성 활용 (Leverage Interoperability for Libraries)

Compose와 XML의 상호 운용성은 아직 Compose를 채택하지 않은 라이브러리를 사용하고 있는 경우도 신경을 써야 합니다. 이전에 살펴보았던 ExoPlayer 등의 예시가 해당됩니다.

이런 경우, [ComposeView³²](#) 또는 [AndroidView³³](#)를 사용하여 Compose UI를 지원하지 않는 UI 라이브러리 또한 컴포저를 함수에 원활하게 통합할 수 있습니다.

6. 마이그레이션 중 테스트 및 모니터링 (Test and Monitor During Migration)

마이그레이션 중에는 앱이 예상대로 작동하는지 확인하기 위해 테스트가 필수적입니다. Compose의 테스트 라이브러리³⁴를 활용하여 다시 작성한 컴포저를 함수의 동작을 검증할 수 있습니다. 또한 성능 프로파일링을 수행하여 Compose를 기준 XML 구현과 비교하고 최소한 성능적인 면에서 동등하거나 향상된 사용자 경험을 보장하는 것이 중요합니다. Compose로 마이그레이션 하기 위해 시간과 리소스를 사용하였으나, 앱 성능이 오히려 더 떨어진다면 팀원들의 신뢰를 잃을 수도 있기 때문에 특히나 신경을 써야 합니다.

요약

Jetpack Compose의 마이그레이션 전략은 프로젝트의 규모 및 팀의 상황에 따라 달라질 수 있습니다. 따라서 각 상황에 맞는 올바른 전략을 선택하는 것이 중요합니다. Compose의 상호 운용성³⁵을 활용하여 Compose와 XML을 동시에 유지하면서, 점진적인 마이그레이션 전략을 세워볼 수도 있습니다. 그리고 마이그레이션 전략과는 관계없이 이 과정에서의 핵심은 개발 전체 프로세스에 걸쳐서 유지 보수의 향상, 개발자의 생산성, 더 나은 사용자 UI 경험과 같은 Compose의 이점을 극대화하면서도 마이그레이션 과정을 통해 발생할 수 있는 위험을 최소화하는 것입니다. 마이그레이션과 관련하여 좋은 전략이나 케이스 스터디를 확인하고 싶으시면 아래 문서와 블로그 포스트를 통해서 더 깊게 알아보실 수 있습니다.

- [Android Official Documentation: Migration strategy³⁶](#)
- [Android Developers: Migrating Sunflower to Jetpack Compose³⁷](#)

³²<https://developer.android.com/develop/ui/compose/migrate/interoperability-apis/compose-in-views>

³³<https://developer.android.com/develop/ui/compose/migrate/interoperability-apis/views-in-compose>

³⁴<https://developer.android.com/develop/ui/compose/testing>

³⁵<https://developer.android.com/develop/ui/compose/migrate/interoperability-apis>

³⁶<https://developer.android.com/develop/ui/compose/migrate/strategy>

³⁷<https://medium.com/androiddevelopers/migrating-sunflower-to-jetpack-compose-f840fa3b9985>

- [Android Developers: Migrating to Jetpack Compose — an interop love story \[part 1\]](#)³⁸
- [Android Developers: Migrating to Jetpack Compose — an interop love story \[part 2\]](#)³⁹

실전 질문

Q) XML에서 Compose로 마이그레이션 할 때 기존 View 기반 레이아웃 내에서 부분적으로 컴포저를 함수로 마이그레이션 하는 전략에 대해서 설명해 주시고, 이러한 접근 방식이 가장 유용한 시나리오는 무엇인가요?

Q) 안드로이드 앱을 화면별로 Jetpack Compose로 마이그레이션 하는 것과 컴포넌트별로 마이그레이션 하는 것의 각 장단점을 경험에 기반하여 말씀해 주세요.

 **Pro Tips for Mastery: XML과 Jetpack Compose를 동시에 사용하면 앱 (APK/AAB) 사이즈에 영향을 미치나요?**

マイグ레이션 중 XML과 Jetpack Compose를 함께 사용하면 앱 크기에 영향을 미칠 수 있지만, 그 효과는 비교적 미미합니다. Jetpack Compose는 일종의 **라이브러리의 집합**이며, **R8 최적화**를 사용하는 경우 일반적으로 Compose UI 라이브러리들을 여럿 추가하는 것으로는 **최대 약 2MB** 정도밖에 늘어나지 않습니다. 메모리 사용량이 아주 민감한 기기에서는 문제가 될 수 있지만, 최신 안드로이드 기기는 일반적으로 심각한 성능 문제나 메모리 부족 문제로 이어질 정도는 아니기 때문에 크게 신경 쓸 정도는 아닙니다.

Q) 9. Compose 성능 테스트를 항상 릴리스 모드(release mode)에서 해야 하는 이유는 무엇인가요?

Jetpack Compose 성능을 테스트할 때는 항상 **R8 최적화**⁴⁰가 활성화된 릴리스 모드에서 앱을 실행해야 합니다. 디버그 모드는 UI 코드에 대한 불필요한 해석, Just-In-Time (JIT) 컴파

³⁸<https://medium.com/androiddevelopers/migrating-to-jetpack-compose-an-interop-love-story-part-1-3693ca3ae981>

³⁹<https://medium.com/androiddevelopers/migrating-to-jetpack-compose-an-interop-love-story-part-2-370fdd978c33>

⁴⁰<https://developer.android.com/build/shrink-code#enable>

일⁴¹, Live Edit Literals와 같은 개발자 도구와 관련된 것들이 추가 오버헤드를 유발하며, 결국 앱 성능에 영향을 미치고 최종적인 사용자 경험을 똑같이 재현하지 못합니다. 따라서 일반적으로 Jetpack Compose로 개발된 앱은 디버그 앱이 릴리스 앱보다 월등히 느린 경우가 많습니다. 이것 때문에 많은 개발자들이 Jetpack Compose의 성능 자체가 전통적인 View 시스템 보다 많이 느리다고 오해하는 경우가 많습니다.

디버그 모드가 Compose에 미치는 영향

Compose는 라이브러리(“unbundled”) 형태로 제공되므로, 디버그 앱을 실행할 때 런타임에 의하여 해석되고 컴파일됩니다. View 시스템의 경우는 안드로이드 OS와 함께 번들로 제공되어 미리 컴파일되는 반면에, Compose 라이브러리들은 디버그 모드에 Compose와 관련한 추가적인 해석 및 JIT 컴파일 오버헤드를 발생시킵니다. 이는 최적화된 릴리스 빌드와 비교하여 상당한 성능 격차를 발생시켜 체감상 훨씬 앱이 느리다고 느껴질 수 있습니다.

안드로이드 팀에 의하면, 전통적인 View와 관련한 코드들은 안드로이드 프레임워크에 이미 통합되어 있기 때문에 빠르게 컴파일되고, 최적화된 형태로 돌아갑니다. 반면 Compose 라이브러리는 Lazy List에서 관리하는 작은 단위의 컴포넌트 뿐만 아니라, 전체 UI 스택을 debuggable한 형태로 실행하기 때문에 디버그 모드에서의 추가적인 오버헤드가 발생합니다.

Live Edit Literals 및 개발자 도구

디버그 빌드는 런타임 업데이트를 지원하기 위해 상수를 getter 함수로 대체하는 [Live Edit Literals](#)⁴²와 같은 개발자 기능을 활성화합니다. 이는 추가적인 산술 비용을 유발하고 최적화를 방해하여, 디버그 모드에서 recomposition 및 렌더링을 더 느리게 합니다.

⁴¹Just-In-Time (JIT) 컴파일은 바이트코드가 실행 직전에 동적으로 기계 코드로 변환되는 런타임 프로세스입니다. 이를 통해 런타임 환경은 실제 실행 패턴을 기반으로 코드를 최적화하여 자주 사용되는 코드 경로의 성능을 향상시킬 수 있습니다.

⁴²<https://developer.android.com/develop/ui/compose/tooling#live-edit-literals>

릴리스 모드에서의 R8 최적화

R8⁴³은 람다 그룹화, 소스 정보 생략, 상수 폴딩, 인터페이스 호출을 더 빠른 정적 호출로 변환하는 등 최적화를 통해 릴리스 빌드의 성능을 크게 향상시킵니다. 이러한 최적화는 앱 시작 시간을 단축하고 메모리 사용량을 줄이며 런타임 실행을 간소화합니다.

안드로이드 팀에 따르면, Compose는 R8 최적화로부터 상당한 이점을 얻습니다. 기본적으로 R8 최적화를 활성화 하는 것만으로도 앱 시작 성능이 약 75%, 프레임 렌더링 성능이 약 60% 향상됩니다. R8은 내부적으로 다양한 최적화를 수행하지만, 아래는 Compose 코드에 가장 큰 영향을 미치는 일부 세부 정보입니다.

- 람다 그룹화: 유사한 람다 구현을 그룹화하여 람다 표현식을 최적화하고 메서드 오버헤드를 줄입니다.
- 소스 정보 생략: 디버그 및 소스 메타데이터를 제거하여 APK 크기를 최소화합니다.
- 상수 폴딩: 컴파일 시 상수 표현식을 단순화하여 런타임 효율성을 향상합니다.
- 인터페이스 호출을 정적 호출로 변환: 동적 인터페이스 호출을 더 빠른 정적 메서드 호출로 대체하여 실행 속도를 크게 향상시킵니다.

Baseline Profiles가 중요한 이유

릴리스 모드에서 Jetpack Compose의 성능을 더욱 향상시키기 위해서는 [Baseline Profiles](#)⁴⁴을 함께 사용하는 것이 좋습니다. Baseline Profiles는 중요한 Compose 메서드를 사전에 컴파일하여 앱 시작 중 런타임 해석 및 JIT 컴파일을 피합니다. 보통 디버그 빌드의 경우 Baseline Profiles를 반영되지 않기 때문에, 실제 릴리스 앱과의 실행 성능을 비교하면 체감될 정도로 월등히 개선되는 것을 느낄 수 있습니다. 자세한 내용은 [Improve Your Android App Performance With Baseline Profiles](#)⁴⁵를 통해 학습하시는 것을 권장합니다.

⁴³R8 최적화는 사용되지 않는 코드 제거, 메서드 인라인ning, 상수 폴딩 및 람다 그룹화와 같은 고급 최적화 적용을 통해 APK 크기를 줄이고 런타임 성능을 향상시키는 안드로이드용 코드 축소 및 최적화 도구입니다. 또한 보안 강화를 위해 코드를 난독화하고 실행을 위해 바이트코드를 더 효율적인 형태로 변환합니다.

⁴⁴<https://developer.android.com/topic/performance/baselineprofiles/overview>

⁴⁵<https://medium.com/proandroiddev/improve-your-android-app-performance-with-baseline-profiles-297f388082e6>

실제 테스트 권장 사항

성능을 정확하게 평가하려면 항상 R8 및 Baseline Profiles가 활성화된 릴리스 모드에서 Compose 앱을 테스트해야 합니다. 앱 시작 및 런타임 성능을 측정하기 위해 [Macrobenchmark⁴⁶](#)와 같은 도구를 활용해 보는 것도 좋습니다. Macrobenchmark로 실제 성능 병목 현상을 살펴보고, 문제가 있는 부분을 명확하게 개선함으로써 최종 사용자에게 원활한 경험을 제공할 수 있습니다.

요약

디버그 모드는 Jetpack Compose의 실제 성능을 왜곡하는 상당한 오버헤드를 유발하기 때문에 실제 사용자가 경험하게 될 릴리스 모드에서의 테스트도 필수적으로 진행해야 합니다. R8 최적화 및 Baseline Profiles는 Compose 앱이 효율적으로 실행되도록 보장하며, Compose 라이브러리의 단점인 Just-In-Time (JIT) 컴파일 방식을 어느 정도 개선해 줍니다. 더 자세한 내용을 살펴보고자 하신다면 [Android Developer Blog: Why should you always test Compose performance in release?](#)⁴⁷를 참조하세요.

실전 질문

Q) R8은 Jetpack Compose 성능 최적화에서 어떤 역할을 하며, 릴리스 빌드에서 어떤 사항을 개선하는지 구체적으로 설명해 주세요

Q) 10. Jetpack Compose에서 자주 사용하시는 Kotlin 관용구(idioms)에 대해서 말씀해 주세요.

Jetpack Compose는 대부분의 시스템이 Kotlin으로 개발되었기 때문에, Kotlin에서 지원하는 언어적 기능을 활용하여 더 표현력 있고 효율적인 UI 개발 경험을 제공합니다. 간결하고 읽기 쉬운 관용적인 Compose 코드를 작성하려면 일반적으로 사용되는 Kotlin 관용구를 잘 알고 있는 것이 깔끔한 코드를 작성하는데 도움이 되고, 전반적으로 코드나 작동 방식에 대한 이해를 돋습니다.

⁴⁶<https://developer.android.com/topic/performance/benchmarking/macrobenchmark-overview>

⁴⁷<https://medium.com/androiddevelopers/why-should-you-always-test-compose-performance-in-release-4168dd0f2c71>

기본 매개변수 (Default Arguments)

Kotlin은 함수 매개변수에 기본값을 지정할 수 있어 여러 함수에 대해 오버로드의 필요성을 줄입니다. 이는 Compose에서 API 사용을 단순화하기 위해 자주 사용되는 기법입니다. 가령, Text 컴포저블은 선택적으로 요구되는 매개변수에 대해 기본값을 제공하여, 최소한의 매개변수만 사용하여 함수를 호출할 수 있도록 허용하면서도 풍부한 커스텀을 지원합니다. 특히 컴포저블 함수는 매개변수의 이름을 지정해 주면 가독성이 훨씬 향상됩니다.

그림 235. Default Arguments Example.kt

```
1 Text("Hello, Android!")
2 // 위의 코드는 아래와 동일하게 동작합니다.
3 Text(
4     text = "Hello, Android!",
5     color = Color.Unspecified, // 기본값
6     fontSize = TextUnit.Unspecified // 기본값
7     // ... 기타 기본 매개변수
8 )
```

기본 매개변수를 통해 복잡한 API 사용을 더 쉽게 하고, 매개변수에 이름을 지정해 줌으로써 코드의 유지보수를 쉽게 하고 전달되는 매개변수에 명확성을 부여합니다.

고차 함수 및 람다 표현식 (Higher-Order Functions and Lambda Expressions)

Compose는 함수를 매개변수로 받는 고차 함수를 광범위하게 사용합니다. 이는 Button과 같이 onClick 람다 함수를 매개변수로 받아 사용자 상호 작용을 처리하는 UI 컴포넌트에서 아주 흔하게 볼 수 있습니다.

그림 236. Higher Order Function Example.kt

```
1 Button(onClick = { showToast("Clicked!") }) { // 람다를 onClick 콜백으로 전달
2     Text("Click Me")
3 }
```

별도의 함수를 정의하는 대신 람다 표현식을 사용하면 동작을 인라인으로 정의하여 코드를 더 가독성 있게 하고, 유지 관리하기 쉽게 만들 수 있습니다. 물론 그렇다고 고차 함수를

너무 많은 매개변수로 사용하면, 컴포넌트를 사용할 때 혼란이 발생할 수 있으니 정확하게 룰을 정해놓고 사용하는 편이 좋습니다.

후행 람다 (Trailing Lambdas)

좀 전에 살펴본 람다 표현식이랑 비슷한 개념이지만 조금 다릅니다. Kotlin은 람다 표현식을 **함수의 마지막 매개변수**로 전달하면 람다의 실행체를 함수 바깥으로 뺄 수 있도록 허용하여 코드를 더 간결하게 만듭니다. 이는 Column과 같이 컴포저블에 마지막 매개변수로 전달되는 content 람да와 같이 일반적인 Composes UI 컴포넌트에서 흔히 사용되는 관용구입니다.

그림 237. Trailing Lambda Example.kt

```
1 Column { // 마지막 람다를 괄호 밖으로 이동
2     Text("Item 1")
3     Text("Item 2")
4 }
```

이는 Compose 레이아웃을 더 가독성 있게 만들고, 쉽게 구조화할 수 있도록 합니다.

스코프 및 수신 객체 (Scopes and Receivers)

Compose API는 종종 스코프가 포함된 람다 함수를 제공하여 특정 컨텍스트 내에서만 특정 Modifier나 프로퍼티에 접근할 수 있도록 합니다. 예를 들어, Row의 content 매개변수는 RowScope를 수신자로 받고, RowScope 내에서는 행 배치에 특화된 정렬 옵션 등을 사용할 수 있습니다.

그림 238. Scopes And Receivers Example.kt

```
1 Row { // RowScope가 암시적으로 제공됨
2     Text(
3         text = "Hello",
4         // RowScope 내에서만 사용 가능한 align 확장 함수
5         modifier = Modifier.align(Alignment.CenterVertically)
6     )
7 }
```

이는 코드 구성을 개선하고 의도된 컨텍스트 외부에서 함수를 오용하는 것을 방지합니다.

위임 속성 (Delegated Properties)

Compose는 상태를 효율적으로 관리하기 위해 위임 속성(by 구문)을 사용합니다. remember 함수는 recomposition으로부터 값을 유지하는 반면, mutableStateOf는 UI를 반응형으로 만듭니다.

그림 239. Delegated Properties Example.kt

```
1 // 'by' 키워드를 사용하여 State 객체에서 값을 직접 위임받음
2 var count by remember { mutableStateOf(0) }
```

이는 값이 변경될 때 자동으로 recomposition을 트리거하여 상태 관리를 단순화합니다.

데이터 클래스 구조 분해 (Destructuring Data Classes)

Kotlin의 구조 분해 기능은 특히 ConstraintLayout과 같이 제약 조건 기반의 레이아웃을 작업할 때 유용합니다.

그림 240. Destructuring Example.kt

```
1 // ConstraintLayout의 createRefs() 예시
2 val (image, title, subtitle) = createRefs()
```

이는 코드를 더 표현력 있게 만들고, 변수 선언을 여러 라인에 걸쳐 하지 않아도 됩니다.

싱글톤 객체 (Singleton Objects)

Kotlin의 object 선언은 일반적으로 MaterialTheme과 같은 테마 시스템에서 사용되는 싱글톤 생성을 단순화합니다.

그림 241. Singleton Object Example.kt

```
1 val primaryColor = MaterialTheme.colorScheme.primary // MaterialTheme은 object
```

이는 애플리케이션 전체에서 일관된 스타일링을 보장합니다.

타입-세이프 빌더 및 DSL (Type-Safe Builders and DSLs)

Jetpack Compose는 선언적 UI 구조를 만들기 위해 Kotlin의 DSL 기능을 활용합니다. 예를 들어, LazyColumn은 계층적 UI 컴포넌트를 읽기 쉬운 방식으로 정의하기 위해 타입-세이프 빌더를 활용합니다.

그림 242. `TypeSafe Builders Example.kt`

```

1 LazyColumn {
2     item { Text("Header") } // 단일 아이템 정의
3     items(listOf("Item 1", "Item 2")) { item -> // 목록 아이템 정의
4         Text(item)
5     }
6 }
```

이는 간결하고 구조화된 UI 레이아웃 정의를 가능하게 합니다.

Kotlin 코루틴

Compose는 내부적으로 코루틴을 적극적으로 사용하고 하여 비동기 작업을 처리하기 위한 방법으로 코루틴을 많이 활용합니다. `rememberCoroutineScope` 함수는 `recomposition`이 발생해도 진행하고 있는 작업을 유지하는 코루틴 스코프를 제공합니다.

그림 243. `Coroutines Example.kt`

```

1 val scope = rememberCoroutineScope() // 컴포지션에 연결된 코루틴 스코프
2 val scrollState = rememberScrollState() // 스크롤 상태
3
4 Button(onClick = {
5     scope.launch { // 코루틴 시작
6         scrollState.animateScrollTo(0) // 스크롤 애니메이션
7     }
8 }) {
9     Text("Scroll to Top")
10 }
```

코루틴을 사용하면 콜백이 필요 없어 비동기 작업을 더 직관적으로 처리할 수 있습니다.

요약

Compose는 더 직관적이고 표현력 있는 UI 개발 경험을 제공하기 위해 Kotlin 관용구를 흔히 사용합니다. 기본 매개변수, 람다 표현식, 후행 람다 및 DSL은 가독성을 향상시키며, 코루틴을 사용하여 비동기를 원활하게 처리합니다. 이러한 Kotlin의 언어적 특성을 이해하면 Compose API가 제공하는 방식의 원리나 개발자의 의도를 이해할 수 있고, 더불어 가독성이 좋고 효율적인 코드를 작성할 수 있습니다.

실전 질문

Q) 컴포저블 함수를 구조화하기 위해 후행 람다나 고차 함수를 자주 사용하는데 이러한 Kotlin 관용구가 Compose API에서 사용되는 예시를 들어주세요.

카테고리 1: Compose Runtime

Compose Runtime은 프로그래밍 모델 및 상태 관리를 위한 핵심 엔진 역할을 하는 Jetpack Compose의 근본적인 부분입니다. Jetpack Compose는 처음부터 이러한 내부 구조에 대한 별다른 지식 없이도 직관적이고 원활하게 작동하도록 설계되었지만, 성능 및 메모리 관리 최적화를 위해서는 Compose Runtime API를 이해하는 것이 사실상 필수적입니다.

Compose Runtime은 내부적으로 상태를 관리하여 개발자가 수동적으로 UI를 업데이트해야 할 필요성을 줄입니다. 한편으로, 사용하는 API가 내부적으로 어떻게 작동하는지에 대한 확실한 이해가 있으면, 상태 관리 및 사이드 이펙트 처리를 다룰 때 의도하지 않은 동작 및 오류를 최소화하면서 더 효율적인 성능으로 앱을 구축할 수 있습니다.

Q) 11. 상태(State)란 무엇이며 이를 관리하는 데 사용되는 API는 무엇인가요?

Jetpack Compose에서 **State**⁴⁸는 앱 시나리오에서 흔히 변경될 수 있는 값이자, UI에서 동적으로 반영되는 데이터를 나타냅니다. 흔히 사용되는 상태는 네트워크 오류에 대한 Snackbar 메시지 노출, 사용자 입력 또는 상호 작용으로 발생하는 애니메이션을 표시하기 위한 용도로 사용됩니다. 상태는 UI 업데이트를 직접 트리거하기 때문에 Compose와 같은 선언적 프레임워크에서 특히 더 중요한데 Compose는 현재 상태를 기반으로 컴포저블을 렌더링 하며, 상태가 변경되면 변경된 상태를 UI에 반영하기 위해 다시 렌더링 합니다.

State와 Composition

Jetpack Compose는 선언적 UI 접근 방식을 따르므로, UI 업데이트는 컴포저블이 변경된 매개변수를 통해 호출될 때만 발생합니다. 이러한 동작은 **composition** 생명주기와 밀접하게 관련 있습니다.

- **초기 Composition:** 컴포저블을 실행하여 UI 트리가 처음 생성되고 렌더링되는 프로세스입니다.
- **Recomposition:** 상태 변경 시 트리가 되며, recomposition은 관련 컴포저블을 업데이트하여 새로운 상태를 반영합니다.

⁴⁸<https://developer.android.com/develop/ui/compose/state>

Compose Runtime은 상태 변경 사항을 자동으로 추적하고, 안드로이드 View 시스템에서 UI를 호출하기 위해 사용하는 `View.invalidate()` 메서드와 유사한 동작을 개발자 대신하여 UI를 업데이트합니다. **Recomposition**은 업데이트된 상태를 반영해야 하는 컴포저블 함수에 대해서만 트리거 됩니다. 아래 예시는 상태 변경에 따라 UI가 자동으로 업데이트되는 방식을 보여줍니다.

그림 244. HelloContent.kt

```
1 @Composable
2 fun HelloContent() {
3     Column(modifier = Modifier.padding(16.dp)) {
4         // remember와 mutableStateOf를 사용하여 상태 정의
5         var name by remember { mutableStateOf("") }
6
7         // name 상태가 비어 있지 않을 때만 Text 표시
8         if (name.isNotEmpty()) {
9             Text(
10                 text = "Hello, $name!",
11                 modifier = Modifier.padding(bottom = 8.dp)
12             )
13         }
14
15         // TextField는 name 상태를 표시하고 업데이트
16         TextField(
17             value = name, // 현재 상태 값
18             onValueChange = { name = it }, // 상태 업데이트 콜백
19             label = { Text("Name") }
20         )
21     }
22 }
```

여기서 `name` 상태가 변경되면 `Text` 및 `TextField` 컴포저블이 자동으로 업데이트되어 UI가 최신의 상태 값과 동기화되도록 보장합니다.

Compose에서 상태 관리하기

Jetpack Compose는 상태를 효과적으로 관리하기 위해 여러 API들을 제공합니다.

1. **remember**: 초기 컴포지션이 발생했을 때 메모리에 객체를 저장하고 recomposition이 발생하면 기존 메모리에 저장된 값을 꺼내옵니다. 따라서, API의 이름 그래도 상태 값을 메모리에 '기억한다'라고 이해하면 됩니다.

그림 245. RememberUsage.kt

```
1 var count by remember { mutableStateOf(0) }
```

2. **rememberSaveable**: recomposition 뿐만 아니라, 화면 회전과 같은 구성 변경 시에도 상태를 유지합니다. Bundle에 저장할 수 있는 유형 또는 그 외 유형에 대해서는 커스텀 saver 객체와 함께 작동할 수 있습니다.
3. **mutableStateOf**: API 이름 그대로 가변적인 상태를 의미합니다. 상태 값이 변경될 때 recomposition을 트리거하는 관찰 가능한 상태 객체를 생성합니다.

remember와 mutableStateOf는 아래와 같이 함께 사용되어 상태 객체를 메모리에 저장할 수 있습니다.

그림 246. MutableStateExamples.kt

```
1 // State<String> 타입
2 val mutableState: MutableState<String> = remember { mutableStateOf("") }
3 // String 타입 (delegate 속성 사용)
4 var value: String by remember { mutableStateOf("") }
5 // 구조 분해 선언 (value: String, setValue: (String) -> Unit)
6 val (value, setValue) = remember { mutableStateOf("") }
```

 **Additional Tips:** 왜 remember와 mutableStateOf가 함께 사용되어야 하나요? 만약 위의 내용을 읽으면서 이와 같은 질문을 하셨다면 아주 잘하고 계신 겁니다. 늘 그렇듯 당연하게 사용되는 API의 동작 원리에 대해서 궁금증을 가지셔야 더 나은 개발자가 될 수 있습니다. remember API는 객체를 메모리에 저 정 한다라고 배웠습니다. 마찬가지로, mutableStateOf는 상태값이 변할 때 recomposition을 트리거하기 위한 관찰 가능한 객체를 생성하는 API입니다. 만약 mutableStateOf만 사용하면 어떻게 될까요? mutableStateOf만 사용하면 상태값이 바뀔 때마다 recomposition을 트리거하게 되는데, 문제는 해당 상태값 자체가 recomposition이 발생할 때마다 메모리에 저장되어있지 않기 때문에 초기화되고, 기대했던 동작과는 다른 결과가 나오게 됩니다. 따라서, 상태 또한 메모리에 저장을 해야 의미가 있기에, remember API를 함께 사용해야 합니다.

요약

State는 Jetpack Compose가 선언형 UI의 근간이 되도록 하는 API이며, State 없는 Jetpack Compose는 존재할 수 없다고 해도 과언이 아닙니다. 개발자 입장에서는 가장 중요한 API라고 여겨야 합니다. State는 recomposition을 트리거하여 데이터 변경 사항을 UI에 반영하고 새로운 상태로 유지할 수 있도록 합니다. 또한, recomposition은 효율적으로 트리거 되므로 개발자는 UI를 수동으로 업데이트하고 다시 렌더링 할 필요가 없습니다. 반면에, 의도하지 않은 recomposition이 발생하면 전반적으로 앱 성능이 저하될 수 있기 때문에, State의 동작 방식을 올바르게 이해하는 것은 Jetpack Compose의 사용방식을 이해하는 것과 동일한 만큼 중요하다고 볼 수 있습니다.

실전 질문

Q) 상태는 recomposition과 어떤 관련이 있으며, recomposition이 발생하면 내부적으로 어떤 일이 발생하나요?

Q) 12. 상태 호이스팅(state hoisting)으로 어떤 이점을 얻을 수 있나요?

상태 호이스팅(State hoisting)⁴⁹은 상태를 상위 수준의 컴포저블 함수로 끌어올리는 것을 의미합니다. Hoisting이란, 기종기와 같은 것으로 무언가를 위쪽으로 끌어올리는 것을

⁴⁹<https://developer.android.com/develop/ui/compose/state-hoisting>

의미합니다. 따라서, 상태 값과 상태 값을 업데이트하는 람다 함수를 컴포저블 매개변수로 전달하고 해당 값은 현재 컴포저블이 아닌 다른 호출자 쪽에서 관리하도록 하는 것입니다. 상태 호이스팅은 근본적으로 단방향 데이터 흐름 원칙을 따르므로 UI를 더 쉽게 관리하고 확장 가능하게 만듭니다.

상태 호이스팅의 특성은 아래와 같습니다.

- **State**는 부모 컴포저블에서 관리합니다. 즉, 상태 호이스팅을 적용하려는 컴포저블은 상태 자체를 가져서는 안 되고, 매개변수로 상태를 받아서도 안 됩니다.
- **Events** 또는 **triggers** (`onClick`, `onValueChange` 등)는 자식에서 값을 바꾸고, 해당 값을 부모로 다시 전달받는 형태로 상태를 업데이트합니다. 따라서, 보편적으로 람다 함수를 매개변수로 넘기고, 해당 컴포저블을 호출하는 쪽에서 업데이트된 값을 콜백으로 받아 상태를 업데이트시키는 형태로 동작합니다.
- 업데이트된 상태는 매개변수로 자식에게 다시 전달되어 단방향 데이터 흐름을 생성합니다.

예제

아래 예제를 통해서 상태 호이스팅이 어떻게 구현되는지 살펴보겠습니다.

그림 247. State Hoisting.kt

```
1 @Composable
2 fun Parent() {
3     var sliderValue by remember { mutableStateOf(0f) }
4
5     // SliderComponent에 상태와 콜백 전달
6     SliderComponent(
7         value = sliderValue,
8         onValueChange = { sliderValue = it } // 상태를 업데이트하는 람다 함수
9     )
10 }
11
12 // SliderComponent는 상태를 직접 관리하지 않음 (Stateless)
13 // 즉, 단방향 데이터 흐름이 성립됨
```

```
14 @Composable
15 fun SliderComponent(value: Float, onValueChange: (Float) -> Unit) {
16     Slider(value = value, onValueChange = onValueChange)
17 }
```

위의 예제에서, `SliderComponent` 컴포저블만 놓고 보았을 때, 해당 컴포저블은 내부적으로 어떠한 상태도 관리하고 있지 않습니다. 따라서 `SliderComponent`는 앞으로 다른 목적을 가진 여러 화면에서 동일한 기능을 하는 컴포넌트로서 재사용할 수 있습니다. 이러한 컴포저블 함수를 `stateless` (상태가 없는)이라고 부릅니다.

만약 아래와 같은 함수는 어떨까요?

그림 248. `Stateful.kt`

```
1 @Composable
2 fun SliderComponent() {
3     var sliderValue by remember { mutableStateOf(0f) }
4
5     Slider(value = sliderValue, onValueChange = { sliderValue = it })
6 }
```

위의 예제에서는 `SliderComponent`가 내부적으로 상태를 가지고 있어서 호출자 쪽에서는 별다른 매개변수를 입력하지 않아도 호출할 수 있고 상태를 내부에서 관리하기 때문에 사용하기 더 쉬워 보입니다. 그러나, 다른 화면에서 `sliderValue` 값을 `Hello, $sliderValue` 와 같은 형태로 약간의 변화가 주어진 요구사항만 생겨도 완전히 다른 컴포저블 함수를 생성해야 합니다. 이러한 컴포저블 함수를 `stateful` (상태를 저장하는)이라고 부릅니다. 보시는 바와 같이 재사용성이 떨어집니다.

 **Additional Tips:** 상태, 상태 호이스팅, `remember` 등 Compose API의 이름은 상당히 직관적인 편입니다. API의 본질을 이해하는 가장 빠른 방법은 API의 이름을 이해하는 것입니다.

상태 호이스팅의 장점

- **더 나은 재사용성:** 상태 호이스팅을 적용하 컴포저블을 상태 없는 형태(stateless) 및 재사용 가능하게 만들 수 있습니다. 상태 및 이벤트 콜백을 전달함으로써 동일한 컴포저블을 특정 구현에 얹매이지 않고 다른 화면이나 컨텍스트에서 사용할 수 있습니다.
- **단순화된 테스트(Simplified Testing):** 상태를 저장하지 않는 컴포저블은 매개변수로 전달된 상태 값에 전적으로 의존하므로 테스트하기가 더 쉽습니다. 이로 인해 예측 가능하고 명확한 테스트 시나리오가 가능해집니다.
- **더 나은 관심사 분리(Better Separation of Concerns):** 상태 관리 로직을 부모 컴포저블 또는 ViewModel로 옮김으로써, UI 컴포넌트가 인터페이스 렌더링에만 집중하도록 합니다. 이러한 역할 분리는 비즈니스 로직과 UI 코드를 구별하여 유지 관리성을 향상시킵니다.
- **단방향 데이터 흐름 지원(Support for Unidirectional Data Flow):** 상태 호이스팅은 Jetpack Compose의 단방향 데이터 흐름 아키텍처와 일치하여 상태가 단 하나의 공급원(single source of truth)에서 흐르도록 보장합니다. 이는 여러 소스가 동일한 상태를 관리하려고 할 때 발생하는 예상치 못한 동작의 발생 가능성을 줄입니다.
- **향상된 상태 관리(Enhanced State Management):** 상태 호이스팅을 사용하면 ViewModel 또는 부모 컴포저블과 같은 상위 수준 컨테이너에서 상태를 중앙 집중화할 수 있습니다. 이를 통해 복잡한 UI 흐름을 관리하고 인스턴스 상태 저장 또는 상태 복원 관리와 같은 작업을 더 쉽게 처리할 수 있습니다.

요약

상태 호이스팅은 더 깔끔하고 테스트 가능한 코드를 만들도록 하며, 단방향 데이터 흐름을 가능하도록 하여 재사용성과 유지 보수성을 향상시킵니다. 컴포저블을 상태를 관리하지 않는 형태로 유지함으로써 개발자는 변화하는 애플리케이션 요구 사항에 따라 기존의 UI를 최대한 재사용하여 유연한 UI 컴포넌트를 만들 수 있습니다.

실전 질문

- Q) 상태 호이스팅이 왜 컴포저블 함수의 재사용성과 테스트 용이성을 향상시키나요? 예시를 들어서 설명해 주세요.
- Q) 어떤 시나리오에서 상태 호이스팅을 사용하지 않고, 컴포저블 내부에 상태를 갖도록 하실 건가요? 해당 시나리오에서는 컴포저블에 상태를 갖도록 하는 것이 상태 호이스팅을 적용하는 것에 비해 어떤 장점을 갖나요?

💡 Pro Tips for Mastery: Stateful vs. Stateless, 코드로 상태 호이스팅 이해하기

상태 호이스팅은 상태 관리를 호출하는 쪽(call site)으로 이동시켜 상태 저장(stateful) 컴포저블을 상태 비저장(stateless) 컴포저블로 변환하는 데 사용되는 디자인 패턴입니다. 이러한 접근 방식은 일반적으로 `remember`를 사용하여 관리되는 내부 상태 변수를 1. '현재의 상태 값'과 2. '상태를 업데이트 하는 콜백 함수'라는 두 개의 매개변수로 대체합니다. 이와 같은 관심사 분리를 통해 더 깔끔하고 재사용 가능하며 테스트 가능한 컴포저블 함수를 만들 수 있습니다.

상태 관리 책임을 호출자에게 이전함으로써 컴포저블은 상태 비저장 상태가 되어 재사용, 테스트 및 유지 관리가 더 쉬워집니다. 예를 들어, `MyTextField` 컴포저블은 부모로부터 직접 텍스트 값과 사용자 입력 처리를 위한 콜백을 받아 명확한 데이터 흐름을 보장할 수 있습니다. 이러한 분리는 컴포저블이 UI 렌더링에만 집중하도록 하고 상태 관리는 호출 컴포넌트에 맡겨 일종의 모듈화를 가능하게 하고 복잡성을 줄입니다.

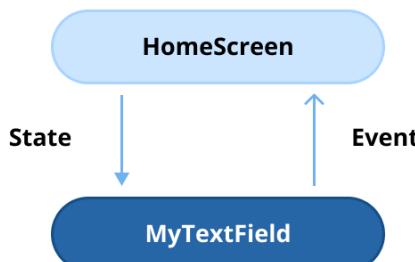


그림 249. state-hoisting

다음으로, 실전 예제 코드를 통하여 **Stateful** 및 **Stateless** 컴포저블 함수의 차이점을 살펴보겠습니다. 이해를 높이기 위해 진행하면서 위 그림을 참조하시면 좋습니다. 사용자 입력을 처리하도록 설계된 `MyTextField`라는 커스텀 텍스트 필드를 떠올려 보세요. `MyTextField`를 **Stateful** 컴포저블 함수로 구현하는 방법은 다음과 같습니다.

그림 250. Stateful.kt

```
1 @Composable
2 fun HomeScreen() {
3     // MyTextField가 자체적인 상태를 관리하므로 호출자는 편합니다.
4     MyTextField()
5 }
6
7 @Composable
8 fun MyTextField() {
9     // remember와 mutableStateOf를 사용하여 내부 상태 관리
10    val (value, onValueChange) = remember { mutableStateOf("") }
11
12    // 내부 상태를 TextField에 전달
13    TextField(value = value, onValueChange = onValueChange)
14 }
```

위 코드 예제에서 `MyTextField`는 상태를 메모리에 저장하고 입력 변경 사항을 추적하는 `remember` 및 `mutableStateOf` 함수를 사용하여 내부 상태를 관리합니다. 이러한 디자인은 자체 상태를 독립적으로 처리하므로 `MyTextField`를 **Stateful** 컴포저블로 만듭니다.

위의 접근 방식에는 장단점이 있습니다. 긍정적인 측면은 호출 사이트(`HomeScreen`)가 상태를 관리할 필요가 없어 구현이 단순화된다는 것입니다. 그러나 단점은 유연성이 감소한다는 것입니다. `MyTextField`가 내부적으로 상태를 관리하므로 외부에서 동작을 제어하거나 커스텀하기가 더 어려워집니다. 이로 인해 다른 컨텍스트에서 컴포저블을 재사용하기가 더 어려워질 수 있습니다.

이러한 제한 사항을 해결하면서 동일한 기능을 달성하는 접근 방식을 살펴보겠습니다.

그림 251. Stateless.kt

```
1 @Composable
2 fun HomeScreen() {
3     // 상태를 HomeScreen에서 관리 (호이스팅 - 끌어 올린다고 이해하면 됩니다)
4     val (value, onValueChanged) = remember { mutableStateOf("") }
5
6     // MyTextField에 상태와 콜백 전달
7     MyTextField(
8         value = value,
9         onValueChanged = onValueChanged
10    )
11 }
12
13 // MyTextField는 상태를 받아서 표시하고 변경 사항을 위임 (Stateless)
14 @Composable
15 fun MyTextField(
16     value: String,
17     onValueChanged: (String) -> Unit // 상태 변경 콜백
18 ) {
19     TextField(value = value, onValueChange = onValueChanged)
20 }
```

예제에서 MyTextField는 매개변수를 통해 변경 사항을 반영하면서 상태 관리를 호출자(HomeScreen) 쪽에 위임하는 **Stateless** 컴포저블로 구현됩니다. 이러한 접근 방식은 이전의 상태 저장(stateful) 구현에 비해 코드가 약간 길어질 수 있지만, 재사용성이 크게 향상되었다는 상당한 이점을 제공합니다. 즉, MyTextField를 stateless 한 상태로 유지함으로써 다른 상황에서도 보다 쉽게 재사용하고 여러 사용 사례에 맞게 조정하여 더 깔끔하고 상태와 독립적인 컴포넌트로 관리할 수 있습니다.

이 디자인 패턴은 상태 관리가 호출 수신자(MyTextField)에서 호출자(HomeScreen)로 “들어 올려지는(hoisted)” **상태 호이스팅(State Hoisting)**이라고 부릅니다. UI 계층 구조에서 상태를 더 높은 UI 계층에서 관리함으로써 다른 컴포넌트 간에 더 유연하고 재사용 가능하며 상태를 개발자가 직접 제어할 수 있도록 합니다.

위의 stateless 예제를 기반으로 사용자로부터 받는 숫자 입력을 필터링하는 텍스트 필드를 만들어야 하는 시나리오를 떠올려 보세요. 이는 아래 예제와 같이 원본 컴포저블을 수정하지 않고도 쉽게 커스텀 수 있는 stateless 컴포저블의 장점을 보여줍니다.

그림 252. State Hoisting.kt

```
1 @Composable
2 fun HomeScreen() {
3     val (value, onValueChanged) = remember { mutableStateOf("") }
4     // derivedStateOf를 사용하여 숫자 필터링 로직 추가 (remember로 키 지정)
5     val processedValue by remember(value) { derivedStateOf { value.filter { !it.isDigit()
6         } } }
7     // 필터링된 값을 표시하고 원본 값 변경 콜백 전달
8     MyTextField(
9         value = processedValue,
10        onValueChanged = onValueChanged // 사용자가 입력 시 원본 'value' 업데이트
11    )
12 }
13
14 @Composable
15 fun MyTextField(
16     value: String,
17     onValueChanged: (String) -> Unit
18 ) {
19     TextField(value = value, onValueChange = onValueChanged)
20 }
```

또한 MyTextField는 특정 요구 사항에 맞게 커스텀하여 다양한 방식으로 재사용할 수 있습니다. 이러한 유연성은 상태 호이스팅의 주요 이점인 외부 제어 및 커스텀을 통해 컴포저블 함수의 재사용성을 향상시킵니다. 결과적으로 컴포넌트는 내부 변경 없이 다른 요구사항에서도 유연하게 사용될 수 있어 코드베이스를 더 깔끔하고 유지 관리하기 쉽게 합니다.

