



Thread & Service



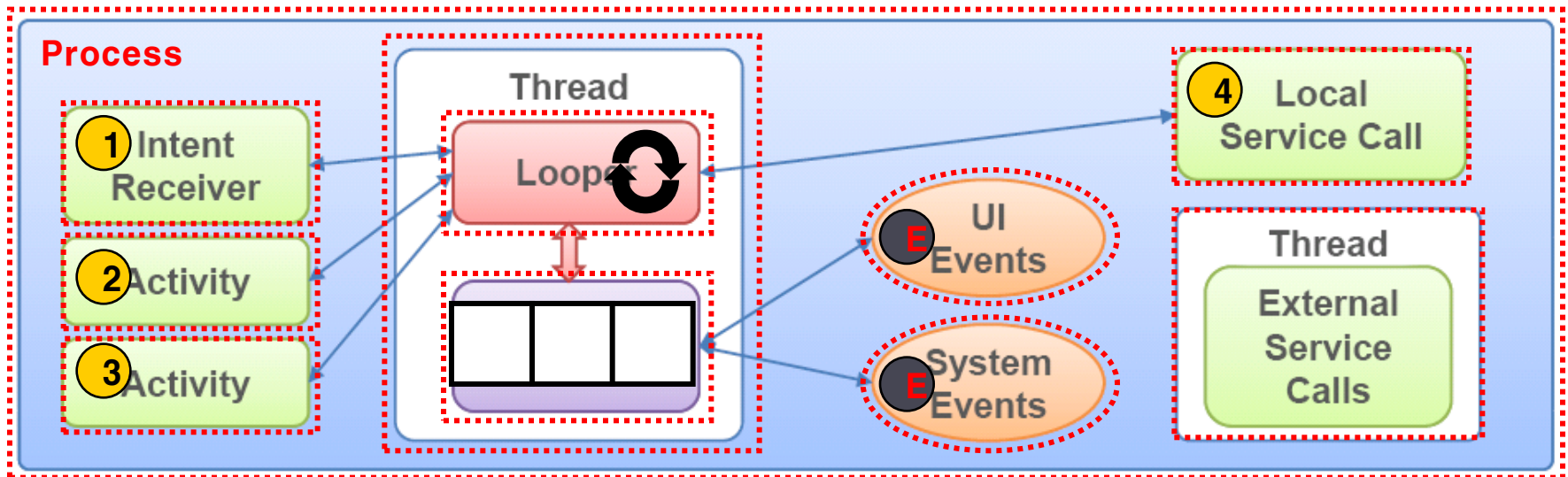


Thread

Thread의 이해

Thread

- 한 프로세스는 기본적으로 하나의 Thread를 가지고 있고, 추가적으로 Thread를 생성 가능 함
- 서로 다른 Thread에서 상대방의 View Component 직접 접근은 할 수 없음
- Handler를 이용하여 Message Queue에 Message를 넣어 주고 받음



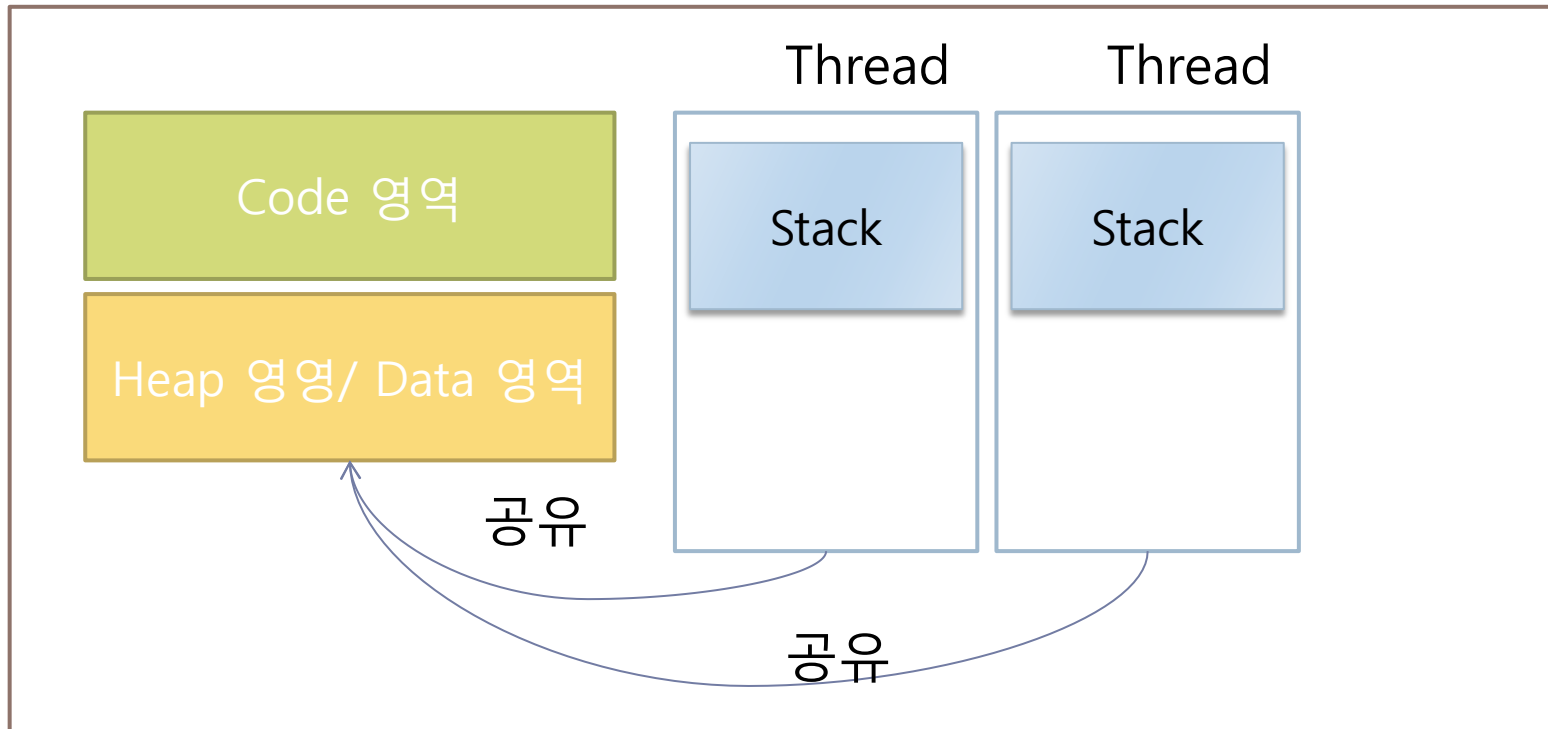
Thread

1. Thread : Process 내의 실행 흐름

Stack 공간을 제외한 Process의 모든 내용을 Thread는 사용할 수 있음

2. Process : 실행 중인 Program <Manifest에 package이름으로 존재> 하거나 <activity의 속성으로 android:process를 이용해 새롭게 지정 가능>

Process



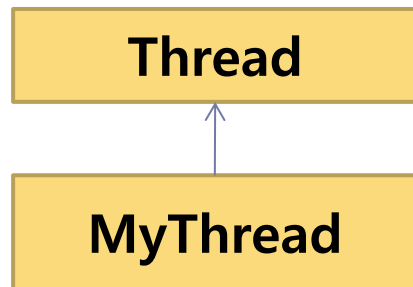
Java의 쓰레드(Thread)

1. Thread class (상속)

- run() : 사용자 Thread의 Handler
- start() : 사용자 Thread를 시작시키는 Method
- 단일 상속 가능하므로, 다른 클래스 상속 못 받음

2. Runnable Interface (구현 상속)

- 다중 상속이 가능한 Thread 지원



run() - override

Thread 선언 방법

```
class MyThread extends Thread {  
    public void run() {...}  
}  
MyThread t1 = new MyThread();  
t1.start();
```

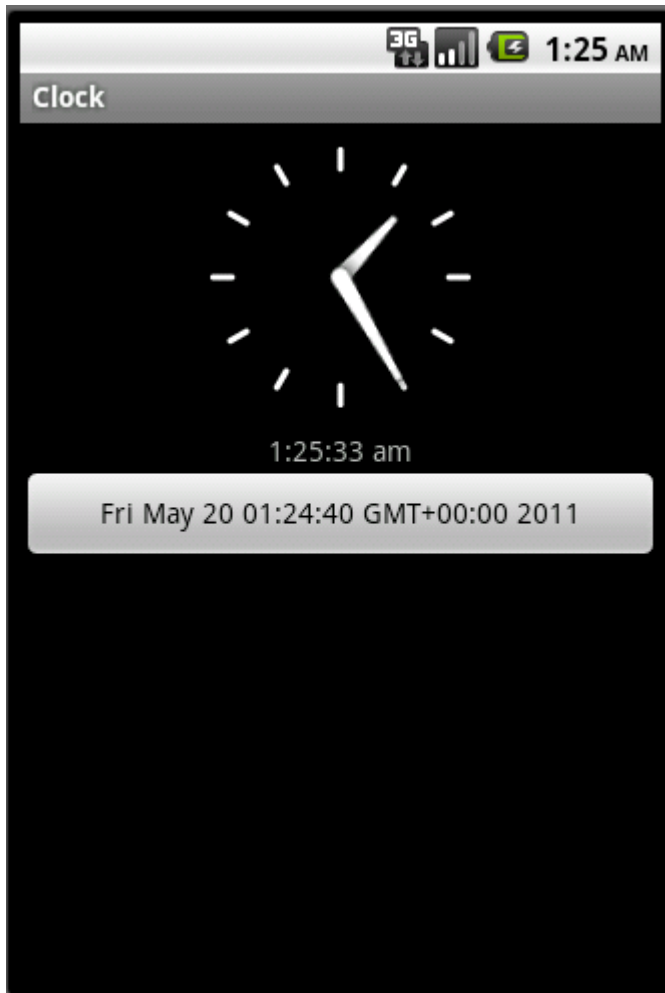
Thread

```
class MyThread implements Runnable {  
    public void run() {...}  
}  
Thread t1 = new Thread(new MyThread());  
t1.start();
```

