# 15장 안드로이드 애플리케이션

이번 장에서는 안드로이드 플랫폼을 대상으로 코틀린 애플리케이션을 개발하는 방법에 대해 이야기한다. 코트린 언어와 구글 공식 지원이 제공하는 훌륭한 프로그래밍 경험으로 인해 이 분야는 모든 코틀린 생태계 중에도 가장 융성한 분야이다. 하지만 의심할 여지도 없이, 안드로이드 플랫폼 기초에 대해 자세히 다루기 위해서는 따로 책을 한권 써도 부족하다. 따라서 여기서 우리 목표는 안드로이드 세계를 친절히 소개하고 독자 여러분에게 이 분야를 더 공부하고 탐구하려는 의욕을 불러 일으키는 것이다.

이번 장을 두 부분으로 나눌 수 있다. 첫번째 부분에서는 안드로이드 스튜디오의 기본 특징을 다룬다. 새로운 프로젝트를 설정하는 방법, 그레이들을 사용해 프로젝트를 설정하고 빌드하는 방법, 안드로이드 액티비티(activity)가 무엇인지, 안드로이드 디바이스 에뮬레이터(device emulator)를 사용해 애플리케이션을 실행하는 방법에 대해 다룬다. 두번째 부분에서는 예제 계산기 애플리케이션을 개발에 초점을 맞춰서 설명하면서 액티비티의 상태를 저장하는 방법이나 코틀린 안드로이드 확장, 젯팩 콤포즈(Jetpack compose) 레이아웃 등의 고급 주제에 대해 설명한다.

[편집자님께: 안드로이드 버전은 5월 초 기준 최신에 맞췄습니다. 책과 다르더라도 제가 보내드린게 맞습니다. 상당부분 달라졌기 때문에 혹시 가능하시다면, 직접 안드로이드 스튜디오로 진행해 보시면서 검증을 해 주시면 더 고맙겠습니다.]

## 구조

* 안드로이드 개발 시작하기
* 액티비티

## 목표

안드로이드 플랫폼에서 실행되는 애플리케이션을 개발하기 위해 안드로이드 스튜디오와 코틀린을 사용하는 기초적인 방법을 이해한다.

## 안드로이드 개발 시작하기

이번 절에서는 안드로이드 스튜디오 IDE를 소개하고, 간단한 "Hello, World" 류의 애플리케이션을 통해 기본적인 프로젝트 구조에 대해 보여준다.

### 안드로이드 스튜디오 프로젝트 설정하기

안드로이드 스튜디오에서 프로젝트를 설정할 때 필요한 단계를 먼저 보여준다. 안드로이드 스튜디오는 구글이 공식적으로 제공하는 안드로이드 IDE이다. 안드로이드 스튜디오는 젯브레인즈 인텔리J 플랫폼을 기반으로 만들어졌기 때문에 지금까지 이 책에서 사용해 온 인텔리J IDEA와 상당히 유사하다. IDEA 자체와 달리 안드로이드 스튜디옹에는 안드로이드 애플리케이션 개발을 위한 구체적인 기능이 기본적으로 포함되어 있다. 인텔리J IDEA와 마찬가지로 안드로이드 스튜디오도 설치하자마자 별다른 설정 없이 코틀린 언어를 바로 지원한다.

안드로이드 스튜디오를 설치하지 않은 독자는 최신 버전을 https://developer.android.com/studio에서 다운로드하고, https://developer.android.com/studio/install.html에 있는 설치 방법을 따라 설치하라. 이번 장에서는 안드로이드 스튜디오 4.2를 사용한다.

시작하면 안드로이드 스튜디오가 초기 화면(welcome screen)을 표시한다. 거기서 "Start a new Android Studio project"를 클랙하면 프로젝트 마법사를 시작할 수 있다. 안드로이드 스튜디오에서 이미 다른 프로젝트를 연 적이 있으면 초기 화면이 표시되지 않고 마지막으로 열었던 프로젝트를 자동으로 적재한다. 이런 경우 IDE의 File | New | New Project ... 명령을 사용해 새 프로젝트를 만들 수 있다.

마법사의 첫번째 단계는 프로젝트에 사용할 첫번째 액티비티의 템플릿을 선택하는 것이다(그림 15.1). 액티비티는 기본적으로 여러분의 애플리케이션에서 사용자가 수행할 수 있는 일을 표현하는 단위다. 예를 들어 메모 편집, 현재 시간 표시, 그리고 우리가 작성할 첫번째 애플리케이션같이 간단한 환영 메시지 표시 등이 액티비티에 속한다. Empty Activity를 선택해서 안드로이드가 빈 스텁(stub) 액티비티를 생성하게 하고 Next를 클릭하자.

그림 15.1: 새 안드로이드 프로젝트에 사용할 액티비티 선택하기

프로젝트 마법사의 다음 단계는 이름, 프로젝트 클래스들이 사용할 공통 패키지, 루트 디렉터리, 디폴트 언어(그림 15.2를 보라) 등의 기본 프로젝트 정보를 선택하는 것이다. 여기서 여러분의 애플리케이션이 지원할 안드로이드 SDK 버전의 최솟값을 지정할 수도 있다. 이 버전 최솟값을 높은(최신 SDK의) 값으로 지정하면 더 강력한 API를 사용할 수 있지만 동시에 애플리케이션이 실행될 수 있는 안드로이드 디바이스(핸드폰)의 비율도 줄어든다. Help me choose 링크를 열어보면 여러 API 버전이 지원하는 내용이 어떻게 다른지 비교하는 표를 볼 수 있다. 여기서 살펴볼 간단한 예제의 경우, 디폴트로 제공하는 옵션을 그대로 남겨둔다.

그림 15.2: 기본 프로젝트 설정 선택

Finish를 클릭하면 안드로이드 스튜디오가 자동으로 필요한 프로젝트 파일을 생성해준다. 이렇게 자동생성되는 파일에는 액티비티 클래스, UI 레이아웃이나 설정 파일 등의 기본적인 애플리케이션 리소스(resource) 등이 포함된다. 그 후 안드로이드 스튜디오는 새로운 프로젝트를 설정하기 시작한다(Building Gradle project info라는 진행 막대를 보라).

그림 15.3은 IDE Project 뷰로 본 새 프로젝트의 구조를 보여준다. 이 뷰는 디폴트로 안드로이드 모드로 프로젝트를 표시해준다. 안드로이드 모드에서는 프로젝트 소스 파일을 자신이 속한 모듈과 소스 루트(일반 소스 파일, 생성된 파일, 리소스 파일 등)으로 모아서 보여준다.

그림 15.3: 예제 프로젝트 구조

안드로이드 뷰의 노드 중 하나인 Gradle Scripts에는 프로젝트 설정을 정의하는 파일들이 들어있다. 이 파일들에 대해 자세히 살펴보자.

