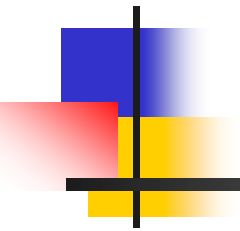
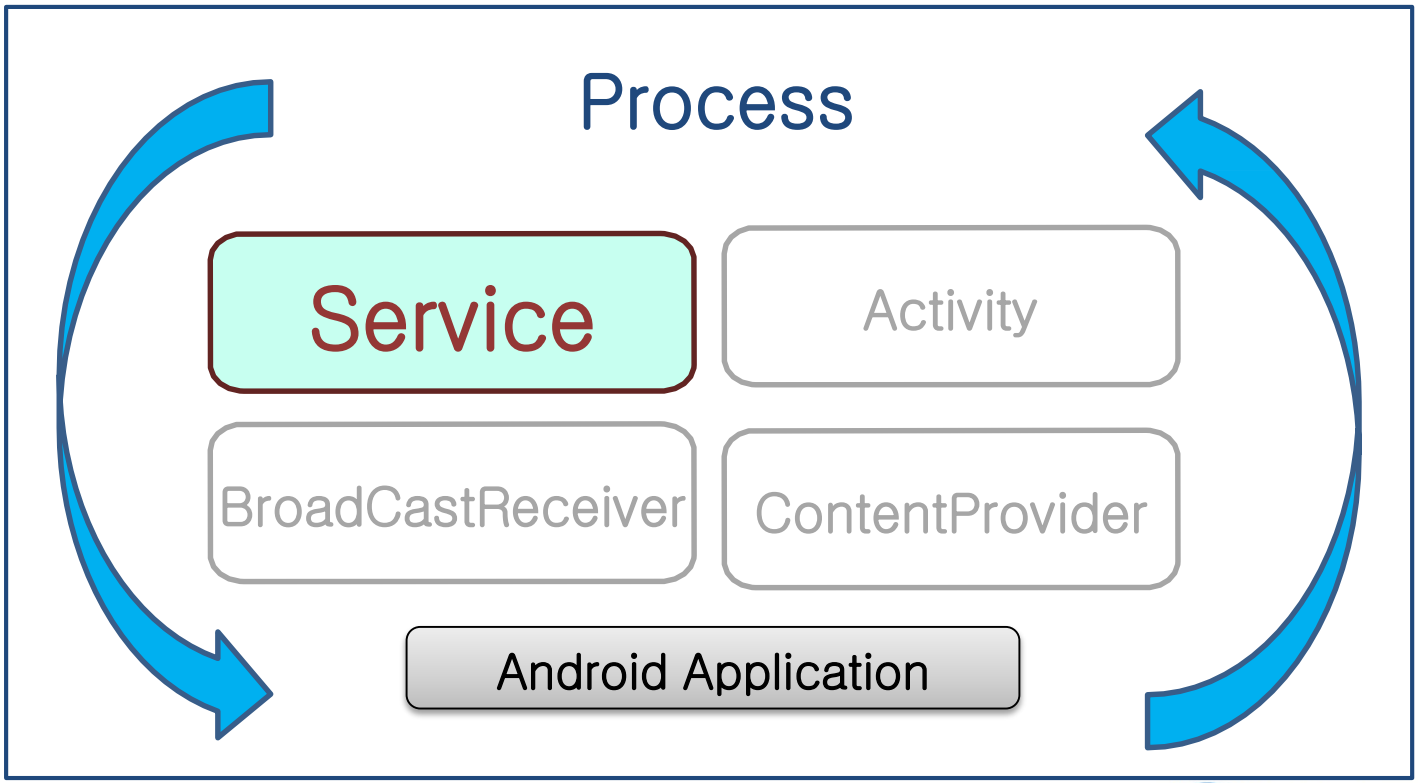


Android Service

배 희호 교수
경복대학교
소프트웨어융합과

Service

- Service는 Android Application을 구성하는 4가지 Component중 하나임
- Activity처럼 사용자와 상호작용 하는 Component가 아니고, Background(화면 뒷단)에서 동작하는 Component를 말함

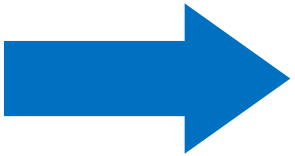


Service

- Android를 개발하면서 아마도 가장 많이 사용되는 Component를 Activity로 알고 있겠지만, 사실 System 측면에서는 Service 임
- Android System 자체가 대부분 Service로 이뤄져 있고, 심지어 Activity가 돌아가기 위해서도 Service를 사용하기 때문 임
- 하지만 우리는 이러한 Service가 존재하는 것을 전혀 인지하지 못한다. 왜일까?
  - 바로 눈에 보이지 않고 Background에서 동작하기 때문임
  - Service는 Activity와 달리 화면을 가지지 않고 오직 Background에서만 동작
- Activity Application은 화면(Activity)이 종료되면 동작하지 않지만 Service는 Background에서 실행되므로 화면과 상관없이 계속 동작함