Thread (Runnable)



08_SimpleClock



- 시간을 표시하는 App
- AnalogClock, DigitalClock, Button
- AnalogClock, DigitalClock은 동작함
- Button은 클릭해야 동작함

Simple Clock – main.xml

```
<AnalogClock  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center_horizontal"  
/>
```

아날로그 시계

```
<DigitalClock  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center_horizontal"  
/>
```

디지털 시계

```
<Button  
    android:id="@+id/button"  
    android:text=""  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
/>
```

시간을 표시할 버튼



Simple Clock – MainActivity.java

```
public class MainActivity extends Activity implements
    View.OnClickListener{
    Button btn;
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        btn = (Button)findViewById(R.id.button);
        btn.setOnClickListener(this);
        btn.setText(new Date().toString());
    }
    public void onClick(View view) {
        updateTime();
    }
    private void updateTime() {
        btn.setText(new Date().toString());
    }
}
```

초기에 시간 표시

클릭하면 시간 표시



Simple Clock – MainActivity.java (Thread 구성)

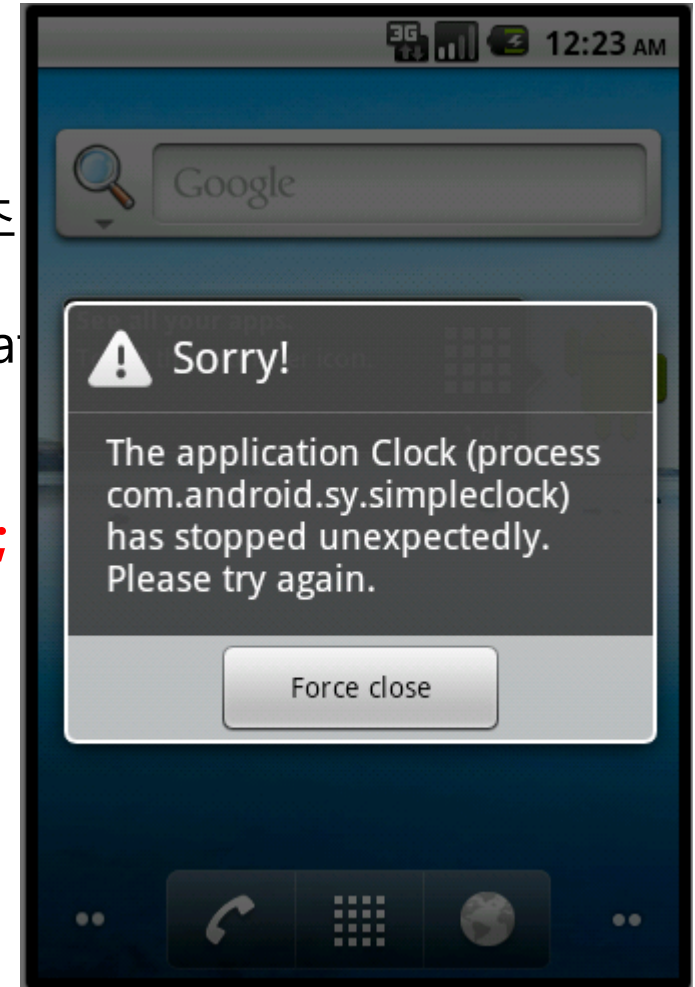
```
public class MainActivity extends Activity implements
    View.OnClickListener{
    Button btn; String str;
    ... 생략 ...
    private void updateTime() {
        btn.setText(str);
    }
    protected void onStart() {
        // TODO Auto-generated method stub
        super.onStart();
        InnerBackgroundThread t1 = new InnerBackgroundThread();
        t1.setDaemon(true);
        t1.start();
    }
    // 뒷 페이지의 InnerBackgroundThread inner class작성
}
```



Simple Clock – MainActivity.java (Thread 구성)

```
class InnerBackgroundThread extends Thread{
    public void run() {
        // TODO Auto-generated method stub
        while(true){
            try {
                Thread.sleep(1000);    // 1초
            } catch (InterruptedException e) {
                // TODO Auto-generated ca
                e.printStackTrace();
            }
            btn.setText(new Date().toString());
        }
    }
}
```

