안드로이드 이벤트(Event) 처리

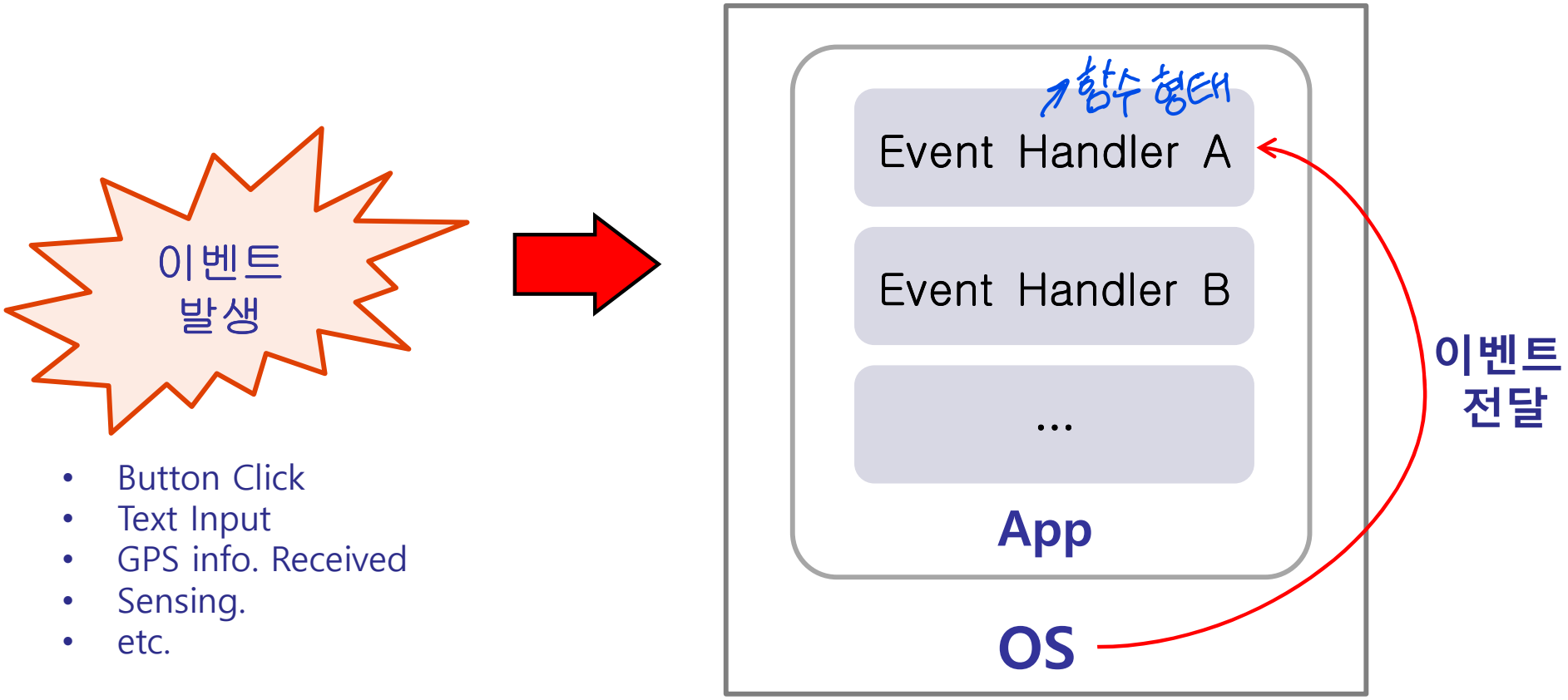
- http://developer.android.com

목차

- Event-Driven Programming
- Android 이벤트 처리 방식
  - 이벤트 처리 메소드 재정의
  - Listener Interface 구현
  - 위젯 이벤트 처리
- 실습

Event-Driven Programming

- 프로그램의 실행 흐름이 사용자 동작과 센서 입출력 등 외부 이벤트에 따라 결정되는 프로그래밍 방식
  - GUI(Graphic User Interface) 프로그램에서 채택
  - 대부분의 프로그래밍 방식



- Button Click
- Text Input
- GPS info. Received
- Sensing.
- etc.