Service

- 예



Service

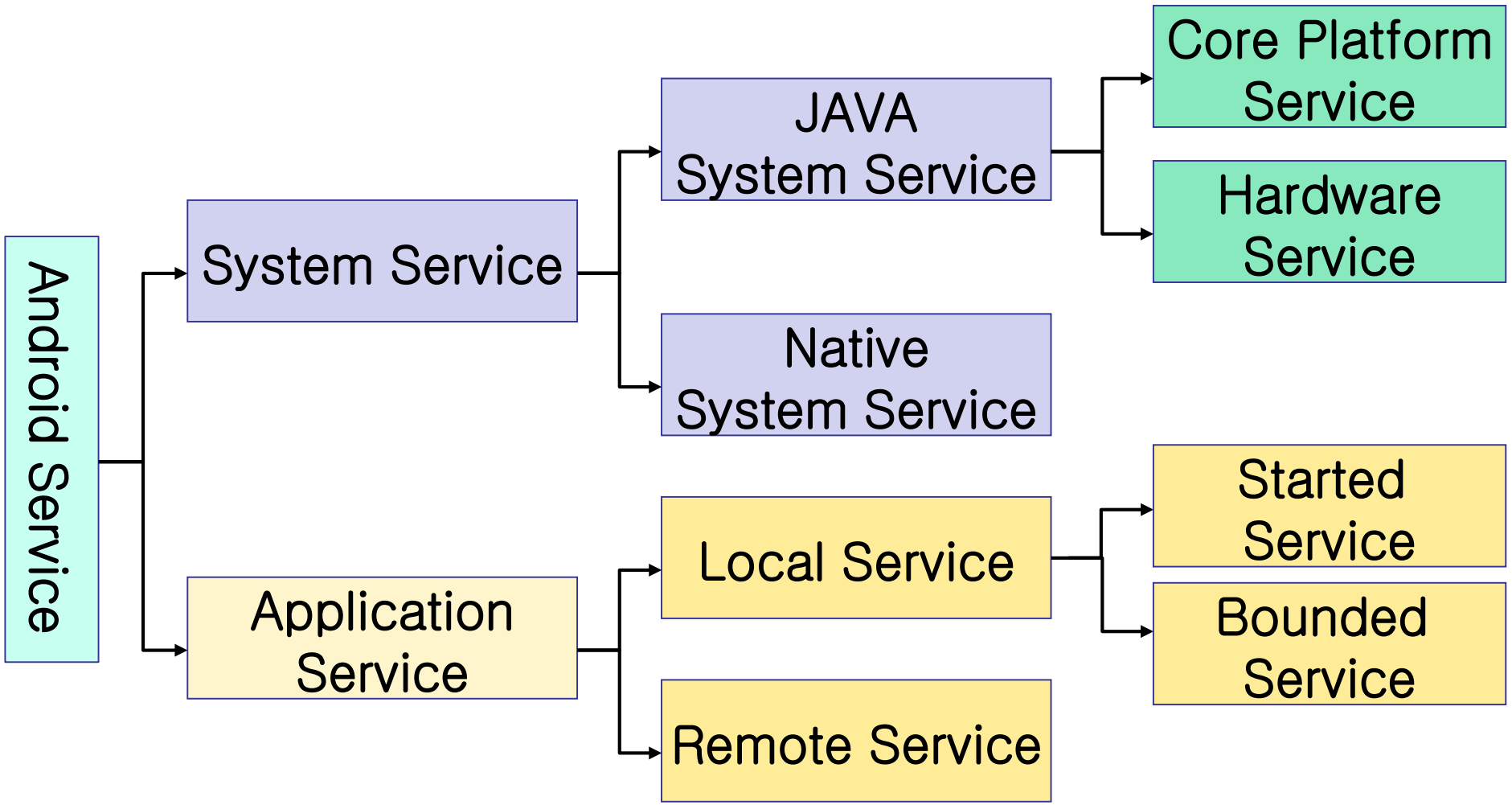
- Service 활용사례
  - 별도의 Thread를 통해서 사용 가능
    - File Download
    - Media Player
  - Android Software Stack 내의 Service 사용
    - Location Manager
    - Media Controller
    - Alarm Manager

Service

- Service 사용 이유
  - Activity가 종료되어도 계속해서 동작하게 하기 위하여 사용
  - Application 개발에 필요한 중요 API를 System Service로 지원

Service

- Service 분류



System Service

- Android Framework가 제공하는 기본 Background Service로, 운영체제가 관리하며 다양한 Hardware 및 System 기능에 접근하기 위해 사용
  - 예) Location Service, Notification Service, Power Manager, Sensor Service 등
- System Service는 App 개발자가 직접 Access하거나 사용할 수 있도록 System API를 통해 노출되며, Android System의 전반적인 운영을 지원하는 핵심 구성 요소임
- System Service는 JAVA 영역에서 동작하는 “JAVA System Service"와 C/C++ 영역에서 동작하는 “Native System Service"로 나뉨

System Service

- 특징
  - System 수준에서 실행
    - Android System에서 시작되고, Lifecycle은 운영체제가 관리(Application이 생성하고, 관리하지 않음)
    - System Service는 Android OS의 핵심 기능(예: 전원 관리, 위치 서비스, 네트워크 관리 등)을 제공하기 위해 설계되었음
    - 이러한 Service는 Framework Layer에 통합되어 있으며, Application이 사용할 수 있는 고수준 API를 제공
  - 항상 실행 중
    - Android 기기의 Booting 시 활성화되며, System 수준에서 항상 실행됨
    - 사용자가 직접 제어할 수 없으며, 운영 체제의 자원을 사용해 동작함

System Service

- 특징
  - 통신
    - System Service는 Client(App)와 Binder IPC(Inter-Process Communication) Mechanism을 사용하여 Data를 주고받음
    - Client와의 통신은 System에서 보안과 성능을 보장
  - 보안
    - System Service에 접근하려면 Permission이 필요
    - Context.getSystemService(String serviceName) 메소드를 통해 액세스함
    - 예) 위치 서비스에 접근하려면 ACCESS_FINE_LOCATION 권한이 필요

System Service

서비스 이름	역할 및 기능
ActivityManager	앱의 수명 주기 관리, 백그라운드 작업 스케줄링
PowerManager	전원 관리, 화면 켜기/끄기, CPU 절전 모드 제어
AlarmManager	지정된 시간에 작업 예약
NotificationManager	알림(Notification) 표시 및 관리
ConnectivityManager	네트워크 연결 관리(Wi-Fi, 모바일 데이터 등)
LocationManager	GPS 및 네트워크 기반 위치 서비스 제공
SensorManager	장치의 센서(가속도계, 자이로스코프 등) 데이터 접근 및 관리
TelephonyManager	전화, SMS, SIM 카드 정보 제공
InputMethodManager	키보드 입력 및 IME(Input Method Editor) 관리
AudioManager	오디오 볼륨 및 스트림 관리
WindowManager	창(Window) 및 화면 레이아웃 관리
ClipboardManager	클립보드 데이터 관리
Vibrator	진동 제어 및 API 제공
PackageManager	설치된 앱 및 권한 관리
WifiManager	Wi-Fi 연결 관리 및 상태 확인
DownloadManager	파일 다운로드 작업 관리

