4 w.

Kotlin

Ivy

가시성 변경자 visibility modifier

- 클래스 외부 접근을 제어

변경자	클래스 멤버	최상위 선언
public (기본)	모든 곳에서 접근 가능	모든 곳에서 접근 가능
internal	같은 모듈 안에서만 접근 가능	같은 모듈 안에서만 접근 가능
protected	하위 클래스 안에서만 접근 가능	최상위 선언에 적용 불가
private	같은 클래스 안에서만 접근 가능	같은 파일 안에서만 접근 가능

중첩된 클래스와 가시성

```
class CustomListView {
    val items = mutableListOf<String>()

    fun setViewHolder() {
        val viewHolder = ViewHolder()
    }

    fun getSelectedItem(position: Int) = items[position]

    class ViewHolder {

        fun bind(position: Int) {
            val item = getSelectedItem(position)
        }

    }
}
```

중첩 클래스 (nested class)

- 바깥쪽 클래스 인스턴스에 대한 접근 권한이 없다.

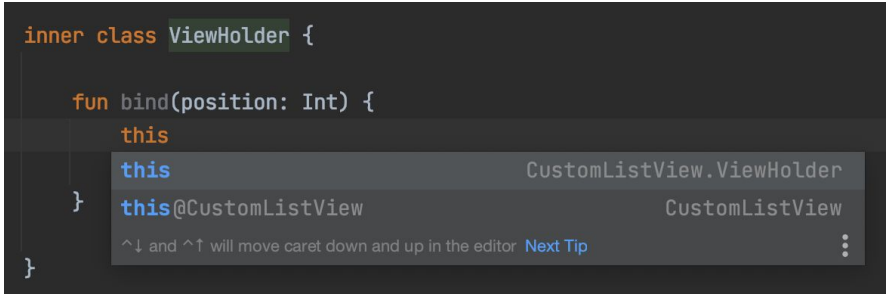
언제 사용할까?

- 도우미 클래스 캡슐화
- 바깥쪽 클래스와 가까이 배치해, 의도를 드러내고 싶을 때

내부 클래스와 가시성

내부 클래스 (inner class)

- 바깥쪽 클래스에 대한 참조를 저장



최상위 함수

- 자바에서는, 모든 코드를 클래스의 메서드로 작성해야 했다.

  → 다양한 정적 메서드를 모아두는 역할만을 담당하는, 특별한 상태나 인스턴스 메서드가 없는 클래스가 생김.

  → 코틀린에서는 이런 클래스가 필요 없다!

```
package com.ivy.memoryassistant.util

fun changeTextStyle() {
    //...
}
```

```
public final void changeStyle() { ButtonExtensionKt.changeTextStyle(); }
```

최상위 프로퍼티

- 프로퍼티도 파일의 최상위 수준에 놓을 수 있다.

```kotlin
val lineSeparator = "\n"

fun changeTextStyle() {
    //...
}
```

```kotlin
const val LINE_SEPARATOR = "\n"
```

```java
@NotNull
public static final String LINE_SEPARATOR = "\n";
```

과제 : Scope 함수, inline 함수 활용

```
public inline fun <R> run(block: () → R): R {
```

```
public inline fun <T, R> T.run(block: T.() → R): R {
```

```
public inline fun <T, R> with(receiver: T, block: T.() → R): R {
```

```
public inline fun <T> T.apply(block: T.() → Unit): T {
```

```
public inline fun <T> T.also(block: (T) → Unit): T {
```

```
public inline fun <T, R> T.let(block: (T) → R): R {
```

```
public inline fun <T> T.takeIf(predicate: (T) → Boolean): T? {
```

```
public inline fun <T> T.takeUnless(predicate: (T) → Boolean): T? {
```

```
public inline fun repeat(times: Int, action: (Int) → Unit) {
```

inline 함수

- 함수 인자 중 1개 이상이 람다여야 inline 키워드를 추가할 수 있다.

```
fun calculateBy(initial: Int, expr:(Int)->Int) = expr(initial)

fun main() {
  println(calculateBy(3) { it * 2 })
}
```

```
public static final int calculate(int initial, @NotNull Function1 expr) {
   Intrinsics.checkNotNullParameter(expr, paramName: "expr");
   return ((Number)expr.invoke(initial)).intValue();
}

public static final void main() {
   int var0 = calculate( initial: 3, (Function1)null.INSTANCE);
   boolean var1 = false;
   System.out.println(var0);
```

