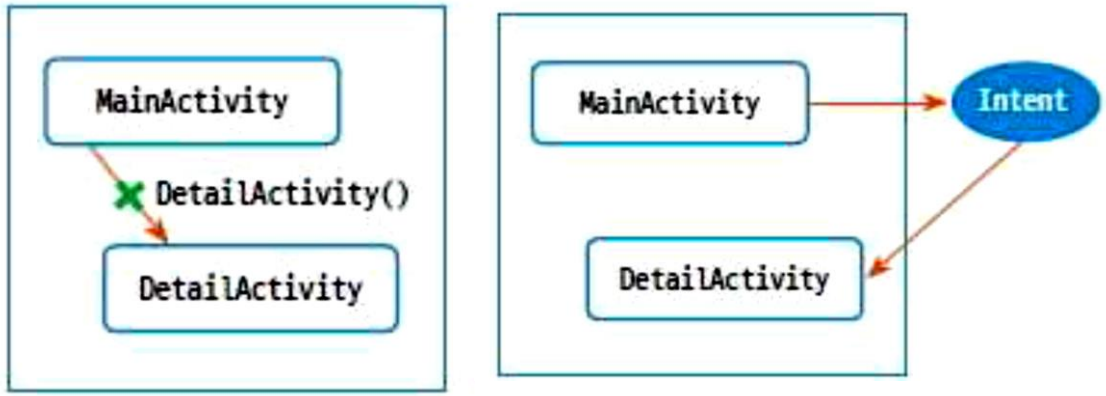
11. 액티비티 컴포넌트

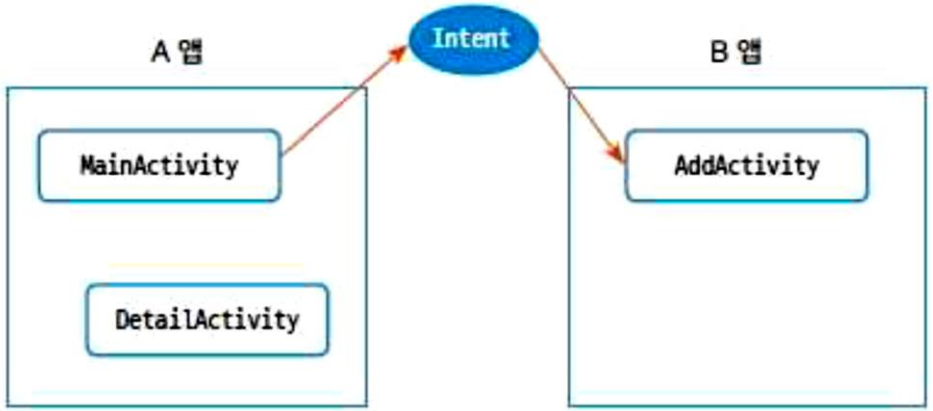
1. 인텐트 이해하기
2. 액티비티 생명주기
3. 액티비티 제어
4. 태스크 관리
5. 액티비티 ANR 문제와 코루틴
6. 할 일 목록 앱 만들기

1. 인텐트 이해하기

인텐트란?

- 인텐트는 한마디로 '컴포넌트를 실행하려고 시스템에 전달하는 메시지'라고 정의
- 안드로이드의 컴포넌트 클래스라면 개발자가 코드에서 직접 생성해서 실행할 수 없다.
- 시스템에서 인텐트의 정보를 분석해서 그에 맞는 컴포넌트를 실행해 줌.
- 외부 앱의 컴포넌트와 연동할 때도 인텐츠 사용.





1. 인텐트 이해하기

- startActivity() 함수가 인텐트를 시스템에 전달
- Intent 생성자의 매개변수는 클래스 타입 레퍼런스 정보

• MainActivity와 DetailActivity 등록

```
<activity
    android:name=".DetailActivity"
    android:exported="true" />
<activity
    android:name=".MainActivity"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

• 인텐트를 시스템에 전달

```
val intent: Intent = Intent(this, DetailActivity::class.java)
startActivity(intent)
```

1. 인텐트 이해하기

- 사후 처리가 필요 없을 때는 startActivity() 함수를 사용
- 사후 처리가 필요할 때는 startActivityForResult() 함수나 ActivityResultLauncher를 사용
- start ActivityForResult() 함수로 인텐트를 발생시켜 액티비티를 실행하는 방법은 안드로이드 초기 버전부터 지금까지 (1~12버전) 잘 사용되어 왔지만, 안드로이드 11 버전이 나올 즈음부터는 androidx의 ActivityResultLauncher를 사용하라고 권장

- startActivityForResult() 함수를 이용하는 전통적인 방법
- startActivityForResult() 함수로 시작할 때 두 번째 매개변수는 개발자가 정하는 요청 코드(requestCode)이며 인텐트를 식별하는 값

• 결과를 돌려받는 액티비티 시작

```
startActivityForResult(intent, 10)
```

1. 인텐트 이해하기

- 화면을 되돌릴 때는 finish() 함수를 이용
- finish() 함수는 현재 화면에 보이는 액티비티를 종료해 달라고 시스템에 요청
- setResult() 함수로는 결과를 어떻게 되돌릴지 지정
- RESULT_OK, 아니면 RESULT_CANCELED 등 상수를 지정

• 결과와 화면 되돌리기

```
intent.putExtra("resultData", "world")
setResult(RESULT_OK, intent)
finish()
```

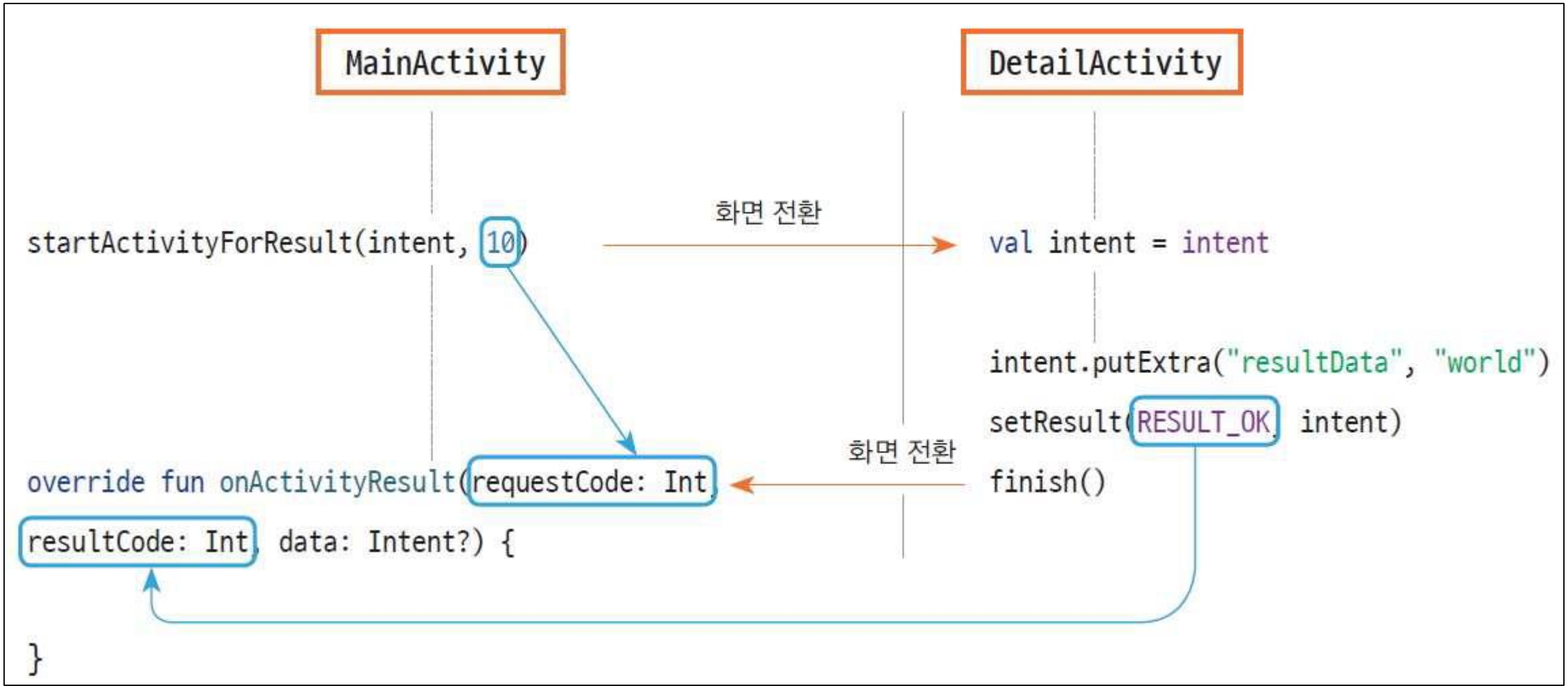
1. 인텐트 이해하기

- 결과가 되돌아와서 다시 자신이 화면에 출력되면 onActivityResult() 함수가 자동으로 호출
- requestCode: 인텐트를 시작한 곳에서 인텐트를 구분하려고 설정한 요청 코드
- resultCode: 인텐트로 실행된 곳에서 돌려받은 결과 코드
- data: 인텐트 객체. 이 객체에 결과 데이터가 있음.

• 결과를 돌려받은 후 처리

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    if (requestCode == 10 && resultCode == Activity.RESULT_OK) {
        val result = data?.getStringExtra("resultData")
    }
}
```

