KOTLIN 빠른 참조

널 안전성, 코루틴, 데이터 클래스, 함수형 프로그래밍 핵심

기본

Hello World

```
fun main() {
    println("Hello, World!")
}
```

변수

```
val name = "Kotlin"        // immutable (prefer)
var count = 0              // mutable
val pi: Double = 3.14159   // explicit type
const val MAX = 100        // compile-time constant
```

기본 타입

`Int, Long`	32비트 / 64비트 부호 있는 정수
`Double, Float`	64비트 / 32비트 부동소수점
`Boolean`	`true` / `false`
`Char`	단일 유니코드 문자
`String`	불변 텍스트, 템플릿 지원
`Unit`	`void`에 해당 (단일 값)
`Nothing`	절대 반환하지 않는 함수 (예: 예외 던지기)

문자열 템플릿

```
val name = "World"
println("Hello, $name!")
println("Length: ${name.length}")
val raw = """line 1
    |line 2""".trimMargin()
```

함수

함수 선언

```
fun add(a: Int, b: Int): Int {
    return a + b
}
fun add(a: Int, b: Int) = a + b  // single expression
```

기본값 및 이름 있는 인수

```
fun greet(name: String, greeting: String = "Hello") {
    println("$greeting, $name!")
}
greet("Alice")                   // Hello, Alice!
greet("Bob", greeting = "Hi")    // Hi, Bob!
```

고차 함수

```
fun operate(a: Int, b: Int, op: (Int, Int) -> Int): Int {
    return op(a, b)
}
val sum = operate(3, 4) { a, b -> a + b }
```

가변 인수

```
fun sum(vararg nums: Int): Int = nums.sum()
sum(1, 2, 3)
val arr = intArrayOf(1, 2, 3)
sum(*arr)                        // spread operator
```

클래스

클래스 정의

```
class Person(val name: String, var age: Int) {
    fun greet() = "Hi, I'm $name"
}
val p = Person("Alice", 30)
println(p.name)
```

상속

```
open class Shape(val sides: Int) { open fun area(): Double = 0.0 }
class Circle(val r: Double) : Shape(0) {
    override fun area() = Math.PI * r * r
}
```

가시성 제한자

`public`	어디서든 접근 가능 (기본값)
`private`	클래스 / 파일 내부에서만
`protected`	클래스 및 서브클래스
`internal`	같은 모듈 내에서만

추상 클래스 및 인터페이스

```
interface Drawable { fun draw() }
abstract class Widget : Drawable { abstract val label: String }
class Button(override val label: String) : Widget() {
    override fun draw() = println("Drawing $label")
}
```

널 안전성

널 가능 타입

```
var name: String? = null     // nullable
val len = name?.length       // safe call: null
val len2 = name?.length ?: 0 // Elvis operator: 0
val len3 = name!!.length     // assert non-null (throws)
```

안전한 연산

`?.`	안전한 호출 — 수신자가 null이면 null 반환
`?:`	엘비스 — null일 때 기본값
`!!`	null 아님 단언 (null이면 예외 발생)
`?.let { }`	null이 아닐 때만 블록 실행
`as?`	안전한 캐스트 — 실패 시 null 반환

스마트 캐스트

```
if (obj is String) println(obj.length)  // auto-cast
when (obj) {
    is Int    -> println(obj + 1)
    is String -> println(obj.uppercase())
}
```

컬렉션

컬렉션 생성

```
val list = listOf(1, 2, 3)         // immutable
val mList = mutableListOf(1, 2, 3) // mutable
val map = mapOf("a" to 1, "b" to 2)
val set = setOf("x", "y", "z")
```

컬렉션 연산

```
val nums = listOf(1, 2, 3, 4, 5)
nums.filter { it > 2 }             // [3, 4, 5]
nums.map { it * 2 }                // [2, 4, 6, 8, 10]
nums.firstOrNull { it > 3 }        // 4
nums.sumOf { it }                  // 15
```

주요 연산

`.filter { }`	조건을 만족하는 요소만 유지
`.map { }`	각 요소를 변환
`.flatMap { }`	매핑 후 펼침
`.groupBy { }`	키 기준으로 Map에 그룹화
`.sortedBy { }`	셀렉터 기준 정렬
`.associate { }`	Map으로 변환 (키-값 쌍)
`.any { } / .all { }`	하나라도/모두 조건 만족 여부 확인
`.fold(init) { }`	초기 누적기로 리듀스

