사용자 인터페이스 정보

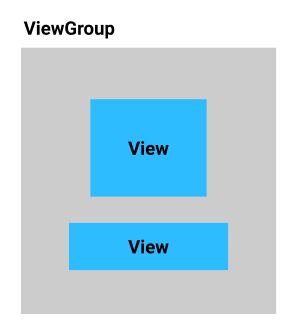
앱의 사용자 인터페이스(UI)는 화면에 표시되는 텍스트, 이미지, 버튼, 기타 여러 유형의 요소입니다. UI를 통해 앱은 사용자에게 콘텐츠를 표시하고 사용자는 앱과 상호작용합니다.

이러한 각 요소를 View라고 합니다. 앱 화면에 표시되는 대부분의 내용은 View입니다. Views는 클릭 가능한 버튼이나 수정 가능한 입력란처럼 상호작용할 수 있습니다.

이 Codelab에서는 텍스트를 표시하는 View 유형인 TextView를 사용해 봅니다.

Android 앱의 Views는 단순히 화면에 자체적으로 플로팅되지 않습니다. Views는 서로 관계가 있습니다. 예를 들어 이미지는 텍스트 옆에 있을 수 있고 버튼은 행을 형성할 수 있습니다. Views를 구성하려면 컨테이너에 배치합니다. ViewGroup은 View 객체가 있을 수 있는 컨테이너로, 내부에 있는 Views를 정렬하는 역할을 합니다. 정렬 또는 *레이아웃*은 앱이 실행되는 Android 기기 화면의 크기와 가로세로 비율에 따라 변경될 수 있고 레이아웃은 기기가 세로 모드인지 가로 모드인지에 따라 조정될 수 있습니다.

ViewGroup의 한 종류인 ConstraintLayout을 통해 내부 Views를 유연한 방식으로 정렬할 수 있습니다.



Layout Editor 정보

Views와 ViewGroups를 정렬하여 사용자 인터페이스를 만드는 것은 Android 앱을 만드는 주요 부분입니다. Android 스튜디오에서는 이 작업을 실행하는 데 도움이 되는 **Layout Editor**라는 도구를 제공합니다. **Layout Editor**를 사용하여 'Hello World!'를 'Happy Birthday!'로 변경하고 나중에 텍스트의 스타일을 지정해 봅니다.

**Layout Editor**를 열면 많은 부분으로 구성되어 있습니다. 이 Codelab에서 대부분의 사용법을 알아봅니다. 아래의 주석이 달린 스크린샷을 통해 **Layout Editor**의 창을 알아보세요. 앱을 변경하면서 각 부분에 관해 자세히 알아봅니다.

* 왼쪽 (1)은 이전에 봤던 **Project** 창입니다. 프로젝트를 구성하는 파일을 모두 나열합니다.
* 가운데에는 앱의 화면 레이아웃을 나타내는 2개의 그림 (4)와 (5)가 있습니다. 왼쪽 그림 (4)는 앱이 실행될 때 표시될 화면의 모습과 가깝습니다. 이를 **Design** 뷰라고 합니다.
* 오른쪽 그림 (5)는 **Blueprint** 뷰이고 특정 작업에 유용할 수 있습니다.
* **Palette** (2)에는 앱에 추가할 수 있는 다양한 유형의 Views 목록이 포함됩니다.
* **Component Tree** (3)은 화면 뷰의 다른 표현입니다. 화면의 모든 뷰가 나열됩니다.
* 맨 오른쪽 (6)은 **Attributes**입니다. View의 속성을 표시하고 변경할 수 있도록 합니다.

**Layout Editor**와 그 구성 방법에 관한 자세한 내용은 [개발자 참조 가이드](https://developer.android.com/studio/write/layout-editor)를 참고하세요.

주석이 달린 전체 **Layout Editor**의 스크린샷

텍스트, 스크린샷, 검은색이(가) 표시된 사진

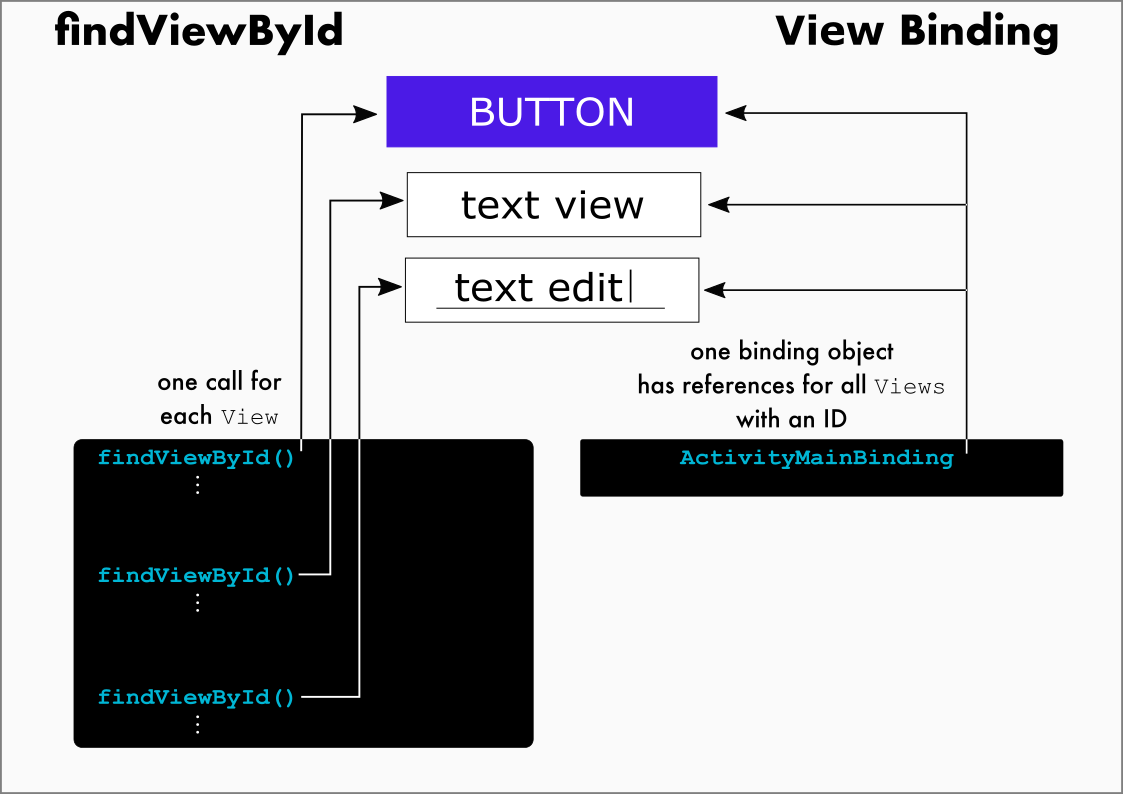
자동 생성된 설명

* **Layout Editor**를 사용하면 Android 앱용 UI를 만들 수 있습니다.
* 앱 화면에 표시되는 대부분의 내용은 View입니다.
* TextView는 앱에서 텍스트를 표시하는 UI 요소입니다.
* ConstraintLayout은 다른 UI 요소의 컨테이너입니다.
* Views는 ConstraintLayout 내에서 가로와 세로로 제한되어야 합니다.
* View를 배치하는 한 가지 방법은 여백을 사용하는 것입니다.
* 여백을 통해 View가 컨테이너의 가장자리에서 떨어진 정도를 설정할 수 있습니다.
* TextView에 글꼴, 텍스트 크기, 색상과 같은 속성을 설정할 수 있습니다.

Xml

* XML(확장성 마크업 언어)은 텍스트를 구성하는 방법이며 태그, 요소, 속성으로 구성됩니다.
* XML을 사용하여 Android 앱의 레이아웃을 정의합니다.
* EditText를 사용하여 사용자가 텍스트를 입력하거나 수정하도록 합니다.
* EditText에는 사용자에게 입력란에 예상되는 내용을 알려주는 힌트가 있을 수 있습니다.
* android:inputType 속성을 지정하여 사용자가 EditText 입력란에 입력할 수 있는 텍스트 유형을 제한합니다.
* RadioGroup으로 그룹화된 RadioButtons를 사용하여 배타적인 옵션 목록을 만듭니다.
* RadioGroup은 세로 또는 가로로 될 수 있고 개발자는 처음에 선택해야 하는 RadioButton을 지정할 수 있습니다.
* Switch를 사용하여 사용자가 두 옵션 간에 전환할 수 있도록 합니다.
* 별도의 TextView를 사용하지 않고 Switch에 라벨을 추가할 수 있습니다.
* ConstraintLayout의 각 하위 요소에는 세로 및 가로 제약 조건이 있어야 합니다.
* 'start' 및 'end' 제약 조건을 사용하여 왼쪽에서 오른쪽(LTR) 및 오른쪽에서 왼쪽(RTL) 방향 언어를 모두 처리합니다.
* 제약 조건 속성의 이름은 layout\_constraint<Source>\_to<Target>Of 형식을 따릅니다.
* View의 너비를 포함되는 ConstraintLayout의 너비와 같게 하려면 시작과 끝을 상위 요소의 시작과 끝으로 제한하고 너비를 0dp로 설정합니다.

