04 20 코틀린

▼ Object (Singleton / anonymous class)

→ class 대신 object ClassName() {} 으로 생성

→ 생성자가 private이므로 생성자를 생성하지 않아도 됨

→ 클래스 이름.method , .property로 접근

```
object CarFactory {
    val cars = mutableListOf<Car>()

    fun makeCar(horsepowers: Int): Car {
        val car = Car(horsepowers)
        cars.add(car)
        return car
    }
}
```

```
val car = CarFactory.makeCar(150)
println(CarFactory.cars.size)
```

+) 익명 nested class 생성에도 object 키워드 사용

```
myButton.setOnClickListener(object: View.OnClickListener {
    override fun onClick(v: View) {
        println(v.id)
    }
})
```

▼ 람다함수

함수형 프로그래밍 언어(Kotlin)

→ 모든 것이 객체(feat 함수)

→ 객체는 일급객체 (함수를 - 변수할당/parmeter 삽입/return 가능) - kotlin/javaScript

네, 코틀린에서 함수는 객체입니다. 코틀린 표준 라이브러리는 함수 인자의 개수에 따라 Function0<R> (인자가 없는 함수), Function1<P1, R> (인자가 1개인 함수) 등의 인터페이스를 제공합니다[1].

람다 표현식(익명함수)

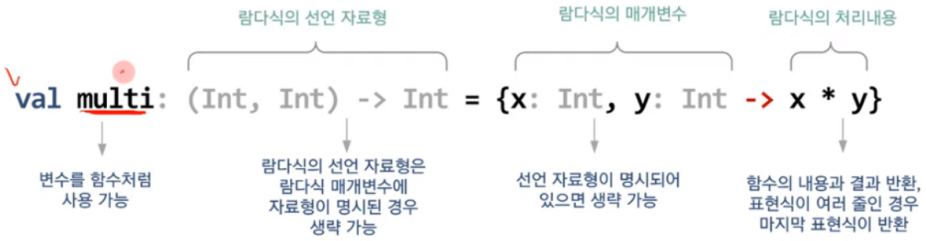
→ 람다함수 == 익명함수

→ **항상 중괄호로 둘러 쌓여 있다.**

→ 마지막 줄은 return을 의미

람다식의 구성

❖ 변수에 지정된 람다식



1. 선언부 or 매개변수 둘중에 1곳에는 type 명시 해야함**(보통 매개변수에 명시)**

```
//생략되지 않은 전체 표현
val multi: (Int, Int) -> Int = {x:Int, y:Imt -> x * y}

// 선언 자료형 생략
val mulit = {x:Int, y:Int -> x * y}

//람다식 매개변수 자료형의 생략
val multi: (Int, Int) -> Int = {x, y -> x * y}

//에러!! 추론 불가
val multi: {X,y -> x *y}
```

▼ **플레이그라운드 예제**

```
// fun String.hi() {println("extended lambda $this")} // 등호없으면 {}
fun String.hi() = println("extended lambda $this") // 등호 있으면 {} 필요없음
//val lambdaFunc = {(num:Int) -> num+num} // ()안에 넣으면 추론이안됨
//val lambdaFunc = {num:Int -> num+num} // 매개변수()안에 넣으면 추론이안됨
// val lambdaFunc(num:Int) {num + num} (변수(val,var)에서는 등호없이 함수선언처럼 안됨)
// fun String.hi() : () -> Unit = {println("extended lambda $this")}
// 불가능 // fun String.hi() : () -> Unit = {() -> println("extended lambda $this")} // 반환형이 없으면 람다에서 -> 쓰지 않는다
// fun String.hi(num:Int) {println("매개변수를 받는데 반환형이 없어요 $num")}
// fun String.hi(num:Int) = println("매개변수를 받는데 반환형이 없어요 $num")
// 불가능 // fun String.hi() = {println("extended lambda $this")}

fun main() {
"확장함수".hi()
println(lambdaFunc(10))

```
val list = listOf(1, 2, 3, 4, 5)

```

//     // 람다식 안에 집어넣기 가능
//     var result = list.fold(0) { sum, element -> sum + element }
//     println(result)

//     result = list.fold(0, { sum, element -> sum + element })
//     println(result)

//     println(result)
}
```

▼ **OnClickListener 예제**

https://toonraon.tistory.com/37

→ 람다식 미적용 (type이 View.OnClickListener인 '익명객체'를 매개변수로 주겠다)

```
// 람다식을 사용하지 않은 코틀린 코드
myButton.setOnClickListener(object: View.OnClickListener {
    override fun onClick(v: View) {
        println(v.id)
    }
})
```

→ JAVA와 비교

```
// 자바의 코드
Button myButton = new Button(this);
myButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        System.out.println(v.getId());
    }
});
```