### 그레이들 빌드 스크립트

안드로이드 스튜디오는 그레이들에 의존한다. 그레이들은 프로젝트 의존관계 관리, 컴파일, 테스트, 패키징 등의 작업을 자동화해주는 유명한 빌드 시스템이다. 프로젝트 설정은 build.gradle 파일에 들어있고, 그루비(Groovy)와 영역 특화 언어를 사용해 작성된다. 그루비는 자바와 코틀린을 연상시키는 언어다(실제로는 그루비가 코틀린 설계에 영향을 끼친 언어에 속한다). 여기서는 그루비나 그레이들에 대해 자세하 다루지는 않고, 안드로이드 애플리케이션에서 코틀린을 사용하는 것과 관련해 몇몇 중요한 내용을 강조해 알려줄 것이다. 안드로이드 스튜디오 마법사가 사용하는 프로젝트 템플릿은 자동으로 다음 파일을 만들어준다.

* 프로젝트 루트 디렉터리에 있는 프로젝트 수준의 build.gradle 파일. 이 파일에는 전체 프로젝트에 대한 공통 설정이 들어간다
* (역시 프로젝트 루트 디렉터리에 있는) 프로젝트에 어떤 모듈이 포함됐는지 지정하는 settings.gradle 파일. 필요하면 추가 설정 명령이 이 파일에 들어갈 수도 있다
* 안드로이드 SDK 디렉터리의 경로나 그레이들 프로세스를 시작할 때 JVM에 전달할 인자 등의 키-값 쌍들을 저장하는 local.properties와 gradle.properties
* app 모듈의 루트 디렉터리에 위치한, 모듈 수준의 build.gradle. 이 파일은 구체적인 모듈별 설정을 담는다

루트의 build.gradle 파일을 살펴보자.

// Top-level build file where you can add configuration options common to all sub-projects/modules.  
buildscript {  
 ext.kotlin\_version = "1.4.31"  
 repositories {  
 google()  
 mavenCentral()  
 }  
 dependencies {  
 classpath "com.android.tools.build:gradle:4.2.0"  
 classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin\_version"  
  
 // NOTE: Do not place your application dependencies here; they belong  
 // in the individual module build.gradle files  
 }  
}  
  
allprojects {  
 repositories {  
 google()  
 mavenCentral()  
 jcenter() // Warning: this repository is going to shut down soon  
 }  
}  
  
task clean(type: Delete) {  
 delete rootProject.buildDir  
}

이 스크립트는 기본적으로 다음 일을 한다.

* 코틀린 표준 라이브러리의 버전을 포함하는 kotlin\_version 프로퍼티를 설정한다. 이 프로퍼티를 다른 스크립트에서 참조할 수 있다
* 그레이들에게 프로젝트를 빌드할 때 com.android.tools.build:gradle과 org.jetbrains.kotlin:kotlin-gradle-plugin 플러그인을 사용하도록 지시한다. 첫번째 플러그인은 안드로이드 모듈을 지원하며 두번째 플러그인은 코틀린 소스코드로 된 프로젝트를 빌드할 수 있게 해준다.
* 바이너리나 소스 코드 등 의존관계 아티팩트를 다운로드할 리포지터리의 목록을 설정한다.
* 프로젝트를 다시 빌드하기 전에 이전 컴파일 결과를 지울 수 있도록 clean 태스크를 추가한다.

settings.gradle 파일은 꽤 간단하다. 디폴트로 만들어진 파일에는 프로젝트를 구성하는 모듈을 그레이들에게 알려주는 include 명령만 한줄 들어있다.

include ':app'

이제 안드로이드 :app 모듈에 있는 build.gradle 설정을 살펴보자.

plugins {  
 id 'com.android.application'  
 id 'kotlin-android'  
}  
  
android {  
 compileSdkVersion 30  
 buildToolsVersion "30.0.3"  
  
 defaultConfig {  
 applicationId "com.example.helloworld"  
 minSdkVersion 27  
 targetSdkVersion 30  
 versionCode 1  
 versionName "1.0"  
  
 testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"  
 }  
  
 buildTypes {  
 release {  
 minifyEnabled false  
 proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'  
 }  
 }  
 compileOptions {  
 sourceCompatibility JavaVersion.VERSION\_1\_8  
 targetCompatibility JavaVersion.VERSION\_1\_8  
 }  
 kotlinOptions {  
 jvmTarget = '1.8'  
 }  
}  
  
dependencies {  
 implementation "org.jetbrains.kotlin:kotlin-stdlib:$kotlin\_version"  
 implementation 'androidx.core:core-ktx:1.3.1'  
 implementation 'androidx.appcompat:appcompat:1.2.0'  
 implementation 'com.google.android.material:material:1.2.1'  
 implementation 'androidx.constraintlayout:constraintlayout:2.0.1'  
 testImplementation 'junit:junit:4.+'  
 androidTestImplementation 'androidx.test.ext:junit:1.1.2'  
 androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'  
}

이 파일에서 맨 처음 하는 일은 루트 빌드 파일(루트의 build.gradle)에 추가했던 안드로이드와 코틀린에 대한 플러그인을 활성화하는 것이다. 여기서 코틀린 플러그인인 kotlin-android는 코틀린 소스 코드를 사용하는 안드로이드 앱 개발을 지원한다. 과거 사용하던 kotlin-android-extensions은 이제 더이상 사용하지 않는다(이제 뷰 검색 및 바인딩에는 뷰 바인딩을 사용한다).

android 블럭에는 여러가지 안드로이드에 대한 설정 파라미터가 들어간다. 애플리케이션 ID, 버전 번호, 지원하는 안드로이드 SDK의 최소 버전 등이 여기 들어있다.

마지막으로 dependencies 블럭에는 모듈이 사용하는 모든 외부 의존관계가 들어간다. 각 의존관계는 의존관계가 쓰이는 설정과 의존관계 내용(보통은 androidx.core:core-ktx:1.3.1 같은 메이븐 좌표를 의존관계 내용으로 넣는다)으로 이뤄진다. 의존관계 설정은 의존관계를 어디에 언제 사용하는가를 지정한다. 예를 들어 implementation 의존관계는 컴파일시 클래스경로(classpath)와 빌드 출력에 모두 추가되지만 현재 모듈에 의존하는 다른 모듈을 컴파일할 때는 추가되지 않는다. testImplementation 의존관계는 모듈의 테스트를 컴파일하고 테스트를 실행하는 동안에는 추가되지만 실행 패키지나 프로덕션 코드에는 추가되지 않는다. 앞의 코드를 보면 알 수 있는 것처럼, 자동으로 코틀린 표준 라이브러리가 의존관계에 추가된다.