Viewbinding



// Old way with findViewById()  
val myButton: Button = findViewById(R.id.my\_button)  
myButton.text = "A button"  
  
// Better way with view binding  
val myButton: Button = binding.myButton  
myButton.text = "A button"  
  
// Best way with view binding and no extra variable  
binding.myButton.text = "A button"

* 뷰 결합을 사용하면 앱의 UI 요소와 상호작용하는 코드를 더 쉽게 작성할 수 있습니다.
* Kotlin의 Double 데이터 유형은 십진수를 저장할 수 있습니다.
* RadioGroup의 checkRadioButtonId 속성을 사용하여 선택된 RadioButton을 확인할 수 있습니다.
* NumberFormat.getCurrencyInstance()를 사용하여 숫자를 통화 형식으로 지정하는 데 이용할 형식 지정 클래스를 가져올 수 있습니다.
* %s와 같은 문자열 매개변수를 사용하여 다른 언어로 쉽게 변환할 수 있는 동적 문자열을 만들 수 있습니다.
* 테스트가 중요합니다!
* Android 스튜디오에서 **Logcat**을 사용하여 앱 비정상 종료와 같은 문제를 해결할 수 있습니다.
* 스택 트레이스는 호출된 메서드 목록을 보여 줍니다. 이는 코드가 예외를 생성하는 경우에 유용할 수 있습니다.
* 예외는 코드가 예상하지 못한 문제를 나타냅니다.
* Null은 '값 없음'을 의미합니다.
* 일부 코드는 null 값을 처리할 수 없으므로 주의해서 사용해야 합니다.
* 추천을 확인할 수 있는 **Analyze > Inspect Code**를 사용하여 코드를 개선합니다.

## RecyclerView용 어댑터 구현하기

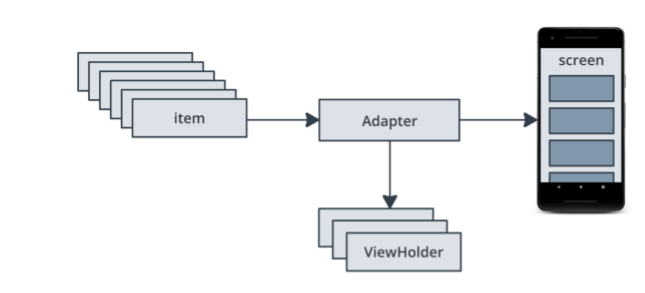
앱에는 Datasource에서 데이터를 가져와 각 Affirmation을 RecyclerView에 항목으로 표시할 수 있도록 형식을 지정하는 방법이 필요합니다.

어댑터는 데이터를 RecyclerView에서 사용할 수 있는 형식으로 조정하는 설계 패턴입니다. 이 경우에는 RecyclerView.에 표시할 수 있도록 loadAffirmations()에서 반환된 목록에서 Affirmation 인스턴스를 가져와 목록 항목 뷰로 전환하는 어댑터가 필요합니다.

앱을 실행하면 RecyclerView가 어댑터를 사용하여 화면에 데이터를 표시하는 방법을 파악합니다. RecyclerView는 목록의 첫 번째 데이터 항목을 위한 새 목록 항목 뷰를 만들도록 어댑터에 요청합니다. 뷰가 생성된 후에는 항목을 그리기 위한 데이터를 제공하도록 어댑터에 요청합니다. 이 프로세스는 RecyclerView에 화면을 채울 뷰가 더 이상 필요하지 않을 때까지 반복됩니다. 한 번에 목록 항목 뷰 세 개만 화면에 들어가는 경우 RecyclerView는 전체 목록 항목 뷰 10개가 아닌 3개만 준비하도록 어댑터에 요청합니다.

RecyclerView를 만들고 사용하는 데는 많은 부분이 관련됩니다. 이러한 부분을 분업이라고 생각할 수 있습니다. 개요를 보여주는 아래 다이어그램에서 구현하는 각 부분에 관해 자세히 알아볼 수 있습니다.

* **항목** - 표시할 목록의 단일 데이터 항목입니다. 앱의 Affirmation 객체 하나를 나타냅니다.
* **어댑터** - RecyclerView에서 표시할 수 있도록 데이터를 가져와 준비합니다.
* **ViewHolder** - 확인을 표시하기 위해 사용하거나 재사용할 RecyclerView용 뷰의 풀입니다.
* **RecyclerView** - 화면에 표시되는 뷰입니다.



[6. 요약](https://developer.android.com/codelabs/basic-android-kotlin-training-recyclerview-scrollable-list?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-recyclerview-scrollable-list#5)

* RecyclerView 위젯을 사용하여 데이터 목록을 표시할 수 있습니다.
* RecyclerView는 어댑터 패턴을 사용하여 데이터를 조정하고 표시합니다.
* ViewHolder는 RecyclerView의 뷰를 만들고 보유합니다.
* RecyclerView는 내장된 LayoutManagers와 함께 제공됩니다. RecyclerView는 항목을 배치하는 방식을 LayoutManagers에 위임합니다.

어댑터를 구현하려면 다음을 따르세요.

* 어댑터의 새 클래스(예: ItemAdapter)를 만듭니다.
* 단일 목록 항목 뷰를 나타내는 맞춤 ViewHolder 클래스를 만듭니다. RecyclerView.ViewHolder 클래스에서 확장합니다.
* ItemAdapter 클래스를 수정하여 RecyclerView.Adapter 클래스에서 확장합니다(맞춤 ViewHolder 클래스 사용).
* 어댑터 내에서 getItemsCount(), onCreateViewHolder(), onBindViewHolder() 메서드를 구현합니다.
* RecyclerView에 추가 콘텐츠를 표시하려면 기본 데이터 모델 클래스와 데이터 소스를 수정합니다. 그런 다음, 목록 항목 레이아웃 및 어댑터를 업데이트하여 데이터를 뷰에 설정합니다.
* 리소스 주석을 사용하여 올바른 유형의 리소스 ID가 클래스 생성자에 전달되도록 합니다.
* **Android 라이브러리의 머티리얼 구성요소**를 사용하여 더 간편하게 앱이 권장 머티리얼 디자인 가이드라인을 준수하도록 합니다.
* MaterialCardView를 사용하여 머티리얼 카드에 콘텐츠를 표시합니다.
* 색상 및 간격의 측면에서 앱을 시각적으로 조금만 수정하면 앱이 더욱 세련되고 일관되게 보이도록 할 수 있습니다.

Intent

*인텐트*는 실행할 작업을 나타내는 객체입니다. 인텐트는 활동을 실행하는 데 가장 많이 사용되지만 다른 용도도 있습니다. 인텐트 유형에는 두 가지가 있습니다. **암시적** 인텐트와 **명시적** 인텐트입니다. **명시적 인텐트**는 매우 구체적이며 실행할 활동을 정확하게 알 수 있고 자체 앱의 화면인 경우가 많습니다.

**암시적 인텐트**는 좀 더 추상적이며 시스템에 링크 열기나 이메일 작성, 전화 걸기와 같은 작업 유형을 알려주고 시스템은 요청 처리 방법을 파악해야 합니다. 인식하지 못한 채로 두 종류의 인텐트가 실제로 작동하는 것을 확인했을 수 있습니다. 일반적으로 자체 앱에서 활동을 표시할 때 명시적 인텐트를 사용합니다.