1차 축약 (SAM이라 onClick 날라감 - View.OnClickListener 인터페이스가 가지는 override 함수가 1개니까 함수명 생략가능하다는 뜻)

```
// 코틀린 1차 축약
myButton.setOnClickListener(View.OnClickListener {
    v: View -> print(v.id)
})
```

2차 축약(setOnClickListenr가 받는 매개변수 type은 View.OnClickListener니까 생략)

```
// 코틀린 2차 축약
myButton.setOnClickListener({
    v: View -> print(v.id)
})
```

3차 축약 (매개변수가 1개? 그럼 생략(it))

```
// 코틀린 3차 축약
myButton.setOnClickListener({
    print(it.id)
})
```

변형(마지막 매개변수가 함수 일 경우 중괄호를 뺄 수 있다.)

```
// 코틀린 4차 축약
myButton.setOnClickListener() {
    print(it.id)
}
```

▼ static? companion!

→ class 밖에서는 companion object 선언 불가

No, companion objects cannot be declared outside a class in Kotlin.

→ 2개 이상 선언 불가

하지만, 클래스 내에는 companion object 를 오직 딱 하나만 선언할 수 있다.
별도의 이름을 부여한다고 해도 같다. 덕분에 static변수들을 한곳에 모이게 할 수 있다.

```
class MyClass5{
    companion object MyCompanion1{
        val prop1 = "나는 Companion object의 속성이다."
        fun method1() = "나는 Companion object의 메소드다."
    }
    companion object MyCompanion2{ // --  에러발생!! Only one companion object is allowed per class
        val prop2 = "나는 Companion object의 속성이다."
        fun method2() = "나는 Companion object의 메소드다."
    }
}
```

▼ BroadCast Receiver (Component)

→ **publish-subscribe design pattern**

→ System or 다른 App 에서 보낸 BroadCast message를 받아서(onReceive) callback(함수 실행)

→ App에서 브로드캐스트 메시지 보내기

intent(context,"브로드캐스트 리시버::class.java")

intent.action("액션명")

sendBroadcast(intent)

→ BroadCast message == intent or event

→ 컴포넌트이지만 예외적으로 동적 리시버는 manifest 등록이 필요없음

▼ 메시지 종류

ACTION_BOOT_COMPLETED

부팅이 끝났을 때 (RECEIVE_BOOT_COMPLETED 권한 등록 필요)

ACTION_CAMERA_BUTTON

카메라 버튼이 눌렸을 때

ACTION_DATE_CHANGED

ACTION_TIME_CHANGED

폰의 날짜, 시간이 수동으로 변했을 때 (설정에서 수정했을 때)

ACTION_SCREEN_OFF

ACTION_SCREEN_ON

화면 on, off

ACTION_AIRPLANE_MODE_CHANGED

비행기 모드

ACTION_BATTERY_CHANGED

ACTION_BATTERY_LOW

ACTION_BATTERY_OKAY

배터리 상태변화

ACTION_PACKAGE_ADDED

ACTION_PACKAGE_CHANGED

ACTION_PACKAGE_DATA_CLEARED

ACTION_PACKAGE_INSTALL

ACTION_PACKAGE_REMOVED

ACTION_PACKAGE_REPLACED

ACTION_PACKAGE_RESTARTED

▼ (정적)리시버는 앱이 동작하지 않아도 이벤트를 받음

Broadcast receivers enable applications to receive intents that are broadcast by the system or by other applications, even when other components of the application are not running [1]. You can declare a broadcast receiver in your manifest file with the `<receiver>` element [1].

So, it is not necessary to use Broadcast Receiver with other components in Android but it can be used to receive intents that are broadcast by the system or by other applications [2] [3].

▼ 동적 리시버 (앱 실행 시에만 인텐트 수신가능 - 생명주기 종속) -

예제

1. manifest에 Reciver객체 이름만 등록 or **등록 안해도 됨**

```
<receiver
    android:name=".broadcastreceiver.TimeCheckBroadcastReceiver"
    android:enabled="true"
    android:exported="true" />
```

2. BroadcaseReceiver()를 상속하는 리시버 객체 생성(onReceive - callback 구현 필수)

```
class TimeCheckBroadcastReceiver : BroadcastReceiver() {

    override fun onReceive(context: Context, intent: Intent) {
        Toast.makeText(context,"Time Check",Toast.LENGTH_SHORT).show()
    }
}
```

3. **registerReceiver(수신할 인텐트 등록)**를 통해 리시버를 사용할 컴포넌트에 연결

```
class MainActivity : AppCompatActivity() {
        ...

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        setBroadcastReceiver()
    }

    private fun setBroadcastReceiver(){
        val intent = IntentFilter(Intent.ACTION_TIME_TICK)
        val receiver = TimeCheckBroadcastReceiver()
        registerReceiver(receiver,intent)
    }
}
```

