Android
002 Build Tools
Gradle

1. Compile

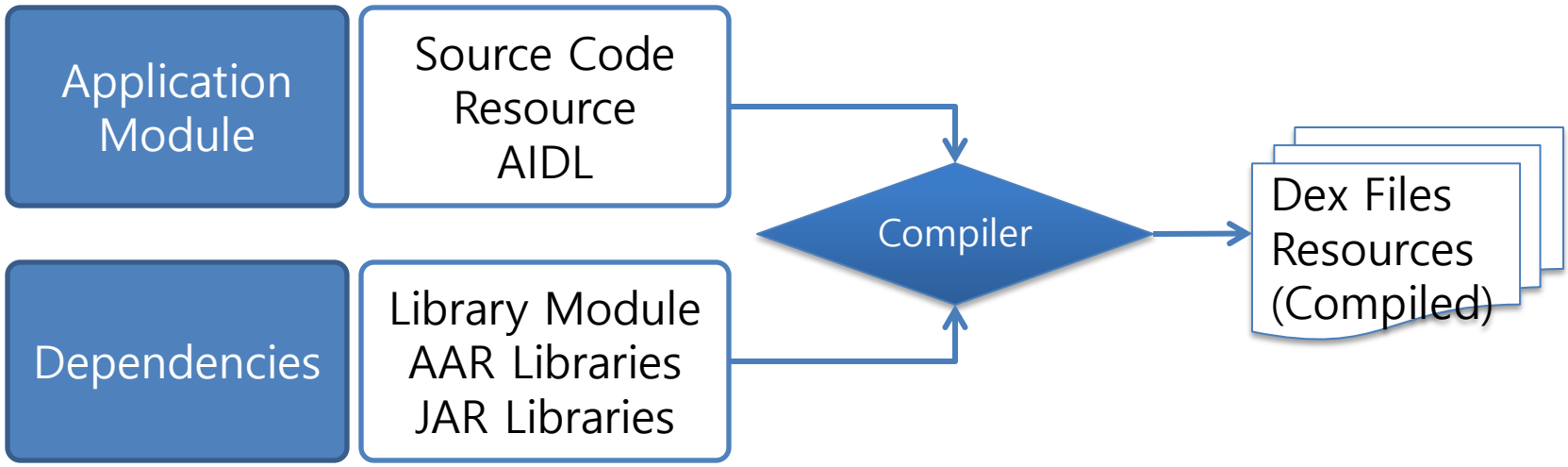
- 리눅스 상에서의 Compile 및 Build 과정

Linux Build = Compile + Link
(실행파일 생성)

