목차

Chapter 1 - 안드로이드 개요와 개발 환경 설치

안드로이드의 주요기능 1

1. 애플리케이션 프레임워크를 통해서 제공되는 API를 사용함으로써 코드를 재사용하여 효율적이고 빠른 애플리케이션 개발 가능
2. 모바일 기기에 최적화된 달빅 또는 아트런타임 제공
3. 2D 그래픽 및 삼차원 그래픽을 최적화하여 표현
4. 모바일용 데이터베이스인 SQLite제공
5. 각종 오디오, 비디오 및 이미지 형식 지원
6. 모바일 기기에 내장된 각종 하드웨어 지원
7. 이클립스 IDE 또는 Android Studio를 통해 빠른 개발 환경 제공

안드로이드의 주요기능 2

1. 롤리팝(5.0) : 다양한 안드로이드 기기 통합지원
2. 마시멜로(6.0) : 앱 권한설정, 지문 인식 지원
3. 누가(7.0) : 가상현실 지원 및 3D게임, 알림 기능향상, 다중 창 열기 지원
4. 오레오(8.0) : PIP, 알림, 자동 채우기, 배터리 강화 등 지원
5. 파이(9.0) : 실내 위치 추적, 향상된 알림, 멀티카메라, 인공지원 확장 지원
6. Android 10.0(Q) : 라이브 캡션, 스마트 재생, 청각 보조, 동작 내비게이션, 어두운 테마, 개인 정보 제어등 지원

순서 : 롤리팝 -> 마시멜로 -> 누가 -> 오레오 -> 파이 -> Q

안드로이드의 특징

1. Kernel : Linux base

> 번외로 리눅스 기반이지만 리눅스가 아닌 이상한 관계임..

2. 개발언어 : Java, Kotlin
3. SDK(Software Development kit) : 많은 라이브러리 제공
4. 오픈 소스
5. 살아있는 언어(즉, 지속적인 업데이트 한다.)

안드로이드 구조

1번	**응용 프로그램**
2번	**응용 프로그램 프레임워크**
3번	**라이브러리 | 안드로이드 런타임(DVM)**
4번	**리눅스 커널**

> 참고로 응용 프로그램 프레임워크의 Activtiy는 안드로이드에서 page 단위이다.

- 응용 프로그램
  - 안드로이드 스마트폰에서 사용하는 일반적인 응용프로그램(앱)
  - 웹 브라우저, 달력, 구글맵, 연락처, 게임 등 사용자 입장에서 사용
- 응용 프로그램 프레임워크
  - 안드로이드 API가 존재하는 곳
  - 안드로이드폰 하드웨어에 접근할 때 API를 통해서만 가능

    > 안드로이드폰 하드웨어 : 블루투스, 카메라, WiFi 등
- 안드로이드 런터임(Android Runtime)
  - Java 코어 라이브러리와 달빅 가상 머신(DVM) 또는 아트 런타임으로 구성
  - 안드로이드는 Java 또는 kotlin 문법으로 프로그래밍하지만 Java 가상 머신을 사용하지 않고 이곳의 달빅 가상 머신이나 아트 런티임 사용
    => 즉, **안드로이드가 Java, kotlin으로 컴파일 가능하지만 JVM이 아닌 DVM 또는 아트런타임 사용한다.**
- 라이브러리
  - 시스템 접근성 때문에 **C언어로 작성됨**
    - 시스템 접근 관련 작업을 위한 라이브러리는 대부분 C/CPP 으로 작성됨
- 리눅스 커널(Linux Kernel)
  - 하드웨어의 운영과 관련된 **저수준의 관리 기능**
  - 메모리 관리, 디바이스 드라이버, 보안 등

> 운영체제 역할을 수행한다. **사실, 커널이 OS의 일부임.**

Chapter 2 - 처음 만드는 안드로이드 애플리케이션

안드로이드 프로젝트 개발 단계

1번	**안드로이드 프로젝트 생성**
2번	화면 디자인 및 편집(.XML)
3번	Kotlin 코드 작성(.kt)
4번	프로젝트 실행 및 결과 확인(AVD)
5번	안드로이드 애플리케이션 개발 완료

2. XML 파일에는 각 위젯의 **ID, 크기 등 속성 값이 들어있다.**
3. Kotlin 코드를 통해서 변수를 만들고 **위젯과 맵핑 및 이벤트를 호출한다.**

사소한 표준 룰

xml에서 위젯 다루기

```
<Button
    android:layout_width = "match_parent"
    android:layout_height = "wrap_content"
    android:id = "@+id/button1"
    android:text = "@string/strBtn1">
    <!-- text = @string/strBtn1 요놈의 의미는 -->
    <!-- [app] -> [res] -> [values] -> [strings.xml]에 저장돼있는 strBtn1 변수가 가지는 text를 이용하겠다는 의미임. -->
    <!-- 그냥 android:text = "버튼입니다" 이렇게 하면 직접적으로 위젯의 이름을 설정한다는 의미 -->
    <Button>
```

`activity_main.xml`에 작성된 `Button`에 대해 접근해야하므로 버튼에 대한 멤버변수를 하나 만들어야함.

```
lateinit var button1 : Button // Button 객체를 가질 변수 선언.
//lateinit이라는 것은 필요할 때 즉, 실행시 초기화 (이벤트 발생시 초기화 하겠다는 의미 즉, 동적 바인딩이다.)
// 이때 val(상수) 선언으로 되어있는지 check 필요함. (상수 선언시, 이벤트로 인한 값 변경 불가)
button = findViewById<Button>(R.id.xml에 작성한 버튼 id)
// 이때 Button 위젯에 관련된 클래스가 import되어 있어야함.
```

`Button`이벤트 생성하기

```
button1.setOnClickListener {
    // 버튼 클릭시 작동할 내용 작성
    // setOnClickListener는 람다식으로 작성됨.
}
```

[정리]

1. 위젯 변수 선언. -> `lateinit var button1 : Button`
2. xml 위젯과 맵핑 -> `button1 = findViewById\<Button>(R.id.xml의 버튼 id)`
3. 이벤트 생성 -> `button1.setOnClickListener {}`

프로젝트에서 사용되는 폴더와 파일의 용도

[Java 폴더]
하위에 **패키지명의 폴더가 있는데, 이는 안드로이드 프로젝트를 생성할 때 입력한 패키지 이름과 동일.**
이곳에 `MainActivity.kt` **있음**

[java (generated) 폴더]
Android Studio 3.2 부터 제공됨. **시스템 내부적으로 사용되므로 딱히 신경쓸게 없다.**

[res 폴더]
앱 개발에 사용되는 **이미지(image), 레이아웃(layout), 문자열(String)** 등이 들어감.

