



Introduzione al linguaggio

Kotlin

A modern programming language
that makes developers happier.



Multiplatform Mobile

Share the logic of your Android and iOS apps while keeping UX native



Server-side

Modern development experience with familiar JVM technology



Multiplatform libraries

Create a library that works across several platforms



Android

Recommended by Google for building Android apps



- Sviluppato da JetBrains a partire dal 2010
- Rivelato con il nome **Project Kotlin** nel luglio 2011
- Requisiti:
 - Linguaggio **conciso, elegante** ed **espressivo**
 - Riduzione della quantità di boilerplate rispetto a Java
 - Totale **interoperabilità** con Java, dato che la maggior parte dei prodotti JetBrains era allora sviluppata in Java, compreso IntelliJ Idea
 - **Velocità di compilazione** alla pari di Java



- Il nome deriva dall'isola russa di Kotlin
- Versione 1.0 rilasciata il 15 febbraio 2016
- Al Google I/O 2017, Google ha annunciato il **supporto** per Kotlin su **Android**
- Nel 2019, Google ha annunciato la scelta di Kotlin come **linguaggio preferito** dalla compagnia per gli sviluppatori di app **Android**

Perché Kotlin per Android?

- **Meno codice e maggiore leggibilità.** Meno tempo impiegato a scrivere codice o a comprendere quello di altri.
- **Linguaggio e ambiente maturi.** Dalla sua creazione nel 2011, Kotlin si è sviluppato continuamente, non solo come linguaggio ma anche come ecosistema con solidi tool. Ora è perfettamente integrato in **Android Studio** e viene utilizzato attivamente da molte aziende per lo sviluppo di applicazioni Android.
- **Supporto Kotlin in Android Jetpack e altre librerie.** Le estensioni KTX aggiungono caratteristiche del linguaggio Kotlin, come le coroutine e le extension functions, alle librerie Android esistenti.
- **Interoperabilità con Java.** È possibile utilizzare Kotlin insieme a Java nelle proprie applicazioni senza dover migrare tutto il codice a Kotlin.

Perché Kotlin per Android?

- **Supporto allo sviluppo multiplatforma.** È possibile utilizzare Kotlin per sviluppare non solo applicazioni Android, ma anche iOS, back-end, web e desktop, condividendo il codice in comune tra le varie piattaforme.
- **Sicurezza del codice.** Meno boilerplate, migliore leggibilità e un compilatore più intelligente portano a un minor numero di errori e ad applicativi più sicuri.
- **Facilità di apprendimento.** Kotlin è facile da imparare, soprattutto per gli sviluppatori Java.
- **Community.** Kotlin gode di un grande supporto e di molti contributi da parte della community, che sta crescendo in tutto il mondo. Secondo Google, già nel 2019 il 60% delle applicazioni nella top 1000 sul Play Store conteneva codice Kotlin.

Chi usa Kotlin



McDonald's leverages Kotlin Multiplatform (KMP) for their global mobile app, enabling them to build a codebase that can be shared across platforms, removing the need for codebase redundancies.

Kotlin Multiplatform



AWS opted for Kotlin over Java for Amazon Quantum Ledger Database (QLDB) thanks to its expressiveness and structured concurrency. They rewrote QLDB in Kotlin, enhancing the user experience, benefiting from its development workflow, and adopting it fully for server-side development.

Server-side



Philips utilizes Kotlin Multiplatform in its HealthSuite digital platform mobile SDK. With KMP, they accelerated the implementation of new features and fostered increased collaboration between Android and iOS developers.

Kotlin Multiplatform



Adobe Experience Platform chose Kotlin for server-side development because of its concise syntax, async capabilities, and interoperability with Java. This shift boosted productivity and improved the developer experience, replacing Java for real-time services.

Server-side



By sharing over 80% of logic across iOS and Android, Forbes now rolls out new features simultaneously on both platforms, keeping the flexibility to withhold or customize features based on the specific platform.