Compile — 소스코드 -> 기계어

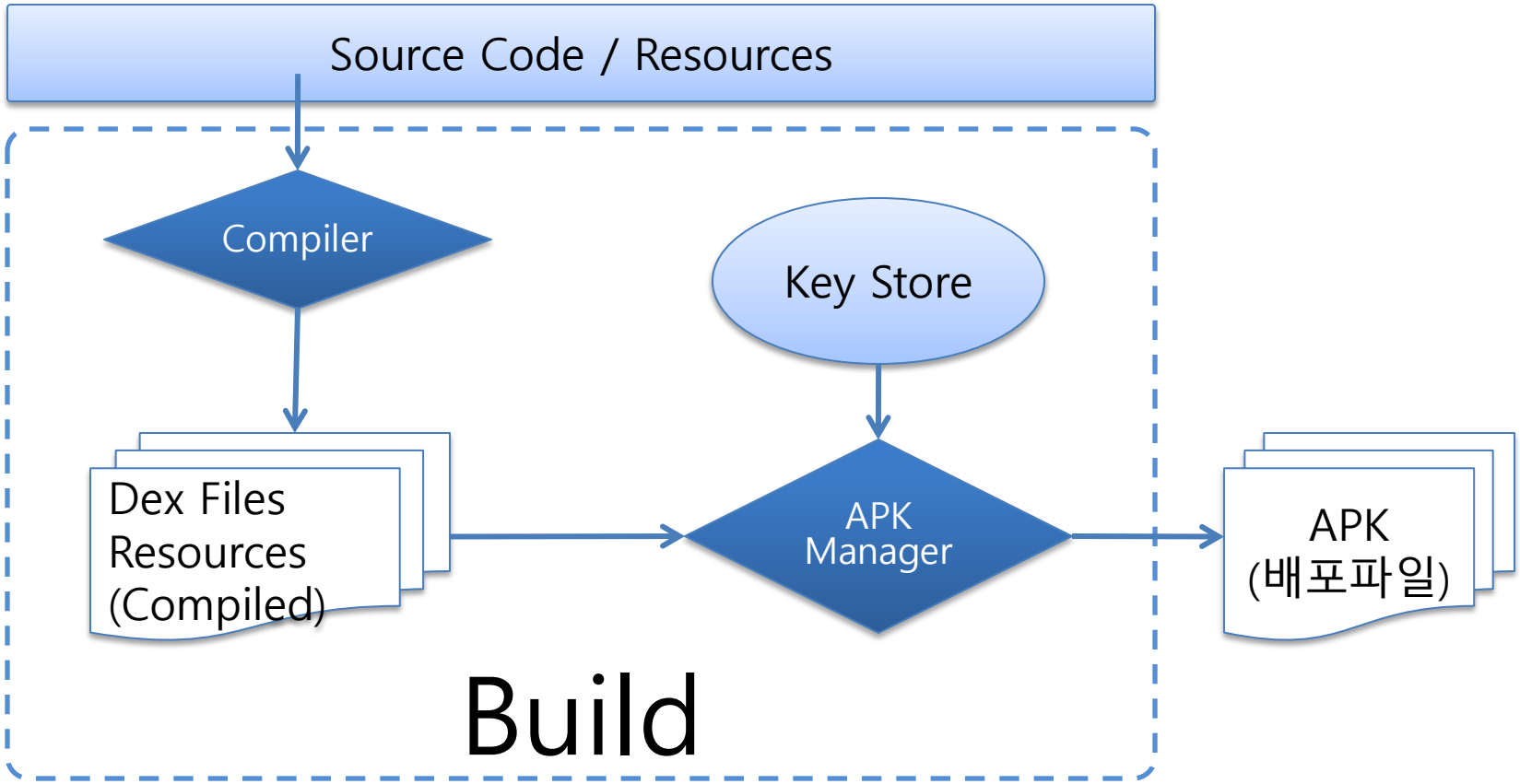
Link — 기계어 + 라이브러리 연결

- 안드로이드 에서의 Compile



2. Build

- 컴파일을 포함하는 개념으로 컴파일된 자원들을 패키징 하여 배포가능한 파일로 만든다



3. Build Tools

- Java에서는 아래 3가지의 빌드 도구가 가장 많이 사용된다

Make

리눅스 빌드툴 - 소스코드를 실행파일로 만든다
(플랫폼 의존성이 있다)

Ant

플랫폼에 독립적이면서 Java IDE를 지원하는 최초의 빌드 툴이다. xml 구조의 설정파일을 사용하며 대부분의 IDE에서 기본적으로 지원하고 있다. (의존성 관리도구가 없다)

Maven

빌드도구로 시작하였으나 현재는 의존성 관리위주로 사용되고 있으며, 국내 Java 진영에서는 가장 범용적으로 사용되고 있다. (기본 규칙(포맷)을 벗어나면 처리가 어렵다)

하지만 앞에서 밝힌바와 같이, 주로 의존성 관리툴로 사용되기 때문에 개발자 입장에서 어려운점은 없었음

Gradle

초기 Gradle 의 복잡성 문제로 -> Maven 으로 대거 개발진영이 이동하였으나 -> 최근 다시 Gradle을 사용하는 추세 (그리고 Google의 전폭적인 지원까지...)

4.1. Gradle 특징

- 아래는 안드로이드 스튜디오에서 지원하는 빌드도구인 Gradle의 특징이다

Groovy DSL	Groovy 스타일 언어지원 -> 컴파일없이 스크립트 실행 (Domain Specific Language 기반)
Gradle Wrapper	Gradle Wrapper 사용으로 실제 머신에 Gradle이 없어도 빌드가 가능하다
Ant + Maven	Ant의 유연성과 Maven의 단순함을 모두 가져왔다.

"Maven의 경우 복잡한 규칙으로 초보자의 학습장벽이 높다" 라고 gradle 진영에서 주장하고 있으나, 문법만을 놓고 본다면 Maven 이 더 쉽다.

* DSL (Domain Specific Language)이란?

1. 범용적으로 사용할 수는 없으나, 특정 영역의 문제(여기서는 빌드)를 해결하기에 적합
2. 비프로그래머가 프로그램을 쉽게 이해할 수 있는 고수준(high-level)의 언어
3. 해당 영역에 한에서는 직관적이며, 자연어에 가까운 형식으로 구현 가능