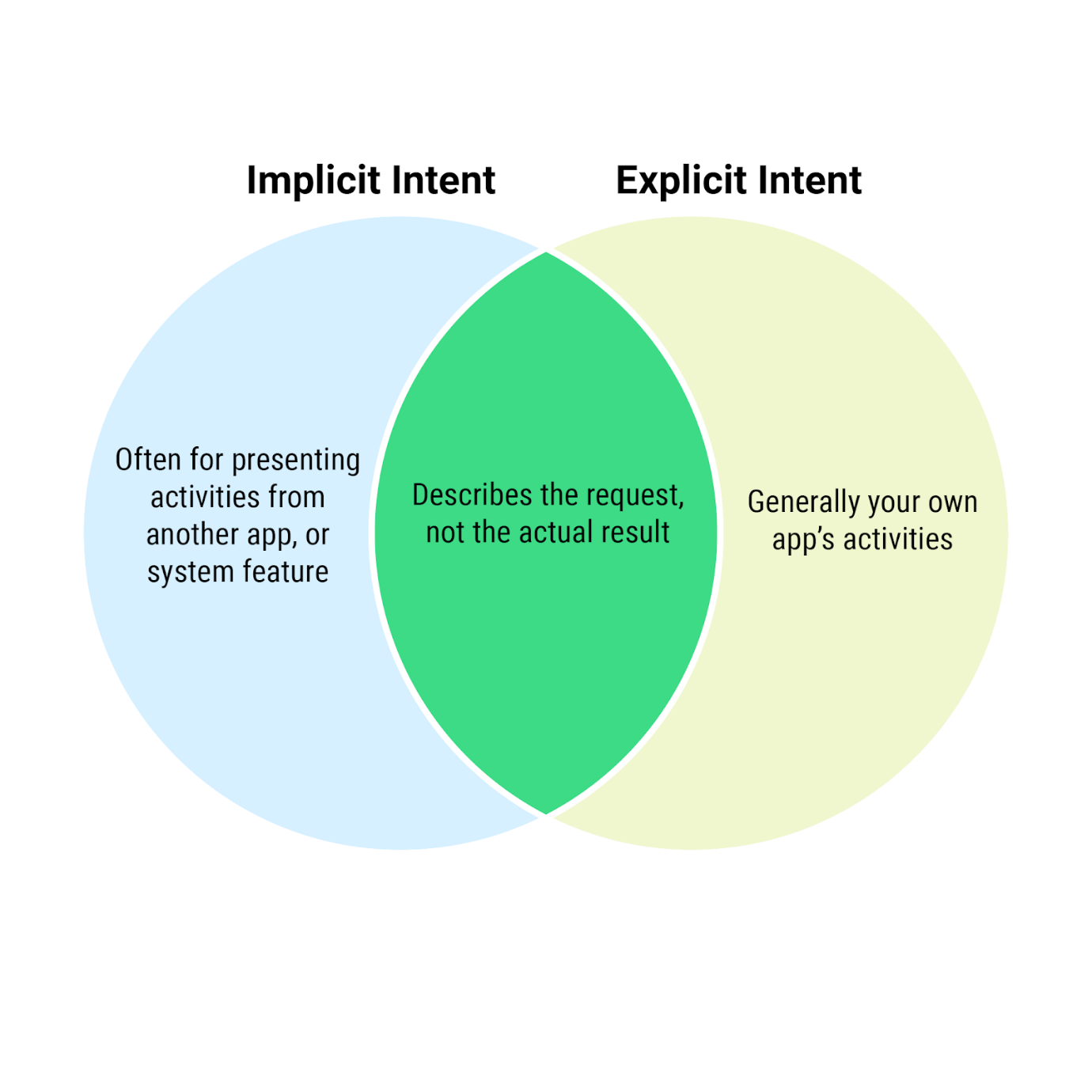
|  |  |
| --- | --- |
|  |  |

그러나 현재 앱과 관련이 없는 작업의 경우 예를 들어 흥미로운 Android 문서 페이지를 발견하여 친구와 공유하려고 할 때 **암시적 인텐트**를 사용합니다. 페이지를 공유하는 데 사용할 앱을 묻는 메뉴가 표시될 수 있습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

개발자는 자체 앱에서 작업 또는 화면 표시에 명시적 인텐트를 사용하고 전체 프로세스를 책임집니다. 일반적으로 암시적 인텐트는 다른 앱이 관련된 작업을 실행하는 데 사용하고 시스템이 최종 결과를 결정합니다. Words 앱에서 두 유형의 인텐트를 모두 사용합니다.



* ACTION\_VIEW는 URI(이 경우 웹 주소)를 사용하는 일반적인 인텐트
* CATEGORY\_APP\_MAPS - 지도 앱을 실행합니다.
* CATEGORY\_APP\_EMAIL - 이메일 앱을 실행합니다.
* CATEGORY\_APP\_GALLERY - 갤러리(사진) 앱을 실행합니다.
* ACTION\_SET\_ALARM - 백그라운드에서 알람을 설정합니다.
* ACTION\_DIAL - 전화를 겁니다.

## [11. 요약](https://developer.android.com/codelabs/basic-android-kotlin-training-activities-intents?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-3-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-activities-intents#10)

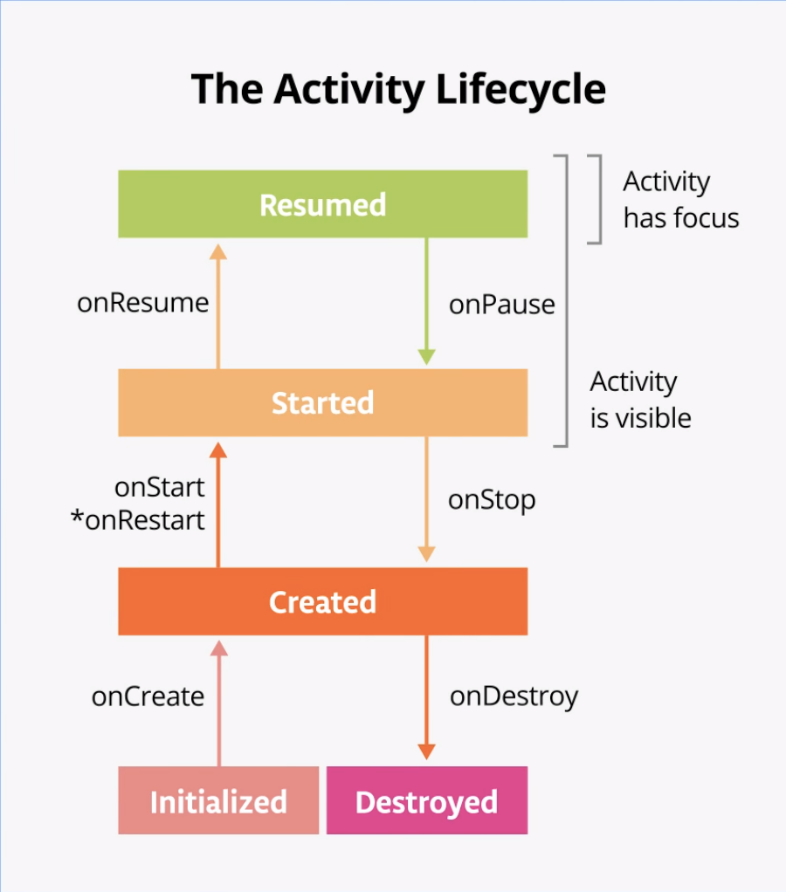
* 명시적 인텐트는 앱의 특정 활동으로 이동하는 데 사용됩니다.
* 암시적 인텐트는 특정 작업(예: 링크 열기, 이미지 공유)에 상응하며 시스템이 인텐트 처리 방법을 결정하도록 합니다.
* 메뉴 옵션을 사용하면 버튼과 메뉴를 앱 바에 추가할 수 있습니다.
* 컴패니언 객체는 재사용 가능한 상수를 유형의 인스턴스가 아닌 유형과 연결하는 방법을 제공합니다.

인텐트를 실행하려면 다음을 실행합니다.

* 컨텍스트 참조를 가져옵니다.
* 명시적인지 암시적인지에 따라 활동이나 인텐트 유형을 제공하는 Intent 객체를 만듭니다.
* putExtra()를 호출하여 필요한 데이터를 전달합니다.
* intent 객체를 전달하는 startActivity()를 호출합니다.