Application Service

- Application 개발자가 Application에서 특정 작업 (Background 작업 등)을 수행하기 위해 정의하고 구현하는 Service
  - 예) 음악 재생, Data 동기화, File Download 등
- 특징
  - Application 개발자가 Service 클래스를 상속해서 구현
  - 개발자가 직접 정의하고 Lifecycle을 관리
  - Activity와 동일 Process(다시 말해 동일 Package 내에서)에서 Service가 실행되는 “Local Service"와 Activity와 다른 Process에서 Service가 실행되는 “Remote Service"로 구분

Application Service

- Local Service
  - Application 내에서만 실행되며, 동일한 Process에서 동작하는 Service
  - 주로 Application의 내부 Component(Activity, Fragment 등)와 통신하기 위해 사용
  - Started Service와 Bound Service로 나뉨
  - 특징
    - 같은 Process에서 실행되므로 IPC(Inter-Process Communication)가 필요하지 않음
    - 구현이 간단하고 성능이 좋음
    - Service와 Component 간의 Data 공유 및 호출이 직접적으로 이루어 짐
    - 자신을 생성한 Activity가 종료되면 종료

Application Service

- Remote Service
  - 다른 Application이나 Process에서 동작하며, Inter-Process Communication (IPC)를 통해 접근할 수 있는 Service
  - 주로 다른 App이나 Process와의 통신이 필요한 경우 사용
  - 특징
    - 별도의 Process에서 실행되므로 성능 Overhead가 발생할 수 있음
    - Data를 주고받기 위해 AIDL(Android Interface Definition Language) 또는 Messenger를 사용해야 함
    - Process가 다르기 때문에 Data 직렬화가 필요
    - 독립적인 Process로 동작하기 때문에 Main Application이 종료되어도 동작하지만, System 자원을 계속 사용하기 때문에 주의해야 함

Application Service

- Started Service(Background Demon, Unbounded Service)
  - 배경에서 계속 실행되는 Process
  - 한 번 시작되면, 명시적으로 중단 (stopSelf() 또는 stopService())하지 않는 한 계속 실행
  - Activity가 startService() 메소드를 호출하여서 Service를 시작
  - 예) 노래 재생
- Bounded Service (원격 호출 Interface)
  - Component(예: Activity)와 Client-Server Model로 Binding
  - 자신의 기능을 메소드로 노출시켜 특정한 기능을 제공
  - Client가 연결을 해제하면 Service가 중단됨
  - Activity가 bindService() 메소드를 호출하여서 Service를 시작
  - 예) COM, CORBA

Application Service

- Intent Service (Started Service의 일종)
  - Background에서 단일 작업을 비동기적으로 처리하도록 설계된 Android에 특화된 Service
  - Client가 여러 작업을 요청해도 Job Queue에 추가되어 순차적으로 처리
  - 작업 완료 후 Service가 자동으로 종료
  - 별도의 Thread에서 실행되므로 UI Thread에 영향을 주지 않음
  - 작업 수행 중 ANR(Application Not Responding)을 방지

Application Service

- IntentService의 주요 메소드

메소드	설명	호출 시점
onCreate()	Service 초기화 작업을 수행 예) Resource 준비, 초기 설정	Service가 처음 생성될 때 한 번 호출
onStartCommand()	Client에서 startService()를 호출할 때 호출. 기본적으로 onHandleIntent()로 작업 전달	Service가 시작될 때 호출 여러 번 호출될 수 있음
onHandleIntent (Intent intent)	Background Thread에서 실행되는 Main 작업 처리 메소드 여기서 비동기 작업 수행	각 Client 요청(Intent)을 처리할 때 호출
onDestroy()	Service가 종료될 때 호출 Resource 정리와 같은 종료 작업 수행	모든 작업이 완료되고 Service가 종료될 때 호출
onBind (Intent intent)	Client가 Bind할 때 호출 IntentService에서는 기본적으로 null을 반환	Client가 bindService()를 호출할 때 호출 일반적으로 IntentService에서는 사용되지 않음

Application Service

- IntentService와 일반 Service의 차이

특징	IntentService	일반 Service
Thread 처리	별도의 작업 Thread에서 실행	기본적으로 UI Thread에서 실행
작업 Queue	요청을 Queue에 추가하고 순차적으로 처리	동시 작업 처리 지원 (직접 구현 필요)
자동 종료	모든 작업 완료 후 자동 종료	명시적으로 종료 호출 필요
주요 사용 목적	비동기 작업 간소화	복잡한 Background 작업 관리

Application Service

- Foreground Service
  - 사용자가 명확히 인식할 수 있도록 실행되는 Service로, 주로 Background에서 중요한 작업을 수행하면서 알림(Notification)을 통해 사용자에게 진행 상태를 표시
  - Android 8.0 (API 26) 이상에서는 Background Service의 제한이 도입되면서 Foreground Service는 필수적인 Service 유형 임

Application Service

- Foreground Service의 주요 특징
  - Notification 필수
    - Foreground Service는 반드시 지속 알림(persistent notification)을 통해 사용자에게 노출
    - 알림은 Service가 실행되는 동안 계속 표시됨
  - Background 제한 우회
    - Android 8.0 이상에서는 Background에서 장시간 실행되는 작업은 Foreground Service로 수행해야 함
  - 사용자 친화적
    - 사용자에게 작업 상태를 알림으로 제공하므로 중요한 작업에 적합
  - 장시간 작업에 적합
    - File Download, Media 재생, GPS 추적과 같은 작업에 자주 사용