- [drawable 폴더]
  이미지 파일을 넣음
- [mipmap 폴더] 디자인 화면이나 앱이 설치된 후에 보이는 런처 아이콘을 넣음
- [layout 폴더] 액티비티(화면)을 구성하는 XML파일 들어있음 기본적으로 `activity_main.xml`이 초기화면임.
- [values 폴더]
  문자열 저장하는 `string.xml` 색상표 저장하는 `colors.xml` 스타일 저장하는 `styles.xml`등이 들어 있음.

[res (generated) 폴더]
Android Studio 3.5 제공됨 **내부적으로 사용되므로 신경쓸 필요없음**

[manifests 폴더]
`AndroidManifest.xml`파일이 들어 있음 **앱의 여러가지 정보를 담고 있는 중요한 파일임**

[Gradle Scripts 폴더]
`Gradle` 빌드 시스템과 관련된 파일 있음
`build.gradle (Module: app)` : 빌드 스크립트 핵심 파일
`local.properties` : 컴파일되는 SDK의 경로가 들어 있음
`gradle.properties` : JVM 관련 메모리가 설정되어 있음

> Gradle : 그루비(Groovy)를 기반으로 한 빌드 도구이다.

Chapter 3 - 안드로이드를 위한 Kotlin 문법

Kotlin의 특징

1. Java와 100% 상호 호환됨
2. **Java보다 문법이 간결함**
3. 프로그램의 안정성을 높여줌 (왜?)
4. `var` OR `val` 예약어를 통해 데이터 형식을 선언하지 않고 변수 선언 가능.

Kotlin으로 Hello world 출력

```
fun main(){
    println("안드로이드를 위한 Kotlin 연습")
    // 세미콜론 없음 python 처럼.
}
```

변수와 데이터 형식

정수형, 실수형, 문자형, 문자열 변수 선언하기.

```
fun main(){
    var var1 : Int = 10
    var var2 : Float = 10.1f  // Float 타입은 f 붙어야함. (자바도 그랬던것 같기도,,,)
    var var3 : Double = 10.2
    var var4 : Char = '안'
    var var5 : String = '안드로이드'

    // var 변수명 : 타입 = 초기값
    // var 변수명 : 타입
    // 각 타입 이름 첫 글자는 항상 대문자!

}
```

Kotlin의 변수 선언 방식

1. 암시적 선언 : 변수의 데이터 형식을 지정하지 않고, 대입되는 값에 따라 자동으로 변수의 데이터에 형식이 지정 ex) `var va1 = 10.1f` Float 타입 의미
   * **단, 초기화 하지 않은 경우에는 데이터 형식을 반드시 명시 해야함.**

```
fun main(){
    var va1 : Int // 초기화 하지 않는 경우 타입 명시
    var va1 : Int = 10 // 정석
```

```
    var va1 = 10 // 암시적 선언
}
```

2. 데이터 형식 변환(타입 캐스팅) : 캐스팅 연산자 사용 (`toInt()`, `toDouble()`) 즉, 정적 Method 사용
   * **코틀린은 b = (float)a 이런 문법 지원 x** (즉, 묵시적 타입 캐스팅 안됨)

```
fun main(){
    var a : Int : "100".toInt() // 문자열 -> 정수
    var b : Double : "100.123".toDouble() // 문자열 -> 실수
}
```

3. Null 사용 : 코틀린은 기본적으로 변수에 Null 넣을 수 없음.
   * **변수 선언시 데이터 형식 뒤 ? 를 붙여야됨**

```
fun main(){
    var noNull : Int = null // 오류
    var okNull : Int? = null // 정상

    var ary = ArrayList<Int>(1) // 1개짜리 배열 리스트 생성
    ary!!.add(100) // 값 100을 추가
    ary!!.add(null) // 오류
}
```

Kotlin 조건문 if, when

---

1. if(이중 분기)

```
fun main(){
    if(조건식){
        // true 실행문
    }
    else{
        false 실행문
    }
}
```

2. when : 스위치랑 유사함.(다중 분기)

```
fun main(){
    when(식){
        값1 -> 실행문 // 값1이면 이 부분 실행
        값2 -> 실행문 // 값2이면 이 부분 실행
        else -> //어디에도 해당 없으면 이 부분 실행 (스위치의 default랑 유사)
```

```
    }
}
```

[종합적인 예시]

```
fun main(){
    var count : Int = 85
    if(count >= 90){
        println("합격")
    }
    else if(count >= 60){
        println("합격인데 좀 에매")
    }else{
        println("fail")
    }

    var jumsu : Int = (count/10) * 10
    when(jumsu){
        100 -> println(100점)
        90 -> printlnt(90점)
        80,70,60 -> println("합격") //여러개 쌉가능
        else -> println("fail")
    }
}
```

[when 심화]
범위로 처리하기 in이라는 키워드 있어야 사용 가능
in min .. max -> **min 부터 max까지**

```
fun main(){
    var jumsu : Int = (count/10) * 10
    when(jumsu){
        in 90 .. 100 -> println("장학생")
        in 60 .. 89 -> println("합격")
        else -> println("fail")
    }
}
```

Kotlin 배열

---

1. 일차원 배열 선언 형식

```
fun main(){
    var array = Array<데이터 타입>(개수, {초기값})
    var array = Array<데이터 타입>(개수) {초기값}
    // 단 타입이 명시되어있으면 초기값은 의무 아님
```

```
    // 변수 선언이랑 비슷한 맥락
}
```

[실사용 예시]

```
fun main(){
    var one = Array<Int>(4, {0}) // 크기 4짜리 1차원 배열 생성 초기값은 0
    one[0] = 10 // 0번 index에 data 10 넣기
    one[3] = 20 // 3번 index에 data 20 넣기
}
```

2. 이차원 배열 선언 형식

```
fun main(){
    var array = Array<배열 데이터 형식>(행 개수, {배열 데이터 형식(열 개수)})
}
```

[실사용 예시]

```
fun main(){
    var two = Array<IntArray>(3 {IntArray(4)})
    var two = Array<IntArray>(3) {IntArray(4)} // 3행 4열 배열 생성
    // IntArray => 정수형 배열 키워드
    // 약간 파이썬 같은 느낌인데..? 리스트안에 리스트 중첩
}
```

[주의사항]
행의 데이터 타입 == 열의 데이터 타입
무조건 IntArray 같은 타입사용해야함

3. 배열 선언하면서 바로 값을 넣기(초기화 값이 아님)

```
fun main(){
    var three : IntArray = intArrayOf(1,2,3) // intArrayOf 이용 (넣은 값 만큼 크기
할당)
}
```

4. `ArrayList` : 연결리스트 같은 느낌이다.

```
fun main(){
    var one = ArrayList<Int>(4)
    one.add(10)
    one.add(20)
```

```
    var hap = one.get(0) + one.get(1)
    // 첫 번째 값 + 두 번째 값
}
```

반복문 : for, while

여기서 `for`문이 조금 다르다

```
fun main(){
    var three : IntArray = intArrayOf(1,2,3)
    for (i in three.indices){
        println(one[i])
    }
    //indices -> 파이썬 range()같은 느낌
}
```

```
fun main(){
    for(변수 in 시작..끝 step 증가량){ // 약간 python range() 풀어서 쓴 느낌
        // for(i in 0..99 step 1) 이때 0 <= i  >= 99 임
        // 실행문
    }

    for(변수 in 배열 명.indices){
        // 배열의 개수만큼 변수에 대입해서 사용하는 방법임
        // 실행문
    }
    for(변수 in 배열 명){
        // 실행문
    }

    // 실사용
    var three : IntArray = intArrayOf(1,2,3)
    for(t in three){ // 파이썬 처럼 쓰면됨
        println(t)
    }
}
```

`while`은 그냥 동일함.