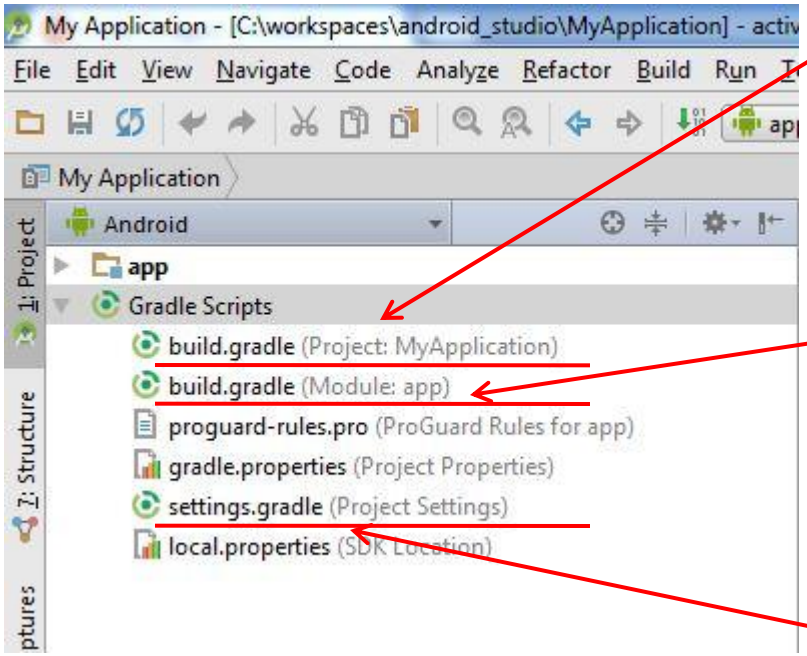
4.2. Gradle 을 사용하는 이유

- Gradle은 기존 빌드툴들의 단점들을 보완하는 형태로 발전하였다.

Wrapper	별도로 Gradle을 설치하지 않아도 된다. gradlew 명령어로 Gradle 바이너리를 자동으로 다운로드 하고 실행한다.
Multi-Project	빌드의 구조화를 제공하고, Multi-Project 빌드가 용이하며, 멀티 프로젝트에 있는 서브 프로젝트간 의존관계를 정의할 수 있다.
Script Component	Build 스크립트 자체를 컴포넌트화 해서 재사용할 수 있다. 기본적으로 안드로이드 스튜디오에서는 프로젝트 build.gradle과 앱 build.gradle 을 기본으로 구성해준다.
Compatibility	안드로이드 스튜디오에서는 프로젝트 로드시 gradle의 버전을 먼저 체크한다. 메이저 업그레이드가 있어도 하위호환성을 유지한다. 예를 들어 1.x > 2.x 로 버전업 되어도 호환성을 고려해서 업데이트 된다.

5. Gradle 설정파일

- 안드로이드 스튜디오에서 프로젝트를 생성하면 아래와 같이 Gradle Scripts 들이 자동으로 생성된다



build.gradle(Project: ...)

프로젝트 전체에 대한 gradle 설정파일

build.gradle(Module: ...)

단일 모듈에 대한 gradle 설정파일

settings.gradle

빌드하려는 모듈에 대한 정의

6. App 빌드를 위한 Gradle 설정

- gradle 설정은 기본적으로 안드로이드 플러그인의 사용선언으로 시작한다
- **build.gradle(Module)**

```
1 apply plugin: 'com.android.application'

2 android {
3     compileSdkVersion 23
      buildToolsVersion "24.0.1"
4     defaultConfig {
          applicationId "com.example.administrator.myapplication"
          minSdkVersion 15
      }
5     buildTypes {
          release {  } debug {  }  custom{  }
      }
6     productFlavors {
          full {    } demo {   }
      }
  }
7 dependencies {
      compile fileTree(dir: 'libs', include: ['*.jar'])
  }
```

① apply plugin
- 모듈 용도 구분

② android
- 앱 관련 설정

③ 앱 버전정의

④ defaultConfig
- 앱 기본설정

⑤ buildType
- 개발과정에 따른 구분

⑥ productFlavors
- 용도에 따른 구분

⑦ dependencies
- 의존성 라이브러리관리

6.1. defaultConfig 설정

- defaultConfig 는 아래 옵션항목들을 설정할 수 있다

속성 이름	DSL default	설명
versionCode	-1	버전코드. 마켓업데이트의 기준이 된다
versionName	null	버전이름. 1.x.x 형태로 기본 규칙이 있다
minSdkVersion	-1	앱 동작을 위한 OS 최소 요구사항
targetSdkVersion	-1	앱 동작시 기준이 되는 OS 버전
packageName	null	패키지명
testPackageName	null	일반적으로 패키지명 + ".test"
testInstrumentationRunner	null	안드로이드 테스트를 위한 Runner 설정
signingConfig	null	배포버전용 sing을 위한 키, 비밀번호 설정
proguardFile		코드 난독화를 위한 프로가드 설정
proguardFiles		proguardFile 의 복수형

6.2. buildTypes 설정

- defaultConfig 는 아래 옵션항목들을 설정할 수 있다

속성 이름	debug 기본값	release, custom 기본값	설명
debuggable	true	false	debuggable 설정
jniDebugBuild	false	false	jni 디버깅 여부
renderscriptDebugBuild	false	false	렌더스크립트 디버깅 여부
renderscriptOptimLevel	3	3	렌더스크립트 최적화 레벨
packageNameSuffix	null	null	패키지명 suffix (.test)
versionNameSuffix	null	null	버전명 suffix (...-alpha)
signingConfig	signing...debug	null	
zipAlign	false	true	Apk 최적화 (false 시 마켓등록 안됨)
runProguard	false	false	
proguardFile			
proguardFiles			

