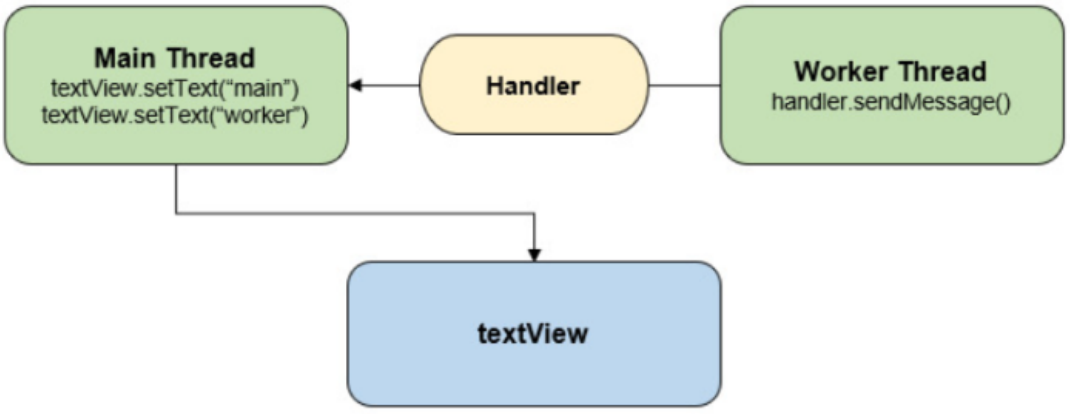
면접 질문 리스트

시작일	@2024년 2월 7일
태그	

기본 기술 질문

스레드

1. **메인 스레드랑 워커 스레드 있음.** 워커는(시간이 오래 걸리는 작업) 동기화 문제 때문에 무조건 핸들러를 거쳐서 메인으로 온다



▼ 스레드를 시작하는 방법

1. thread class를 상속받고 run을 오버라이딩, 정의, 실행
2. runnable 인터페이스를 확장해 run()구현, 정의 후 실행

   → 이게 좋음! 필요한 기능만 갖고(run하나만 있음), 클래스간 결합도도 낮춤. 상속받으면 private이 아닌 메서드나 변수를 자식이 다 받아감

- 코틀린에서는 thread(start = true) {} 요렇게 해서 블럭 안 코드 실행 가능!

▼ 살아있는 스레드를 종료하는 방법

- 직접적으로 스레드를 종료 할 수는 없음
- interrupt() 메서드를 사용

    - 스레드가 **일시 정지**라면 → interruptedException 발생
    - getIsInterrupted()를 통해 가능한지 먼저 확인할 것

▼ 안드로이드에서 멀티 스레딩을 하는 이유

- 앱이 눈에 띄는 일시중지 없이 사용자에게 표시되도록 하려면 기본 스레드가 16ms 마다 또는 더 자주 화면을 업데이트하거나 초당 약 60프레임으로 화면을 업데이트 해야 함. 이 속도에서 인간은 프레임 변경이 완전히 원활하다고 인식

▼ SOLID

- SRP : single responsibility principle : 단일 책임 원칙
- OCP : open-closed principle : 개방-폐쇄 원칙(확장에는 열려있고 변경에는 닫혀있음)
- LSP : Liskov Substitution Principle : 리스코프 치환 원칙(부모 클래스의 인스턴스는 자식 클래스의 인스턴스로 교체 가능)→ 상속 위함!
- ISP : Interface Segregation Principle : 인터페이스 분리 원칙(인터페이스는 꼬오오옥 필요한 기능만 포함해야함)
- DIP : Dependency Inversion Principle : 의존성 역전 원칙(구체적인 구현이 아니라 추상 클래스와 같은 추상화에 의존해야합니다)
    - 추상화 : 개념을 단순화하고 필수적인 요소만을 강조해 모델링하는 과정. 세부사항을 숨기고 핵심 개념을 이해하기 쉽게 해줌

▼ 자바 코틀린 차이

- 코틀린은 문법이 간결하고 표현력이 높음.
- null 안정성. (타입 시스템을 통해)
- 함수형 프로그래밍 지원
    - **함수형 프로그래밍** ; 대입문을 사용하지 않는 프로그래밍
    - *부수 효과가 없는 순수 함수를 1급 객체로 간주하여 파라미터나 반환값으로 사용할 수 있으며, 참조 투명성을 지킬 수 있음*
        - 부수효과 : (변수의 값 변경/자료 구조를 제자리에서 수정/객체의 필드값을 설정/예외나 오류가 발생하며 실행이 중단됨/콘솔 또는 파일 I/O가 발생)
        - 순수함수 : 부수효과 없는 함수(스레드 안정성)
        - 1급 객체 : 변수나 데이터 구조 안에 담을 수 있고, 파라미터로 전달도 가능하고, 반환값으로도 사용되는 친구

    - 참조 투명성 : 동일인자에 대해 동일 반환

  예시

```
Function<String, String> function = word -> word.toU
```

- 확장 함수와 확장 프로퍼티 제공(String.lastChar←프로퍼티)
- kotlin의 null check → val length = name?.length(null아닐때만 접근)
- kotlin의 null 안전 cast → val intValue: Int? = number as? Int (실패시 null 반환)
- nonnull 연산자 !! → 안전하지 않음
- kotlin에서 Int는 있으나 integer는 없음. 후자는 null같은게 있는 경우 쓰이는걸로, 컴파일 타임에 컴파일러가 알아서 결정해줌

▼ JAVA에는 없고 Kotlin에는 있는것은?

- null safety
- 연산자 오버로딩
- 코루틴
- 범위 표현식
- 스마트 캐스트 (n is Num)
- 동반 객체 (companion obj)

▼ Interface와 abstract의 특징과 차이

- 인페는 클래스 여러개 받을 수 있음. 멤버 변수는 없고, 추상함수, 디폴트 함수, 정적 함수, 추상 프로퍼티, 상수만 있단다~
- 후자는 **하나 이상의 추상 메서드가 있는 클래스**임...**추상 메서드는 선언만 있고 구현은 없음**. **생성자도 있음**. **다른 클래스가 하나만 받을 수 있음**. 공통된 특성과 동작이 있는 클래스들의 베이스

