KOTLIN クイックリファレンス

Null 安全性、コルーチン、データクラス、関数型プログラミングの基礎

基本

Hello World

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

変数

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

基本型

`Int, Long`	32 ビット / 64 ビット符号付き整数
`Double, Float`	64 ビット / 32 ビット浮動小数点
`Boolean`	`true` / `false`
`Char`	単一の Unicode 文字
`String`	イミュータブルなテキスト、テンプレート対応
`Unit`	`void` に相当（単一の値）
`Nothing`	絶対に返らない関数（例：throws）

文字列テンプレート

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

関数

関数の宣言

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

デフォルト引数と名前付き引数

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

高階関数

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

可変長引数

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

クラス

クラス定義

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

継承

```
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
}
```

可視性修飾子

`public`	どこからでも見える（デフォルト）
`private`	クラス / ファイル内のみ
`protected`	クラスとサブクラス
`internal`	同一モジュール内のみ

抽象クラスとインターフェース

```
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")
}
```

Null 安全性

Null 許容型

```
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)
```

安全な操作

`?.`	安全呼び出し — レシーバーが null なら null を返す
`?:`	エルビス演算子 — null のときのデフォルト値
`!!`	非 null アサーション（null なら throws）
`?.let { }`	非 null の場合のみブロックを実行
`as?`	安全キャスト — 失敗時は null を返す

スマートキャスト

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

コレクション

コレクションの作成

```
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")
```

コレクション操作

```
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
```

よく使う操作

`.filter { }`	述語にマッチする要素を保持
`.map { }`	各要素を変換
`.flatMap { }`	マップしてフラット化
`.groupBy { }`	キーで Map にグループ化
`.sortedBy { }`	セレクターでソート
`.associate { }`	Map に変換（キーと値のペア）
`.any { } / .all { }`	いずれか / すべてが述語にマッチするか確認
`.fold(init) { }`	初期アキュムレータで集約

コルーチン

基本的なコルーチン

```
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() })
```

コルーチンビルダー

`launch { }`	ファイアアンドフォーゲットのコルーチン（Job を返す）
`async { }`	結果付きの Deferred<T> を返す
`runBlocking { }`	ブロッキングコードとサスペンドコードを橋渡し
`withContext(dispatcher)`	コルーチンコンテキストを切り替え
`coroutineScope { }`	構造化並行性スコープ

ディスパッチャー

`Dispatchers.Default`	CPU 集約的な処理（スレッドプール）
`Dispatchers.IO`	ブロッキング I/O 操作
`Dispatchers.Main`	メイン / UI スレッド（Android、Swing）
`Dispatchers.Unconfined`	呼び出しスレッドで開始し、任意のスレッドで再開

拡張

拡張関数

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

拡張プロパティ

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

演算子オーバーロード

```
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)
```

データクラス

データクラス

```
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
```

自動生成されるメンバー

`equals()`	プロパティに基づく構造的等価性
`hashCode()`	`equals()` と一致
`toString()`	`User(name=Alice, age=30)`
`copy()`	変更済みのコピーを作成
`componentN()`	分割代入のサポート

enum クラス

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

sealed クラス

sealed クラス階層

```
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>()
}
```

網羅的な when

```
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
```

sealed vs enum

`Sealed class`	サブクラスが異なる状態を保持できる
`Sealed interface`	多重継承が可能
`Enum class`	固定されたシングルトンインスタンスのセット
`data object`	`toString()` をオーバーライドするシングルトン

スコープ関数

スコープ関数の比較

`let`	コンテキストは `it`、ラムダ結果を返す
`run`	コンテキストは `this`、ラムダ結果を返す
`with(obj)`	コンテキストは `this`、ラムダ結果を返す
`apply`	コンテキストは `this`、コンテキストオブジェクトを返す
`also`	コンテキストは `it`、コンテキストオブジェクトを返す

let と apply

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

run と 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)
```