Android 이벤트 처리(이벤트 핸들러 구현) 방식

1. 상속 후 기본 이벤트 처리 메소드 재정의

- 콜백 메소드
- View를 상속받을 경우 기본 이벤트 처리 메소드를 재정의하여 기능 구현

2. Listener Interface 구현 (기본)

- 이벤트 처리 Listener 객체를 구현하여 이벤트 처리가 필요한 View에 등록

3. 위젯 이벤트 처리

- 클릭, 롱클릭과 같이 자주 사용하는 기본 이벤트
- XML의 View 속성에 처리 메소드 명을 등록한 후 구현

이벤트 처리 유형 – 1.이벤트 처리 메소드 재정의

View의 대표적 이벤트 처리 메소드

- override fun onTouchEvent(event: MotionEvent?) : Boolean
- override fun onKeyDown(keycode: Int, event: keyEvent?) : Boolean
- override fun onKeyUp(keycode: Int, event: keyEvent?) : Boolean

고려사항

- View 를 상속하여 Custom View 를 구현할 때만 가능
- 모든 이벤트 종류에 대한 메소드가 정의되어 있지는 않음

Android Studio 기능 활용

- 코드의 View 클래스 내부에서 Ctrl + O → [Select Method to Override/Implement] → 해당 메소드 선택

이벤트 처리 메소드 재정의 예

```
class MyOuterView : View {
    constructor(context: Context?) : super(context)
    constructor(context: Context?, attrs: AttributeSet?) : super(context, attrs)
    constructor(context: Context?, attrs: AttributeSet?, defStyleAttr: Int) : super(
        context,
        attrs,
        defStyleAttr
    )

    override fun onDraw(canvas: Canvas) {
        super.onDraw(canvas)
        canvas?.drawColor(Color.LTGRAY)
        val paint = Paint()
        paint.setColor(Color.BLUE)
        canvas?.drawCircle(200.toFloat(), 200.toFloat(), 100.toFloat(), paint)
    }

    override fun onTouchEvent(event: MotionEvent?): Boolean {
        Toast.makeText(context, text: "Touch!!!", Toast.LENGTH_SHORT).show()
        return false → 처리X 상위로 올라가서 찾아감
    }
}
```

생성자를 추가하고자 할 경우 상위클래스(View)에 커서를 놓은 후 [alt + Ins.] 키 클릭
→ [Secondary Constructor] 선택 후 추가

MyOuterView 를 Touch 할 경우 실행

클래스 내부에서 [CTRL + O] 키를 눌러 추가 가능

true → 처리 O

이벤트 처리 유형 – 2.Listener Interface 구현 1

- 가장 기본적인 UI 이벤트 처리 방법
  - View에 필요한 UI 이벤트 처리 리스너를 구현 후 View에 연결

- 각 UI 이벤트에 대한 Listener Interface 존재
  - 하나의 인터페이스 당 하나의 구현 함수가 존재
    → 코틀린의 SAM(Single Abstract Method) 사용 가능