메소드와 전역변수, 지역변수

---

1. 메소드 : 기본 메서드인 `main()`함수 외에 사용자가 메소드를 추가로 생성할 수 있음
2. 전역, 지역변수 : 알잖아...
3. 함수에 리턴타입 지정 가능

```
fun add(x:Int, y:Int) : Int {
    // add()의 리턴 타입은 Int임.
    // 반드시 return 할 경우 타입 명시해야함.
}
```

*** 리턴 타입 명시 하지않을 경우 해당 메서드는 Void타입이다. * main() 안에 메서드 정의 가능**

예외처리

`try-catch` 문을 통해서 가능.

```
fun main(){
    var num1 : Int = 100
    var num2 : Int = 0
    try{
        println(num1/num2)
    }catch(e : ArithmerticException){
        println("계산에 문제가 있습니다.")
    }
}
```

클래스 정의와 인스턴스 생성

클래스 : 필드변수와 메소드로 구성

```
Class Car{
    var color : String = ""
    var speed : Int = 0

    fun upSpeed(value : Int){
        if((speed + value) >= 200)
            speed = 200
        else
            speed = speed + value
    }

    fun downSpeed(value : Int){
        if(speed-value <= 0 )
            speed = 0
        else
            speed = speed - value
    }
}

fun main(){
```

```
    var myCar1 : Car = Car() // 이때 컴파일러가 알아서 기본 생성자 만들어 둔다.
}
```

[사용자 생성자 만들기]
`constructor`키워드 사용

```
constructor(color:String, speed:Int){
    this.speed = speed
    this.color = color
}
```

메소드 오버로딩(overloading) And 메소드 오버라이드(override)

Method overloading : 리턴타입에는 영향없고 매개변수만 상관있음

> 한 클래스 내에서 메소드의 이름이 같아도 **파라미터의 개수나 데이터 형식만 다르면 여러 개를 선언할 수 있음**

Method override : 이름, 매개변수 같은데 실행문만 다름.

정적 필드, 정적 메소드, 상수 필드

---

1. 정적필드(Static field) : 인스턴스 생성 없이 클래스 자체적으로 사용됨. `companion object{}` **안에 작성하여 정적 필드 만듦**
2. 정적메소드(Static method) : `companion object{}` 안에 작성 하면 됨
3. 상수 필드 : `Static field`에 초기값 입력 후 `const val`로 선언 (val도 상수 변수 의미 단, **정적 필드의 상수화는 const키워드 필요**)

```
Class Car{
    var color : String = ""
    var speed : Int = 0

    companion object{ // 정적 필드, 정적 메서드
        var carCount : Int = 0
        fun upSpeed(value : Int){
            if((speed + value) >= 200)
                speed = 200
            else
                speed = speed + value
        }

        fun downSpeed(value : Int){
            if(speed-value <= 0 )
                speed = 0
            else
```

```
                speed = speed - value
        }
    }
}

fun main(){
    var myCar1 : Car = Car() // 이때 컴파일러가 알아서 기본 생성자 만들어 둔다.
}
```

클래스 상속(inheritance)

클래스 상속 : 기존 클래스를 그대로 물려받으면서 필요한 메서드를 추가로 정의하는 것.

- 슈퍼클래스 : 부모 클래스라고 부름
- 서브클래스 : 슈퍼클래스의 모든 필드, 메서드를 상속받음

`open`이라는 키워드가 있어야 **오버라이딩 가능** 오버라이딩 할땐, `override`라고 명시해야함 (Java의 어노테이션과 유사)

서브 클래스 선언시에 타입 지정 필요함

```
open class Car{
    // 슈퍼클래스 누군가 상속 받을 예정이라 open 키워드 사용
    var color : String = ""
    var speed : Int = 0

    open fun upSpeed(value : Int){
        if((speed + value) >= 200)
            speed = 200
        else
            speed = speed + value
    }

    open fun downSpeed(value : Int){
        if(speed-value <= 0 )
            speed = 0
        else
            speed = speed - value
    }
}
class Automobile : Car() {
    //Car() 이렇게 하면 부모 생성자 까지 호출하면서 가져오는 것.
}
class Automoblie : Car {
    constructor() {
        // 요렇게 하면 Car() 안해도 됨.
    }
}
```

추상 클래스와 추상 메서드

- 추상(abstract) 클래스
  - 인스턴스화를 금지하는 클래스 -> `var animal : Animal()` 이런거 못함 `var animal : Animal` 이래야함
  - Class 앞에 abstract 키워드 붙여서 사용(Java랑 동일)
- 추상 메소드(abstract method)
  - **본체가 없는 메소드**
  - abstract 키워드 붙여서 사용
    추상 메서드를 포함하는 클래스는 추상 클래스 밖에없음.

추상 클래스와 추상 메서드를 사용하는 목적 : 공통적으로 사용되는 기능을 추상 메드로 선언하고 상속 후 구체화함.

즉, 구현한다(implement) 는 의미..?

인터페이스(interface)와 다중 상속

인터페이스(interface) : 추상 클래스와 성격이 비슷.

- > 인터페이스는 상속 보단 구현한다는 표현을 함.

```
abstract class Animal { // 추상 클래스
    var name : String = ""
    abstract fun move() // 추상 메서드
}

interface iAnimal { // 인터페이스
    abstract fun eat() // 추상 메서드
}

class iCat : iAnimal { // iAnimal 구현
    override fun eat() { // 오버라이드
        println("생선을 좋아한다.")
    }
}

class iTiger : Animal(), iAnimal { // 다중 상속 흉내 (Animal은 상속, iAnimal은 구현)
    override fun move() {
        println("네 발로 이동한다.")
    }
    override fun eat() {
        println("멧돼지를 잡아 먹는다.")
    }
}

class Eagle : Animal() {
```

```
    var home : String = ""
    override fun move() {
        println("날개로 날아간다.")
    }
}

fun main() {
    var cat = iCat() // 인터페이스는 인스턴스화 가능
    cat.eat()

    var tiger = iTiger()
    tiger.move()
    tiger.eat()
}
```

람다식(Lambda expression)