Q) 13. remember와 rememberSaveable의 차이점은 무엇인가요?

Jetpack Compose에서 **상태 관리(state management)**는 UI가 데이터 변경에 따라 동적으로 업데이트되도록 하는 핵심적인 개념입니다. remember와 rememberSaveable은 모두 recomposition로부터 상태를 유지하도록 하는 API이지만 서로 다른 목적을 가지고 있으며 각자 다른 시나리오에서 적합합니다.

remember 이해하기

- 목적:** remember API는 메모리에 값을 저장하고 recomposition가 발생했을 경우 메모리에 저장된 값을 꺼내와 상태를 유지합니다. 그러나 화면 회전이나 프로세스 재시작과 같은 구성 변경 중에는 상태를 유지하지 않습니다.
- 유즈 케이스:** 상태가 구성 변경 후에도 유지될 필요가 없을 때 remember를 사용합니다. 예를 들어, 화면이 회전되거나 사용자가 언어 설정 등을 바꾸었을 경우, 정보가 날아가도 상관없는 상태의 경우 remember가 적합합니다.

그림 253. RememberExample.kt

```

1 @Composable
2 fun RememberExample() {
3     var count by remember { mutableStateOf(0) }
4
5     Button(onClick = { count++ }) {
6         Text("Clicked $count times")
7     }
8 }
```

- 동작:** remember는 현재 컴포지션 생명주기 내에서만 상태를 저장하므로 기기가 회전하면 count 변수가 0으로 재설정됩니다.

rememberSaveable 이해하기

- 목적:** rememberSaveable API는 구성 변경 시에도 상태를 유지하여 remember 보다 더 넓은 범위에서 상태를 저장하고 복원합니다. 안드로이드 SDK의 Bundle에 저장할 수 있는 값에 한하여 자동으로 저장하고 복원합니다.

- **유즈 케이스:** 유저 인풋 입력이나 내비게이션 상태와 같이 구성 변경 후에도 유지되어야 하는 상태에 대해서는 rememberSaveable을 사용해야 합니다.

그림 254. RememberSaveableExample.kt

```
1 @Composable
2 fun RememberSaveableExample() {
3     var text by rememberSaveable { mutableStateOf("") }
4
5     OutlinedTextField(
6         value = text,
7         onValueChange = { text = it },
8         label = { Text("Enter text") }
9     )
10 }
```

- **행동:** text 값은 화면 회전이나 구성 변경 시에도 유지되어 원활한 사용자 경험을 보장합니다.



Additional Tips: 그러면 무조건 rememberSaveable를 사용하는 것이 recomposition 뿐만 아니라, 상태를 구성 변경 등으로부터 유지하여 더 원활한 사용자 경험을 만들 수 있지 않나요? 아주 좋은 질문입니다. 조금 아래에서 더 살펴볼 건데 remember와 rememberSaveable는 API 이름이 닮았지만, 내부 구현체는 완전히 다른 구조로 이루어져 있습니다. rememberSaveable는 내부적으로 remember 보다 훨씬 다양한 내부 처리 및 오버헤드가 발생하기 때문에, 무턱대고 모든 상황에 rememberSaveable를 사용하는 것은 오히려 앱 성능을 떨어뜨리고 사용자 경험을 방해할 수도 있습니다. 개발에 있어서 trade-off 없이 모든 상황에서 백퍼센트 좋기만 한 완전한 솔루션은 거의 존재하지 않습니다.

주요 차이점

기능	<code>remember</code>	<code>rememberSaveable</code>
지속성	현재 컴포지션 생명주기 동안에만 상태 유지.	컴포지션 및 구성 변경 시 상태 유지.
저장 위치	메모리에 값 저장.	메모리에 값을 저장하고 Bundle에 저장.
커스텀 Saver 지원	해당 없음.	복잡한 객체에 대한 커스텀 saver 지원.

사용 시기

- 애니메이션이나 임시적인 UI 상태와 같이 현재 컴포지션을 넘어서 유지될 필요가 없는 일시적인 상태에는 `remember`를 사용합니다.
- 사용자 입력, 선택 상태 또는 양식 데이터와 같이 구성 변경 시에도 유지되어 더 나은 사용자 경험을 제공할 수 있는 상태에는 `rememberSaveable`를 사용합니다.

요약

`remember`와 `rememberSaveable`은 Jetpack Compose에서 상태를 관리하기 위한 필수적인 API입니다. `remember`는 단일 컴포지션 내의 **일시적 상태(transient state)**에 적합한 반면, `rememberSaveable`은 구성 변경 시 **상태 지속성(state persistence)**을 보장합니다. 그러나 `rememberSaveable`은 오버헤드가 발생하기 때문에 모든 시나리오에 항상 최상의 선택은 아닙니다. 이들의 차이점을 이해하고 각각을 상황에 맞게 적절하게 사용하면 애플리케이션 요구 사항에 따라 가장 효율적인 API를 선택할 수 있습니다.

실전 질문

Q) `remember`이 아닌 `rememberSaveable`를 사용해야하는 시나리오는 무엇이며, 어떤 트레이드오프를 고려해야 하나요?

Q) 기본적으로 지원되지 않는 커스텀 비원시(non-primitive) 값을 `rememberSaveable`로 저장하려면 어떻게 해야 하나요?

Pro Tips for Mastery: remember 및 rememberSaveable 내부 구조

지금까지 remember와 rememberSaveable의 목적에 대해 살펴보았으니, 이제 내부 구현체를 살펴보고 동작 구조에 대해서 더 자세하게 살펴보겠습니다.

remember 내부 구조 이해하기

remember 함수의 내부 구현은 다음과 같습니다.

그림 255. remember.kt

```

1 @Composable
2 inline fun <T> remember(crossinline calculation: @DisallowComposableCalls () -> T): T =
3     currentComposer.cache(false, calculation)

```

코드에서 볼 수 있듯이 remember는 내부적으로 Composer 인스턴스에서 cache 함수를 호출합니다. cache 함수가 구현되는 방식은 다음과 같습니다.

그림 256. cache.kt

```

1 @ComposeCompilerApi
2 inline fun <T> Composer.cache(invalid: Boolean, block: @DisallowComposableCalls () ->
3     → T): T {
4     @Suppress("UNCHECKED_CAST")
5     return rememberedValue().let {
6         if (invalid || it === Composer.Empty) { // 값이 유효하지 않거나 초기화되지
7             ↳ 않았는지 확인
8             val value = block() // 계산(remember 매개변수인 `calculation`) 블록 실행
9             updateRememberedValue(value) // 컴포지션 데이터에 값 저장
10            value
11        } else it // 이전에 기억된 값 반환
12    } as T
13 }

```

위의 코드는 remember가 내부적으로 어떻게 작동하는지 보여줍니다. Compose 컴파일러 플러그인 API와 상호 작용하여 컴포지션 데이터에 값을 캐시합니다. 구체적으로, 값이

유효하지 않거나 초기화되지 않았는지(Composer.Empty로 표시됨) 확인합니다. 초기화 되지 않았다면, 제공된 블록 람다 함수를 사용하여 값을 계산하고 컴포지션 데이터에 저장한 다음 반환합니다. 그렇지 않으면 이전에 기억된 값을 단순히 복원하여 반환합니다. 이와 같은 메커니즘은 중복적인 계산을 피하면서 recomposition이 발생해도 값이 효율적으로 유지되도록 보장합니다.

rememberSaveable 내부 구조 이해하기

rememberSaveable 함수의 내부 구현은 다음과 같습니다.

그림 257. rememberSaveable.kt

```
1 @Composable
2 fun <T : Any> rememberSaveable(
3     vararg inputs: Any?, // 상태 재계산을 트리거 하기 위한 입력 키
4     saver: Saver<T, out Any> = autoSaver(), // 상태 저장 및 복원 로직
5     key: String? = null, // 상태 저장을 위한 고유 키
6     init: () -> T // 초기값 계산 람다
7 ): T {
8     // 현재 컴포지션 해시로부터 기본 키 생성
9     val compositeKey = currentCompositeKeyHash
10    // 사용자가 제공한 키 또는 생성된 키 사용
11    val finalKey = if (!key.isNullOrEmpty()) {
12        key
13    } else {
14        compositeKey.toString(MaxSupportedRadix)
15    }
16    @Suppress("UNCHECKED_CAST")
17    (saver as Saver<T, Any>) // Saver 타입 캐스팅
18
19    // 현재 SaveableStateRegistry 가져오기
20    val registry = LocalSaveableStateRegistry.current
21
22    // 상태를 관리할 Holder 생성 및 기억
23    val holder = remember(saver, registry, finalKey) { // 키 변경 시 재실행
```

```
24     // 레지스트리에서 복원 시도 또는 init으로 초기화
25     val restored = registry?.consumeRestored(finalKey)?.let {
26         saver.restore(it)
27     }
28     val finalValue = restored ?: init()
29     SaveableHolder(saver, registry, finalKey, finalValue, inputs)
30 }
31
32     // 입력 키가 변경되지 않았다면 저장된 값 사용, 아니면 init 재실행
33     val value = holder.getValueIfInputsDidntChange(inputs) ?: init()
34
35     // recomposition 시 상태 업데이트 및 레지스트리에 등록
36     SideEffect {
37         holder.update(saver, registry, finalKey, value, inputs)
38     }
39
40     return value
41 }
```

약간 복잡해 보일 수 있지만 단계별로 분석해 보겠습니다. rememberSaveable 함수는 구성 변경 및 프로세스 종료 시에도 상태를 저장하고 복원하는 기능을 추가적으로 지원하여 remember 보다 더 넓은 스코프에서의 데이터 복원을 가능하게 합니다. 아래는 내부 구현 분석입니다.

1. **키 생성(Key Generation)**: key 매개변수를 통해 사용자가 커스텀 키를 제공할 수 있습니다. 제공되지 않으면 현재 컴포지션 해시를 사용하여 복합 키가 자동으로 생성됩니다.

```

1 val compositeKey = currentCompositeKeyHash
2 val finalKey = if (!key.isNullOrEmpty()) {
3     key
4 } else {
5     compositeKey.toString(MaxSupportedRadix)
6 }

```

2. 상태 복원(**State Restoration**): LocalSaveableStateRegistry는 주어진 키에 대해 이전에 저장된 값을 검색하는 데 사용됩니다. 저장된 값이 존재하면 제공된 Saver를 사용하여 복원됩니다.

```

1 val registry = LocalSaveableStateRegistry.current
2 val restored = registry?.consumeRestored(finalKey)?.let {
3     saver.restore(it)
4 }

```

3. 기본값 초기화(**Default Value Initialization**): 복원된 값이 없으면 init 람다를 사용하여 기본값이 초기화됩니다.

```

1 val finalValue = restored ?: init()

```

4. **Saveable Holder**: 상태, saver, 레지스트리 및 입력을 관리하기 위해 SaveableHolder가 생성됩니다.

```

1 SaveableHolder(saver, registry, finalKey, finalValue, inputs)

```

5. 입력 변경 처리(**Input Change Handling**): rememberSaveable에 대한 입력이 변경되면 상태가 무효화되고 값이 다시 초기화됩니다.

```
1 val value = holder.getValueIfInputsDidntChange(inputs) ?: init()
```

6. 사이드 이펙트(Side Effects): SideEffect는 recomposition 중에 업데이트된 상태가 레지스트리에 저장되도록 보장합니다.

```
1 SideEffect {  
2     holder.update(saver, registry, finalKey, value, inputs)  
3 }
```

rememberSaveable의 내부 구현을 하나씩 살펴보았습니다. remember와 달리 LocalSaveableStateRegistry를 사용하여 화면 회전과 같은 구성 변경 시에도 상태를 보존함으로써 상태의 복원 범위를 넓혔습니다. 또한 saver 매개변수를 통해 개발자는 커스텀 직렬화 및 역직렬화 로직을 정의할 수 있어 복잡한 객체를 원활하게 처리하는 것도 가능합니다.

Q) 14. 컴포저블 함수 내에서 안전하게 코루틴 스코프(coroutine scope)를 생성하는 방법은 무엇인가요?

Jetpack Compose에서 rememberCoroutineScope⁵⁰는 컴포저블 함수 내에서 코루틴 스코프를 안전하게 생성하고 관리하는 공식적으로 권장하는 접근 방식입니다. 이는 코루틴 스코프가 컴포지션에 연결되도록 보장하여 잠재적인 메모리 누수 및 부적절한 리소스 사용을 방지합니다.

rememberCoroutineScope를 사용하는 이유

Jetpack Compose의 rememberCoroutineScope는 컴포저블이 컴포지션을 벗어날 때 활성 중인 코루틴 스코프를 자동으로 취소합니다. 이는 내부적으로 컴포저블 함수의 내부 생명주기를 인지하게 작동하고 있음을 의미하고, 이를 통해 개발자는 수동적으로 생명주기를 관리할 필요 없이 컴포저블에서 코루틴을 안전하게 런칭 및 취소할 수 있습니다.

사용 예제

⁵⁰<https://developer.android.com/develop/ui/compose/side-effects#remembercoroutinescope>

그림 258. Using rememberCoroutineScope

```
1 @Composable
2 fun CounterWithReset() {
3     var count by remember { mutableStateOf(0) }
4     // 컴포지션에 연결된 코루틴 스코프 가져오기
5     val coroutineScope = rememberCoroutineScope()
6
7     Column(
8         modifier = Modifier.padding(16.dp),
9         horizontalAlignment = Alignment.CenterHorizontally
10    ) {
11        Text("Count: $count", style = MaterialTheme.typography.h5)
12        Spacer(modifier = Modifier.height(8.dp))
13        Button(onClick = { count++ }) {
14            Text("Increment")
15        }
16        Spacer(modifier = Modifier.height(8.dp))
17        Button(onClick = {
18            // 버튼 클릭 시 코루틴 시작
19            coroutineScope.launch {
20                // 리셋을 위한 지연 시뮬레이션
21                delay(1000)
22                // 지연 후 상태 업데이트
23                count = 0
24            }
25        }) {
26            Text("Reset After 1s")
27        }
28    }
29 }
```

작동 방식

- 컴포지션 인식(Composition Awareness)**: rememberCoroutineScope에 의해 생성된 코루틴 스코프는 컴포지션으로 범위가 지정됩니다. 즉, 해당 스코프 내에서 시작된 모든 코루틴 작업은 컴포저블이 컴포지션에서 제거될 때 취소됩니다.
- 상태 관리(State Management)**: remember API는 값을 메모리에 저장하고, recomposition이 발생했을 때 저장된 값을 복원합니다. rememberCoroutineScope 또한 내부적으로 유사하게 동작하는데, 상황에 따라 remember 함께 적절하게 사용하면 비동기 작업을 안전하게 관리하는 데 도움이 됩니다.
- 메모리 누수 방지(Avoids Memory Leaks)**: GlobalScope를 사용하거나 코루틴 스코프를 수동으로 관리하는 것과 달리, rememberCoroutineScope는 새 코루틴 스코프를 생성하고 컴포저블이 더 이상 사용되지 않을 때 리소스가 적절하게 정리되도록 보장합니다.

모범 사례

- 컴포지션 생명주기에 연결된 가벼운 UI 관련 작업에는 rememberCoroutineScope를 사용합니다. 컴포지션 범위를 넘어서 장기적으로 실행되어야 하는 작업의 경우 viewModelScope 또는 lifecycleScope와 같은 더 넓은 코루틴 스코프를 사용하여 적절한 생명주기 관리를 보장하면서 예기치 않은 작업 취소를 방지하는 것이 좋습니다.
- 컴포저블 내에서 직접 코루틴 스코프를 생성하는 것은 수동 동적으로 관리가 필요하고, 예상치 못한 메모리 누수를 유발할 수 있으므로 가능한 피해야 합니다.
- rememberCoroutineScope를 사용하더라도 컴포저블 내에서 해당 스코프를 사용한 비동기 로직, 특히 비즈니스 로직과 관련된 로직을 직접적으로 호출하기보다는, 더 나은 유지 관리성 및 성능을 위해 복잡한 작업을 ViewModel 또는 다른 아키텍처 계층으로 위임하는 것이 옳습니다. 또한, rememberCoroutineScope는 기본적으로 메인 스레드를 사용하기 때문에 비즈니스 로직을 해당 스코프에서 수행하는 것은 옳지 않습니다.

요약

rememberCoroutineScope는 컴포저블 함수 내에서 코루틴 스코프를 생성하기 위한 유용하고 안전한 API입니다. 코루틴 스코프를 컴포지션의 생명주기에 연결함으로써 리소스의 적절한 정리와 메모리 누수를 방지할 수 있습니다. 그러나 기본적으로 rememberCoroutineScope는 메인 스레드에서 코루틴 작업을 실행하므로 신중하게 사용해야 하고, 네트워크 요청이나 데이터베이스 쿼리와 같은 비즈니스 로직을 직접 실행하는 것은 피해야 합니다.

실전 질문

Q) 컴포저블 내에서 직접 코루틴 스코프를 생성하고 시작하면 어떤 위험성이 발생할 수 있나요? `rememberCoroutineScope`를 사용해야 하는 이유가 무엇인가요?

💡 Pro Tips for Mastery: `rememberCoroutineScope` 내부 구조

컴포저블 함수 내에서 `rememberCoroutineScope`를 사용하여 안전하게 코루틴 스코프를 런칭하는 방법을 살펴보았습니다. 이제 내부적으로 어떻게 작동하고 왜 해당 함수가 컴포지션의 생명주기를 인식하는지 내부 구현을 더 자세히 살펴보겠습니다.

`rememberCoroutineScope` 내부 구현

아래는 `rememberCoroutineScope`의 내부 구현입니다.

그림 259. `rememberCoroutineScope.kt`

```
1 @Composable
2 inline fun rememberCoroutineScope(
3     crossinline getContext: @DisallowComposableCalls () -> CoroutineContext =
4     { EmptyCoroutineContext }
5 ): CoroutineScope {
6     val composer = currentComposer
7     // CompositionScopedCoroutineScopeCanceller 인스턴스를 remember로 감싸서 유지
8     val wrapper = remember {
9         CompositionScopedCoroutineScopeCanceller(
10             // 실제 코루틴 스코프 생성
11             createCompositionCoroutineScope(getContext(), composer)
12         )
13     }
14     // 저장된 코루틴 스코프 반환
15     return wrapper.coroutineScope
16 }
```

위의 예시에서 알 수 있듯이, `CompositionScopedCoroutineScopeCanceller`와 `createCompositionCoroutineScope`라는 두 가지 핵심 함수로 구성됩니다.

1. CompositionScopedCoroutineScopeCanceller

CompositionScopedCoroutineScopeCanceller 클래스는 코루틴 스코프가 컴포지션 생명주기를 인식하도록 보장합니다. 내부 구현은 다음과 같습니다.

그림 260. CompositionScopedCoroutineScopeCanceller.kt

```
1 internal class CompositionScopedCoroutineScopeCanceller(
2     val coroutineScope: CoroutineScope
3 ) : RememberObserver { // 컴포지션 생명주기 관찰
4     override fun onRemembered() {
5         // nothing
6     }
7
8     override fun onForgotten() {
9         // 컴포지션을 떠날 때 현재 실행 중인 스코프 취소
10        coroutineScope.cancel(LeftCompositionCancellationException())
11    }
12
13    override fun onAbandoned() {
14        // 컴포지션이 버려질 때 현재 실행 중인 스코프 취소
15        coroutineScope.cancel(LeftCompositionCancellationException())
16    }
17 }
```

핵심은 아래와 같습니다.

- **RememberObserver 구현**: 컴포저블의 생명주기를 추적할 수 있습니다.
- **컴포지션 인식**: 컴포저블이 컴포지션에서 제거되면 onForgotten 또는 onAbandoned 메서드가 트리거 되고 coroutineScope.cancel()을 사용하여 현재 실행 중인 코루틴 스코프가 취소됩니다.
- **안전한 리소스 정리**: 스코프에서 시작된 모든 코루틴이 컴포저블이 컴포지션을 떠날 때 취소되도록 보장합니다.

2. `createCompositionCoroutineScope`

`createCompositionCoroutineScope` 함수는 주어진 코루틴 컨텍스트에 대해 새 코루틴 스코프를 생성하는 역할을 합니다. 내부 구현체는 다음과 같습니다.

그림 261. `createCompositionCoroutineScope.kt`

```

1 internal fun createCompositionCoroutineScope(
2     coroutineContext: CoroutineContext,
3     composer: Composer
4 ) = if (coroutineContext[Job] != null) {
5     // Job이 이미 존재하면 예외 발생 (독립 스코프 보장)
6     CoroutineScope(
7         Job().apply {
8             completeExceptionally(
9                 IllegalArgumentException(
10                    "CoroutineContext supplied to " +
11                    "rememberCoroutineScope may not include a parent job"
12                )
13            )
14        }
15    )
16 } else {
17     // Composer의 applyCoroutineContext와 새 Job 결합하여 스코프 생성
18     val applicationContext = composer.applyCoroutineContext
19     CoroutineScope(applicationContext + Job(applicationContext[Job]) + coroutineContext)
20 }

```

핵심적인 동작은 다음과 같습니다.

- **상위 Job 제한**: 제공된 `CoroutineContext`에서 `Job`을 포함하고 있는 경우 예외가 발생합니다. 이는 `rememberCoroutineScope`가 기존의 상위 `Job`에 종속되는 것이 아닌 독립적인 스코프를 생성하도록 보장합니다.
- **Job 관리**: 스코프를 관리하기 위해 새로운 `Job`이 코루틴 컨텍스트에 추가됩니다.

- **Composer와의 통합:** 함수는 `composer.applyCoroutineContext`를 새로 생성된 Job과 결합하여 코루틴 스코프를 설정합니다.

rememberCoroutineScope 내부 작동 방식

1. **스코프 생성:** `createCompositionCoroutineScope`를 사용하여 새 코루틴 스코프가 생성되고 `CompositionScopedCoroutineScopeCanceller`에 전달됩니다.
2. **컴포지션 생명주기 인지:** `CompositionScopedCoroutineScopeCanceller`는 컴포지션 생명주기를 관찰합니다. 컴포저블이 컴포지션에서 제거되면 메모리 누수를 방지하기 위해 스코프가 취소됩니다.
3. **안정성:** 이와 같은 동작으로 컴포저블이 컴포지션을 종료할 때, 해당 스코프 내에서 시작된 모든 코루틴을 자동으로 취소하도록 보장합니다.

요약

`rememberCoroutineScope` 함수는 연결된 컴포지션의 생명주기를 인지하는 코루틴 스코프를 생성합니다. `CompositionScopedCoroutineScopeCanceller`를 사용하여 컴포저블이 컴포지션에서 제거될 때 스코프를 취소하여 현재 실행 중인 코루틴 스코프를 안전하게 취소하고, 리소스를 적절하게 정리할 수 있습니다. 내부 구조를 이해하면 Jetpack Compose가 UI 개발을 위해 얼마나 안전하고 효율적인 API를 설계했는지 이해할 수 있습니다.

Q) 15. 컴포저블 함수 내에서 발생하는 사이드 이펙트를 어떻게 처리하나요?

Jetpack Compose는 컴포저블의 `recomposition` 범위 외부에서 작업을 처리할 수 있는 여러 [사이드 이펙트 핸들러 API\(side-effect handler APIs\)](#)⁵¹를 제공합니다. 해당 API는 안드로이드 프레임워크와 상호 작용, 컴포지션 이벤트 관리, 및 상태 변경에 기반한 사이드 이펙트 핸들링과 같은 시나리오에서 필수적입니다. 주요한 세 가지 사이드 이펙트 핸들러 API인 `LaunchedEffect`, `DisposableEffect`, `SideEffect`를 살펴보겠습니다.

⁵¹<https://developer.android.com/develop/ui/compose/side-effects>

1. LaunchedEffect: 컴포저블 범위 내에서 suspend 함수 실행하기

LaunchedEffect는 컴포저블의 컴포지션 생명주기 내에서 새로운 코루틴 스코프를 생성하고 실행합니다. 생성된 코루틴은 LaunchedEffect에 전달된 키(key)가 변경되면 취소되고 다시 시작됩니다. 컴포저블이 컴포지션에 진입할 때 수행해야 하는 데이터 로딩하기, 애니메이션 시작 또는 이벤트 수신과 같은 작업에 유용합니다.

주요 특징은 아래와 같습니다.

- 컴포저블이 컴포지션에 진입할 때 한 번 실행됩니다.
- 키가 변경되면 자동으로 취소되고 다시 시작됩니다.
- 컴포지션 생명주기를 인지합니다. 즉, 컴포저블이 컴포지션을 떠날 때 코루틴 작업이 자동으로 취소됩니다.

가령, 리스트에서 특정 아이템을 클릭할 때 네트워크에서 아주 가벼운 데이터를 추가적으로 가져와야 하는 경우 및 애니메이션을 실행해야 하는 경우 LaunchedEffect를 사용하여 처리할 수 있습니다. 또한, LaunchedEffect에 매개변수로 전달된 키 값이 변경될 때마다 작업을 취소하고 다시 실행할 수 있습니다.

그림 262. LaunchedEffect.kt

```
1 var selectedPoster: Poster? by remember { mutableStateOf(null) }
2
3 // selectedPoster가 변경될 때마다 LaunchedEffect 재실행
4 LaunchedEffect(key1 = selectedPoster) {
5     selectedPoster?.let { poster ->
6         // 네트워크에서 추가 정보 가져오기 위해 ViewModel에 이벤트 전송
7         viewModel.fetchPosterDetails(poster.id)
8
9         // Fade 애니메이션 실행
10        .. // suspend 함수 실행
11    }
12 }
```

LaunchedEffect를 사용하여 flow를 안전하게 관찰할 수도 있습니다. LaunchedEffect에 의해 시작된 코루틴은 컴포저블 함수가 컴포지션을 떠날 때 자동으로 취소되어 메모리 누수를

방지합니다. 또한 코루틴은 recomposition 중에 다시 시작되지 않습니다. 사이드 이펙트가 컴포지션의 생명주기와 일치하여 key 값의 영향을 받지 않도록 하려면, Unit 또는 true와 같은 상수 값을 키 매개변수로 전달할 수 있습니다. 이는 공식적으로 권장되는 방법입니다. 따라서, 키가 변경되지 않는 한 효과가 한 번만 실행되도록 보장합니다.

그림 263. LaunchedEffect.kt

```

1 LaunchedEffect(key1 = Unit) { // 키가 변경되지 않으므로 한 번만 실행됨
2     stateFlow
3         .distinctUntilChanged() // 중복 값 필터링
4         .filter { it.marked } // 특정 조건 필터링
5         .collect { value -> // Flow 구독
6             // Flow에서 보내는 값 수신 후 처리
7         }
8 }
```

2. DisposableEffect: 메모리 해지 등이 필요한 사이드 이펙트 실행에 적합

DisposableEffect는 컴포저블의 라이프 사이클에 따라 리소스 해지 및 실행 중인 태스크를 정리하기 위해 데 사용됩니다. LaunchedEffect와 달리 컴포저블이 컴포지션을 떠날 때 리소스를 해제하기 위해 DisposableEffectScope를 제공하고 해당 스코프 안에서 onDispose 람다를 통해 해지 작업을 수행할 수 있습니다.

주요 특징은 아래와 같습니다.

- 리스너, 관찰자 또는 구독과 같은 리소스 관리에 이상적입니다.
- onDispose 콜백으로 적절한 해지 및 리소스 정리를 보장합니다.

예를 들어, 생명주기 이벤트를 기반으로 웹 서버에 애널리틱스 관련 이벤트를 보내야 하는 경우 LifecycleObserver를 사용하여 구현할 수 있습니다. 이때, DisposableEffect를 함께 사용하여 컴포저블이 컴포지션에 진입할 때 관찰자를 등록하고 컴포저블이 컴포지션을 떠날 때 자동으로 등록 해제할 수 있습니다. 이는 리소스의 적절한 정리를 보장하고 메모리 누수를 방지합니다.

그림 264. DisposableEffect.kt

```
1 @Composable
2 fun HomeScreen(
3     lifecycleOwner: LifecycleOwner = LocalLifecycleOwner.current,
4     onStart: () -> Unit, // 'started' 분석 이벤트 전송
5     onStop: () -> Unit // 'stopped' 분석 이벤트 전송
6 ) {
7     // 새 람다가 제공될 때 현재 람다를 안전하게 업데이트
8     val currentOnStart by rememberUpdatedState(onStart)
9     val currentOnStop by rememberUpdatedState(onStop)
10
11    // `lifecycleOwner`가 변경되면 사이드 이펙트를 폐기하고 재설정
12    DisposableEffect(lifecycleOwner) {
13        // 람다를 트리거하는 관찰자 생성
14        val observer = LifecycleEventObserver { _, event ->
15            if (event == Lifecycle.Event.ON_START) {
16                currentOnStart()
17            } else if (event == Lifecycle.Event.ON_STOP) {
18                currentOnStop()
19            }
20        }
21
22        // 생명주기에 관찰자 추가
23        lifecycleOwner.lifecycle.addObserver(observer)
24
25        // 컴포지션을 떠날 때 관찰자 제거
26        onDispose {
27            lifecycleOwner.lifecycle.removeObserver(observer)
28        }
29    }
30 }
```

3. SideEffect: Compose의 상태를 non-Compose 코드로 실행하기

SideEffect는 매 recomposition 직후에 적용해야 하는 작업을 실행하는 데 사용됩니다. 컴포저블이 recompose된 후 실행을 보장하므로 ViewModel이나 외부 라이브러리의 UI 상태 업데이트와 같이 컴포지션의 일부가 아닌 외부 시스템과 Compose 상태를 동기화하는 데 적합합니다.

주요 특징은 아래와 같습니다.

- 매 recomposition 후 실행이 보장됩니다.
- non-Compose 컴포넌트와의 상태 동기화에 유용합니다.

예를 들어, 애널리틱스 라이브러리를 사용하여 커스텀 메타데이터(가령, “user properties”)를 포함하여 사용자 세그먼트를 세분화하는 기능을 개발하려는 경우 SideEffect를 사용하여 현재 사용자의 데이터가 원활하게 애널리틱스 서버로 전달되도록 할 수 있습니다. 따라서, 앱 데이터 분석 도구들이 애플리케이션의 데이터와 동기화 상태를 유지할 수 있도록 하는 등, 여러 시나리오에서 활용될 수 있습니다.

그림 265. SideEffect.kt

```
1 @Composable
2 fun rememberFirebaseAnalytics(user: User): FirebaseAnalytics {
3     val analytics: FirebaseAnalytics = remember {
4         FirebaseAnalytics.getInstance(LocalContext.current) // Context 필요
5     }
6
7     // 컴포지션 이후 현재 User의 userType으로 FirebaseAnalytics 업데이트
8     // 향후 분석 이벤트에 해당 메타데이터가 함께 전달되도록 보장
9     SideEffect {
10         analytics.setUserProperty("userType", user.userType)
11     }
12     return analytics
13 }
```

SideEffect를 사용하는 또 다른 시나리오는 아래 예제와 같이 recomposition이 완료된 후에만 Lottie 애니메이션을 시작하거나 non-Composable 작업을 트리거해야 하는 경우입니다.

그림 266. SideEffect.kt

```
1 // LottieAnimationView 인스턴스가 있다고 가정
2 val lottieAnimationView = rememberLottieAnimationView()
3
4 SideEffect {
5     // 최신 recomposition 트리가 후에만 애니메이션 재생
6     lottieAnimationView.playAnimation()
7 }
```

요약

각 사이드 이펙트 핸들러 API는 고유한 목적을 가지고 있습니다.

- 코루틴 기반 작업을 시작하거나 키 매개변수 변경에 따라 작업을 다시 시작하려면 LaunchedEffect를 사용하는 것이 적합합니다.
- 컴포저블의 컴포지션 생명주기를 인지하고, 리소스를 올바르게 관리하고 폐기하려면 DisposableEffect를 사용하는 것이 적합합니다.
- 매 recomposition 직후에 실행해야 하는 작업을 수행하고, 외부 시스템을 Compose 상태와 동기화하기 위한 목적으로는 SideEffect가 적합합니다.

주요 사이드 이펙트 핸들러 API 3가지에 대해서 살펴보았고, 언제 어떻게 사용해야 하는지 이해하면 더 깔끔한 선언적인 접근 방식을 유지하면서 사이드 이펙트를 효과적으로 관리할 수 있습니다. 각 API와 내부 작동 방식에 대한 더 깊은 이해가 필요하다면 [Understanding the Internals of Side-Effect Handlers in Jetpack Compose](#)⁵²를 읽어보시면 도움이 됩니다.

실전 질문

- Q) LaunchedEffect는 컴포저블에서 suspend 함수를 관리하는 데 어떤 도움이 되며, 키 매개변수 값이 변경되면 어떤 동작이 발생하나요?
- Q) LaunchedEffect 대신 DisposableEffect를 사용해야 하는 경우는 언제인가요?
- Q) SideEffect의 사용 사례를 설명하고 LaunchedEffect와 어떻게 다른지 설명해 주세요.

⁵²<https://medium.com/proandroiddev/understanding-the-internals-of-side-effect-handlers-in-jetpack-compose-d55fbf914fde>

Q) 16. `rememberUpdatedState`는 왜 사용하고 어떻게 작동하나요?

`rememberUpdatedState`⁵³ 함수는 컴포지션 컨텍스트 내에서 람다 함수에 대한 상태 업데이트를 안전하게 처리하는 런타임 API입니다. 최초 컴포지션을 통해서 생성되었더라도 람다나 콜백의 상태 값에 대해 항상 최신 상태를 유지하도록 보장합니다.

컴포저블에서 콜백이나 람다를 생성할 때 해당 콜백 내에서 참조된 상태 값은 함수가 이미 생성된 경우 자동으로 업데이트되지 않을 수 있습니다. 이를 위해 `rememberUpdatedState`가 유용하게 사용됩니다. 람다 함수 내에서 항상 최신 상태 값을 사용할 수 있도록 하는 메커니즘을 제공하여 람다 내부의 상태가 업데이트되지 않는 잠재적인 문제를 방지합니다.

동작 구조

`rememberUpdatedState` API는 상태의 가장 최근 값을 기억하고 상태가 변경될 때마다 업데이트하고, 컴포저블 또는 람다 내에서 값을 읽고 가장 최신의 상태를 반영하는 `State<T>` 객체를 반환합니다.

함수 시그니처는 다음과 같습니다.

```
1 @Composable  
2 fun <T> rememberUpdatedState(newValue: T): State<T>
```

사용 사례

`rememberUpdatedState`는 다음과 같은 시나리오에서 특히 유용합니다.

1. **콜백이 사이드 이펙트 내부로 전달될 때**: 사이드 이펙트 내에서 람다나 콜백이 최초 컴포지션을 통해 이미 생성되었지만, 지속적으로 새로 업데이트된 람다 및 콜백을 실행해야 할 때 유용합니다.
2. **애니메이션 또는 사이드 이펙트 API**: `recomposition`을 넘어 지속되는 `LaunchedEffect`, `DisposableEffect` 또는 애니메이션과 함께 사용할 때 유용합니다.

설명만 읽어서는 처음에 이해가 잘 안 될 수 있습니다. 바로 `rememberUpdatedState`에 대한 예시 코드를 살펴보겠습니다.

⁵³<https://developer.android.com/develop/ui/compose/side-effects#rememberupdatedstate>

그림 267. RememberUpdatedStateExample.kt

```
1 @Composable
2 fun TimerWithCallback(
3     onTimeout: () -> Unit, // 시간이 초과되었을 때 호출될 콜백
4     timeoutMillis: Long = 5000L // 타임아웃 시간 (밀리초)
5 ) {
6     // onTimeout 콜백의 최신 버전을 기억
7     val currentOnTimeout by rememberUpdatedState(onTimeout)
8
9     // TimerWithCallback의 생명주기와 일치하는 효과 생성
10    // TimerWithCallback이 recompose되더라도 딜레이가 다시 시작되지 않음
11    LaunchedEffect(true) { // key를 true로 설정하여 한 번만 실행
12        delay(timeoutMillis) // 딜레이가 발생하는 동안 `onTimeout`의 구현체가 바뀐다면?
13        currentOnTimeout() // 최신으로 업데이트 된 콜백이 사용되도록 보장
14    }
15
16    Text(text = "Timer running for $timeoutMillis milliseconds")
17 }
```

설명

rememberUpdatedState를 사용하지 않는다면 타이머가 실행되는 동안 컴포지션의 실행될 당시의 구현체 값은 캡처되었기 때문에, onTimeout 콜백의 구현체 자체가 변경되면 초기에 딜레이가 발생하기 이전의 콜백이 호출될 수 있습니다. 즉, 딜레이가 발생하는 동안 onTimeout의 구현체가 변경되면, 구현체가 변경되기 이전의 onTimeout 함수가 계속 호출되면서 버그를 유발하는 것입니다. 이를 방지하기 위해 rememberUpdatedState를 사용하여 onTimeout 콜백은 항상 최신 상태로 업데이트하고, 딜레이가 완료되었을 때 가장 최신 함수가 호출되도록 보장합니다.

주요 이점

- **오래된 상태 방지(Avoids Stale State)**: 장기적인 작업을 수행하는 사이드 이펙트 내에서 상태에 대해 오래된 참조로 인한 문제를 방지합니다.

- **안전한 컴포지션 처리(Safe Composition Handling)**: LaunchedEffect 또는 DisposableEffect와 같은 컴포지션의 생명주기를 인식하는 API와 원활하게 작동합니다.
- **간단한 통합(Simple Integration)**: 람다 상태가 항상 최신 값을 반영하도록 보장하기 위해 람다를 단순히 rememberUpdatedState으로 감싸주기만 하면 됩니다.

요약

rememberUpdatedState는 장기적인 작업이 수행되어야 하는 사이드 이펙트 내에서 콜백이나 람다 함수에 대한 상태 업데이트를 관리하기 위한 사이드 이펙트 핸들러 API 중 하나입니다. 가장 최근 상태 값이 사용되도록 보장하여 오래된 컴포지션 초기에 캡처된 람다 함수가 실행되는 잠재적인 문제를 방지할 수 있습니다.

실전 질문

Q) LaunchedEffect를 사용하여 어느 정도의 딜레이 이후 람다 함수를 트리거하는 경우에 최신 상태를 반영하는 람다가 호출되도록 어떻게 보장할 수 있나요?

Pro Tips for Mastery: rememberUpdatedState 내부 구조

rememberUpdatedState의 동작을 살펴보았는데, 처음에는 좀 복잡해 보일 수 있습니다. 그러나 내부 구현은 의외로 매우 간단합니다.

그림 268. rememberUpdatedState.kt

```

1 @Composable
2 fun <T> rememberUpdatedState(newValue: T): State<T> =
3     // remember를 사용하여 State 객체를 생성하고 기억합니다.
4     remember { mutableStateOf(newValue) }
5     // 매 recomposition마다 최신 newValue로 State의 값을 업데이트합니다.
6     .apply { value = newValue }
```

위에서 볼 수 있듯이 rememberUpdatedState는 제공된 newValue를 상태로 저장하고 apply 스코프 함수를 사용하여 값을 업데이트합니다. rememberUpdatedState를 사용하는 컴포저블 함수가 recompose 될 때마다 함수가 호출되고 이전에 기억된 상태가 새 newValue 매개변수로 업데이트됩니다. API 사용법이 표면적으로는 복잡해 보이지만 내부적으로는 작동 방식이 간단합니다.

Q) 17. `produceState`의 목적은 무엇이며 어떻게 작동하나요?

`produceState`⁵⁴ 함수는 새로 시작된 코루틴에 의해 값이 생성되는 State 객체를 만드는 데 도움이 됩니다. 비동기적으로 불러와야 하는 데이터나 코루틴을 활용하여 연산이 필요한 동작과 Compose 간의 다리 역할을 합니다. 이는 비동기 작업에 의존하는 상태를 관리하거나 non-Compose 상태를 Compose 상태로 변환해야 할 때 특히 유용합니다.

컴포저블이 관찰할 수 있는 State 객체를 생성하고, 상태 값을 업데이트하기 위해 생산자 (producer) 코루틴을 실행하며, 컴포저블이 컴포지션을 떠날 때 코루틴 스코프를 자동으로 취소합니다. 이 방식을 통해 컴포저블 함수 내에서 비동기 데이터를 처리하고 Compose UI에 원활하게 상태를 반영할 수 있는 간단하고 선언적인 방법이 가능합니다.

구문 (Syntax)

`produceState` 함수의 시그니처는 다음과 같습니다.

그림 269. `ProduceState.kt`

```

1 @Composable
2 fun <T> produceState(  

3   initialValue: T, // 상태의 초기값  

4   vararg keys: Any?, // 코루틴을 재시작을 위한 키 값  

5   producer: suspend ProduceStateScope<T>.() -> Unit // 상태 값을 업데이트하는 suspend  

6       ↳ 람다  

6   ): State<T> // 관찰 가능한 State 객체 반환
```

- `initialValue`: 생산자가 코루틴을 실행하고 상태를 업데이트를 시작하기 전 초기값입니다.
- `keys`: 생산자의 코루틴은 이 키값에 의존성을 갖습니다. 키 값 중 하나라도 변경되면 기존에 돌아가던 생산자 코루틴은 취소되고, 다시 시작됩니다.
- `producer`: 코루틴 동작을 수행하고 상태를 업데이트하는 `suspend` 람다입니다. `ProduceStateScope` 내에서 실행됩니다.

⁵⁴<https://developer.android.com/develop/ui/compose/side-effects#producestate>

사용 예제

아래는 `produceState`를 사용하여 네트워크에서 데이터를 가져오는 실제 예시입니다.