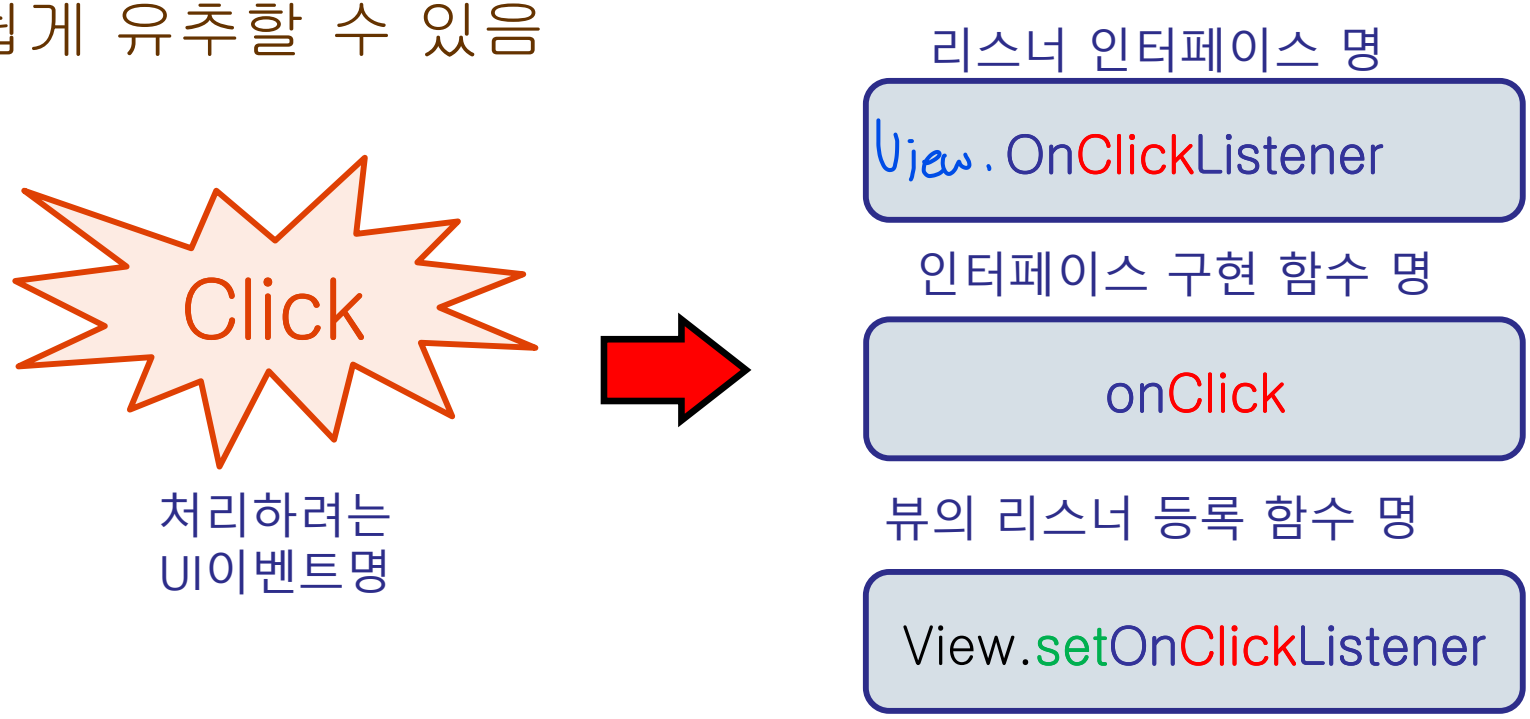
> - *인터페이스명 : 구현 메소드*
> - **View.OnClickListener** : fun onClick(...)
> - **View.OnKeyListener** : fun onKey(...)
> - **View.OnLongClickListener** : fun onLongClick(...)

- 적용 절차
  - 구현하려는 UI 이벤트 처리 *Listener Interface를 구현* → *인터페이스 구현 클래스 작성*
  - *객체 생성*
  - 생성한 객체를 이벤트 처리가 *필요한 View에 등록*

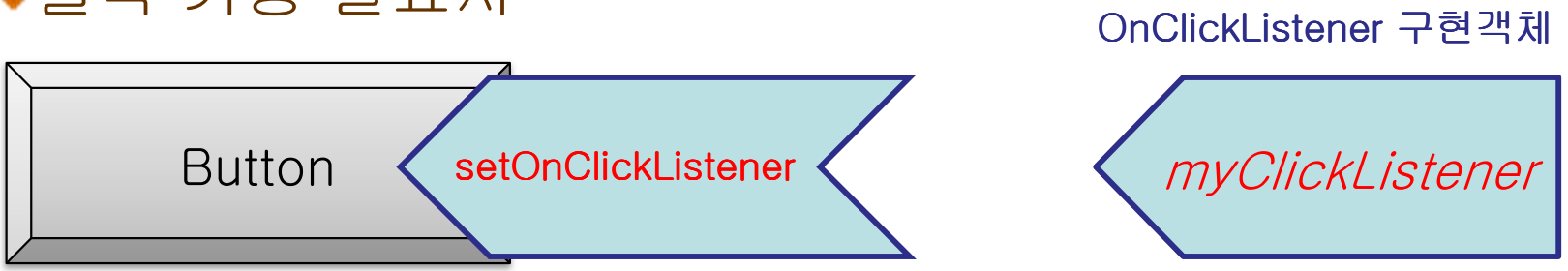
이벤트 처리 유형 – 2.Listener Interface 구현 2

Listener Interface 사용 시의 명칭

- 명칭을 구성하는 규칙이 있으므로 이름을 통해 각 요소를 쉽게 유추할 수 있음



- 클릭 기능 필요시



이벤트 처리 유형 – 2.Listener Interface 구현 3

Listener Interface 구현 유형

A. 리스너 인터페이스 구현 클래스 작성

B. Activity가 리스너 인터페이스 구현

- 커스텀 뷰일 경우 해당 뷰가 리스너 인터페이스 구현

C. 익명 객체(object)로 구현

D. SAM 적용: 인터페이스 선언 없이 함수만 적용

- 사용이 약속되어진 객체가 함수가 하나만 갖고 있을 경우 객체를 생략하고 함수의 본체만 적용

고려사항

- 여러 액티비티에서 재사용 할 필요가 있는가?
- 액티비티 내에서 반복 사용하는가?
- 처리하는 코드 길이는 어느 정도인가?