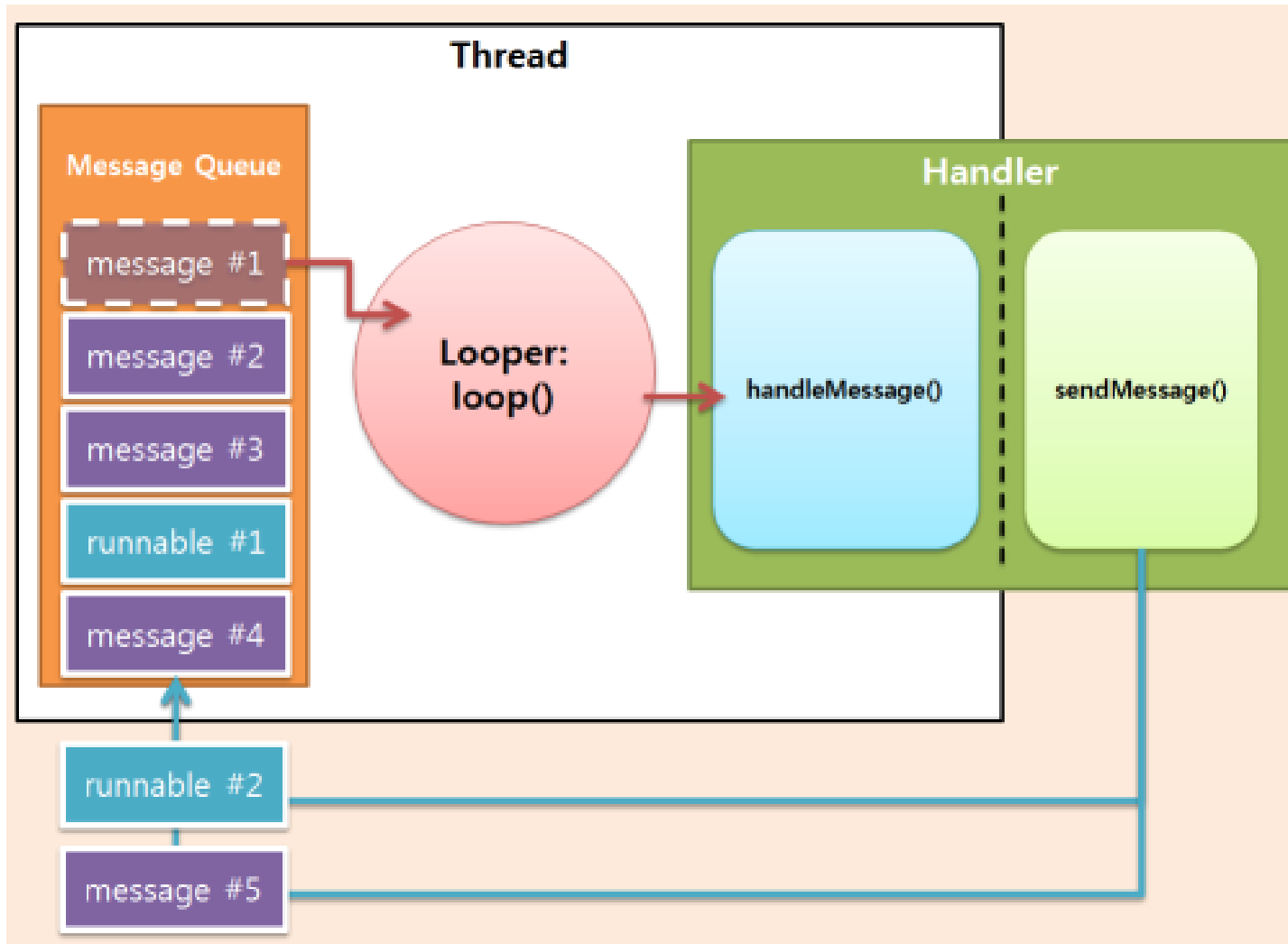
inner class



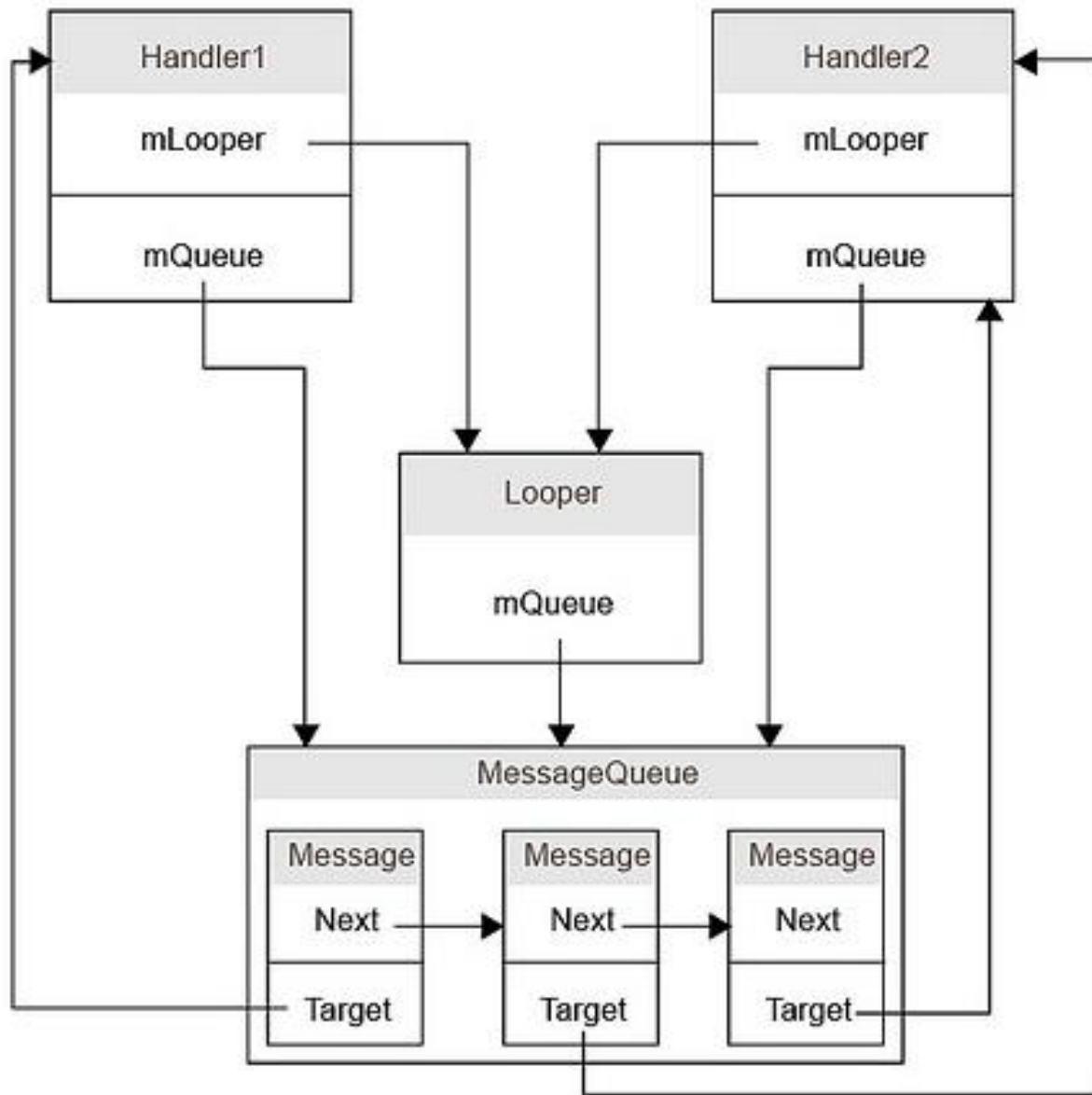
Handler의 사용

Handler

- Handler를 이용하여 MessageQueue에 Message를 넣어 **주고 받음**



Handler, Looper and MessageQueue



Handler

- Handler는 메시지가 들어가는 입구이자 출구 역할을 함

Handler: 스레드간 통신을 담당하며, Message를 통해 백그라운드 스레드와 메인 스레드 사이에서 백그라운드 스레드에서 값이 변경된 것을 알리거나 메인 스레드의 값을 갱신하도록 요청함

- ✓ 입구 역할을 하는 메서드 : 일반적으로 Handler가 속한 스레드가 아닌 외부 스레드에서 호출됨 // handleMessage()
- ✓ 출구 역할을 하는 메서드 : 항상 Handler가 속한 스레드 내부에서 호출 됨 // sendMessage(), sendMessage()
- ✓ View Component는 Multi Thread의 접근을 허가하지 않고 있음 (동기화 문제 발생 방지) - 오류 발생



Handler

Handle의 동작 원리

- Handler의 처리 대상 : Message 오브젝트, Runnable 오브젝트
 - ✓ Runnable 오브젝트 – run() 메서드를 호출하여 처리
 - ✓ Message 오브젝트 – handleMessage() 메서드를 호출하여 처리
 - ✓ handleMessage()메서드를 오버라이드 하여 원하는 메시지를 처리하는 루틴을 구현함
- Handler를 이용한 Message 객체 전달
 1. Handler의 **sendMessage**를 사용하여 MessageQueue에 메시지를 넣음
 2. Message는 '어떤 스레드'에 속해 있는 MessageQueue에서 나옴
 3. Looper는 MessageQueue의 메시지를 꺼내 Handler에 전달함
 4. Handler는 **handleMessage**를 통해 메시지를 처리함



Handler

Handle의 동작 원리

- Handler를 이용한 Runnable 객체 전달
 1. Handler의 **post**를 사용하여 MessageQueue에 메시지를 넣음
boolean post (Runnable r)
 2. **Runnable**은 '어떤 스레드'에 속해 있는 MessageQueue에서 나옴
 3. Looper는 MessageQueue의 메시지를 꺼내 Handler에 전달함
 4. Runnable의 **run 메서드**가 실행됨코드 상으로 Message 전달 보다 간단함



Handler 선언 방법

```
private final Handler handler = new Handler() {  
    @Override  
    public void handleMessage(Message msg){  
        super.handleMessage(msg);  
        // 여기에 메인 스레드에서 해야 할 작업 기술  
        msg는 받은 메시지가 들어 있음  
    }  
};
```

주요 메서드 // **보내는 메시지**

boolean Handler.sendMessage (int what)

boolean Handler.**sendMessage** (Message msg)

boolean Handler.sendMessageAtFrontOfQueue (Message msg)



Handler 선언 방법

- **Thread가 Activity 내부에 존재할 때는 Thread가 Activity의** 멤버인 `str`을 직접 사용할 수 있으므로 **Message기술 불필요** (`sendEmptyMessage` 사용으로 충분하며, `sendMessage` 사용도 가능함)
- 일반적인 경우 **Thread와 Activity class는 분리되어 작성하며**, 이 경우는 멤버 참조를 직접할 수 없으므로 **Message 기술이 필요함** (`sendMessage` 사용이 필요하며, 전달 값이 특별히 없을 경우 `sendEmptyMessage` 사용도 가능함)
- 우선 순위 변경을 위해 '`sendMessageAtFrontOfQueue`' 메소드를 사용함



Message의 사용

static Message obtain (Handler h, int what, int arg1, int arg2, Object obj)

Message를 통신할 때마다 new 연산자로 생성 하게 되면 메모리 소모가 많고 속도도 느리기 때문에 **Android는 시스템에 메시지 풀을 두어 캐시를 유지하며 obtain 메서드를 사용해 꺼내도록 함**

필드	설명
int what	메시지의 의미를 설명, 의미가 정해져 있지는 않으며, 구분이 필요한 경우 사용
int arg1	메시지의 추가 정보
int arg2	메시지의 추가 정보
Object obj	정수만으로 메시지를 기술할 수 없을 때 객체를 보냄
Messenger replyTo	메시지에 대한 응답을 받을 객체 지정



Simple Clock – MainActivity.java (Thread 구성 1)

```
class InnerBackgroundThread extends Thread{  
    public void run() {  
        // TODO Auto-generated method stub  
        while(true){  
            try {  
                Thread.sleep(1000);    // 1초 쉬기  
            } catch (InterruptedException e) {  
                // TODO Auto-generated catch block  
                e.printStackTrace();  
            }  
            str = new Date().toString();  
            mHandler.sendMessage(0);  
        }  
    }  
}
```

inner class 수정



Simple Clock – MainActivity.java (Thread 구성 1)

```
// Activity는 핸들러를 구현하여 받은 메시지가 있는 경우 처리함
Handler mHandler = new Handler() {
    public void handleMessage(Message msg){
        super.handleMessage(msg);
        if (msg.what == 0){ // msg 구분을 위해 사용
            updateTime();
        }
    }
};
```

handler의 구현

QUIZ.1

Thread를 상속 받아 구성된 것을

→Runnable을 이용한 Thread로 구현하시오.