익명함수 형태로 간단히 표현 한 것

[파라미터 2개를 받아서 합계를 출력하는 일반적인 메소드 형식]

```
fun addNumber(n1:Int, n2:Int) : Int{
    return n1 * n2
}
```

[람다식 표현]

```
val addNumber = {n1:Int, n2:Int -> n1 + n2} // var을 사용해서 선언도 가능하지만 추천
x
                  // 매개변수 -> 실행문
```

람다식 특징

1. 람다식은 {}로 감싸며 fun 예약어 사용안함
2. {}안쪽 -> 왼쪽은 매개변수, 오른쪽은 함수의 실행문
3. -> 오른쪽 문잔이 여러 Line이면 세미콜론으로 구분 (**한줄로 작성하는 경우 적용됨**)
4. 내용 중 마지막 문장은 반환 값(return)

추후 Event Listener 사용시 많이 적용됨

람다식 심화

람다식의 구성

변수에 지정된 람다식 (**변수를 함수처럼 사용가능**)

`val multi: (Int, Int) -> Int = {x:Int, y:Int -> x*y}`
변수명 : 람다식의 선언 자료형(매개변수에 자료형이 명시된 경우 생략가능) = {매개변수:자료형 -> 실행문}
람다식의 선언 자료형을 명시했을 경우 매개변수 타입 생략 가능

[Void OR 매개변수가 하나 있을 때]

- `val greet: ()->Unit = {println("Hello World!")}` But, ()-> Unit 생략가능
- `val square:(Int)->Int = {x -> x*x}`

[람다식 내부에 람다식 있는 경우] - 해당 문법은 자주 사용하지 않음

- `val nestedLambda: () -> () -Unit = {{println("nested")}}`

[선업부의 자료형 생략]

- `val greet = {println("Hello World!")}` -> 추론 가능
- `val square = {x:Int -> x * x}` -> **선언 부분을 생략하려면 x의 자료형 명시해야함**
- `val nestedLambda = {{println("nested")}}` -> 추론 가능

[매개변수를 람다식으로 사용하는 메서드]

```
fun main(){

    var result : Int // 초기값 없는 변수 선언

    //람다식을 매개변수와 인자로 사용한 함수
    result = highOrder({x, y -> x + y}, 10, 20) // 람다식, 10, 20 매개변수로 넘김
    println(result)
}

fun highOrder(sum: (Int, Int)->Int, a: Int, b:Int) : Int {
    /*
        (람다식, 변수 a, 변수b )
        sum이라는 람다식은 정수 자료형 2개를 매개변수로 받아 정수형을 리턴하는 람다식임.
    */
    return sum(a, b) // 람다식 호출
}
```

[Call by Value] **람다식을 Call By Value로 호출하면 메서드의 매개변수는 일반 변수 자료형 사용 됨**

```
fun main(){
    val result = CallByValue(lambda()) // 람다식 함수 호출
    println("result")
}

fun callByValue(b: Boolean) : Boolean { // 일반 변수 자료형으로 선언된 매개변수
    println("callByValue function")
}

val lambda: () -> Boolean = { // 람다 표현식
```

```
    println("lambda function")
    true // 반환 값
}
```

[Call by Name] **람다식을 Call by Name으로 호출하면 메서드의 매개변수는 람다식 함수 자료형으로 선언해야 함**

```
fun main(){
    val result = callByName(lambda) // 람다식 이름으로 호출
    println("result")
}

fun callByName(b: () -> Boolean) : Boolean { // 람다식 함수 자료형으로 선언된 매개변수
    println("callByName function")
}

val lambda: () -> Boolean = { // 람다 표현식
    println("lambda function")
    true // 반환 값
}
```

[한눈에 비교하기]

Call by Value	Call by Name
`fun callByValue(b:Boolean) : Boolean {}`	`fun callByName(b:()->Boolean) : Boolean {}`

[다른 함수의 참조에 의한 호출]

```
fun sum(x:Int, y:Int) = x+y

fun funcParam(a:Int, b:Int, c:(Int, Int)->Int) : Int{
    return c(a, b)
}

funcParam(3, 2, sum) // 오류, sum은 람다식 아님
funcParam(3, 2, ::sum) // 정상
```

고차함수

[고차함수] : **함수를 매개변수로 전달받거나 반환하는 함수를 의미함** (Callback에서 자주 쓰임)

```
fun calculator(a:Int, b:Int, operation: (Int, Int) -> Int) = operation(a, b) // operation이라는 람다식 사용됨
```

```
fun main(){
    val sum = {x:Int, y:Int -> x+y}
    println( calculator(1, 2, sum)) // 미리 정의한 람다식을 사용한 경우
    println( calculator(4, 3, {a:Int, b:Int -> a-b})) // 매개변수에 람다식을 넣어서
사용한 경우
}
```

[코드 분석]

`calculator(1, 2, sum)` 여기서 1, 2가 operation 즉, 람다식(sum의 a, b로 들어가서 a+b 형태로 리턴됨)

[응용 : 사칙연산 람다식]

```
fun main() {

    val sum = {a:Int, b:Int -> a+b} // 더하기 람다식
    val sub = {a:Int, b:Int -> a-b}
    val mul = {a:Int, b:Int -> a*b}
    val div = {a:Int, b:Int -> a/b}
    var a:Int = 10
    var b:Int = 5
    calc(a,b, sum)
    calc(a,b, sub)
    calc(a,b, mul)
    calc(a,b, div)

}

fun calc(a:Int, b:Int, option:(Int, Int)->Int) = option(a, b)
```

Chpater 4 - 기본 위젯 익히기

뷰(View) 클래스

- **안드로이드 화면에서 실제로 사용되는 것들은 모두 View 클래스의 상속을 받음**
    - 따라서 View 클래스의 속성과 메서드를 상속받는다.
- 위젯이라고 부름
- 레이아웃
    - 다른 위젯을 담을 수 있는 위제을 특별히 레이아웃이라고 함
    - 레이아웃도 크게 보면 위젯에 포함됨

뷰와 뷰그룹

- 안드로이드에서의 위젯
  - 다른 프로그램 언어에서 컨트롤이라고 부르는 것들

> 버튼, 텍스트뷰, 에디트텍스트, 라디오버튼, 이미지뷰 등

* 껍데기는 XML에 작성, 실제 구현은 kt파일

뷰 클래스 계층도

- **안드로이드 화면에 나타나는 모든 위젯은 `View` 하위에 존재함**(최상위 클래스는 Object)

View클래스의 XML속성

---

1. 버튼의 속성
   - xml 속성이 거의 없고 대개 상위 클래스인 `TextView` Or `View`에서 상속받는다.
     - **크기 관련 속성 layout_width, layout_height 필수 요소임**
     - id의 경우 필수는 아니지만 작성하지 않으면 이벤트 사용 불가
2. id 속성
   - **코틀린 코드에서 버튼 등의 위젯에 접근할 때 id 속성에 지정한 아이디를 사용** (findViewById(R.id.xml의 id))

xml에서 `android:id = '@+id/~~~'` 형식으로 쓰인다.
이를 코틀린에서 `위젯 변수 = findViewById<위젯>(R.id.위젯id)`

> btn1 = findViewById<Button>(R.id.Btn1) 이런 형식이다.