Listener Interface 구현 예 1

2-A. 리스너 인터페이스 구현 클래스 작성

```
/* 2-A. 리스너 인터페이스 구현클래스 작성 방식*/
class MainActivity : AppCompatActivity() {
    val mainBinding by lazy {
        ActivityMainBinding.inflate(layoutInflater)
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(mainBinding.root)
        val myClick = MyClick( context: this)      // 리스너 인터페이스 구현클래스 객체 생성
        mainBinding.button.setOnClickListener(myClick)  // button 에 이벤트 객체 연결
    }

    /*리스너 인터페이스 구현 클래스*/
    class MyClick (val context: Context) : View.OnClickListener {
        override fun onClick(v: View?) {
            Toast.makeText( context,  text: "리스너 인터페이스 구현 클래스", Toast.LENGTH_SHORT).show()
        }
    }
}
```

이벤트 객체 생성 후 이벤트를 수행할 View 에 연결

클릭에 해당하는 OnClickListener 인터페이스

Listener Interface 구현 예 2

2-B. Activity가 리스너 인터페이스 구현

Acitivity가 리스너 인터페이스 상속

```
/* 2-B. Activity가 리스너 인터페이스 구현*/
class MainActivity : AppCompatActivity(), View.OnClickListener {
    val mainBinding by lazy {
        ActivityMainBinding.inflate(layoutInflater)
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(mainBinding.root)
        mainBinding.button.setOnClickListener(this) // button 에 이벤트 객체 연결
    }

    override fun onClick(v: View?) {
        Toast.makeText( context: this, text: "Activity가 리스너 인터페이스 구현", Toast.LENGTH_SHORT).show()
    }
}
```

Activity 자신이 리스너 인터페이스 구현 객체이므로 this 로 지정

Listener Interface 구현 예 3

2-C. 익명 리스너 인터페이스 객체 구현

```
/* 2-C. 익명 리스너 인터페이스 객체 구현*/
class MainActivity : AppCompatActivity() {
    val mainBinding by lazy {
        ActivityMainBinding.inflate(layoutInflater)
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(mainBinding.root)
        mainBinding.button.setOnClickListener(myClick)  // button 에 이벤트 객체 연결
    }

    val myClick = object: View.OnClickListener {
        override fun onClick(v: View?) {
            Toast.makeText( context: this@MainActivity,
                text: "익명 리스너 인터페이스 구현", Toast.LENGTH_SHORT).show()
        }
    }
}
```

리스너 인터페이스에서
직접 객체 생성

Listener Interface 구현 예 4

2-D. SAM(Single Abstract Method) 적용

- Interface 의 함수가 하나일 경우에만 적용 가능 → Listener Interface 에 적용 가능

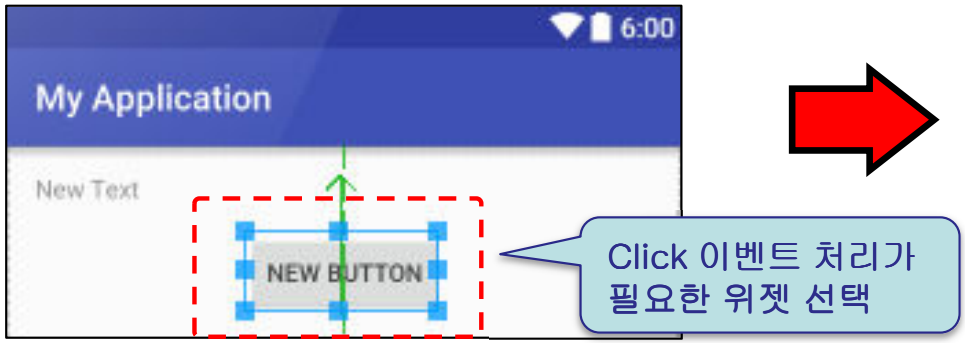
```
/* 2-D. SAM 적용*/
class MainActivity : AppCompatActivity() {
    val mainBinding by lazy {
        ActivityMainBinding.inflate(layoutInflater)
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(mainBinding.root)
        mainBinding.button.setOnClickListener{ it: View!
            Toast.makeText( context: this@MainActivity,
                text: "SAM 적용", Toast.LENGTH_SHORT).show()
        }  // button 에 이벤트 객체 연결
    }
}
```