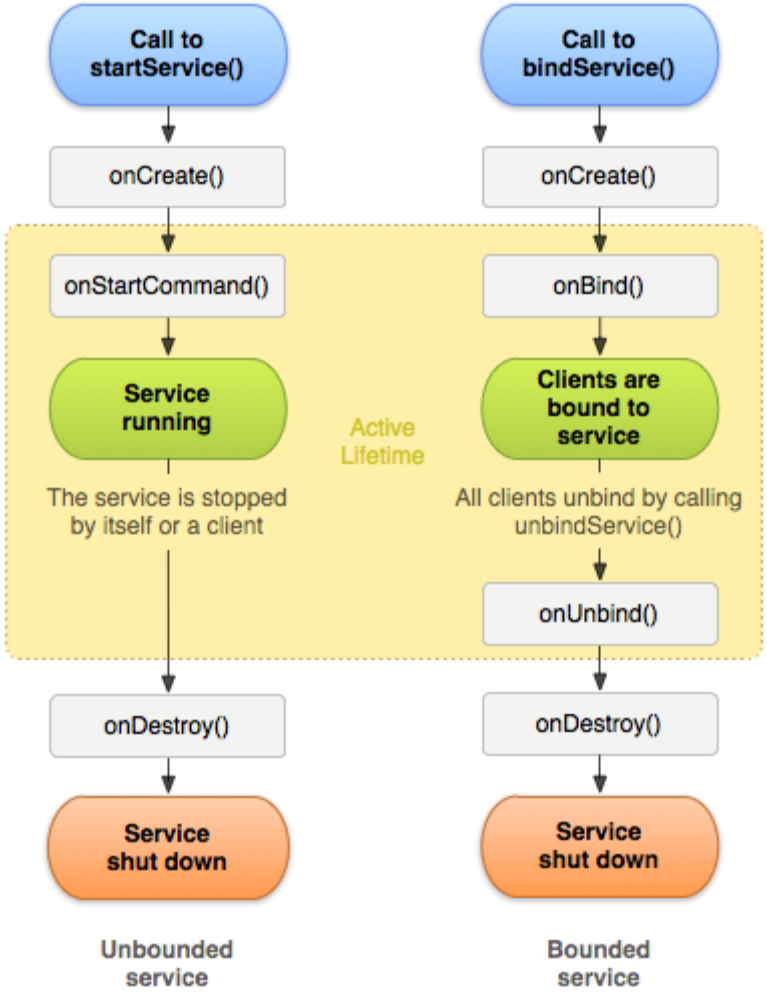
Application Service

- Android SDK의 Service 클래스를 확장한 것으로, UI없이 주기적으로 특정한 일을 수행하는 Background Process를 가르킴

- Application Service는 다음의 2가지를 제공
  - Service 시작, 종료
  - Binding을 통한 Service 메소드 호출
  - Application에서 Service를 생성할 때 단순히 Background Job을 실행 시키는 경우와 Binding을 통해 메소드를 이용하는 경우에 따라서 startService(), bindService() API를 이용
  - 두 경우에 따라 Lifecycle이 달라짐

Application Service

- LifeCycle



Application Service

- Service = Deamon = Background Program
- Background에서 실행되는 정해지지 않은 시간 동안 항상 동작하는 Process
- 사용자와 직접 상호작용은 하지 않음으로 Visual한 사용자 Interface가 없음 (UI가 없음)
- 사용자의 입력과는 무관하게 지속적인 처리나 규칙적인 처리, 또는 Event 처리를 수행하는 작업의 경우에 Service를 이용하면 좋음
  - 예) MP3 Player, Chatting Program, File Download 등…
- 사용자와 통신할 수 있는 방법이 필요
  - Notification
- Service는 일반적으로 Application에 의하여 시작됨
- Activity와 연결되어 있지 않아도 끊임없이 실행이 됨
- 연산이나 메소드 등의 Service를 제공하는 것이 주 임무

Application Service

- 사용자 인터페이스(UI) 없이 Background에서 실행되는 Component
  - 배경 음악을 재생
  - Web Site에서 주기적으로 Data를 읽음
  - 주기적으로 Phone의 사용량을 계산
  - Application의 Update를 주기적으로 검사

Application Service

- Android Service의 시작
  - 수동으로 시작하기
    - API를 호출
  - 자동으로 시작하기
    - IPC(Process간의 통신)를 사용

- Android Service의 주의사항
  - Service는 더 이상 필요 없을 때까지, 또는 Memory를 더 많이 확보해야 할 때까지 실행
  - 이로 인해 CPU와 Network(무선 랜)의 사용량이 많아져서 Battery의 사용시간이 줄어들 수 있음

Application Service

- Service 구현 과정
  - Android SDK에서 제공하는 Service용 기본 클래스 상속
  - 새로운 Service를 구현하는데 필요한 메소드를 Override
  - AndroidManifest.xml File에 적절한 내용을 넣어 Android System과 연동
  - Android Service 클래스의 구현은 Activity의 구현과 비슷함

Application Service

- Service 만들기
  - IBinder onBind(Intent intent)
    - 다른 Component가 bindService()를 호출해서 Service와 연결을 시도하면 이 메소드가 호출
    - 이 메소드에서 IBinder를 반환해서 Service와 Component가 통신하는데 사용하는 Interface를 제공해야 함
    - Started Service를 구현한다면 null을 반환하면 됨
    - 이 메소드에서 Service와 Activity간에 연결을 설정하여 Activity에서 Service안의 메소드를 호출할 수 있게 함

Application Service

- Service 만들기
  - void onDestory()
    - stopService()가 호출되거나 stopSelf()가 호출되었을 때 수행되는 메소드
    - 보통 사용했던 자원들을 해제하는 작업을 함
    - 주로 Thread, Listener, BroadcastReceiver와 같은 자원들을 정리하는데 사용하면 됨
    - TaskKiller에 의해 Service가 강제 종료될 경우에는 이 메소드가 호출되지 않음

Application Service

- 새로운 Service를 만들고 난 뒤에는 이를 ApplicationManifest.xml File에 등록해야 함

```
<service
    android:name=“.MyService”
    android:enabled=“true”
    android:exported=“false”/>
```

  - name 속성 : Service 클래스의 이름 지정
  - enabled 속성 : Service 사용 여부
    - enabled="true"를 추가해주면 Android System이 자동적으로 Service를 시작하게 됨
  - exported 속성 : 외부에서 Service 제어
    - exported= "false"로 지정함으로써 해당 Service를 자신의 App에서만 사용할 수 있고, 외부의 다른 Application이 현재 사용되고 있는Service를 강제로 종료할 수 없게 할 수 있음

Application Service

- 주의할 점
  - Service는 Main Thread에서 실행
    - 만약 Service가 CPU 자원을 많이 소모하는 작업이라면 Service안에서 Thread를 생성해서 작업하는 게 좋음
  - App이 실행 중일 때만 필요한 기능이라면 Thread를 사용하고, App이 실행 중이지 않을 때 실행되어야 한다면 Service를 이용해야 함