* 터치했을 때 어떤 동작이 필요한 경우 id 지정

3. `layout_width`, `layout_height` 속성 : 모든 위젯에 필수로 들어감
   - `match_parent` : 부모의 너비에 꽉차는 크기
   - `warp_content`: 위젯 내부의 컨텐츠에 크기를 맞춤 (ex. Button에 text속성이 버튼입니다. 경우 크기는 버튼입니다. 글자에 맞춤)
4. 값을 숫자로 직접 지정하는 경우 : **단위에 주의할 것** (단위 : px)
   - AVD는 해상도가 1080 x 1920인 경우 너비(width)를 1080px, 높이를 1920px로 지정하면 match_parent처럼 보임.
5. `background` 속성 : 레이아웃 색상 지정
6. `padding` 및 `layout_margin` 속성
   - padding : **레이아웃의 경계선과 위젯 사이에** 여백을 둘 수 있다.
   - margin : **위젯과 위젯 사이에** 여백을 둘 수 있다.
7. `visibility`속성 : 위젯을 보일 것인지 여부 결정(뭐 특정 버튼 누르면 보이도록 하는 방법에 쓰임)
   - `visible` : 디폴트로 설정되어 있으며, **보이는 상태**
   - `invisible` : 안보이는 상태이지만, **자신의 자리 유지함.**

    - `gone` : 안 보이는 상태, **원래 자신의 자리도 사라짐.**

8. `enabled`, `clickable` 속성 : Boolean타입의 값을 가지고 디폴트 값은 true임.
    - `enabled` : 위젯의 동작 여부
    - `clickable` : 클릭이나 터치가 동작 여부

9. `rotation` : 위젯을 회전시켜 출력 (값은 각도)

TextView

- `text` 속성 : 문자열 형식으로 값을 직접 입력하거나 `@string/변수명` 형식으로 지정 후 string.xml파일에 지정 가능
- `textColor` 속성 : 글자 색상
- `textSize` 속성 : 글자 크기(단위 : dp, px, in, mm, sp)
- `typeface` 속성 : 글자의 글꼴 지정
- `textStyle` 속성 : 글자 스타일 지정(글꼴이랑 착각 금지)
- `singleLine` 속성 : **글이 길어 줄이 넘어갈 경우 강제로 한 줄까지만 출력 그뒤에 ...처리

Kotlin코드로 XML속성 설정

```
override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // 변수 선언
        var tv1 : TextView
        var tv2 : TextView
        var tv3 : TextView

        // XML <-> kt 맵핑
        tv1 = findViewById<TextView>(R.id.textView1)
        tv2 = findViewById<TextView>(R.id.textView2)
        tv3 = findViewById<TextView>(R.id.textView3)

        tv1.setText("안녕하세요?") // android:text
        tv1.setTextColor(Color.RED) // android:textColor
        tv2.setTextSize(30.0f) // android:textsize
        tv2.setTypeface(android.graphics.Typeface.SERIF,
android.graphics.Typeface.BOLD_ITALIC) // android:typeface
        tv3.setText("가나다라마바사아자차카타파하가나다라마바사아자차카타파하갸냐댜랴먀
뱌샤")
        tv3.setSingleLine() // android:singleLing
    }
```

Button과 EditText

[클래스 계층도]

- Object
  - View
    - TextView
      - Button, EditText, RadioButton, CheckBox, ToggleButton 등.

[버튼 이벤트 생성]

`변수.setOnClickListener{}` 콜백 함수라고 부름(람다식 형태)
`변수.setOnTouchListener{view, motionEvent}`
*** 두 콜백함수의 차이점 : `setOnClickListener`의 경우 단순 터치시 동작함(터치만 인식함) 반면, `setOnTouchListener`는 터치, 드래그, 손가락 방향 등 다양한 액션을 인식 할 수 있다.**

[EditText 값 가져오기]
`String 변수 = EditText객체.getText().toString()` == `String 변수 = EditText객체.text.toString()` **차이점은 kotlin 메서드(getText()) 이냐 XML 속성(text)이냐??** -> just 뇌피셜

[onCreate 메서드] : 이놈 호출시 Main_activity 페이지 생성됨 (lateinit)

CompoundButton 클래스

[클래스 계층도]

- Object
  - View
    - TextView
      - Button
        - CompoundButton
          - Check Box, RadioButton, Switch, ToggleButton

특징 : **공통적으로 체크 또는 언체크 상태를 가질 수 있음**

1. 체크박스 : **독립 컨포넌트**
   - 여러 개의 체크박스가 있어도 서로 독립적으로 동작한다는 특징이 있어 여러 개를 동시에 체크할 수 있음.

[이벤트 구성] : `setOnCheckedChangeListener`

```
// 체크박스가 변경될 때 동작하는 람다식
변수명.setOnCheckedChangeListener{compoundButton, b ->
    // 이때 b는 boolean 타입을 말함 (체크 됨/안됨)
    // 이벤트 실행 부
    if(변수.isChecked == true){
        // 체크 된 상태
    }else{
        // 체크 안된 상태
    }
}
```

2. 스위치와 토글버튼 : 주 용도는 온/오프 상태 표시
    - 똑같이 이벤트는 `setOnCheckedChangeListener` 사용함.
3. 라디오버튼과 라디오그룹
    - **여러 개 중 하나만 선택해야 하는 경우에 사용**됨 (즉, 중복 x 단, **라디오 그룹과 같이 사용해야함**), **각 라디오 버튼마다 id가 있어야함**(없으면 선택된 것으로 fix되어버림)
    - **라디오그룹은 TextView 하위의 위젯들과는 성격이 조금 다르다**

        > `clearCheck()` 해당 라디오 그룹 내부의 체크된 것을 모두 해제함

```
<!-- 아래는 일부 속성이 생략된 코드임 -->
<RadioGroup
    android:id="@+id/rGroup1:"
    <RadioButton
        android1:id="@+id/radio1"
        android:text = "남성"/>
    <RadioButton
        android1:id="@+id/radio2"
        android:text = "여성"/>
    <!-- 라디오 그룹 내부에 2개의 라디오 버튼이 있음 -->
</RadioGroup>
<!-- 그룹 내부에 2개의 버튼이 있어서 둘 중 하나만 선택 가능함. -->
<!-- 그룹 밖에 2개 나열 하면 중복 선택 가능함 -->
```

이미지뷰와 이미지버튼

[클래스 계층도]

- Object
    - View
        - ImageView
            - ImageButton

특징 : 이미지뷰에 보여줄 그림 파일은 일반적으로 프로젝트의 [res] -> [drawable] 폴더에 있어야 함.