## 액티비티

이제 자바 루트 아래(**자바** 루트라는 점에 유의하라) 있는 소스 파일을 살펴보자. 이 디렉터리에 자바와 코틀린 파일이 함께 들어갈 수 있다. MainActivity.kt 파일을 찾아서 열면 에디터 창에서 해당 파일이 열린다. 다음 코드를 볼 수 있다.

package com.example.helloworld  
  
import androidx.appcompat.app.AppCompatActivity  
import android.os.Bundle  
  
class MainActivity : AppCompatActivity() {  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 setContentView(R.layout.activity\_main)  
 }  
}

이 클래스는 여려분이 프로젝트 마법사에서 선택한 템플릿을 바탕으로 안드로이드 스튜디오가 생성한 액티비티 클래스다. 모든 액티비티 클래스는 안드로이드 SDK에 속한 Activity 클래스를 상속한다. 생성된 클래스는 더 구체적인 AppCompatActivity를 상속하는데, 이 클래스는 애플리케이션 이름을 표시할 수 있는 툴바와 여러가지 대화식 UI 콤포넌트를 추가 제공한다.

안드로이드 OS는 액티비티 인스턴스를 생성한 뒤 onCreate() 메서드를 호출해준다. 따라서 보통 이 메서드 안에 초기화 관련 코드를 넣어둔다. 특히 이 메서드는 액티비티의 뷰를 설정한다.

setContentView(R.layout.activity\_main)

안드로이드 프로젝트를 컴파일하는 동안 R 클래스가 자동으로 생성된다. R 안에는 res 디렉터리에 들어있는 모든 리소스에 해당하는 식별자가 들어있다. 특히, R.layout.activity\_main은 res/layout 디렉터리 안에 있는 activity\_main.xml 파일에 해당한다. 액티비티 뷰를 구성하는 UI 콤포넌트를 지정해주는 이 파일을 레이아웃 XML 파일이라 부른다. 이 파일을 열면, 안드로이드 스튜디오는 콤포넌트를 드래그 앤 드롭하고 애트리뷰트 창의 콤포넌트 프로퍼티를 변경함으로써 UI를 편집할 수 있는 UI 디자이너 도구를 열어준다. 예를 들어 중앙의 TextView를 선택하고 텍스트 크기를 36으로 변경하면 그림 15.4 같은 결과를 볼 수 있다.

그림 15.4: UI 디자이너

디자이너 창 오른쪽 위쪽의 Code, Split, Design 탭 중에 Code를 클릭하면 에디터가 코드 뷰로 변경되고 XML 파일 내용을 보여줄 것이다. 텍스트 크기를 변경한 이후 XML 파일 내용은 다음과 같다.

<?xml version="1.0" encoding="utf-8"?>  
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:app="http://schemas.android.com/apk/res-auto"  
 xmlns:tools="http://schemas.android.com/tools"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 tools:context=".MainActivity">  
  
 <TextView  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="Hello World!"  
 android:textSize="36sp"  
 app:layout\_constraintBottom\_toBottomOf="parent"  
 app:layout\_constraintLeft\_toLeftOf="parent"  
 app:layout\_constraintRight\_toRightOf="parent"  
 app:layout\_constraintTop\_toTopOf="parent" />  
  
</androidx.constraintlayout.widget.ConstraintLayout>

앞으로 계산기 애플리케이션의 더 복잡한 UI 레이아웃 예제를 살펴볼 것이다. 당장은 이 "Hello, World" 샘플이 안드로이드 디바이스에서 어떤 모습으로 보일지 살펴보자.

### 에뮬레이터 사용하기

이제 안드로이드 디바이스 에뮬레이터를 사용해 예제 애플리케이션을 실행해볼 수 있다. 에뮬레이터를 사용해 본 적이 없다면 맨 처음 해야 할 일은 가상 디바이스(virtual device)를 만드는 것이다. 안드로이드 스튜디오 메뉴에서 Tools | AVD Manager 명령을 선택하고 Android Virtual Device Manager 대화창 안에 있는 Create Virtual Device... 버튼을 클릭하라.

그 이후 표시되는 대화창 안에서(그림 15.5) 핸드폰 모델을 선택할 수 있다. 우리 예제의 경우 디폴트 장치로도 충분하다. 따라서 Next를 클릭하라.

그림 15.5: 가상 디바이스 설정 선택

System Image 대화창에서 에뮬레이터에서 사용할 안드로이드 OS 이미지를 선택할 수 있다. 맨 처음 어떤 이미지를 사용하려면 Download 링크(그림 15.6)를 통해 해당 이미지를 다운로드 해야만 한다. 다운로드가 끝나면 Next를 클릭하라.

그림 15.6: 가상 디바이스에 사용할 안드로이드 OS 이미지 선택

가상 디바이스 설정 마법사의 마지막 화면인 Verify Configuration 대화창에서 새 디바이스의 이름과 디바이스의 디폴트 방향(orientation)을 선택할 수 있다. Finish를 클릭하면 가상 디바이스 관리자 화면으로 돌아간다. 새로 추가한 디바이스가 목록에 있는지 확인한 후, AVD 창을 닫자.

이제 새 에뮬레이터가 메인 액티비티를 실행하는 모습을 살펴보자. Run | Run 'app' 명령을 실행하거나 Run 툴바 버튼(그림 15.7을 보라)을 클릭하라.

그림 15.7: Run 명령어 사용하기

안드로이드 스튜디오는 가상 디바이스를 선택하는 대화창을 표시한다. 조금 전에 생성한 가상 디바이스를 선택하고 OK를 클릭하라. IDE가 에뮬레이터를 실행하고 운영체제를 부팅한 다음, 실행할 애플리케이션의 메인 액티비티를 시작한다. 에뮬레이터의 모양은 약간 다를 수 있지만 그림 15.8과 비슷한 모양을 볼 수 있어야 한다.

그림 15.8: 안드로이드 에뮬레이터에서 실행중인 애플리케이션

물리적인 스마트폰이나 태블릿을 사용하는 것처럼 안드로이드 에뮬레이터와 상호작용할 수 있다. 폰 모양 화면 옆에 있는 패널은 장치 회전, 볼륨 조절, 스크린샷(screenshot) 등의 기능을 제공한다. 애플리케이션을 에뮬레이터에 배포해 테스트해볼 수도 있다. 이를 위해서는 Run 대신 Debug 명령을 사용해 디버깅 모드에서 애플리케이션을 시작하라.

이제 기본적인 프로젝트 구조를 이해했으므로, 애플리케이션이 조금 더 사용자와 상호작용을 할 수 있게 만들어보자.

## 액티비티

이번 장의 나머지 부분은 계산기 애플리케이션 예제 위주로 되어있다. 이번 절을 진행하면서 레이아웃 XML과 젯팩 콤포즈 DSL을 사용해 액티비티 UI를 정의하는 방법을 살펴보고, 합성 프로퍼티를 통해 UI관련 코드를 쉽게 작성하는 방법과 액티비티 생명주기에 대해 살펴본다. 그리고 액티비티의 상태를 언제 어떻게 저장해야 하는지 살펴본다.

### 애플리케이션 UI 설계하기

앞에서 본 Hello, World 예제를 바탕으로 계산기 애플리케이션을 만든다. 먼저 strings.xml에 리소스 파일에 있는 애플리케이션 제목을 바꾸자.