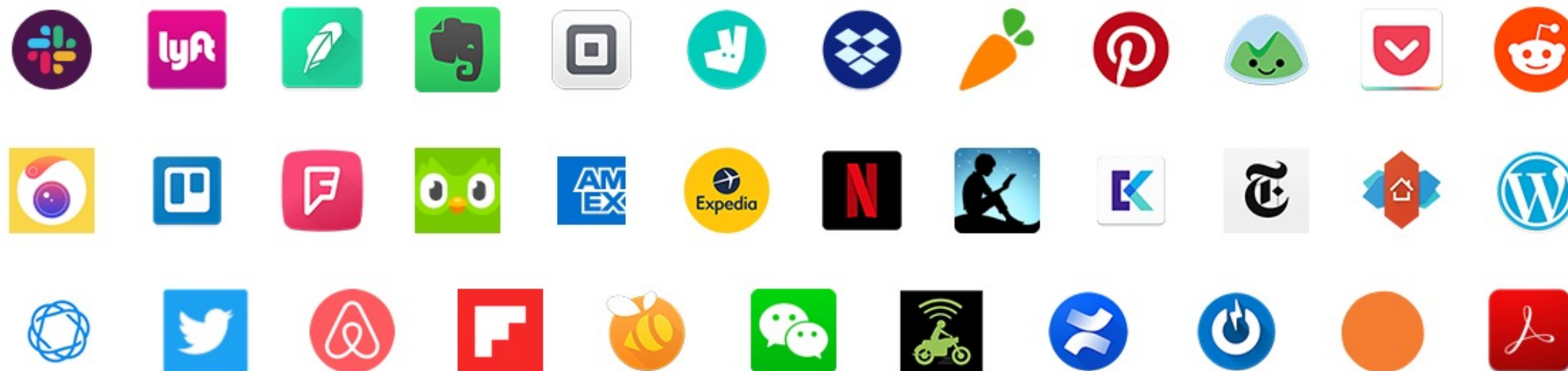
Kotlin Multiplatform



Atlassian adopted Kotlin for the Jira Software cloud, leveraging its Java compatibility for seamless integration and minimizing migration challenges. Kotlin's ease of use improved developer productivity and boosted team satisfaction and efficiency.

Server-side

App create con Kotlin



Linguaggio semplice

```
fun main() {  
    val name = "stranger"    // Declare your first variable  
    println("Hi, $name!")    // ...and use it!  
    print("Current count:")  
    for (i in 0..10) {        // Loop over a range from 0 to 10  
        print(" $i")  
    }  
}
```

```
Hi, stranger!  
Current count: 0 1 2 3 4 5 6 7 8 9 10
```

Asincrono

```
import kotlinx.coroutines.*

suspend fun main() { // A function that can be suspended and resumed later
    val start = System.currentTimeMillis()
    coroutineScope { // Create a scope for starting coroutines
        for (i in 1..10) {
            launch { // Start 10 concurrent tasks
                delay(3000L - i * 300) // Pause their execution
                log(start, "Countdown: $i")
            }
        }
    }
    // Execution continues when all coroutines in the scope have finished
    log(start, "Liftoff!")
}

fun log(start: Long, msg: String) {
    println("$msg " +
        "(on ${Thread.currentThread().name}) " +
        "after ${((System.currentTimeMillis() - start)/1000F}s)")
}
```

Asincrono

```
import kotlinx.coroutines.*

suspend fun main() {
    val start = System.currentTimeMillis()
    coroutineScope {
        for (i in 1..10) {
            launch {
                delay(3000L - i * 300)
                log(start, "Countdown: $i")
            }
        }
    }
    // Execution continues when all co
    log(start, "Liftoff!")
}

fun log(start: Long, msg: String) {
    println("$msg " +
        "(on ${Thread.currentThreadName} " +
        "after ${System.currentTimeMillis() - start}ms)"
    )
}
```

```
Countdown: 10 (on DefaultDispatcher-worker-2 @coroutine#10) after 0.15s
Countdown: 9 (on DefaultDispatcher-worker-2 @coroutine#9) after 0.451s
Countdown: 8 (on DefaultDispatcher-worker-2 @coroutine#8) after 0.75s
Countdown: 7 (on DefaultDispatcher-worker-2 @coroutine#7) after 1.05s
Countdown: 6 (on DefaultDispatcher-worker-2 @coroutine#6) after 1.35s
Countdown: 5 (on DefaultDispatcher-worker-2 @coroutine#5) after 1.65s
Countdown: 4 (on DefaultDispatcher-worker-2 @coroutine#4) after 1.95s
Countdown: 3 (on DefaultDispatcher-worker-2 @coroutine#3) after 2.25s
Countdown: 2 (on DefaultDispatcher-worker-2 @coroutine#2) after 2.543s
Countdown: 1 (on DefaultDispatcher-worker-2 @coroutine#1) after 2.851s
Liftoff! (on DefaultDispatcher-worker-2 @coroutine#1) after 2.851s
```