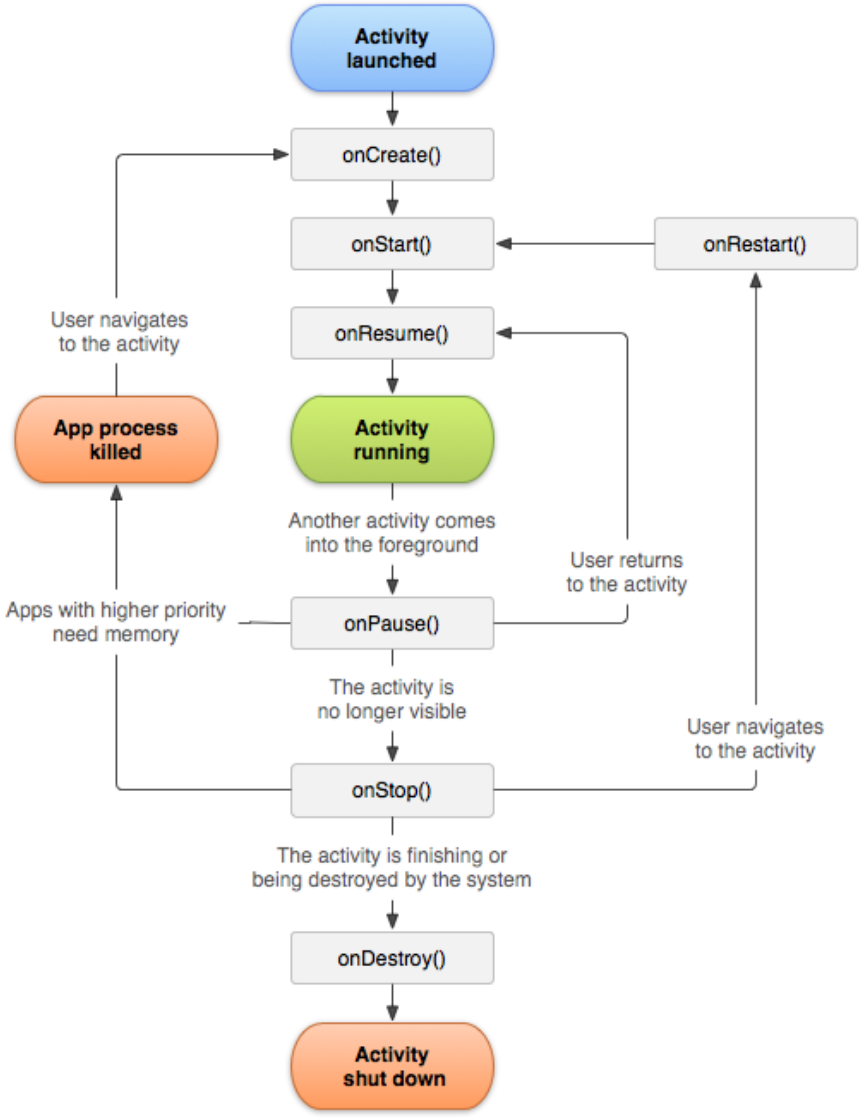
▼ Scope func / 각각 어떤 차이가 있을까?

→ 범위 함수. 객체의 범위 내에서 간결한 코드 작성 가능.

1. let : null아닌 경우만 블록 내에서 해당 객체 사용 가능
2. run : 객체의 컨텍스트 내에서 코드 블록 실행. null아닌 경우만 돌아. 반환값은 맘대로

    a. runCatching이라고 Result를 mapping해서 사용할 수 있는 친구도 있음
3. with : 특정 객체의 컨텍스트 내에서 코드 블록 실행. 객체 자체를 받아 사용. 참조 노필요
4. apply : 객체 속성 초기화하거나 수정하는데 씀. 객체 자체를 받음. 객체 속성에 연속적으로 접근해 값 설정 가능
5. also : 객체를 받아서 사용하는 let과 유사. 전달받은 객체를 유지한채 다른 용도로 사용 가능

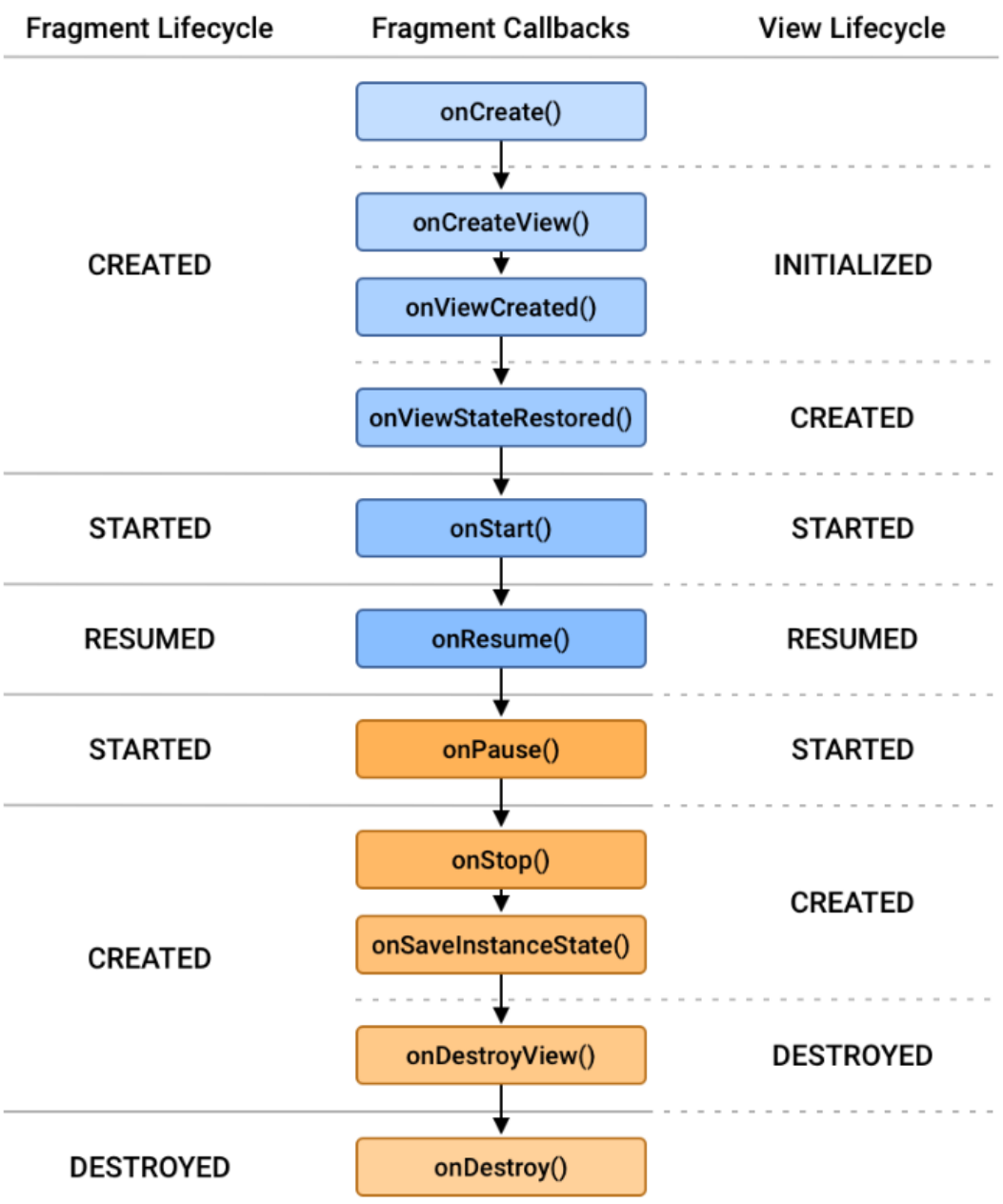
▼ Activity 생명주기



1. onCreate : 액티비티 첫 생성. 초기화, 인페설정 등

2. onStart : activity가 화면에 보여지기 직전
3. onResume: 액티비티가 사용자랑 상호작용을 시작함.
4. onPause : 일시 중지 되거나, 다른 액티비티가 화면 가려서 포그라운드에서 사라짐. 데이터 저장, 네트워크 연결 해제 등 함(다이얼로그 띄울 때)
5. onStop : 더이상 안보이고 화면에서 완전히 가려짐. 자원해제나 정리
6. onRestart : 5번에서 다시 시작되기 전에 호출되는 메서드
7. onDestroy : 액티비티 소멸~할당된 자원 해제하고 종료