Runnable을 구현한 클래스명은 **InnerBackgroundRunnable**로 하시오.



Simple Clock – MainActivity.java (Thread 구성 2)

```
class BackgroundThread extends Thread{
    String str;
    Handler mHandler;
    BackgroundThread (Handler handler){
        mHandler = handler;
    }
    public void run() {
        // TODO Auto-generated method stub
        while(true){
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
```

별도 class 구성

내용은 비슷함
str, mHandler
의 내용 수정



Simple Clock – MainActivity.java (Thread 구성 2)

별도 class

```
str = new Date().toString();  
//Message msg = Message.obtain();  
//msg.what = 0; msg.obj = str;  
Message msg = Message.obtain(mHandler, 0, str);  
mHandler.sendMessage(msg);
```

```
}
```

```
}
```

```
}
```

// Activity쪽에

```
Handler oHandler = new Handler() {  
    public void handleMessage(Message msg){  
        super.handleMessage(msg);  
        if (msg.what == 0){ // msg 구분을 위해 사용  
            str = (String)msg.obj;  
            updateTime();  
        }  
    }  
}
```

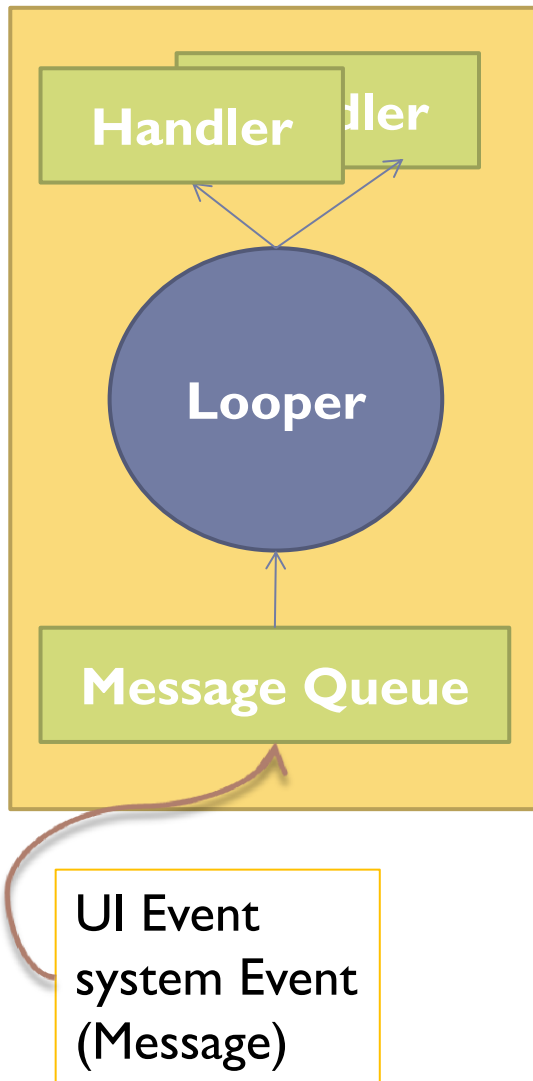
```
}
```

```
};
```




Looper의 사용

Looper



- Message는 스레드간 통신이므로 바로 처리되는 것은 아님
- Message Queue에 쌓아 두었다가 순서대로 처리함
- 큐에서 메시지를 꺼내 Component의 Handler에 전달하는 것이 Looper임
- UI관리를 하는 메인 스레드는 기본적으로 Looper를 가지고 있음
- 작업 스레드는 Looper를 가지고 있지 않아 run 메서드의 코드만 실행하고 종료함
- 작업 스레드가 Message를 받기 위해서는 Looper를 직접 프로그래밍 해야 함

Handler/ Looper

Handle의 생성자

- Handler는 혼자 존재할 수 없지만, 기본 생성자를 제공해주고 있음
 - 기본 생성자를 통해 Handler를 생성하면, 새롭게 생성된 Handler는 해당 Handler의 생성자가 호출된 바로 그 스레드의 MessageQueue 그리고 Looper에 자동 연결 됨
 - Handler와 연결된 Thread를 명시적으로 지정하고 싶을 때는 제공되는 Handler(Looper looper) 생성자를 사용함

Looper의 필요성

- ✓ Handler는 특정 스레드와 연결되어 있어야 하며,
- ✓ 그 스레드는 메시지를 담을 수 있는 MessageQueue와 MessageQueue에서 메시지를 꺼내 넘겨줄 Looper가 필요함
- ✓ 메인 스레드에는 기본적으로 Looper가 존재하지만, 작업 스레드는 기본으로 Looper를 가지지 않으며 run 메서드의 코드만 실행함



Looper

Looper

- 하나의 스레드는 하나의 Looper만 갖을 수 있음
(Looper: 스레드 = 1:1)
- MessageQueue가 비어 있는 동안은 아무일도 안하고 있다가, 메시지가 들어오면 해당 메시지를 꺼내 적절한 Handler로 전달함
- Looper가 동작하고 있는 스레드는 메시지를 처리하는 일 이외에 다른 일을 수행할 수 없음
- 하나의 메시지를 처리하는데 너무 오랜 시간이 걸리는 작업을 수행하면 안됨 (다른 메시지들이 MessageQueue에 쌓여 기다리게 됨)



Looper

Looper 생성 방법 1

- 임의의 스레드에서 직접 Looper를 생성

```
Thread t = new Thread(new Runnable() {  
    public void run() {  
        Looper.prepare();  
        Looper.loop();  
  
        handler = new Handler();  
    }  
});  
t.start();
```

- loop 메서드는 무한 루프를 도는 작업을 하므로, 생성된 스레드는 외부 스레드에서 Looper의 quit 메서드를 호출해 주지 않으면 종료 되지 않음



Looper

Looper 생성 방법 2

- HandlerThread 클래스를 이용하여 Looper를 생성

```
HandlerThread t = new HandlerThread("My Handler Thread");  
t.start();  
handler = new Handler(t.getLooper());
```

- HandlerThread는 Looper를 갖고 있으며, getLooper 메서드를 통해 포함된 Looper를 얻어오거나, quit() 메서드를 통해 Looper의 무한 루프를 정지 할 수 있음

→ Looper가 필요한 경우 HandlerThread를 사용하는 것이 편리함

- Handler는 유용하지만, 주요 스레드를 너무 오래 점유하게 되면 성능과 직결된 문제가 발생할 수 있으므로, 실수 할 경우 예상할 수 없는 결과를 초래할 수도 있음
- 생성된 Handler가 과연 어떤 Thread의 MessageQueue에 연결되어 있는지 확인하는 것이 필요함



Handler

Handler 활용

- Handler의 유용 메서드 3가지
 - ✓ `sendMessageAtFrontOfQueue` : 특정 메시지를 `MessageQueue`의 가장 처음으로 전달함. 현재 수행중인 메시지 처리 구문이 끝나면 바로 해당 메시지가 처리됨
 - ✓ `sendMessageDelayed` : 최소한 정해진 시간이 지난 후에 해당 메시지가 처리됨 (해당 시간 이전에 메시지가 처리되지 않는 것만 보장됨)
 - ✓ `removeMessage` : 특정 조건을 만족하는 메시지를 삭제함



LooperTest – MainActivity.java

```
class BackgroundThreadwLooper extends Thread {  
  
    private Handler mHandler;  
    BackgroundThreadwLooper (Handler handler){  
        mHandler = handler;  
    }  
    @Override  
    public void run() {  
        Looper.prepare();  
        Looper.loop();  
    }  
  
    //  
    public Handler mBackHandler = new Handler () {  
        ... // 뒷 장에 구현해 놓았음  
    }  
  
}
```

looper의 구현



LooperTest – MainActivity.java

```
public Handler mBackHandler = new Handler () {  
    public void handleMessage(Message msg){
```

handler의 구현

```
        Message retmsg = Message.obtain();
```

```
        retmsg.what=msg.what;
```

```
        switch(msg.what){
```

```
        case 0:
```

```
            retmsg.arg1 = msg.arg1 * msg.arg1;
```

```
            break;
```

```
        case 1:
```

```
            retmsg.obj = new Double(Math.sqrt((double)msg.arg1));
```

```
            break;
```

```
        }
```

```
        mHandler.sendMessage(retmsg);
```

```
    }
```

```
};
```



LooperTest – MainActivity.java

```
public class MainActivity extends Activity implements  
View.OnClickListener {
```

```
...
```

```
public void onClick(View view) {
```

```
    Message msg = null;
```

```
    int what=0, i=0;
```

```
    switch(view.getId()){
```

```
        case R.id.button1:
```

```
            what = 0; break;
```

```
        case R.id.button2:
```

```
            what = 1; break;
```

```
    }
```

```
    i = Integer.parseInt(input.getText().toString());
```

```
    msg = Message.obtain(t1.mBackHandler, what, i, 0, null);
```

```
    t1.mBackHandler.sendMessage(msg);
```

```
}
```

```
...
```

```
}
```