활동 수명 주기 단계

Activity 클래스 자체와 Activity의 모든 서브클래스(예: AppCompatActivity)는 일련의 수명 주기 콜백 메서드를 구현합니다. Android에서는 활동이 한 상태에서 다른 상태로 이동할 때 이러한 콜백을 호출하고 개발자는 이러한 메서드를 자체 활동에서 재정의하여 수명 주기 상태 변경에 응답해 작업을 실행할 수 있습니다. 다음 다이어그램은 사용할 수 있는 재정의 가능한 콜백과 함께 수명 주기 상태를 보여줍니다.



onCreate()에서 레이아웃을 확장하거나 클릭 리스너를 정의하거나 뷰 결합을 설정합니다. onCreate() 수명 주기 메서드는 활동이 초기화된 직후(새 Activity 객체가 메모리에 만들어질 때) 한 번 호출됩니다. onCreate()가 실행되면 활동이 생성됨으로 간주됩니다.

**onCreate()** 메서드를 재정의할 때 슈퍼클래스 구현을 호출하여 활동 생성을 완료해야 하므로 활동 내에서 **super.onCreate()**를 즉시 호출해야 합니다. 다른 수명 주기 콜백 메서드의 경우에도 마찬가지입니다.

onStart() 수명 주기 메서드는 onCreate() 직후에 호출됩니다. onStart()가 실행되면 활동이 화면에 표시됩니다. 활동을 초기화하는 데 한 번만 호출되는 onCreate()와 달리 onStart()는 활동의 수명 주기에서 여러 번 호출될 수 있습니다. onStart()는 상응하는 onStop() 수명 주기 메서드와 페어링됩니다. 사용자가 앱을 시작한 후 기기 홈 화면으로 돌아오면 활동이 중지되고 더 이상 화면에 표시되지 않습니다.

onResume() - 활동 포커스를 제공하고 사용자가 상호작용할 수 있도록 활동을 준비합니다.

이름에도 불구하고 onResume() 메서드는 다시 시작할 대상이 없어도 시작 시 호출됩니다.

onDestroy() 메서드의 실행은 활동이 완전히 종료되었으며 가비지 컬렉션될 수 있음을 의미합니다. [가비지 컬렉션](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science))은 더 이상 사용하지 않을 객체의 자동 정리를 나타냅니다. onDestroy()가 호출되면 시스템은 이러한 리소스가 삭제될 수 있음을 인식하고 메모리 정리를 시작합니다.

수동으로 활동의 [finish()](https://developer.android.com/reference/android/app/Activity.html#finish()) 메서드를 호출하거나 사용자가 앱을 강제 종료하는 경우에도 완전히 종료될 수 있습니다. 예를 들어 사용자는 최근 화면에서 앱을 강제 종료하거나 닫을 수 있습니다. Android 시스템은 앱이 오랫동안 화면에 표시되지 않으면 자체적으로 활동을 종료할 수도 있습니다. Android는 배터리를 보존하고 앱의 리소스를 다른 앱에서 사용할 수 있도록 이 작업을 실행합니다.

**onCreate()**와 **onDestroy()**가 단일 활동 인스턴스의 전체 기간에 한 번만 호출됩니다. **onCreate()**는 앱을 맨 처음 초기화할 때, **onDestroy()**는 앱에서 사용하는 리소스를 정리할 때 호출됩니다.

활동은 사용자가 활동에서 벗어날 때마다 완전히 닫히지 않습니다.

* 활동이 화면에 더 이상 표시되지 않으면 이는 활동이 *백그라운드*에 배치되는 것입니다. 이와 반대의 경우는 활동이 *포그라운드*에 있거나 화면에 표시되는 것입니다.
* 사용자가 앱으로 돌아오면 동일한 활동이 다시 시작되어 화면에 다시 표시됩니다. 수명 주기에서 이 부분을 앱의 *표시* 수명 주기라고 합니다.

onPause()가 호출되면 앱에 더 이상 포커스가 없습니다. onStop() 이후에는 앱이 더 이상 화면에 표시되지 않습니다. 활동이 중지되었지만 Activity 객체는 여전히 백그라운드에서 메모리에 있습니다. 활동은 소멸되지 않았습니다.  Android는 활동 리소스를 유지합니다.

활동이 포그라운드로 돌아오면 onCreate() 메서드가 다시 호출되지 않습니다. 활동 객체는 소멸되지 않았으므로 다시 만들지 않아도 됩니다. onCreate() 대신 onRestart() 메서드가 호출됩니다. 이번에는 활동이 포그라운드로 돌아올 때 **Desserts Sold** 수가 유지됩니다.

**onStart()**와 **onStop()**이 사용자가 활동에서 나가거나 활동으로 이동할 때 여러 번 호출된다는 점입니다.

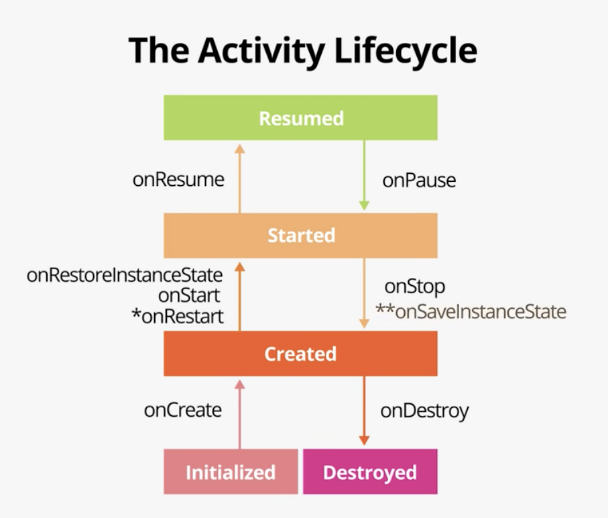
onRestart() 메서드는 onCreate()와 매우 유사합니다. onCreate()나 onRestart()는 활동이 표시되기 전에 호출됩니다. onCreate() 메서드는 처음에만 호출되고 onRestart()는 그 후에 호출됩니다. onRestart() 메서드는 활동이 처음으로 시작되지 **않은** 경우에만 호출하려는 코드를 배치하는 위치입니다.

onPause()만 사용한 중단은 보통 활동으로 돌아가거나 다른 활동 또는 앱으로 이동하기 전에 잠시 지속됩니다. 일반적으로 UI를 계속 업데이트하여 나머지 앱이 멈춘 것처럼 보이지 않도록 하는 것이 좋습니다.

onResume() 메서드는 활동에 포커스가 있을 때 호출되고 onPause()는 활동에 포커스가 없을 때 호출됩니다.

## ****onSaveInstanceState()를 사용하여 번들 데이터 저장****

onSaveInstanceState() 메서드는 Activity가 소멸되면 필요할 수 있는 데이터를 저장하는 데 사용하는 콜백입니다. 수명 주기 콜백 다이어그램에서 onSaveInstanceState()는 활동이 중지된 후 호출됩니다. 또한 앱이 백그라운드로 전환될 때마다 호출됩니다.



**onSaveInstanceState()** 재정의에는 두 가지가 있습니다. 하나는 **outState** 매개변수만 포함하고 다른 하나는 **outState** 매개변수와 **outPersistentState** 매개변수를 포함합니다.

onSaveInstanceState() 콜백은 onPause()와 onStop() 바로 다음에 발생합니다.

[Bundle](https://developer.android.com/reference/kotlin/android/os/Bundle)은 키-값 쌍 모음으로, 키가 항상 문자열입니다. Int 및 Boolean 값과 같은 간단한 데이터를 번들에 넣을 수 있습니다. 시스템이 이 번들을 메모리에 유지하므로 번들의 데이터를 작게 유지하는 것이 좋습니다. 이 번들의 크기도 제한되지만 기기마다 크기는 다릅니다. 너무 많은 데이터를 저장하면 TransactionTooLargeException 오류로 인해 앱이 비정상 종료될 수 있습니다.

putInt() 메서드와 putFloat() 및 putString()과 같은 Bundle 클래스의 유사한 메서드는 두 가지 인수를 사용합니다. 키 문자열(KEY\_REVENUE 상수)과 저장할 실제 값입니다.

## ****onCreate()를 사용하여 번들 데이터 복원****

액티비티 상태는 onCreate(Bundle)이나 [onRestoreInstanceState(Bundle)](https://developer.android.com/reference/android/app/Activity#onRestoreInstanceState(android.os.Bundle))에서 복원할 수 있습니다. onSaveInstanceState() 메서드로 채워진 Bundle은 두 수명 주기 콜백 메서드에 모두 전달됩니다.