▼ Fragment 생명 주기

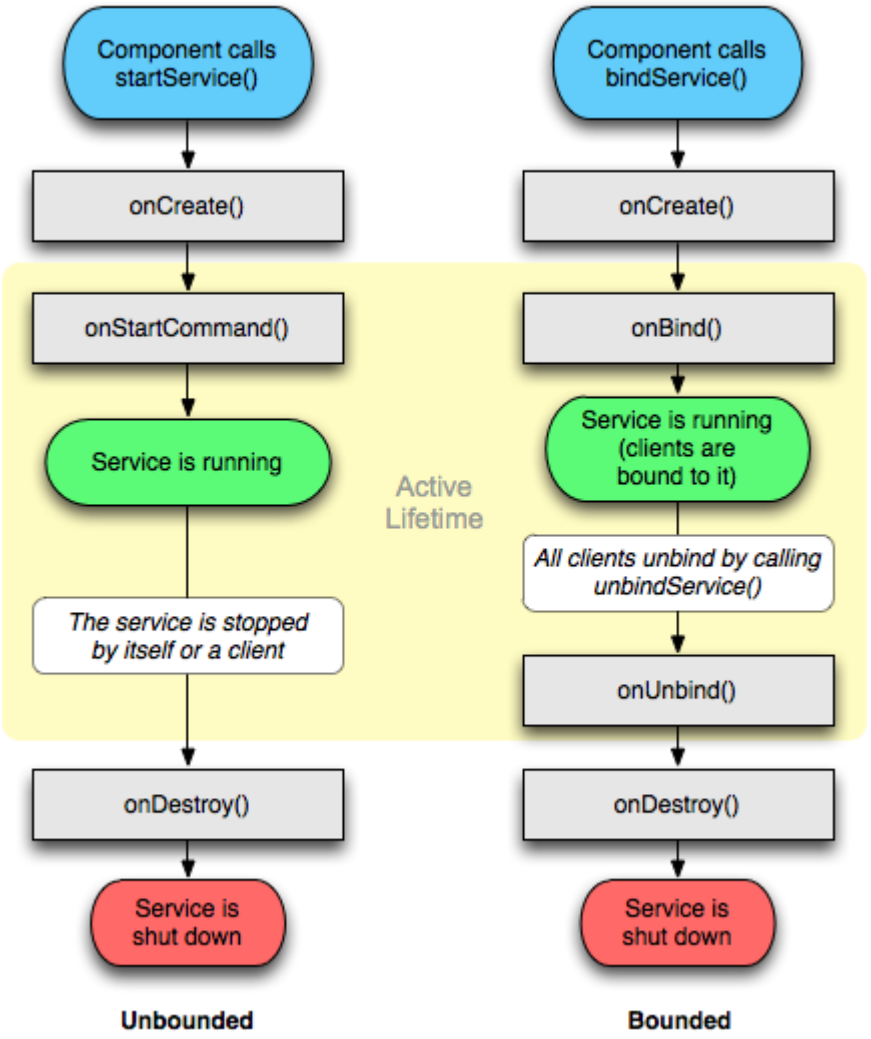


- + onAttatch : fragment가 activity에 처음 연결될 때
- onCreate : frag 생성시. 초기화나 interface설정 등을 수행
- onCreateView : 레이아웃 그리기 시작
- onViewCreated : 레이아웃 그리는거 끝남. UI관련 작업을 함(초기화)
- + onStart : 화면에 보여지기 시작
- onResume : 유저와 상호작용을 시작함

- onPause : 포그라운드에서 잠깐 사라짐. 특별히 하는 건 없음
- onStop : 완전히 화면에서 가려질때. 자원 해제 시작함
- onDestroyView : UI 소멸될ㄸㅐ ~ UI 관련 리소스 해제는 여기서 시작합니다
- onDestroy : 소멸될ㄸㅐ ~ ↔ onCreate 자원 해제하고 종료 시작 → 위에거랑 차아는?
- onDetatch : activity에서 완전히 떨어질 때 ↔ onAttatch

▼ 4대 컴포넌트

1. activity
    a. UI. 사용자 입력을 처리하고, 다른 액티비티와의 화면 전환, 상호작용 등
    b. MVVM , MVI, MVP, MVC 등에서 V에 해당하는 친구
2. content provider
    a. 데이터 관리하고 다른 앱과 데이터 공유위해 인페 제공
    b. 다른 앱과의 상호작용을 통해 데이터를 교환~
    c. 파일 / DB / 네트워크 등의 소스에 접근
    d. 생명주기는 onCreate하나
3. broadcast receiver
    a. 브로드캐스트 메세지 수신 (예를 들면....알림? 주로 노티)
    b. 시스템 이벤트도 수신 가능 (배러리 부족~ 시간 변경 등)
    c. 다른 앱의 이벤트도 수신 가능
    d. 열공클럽의 댓글 토글에 사용중
    e. 생명주기 별도로 없음
4. service
    a. 백그라운드에서 실행됨. UI 없음 당연함~~
    b. 데이터 동기화 음악 재생 등 백그라운드에서 오~~~~래 돌아가는거
    c. 타이머로 쓰는중
    d. 앱이 활성화되지 않아도 잘 된답니다?
    e. 생명주기는 아래와 같다



▼ 흔한 패턴 M몇글자 (4개)

- MVC : model - view - controller

  장점

  - 가장 간단하고 만만
  - 모델은 데이터
  - 컨트롤러는 사용자 입력 처리
  - 범용적

  단점

  - 컨트롤러 사이즈 컨트롤 안됨 푸항항 ~
  - 데이터 흐름이 복잡해질 수 있음

- MVP : model - view - presenter
  - 위랑 별 차이 없는 애임

    - 모델 역할 똑같고, 뷰도 똑같음
    - 프레젠터는 모델과 뷰 사이 중간 역할
    - 뷰와 프레젠터 사이 인터페이스를 통한 느슨한 결합 유지

    → 둘 사이 차이점은

mvc	mvp
뷰와 모델이 긴밀하게 결합	뷰는 모델과 느슨하게 결합
컨트롤러와 뷰 레이어는 동일한 activity/fragment안에 있음	view-presenter-model간의 통신은 인터페이스를 통해 이루어짐
사용자 입력은 컨트롤러에 의해 처리	사용자 입력은 view에 의해 처리
컨트롤러와 뷰는 다대일	뷰와 프레젠터는 일대일
단위테스트 제한된 지원	단위 테스트가 강력하게 지원