그림 270. `ProduceStateExample.kt`

```
1  @Composable
2  fun UserProfile(userId: String, viewModel: UserViewModel) { // ViewModel 주입 (예시)
3      // userId가 변경되면 produceState 재실행
4      val userState: State<User?> by produceState<User?>(initialValue = null, key1 =
5          userId) {
6          // 생산자 랍다: 비동기 작업 수행 및 상태 업데이트
7          value = viewModel.fetchUserFromNetwork(userId) // 간단한 네트워크 요청 실행
8      }
9
10     // 로딩 상태 또는 사용자 정보 표시
11     if (userState == null) {
12         Text("Loading...")
13     } else {
14         Text("User: ${userState?.name}")
15     }
16
17     // 예시 ViewModel 함수
18     class UserViewModel : ViewModel() {
19         suspend fun fetchUserFromNetwork(userId: String): User {
20             // 네트워크 지연 시뮬레이션 (실제는 네트워크 통신 로직이 들어가야 함)
21             delay(2000)
22             return User(name = "skydives ($userId)") // userId 포함하여 반환
23         }
24     }
25
26     data class User(val name: String)
```

예제 설명

- produceState는 초기값이 null인 State<User?> 객체를 생성합니다. userId가 키로 사용되어 userId가 변경되면 코루틴이 재시작됩니다.
- 비동기 로직인 생산자 코루틴은 사용자 데이터를 비동기적으로 가져와 Compose의 상태로 만들고 해당 상태 값인 value를 업데이트합니다.
- value가 변경되면 UserProfile 컴포저블이 recompose되어 업데이트된 데이터를 반영합니다.

produceState의 이점

- 선언적(Declarative):** 비동기 작업을 실행하고 그 결과를 Compose의 상태로 만들기 위해 깔끔하고 Compose의 선언적 방식을 있는 그대로 사용합니다.
- 컴포지션 생명주기 인지(Composition-aware):** 컴포저블이 컴포지션을 떠날 때 코루틴을 자동으로 취소하여 리소스 누수 위험을 방지합니다.
- 유연성(Flexible):** 외부 suspend 함수와도 잘 작동하고 의존성(키)이 변경될 때 코루틴 작업을 다시 시작할 수 있습니다.

모범 사례

생산자 코루틴이 필요할 때만 다시 시작되도록 의미 있는 keys 값을 사용해야 합니다. 또한 produceState는 withContext를 사용하여 명시적으로 코루틴 디스패처를 변경하지 않으면 기본적으로 메인 스레드에서 실행됩니다. 따라서, 메인 스레드를 차단하지 않으려면 produceState 내에서 직접적으로 무겁거나 장기적으로 실행되는 코루틴 작업을 수행하면 안 되고, 또는 withContext를 사용하여 명시적으로 디스패처를 변경해야 합니다.

요약

produceState는 코루틴 작업이 컴포지션의 생명주기를 인식하도록 하며, 선언적 방식으로 비동기 처리 로직을 Compose의 상태로 만드는 사이드 이펙트 핸들러 API 중 하나입니다. 코루틴 작업으로부터 Compose의 상태 값을 생성할 수 있도록 하여 비동기로 데이터를 불러오고 곧바로 recompose를 수행하여 Compose UI에 반영하는 과정을 단순화합니다.

실전 질문

Q) 컴포저블 함수에서 코루틴 작업을 시작하고 그 결과를 상태로 관찰해야 하는 시나리오에서 LaunchedEffect나 rememberCoroutineScope를 사용하지 않고 구현하는 방법에 대해 설명해 주세요.

💡 Pro Tips for Mastery: produceState 내부 구조

produceState 함수의 내부 구현을 살펴보면 몇 가지 흥미로운 사실이 있습니다.

그림 271. produceState.kt

```
1 @Composable
2 fun <T> produceState(
3     initialValue: T,
4     key1: Any?, // 키 매개변수 (가변 인자 아님)
5     producer: suspend ProduceStateScope<T>.() -> Unit
6 ): State<T> {
7     // remember와 mutableStateOf를 사용하여 상태 생성
8     val result = remember { mutableStateOf(initialValue) }
9     // LaunchedEffect를 사용하여 생산자 코루틴 안전하게 실행
10    LaunchedEffect(key1) { // 키가 변경되면 재시작
11        // ProduceStateScopeImpl 인스턴스를 사용하여 생산자 실행
12        ProduceStateScopeImpl(result, coroutineContext).producer()
13    }
14    return result // 생성된 State 객체 반환
15 }
16
17 // ProduceStateScope 인터페이스를 구현하고 있음
18 private class ProduceStateScopeImpl<T>(
19     state: MutableState<T>, // 상태 업데이트를 위한 MutableState 참조
20     override val coroutineContext: CoroutineContext // 코루틴 컨텍스트
21 ) : ProduceStateScope<T>, MutableState<T> by state { // MutableState 위임
22
23     // 생산자 코루틴이 취소될 때 정리 작업을 수행하는 suspend 함수
```

```

24    override suspend fun awaitDispose(onDispose: () -> Unit): Nothing {
25        try {
26            // 코루틴이 취소될 때까지 일시 중단
27            suspendCancellableCoroutine<Nothing> { }
28        } finally {
29            // 취소 시 onDispose 람다 실행
30            onDispose()
31        }
32        // 실제로는 여기까지 도달하지 않음
33        throw RuntimeException("awaitDispose returned")
34    }
35 }

```

`produceState` 함수는 내부적으로 `remember`와 `mutableStateOf`를 사용하여 상태를 생성합니다. 또한 `LaunchedEffect`를 활용하여 `producer`를 새 코루틴 스코프에서 안전하게 시작합니다. 따라서, 해당 코루틴 스코프는 컴포저블 함수가 컴포지션을 나갈 때 자동으로 취소되어 적절한 리소스 관리와 잠재적 메모리 누수를 방지합니다.

Q) 18. `snapshotFlow`를 사용해 본 경험이 있을까요? 사용 시 주의 사항은 무엇인가요?

`snapshotFlow`⁵⁵는 Compose의 상태를 Flow로 변환하는 함수입니다. Compose가 내부적으로 상태 변경을 효율적으로 관리하고 관찰하는 데 사용하는 Snapshot 시스템 내에서 변경 사항을 관찰합니다. 관찰된 상태가 변경될 때마다 Flow는 업데이트된 값을 내보냅니다.

간단히 말해, `snapshotFlow`는 Compose의 상태 변경을 수신하고 이 값을 Flow로 방출하여 상태 업데이트를 관리합니다. 즉, 반응형 기반인 Compose의 상태를 또 다른 반응형 기반인 Flow로 전환하여 업데이트된 값을 관찰할 수 있도록 하는 API입니다.

snapshotFlow의 주요 특징

- **상태 관찰(State Observation):** Snapshot 시스템을 사용하여 Compose의 상태 변화를 관찰합니다.

⁵⁵<https://developer.android.com/develop/ui/compose/side-effects#snapshotFlow>

- **스레드 안전성(Thread Safety)**: 상태 읽기 및 쓰기가 Compose의 스냅샷 스코프 내에서 발생하도록 보장하여 경쟁 조건(race condition)을 방지합니다.
- **유휴 건너뛰기(Idle Skipping)**: 상태 값이 변경될 때만 Flow 값으로의 방출이 발생하고 recomposition이 발생하지 않는 중에는 업데이트를 건너뛰도록 보장합니다.
- **취소 인지(Cancellation-Aware)**: Flow를 수집하는 코루틴이 취소될 때 구독 또한 자동으로 취소하여, 컴포지션의 생명주기를 인지하는(composition-aware) 형태로 동작하여 부적절한 메모리 누수를 방지합니다.

유즈 케이스

- **코루틴과의 인터페이스(Interfacing with Coroutines)**: 변환, 플로우 결합 또는 필터링과 같은 추가적인 Flow의 연산 작업이 필요한 경우 Compose의 상태를 코루틴 플로우로 변경하여 여러 가지 이점을 얻을 수 있습니다.
- **비 UI 사이드 이펙트(Non-UI Side Effects)**: 애널리틱스 이벤트 전송이나 백엔드 호출이 필요한 경우와 같이 UI에 직접적으로 관련이 없는 사이드 이펙트 작업을 수행하기에 적합합니다.

snapshotFlow 사용법

다음은 snapshotFlow를 사용하여 상태를 관찰하고 변경될 때마다 작업을 수행하는 코드입니다.

그림 272. SnapshotFlowExample.kt

```
1 @Composable
2 fun SnapshotFlowExample(viewModel: MyViewModel) {
3     val count by viewModel.count.collectAsState()
4
5     LaunchedEffect(Unit) { // 컴포지션 진입 시 단 한 번만 실행
6         snapshotFlow { count } // count 상태 변경 감지
7         .collect { value ->
8             // 최신 값으로 사이드 이펙트 작업 수행
9             // 예: 로그 기록, 분석 이벤트 전송 등 무겁지 않은 동작
10            println("Count value changed to: $value")
11        }
12    }
13 }
```

```
11         }
12     }
13
14     Text(text = "Count: $count")
15 }
```

또 다른 시나리오는, 사용자가 리스트를 스크롤하여 필요할 때마다 Pagination에 필요한 데이터를 불러와야 할 때 snapshotFlow를 사용하여 해결할 수 있습니다. 해당 접근 방식은 shouldLoadMore 조건이 true가 될 때만 pagination을 위한 요청을 전송하여 중복 적으로 동일한 아이템 셋 요청 작업을 방지합니다.

그림 273. SnapshotFlowExample.kt

```
1 val listState = rememberLazyListState()
2 val threshold = 2
3
4 LazyColumn(state = listState) {
5     // ... 아이템 목록 ...
6 }
7
8 // 현재 ViewModel이 이미 추가 목록을 불러오고 있는지에 대한 상태 관찰
9 val isLoadingShorts by shortsViewModel.isLoadingShorts.collectAsStateWithLifecycle()
10 // Pagination을 구현하기 위해 추가 목록을 불러 와야하는지 말아야 하는지 판단
11 val shouldLoadMore by remember {
12     derivedStateOf {
13         val totalItemCount = lazyListState.layoutInfo.totalItemCount
14         val lastVisibleItemIndex =
15             lazyListState.layoutInfo.visibleItemsInfo.lastOrNull()?.index ?: 0
16         (lastVisibleItemIndex + threshold >= totalItemCount) && !isLoadingShorts
17     }
18 }
19
20 // listState가 변경될 때마다 LaunchedEffect 재실행
21 LaunchedEffect(listState) {
22     snapshotFlow { shouldLoadMore } // 첫 번째 보이는 아이템 인덱스 관찰
```

```
22     .map { index -> index > 0 } // 첫 번째 아이템을 지나 스크롤했는지 여부로 변환
23     .distinctUntilChanged() // 값이 실제로 변경될 때만 방출
24     .filter { it } // shouldLoadMore가 true일 때만 진행
25     .collect {
26         // pagination 구현을 위해 추가 항목을 불러오는 이벤트 전송
27         shortsViewModel.handleShortsEvent(event = ShortsEvent.LoadMore)
28     }
29 }
```

이 예제는 `snapshotFlow`를 활용하여 `LazyColumn`의 스크롤 상태를 모니터링하고 사용자가 현재 리스트의 특정 아이템(전체 아이템 사이즈 - threshold)이 화면에 보여지는 순간 추가 항목 로드를 위해서 이벤트를 호출합니다. 특히, `map`, `distinctUntilChanged`, `filter`와 같은 연산자를 사용하여 추가 이벤트를 트리거해도 되는지를 판단하고, 스크롤 작업당 이벤트가 한 번만 발생하도록 보장합니다.

`snapshotFlow` 내부 작동 방식

1. 람다 매개변수로 제공된 Compose의 상태를 Snapshot 관찰자 내에서 지속적으로 관찰합니다.
2. 람다 매개변수에 상태 변수가 들어오는 순간 Compose는 해당 상태에 대해서 종속성을 등록합니다.
3. 상태가 변경되면 Compose는 `snapshotFlow`에 업데이트 사항을 알리고 새 값을 방출하도록 트리거합니다.

명심해야 할 사항

`snapshotFlow`를 사용할 때 다음 사항을 염두에 두어야 합니다.

- **스레드 안전성 및 수집(Thread Safety and Collection)**: 수집된 플로우가 일반적으로 `LaunchedEffect`를 사용하는 코루틴 스코프 내에서 처리되도록 하여 컴포지션 외부에서 수집하는 것을 방지하고 메모리 누수를 방지합니다.
- **방출 빈도(Emission Frequency)**: `snapshotFlow`는 람다 내에서 읽은 스냅샷 상태가 변경될 때마다 값을 방출하며, 이는 생각보다 자주 발생할 수 있습니다.

`distinctUntilChanged()` 또는 `debounce()`와 같은 연산자를 사용하여 불필요한 방출을 줄이고 성능을 최적화합니다.

- **스냅샷 격리(Snapshot Isolation):** Flow는 Compose의 스냅샷 시스템에 최대한 동기화되어 동작하는데, 즉, 값은 격리된 스냅샷을 기반으로 방출됩니다. 스냅샷 생명주기 내에서 예상되는 동작을 보장하기 위해 다른 Flow 또는 suspend 함수와 결합할 때는 주의해야 합니다.

요약

`snapshotFlow`는 상태 관찰 시스템과 Kotlin의 코루틴 Flow를 연결하는 사이드 이펙트 핸들러 API 중 하나입니다. Compose의 상태 변경에 반응할 수 있는 Flow를 생성하도록 하여 분석, 데이터 동기화 및 기타 비 UI 동작에 대한 시나리오를 처리하는데 적합합니다. `snapshotFlow`는 폐나 현업에서 자주 사용되기도 하고 강력한 API이기 때문에 효과적으로 사용하는 방법을 이해하면, 앱 개발에 있어서 더 다양한 복잡한 상황을 단순화하는데 활용할 수 있습니다.

실전 질문

Q) `snapshotFlow`를 사용하는 것이 선호되는 시나리오는 무엇이며, 이는 Compose의 스냅샷 시스템과 어떠한 연관이 있나요? 또한, Flow에서 값을 방출하는 동작을 어떻게 최적화하시겠습니까?

Q) 19. `derivedStateOf`가 필요한 시나리오는 무엇이고, `recomposition` 최적화에 어떻게 도움이 되나요?

`derivedStateOf`는 하나 이상의 상태 객체에서 파생된 값을 계산하는 컴포저블 API입니다. 종속 상태 중 하나가 변경될 때만 파생된 값이 다시 계산되도록 보장하여 반응형 상태 관계를 관리하는 효과적인 도구입니다.

주요 특징 중 하나는 종속 상태가 자주 업데이트되더라도 계산된 값 자체가 변경될 때만 `recomposition`을 트리거하여 `recomposition`을 최적화하는 것입니다. 이로 인해 `derivedStateOf`는 빈번한 상태 업데이트가 있는 시나리오에서 성능을 개선하고 불필요한 `recomposition`을 방지하는 데 유용합니다.

`derivedStateOf`는 중복된 `recomposition`을 방지하도록 설계되었지만 연산 작업이 요구되기 때문에 약간의 오버헤드를 동반합니다. 따라서 `recomposition` 방지가 중요한 상황에서만 신중하게 사용하여 파생된 상태 유지 관리의 추가 비용보다 이점이 크도록 해야 합니다.

 **Additional Tips:** derived란 파생된 이라는 사전적 의미를 갖습니다. 즉, 'derivedStateOf'는 '~으로부터 파생된 상태'라는 뜻을 지닙니다. API의 본질을 이해하는 가장 빠른 방법은 API의 이름을 이해하는 것입니다.

derivedStateOf 사용 시기

- **파생 데이터(Derived Data):** 필터링된 목록이나 결합된 텍스트와 같이 기존 상태 값에서 연산이 필요할 때 유용합니다.
- **Recomposition 방지(Avoiding Recomposition):** 파생된 값이 자주 변경되는 상태에 의존하지만 파생된 값이 변경될 때만 recomposition을 원할 때 사용하기 적합합니다.

derivedStateOf 사용 방법

다음은 검색 쿼리를 기반으로 항목 목록을 필터링하는 실제 예시입니다.

그림 274. DerivedStateOfExample.kt

```
1 @Composable
2 fun DerivedStateExample(items: List<String>, searchQuery: String) {
3     // searchQuery 또는 items가 변경될 때만 filteredItems 재계산
4     val filteredItems by remember(searchQuery, items) {
5         derivedStateOf {
6             items.filter { it.contains(searchQuery, ignoreCase = true) }
7         }
8     }
9
10    Column {
11        Text("Search results:")
12        filteredItems.forEach { item ->
13            Text(item)
14        }
15    }
16}
```

```
15     }  
16 }
```

위의 코드는 다음과 같은 동작을 포함하고 있습니다.

- `derivedStateOf`라는 API의 이름처럼 `filteredItems`는 `items` 및 `searchQuery`를 통해 파생된 상태입니다.
- `derivedStateOf`는 `searchQuery` 또는 `items`가 변경될 때만 목록이 다시 계산되도록 보장합니다.

derivedStateOf 작동 방식

1. `derivedStateOf`의 람다 매개변수 내에서 사용된 모든 상태들의 변화를 관찰합니다.
2. 관찰된 상태 중 하나가 변경되면 새 값을 계산합니다.
3. 새로 계산된 값이 이전 값과 다른 경우에만 `recomposition`을 트리거합니다.

명심해야 할 주요 사항

- 파생된 상태가 `recomposition`으로부터 값을 유지하도록 하려면 항상 `remember`와 함께 `derivedStateOf`를 사용합니다.
- 원활한 성능을 보장하기 위해 `derivedStateOf`에 아주 무거운 연산식을 사용하지 않는 것이 좋습니다. 오히려 해당 계산을 통한 오버헤드가 `recomposition`을 통해 UI를 업데이트하는 비용보다 더 크면 성능에 역효과가 날 수 있습니다.
- 고로, 불필요한 `recomposition`을 피하는 것이 중요한 경우에만 신중하게 사용하는 것이 좋습니다.

고급 예제: 실시간으로 파생 상태 계산하기

그림 275. RealTimeDerivedStateExample.kt

```
1 @Composable
2 fun RealTimeDerivedStateExample() {
3     var text by remember { mutableStateOf("") }
4     // text 상태로부터 입력 유효성을 계산
5     val isValid by remember {
6         derivedStateOf { text.length >= 5 }
7     }
8
9     Column {
10         TextField(value = text, onValueChange = { text = it })
11         // isValid 상태가 변경될 때만 아래 Text가 recompose됨
12         if (isValid) {
13             Text("Valid input")
14         } else {
15             Text("Input must be at least 5 characters")
16         }
17     }
18 }
```

위 예제를 통해 다음 내용을 살펴볼 수 있습니다.

- 입력 유효성(isValid)은 텍스트 상태에서 파생됩니다.
- recomposition은 텍스트가 변경될 때마다 트리거 되는 것이 아닌, isValid가 변경될 때만 발생합니다.

잘못된 사용법

두 Compose 상태 객체를 결합할 때 흔히 저지르는 실수는 “상태를 파생”하기 때문에 항상 derivedStateOf를 사용해야 한다고 가정하는 것입니다. 아래 예시와 같이 생각보다 derivedStateOf는 대부분의 경우 불필요한 오버헤드를 유발하고 실질적으로 연산의 효율을 개선하지 않는 경우가 많습니다.

그림 276. IncorrectDerivedStateExample.kt

```

1 // derivedStateOf의 잘못된 사용 사례입니다.
2 var firstName by remember { mutableStateOf("") }
3 var lastName by remember { mutableStateOf("") }
4
5 // 아래 코드는 오버헤드만 추가할 뿐 어떠한 성능적인 개선이 없습니다.
6 val fullNameBad by remember { derivedStateOf { "$firstName $lastName" } }
7 // 그냥 아래와 같이 사용하는 것이 더 효율적입니다.
8 val fullNameCorrect = "$firstName $lastName"

```

위의 예제에서 `firstName` 및 `lastName`이라는 값은 일반적으로 변경될 가능성이 적으며, `recomposition`을 자주 유발하지 않기 때문에 단순히 문자열을 합치기 위한 수준의 작업을 `derivedStateOf`로 래핑하면 오히려 불필요한 복잡성과 오버헤드가 추가됩니다. 따라서 파생된 값을 이루는 상태들이 자주 업데이트되고, 상태가 변경될 때 다시 계산하여 명확한 성능적인 이점이 있는 경우에만 `derivedStateOf`를 사용하는 것이 좋습니다.

요약

`derivedStateOf`는 여러 상태들로부터 최적화된 파생 상태를 만들기 위한 사이드 이펙트 핸들러 API 중 하나입니다. 연산이 발생할 때마다 `recomposition`을 하는 것이 아니라, 필요할 때만 `recomposition`이 발생하도록 하여 UI를 더 효율적으로 만들고 코드를 선언적 UI 스타일에 한층 더 가깝게 만듭니다. 반면, `derivedStateOf`를 잘못 사용하면 오히려 오버헤드가 발생할 수 있기 때문에, 적절한 상황에 잘 사용하면 Compose 앱의 성능을 최적화시키면서 여러 가지 조합된 상태로부터 파생 상태를 효과적으로 생성할 수 있습니다.

실전 질문

Q) 어떤 시나리오에서 `derivedStateOf`를 사용하는 것이 효과적인가요? 혹은 어떤 시나리오에서 `derivedStateOf`를 사용하지 말아야 하나요?

Q) 20. 컴포저블 함수 또는 컴포지션의 생명주기는 어떻게 되나요?

컴포저블 함수는 안드로이드 View나 Activity와 같은 전통적인 생명주기를 가지지 않습니다. 대신 Compose 런타임에 의해 구동되는 **컴포지션을 인지하는 생명주기**(composition-aware

`lifecycle`) 를 갖습니다.

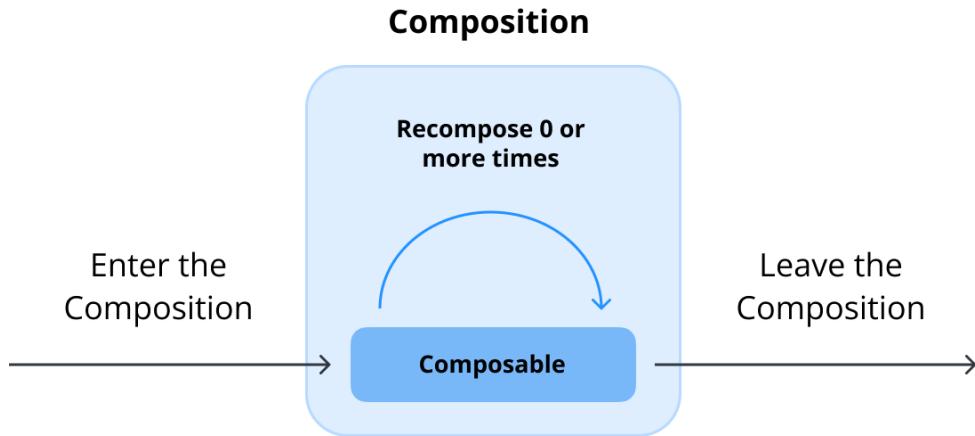


그림 277. `composable-lifecycle`

이 생명주기는 UI 상태가 변경됨에 따라 컴포저블 함수가 효율적으로 호출되고, recompose 되고, 폐기되도록 보장합니다. 컴포저블 함수의 생명주기를 각 단계별로 살펴보도록 하겠습니다.

1. 초기 컴포지션 (Initial Composition)

컴포저블 함수가 최초로 실행되는 단계입니다. 이 단계 동안 아래와 같은 동작이 수행됩니다.

- 함수는 주어진 상태를 기반으로 초기 UI 컴포넌트를 생성합니다.
- LaunchedEffect 또는 remember와 같은 사이드 이펙트(side-effect)가 초기화되고 향후 recomposition 발생 시 데이터 복원을 위해 필요한 데이터를 메모리에 저장합니다.
- UI 계층 구조가 구축되고 컴포지션 트리에 추가됩니다.

예를 들어, 아래 Greeting 함수는 name이라는 주어진 매개변수 값에 대해 최초로 컴포지션을 구성합니다.

그림 278. Initial Composition.kt

```
1 @Composable
2 fun Greeting(name: String) {
3     Text(text = "Hello, $name!")
4 }
```

2. 리컴포지션 (Recomposition)

주어진 매개변수에 다른 값이 들어오거나, 컴포저블 함수 내의 상태에 변화가 감지되었을 때 Recomposition이 발생합니다. Recomposition은 다음과 같은 동작을 따릅니다.

- 컴포지션 트리 전체를 recompose 하는 것이 아니라, 업데이트가 필요한 부분만 recompose 됩니다 (스마트 Recomposition).
- Compose는 업데이트가 필요하지 않은 UI 트리 부분을 건너뛰어 성능을 최적화합니다.
- remember 등과 같이 메모리에 값을 저장하는 사이드 이펙트 API의 경우는 값이 recomposition이 발생해도 유지됩니다.

예를 들어, 아래의 코드에서 count라는 상태가 변경되면 버튼 내의 Text만 recompose됩니다.

그림 279. Recomposition.kt

```
1 @Composable
2 fun Counter() {
3     var count by remember { mutableStateOf(0) }
4
5     Button(onClick = { count++ }) {
6         Text(text = "Clicked $count times")
7     }
8 }
```

3. 컴포지션 떠나기 (Leaving the Composition)

컴포저블 함수가 컴포지션에서 제거될 때(가령, 다른 화면으로 이동하는 등), 컴포지션을 떠납니다. 이 단계 동안 다음과 같은 동작이 수행됩니다.

- 컴포저블에 연결된 모든 리소스가 자동으로 정리됩니다.
- DisposableEffect와 같은 사이드 이펙트의 경우 리소스를 해제합니다.
- rememberCoroutineScope의 경우 현재 실행 중인 코루틴 작업을 안전하게 취소합니다.

예를 들어, DisposableEffect API를 사용 중인 컴포저블 함수가 컴포지션을 떠날 때 onDispose 블록이 호출됩니다.

그림 280. Leaving Composition.kt

```
1 @Composable
2 fun DisposableExample() {
3     DisposableEffect(Unit) {
4         println("컴포지션 진입")
5         onDispose {
6             println("컴포지션 떠남")
7         }
8     }
9     Text(text = "사용 중인 컴포저블")
10 }
```

컴포지션 생명주기의 주요 사항

- **컴포지션 단계(Phases of Composition)**: 컴포지션 생명주기는 초기 컴포지션, recomposition, 폐기의 세 가지 주요 단계로 구성됩니다. 초기 컴포지션은 컴포저블 함수가 컴포지션 트리에 들어갈 때 발생하며, recomposition은 상태 변경이 특정 UI 컴포넌트에 대한 업데이트를 필요로 할 때 발생합니다.
- **건너뛰기 및 최적화(Skipping and Optimization)**: Compose는 상태값 변경에 대해 업데이트하지 않아도 되는 컴포저블 함수에 대한 recomposition을 지능적으로 건너뛰어 불필요한 UI 업데이트를 줄입니다. 이는 remember, derivedStateOf 및 Stability와 같은 메커니즘을 통해 성능을 향상시킬 수 있습니다.

- **폐기 및 정리(Disposal and Cleanup)**: 컴포저블 함수가 컴포지션을 떠나면 UI 트리에서 제거됩니다. LaunchedEffect에서 실행된 코루틴이나 DisposableEffect를 통해 수행하는 구독 패턴 같이 사이드 이펙트가 시작되었거나, remember를 사용하여 상태가 생성된 경우 메모리 누수를 방지하기 위해 적절하게 정리되는데, 이를 컴포지션을 인지(composition-aware)하고 있다고 부릅니다.

요약

컴포저블 함수의 생명주기는 **초기 컴포지션, recomposition, 컴포지션 떠나기**의 세 가지 주요 단계로 이루어져 있습니다. 각 단계는 효율적인 렌더링, 반응형 업데이트 및 리소스의 적절한 정리를 보장하기 위해 중요한 역할을 합니다. 전통적인 안드로이드의 Activity, Fragment, ViewModel 등의 생명주기를 이해해야 하듯이, 컴포저블 함수의 생명주기를 올바르게 이해해야 효율적인 UI 컴포넌트를 설계할 수 있습니다.

실전 질문

Q) 컴포저블 함수의 각 생명주기를 단계별로 설명하고 상태 변경 시 Compose가 recomposition을 어떻게 처리하는지 설명해 주세요.

Q) 21. SaveableStateHolder에 대해서 설명해 주세요. 그리고 어떤 시나리오에서 유용한가요?

Jetpack Compose에서 내비게이션 시스템과 같이 동적이면서 다중으로 구성된 화면을 관리할 때 개별적인 화면의 상태를 보존하고 복원하는 것은 생각보다 까다롭습니다. 이때 **SaveableStateHolder**⁵⁶가 유용할 수 있는데, 컴포저블이 내비게이션 또는 화면 전환과 같이 컴포지션에서 일시적으로 제거되더라도 상태를 유지하도록 보장합니다.

SaveableStateHolder는 고유 키와 연결된 컴포저블의 상태를 관리하고 보존하기 위해 rememberSaveable과 함께 작동하는 Compose Runtime API입니다. 컴포저블이 컴포지션에서 제거되면(가령, 새 화면으로 이동 등) 해당 상태가 저장되고 컴포지션에 다시 진입할 때 자동으로 복원됩니다.

⁵⁶<https://developer.android.com/reference/kotlin/androidx/compose/runtime/saveable/SaveableStateHolder>

예제： 내비게이션과 함께 SaveableStateHolder 사용하기

아래는 내비게이션 시스템을 직접 구현한다고 했을 때 SaveableStateHolder를 사용하여 상태를 관리하는 방법을 보여줍니다. 각 화면은 컴포저블 간의 전환이 발생해도 상태를 독립적으로 유지합니다.

그림 281. SaveableStateHolderNavigation.kt

```
1 @Composable
2 fun <T : Any> Navigation(
3     currentScreen: T, // 현재 표시할 화면을 나타내는 키 (Any 타입)
4     modifier: Modifier = Modifier,
5     content: @Composable (T) -> Unit // 현재 화면 키에 해당하는 컴포저블 콘텐츠
6 ) {
7     // SaveableStateHolder 인스턴스를 rememberSaveable로 생성 및 유지
8     val saveableStateHolder = rememberSaveableStateHolder()
9
10    Box(modifier) {
11        // AnimatedContent를 사용하여 화면 전환 애니메이션 적용 (선택 사항)
12        AnimatedContent(
13            targetState = currentScreen,
14            transitionSpec = {
15                fadeIn(animationSpec = tween(300)) togetherWith fadeOut(animationSpec
16                → = tween(300))
17            },
18            label = "NavigationAnimation"
19        ) { targetScreen ->
20            // SaveableStateProvider로 현재 화면 콘텐츠를 래핑하여 상태 저장/복원
21            // key는 targetScreen으로, 화면별로 상태를 구분하여 저장
22            saveableStateHolder.SaveableStateProvider(key = targetScreen) {
23                content(targetScreen) // 실제 화면 콘텐츠 렌더링
24            }
25        }
26    }
}
```

```
27
28 @Composable
29 fun SaveableStateHolderExample() {
30     // 현재 화면 상태를 rememberSaveable로 관리 (구성 변경 시 유지)
31     var screen by rememberSaveable { mutableStateOf("screen1") }
32
33     Column {
34         // 화면 전환 버튼
35         Row(
36             modifier = Modifier.fillMaxWidth(),
37             horizontalArrangement = Arrangement.SpaceEvenly
38         ) {
39             Button(onClick = { screen = "screen1" }) { Text("Go to screen1") }
40             Button(onClick = { screen = "screen2" }) { Text("Go to screen2") }
41         }
42
43         // SaveableStateHolder를 사용하는 Navigation 컴포저블 호출
44         Navigation(screen, Modifier.fillMaxSize()) { currentScreen ->
45             // 현재 화면 키에 따라 다른 화면 표시
46             when (currentScreen) {
47                 "screen1" -> Screen1()
48                 "screen2" -> Screen2()
49             }
50         }
51     }
52 }
53
54 // 화면 1 컴포저블
55 @Composable
56 fun Screen1() {
57     // rememberSaveable을 사용하여 화면 상태(카운터) 유지
58     var counter by rememberSaveable { mutableStateOf(0) }
59     Column(
60         modifier = Modifier.padding(16.dp),
```

```
61     horizontalAlignment = Alignment.CenterHorizontally
62 ) {
63     Text("Screen 1")
64     Spacer(modifier = Modifier.height(16.dp))
65     Button(onClick = { counter++ }) {
66         Text("Counter: $counter")
67     }
68 }
69 }
70
71 // 화면 2 컴포저블
72 @Composable
73 fun Screen2() {
74     // rememberSaveable을 사용하여 화면 상태(텍스트) 유지
75     var text by rememberSaveable { mutableStateOf("") }
76     Column(
77         modifier = Modifier.padding(16.dp),
78         horizontalAlignment = Alignment.CenterHorizontally
79     ) {
80         Text("Screen 2")
81         Spacer(modifier = Modifier.height(16.dp))
82         TextField(
83             value = text,
84             onValueChange = { text = it },
85             label = { Text("Enter text") }
86         )
87     }
88 }
```

주요 개념 살펴보기

위의 예제를 통해서 다음과 같은 내용을 살펴볼 수 있습니다.

1. **내비게이션 래퍼(Navigation Wrapper)**: Navigation 컴포저블은 현재 화면을 가져와

SaveableStateProvider로 래핑합니다. 이를 통해 각 화면의 상태가 화면 키를 기반으로 독립적으로 저장되고 복원되도록 보장합니다.

2. **상태 유지(State Retention)**: 각 화면(Screen1 및 Screen2)은 rememberSaveable을 사용하여 상태를 유지합니다. 즉, Screen1은 카운터를 유지하고 Screen2는 텍스트 입력을 유지합니다.
3. **동적 상태 처리(Dynamic State Handling)**: 내비게이션 버튼을 사용하여 화면 간 전환이 발생해도 각 화면의 상태가 보존되어 상태 손실을 방지합니다.

SaveableStateHolder의 장점

- **화면 간 상태 지속성(State Persistence Across Screens)**: 다른 화면 간 전환 시에도 각 화면의 상태를 유지합니다. 즉, 컴포저블 간 교체가 발생하면서 이미 컴포지션을 벗어난 컴포저블 함수에 대해서도 상태를 유지합니다.
- **단순화된 상태 관리(Simplified State Management)**: 상태 저장 및 복원 작업을 자동으로 처리하여 보일러 플레이트 코드를 줄입니다.
- **구성 변경 처리(Configuration Change Handling)**: rememberSaveable과 함께 원활하게 작동하여, 화면 회전과 같은 구성 변경 중이 발생해도 상태를 보존합니다.

요약

SaveableStateHolder를 사용하여 커스텀 내비게이션을 구현하고자 하는 경우, 별도의 복잡성 없이 개별 화면의 상태가 보존되도록 할 수 있습니다. 특히 구성 변경이 발생해도 상태가 유지되기 때문에 사용자 경험이 우선시 되고, 사용자 화면 간에 전환이 잦은 시스템을 구현하는데 유용합니다. Compose 전용 Jetpack Navigation 라이브러리⁵⁷를 사용하는 경우에도 SaveableStateProvider는 컴포저블 함수 간 전환이 발생해도 UI 상태가 유지되어야 하는 시나리오에서 여전히 유용하게 사용될 수 있습니다.

실전 질문

Q) 여러 화면이 있는 탭 기반 사용자 인터페이스에서 Jetpack Navigation 라이브러리를 사용하지 않고 각 탭이 화면 전환 시에도 스크롤 위치나 입력 상태를 유지하도록 하는 방법에 대해 설명해 주세요.

⁵⁷<https://developer.android.com/develop/ui/compose/navigation>

Q) 22. 스냅샷 시스템(snapshot system)이란 무엇이며 어디에 사용하나요?

Jetpack Compose에서 상태 관리는 스냅샷 시스템에 의해 작동하며, 이는 특정 시점의 애플리케이션 내 모든 관찰 가능한 객체의 상태를 캡처합니다. `Snapshot.takeSnapshot()`은 현재 상태의 읽기 전용(read-only) 스냅샷을 생성하여 값을 수정하지 않고 캡처된 값을 검사하거나 일시적으로 사용할 수 있도록 합니다. 이와 같은 접근 방식은 상태 값에 종속 기능을 디버깅하거나 구현하는 동안 안전성과 일관성을 보장합니다.

Compose의 스냅샷은 게임의 세이브 기능이랑 비슷합니다. 마치 해리포터에 등장하는 타임터너를 사용해서 과거의 특정 지점으로 시간 여행을 하는 것처럼 이해하면 어렵지 않습니다. 따라서 스냅샷은 특정 시점의 모든 관찰 가능한 데이터를 상태로 나타냅니다. 스냅샷을 찍으면 모든 `MutableState` 객체의 상태가 고정되며 해당 객체를 다른 곳에서 변경하더라도 당시 고정된 값에 대해 지속적인 관찰이 가능하기 때문에, 걱정 없이 안전하게 값을 읽을 수 있습니다.

`Snapshot.takeSnapshot()`을 사용하는 이유

`Snapshot.takeSnapshot()`의 주요 목적은 현재 상태에 대해 값의 변동이 없는 읽기 전용 상태를 생성하는 것입니다. 이는 다음과 같은 경우에 특히 유용합니다.

- 값을 변경하지 않고 현재 상태를 디버깅하거나 분석할 때.
- 현재 상태를 기반으로 연산 작업 또는 임시 작업을 수행할 때.
- 멀티스레드 환경에서 상태 값을 읽을 때 스레드 안전성을 보장해야 할 때.

`Snapshot.takeSnapshot()` 사용하기

User 클래스 내에서 `name`이라는 `MutableState`를 프로퍼티로 가지는 클래스를 예로 들어봅시다. 아래 예시는, 현재 상태에 대해 스냅샷을 찍고 데이터를 일관되고 안전하게 읽는 방법을 보여줍니다.

그림 282. SnapshotExample.kt

```
1 import androidx.compose.runtime.MutableState
2 import androidx.compose.runtime.mutableStateOf
3 import androidx.compose.runtime.snapshots.Snapshot
4
5 class User {
6     var name: MutableState<String> = mutableStateOf("")
7 }
8
9 fun main() {
10     val user = User()
11
12     // 초기 이름 설정
13     user.name.value = "skydoves"
14
15     // 현재 상태의 읽기 전용 스냅샷 생성
16     val snapshot = Snapshot.takeSnapshot()
17
18     // 스냅샷 생성 후 상태 수정
19     user.name.value = "Android"
20
21     println("Current name: ${user.name.value}") // 출력: Android
22
23     // 스냅샷에 진입하여 캡처된 값 읽기
24     snapshot.enter {
25         // 스냅샷을 찍을 당시의 상태 값에 접근
26         println("Snapshot name: ${user.name.value}") // 출력: skydoves
27     }
28
29     // 사용 후 스냅샷 폐기하여 리소스 해제
30     snapshot.dispose()
31 }
```

예제 설명

위의 예제가 어떻게 동작하는지 하나씩 살펴보도록 하겠습니다.

1. **초기 상태:** User 클래스에는 `MutableState`를 타입으로 가지는 `name` 프로퍼티가 있습니다. 초기값은 "skydoves"로 설정됩니다.
2. **스냅샷 생성:** `Snapshot.takeSnapshot()`을 사용하여 스냅샷을 생성합니다. 이 순간 `name` 값은 "skydoves"로 고정됩니다.
3. **상태 수정:** 스냅샷을 생성한 후 `name`이 "Android"로 업데이트됩니다. 스냅샷이 찍힌 이후의 변경 사항은 스냅샷에 영향을 전혀 미치지 않습니다.
4. **스냅샷 진입:** `enter` 함수는 해당 블록 내에서 캡처된 상태를 일시적으로 복원합니다. 블록 내부에서 `name`은 여전히 "skydoves"로 출력됩니다.
5. **스냅샷 폐기:** 리소스를 해제하기 위해 스냅샷이 폐기됩니다.

`Snapshot.takeSnapshot()`의 주요 특징

1. **읽기 전용(Read-Only):** `Snapshot.takeSnapshot()`을 사용하여 생성된 스냅샷은 엄격하게 읽기 전용(read-only)입니다. 스냅샷 내부에서 상태를 수정하려고 하면 `IllegalStateException`이 발생합니다.
2. **스레드 안전성(Thread Safety):** 상태를 고정함으로써 스냅샷은 동시 값의 변화에 대한 걱정 없이 안전하게 값을 읽을 수 있도록 보장합니다.
3. **격리(Isolation):** 스냅샷 외부에서 변경된 상태는 캡처된 상태에 영향을 미치지 않습니다. 마찬가지로 스냅샷은 프로그램의 현재 상태에 영향을 미치지 않습니다.

읽기 전용 스냅샷의 이점

- **디버깅(Debugging):** 스냅샷은 상태에 대한 일관된 시점을 제공하여 디버깅 등 문제 분석을 용이하게 합니다.
- **일관성(Consistency):** 스냅샷은 읽기 전용이므로 값을 읽는 동안 동안 실수로 상태를 수정되지 않도록 보장합니다.
- **단순성(Simplicity):** 상태를 격리함으로써 스냅샷은 개발 중 의도하지 않은 사이드 이펙트를 유발할 위험성을 줄입니다.

 **Additional Tips:** Snapshot.takeSnapshot()는 Compose의 런타임 API를 개발하는데 이미 내부적으로 상당히 많이 활용되고 있습니다. 가령, snapshotFlow API를 구현하기 위해서 내부적으로 여러 번 활용되는데, 이러한 스냅샷 시스템을 얼마나 잘 이해하고 실용적으로 잘 활용할 줄 아느냐가 Jetpack Compose를 얼마나 깊이 있게 이해하느냐의 척도라고 봐도 무방합니다.