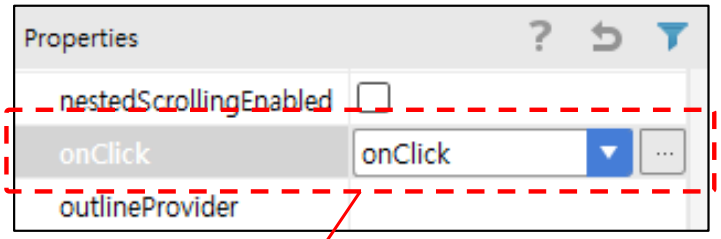
View.setOnClickListener 에는 OnClickLister 가 전달되도록 약속되어 있으므로 객체 및 함수명 생략 가능

이벤트 처리 유형 – 3.위젯 이벤트 처리

View의 XML 속성으로 이벤트 처리 지정

- 이벤트를 처리하는 뷰(위젯)의 XML onClick 속성에 이벤트를 처리할 메소드 명 등록 (long click 도 존재)





해당 위젯이 속한 Activity 멤버 메소드로 추가

메소드 선언 형식 :
fun *메소드명* (view : View) : Unit { … }

```
fun onClick(view : View) {
    // 이벤트 처리 내용
    // 여러 위젯에서 사용할 경우 when 문으로 ID 구분
}
```

이벤트 핸들러의 우선 순위

- 하나의 이벤트에 대하여 여러 이벤트 리스너를 작성할 경우의 우선 순위
  - 구현한 이벤트 리스너가 있을 경우 우선 수행
  - 뷰 자체의 콜백 메소드를 구현하였을 경우 수행 (커스텀 뷰 사용 시)
  - 액티비티에 콜백 메소드를 구현하였을 경우 수행

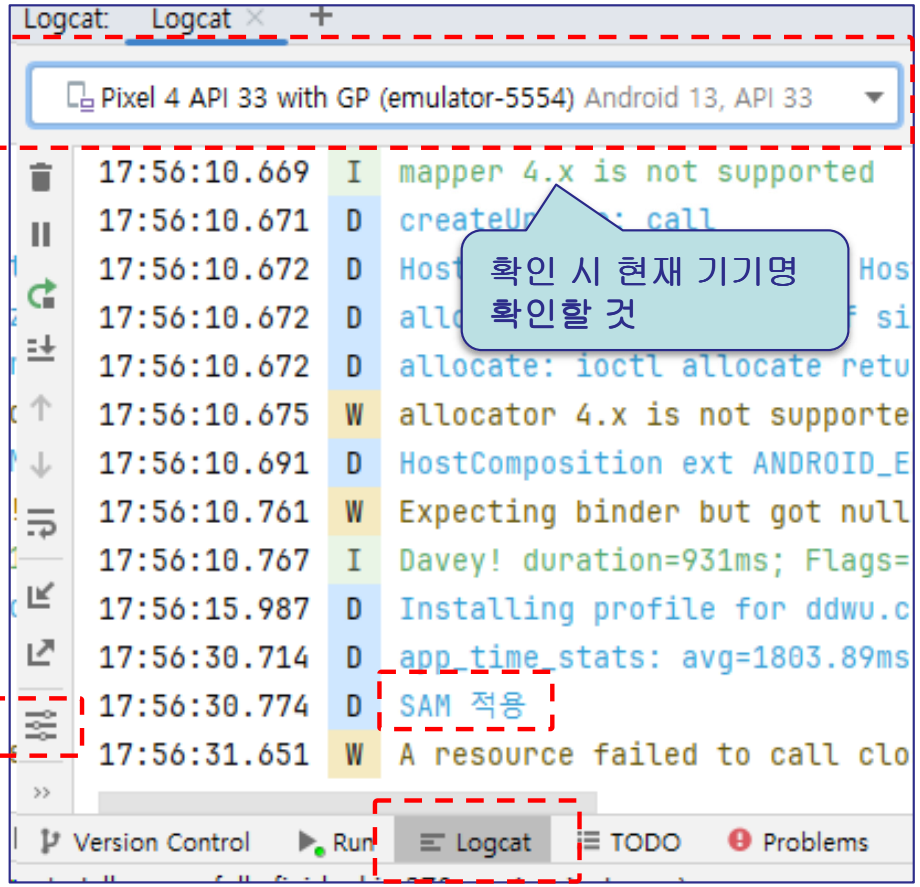
- 이벤트 리스너의 **반환값** 설정
  - true를 반환하면 추가적인 이벤트 처리 진행을 생략
  - false를 반환하면 이벤트 처리를 다음 순위의 리스너에게 이벤트 진행