**Thread 쪽 Handler에
Message 보내기**





Service

Service 이해, 생명 주기

Service의 이해

Service는 UI와 상관 없이 오랫동안 존재하면서 실행되는 코드임
예를 들면 재생 목록에서 노래를 재생하는 미디어 플레이어 같은 것을 Service라고 할 수 있으며, 미디어 플레이어 APP는 사용자가 곡을 선택하고 재생을 시작하는 하나 이상의 Activity를 가지고 있지만, Activity가 음악을 재생하는 것이 아님



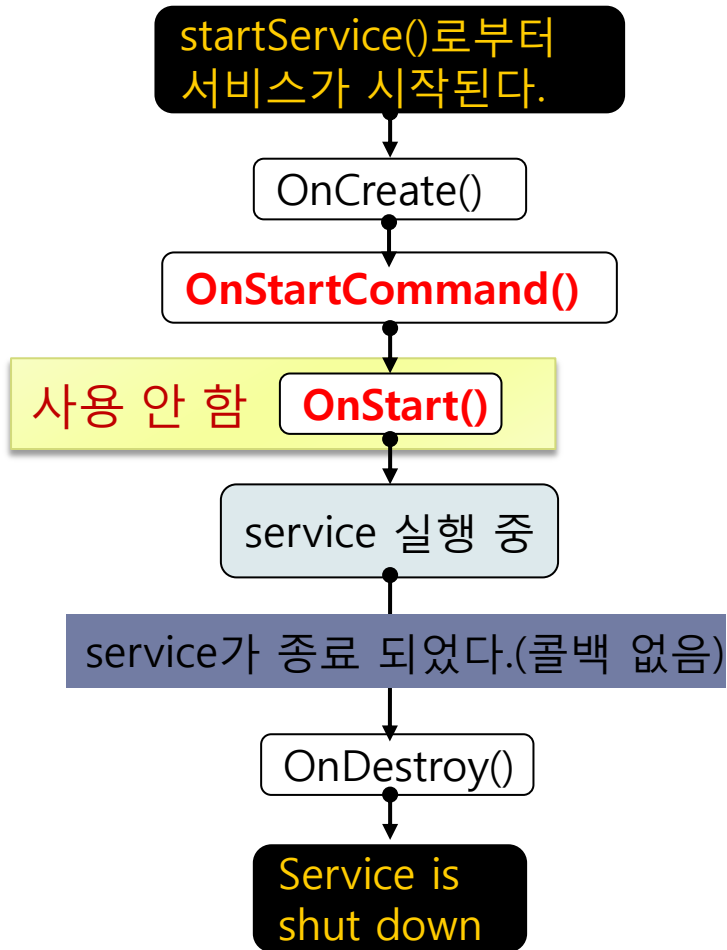
Service의 이해

- 화면 출력과 상관 없이 지속적으로 동작하는 것 (background 동작)
- 별도의 UI 처리 방법을 제공하지 않음, Activity를 위한 연산, 메서드 등의 서비스를 제공함
- 사용 형태 (한 가지 형태 또는 두 가지를 모두 지원하는 형태 가능)
 1. Background 데몬: MP3와 같이 **배경에서 계속 실행되는 프로세스**, 클라이언트가 기동시켜 놓기만 하면 사용자의 명령이 없어도 지속적으로 실행됨
 2. IPC (Inter-Process Communication) : **클라이언트를 위해 특정한 기능을 제공하는 역할**을 하는 것으로, 자신의 기능을 메서드로 호출시키며 클라이언트는 메서드를 호출함으로써 서비스를 이용함
- 사용 형태에 따라 **서비스의 생명주기가 다름**

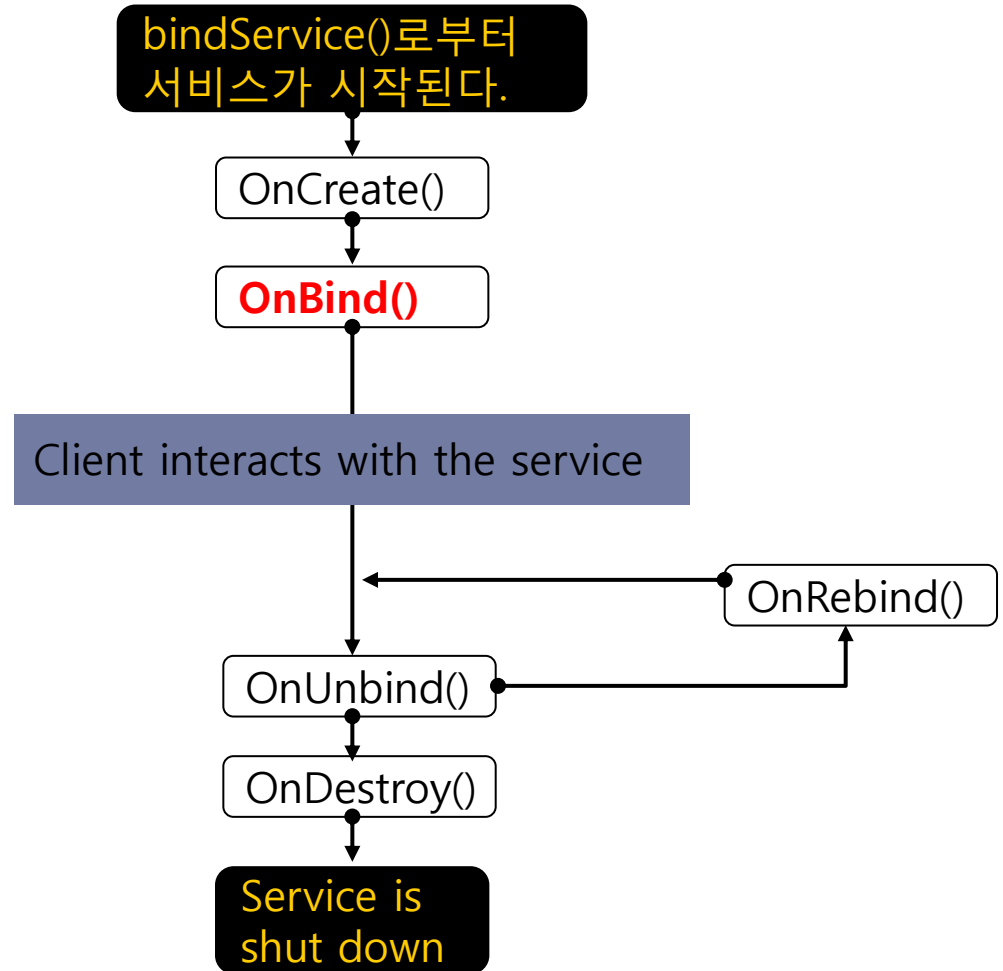


Service의 생명 주기

background 방식의 생명주기



IPC의 경우 생명주기



Service의 이해

- Java의 Thread를 Main Thread를 수행하면서 다른 Thread를 생성하여 처리할 수 있지만, 다른 Application에서 호출하여 사용할 수 있도록 하기위해서는 Android의 구성 요소로 작성해야 함
- 생명 주기를 가지고 있어 실행 시작, 종료에 대한 처리를 콜백 메서드에서 하도록 할 수 있음
- 생명 주기 (시작 부분) // 2.0 이상
 - 1) startService() 처음 실행일 경우
onCreate() → onStartCommand() → onStart()
 - 2) startService() 재호출의 경우
onStartCommand() → onStart()
 - 3) 서비스가 강제 종료 되었다가 다시 실행되는 경우
onCreate() → onStartCommand()



Service의 이해

<< onStartCommand의 필요성 >>

- Service가 정상 종료되지 않고 Background에 남을 수 있음
 1. Application이 StartService 호출
 2. Service의 onCreate, onStart() 함수 호출되고 작업을 수행하기 위해 백그라운드 스레드 생성
 3. 시스템이 메모리 부족으로 현재 작동 중인 서비스 강제 종료
 4. 잠시 후, 메모리 상황이 호전되어 Service가 재시작
(onCreate()만 실행됨, onStart() 실행되지 않음)

위와 같이 다시 재시작 된 Service에 onStart() 부분에 적혀 있는 일을 모르기 때문에 생성된 채로 일을 수행하지 못하고 있으며, 언제 종료되어야 할지도 알 수 없음

onStart()를 사용하지 않고, onStartCommand() 콜백이 사용됨



Service의 이해

<< onStartCommand의 이해>>

- onStartCommand()는 결과값을 반환함
- 안드로이드 플랫폼은 이 결과값을 기반으로 작동중인 서비스가 강제로 종료될 경우 어떤 일을 수행할지 판단함
 - ✓ START_STICKY : 기존에는 프로세스가 강제 종료된 후 Service가 재 시작될 때 onStart() 콜백이 호출되지 않았지만, START_STICKY 형태로 호출된 Service는 null Intent가 담긴 onStartCommand() 콜백 함수 호출
 - ✓ START_NOT_STICKY : 안드로이드 플랫폼에 의해 프로세스가 강제 종료된 후, 다시 시작되지 않고 종료된 상태로 남음
이러한 방식은 특정 Intent로 주어진 명령을 수행하는 동안에만 Service가 실행되면 되는 경우 적당함