1. 인텐트 이해하기


MainActivity
DetailActivity
화면 전환
startActivityForResult(intent, 10)
val intent = intent
intent.putExtra("resultData", "world")
setResult(RESULT_OK, intent)
화면 전환
override fun onActivityResult(requestCode: Int,
finish()
resultCode: Int, data: Intent?) {
}

1. 인텐트 이해하기

- 액티비티 화면 되돌리기 – ActivityResultLauncher
  - ActivityResultLauncher 는 액티비티에서 다양한 결과에 대한 사후 처리를 제공

  - Contract
    - Contract는 ActivityResultLauncher로 실행될 요청을 처리하는 역할
    - Contract는 ActivityResultContract를 상속받은 서브 클래스
    - PickContact : 선택한 연락처의 Uri 획득
    - RequestPermission : 권한 요청, 허락 여부 파악
    - RequestMultiplePermissions : 여러 권한을 동시에 요청
    - StartActivityForResult : 인텐트 발생, 액티비티 실행 결과 획득
    - TakePicturePreview : 사진 촬영 후 비트맵 획득
    - TakePicture : 사진 촬영, 저장, 비트맵 획득

1. 인텐트 이해하기

- ActivityResultLauncher
  - ActivityResultLauncher는 registerForActivityResult() 함수로 만드는 객체이며 함수의 매개변수에 실제 작업자인 Contract 객체와 결과를 처리하는 Callback 객체를 등록
- launch
  - launch 함수를 호출하는 순간 ActivityResultLauncher에 등록된 Contract 객체가 실행

• ActivityResultLauncher 생성

```
val requestLauncher: ActivityResultLauncher<Intent> = registerForActivityResult(
    ActivityResultContracts.StartActivityForResult())
    {
        val resultData = it.data?.getStringExtra("result")
        binding.mainResultView.text = "result : $resultData"
    }
```

Contract

Callback

• ActivityResultLauncher 실행

```
val intent: Intent = Intent(this, DetailActivity::class.java)
requestLauncher.launch(intent)
```

1. 인텐트 이해하기

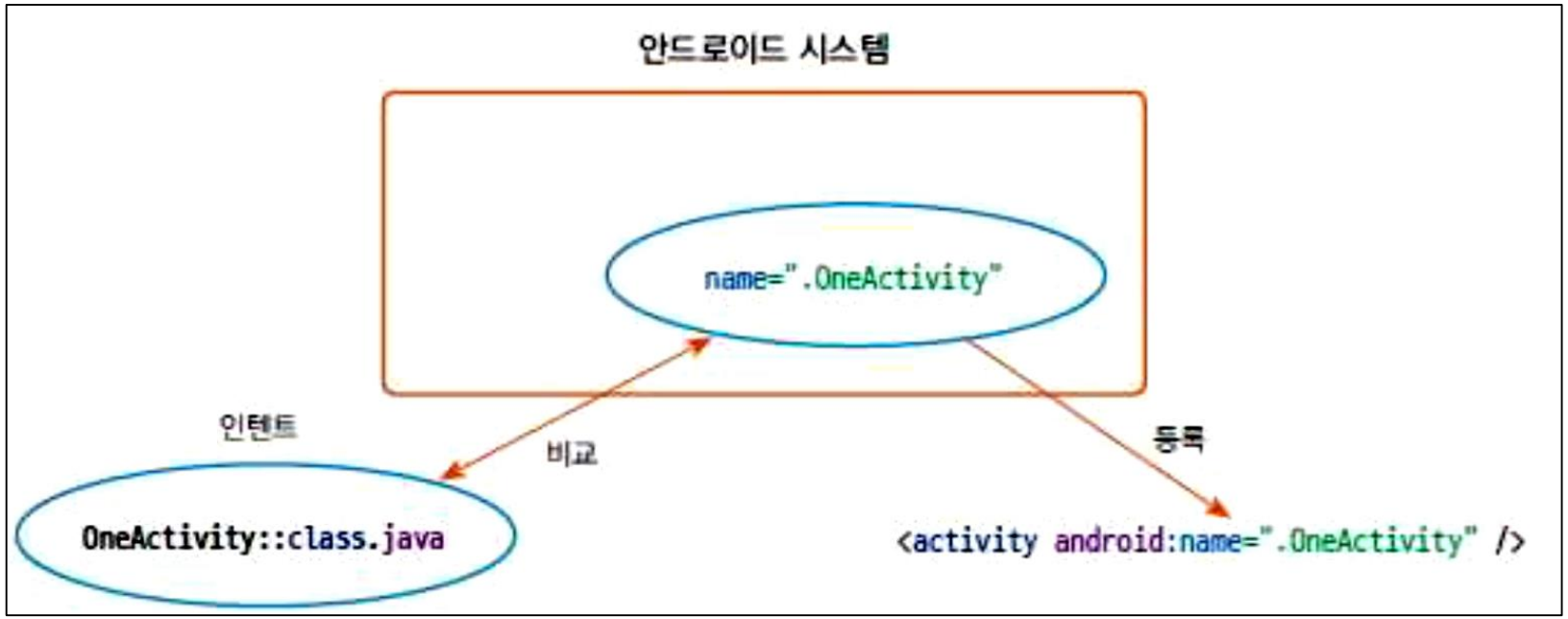
- 인텐트 필터
  - 인텐트는 실행할 컴포넌트 정보를 어떻게 설정하는지에 따라 2가지로 나뉩니다.
    - 명시적 인텐트: 클래스 타입 레퍼런스 정보를 활용한 인텐트
    - 암시적 인텐트: 인텐트 필터 정보를 활용한 인텐트
  - 클래스 타입 레퍼런스를 이용하는 것을 명시적 인텐트
  - 암시적 인텐트는 매니페스트 파일에 선언된 인텐트 필터를 이용

• 암시적 인텐트

```
<activity android:name=".OneActivity" />
<activity android:name=".TwoActivity"
    android:exported="true">
    <intent-filter>
        <action android:name="ACTION_EDIT" />
    </intent-filter>
</activity>
```