6.3. buildTypes 적용하기 1 - 코드적용

- buildType 에 따른 실 코드를 소스코드에는 아래와 같이 적용할 수 있다

```
apply plugin: 'com.android.application'

android {
    buildTypes {
        release {
            buildConfigField "String", "MYURL", "₩"http://real.seoul.go.kr₩""
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
        debug {
            buildConfigField "String", "MYURL", "₩"http://test.seoul.go.kr₩""
        }
    }
}
```

- 소스코드에서 사용하기

```
String myUrl = BuildConfig.MYURL;
```


6.4. buildTypes 적용하기 2 - 리소스적용

- buildType 에 따른 실 코드를 리소스에는 아래와 같이 적용할 수 있다

```
apply plugin: 'com.android.application'
android {
    buildTypes {
        release {
            resValue "string", "FB_KEY", "REAL_fjjhafdij02084ljfljasdoi034841ifkjkls931lkj"
        }
        debug {
            resValue "string", "FB_KEY", "TEST_fjjhafdij02084ljfljasdoi034841ifkjkls931lkj"
        }
    }
}
```

- 소스코드에서 사용하기

```
Resources res = getResources();
String resValue = res.getString(R.string.FB_KEY);
```

- 설정파일에서 사용하기

```
<meta-data android:name="com.facebook. ApplicationID" android:value="@string/FBKEY"/>
```

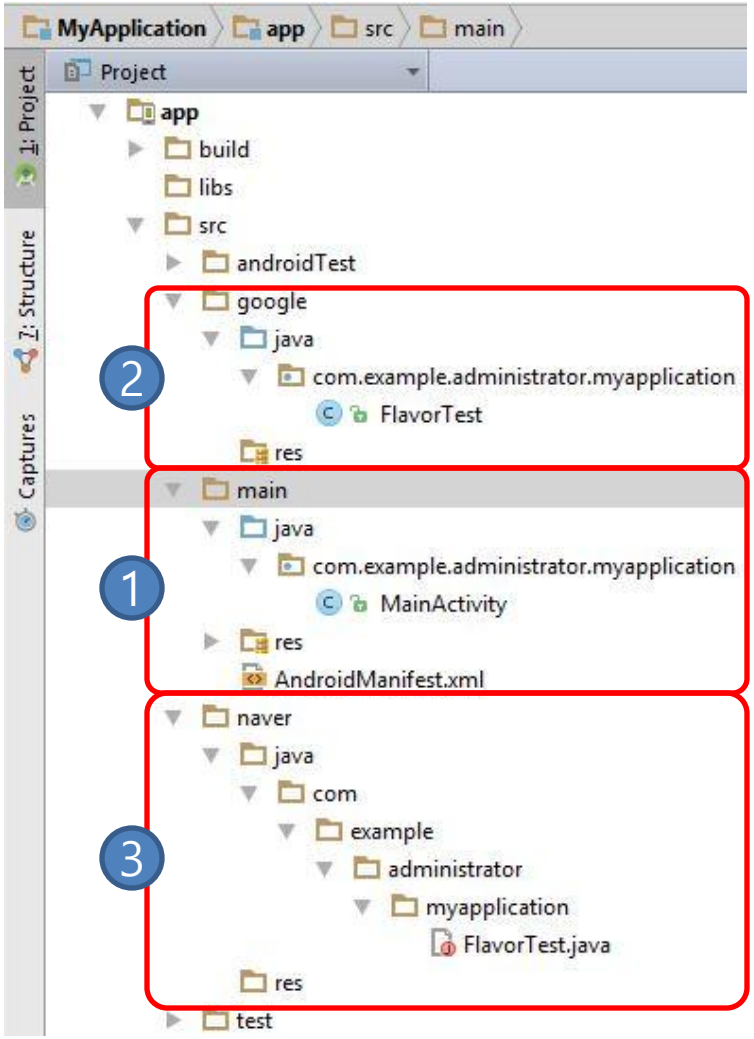

6.5. productFlavors 적용하기 1 - 설정

- productFlavors 를 통해 용도별로 배포파일을 생성해 줄 수 있다

```
android {
    defaultConfig {
        applicationId "com.example.administrator.myapplication"
    }

    productFlavors {
        google {
            versionName "1.0.1-google"
        }
        naver {
            applicationId "com.example.administrator.myapplication.naver"
            versionName "1.0.1-naver"
        }
    }
}
```

(참고 1) productFlavors 는 개별 라이브러리 모듈에는 적용할 수 없다
(참고 2) buildTypes 와 productFlavors 를 두 가지 이상 설정 시
apk 파일개수 = (buildTypes 개수 * productFlavors의 개수)

6.6. productFlavors 적용하기 2 - 폴더생성



- main 소스에서 각 flavor를 참조시 해당 flavor의 동일한 클래스명을 참조 한다. **(* 단 main에 존재하는 클래스는 개별 flavor에 만들면 안된다)**