Service의 이해

- onStartCommand()는 결과값 계속 ...
 - ✓ START_REDELIVER_INTENT : START_NOT_STICKY와 유사함
단, 프로세스가 강제로 종료되는 경우 (stopSelf()가 호출되기 전에 종료되는 경우), Intent가 다시 전달 되어 Service가 재시작함 (단, 여러 차례 시도한 후에도 작업이 종료되지 않으면 Service는 재시작되지 않음.) 이 모드는 전달받은 명령을 반드시 수행해야 하는 Service에 유용함
 - ✓ START_STICKY_COMPATIBILITY : 기존의 기준으로 제작된 어플리케이션과의 호환성을 위해 존재함 (null intent를 보내지 않고 기존과 동일하게 작동)



Service 생성 방법

```
public class MyService extends Service {  
    @Override  
    public void onCreate() {  
        super.onCreate();  
    }  
    @Override  
    public void onDestroy() {  
        super.onDestroy();  
    }  
    @Override  
    public int onStartCommand(Intent intent, int flags, int startId) {  
        return super.onStartCommand(intent, flags, startId);  
    }  
    @Override  
    public IBinder onBind(Intent arg0) {  
        return null;  
    }  
}
```

서비스는 어떤 식으로 사용될 것인가를 클라이언트가 어떤 메서드로 서비스를 기동시키는가에 의해 결정하므로 onCreate, onStartCommand, onBind, onDestroy 메소드를 정의해야 함

Service AndroidManifest.xml

AndroidManifest.xml

```
<service android:name=".MyService" android:enabled="true" />
```

```
public class MyActivity extends Activity {  
    // service 시작  
    Intent i = new Intent(MyActivity.this, MyService.class);  
    startService(i);  
  
    // service 종료  
    Intent i = new Intent(MyActivity.this, MyService.class);  
    stopService(i);  
}
```

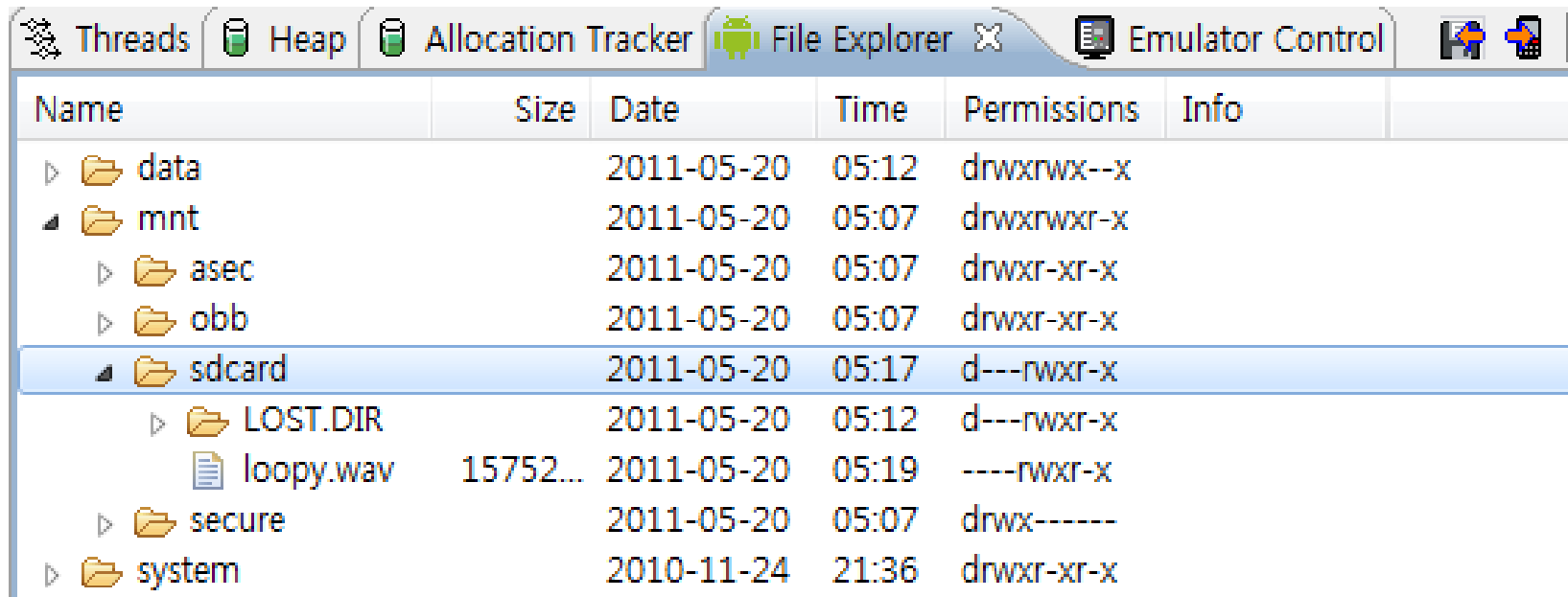


Service의 사용

- wav를 play하는 프로그램인 ServiceApp 프로그램을 Service 방식으로 구현한 것



SD Card에 파일 push // emulator 생성시 용량 지정해야 함



The screenshot shows the File Explorer tab in Android Studio. The directory tree is as follows:

Name	Size	Date	Time	Permissions	Info
data		2011-05-20	05:12	drwxrwx--x	
mnt		2011-05-20	05:07	drwxrwxr-x	
asec		2011-05-20	05:07	drwxr-xr-x	
obb		2011-05-20	05:07	drwxr-xr-x	
sdcard		2011-05-20	05:17	d---rwxr-x	
LOST.DIR		2011-05-20	05:12	d---rwxr-x	
loopy.wav	15752...	2011-05-20	05:19	----rwxr-x	
secure		2011-05-20	05:07	drwx-----	
system		2010-11-24	21:36	drwxr-xr-x	

// ServiceApp 을 다운로드 받아 실습해 보세요.

push가 되지 않을 경우는

adb를 이용하여 push 함 (adb push c:\temp\iam.wav /sdcard/)

IPC(Inter Process Communication)

- AIDL(Android Interface Definition Language)를 이용해서 작성된 .aidl 파일을 클라이언트로 정의해서 **Binder**를 통해 다른 애플리케이션에서 접근할 수 있도록 구현하는 서비스



IPC 작성 과정

1. **AIDL 파일 작성** : src 폴더에 XX.aidl file 생성

2. **Interface Implement**

: Service class를 상속 받은 Server 역할을 하는 class 생성

2.1 XX.Stub class Instance 생성 :

→ 추상 method 구현(원격으로 접속할 method)

2.2 Service에 onBind() 구현

3. **Service의 Bind** : Service에 연결할 Client 역할을 하는 class 구현

3.1 ServiceConnection 생성

3.2 bindService() 정의

3.3 원격 method 호출

3.4 unbindService() 정의

4. **AndroidManifest.xml** 에 service, intent-filter등록



AIDL file 작성 규칙

1. src 폴더에 XX.aidl file 생성

- **interface**로 선언하며, 내부에 선언된 메서드의 매개 인자는 반드시 in, out, inout을 구분해야함. in은 자바의 기본 자료형을 사용했을 때 지정하며, out과 inout으로 선언된 매개 인자는 해당 서비스 내부에서 값 변경 가능하며 변경된 값은 다시 호출한 애플리케이션에도 전달 됨
- String, CharSequence, List, Map 클래스를 선언할 수 있음
- 직렬화를 구현하려면(객체를 주고 받으려면) Parcelable 클래스를 상속받아 구현함
- 작성된 파일은 aidl 확장자로 /src 폴더에 저장. AIDL 컴파일러가 컴파일 후 /gen 폴더에 *.aidl 파일을 *.java (interface) 파일로 생성함 (자동 생성 됨)
- 생성된 **interface** 파일 내부에 **Stub**라는 추상 클래스를 서비스에서 **구현**해야 함 (인터페이스를 상속받는 객체 생성 후, 메서드 구현)



AIDL file 작성 규칙

형	내용
Primitive	in 파라미터로만 사용 가능
String	in 파라미터로만 사용 가능
Parcelable	Parcelable 인터페이스를 implements 한 클래스
AIDL interface	AIDL에서 정의한 interface 형
List	내부에 입력할 수 있는 것은 Parcelable형, AIDL 인터페이스형, List<String>과 같이 Generic 지정 가능
Map	내부에 입력할 수 있는 것은 Parcelable형, AIDL 인터페이스형, List<String>과 같이 Generic 지정은 지원되지 않음
배열	사용 가능한 형의 배열형, Primitive 형 등을 out 파라미터로 지정 가능



AIDL file 작성의 예

```
package com.android.ipctest;
interface IStrService
{
    int addString(in String str);
    String getString (in int index);
    boolean getStrings(in int[] indexes, out String[] str);
    List<String> getStringList(in int start, in int end);
}
```



Interface Implement

2. Service class를 상속 받은 Server 역할을 하는 class 생성

2.1. XX.Stub calss Instance 생성 :

→ 추상 method 구현(원격으로 접속할 method)