[이미지 불러오기]
`android:src = "@drawable/그림 아이디(그림 파일 이름)`

[관련 속성]

1. src : 이미지의 경로
2. maxHeight/maxWidth : 이미지의 크기 지정
3. scaleType : 이미지의 확대/축소 방식 지정

[mipmap 폴더] : 여기 있는 앱 아이콘 그림을 화면에 출력하면 **OS가 알아서 현재 안드로이드 폰의 해상도에 적절한 이미지를 알아서 선택함**

CallBack 함수 참고

[callback 함수]

1. **다른 함수의 인자로써 이용되는 함수(고차함수)**
2. 어떤 이벤트에 의해 호출되어지는 함수.(이벤트 리스너)

OS -> 이벤트 인식 -> 콜백 호출

> `setOnCableConnected`로 설정한 함수가 케이블을 연결할 때 마다 호출되므로 -> `onCableConnected`는 어떤 이벤트에 의해 호출되어지는 함수 즉, 콜백 함수 라고 할 수 있다.

안드로이드에서 콜백 의미

1. 정의 : 함수의 파라미터로 들어온 함수를 콜백함수라고 함. 이는 특별한 문법이 있는게 아니고 **호출하는 방식의 측면으로 봐야함**
2. 용도 : **어떤 함수가 호출되고 순차적으로 다음 작업을 실행해야 할 때 사용됨.**

```
fun main(){
    first(::Second) // call by name
}
fun first(second : () -> Unit) : Unit{ // 콜백 함수 리턴 타입 = Unit, first함수 리턴 타입 = Unit
    print("first on\n")
    second() // 콜백 함수 실행.
}
fun second():Unit{ //반환 타입 void
    print("second on")
}
// 실행 순서 main -> first -> second
// 리턴 순서 second -> first -> main
```

chpater 5 - 레이아웃 익히기

레이아웃의 기본 개념

레이아웃은 ViewGroup 클래스로부터 상속받으며 내부에 무엇을 담는 용도로 쓰임(위젯, 레이아웃)
`View` : 컨테이너 역할하는 애들 집합임

레이아웃의 대표적인 속성

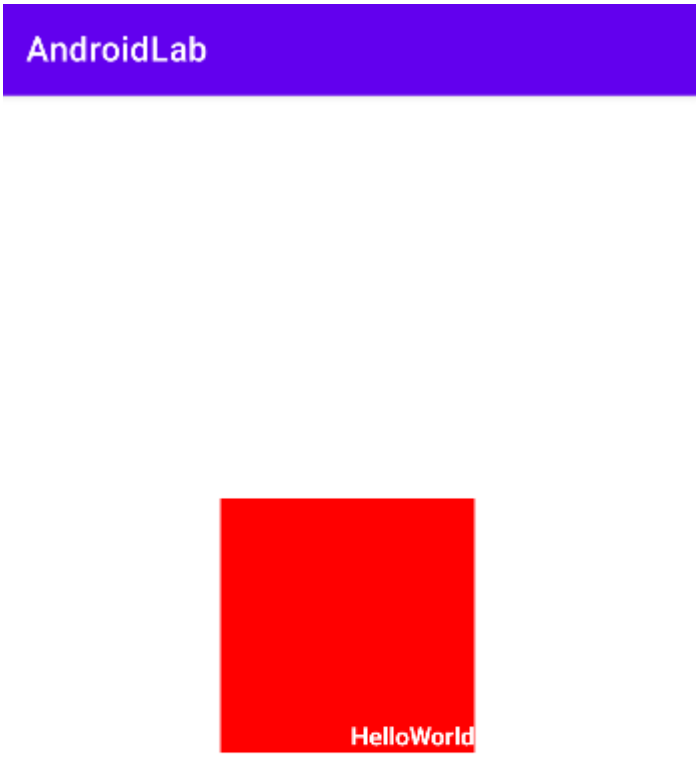
1. `orientation` : 수평, 수직 여부 결정(`vertical` : 수직, `horizontal` : 수평)
2. `gravity` : 위젯의 정렬 방향을 좌픅, 우측, 중앙 등으로 설정
    - 레이아웃 기준 위젯을 정렬함 (**해당 레이아웃에 있는 위젯을 어디에 배치할지 정해지는 것임**)

3. `layout_gravity` : 위젯 자신의 위치를 부모의 어느곳에 위치할지 결정함.(**위젯을 레아아웃 어느 방향에 배치할지 설정**)
4. `padding` : 레이아웃 안에 배치할 위젯의 여백을 설정
5. `layout_weight` : 차지하는 공간의 가중값을 설정 (위젯들이 동일한 비율로 화면에 배치되도록)
6. `baselineAligned` : 위젯을 보기 좋게 정렬함(이때 위젯 테두리 맞춤이 아니라 위젯 내부 컨텐츠에 맞춰 정렬함)

[`gravity` VS `layout_gravity`]

```
<Button
    android:layout_gravity = "center"
    android:gravity = "bottom|right"
    android:background = "#ff0000"
    android:text = "Hello World"
    android:textcolor = "#000000"/>
```

이렇게 배치를 한다면 예상되는 화면은 아래와 같다



출처

* 레이아웃도 View 클래스의 하위 클래스이므로 View클래스의 XML속성과 메서드 모두 사용 가능함.

레이아웃의 종류

---

1. LinearLayout : 그냥 순차적으로 쌓기
2. RelativeLayout : **어떤 대상 기준 어느 방향인지 설정 가능**(다른 위젯으로 부터 상대적인 위치 지정)
3. FrameLayout : 주로 탭 같은거 만들 때 사용
4. TalbeLayout : 테이블 처럼 가능(행 확장은 힘듬)
5. GridLayout : 테이블 처럼 만드는 것이지만 행 확장 등, 확장성 좋음

[한 화면에서 위젯을 **수평과 수직으로 다양하게 배치해야 하는 경우**]

- 리니어레이아웃 내부에 리니어레이아웃 생성하는 방식을 사용함.

```
<LinearLayout>
    <LinearLayout>
        위젯...
    </LinearLayout>
    <LinearLayout>
        위젯...
    </LinearLayout>
    <LinearLayout>
        위젯...
    </LinearLayout>
    <LinearLayout>
        위젯...
    </LinearLayout>
</LinearLayout>
```

[잠깐!]

> match_parent : 자신의 상위 객체에 같은 크기가 됨
> wrap_content : 내용을 담을 크기면 충분함.
> 중복 리니어레이아웃 작성시 주의해서 속성을 부여해야함.
> 만약 layout_weight 속성이 있는 경우 비율이 동일 하기 때문에 match_parent를 하더라도 모든 위젯이 보인다.