① **main 폴더**
프로젝트 생성 시 생성되는 폴더로 모든 기본로직이 작성된다

② **google 폴더**
flavor에서 지정한 폴더
app-google-***로 생성되는 apk 파일에서만 참조된다

③ **naver 폴더**
app-naver-***로 생성되는 apk 파일에서만 참조된다

6.7. dependencies 설정

- 의존성 있는 라이브러리를 관리하기 위한 옵션

```
dependencies {
1   compile project(':facebookSDK')
2   compile fileTree(dir: 'libs', include: ['*.jar'])
3   compile 'com.android.support:support-v7:23.4.0'
4   compile files('libs/retrofit-1.8.0.jar')
}
```

① **동일 프로젝트내의 모듈 지정**
project(':프로젝트명')

② **의존성 모듈이 있는 폴더 전체 지정**
fileTree(dir:'폴더명', include:['*.파일확장자'])

③ **외부저장소에 있는 라이브러리 지정**
비단축형 -> **group** : 'com.android.support', **name** : 'support-v7' , **version** : '23.4.0'

④ **개별 파일에 대한 지정**
files('파일명')

7. ARR(라이브러리) 빌드를 위한 Gradle 설정

- 플러그인 설정을 library 로 변경한다
- **build.gradle(Module)**

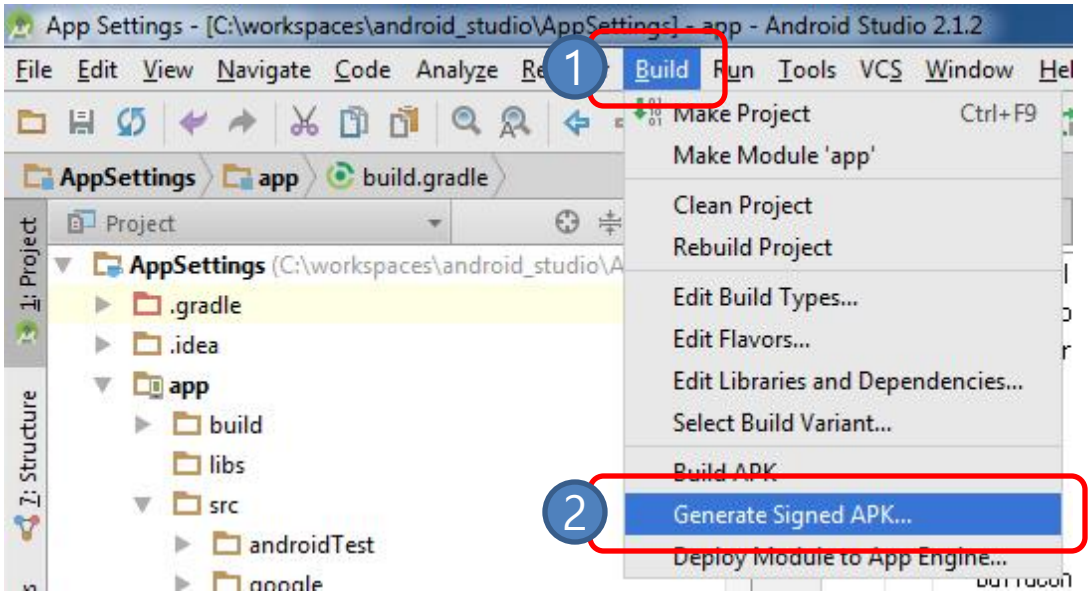
```
1 apply plugin: 'com.android.library'

android {
    compileSdkVersion 23
    ...
2   defaultConfig {
        applicationId "com.example.administrator.myapplication"
        ...
    }

}
```

① apply plugin
- com.android.library로 변경

② defaultConfig
- applicationId 삭제

8.1. keystore 생성하기 1 - 생성메뉴

- build된 파일을 마켓에 등록하기 위해서는 암호화된 key를 이용한 singing 작업이 필요한데 key store 는 이런 암호화된 키를 보관하는 저장소다