Asincrono

```
import kotlinx.coroutines.*

suspend fun main() {
    val start = System.currentTimeMillis()
    coroutineScope {
        for (i in 1..10) {
            launch {
                delay(3000L - i * 300L)
                log(start, "Countdown: $i")
            }
        }
    }
    // Execution continues when all coroutines finish
    log(start, "Liftoff!")
}

fun log(start: Long, msg: String) {
    println("$msg " +
        "(on ${Thread.currentThread().name} " +
        "after ${System.currentTimeMillis() - start}ms)")
}
```

Vedremo le coroutine
in Android
nelle prossime lezioni

```
Countdown: 10 (on DefaultDispatcher-worker-2 @coroutine#10) after 0.15s
Countdown: 9 (on DefaultDispatcher-worker-2 @coroutine#9) after 0.451s
Countdown: 8 (on DefaultDispatcher-worker-2 @coroutine#8) after 0.75s
Countdown: 7 (on DefaultDispatcher-worker-2 @coroutine#7) after 1.05s
Countdown: 6 (on DefaultDispatcher-worker-2 @coroutine#6) after 1.35s
Countdown: 5 (on DefaultDispatcher-worker-2 @coroutine#5) after 1.65s
Countdown: 4 (on DefaultDispatcher-worker-2 @coroutine#4) after 1.95s
Countdown: 3 (on DefaultDispatcher-worker-2 @coroutine#3) after 2.25s
Countdown: 2 (on DefaultDispatcher-worker-2 @coroutine#2) after 2.543s
Countdown: 1 (on DefaultDispatcher-worker-2 @coroutine#1) after 2.851s
Liftoff! (on DefaultDispatcher-worker-2 @coroutine#1) after 2.851s
```

Object-oriented

```
abstract class Person(val name: String) {
    abstract fun greet()
}

interface FoodConsumer {
    fun eat()
    fun pay(amount: Int) = println("Delicious! Here's $amount bucks!")
}

class RestaurantCustomer(name: String, val dish: String) : Person(name), FoodConsumer {
    fun order() = println("$dish, please!")
    override fun eat() = println("*Eats $dish*")
    override fun greet() = println("It's me, $name.")
}

fun main() {
    val sam = RestaurantCustomer("Sam", "Mixed salad")
    sam.greet() // An implementation of an abstract function
    sam.order() // A member function
    sam.eat()   // An implementation of an interface function
    sam.pay(10) // A default implementation in an interface
}
```

Object-oriented

```
abstract class Person(val name: String) {
    abstract fun greet()
}

interface FoodConsumer {
    fun eat()
    fun pay(amount: Int) = println("Delicious! Here's $amount bucks!")
}

class RestaurantCustomer(name: String, val dish: String) : Person(name), FoodConsumer {
    fun order() = println("$dish, please!")
    override fun eat() = println("*Eats $dish*")
    override fun greet() = println("It's me, $name.")
}

fun main() {
    val sam = RestaurantCustomer("Sam", "Mixed salad")
    sam.greet() // An implementation of an abstract function
    sam.order() // A member function
    sam.eat() // An implementation of an interface function
    sam.pay(10) // A default implementation in an interface
}
```

```
It's me, Sam.
Mixed salad, please!
*Eats Mixed salad*
Delicious! Here's 10 bucks!
```

Funzionale

```
fun main() {  
    // Who sent the most messages?  
    val frequentSender = messages  
        .groupBy(Message::sender)  
        .maxByOrNull { (_, messages) -> messages.size }  
        ?.key // Get their names  
    println(frequentSender)  
  
    // Who are the senders?  
    val senders = messages  
        .asSequence() // Make operations lazy (for a long call chain)  
        .filter { it.body.isNotBlank() && !it.isRead } // Use lambdas...  
        .map(Message::sender) // ...or member references  
        .distinct()  
        .sorted()  
        .toList() // Convert sequence back to a list to get a result  
    println(senders)  
}  
  
data class Message( // Create a data class  
    val sender: String,  
    val body: String,  
    val isRead: Boolean = false, // Provide a default value for the argument  
)  
  
val messages = listOf( // Create a list  
    Message("Ma", "Hey! Where are you?"),  
    Message("Adam", "Everything going according to plan today?"),  
    Message("Ma", "Please reply. I've lost you!"),  
)
```