<resources>  
 <string name="app\_name">Calculator<</string>  
</resources>

이제 메인 액티비티의 UI 정의가 들어있는 activity\_main.xml을 열자. 오른쪽 위의 Code 버튼을 눌러서 XML 파일을 텍스트 형태로 표시한 후, 다음과 같이 내용을 바꾸자. (참고로 지면 관계상 xmlns:a="http://schemas.android.com/apk/res/android"라고 XML 네임스페이스 이름을 a로 줬는데, 젯팩 등의 관련 도구에서 제대로 XML을 처리하지 **못하는 경우가 생기므로** 이 예제를 시험해볼 때는 **반드시** 이름을 xmlns:android="http://schemas.android.com/apk/res/android"로 바꾸고, a:layout\_width등도 android:layout\_width등으로 android: 접두사를 사용해야 한다)

<?xml version="1.0" encoding="utf-8"?>  
<RelativeLayout xmlns:a="http://schemas.android.com/apk/res/android"  
 xmlns:tools="http://schemas.android.com/tools"  
 a:id="@+id/relative1" a:layout\_width="match\_parent" a:layout\_height="match\_parent"  
 tools:context=".MainActivity">  
  
 <TableLayout  
 a:layout\_width="match\_parent" a:layout\_height="match\_parent" a:stretchColumns="3">  
  
 <TextView  
 a:id="@+id/txtResult" a:layout\_width="match\_parent"  
 a:layout\_height="wrap\_content" a:textSize="40sp" />  
  
 <TableRow a:layout\_width="match\_parent" a:layout\_height="match\_parent">  
  
 <Button a:id="@+id/btn7" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:text="7" />  
  
 <Button a:id="@+id/btn8" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:text="8" />  
  
 <Button a:id="@+id/btn9" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:text="9" />  
  
 <Button a:id="@+id/btnPlus" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:layout\_gravity="end|center\_vertical"  
 a:text="+" />  
 </TableRow>  
  
 <TableRow a:layout\_width="match\_parent" a:layout\_height="match\_parent">  
  
 <Button a:id="@+id/btn4" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:text="4" />  
  
 <Button a:id="@+id/btn5" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:text="5" />  
  
 <Button a:id="@+id/btn6" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:text="6" />  
  
 <Button  
 a:id="@+id/btnMinus" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:layout\_gravity="end|center\_vertical"  
 a:text="-" />  
 </TableRow>  
  
 <TableRow a:layout\_width="match\_parent" a:layout\_height="match\_parent">  
  
 <Button a:id="@+id/btn1" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:text="1" />  
  
 <Button a:id="@+id/btn2" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:text="2" />  
  
 <Button a:id="@+id/btn3" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:text="3" />  
  
 <Button a:id="@+id/btnTimes" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:layout\_gravity="end|center\_vertical"  
 a:text="\*" />  
 </TableRow>  
  
 <TableRow a:layout\_width="match\_parent" a:layout\_height="match\_parent">  
  
 <Button a:id="@+id/btn0" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:text="0" />  
  
 <Button a:id="@+id/btnPoint" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:text="." />  
  
 <Button a:id="@+id/btnSign" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:text="+/-" />  
  
 <Button a:id="@+id/btnDivide" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:layout\_gravity="end|center\_vertical"  
 a:text="/" />  
 </TableRow>  
  
 <TableRow a:layout\_width="match\_parent" a:layout\_height="match\_parent">  
  
 <Button a:id="@+id/btnBackspace" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:text="&lt;-" />  
  
 <Button a:id="@+id/btnClear" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:text="C" />  
  
 <Space a:layout\_width="wrap\_content" a:layout\_height="wrap\_content" />  
  
 <Button a:id="@+id/btnCalc" a:layout\_width="wrap\_content"  
 a:layout\_height="wrap\_content" a:layout\_gravity="end|center\_vertical"  
 a:text="=" />  
 </TableRow>  
 </TableLayout>  
</RelativeLayout>

그림 15.9는 이 계산기 UI의 모습을 미리보기한 것이다.

그림 15.9: 계산기 UI 미리보기

레이아웃 XML의 형식에 대해 설명하는 것은 이 책의 범위를 벗어난다. 따라서 여기서 세부 사항을 자세히 다루지는 않고, 몇가지 기본적인 내용만 설명한다. 레이아웃은 내포된 뷰들을 정해진 방식으로 배열하는 컨테이너라 할 수 있다(안드로이드 레이아웃을 어떤 LayoutManager가 정해진 스윙 컨테이너라고 생각할 수 있다). 계산기 UI는 대부분 정해진 위치에 있는 요소들로 이뤄지므로, 테이블 레이아웃을 사용해 내포된 콤포넌트를 열과 행에 배치했다. 계산기의 기본 콤포넌트 자체는 Button(텍스트를 표시한 간단한 버튼)이나 TextView(스윙 JLabel처럼 읽기 전용 텍스트를 표시하는 콤포넌트)으로 이뤄진다.

a:id 애트리뷰트에 유의하라. 데이터를 사용자에게 제공하거나 사용자로부터 데이터를 얻는 콤포넌트에 대해 이 애트리뷰트를 사용했다. 이 애트리뷰트는 해당 엘리먼트를 자바나 코틀린 코드에서 참조할 수 있는 수단을 제공한다. 다음 절에서 액티비티 클래스에 대해 설명하면서 어떻게 이 애트리뷰트를 통해 콤포넌트를 참조할 수 있는지 설명한다.

### 액티비티 클래스 정의하기

UI에 몇가지 행동을 부여하는 액티비티 클래스를 구현하자. 계산기의 비즈니스 로직에 대해서는 자세히 다루지 않고 독자들에게 연습문제로 남겨둔다. 대신, 안드로이드와 관련있는 세부 사항에 대해 계속 다룰 것이다.

무엇보다 사용자가 지금까지 입력한 내용을 계산기의 디스플레이에 해당하는 txtResult 콤포넌트에 표시하고 싶다. 이를 위해서는 findViewById() 함수에 텍스트 뷰의 식별자를 넘겨야 한다. Hello, World 예제에서 설명한 R 클래스를 통해 식별자를 참조할 수 있기 때문에, 레이아웃 XML에 있는 실제 문자열을 넣을 필요는 없다.

private val txtResult by lazy { findViewById<TextView>(R.id.txtResult) }

이 뷰 참조를 사용해 뷰 내부 멤버에 접근할 수 있다. 예를 들어 setOnClickListener()를 호출하면 버튼을 클릭할 때 원하는 액션이 일어나게 할 수 있다.

findViewById<Button>(R.id.btn0).setOnClickListener { appendText("0") }

데이터를 읽거나 쓸 때는 TextView의 text 프로퍼티를 사용해 콤포넌트의 텍스트 컨텐트에 접근할 수 있다.

private fun clearText() {  
 txtResult.text = "0"  
}

다음은 계산기 MainActivity 클래스의 전체 소스 코드다.