    장점

    - 코드 재사용성 모듈화 촉진
    - 테스트와 유지보수 용이
    - UI / 비즈니스 로직 분리

    단점

    - 구현에 따라 프레젠터의 역할 및 책임 정의 어려움
    - view / presenter 사이의 인터페이스 설계에 신경써야함
- MVVM : model - view - viewmodel
    - 모델은 데이터
    - 뷰는 인페
    - 뷰모델은 뷰와 모델 사이의 매개체
    - 뷰모델은 뷰와 완전히 분리
    - 뷰모델은 뷰에 대한 상태 및 동작 관리, 사용자 입력 및 이벤트 처리 담당
    - 유연하고 확장 가능한 구조 제공 : UI, 비즈니스 로직 분리 촉진

    장점

    - Command pattern으로 사용자 인터랙션 처리 가능(.invoke)
    - UI 개발자 / 비즈니스 로직 개발자간 협업 용이

단점

- 러닝커브가 가파름다. 복잡한 앱은 오버헤드 발생
- 뷰모델이 커질 가능성 존재
- 앱이 작으면 과도한 추상화~

- MVI : model - view - intent
  - 모델은 앱의 상태
  - 인텐트는 사용자나 앱 내의 액션들(프레젠터가 이를 업데이트)
  - MVI에서 모델의 업데이트는 redux의 개념을 빌려옴
    - state : 앱 상태 홀더
    - Action : state 바꾸기 위한 명령
    - Reducer : 이전의 state + action받아서 새로운 state만들어주는 순수 함수
  - compose와 함께 사용됨

  장점
  - 앱의 상태는 하나라, 상호작용이 많아져도 상태 충돌이 없음
  - 데이터 흐름 정해져있어 흐름 이해 쉽고 관리도 쉬움
  - 각각 값이 불변, 스레드 안정성

  단점
  - 약간의 러닝커브
  - 작은 변경도 무조건 intent, 아주 작은 앱도 최소한의 intent와 model 있어야 함
  - model 업데이트 위해 매번 새로운 인스턴스가 필요. 너무 많은 객체 → gc 자주 작동

▼ DI

Dependency Injection : 의존성 주입

클래스를 종속성들과 독립적으로 만들기위한 디자인 패턴

객체의 생성과 사용을 분리해 이를 달성. SOLID의 SRP, DIP를 가능케함

인스턴스를 클래스 외부에서 주입하기 위해서는 인스턴스의 생명주기 관리가 필요함

이를 알아서 관리해주는게 Dagger / Dagger - Hilt

- Dagger의 단점
  - annotation / annotation processing, 각 annotation에 대한 역할, module / component 간의 관계, scope 개념 등 라이브러리에 대한 아주 많은 이해 필요로함

    → 러닝커브가 높아요

    → 프로젝트 상황에 따라 초기 DI 환경 구축하는데 비용이 많이 듬.
- Koin의 단점
  - 쉬우나, DI 보다는 kotlin dsl을 활용한 service locator pattern이다
    - 로케이터에 객체 초기화 방법 등록, 해당 객체를 필요로 하는 곳에서 로케이터를 통해 객체를 제공받도록 하는 방법
    - 객체를 등록하는걸 동적으로 관리하므로 컴파일시에 어떤 문제가 있는지 알 수 없고,
    - 로케이터에 없는 객체를 요구할경우 런타임 에러가 생긴다)
    - 그리고 모든 클래스들이 서비스 로케이터에 종속되게된다(사용하려면)

    → 결과적으로 프로젝트 규모가 커질수록 컴파일타임에 처리하는 대거보다 런타임 퍼포먼스가 떨어질 수밖에 없다.
- Hilt!

  구글에서 발표한 Android 전용 DI 라이브러리
  - Dagger2 기반.
  - annotation을 사용해 컴파일 시간에 해당 코드 생성, 런타임에 빠름
  - 나는 요거 썼더니 사용하는 annotation익히기도 너무 쉬웠고, 뷰모델과 모듈들을 주입하는데 주로 사용할 수 있었음
  - Annotiation설명
    - @HiltAndroidApp : 앱의 애플리케이션 클래스에 지정, 힐트 초기화. 힐트가 앱의 컴포넌트 계층 구조를 구성하고 DI를 가능하게 함
    - @AndroidEntryPoint : 액티비티, 프래그먼트, 서비스, 브로드캐스트 리시버 등에 이 어노테이션 지정해 해당 클래스에 종속성 주입할 수 있게함
    - @Inject : 주입할 필드, 생성자 또는 메서드에 지정됨. 힐트는 해당 필드나 생성자에 대한 인스턴스를 자동으로 생성하고 주입함
    - @Module : 힐트 모듈을 정의하는 클래스에 지정됨. 모듈은 주입할 종속성을 제공하는 방법을 정의

        - @Provides : 모듈 내의 메서드에 지정됨. 종속성 제공하고 엔포로 주입될 친구임
        - @InstallIn : 모듈을 특정 컴포넌트에 설치하는데 사용됨. @SingletonComponent, @ActivityRetainedComponent @ActivityRetainedComponent와 같은 힐트 컴포넌트가 대상
        - @SingletonComponent : 앱의 전역 범위에서 단일 인스턴스를 유지하는데 사용되는 컴포넌트. 앱의 라이프사이클동안 한 번만 생성되고 유지됨

▼ synchronized란

- 멀티 스레드 환경에서 동기화위해 사용되는 키워드. 동시에 여러 스레드가 접근할 수 있는 공유 자원에 대해 안전한 접근 보장
- room초기 인스턴스 생성할때 사용(여러 스레드에서 접근하더라도 하나만 생성되게)

```
companion object {
        private var INSTANCE: AppDatabase? = null

        fun getInstance(context: Context): AppDatabase? {
            if (INSTANCE == null) {
                synchronized(AppDatabase::class) {
                    INSTANCE = Room.databaseBuilder(contex
                }
            }
            return INSTANCE
        }
```

▼ AsynchTask란

- UI스레드를 적절하고 쉽게 사용하도록 고안된 방법

▼ serializable / parcelable

둘 다 객체를 직렬화 하는인터페이스

- 속도(빠르기) : Parcelable > Serializable
    - Serializable는 java reflection사용, 처리과정에서 오버헤드 발생.
    - Parcelable는 명시적으로 필드를 읽고 쓰는데 집중
- 메모리 사용 : Parcelable > Serializable

    - Parcelable은 Android Binder 메커니즘 사용함. 명시적으로 작성되므로 클래스 구조의 변경에 유연하게 대응 가능
    - Serializable은 자바 직렬화 메커니즘 사용하므로 제약 있을 수 있음. Serializable구조가 변경되면 역직렬화에 문제 생김
    - • 이 과정에서는 리플렉션이 사용되지 않으므로 가비지가 생성되지 않습니다.
- 명시성 : Parcelable < Serializable
    - Parcelable은 직렬화 및 역직렬화 과정을 명시적으로 작성해야함
    - Serializable는 특별한 메서드 필요하지않음
- 호환성 : Parcelable > Serializable
    - 자바 직렬화 메커니즘을 사용하므로 직렬 / 역직렬화에 제약 있음.
    - Parcelable은 명시적으로 작성되므로 클래스 구조의 변경에 유연하게 대응 할 수 있음
- 비슷한건 Gson이 있음