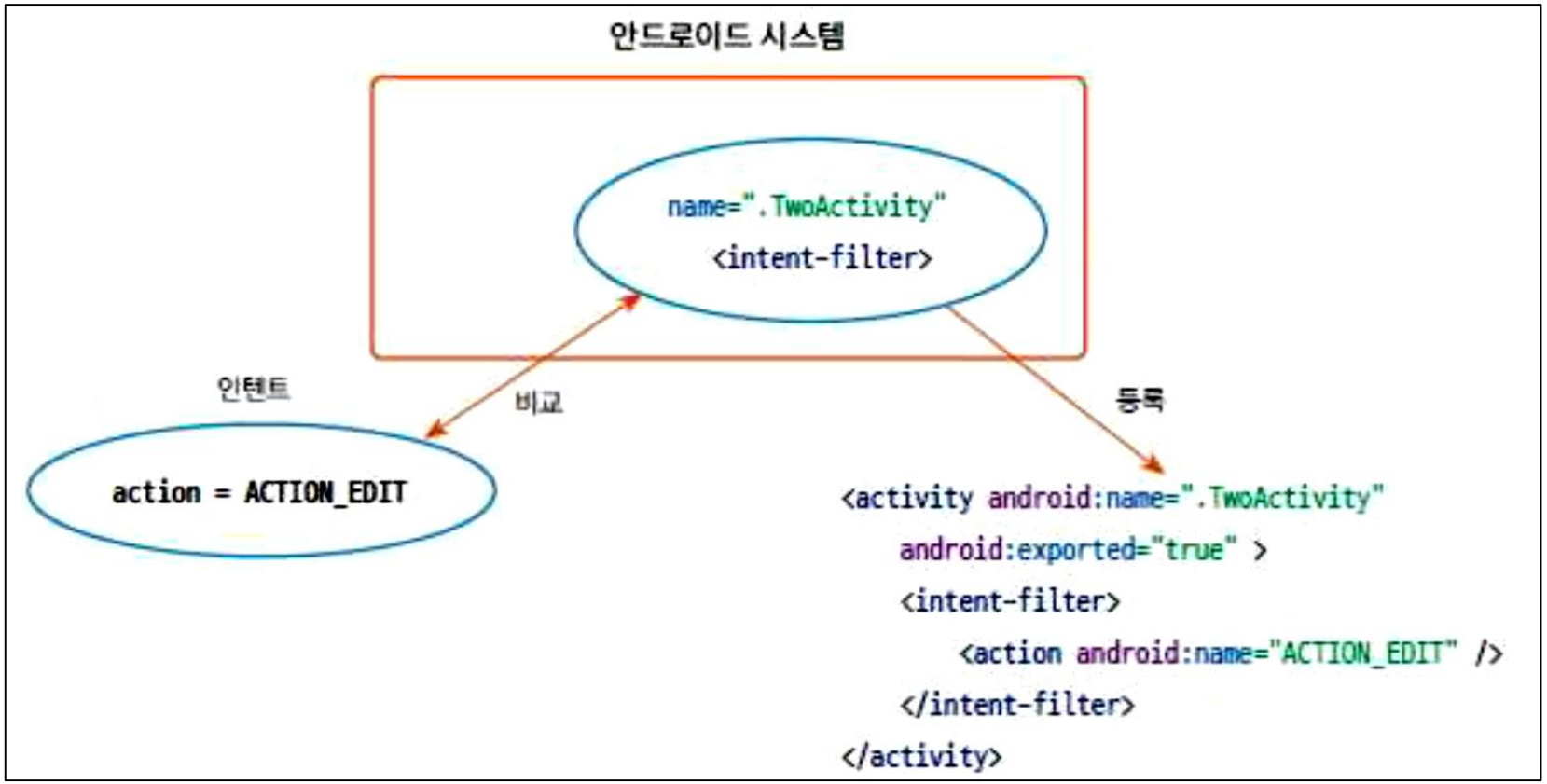
1. 인텐트 이해하기

- 명시적 인텐트



1. 인텐트 이해하기

- 암시적 인텐트



1. 인텐트 이해하기

- 인텐트 필터 하위에는 <action>, <category>, <data> 태그를 이용해 정보를 설정
- 어떤 정보를 설정할 것인지는 개발자의 선택
  - <action>: 컴포넌트의 기능을 나타내는 문자열.
  - <category>: 컴포넌트가 포함되는 범주를 나타내는 문자열.
  - <data>: 컴포넌트에 필요한 데이터 정보

• 자동으로 만들어지는 메인 액티비티

```
<activity android:name=".MainActivity">
    android:exported="true" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

• 외부 앱과 연동하는 인텐트 필터 설정(메인 액티비티 외)

```
<activity android:name=".TwoActivity"
    android:exported="true">
    <intent-filter>
        <action android:name="ACTION_EDIT" />
        <category android:name="android.intent.category.DEFAULT"/ >
        <data android:scheme="http" />
    </intent-filter>
</activity>
```

1. 인텐트 이해하기

• 인텐트의 프로퍼티를 이용하는 방법

```
val intent = Intent()
intent.action = "ACTION_EDIT"
intent.data = Uri.parse("http://www.google.com")
startActivity(intent)
```

• 인텐트의 생성자를 이용하는 방법

```
val intent = Intent("ACTION_EDIT", Uri.parse("http://www.google.com"))
startActivity(intent)
```

• mimeType 설정

```
<activity android:name=".TwoActivity">
    <intent-filter>
        <action android:name="ACTION_EDIT" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="image/*" />
    </intent-filter>
</activity>
```

• 타입 정보 설정

```
val intent = Intent("ACTION_EDIT")
intent.type = "image/*"
startActivity(intent)
```

1. 인텐트 이해하기

액티비티 인텐트 동작 방식

- 실행할 액티비티가 없을 때, 1개일 때, 여러 개일 때 시스템이 어떻게 처리하는지?
  - 없을 때: 인텐트를 시작한 곳에 오류가 발생.
  - 1개일 때: 문제없이 실행.
  - n개일 때: 사용자 선택으로 하나만 실행.

```
android.content.ActivityNotFoundException: No Activity found to handle Intent {
act=ACTION_HELLO }
```

1. 인텐트 이해하기

- 인텐트로 실행할 액티비티가 없을 수도 있는 상황을 고려

```
val intent = Intent("ACTION_HELLO")
try {
    startActivity(intent)
} catch (e: Exception) {
    Toast.makeText(this, "no app...", Toast.LENGTH_SHORT).show()
}
```

- 액티비티가 여러 개라면 사용자가 선택하는대로 하나만 실행

```
val intent = Intent(Intent.ACTION_VIEW, Uri.parse("geo:37.7749,127.4194"))
startActivity(intent)
```

- 특정 앱의 액티비티를 실행하고 싶다면 해당 앱의 패키지명을 지정

```
val intent = Intent(Intent.ACTION_VIEW, Uri.parse("geo:37.7749,127.4194"))
intent.setPackage("com.google.android.apps.maps")
startActivity(intent)
```

1. 인텐트 이해하기

패키지 공개 상태

- 안드로이드 11(API 레벨 30) 버전부터는 앱의 패키지 공개 상태를 지정하지 않으면 외부 앱의 패키지(앱의 식별자) 정보에 접근할 수 없게 됨.
- 외부 앱을 연동하더라도 패키지 정보를 활용하지 않는다면 아무런 문제가 없음.
- 다음과 같은 함수를 사용할 때는 패키지 공개 상태에 따라 영향을 받음.
  - PackageManager.getPackageInfo()
  - PackageManager.queryIntentActivities()
  - Intent.resolveActivity()
  - PackageManager.getInstalledPackages()
  - PackageManager.getInstalledApplications()
  - bindService()

1. 인텐트 이해하기

- PackageManager의 getPackageInfo() 함수를 이용해 매개변수에 지정한 패키지 문자열로 식별되는 앱의 정보를 가져오는 코드