요약

Snapshot.takeSnapshot()은 특정 시점의 앱 상태를 읽기 전용의 형태로 격리시켜주는 강력한 도구입니다. 특히 디버깅, 상태 분석 등 다양한 환경에서 영향을 받지 않고 일관된 데이터 스냅샷이 필요한 작업을 수행하는 데 유용합니다. 해당 API를 이해하고 적절하게 활용하면 Jetpack Compose에서 더 안전하고 예측 가능한 애플리케이션을 구축할 수 있습니다. 스냅샷 시스템에 대한 더 깊은 이해를 원하시면 [Introduction to the Compose Snapshot System by Zach Klippenstein⁵⁸](#)을 살펴보실 수 있습니다.

실전 질문

Q) 컴포저블 함수에서 직접 상태를 관찰하는 것보다 스냅샷을 찍는 것 (Snapshot.takeSnapshot())이 선호되는 시나리오에 대해 설명해 주세요.

 **Pro Tips for Mastery:** 가변적인 스냅샷(**mutable snapshot**)은 어떻게 생성하나요?

가변적인 스냅샷은 Snapshot.takeMutableSnapshot() API를 사용하여 생성할 수 있습니다. 해당 API를 사용하면 전역 상태에 아무 영향을 주지 않고 상태 값을 안전하게 수정할 수 있는 독립적인 상태를 만들 수 있습니다. 해당 스냅샷에 대한 변경 사항은 apply() 함수를 사용하여 명시적으로 업데이트할 때까지 로컬로 스냅샷 형태로 유지됩니다. 이러한 메커니즘은 상태 변경을 테스트하거나 실험적인 용도로 활용하는 데 특히 유용합니다.

가변적인 스냅샷은 상태에 대해 일종의 복사본을 만드는 것이라고 이해하면 쉽습니다. 이를 통해 다음과 같은 작업을 수행할 수 있습니다.

- 전역 상태에 영향을 주지 않고 로컬로 상태 수정.

⁵⁸<https://blog.zachklipp.com/introduction-to-the-compose-snapshot-system/>

- 변경 사항을 반영하기 전에 안전하게 테스트하거나 유효성 검사.
- 필요한 경우 원하지 않는 변경 사항 폐기.

apply()를 사용하여 스냅샷을 적용하면 해당 변경 사항이 전역 상태로 전파됩니다. 여러 스냅샷이 동일한 상태를 수정하려는 경우는 시스템은 미리 정의해 놓았거나, 커스텀 된 충돌 해결 정책을 사용하여 충돌 문제를 처리합니다.

가변적인 스냅샷 생성 및 사용하기

아래는 가변적인 스냅샷을 만들고 그 스코프 내에서 상태를 수정하는 방법을 살펴봅니다.

그림 283. MutableSnapshotExample.kt

```
1 import androidx.compose.runtime.MutableState
2 import androidx.compose.runtime.mutableStateOf
3 import androidx.compose.runtime.snapshots.Snapshot
4
5 class User {
6     var name: MutableState<String> = mutableStateOf("")
7 }
8
9 fun main() {
10     val user = User()
11     user.name.value = "skydives" // 초기값 설정
12     println("Initial name: ${user.name.value}") // 출력: skydives
13
14     // 가변적인 스냅샷 생성
15     val mutableSnapshot = Snapshot.takeMutableSnapshot()
16
17     // 스냅샷 내부에서 상태 수정
18     mutableSnapshot.enter {
19         user.name.value = "Android"
20         println("Inside snapshot: ${user.name.value}") // 출력: Android
21     }
22 }
```

```
23 // 전역 상태는 아직 영향을 받지 않음  
24 println("After snapshot but before apply: ${user.name.value}") // 출력: skydoves  
25  
26 // 스냅샷을 적용하여 변경 사항을 전역 상태로 전파  
27 mutableSnapshot.apply()  
28 println("After applying snapshot: ${user.name.value}") // 출력: Android  
29  
30 // 사용 후 스냅샷 폐기 (리소스 해제)  
31 mutableSnapshot.dispose()  
32 }
```

예제의 단계

- 초기 상태:** User 객체가 생성되고 내부 name 프로퍼티의 초기 상태 값은 "skydoves"입니다.
- 스냅샷 생성:** Snapshot.takeMutableSnapshot()을 사용하여 가변적인 스냅샷을 생성합니다.
- 스냅샷에서 상태 수정:** 스냅샷의 enter 스코프 내에서 name 프로퍼티가 "Android"로 업데이트됩니다. 이러한 변경 사항은 로컬 스냅샷에 독립적으로 저장되며, 전역 상태에는 영향을 미치지 않습니다.
- 적용 전 상태 확인:** enter 스코프를 빠져나온 후 전역 상태는 변경되지 않은 상태 ("skydoves")로 유지됩니다.
- 스냅샷 적용:** apply() 함수를 호출하여 스냅샷의 변경 사항을 전역 상태에 반영합니다.

주요 사항

- 격리(Isolation):** 스냅샷 내부에서 변경된 사항은 수동적으로 적용될 때까지 로컬에 따로 격리됩니다.
- 명시적 적용(Explicit Application):** 변경 사항을 전역 상태로 반영하려면 apply() 함수를 호출하면 됩니다.
- 안전성(Safety):** 스냅샷을 적용하지 않기로 결정했다면, 스냅샷을 폐기하는 함수인 dispose를 호출하고, 변경 사항을 함께 폐기합니다.

가변적인 스냅샷의 장점

1. **상태 실험(State Experimentation)**: 라이브 애플리케이션 상태에 영향을 주지 않고 안전하게 상태를 수정합니다.
2. **되돌리기 가능(Revertibility)**: 스냅샷을 곧바로 적용하지 않음으로써 원하지 않는 변경 사항을 간단히 폐기합니다.
3. **충돌 해결(Conflict Resolution)**: 여러 스냅샷이 동일한 상태를 수정할 때 충돌 변경 사항을 처리하기 위해 커스텀 충돌 해결 정책을 정의할 수 있습니다.

가변적인 스냅샷 사용 시나리오

- 즉시 반영하지 않고 상태 변경을 안전하게 테스트하기 위해.
- 실행 취소/다시 실행 기능과 같은 시나리오(가령, 텍스트 편집기와 같은)에서 임시적으로 상태 변화를 관리하기 위해.
- 상태에 대해 격리 및 유효성 검사가 필요한 시나리오.

요약

`Snapshot.takeMutableSnapshot()`으로 가변적인 스냅샷을 생성하면 Jetpack Compose에서 상태 관리를 세밀하게 제어할 수 있습니다. 특히 상태에 대한 변경 사항을 격리하여 전역 상태에 적용하기 전에 실험, 테스트 및 유효성 검사를 수행해야 하는 시나리오에 유용합니다. 즉, 상태에 대해 복잡한 상호 작용이 요구되는 시나리오에서 안전성, 유연성 및 일관성을 보장합니다.

Q) 23. State에 List 유형을 바로 넣어서 사용하면 어떤 문제가 발생하나요? `mutableStateListOf`와 `mutableStateMapOf`에 대해서 설명해 주세요.

State는 Compose의 스냅샷 시스템과 함께 작동하며 값이 변경될 때마다 `recomposition`을 트리거하여 UI를 동적으로 업데이트하는 아주 중요한 인터페이스입니다. 그러나 List 또는 Map과 같은 컬렉션을 타입으로 받는 경우 해당 컬렉션에서 제공하는 표준적인 add/remove 등과 같은 메서드는 State에 변경 사항을 알리지 않습니다. 어찌 보면 Compose API와 Kotlin의 컬렉션은 완전히 별개의 API이니 당연한 결과입니다. 결과적으로 State와 가변성을 가진 Kotlin 컬렉션을 함께 사용할 경우 아이템 변경 사항에 대해 `recomposition`을 트리거하지 않아 개발자가 예상한 것과는 다르게 동작합니다. 아래 예시를 통해서 살펴봅시다.

그림 284. MutableStateWithList.kt

```
1 val mutableList by remember { mutableStateOf(mutableListOf("skydoves", "android")) }
2
3 LazyColumn {
4     item {
5         Button(
6             onClick = { mutableList.add("kotlin") } // recomposition을 트리거하지
7             ↪ 않습니다.
8         ) {
9             Text(text = "Add")
10        }
11
12     // 버튼을 클릭하더라도 아이템 항목은 변화가 없습니다.
13     items(items = mutableList) { item ->
14         Text(text = item)
15     }
16 }
```

이 경우 `mutableList`에 항목을 추가해도 Compose에 변경 사항이 올바르게 전달되지 않아 UI가 예상대로 업데이트되지 않습니다. Compose는 컬렉션의 아이템 변화를 제대로 추적하기 위해서 `mutableStateListOf`와 `mutableStateMapOf`라는 별도의 API를 제공합니다. 이와 같이 Compose에서 별도로 제공하는 컬렉션은 Compose의 스냅샷 시스템과 통합되어 아이템이 변경되었을 때 `recomposition`이 발생하도록 보장합니다. 따라서 해당 API를 사용하여 UI 업데이트를 반응형으로 유지하면서 리스트 및 맵을 관찰 가능하고 효율적으로 관리할 수 있습니다.

mutableStateListOf

`mutableStateListOf`는 일반적인 `MutableList`처럼 동작하지만, 내부적으로는 Compose에 최적화된 `SnapshotStateList`를 생성합니다. 따라서 해당 리스트가 가진 아이템에 대한 모든 변경 사항은 해당 상태가 사용되는 곳에 한하여 `recomposition`을 트리거합니다.

그림 285. MutableListStateList.kt

```
1 // SnapshotStateList 생성
2 val items = remember { mutableStateListOf("android", "kotlin", "skydoves") }
```

항목을 추가하거나 제거하는 등 아이템을 수정하면 Compose는 즉시 변경 사항을 감지하고 자동적으로 상태와 UI를 업데이트합니다.

그림 286. MutableListStateList Example.kt

```
1 items.add("Jetpack Compose") // 목록 변경 시 recomposition 트리거
2 items.removeAt(0)
```

mutableStateMapOf

mutableStateMapOf는 MutableMap과 유사하게 동작하는 SnapshotStateMap을 생성하며, 데이터가 변경될 때 상태 및 UI 업데이트를 보장합니다.

그림 287. MutableStateMap.kt

```
1 // SnapshotStateMap 생성
2 val userSettings = remember { mutableStateMapOf("theme" to "dark", "notifications" to
   ↴ "enabled") }
```

값을 업데이트하면 해당 상태를 참조하는 UI 컴포넌트에서 recomposition이 트리거됩니다.

그림 288. MutableStateMap Example.kt

```
1 userSettings["theme"] = "light" // 맵 변경 시 recomposition 트리거
2 userSettings.remove("notifications")
```

일반적인 사용법

mutableStateListOf 및 mutableStateMapOf는 Compose에서 반응성을 유지하면서 UI 관련 상태를 보유하는 데 사용됩니다. 또한, Compose의 스냅샷 시스템과 통합되므로 필요한 UI 컴포넌트만 recompose되도록 보장합니다.

그림 289. UserViewModel.kt

```
1 class UserViewModel : ViewModel() {
2     // ViewModel 내에서 SnapshotStateList 사용
3     private val _mutableUserList = mutableStateListOf("skydoves", "kotlin", "android")
4     // UI 레이어에는 불변 List 또는 StateFlow로 노출 권장
5     val userList: List<String> = _mutableUserList // 또는 StateFlow 사용
6
7     fun addUser(user: String) {
8         _mutableUserList.add(user) // 리스트 변경 시 관찰자에게 알림
9     }
10
11    fun removeUser(user: String) {
12        _mutableUserList.remove(user)
13    }
14 }
15
16 @Composable
17 fun UserListScreen() {
18     val userViewModel: UserViewModel = viewModel()
19     // ViewModel의 상태를 가져와 UI에 표시
20     val userList = userViewModel.userList
21
22     LazyColumn {
23         items(userList) { user ->
24             Text(user)
25         }
26     }
27     // ... (추가/삭제 버튼 등)
28 }
```

아래는 mutableStateMapOf에 대한 사용 예시입니다.

그림 290. SettingsViewModel.kt

```
1 class SettingsViewModel : ViewModel() {
2     // ViewModel 내에서 SnapshotStateMap 사용
3     private val _mutableSettingMap = mutableStateMapOf("theme" to "light", "language"
4             → to "English")
5     // UI 레이어에는 불변 Map 또는 StateFlow로 노출 권장
6     val settings: Map<String, String> = _mutableSettingMap // 또는 StateFlow 사용
7
8     fun updateTheme(theme: String) {
9         _mutableSettingMap["theme"] = theme // 맵 변경 시 관찰자에게 알림
10    }
11
12    fun removeSetting(key: String) {
13        _mutableSettingMap.remove(key)
14    }
15
16 @Composable
17 fun SettingsScreen() {
18     val settingsViewModel: SettingsViewModel = viewModel()
19     val settings = settingsViewModel.settings
20
21     Column {
22         settings.forEach { (key, value) ->
23             Text("$key: $value")
24         }
25     }
26     // ... (설정 변경 UI 등)
27 }
```

요약

`mutableStateListOf` 및 `mutableStateMapOf`는 Jetpack Compose에서 Kotlin에서 제공하는 컬렉션과 유사하게 동작하지만 상태 시스템과 원활하게 통합되어 효율적인 recomposition을 가능하게 합니다. 일반적으로 리스트 또는 맵으로 상태를 관리하기 위해 사용되며, 종종 네트워크 응답이나 데이터베이스 쿼리와 같은 도메인 로직에서 검색된 데이터를 보유하여 컴포저블 함수에서 관찰하고 동적으로 UI를 업데이트할 수 있도록 합니다.

실전 질문

- Q) `mutableStateOf`로 래핑 된 `MutableList`에 아이템을 추가하거나 삭제해도 recomposition이 트리거 되지 않는 이유는 무엇인가요?
- Q) `LazyColumn`에서 리스트의 아이템을 동적으로 추가하거나 제거할 때 UI 업데이트를 효율적으로 추적하려면 리스트의 상태를 어떻게 관리해야 하나요?

Q) 24. 컴포저블 함수에서 Kotlin의 Flow를 메모리 누수 없이 안전하게 관찰하는 방법은 무엇인가요?

컴포저블 함수 내에서 `Flow`를 상태로서 관찰하는 것은 선언적 특성을 가지는 컴포즈에서 필수입니다. 그러나 관찰 및 관찰 해지를 똑바로 처리하지 않으면 메모리 누수 및 과도한 recomposition이 발생하거나 앱 전반적인 성능 문제가 발생할 수 있습니다. Compose에서 안전하게 `flow`를 관찰하기 위해 널리 사용되는 두 가지 기본 접근 방식은 `collectAsState`와 `collectAsStateWithLifecycle`입니다. 최적의 성능 및 생명주기 관리를 위해 이들의 차이점과 각각 언제 사용해야 하는지 올바르게 이해하는 것이 중요합니다.

collectAsState

`collectAsState`는 `Flow`를 수집하고 이를 `State` 객체로 변환해 주는 편리한 API입니다. 이를 통해 수집된 데이터를 컴포저블 함수 내에서 직접 상태와 동일한 형태로 사용할 수 있고, `flow`가 새 값을 방출할 때마다 상태값에 반영되어 recomposition을 트리거할 수 있습니다.

그림 291. CollectAsState.kt

```
1 @Composable
2 fun UserProfileScreen(viewModel: UserViewModel) {
3     // viewModel.userNameFlow를 State로 수집, 초기값은 "skydoves"
4     val userName by viewModel.userNameFlow.collectAsState(initial = "skydoves")
5
6     Column {
7         Text(text = "User: $userName")
8     }
9 }
```

위의 예제에서 다음과 같은 내용을 살펴볼 수 있습니다.

- viewModel의 userNameFlow(Flow 타입)가 Compose의 상태로서 수집됩니다.
- userNameFlow Flow가 새 값을 방출할 때마다 UI가 recompose됩니다.
- 관찰은 컴포지션 내에서 발생하며 컴포저블 함수가 컴포지션을 떠날 때 중지됩니다.

collectAsState는 대부분의 시나리오에서 정상적으로 작동하지만 안드로이드 생명주기를 인지하고 있는 것은 아닙니다. Compose 자체가 Compose Multiplatform에 기반하여 설계되었기 때문에, Compose Runtime 자체는 안드로이드와 완벽하게 독립적인 구조이기 때문입니다. 따라서, collectAsState는 안드로이드와는 전혀 관련 없는 API이므로 안드로이드의 생명주기를 당연히 인식하지 못합니다. 즉, 컴포저블 함수가 메모리에 남아 있지만 활발하게 사용되지 않는 시나리오(가령, 사용자가 다른 화면으로 이동한 경우)에도 Flow가 계속 수집되어 잠재적으로 불필요한 리소스 사용으로 이어질 수 있습니다. 의외로 많은 주니어뿐만 아니라, 시니어 개발자분들도 이 부분을 놓치고 있습니다.

collectAsStateWithLifecycle

collectAsStateWithLifecycle은 Flow 수집이 안드로이드의 생명주기에 바인딩되도록 보장하는 안드로이드 전용 API입니다. 컴포저블 함수가 포그라운드에 있지 않을 때 Flow 수집을 자동으로 일시 중지하여 불필요한 백그라운드 작업을 방지합니다. 해당 API는 안드로이드 생명주기를 인식하도록 설계된 androidx.lifecycle-runtime-compose 패키지에서 제공됩니다.

그림 292. collectAsStateWithLifecycle.kt

```
1 @Composable
2 fun UserProfileScreen(viewModel: UserViewModel) {
3     // lifecycle-aware 방식으로 Flow 수집
4     val userName by viewModel.userNameFlow.collectAsStateWithLifecycle(initialValue =
5         "skydoves")
6
7     Column {
8         Text(text = "User: $userName")
9     }
}
```

collectAsStateWithLifecycle의 주요 이점은 다음과 같습니다.

- 호스트 Activity 또는 Fragment의 Lifecycle을 추적하고 인지합니다.
- UI가 백그라운드에 있을 때(가령, 화면 회전 또는 내비게이션 중) Flow 수집이 자동으로 일시 중지됩니다.
- 컴포지션을 완전히 떠나거나, 더 이상 필요하지 않을 때 Flow가 계속 실행되지 않도록 하여 메모리 누수를 방지합니다.

올바른 접근 방식 선택하기

- 컴포저블이 컴포지션에 있는 동안 flow 수집이 항상 활성 상태여야 하는 경우 collectAsState를 사용해야 합니다.
- 반면에, 컴포저블이 보이지 않을 때 안드로이드의 생명주기에 따라 Flow 수집을 멈춰 불필요한 작업을 피하기 위해서는 collectAsStateWithLifecycle을 사용해야 합니다.

요약

Compose를 사용하는 안드로이드 앱에서 안전하게 Flow를 수집하려면 안드로이드의 생명주기를 올바르게 인지하고 그에 맞게 처리해야 합니다. collectAsState는 상태가 변경될 때 recomposition을 보장하지만 안드로이드 UI가 비활성 상태일 때 수집을 일시 중지하지

않습니다. 반면에 `collectAsStateWithLifecycle`은 안드로이드 UI가 백그라운드에 있을 때 수집을 일시 중지하여 불필요한 작업을 줄이고 메모리 누수를 방지합니다. 어떤 메서드를 선택해야 할지는 각 상황에 따라 달라집니다. 따라서 모든 경우에 단 하나의 API만 사용하는 것도 옳지 않습니다. 이와 관련하여 더 깊은 이해를 원하시면 [Consuming flows safely in Jetpack Compose](#)⁵⁹를 읽어보시는 것을 권장합니다.

실전 질문

Q) `collectAsState`는 안드로이드 생명주기를 인지하지 못하기 때문에 UI가 보이지 않을 때도 계속 Flow를 수집하여 잠재적인 메모리 누수를 유발할 수 있습니다. 이 문제를 어떻게 해결할 수 있나요? 만약 `collectAsStateWithLifecycle`를 사용하지 않고 직접 해결해야 한다면 어떻게 구현하실 건가요?

Q) 25. `CompositionLocals`의 역할과 목적에 대해 설명해 주세요.

`CompositionLocal`⁶⁰은 컴포저블 함수의 매개변수를 통해 데이터를 명시적으로 전달하지 않고도 컴포지션 트리를 통해 데이터를 암시적으로 전달할 수 있는 Jetpack Compose 메커니즘입니다. 이는 깔끔하고 확장 가능한 UI 아키텍처를 유지하는 데 도움이 되며 복잡한 UI 트리 구조에서 수동으로 매개변수를 전달할 필요성을 줄입니다.

각 컴포저블 함수가 상위 계층을 통해 지속적으로 데이터를 아래로 흘려보내는 방법 대신, `CompositionLocal`을 사용하면 UI 계층 구조의 모든 수준에서 특정 데이터에 접근할 수 있도록 하여 더 유연한 접근 방식을 가능하게 합니다. 특히 이러한 메커니즘은 텍스트 스타일, 내비게이션 핸들러와 같은 전역적인 값을 구성하는 경우, 혹은 시스템 전반적인 영역에 영향을 미치는 테마나 상위 계층에서 주입받은 의존성을 하위 계층에서 사용하고 관리하는데 유용합니다.

`CompositionLocal`을 사용하는 이유

Jetpack Compose는 선언적 접근 방식을 따르므로 UI 컴포넌트가 대체로 재사용 가능하고 직관적입니다. 그러나 레이아웃의 가장 높은 계층에서 제공된 데이터가 컴포저블 계층 구조 아래 깊숙한 곳에서 필요한 경우 문제가 하나 발생합니다. 간단한 UI 구조라면 단순히

⁵⁹<https://medium.com/androiddevelopers/consuming-flows-safely-in-jetpack-compose-cde014d0d5a3>

⁶⁰<https://developer.android.com/develop/ui/compose/compositionlocal>

매개변수로 넘겨서 처리하면 되지만, 10개 이상의 계층에 걸쳐 데이터를 전달해야만 하는 시나리오와 같이 깊게 중첩된 컴포저블이 있는 복잡한 레이아웃에서는 과도한 매개변수 전달로 컴포저블 함수가 비대해질 뿐더러 매우 비효율적으로 동작하게 됩니다.

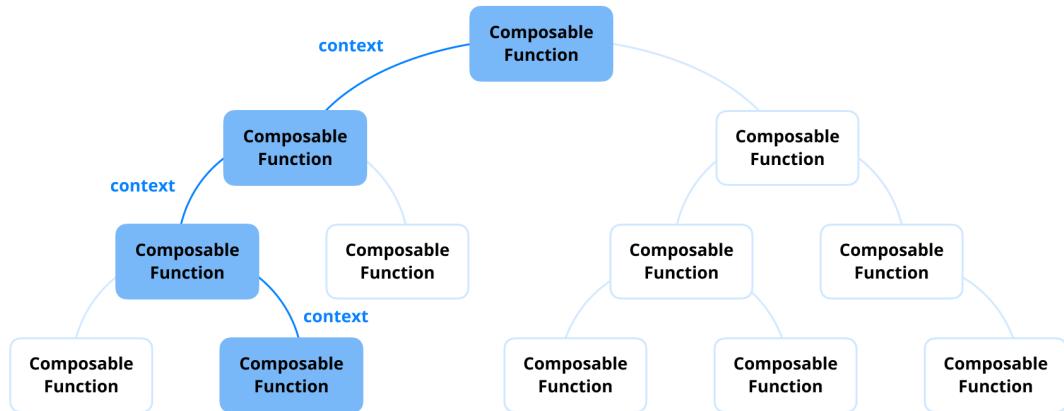


그림 293. composition-locals-sample

이와 같은 문제를 해결하기 위해 Jetpack Compose는 Compose 트리를 통해 암시적 데이터 전파를 가능하게 하는 `CompositionLocal` 메커니즘을 제공합니다. 이를 통해 모든 수준의 컴포저블이 명시적으로 매개변수를 전달하지 않고도 필요한 정보에 접근할 수 있어, 특히 깊게 중첩되고 복잡한 UI 계층 구조에서 제한된 범위 내에서 제공된 데이터에 접근하는 것이 더 깔끔하고 쉬워집니다.

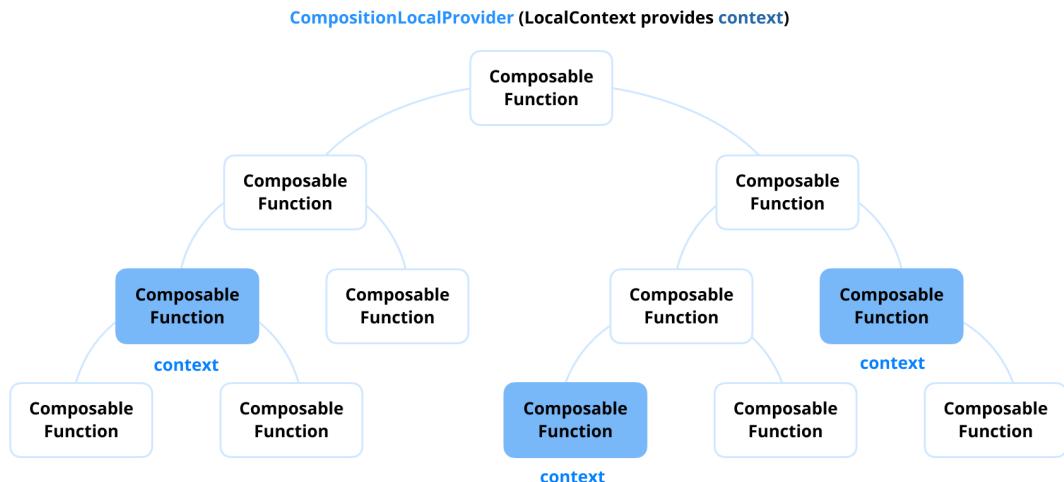


그림 294. composition-locals

대부분의 경우 `CompositionLocal`은 한 번 초기화되고 나면 변경되지 않는 비교적 정적인 정보를 전달하는 데 사용됩니다. 일반적인 예는 Compose UI 컴포넌트 전체에서 일관된 디자인을 유지하기 위한 `MaterialTheme` 객체입니다. 실제 라이브러리의 내부 구현을 살펴보겠습니다.

그림 295. MaterialTheme.kt

```

1 @Composable
2 fun MaterialTheme(
3     colorScheme: ColorScheme = MaterialTheme.colorScheme, // 기본값 사용
4     shapes: Shapes = MaterialTheme.shapes,
5     typography: Typography = MaterialTheme.typography,
6     content: @Composable () -> Unit
7 ) {
8     // ... 내부 로직 ...
9
10    // CompositionLocalProvider를 사용하여 테마 관련 값을 하위 컴포저블에 제공
11    CompositionLocalProvider(
12         LocalColorScheme provides colorScheme,
13         LocalShapes provides shapes,
14         LocalTypography provides typography,
  
```

```
15      // 다른 CompositionLocal 값들도 여기에 제공될 수 있습니다.  
16      // LocalContentColor 등  
17  ) {  
18      content() // MaterialTheme의 자식 컴포저블들  
19  }  
20 }
```

사용 예제

CompositionLocal을 사용하여 해당 스코프 내의 UI 계층에 존재하는 그 어떠한 Composable에서든지 사용자 객체를 관리할 수 있도록 하는 예제입니다.

그림 296. **CompositionLocalExample.kt**

```
1 // String 타입의 CompositionLocal 정의 (기본값 "skydoves")  
2 val LocalUser = compositionLocalOf { "skydoves" }  
3  
4 @Composable  
5 fun UserProfile() {  
6     Column {  
7         // LocalUser의 현재 값 접근  
8         Text(text = "User: ${LocalUser.current}")  
9         UserDetails()  
10    }  
11 }  
12  
13 @Composable  
14 fun UserDetails() {  
15     // 여기서도 LocalUser의 현재 값 접근 가능  
16     Text(text = "Welcome, ${LocalUser.current}!")  
17 }  
18  
19 @Composable  
20 fun App() {
```

```
21 // CompositionLocalProvider를 사용하여 LocalUser에 "Android" 값 제공
22 CompositionLocalProvider(LocalUser provides "Android") {
23     // 이 범위 내의 모든 UserProfile 호출은 "Android" 값을 사용
24     UserProfile()
25 }
26 }
```

이 예제에서 LocalUser 객체는 상위 수준(App 컴포저블)에서 제공되는데, 해당 값을 매개변수를 통해서 전달하지 않고 UserProfile 및 UserDetails과 같은 컴포저블 함수에서 암시적으로 접근할 수 있습니다. 이와 같이 유저 프로필처럼 일반적으로 잘 변화하지 않지만 변화하는 순간 앱 전반에 걸쳐 업데이트가 필요하며 앱의 다양한 곳에서 접근이 필요한 경우, 루트 컴포저블에서 해당 객체를 CompositionLocal을 통해 제공하면 데이터에 대한 일관성과 동기화 문제를 간단하게 해결할 수 있습니다.

요약

CompositionLocal은 매개변수를 명시적으로 전달할 필요 없이 Compose 트리 전체에서 암시적 데이터 공유를 가능하게 합니다. 테마, 사용자 세션 및 내비게이션 핸들러와 같은 전역적인 구성을 관리하는 데 특히 유용합니다.

실전 질문

- Q) CompositionLocal이란 무엇이며, 보통 어떤 시나리오에서 유용하게 사용하시나요?
- Q) CompositionLocalProvider는 어떻게 작동하며, 만약 값이 제공되지 않은 CompositionLocal에 접근하려고 하면 어떤 일이 발생하나요?

Pro Tips for Mastery: **CompositionLocal**을 신중하게 사용해야 하는 이유

CompositionLocal은 명시적으로 매개변수를 제공하지 않고 UI 트리를 통해 데이터를 암시적으로 전달하는 메커니즘입니다. 재사용성을 향상시키고 데이터 전파를 단순화하지만 부적절하게 사용하면 과도한 recomposition을 유발하여 앱 전반적인 성능에 영향을 미칠 수 있습니다. Jetpack Compose는 CompositionLocal을 생성하기 위한 두 가지 API인 `compositionLocalOf`과 `staticCompositionLocalOf`을 제공합니다. 효율적인 상태 관리를 위해서는 이들의 차이점을 올바르게 이해하는 것이 중요합니다.

compositionLocalOf: 세분화된 Recomposition을 통한 동적 상태 관리에 효율적

compositionLocalOf는 동적 CompositionLocal입니다. 즉, 해당 값이 변경되면 그 값을 읽고 있는 컴포저블에 대해 recomposition을 트리거합니다. 따라서 자주 변경되지만 어느 정도의 범위에 걸쳐서 업데이트가 필요한 시나리오에 적합합니다.

그림 297. DynamicCompositionLocal.kt

```
1 // 동적 CompositionLocal 생성
2 val LocalUser = compositionLocalOf { "skydoves" }
3
4 @Composable
5 fun UserScreen() {
6     var user by remember { mutableStateOf("skydoves") }
7
8     Column {
9         Button(onClick = { user = "android" }) { // 버튼 클릭 시 user 상태 변경
10            Text("Change User")
11        }
12        // CompositionLocalProvider로 새로운 user 값 제공
13        CompositionLocalProvider(LocalUser provides user) {
14            UserProfile() // UserProfile만 recompose됨
15
16            OtherComposable() // LocalUser를 읽지 않는다면 recompose되지 않음
17        }
18    }
19 }
20
21 @Composable
22 fun UserProfile() {
23     // LocalUser의 현재 값 읽기
24     Text("User: ${LocalUser.current}")
25 }
```

위의 예제는 아래와 같이 동작합니다.

- LocalUser는 recomposition 중에 변경되는 동적인 값을 보유합니다.
- 버튼을 클릭하면 user 상태 값이 업데이트되고, 해당 값을 읽고 있는 UserProfile만 recompose 되어 성능을 최적화합니다.

staticCompositionLocalOf: 정적 값에 효율적

compositionLocalOf와 달리 staticCompositionLocalOf는 컴포지션에서 해당 값을 어디에서 읽고 있는지 따로 추적하지 않습니다. 대신 제공된 값이 변경되면 Compose는 제공자 내에 속한 모든 UI 트리에 대해서 광범위하게 recomposition을 트리거합니다.

그림 298. StaticCompositionLocal.kt

```
1 // 정적 CompositionLocal 생성
2 val LocalThemeColor = staticCompositionLocalOf { Color.Black }
3
4 @Composable
5 fun ThemedScreen() {
6     var themeColor by remember { mutableStateOf(Color.Blue) }
7
8     Column {
9         Button(onClick = { themeColor = Color.Green }) { // 테마 색상 변경
10            Text("Change Theme")
11        }
12        // 새 themeColor 값 제공
13        CompositionLocalProvider(LocalThemeColor provides themeColor) {
14            // 이 블록 전체가 recompose됨
15            ThemedContent()
16        }
17    }
18 }
19
20 @Composable
21 fun ThemedContent() {
22     // LocalThemeColor의 현재 값 읽기
```

```
23     Box(modifier = Modifier.background(LocalThemeColor.current).size(100.dp))
24 }
```

위의 예제는 아래와 같이 동작합니다.

- LocalThemeColor는 정적 테마 색상을 보유합니다.
- themeColor가 업데이트되면 전체 CompositionLocalProvider 스코프 내의 모든 컴포저블 트리가 recompose 됩니다.
- compositionLocalOf와 달리 특정 스코프에 대해서만 recomposition 하도록 제한하지 않고 전체 트리를 전부 recomposition 하기 때문에, 테마와 같이 거의 변경되지 않는 전역 값에 적합합니다.

Recomposition 고려 사항

CompositionLocal을 잘못 사용하면 불필요한 recomposition이 발생하여 앱 전반적인 성능에 영향을 미칠 수 있습니다. compositionLocalOf와 staticCompositionLocalOf의 주요 차이점은 다음과 같습니다.

API	Recomposition 동작	사용 시나리오
compositionLocalOf	current가 호출된 곳에 한하여 다시 트리거	자주 변경되는 상태 (가령, 사용자 환경 설정, UI의 로컬라이제이션 업데이트)
staticCompositionLocalOf	제공자 스코프 내 전체 UI 트리에 대해 recomposition 트리거	전역 정적 값 (가령, 테마 등)

성능을 최적화 하기 위해서 다음 내용을 숙지하고 있으면 좋습니다.

- 거의 변경되지 않는 전역 값에는 staticCompositionLocalOf를 사용하여 과도한 recomposition을 방지합니다.

- 범위가 작고 자주 변경되는 동적인 값에는 `compositionLocalOf`를 사용하여 업데이트가 특정 컴포저블에만 영향을 미치도록 합니다.
- 매우 동적인 데이터에는 `CompositionLocal` 사용 자체를 지양하고 대신 `remember`와 `State`를 사용하여 로컬 컴포저블에서 UI 상태를 효율적으로 관리하는 것이 좋습니다.

요약

`CompositionLocal`은 Compose 계층 구조 내에서 데이터를 전달하는 데 유용한 도구이지만 불필요한 `recomposition`을 방지하기 위해 신중하게 사용해야 합니다. `compositionLocalOf`는 자주 변경되는 값에 대한 세분화된 제어를 제공하는 반면, `staticCompositionLocalOf`는 안정적인 전역 구성에 더 적합합니다. 올바른 시나리오에 맞는 올바른 API를 선택하면 더 효율적이고 좋은 성능의 앱을 구축할 수 있습니다.

카테고리 2: Compose UI

Compose UI는 Box, Column, Row, MaterialTheme과 같은 UI 관련 컴포넌트들을 포함합니다. 레이아웃, 드로잉, 사용자 입력 박스 등을 포함하여 기기와 상호 작용하는 데 필요한 UI 라이브러리 세트를 제공하며, Compose Material⁶¹, Compose Foundation⁶², Compose UI⁶³ 등 여러 라이브러리를 포함합니다.

플랫폼과 완전히 독립적인 Compose Compiler 및 Compose Runtime과 달리, Jetpack Compose UI는 안드로이드용으로 설계되었으며 Compose Runtime 위에서 작동하도록 개발되었습니다. Compose UI는 이러한 플랫폼 독립적인 Compose Compiler 및 Compose Runtime의 클라이언트 역할을 합니다. JetBrains는 크로스 플랫폼 개발을 위해 Compose Multiplatform⁶⁴을 제공하는데, 이는 Kotlin Multiplatform⁶⁵을 사용하여 안드로이드, iOS, WebAssembly(Wasm) 및 데스크톱을 포함한 여러 플랫폼에서 Compose UI를 빌드할 수 있도록 합니다. Jetpack Compose의 채택률이 지속적으로 증가함에 따라 Compose Multiplatform 생태계도 확장되고 있지만, 이 책에서는 안드로이드용 Compose UI에만 집중합니다.

일부 API는 성능에 직접적인 영향을 미칠뿐더러, UI 레이아웃이 얼마나 효율적으로 동작하는지를 결정하므로 Compose UI의 핵심 구성 요소를 올바르게 이해하는 것이 중요합니다. 특히 Modifier와 같은 API를 제대로 이해하는 것이 Compose UI를 올바르게 사용하는 데 있어 반 이상을 차지한다고 봐도 무방합니다. 이번 카테고리의 목적은 우리가 일상에서 매일 사용하는 Compose UI API에 대한 더 깊은 이해를 얻기 위함입니다. 따라서 모든 Compose UI와 관련된 API를 다루는 것은 아니고, 우리가 일반적으로 앱 UI를 개발하는 데 있어서 가장 필수적인 개념에 중점을 두고 살펴볼 예정입니다.

Q) 26. Modifier란 무엇인가요?

Modifier⁶⁶는 스타일, 동작을 적용하고 컴포저블을 연계적으로 수정할 수 있도록 하는 Compose UI의 초석이라고 할 수 있는 API입니다. 패딩, 크기, 정렬, 클릭 동작, 배경 및 상호 작용 설정과 같은 UI 컴포넌트의 스타일을 정의하거나 사용자와 상호작용을 가능하게 하는

⁶¹<https://developer.android.com/jetpack/androidx/releases/compose-material>

⁶²<https://developer.android.com/jetpack/androidx/releases/compose-foundation>

⁶³<https://developer.android.com/jetpack/androidx/releases/compose-ui>

⁶⁴<https://www.jetbrains.com/compose-multiplatform/>

⁶⁵<https://kotlinlang.org/docs/multiplatform.html>

⁶⁶<https://developer.android.com/develop/ui/compose/modifiers>

유연한 방법들을 제공합니다. Compose는 선언적 UI 패러다임을 따르기 때문에 Modifier는 핵심 로직을 수정하지 않고도 UI 컴포넌트를 재사용할 수 있고 유지 관리 가능하게 만듭니다.

Modifier는 메서드를 체이닝 형태로 연결하는 방식으로 적용되며, 여러 다양한 스타일을 적용하기 위해 구조화된 접근 방식을 사용할 수 있습니다. 체인의 각 함수는 이전에 적용된 수정을 유지하면서 새 Modifier 인스턴스를 반환합니다. 이는 불변성을 보장하고 Compose가 UI 업데이트를 처리하는 방식을 최적화하여 성능을 향상시킵니다.

일반적으로 Modifier는 레이아웃 계층 구조를 통해 원활하게 전파되어 루트 컴포저블에서 정의된 속성이 UI 전체에서 확장되고 유지될 수 있도록 합니다. 이러한 접근 방식은 효율성을 향상시키면서 레이아웃 구성 및 스타일링의 일관성을 향상시킵니다. 또한 Modifier는 상태를 저장하지 않으며(stateless)이며, 표준적인 Kotlin 객체로 구현되어 있기 때문에 아래와 같이 함수 체인을 사용하여 쉽게 생성하고 구성할 수 있습니다.

그림 299. ModifierExample.kt

```
1 @Composable
2 fun Greeting(name: String) {
3     Column(
4         modifier = Modifier
5             .padding(24.dp) // Column 주위에 패딩 추가
6             .fillMaxWidth() // Column이 사용 가능한 최대 너비를 채우도록 함
7     ) {
8         Text(text = "Hello,")
9         Text(text = name)
10    }
11 }
```

위의 예제는 다음 동작을 포함합니다.

- padding(24.dp)는 Column 주위에 간격을 추가하여 화면 가장자리에 닿지 않도록 합니다.
- fillMaxWidth()는 Column이 사용 가능한 최대 너비로 확장되도록 하여 레이아웃의 일관성을 보장합니다.

💡 Additional Tips: Jetpack Compose는 전통적인 XML 시스템과는 다르게 마진(margin)이라는 개념이 존재하지 않습니다. 따라서 Modifier 체인에서 padding의 순서를 조정함으로써 margin과 동일한 동작을 구현할 수 있습니다.

Modifier 순서의 중요성

Modifier를 사용할 때 가장 중요한 요소는 바로 체이닝 시 각 메서드들이 적용되는 순서입니다. Modifier는 순차적으로 호출될 수 있는데, 각 함수는 이전 함수를 래핑하고 그 위에 구축되어 컴포넌트의 최종 모양과 동작에 직접적인 영향을 미치는 복합 구조를 형성합니다.

이러한 레이어링 프로세스는 각 Modifier가 제어되고 예측 가능한 방식으로 적용되도록 보장합니다. 이는 트리 순회와 유사하게 작동하며, 수정 사항이 위에서 아래로 단계별로 적용되어 UI의 최종 레이아웃 및 상호 작용 속성에 영향을 미칩니다.

예를 들어, padding 전에 clickable을 적용하면 패딩을 포함한 전체 영역을 클릭할 수 있게 되지만, padding을 먼저 적용하면 클릭 가능한 영역이 내부 콘텐츠로만 제한됩니다.