Application Service

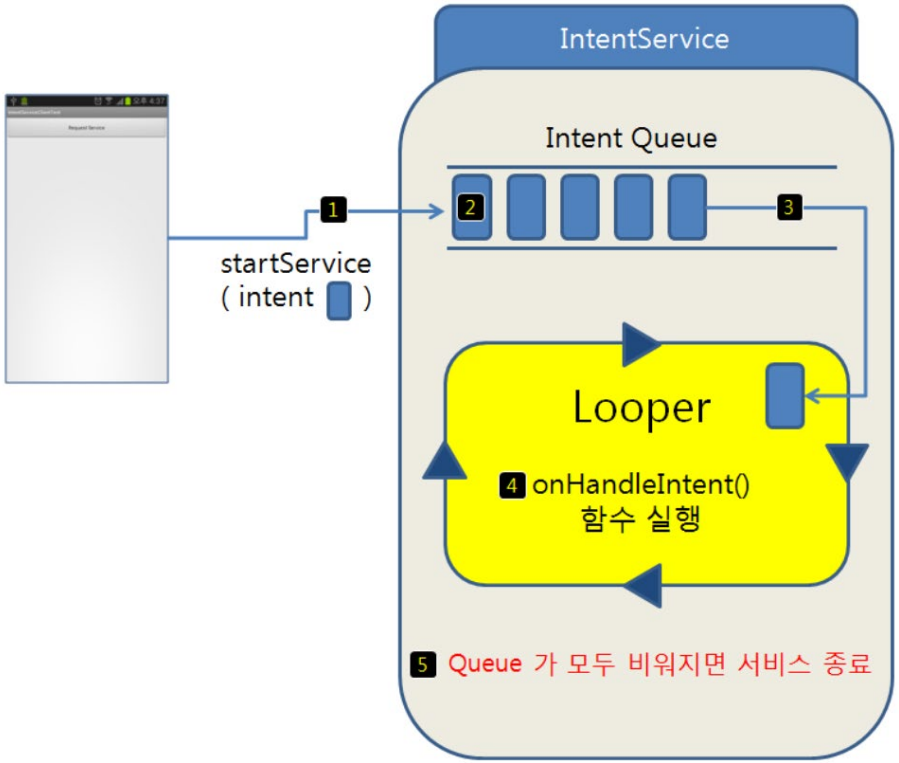
- Started Service(시작 타입의 서비스)
  - startService() 메소드를 호출하면 시작됨
  - 한번 시작되면 Background에서 무기한(unbounded)으로 실행
  - 특정 Service를 동작시켜두는 데 목적을 둔 형태의 Service
  - Service를 동작 시킨 Component는 Service를 중단하는 것 이외에 어떤 제어도 할 수 없음
  - 동작된 Service는 알아서 Background로 작업을 수행하고 종료
  - 호출자에게 결과를 반환할 수 없음
  - 예) File Download, 음악재생 등이 있음

Application Service

- Bounded Service (연결 타입의 서비스)
  - bindService() 메소드 호출 후에 시작
  - Client와 Server와 같이 동작함
    - Activity는 Service에게 어떤 요청을 하고 Service는 결과값을 반환하여 Service가 Server 역할을 함
  - 하나의 Service에 다수의 Activity 연결될 수 있음
  - Process간 통신에도 사용
  - Service Binding은 연결된 Activity가 사라지면 Service도 소멸됨 (Background에서 무한히 실행되지는 않음)
  - Application 안의 기능을 외부에 제공하는 경우에 많이 사용