코루틴

기본 코루틴

```
import kotlinx.coroutines.*
fun main() = runBlocking {
    launch { delay(1000); println("World") }
    println("Hello")
}
```

Async / Await

```
val deferred = async { fetchData() }
val result = deferred.await()
// parallel: launch multiple async, await all
val (a, b) = awaitAll(async { fetchA() }, async { fetchB() })
```

코루틴 빌더

`launch { }`	결과 없는 코루틴 시작 (Job 반환)
`async { }`	결과를 가지는 Deferred<T> 반환
`runBlocking { }`	블로킹 코드와 일시 중단 코드 연결
`withContext(dispatcher)`	코루틴 컨텍스트 전환
`coroutineScope { }`	구조적 동시성 스코프

디스패처

`Dispatchers.Default`	CPU 집약적 작업 (스레드 풀)
`Dispatchers.IO`	블로킹 I/O 작업
`Dispatchers.Main`	메인/UI 스레드 (Android, Swing)
`Dispatchers.Unconfined`	호출 스레드에서 시작, 아무 스레드에서 재개

확장

확장 함수

```
fun String.isPalindrome(): Boolean {
    return this == this.reversed()
}
println("racecar".isPalindrome())  // true
```

확장 프로퍼티

```
val String.wordCount: Int
    get() = this.split("\\s+".toRegex()).size
println("hello world".wordCount)   // 2
```

연산자 오버로딩

```
data class Vec(val x: Double, val y: Double) {
    operator fun plus(other: Vec) = Vec(x + other.x, y + other.y)
}
val v = Vec(1.0, 2.0) + Vec(3.0, 4.0)  // Vec(4.0, 6.0)
```

데이터 클래스

데이터 클래스

```
data class User(val name: String, val age: Int)
val u1 = User("Alice", 30)
val u2 = u1.copy(age = 31)        // non-destructive copy
val (name, age) = u1               // destructuring
```

자동 생성 멤버

`equals()`	프로퍼티 기반 구조적 동등성
`hashCode()`	`equals()`와 일관성 유지
`toString()`	`User(name=Alice, age=30)`
`copy()`	수정된 복사본 생성
`componentN()`	구조 분해 지원

열거형 클래스

```
enum class Direction { NORTH, SOUTH, EAST, WEST }
val dir = Direction.NORTH
when (dir) { Direction.NORTH -> "up"; else -> "other" }
```

봉인 클래스

봉인 클래스 계층

```
sealed class Result<out T> {
    data class Success<T>(val data: T) : Result<T>()
    data class Error(val message: String) : Result<Nothing>()
    data object Loading : Result<Nothing>()
}
```

완전한 when 표현식

```
fun handle(result: Result<String>): String = when (result) {
    is Result.Success -> result.data
    is Result.Error   -> "Error: ${result.message}"
    is Result.Loading -> "Loading..."
}   // no else needed — compiler checks exhaustiveness
```

봉인 클래스 vs 열거형

`Sealed class`	서브클래스가 서로 다른 상태를 가질 수 있음
`Sealed interface`	다중 상속 허용
`Enum class`	고정된 싱글턴 인스턴스 집합
`data object`	`toString()`이 재정의된 싱글턴

스코프 함수

스코프 함수 비교

`let`	컨텍스트를 `it`으로 참조, 람다 결과 반환
`run`	컨텍스트를 `this`로 참조, 람다 결과 반환
`with(obj)`	컨텍스트를 `this`로 참조, 람다 결과 반환
`apply`	컨텍스트를 `this`로 참조, 컨텍스트 객체 반환
`also`	컨텍스트를 `it`으로 참조, 컨텍스트 객체 반환

let & apply

```
val name: String? = "Alice"
name?.let { println("Name is $it") }
val person = Person("Bob", 25).apply {
    age = 26                       // configure object
}
```

run & with

```
val result = "Hello".run { uppercase() + " WORLD" }
val info = with(person) { "$name is $age years old" }
```

also

```
val numbers = mutableListOf(1, 2, 3)
    .also { println("Original: $it") }
    .also { it.add(4) }
// also is useful for side effects (logging, validation)
```