2.2. Service에 onBind() 구현

```
public class StrService extends Service{
    private IStrService.Stub mStrService = new IStrService.Stub() {
        private List<String> mStrs = new ArrayList<String>();
        @Override
        public int addString(String str) throws RemoteException {
            } // 추상 메서드의 Override
    };
    @Override
    public IBinder onBind(Intent intent) {
        return mStrService; // onBind Override
    }
}
```



Service의 Bind

3. Service에 연결할 Client 역할을 하는 class 구현

3.1 ServiceConnection 생성

3.2 bindService() 정의

3.3 원격 method 호출

3.4 unbindService() 정의



Service의 Bind

```
public class IPCTestActivity extends Activity {  
    private IStrService mService;  
    private ServiceConnection mConnection  
        = new ServiceConnection() {  
        @Override  
        public void onServiceConnected(ComponentName name, IBinder service) {  
            mService = IStrService.Stub.asInterface(service);  
            putTimeStamp();  
        }  
        @Override  
        public void onServiceDisconnected(ComponentName name) {  
            mService = null;  
        }  
    };  
    ..... // 뒷 장에 계속
```



Service의 Bind

```
// private IStrService mService;  
  
...  
try {  
    mLastIndex = mService.addString(new Date().toString());  
} catch (RemoteException e){ }  
  
try {  
    String s = mService.getString(mLastIndex);  
} catch (RemoteException e) { }  
  
...  
}
```

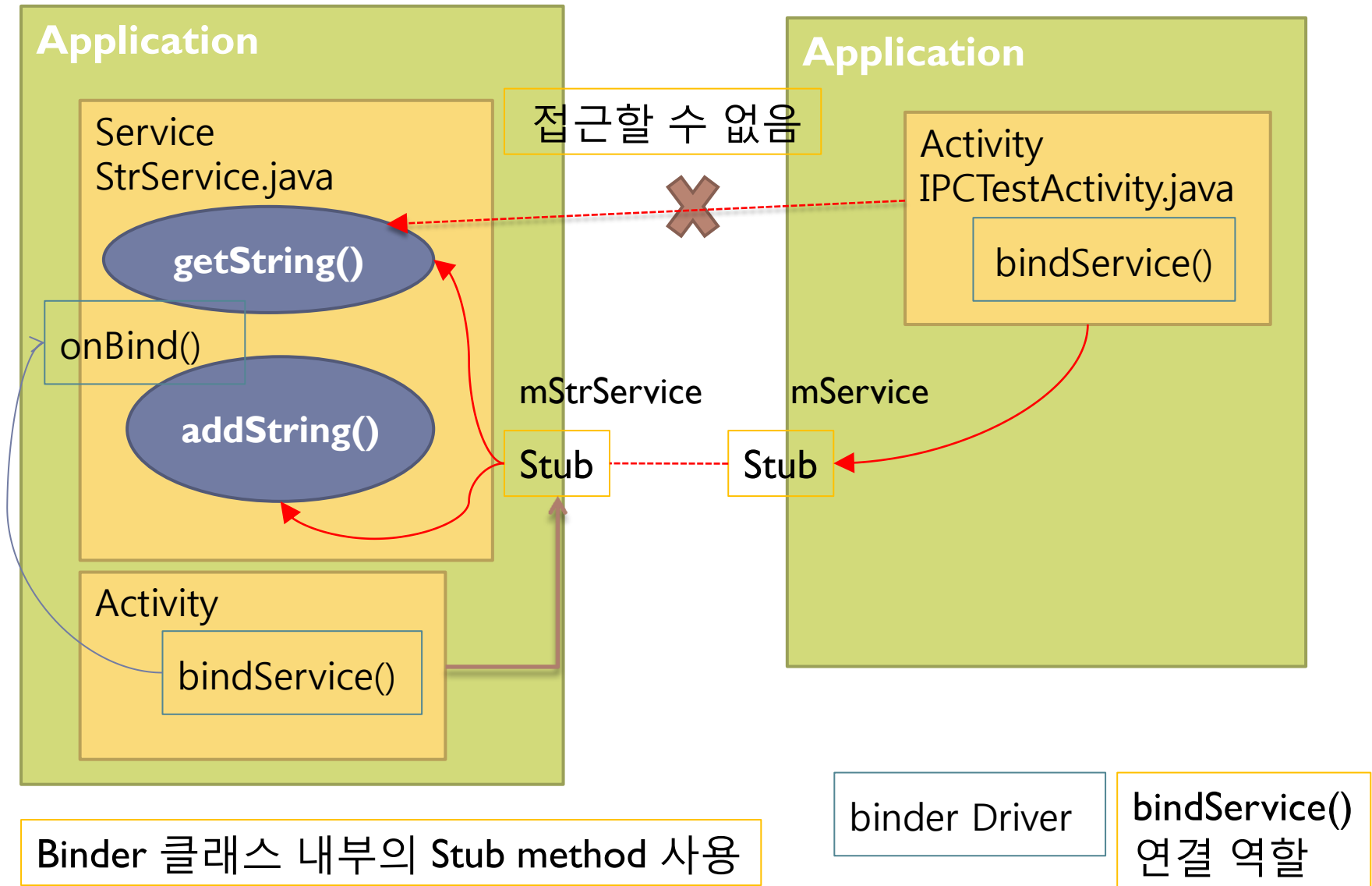


AndroidManifest.xml에 Service 등록

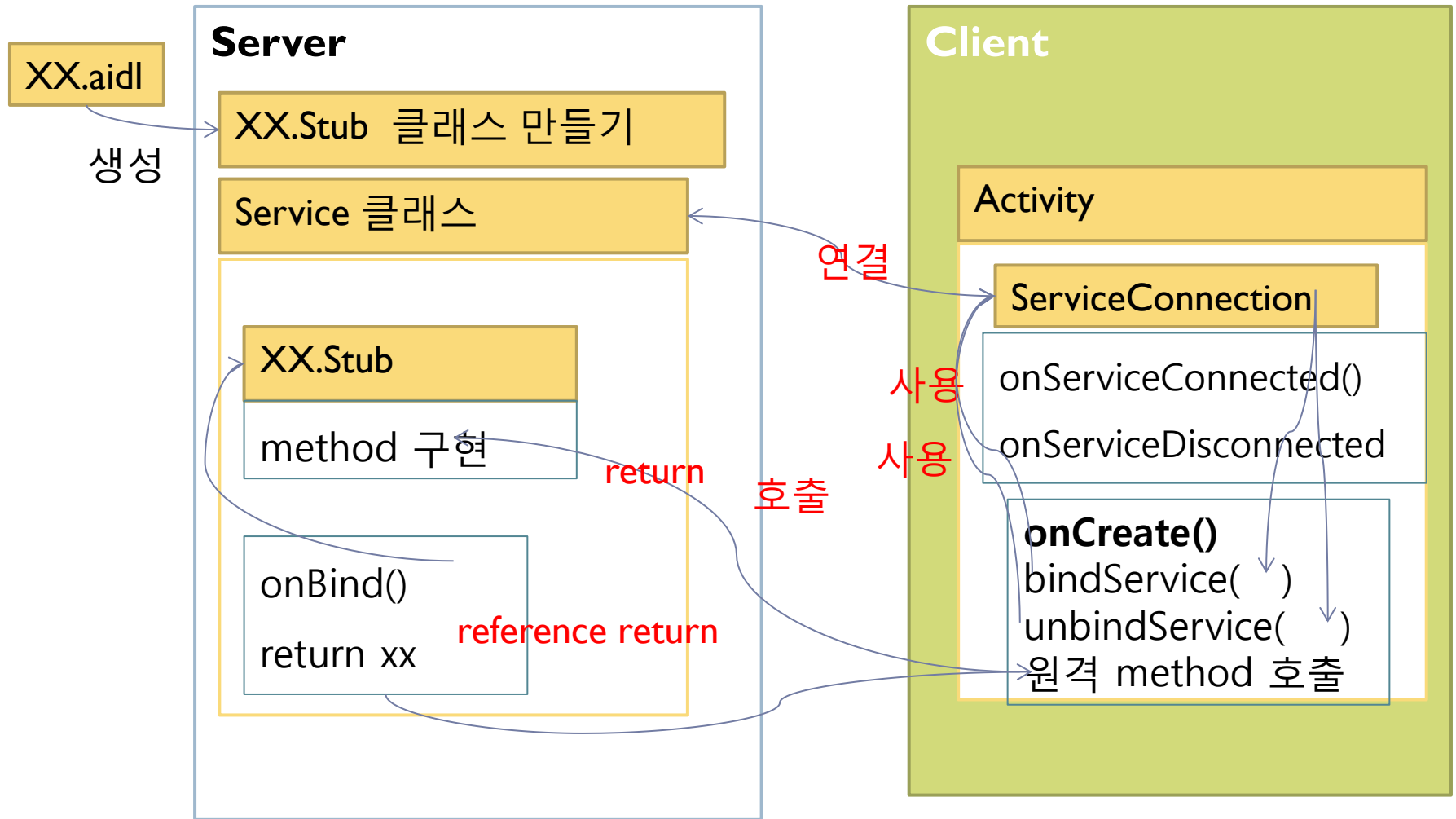
```
<service android:name = ".StrService">  
</service>
```