```
val packageInfo = packageManager.getPackageInfo("com.example.test_outter", 0)
val versionName = packageInfo.versionName
```

- 안드로이드 10 버전까지 별문제 없이 실행되었지만 11 버전부터는 다음과 같은 오류가 발생

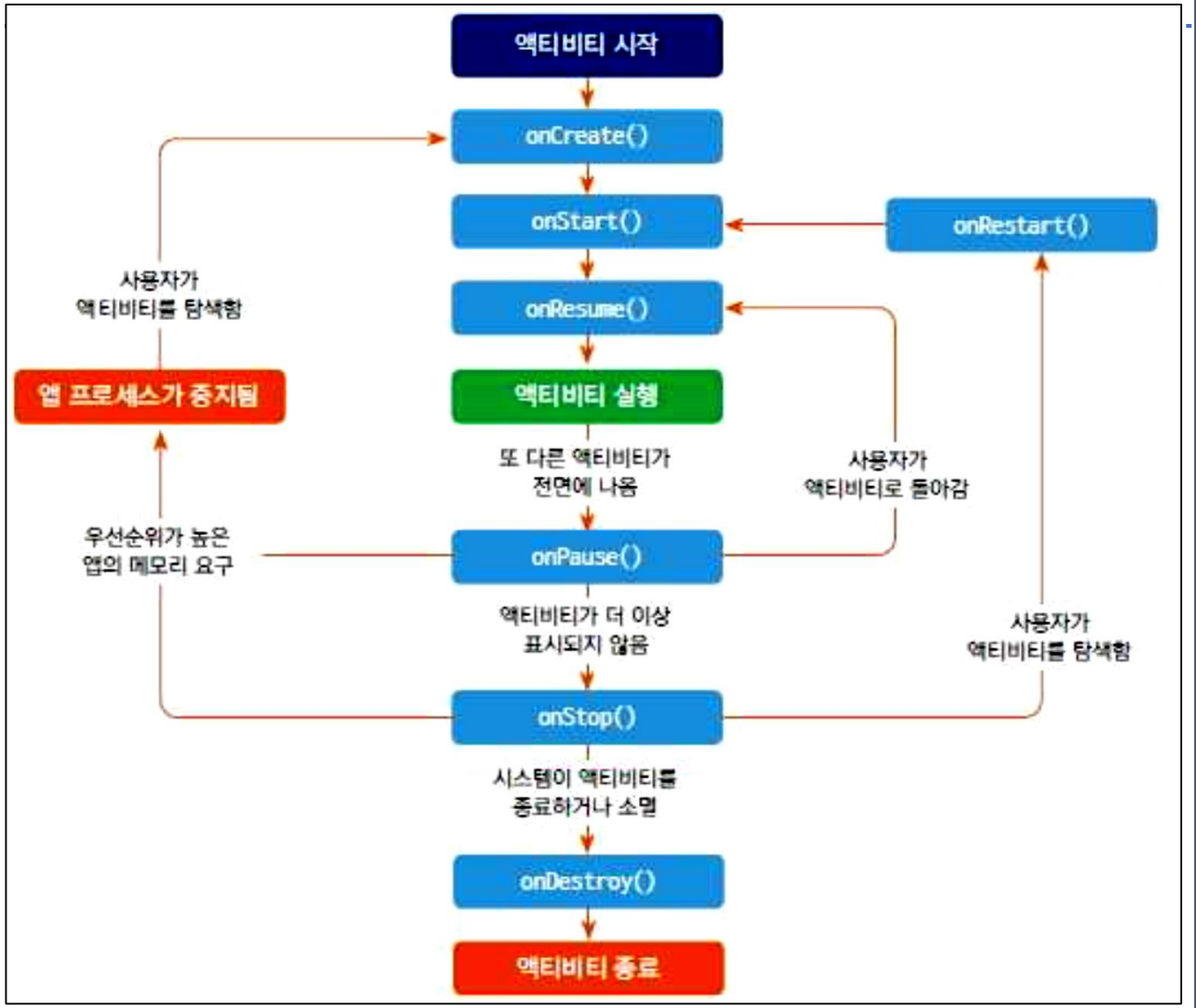
```
Caused by: android.content.pm.PackageManager$NameNotFoundException: com.example.test_outter
```

- 정상으로 실행하려면 매니페스트 파일에 외부 앱의 정보에 접근하겠다고 선언

```
<manifest ... 생략 ... >
    <queries>
        <package android:name="com.example.test_outter" />
    </queries>
    (... 생략 ...)
</manifest>
```

2. 액티비티 생명주기

- 액티비티의 상태
  - 생명주기란 액티비티가 생성되어 소멸하기까지의 과정
  - 액티비티의 상태는 다음처럼 크게 3가지로 구분
    - 활성: 액티비티 화면이 출력되고 있고 사용자가 이벤트를 발생시킬 수 있는 상태
    - 일시 정지: 액티비티의 화면이 출력되고 있지만 사용자가 이벤트를 발생시킬 수 없는 상태
    - 비활성: 액티비티의 화면이 출력되고 있지 않는 상태



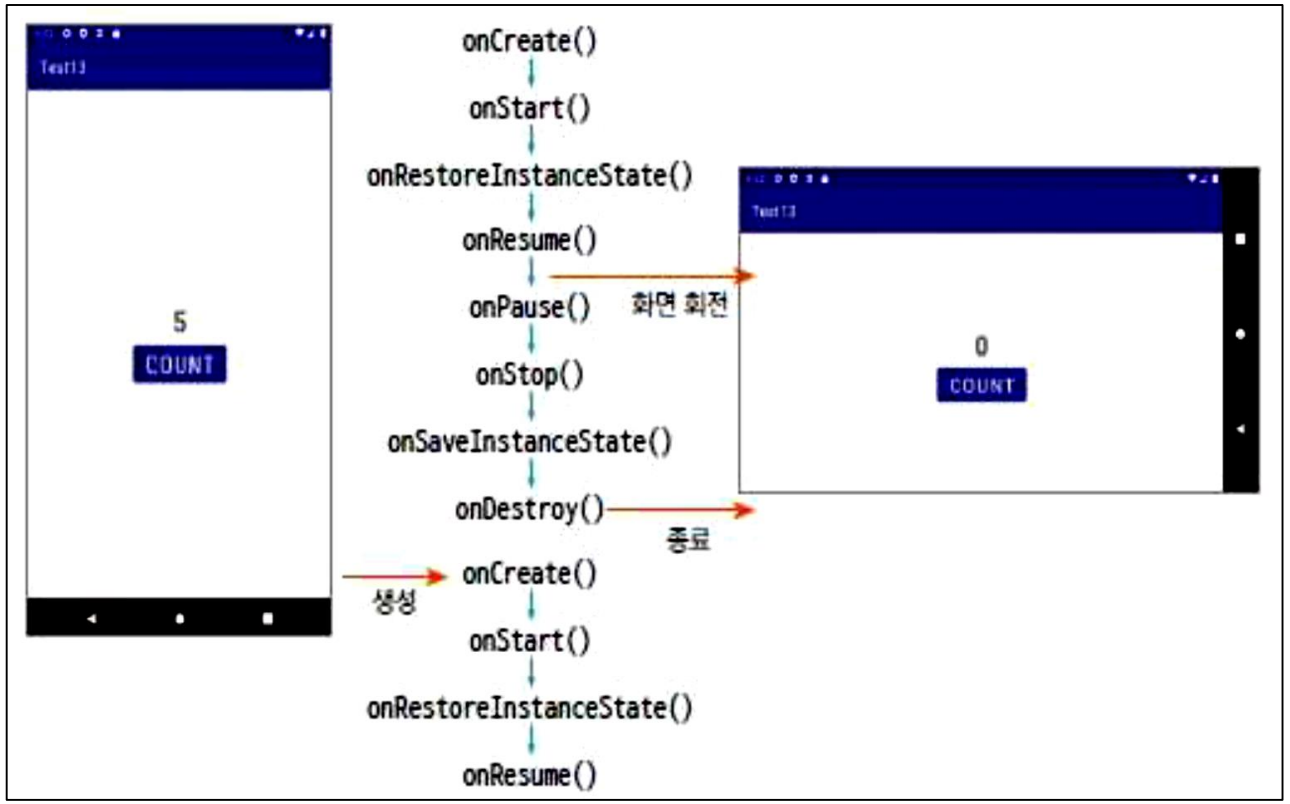
2. 액티비티 생명주기

- 활성 상태
  - 액티비티가 실행되어 화면에 나오고 사용자 이벤트를 처리할 수 있는 상태
  - 처음 실행된 액티비티는 onCreate() → onStart() → onResume() 함수까지 호출
- 일시 정지 상태
  - 일시 정지 상태는 onPause() 함수까지 호출된 상태
  - 액티비티가 화면에 보이지만 포커스를 잃어 사용자 이벤트를 처리할 수 없는 상태
- 비활성 상태
  - 비활성 상태란 액티비티가 종료되지 않고 화면에만 보이지 않는 상태
  - 활성 상태에서 비활성 상태가 되면 onPause() → onStop() 함수까지 호출
  - 액티비티가 종료된다는 것은 onDestroy()까지 호출되었다는 의미

2. 액티비티 생명주기

액티비티의 상태 저장

- 액티비티가 종료되면 객체가 소멸하므로 액티비티의 데이터는 모두 사라짐.
- 상태를 저장한다는 것은 액티비티가 종료되어 메모리의 데이터가 사라지더라도 다시 실행할 때 사용자가 저장한 데이터로 액티비티의 상태를 복원하겠다는 의미
- 화면을 회전하면 액티비티가 종료되었다가 나옴. 따라서 액티비티의 데이터는 초기화 됨.



2. 액티비티 생명주기

- 액티비티를 종료할 때 저장했다가 복원해야 할 데이터가 있다면 **Bundle이라는 객체에** 담아 주면 됨
- onCreate(), onSaveInstance State(), onRestoreInstanceState() 함수는 매개변수를 가지며 모두 Bundle 객체

• 번들 객체 사용

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
}

override fun onRestoreInstanceState(savedInstanceState: Bundle) {
    super.onRestoreInstanceState(savedInstanceState)
}

override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
}
```