4. unregisterReceiver를 통해 리시버 사용해제

```
}
private fun cancelBroadcastReceiver(){
    if(receiver != null && isRegistered){
        unregisterReceiver(receiver)
        isRegistered = false
    } else {
        Toast.makeText(this,"Broadcast Receiver have not been registered",Toast.LENGTH_SHORT).show()
    }
}
```

▼ 정적 리시버 (앱 설치 시부터 인텐트 수신가능 - 생명주기 비종속 - 앱꺼져있어도 됨)

- **정적 리시버**
  - AndroidManifest.xml 파일에 등록되며 라이프 사이클과 무관하게 동작
  - 앱이 설치되면 즉시 사용 가능하며 등록과 해지가 자유롭지 못함

예제

1. manifest에 리시버 정보 등록(리시버 class명, 액션)

```
<receiver android:name=".BroadcastSideReceiver">
        <intent-filter>
                <action android:name="com.superdroid.test.Broadcasting.action.FILE_DOWNLOADED" />
    </intent-filter>
</receiver>
```

2. 리시버 class 및 onClick 구현

```java
public class BroadcastSideReceiver extends BroadcastReceiver {
        public void onReceive(Context context, Intent intent) {
        String fileName = intent.getStringExtra("FILE_NAME");
        Toast.makeText(context, "REcevier!!!", Toast.LENGTH_LONG).show();
    }
}
```

▼ 브로드캐스트 == intent **(broadcast == message == intent)**

— 브로드캐스트 == 인텐트 <> 브로드캐스트 리시버는 다른 개념입니다.

명시적 - 이 이벤트를 처리할 대상(패키지==앱)가 명확하다. (단일 앱 수신)

암시적 - 이 이벤트를 처리할 대상(패키지==앱)이 달라질 수 있다. (다중 앱 수신)

명시적 브로드캐스트는 미리 알려진 컴포넌트에 대상 속성을 포함하는 브로드캐스트입니다. 반면, 암시적 브로드캐스트는 대상 속성이 없으며, 앱에 특정하게 타겟팅되지 않은 브로드캐스트입니다. 예를 들어, 수신된 SMS 메시지의 작업은 암시적 브로드캐스트입니다.

API26 이후부터 **일부 액션만 manifest에 등록 가능**

>> 동적 리시버의 사용이 필수가 됐다는 뜻

BroadCastReceiver 오레오 이후 변경된 점

안드로이드개발자홈페이지

앱은 더 이상 명시적 브로드캐스트를 제외한 리시버를 AndroidManifest.xml에 등록할 수 없습니다. 암시적 브로드캐스트 리시버는 런타임에 Context.registerReceiver()를 통해서만 등록이 가능합니다. 단, 서명 권한이 요구되는 브로드캐스트는 암시적 브로드캐스트 제한에서 제외됩니다. 이는 동일한 인증서로 서명된 앱으로만 브로드캐스트가 전송되기 때문입니다.

API 26이전 :

```
<receiver
android:name=".MyBroadcastReceiver">
    <intent-filter>
        <action
android:name="com.example.broadcast.
MY_NOTIFICATION" />
    </intent-filter>
</receiver>
```

```
public class MyBroadcastReceiver
extends BroadcastReceiver {
    @Override
    public void onReceive(Context
context, Intent intent) {
        // Handle the broadcast
event
    }
}
```

API 26 이후

API 레벨 26 이상에서는 암시적 브로드캐스트의 broadcast receiver를 manifest에 더 이상 등록할 수 없습니다. 대신에 Context.registerReceiver() 메서드를 사용하여 런타임에 리시버를 등록해야 합니다. 아래는 예시 코드입니다.

```
MyBroadcastReceiver receiver = new
MyBroadcastReceiver();
IntentFilter filter = new
IntentFilter("com.example.broadcast.
MY_NOTIFICATION");
registerReceiver(receiver, filter);
```

```
public class MyBroadcastReceiver
extends BroadcastReceiver {
    @Override
    public void onReceive(Context
context, Intent intent) {
        // Handle the broadcast
event
    }
}
```

Broadcast란?

Android Application이나 Android System에서 Publish-subscribe pattern처럼 이벤트를 송/수신 하는 컴포넌트입니다.
이벤트는 intent객체에 담겨서 발송되게 됩니다.

Broadcast의 intent 뜯어보기

이벤트는 intent에 문자열로 담겨서 전송되며, 구조는 아래와 같습니다.

```
//android.intent.action.PACKAGE_FULLY_REMOVED이벤트 broadcast의 intent
Intent {
    act=android.intent.action.PACKAGE_FULLY_REMOVED
    dat=package:com.example.broadcasttestc
    flg=0×5000010
    cmp=com.example.broadcasttesta/.MyBroadcastReceiver (has extras)
}
```

그리고 이는 아래처럼 호출하여 문자열로 획득할 수 있습니다.

```
intent.action
intent.data
intent.flags
intent.component
```

https://greensky0026.tistory.com/217