inline 함수

```
public static final int calculate(int initial, @NotNull Function1 expr) {
   Intrinsics.checkNotNullParameter(expr,  paramName: "expr");
   return ((Number)expr.invoke(initial)).intValue();
}

public static final void main() {
   int var0 = calculate( initial: 3, (Function1)null.INSTANCE);
   boolean var1 = false;
   System.out.println(var0);
```

```
public static final int calculate(int initial, @NotNull Function1 expr) {
   int $i$f$calculate = 0;
   Intrinsics.checkNotNullParameter(expr,  paramName: "expr");
   return ((Number)expr.invoke(initial)).intValue();
}

public static final void main() {
   int initial$iv = 3;
   int $i$f$calculate = false;
   int var3 = false;
   int var4 = initial$iv * 2;
   $i$f$calculate = false;
   System.out.println(var4);
}
```

inline 함수의 이점

- 함수 본문을 번역한 바이트 코드로 컴파일 한다.
- 메서드를 호출해도 call stack 부하가 없다.

let

```
public inline fun <T, R> T.let(block: (T) → R): R {
```

```
fun refresh() {
    taskId?.let { start(it, forceRefresh: true) }
}
```

- null 이 될 수 있는 식을 다룬다.
- 변수가 필요 없을 때는 '_' 으로 바꾼다.

run - receiver 가 없는 경우

```
public inline fun <R> run(block: () -> R): R {
```

```
fun main() {
  val result = run {
    println("결과를 반환합니다")
    true
  }
}
```

- class 안에서 run을 사용할 경우 인스턴스 receiver(this)와 run의 receiver가 구분되지 않으므로, 구분하기 위해 kotlin.run { } 으로 호출된다.

run - receiver 가 있는 경우, this 참조가 필요한 경우

```
public inline fun <T, R> T.run(block: T.() -> R): R {
```

```
private fun showFilteringPopUpMenu() {
    val view = activity?.findViewById<View>(R.id.menu_filter) ?: return
    PopupMenu(requireContext(), view).run {  this: PopupMenu
        menuInflater.inflate(R.menu.filter_tasks, menu)

        setOnMenuItemClickListener {  it: MenuItem!
            viewModel.setFiltering(
                when (it.itemId) {
                    R.id.active -> TasksFilterType.ACTIVE_TASKS
                    R.id.completed -> TasksFilterType.COMPLETED_TASKS
                    else -> TasksFilterType.ALL_TASKS
                }
            )
            viewModel.loadTasks( forceUpdate: false)
            true  ^setOnMenuItemClickListener
        }
        show()
    }
}
```

with

```
public inline fun <T, R> with(receiver: T, block: T.() -> R): R {
```

```
val str: String? = null
with(str) { // str 이 nullable 이면 this 도 nullable

}
```

```
override fun <T : ViewModel> create(modelClass: Class<T>) =
        with(modelClass) {  this: Class<T>
            when {
                isAssignableFrom(StatisticsViewModel::class.java) ->
                    StatisticsViewModel(
                        GetTasksUseCase(tasksRepository)
                    )  ^with
```

apply, also

```
public inline fun <T> T.apply(block: T.() → Unit): T {
```

```
public inline fun <T> T.also(block: (T) → Unit): T {
```

```
val view = inflater.inflate(R.layout.taskdetail_frag, container, attachToRoot: false)
viewDataBinding = TaskdetailFragBinding.bind(view).apply { this: TaskdetailFragBinding
    viewmodel = viewModel
}
```

- 둘의 차이점은 객체를 참조로 받을지 / 인자로 받을지에 대한 것

takeIf, takeUnless

```
public inline fun <T> T.takeIf(predicate: (T) → Boolean): T? {
```

```
public inline fun <T> T.takeUnless(predicate: (T) → Boolean): T? {
```

```
public inline fun <T> T.takeIf(predicate: (T) → Boolean): T? {
    contract { this: ContractBuilder
        callsInPlace(predicate, InvocationKind.EXACTLY_ONCE)
    }
    return if (predicate(this)) this else null
}
```

```
public inline fun <T> T.takeUnless(predicate: (T) → Boolean): T? {
    contract { this: ContractBuilder
        callsInPlace(predicate, InvocationKind.EXACTLY_ONCE)
    }
    return if (!predicate(this)) this else null
}
```

```
fun showProgress() {
    takeIf { progressBar.isShown } ?: run {
        progressBar.visibility = View.VISIBLE
        progressBar.show()
        hideButtonText()
    }
}

fun hideProgress() {
    takeUnless { progressBar.isShown } ?: run {
        progressBar.visibility = View.GONE
        progressBar.hide()
    }
}
```