그림 300. ModifierOrder.kt

```
1 @Composable
2 fun ArtistCard(onClick: () -> Unit) {
3     Column(
4         modifier = Modifier
5             .clickable(onClick = onClick) // clickable이 먼저 적용됨
6             .padding(21.dp)
7             .fillMaxWidth()
8     ) {
9         // ... 콘텐츠 ...
10    }
11 }
```

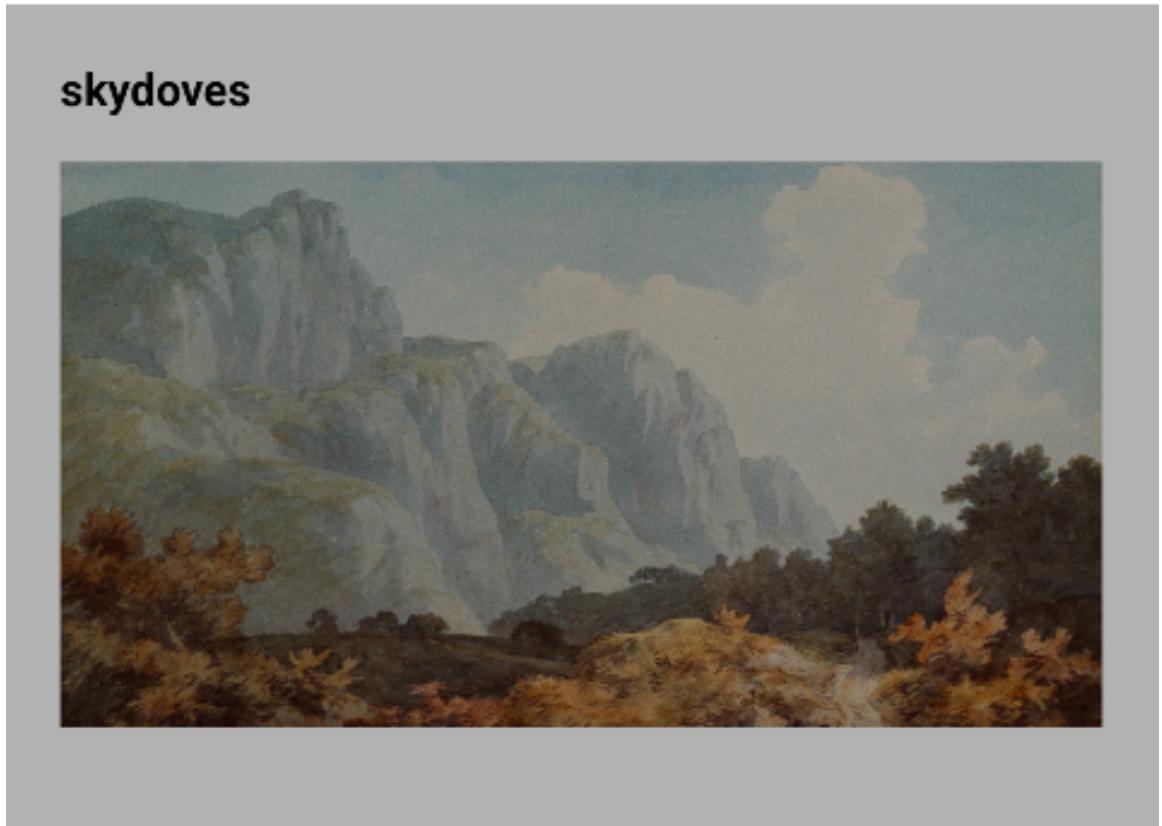


그림 301. compose-order-example

아래의 버전에서는 패딩쪽 영역도 클릭 가능합니다. 간단히 순서를 바꾸는 것만으로도 동작 구조가 달라집니다.

그림 302. ModifierOrderReversed.kt

```
1 @Composable
2 fun ArtistCard(onClick: () -> Unit) {
3     Column(
4         modifier = Modifier
5             .padding(21.dp) // padding이 먼저 적용됨
6             .clickable(onClick = onClick)
7             .fillMaxWidth()
8     ) {
9         // ... 콘텐츠 ...
}
```

```
10     }  
11 }
```

이제 내부 콘텐츠만 클릭 가능하며 패딩은 클릭해도 리스너가 호출되지 않습니다. 이는 Modifier 순서가 Compose UI의 동작에 어떤 영향을 미치는지 보여주는 좋은 사례입니다.

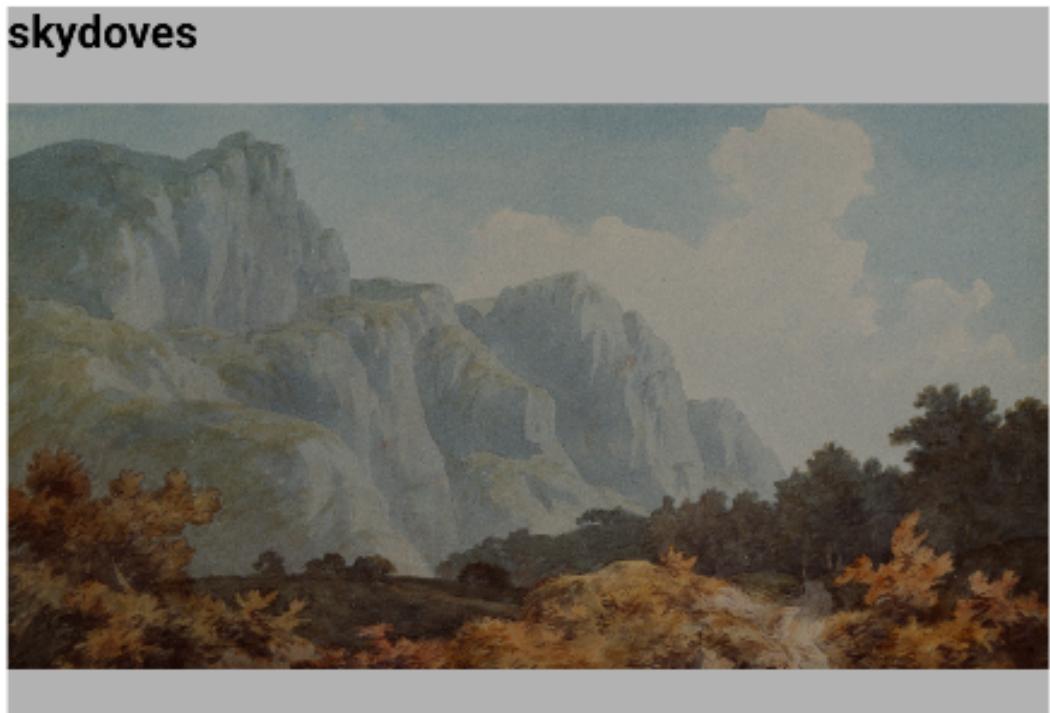


그림 303. compose-order-example

다음은 Modifier 함수 순서가 레이아웃에 미치는 영향을 보여주는 또 다른 좋은 사례입니다. 해당 구현은 Modifier 순서를 조정하는 것만으로도 전통적인 XML 시스템에서는 구현하기 까다로운 온라인 인디케이터를 쉽게 구현할 수 있음을 보여줍니다.

그림 304. OnlineIndicator.kt

```
1 @Composable
2 fun OnlineIndicator(modifier: Modifier = Modifier) {
3     Box(
4         modifier = modifier
5             .size(60.dp) // 전체 레이아웃 크기 설정
6             .background(Color.Gray, CircleShape) // 외부 배경 적용
7             .padding(4.dp) // 내부 콘텐츠 주위에 패딩 추가
8             .background(Color.Green, CircleShape), // 내부 배경 적용 (원의 중심)
9     )
10 }
```

위의 예제 컴포저블 함수에 @Preview 어노테이션을 추가하면 Android Studio에서 Compose 미리 보기 기능이 활성화되어, Modifier의 스타일링 및 레이어링이 의도한 대로 작동하는지 미리 확인할 수 있습니다.

**그림 305. compose-online-indicator**

보편적으로 사용되는 Modifier 함수 알아보기

Jetpack Compose는 UI 레이아웃, 모양 및 상호 작용을 조정하기 위해 기본적으로 다양한 Modifier API를 제공합니다.

크기 조절 및 제약 조건 (Sizing and Constraints)

기본적으로 Compose 레이아웃은 자식을 감싸지만 size, fillMaxSize, fillMaxWidth 또는 fillMaxHeight를 사용하여 제약 조건을 지정할 수 있습니다.

그림 306. SizeModifier.kt

```
1 @Composable
2 fun ArtistCard() {
3     Row(
4         modifier = Modifier.size(width = 400.dp, height = 100.dp) // 고정 크기 설정
5     ) {
6         Text(text = "skydoves")
7     }
8 }
```

경우에 따라 부모 규칙으로 인해 제약 조건이 무시될 수 있습니다. 부모 제약 조건에 관계없이 고정 크기를 강제해야 하는 경우 requiredSize()를 사용해야 합니다.

그림 307. RequiredSize.kt

```
1 @Composable
2 fun ArtistCard() {
3     Row(
4         modifier = Modifier.size(400.dp, 100.dp) // 부모 크기 제약
5     ) {
6         Text(
7             text = "skydoves",
8             modifier = Modifier.requiredSize(150.dp) // 자식에게 고정 크기 강제
9         )
10    }
```

```
10     }
11 }
```

여기서 부모 컴포저블의 높이가 100.dp로 설정되어 있어도 requiredSize() 때문에 텍스트는 여전히 150.dp가 됩니다.

💡 Pro Tips for Mastery: Compose에서 레이아웃은 부모가 제약 조건을 설정하고 자식이 이를 따를 것으로 예상하는 제약 조건 기반 시스템을 따릅니다. 그러나 requiredSize와 같은 Modifier는 이러한 제약 조건을 재정의하거나 커스텀 레이아웃을 개발할 수 있도록 합니다. 자식이 제약 조건을 무시하면 시스템은 기본적으로 중앙에 배치되는데, 이러한 동작 또한 wrapContentSize로 조정할 수 있습니다.

레이아웃 위치 지정 (Layout Positioning)

현재 컴포넌트의 위치를 이동하려면 offset()을 사용하여 상대적인 위치를 지정할 수 있습니다. 패딩과 달리 오프셋은 컴포넌트 크기를 변경하지 않고 이동시킵니다.

그림 308. OffsetModifier.kt

```
1 @Composable
2 fun ArtistCard() {
3     Column {
4         Text(text = "skydoves")
5         Text(
6             text = "Last seen online",
7             modifier = Modifier.offset(x = 10.dp) // 오른쪽으로 10dp 이동
8         )
9     }
10 }
```

이렇게 하면 두 번째 Text 컴포넌트가 오른쪽으로 10.dp 이동합니다.

범위 지정 Modifier (Scoped Modifiers)

일부 Modifier는 특정 컴포저블 스코프 내에서만 사용할 수 있습니다. 예를 들어, `matchParentSize()`는 `Box` 내에서만 사용할 수 있으며 자식 컴포넌트가 부모의 정확한 사이즈에 들어맞도록 보장합니다.

그림 309. MatchParentSize.kt

```
1 @Composable
2 fun MatchParentSizeExample() {
3     Box(modifier = Modifier.fillMaxWidth()) { // Box는 크기를 가짐
4         Spacer(
5             modifier = Modifier
6                 .matchParentSize() // Box 크기와 동일하게 설정
7                 .background(Color.LightGray)
8         )
9         Text(text = "skydoves") // Spacer 위에 텍스트 배치
10    }
11 }
```

여기서 `Spacer`는 부모 컴포저블인 `Box`의 사이즈와 동일하게 조정되며 `Text`의 배경 색상 역할을 합니다. 실제로는 `Text`의 `Modifier`에 배경색상을 바로 추가하면 되기 때문에 비효율적인 코드지만, 이해를 돋기 위한 예제일 뿐이니 오해하시면 안 됩니다.

skydoves

그림 310. compose-scoped-modifiers

마찬가지로 `weight()` Modifier는 `Row` 및 `Column` 내에서만 사용할 수 있습니다. 자식 컴포저블이 같은 레벨에 존재하는 다른 컴포저블에 대해서 상대적으로 유연한 공간을 차지하도록 허용합니다.

그림 311. WeightModifier.kt

```
1 @Composable
2 fun WeightedRow() {
3     Row(modifier = Modifier.fillMaxWidth()) {
4         // 첫 번째 Box가 두 번째 Box보다 상대적으로 2배의 가중치를 가짐
5         Box(modifier = Modifier.weight(2f).height(50.dp).background(Color.Red))
6         Box(modifier = Modifier.weight(1f).height(50.dp).background(Color.Blue))
7     }
8 }
```

이 예제에서 첫 번째의 빨간색 Box는 두 번째의 파란색 Box보다 두 배 더 크게 랜더링됩니다.

**그림 312. compose-weight****요약**

Modifier는 개발자가 크기, 레이아웃 및 상호 작용과 관련된 동작을 적용하여 컴포저블을 쉽게 커스텀할 수 있도록 하는 Jetpack Compose의 핵심적인 API입니다. Modifier가 적용되는 순서는 실제 구현과 동작에 영향을 미치며, 스코프와 관련된 Modifier는 해당 부모 컴포저블에 따라 신중하게 사용해야 합니다. Modifier를 효율적으로 이해하고 적용한다면 기존의 전통적인 XML 시스템에서 구현이 어려웠던 UI를 매우 쉽게 구현할 수 있으며, 유연하고 재사용성이 높은 UI 컴포넌트를 개발할 수 있습니다.

실전 질문

Q) Modifier의 순서가 중요한 이유를 설명해 주세요. Modifier의 순서 변경으로 다른 동작을 하게 되는 시나리오를 예로 들어서 설명해 주세요.

💡 Pro Tips for Mastery: Modifier 사용 규칙

Modifier는 컴포저블 함수를 쉽게 스타일링하고 구성하는 데 매우 중요한 역할을 합니다. 그러나 부적절하게 사용하면 “Modifier 순서의 중요성” 섹션에서 살펴본 바와 같이 개발자가 의도하지 않은 전혀 다른 동작이 발생할 수 있습니다. 따라서 오용을 방지하고 Compose 컴포넌트의 일관성을 보장하려면, 팀 내에서 Modifier 사용에 대한 명확한 규칙이나 지침을 설정하는 것이 좋습니다.

1. 가장 바깥쪽 레이아웃에 Modifier 적용하기

Modifier는 레이아웃 계층 구조 내에서 임의의 구조에 적용해도 되는 것이 아니라, 컴포넌트 내 최상위 컴포저블에 적용해야 합니다. 일단 컴포넌트 자체가 덜 직관적으로 동작하기 때문에 예기치 않은 동작이 발생하거나, 해당 컴포넌트를 사용하는 미래의 본인 혹은 다른 팀원이 오용하게 될 가능성이 매우 높습니다.

예를 들어, 둥근 모양을 갖도록 설계된 RoundedButton 컴포넌트를 떠올려 볼 수 있습니다.

그림 313. CorrectModifierUsage.kt

```
1 @Composable
2 fun RoundedButton(
3     modifier: Modifier = Modifier, // 외부에서 전달된 modifier
4     onClick: () -> Unit
5 ) {
6     Button(
7         // Button에 modifier와 clip 적용
8         modifier = modifier.clip(RoundedCornerShape(32.dp)),
9         onClick = onClick
10    ) {
11        Text(
12            modifier = Modifier.padding(10.dp), // Text 내부 패딩
```

```
13         text = "Rounded"  
14     )  
15 }  
16 }
```

여기서 modifier는 컴포넌트의 가장 바깥쪽 요소인 Button에 올바르게 적용하고 있습니다. 그러나 modifier를 그 보다 아래 계층 컴포넌트인 Text에 직접 적용하면 다음과 같은 문제가 발생할 수 있습니다.

그림 314. IncorrectModifierUsage.kt

```
1 @Composable  
2 fun RoundedButton(  
3     modifier: Modifier = Modifier,  
4     onClick: () -> Unit  
5 ) {  
6     Button(  
7         // Button 자체에는 기본 Modifier만 적용됨  
8         modifier = Modifier.clip(RoundedCornerShape(32.dp)),  
9         onClick = onClick  
10    ) {  
11        Text(  
12            // Text에 외부 modifier 적용 (잘못된 사용)  
13            modifier = modifier.padding(10.dp),  
14            text = "Rounded"  
15        )  
16    }  
17 }
```

RoundedButton의 이름 자체가 Button을 명시적으로 표기하고 있기 때문에, Modifier의 스타일링 적용 기준 또한 버튼 수준에서 이루어져야 합니다. 계층 구조 내에서 Modifier를 일관성 없게 적용하면 해당 컴포넌트를 사용하는 개발자를 혼란스럽게 하고 의도하지 않은 결과를 초래할 수 있습니다. 반드시 한 번은 문제가 생길 수밖에 없습니다.

2. 단일 Modifier 매개변수 사용하기

컴포저를 내 다른 요소를 제어하기 위해 여러 Modifier 매개변수를 제공하는 것이 논리적으로 보일 수 있지만, 이러한 접근 방식은 불필요한 복잡성을 유발하고 명확성을 감소시킬 수 있습니다.

그림 315. MultipleModifiers.kt

```
1 @Composable
2 fun RoundedButton(
3     modifier: Modifier = Modifier, // 버튼용 Modifier
4     textModifier: Modifier = Modifier, // 텍스트용 Modifier (권장하지 않음)
5     onClick: () -> Unit
6 ) {
7     Button(
8         modifier = modifier.clip(RoundedCornerShape(32.dp)),
9         onClick = onClick
10    ) {
11        Text(
12            modifier = textModifier.padding(10.dp),
13            text = "Rounded"
14        )
15    }
16 }
```

여러 Modifier 매개변수를 사용하는 대신 슬롯 기반 접근 방식을 통해 유연성을 제공하는 것이 좋습니다. 이를 통해 사용자는 외부 커스텀을 위한 단일 Modifier를 유지하면서 내부 콘텐츠를 정의할 수 있습니다.

그림 316. SlotBasedApproach.kt

```
1 @Composable
2 fun RoundedButton(
3     modifier: Modifier = Modifier,
4     onClick: () -> Unit,
5     // content 슬롯을 통해 내부 콘텐츠 제공
6     content: @Composable RowScope.() -> Unit
7 ) {
8     Button(
9         modifier = modifier.clip(RoundedCornerShape(32.dp)),
10        onClick = onClick,
11        content = content // content 람다 전달
12    )
13 }
14
15 // 사용 예시
16 RoundedButton(onClick = { /* ... */ }) {
17     // 여기에 버튼 내부 콘텐츠 정의 (아이콘 및 텍스트 등)
18     Icon(Icons.Filled.Add, contentDescription = null)
19     Spacer(Modifier.width(4.dp))
20     Text("Add Item")
21 }
```

이러한 접근 방식은 API를 더 깔끔하게 유지하고 유연하게 만들며 예측 가능성 원칙을 부여합니다.

3. 여러 컴포넌트에서 Modifier 매개변수 재사용 하지 않기

경우에 따라서는 동일한 Modifier 체인을 여러 컴포저블에서 재사용⁶⁷하기 위해 Modifier를 변수로 추출하고 상위 범주로 호이스팅하여 이득을 취할 수도 있습니다. 그러나 흔한 실수는 레이아웃의 여러 컴포저블에서 동일한 Modifier 매개변수 인스턴스를 재사용하는 것입니다.

⁶⁷<https://developer.android.com/develop/ui/compose/modifiers#reusing-modifiers>

Modifier 생성을 줄여서 효율적으로 보일 수 있지만 의도하지 않은 사이드 이펙트와 예측할 수 없는 동작을 유발할 수 있습니다.

예를 들어, 아래와 같은 예시를 떠올려 볼 수 있습니다.

그림 317. ReusedModifier.kt

```
1 @Composable
2 fun MyButtons(
3     modifier: Modifier = Modifier, // 단일 Modifier 매개변수
4     onClick: () -> Unit
5 ) {
6     Column(modifier = modifier) { // Column에 modifier 적용
7         Button(
8             modifier = modifier, // Button에도 동일한 modifier 적용 (문제 발생 가능)
9             onClick = onClick
10        ) {
11            Text(
12                modifier = modifier.padding(10.dp), // Text에도 동일한 modifier 적용
13                // → (문제 발생 가능)
14                text = "Rounded"
15            )
16        }
17     Button(
18         modifier = modifier, // 두 번째 Button에도 동일한 modifier 적용
19         onClick = onClick
20     ) {
21        Text(
22            modifier = modifier.padding(10.dp), // 두 번째 Text에도 동일한 modifier
23            // → 적용
24            text = "Not Rounded"
25        )
26    }
}
```

언뜻 보기에는 괜찮아 보이지만 호출자 쪽에서 modifier를 수정하면 언제든지 개발자의 의도와는 전혀 다른 결과가 발생할 수 있습니다. 예를 들어, 호출자 쪽에서 아래와 같이 Modifier 객체를 넘긴다고 해봅시다.

그림 318. ParentComposable.kt

```
1 MyButtons(  
2     modifier = Modifier  
3         .clip(RoundedCornerShape(32.dp)) // 최상위 Column과 모든 자식에 적용됨  
4         .background(Color.Blue)  
5 ) {}
```

동일한 Modifier 인스턴스가 모든 중첩된 컴포저블에 적용되므로 최상위 수준의 변경 사항이 의도하지 않은 방식으로 전체 계층 구조에 영향을 미칩니다. 따라서 각 컴포저블에는 자체 Modifier 인스턴스가 필요합니다.



Additional Tips: 우리가 개발을 학습할 때 “불필요한 인스턴스 생성을 최소화하고, 최대한으로 재활용하여 프로그램의 성능을 최적화 하자”라는 이론이 Modifier를 설계할 때는 조금 다르게 적용되어야 합니다. 애초에 Compose UI 컴포넌트들은 각 컴포넌트 별로 각자 고유한 Modifier를 갖도록 설계되었다는 점을 주목해야 합니다. 따라서 비슷한 동작을 하는 UI 컴포넌트들이 쪽 나열되어 있다고 하더라도, 해당 컴포넌트마다 각자의 새로운 Modifier를 사용하는 것이 이상적입니다. 실제로 Modifier 객체를 많이 사용한다고 앱에 아주 큰 성능적인 영향을 미치는 것도 아닙니다.

요약

Jetpack Compose에서 Modifier를 효과적으로 사용하면 컴포넌트가 예측 가능하고 유지보수하기 쉬워집니다. 각 동작을 올바르게 이해하고, API를 최대한 단순하게 유지하며, 의도하지 않은 동작을 방지하는 것이 깔끔하고 견고한 UI 컴포넌트를 구축하는 것의 핵심이며, Modifier 가이드라인을 팀 내에서 잘 정의해 놓으면 미래의 본인 또는 팀원들을 위해

아주 좋습니다. 이와 같은 주제에 대해 더 학습하고 싶으시다면 [Compose Component API Guidelines](#)⁶⁸ 및 [Jetpack Compose Rules from Twitter](#)⁶⁹를 읽어보시는 것을 권장합니다.

💡 Pro Tips for Mastery: 커스텀 Modifier 만들기

Modifier를 사용하면 개발자가 체이닝 방식으로 스타일을 원활하게 적용할 수 있어 커스텀 Modifier를 잘 만들면 다양한 컴포넌트에 활용하기 매우 편해집니다. 특히 프로젝트 전반적으로 자주 사용되는 스타일링에 대해 커스텀 Modifier를 하나 만들어두면 여러 시나리오에서 재사용이 쉽고 코드 일관성 또한 향상시킬 수 있습니다. 커스텀 Modifier는 Modifier 팩토리 메서드, composed {}, Modifier.Node 등을 통해 만들 수 있습니다.

Composable Modifier Factory

[Composable Modifier Factory](#)⁷⁰를 사용하면 애니메이션 및 테마 설정과 같은 상위 수준 기능을 활성화하는 커스텀 Modifier를 만들 수 있습니다. 해당 메서드는 Modifier가 연계적으로 적용될 수 있다라는 Modifier의 특성을 활용하여 컴포저블 함수 어디에나 쉽게 커스텀 Modifier가 사용될 수 있도록 합니다.

가령, 다음 함수는 컴포넌트가 활성화 또는 비활성화될 때 페이드 효과를 부여합니다.

그림 319. Composable Modifier Factory.kt

```

1 @Composable
2 fun Modifier.fade(enabled: Boolean): Modifier {
3     // enabled 상태에 따라 alpha 값 애니메이션
4     val alpha by animateFloatAsState(if (enabled) 1.0f else 0.5f) // 활성화 시 1.0,
5     // graphicsLayer를 사용하여 alpha 적용
6     return this.graphicsLayer { this.alpha = alpha }
7 }
```

위의 방식은 Compose 상태 및 애니메이션과의 직접적인 사용을 허용하기 때문에 동적인 UI 동작을 구현하는데 유용합니다.

⁶⁸<https://github.com/androidx/androidx/blob/androidx-main/compose/docs/compose-component-api-guidelines.md#accepter-modifier-parameter>

⁶⁹<https://mrmans0n.github.io/compose-rules/rules/#modifiers>

⁷⁰https://developer.android.com/develop/ui/compose/custom-modifiers#create_a_custom_modifier_using_a_composable_modifier_factory

composed {}

composed {} 함수는 Compose UI의 초창기 버전에서부터 커스텀 Modifier를 만드는 데 권장되었지만, 현재는 성능 문제로 인해 권장되는 방법은 아닙니다. 위에서 살펴본 Modifier 팩토리와 달리 **composed {}** 는 컴포지션 내에서 초기화 및 업데이트될 때 추가적인 recomposition 및 오버헤드를 유발합니다.

composed {} 의 사용법은 다음과 같습니다.

그림 320. composed.kt

```

1 // 현재는 권장되지 않는 방식입니다.
2 fun Modifier.customPadding(value: Dp) = composed {
3     // composed 블록 내에서 다른 Modifier 호출
4     padding(value)
5 }
```

위의 접근 방식은 컴포지션 내에서 동적인 변경을 허용하지만, 상태를 핸들링하는 Modifier에 대해 더 효율적인 대안을 제공하는 **Modifier.Node**를 사용하여 마이그레이션 하는 것이 가장 이상적입니다.

Modifier.Node

Modifier.Node⁷¹는 효율성과 생명주기 관리 기능으로 인해 커스텀 Modifier를 구현하는 가장 선호되는 방법입니다. 개발자가 성능을 유지하면서 레이아웃, 드로잉 및 상호 작용에 대한 저수준 제어가 있는 Modifier를 정의할 수 있습니다.

Modifier.Node 구현은 세 가지 구성 요소로 이루어집니다.

1. **Modifier.Node**: Modifier의 동작을 정의합니다.
2. **ModifierNodeElement**: 노드를 생성하거나 업데이트하기 위한 상태를 저장하지 않는 팩토리를 제공합니다.
3. **Modifier Factory**: UI에서 사용할 Modifier에 대한 익스텐션을 정의합니다.

다음 예시는 **Modifier.Node**를 사용하여 원을 그리는 Modifier를 구현합니다.

⁷¹<https://developer.android.com/develop/ui/compose/custom-modifiers#implement-custom>

그림 321. ModifierNode.kt

```
1 import androidx.compose.ui.Modifier
2 import androidx.compose.ui.draw.DrawModifier
3 import androidx.compose.ui.graphics.Color
4 import androidx.compose.ui.graphics.drawscope.ContentDrawScope
5 import androidx.compose.ui.node.DrawModifierNode
6 import androidx.compose.ui.node.ModifierNodeElement
7 import androidx.compose.ui.platform.InspectorInfo
8
9 // 1. Modifier Factory (익스텐션)
10 fun Modifier.circle(color: Color): Modifier = this.then(CircleElement(color))
11
12 // 2. ModifierNodeElement (Node 생성 및 업데이트)
13 private data class CircleElement(val color: Color) : ModifierNodeElement<CircleNode>()
14     {
15         // 노드 생성
16         override fun create(): CircleNode = CircleNode(color)
17
18         // 노드 업데이트 (색상 변경 시)
19         override fun update(node: CircleNode) {
20             node.color = color
21         }
22
23         // Layout Inspector 정보 (선택 사항)
24         override fun InspectorInfo.inspectableProperties() {
25             name = "circle"
26             properties["color"] = color
27         }
28
29 // 3. Modifier.Node (실제 그리기 로직 구현)
30 private class CircleNode(var color: Color) : DrawModifierNode, Modifier.Node() {
31     // DrawModifierNode 인터페이스 구현
32     override fun ContentDrawScope.draw() {
```

```
33     // 콘텐츠를 먼저 그린 후 원을 그림 (순서 변경 가능)
34     drawContent()
35     // 원 그리기 로직
36     drawCircle(color = color, radius = size.minDimension / 2)
37 }
38 }
39
40 // 사용 예시
41 @Composable
42 fun CircleExample() {
43     Box(
44         modifier = Modifier
45             .size(150.dp)
46             .circle(Color.Blue) // 커스텀 circle Modifier 사용
47             .background(Color.LightGray) // 배경 추가 (선택 사항)
48     ) {
49         // Box 내 다른 콘텐츠 (선택 사항)
50     }
51 }
```

위의 접근 방식은 **Modifier.Node**가 recomposition이 발생해도 상태를 보존하며, 생명주기에 대해 더 나은 핸들링을 제공하므로 **composed {}** 를 사용하는 것보다 훨씬 효율적입니다.



Additional Tips: 그렇다면 **composed {}** 를 완전히 걷어내고 **Modifier.Node**만 사용해야 하나요? 이는 현재 팀의 전반적인 상황에 따라 유동적으로 선택하시는 것이 좋습니다. 비즈니스 관점에서 당장 새로운 기능을 출시해야 하는 상황에 **Modifier.Node**를 개발하고 있을 시간이 없으면 **composed {}** 로 빠르게 개발하고 배포하는 것이 그 상황에서는 옳은 판단일 수도 있습니다.

요약

커스텀 Modifier를 만들면 미리 정의된 스타일과 동작의 재사용성이 향상됩니다. **Composable Modifier Factory**는 상태를 보유해야 하는 형태의 스타일링을 쉽게 통합할 수 있도록 하는 반면, **Modifier.Node**는 복잡한 동작에 대한 최적의 성능을 제공합니다. 이전에 권장되었던 **composed {}** 메서드는 비효율성으로 인해 되도록 사용하지 않는 것이 좋습니다. 따라서, **Modifier.Node**를 활용하여 개발자는 원활한 성능을 유지하면서 매우 효율적이고 재사용 가능한 커스텀 Modifier를 개발할 수 있습니다.

Q) 27. Layout이란 무엇인가요?

Layout 컴포저블은 자식 컴포저블의 크기 측정 및 위치 지정에 대한 완전한 제어를 제공하는 저수준 API입니다. 미리 정의된 동작을 제공하는 Column, Row 또는 Box와 같은 고수준 레이아웃 컴포넌트와 달리 Layout을 사용하면 개발자가 특정 요구 사항에 맞는 커스텀 레이아웃을 만들 수 있습니다.

Layout 작동 방식

Layout 컴포저블은 자식 컴포저블이 측정되고 배치되는 방식을 정의하는 함수를 제공합니다. 두 가지 주요 단계로 구성됩니다.

1. **측정 단계(Measurement Phase)**: 부모가 제공한 제약 조건에 따라 각 자식 컴포저블의 크기를 결정합니다.
2. **배치 단계(Placement Phase)**: 사용 가능한 공간 내에 각 자식 컴포저블을 배치합니다.

기본적인 Layout 컴포저블은 다음과 같이 구성됩니다.

그림 322. CustomLayout.kt

```
1 @Composable
2 fun CustomLayout(
3     modifier: Modifier = Modifier,
4     content: @Composable () -> Unit // 자식 컴포저블을 받는 람다 함수
5 ) {
6     Layout(
7         content = content, // 자식 컴포저블 콘텐츠
8         modifier = modifier // 외부에서 전달된 Modifier 적용
9     ) { measurables, constraints -> // 측정 가능한 자식 목록과 부모 제약 조건
10         // 1. 자식 측정
11         val placeables = measurables.map { measurable ->
12             // 각 자식에게 제약 조건을 전달하여 측정
13             measurable.measure(constraints)
14         }
15
16         // 2. 레이아웃 크기 결정
17         // 예: 모든 자식 높이의 합으로 높이 결정, 너비는 최대 제약 조건 사용
18         val width = constraints.maxWidth
19         val height = placeables.sumOf { it.height }
20
21         // 결정된 크기로 레이아웃 정의
22         layout(width, height) {
23             // 3. 자식 배치
24             var yPosition = 0
25             placeables.forEach { placeable ->
26                 // 각 자식을 (0, yPosition)에 배치
27                 placeable.placeRelative(x = 0, y = yPosition)
28                 // 다음 자식의 y 위치 업데이트
29                 yPosition += placeable.height
30             }
31         }
32     }
}
```

Layout의 주요 구성 요소

1. **자식 측정(Measuring Children)**: measure() 함수는 부모로부터 제약 조건을 적용하여 각 자식 컴포저블을 측정하는 데 사용됩니다.
2. **레이아웃 크기 결정(Determining Layout Size)**: layout() 함수는 레이아웃의 최종 너비와 높이를 정의합니다.
3. **자식 배치(Placing Children)**: placeRelative() 함수는 레이아웃 내에서 각 자식 컴포저블이 배치될 위치를 결정합니다.

Layout 유즈 케이스

Layout 컴포저블은 Column, Row, Box와 같은 표준 레이아웃 컴포넌트가 달성하지 못하는 특수한 디자인 요구 사항에 필요한 커스텀 컴포넌트를 구현해야하는 시나리오에서 유용합니다. 자식 컴포저블이 측정되고 배치되는 방식에 대한 완전한 제어가 필요한 경우 Layout을 사용하여 커스텀 측정 및 배치 로직을 정의할 수 있습니다.

비표준 배열(가령, 지그재그로 그려지는 그리드), 겹치는 요소 또는 콘텐츠에 따라 동적으로 크기가 조절되는 컴포넌트와 같이 복잡한 UI 구조를 구현할 때 특히 유용합니다. Layout은 제약 조건 및 측정에 직접 접근할 수 있으므로 개발자는 recomposition 및 불필요한 재측정을 효율적으로 관리하여 UI 성능을 최적화할 수 있습니다.

또한 특정 동작 및 스타일링을 캡슐화하는 재사용 가능한 커스텀 레이아웃 컴포넌트를 구축하여 UI 코드를 더 깔끔하고 유지 관리하기 쉽게 만드는 데 유용합니다. 디자인에 정밀한 정렬 규칙, 적응형 레이아웃 또는 기존 레이아웃 컴포저블로 달성할 수 없는 완전히 고유한 배열이 필요한 경우 Layout을 사용하는 것이 적합합니다.

요약

Layout 컴포저블은 측정 및 배치에 대한 세분화된 제어를 제공하여 커스텀 레이아웃을 개발하는 데 사용됩니다. 개발자가 자식 크기 및 위치 지정 방법을 정의할 수 있도록 하므로 표준 레이아웃 컴포넌트를 넘어서는 커스텀 UI를 개발해야 하는 경우 유용합니다.

실전 질문

Q) Row 또는 Column과 같은 표준 컴포넌트 대신 Layout 컴포저블로 컴포넌트를 구현해 본 경험이 있나요? 어떤 장점이 있나요?

Q) LazyVerticalGrid를 사용하여 달성할 수 없는 지그재그 형태의 그리드 레이아웃을 구축한다고 가정했을 때, Layout을 사용하여 어떻게 구현해 볼 수 있을까요?

Pro Tips for Mastery: SubcomposeLayout의 개념 및 동작 원리

SubcomposeLayout은 레이아웃 내에서 동적 컴포지션을 허용하는 저수준 API입니다. UI 컴포넌트가 부모 레이아웃 패스와 독립적으로 하위 컴포넌트를 recompose해야 할 때 주로 사용됩니다. 이는 자식 콘텐츠 크기가 비동기 데이터에 의존하거나 여러 측정 패스가 필요할 때 특히 유용합니다.

SubcomposeLayout 작동 방식

Jetpack Compose에서 표준 레이아웃 시스템은 각 컴포저블이 recomposition당 한 번 측정되는 단일 측정 모델을 따릅니다. 그러나 특정 UI 구조는 자식 컴포저블을 여러 번 측정해야 하거나 레이아웃 제약 조건이 정해질 때까지 컴포지션을 지연시켜야 합니다. SubcomposeLayout은 측정 단계 중 자식의 컴포지션을 제약 조건이 정해질 때까지 지연하는 것을 허용하여 특수한 경우에 유연성을 제공합니다.

SubcomposeLayout의 일반적인 사용 사례는 다음과 같습니다.

- 일반적인 Layout 또는 LayoutModifier로 처리할 수 없는 컴포지션 중 부모가 가지는 제약 조건에 접근해야 하는 경우(가령, BoxWithConstraints와 같이 부모 컴포저블의 제약 조건을 자식 컴포저블이 알아야 하는 동작하는 형태).
- 한 자식의 크기에 따라 다른 자식을 측정하거나 배치해야 하는 경우. SubcomposeLayout을 사용하는 대표적인 이유 중 하나입니다.
- 긴 리스트에서 보이는 항목만 렌더링 하고 필요할 때까지(스크롤 등일 발생할 때까지) 다른 항목을 지연시키는 등 사용 가능한 공간을 기반으로 항목을 지연 구성(lazily compose)하려는 경우.
- 컴포저블 크기가 컴포지션 타임에 동적으로 정해져야 하는 콘텐츠의 경우.
- 레이아웃 로직이 여러 독립적인 단계에 걸쳐 자식을 측정하고 배치해야 하는 경우.
- 입력이 변경될 때만 recompose 되어야 하는 LazyList의 헤더와 같은 동적 UI를 처리하는 경우.

사용 예제

아래는 SubcomposeLayout을 사용하여 자식 컴포저블을 동적으로 측정하는 방법을 보여줍니다.

그림 323. SubcomposeLayoutExample.kt

```
1 @Composable
2 fun DynamicContentLayout() {
3     SubcomposeLayout { constraints -> // 부모로부터 받은 제약 조건
4         // "content" 슬롯에 대한 하위 컴포지션 수행
5         val placeables = subcompose("content") {
6             // 여기에 동적으로 표시될 콘텐츠 정의
7             Text(text = "Hello, skydoves!")
8         }.map { // 각 하위 컴포저블 측정
9             it.measure(constraints)
10        }
11
12         // 측정된 크기 계산 (여기서는 첫 번째 요소만 사용)
13         val measurable = placeables.firstOrNull()
14
15         // 레이아웃 크기 결정
16         val width = measurable?.width ?: 0
17         val height = measurable?.height ?: 0
18
19         // 레이아웃 배치
20         layout(width, height) {
21             // 측정된 자식 컴포저블 배치
22             measurable?.placeRelative(0, 0)
23         }
24     }
25 }
```

위의 예제를 하나씩 살펴보겠습니다.

- subcompose("content")는 텍스트 콘텐츠를 동적으로 구성합니다.

- `measure` 함수는 주어진 제약 조건에 따라 자식 크기를 계산합니다.
- `layout` 함수는 측정된 자식을 적절하게 배치합니다.

고려 사항

`SubcomposeLayout`은 상당한 유연성을 제공하지만 잠재적인 성능 비용으로 인해 신중하게 사용하는 것이 좋습니다. 자식 컴포넌트를 여러 번 재측정하고 구성할 수 있으므로 불필요한 `recomposition`이 발생하여 성능을 저하를 유발할 수 있습니다. 따라서, 여러 횟수의 측정이 요구되는 동적인 UI 컴포넌트와 같이 표준 레이아웃으로 충분하지 않은 경우에 가장 적합합니다. `SubcomposeLayout`을 사용할 때는 효율성을 유지하기 위해 필요할 때만 `recomposition`이 발생하도록 해야 합니다.

요약

`SubcomposeLayout`은 레이아웃 패스 내에서 동적 하위 컴포지션을 허용하는 유용한 API입니다. 콘텐츠 측정 및 `recomposition`을 부모 레이아웃과 분리해야 하는 시나리오에 가장 적합합니다. 상당한 유연성을 제공하지만 성능 함정을 피하기 위해 신중하게 사용해야 합니다.

Q) 28. Box에 대해서 아는 대로 다 설명해 주세요.

`Box`는 여러 자식 컴포저블을 부모 내에 자유롭게 쌓을 수 있도록 설계된 Jetpack Compose의 기본적인 레이아웃 컴포넌트입니다. 자식 컴포저블을 상대적으로 배치하여 오버레이 효과, 정렬 제어 및 레이어링을 가능하게 합니다. 이로 인해 `Box`는 배경, 이미지 위의 아이콘 또는 플로팅 UI 컴포넌트가 필요한 많은 시나리오에 특히 유용합니다.

Box 동작 방식

`Box`는 기본적으로 자식을 왼쪽 상단 정렬로 배열하지만 `contentAlignment` 매개변수를 사용하여 정렬을 커스텀할 수 있습니다. 또한 `Modifier` 속성을 통해 크기, 패딩, 배경 및 클릭 상호 작용을 커스텀할 수 있습니다.

자식을 정해진 순서대로 정렬하는 `Column` 및 `Row`와 달리 `Box`는 자식을 스택처럼 오버레이합니다. 이로 인해 UI 컴포넌트를 레이어링하는 데 이상적입니다.

사용 예제

다음 예제는 `Box`를 사용하여 이미지 위에 텍스트를 오버레이하는 방법을 보여줍니다.

그림 324. BoxExample.kt

```
1 @Composable
2 fun ImageWithOverlay() {
3     Box(
4         modifier = Modifier.size(200.dp), // Box 크기 설정
5         contentAlignment = Alignment.BottomCenter // 콘텐츠 정렬 (아래 중앙)
6     ) {
7         // 배경 이미지
8         Image(
9             painter = painterResource(id = R.drawable.skydoves_image),
10            contentDescription = "Background Image",
11            contentScale = ContentScale.Crop, // 이미지 스케일링 (선택 사항)
12            modifier = Modifier.matchParentSize() // Box 크기에 맞춤
13        )
14         // 오버레이 텍스트
15         Text(
16             text = "Hello, skydoves!",
17             color = Color.White,
18             modifier = Modifier
19                 .background(Color.Black.copy(alpha = 0.5f)) // 반투명 배경
20                 .padding(8.dp) // 텍스트 패딩
21                 // Box 내에서 정렬 (선택 사항, contentAlignment이 이미 설정됨)
22                 // .align(Alignment.BottomCenter)
23         )
24     }
25 }
```

결과는 아래와 같습니다.



그림 325. compose-box

위의 예제는 다음과 같이 동작합니다.