package com.example.helloworld  
  
import androidx.appcompat.app.AppCompatActivity  
import android.os.Bundle  
import android.widget.Button  
import android.widget.TextView  
import android.widget.Toast  
import java.math.BigDecimal  
import java.math.RoundingMode  
  
class MainActivity : AppCompatActivity() {  
 enum class OpKind {  
 ADD, SUBTRACT, MULTIPLY, DIVIDE  
 }  
  
 companion object {  
 fun OpKind.compute(a: BigDecimal, b: BigDecimal) = when (this) {  
 OpKind.ADD -> a + b  
 OpKind.SUBTRACT -> a - b  
 OpKind.MULTIPLY -> a \* b  
 OpKind.DIVIDE ->a.divide(b, 10, RoundingMode.HALF\_EVEN)  
 }  
 }  
  
 private val txtResult by lazy<TextView> { findViewById(R.id.txtResult) }  
 private var lastResult: BigDecimal = BigDecimal.ZERO;  
 private var lastOp: OpKind? = null  
 private var waitingNextOperand: Boolean = false  
  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 setContentView(R.layout.activity\_main)  
  
 findViewById<Button>(R.id.btn0).setOnClickListener{ appendText("0") }  
 findViewById<Button>(R.id.btn1).setOnClickListener{ appendText("1") }  
 findViewById<Button>(R.id.btn2).setOnClickListener{ appendText("2") }  
 findViewById<Button>(R.id.btn3).setOnClickListener{ appendText("3") }  
 findViewById<Button>(R.id.btn4).setOnClickListener{ appendText("4") }  
 findViewById<Button>(R.id.btn5).setOnClickListener{ appendText("5") }  
 findViewById<Button>(R.id.btn6).setOnClickListener{ appendText("6") }  
 findViewById<Button>(R.id.btn7).setOnClickListener{ appendText("7") }  
 findViewById<Button>(R.id.btn8).setOnClickListener{ appendText("8") }  
 findViewById<Button>(R.id.btn9).setOnClickListener{ appendText("9") }  
 findViewById<Button>(R.id.btnPoint).setOnClickListener{ appendText(".") }  
 findViewById<Button>(R.id.btnSign).setOnClickListener{  
 val currentText = txtResult.text.toString()  
 txtResult.text = when {  
 currentText.startsWith("-") ->  
 currentText.substring(1, currentText.length)  
 currentText != "0" -> "-$currentText"  
 else ->return@setOnClickListener  
 }  
 }  
 findViewById<Button>(R.id.btnBackspace).setOnClickListener{  
 val currentText = txtResult.text.toString()  
 val newText = currentText.substring(0, currentText.length - 1)  
 txtResult.text =  
 if (newText.isEmpty() || newText == "-") "0" else newText  
 }  
 findViewById<Button>(R.id.btnClear).setOnClickListener{ clearText() }  
 findViewById<Button>(R.id.btnPlus).setOnClickListener{ calc(OpKind.ADD) }  
 findViewById<Button>(R.id.btnMinus).setOnClickListener{ calc(OpKind.SUBTRACT) }  
 findViewById<Button>(R.id.btnTimes).setOnClickListener{ calc(OpKind.MULTIPLY) }  
 findViewById<Button>(R.id.btnDivide).setOnClickListener{ calc(OpKind.DIVIDE) }  
 findViewById<Button>(R.id.btnCalc).setOnClickListener{ calc(null) }  
  
 clearText()  
 }  
  
 private fun clearText() {  
 txtResult.text = "0"  
 }  
 private fun appendText(text: String) {  
 if (waitingNextOperand) {  
 clearText()  
 waitingNextOperand = false  
 }  
 val currentText = txtResult.text.toString()  
 txtResult.text =  
 if (currentText == "0") text else currentText + text  
 }  
 private fun calc(nextOp: OpKind?) {  
 if (waitingNextOperand) {  
 lastOp = nextOp  
 return  
 }  
 val currentValue = BigDecimal(txtResult.text.toString())  
 val newValue = try {  
 lastOp?.compute(lastResult, currentValue) ?: currentValue  
 } catch (e: ArithmeticException) {  
 lastOp = null  
 waitingNextOperand = true  
 Toast.makeText(  
 applicationContext,  
 "Invalid operation!",  
 Toast.LENGTH\_SHORT  
 ).show()  
 return  
 }  
  
 if (nextOp != null) lastResult = newValue  
 if (lastOp != null) txtResult.text = newValue.toPlainString()  
  
 lastOp = nextOp  
 waitingNextOperand = nextOp != null  
 }  
}

이제 계산기 앱을 실행해 실제 어떻게 작동하는지 살펴볼 수 있다. 그림 15.10은 안드로이드 에뮬레이터에서 이 앱을 실행한 경우를 보여준다.

그림 15.10: 작동중인 계산기 애플리케이션

## 젯팩 뷰 바인딩

findViewById()를 자주 사용하면 코드가 지저분해 보인다. 특히 txtResult를 저장했던 것처럼 뷰 콤포넌트를 클래스 프로퍼티로 저장하는 경우 상당히 코드가 길어지고 작성하기 번거롭다. 자바에서는 버터나이프(Butterknife)나 안드로이드 데이터 바인딩(Android Data Binding)을 사용해 자동으로 뷰 참조를 클래스 필드에 넣을 수 있지만 매번 대상 필드에 애너테이션을 붙여야 했고, 코틀린에서는 코틀린 안드로이드 확장(Kotlin Android Extension)이 자동으로 생성해주는 필드(synthetics)를 통해 뷰를 쉽게 참조할 수 있었다. 하지만 앞으로는 코틀린 안드로이드 확장은 더 이상 쓰이지 않고, https://developer.android.com/jetpack 에서 볼 수 있는 안드로이드 젯팩(Android Jetpack)이 제공하는 뷰 바인딩(view binding)을 통해 콤포넌트에 접근해야 한다.

젯팩 뷰 바인딩을 쓰려면 안드로이드 스튜디오 3.6 이상을 사용해야 하며, :app 모듈의 build.gradle 내부에 있는 android 블럭에 다음을 추가해야 한다.

android {   
 ... (나머지 생략)  
 buildFeatures {  
 viewBinding true  
 }  
}

뷰 바인딩을 활성화하면 컴파일이 이뤄지는 동안 각 XML 레이아웃 파일마다 바인딩 클래스가 생성된다. 바인딩 클래스의 이름은 XML 파일 이름을 낙타등 표기법(camelcase, 밑줄은 없애고, 각 단어의 첫 글자를 대문자로 표시)으로 바꾸고, 맨 끝에 Binding을 추가한 이름으로 정해진다. 따라서 계산기 예제의 경우 클래스 이름은 ActivityMainBinding이 된다.