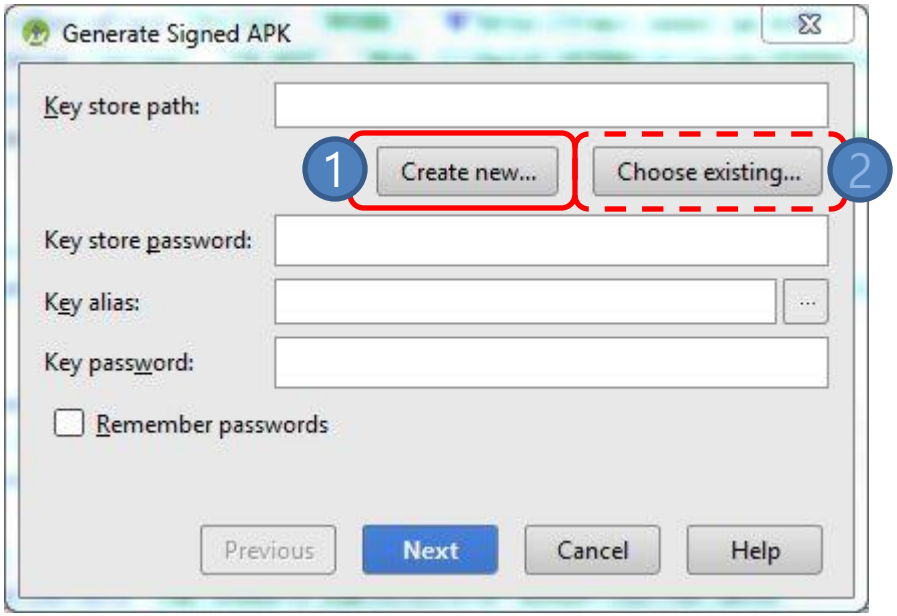


① 상단 Build 메뉴 클릭

② Generate Signed APK 클릭

8.2. keystore 생성하기 2

- Create new 버튼을 클릭해서 Key store 를 새로 만들어 준다

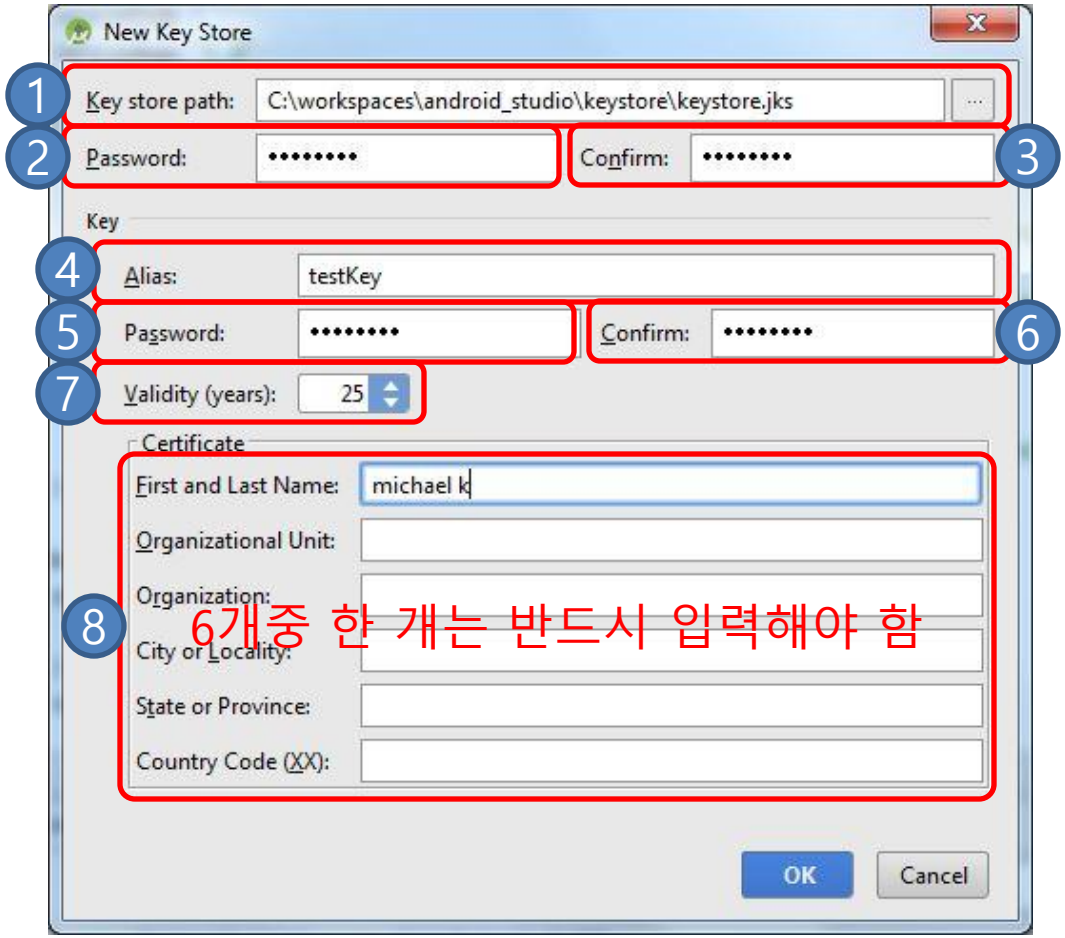


① [Create new...] 클릭

② 기존에 만들어진 키스토어가 있는 경우 [Choose existing...] 클릭

8.3. keystore 생성하기 3 - 키스토어, 키 생성