- Box는 Image와 Text가 모두 Box의 사이즈 내에 위치하도록 보장합니다.
- contentAlignment = Alignment.BottomCenter는 모든 내부 콘텐츠를 Box의 하단 중앙에 배치합니다.
- Text에는 더 나은 가독성을 위해 반투명 배경을 사용하고 있습니다.

주요 특징

Box는 UI 디자인에 다양한 용도로 사용할 수 있는 대표적인 두 가지 기능을 제공합니다. 첫째, 단일 레이아웃 내에 여러 자식 컴포저블을 겹치도록 쌓을 수 있어 UI 컴포넌트 오버레이에 유용합니다. 둘째, Box에서 제공하는 contentAlignment 또는 자식 수준의 컴포저블에서 사용이 가능한 Modifier.align()을 통해 정렬 제어를 구현할 수 있습니다.

요약

Box는 부모 내에 UI 컴포넌트를 쌓을 수 있도록 설계된 유용하면서도 간단한 레이아웃 컴포저블입니다. 콘텐츠 오버레이, 배경 효과 생성 및 요소 정렬 제어에 특히 유용합니다.

실전 질문

Q) 어떤 시나리오에서 Column이나 Row 대신 Box를 사용하는 것이 효율적이고, Box는 자식 컴포저블을 어떻게 다르게 처리하나요?

Q) Box의 contentAlignment 매개변수는 자식 컴포저블에서 각각 Modifier.align()을 사용하는 것과 어떻게 다른가요? 둘 다 함께 사용할 수 있나요?

Pro Tips for Mastery: BoxWithConstraints 살펴보기

BoxWithConstraints는 컴포지션 중 부모의 레이아웃 제약 조건에 대한 접근을 제공하는 고급 레이아웃 API입니다. 일반 Box와 달리 개발자가 사용 가능한 공간 및 크기 제약 조건에 따라 동적으로 UI 결정을 내릴 수 있도록 하여 반응형 디자인 및 적응형 레이아웃에 유용합니다.

BoxWithConstraints 작동 방식

BoxWithConstraints 컴포저블은 컨텐츠 람다 함수 내에서 maxWidth, maxHeight, minWidth, minHeight등과 같은 프로퍼티를 담고 있는 Constraints 스코프를 제공합니다. 해당 값들은 해당 BoxWithConstraints 컴포저블이 랜더링 될 수 있는 크기를 나타내므로 주어진 제약 조건에 따라 자식 UI들을 유동적으로 스타일링할 수 있습니다.

이로 인해 BoxWithConstraints는 UI 컴포넌트가 화면 크기나 제약 조건 또는 부모 레이아웃 크기에 따라 동적으로 랜더링 되어야 하는 상황에서 특히 유용합니다.

사용 예제

다음 예제는 너비에 제약 조건에 따라 텍스트 크기를 변경하기 위해 BoxWithConstraints를 활용하는 방법입니다.

그림 326. BoxWithConstraintsExample.kt

```
1 @Composable
2 fun ResponsiveText() {
3     BoxWithConstraints(
4         modifier = Modifier.fillMaxWidth()
5     ) {
6         // maxWidth 제약 조건에 따라 텍스트 크기 결정
7         val textSize = if (maxWidth < 300.dp) 14.sp else 20.sp
8         Text(
9             text = "Hello, skydoves!",
10            fontSize = textSize
11        )
12    }
13 }
```

위의 예제는 다음과 같은 동작을 포함합니다.

- BoxWithConstraints 컴포저블은 해당 컴포저블 내에서 사용할 수 있는 최대 가로 사이즈인 maxWidth를 제공합니다.
- maxWidth가 300.dp 미만이면 텍스트 크기가 14.sp로 설정되고, 그렇지 않으면 20.sp로 설정됩니다.
- 이를 통해 텍스트가 사용 가능한 화면 너비에 따라 동적으로 변화할 수 있도록 합니다.

주요 특징

BoxWithConstraints는 컴포지션 스코프 내에서 레이아웃 제약 조건을 제공하여 반응형 디자인을 가능하게 합니다. 개발자가 사용 가능한 사이즈에 따라 동적으로 변화하는 UI 컴포넌트를 만들 수 있도록 합니다. 실시간으로 제약 조건 값을 제공하므로 화면 크기에 따라 조건부로 레이아웃을 변경하거나, 타이포그래피를 조정하거나, UI 컴포넌트를 재배열하는 데 사용할 수 있습니다.

그러나 BoxWithConstraints는 일반 Box에 비해 오버헤드를 발생시키며 제공하는 기능이 더 많다고 모든 상황에서 Box를 대체하기 위한 용도로 사용해서는 절대 안 됩니다. BoxWithConstraints의 내부 구현을 살펴보면 결국 SubcomposeLayout을 사용하여 구현되었기 때문입니다.

그림 327. BoxWithConstraints Internals.kt

```

1 @Composable
2 fun BoxWithConstraints(
3     modifier: Modifier = Modifier,
4     contentAlignment: Alignment = Alignment.TopStart,
5     propagateMinConstraints: Boolean = false,
6     content: @Composable BoxWithConstraintsScope.() -> Unit
7 ) {
8     val measurePolicy = rememberBoxMeasurePolicy(contentAlignment,
9         → propagateMinConstraints) // MeasurePolicy 캐싱 가능
10    SubcomposeLayout(modifier) { constraints -> // SubcomposeLayout 사용
11        val scope = BoxWithConstraintsScopeImpl(this, constraints) // 제약 조건 전달
12        // subcompose를 사용하여 content 람다를 동적으로 컴포즈
13        val measurables = subcompose(Unit) { scope.content() }
14        // 측정 정책을 사용하여 측정 및 레이아웃 수행
15        with(measurePolicy) { measure(measurables, constraints) }
16    }
}

```

따라서 레이아웃 제약 조건에 접근하는 것이 필수적인 시나리오에 가장 적합하며, 불필요한 성능 비용을 피하기 위해 필요할 때만 사용해야 합니다.

요약

BoxWithConstraints는 부모 제약 조건에 따라 동적으로 UI 조정이 필요할 때 사용하기 적합한 컴포넌트입니다. 특히 UI 컴포넌트가 다른 화면의 크기나 레이아웃 제한에 따라 유동적으로 변화해야 하는 반응형 디자인을 만드는 데 꽤나 유용합니다. BoxWithConstraints를 적절한 상황에 잘 활용하면 개발자는 다양한 기기 구성에서 컴포저블이 유연하게 하고, 반응형 디자인을 설계할 수 있습니다.

Q) 29. Arrangement와 Alignment의 차이점에 대해서 설명해 주세요.

Jetpack Compose에서 Arrangement와 Alignment는 모두 레이아웃 내 UI 컴포넌트의 위치를 지정하기 위해 사용되지만 서로 다른 목적을 가지고 있습니다. 이들의 차이점을 잘 이해하면

UI 컴포넌트를 효과적으로 구성할 수 있습니다.

Arrangement란 무엇인가?

Arrangement는 Row 또는 Column과 같이 항목을 단일 방향으로 정렬하는 레이아웃 내에서 여러 자식 컴포저블의 간격 및 분포를 제어합니다. 레이아웃의 **주축(main axis)** 을 따라 자식이 어떻게 배치되는지를 결정합니다.

예를 들어, Row에서 Arrangement는 항목이 **수평으로** 간격을 두는 방식을 정의하고, Column에서는 **수직으로** 간격을 두는 방식을 정의합니다.

Arrangement 예제

그림 328. ArrangementExample.kt

```
1 @Composable
2 fun RowWithArrangement() {
3     Row(
4         modifier = Modifier.fillMaxWidth(),
5         // 자식 사이에 공간을 균등하게 분배
6         horizontalArrangement = Arrangement.SpaceBetween
7     ) {
8         Text(text = "Hello")
9         Text(text = "skydoves")
10    }
11 }
```

위의 예제는 다음과 같이 동작합니다.

- Arrangement.SpaceBetween은 두 Text 컴포저블이 사용 가능한 너비에 걸쳐 균등하게 간격을 두도록 보장합니다.
- 첫 번째 Text는 시작 부분에 배치되고 두 번째 Text는 끝 부분에 배치되며 그 사이에 공간이 분배됩니다.

Alignment란 무엇인가?

Alignment는 자식 컴포저블이 **교차축(cross axis)** 을 따라 부모 내에서 어떻게 배치되는지를 결정합니다. Box, Row, Column과 같은 레이아웃에서 컴포넌트가 컨테이너 경계에 상대적으로 어떻게 정렬되는지를 제어하는 데 사용됩니다.

- Row에서 Alignment는 **수직** 위치를 지정하는 데 영향을 미칩니다.
- Column에서 Alignment는 **수평** 위치를 지정하는 데 영향을 미칩니다.
- Box에서 Alignment는 수평 및 수직 위치 모두를 지정하는 데 영향을 미칩니다.

Alignment 예제

그림 329. AlignmentExample.kt

```
1 @Composable
2 fun ColumnWithAlignment() {
3     Column(
4         modifier = Modifier.fillMaxSize(),
5         // 자식을 수평 중앙에 정렬
6         horizontalAlignment = Alignment.CenterHorizontally
7     ) {
8         Text(text = "Hello")
9         Text(text = "skydoves")
10    }
11 }
```

위의 예제는 다음과 같이 동작합니다.

- horizontalAlignment = Alignment.CenterHorizontally는 두 Text 요소가 Column 내에서 수평 기준으로 중앙에 정렬되도록 보장합니다.
- Column은 여전히 자식을 **수직으로** 쌓지만 **수평 축**을 따라 중앙에 정렬됩니다.

주요 차이점

Arrangement는 **주축을 따라 여러** 자식 요소의 위치를 지정하는 데 사용됩니다(Row의 경우 수평, Column의 경우 수직). 반면 Alignment는 **교차축을 따라** 부모 내 **개별** 요소의 위치를 지정하는 데 사용됩니다.

Jetpack Compose에서 효율적이고 잘 구조화된 레이아웃을 구축하려면 이들을 올바르게 사용하는 것이 필수적입니다. Arrangement와 Alignment는 다른 목적을 가지고 있으므로 어떤 것을 사용해야 할지 결정할 때 종종 혼동을 일으킬 수 있습니다. 이들의 고유한 역할을 이해하면 더 나은 레이아웃 결정을 내리고 불필요한 복잡성을 피하는 데 도움이 됩니다.



Additional Tips: Arrangement와 Alignment의 이름도 비슷하고 처음에 기억하기 매우 헷갈립니다. “Arrangement”은 “어레인“지먼트 -> “Array(배열)”랑 발음이 비슷합니다. 따라서 항목들이 배열처럼 쪽 늘어진 **주 축(main axis)** 을 기준으로 생각하세요. 배열이 여러 아이템을 순차적으로 담는 것처럼, Arrangement는 Row에서는 가로 방향, Column에서는 세로 방향으로 여러 컴포저블이 어떻게 배치되는지를 제어합니다. 반면 “Alignment”은 얼“라인” -> “Line(선)”이라는 단어를 연상하게 합니다. 즉, **교차 축(cross axis)** 상에서의 십자가처럼 선을 교차시켜 **단일 아이템**의 위치를 정한다라는 느낌을 상상하시면 됩니다.

요약

Arrangement와 Alignment는 모두 UI 컴포넌트 위치 지정에 도움이 되지만 다른 축에서 작동하며 각자의 목적을 가집니다. Arrangement는 주축을 따라 여러 항목의 간격을 제어하고, Alignment는 교차축을 따라 부모 컨테이너 내 항목의 위치를 결정합니다.

실전 질문

Q) 항목이 화면 전체에 균등하게 간격을 두고 상단에 정렬되어야 하는 Row를 떠올려 보세요. Arrangement와 Alignment 조합을 어떻게 사용해서 구현하실 건가요?

Q) Row에서 horizontalAlignment를 설정해도 동작하지 않는데, 반면 Column에서는 작동하는 이유는 무엇인가요? Compose 레이아웃 규칙 중 어떤 것이 이러한 동작을 유발하나요?

Q) 30. Painter에 대해서 설명해 주세요.

Painter는 이미지, 벡터 그래픽 및 drawable 콘텐츠를 렌더링 하는 데 사용되는 추상 클래스입니다. 크기 조절, 틴팅 및 커스텀 드로잉 로직과 같은 기능을 지원하면서 이미지를 로드하고 표시하는 유연한 방법을 제공합니다.

Painter 작동 방식

전통적인 이미지 로딩 접근 방식과 달리 Painter는 이미지 리소스를 표시하는 UI 컴포넌트와 분리되어 있습니다. 이는 drawable 리소스를 로드하기 위한 painterResource 또는 벡터 기반 이미지를 처리하는 VectorPainter를 반환하는 rememberVectorPainter와 같이 다른 이미지 소스로 작업할 때 유용합니다.

Compose UI에서 다음과 같은 방식으로 페인터를 구현할 수 있습니다.

- res/drawable 폴더에서 이미지를 로드하기 위한 painterResource(id).
- 단색으로 영역을 채우기 위한 ColorPainter(color).
- ImageVector에서 동적으로 VectorPainter를 생성하기 위한 rememberVectorPainter(image = ImageVector).

사용 예제

다음 예제는 Image 컴포저블과 함께 Painter를 사용하는 방법을 보여줍니다.

그림 330. PainterExample.kt

```
1 @Composable
2 fun DisplayImage() {
3     // drawable 리소스에서 Painter 로드
4     val painter: Painter = painterResource(id = R.drawable.skydoves_image)
5     Image(
6         painter = painter, // Image 컴포저블에 Painter 전달
7         modifier = Modifier.size(100.dp), // 이미지 크기 설정
8         contentDescription = "Sample Image", // 접근성을 위한 설명
9     )
10 }
```

위의 예제는 아래와 같이 동작합니다.

- painterResource 함수는 drawable 리소스를 사용하여 이미지를 로드합니다.
- Image 컴포저블은 지정된 크기로 이미지를 렌더링 하기 위해 Painter를 사용합니다.

벡터 이미지의 경우 rememberVectorPainter를 통해 VectorPainter를 생성할 수 있습니다.

그림 331. VectorPainterExample.kt

```
1 @Composable
2 fun DisplayVector() {
3     // Icons.Default.Star ImageVector에서 VectorPainter 생성
4     val vectorPainter: Painter = rememberVectorPainter(image = Icons.Default.Star)
5     Image(
6         painter = vectorPainter, // Image 컴포저블에 VectorPainter 전달
7         modifier = Modifier.size(50.dp), // 아이콘 크기 설정
8         contentDescription = "Vector Icon", // 접근성 설명
9     )
10 }
```

위의 예제는 아래와 같이 동작합니다.

- rememberVectorPainter는 ImageVector에서 VectorPainter를 생성합니다.
- Image 컴포저블은 VectorPainter를 사용하여 이미지의 퀄리티 손실이 없는 벡터 이미지를 렌더링 합니다.

주요 특징

Painter 객체는 drawable 리소스를 Compose UI에 렌더링 할 수 있는 형태로 나타내며 안드로이드의 전통적인 Drawable API를 대체하여 사용할 수 있습니다. 이미지나 그래픽이 렌더링되는 방식을 정의할 뿐만 아니라 이를 사용하는 컴포저블의 크기 측정 및 레이아웃에도 영향을 미칩니다.

커스텀 페인터를 만들려면 Painter 클래스를 상속받고 커스텀 그래픽 렌더링을 위해 필요한 DrawScope를 제공하는 onDraw 메서드를 구현해야 합니다. 이를 통해 컴포저블 내에서 콘텐츠

가 그려지는 방식을 완전히 커스텀할 수 있습니다. 자세한 내용은 [공식 문서](#)⁷²를 읽어보시면 도움 됩니다.

요약

Painter는 크기 조절 및 커스텀 유연성을 제공하면서 이미지 및 벡터 렌더링을 단순화하는 추상화 개념입니다. Painter 및 VectorPainter를 사용하여 개발자는 이미지 리소스를 효율적으로 로드하고 Compose 친화적인 방식으로 비트맵 및 벡터 기반 그래픽을 모두를 손쉽게 렌더링 할 수 있습니다.

실전 질문

Q) Jetpack Compose에서 커스텀 Painter를 만들어 본 적이 있나요? 어떤 시나리오에서 사용했고, 드로잉 로직을 어떻게 구현했나요?

Q) 31. 네트워크에서 받아오는 이미지는 어떻게 로딩하나요?

Jetpack Compose는 네트워크 이미지 로딩에 대해 따로 솔루션을 제공하지 않지만 **Coil**, **Glide**, **Landscapist**와 같은 서드파티 라이브러리를 사용하여 URL에서 이미지를 효율적으로 로드하고 표시할 수 있습니다. 해당 라이브러리는 Jetpack Compose 또는 Kotlin Multiplatform에서도 잘 동작하며 캐싱 및 플레이스홀더 처리와 같은 편리한 기능들을 제공합니다.

네트워크 URL로부터 이미지를 받아와 직접 로딩 시스템을 구현해 볼 수도 있지만, 네트워크에서 이미지를 다운로드하고, 해상도에 맞게 크기를 조절하거나, 캐싱 알고리즘, 렌더링 전략 및 효율적인 메모리 관리와 같은 여러 복잡한 작업 또한 직접 구현해야 합니다. 이러한 여러 가지 측면 때문에 캐싱 전략, 변환 및 비동기 로딩에 대한 지원과 함께 실제 프로덕션에 사용되어 수많은 유저들의 이슈 리포트를 거쳐 어느 정도 탄탄하게 검증이 끝난 솔루션을 사용하는 것이 오히려 리소스 절약 측면에서 압도적으로 유리합니다.

Coil

Coil⁷³은 Jetpack Compose 및 Kotlin Multiplatform에 최적화된 이미지 로딩 라이브러리입니다. 완전히 Kotlin으로 작성되었으며 API는 완전히 Kotlin 지향적으로 작성되었습니다.

⁷²<https://developer.android.com/develop/ui/compose/graphics/images/custompainter>

⁷³<https://github.com/coil-kt/coil>

주목할 만한 점은 내부적으로 OkHttp 및 Coroutines와 같이 안드로이드 프로젝트에서 이미 널리 사용되는 라이브러리를 사용하기 때문에 해당 기능들을 완전히 처음부터 구현하는 솔루션보다는 오히려 사이즈나 동작 측면에서 더 가벼울 수 있습니다. Coil은 Jetpack Compose도 지원하는데 변환, 애니메이션 GIF 지원, SVG 지원 및 비디오 프레임 지원과 같은 유용한 기능을 제공합니다.

그림 332. CoilExample.kt

```
1 import coil.compose.AsyncImage
2
3 AsyncImage(
4     model = "https://example.com/image.jpg", // 이미지 URL
5     contentDescription = null, // 접근성 설명
6     // 추가 옵션 (플레이스홀더, 오류 이미지, 변환 등)
7     // placeholder = painterResource(R.drawable.placeholder),
8     // error = painterResource(R.drawable.error),
9     // contentScale = ContentScale.Crop,
10    // modifier = Modifier.size(128.dp)
11 )
```

Glide

Glide⁷⁴는 예전부터 전통적인 XML 레이아웃의 네트워크 이미지 로딩 솔루션으로 가장 널리 사용되던 이미지 로딩 라이브러리입니다. Compose 지원을 제공하긴 하지만 마지막 업데이트는 2023년 9월로 현재 Compose 라이브러리는 여전히 베타 상태이며 유지 관리가 아예 되지 않고 있습니다. Glide는 애니메이션 GIF 지원, 플레이스홀더, 변환, 캐싱 및 효율적인 리소스 재사용과 같은 유용한 기능을 제공하여 안드로이드 애플리케이션에서 이미지를 처리하는 데 여전히 좋은 선택입니다.

⁷⁴<https://github.com/bumptech/glide>

그림 333. GlideExample.kt

```
1 import com.bumptech.glide.integration.compose.ExperimentalGlideComposeApi
2 import com.bumptech.glide.integration.compose.GlideImage
3
4 @OptIn(ExperimentalGlideComposeApi::class)
5 // ...
6
7 GlideImage(
8     model = myUrl, // 이미지 URL 또는 모델
9     contentDescription = getString(R.string.picture_of_cat), // 접근성 설명
10    modifier = Modifier.padding(padding).clickable(onClick = onClick).fillMaxSize(), //
11        ↪ 사이즈 조절 등
12    // 추가 옵션 (플레이스홀더, 오류, 변환 등)
13    // loading = placeholder(R.drawable.placeholder),
14    // failure = placeholder(R.drawable.error),
15    // requestBuilderTransform = { it.circleCrop() }
16 )
```

💡 Fun Facts: Glide는 Google에서 근무했던 엔지니어 단 한 명에 의해 관리되었습니다. 사실상 Google의 언오피셜 한 네트워크 이미지 솔루션이라고 봐도 무방합니다. 하지만 해당 라이브러리를 개발하던 엔지니어가 다른 회사로 이직하는 바람에 해당 라이브러리는 최소한의 유지보수만 이루어지고 있으며, 2023년 9월 이후 아예 업데이트가 없습니다.

Landscapist

Landscapist⁷⁵는 Glide, Coil 또는 Fresco의 코어 시스템을 사용하여 네트워크 또는 로컬 리소스에서 이미지를 가져와 Jetpack Compose 및 Kotlin Multiplatform에 가장 최적화된 성능으로 이미지를 로딩하는 라이브러리입니다. Jetpack Compose의 성능에 최적화되어 있으며 recomposition 오버헤드를 줄이기 위해 성능에 민감한 부분까지 모두 고려되었습니다.

⁷⁵<https://github.com/skydoves/landscapist>

대부분의 컴포저블 함수는 **Restartable**이면서 **Skippable** 하기 때문에 Compose 컴파일러 메트릭에서 관찰된 바와 같이 최적화된 recomposition 성능을 제공합니다.

Landscapist는 **Baseline Profiles**를 통해 성능을 향상시켜 더 빠른 스타트업 퍼포먼스와 최적화된 런타임 실행을 보장합니다. 또한 **ImageOptions**, 상태 리스너, 커스텀 컴포저블, Android Studio 미리 보기 호환성, **ImageComponent** 및 **ImagePlugin**과 같은 모듈식 컴포넌트를 포함하여 광범위한 커스텀 옵션을 제공합니다. 또한 플레이스홀더, 애니메이션(circular reveal animation, 크로스페이드), 변환(블러), 색상 팔레트 추출과 같은 다양한 플러그인 기능을 지원하여 개발자가 이미지 로딩을 위해 고려해야 할 부분을 최소화하고 다른 부분에 집중할 수 있도록 돕습니다.

그림 334. LandscapistGlideExample.kt

```
1 import com.skydoves.landscapist.glide.GlideImage
2 import com.skydoves.landscapist.components.rememberImageComponent
3 import com.skydoves.landscapist.placeholder.shimmer.ShimmerPlugin
4 import com.skydoves.landscapist.placeholder.shimmer.Shimmer
5
6 GlideImage( // 또는 CoilImage, FrescoImage
7     imageModel = { imageUrl }, // 이미지 모델 제공 (람다 사용 권장)
8     modifier = modifier,
9     // ImageComponent를 사용하여 플러그인 구성
10    component = rememberImageComponent {
11        // 이미지 로딩 중 shimmer 효과 표시 (선택 사항)
12        +ShimmerPlugin(
13            baseColor = Color.DarkGray, // shimmer 기본 색상
14            highlightColor = Color.LightGray // shimmer 하이라이트 색상
15        )
16        // +CrossfadePlugin() // 크로스페이드 애니메이션 추가 (선택 사항)
17        // +PlaceholderPlugin.Failure(painterResource(id = R.drawable.error)) // 로딩 실패
18        // 시 이미지 표시
19    },
20    // 요청 실패 시 에러 텍스트 메시지 표시
21    failure = {
22        Text(text = "image request failed.")
23    },
24 }
```

```
23 // 로딩 중 플레이스홀더 (선택 사항)  
24 // loading = { CircularProgressIndicator() }  
25 )
```

💡 Fun Facts: Landscapist는 이 책의 저자인 **skydoves** (엄재웅)에 의해 개발되어 현재는 세계적인 제품인 X (Twitter), 아자르, 카카오뱅크, Stream Chat/Video SDK 등에 사용되고 있습니다. Jetpack Compose의 초기 알파 프리뷰 단계인 2020년에 처음 개발되어, 당시 최초의 Jetpack Compose 이미지 로딩 솔루션 중 하나입니다.

요약

이미지 로딩은 사용자 프로필 및 네트워크에서 다양한 콘텐츠를 로딩해야 하는 사례와 같이 현대 애플리케이션 개발에서 빼놓을 수 없습니다. Coil, Glide 또는 Landscapist를 사용하여 네트워크 또는 로컬 이미지에서 데이터를 받아와 효율적으로 이미지를 랜더링하고, 필요에 따라 변환, 애니메이션 GIF 지원, SVG 지원 및 비디오 프레임 지원과 같은 다양한 플러그인을 통해 기능을 더 풍부하게 가져갈 수 있습니다.

실전 질문

Q) Jetpack Compose에서 이미지를 로드하기 위해 사용해 본 서드파티 라이브러리는 무엇이 있으며, 다른 왜 해당 라이브러리를 사용하셨나요? 각 라이브러리 별로의 트레이드오프가 무엇이라고 생각하나요?

Q) 32. UI 버벅거림(jank)을 피하면서 리스트에 수백 개의 항목을 효율적으로 렌더링하려면 어떻게 해야 하나요?

Jetpack Compose에서 수백 또는 수천 개의 아이템을 표시할 때 Column과 같은 레이아웃을 사용하면 불필요한 컴포지션 및 렌더링으로 인해 성능 문제가 발생할 수 있습니다. Jetpack Compose는 UI 버벅거림을 피하고 효율성을 개선하기 위해 필요에 따라 아이템을 동적으로

구성하고 재활용하는 **LazyColumn**, **LazyRow**, **LazyGrid**와 같은 **Lazy List**⁷⁶라는 최적화된 리스트 컴포넌트를 제공합니다.

수직 나열을 위한 LazyColumn

LazyColumn은 현재 화면에 보이는 아이템만 랜더링 하고 화면 밖 항목을 재활용하여 대규모 아이템을 효율적으로 렌더링 하도록 설계되었습니다. 이는 메모리 사용량을 크게 줄이고 스크롤 성능을 향상시킵니다.

그림 335. LazyColumnExample.kt

```
1 @Composable
2 fun ItemList() {
3     LazyColumn {
4         // 1000개의 아이템 정의
5         items(1000) { index ->
6             Text(text = "Item #$index", modifier = Modifier.padding(8.dp))
7         }
8     }
9 }
```

위의 예제는 다음과 같이 동작합니다.

- **LazyColumn**은 보이는 항목만 로드하여 불필요한 컴포지션을 방지합니다.
- **items** 함수는 한 번에 모두 미리 로드하지 않고 1000개의 아이템을 필요할 때마다 동적으로 로드합니다.

수평 나열을 위한 LazyRow

수평 스크롤 리스트의 경우 **LazyRow**를 사용할 수 있습니다. **LazyColumn**과 유사하게 작동하여 현재 화면에 노출되어야 하는 필요한 항목만 랜더링 하도록 보장합니다.

⁷⁶<https://developer.android.com/develop/ui/compose/lists#lazy>

그림 336. LazyRowExample.kt

```
1 @Composable
2 fun HorizontalItemList() {
3     LazyRow {
4         items(500) { index ->
5             Text(text = "Item #$index", modifier = Modifier.padding(8.dp))
6         }
7     }
8 }
```

이와 같은 접근 방식으로 많은 수의 아이템을 랜더링 해야 하는 경우 아이템에 대한 UI 랜더링 지연을 통해 성능을 최적화합니다. 전통적인 XML에서의 RecyclerView와 비슷한 동작을 구현하지만, 실제로 구현하기 위해 작성해야 하는 코드는 훨씬 간단합니다.

그리드 레이아웃을 위한 LazyVerticalGrid

그리드 구조가 필요한 레이아웃의 경우 LazyVerticalGrid를 사용하여 아이템을 열과 행으로 구성하면서 LazyColumn/LazyRow와 유사하게 렌더링을 효율적으로 처리할 수 있습니다.

그림 337. LazyGridExample.kt

```
1 @Composable
2 fun GridItemList() {
3     LazyVerticalGrid(
4         // 고정된 3개의 열 정의
5         columns = GridCells.Fixed(3),
6         modifier = Modifier.fillMaxSize()
7     ) {
8         items(300) { index ->
9             Text(text = "Item #$index", modifier = Modifier.padding(8.dp))
10        }
11    }
12 }
```

마찬가지로, 현재 화면에 보여야 하는 그리드 항목만 렌더링 되도록 보장합니다. 수직이 아닌 수평 형태의 그리드 레이아웃을 구현하려면 LazyHorizontalGrid를 사용하시면 됩니다.

키(Keys)를 사용한 성능 최적화

기본적으로 각 아이템의 상태는 리스트 또는 그리드에서의 위치와 연결됩니다. 그러나 아이템이 삽입, 제거 또는 재정렬되는 등 데이터 세트가 변경되면 위치가 이동하기 때문에 아이템이 상태를 잃을 수 있습니다. 이와 같이 지속적으로 재활용되는 리스트에서 각 아이템마다 상태를 보존하려면 아이템에 고유한 **키(key)** 값을 할당하여 리스트에서의 위치가 수정되어도 불필요한 recomposition을 트리거하지 않고, 상태 또한 일관되게 유지할 수 있습니다.

그림 338. LazyColumnWithKeys.kt

```
1 @Composable
2 fun KeyedItemList(items: List<Item>) { // Item 데이터 클래스 가정
3     LazyColumn {
4         items(
5             items = items,
6             // 각 아이템에 고유한 키 값 제공
7             key = { item -> item.id } // item의 고유한 식별자 사용
8         ) { item ->
9             Text(text = item.name, modifier = Modifier.padding(8.dp))
10        }
11    }
12 }
13
14 // 예시 Item 데이터 클래스
15 data class Item(val id: String, val name: String)
```

이 예제에서 key = { it.id }는 각 아이템에 고유 식별자를 할당하여 리스트 아이템이 변경될 때 불필요한 recomposition을 피하고 성능을 최적화시킵니다. Lazy List를 최적화하기 위한 모범 사례는 공식 문서 [레이지 레이아웃 키 사용](#)⁷⁷을 참고하시면 좋습니다.

⁷⁷<https://developer.android.com/develop/ui/compose/performance/bestpractices#use-lazylist-keys>

 **Additional Tips:** LazyColumn 및 LazyRow와 같은 Lazy List 컴포넌트는 key 매개변수를 선택적으로 받고 있지만, 저자가 GDE 활동을 하며 들었던 구글 관계자들의 발언에 의하면 사실상 이는 필수적인 매개변수로 취급해야 합니다. 즉, Lazy List의 성능을 최대한으로 최적화시키고, 상태를 안정적으로 보장하기 위해서는 반드시 key 매개변수를 올바르게 사용해야 해야 Compose 팀에서 기대했던 만큼의 컴포넌트의 성능이 뽑힙니다. 주의해야 할 사항은 key 값은 **반드시** 아이템의 고유한 식별자로서 사용하셔야 합니다. 즉, 여러 아이템이 동일한 key 값을 할당받는다면 앱에서 **100% exception**이 발생하기 때문에, 유의해서 사용해야 합니다. 특히 백엔드에서 데이터를 내려받을 때 각 아이템마다의 고유한 id 값을 반드시 내려 달라고 요청하는 것이 좋고, 혹시라도 중복적인 id 값을 가진 응답이 내려올 경우 exception이 발생하고 앱 크래시로 이어질 가능성이 높기 때문에, items.distinctBy { it.id }와 같이 중복적인 아이템은 제거하고 Lazy List로 넘겨주는 방법 등이 있습니다.

요약

Jetpack Compose에서 대량의 아이템을 효율적으로 렌더링 하려면 Column 및 Row와 같은 레이아웃 대신 **LazyColumn**, **LazyRow**, **LazyGrid**와 같은 Lazy List를 사용해야 합니다. 성능을 최적화하고 각 아이템마다 상태를 보존하려면 **키(keys)** 를 적용하여 동적적으로 리스트를 처리할 때 최소한의 recomposition이 발생하도록 할 수 있습니다. 이러한 최적화를 효율적으로 활용하면 UI 버벅거림을 줄이고 부드럽고 효율적인 리스트 렌더링을 구현할 수 있습니다.

실전 질문

Q) 실시간으로 메시지를 처리해야 하는 채팅 화면을 개발한다고 가정해 봅시다. 데이터 레이어 쪽에서는 Flow를 통해서 지속적으로 새로운 채팅 메시지가 들어오고 있고, UI에서는 이를 즉시 랜더링 해야 합니다. 스크롤을 부드럽게 유지하고 최소한의 recomposition으로 성능이 최적화된 화면을 개발하기 위해 레이아웃을 어떻게 설계하실 건가요?

Q) LazyColumn 또는 LazyGrid에서 키를 사용하는 것이 리스트 아이템 업데이트 시 UI 성능과 안정성을 유지하는 데 어떤 도움이 되나요?

Q) 33. Lazy List를 활용하여 페이지네이션(pagination) 구현하는 방법을 설명해 주세요.

페이지네이션은 원활한 UI 성능을 유지하면서 리스트에서 대규모의 데이터를 효율적으로 처리하기 위해 필수적입니다. 구글에서 공식적으로 제공하는 [Jetpack Paging 라이브러리](#)⁷⁸를 포함하여 다양한 솔루션이 존재합니다. 그러나 서드파티 라이브러리 없이 사용자가 리스트를 스크롤하여 현재 로드한 마지막 아이템에 도달할 때 더 많은 데이터를 로드하여 무한 스크롤을 구현할 수 있습니다.

스크롤 위치 감지하기

일반적으로 페이지네이션을 직접 구현하기 위해서는 디스플레이 상으로 마지막으로 보이는 아이템에 도달하는 시점을 감지한 다음 추가적인 데이터를 로드하는 것입니다. 이는 LazyListState를 사용하여 아래와 같은 형태로 구현할 수 있습니다.

그림 339. PaginationExample.kt

```
1 @Composable
2 fun PaginatedList(viewModel: ListViewModel) {
3     val listState = rememberLazyListState() // LazyList의 스크롤 상태 기억
4     // ViewModel에서 아이템 목록과 로딩 상태 관찰
5     val items by viewModel.items.collectAsStateWithLifecycle()
6     val isLoading by viewModel.isLoading.collectAsStateWithLifecycle()
7     // 다음 페이지 로드 트리거를 위한 threshold (예: 마지막에서 2번째 아이템이 보일 때로
8     // 계산하여 미리 로드)
9     val threshold = 2
10    // 로드 필요 여부를 계산하는 derivedStateOf (불필요한 계산 방지)
11    val shouldLoadMore by remember {
12        derivedStateOf {
13            val layoutInfo = listState.layoutInfo
14            val totalItemCount = layoutInfo.totalItemCount
15            // 마지막으로 보이는 아이템 인덱스 가져오기
16        }
17    }
18 }
```

⁷⁸<https://developer.android.com/topic/libraries/architecture/paging/v3-overview>

```
15     val lastVisibleItemIndex = layoutInfo.visibleItemsInfo.lastOrNull()?.index
16     → ?: -1
17
18     // 로딩 중이 아니면서, 마지막 아이템 근처에 도달했을 때 true 반환
19     (lastVisibleItemIndex != -1 && lastVisibleItemIndex + threshold >=
20      → totalItemCount) && !isLoading
21   }
22
23   // 방법 1: shouldLoadMore 상태가 true로 변경될 때만 loadMoreItems 호출
24   LaunchedEffect(shouldLoadMore) { // shouldLoadMore를 key로 사용
25     if (shouldLoadMore) {
26       viewModel.loadMoreItems()
27     }
28
29   /* 방법 2: 또는 snapshotFlow 사용 (더 세밀한 제어 가능)
30   LaunchedEffect(listState) { // listState를 key로 사용
31     snapshotFlow { shouldLoadMore }
32       .distinctUntilChanged() // 상태 변경 시에만 방출
33       .filter { it } // true일 때만 진행
34       .collect {
35         viewModel.loadMoreItems()
36       }
37   }
38 */
39
40 LazyColumn(
41   state = listState, // LazyListState 연결
42   modifier = Modifier.fillMaxSize()
43 ) {
44   items(items) { item ->
45     Text(modifier = Modifier.padding(8.dp), text = "$item")
46   }
}
```

```
47  
48     // 로딩 중일 때 하단에 프로그레스 바 표시  
49     item {  
50         if (isLoading) {  
51             Box(modifier = Modifier.fillMaxWidth().padding(16.dp), contentAlignment  
52                 = Alignment.Center) {  
53                 CircularProgressIndicator()  
54             }  
55         }  
56     }  
57 }
```

위의 예제는 다음과 같이 동작합니다.

- LazyListState는 스크롤 위치를 모니터링하는 데 사용됩니다.
- derivedStateOf 또는 snapshotFlow는 마지막으로 보이는 아이템의 인덱스를 추적하고 threshold 등을 사용하여 로딩이 필요한 시점을 계산하고 관찰합니다.
- 계산된 임계값에 따라 특정 아이템 인덱스에 도달하면 loadMoreItems()가 트리거되어 다음 데이터를 요청합니다.

페이지네이션 관리를 위한 ViewModel

페이지네이션 로직을 관리하기 위해 ViewModel을 사용하여 데이터를 점진적으로 로드할 수 있습니다.

그림 340. ListViewModel.kt

```
1 class ListViewModel : ViewModel() {
2     // mutableStateListOf를 사용하여 Composes UI에서 아이템 변경 사항을 관찰 가능
3     private val _items = mutableStateListOf<Int>()
4     // 외부로 노출되는 읽기 전용 StateFlow
5     val items: StateFlow<List<Int>> = MutableStateFlow(_items)
6     // 로딩 상태 Flow
7     private val _isLoading = MutableStateFlow(false)
8     val isLoading: StateFlow<Boolean> = _isLoading.asStateFlow()
9
10    private var currentPage = 0 // 현재 페이지 번호
11
12    // 다음 페이지 로드 함수
13    fun loadMoreItems() {
14        // 이미 로딩 중이면 중복 요청 방지
15        if (_isLoading.value) return
16
17        _isLoading.value = true // 로딩 시작 상태 업데이트
18        viewModelScope.launch {
19            try {
20                delay(1000) // 네트워크 요청을 시뮬레이션 하기 위한 딜레이
21                // 새 아이템 생성 (실제로는 API 호출 결과 사용)
22                val newItems = List(20) { (currentPage * 20) + it }
23                _items.addAll(newItems) // 리스트에 새 아이템 추가
24                currentPage++ // 다음 페이지 번호 증가
25            } finally {
26                _isLoading.value = false // 로딩 완료 상태 업데이트
27            }
28        }
29    }
30
31    init {
32        // 초기 데이터 로드
33        // 예시를 위해 init block에서 호출하고 있지만, UI 화면의 상태에 따라
```

```
34     // Lazy initialization 방식을 통해 호출하는 것을 권장합니다.  
35     loadMoreItems()  
36 }  
37 }
```

위의 구현은 다음 동작을 포함합니다.

- `_items`는 새로운 페이지가 로드될 때마다 업데이트되는 `SnapshotStateList` 타입으로, `StateFlow`인 `items`를 통해서 `ViewModel` 외부로 읽기 전용 타입으로 노출됩니다.
- `loadMoreItems()`는 호출될 때 새 데이터를 가져옵니다.
- `isLoading`은 페이지가 이미 로드 중일 때 중복 요청을 방지합니다.

요약

`Lazy List`에서 추가적인 데이터 로드가 필요할 때를 감지하기 위해 `LazyListState`를 사용하여 페이지네이션을 효율적으로 구현할 수 있습니다. 이러한 접근 방식은 필요할 때만 새 데이터를 요청하여 불필요한 오버헤드 및 `recomposition`을 줄이고 성능을 최적화시킬 수 있습니다. 또한, `threshold` 값을 적절하게 설정해 주면, 유저들이 스크롤을 자연스럽게 넘기면서 마지막 아이템에 직접적으로 도달하기 전에 미리 데이터를 로드하여 더 부드러운 사용자 경험을 제공할 수 있습니다.

실전 질문

Q) 페이지네이션에서 다음 데이터를 로드하는 시점을 감지하기 위해 어떤 API 또는 상태 메커니즘을 사용하실 건가요?

Q) 페이지네이션에서 `LazyListState`는 어떤 역할을 하며, `derivedStateOf` 및 `snapshotFlow`는 데이터 로딩 로직 최적화에 어떻게 도움이 되나요? `snapshotFlow`를 사용할 때 `distinctUntilChanged()`를 함께 사용하면 어떤 이점이 있나요?

Q) 사용자가 목록을 빠르게 스크롤하다 보면 페이지네이션 중 중복 네트워크 호출이나 데이터 로딩이 발생할 가능성이 있는데 이를 어떻게 방지할 건가요?

Q) 34. Canvas는 어떤 역할을 하나요?

Canvas는 커스텀 그래픽, 애니메이션 및 시각 효과를 만들 수 있도록 개발자가 직접 원, 선, 도형 등을 그리거나 이미지를 렌더링 할 수 있는 일종의 스케치북과 같은 API입니다. 표준 UI 컴포넌트와 달리 Canvas는 DrawScope 인터페이스를 제공하여 **드로잉 커맨드**를 사용하여 렌더링을 세부적으로 제어할 수 있도록 합니다.

Canvas는 Jetpack Compose의 드로잉 시스템 매커니즘에 기반하여 drawRect, drawCircle, drawPath, drawText, drawImage와 같은 함수를 통해 전통적인 XML의 Canvas와 유사한 형태로 동작합니다. 기본적으로 제공되는 함수를 통해 개발자는 커스텀 도형, 이미지, 벡터 그래픽을 렌더링하고 색상, 크기, 스트로크 스타일 및 변환을 효율적으로 제어할 수 있습니다.

사용 예제

다음 예제를 통해 Canvas 컴포저블 내부에 간단한 원을 그리는 방법을 살펴보겠습니다.