이 바인딩 클래스를 사용하려면 액티비티 클래스에서 다음과 같이 바인딩 클래스에 대한 참조를 선언한다.

private lateinit var binding: ResultProfileBinding

그리고 onCreate에서 레이아웃을 inflate로 가져와서 setContentView로 설정하자.

override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 binding = ActivityMainBinding.inflate(layoutInflater)  
 setContentView(binding.root)  
   
 ...  
 }

이제 위 코드의 ... 부분에 각 콤포넌에 대한 리스너를 다음과 같이 설정할 수 있다.

binding.run{  
 btn0.setOnClickListener{ appendText("0") }  
 btn1.setOnClickListener{ appendText("1") }  
 btn2.setOnClickListener{ appendText("2") }  
 btn3.setOnClickListener{ appendText("3") }  
 btn4.setOnClickListener{ appendText("4") }  
 btn5.setOnClickListener{ appendText("5") }  
 btn6.setOnClickListener{ appendText("6") }  
 btn7.setOnClickListener{ appendText("7") }  
 btn8.setOnClickListener{ appendText("8") }  
 btn9.setOnClickListener{ appendText("9") }  
 btnPoint.setOnClickListener{ appendText(".") }  
 btnSign.setOnClickListener{  
 val currentText = txtResult.text.toString()  
 txtResult.text = when {  
 currentText.startsWith("-") ->  
 currentText.substring(1, currentText.length)  
 currentText != "0" -> "-$currentText"  
 else ->return@setOnClickListener  
 }  
 }  
 btnBackspace.setOnClickListener{  
 val currentText = txtResult.text.toString()  
 val newText = currentText.substring(0, currentText.length - 1)  
 txtResult.text =  
 if (newText.isEmpty() || newText == "-") "0" else newText  
 }  
 btnClear.setOnClickListener{ clearText() }  
 btnPlus.setOnClickListener{ calc(OpKind.ADD) }  
 btnMinus.setOnClickListener{ calc(OpKind.SUBTRACT) }  
 btnTimes.setOnClickListener{ calc(OpKind.MULTIPLY) }  
 btnDivide.setOnClickListener{ calc(OpKind.DIVIDE) }  
 btnCalc.setOnClickListener{ calc(null) }  
 }  
  
 clearText()

한편, txtResult를 저장할 필요가 없이, binding에서 얻어서 사용하면 된다. 예를 들어 clearText()는 다음과 같이 바뀐다.

private fun clearText() {  
 binding.txtResult.text = "0"  
 }

뷰를 편하게 사용하는 방법을 살펴봤다. 다음으로는 액티비티의 상태를 유지하는 방법을 살펴보자.

### 액티비티 상태 유지하기

계산기 애플리케이션을 시험하다 보면 눈에 띄는 문제가 있음을 깨달을 수 있다. 숫자를 몇자리 입력한 후, 디바이스를 회전시켜보자. 이를 위해서는 에뮬레이터의 옆에 있는 패널의 Rotate Left/Rotate Right 버튼을 클릭하면 된다. 결과는 그림 15.11과 같다.

그림 15.11: 디바이스 회전시 액티비티가 재설정됨

원래 화면에 보이던 숫자가 0으로 변경됐음을 알 수 있다. 실제로 디바이스가 회전하면 액티비티의 모든 상태가 초기화 코드에 제공한 값으로 재설정된다. 이유는 디바이스 설정(예: 화면 회전)이 변경될 때마다 시스템이 액티비티를 없애고 처음부터 다시 만들기 때문이다. 비슷한 경우로, 액티비티가 사용자에게 표시되지 않는 상황에서 더 우선순위가 높은 애플리케이션이 시스템 자원을 더 요구하는 경우를 들 수 있다. 이런 일이 벌어지면 안드로이드가 우선 순위가 낮은 프로세스와 액티비티를 끝낼 수 있다.

하지만 액티비티가 다시 초기화 되더라도 상태를 유지해서 액티비티가 시스템에 의해 강제적으로 파괴되거나 재생성되더라도 문제가 생기지 않아야만 하는 경우에는 어떻게 해야 할까? 안드로이드는 번들(bundle)이라 불리는 해법을 제공한다. 키와 값을 짝지은 형태로 직렬화한 데이터를 저장하기 위해 번들을 사용한다. 눈치챘겠지만, onCreate() 메서드에는 Bundle 타입의 파라미터가 있다. 이 파라미터를 통해 이전에 액티비티가 실행될 때 저장해둔 데이터가 전달된다. 번들에 데이터를 넣으려면 Activity의 onSaveInstanceState() 메서드를 오버라이드해야 한다.

예를 들어 계산기의 잘못된 동작을 수정하려면 계산기의 상태를 구성하는 디스플레이에 표시된 텍스트, lastResult, lastOp, waitingNextOperand 인스턴스 변수의 값을 저장해야 한다. 첫 단계는 onSaveInstanceState()를 오버라이드해서 번들 객체에 적당한 값을 써 넣는 것이다.

override fun onSaveInstanceState(outState: Bundle) {  
 super.onSaveInstanceState(outState)  
 outState.putString("currentText", txtResult.text.toString())  
 outState.putSerializable(::lastResult.name, lastResult)  
 outState.putSerializable(::lastOp.name, lastOp)  
 outState.putBoolean(::waitingNextOperand.name, waitingNextOperand)  
}

이제 설정 변경이나 다른 프로세스에 디바이스 자원을 양보해야 하는 등의 이유로 인해 액티비티가 일시적으로 파괴되고 나중에 재생성되더라도 시스템이 번들을 유지하고 액티비티 초기화시 onCreate() 메서드에게 다시 넘겨준다. 두번째 단계는 번들에서 값을 읽는 코드를 onCreate() 구현에 추가하는 것이다.

override fun onCreate(savedInstanceState: Bundle?) {  
 ...  
   
 clearText()  
   
 savedInstanceState?.let {  
 binding.txtResult.text = it.getString("currentText")  
 lastResult = it.getSerializable(::lastResult.name) as BigDecimal  
 lastOp = it.getSerializable(::lastOp.name) as OpKind?  
 waitingNextOperand = it.getBoolean(::waitingNextOperand.name)  
 }  
}

onCreate()에 null이 전달될 수도 있다. 이런 경우는 번들이 없는 경우다. 예를 들어 액티비티가 맨 처음 만들어질 때는 번들이 없다.

직렬화 가능한 값이라면 무엇이든 번들에 저장할 수 있다는 점에 유의하라. 직렬화가 불가능한 객체를 저장해야 한다면 Serializable 인터페이스를 구현하거나 번들에 해당 객체를 직접 쓰는 대신 다른 직렬화가능한 데이터 형태로 변경하거나, 원래의 객체를 부분부분 나눠서 직렬화한 후 다시 복구해야 한다.

추가로 언급할만한 내용으로는, 번들 직렬화가 메인 스레드에서 이뤄지며 시스템 프로세스의 메모리를 잡아먹기 때문에 상대적으로 크기가 작은 일시적인 데이터를 저장할 때만 번들을 사용해야 한다는 점이 있다. 다른 경우에는 사용자 선호도(user preference)나 SQLite 데이터베이스를 사용해야 한다.

onCreate()와 onSaveInstanceState() 메서드는 액티비티의 상태가 생명주기상 새로운 상태로 전이될 때 모두 안드로이드 운영체제가 호출해주는 생명주기 콜백(lifecycle callback)이라는 메서드에 속한다. 예를 들어 Resumed 상태는 포그라운드(foreground)에서 실행되는 액티비티와 연관이 있고, Paused 상태는 백그라운드로 옮겨졌지만 여전히 사용자에게 보이는 액티비티에 해당하며, Stopped 상태는 액티비티를 사용자가 전혀 볼 수 없게 됐다는 뜻이다.