2. 액티비티 생명주기

- onSaveInstanceState() 함수의 매개변수로 전달되는 Bundle에 데이터를 담아 주면 자동으로 데이터를 파일로 저장해 줌

```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    outState.putString("data1", "hello")
    outState.putInt("data2", 10)
}
```

번들에 데이터 저장

- 다시 액티비티가 생성되어 실행될 때 캐싱 파일이 있다면 그 내용을 읽어서 번들 객체에 담아 onCreate(), onRestoreInstanceState() 함수의 매개변수로 전달

```
override fun onRestoreInstanceState(savedInstanceState: Bundle) {
    super.onRestoreInstanceState(savedInstanceState)
    val data1 = savedInstanceState.getString("data1")
    val data2 = savedInstanceState.getInt("data2")
}
```

번들에 저장된 데이터 가져오기

3. 액티비티 제어

■ 소프트 키보드 제어하기

▪ 입력 매니저

- 특정한 순간에 키보드를 올리거나 내려야 할 수도 있음.
- InputMethodManager 클래스가 지원

```
public boolean hideSoftInputFromWindow(IBinder windowToken, int flags)
public boolean showSoftInput(View view, int flags)
public void toggleSoftInput(int showFlags, int hideFlags)
```

• 키보드 올리고 내리기

```
val manager = getSystemService(INPUT_METHOD_SERVICE) as InputMethodManager
binding.showInputButton.setOnClickListener {
    binding.editView.requestFocus()   // 뷰에 포커스 강제
    manager.showSoftInput(binding.editView, InputMethodManager.SHOW_IMPLICIT)
}
binding.hideInputButton.setOnClickListener {
    manager.hideSoftInputFromWindow(currentFocus?.windowToken,
        InputMethodManager.HIDE_NOT_ALWAYS)
}
```

3. 액티비티 제어

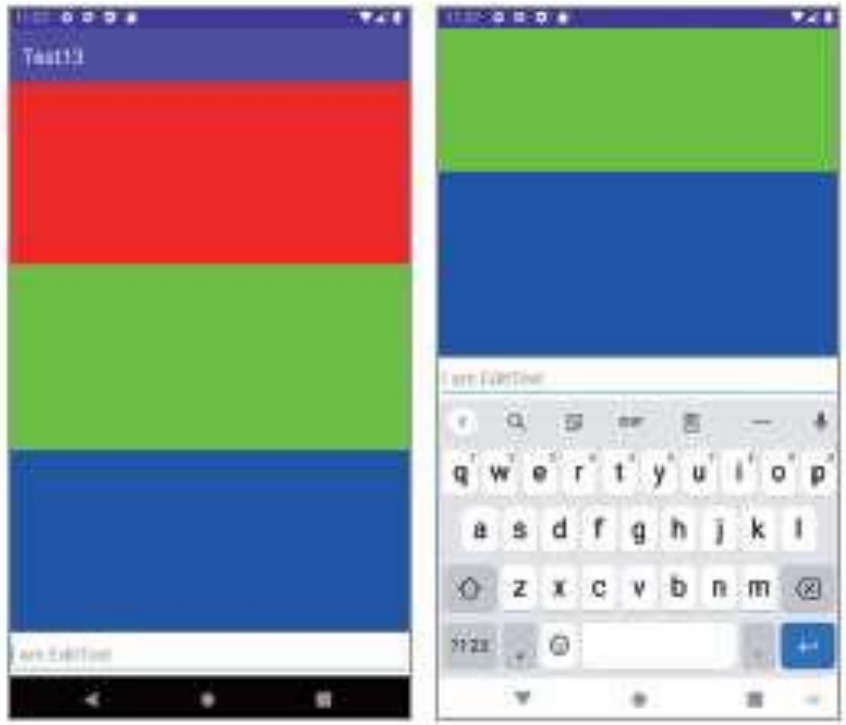
- 입력 모드 설정
- 키보드가 올라올 때 액티비티가 함께 위로 올라오게 할 수도 있고 액티비티의 크기를 조정할 수도 있음
- 매니페스트 파일에서 <activity> 태그의 windowSoftInputMode 속성을 이용
  - adjustPan: 키보드가 올라올 때 입력 에디트 텍스트에 맞춰 화면을 위로 올림
  - adjustResize: 키보드가 올라올 때 액티비티의 크기를 조정
  - adjustUnspecified: 상황에 맞는 옵션을 시스템이 알아서 설정. ← 기본값
  - stateHidden: 액티비티 실행 시 키보드가 자동으로 올라오는 것을 방지.
  - stateVisible: 액티비티 실행 시 키보드가 자동으로 올라옴
  - stateUnspecified: 시스템이 적절한 키보드 상태를 설정하거나 테마에 따름. ← 기본값

• 키보드 올라올 때 설정

```
<activity android:name=".SettingActivity" android:windowSoftInputMode="adjustPan">
```

3. 액티비티 제어

- 기본상태



- adjustResize

3. 액티비티 제어

▪ 방향과 전체 화면 설정하기

- 액티비티의 방향을 고정하고 싶다면 매니페스트 파일의 <activity> 태그의 screenOrienta tion 속성을 이용
- 값은 landscape나 portrait를 지정

• 화면 방향 고정하기

```
<activity android:name=".SettingActivity" android:screenOrientation="landscape">
```

3. 액티비티 제어

- 액티비티 코드에서 전체 화면으로 출력되게 설정
- API 레벨 29까지는 window.setFlags() 함수를 이용해 전체 화면을 지정
- API 레벨 30부터는 WindowInsetsController라는 클래스의 함수를 이용해 액티비티 창을 설정

• 전체 화면 출력 코드(API 레벨 30 이후)

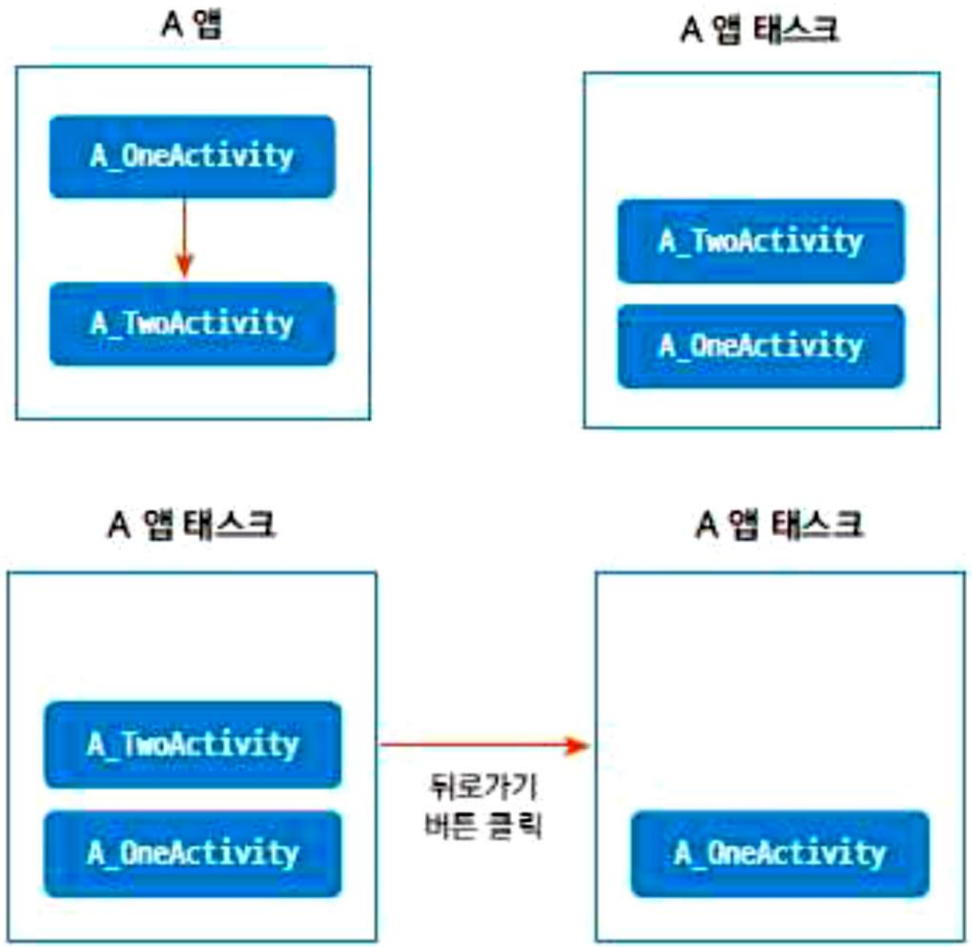
```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.R) {
    window.setDecorFitsSystemWindows(false)
    val controller = window.insetsController
    if (controller != null) {
        controller.hide(WindowInsets.Type.statusBars() or
        WindowInsets.Type.navigationBars())
        controller.systemBarsBehavior =
            WindowInsetsController.BEHAVIOR_SHOW_TRANSIENT_BARS_BY_SWIPE
    }
} else {
    window.setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
    WindowManager.LayoutParams.FLAG_FULLSCREEN)
}
```