▼ FCM

1. POST_NOTIFICATION 권한 얻기
2. 토큰 등록
3. 메세지 받기

```
public class MyFirebaseMessagingService extends FirebaseMe

    @Override
    public void onNewToken(@NonNull String token) {
        super.onNewToken(token);
        //token을 서버로 전송
    }

    @Override
    public void onMessageReceived(@NonNull RemoteMessage r
        super.onMessageReceived(remoteMessage);
        //수신한 메시지를 처리
    }
}
```

▼ LiveData 와 Mutable LiveData

후자는 데이터 변경 가능, 전자는 읽기 전용

▼ 다른 비동기처리 말고 Coroutine 사용하는 이유

- 비교되는 rxjava보다 러닝커브가 낮은 편이다
- 가독성 : 동기 코드와 유사한 구문을 사용해 비동기 작업 표현 가능. 콜백보다 가독성이 좋음
- 순차적인 코드로 비동기 작업 작성 가능. 즉, 비동기 작업 여러개로 나누고 각 단계 순차 실행 가능
- 상태 유지 : 각 단계에서 상태 저장하고 다음 단계에서 이어서 실행 할 수 있음
- 에러 처리 : try - catch 사용 가능
- 확장성 : 일반적인 비동기 작업 이외에도 다양한 동시성 작업에 유용함. 여러개 비동기 작업 병렬로 실행하거나, 동기화 필요한 작업 순차로 실행 등 다양한 동시성 패턴 지원

- 사용한 서드파티 예시

▼ Coroutine Dispatcher 종류와 특징

디스패처는 코루틴 라이브러리에서 사용되는 스레드 관리 도구

- Dispatchers.Main : 메인스레드에서 동작. UI 업데이트와 사용자 상호작용 처리에 적합
- Dispatchers.IO : 네트워크 요청, 파일 입출력 작업등을 처리하기에 적합한 스레드 풀. 백그라운드에서 CPU를 많이 사용하지 않는 작업에 적합.
- Dispatchers.Default :CPU 집약적인 작업을 처리하기에 적합한 스레드풀. IO와 UI작업을 포함한 일반적인 작업에 사용됨
- Dispatchers.Unconfined : 현재 실행중인 스레드 그대로 사용. 특정 스레드에 구속되지 않으므로 사용 시 주의가 필요합니다

▼ Jetpack Navigation 에서 popUpTo와 popUpToInclusive

- popUpTo
  - 역방향 탐색시 백스택에서 제거할 목적지를 지정하는 속성
  - 목적지의 ID나 참조를 받을 수 있음. 이를 통해 역방향 탐색 중 해당 목적지 이후의 모든 프래그먼트 제거 가능

    - popUpTo = “@+id/destination_home”은 home까지 역방향으로 탐색하며 그 이후의 모든 프래그먼트를 제거함
- popUpToInclusive
    - 지정된 목적지를 포함하고, 그 이전의 모든 프래그먼트를 제거함. 대상 목적지도 같이 제거되는것.

▼ 동기와 비동기

동기 : 작업이 순차적, 기다렸다가 실행됨. 작업이 블로킹되고 대기될 수 있음

비동기 : 작업이 동시적. 이전 작업이 완료되기를 기다리지 않고 다음 작업을 진행. 각 작업은 별도의 스레드 또는 프로세스에서 실행됨. 작업간 상호 의존 없음

▼ 블로킹과 넌블로킹

- 블로킹 : 호출된 함수나 작업이 완료될 때까지 현재 실행 흐름이 멈춤. 파일을 읽을 때 블로킹을 수행하면 파일이 완전히 읽혀질 때까지 다음 코드는 실행되지 않음.
- 논블로킹 : 호출된 함수 또는 작업이 즉시 반환됨. 현재 실행 흐름이 멈추지않고 계속 진행됨. 나중에 완료되면 알람 받을 수 있음

▼ Access Token / Refresh Token

- Access
    - 인증된 사용자의 리소스에 대해 접근 권한을 나타내는 토큰. 기간이 짧음(분~시간)
    - 클라이언트에서 API요청할때 인증 수단으로 사용
    - 암호화되어 전송되어야하고, 얘는 권한과 범위를 포함함
- Refresh
    - Access 갱신을 위한 토큰
    - 위에 애보다 길다. 몇일 몇 주 대충 그럼
    - 새 Access를 얻기위해 사용함
    - 주로 사용자 인증 세션 유지하고, 재인증없이 액세스 갱신하는데 사용됨

▼ TCP / UDP

- TCP
    - 연결지향적. 연결 미리 설정하고 신뢰성 있는 데이터 전송 보장
    - 신뢰성. 패킷 오류나면 다시 보냄

        - 흐름 제어와 혼잡 제어. 흐름 조절하고 네트워크 혼잡 방지하기 위해 흐름 제어 및 혼잡 제어 메커니즘 사용함
        - 순차적 데이터 전송. 패킷 순서 보장해 데이터 원래 순서대로 수신함
        - 상대적으로 높은 지연. 연결 설정 및 신뢰성을 위한 추가 작업 때문에 지연 발생할 수 있음
    - UDP
        - 비연결 지향적. 즉시 전송함
        - 신뢰성 없음. 재전송 안함
        - 흐름 제어나 혼잡 제어 지원 안 함
        - 높은 성능. 간단하고 빨라서
        - 다중 캐스트와 브로드캐스트 지원 : 하나의 송신자가 여러개의 수신자에게 데이터 전송 가능

▼ 리스트뷰 / 리사이클러뷰 차이

리스트뷰는 무한대로 뷰 생성

리사이클러뷰는 처음 만들어지는 개수는 정해져있음. 뷰홀더를 이용해 뷰를 재사용

메모리에 좀 더 나은 선택

▼ Android 에서 Context란?

- 안드로이드 시스템이 만들어놓은 ID. 앱 또는 컴포넌트를 관리하기 위함. 현재 내가 사용하고 있는 앱의 환경 정보에 접근 가능한 수단
    - Application context : 어플리케이션이 살아있는 동안 살아있는 컨텍스트
        - 토스트를 제외하고, 다이얼로그를 띄우는 등 GUI 와 관련된 작업은 못함
    - Activity context : 액티비티가 살아있는 동안 살아있는 컨텍스트
        - UI 작업 가능. 다이얼로그 띄우는 등!

▼ Restful API

- Representational State Transfer : API 작동 방식에 조건을 부과하는 아키텍쳐
    - 자원 / 표현 / 으로 구분하여, 해당 자원의 상태 정보를 주고받는것
- URI로 자원을 명시, method로 자원에 대한 CRUD 적용
- GET/POST/PUT/PATCH/DELETE

- 서버 / 클라이언트 구조, 무상태, 캐시 처리 가능, 계층화, 인터페이스 일관성을 가진다

▼ retrofit에서 @body와 @field의 차이점은?