활동이 다시 생성되는 경우 **onRestoreInstanceState()** 콜백은 번들과 함께 **onStart()** 후에 호출됩니다. 대부분의 경우 **onCreate()**에서 액티비티 상태를 복원합니다. 그러나 **onRestoreInstanceState()**는 **onStart()** 후에 호출되므로 **onCreate()**가 호출된 후 일부 상태를 복원해야 한다면 **onRestoreInstanceState()**를 사용하면 됩니다.

## [6. 요약](https://developer.android.com/codelabs/basic-android-kotlin-training-activity-lifecycle?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-3-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-activity-lifecycle#5)

## ****활동 수명 주기****

* 활동 수명 주기는 활동이 이동하는 일련의 상태입니다. 활동 수명 주기는 활동이 처음 만들어질 때 시작되고 활동이 소멸될 때 종료됩니다.
* 사용자가 활동 간에 그리고 앱 안팎으로 이동할 때 각 활동은 활동 수명 주기의 상태 간에 이동합니다.
* 활동 수명 주기의 각 상태에는 Activity 클래스에서 재정의할 수 있는 상응하는 콜백 메서드가 있습니다. 수명 주기 메서드의 핵심 집합은 다음과 같습니다. [onCreate()](https://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle))[onStart()](https://developer.android.com/reference/android/app/Activity.html#onStart())[onPause()](https://developer.android.com/reference/android/app/Activity.html#onPause())[onRestart()](https://developer.android.com/reference/android/app/Activity.html#onRestart())[onResume()](https://developer.android.com/reference/android/app/Activity.html#onResume())[onStop()](https://developer.android.com/reference/android/app/Activity.html#onStop())[onDestroy()](https://developer.android.com/reference/android/app/Activity.html#onDestroy())
* 활동이 수명 주기 상태로 전환될 때 발생하는 동작을 추가하려면 상태의 콜백 메서드를 재정의합니다.
* Android 스튜디오에서 스켈레톤 재정의 메서드를 클래스에 추가하려면 **Code > Override Methods**를 선택하거나 Control+o(Mac은 Command+o)를 누릅니다.

## ****로그로 로깅****

* Android Logging API와 특히 [Log](https://developer.android.com/reference/android/util/Log) 클래스를 사용하여 Android 스튜디오 내 Logcat에 표시되는 짧은 메시지를 작성할 수 있습니다.
* Log.d()를 사용하여 디버그 메시지를 작성합니다. 이 메서드는 두 가지 인수를 사용합니다. 로그 태그(일반적으로 클래스 이름)와 로그 메시지(짧은 문자열)입니다.
* Android 스튜디오에서 **Logcat** 창을 사용하여 작성한 메시지를 비롯한 시스템 로그를 확인합니다.

## ****액티비티 상태 유지****

* 앱이 백그라운드로 전환되면 onStop()이 호출된 직후 앱 데이터를 번들에 저장할 수 있습니다. EditText의 콘텐츠와 같은 일부 앱 데이터는 자동으로 저장됩니다.
* 번들은 키와 값의 모음인 [Bundle](https://developer.android.com/reference/kotlin/android/os/Bundle)의 인스턴스입니다. 키는 항상 문자열입니다.
* onSaveInstanceState() 콜백을 사용하여 앱이 자동으로 종료된 경우에도 유지하려는 번들에 기타 데이터를 저장합니다. 번들에 데이터를 넣으려면 put으로 시작하는 번들 메서드(예: putInt())를 사용합니다.
* onRestoreInstanceState() 메서드 또는 더 일반적인 onCreate()의 번들에서 데이터를 다시 가져올 수 있습니다. onCreate() 메서드에는 번들을 보유하는 savedInstanceState 매개변수가 있습니다.
* savedInstanceState 변수가 null이면 활동이 상태 번들 없이 시작되어 검색할 상태 데이터가 없습니다.
* 키를 사용하여 번들에서 데이터를 검색하려면 get으로 시작하는 Bundle 메서드(예: getInt())를 사용합니다.

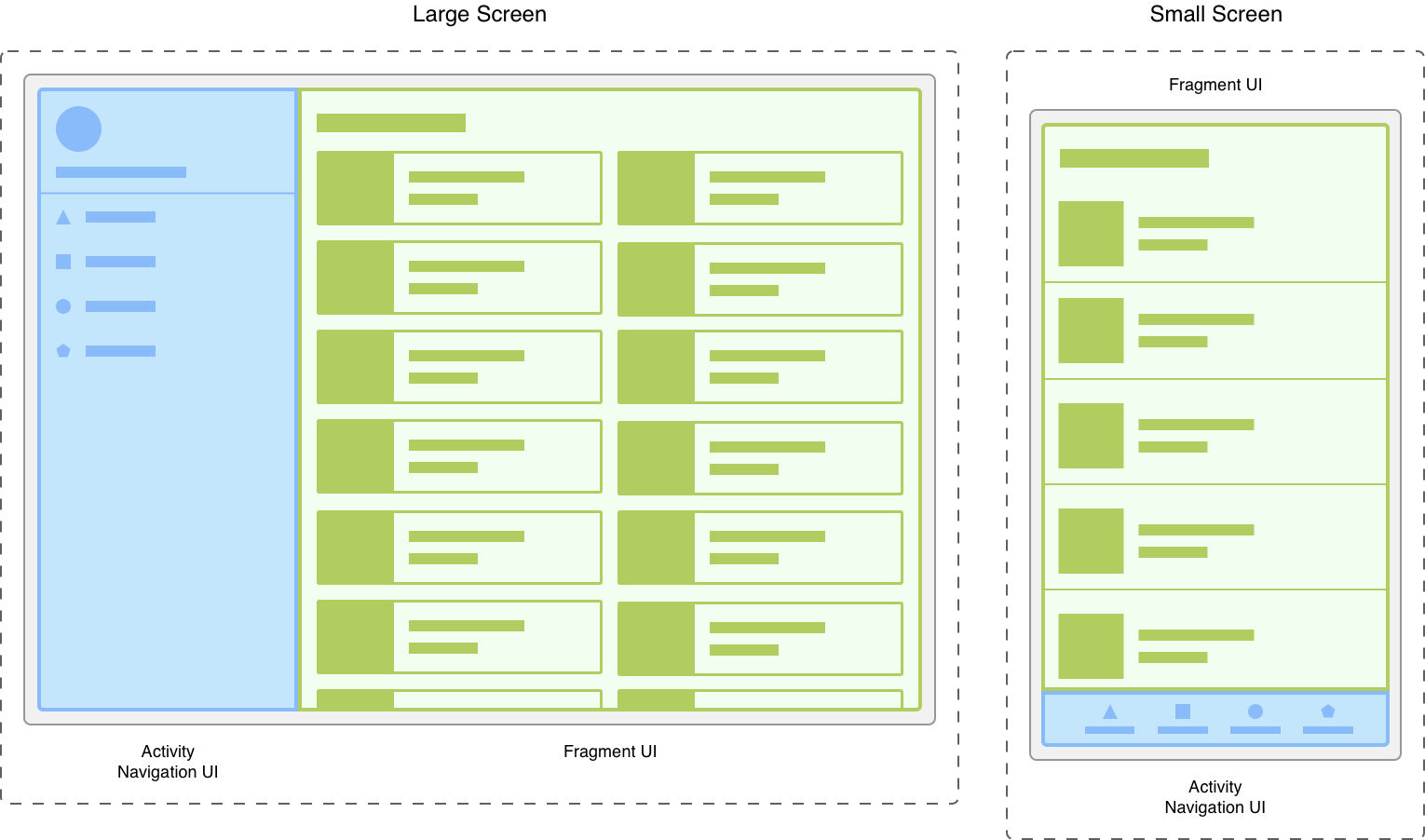
## ****구성 변경****

* 구성 변경은 기기 상태가 매우 급격하게 변경되어 시스템이 변경사항을 확인하는 가장 쉬운 방법이 활동을 소멸시키고 다시 빌드하는 것일 때 발생합니다.
* 구성 변경은 사용자가 기기를 세로 모드에서 가로 모드로 또는 가로 모드에서 세로 모드로 회전할 때 가장 흔하게 발생합니다. 기기 언어가 변경되거나 하드웨어 키보드가 연결될 때도 구성 변경이 발생할 수 있습니다.
* 구성 변경이 발생하면 Android는 모든 활동 수명 주기의 종료 콜백을 호출합니다. 그런 다음 Android는 처음부터 활동을 다시 시작하여 모든 수명 주기 시작 콜백을 실행합니다.
* Android는 구성 변경으로 인해 앱을 종료할 때 onCreate()에서 사용할 수 있는 상태 번들로 활동을 다시 시작합니다.
* 프로세스 종료와 마찬가지로 앱 상태를 onSaveInstanceState()의 번들에 저장합니다.