[응용 - 빨 노 검 파란색 레이아웃 배치하기]

```
<!--전체 레이아웃-->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"><!--수직으로 설정-->

    <!--빨간색/노란색/검은색 위해 만든 곳-->
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:orientation="horizontal"><!--수평-->

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_weight="1"
            android:background="#FF0000"><!--빨간색-->
        </LinearLayout>

        <!--노란색/검은색 나누기 위함-->
        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent"
```

```
            android:orientation="vertical"
            android:layout_weight="1"><!--수직-->

            <LinearLayout
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:layout_weight="1"
                android:background="#FFFF00"><!--노란색-->
            </LinearLayout>

            <LinearLayout
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:layout_weight="1"
                android:background="#000000"><!--검은색-->
            </LinearLayout>
        </LinearLayout>
    </LinearLayout>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:background="#0000FF"><!--파란색-->
    </LinearLayout>
</LinearLayout>
```

레이아웃 구조 : **전체 레이아웃을 감싸는 것** 1개, **각 색상을 채워넣을 레이아웃** 4개(빨 노 검 파), **빨/노/검 구역 나눌 레이아웃** 1개, **노/검 구역 나눌 레이아웃** 1개

Kotlin 코드로 레이아웃 만들기

지금까지는 `activity_main.xml`에 화면을 구성한 후 `MainActivity.kt`의 `setContentView()` 메서드로 화면을 출력했음.

> `setContentView()`메서드가 호출 되어야 Activity 생성됨 즉, lateinit 임.

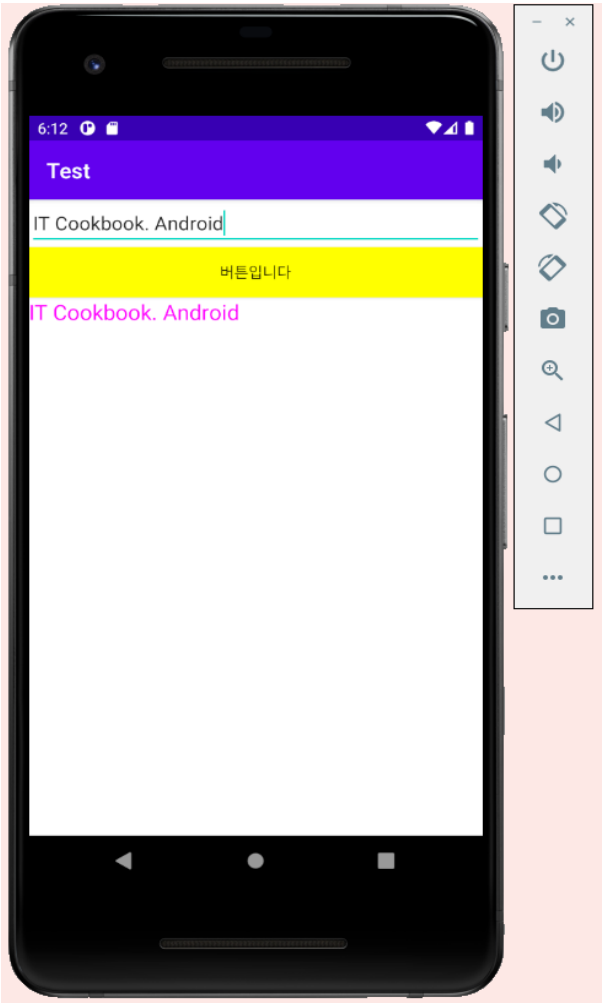
---

* XML 없이 Kotlin을 작성하기 위해서는 기존의 **`setContentView()`**를 주석처리 Or 제거해야한다.

```
// XML없이 리니어레이아웃 생성하기

val params = LinearLayout.LayoutParams(
             LinearLayout.LayoutParmas.MATCH_PARENT,
             LinearLayout.LayoutParmas.MATCH_PARENT)

// 이때 params 변수는 상수로 선언하는 것을 추천함.

val baseLayout = LinearLayout(this) // 리니어 레이아웃 생성
```

```
// 레이아웃 속성 정의
baseLayout.orientation = LinearLayout.VERTICAL // 수직
baseLayout.setBackgroundColor(Color.rgb(0, 255, 0))

setContentView(baseLayout, params) // R.id.activity_main 같은 의미

/*
순서를 보면
레이아웃 속성 크기 정의하고
레이아웃 속성 정의하고
setContentView() 불러온다.
*/
```

```
//버튼 만들기
val btn = Button(this) // 버튼 객체 생성(생각해보면 클래스이름() 형태라 인스턴스 만드
는 느낌임)
btn.text = "버튼입니다" // 버튼 텍스트 속성
btn.setBackgroundColor(Color.MAGENTA)
baseLayout.addView(btn) // 레이아웃에 해당 버튼 위젯 삽입

btn.setOnClickListener { // 이벤트
    Toast.makeText(applicationContext, "코드로 생성한 버튼입니다",
Toast.LENGTH_SHORT).show()
}
```

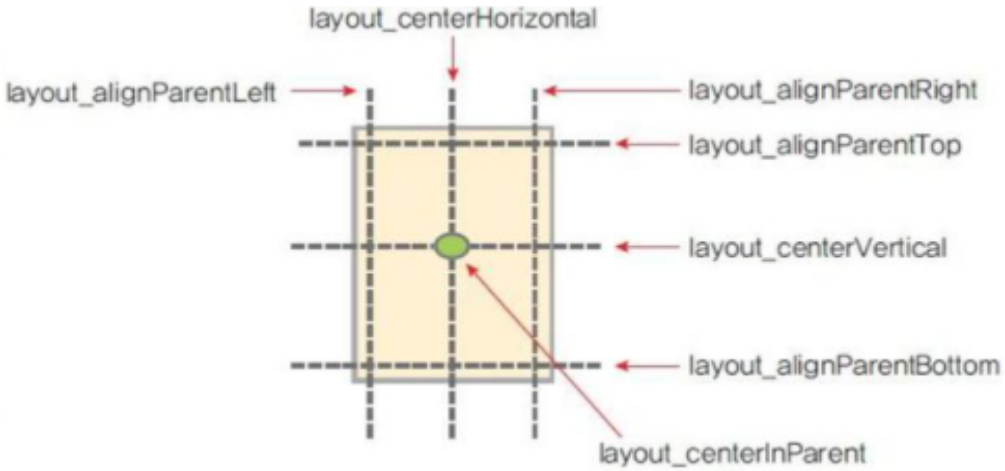


[위 그림 만들기]

```
val params = LinearLayout.LayoutParams(
             LinearLayout.LayoutParmas.MATCH_PARENT,
             LinearLayout.LayoutParmas.MATCH_PARENT)

val baseLayout = LinearLayout(this) // 리니어 레이아웃 생성
// 레이아웃 속성 정의
baseLayout.orientation = LinearLayout.VERTICAL // 수직
baseLayout.setBackgroundColor(Color.rgb(0, 255, 0))

setContentView(baseLayout, params) // R.id.activity_main 같은 의미

val edittext = EditText(this) // Edit Text 객체 생성
baseLayout.addView(edittext) // 레이아웃에 edittetx 삽입
//레이아웃 변수.addView(삽입할 객체)

val btn = Button(this) // 버튼 객체 생성
btn.text = "버튼입니다"
btn.setBackgroundColor(Color.Yellow)
baseLayout.addView(btn) // 레이아웃에 삽입

val textview = TextView(this) // 텍스트뷰 객체 생성
baseLayout.addView(textview) // 레이아웃에 삽입
```

```
btn.setOnClickListener{ //버튼 이벤트 생성
    textview.text = edittext.text.toString() // EditText의 텍스를 바로 문자열로 캐스팅하고 텍스트뷰에 저장.
```