부모 클래스의 생명주기 콜백에는 액티비티가 제대로 작동하기 위해 필요한 공통 코드가 들어있기 때문에, 생명주기 콜백을 오버라이드한 메서드는 반드시 부모 클래스의 메서드를 호출해야만 한다.

IDE 팁: 안드로이드 스튜디오는 생명주기 콜백을 오버라이드하면서 부모 클래스의 메서드를 호출하지 않는 경우 오류를 보고해준다(그림 15.12)

## 젯팩 콤포즈 레이아웃 DSL

[안코가 deprecated이기 때문에, 아직 개발중이지만 구글에서 밀어주고 있는 젯팩 콤포즈로 바꾸겠습니다]

2019년까지는 안코(Anko)라는 라이브러리가 활발히 유지보수되고 쓰였다. 안코는 안드로이드 애플리케이션 개발을 단순화해주는 여러가지 향상된 기능을 제공하는 코틀린 프레임워크다. 안코의 주요 특징은 다음과 같다.

* 안코 커먼즈(Anko Commons) 라이브러리와 함께, 대화창, 인텐트(intent, 새로운 액티비티를 시작하기 위해 사용하는 객체), 로깅(logging), 애플리케이션 리소스 등을 다룰 수 있는 다양한 도우미 함수를 제공한다
* UI 레이아웃을 지정할수 있는 영역 특화 언어(Anko Layouts)를 제공한다
* 안드로이드의 로컬 저장소로 흔히 사용되는 SQLite 접근을 편하게 해주는 API(Anko SQLite)를 제공한다

하지만 코틀린이 안드로이드의 주요 언어로 사용되고, 다양한 코틀린 기반 툴들이 등장했기 때문에, 2019년 말 이후 안코는 더 이상 지원이 이뤄지지 않고 있다. 몇가지 안코에 대한 대안을 정리하면 다음과 같다.

* 레이아웃 DSL은 젯팩 콤포즈(Jetpack Compose, https://developer.android.com/jetpack/compose)나 스플리티즈(Splitties)의 Views DSL(https://github.com/LouisCAD/Splitties/tree/main/modules/views-dsl) 등이 있다.
* 안코 커먼즈에 해당하는 일반적인 유틸리티로는 안드로이드 KTX(https://developer.android.com/kotlin/ktx)나 스플리티즈(https://github.com/LouisCAD/Splitties) 등이 있다.
* SQLite에 대한 유틸리티로는 룸(Room, https://developer.android.com/topic/libraries/architecture/room)이나 SQLdelight(https://github.com/cashapp/sqldelight)가 있다.

특히 젯팩과 KTX, 룸은 모두 구글이 지원하고 있기 때문에 내부를 살펴보고 프로젝트에 적용해볼만하다. 여기서는 간단한 예제로 계산기 앱의 레이아웃을 젯팩을 사용해 다시 작성해보자. 다른 예제나 젯팩 콤포즈에 대한 자세한 내용은 https://developer.android.com/jetpack/compose 를 보라.

젯팩 콤포즈를 사용하려면 카나리 버전의 안드로이드 스튜디오가 필요하다. 먼저 https://developer.android.com/studio/preview에서 카나리 15 버전을 설치하라. 그 후, :app의 그레이들 빌드파일의 android 안의 buildFeatures에서 젯팩 콤포즈를 활성화해야 한다. 그리고 마지막에 composeOptions에 코틀린 컴파일러 확장 버전을 기술하라.

android {  
 ...  
 buildFeatures {  
 compose true  
 }  
 ...  
 composeOptions {  
 kotlinCompilerExtensionVersion '1.0.0-beta01'  
 }  
}

그 후 다음과 같이 젯팩 툴킷에 대한 의존관계를 추가하라.

dependencies {  
 implementation 'androidx.compose.ui:ui:1.0.0-beta01'  
 // 프리뷰 등 툴링 지원  
 implementation 'androidx.compose.ui:ui-tooling:1.0.0-beta01'  
 // Border, Background, Box, Image, Scroll, shapes, animations 등 파운데이슨 클래스  
 implementation 'androidx.compose.foundation:foundation:1.0.0-beta01'  
 // 머티리얼 디자인 관련  
 implementation 'androidx.compose.material:material:1.0.0-beta01'  
 implementation 'androidx.compose.material:material-icons-core:1.0.0-beta01'  
 implementation 'androidx.compose.material:material-icons-extended:1.0.0-beta01'  
 // 액티비티 통합  
 implementation 'androidx.activity:activity-compose:1.3.0-alpha03'  
 // 뷰모델 통합  
 implementation 'androidx.lifecycle:lifecycle-viewmodel-compose:1.0.0-alpha02'  
 // 옵저버블(Observable) 통합  
 implementation 'androidx.compose.runtime:runtime-livedata:1.0.0-beta01'  
 implementation 'androidx.compose.runtime:runtime-rxjava2:1.0.0-beta01'  
  
 // UI 테스트  
 androidTestImplementation 'androidx.compose.ui:ui-test-junit4:1.0.0-beta01'  
}

설정을 변경하면 IDE가 이를 감지하고 그림 15.13처럼 그레이들 동기화를 하라는 메시지를 표시한다.

그림 15.12: 안드로이드 스튜디오가 그레이들 동기화를 권장함

Sync Now를 클릭하면 안드로이드 스튜디오가 내부 프로젝트 모델을 그레이들 설정에 따라 업데이트한다. 이제 여러가지 젯팩 콤포즈 기능을 활용할 수 있다.

이제 DSL로 뷰 레이아웃을 기술해보자. 콤포넌트를 @Composable 애너테이션이 붙은 함수를 활용해 작성하기 때문에, 함수 내부에 상태를 가두고 리스너가 이 상태를 활용하는 방식으로 외부와의 접점을 최소화하기도 한다. 또는 액티비티 클래스에 속한 필드를 활용할 수도 있다. 이 예제에서는 두가지방식을 모두 보여준다.