4. 태스크 관리

시스템에서 태스크 관리

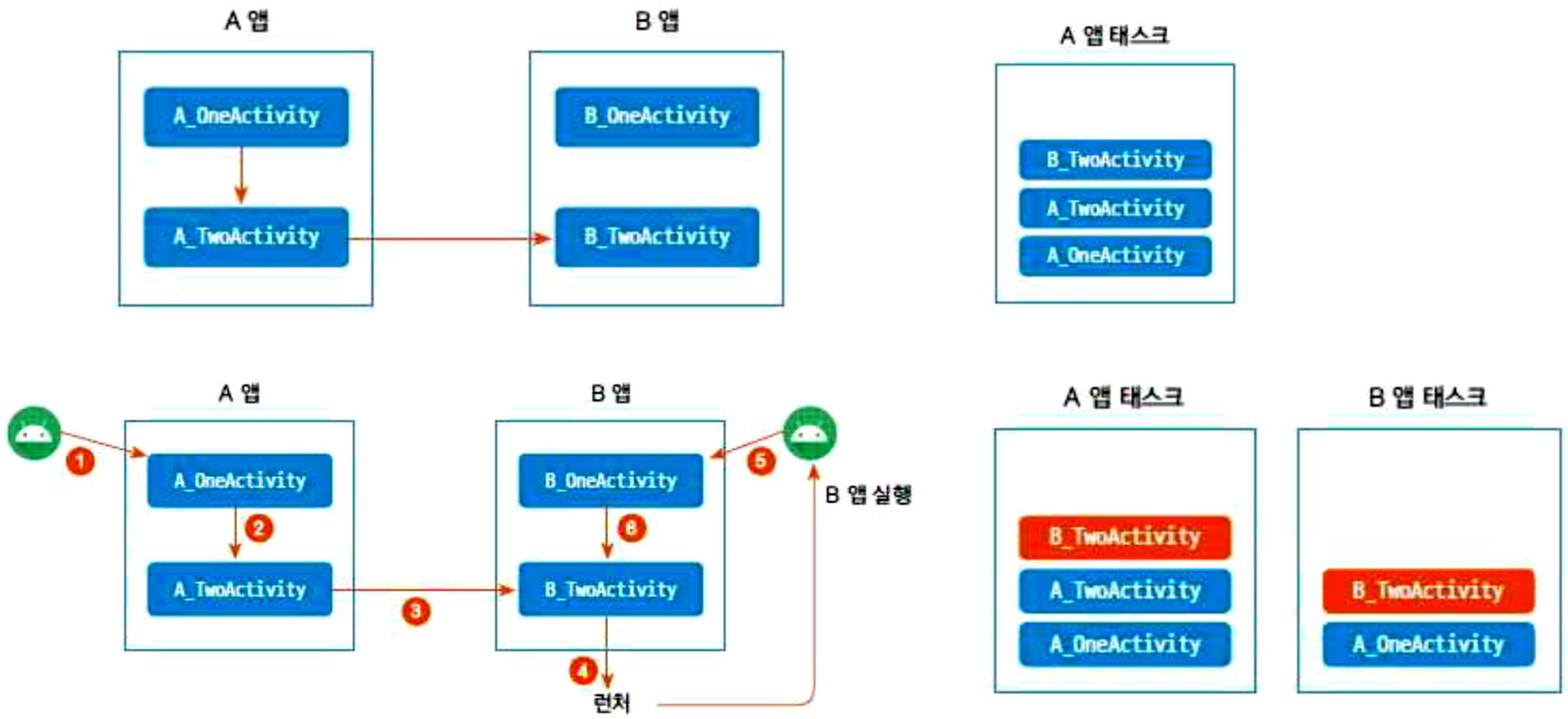
- 액티비티 태스크란 앱이 실행될 때 시스템에서 액티비티의 각종 정보를 저장하는 공간

- 사용자가 기기의 뒤로가기 버튼을 누르면 이 태스크에서 위쪽에 있는 액티비티를 종료



4. 태스크 관리

- 앱과 앱이 연동되어 실행되는 구조



4. 태스크 관리

태스크 제어

- 태스크에 제어하는 2가지 방법
  - 액티비티가 등록되는 매니페스트 파일의 <activity> 태그의 launchMode를 이용
  - 인텐트의 flags 정보를 설정하여 제어
- <activity> 태그의 launchMode 속성으로 실행 모드를 설정

• 매니페스트 파일에서 launchMode로 제어

```
<activity android:name=".TwoActivity" android:launchMode="singleTop">
```

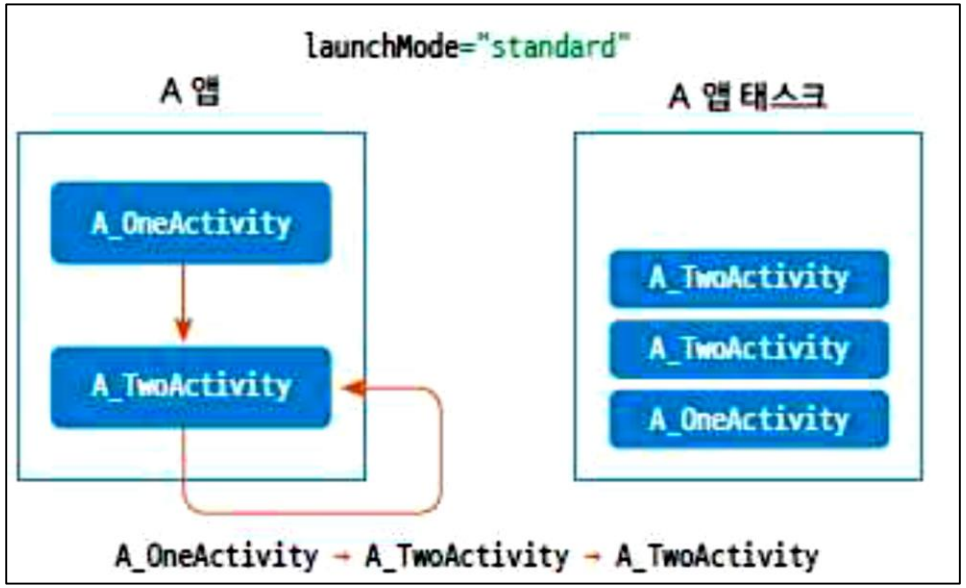
4. 태스크 관리

- 코드에서는 인텐트를 발생시키기 전에 인텐트의 flags 속성에 설정

• 코드에서 flags 속성으로 제어

```
val intent = Intent(this, TwoActivity::class.java)
intent.flags = Intent.FLAG_ACTIVITY_SINGLE_TOP
startActivity(intent)
```