렐러티브레이아웃




```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:text="위쪽" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_centerVertical="true"
        android:text="좌측" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:text="중앙" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_centerVertical="true"
```

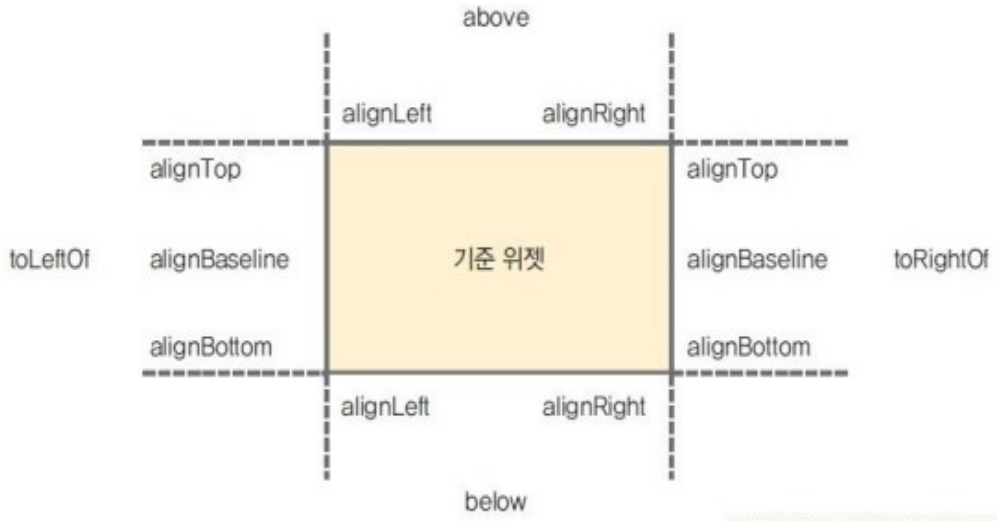
```
        android:text="우측" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_centerHorizontal="true"
        android:text="아래" />

</RelativeLayout>
```

결과 화면

다른 위젯의 상대 위치에 배치

렐러티브 레이아웃 안에서 다른 위젯의 특정한 곳에 배치하는 방법도 있음
다른 위젯의 아이드를 지정하면 됨 **(@+id/기준 위젯 아이디)**

> ex. 버튼을 기준으로 alignTop 위치에 두고싶다 => `android:layout_alignTop = "@+id/basebtn"`



여기서 above, toRightOf, toLeftOf, below는 기준 위젯에서 상하좌우를 의미하며, 나머지 다른 세부 속성은 상하좌우에 대한 구체적인 위치를 의미

예를 들어, 왼쪽 위치의 상단에 두고싶으면

```
<Button
    android:layout_alignTop ="@+id/baseBtn"
    android:layout_toLeftOf = "@+id/baseBtn"/>
    <!-- 이렇게 2개 필요함 -->
```

[응용]

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <Button
        android:id="@+id/baseBtn1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:text="기준1" />

    <Button
        android:id="@+id/baseBtn2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_centerVertical="true"
        android:text="기준2" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
        android:layout_above="@+id/baseBtn2"
        android:layout_toRightOf="@+id/baseBtn1"
        android:text="1번" />
        <!-- 기준 1의 오른쪽, 기준 2의 상단 부분에 위치 -->
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_below="@+id/baseBtn1"
        android:text="2번" />
        <!-- 레이아웃의 오른쪽, 기준1의 아래 부분에 위치 -->
</RelativeLayout>
```

어떤 결과를 보일지 한번 해보길 바란다.

[응용]

```
<RelativeLayout
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_height="match_parent"
    android:layout_width="match_parent"
    android:orientation="horizontal"
    android:padding="10dp">


    <TextView
        android:id="@+id/number"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="전화번호" />

    <EditText
        android:id="@+id/editText1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_toRightOf="@+id/number"
        android:hint="000 - 0000 - 0000"
        android:gravity="center"/>
    <Button
        android:id="@+id/btn1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/editText1"
        android:layout_alignRight="@+id/editText1"
        android:text="취소"/>
        <!-- editText 기준으로 아래 오른쪽 -->
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toLeftOf="@+id/btn1"
        android:layout_margin="30dp"
```

```
        android:layout_alignBaseline="@+id/btn1"
        android:text="확인"/>
        <!-- 취소 버튼 기준으로 왼쪽 같은 라인 -->
</RelativeLayout>
```

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding = '10dp'
    android:orientation="vertical">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">
        <TextView
            android:id="@+id/tv1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text = "전화번호">
        </TextView>
        <EditText
            android:id="@+id/et1"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="000 - 0000 - 0000"
            android:gravity="center">
        </EditText>
    </LinearLayout>
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:layout_gravity="right">
        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="입력"
            android:layout_margin="10dp">
        </Button>
        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="취소"
            android:layout_margin="10dp">
        </Button>
    </LinearLayout>
</LinearLayout>
```

테이블 레이아웃

`TableRow`와 함께 사용됨

행의 수 : `TableRow`의 수 열의 수 : `TableRow`안에 포함됨 위젯의 수

[관련 속성]

1. `layout_span` : 열을 합쳐서 표시하라 의미
2. `layout_colum` : 지정된 열에 현재 위젯 표시
3. `stretchColumns` : `TableLayout`자체에 설정하는 속성임. 지정된 열의 넓이를 늘리라는 의미 값이 *인 경우 모두 같은 크기로 확장됨.

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/tableLayout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TableRow>

        <Button
            android:layout_width="60dp"
            android:text="1" />

        <Button
            android:layout_width="60dp"
            android:layout_span="2"
            android:text="2" />

        <Button
            android:layout_width="60dp"
            android:text="3" />
    </TableRow>

    <TableRow>

        <Button
            android:layout_width="60dp"
            android:layout_column="1"
            android:text="4" />
        <!-- 1열이라고 지정을 했음 -->
        <Button
            android:layout_width="60dp"
            android:text="5" />

        <Button
            android:layout_width="60dp"
            android:text="6" />
    </TableRow>
```

```
</TableLayout>
```