Fragment 및 탐색 구성 요소

[Fragment](https://developer.android.com/guide/fragments)는 재사용 가능한 UI의 부분입니다. Fragment를 하나 이상의 Activity에 재사용하고 삽입할 수 있습니다. 위 스크린샷에서 탭을 탭하면 다음 화면을 표시하기 위한 인텐트가 트리거되지 않습니다. 대신 탭을 전환하면 이전 Fragment가 다른 Fragment로 교체됩니다. 이 모든 작업은 다른 Activity을 실행하지 않고 실행됩니다.

태블릿 기기의 마스터/세부정보 레이아웃과 같이 단일 화면에 여러 프래그먼트를 한 번에 표시할 수도 있습니다. 아래 예에서 왼쪽의 탐색 UI와 오른쪽의 콘텐츠는 모두 별도의 프래그먼트에 각각 포함될 수 있습니다. 두 프래그먼트가 동일한 활동에 동시에 존재합니다.



## [3. 프래그먼트 및 프래그먼트 수명 주기](https://developer.android.com/codelabs/basic-android-kotlin-training-fragments-navigation-component?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-3-pathway-2%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-fragments-navigation-component#2)

프래그먼트는 간단하게 말해 앱의 사용자 인터페이스에서 재사용 가능한 부분입니다. 활동과 마찬가지로 프래그먼트는 수명 주기가 있고 사용자 입력에 응답할 수 있습니다. 프래그먼트는 활동이 화면에 표시될 때 활동의 뷰 계층 구조 내에 항상 포함됩니다. 재사용성과 모듈성을 강조하므로 단일 활동에서 여러 프래그먼트를 동시에 호스팅할 수도 있습니다. 각 프래그먼트는 별도의 자체 수명 주기를 관리합니다.

프래그먼트 수명 주기에는 [*Lifecycle.State*](https://developer.android.com/reference/kotlin/androidx/lifecycle/Lifecycle.State) 열거형으로 표현되는 다섯 가지 상태가 있습니다.

* INITIALIZED: 프래그먼트의 새 인스턴스가 인스턴스화되었습니다.
* CREATED: 첫 번째 프래그먼트 수명 주기 메서드가 호출됩니다. 이 상태에서 프래그먼트와 연결된 뷰도 만들어집니다.
* STARTED: 프래그먼트가 화면에 표시되지만 '포커스'가 없으므로 사용자 입력에 응답할 수 없습니다.
* RESUMED: 프래그먼트가 표시되고 포커스가 있습니다.
* DESTROYED: 프래그먼트 객체의 인스턴스화가 취소되었습니다.

또한 활동과 마찬가지로 [Fragment](https://developer.android.com/reference/android/app/Fragment) 클래스는 수명 주기 이벤트에 응답하기 위해 재정의할 수 있는 여러 메서드를 제공합니다.

* onCreate(): 프래그먼트가 인스턴스화되었고 CREATED 상태입니다. 그러나 이에 상응하는 뷰가 아직 만들어지지 않았습니다.
* onCreateView(): 이 메서드에서 레이아웃을 확장합니다. 프래그먼트가 CREATED 상태로 전환되었습니다.
* onViewCreated(): 뷰가 만들어진 후 호출됩니다. 이 메서드에서 일반적으로 findViewById()를 호출하여 특정 뷰를 속성에 바인딩합니다.
* onStart(): 프래그먼트가 STARTED 상태로 전환되었습니다.
* onResume(): 프래그먼트가 RESUMED 상태로 전환되었고 이제 포커스를 보유합니다(사용자 입력에 응답할 수 있음).
* onPause(): 프래그먼트가 STARTED 상태로 다시 전환되었습니다. UI가 사용자에게 표시됩니다.
* onStop(): 프래그먼트가 CREATED 상태로 다시 전환되었습니다. 객체가 인스턴스화되었지만 더 이상 화면에 표시되지 않습니다.
* onDestroyView(): 프래그먼트가 DESTROYED 상태로 전환되기 직전에 호출됩니다. 뷰는 메모리에서 이미 삭제되었지만 프래그먼트 객체는 여전히 있습니다.
* onDestroy(): 프래그먼트가 DESTROYED 상태로 전환됩니다.

아래 차트는 프래그먼트 수명 주기와 상태 간 전환을 간략히 보여줍니다.



프래그먼트 수명 주기에서 onCreate()는 뷰가 만들어지기 전에 호출되므로 여기서 레이아웃을 확장할 수 없습니다. 대신 onCreateView()에서 확장합니다. 그런 다음 뷰를 만든 후 onViewCreated() 메서드가 호출되고 여기서 속성을 특정 뷰에 바인딩할 수 있습니다.

프래그먼트에서는 레이아웃이 onCreateView()에서 확장됩니다. 뷰를 확장하고 \_binding 값을 설정한 다음 루트 뷰를 반환하여 onCreateView()를 구현합니다.

프래그먼트를 사용할 때 onCreateOptionsMenu() 메서드와 약간의 미묘한 차이가 있습니다. Activity 클래스에는 menuInflater라는 전역 속성이 있지만 프래그먼트에는 이 속성이 없습니다. 메뉴 인플레이터가 대신 onCreateOptionsMenu()로 전달됩니다. 또한 프래그먼트와 함께 사용되는 onCreateOptionsMenu() 메서드에는 return 문이 필요하지 않습니다.

프래그먼트는 [Context](https://developer.android.com/reference/android/content/Context)가 아님. this(프래그먼트 객체 참고)를 레이아웃 관리자의 컨텍스트로 전달할 수 없습니다. 그러나 프래그먼트는 대신 사용할 수 있는 context 속성을 제공합니다.

context

*/\*\**

*\* Return the {@link Context} this fragment is currently associated with.*

*\**

*\* @see #requireContext()*

*\*/*

@Nullable

public Context getContext() {

return mHost == null ? null : mHost.getContext();

}

requireContext

*/\*\**

*\* Return the {@link Context} this fragment is currently associated with.*

*\**

*\* @throws IllegalStateException if not currently associated with a context.*

*\* @see #getContext()*

*\*/*

@NonNull

public final Context requireContext() {

Context context = getContext();

if (context == null) {

throw new IllegalStateException("Fragment " + this + " not attached to a context.");

}

return context;

}

둘 다 return 하는 값은 동일하다.

requireContext는 현재 context 가 null 일 경우 Exception 발생. context가 null이 아닐 경우가 확실할 때 사용 혹은 사용할 때 throw 로 Exception 처리. Null이 아닌 Context를 전달해주어야 한다면 **requireContext()**를 사용해야 한다.

프래그먼트는 인텐트에 직접 액세스할 수 없으므로 activity.intent를 사용하여 참조해야 합니다.

## [7. Jetpack 탐색 구성요소](https://developer.android.com/codelabs/basic-android-kotlin-training-fragments-navigation-component?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-3-pathway-2%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-fragments-navigation-component#6)

* 탐색 그래프: 탐색 그래프는 앱에서 탐색을 시각적으로 보여주는 XML 파일입니다. 파일은 개별 활동 및 프래그먼트에 상응하는 *대상*과 한 대상에서 다른 대상으로 이동하려고 코드에서 사용할 수 있는 대상 사이의 작업으로 구성됩니다. 레이아웃 파일과 마찬가지로 Android 스튜디오는 탐색 그래프에 대상과 작업을 추가하는 시각적 편집기를 제공합니다.
* NavHost: NavHost는 활동 내에서 탐색 그래프의 대상을 표시하는 데 사용됩니다. 프래그먼트 간에 이동하면 NavHost에 표시되는 대상이 업데이트됩니다. MainActivity에서 NavHostFragment라는 기본 제공 구현을 사용합니다.
* NavController: NavController 객체를 사용하면 NavHost에 표시되는 대상 간 탐색을 제어할 수 있습니다. 인텐트를 사용할 때는 startActivity를 호출하여 새 화면으로 이동해야 했습니다. 탐색 구성요소를 사용하면 NavController의 navigate() 메서드를 호출하여 표시되는 프래그먼트를 교체할 수 있습니다. NavController를 사용하면 시스템 '위로' 버튼에 응답하여 이전에 표시된 프래그먼트로 다시 이동하는 것과 같은 일반적인 작업을 처리할 수도 있습니다.