- 스탠더드로 설정
  - 기본값
  - 인텐트가 발생하면 항상 객체가 생성되고 태스크에 등록

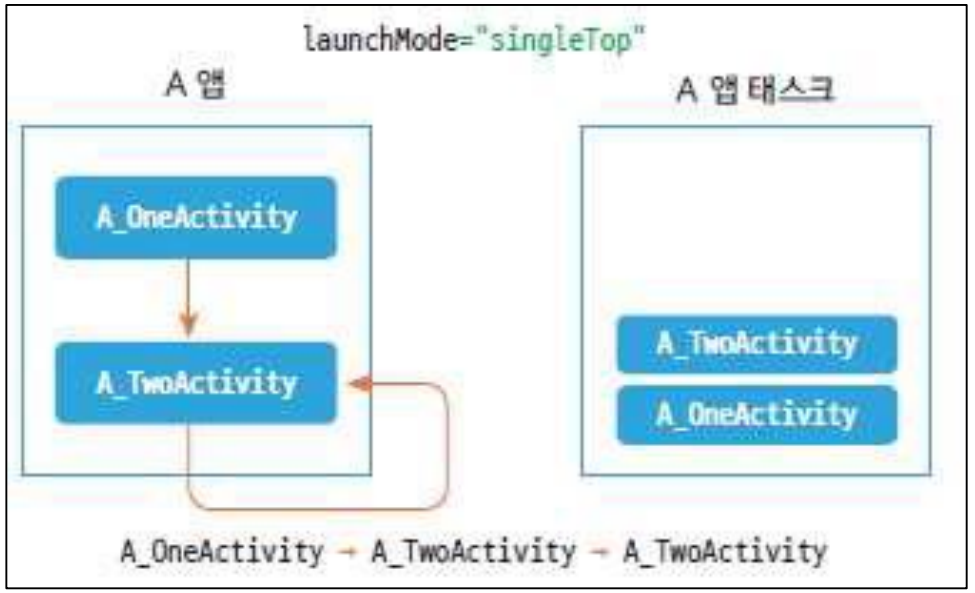


4. 태스크 관리

- 싱글 톱으로 설정
  - 액티비티 정보가 태스크의 위쪽에 있을 때 인텐트가 발생해도 객체를 생성하지 않음.
  - 기존 객체의 onNewIntent() 함수가 자동으로 호출

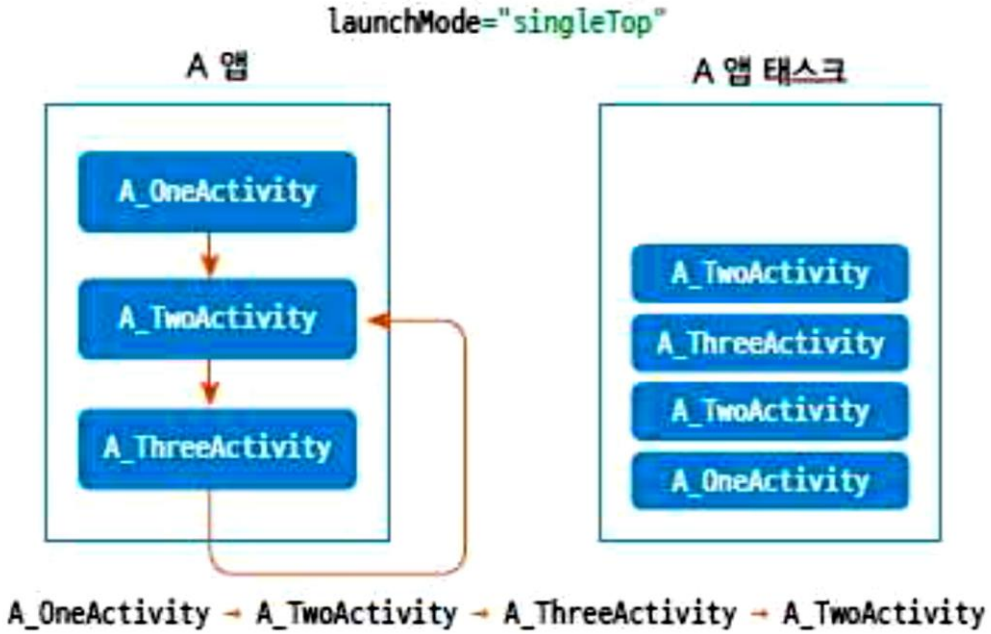
• onNewIntent() 함수

```
override fun onNewIntent(intent: Intent?) {
    super.onNewIntent(intent)
}
```



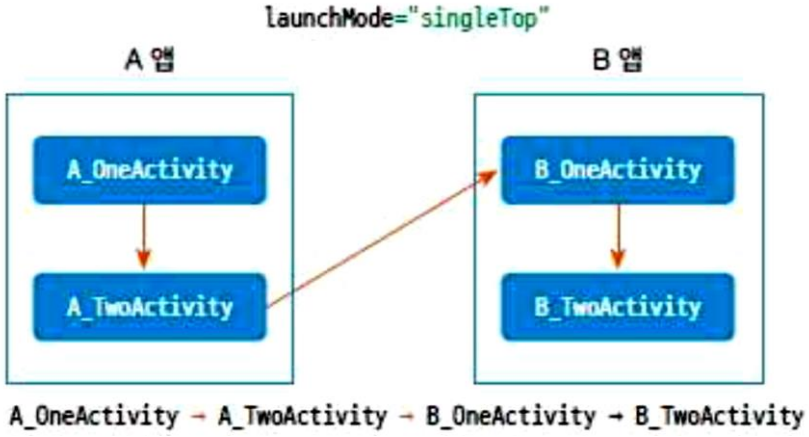
4. 태스크 관리

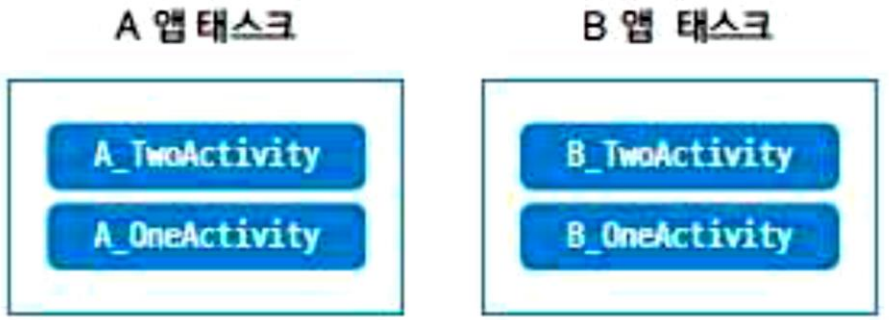
- singleTop으로 설정한 액티비티 객체가 태스크 위쪽이 아닌 곳에 있는 경우



4. 태스크 관리

- **싱글 태스크로 설정**
  - singleTask로 설정하면 새로운 태스크를 만들어 등록
  - single Task 설정은 같은 앱에서는 적용되지 않으며 다른 앱의 액티비티를 인텐트로 실행할 때에만 적용



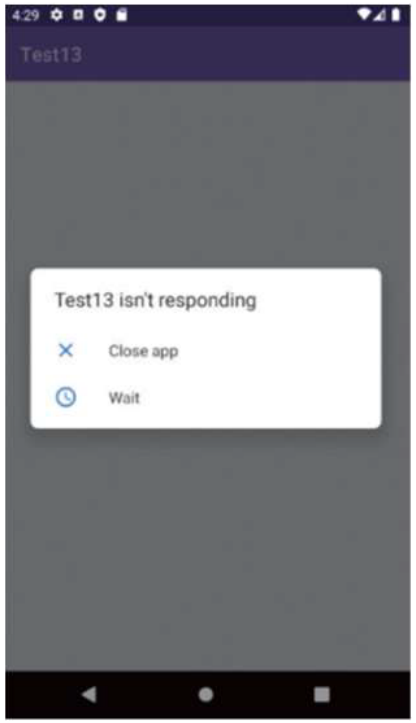


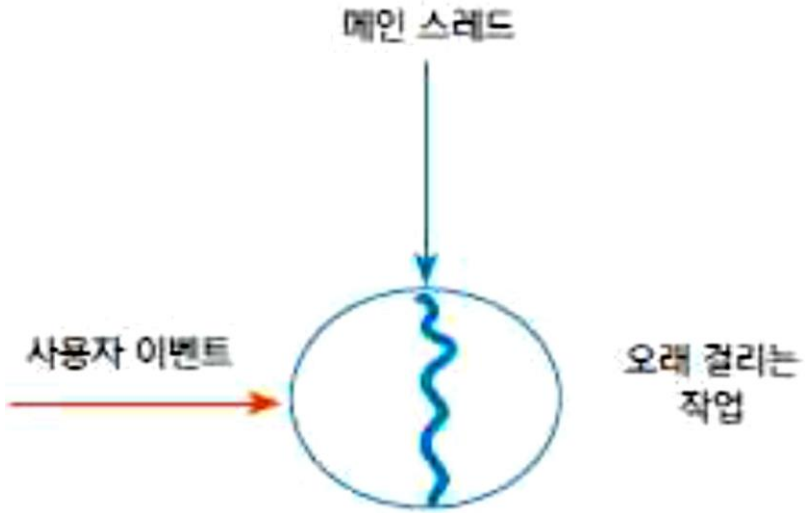
4. 태스크 관리

- **싱글 인스턴스로 설정**
  - singleInstance로 설정하면 싱글 태스크처럼 새로운 태스크를 만들어 등록하는데, 그 태스크에는 해당 설정이 적용된 액티비티 하나만 등록