참고 – Logcat 의 사용

- 각종 디버그 정보 출력 및 확인
- Log.d (*TAG문자열, 출력문자열*) // d, e, w, I, v

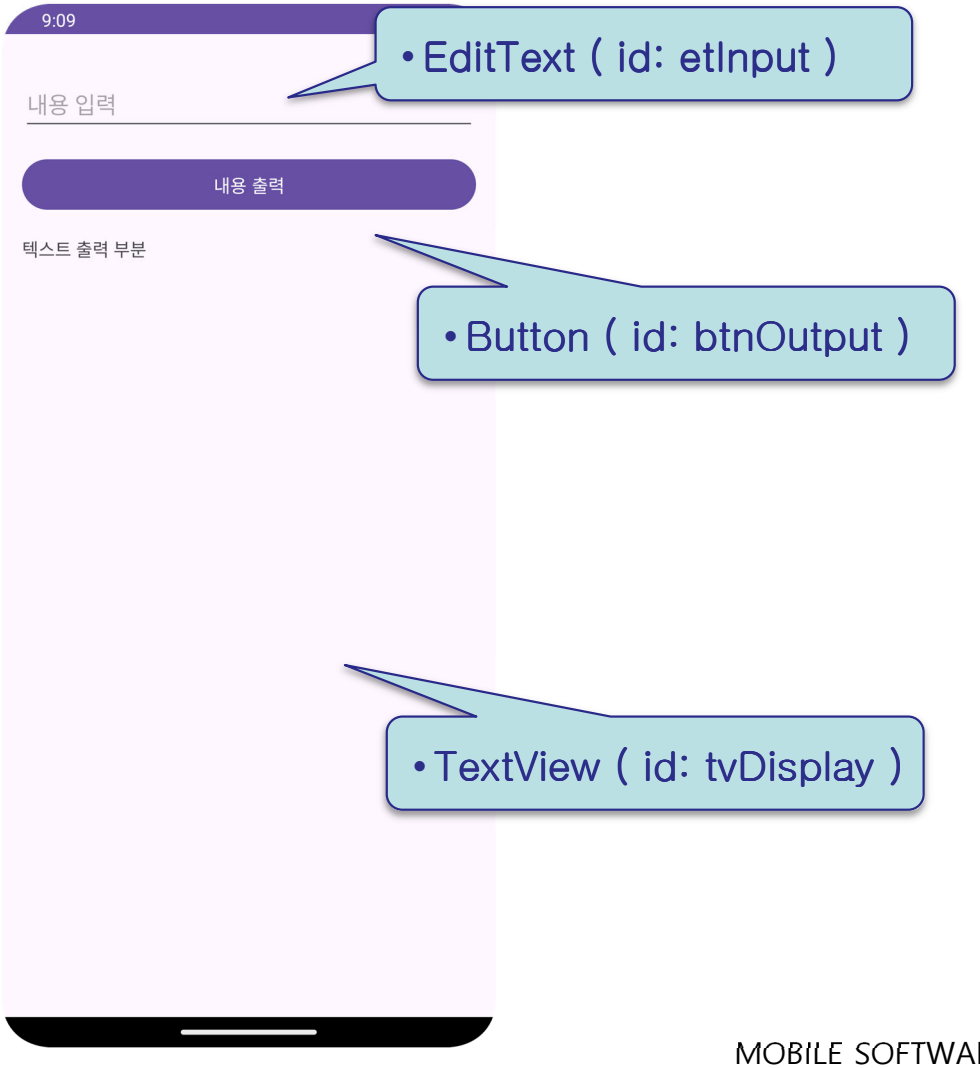
```
const val TAG = "MainActivity" // TAG명

/* 2-D. SAM 적용*/
class MainActivity : AppCompatActivity() {
    val mainBinding by lazy {
        ActivityMainBinding.inflate(layoutInflater)
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(mainBinding.root)
        mainBinding.button.setOnClickListener{ it: View!
            Log.d(TAG, msg: "SAM 적용")
        } // button 에 이벤트 객체 연결
    }
}
```