- body는 json 형태
- field는 데이터를 formurlencoded하는 방식. (hashmap)

▼ JetPack

- 안드로이드 앱을 쉽고 빠르게 구축하는데 도움이 되는 도구들의 모음.
- androidX가 jetPack의 라이브러리를 묶은 패키지명
- 뷰모델, 뷰바인딩, 네비게이션, 룸, 페이징 등.. 많음

▼ **안드로이드 URI 스킴, 유니버설 링크, 딥링크***

딥링크란? 앱에서 링크에 해당하는 컨텐츠로 이동시킬 수 있는 링크

- URI 스킴
  - {scheme}://{path}?xxxx=1234 과 같은 패턴
  - 별다른 제약 없이 직접 원하는 스킴을 정해서 만들 수 있음
  - `kakaotalk://me`
  - `heydealer://register?car=xxxx&action=review_list`

  단점

  1. 앱이 설치되어있을때만 실행할 수 있음.
  2. 원하는 스킴을 아무나 만들 수 있어서, 같은 스킴을 사용하는 경우가 생김
  3. 같은 스킴일 경우, android는 어떤 앱으로 열지 선택하는 팝업이 뜨고, ios는 마지막으로 설치한 앱이 열림

  → 같은 스킴을 정의하면 악의적으로 내 앱을 열게 할 수도 있는것
- 앱링크 / 유니버셜 링크
  - 항상 웹사이트 형태의 딥링크 url로 만들어짐
  - https://www.heydealer.com/chat
    https://deeplink.heydealer.com/cars/xxx
    https://heydealer.com/events?id=123
  - 고유한 웹사이트 주소(도메인)으로 만들어지기때문에 항상 유일하다.
  - 이 도메인이 이 앱의 소유자라는것을 인증해줘야 제대로 동작한다.

    - 소유자 인증 : 해당 도메인이 앱의 고유 id 정보가 포함되어있는 파일을 업로드하도록 하고, 해당 파일을 확인해서 인증처리하는 방식
        - android에서는 딥링크 만들고, google search console에 인증함

    동작 방법

    1. 앱이 설치되어있다면 화면 실행됨
    2. 설치되어있지않다면, 웹 링크 열려고 시도 → 웹페이지 있다면 열림
    3. 아니라면 앱마켓 열도록 처리하면 됨

    단점

    1. 브라우저 주소 입력창에 딥링크 입력하면 동작하지 않음
    2. 안되는 브라우저도 있음

- Deffered DeepLink
    - 앱이 설치되어있으면 바로 열리면서 해당 컨텐츠 실행
    - 앱이 설치되어있지 않으면 동작을 지연시켰다가 앱 설치 이후 실행되었을때 해당 컨텐츠를 실행
    - 이걸 지원하는건 firebase dynamic deeplink 사용중이나.... 바꿔야함. 곧 종료됨

▼ 안드로이드 스타일

- *스타일*은 특정 뷰 유형의 속성을 지정. 예를 들어, 특정한 하나의 스타일을 사용해 버튼 속성을 지정하는 것
- *테마*는 스타일, 레이아웃, 위젯 등으로 참조할 수 있는 명명된 리소스 모음을 정의. 테마는 `colorPrimary`와 같은 시맨틱 이름을 Android 리소스에 할당.

스타일 및 테마는 함께 작동하도록 설계됨.

▼ viewholder pattern이란

- 뷰 객체를 뷰홀더에 보관함으로써 findViewById()(뷰를 찾을때마다)와 같은 반복적 호출 메서드를 줄여 효과적으로 속도를 개선할 수 있는 패턴

```
override fun onCreateViewHolder(parent: ViewGroup, viewTyp
        val inflater = LayoutInflater.from(parent.context)
        val binding = ItemAddressListBinding.inflate(infla
        return ViewHolder(binding)
    }
```

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

    override fun onBindViewHolder(holder: ViewHolder, posi
        holder.bind(itemList[position])
    }

    inner class ViewHolder(
        private val binding: ItemAddressListBinding
    ) : RecyclerView.ViewHolder(binding.root) {

        fun bind(item: AddressListLayout) {
            binding.tvListRoad.text = item.road
            binding.tvListAddress.text = item.address
            binding.addressItemLayout.setOnClickListener {
                itemClickListener.onItemClick(it, position
            }
        }
    }
```

▼ 인텐트 종류

- 명시적 인텐트 : 작업을 수행하기 위한 컴포넌트를 정확히 지목하는것.
  - ex) startActivity 같은것

```
val intent = Intent(this@TestActivity, MainActivity::cl
```

- 암시적 인텐트
  - 해당 작업을 할 수 있는 컴포넌트 전부에게 전달함.
  - 주로 공유하기 버튼을 누를때 많이 사용됨.
  - 해당 URI를 띄울 수 있는 앱들이 하단에 나타나며 사용자가 고르는 것

```
val intent = Intent(Intent.ACTION_VIEW, Uri.parse("http
```

- intent-filter

- 앱에 Intent가 전달되었을 때 Intent Filter에 기술된 조건에 맞는 Intent만 통과시킴 (말 그대로 필터링)

```
ex) A 앱에서 Intent에 ACTION_VIEW, CATEGORY_BROWSABLE를 설정한 후 전송하면?

- 암시적 인텐트이므로 CATEGORY_DEFAULT가 자동 추가된 후 전송

- 설치된 앱들 중 Intent Filter에 ACTION_VIEW, CATEGORY_BROWSABLE,
  CATEGORY_DEFAULT가 정의된 앱들의 목록이 등장함
  (해당 앱들의 Filter가 Intent를 통과시킴)