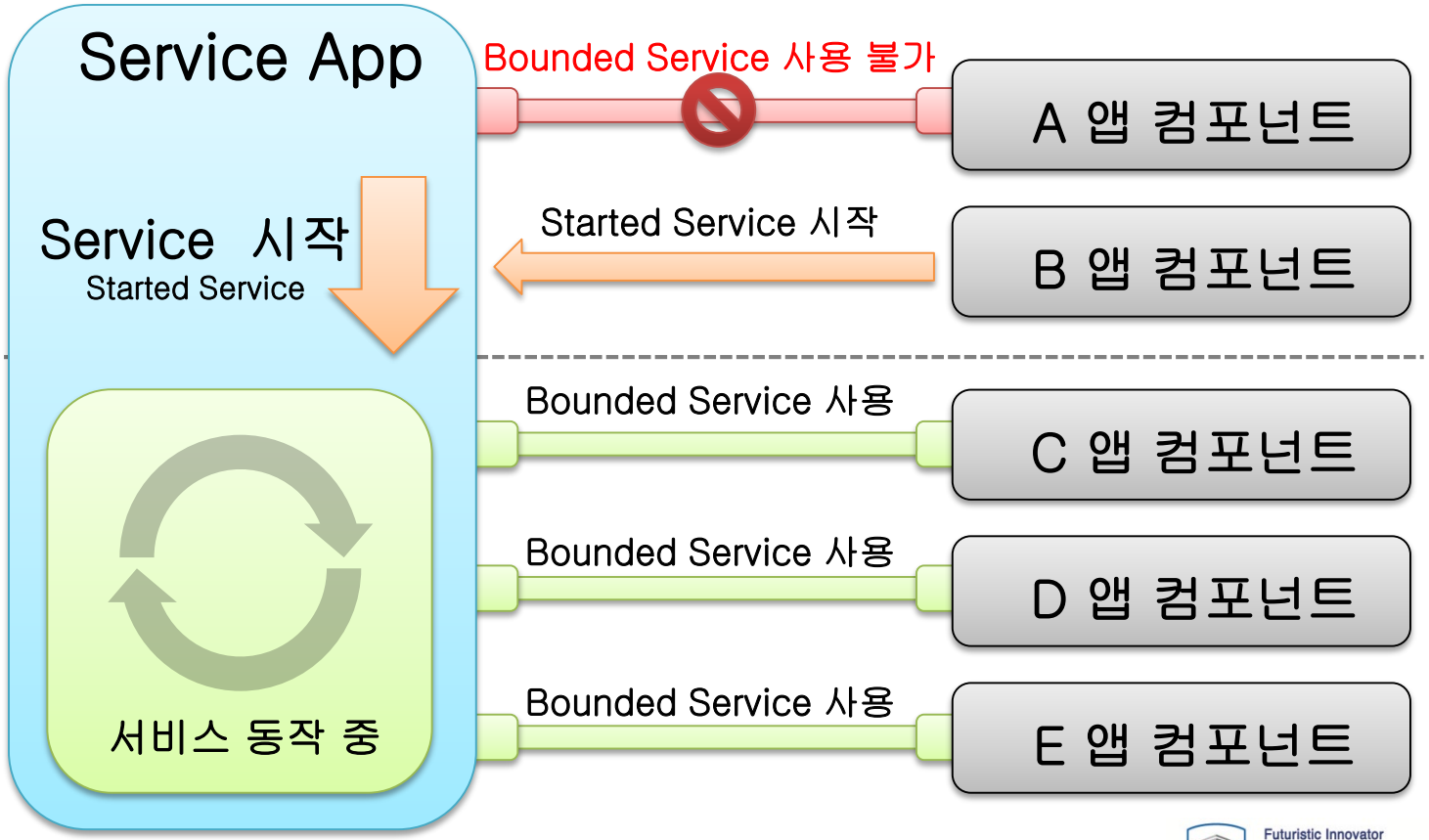
Application Service

- Intent Service
  - 일반 Service와 다르게 요청이 끝나면, 자동으로 Service가 종료
  - 다른 Service들과 다르게 onHandleIntent() 메소드 하나만을 통해 작업을 처리할 수 있음



Application Service

- 왜 Service가 구분되었을까?
  - Started와 Bounded 형태로 Service가 구분된 이유 살펴보기 전에 먼저 둘의 기본 관계를 이해할 필요가 있음

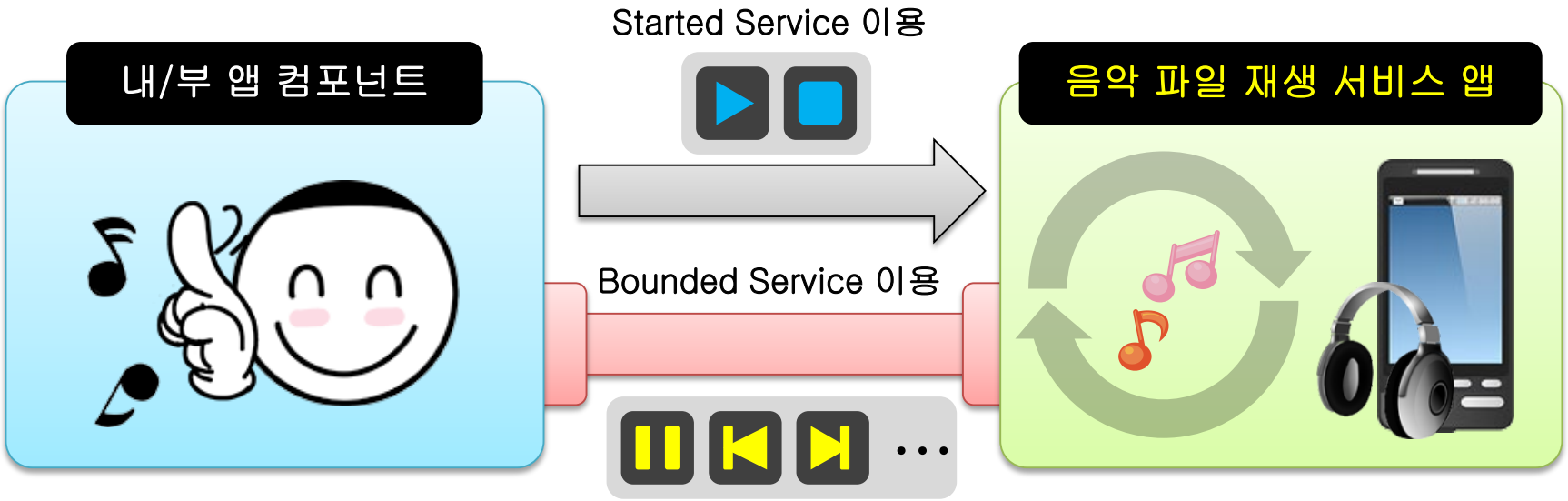


Application Service

- 왜 Service가 구분되었을까?
  - Bounded는 Started 형태의 Service가 실행되어야만 사용할 수 있는데, 그 이유는 음악 재생기를 비유하면 이해하기 쉬움
  - 음악을 재생하기 전에 일시 정지나 다음 곡 재생 등의 기능이 의미가 없지 않은가!
  - 여기까지 생각해보면 Bounded는 Started에 의존적이라 둘의 구별 없이 그냥 Service라 보면 될 것 같음
  - 그리고 이 연관 관계가 Android Service의 기본 임
  - 하지만 Service를 사용하다 보면 Service의 실행과는 상관없이 Bounded 형태만 필요할 때가 많음
  - 예를 들면 앞서 설명된 계산 서비스 같은 경우
  - 이를 위해 Android는 Started와 Bounded의 관계를 끊고 각각 독립적으로 사용할 수 있는 방법을 제공

Application Service

- Started Service와 Bounded Service
  - 지금까지 2가지 형태 Service를 분리해서 설명했지만, 구현할 App에 따라 각 형태의 특징을 모두 활용해야 하는 경우가 있음
  - 이 2가지 형태는 구현할 Service의 목적에 따라 선택적으로 사용하거나 모두 활용하여 사용될 수도 있음



Application Service

- Service 사용시 주의할 점
  - Android는 LINUX 기반이기 때문에 Framework 단에는 LINUX로 구현되어 있음
  - Memory 관리 또한 Linux 정책을 따름 (Linux Kernel에 의해서 관리)
    - 결국 여러 Process들을 Kernel에서 관리
    - 하나의 Process 안에는 Application, 4대 Component, Thread 등을 구성하고 있어서, 4대 Component의 운명 또한 LINUX Kernel에게 달려있음
  - Service는 Main Thread에서 실행
    - 만약 Service가 CPU 자원을 많이 소모하는 작업이라면 Service안에서 Thread로 작업하는 것이 좋음
  - App이 실행 중일 때 만 필요한 기능이라면 Thread를 사용하는 게 맞고, App이 실행 중이지 않을 때 실행되어야 한다면 Service를 이용해야 함