- 새로운 Key Store 와 함께 Signing에 사용할 Key를 생성해준다



Key Store

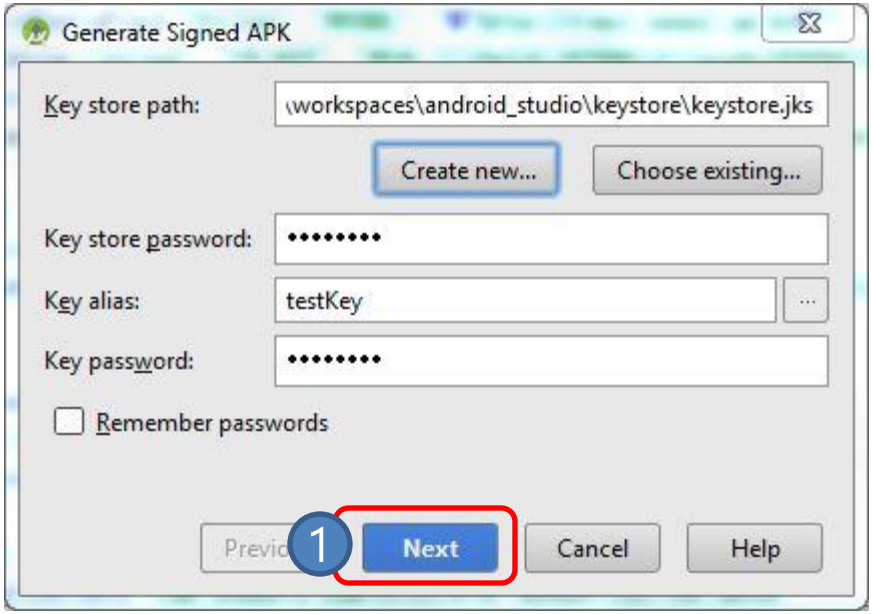
① Key store path :
키스토어 저장경로
② Password
키스토어 비밀번호
③ Confirm
키스토어 비밀번호 확인

Key

④ Alias : 암호화 Key 이름
⑤ Password : Key 비밀번호
⑥ Confirm : Key 비밀번호 확인
⑦ Validity : 사용기한
⑧ Certificate : 부가정보 중 한개는 반드시 입력해야 한다