- 목록에서 앱을 선택하면 해당 앱으로 Intent 전달하며 컴포넌트를 실행함
```

▼ 인플레이션이란

- xml 레이아웃의 내용이 메모리에 객체화되는 과정

▼ 데이터 바인딩 / 뷰바인딩

- 데이터 바인딩 : 두개의 데이터 소스를 함께 연결하고 동기화 상태를 유지하는 일반적인 기술
  - 단방향 : event로 흐름이 이루어지고, 뷰가 바로 업데이트되지않음
  - 양방향 : 데이터 변경에 따라, 프레임워크를 통해 양방향으로 이루어지며, 뷰 즉각 업데이트됨

  단점

  1. 강결합된 코드베이스 : 디버깅을 어렵게함.
  2. 관심사의 분리가 안됨 : UI와 데이터를 조작하는 코드가 섞여있음
  3. 생성된 코드이기때문에, 디버깅이 어려움. 생성된 코드는 프로젝트에서 보이지도 않아서, 발생하는 문제를 추적하기 어려움
- 뷰바인딩 : xml 파일을 해석해서 객체로 변환함. 자동으로 형면환을 해줌(텍스트뷰는 텍스트뷰로). 연결된 레이아웃에 존재하는 뷰 그냥 가져오면됨!
  - 자동으로 뷰에 대한 커넥션을 생성해주는 기능. binding 파일도 바로 찾을 수 있고, binding. 으로 뷰들에도 바로 접근이 가능

▼ RxJava / RxKotlin

- Reactivie Programming : 주변환경과 끊임없이 상호작용하면서, 환경이 변하면 이벤트를 받아 동작하는 프로그래밍
- 비동기 및 이벤트 기반 프로그래밍을 하기 위한 라이브러리.
- 옵저버 패턴 사용
- 관찰 가능한 시퀀스를 사용함

> 💡 **Rxjava**
>
> 메인라이브러리
>
> **RxKotlin**
>
> 코틀린에서 필요한 추가적인 기능 제공(함수형 프로그래밍)
>
> **RxAndroid**
>
> 안드로이드에서의 쓰레드를 Rxjava에서 사용하는 스케줄러와 연동하기 위해 사용
>
> **RxBinding**
>
> Edittext 등의 컴포넌트를 옵저버블 형태로 만들어 주는 것

▼ 코루틴이란

- 협력형 멀티 태스킹
- 여러개의 진입점과 탈출점을 지님
- 동시성을 프로그래밍

> 💡 동시성 프로그래밍이란 오른쪽 손에만 펜을 쥐고서 왼쪽 도화지에 사람 일부를 조금 그리고, 오른쪽 도화지에 가서 잠시 또 사람을 그리고, 다시 왼쪽 도화지에 사람을 찔끔 그리고... 이 행위를 아주 빨리 반복하는 것이다. 사실 내가 쥔 펜은 한 순간에 하나의 도화지에만 닿는다. 그러나 이 행위를 멀리서 본다면 마치 동시에 그림이 그려지고 있는 것 처럼 보일 것이다. 이것이 동시성 프로그래밍이다.

💡 위의 코드에서는 메인 쓰레드에 코루틴이 두 개가 있다. 하나는 왼쪽 도화지에 그림을 그리는 코드고 다른 하나는 오른쪽 도화지에 그림을 그리는 코드다. 메인 쓰레드가 실행되면서 먼저 왼쪽 코루틴인 `drawPersonToPaperA()`라는 함수를 만났다고 가정해보자. 해당 함수는 가상 코루틴 빌더인 `startCoroutine {}` 블럭으로 인해 코루틴이 되고, 함수를 중간에 나갔다가 다시 들어올 수 있는 힘을 얻게된다. `drawPersonToPaperA()`가 호출되어 `suspend` 함수인 `drawHead()`를 만나게 되면 이 코루틴을 잠시 빠져나간다.

왼쪽 코루틴을 빠져나갔지만 그렇다고 메인쓰레드가 가만히 놀고있진 않는다. 다른 `suspend` 함수들을 찾거나 resume되어지는 다른 코드들을 찾는다. 왼쪽 코루틴의 경우 2초 동안 `drawHead()` 작업을 하게된다. 그러나 `delay(2000)`는 쓰레드를 블락시키지 않으므로 다른 일들을 할 수가 있다. 뿐만 아니라 `drawHead()`함수 안에서 다른 쓰레드를 실행시킨다면 병행적으로도 실행이 가능하다. 왼쪽 코루틴을 빠져나온 쓰레드가 오른쪽 코루틴을 만나게 되어 또 한번 `suspend` 함수를 만나게 되면 아까 도화지 그림에서 설명한 것과 같은 현상이 일어난다. 아까 오른손에 펜을 쥐고 왼쪽과 오른쪽 도화지를 아주빠르게 왔다 갔다 하면서 그림을 그리는 것 같은 셈이다. **이렇게 코루틴을 사용하여 쓰레드 하나에서 동시성 프로그래밍이 가능하다**

▼ Coroutine vs Rxjava

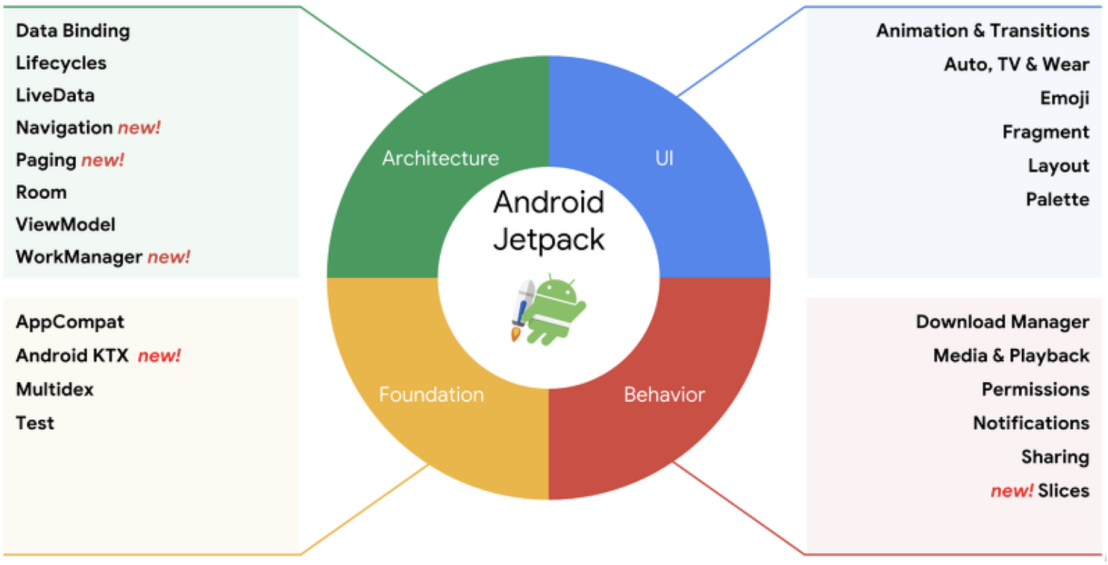
이 글 참고

▼ pending intent

- PendingIntent 동작 -> 다른 프로세스(앱) 권한을 허가 -> 본인 앱에서 Intent를 실행하는 것처럼 사용하게 하는 것

▼ AAC란?

- 안드로이드 아키텍쳐 컴포넌트. 테스트와 유지보수가 쉬운 앱을 디자인하도록 돕는 라이브러리의 모음.



저기서 초록색...

▼ Retrofit Interceptor / Converter

- interceptor : 서버로부터 데이터 송수신할때 가로채서 어떤 처리를 해주는 것. 별도의 파싱 없이 처리 가능