Application Service

- Service 사용시 주의할 점
  - Memory부족이나, 과부하 등과 같은 현상이 발생했을 때 LINUX Kernel이 Process를 강제로 종료시킬 수 있음
  - Android UI 관련 처리는 UI Thread에서 처리하는데 Service를 그냥 생성하면, 기본적으로 이 Main Thread에게 붙는데, 만약 Service에게 많은 일을 할당하면 Main Thread에게 부담이 됨
  - Service에게 많은 일을 해야 하는 경우는, Service의 일은 곧 Main Thread이고, Main Thread는 그 일을 처리하느라고 UI Update를 신경 쓰지 못하고 사용자는 멈춰있는 Application을 마주하게 됨
  - Google은 사용자가 Application UI Update를 오랫동안 기다려야 하고 늦은 응답에 대해 무한정 기다려주는 정책을 펼치지 않음. 이 때 ANR이라는 것을 발동시켜 강제로 Process를 죽여 버림

Application Service

- Service 사용시 주의할 점
  - ANR을 방지하기 위해서는 만약 많은 일을 Service가 처리해야 한다면, 별도의 Work Thread를(Background Thread) 만들어 처리해야 함

Service 클래스

- Android에서 사용되는 모든 Service의 기본의 되는 클래스
- Service는 Background에서 장기 작업을 수행하기 위해 설계된 Android Component
- 사용자가 App을 종료하거나 화면 전환을 해도 Background에서 실행되며, 주로 UI와 직접적으로 상호작용하지 않음
- Service는 UI 작업 없이 비동기적으로 Data를 처리하거나 다른 App 또는 System과 통신할 때 사용
- 이 클래스를 상속받았을 때, Service의 모든 작업을 처리할 새로운 Thread를 생성하는 것이 중요함
- Service는 기본적으로 Application의 Main Thread를 사용하기 때문에, 이를 그대로 사용하면 App에서 사용하는 Activity의 속도가 느려질 가능성이 큼

Service 클래스

- Service의 주요 특징
  - Background 실행
    - UI Thread와 별개로 작업을 수행하지만, 기본적으로 Main Thread에서 실행
    - CPU 집약적이거나 오래 지속되는 작업에는 별도의 Thread 또는 WorkManager, IntentService를 사용하는 것이 좋음
  - Lifecycle 관리
    - onCreate(), onStartCommand(), onDestroy() 등의 메소드를 통해 Lifecycle을 관리
  - UI 제공 없음
    - 직접적으로 화면에 표시되지 않으며, UI와 상호작용할 수 없음
    - UI 작업이 필요할 경우 BroadcastReceiver 또는 Handler를 통해 Activity에 전달

Service 클래스

- Service의 주요 Lifecycle 메소드
  - void onCreate()
    - 호출 시점
      - Service가 최초로 생성될 때 호출
      - Service가 이미 실행 중인 경우, 이 메소드는 다시 호출되지 않음
    - 주요 역할
      - Service 초기화 작업 수행
      - Resource 설정, Thread 생성 등 초기 준비 작업에 사용

Service 클래스

- Service의 주요 Lifecycle 메소드
  - int onStartCommand(Intent intent, int flags, int startId)
    - 호출 시점
      - Service가 startService()로 시작될 때 호출
      - 여러 번 호출될 수 있음
        (예: startService가 반복 호출된 경우)
    - 주요 역할
      - Service의 주요 작업 로직을 처리
      - 여기서 Intent를 다루고 작업을 수행
      - Service가 종료되었을 때의 동작을 return 값으로 결정
    - 매개 변수
      - intent : Client가 Service를 시작할 때 전달하는 것
      - flags : Service의 요청에 대한 추가 정보
      - startId : Service 요청에 대한 고유한 식별자

Service 클래스

- int onStartCommand(Intent intent, int flags, int startId)
  - 반환값을 통해 Service의 재 시작 방식을 제어할 수 있음

반환 값 상수	설명
START_STICKY	✓ 표준 방식 ✓ Service가 Runtime에 의해 종료되면 항상 재 시작되며, 재 시작 될 때마다 onStartCommand()가 호출. 이때 전달되는 intent는 null 임 ✓ 지속적인 Background 작업이 필요한 경우나 음악 재생 Service 등에 적합한 방식
START_NOT_STICKY	✓ Service가 Runtime에 의해 종료되어도 startService()를 다시 호출하지 않으면 해당 Service는 중지됨 ✓ Update나 Network Polling과 같이 규칙적인 처리를 다루는 Service에 적합 (중지되어도 다음 예약 시점에 다시 호출됨)
START_REDELIVER_INTENT	✓ Service가 Runtime에 의해 종료되면 startService()를 다시 호출하거나, Process가 stopSelf()를 호출하기 전에 종료된 경우에만 재 시작됨. 후자의 경우에는 onStartCommand()가 호출되며, 처리가 덜된 초기 Intent가 전달됨 ✓ Service가 요청 받은 명령을 반드시 처리 완료해야 하는 경우에 적합한 방식

Service 클래스

- Int onStartCommand(Intent intent, int flags, int startId)
  - flags 인자를 통해 Service가 어떻게 시작됐는지 알아낼 수 있음

flag	설명
START_FLAG_RETRY	재 시작 방식이 START_STICKY이면서 Service가 비정상적인 종료를 당한 후에 재 시작되었음을 나타냄
START_FLAG_REDELIVER	재 시작 방식이 START_REDELIVER_INTENT이면서 Service가 stopSelf()를 호출하기 전에 비정상적인 종료를 당한 후에 재 시작되었음을 나타냄

Service 클래스

- Service의 주요 Lifecycle 메소드
  - IBinder onBind(Intent intent)
    - 호출 시점
      - Service가 bindService()로 호출될 때 실행
      - Bounded Service의 Client가 Service를 연결하려고 할 때 사용
    - 주요 역할
      - IBinder 객체를 반환하여 Client와의 통신을 제공
      - Bounded Service는 연결된 Client가 모두 연결을 끊으면 종료됨
    - 반환 값
      - IBinder 객체 : Client와의 통신 Interface 제공
      - null : Bounded Service가 아닌 경우