9.1. sign 하기 - 생성완료 및 적용

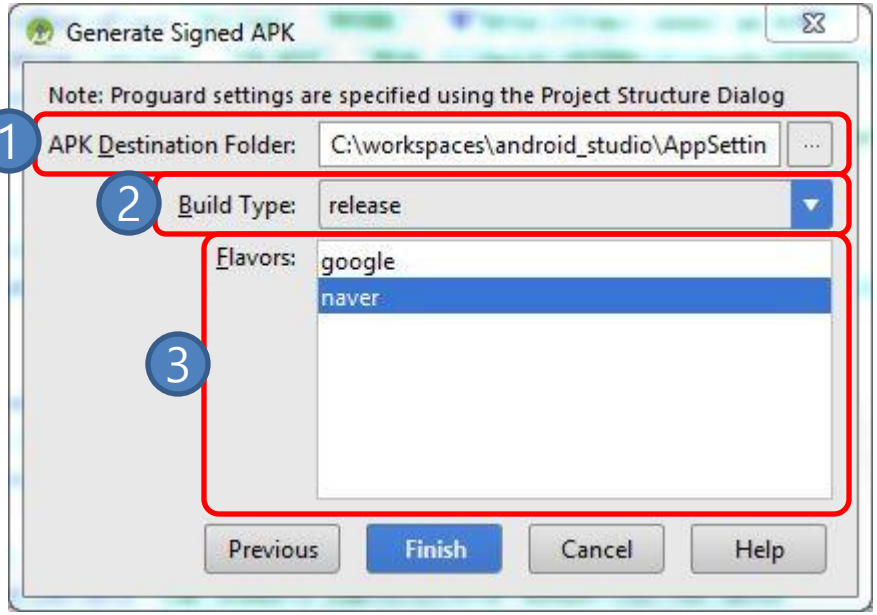
- 키 생성완료 화면 Next 버튼을 클릭한다



① [Next] 클릭

9.2. sign 하기 - signed 파일 저장 경로 설정

- 키 생성완료 화면 Next 버튼을 클릭한다



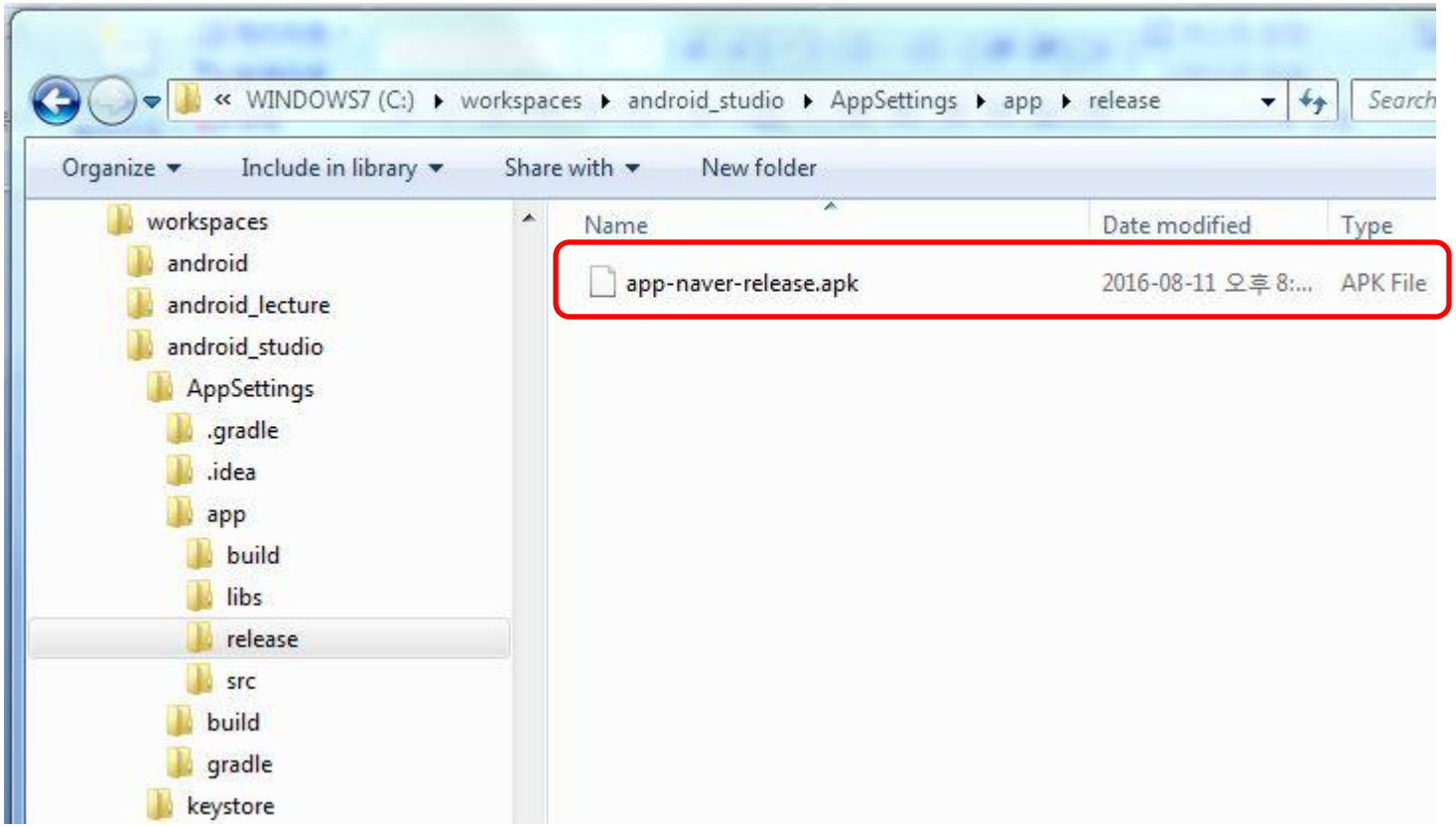
① **APK Destination Folder**
signed 된 apk 파일이 저장될 경로

② **Build Type**
signing 할 Build Type 선택

③ **Flavors**
signing 할 Flavor 선택

9.3. sign 하기 - signed apk 생성완료

- 아래와 같이 설정된 저장경로에 signed 된 apk 파일이 생성된다



10. gradle signed 설정

- signingConfigs 정보를 설정한 후 build type 중에 sign 되어야 할 Type에 signingConfig 옵션을 추가해 준다 *(* 옵션명에는 끝에 s가 없다)*

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 23
    ...
1   signingConfigs {
        release {
            storeFile file("keystore/keystore.jks")
            storePassword "12345678"
            keyAlias "testKey"
            keyPassword "12345678"
        }
    }
    buildTypes {
        release {
2           signingConfig signingConfigs.release
        }
    }
}
```

옵션명

① **signingConfigs**
- sign 정보를 설정한다

② **signingConfig**
- buildType 에 따른 sign 설정을 해준다

11. Signing Process

App signing process

1. generate a private key ***(keytool)***
2. compile to get the unsigned APK -> unaligned unsigned APK
3. Sign app in release mode using private key ***(jarsigner)*** -> unaligned signed APK
4. align the APK ***(zipalign)*** -> aligned signed APK

Why do we need the intermediate app-debug-unaligned.apk at all?

zipalign must only be performed **after** the .apk file has been signed with your priva te key. If you perform zipalign before signing, then the signing procedure will undo the alignment.

What is the advantage? zipalign?

The advantage is that aligned APKs are optimized for RAM usage so they will cons ume less RAM in the devices.
zipalign is an archive alignment tool that provides important optimization to Androi d application (.apk) files. ....The benefit is a reduction in the amount of RAM consu med when running the application.