## Safe Args 플러그인

프래그먼트 간에 데이터를 전달할 때 유형 안전성을 지원하는 Gradle 플러그인

## [8. 탐색 그래프 사용](https://developer.android.com/codelabs/basic-android-kotlin-training-fragments-navigation-component?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-3-pathway-2%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-fragments-navigation-component#7)

탐색 그래프(또는 줄여서 NavGraph)는 앱 탐색의 가상 매핑입니다. 각 화면(또는 이 경우의 프래그먼트)는 이동할 수 있는 가능한 '대상'이 됩니다. NavGraph는 각 대상이 서로 관련되는 방식을 보여주는 XML 파일로 나타낼 수 있습니다.

탐색 그래프의 대상은 FragmentContainerView로 사용자에게 표시됩니다. XML 파일을 만들고 가능한 대상을 정의하기만 하면 됩니다. 그런 다음 생성된 코드를 사용하여 프래그먼트 간에 이동할 수 있습니다.

MainActivity의 용도를 변경하여 프래그먼트의 NavHost 역할을 할 FragmentContainerView를 포함합니다. 이 시점부터 앱의 모든 탐색은 FragmentContainerView 내에서 실행됩니다.

android:name="androidx.navigation.fragment.NavHostFragment"

NavHostFragment로 설정하면 FragmentContainerView가 프래그먼트 간에 이동할 수 있습니다.

app:defaultNavHost="true"

프래그먼트 컨테이너가 탐색 계층 구조와 상호작용할 수 있습니다. 예를 들어 시스템 뒤로 버튼을 누르면 컨테이너는 새 활동이 표시될 때와 마찬가지로 이전에 표시된 프래그먼트로 다시 이동합니다.

app:navGraph="@navigation/nav\_graph"

앱의 프래그먼트가 서로 이동할 수 있는 방법을 정의하는 XML 파일을 가리킵니다.

## [12. 요약](https://developer.android.com/codelabs/basic-android-kotlin-training-fragments-navigation-component?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-3-pathway-2%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-fragments-navigation-component#11)

* 프래그먼트는 활동에 삽입할 수 있는 재사용 가능한 UI의 부분입니다.
* 프래그먼트 수명 주기는 활동 수명 주기와 다르며 뷰 설정이 onCreateView()가 아닌 onViewCreated()에서 발생합니다.
* FragmentContainerView는 프래그먼트를 다른 활동에 삽입하는 데 사용되고 프래그먼트 간 탐색을 관리할 수 있습니다.

탐색 구성요소 사용

* FragmentContainerView의 navGraph 속성을 설정하면 활동 내에서 프래그먼트 간에 이동할 수 있습니다.
* NavGraph 편집기를 사용하면 탐색 작업을 추가하고 다양한 대상 간에 인수를 지정할 수 있습니다.
* 인텐트를 사용하여 탐색하려면 extras를 전달해야 하지만 탐색 구성요소는 SafeArgs를 사용하여 탐색 작업의 클래스와 메서드를 자동 생성하므로 인수를 통해 유형 안전성을 보장합니다.

프래그먼트 사용 사례

* 탐색 구성요소를 사용하면 여러 앱이 단일 활동 내에서 전체 레이아웃을 관리할 수 있으며 모든 탐색은 프래그먼트 간에 발생합니다.
* 프래그먼트를 통해 태블릿의 마스터/세부정보 레이아웃이나 동일한 활동 내의 여러 탭과 같은 일반적인 레이아웃 패턴이 가능해집니다.

## [3. 앱 아키텍처 알아보기](https://developer.android.com/codelabs/basic-android-kotlin-training-viewmodel?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-3-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-viewmodel#2)

아키텍처는 앱에서 클래스 간의 책임을 할당하는 가이드라인을 제공합니다. 앱 아키텍처가 잘 디자인되어 있으면 향후 앱을 확장하고 더 많은 기능을 포함할 수 있습니다. 또한 팀 공동작업도 더 간편합니다.

**관심사 분리**

관심사 분리 디자인 원칙은 각각 별개의 책임이 있는 여러 클래스로 앱을 나눠야 한다는 원칙입니다.

**모델에서 UI 만들기**

또 하나의 중요한 원칙은 모델에서 UI를 만들어야 한다는 것입니다. 가급적 지속적인 모델을 권장합니다. 모델은 앱의 데이터 처리를 담당하는 구성요소로, 앱의 Views 객체 및 앱 구성요소와 독립되어 있으므로 앱의 수명 주기 및 관련 문제의 영향을 받지 않습니다.

Android 아키텍처의 기본 클래스 또는 구성요소는 UI컨트롤러(활동/프래그먼트), ViewModel, LiveData, Room

### 

UI 컨트롤러는 화면에 뷰를 그리고 사용자 이벤트나 사용자가 상호작용하는 다른 모든 UI 관련 동작을 캡처하여 UI를 제어. 앱의 데이터 또는 데이터에 관한 모든 의사 결정 로직은 포함 안 함

데이터에 관한 의사 결정 로직은 ViewModel에 추가. [ViewModel](https://developer.android.com/topic/libraries/architecture/viewmodel)은 뷰에 표시되는 앱 데이터의 모델입니다. 모델은 앱의 데이터 처리를 담당하는 구성요소로, 아키텍처 원칙에 따라 모델에서 UI가 도출. 활동이나 프래그먼트가 소멸되고 다시 생성될 때 폐기되지 않는 앱 관련 데이터를 저장. ViewModel 객체는 활동 또는 프래그먼트 인스턴스처럼 소멸되지 않음.

앱에 ViewModel을 구현하려면 아키텍처 구성요소 라이브러리에서 가져온 ViewModel 클래스를 확장하고 이 클래스 내에 앱 데이터를 저장합니다.

|  |  |
| --- | --- |
| 프래그먼트/활동(UI 컨트롤러)의 책임 | ViewModel의 책임 |
| 활동 및 프래그먼트는 뷰와 데이터를 화면에 그리고, 사용자 이벤트에 응답합니다. | ViewModel은 UI에 필요한 모든 데이터를 보유하고 처리합니다. 뷰 계층 구조(예: 뷰 결합 객체)에 액세스하거나 활동 또는 프래그먼트의 참조를 보유해서는 안 됩니다. |

ViewModel은 추상 클래스이므로 앱에 사용할 수 있도록 확장해야 합니다.

ViewModel을 UI 컨트롤러(활동/프래그먼트)에 연결하려면 UI 컨트롤러 내에 ViewModel에 관한 참조(객체)를 만듭니다.

속성 위임 접근 방식을 사용해 viewModel 객체의 책임을 viewModels라는 별도의 클래스에 위임합니다. 즉, viewModel 객체에 액세스하면 이 객체는 대리자 클래스 viewModels에 의해 내부적으로 처리됩니다. 대리자 클래스는 첫 번째 액세스 시 자동으로 viewModel 객체를 만들고 이 값을 구성 변경 중에도 유지했다가 요청이 있을 때 반환합니다.

활동과 프래그먼트는 화면에 뷰와 데이터를 그리고, ViewModel은 UI에 필요한 모든 데이터를 보유하고 처리합니다.

## 지원 속성

## ViewModel 내에서는 데이터를 수정할 수 있어야 하므로 데이터는 private 및 var이어야 합니다. ViewModel 외부에서는 데이터를 읽을 수 있지만 수정할 수는 없어야 하므로 데이터는 public 및 val로 노출되어야 합니다. 이 동작을 실현하기 위해 Kotlin에는 [지원 속성](https://kotlinlang.org/docs/reference/properties.html#backing-properties)이라는 기능이 있습니다.

지원 속성을 사용하면 정확한 객체가 아닌 getter에서 무언가를 반환할 수 있습니다.

getter 메서드와 setter 메서드 중 하나 또는 둘 모두를 재정의하여 고유한 맞춤 동작을 제공할 수 있습니다. 지원 속성을 구현하려면 읽기 전용 버전의 데이터를 반환하도록 getter 메서드를 재정의합니다.