5. 액티비티 ANR 문제와 코루틴

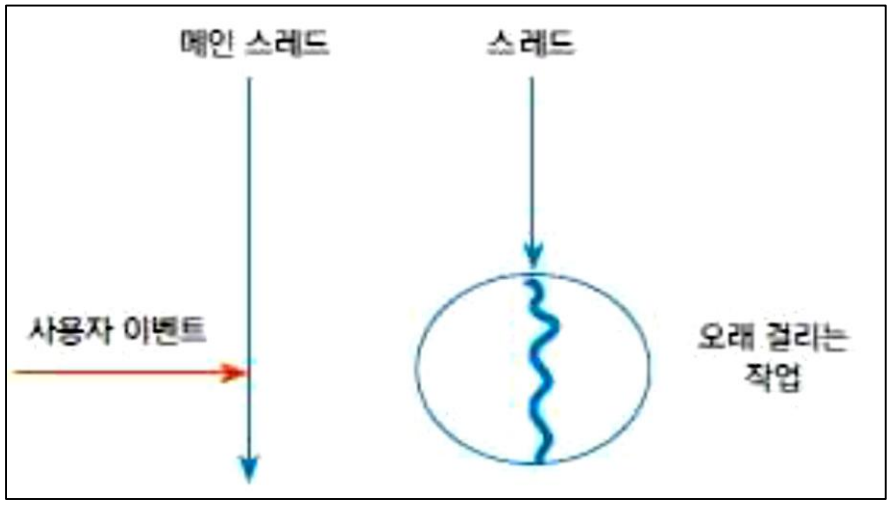
- **ANR 문제란?**
  - ANR은 액티비티가 응답하지 않는 오류 상황
  - 시스템에서 액티비티를 실행하는 수행 흐름을 메인 스레드 또는 화면을 출력하는 수행 흐름이라는 의미에서 UI 스레드라고 함.

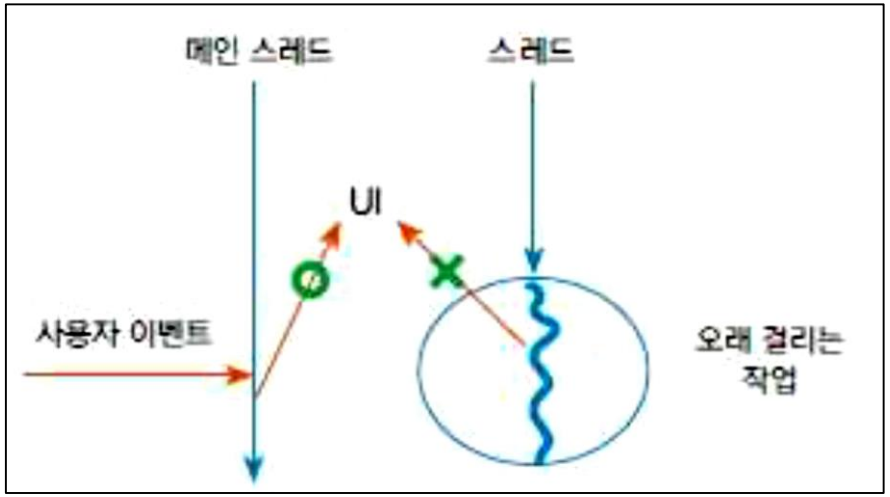




5. 액티비티 ANR 문제와 코루틴

- ANR 문제를 해결하는 방법은 액티비티를 실행한 메인 스레드 이외에 실행 흐름(개발자 스레드)을 따로 만들어서 시간이 오래 걸리는 작업을 담당하게 하면 됨.
- 이 방법으로 대처하면 ANR 오류는 해결되지만 화면을 변경할 수 없다는 또 다른 문제발생





5. 액티비티 ANR 문제와 코루틴

- **코루틴으로 ANR 오류 해결**
  - 코루틴이란?
  - 비동기 경량 스레드
  - 프로그래밍 언어에서 제공하는 기능
  - 코루틴을 사용하라고 권하면서 다음과 같은 장점이 있다고 소개
    - 경량
    - 메모리 누수가 적음
    - 취소 등 다양한 기능을 지원
    - 많은 제트팩 라이브러리에 적용되어 있음

5. 액티비티 ANR 문제와 코루틴

- 5. 액티비티 ANR 문제와 코루틴

- ANR 오류가 발생 코드

• 코루틴 등록

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.9'
```

• 시간이 오래 걸리는 작업 예

```
var sum = 0L
var time = measureTimeMillis {
    for (i in 1..2_000_000_000) {
        sum += i
    }
}
Log.d("kkang","time : $time")
binding.resultView.text = "sum : $sum"
```

▶ 실행 결과

5. 액티비티 ANR 문제와 코루틴

- ANR 오류를 해결하고자 스레드-핸들러 구조로 다시 작성

• 스레드-핸들러 구조로 작성한 소스

```
val handler = object: Handler() {
    override fun handleMessage(msg: Message) {
        super.handleMessage(msg)
        binding.resultView.text = "sum : ${msg.arg1}"
    }
}
```

```
thread {
    var sum = 0L
    var time = measureTimeMillis {
        for (i in 1..2_000_000_000) {
            sum += i
        }
        val message = Message()
        message.arg1 = sum.toInt()
        handler.sendMessage(message)
    }
    Log.d("kkang", "time : $time")
}
```

5. 액티비티 ANR 문제와 코루틴

- 코드를 코루틴으로 작성
  - 코루틴을 구동하려면 먼저 스코프를 준비
  - 스코프에서 코루틴을 구동
  - 코루틴 스코프는 CoroutineScope를 구현한 클래스의 객체
  - 직접 구현할 수도 있고 Global Scope, ActorScope, ProducerScope 등 코틀린 언어가 제공하는 스코프를 이용할 수도 있음.
- 디스패처는 이 스코프에서 구동한 코루틴이 어디에서 동작해야 하는지를 나타냄
  - Dispatchers.Main: 액티비티의 메인 스레드에서 동작하는 코루틴을 만듬
  - Dispatchers.IO: 파일에 읽거나 쓰기 또는 네트워크 작업 등에 최적화
  - Dispatchers.Default: CPU를 많이 사용하는 작업을 백그라운드에서 실행

• 코루틴으로 작성한 소스

```
val channel = Channel<Int>()
val backgroundScope = CoroutineScope(Dispatchers.Default + Job())
backgroundScope.launch {
    var sum = 0L
    var time = measureTimeMillis {
        for (i in 1..2_000_000_000) {
            sum += i
        }
    }
    Log.d("kkang", "time : $time")
    channel.send(sum.toInt())
}
val mainScope = GlobalScope.launch(Dispatchers.Main) {
    channel.consumeEach {
        binding.resultView.text = "sum : $it"
    }
}
```

백그라운드에서 동작(시간이 오래 걸리는 작업)

메인 스레드에서 동작(화면에 결괏값 표시)

실습 : 할 일 목록 앱 만들기

- 1단계. 모듈 생성과 빌드 그래들 설정하기
  - Ch13_Activity라는 이름으로새로운 모듈
  - 그래들파일에 뷰바인딩을 사용하도록 설정
- 2단계. 할 일 등록 액티비티 생성하기
  - [New → Activity → Empty Activity]
  - 액티비티 이름에 AddActivity를 입력
- 3단계. 리소스 & 소스 파일 복사하기
  - res 디렉터리 아래에 drawable, layout, menu 디렉터리를 현재 모듈의 res 디렉터리로 복사
  - 소스가 든 디렉터리에서 AddActivity.kt, Main Activity.kt, MyAdapter.kt 파일을 현재 모듈의 소스 영역에 복사

- 4단계. 할 일 등록 액티비티 작성하기
  - AddActivity.kt 파일을 열어 내용을 추가
- 5단계. 메인 액티비티 작성하기
  - MainActivity.kt 파일을 열고 내용을 추가
- 6단계. 앱 실행하기