import androidx.appcompat.app.AppCompatActivity  
import android.os.Bundle  
import android.widget.Toast  
import androidx.activity.compose.setContent  
import androidx.compose.foundation.layout.\*  
import androidx.compose.material.Button  
import androidx.compose.material.Text  
import androidx.compose.material.TextField  
import androidx.compose.runtime.Composable  
import androidx.compose.runtime.mutableStateOf  
import androidx.compose.runtime.remember  
import androidx.compose.runtime.getValue // by remember{mutableStateOf(...)} 를 사용하려면 꼭 써야 함   
import androidx.compose.runtime.setValue // by remember{mutableStateOf(...)} 를 사용하려면 꼭 써야 함   
import androidx.compose.ui.Modifier  
import androidx.compose.ui.text.TextStyle  
import androidx.compose.ui.unit.dp  
import androidx.compose.ui.unit.sp  
import java.math.BigDecimal  
import java.math.RoundingMode  
  
class MainActivity: AppCompatActivity() {  
 enum class OpKind {  
 ADD, SUBTRACT, MULTIPLY, DIVIDE  
 }  
  
 companion object {  
 fun OpKind.compute(a: BigDecimal, b: BigDecimal) = when (this) {  
 OpKind.ADD -> a + b  
 OpKind.SUBTRACT -> a - b  
 OpKind.MULTIPLY -> a \* b  
 OpKind.DIVIDE ->a.divide(b, 10, RoundingMode.HALF\_EVEN)  
 }  
 }  
  
 private var lastResult: BigDecimal = BigDecimal.ZERO;  
 private var lastOp: OpKind? = null  
 private var waitingNextOperand: Boolean = false  
 private var textSaved: String = "" // MainActivityLayout안의 텍스트를 손쉽게 저장하 위해  
  
 // 버튼을 쉽게 만들기 위한 @Composable 함수  
 @Composable  
 fun calcButton(text:String, callback:()->Unit) =  
 Button(callback, Modifier.padding(10.dp)){Text(text)}  
  
 @Composable  
 fun MainActivityLayout(last: String) {  
 var text by remember{mutableStateOf("")}  
  
 Column(Modifier.fillMaxHeight()) {  
 TextField(value=text, onValueChange = {textSaved=it}, label={}, textStyle = TextStyle(fontSize=36.sp))  
 Row {  
 Column(Modifier.weight(3f)){  
 Row {  
 calcButton("7"){ text+='7' }  
 calcButton("8"){ text+='8' }  
 calcButton("9"){ text+='9' }  
 }  
 Row {  
 calcButton("4"){ text+='4' }  
 calcButton("5"){ text+='5' }  
 calcButton("6"){ text+='6' }  
 }  
 Row {  
 calcButton("1"){ text+='1' }  
 calcButton("2"){ text+='2' }  
 calcButton("3"){ text+='3' }  
 }  
 Row {  
 calcButton("0"){ text+='0' }  
 calcButton("."){ text+='.' }  
 calcButton("+/-"){  
 text = when {  
 text.startsWith("-") ->  
 text.substring(1, text.length)  
 text != "0" -> "-$text"  
 else -> text  
 }  
 }  
 }  
 Row {  
 calcButton("<-"){  
 text = text.dropLast(1).let{  
 if(it=="-") "" else it  
 }  
 }  
 calcButton("C"){ text="" }  
 }  
 }  
 Column(Modifier.weight(1f)){  
 }  
 Column(Modifier.weight(1f)) {  
 fun calc(nextOp: OpKind?) {  
 if (waitingNextOperand) {  
 lastOp = nextOp  
 return  
 }  
 val currentValue = BigDecimal(text)  
 val newValue = try {  
 lastOp?.compute(lastResult, currentValue) ?: currentValue  
 } catch (e: ArithmeticException) {  
 lastOp = null  
 waitingNextOperand = true  
 Toast.makeText(  
 applicationContext,  
 "Invalid operation!",  
 Toast.LENGTH\_SHORT  
 ).show()  
 return  
 }  
  
 if (nextOp != null) lastResult = newValue  
 if (lastOp != null) text = newValue.toPlainString()  
  
 lastOp = nextOp  
 waitingNextOperand = nextOp != null  
 }  
  
 calcButton("+"){ calc(OpKind.ADD) }  
 calcButton("-"){ calc(OpKind.SUBTRACT) }  
 calcButton("\*"){ calc(OpKind.MULTIPLY) }  
 calcButton("/"){ calc(OpKind.DIVIDE) }  
 calcButton("="){ calc(null) }  
 }  
 }  
 }  
 }  
  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
  
 var currentText = ""  
  
 savedInstanceState?.let {  
 lastResult = it.getSerializable(::lastResult.name) as BigDecimal  
 lastOp = it.getSerializable(::lastOp.name) as OpKind?  
 waitingNextOperand = it.getBoolean(::waitingNextOperand.name)  
 currentText = it.getString("currentText") ?: ""  
 }  
  
 setContent { MainActivityLayout(currentText) }  
 }  
  
  
 override fun onSaveInstanceState(outState: Bundle) {  
 super.onSaveInstanceState(outState)  
 outState.putString("currentText", textSaved)  
 outState.putSerializable(::lastResult.name, lastResult)  
 outState.putSerializable(::lastOp.name, lastOp)  
 outState.putBoolean(::waitingNextOperand.name, waitingNextOperand)  
 }  
}

주석에도 적어뒀지만, @Composable 함수 내부에서 var text by remember{mutableStateOf("")} 처럼 상태를 저장한 경우, getValue와 setValue를 임포트해야만 한다. MainActivityLayout는 콤포즈가 제공하는 Column과 Row 레이아웃을 사용해 내부에 버튼을 배치하면서, Modifier.padding(10.dp)으로 버튼 주변의 네 방향에 패딩을 10dp씩 줬다. 이렇게 구성한 계산기를 실행하면 그림 15.10과 유사한 모양의 계산기를 볼 수 있다.

## 결론

이번 장에서는 안드로이드 스튜디오 IDE의 기본 기능을 배우고, 코틀린 언어를 사용해 안드로이드 앱을 만들어 봤다. 안드로이드 액티비티의 개념을 소개하고, UI 레이아웃을 기술하는 방법을 맛봤으며, 구글이 제공하는 젯팩의 뷰 바인딩과 콤포즈에 대해 살펴봤다.

다음 장에서는 Ktor 프레임워크를 사용해 웹 애플리케이션을 개발하는 방법을 배운다. 기본적인 Ktor의 기능을 다루고, 클라이언트 요청을 처리하기 위해 라우팅 메커니즘을 사용하는 방법도 살펴본다. 그리고 HTML 컨텐트를 생성할 때 쓰려는 목적으로 만들어진 DSL에 대해서도 살펴본다.

## 문제

1. 안드로이드 스튜디오에서 프로젝트를 어떻게 설정하는지 설명하라.
2. 안드로이들 스튜디오의 그레이들 설정에 대해 설명하라. 새로운 의존관계를 어디에 추가해야 하는가?
3. 애플리케이션을 실행하기 위한 가상 디바이스를 설정하는 방법을 설명하라.
4. 액티비티란 무엇인가? 안드로이드 애플리케이션에서 UI를 어떻게 기술하는가?
5. 젯팩이 제공하는 뷰 바인딩에 대해 설명하라. 어떻게 하면 젯팩 뷰 바인딩을 사용할 수 있는가?
6. 액티비티가 파괴될 때 액티비티의 현재 상태를 저장했다 나중에 복구하는 방법은 무엇인가?
7. 젯팩 콤포즈의 UI 레이아웃 DSL에 대해 설명하라.