Service 클래스

- Service의 주요 Lifecycle 메소드
  - boolean onUnbind(Intent intent)
    - 호출 시점
      - bindService()로 연결된 Client가 unbindService()를 호출할 때 실행
      - 마지막 Client가 Service를 연결 해제할 때 호출
    - 주요 역할
      - Resource 정리나 특정 작업을 종료
  - void onRebind(Intent intent)
    - 호출 시점
      - onUnbind()가 호출된 이후, Client가 다시 bindService()를 호출할 때 실행
    - 주요 역할
      - Client가 다시 연결될 때 필요한 작업을 수행

Service 클래스

- Service의 주요 Lifecycle 메소드
  - void onDestroy()
    - 호출 시점
      - Service가 종료될 때 호출
        - 외부에서 stopService()를 호출하거나
        - 내부에서 stopSelf()를 호출 시 호출됨
      - System에 의해 강제 종료된 경우에는 호출되지 않을 수 있음
    - 주요 역할
      - Resource 정리, Thread 중지, Notification 해제 등의 종료 작업 수행

ServiceConnection 클래스

- ServiceConnection 클래스는 Android에서 Bounded Service를 Client(예: Activity 또는 Fragment)에 연결하거나 해제하는 데 사용되는 Interface
- Client와 Service 간의 통신을 설정하고, Service 연결 상태를 Monitoring하는 데 중요한 역할을 함

ServiceConnection 클래스

- ServiceConnection의 주요 역할
  - Service 연결
    - bindService() 메소드를 통해 Service를 연결하면 onServiceConnected()가 호출
    - 연결된 Service의 IBinder 객체를 Client에 제공하여, Service와 통신할 수 있는 Interface를 설정
  - Service 연결 해제
    - Service가 비정상적으로 종료되거나 unbindService()가 호출되면 onServiceDisconnected()가 호출
  - Service 상태 관리
    - 연결 상태 변화를 감지하고 필요한 작업을 수행할 수 있음

ServiceConnection 클래스

- ServiceConnection의 주요 메소드
  - void onServiceConnected(ComponentName name, IBinder service)
    - 호출 시점
      - bindService() 메소드를 통해 Service가 연결되었을 때 호출
    - 매개변수
      - ComponentName name: 연결된 Service의 Component 정보 (예: 패키지 이름, 클래스 이름)
      - IBinder service: Service와의 통신을 위한 Interface 객체
    - 주요 역할
      - IBinder 객체를 이용해 Service와 상호작용하거나 Data를 주고받을 수 있는 Interface를 설정

ServiceConnection 클래스

- ServiceConnection의 주요 메소드
  - void onServiceDisconnected(ComponentName name)
    - 호출 시점
      - Service 연결이 비정상적으로 끊어졌을 때 호출
      - 예) Service가 종료되거나 Process가 종료될 때 발생
    - 매개변수
      - ComponentName name: 연결이 끊어진 Service의 Component 정보
    - 주요 역할
      - Service 연결이 끊어졌을 때 자원을 해제하거나 재연결 시도를 수행

ServiceConnection 클래스

- Service 기능을 호출할 경우 처리해야 할 2가지 예외
  - DeadObjectException
    - 발생하면 Service 연결이 비정상적으로 종료 되었음을 의미함
    - 연결이 끊겼을 때의 절차를 수행해야 함
    - 필요한 경우 onServiceDisconnectied() 메소드를 수동으로 호출해야 함
  - RemoteException
    - Process간의 연결 과정에 문제가 있음을 나타내는 범용 예외 상황
    - 발생한 경우 Service와의 연결을 새로 초기화 하는 것이 좋음

IntentService 클래스

- IntentService는 Android의 Service 클래스를 확장하여, Background에서 작업을 비동기적으로 처리하도록 설계된 특수한 Service
- 주요 특징은 작업 요청(Intent)을 큐(Queue)에 저장하고, 하나씩 처리하며 모든 작업이 완료되면 자동으로 Service를 종료한다는 것임

IntentService 클래스

- IntentService의 주요 특징
  - Background Thread에서 실행
    - IntentService는 별도의 작업 Thread(HandlerThread)를 생성하여 작업을 실행하므로, Main Thread를 차단하지 않음
    - 직접 Thread를 관리할 필요 없이 비동기 작업을 처리할 수 있음
  - 작업 Queue 방식
    - startService()로 전달된 Intent가 FIFO(First In, First Out) 방식으로 처리
    - 하나의 작업이 끝난 후 다음 작업이 실행

IntentService 클래스

- IntentService의 주요 특징
  - 자동 종료
    - 모든 작업이 완료되면 IntentService가 자동으로 종료
    - 명시적으로 stopSelf()를 호출하지 않아도 됨
  - Thread-safe
    - 내부적으로 작업 Queue를 사용하기 때문에 동시에 여러 요청이 들어와도 안전하게 처리됨

IntentService 클래스

- IntentService 클래스의 주요 메소드
  - void onHandleIntent(Intent intent)
  - 호출 시점
    - Service에 작업 요청(Intent)이 들어올 때마다 호출
    - Background Thread에서 실행되므로 Main Thread를 차단하지 않음
  - 주요 역할
    - 전달받은 Intent를 처리하는 작업 로직을 구현
    - 모든 작업이 완료되면 IntentService는 자동으로 종료
  - 매개변수
    - Intent intent: 작업 요청 시 전달된 Data가 포함된 객체

IntentService 클래스

- IntentService의 주요 메소드 간 관계
  - startService(Intent)가 호출되면
    - onCreate() → onStartCommand() → onHandleIntent() 순으로 호출
  - Intent 요청이 들어올 때마다
    - onHandleIntent(Intent intent)에서 작업 처리
  - 모든 작업이 완료되면
    - onDestroy()가 호출되고 Service가 종료

IntentService 클래스

- IntentService의 장점
  - Thread 관리의 단순화
    - 별도의 Thread를 생성하고 관리할 필요 없이 Background 작업을 쉽게 구현할 수 있음
  - 자동 종료
    - 작업이 완료되면 자동으로 종료되어 Memory 누수를 방지
  - 안전한 작업 처리
    - 작업이 Queue로 관리되기 때문에 동시에 여러 요청이 들어와도 충돌 없이 처리
  - 직관적인 구현
    - 작업 처리 로직을 onHandleIntent()에 작성하여 간결하게 구현할 수 있음