// Declare private mutable variable that can only be modified  
// within the class it is declared.  
private var \_count = 0  
// Declare another public immutable field and override its getter method.  
// Return the private property's value in the getter method.  
// When count is accessed, the get() function is called and  
// the value of \_count is returned.  
val count: Int  
 get() = \_count

## [6. ViewModel의 수명 주기](https://developer.android.com/codelabs/basic-android-kotlin-training-viewmodel?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-3-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-viewmodel#5)

프레임워크는 활동이나 프래그먼트의 범위가 유지되는 동안 [ViewModel](https://developer.android.com/reference/androidx/lifecycle/ViewModel.html)을 유지합니다. ViewModel은 소유자가 화면 회전과 같은 구성 변경으로 인해 소멸되는 경우에도 소멸되지 않습니다. 소유자의 새 인스턴스는 다음 다이어그램과 같이 기존 ViewModel 인스턴스에 다시 연결됩니다.

## 

ViewModel은 연결된 프래그먼트가 분리되거나 활동이 완료되면 소멸됩니다. ViewModel이 소멸되기 직전에 onCleared() 콜백이 호출됩니다.

## [14. 요약](https://developer.android.com/codelabs/basic-android-kotlin-training-viewmodel?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-3-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-viewmodel#13)

* Android 앱 아키텍처 가이드라인에서는 책임이 서로 다른 클래스를 분리하고 모델에서 UI를 만들도록 권장합니다.
* UI 컨트롤러는 Activity 또는 Fragment와 같은 UI 기반 클래스입니다. UI 컨트롤러에는 UI 및 운영체제 상호작용을 처리하는 로직만 포함해야 합니다. UI에 표시할 데이터의 소스여서는 안 됩니다. UI에 표시할 데이터와 모든 관련 로직은 ViewModel에 배치합니다.
* ViewModel 클래스는 UI 관련 데이터를 저장하고 관리합니다. ViewModel 클래스를 사용하면 화면 회전과 같이 구성을 변경할 때도 데이터를 유지할 수 있습니다.
* ViewModel은 권장되는 Android 아키텍처 구성요소 중 하나입니다.

## [3. Livedata란?](https://developer.android.com/codelabs/basic-android-kotlin-training-livedata?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-3-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-livedata#2)

[LiveData](https://developer.android.com/topic/libraries/architecture/livedata)는 수명 주기를 인식하는 식별 가능한 데이터 홀더 클래스입니다.

LiveData의 특성은 다음과 같습니다.

* LiveData는 데이터를 보유합니다. LiveData는 모든 유형의 데이터에 사용할 수 있는 래퍼입니다.
* LiveData는 식별 가능합니다. 즉, LiveData 객체에서 보유한 데이터가 변경되면 관찰자에 알림이 제공됩니다.
* LiveData는 수명 주기를 인식합니다. LiveData에 관찰자를 연결하면 관찰자는 [LifecycleOwner](https://developer.android.com/topic/libraries/architecture/lifecycle#lco)(일반적으로 활동 또는 프래그먼트)와 연결됩니다. LiveData는 [STARTED](https://developer.android.com/reference/android/arch/lifecycle/Lifecycle.State.html#STARTED) 또는 [RESUMED](https://developer.android.com/reference/android/arch/lifecycle/Lifecycle.State.html#RESUMED) 같은 활성 수명 주기 상태인 관찰자만 업데이트합니다. LiveData 및 관찰에 관한 자세한 내용은 [여기](https://developer.android.com/topic/libraries/architecture/livedata.html#work_livedata)에서 알아볼 수 있습니다.
* *viewLifecycleOwner*를 첫 번째 매개변수로 observe() 메서드에 전달합니다. *viewLifecycleOwner*는 [프래그먼트의 뷰](https://developer.android.com/reference/kotlin/androidx/fragment/app/Fragment.html#getView()) 수명 주기를 나타냅니다. 이 매개변수를 사용하면 LiveData가 GameFragment 수명 주기를 인식하고 GameFragment가 활성 상태(STARTED 또는 RESUMED)일 때만 관찰자에 알릴 수 있습니다.
* 두 번째 매개변수로 람다를 추가합니다.

### **데이터 결합**

데이터 결합 라이브러리는 [Android Jetpack 라이브러리](https://developer.android.com/jetpack)의 일부입니다. 데이터 결합은 선언적 형식을 사용하여 레이아웃의 UI 구성요소를 앱의 데이터 소스에 바인딩합니다.

데이터 결합은 코드에서 데이터를 뷰 + 뷰 결합에 결합(뷰를 코드에 결합)하는 것

UI 컨트롤러에서 뷰 결합 사용의 예

binding.textViewUnscrambledWord.text = viewModel.currentScrambledWord

레이아웃 파일에서 데이터 결합 사용의 예

android:text="@{gameViewModel.currentScrambledWord}"

데이터 결합을 사용할 때 주요 이점은 활동에서 많은 UI 프레임워크 호출을 삭제할 수 있어 파일이 더욱 단순해지고 더 손쉬운 유지관리가 가능하다는 점입니다. 또한 앱 성능이 향상되며 메모리 누수 및 null 포인터 예외를 방지할 수 있습니다.

## [9. 결합 표현식 사용하기](https://developer.android.com/codelabs/basic-android-kotlin-training-livedata?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-3-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-livedata#8)

결합 표현식은 레이아웃 내에서 레이아웃 속성을 참조하는 속성(attribute properties)(예: android:text)에서 작성됩니다. 레이아웃 속성은 <variable> 태그를 통해 데이터 결합 레이아웃 파일의 상단에서 선언됩니다. 종속 변수 중 하나라도 변경되면 'DB 라이브러리'가 결합 표현식을 실행하고 이에 따라 뷰를 업데이트합니다. 이러한 변경 감지는 데이터 결합 라이브러리를 사용할 때 무료로 제공되는 훌륭한 최적화 기능입니다.

### **결합 표현식의 구문**

결합 표현식은 @ 기호로 시작하고 중괄호 {}로 래핑됩니다. 다음 예에서 TextView 텍스트는 user 변수의 firstName 속성으로 설정됩니다.

예:

<TextView android:layout\_width="wrap\_content"  
          android:layout\_height="wrap\_content"  
          android:text="@{user.firstName}" />

## 2단계: 점수 및 단어 수에 결합 표현식 추가하기

### **데이터 결합 표현식의 리소스**

데이터 결합 표현식은 다음 구문을 사용하여 앱 리소스를 참조할 수 있습니다.

android:padding="@{@dimen/largePadding}"

위의 예에서는 padding 속성에 dimen.xml 리소스 파일의 largePadding 값이 할당됩니다.

레이아웃 속성을 리소스 매개변수로 전달할 수도 있습니다.

android:text="@{@string/example\_resource(user.lastName)}"

strings.xml

<string name="example\_resource">Last Name: %s</string>

위의 예에서 example\_resource는 %s 자리표시자가 있는 문자열 리소스입니다. user.lastName을 결합 표현식의 리소스 매개변수로 전달합니다. 여기서 user는 레이아웃 변수입니다.

## [13. 요약](https://developer.android.com/codelabs/basic-android-kotlin-training-livedata?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-3-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-livedata#12)

* LiveData는 데이터를 보유합니다. LiveData는 모든 데이터에 사용할 수 있는 래퍼입니다.
* LiveData는 식별 가능합니다. 즉, LiveData 객체에서 보유한 데이터가 변경되면 관찰자에 알림이 제공됩니다.
* LiveData는 수명 주기를 인식합니다. LiveData에 관찰자를 연결하면 관찰자는 [LifecycleOwner](https://developer.android.com/topic/libraries/architecture/lifecycle#lco)(일반적으로 활동 또는 프래그먼트)와 연결됩니다. LiveData는 [STARTED](https://developer.android.com/reference/android/arch/lifecycle/Lifecycle.State.html#STARTED) 또는 [RESUMED](https://developer.android.com/reference/android/arch/lifecycle/Lifecycle.State.html#RESUMED) 같은 활성 수명 주기 상태인 관찰자만 업데이트합니다. LiveData 및 관찰에 관한 자세한 내용은 [여기](https://developer.android.com/topic/libraries/architecture/livedata.html#work_livedata)에서 알아볼 수 있습니다.
* 앱은 데이터 결합 및 결합 표현식을 사용하여 레이아웃에서 LiveData 변경사항을 수신할 수 있습니다.
* 결합 표현식은 레이아웃 내에서 레이아웃 속성을 참조하는 속성(attribute properties)(예: android:text)에서 작성됩니다.