그림 341. CanvasExample.kt

```
1 @Composable
2 fun DrawCircleCanvas() {
3     Canvas(modifier = Modifier.size(200.dp)) { // 고정 크기의 Canvas
4         // Canvas 중앙에 파란색 원 그리기
5         drawCircle(
6             color = Color.Blue,
7             radius = size.minDimension / 2, // Canvas 크기에 기반한 반지름
8             center = center // Canvas 중앙 좌표
9         )
10    }
11 }
```

위의 예제는 다음 동작을 포함합니다.

- Canvas 컴포저블의 사이즈는 200.dp로 지정됩니다.
- drawCircle은 캔버스 중앙에 파란색 원을 렌더링 하는 데 사용됩니다.
- size.minDimension / 2는 원이 캔버스 경계 내에 딱 들어맞도록 계산된 반지름 값입니다.

결과는 아래와 같습니다.

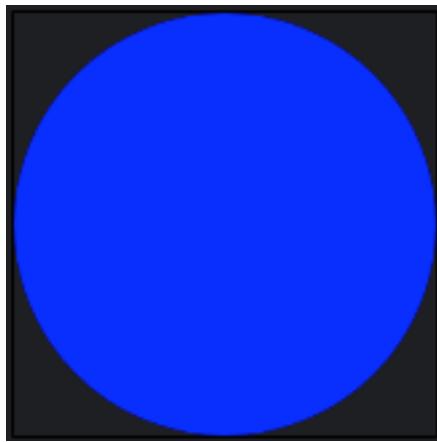


그림 342. compose-canvas0

기본 변환 (Basic Transformations)

Canvas 컴포저블은 개발자가 동적이고 상호 작용 가능한 UI 컴포넌트를 만들 수 있도록 다양한 변환 및 드로잉 함수를 제공합니다. 대표적인 기능은 다음과 같습니다.

- **크기 조절 (scale)**: 지정된 배율로 드로잉 대상 확대 또는 축소.
- **이동 (translate)**: 드로잉 영역 내에서 X 및 Y 축을 따라 대상 이동.
- **회전 (rotate)**: 피벗 점 주위로 대상 회전.
- **인셋 (inset)**: 패딩을 적용하여 드로잉 경계 조정.
- **다중 변환 (withTransform)**: 더 나은 성능을 위해 단일 작업에서 여러 변환 결합.
- **텍스트 그리기 (drawText)**: 정밀한 위치 지정 및 커스텀로 텍스트 수동 렌더링.

이와 같은 변환 API는 [컴포저블 생명주기의 Drawing 단계](#)⁷⁹ 중에 적용되는데, 이는 컴포넌트의 레이아웃 크기나 위치 지정에 영향을 미치지 않음을 의미합니다. 결과적으로 변환을 사용하여 크기나 위치를 수정해도 실제 레이아웃의 경계는 변경되지 않으며, 컴포넌트가 지정된 레이아웃 공간에서 서로 겹쳐지거나 벗어날 수 있습니다.

⁷⁹<https://developer.android.com/develop/ui/compose/phases>

요약

Canvas는 화면에 커스텀 그래픽을 렌더링하는 매우 유연한 방법을 제공합니다. 도형, 텍스트, 이미지 그리기 및 변환 기능을 활용하여 개발자는 풍부한 시각적 및 커스텀 경험을 만들 수 있습니다. 기존의 전통적인 XML 레이아웃에서 개발하는 커스텀 뷰와 같이, 일반적인 Compose UI 라이브러리가 제공하지 않는 형태의 커스텀 UI(가령, 차트나 컬러 피커 등)가 필요한 경우에 필수적입니다. Canvas API에 대한 더 학습하고 싶으시면 [Compose의 그래픽](#)⁸⁰ 문서를 참고하시길 바랍니다.

실전 질문

Q) 커스텀 애니메이션이 들어간 원형 프로그레스바를 Canvas를 사용해서 어떻게 구현할지 설명해 주세요.

Q) 35. graphicsLayer를 어떻게 활용하나요?

graphicsLayer는 개발자가 컴포저블에 변환, 클리핑 및 합성 효과를 적용할 수 있도록 하는 유용한 Modifier입니다. 컴포저블을 별도의 **드로잉 레이어**로 렌더링하여 작동하며, 격리된 렌더링, 캐싱 및 오프스크린 래스터화와 같은 최적화를 가능하게 합니다. 수동적으로 드로잉을 제어하는 Canvas와는 달리 graphicsLayer는 컴포저블의 구성 가능성을 유지하여 훨씬 선언적인 방법으로 컴포넌트의 모양을 수정할 수 있습니다.

graphicsLayer 작동 방식

컴포저블이 Modifier.graphicsLayer로 래핑되면 모든 드로잉 작업이 나머지 UI와는 별도로 분리되는 **격리된 레이어**를 생성합니다. 즉, **크기 조절, 이동, 회전, 알파 변경 및 클리핑**과 같은 변환을 주변 컴포저블에 영향을 주지 않고 적용할 수 있습니다. 또한 graphicsLayer는 하드웨어 가속을 사용하므로 과도한 recomposition을 트리거 하지 않고도 이러한 효과를 효율적으로 적용할 수 있습니다.

사용 예제

다음 예제는 graphicsLayer를 사용하여 Image 크기를 조절하는 방법을 보여줍니다.

⁸⁰<https://developer.android.com/develop/ui/compose/graphics/draw/overview#draw-image>

그림 343. GraphicsLayerScaleExample.kt

```
1 @Composable
2 fun ScaledImage() {
3     Image(
4         painter = painterResource(id = R.drawable.skydoves_image),
5         contentDescription = "Scaled Image",
6         modifier = Modifier
7             .size(200.dp) // graphicsLayer 전에 크기 지정 권장
8             .graphicsLayer {
9                 scaleX = 1.5f // 가로 1.5배 확대
10                scaleY = 1.2f // 세로 1.2배 확대
11            }
12        )
13 }
```

위의 예제는 다음과 같이 동작합니다.

- scaleX = 1.5f는 이미지를 가로로 1.5배 확대합니다.
- scaleY = 1.2f는 이미지를 세로로 1.2배 확대합니다.

변환 적용하기

graphicsLayer는 이동, 회전 및 원점 기반 크기 조절과 같은 다양한 변환을 가능하게 합니다. 각 변환을 예제와 함께 살펴보겠습니다.

이동 (요소 이동)

그림 344. GraphicsLayerTranslationExample.kt

```
1 @Composable
2 fun TranslatedImage() {
3     Image(
4         painter = painterResource(id = R.drawable.skydoves_image),
5         contentDescription = "Translated Image",
6         modifier = Modifier
7             .size(200.dp)
8             .graphicsLayer {
9                 translationX = 50.dp.toPx() // 오른쪽으로 50dp 이동
10                translationY = -20.dp.toPx() // 위쪽으로 20dp 이동
11            }
12        )
13 }
```

- translationX는 이미지를 오른쪽으로 50dp 이동합니다.
- translationY는 이미지를 위쪽으로 20dp 이동합니다.

회전 (X, Y 또는 Z 축 주위 회전)**그림 345. GraphicsLayerRotationExample.kt**

```
1 @Composable
2 fun RotatedImage() {
3     Image(
4         painter = painterResource(id = R.drawable.skydoves_image),
5         contentDescription = "Rotated Image",
6         modifier = Modifier
7             .size(200.dp)
8             .graphicsLayer {
9                 rotationX = 45f // X축 기준 45도 회전
10                rotationY = 30f // Y축 기준 30도 회전
11                rotationZ = 90f // Z축(화면 축) 기준 90도 회전
12            }
13 }
```

```
12         transformOrigin = TransformOrigin.Center // 회전 중심 설정 (선택 사항)
13     }
14 )
15 }
```

- rotationX는 이미지를 수평으로 회전합니다.
- rotationY는 이미지를 수직으로 회전합니다.
- rotationZ는 이미지를 화면 축 주위로 회전합니다.

클리핑 및 모양 지정 (Clipping and Shaping)

graphicsLayer는 Shape를 정의하고 clip 속성을 활성화하여 커스텀 클리핑을 지원합니다.

그림 346. GraphicsLayerClipExample.kt

```
1 @Composable
2 fun ClippedBox() {
3     Box(
4         modifier = Modifier
5             .size(200.dp)
6             .background(Color.Blue) // 배경을 먼저 적용해서 클리핑 영역을 살펴보는 예제
7             .graphicsLayer {
8                 clip = true // 클리핑 활성화
9                 shape = CircleShape // 원형 모양으로 클리핑
10            }
11            // .background(Color.Blue) // 여기에 배경을 적용하면 클리핑된 영역만 채워짐
12        )
13    }
```

- CircleShape은 콘텐츠가 원형 모양으로 클리핑되도록 보장합니다.
- clip = true 속성은 클리핑을 활성화합니다.

알파 (불투명도 제어)

graphicsLayer는 alpha를 사용하여 투명도를 조정할 수 있습니다. 1.0f는 완전히 불투명하고 0.0f는 완전히 투명합니다.

그림 347. GraphicsLayerAlphaExample.kt

```
1 @Composable
2 fun TransparentImage() {
3     Image(
4         painter = painterResource(id = R.drawable.skydoves_image),
5         contentDescription = "Transparent Image",
6         modifier = Modifier
7             .size(200.dp)
8             .graphicsLayer {
9                 alpha = 0.5f // 50% 투명도 설정
10            }
11        )
12 }
```

alpha = 0.5f는 이미지를 **50% 투명도**로 설정합니다.

합성 전략 (Compositing Strategies)

graphicsLayer는 콘텐츠가 렌더링되는 방식을 결정하는 다양한 합성 전략을 제공합니다.

1. **Auto (기본값)**: 속성에 따라 렌더링을 자동으로 최적화합니다.
2. **Offscreen**: 합성하기 전에 콘텐츠를 오프스크린 텍스처로 렌더링합니다.
3. **ModulateAlpha**: 전체 레이어 대신 드로잉 작업별로 alpha를 적용합니다.

다음은 블렌딩 효과를 구현하는 오프스크린 렌더링 예시입니다.

그림 348. GraphicsLayerOffscreenExample.kt

```
1 @Composable
2 fun OffscreenBlendEffect() {
3     Image(
4         painter = painterResource(id = R.drawable.skydoves_image),
5         contentDescription = "Blended Image",
6         modifier = Modifier
7             .size(200.dp)
8             .graphicsLayer {
9                 compositingStrategy = CompositingStrategy.Offscreen // 오프스크린 전략
10                → 사용
11                // 여기에 BlendMode 등 적용 가능
12            }
13    )
```

BlendMode와 같은 효과가 다른 컴포넌트에는 영향을 주지 않고, 해당 컴포저블에만 적용되도록 보장합니다.

컴포저블로부터 Bitmap 획득하기

Compose 1.7.0부터 graphicsLayer를 사용하여 컴포저블을 비트맵으로 추출할 수 있습니다. 안드로이드의 전통적인 Bitmap은 구조가 준수하게 잡혀있는 편이기 때문에, 여전히 Composable 함수로부터 Bitmap을 획득하여 여러 형태로 사용해볼 수 있습니다. 가령, 가우시안 블러(gaussian blur) 효과를 구현하는 등 여전히 이미지에 산술적인 연산이 필요한 경우에 유의미할 수 있습니다. 전통적인 XML에서는 View 객체로부터 Bitmap을 획득할 수 있는 방법이 존재했던 반면에, Jetpack Compose는 최근 들어서야 graphicsLayer API를 통해 구현할 수 있게 되었습니다.

그림 349. GraphicsLayerToBitmapExample.kt

```
1 val coroutineScope = rememberCoroutineScope()
2 // rememberGraphicsLayer()를 사용하여 GraphicsLayerController 생성 및 기억
3 val graphicsLayer = rememberGraphicsLayer()
4
5 Box(
6     modifier = Modifier
7         // drawWithContent를 사용하여 그리기 제어
8         .drawWithContent {
9             // graphicsLayer에 그리기 내용 기록
10            graphicsLayer.record {
11                // 원래 콘텐츠 그리기
12                this@drawWithContent.drawContent()
13            }
14            // 기록된 레이어 그리기
15            drawLayer(graphicsLayer)
16        }
17        .clickable {
18            coroutineScope.launch {
19                // 기록된 GraphicsLayer에서 ImageBitmap 생성 시도
20                val bitmap = try {
21                    graphicsLayer.toImageBitmap()
22                } catch (e: Exception) {
23                    // 오류 처리 (예: 캡처 실패)
24                    Log.e("GraphicsLayer", "Failed to capture bitmap", e)
25                    null
26                }
27                // 비트맵 저장 또는 공유
28                bitmap?.let { /* 비트맵 처리 로직 */ }
29            }
30        }
31        .background(Color.White)
32    ) {
33        Text("Hello Compose", fontSize = 26.sp)
```

이와 같은 방식을 통해 전체 UI를 다시 그리지 않고 컴포저블을 Bitmap 객체로 추출합니다.

요약

graphicsLayer는 컴포저블에 변환, 클리핑, 투명도 및 합성을 적용하는 효율적인 방법을 제공합니다. 크기 조절, 회전, 이동 또는 심화적인 렌더링을 위한 것이든 graphicsLayer는 UI 유연성과 성능을 향상시켜 Compose 애플리케이션에서 커스텀 시각 효과를 위한 중요한 API가 되었습니다.

실전 질문

Q) 70%의 투명도와 원본 사이즈보다 1.2배 크기의 원 모양으로 클리핑 된 프로필 이미지 컴포넌트를 구현해야 한다면, 어떻게 구현하실 건가요?

Q) graphicsLayer API의 사용 목적은 무엇이며, scale, rotate, alpha와 같은 다른 Modifier 대신 이것을 사용하는 이유가 무엇인가요? 또한 graphicsLayer는 렌더링 성능과 컴포저블 격리에 어떤 영향을 미치나요?

Q) 36. Jetpack Compose에서 애니메이션을 어떻게 구현하나요?

Jetpack Compose는 UI 상태 간의 부드럽고 시각적인 효과와 함께 전환을 가능하게 하는 선언적 애니메이션 시스템을 제공합니다. Compose에서 기본적으로 제공하는 애니메이션 API는 개발자가 컴포저블 가시성, 콘텐츠 변경, 크기 조정 및 속성 전환 등에 쉽게 애니메이션을 구현할 수 있도록 합니다. 가장 주목할 만한 부분은 전통적인 XML에서의 애니메이션보다 월등히 구현하기 쉽고, 잘 활용한다면 최적의 성능을 유지하면서 사용자 경험을 향상시킬 수 있습니다.

AnimatedVisibility로 가시성 애니메이션 구현하기

AnimatedVisibility를 사용하면 컴포저블이 나타나고 사라지는 페이드 인/아웃 전환 애니메이션을 구현할 수 있습니다. 컨텐츠가 나타날 때는 페이드 인에 추가적인 효과를 적용할 수 있고 사라질 때는 페이드 아웃에 축소 효과 등을 적용할 수 있습니다. EnterTransition 및 ExitTransition을 사용하여 커스텀 전환을 구현할 수도 있습니다.

그림 350. AnimatedVisibilityExample.kt

```
1 @Composable
2 fun AnimatedVisibilityExample() {
3     var isVisible by remember { mutableStateOf(true) }
4
5     Column {
6         Button(onClick = { isVisible = !isVisible }) {
7             Text("Toggle Visibility")
8         }
9
10        AnimatedVisibility(
11            visible = isVisible,
12            // 진입 애니메이션: 페이드 인 + 수직 확장
13            enter = fadeIn(animationSpec = tween(300)) + expandVertically(),
14            // 퇴장 애니메이션: 페이드 아웃 + 수직 축소
15            exit = fadeOut(animationSpec = tween(300)) + shrinkVertically()
16        ) {
17            // 애니메이션될 컨텐츠
18            Box(
19                modifier = Modifier
20                    .size(100.dp)
21                    .background(Color.Blue)
22            )
23        }
24    }
25 }
```

- Box는 나타날 때 페이드 인 + 수직으로 점점 커지는 확장 애니메이션이 동작하고, 사라질 때 천천히 페이드 아웃 및 축소 애니메이션이 동작합니다.
- EnterTransition 및 ExitTransition을 사용하여 애니메이션 효과를 커스텀할 수 있습니다.

Crossfade를 사용하여 부드러운 전환 만들기

Crossfade는 페이드 효과를 사용하여 두 컴포저블 간의 전환을 애니메이션화하여 컴포넌트 전환 또는 상태 변경을 자연스럽게 합니다.

그림 351. CrossfadeExample.kt

```
1 @Composable
2 fun CrossfadeExample() {
3     var selectedScreen by remember { mutableStateOf("Home") }
4
5     Column {
6         Row {
7             Button(onClick = { selectedScreen = "Home" }) { Text("Home") }
8             Button(onClick = { selectedScreen = "Profile" }) { Text("Profile") }
9         }
10
11         // targetState가 변경될 때 콘텐츠가 크로스페이드됨
12         Crossfade(targetState = selectedScreen, label = "Screen Crossfade") { screen ->
13             when (screen) {
14                 "Home" -> Text("Home Screen", fontSize = 24.sp)
15                 "Profile" -> Text("Profile Screen", fontSize = 24.sp)
16                 else -> Text("Undefined Screen", fontSize = 24.sp)
17             }
18         }
19     }
20 }
```

- Home과 Profile 간 컴포저블 전환 시 콘텐츠가 부드럽게 크로스페이드됩니다.
- 탭 내비게이션 및 화면/이미지/컴포넌트 전환 등에 유용합니다.

AnimatedContent를 사용하여 콘텐츠 전환에 애니메이션 구현하기

AnimatedContent는 레이아웃을 유지하면서 다른 상태 변경으로 인한 컨텐츠 간 전환을 부드럽게 합니다.

그림 352. AnimatedContentExample.kt

```
1 @Composable
2 fun AnimatedContentExample() {
3     var count by remember { mutableStateOf(0) }
4
5     Row {
6         Button(onClick = { count++ }) { Text("Increment") }
7
8         Spacer(Modifier.width(16.dp))
9
10        // targetState가 변경될 때 콘텐츠 애니메이션
11        AnimatedContent(
12            targetState = count,
13            label = "Count Animation",
14            // 전환 애니메이션 정의 (선택 사항)
15            transitionSpec = {
16                // 숫자가 증가하면 위로 슬라이드, 감소하면 아래로 슬라이드
17                if (targetState > initialState) {
18                    slideInVertically { height -> height } + fadeIn() togetherWith
19                        slideOutVertically { height -> -height } + fadeOut()
20                } else {
21                    slideInVertically { height -> -height } + fadeIn() togetherWith
22                        slideOutVertically { height -> height } + fadeOut()
23                }.using(
24                    // 크기 변경 애니메이션 비활성화
25                    SizeTransform(clip = false)
26                )
27            }
28        ) { targetCount ->
29            // 애니메이션될 콘텐츠
30            Text(text = "Count: $targetCount", fontSize = 24.sp)
31        }
32    }
```

- count가 업데이트될 때마다 Text가 전환 애니메이션을 자연스럽게 수행합니다.
- 애니메이션은 다른 애니메이션 스펙(animation specs)으로 커스텀할 수 있습니다.

animate*AsState를 사용하여 세밀한 애니메이션 구현하기

animate*AsState 함수는 단일 값을 부드럽게 애니메이션화하도록 설계된 가장 간단한 애니메이션 API입니다. 대상 값만 지정하면 API가 현재 상태에서 목표 상태로의 전환을 부드러운 애니메이션으로 처리할 수 있습니다. animate*AsState 함수는 종류가 다양한데, 대표적으로 animateDpAsState, animateFloatAsState, animateIntAsState, animateOffsetAsState, animateSizeAsState, animateValueAsState를 포함한 다양한 값 기반의 애니메이션을 지원하여 유연하고 훨씬 세밀한 값에 기반한 애니메이션을 구현할 때 유리합니다.

아래 예제는 컴포넌트 크기를 동적으로 조정하는 animateDpAsState를 보여줍니다.

그림 353. AnimateAsStateExample.kt

```
1 @Composable
2 fun AnimateAsStateExample() {
3     var isExpanded by remember { mutableStateOf(false) }
4
5     // isExpanded 상태에 따라 Dp 값 애니메이션
6     val boxSize by animateDpAsState(
7         targetValue = if (isExpanded) 200.dp else 100.dp,
8         // 애니메이션 스펙 설정 (500ms 동안 tween)
9         animationSpec = tween(durationMillis = 500), label = "Box Size Animation"
10    )
11
12    Column {
13        Button(onClick = { isExpanded = !isExpanded }) {
14            Text("Toggle Size")
15        }
16    }
```

```
17     Box(  
18         modifier = Modifier  
19             .size(boxSize) // 애니메이션된 Dp 값 사용  
20             .background(Color.Green)  
21     )  
22 }  
23 }
```

- `animateDpAsState`는 500밀리 초 동안 상태에 따라 Dp 값을 증가시키거나 감소시킵니다.
- 위의 구현은 버튼 토글 시 부드러운 크기 변환 애니메이션을 보여줍니다.

animateContentSize로 크기 변경 처리하기

`animateContentSize`는 수동적으로 애니메이션 콜백 등을 구현할 필요 없이 레이아웃 크기 변경을 자동으로 감지화 하고 해당 값을 알아서 추적하여 애니메이션화합니다. 사용방법이 굉장히 간단하고, 사이즈 관련 애니메이션을 빠르게 추가하고 싶은 경우 매우 유용합니다.

그림 354. `AnimateContentSizeExample.kt`

```
1 @Composable  
2 fun AnimateContentSizeExample() {  
3     var expanded by remember { mutableStateOf(false) }  
4  
5     Column {  
6         Button(  
7             onClick = { expanded = !expanded }  
8         ) {  
9             Text(if (expanded) "Collapse" else "Expand")  
10        }  
11    }  
12  
13    Box(  
14        modifier = Modifier  
15            .background(Color.Red)  
16            .animateContentSize() // 콘텐츠 크기 변경 시 자동으로 애니메이션 적용  
17    }
```

```
16           .padding(16.dp)
17     ) {
18         // expanded 상태에 따라 텍스트 변경
19         Text(
20             text = if (expanded) {
21                 "Lorem ipsum dolor sit amet, consectetur adipiscing elit. " +
22                 "Sed do eiusmod tempor incididunt ut labore et dolore magna aliqua."
23             } else {
24                 "Short Text"
25             },
26             fontSize = 18.sp
27         )
28     }
29 }
30 }
```

- 콘텐츠가 변경될 때 Box의 크기가 부드럽게 늘어나거나 줄어듭니다.
- 추가적인 애니메이션 로직이 필요하지 않고, 크기 변동을 알아서 감지하여 애니메이션이 자동으로 구현됩니다.

요약

Jetpack Compose는 사용이 쉽고 원활한 애니메이션 구현을 위한 API를 기본적으로 제공하여, 전통적인 XML 기반 애니메이션에 비해 훨씬 쉽게 구현이 가능합니다. 보편적으로 사용하는 API를 요약하자면 다음과 같습니다.

- **AnimatedVisibility**: 컴포저블을 자연스럽게 나타나게 하거나 사라지게 합니다.
- **Crossfade**: UI 상태 간 부드러운 전환을 제공합니다.
- **AnimatedContent**: 콘텐츠 업데이트를 동적으로 애니메이션화합니다.
- **animate*AsState**: 값에 기반한 애니메이션을 실행하여 크기, 불투명도, 위치 등 세밀한 값을 조정할 수 있게 합니다.
- **animateContentSize**: 컴포저블의 크기를 추적하여 별도의 구현 없이 자동적으로 크기 변화 애니메이션을 구현할 수 있도록 합니다.

실전 질문

Q) 콘텐츠에 따라 크기가 동적으로 변하는 텍스트 블록에 크기 관련 애니메이션 효과를 어떻게 구현하실 건가요?

Q) Canvas, Painter 및 애니메이션을 사용하여 로딩 중인 컴포넌트에 대해 쉬머(shimmer) 애니메이션 효과를 주고 싶은 경우 어떻게 구현하시겠습니까?

Q) 37. 화면 간 내비게이션을 어떻게 구현하나요?

최근 best practice라고 일컫는 싱글 액티비티 구조가 선호됨에 따라 XML 기반 프로젝트에서도 내비게이션 시스템은 안드로이드 개발에 있어 필수적인 부분으로 자리하고 있습니다. Jetpack Compose에서는 Fragment 개념이 존재하지 않기 때문에, 내비게이션 스택, 컨트롤러 및 UI 상태를 관리하기 위한 전용 내비게이션 시스템이 필요합니다. Compose 프로젝트에서 내비게이션을 구현하는 두 가지 기본 접근 방식은 수동적으로 내비게이션 시스템을 구현하는 방법과 스택 처리 및 상태 보존을 내부적으로 모두 처리해 주는 Jetpack Compose Navigation 라이브러리를 사용하는 방법이 있습니다.

내비게이션 직접 구현하기 (Manual Navigation)

컴포저블의 상태를 관리하고 보존하기 위해 고유 키를 기반으로 rememberSaveable과 함께 작동하는 Compose Runtime API인 SaveableStateHolder를 사용하여 수동으로 내비게이션 시스템을 구현할 수 있습니다. 컴포저블이 컴포지션에서 제거될 때(가령, 새 화면으로 이동할 때) 해당 상태는 자동으로 저장되고 컴포지션에 다시 진입할 때 복원됩니다.

아래 예제는 내비게이션을 직접 구현하는 과정에서 상태 관리를 위해 SaveableStateHolder를 사용하는 방법을 보여줍니다. 각 화면은 앞뒤로 상태를 독립적으로 유지하지만, 다른 화면으로 전환 후 돌아오면 기존 화면의 상태가 복원됩니다.

그림 355. SaveableStateHolderNavigation.kt

```
1 @Composable
2 fun <T : Any> Navigation(
3     currentScreen: T,
4     modifier: Modifier = Modifier,
5     content: @Composable (T) -> Unit
6 ) {
7     // SaveableStateHolder 생성 및 기억
8     val saveableStateHolder = rememberSaveableStateHolder()
9
10    Box(modifier) {
11        // AnimatedContent를 사용하여 화면 전환 시 애니메이션 효과 적용
12        AnimatedContent(targetState = currentScreen, label = "NavigationContent") {
13            targetScreen ->
14                // SaveableStateProvider로 현재 화면 콘텐츠 래핑 (key로 화면 상태 구분)
15                saveableStateHolder.SaveableStateProvider(key = targetScreen) {
16                    content(targetScreen) // 실제 화면 콘텐츠 렌더링
17                }
18        }
19    }
20
21 // 메인 컴포저블 (화면 전환 버튼 및 Navigation 호출)
22 @Composable
23 fun SaveableStateHolderExample() {
24     var screen by rememberSaveable { mutableStateOf("screen1") }
25
26     Column {
27         // 화면 전환 버튼
28         Row(
29             modifier = Modifier.fillMaxWidth(),
30             horizontalArrangement = Arrangement.SpaceEvenly
31         ) {
32             Button(onClick = { screen = "screen1" }) { Text("Go to screen1") }
```

```
33             Button(onClick = { screen = "screen2" }) { Text("Go to screen2") }
34         }
35
36     // Navigation 컴포저블을 사용하여 화면 표시
37     Navigation(currentScreen = screen, modifier = Modifier.fillMaxSize()) {
38         currentScreen ->
39         when (currentScreen) {
40             "screen1" -> Screen1()
41             "screen2" -> Screen2()
42         }
43     }
44 }
45
46 // 화면 1 컴포저블
47 @Composable
48 fun Screen1() {
49     var counter by rememberSaveable { mutableStateOf(0) }
50     Column(
51         modifier = Modifier.padding(16.dp),
52         horizontalAlignment = Alignment.CenterHorizontally
53     ) {
54         Text("Screen 1")
55         Spacer(Modifier.height(16.dp))
56         Button(onClick = { counter++ }) {
57             Text("Counter: $counter")
58         }
59     }
60 }
61
62 // 화면 2 컴포저블
63 @Composable
64 fun Screen2() {
65     var text by rememberSaveable { mutableStateOf("") }
```

```
66     Column(  
67         modifier = Modifier.padding(16.dp),  
68         horizontalAlignment = Alignment.CenterHorizontally  
69     ) {  
70         Text("Screen 2")  
71         Spacer(Modifier.height(16.dp))  
72         TextField(  
73             value = text,  
74             onValueChange = { text = it },  
75             label = { Text("Enter text") }  
76         )  
77     }  
78 }
```

위에서 살펴본 예시와 같이 `SaveableStateHolder`와 `rememberSaveable`을 사용하여 내비게이션 시스템을 직접 구현할 수 있습니다. 그러나 실제 현실에서 사용하기 위한 내비게이션 시스템은 훨씬 더 복잡합니다. 이전 화면으로 결과 값을 전달하거나, 특정 화면으로 전환하는 시나리오, 혹은 딥 링킹 기능과 같은 것을 고려해 보면 직접 새롭게 구현한다는 것이 결코 만만한 작업은 아닙니다. 이러한 시스템적인 기능을 효율적으로 처리하기 위한 전용 라이브러리나 솔루션을 사용하고 다른 중요한 것에 집중하는 것이 리소스를 절약할 수 있는 방법 중 하나입니다.

또한 Jetpack ViewModel을 사용하는 경우 해당 생명주기는 내비게이션 상태를 기반으로 관리해야 합니다. 즉, `ViewModel`이 의존성 주입 프레임워크와 통합되면 수동으로 관리하기가 더욱 어려워질 수 있습니다. 특히 복잡한 화면 구조를 이루는 유즈 케이스에서 더 정교하고 확장 가능한 내비게이션을 위해서는 Jetpack Compose의 내비게이션 라이브러리를 사용하는 것이 가장 이상적입니다.

Jetpack Compose Navigation

Jetpack Compose는 Jetpack 내비게이션 시스템의 기존 인프라를 활용하면서 컴포저블 간의 원활한 내비게이션을 가능하게 하는 [Compose용 내비게이션 라이브러리](#)⁸¹를 제공합니다. 구조화된 내비게이션 관리, 상태 처리 및 딥 링킹 기능, 안전한 인수(safe arguments) 전달

⁸¹<https://developer.android.com/develop/ui/compose/navigation>

기능 등을 제공하여 Compose에서 내비게이션 시스템이 필요한 경우 해당 라이브러리를 통해 대부분의 문제는 해결할 수 있습니다.

Compose 내비게이션 시스템은 세 가지 주요 컴포넌트로 이루어져 있습니다.

- **NavHost**: 내비게이션 그래프를 정의하고 컴포저블 화면을 라우트(routes)와 연결합니다.
- **NavController**: 내비게이션 스택을 관리하고 목적지(destinations) 간의 내비게이션을 처리합니다.
- **Composable Destinations**: 내비게이션 그래프 내의 개별 화면입니다.

이제 Jetpack Compose Navigation 라이브러리를 사용하여 내비게이션 시스템을 구현하는 방법을 살펴보겠습니다.

1. NavHost로 내비게이션 정의하기

NavHost는 내비게이션 그래프를 관리하는 역할을 합니다. 시작 대상과 다른 화면 간의 가능한 라우트를 모두 정의합니다.

그림 356. AppNavHost.kt

```
1 import androidx.navigation.compose.NavHost
2 import androidx.navigation.compose.composable
3 import androidx.navigation.compose.rememberNavController
4 import kotlinx.serialization.Serializable // Kotlinx Serialization 사용 예시
5
6 // 화면 경로 정의 (Type-safe Navigation 사용 권장)
7 @Serializable sealed interface PokedexScreen {
8     @Serializable data object Home: PokedexScreen
9     @Serializable data object Details: PokedexScreen // 예시: data object 사용
10 }
11
12 @Composable
13 fun AppNavHost(
14     modifier: Modifier = Modifier,
```

```
15     navController: NavHostController = rememberNavController(), // NavController 생성
    ↵ 및 기억
16 ) {
17     NavHost(
18         modifier = modifier,
19         navController = navController,
20         startDestination = PokedexScreen.Home // 시작 화면 지정
21 ) {
22     // Home 화면 정의
23     composable<PokedexScreen.Home> { // 타입-세이프 composable 사용
24         HomeScreen(
25             // Details 화면으로 이동하는 콜백 전달
26             onNavigateToDetails = { navController.navigate(PokedexScreen.Details) }
27         )
28     }
29     // Details 화면 정의
30     composable<PokedexScreen.Details> {
31         DetailsScreen(
32             // 뒤로 가기 콜백 전달 (예시)
33             // onBack = { navController.popBackStack() }
34         )
35     }
36 }
37 }
```

위의 코드는 다음과 같이 동작합니다.

- NavHost는 모든 화면을 정의하고, Home을 시작 화면으로 설정하여 내비게이션 그래프를 생성합니다.
- composable<Home> 및 composable<Details>는 내비게이션 그래프 내의 각 화면을 나타내며 이들 간의 내비게이션을 가능하게 합니다.
- NavController는 특정 화면으로의 내비게이션을 가능하게 하고, 내비게이션 간에 매개 변수를 전달할 수 있도록 하며, 백 스택 및 현재 대상을 관리하며, NavHost 내의 전체 내비게이션 동작을 제어합니다.

2. 화면 간 내비게이션하기

HomeScreen에는 제공된 내비게이션 콜백을 사용하여 DetailsScreen으로의 내비게이션을 트리거하는 버튼이 포함되어 있습니다.

그림 357. HomeScreen.kt

```

1 @Composable
2 fun HomeScreen(
3     onNavigateToDetails: () -> Unit // Details 화면으로 이동하는 콜백
4 ) {
5     Column {
6         Button(onClick = onNavigateToDetails) { // 버튼 클릭 시 콜백 실행
7             Text(text = "Navigate to Details")
8         }
9     }
10 }
```

- onNavigateToDetails 람다는 HomeScreen에 매개변수로 전달되어 내비게이션 로직을 분리합니다.
- 버튼을 클릭하면 navController.navigate(PokedexScreen.Details) 코드를 통해 사용자를 DetailsScreen으로 이동시킵니다.

Jetpack Compose Navigation 라이브러리는 전환 애니메이션, 딥 링크⁸², 타입-세이프티⁸³, 중첩 내비게이션⁸⁴, 테스트 전략⁸⁵, Hilt 통합 기능⁸⁶ 지원을 포함한 광범위한 기능을 제공합니다. 또한 내비게이션 시스템 내에서 ViewModel 생명주기를 효율적으로 관리하기 위한 내부 시스템 전용 viewModelStore를 제공합니다.

JetBrains는 Kotlin Multiplatform (KMP) 사용자에게 KMP 기반 내비게이션⁸⁷을 지원하기

⁸²<https://developer.android.com/develop/ui/compose/navigation#deeplinks>

⁸³<https://developer.android.com/guide/navigation/design/type-safety>

⁸⁴<https://developer.android.com/develop/ui/compose/navigation#nested-nav>

⁸⁵<https://developer.android.com/develop/ui/compose/navigation#testing>

⁸⁶<https://developer.android.com/training/dependency-injection/hilt-jetpack#viewmodel-navigation>

⁸⁷<https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-navigation-routing.html>

위해 Jetpack Compose Navigation 라이브러리를 포크 하여 멀티 플랫폼에서 사용할 수 있는 솔루션을 공개하기도 하였습니다.

요약

내비게이션은 현대 안드로이드 앱 개발에서 필수적인 기능이 되었습니다. 니즈에 따라서 수동으로 구현할 수도 있지만, 딥 링크 지원, 타입-세이프티, Hilt 통합 및 원활한 ViewModel 생명주기 관리와 같이 심화적인 기능을 위해서는 Jetpack Compose Navigation 라이브러리를 사용하는 편이 좋습니다. Google의 공식 프로젝트인 [Now in Android](#)⁸⁸를 통하여 Jetpack Compose Navigation의 실전 사용 사례를 보여주는 예제를 탐색해 보실 수 있습니다.

실전 질문

Q) Navigation 라이브러리를 사용하지 않고 멀티 화면으로 구성된 Compose 앱에서, 화면 간 내비게이션 기능과 화면 간 이동으로부터 상태를 보존하려면 어떤 방법을 사용할 수 있을까요?

Q) Jetpack Compose Navigation에서 NavHost 및 NavController 시스템은 백 스택 및 ViewModel 생명주기를 어떻게 처리하나요?

Q) 38. Compose Preview는 어떻게 작동하고 사용하고 계신 Preview 어노테이션은 어떤 것이 있나요?

Jetpack Compose의 장점 중 하나는 Android Studio의 [Preview](#)⁸⁹ 기능입니다. 이 기능을 통해 개발자는 전체 프로젝트를 컴파일하지 않고도 UI 컴포넌트를 점진적으로 빌드하고 시각화할 수 있습니다. 컴포저를 함수를 미리보기 창에 직접 렌더링함으로써 Compose Previews는 개발 워크플로를 간소화하고 UI 변경 사항을 확인하는 데 필요한 시간을 줄이며 생산성을 향상시킵니다.

[Compose UI Tooling preview](#) 라이브러리⁹⁰는 Android Studio에서 미리보기 경험을 향상시키는 여러 어노테이션을 제공하여 컴포저블을 시각화하고 테스트하기 쉽게 만듭니다. 이러한 어노테이션이 어떻게 개발 프로세스를 간소화하고 Compose 미리보기 작업 시 효율성을 올려주는지 살펴보겠습니다.

⁸⁸<https://github.com/android/nowinandroid/blob/main/app/src/main/java/com/google/samples/apps/nowinandroid/ui/NiaApp.kt>

⁸⁹<https://developer.android.com/develop/ui/compose/tooling/previews>

⁹⁰<https://developer.android.com/develop/ui/compose/tooling>

@Preview를 사용하여 Composable 렌더링하기

@Preview 어노테이션은 Jetpack Compose 미리보기 시스템의 가장 근본이 됩니다. 모든 컴포저블 함수에 적용할 수 있으며 다중 어노테이션을 허용하므로 아래 예제와 같이 동일한 함수에 여러 @Preview 어노테이션을 추가하여 다양한 설정 값 또는 디바이스에서 미리보기를 활성화할 수 있습니다.

그림 358. SimplePreview.kt

```
1 @Preview(name = "light mode")
2 @Preview(name = "dark mode", uiMode = Configuration.UI_MODE_NIGHT_YES)
3 @Composable
4 private fun MyPreview() {
5     MaterialTheme {
6         Text(
7             text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit.",
8             color = if (isSystemInDarkTheme()) {
9                 Color.White
10            } else {
11                Color.Yellow // 라이트 모드에서는 노란색
12            }
13        )
14    }
15 }
```

- @Preview 어노테이션은 MyPreview 컴포저블을 Android Studio의 **미리보기** 창에 렌더링하도록 지시합니다.
- 다양한 UI 상태를 시각화하기 위해 하나의 컴포저블 함수에 대해서 여러 미리보기를 만들 수 있습니다.

Android Studio에서 아래 결과를 볼 수 있습니다.



그림 359. compose-preview

미리보기 커스텀하기

Jetpack Compose는 다양한 매개변수를 사용하여 `@Preview`를 커스텀할 수 있도록 합니다.

그림 360. CustomPreview.kt

```
1 @Preview(  
2     name = "Dark Mode Pixel 4 XL", // 미리보기 이름  
3     showBackground = true, // 배경 표시 여부  
4     backgroundColor = 0xFF212121, // 배경색 지정 (ARGB)  
5     uiMode = Configuration.UI_MODE_NIGHT_YES, // UI 모드 (다크 모드)  
6     device = Devices.PIXEL_4_XL, // 시뮬레이션할 기기  
7     // locale = "ko", // 로케일 설정 (선택 사항)  
8     // fontScale = 1.5f // 글꼴 크기 배율 설정 (선택 사항)  
9 )  
10 @Composable  
11 fun DarkModePreview() {  
12     // Greeting 컴포저블 (내용은 예시)
```

```
13     MaterialTheme { // 테마 적용 권장
14         Greeting(name = "skydoves")
15     }
16 }
17
18 @Composable
19 fun Greeting(name: String) {
20     Text("Hello $name!")
21 }
```

@Preview 어노테이션에서 대표적으로 제공하는 커스텀 매개변수는 다음과 같습니다.

- name: 미리보기에 이름을 할당합니다. 하나의 컴포저블에 대해서 여러 미리보기를 생성하는 경우 헷갈릴 수 있는데 혼선을 방지하기 위해 유용합니다.
- showBackground: 컴포저를 뒤에 배경을 표시합니다. 배경이 없으면 기본값으로 투명색 상이 적용되는데, 컴포넌트 색상이 Android Studio의 IDE 배경색이랑 같으면 잘 안 보일 수 있기 때문에 일반적으로 배경을 표시하는 편이 좋습니다.
- backgroundColor: 커스텀 배경색을 정의합니다.
- uiMode: **다크 모드**와 같은 다양한 시스템 모드를 적용합니다.
- device = Devices.PIXEL_4_XL: **Pixel 4 XL** 디바이스의 해상도를 가진 미리보기를 적용 합니다.

@PreviewParameter를 사용한 미리보기

@PreviewParameter 어노테이션은 PreviewParameterProvider를 활용하여 컴포저블 함수에 미리보기 인스턴스 의존성을 주입할 수 있도록 합니다. **PreviewParameterProvider**⁹¹ 인터페이스를 구현하는 커스텀 클래스를 정의하고 이를 사용하여 컴포저블에 데이터 정보를 제공할 수 있습니다. 이 방법을 통해 아래 예제와 같이 다양한 커스텀 데이터를 **동적 미리보기 (dynamic previews)** 를 통해 보여지도록 할 수 있습니다.