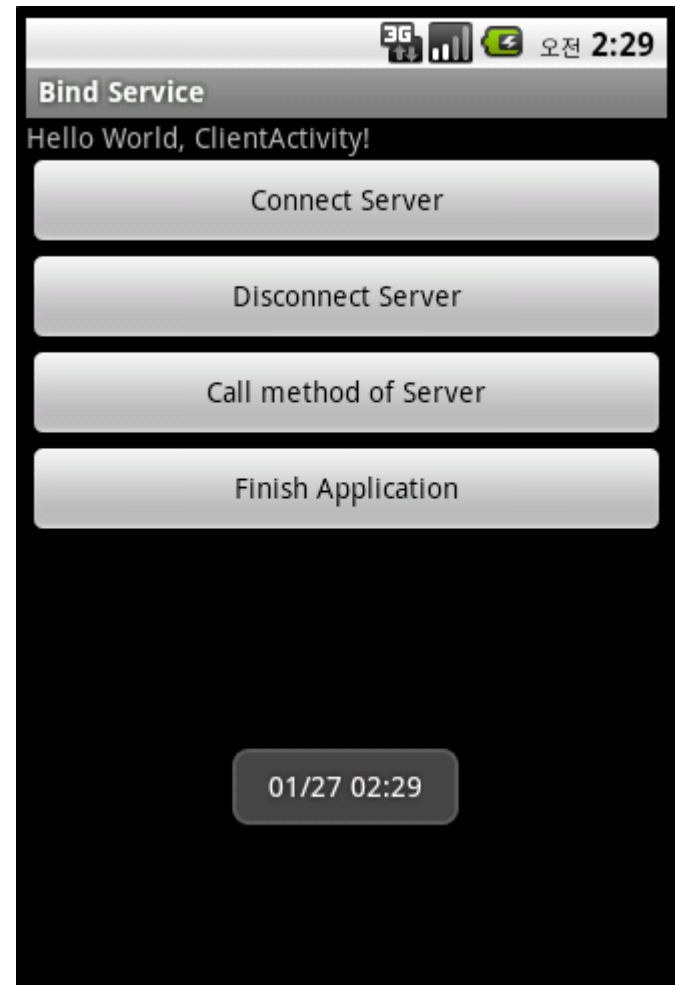
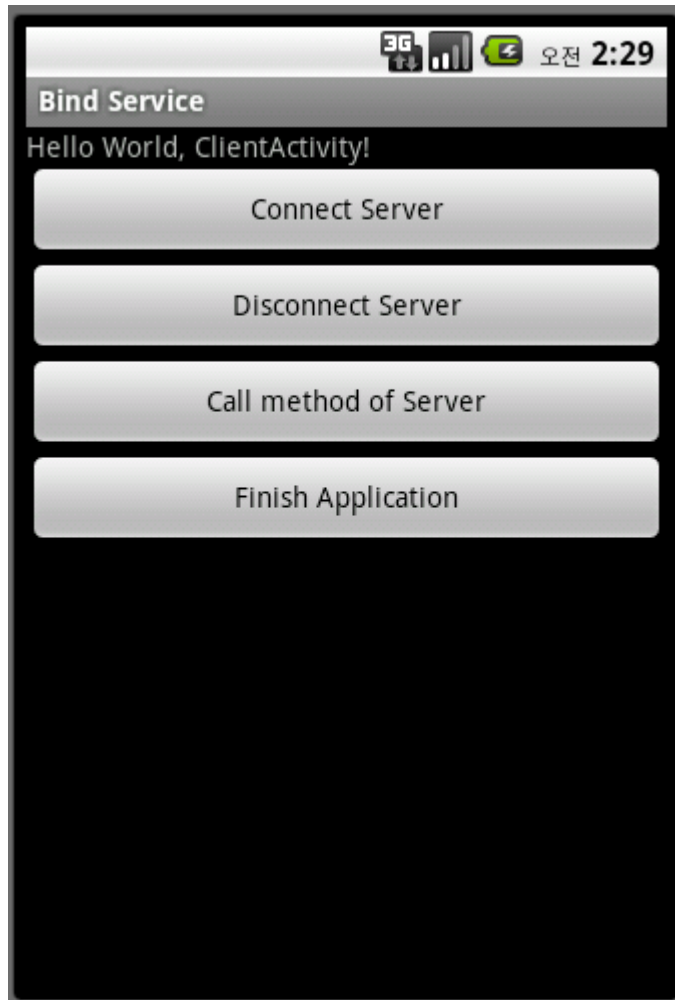
IPC



IPC



IPC의 예



Messenger

준비된 AIDL인 Messenger를 사용하자!

Messenger 클래스

스레드 간 / 프로세스간 통신

- Handler 클래스 : 서로 다른 스레드 간 메시지를 주고 받을 수 있음
- Messenger 클래스 : 서로 다른 어플리케이션, 서로 다른 프로세스에 존재하는 Handler로 메시지를 주고 받을 수 있음

Messenger 클래스

- 특정 Handler 인스턴스의 레퍼런스를 갖고 있으며, 이를 이용하여 해당 Handler로 메시지를 보낼 수 있습니다. 이를 이용하여 프로세스간 메시지 기반 커뮤니케이션을 수행할 때 활용될 수 있습니다.
- Messenger는 특정 Handler를 감싸는 클래스로, Messenger가 Parcelable 인터페이스를 구현하고 있음
- Handler 자체는 다른 프로세스로 넘겨 줄 수 없지만, 이를 Messenger로 감싸면, 해당 Handler로 원격에서 메시지를 전할 수 있는 Messenger 인스턴스를 생성할 수 있고, 이 Messenger 인스턴스는 한 프로세스에서 다른 프로세스로 이동할 수 있음
- AIDL을 정의하지 않고도 간편하게 Message에 기반한 IPC 작업을 수행할 수 있음



Messenger 클래스

Messenger 클래스 활용

- 만일, 서비스 측에서 클라이언트 쪽으로 응답 메시지를 보내야하는 경우 Message의 replyTo 필드를 활용함
 1. 클라이언트에 응답 메시지를 수신할 Handler 인스턴스를 구현할 수, 해당 Handler를 참조하는 Messenger를 구성함
 2. 서비스 쪽으로 메시지를 전달할 때 Message replyTo 필드에 해당 Messenger 인스턴스를 첨부함
 3. Message를 수신한 서비스 핸들러는 replyTo 필드의 Messenger 인스턴스를 통해 응답 메시지를 클라이언트로 전달할 수 있음



Messenger 클래스

Messenger 클래스 동작 원리

- 일종의 안드로이드 플랫폼에서 미리 정의해 둔 AIDL로 볼 수 있음
- Messenger 프레임워크 소스를 보면 Messenger 인터페이스를 정의한 `IMessenger.aidl`이 존재하며, `send(Message msg)` 라는 IPC 메서드를 구현하고 있음
- 이 AIDL 인터페이스의 실제 stub 클래스는 Handler 클래스 아래 `private` 클래스로 정의되어 있어 자기 자신의 메시지 큐에 메시지를 넣도록 구현되어 있음



MessengerTest – MessengerClient.java

```
public class MessengerClient extends Activity implements
View.OnClickListener {
    Messenger mService = null;

    private ServiceConnection mConnection = new ServiceConnection()
    {
        @Override
        public void onServiceConnected(ComponentName className,
        IBinder service){
            mService = new Messenger(service);
        }

        @Override
        public void onServiceDisconnected(ComponentName arg0) {
            // TODO Auto-generated method stub
            mService = null;
        }
    };
};
```

handler의 구현

MessengerTest – MessengerClient.java

```
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
  
    bindService (new Intent(MessengerClient.this,  
        MessengerService.class), mConnection,  
        Context.BIND_AUTO_CREATE);  
    ....  
}  
public void onClick(View view) {  
    Message msg = Message.obtain();  
    ...  
    msg.arg1 = Integer.parseInt(input.getText().toString());  
    msg.replyTo = oMessenger;  
    try {  
        mService.send(msg);  
    } catch (RemoteException e) { }  
}
```

handler의 구현



MessengerTest – MessengerService.java

```
public class MessengerService extends Service{

    Messenger mMessenger = new Messenger(new Handler(){
        public void handleMessage(Message msg){
            super.handleMessage(msg);
            Message retmsg = Message.obtain();
            ...
            try {
                msg.replyTo.send(retmsg);
            } catch (RemoteException e) { }
        }
    });

    @Override
    public IBinder onBind(Intent intent) {
        return mMessenger.getBinder();
    }
}
```

handler의 구현



Parcelable 인터페이스

Parcelable 인터페이스

- 객체를 전달 가능한 형태로 포장하는 것
- Parcel : 프로세스간 통신을 위한 메시지 저장 장치
- Bundle : 꾸러미 입출력 기능을 제공하므로 같은 프로세스의 세션간 데이터 저장 및 복구에도 사용함

