KOTLIN RIFERIMENTO RAPIDO

Null safety, coroutine, data class, programmazione funzionale

Basi

Hello World

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

Variabili

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

Tipi di Base

`Int, Long`	Interi con segno a 32/64 bit
`Double, Float`	Floating point a 64/32 bit
`Boolean`	`true` / `false`
`Char`	Singolo carattere Unicode
`String`	Testo immutabile, supporta template
`Unit`	Equivalente a `void` (valore singolo)
`Nothing`	La funzione non ritorna mai (es. lancia eccezione)

Template di Stringa

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

Funzioni

Dichiarazione di Funzione

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

Argomenti Predefiniti e Nominali

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

Funzioni di Ordine Superiore

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

Vararg

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

Classi

Definizione di Classe

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

Ereditarietà

```
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
}
```

Modificatori di Visibilità

`public`	Visibile ovunque (predefinito)
`private`	Visibile dentro la classe / file
`protected`	Classe e sottoclassi
`internal`	Solo nello stesso modulo

Classi Astratte e Interfacce

```
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 Safety

Tipi Nullable

```
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)
```

Operazioni Sicure

`?.`	Safe call — restituisce null se il ricevente è null
`?:`	Elvis — valore predefinito quando null
`!!`	Asserzione non-null (lancia se null)
`?.let { }`	Esegue il blocco solo se non null
`as?`	Cast sicuro — restituisce null in caso di fallimento

Smart Cast

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

Collezioni

Creazione di Collezioni

```
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")
```

Operazioni sulle Collezioni

```
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
```

Operazioni Comuni

`.filter { }`	Mantieni gli elementi che soddisfano il predicato
`.map { }`	Trasforma ogni elemento
`.flatMap { }`	Mappa e appiattisce
`.groupBy { }`	Raggruppa per chiave in una Map
`.sortedBy { }`	Ordina per selettore
`.associate { }`	Trasforma in Map (coppie chiave-valore)
`.any { } / .all { }`	Verifica se qualcuno/tutti soddisfano il predicato
`.fold(init) { }`	Riduce con accumulatore iniziale

Coroutine

Coroutine di Base

```
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() })
```

Builder di Coroutine

`launch { }`	Coroutine fire-and-forget (restituisce Job)
`async { }`	Restituisce Deferred<T> con risultato
`runBlocking { }`	Collega codice bloccante e sospeso
`withContext(dispatcher)`	Cambia il contesto della coroutine
`coroutineScope { }`	Scope con concorrenza strutturata

Dispatcher

`Dispatchers.Default`	Lavoro CPU-intensivo (pool di thread)
`Dispatchers.IO`	Operazioni I/O bloccanti
`Dispatchers.Main`	Thread principale/UI (Android, Swing)
`Dispatchers.Unconfined`	Inizia nel thread del chiamante, riprende in qualsiasi thread

Estensioni

Funzioni di Estensione

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

Proprietà di Estensione

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

Overloading degli Operatori

```
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)
```

Data Class

Data Class

```
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
```

Membri Generati Automaticamente

`equals()`	Uguaglianza strutturale basata sulle proprietà
`hashCode()`	Coerente con `equals()`
`toString()`	`User(name=Alice, age=30)`
`copy()`	Crea una copia modificata
`componentN()`	Supporto alla destrutturazione

Enum Class

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

Sealed Class

Gerarchia Sealed

```
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>()
}
```

When Esaustivo

```
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`	Le sottoclassi possono avere stato diverso
`Sealed interface`	Permette ereditarietà multipla
`Enum class`	Insieme fisso di istanze singleton
`data object`	Singleton con override di `toString()`

Funzioni Scope

Confronto delle Funzioni Scope

`let`	Contesto come `it`, ritorna il risultato della lambda
`run`	Contesto come `this`, ritorna il risultato della lambda
`with(obj)`	Contesto come `this`, ritorna il risultato della lambda
`apply`	Contesto come `this`, ritorna l'oggetto contesto
`also`	Contesto come `it`, ritorna l'oggetto contesto

let e apply

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

run e 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)
```