⁹¹<https://developer.android.com/reference/kotlin/androidx/compose/ui/tooling/preview/PreviewParameterProvider>

그림 361. ParameterizedPreview.kt

```
1 // 미리보기용 데이터 클래스
2 public data class User(
3     val name: String,
4 )
5
6 // User 데이터를 제공하는 PreviewParameterProvider 구현
7 public class UserPreviewParameterProvider: PreviewParameterProvider<User> {
8     override val values: Sequence<User>
9         get() = sequenceOf(
10             User("user1"),
11             User("user2"),
12             User("user3") // 필요에 따라 더 많은 데이터 추가 가능
13         )
14 }
15
16 // @PreviewParameter를 사용하여 데이터 주입
17 @Preview(name = "UserPreview")
18 @Composable
19 private fun UserPreview(
20     @PreviewParameter(provider = UserPreviewParameterProvider::class) user: User
21 ) {
22     // MaterialTheme 등으로 감싸는 것이 좋음
23     Surface { // 배경 등 기본 스타일링 적용
24         Text(text = user.name, color = Color.Black) // 예시: 검은색 텍스트
25     }
26 }
```

이는 주입된 데이터 리스트를 기반으로 여러 미리보기를 생성합니다. 여기서 또 하나의 팁은, Jetpack Compose UI tooling 라이브러리에는 아래 미리보기 이미지와 같이 미리 정의된 샘플 텍스트 문자열을 제공하는 [LoremIpsum⁹²](#)이라는 PreviewParameterProvider를 제

⁹²<https://cs.android.com/androidx/platform/frameworks/support/+/androidx-main:compose/ui/ui-tooling-preview/src/androidMain/kotlin/androidx/compose/ui/tooling/preview/datasource/LoremIpsum.android.kt;l=49?q=LoremIpsum%20compose>

공합니다. 따라서 미리보기를 적용하는 컴포저블에 Text가 포함된 경우 해당 값을 사용하여 빠르게 처리할 수 있습니다.

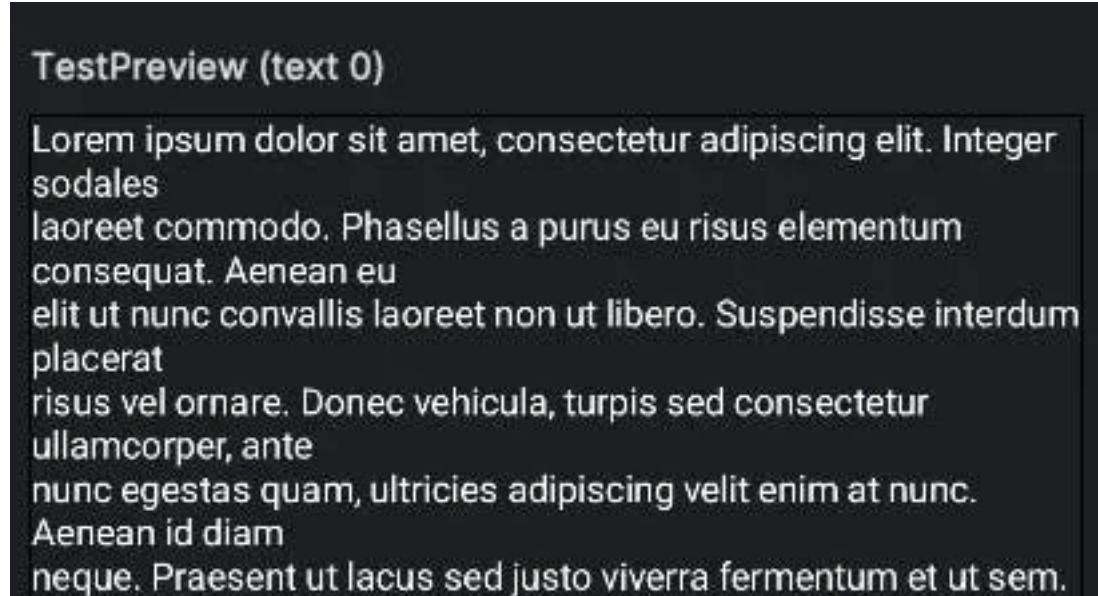


그림 362. compose-preview

아래 코드에서 볼 수 있듯이 일반적인 PreviewParameterProvider처럼 클래스를 사용할 수 있습니다.

그림 363. LoremIpsum.kt

```
1 @Preview  
2 @Composable  
3 private fun TestPreview(@PreviewParameter(provider = LoremIpsum::class, limit = 2)  
    → text: String) { // limit으로 개수 제한 가능  
    Text(text = text, color = Color.Black)  
}
```

미리보기의 상호작용 모드 (Interactive Mode)

Android Studio는 앱을 실행하지 않고도 사용자가 컴포저블과 상호 작용이 가능한 미리보기를 지원합니다.

그림 364. InteractivePreview.kt

```
1 @Preview(showBackground = true) // 배경 표시
2 @Composable
3 fun InteractivePreview() {
4     var count by remember { mutableStateOf(0) }
5
6     Column(horizontalAlignment = Alignment.CenterHorizontally) { // 가운데 정렬 추가
7         Text("Count: $count")
8         Spacer(Modifier.height(8.dp)) // 간격 추가
9         Button(onClick = { count++ }) {
10             Text("Increment")
11         }
12     }
13 }
```

클릭 가능한 컴포넌트, 애니메이션 및 상태 업데이트를 미리보기 창에서 직접 눌러보거나 시각적으로 테스트할 수 있습니다. 아래 그림과 같이 미리보기 창에서 **상호작용 모드**를 활성화할 수 있습니다.

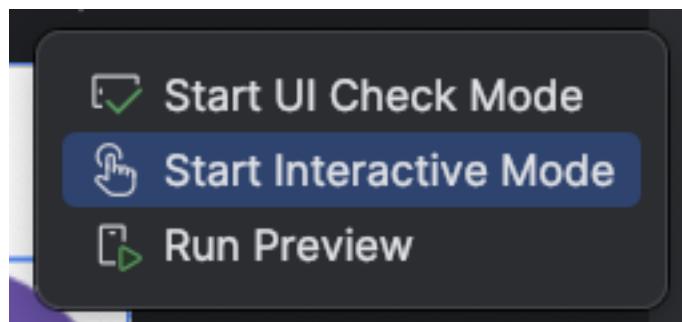


그림 365. compose-preview-interactive-mode

이제 미리보기 창에서 아래 스크린샷과 같이 직접 컴포저블 함수와 상호 작용할 수 있습니다.

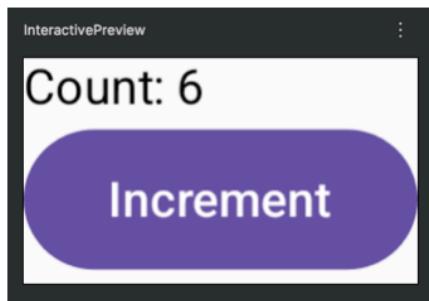


그림 366. compose-preview-interactive-mode

멀티 프리뷰 어노테이션 (MultiPreview Annotations)

앞서 언급했듯이 @Preview 어노테이션은 반복적으로 적용이 가능하므로 동일한 컴포저블 함수에 여러 어노테이션을 적용할 수 있습니다. Jetpack Compose는 @PreviewLightDark, @PreviewFontSize, @PreviewDynamicColors, @PreviewScreenSizes와 같이 다양한 UI 조건을 빠르게 테스트할 수 있도록 **멀티 프리뷰 어노테이션** 프리셋을 제공합니다.

예를 들어, **라이트 및 다크 모드**에서의 미리보기를 모두 확인하려는 경우, 아래 코드와 같이 수동적으로 미리보기 어노테이션을 여러 개 적용할 수도 있습니다.

그림 367. MultiPreviewAnnotations.kt

```
1 @Preview(name = "light mode")
2 @Preview(name = "dark mode", uiMode = Configuration.UI_MODE_NIGHT_YES)
3 @Composable
4 private fun MyPreview() {
5     ..
6 }
```

그러나, 이를 @PreviewLightDark 어노테이션으로 간단히 대체할 수 있습니다.

그림 368. MultiPreviewAnnotations.kt

```
1 @PreviewLightDark
2 @Composable
3 private fun MyPreview() {
4     ..
5 }
```

또한, 아래 스크린샷과 같이 `@PreviewScreenSizes` 및 `@PreviewFontSizeScale`과 같은 프리셋 미리 보기 어노테이션을 사용해서 Android Studio 내에서 컴포저블을 여러 형태로 쉽게 시각화할 수 있습니다. 해당 어노테이션을 사용하면 수동으로 미리보기 관련 어노테이션을 여러 개 사용하지 않고도 다양한 화면 크기 및 글꼴 배율에서 효율적인 테스트가 가능합니다.

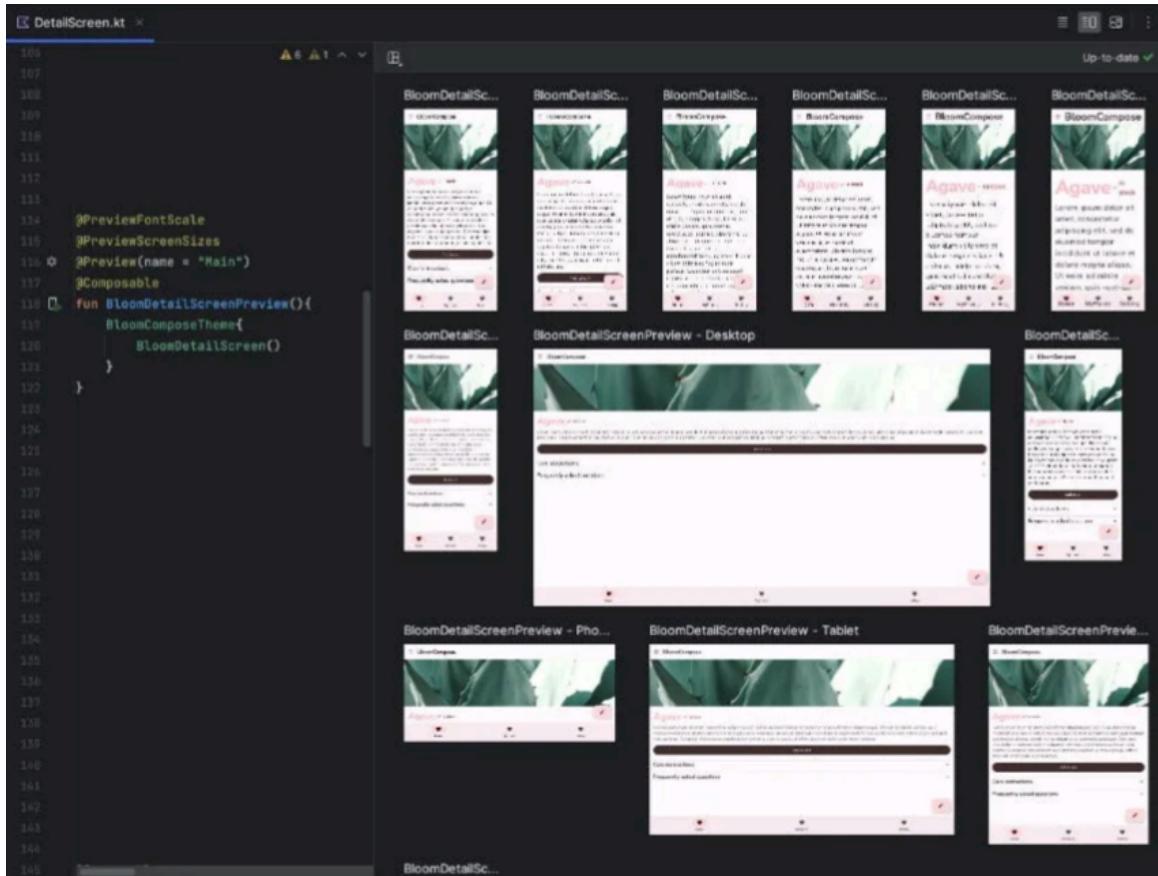


그림 369. compose-preview

요약

Jetpack Compose의 `@Preview`는 Android Studio 내에서 실시간 렌더링, 커스텀 및 상호 작용 모드를 제공하여 UI 개발 생산성을 높여줍니다. 테마 설정, 기기 구성, 다크 모드 미리보기 등 여러 매개변수와 같은 커스텀 옵션 또한 지원하여 전체 프로젝트를 직접 에뮬레이터나 물리적인 디바이스에 컴파일하지 않고도 컴포넌트를 시각화할 수 있습니다.

실전 질문

Q) `@Preview` 어노테이션은 어떻게 개발 워크플로를 개선하며, 이와 함께 사용해 본 주요 어노테이션(다크 테마, 화면 크기, 멀티 프리뷰 어노테이션 등)은 어떤 것이 있나요?

Q) 39. Compose UI 컴포넌트 또는 스크린 유닛 테스트를 작성해 본 적이 있나요? 어떤 시나리오에서 작성해 봤나요?

Jetpack Compose UI 컴포넌트 테스트는 UI 정확성, 안정성 및 사용성을 확인합니다. Compose는 개발자가 UI 테스트를 효율적으로 작성할 수 있도록 전용 테스트 라이브러리⁹³를 제공합니다. 해당 라이브러리는 Jetpack의 **ComposeTestRule**을 기반으로 구축되어 UI 상호 작용, 동기화 및 동작 검증을 위한 API를 제공합니다.

Compose UI 테스트 설정하기

Compose 테스트는 **AndroidJUnit4**를 사용하여 작성할 수 있으며, 테스트 환경을 구성하는 하는 **ComposeTestRule**이라는 Compose 전용 테스트 규칙이 필요합니다. 해당 규칙을 통해 UI와의 상호 작용이 가능하고, 테스트 작업과 recomposition 간의 동기화가 보장됩니다.

그림 370. verifyTextDisplayed.kt

```
1 import androidx.compose.ui.test.junit4.createComposeRule
2 import androidx.compose.ui.test.*
3 import androidx.test.ext.junit.runners.AndroidJUnit4
4 import org.junit.Rule
5 import org.junit.Test
6 import org.junit.runner.RunWith
7
8 @RunWith(AndroidJUnit4::class)
9 class MyComposeTest {
10
11     @get:Rule
12     val composeTestRule = createComposeRule() // Compose 테스트 규칙 생성
13
14     @Test
15     fun verifyTextDisplayed() {
16         // 테스트할 UI 설정
17         composeTestRule.setContent {
18             Text("Hello, skydoves!")
```

⁹³<https://developer.android.com/develop/ui/compose/testing>

```
19         }
20
21     // "Hello, skydoves!" 텍스트를 가진 노드 찾기
22     composeTestRule
23         .onNodeWithText("Hello, skydoves!")
24         .assertExists() // 노드가 존재하는지 검증
25     }
26 }
```

위의 예제에서 `setContent`는 테스트를 위한 UI를 초기화하고(`composition` 생성), `onNodeWithText`는 일치하는 텍스트를 가진 컴포저블을 찾아 실제 존재하는지 검증합니다.

UI 상호 작용 테스트하기

Compose는 버튼 클릭, 텍스트 입력, 목록 스크롤과 같은 사용자 상호 작용을 시뮬레이션하는 API를 제공합니다.

그림 371. `clickButtonUpdatesText.kt`

```
1 @Test
2 fun clickButtonUpdatesText() {
3     composeTestRule.setContent {
4         var text by remember { mutableStateOf("Hello, skydoves!") }
5         Column {
6             Text(text) // 현재 텍스트 표시
7             Button(onClick = { text = "Hello, Kotlin!" }) { // 클릭 시 텍스트 변경
8                 Text("Click me")
9             }
10        }
11    }
12
13    // "Click me" 버튼 찾아서 클릭 수행
14    composeTestRule.onNodeWithText("Click me").performClick()
15    // "Hello, Kotlin!" 텍스트가 존재하는지 확인
```

```
16     composeTestRule.onNodeWithText("Hello, Kotlin!").assertExists()  
17 }
```

여기서 `performClick()`은 버튼 클릭을 시뮬레이션하여 텍스트를 업데이트하고, 해당 동작이 정상적으로 반영되었는지 `assertExists()` API를 통해 검증합니다.

동기화 및 유휴 리소스 (Synchronization and Idling Resources)

Jetpack Compose UI 테스트는 단일 스레드에서 실행되므로 Compose는 **유휴 리소스(Idling Resources)**를 통해 테스트 동기화를 보장할 수 있습니다. 하지만 종종 코루틴이나 애니메이션을 테스트할 때 명시적인 동기화가 필요할 수 있습니다.

그림 372. testLoadingState.kt

```
1 @Test  
2 fun testLoadingState() {  
3     composeTestRule.setContent {  
4         var isLoading by remember { mutableStateOf(true) }  
5         // LaunchedEffect를 사용하여 딜레이 후 로딩 상태 변경  
6         LaunchedEffect(Unit) {  
7             delay(2000) // 2초 지연  
8             isLoading = false  
9         }  
10  
11         if (isLoading) {  
12             CircularProgressIndicator(modifier = Modifier.testTag("loadingIndicator")) //  
13             → 테스트 태그 추가  
14         } else {  
15             Text("Loaded")  
16         }  
17  
18         // 로딩 인디케이터가 사라질 때까지 대기 (예시)  
19         // composeTestRule.waitUntilDoesNotExist(hasTestTag("loadingIndicator"), 3000)  
20 }
```

```

21    // "Loaded" 텍스트가 정확히 하나 존재할 때까지 최대 3초 대기
22    composeTestRule.waitUntilExactlyOneExists(hasText("Loaded"), 3000)
23
24    // 로딩 완료 후 "Loaded" 텍스트가 표시되는지 확인
25    composeTestRule.onNodeWithText("Loaded").assertIsDisplayed()
26 }

```

여기서 `waitFor()`은 텍스트 업데이트를 확인하기 전에 로딩 상태가 완료될 때까지 기다립니다. 아래와 같이 기본적으로 `waitFor()` 관련 API들을 제공합니다.

```

1 // 매처와 일치하는 노드가 하나 이상 존재할 때까지 대기
2 composeTestRule.waitForAtLeastOneExists(matcher, timeoutMs)
3
4 // 매처와 일치하는 노드가 존재하지 않을 때까지 대기
5 composeTestRule.waitForDoesNotExist(matcher, timeoutMs)
6
7 // 매처와 일치하는 노드가 정확히 하나 존재할 때까지 대기
8 composeTestRule.waitForExactlyOneExists(matcher, timeoutMs)
9
10 // 매처와 일치하는 노드 수가 지정된 개수가 될 때까지 대기
11 composeTestRule.waitForNodeCount(matcher, count, timeoutMs)

```

Compose에서 단위 테스트를 작성하고 특정 조건이 충족될 때까지 기다리는 API에 대해서는 [Alternatives to Idling Resources in Compose tests: The `waitFor` APIs⁹⁴](#)를 확인하시면 더 자세히 학습하실 수 있습니다. UI 테스트를 작성할 때 의외로 자주 사용되는 API라서 추가학습을 권장합니다.

지연 목록(Lazy Lists) 테스트하기

`LazyColumn`과 같은 스크롤 가능한 콘텐츠의 경우 `assertIsDisplayed()` 또는 `assertDoesNotExist()`를 통해 컴포넌트가 현재 화면에 보이는지 확인하거나, `performScrollToNode()`를 통해 화면 밖에 있는 컴포넌트를 테스트할 수 있습니다.

⁹⁴<https://medium.com/androiddevelopers/alternatives-to-idling-resources-in-compose-tests-8ae71f9fc473>

그림 373. scrollToItem.kt

```
1 @Test
2 fun scrollToItem() {
3     val list = List(100) { "item$it" } // 100개의 아이템 생성
4
5     composeTestRule.setContent {
6         LazyColumn(modifier = Modifier.testTag("lazyColumn").height(200.dp)) { // 높이
7             → 제한 추가
8                 items(items = list) { value ->
9                     Text(value, Modifier.testTag(value).height(50.dp)) // 아이템 높이 지정
10                }
11            }
12
13        // 초기에는 item50이 보이지 않음을 확인
14        composeTestRule.onNodeWithTag("item50").assertDoesNotExist()
15        // item50 텍스트를 가진 노드로 스크롤
16        composeTestRule.onNode(hasScrollAction()).performScrollToNode(hasText("item50")) //
17             → hasScrollAction() 추가
18        // 스크롤 후 item50이 표시되는지 확인
19        composeTestRule.onNodeWithTag("item50").assertIsDisplayed()
}
```

해당 테스트는 LazyColumn을 50번 대 아이템까지 화면을 스크롤하여, 처음에는 화면에 보이지 않는 아이템이 정상적으로 잘 나타나는지 확인합니다.

UI 시맨틱스 및 접근성 검증하기

Compose의 테스트 프레임워크는 **시맨틱스 속성(semantic properties)** 을 지원하여 콘텐츠 설명과 같은 접근성 속성을 검증할 수 있습니다.

그림 374. testContentDescription.kt

```
1 @Test
2 fun testContentDescription() {
3     composeTestRule.setContent {
4         Icon(imageVector = Icons.Default.Home, contentDescription = "Home Icon")
5     }
6
7     // contentDescription이 "Home Icon"인 노드 찾아서 존재하는지 확인
8     composeTestRule.onNodeWithContentDescription("Home Icon").assertExists()
9 }
```

onNodeWithContentDescription을 사용하여 접근성 레이블이 올바르게 할당되었는지 검증합니다.

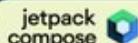
Jetpack Compose UI 테스트 작성은 노드 조회, UI 상호 작용, 리소스 유무 상태 등을 위한 효율적인 라이브러리 덕분에 전통적인 XML 기반 레이아웃에 대해 UI 테스트를 작성하는 것이 월등히 쉽습니다. 특히 XML 기반 레이아웃의 경우는 모든 컴포넌트가 layout XML 파일 안에 집약적으로 사용되고 있고 각 컴포넌트의 재사용성이 좋지 않아서 컴포넌트 별 독립적인 UI 테스트 작성도 쉽지 않았고, 그렇다고 화면 하나에 대해서 통째로 UI 테스트를 작성하는 것도 쉽지 않았습니다.

Jetpack Compose의 UI 테스트 라이브러리는 이를 월등히 개선시켜 주어, 누구나 쉽게 테스트 UI를 작성할 수 있도록 접근성을 대폭 낮추었습니다. UI 테스트 관련 API를 깔끔하게 정리해 놓은 [Jetpack Compose 테스트 치트 시트](#)⁹⁵를 통해 활용할 수 있는 API를 확인해 보시는 것을 권장합니다.

⁹⁵<https://developer.android.com/develop/ui/compose/testing/testing-cheatsheet>

Testing cheat sheet

v1.1.0



Finders

```
onNode(matcher)
onNodeWithContentDescription
onNodeWithTag
onNodeWithText
onRoot
```

```
onAllNodes(matcher)
onAllNodesWithContentDescription
onAllNodesWithTag
onAllNodesWithText
```

OPTIONS: useUnmergedTree: Boolean

Matchers

```
has[No]ClickAction
hasContentDescription[Exactly]
hasImeAction
hasProgressBarRangeInfo
has[No]ScrollAction
hasScrollTo[Index|Key|Node]Action
hasSetTextAction
hasSetTextAction
hasTextDescription
hasTestTag
hasText[Exactly]
is[Not]Dialog
is[Not]Enabled
is[Not]Focused
is[Not]Selected
isHeading
isOff
isOn
isPopup
isSelectable
isToggable
isFocusable
isRoot
```

HIERARCHICAL

```
hasParent
hasAnyChild
hasAnySibling
hasAnyDescendant
hasAnyAncestor
```

SELECTORS

```
filter(matcher)
filterToOne(matcher)
onAncestors
onChild
onChildAt
onChildren
onFirst
onLast
onParent
onSibling
onSiblings
```

Assertions

```
assert(matcher)
assertExists
assertDoesNotExist
assertContentDescriptionContains
assertContentDescriptionEquals
assertIs[Not]Displayed
assertIs[Not]Enabled
assertIs[Not]Selected
assertIs[Not]Focused
assertIsOn
assertIsOff
assertIsToggable
assertIsSelectable
assertTextEquals
assertTextContains
assertValueEquals
assertRangeInfoEquals
assertHas[No]ClickAction
```

COLLECTIONS

```
assertAll
assertAny
assertCountEquals(Int)
```

BOUNDS

```
assert[Width|Height]IsEqualTo
assertIsEqualTo
assert[Width|Height]IsAtLeast
assertTouch[Width|Height]IsEqualTo
assertTopPositionInRootIsEqualTo
assertLeftPositionInRootIsEqualTo
getAlignmentLinePosition(Baseline)
getUnclippedBoundsInRoot
```

Actions

```
performClick
performTouchInput
performMultiModalInput
performScrollTo
performSemanticsAction
performKeyPress
performImeAction
performTextClearance
performTextInput
performTextReplacement
```

TOUCH INPUT

```
click
doubleClick
longClick
pinch
swipe
swipe[Down|Left|Right|Up]
swipeWithVelocity
```

TOUCH INPUT PARTIAL

```
down
moveTo
movePointerTo
moveBy
movePointerBy
move
up
cancel
```

ComposeTestRule

```
@get:Rule
val testRule =
    createComposeRule()
```

```
setContent {
    density
    runOnUiThread { }
    runOnUiThread { }
    waitForIdle()
    waitUntil { }
    awaitIdle()
    [un]registerIdlingResource()
    mainClock.advanceTime
    mainClock.currentTime
    mainClock.advanceTimeBy()
    mainClock.advanceTimeByFrame()
    mainClock.advanceTimeUntil { }
```

AndroidComposeTestRule

```
@get:Rule
val testRule =
    createAndroidComposeRule<Activity>()
```

```
ComposeTestRule.* +
    activity
    activityRule
```

Debug

```
onNode(...).*
```

```
printToString()
printToLog()
captureToImage()
```

요약

Compose UI 테스트는 정확성, 안정성 및 접근성을 검증하고, 컴포넌트가 개발자의 의도에 맞게 동작하는지를 확인합니다. `ComposeTestRule`를 통해 UI 상호작용을 가능하게 하고, 동기화 메커니즘 활용, 자연 목록 검증, 접근성 시맨틱스 확인 등을 통해 더 면밀하게 컴포넌트의 동작을 검증할 수 있습니다.

실전 질문

Q) 컴포저블이 올바른 UI 컴포넌트를 랜더링 하는지 확인하기 위해 UI 테스트를 어떻게 작성하면 좋을까요? 만약 리스트의 50번째 항목과 같이 리스트의 하단에 위치한 아이템의 경우는 어떻게 확인하나요?

Q) `performClick()`과 같은 상호작용 테스트 API나 `assertExists()` 또는 `assertTextEquals()`와 같은 `assert` 관련 API는 어떤 시나리오에서 유용한지 설명해 주세요.

Q) 40. 스크린샷 테스트(screenshot testing)란 무엇이며, UI 일관성을 보장하는 데 어떻게 도움이 되나요?

스크린샷 테스트는 실제 기기에서 앱을 실행하지 않고 UI 렌더링 결과를 확인하는 효과적인 방법입니다. 특히 코드 리뷰 단계에서 컴포넌트 변경사항에 대한 새로운 스크린샷을 이전 스크린샷과 비교하여 변경 사항을 시각적인 형태로 감지할 수 있어 수정 사항을 쉽고 빠르게 식별할 수 있습니다. 따라서 팀 구성원들이 코드 리뷰 단계에서 컴포넌트의 변화를 직관적으로 판별할 수 있도록 하여 팀 효율성을 향상시킬 수 있습니다.

Jetpack Compose에서 스크린샷 테스트를 수행하는 세 가지 방법이 있습니다. **Google의 공식 Gradle 플러그인**과 오픈소스 커뮤니티에서 유명한 **Paparazzi** 및 **Roborazzi**입니다. 각 접근 방식은 UI 스냅샷을 효율적으로 캡처하고 비교하는 데 각자만의 이점을 제공합니다.

Compose 스크린샷 테스트 플러그인

Google에서 공식적으로 제공하는 **Compose 스크린샷 테스트 플러그인**⁹⁶은 **Compose Preview**⁹⁷와 함께 작동하여 개발자가 UI 스냅샷을 원활하게 생성하고 비교할 수 있도

⁹⁶<https://developer.android.com/studio/preview/compose-screenshot-testing>

⁹⁷<https://developer.android.com/develop/ui/compose/tooling/previews>

록 합니다. 이 방법은 UI 일관성을 확인하고 의도하지 않은 레이아웃 변경 사항을 감지하는데 유용합니다.

스크린샷 테스트는 UI 스냅샷을 캡처하고 이전에 생성되었던 스냅샷 이미지와 비교합니다. 차이점이 감지되면 테스트가 실패하고 변경 사항을 상세하게 알려주는 HTML 리포트를 생성합니다.

Compose Preview 스크린샷 테스트 도구를 사용하여 다음을 항목들을 수행할 수 있습니다.

- 스크린샷 테스트를 위한 컴포저블 미리보기 선택.
- 비교를 위한 참조 이미지 생성.
- UI 변경 사항 자동 감지 및 HTML 리포트 생성.
- 테스트 범위를 확장하기 위해 uiMode 및 fontScale과 같은 @Preview 매개변수 사용.
- 모듈화를 위해 screenshotTest 소스 세트를 사용하여 테스트 구성.

이 접근 방식은 UI 일관성을 보장하고 시각적 회귀를 효율적으로 감지하는 데 도움이 됩니다.

Class com.google.samples.apps.nowinandroid.feature.foryou.FоХуScreenshotTests

all > com.google.samples.apps.nowinandroid.feature.foryou > FоХуScreenshotTests

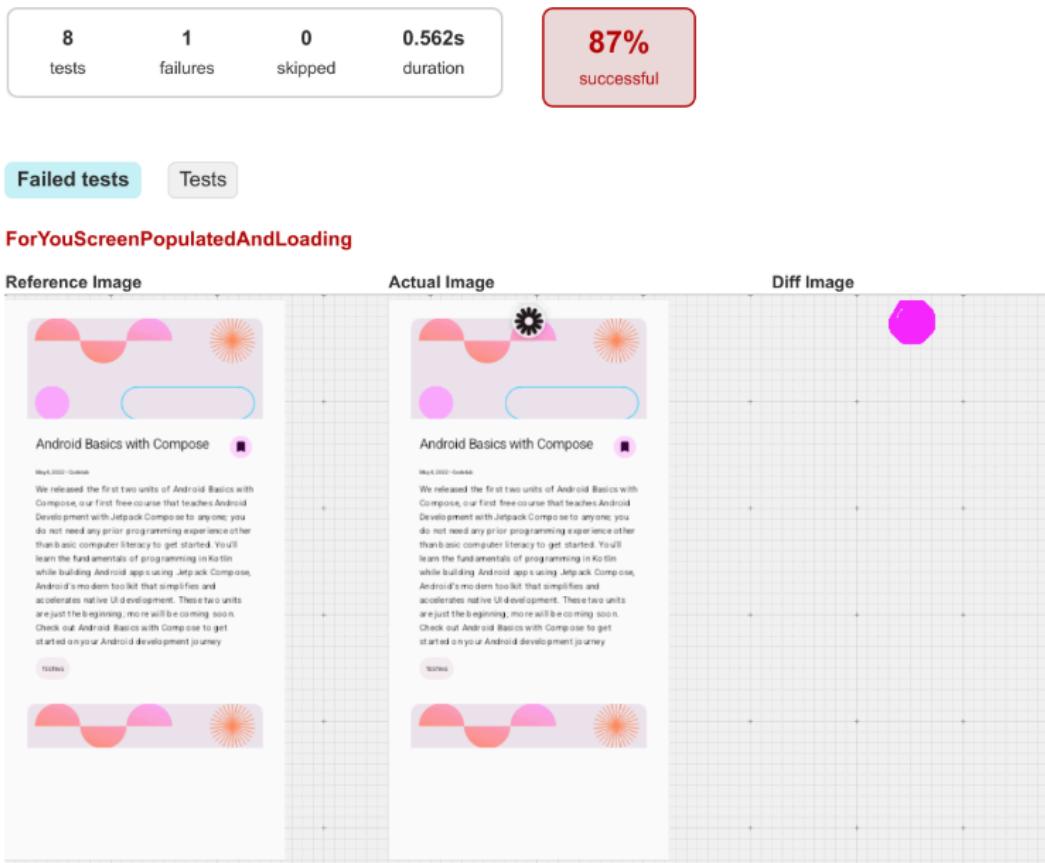


그림 376. compose-snapshot-testing

Paparazzi

Paparazzi⁹⁸는 Cash App에서 개발한 오픈 소스 라이브러리로, 에뮬레이터나 실제 기기 없이 스크린샷 테스트를 가능하게 합니다. 모든 작업은 JVM에서 실행되므로 UI 스냅샷을 캡처하는 빠르고 효율적인 방법입니다. Paparazzi는 JVM에서 직접 Compose UI를 렌더링하고 비교를 위해 픽셀 단위로 완벽한 스크린샷을 캡처하는 방식으로 작동합니다.

다음 예제를 사용하면 실제 기기나 에뮬레이터 없이 Android Studio에서 직접 Compose UI

⁹⁸<https://github.com/cashapp/paparazzi>

화면을 렌더링할 수 있습니다.

그림 377. Paparazzi.kt

```
1 import app.cash.paparazzi.DeviceConfig.Companion.PIXEL_5
2 import app.cash.paparazzi.Paparazzi
3 import org.junit.Rule
4 import org.junit.Test
5
6 class LaunchViewTest {
7     @get:Rule
8     val paparazzi = Paparazzi(
9         deviceConfig = PIXEL_5, // 기기 종류 설정
10        theme = "android:Theme.Material.Light.NoActionBar" // 테마 설정
11        // ... 더 많은 옵션은 문서 참조
12    )
13
14    @Test
15    fun launchView() {
16        // XML 레이아웃 인플레이트
17        val view = paparazzi.inflate<LaunchView>(R.layout.launch)
18        // 또는 프로그래밍 방식으로 View 생성
19        // val view = LaunchView(paparazzi.context)
20
21        view.setModel(LaunchModel(title = "paparazzi")) // 모델 설정
22        paparazzi.snapshot(view) // 스냅샷 생성
23    }
24
25    @Test
26    fun launchComposable() {
27        // Composable 스냅샷 생성
28        paparazzi.snapshot {
29            MyComposable()
30        }
31    }
```

Roborazzi

Roborazzi⁹⁹는 Jetpack Compose를 포함한 안드로이드 스크린샷 테스트를 위해 설계된 오픈 소스 라이브러리입니다. 스냅샷 비교를 통해 UI 상태를 캡처하고, UI 변경 사항을 확인하기 위한 간단하고 유연한 API를 제공합니다.

Roborazzi는 Robolectric¹⁰⁰ 위에서 동작하여, Hilt와 함께 테스트를 실행할 수 있고 보다 현실적인 환경에서 UI 컴포넌트와 상호 작용할 수 있습니다. Robolectric은 JVM 환경이 아니라 에뮬레이터나 실제 디바이스에서 실행되는 환경이기 때문에, 실제 렌더링 환경에서의 스크린샷을 캡처함으로써 Paparazzi가 커버하지 못하는 한계점을 개선하여 런타임 의존성 주입 및 기타 시스템 수준의 상호 작용 등 기존 호환성을 보장하면서 테스트 프로세스를 더 효율적이고 신뢰할 수 있게 만듭니다.

또한 Compose Multiplatform 지원¹⁰¹, Compose Preview 통합¹⁰², AI 기반 이미지 검증¹⁰³ 등과 같은 유용한 기능을 포함합니다.

요약

스크린샷 테스트는 UI 변경 사항을 직관적으로 추적하고 디자인 일관성을 보장하는 신뢰할 수 있는 방법을 제공합니다. Google Compose 스크린샷 테스트 플러그인, Paparazzi, Roborazzi는 각자의 이점을 제공하기 때문에, 현재 프로젝트의 구조나 워크플로에 맞는 솔루션을 골라서 통합하는 것이 좋습니다. 스크린샷 테스트를 채택함으로써 팀은 시각적 회귀를 조기에 식별하고 코드 리뷰 중 협업을 개선하며 앱 버전 전체에서 일관되고 올바른 UI 경험을 유지할 수 있습니다.

실전 질문

Q) 팀 워크플로의 일부로 스크린샷 테스트를 사용해 본 적이 있나요? 개발 또는 코드 리뷰 프로세스를 어떻게 개선했으며, 경험하신 구체적인 이점은 무엇인가요?

⁹⁹<https://github.com/takahirom/roborazzi>

¹⁰⁰<https://github.com/robolectric/robolectric>

¹⁰¹<https://takahirom.github.io/roborazzi/compose-multiplatform.html>

¹⁰²<https://takahirom.github.io/roborazzi/preview-support.html>

¹⁰³<https://takahirom.github.io/roborazzi/ai-powered-image-assertion.html>

Q) 41. Jetpack Compose에서 접근성을 어떻게 보장하나요?

Jetpack Compose에서 접근성 보장¹⁰⁴은 스크린 리더와 같은 어시스턴트 기능을 통해 앱 화면이나 기능을 쉽게 해석하고 상호 작용할 수 있도록 합니다. Compose는 선언적 UI 모델을 유지하면서 접근성 기능을 더 쉽게 구현할 수 있는 유연한 API를 제공합니다.

시맨틱스 Modifier (Semantics Modifier)

Compose 접근성 시스템의 핵심에는 semantics Modifier가 있습니다. UI 컴포넌트가 접근성 서비스에 의해 어떻게 해석되어야 하는지 설명하는 모든 내용을 담고 있습니다.

그림 378. SemanticsModifier.kt

```
1 Modifier.semantics {  
2     // 역할 부여 (예: 버튼 역할)  
3     // role = Role.Button  
4     // 콘텐츠 설명 설정  
5     contentDescription = "전송 버튼" // "Send Button"  
6     // 클릭 액션 설명 추가  
7     // onClick(label = "메시지 전송", action = { /* 클릭 처리 */ true })  
8 }
```

해당 Modifier는 콘텐츠 설명, 역할 또는 커스텀 작업과 같은 메타데이터를 어시스턴트 도구에 전달합니다. 그러나 Text, Button, Icon과 같은 대부분의 컴포저블에는 이미 내장된 시맨틱스가 있기 때문에 일반적으로 커스텀 컴포넌트에만 수동적으로 추가해도 됩니다.

이미지 및 아이콘을 위한 contentDescription

contentDescription 매개변수는 Image 및 Icon 컴포넌트의 기본 접근성 매개변수입니다. 시각적 콘텐츠에 대한 텍스트 컨텍스트를 제공합니다.

¹⁰⁴<https://developer.android.com/develop/ui/compose/accessibility>

그림 379. IconWithContentDescription.kt

```

1  Icon(
2      imageVector = Icons.Default.Send,
3      contentDescription = "전송" // "Send"
4 )

```

이미지가 어떤 중요한 정보를 포함하고 있는 것이 아니라 단순히 꾸미는 역할만 하는 경우 null 값을 전달하여 접근성 서비스에서 제외할 수 있습니다.

그림 380. DecorativeImage.kt

```

1  Image(
2      painter = painterResource(id = R.drawable.divider),
3      contentDescription = null // 별다른 역할을 하지 않는 이미지는 설명을 null로 설정
4 )

```

그룹화된 콘텐츠에 대한 시맨틱스 병합 (Merging Semantics)

여러 컴포넌트를 하나의 단일 논리 단위로 표시해야 하는 경우 Modifier.clearAndSetSemantics {} 또는 Modifier.semantics(mergeDescendants = true)를 사용하여 접근성을 목적 단위로 그룹화할 수 있습니다.

그림 381. MergedSemantics.kt

```

1  Column(
2      // 자식 요소들의 시맨틱스를 병합하여 하나로 읽도록 함
3      modifier = Modifier.semantics(mergeDescendants = true) {}
4  ) {
5      Text("항공편: NZ123") // "Flight: NZ123"
6      Text("출발: 오전 10:30") // "Departure: 10:30 AM"
7 }

```

이를 통해 어시스턴트 기능이 해당 콘텐츠를 단일 항목으로 읽도록 보장합니다.

커스텀 접근성 작업 (Custom Accessibility Actions)

스크린 리더나 다른 접근성 도구를 사용하는 사용자의 상호 작용을 향상시키기 위해 아래와 같이 커스텀 작업을 추가할 수 있습니다.

그림 382. CustomSemanticsAction.kt

```

1 Modifier.semantics {
2     // 커스텀 클릭 액션 추가
3     customActions = listOf(
4         CustomAccessibilityAction(
5             label = "북마크하려면 두 번 탭하세요", // "Double tap to bookmark"
6             action = { /* 북마크 처리 로직 */ true }
7         )
8     )
9     // 기본 클릭 액션도 설정 가능
10    // onClick("북마크") { ... }
11 }
```

이를 통해 어시스턴트 기술 사용자를 위한 더 친절하고 상세한 UI 경험을 제공할 수 있습니다.

접근성 테스트하기

[Accessibility Scanner](#)¹⁰⁵ 또는 Compose UI 테스트의 AccessibilityTestRule을 사용하여 접근성 레이블, 역할 및 계층 구조를 검증할 수 있습니다. 또한 Compose는 적절한 접근성 동작을 보장하기 위해 테스트에서 시맨틱스 단언(assertion)을 허용합니다.

그림 383. ComposeAccessibilityTest.kt

```

1 // contentDescription이 "전송"인 노드가 존재하는지 검증
2 composeTestRule.onNodeWithContentDescription("Send").assertExists()
```

Compose 테스트에서 onNodeWithContentDescription을 사용하여 접근성 레이블이 올바르게 할당되었는지 확인할 수 있습니다.

¹⁰⁵<https://support.google.com/accessibility/android/answer/6376570?hl=en>

요약

Jetpack Compose는 개발자가 접근성 UI를 쉽게 구현할 수 있도록 semantics, contentDescription, mergeDescendants와 같은 구조화된 API를 제공합니다. 시각적 요소를 적절하게 어노테이션하고 관련 콘텐츠를 그룹화함으로써 보조 기술에 의존하는 사용자를 포함하여 더 넓은 범위의 사용자가 애플리케이션을 사용할 수 있도록 보장할 수 있습니다.

실전 질문

Q) 시맨틱스 Modifier의 목적은 무엇인가요?

Q) Compose 컴포넌트에 접근성을 추가할 때 UI 컴포넌트를 그룹화하려면 어떻게 해야 하나요?