[Compact View] 선택 시 패키지 정보 생략



실습 01

- 다음과 같은 화면을 구성한 후 각각의 이벤트 처리 방법을 적용하여 앱을 구현하시오.
  - EditText 에 문자열 입력 후 버튼 클릭 시 TextView에 출력



- Exam01.zip 사용
- viewBinding 적용할 것

1. 별도의 리스너 인터페이스 구현 클래스 작성

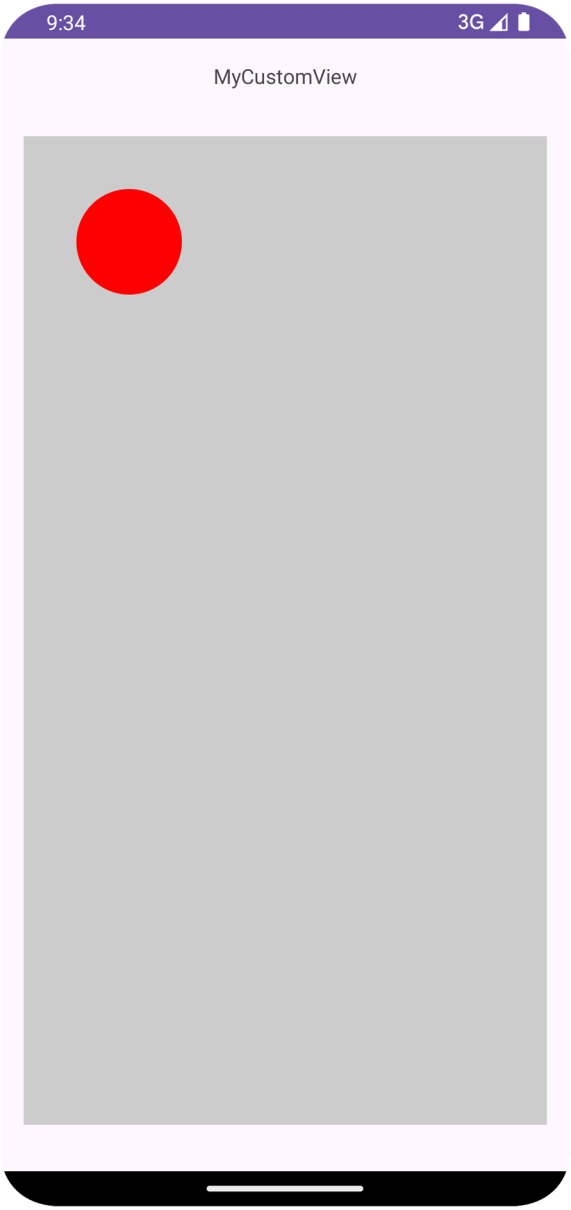
2. 익명 객체 구현으로 작성

3. SAM 적용

※ 각 방법으로 작성한 후 정상적으로 동작하면 주석 처리하여 다음 방법을 적용하여 작성

실습 02

- 커스텀 뷰인 MyCustomView 를 사용하여 각각의 이벤트 처리 방법을 구현하시오.



- Exam02.zip 사용
- viewBinding 적용할 것

1. MyCustomView 클래스의 onTouchEvent(..) 메소드를 재정의하여 터치한 위치에 원 표시

2. MainActivity 에 익명 내부 클래스의 임시객체 구현으로 1번과 동일한 기능 작성
   - OnTouchListener 사용, MyCustomView의 onTouchEvent(...) 는 주석처리 후 구현

3. 롱클릭할 경우 원의 색상이 바뀌는 기능을 익명 내부 클래스의 임시 객체 구현 방법으로 작성
   - OnLongClickListener 구현
   - 앞서 작성한 onTouchListener 의 onTouch(..) 의 반환값은 false 로 변경)
   - 좌표를 바꿨을 때 한번만 실행하도록 구현