```
private val client = OkHttpClient.Builder()
    .addNetworkInterceptor(commonNetworkInterceptor)
    .build()

private val retrofit = Retrofit.Builder()
    .baseUrl(BASE_URL)
    .client(client)
    .addConverterFactory(GsonConverterFactory.create())
    .build()
```

- converter : 기본적으로 Retrofit은 HTTP 요청 본문을 Okhttp의 ResponseBody 형식과 @Body에 이용하는 RequestBody 타입만 역직렬화 하는것.
    - 컨버터를 통해 이외의 형식들을 변환가능. 그 종류에는 Gson, Jackson, Moshi, Protobuf 등등

▼ Android 위험 권한

    - 런타임 권한
    - 개인정보 또는 개인정보를 만들어낼 수 있는 단말의 주요 장치에 접근할 때 부여
    - 위치 / 카메라 / 마이크 / 연락처/ 전화 / 문자 / 일정 / 센서 등

▼ companion object 와 object의 차이점

- compaion obj는 클래스에서 단일을 보장
- object는 전체 어플리케이션에서 단일을 보장. constant같은것을 정의하는데 사용

- Compose 최적화 하는 방법 (사례로 들기)

코루틴 딥다이브

1. Android의 Kotlin 코루틴은 무엇이며 기존 스레딩과 어떻게 다릅니까?

2. 코루틴 맥락에서 "suspending functions"의 개념을 설명할 수 있습니까?

3. 코루틴은 Android 애플리케이션에서 동시성 및 병렬성을 관리하는 데 어떻게 도움이 됩니까?

4. 시작, 비동기 및 runBlocking 코루틴 빌더 간의 차이점을 설명하십시오.

5. 코루틴 관리에서 CoroutineScope 및 CoroutineContext의 역할은 무엇입니까?

6. 구조화된 동시성이 Android 코루틴에서 작동하는 방식을 설명합니다.

7. 작업의 목적은 무엇이며 코루틴의 수명 주기를 관리하는 데 어떻게 사용할 수 있습니까?

8. Dispatchers.Main, Dispatchers.IO 및 Dispatchers.Default의 차이점을 설명하십시오.

9. Kotlin 코루틴에서 예외를 어떻게 처리할 수 있습니까?

10. CoroutineDispatcher는 무엇이며 코루틴 실행을 관리하는 데 왜 중요합니까?

11. "withContext" 기능은 어떻게 작동하며 언제 사용해야 합니까?

12. Android 코루틴에서 "flow"를 사용하면 어떤 이점이 있으며 간단한 suspending function 사용하는 것과 어떻게 다릅니까?

13. Kotlin 코루틴에서 "StateFlow" 및 "SharedFlow"의 역할을 설명합니다.

14. "callbackFlow"와 "channelFlow"의 차이점은 무엇입니까?

15. Kotlin 코루틴의 "cold" 및 "hot" 흐름 개념을 설명할 수 있습니까?

16. Flow와 관련하여 "buffer"는 어떻게 작동하며 언제 사용해야 합니까?

17. "flowOn"이란 무엇이며 Flow의 실행 컨텍스트에 어떤 영향을 줍니까?

18. Flow로 작업할 때 "collect"와 "onEach"의 차이점을 설명하십시오.

19. Android 코루틴에서 배압 처리를 어떻게 구현할 수 있습니까?

20. 코루틴에서 리소스를 관리하는 모범 사례는 무엇입니까?

21. 코루틴은 RxJava와 성능 및 유용성 측면에서 어떻게 비교됩니까?

22. Flows에서 "combine" 및 "zip" 연산자의 목적과 사용법을 설명하십시오.

23. Kotlin 코루틴에서 "액터"는 무엇이며 동시성 관리에 어떻게 사용됩니까?

24. "supervisorScope"는 "coroutineScope"와 어떻게 다른가요?

25. Kotlin 코루틴의 "Mutex" 개념을 설명하고 사용 사례를 제공합니다.

26. 네트워크 호출을 위해 Retrofit과 함께 코루틴을 어떻게 사용할 수 있습니까?

27. Kotlin 코루틴에서 "debounce"의 역할은 무엇이며 성능을 최적화하는 데 어떻게 사용할 수 있습니까?

28. Android 애플리케이션에서 코루틴과 흐름을 어떻게 테스트할 수 있습니까?

29. Kotlin 코루틴에서 재시도 및 시간 초과를 처리하는 가장 좋은 방법은 무엇입니까?

30. 실행 중인 코루틴을 취소하는 방법과 하위 코루틴에 미치는 영향을 설명하십시오.

31. 코루틴을 사용하여 어떻게 병렬 분해를 구현할 수 있습니까?

32. Kotlin 코루틴에서 "select" 표현식의 목적은 무엇입니까?

33. Room Database 및 LiveData와 함께 코루틴을 어떻게 사용할 수 있습니까?

34. CoroutineStart의 개념과 다양한 옵션에 대해 설명하십시오.

35. CoroutineName과 CoroutineExceptionHandler는 무엇이며 왜 중요한가요?

36. 흐름으로 작업할 때 "launchIn"과 "collectIn"의 차이점을 설명하십시오.

37. 코루틴은 Android 애플리케이션에서 백그라운드 작업 및 UI 업데이트를 관리하는 데 어떻게 도움이 됩니까?

38. "callbackFlow"란 무엇이며 콜백을 흐름으로 변환하는 데 어떻게 사용할 수 있습니까?

39. Kotlin 코루틴의 "동시성" 및 "병렬성" 개념을 설명하십시오.

40. "coroutineScope"는 무엇이며 코루틴 계층 구조를 관리하는 데 왜 중요합니까?

41. Kotlin 코루틴은 Java의 CompletableFuture 및 기타 동시성 모델과 어떻게 비교됩니까?

42. Kotlin 코루틴에서 "reduce" 및 "fold" 작업의 역할을 설명하십시오.

43. Kotlin 코루틴에서 "transform" 및 "transformLatest" 연산자를 사용하면 어떤 이점이 있습니까?

44. Kotlin 코루틴을 사용하여 스로틀링 메커니즘을 어떻게 구현할 수 있습니까?

45. Flow 변환에서 "emit" 및 "emitAll"의 역할을 설명하십시오.

46. Kotlin 코루틴의 "flatMapConcat", "flatMapMerge" 및 "flatMapLatest"는 무엇이며 서로 어떻게 다릅니까?

47. 실시간 업데이트 및 데이터 관리를 위해 Firebase API와 함께 Kotlin 코루틴을 어떻게 사용할 수 있습니까?

48. 채널을 사용하는 Kotlin 코루틴의 "produce" 및 "consume" 개념을 설명하십시오.

49. 코루틴 취소 및 리소스 정리를 처리하는 모범 사례는 무엇입니까?

50. Kotlin 코루틴을 사용하여 Android 애플리케이션에서 MVI(Model-View-Intent) 아키텍처를 구현하려면 어떻게 해야 합니까?

추가

▼ **성능 측면에서 스레드는 코루틴과 어떻게 비교되는지**

일반적으로 코루틴은 CPU 사용량 측면에서 더 효율적인 경향이 있으며, I/O 바인딩 작업 측면에서는 스레드가 더 나을 수 있습니다.

▼ **정지 기능은 기본적으로 메인 스레드에서 실행되는지, 아니면 다른 스레드에서 실행?**

기본적으로 일시 중지 기능은 백그라운드 스레드에서 실행

경험 및 가치관 질문

- 본인이 직무에 적합한 이유
- 맡고싶은 분야?
- 왜 안드로이드?

- 경력사항중 회사에 기여할 수 있는 부분?
- 성격의 장단점은? (사례도 함께 들을것)
- 가장 자랑하고싶은 경험
- 가장 큰 실패
- 가장 집요하게 파고든거
- 가장 가치있다고 생각하는 경험