Funzionale

```
fun main() {  
    // Who sent the most messages?  
    val frequentSender = messages  
        .groupBy(Message::sender)  
        .maxByOrNull { (_, messages) -> messages.size }  
        ?.key // Get their names  
    println(frequentSender)  
  
    // Who are the senders?  
    val senders = messages  
        .asSequence() // Make operations lazy (for a long call chain)  
        .filter { it.body.isNotBlank() && !it.isRead } // Use lambdas...  
        .map(Message::sender) // ...or member references  
        .distinct()  
        .sorted()  
        .toList() // Convert sequence back to a list to get a result  
    println(senders)  
}  
  
data class Message( // Create a data class  
    val sender: String,  
    val body: String,  
    val isRead: Boolean = false, // Provide a default value for the argument  
)  
  
val messages = listOf( // Create a list  
    Message("Ma", "Hey! Where are you?"),  
    Message("Adam", "Everything going according to plan today?"),  
    Message("Ma", "Please reply. I've lost you!"),  
)
```

Ma
[Adam, Ma]

Ideale per il testing

```
import org.junit.Test // Tests
import kotlin.test.*

class SampleTest {
    @Test
    fun `test sum`() { // Write test names with whitespaces in backticks
        val a = 1
        val b = 41
        assertEquals(42, sum(a, b), "Wrong result for sum($a, $b)")
    }

    @Test
    fun `test computation`() { // Use lambda returning the test subject
        assertTrue("Computation failed") {
            setup()
            compute()
        }
    }
}

fun sum(a: Int, b: Int) = a + b // Source code
fun setup() {}
fun compute() = true
```

- ✓ Passed: test sum
- ✓ Passed: test computation

Ideale per il testing

```
import org.junit.Test
import kotlin.test.*

class SampleTest {
    @Test
    fun `test sum`() {
        val a = 1
        val b = 41
        assertEquals(42, sum(a, b))
    }

    @Test
    fun `test computation`() {
        assertTrue("Computation failed") {
            setup()
            compute()
        }
    }
}

fun sum(a: Int, b: Int) = a + b
fun setup() {}
fun compute() = true
```

// Tests

// Write test names with whitespaces in backticks

turning the test subject

// Sources

Non vedremo il testing nel dettaglio.

Può quindi essere argomento di approfondimento all'esame!

- ✓ Passed: test sum
- ✓ Passed: test computation

Interoperabile

```
// Write Kotlin code, compile it to JavaScript, and run it in the browser
// Use existing JavaScript APIs and libraries
```

```
import kotlinx.browser.window
```

```
fun main() {
    val body = window.document.body
    body?.innerHTML += "<b>Hello, <i>Kotlin</i></b>"
    window.setInterval({
        body?.innerHTML += "!"
    }, 1000)
}
```

```
// Use Kotlin wrappers to build applications with JavaScript frameworks
// Such as React
```

```
import react.*
import react.dom.*
import kotlinx.html.js.onClickFunction

val counter = functionalComponent<Props> {
    val (count, setCount) = useState(0)
    button {
        attrs.onClickFunction = { setCount(count + 1) }
        +count.toString()
    }
}
```

Hello World

- Il codice Kotlin è definito in **package**
- Il punto di ingresso di un'applicazione Kotlin è la funzione **main**, che può essere dichiarata senza alcun parametro. Il tipo di ritorno non è specificato, il che significa che la funzione non restituisce nulla
- La funzione **println** scrive una riga sullo standard output. È importata implicitamente. Inoltre i punti e virgola sono facoltativi

```
package org.kotlinalang.play

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

Variabili

- Kotlin è un linguaggio a **tipizzazione statica**
- È possibile:
 - Dichiarare esplicitamente il tipo delle variabili
 - Lasciare che sia il compilatore a inferire il tipo
- Le variabili immutabili (che non possono essere riassegnate) sono definite con la parola chiave **val**
- Le variabili mutabili (che possono essere riassegnate) utilizzano la parola chiave **var**

```
val a: Int = 1  
val b = 3
```

Costanti

- Per motivi di performance esiste anche la keyword **const**, per la creazione di proprietà immutabili il cui valore è noto a compile-time
- Una proprietà **const** deve soddisfare i seguenti requisiti:
 - Deve essere inizializzata con un valore di tipo **String** o di un tipo primitivo
 - Non può essere un getter personalizzato
 - Deve essere una proprietà top-level, un membro di un object o la dichiarazione un companion object (concetti che introdurremo nelle prossime slide)

Stringhe

- Kotlin offre alcune funzionalità molto comode per la gestione delle stringhe
- Tramite gli **String Templates** è possibile interpolare variabili ed espressioni direttamente all'interno di una stringa
- Utilizzando le triple virgolette, è possibile definire una **Multiline String**

```
val greeting = "Kotliner"
println("Hello $greeting")
println("Hello ${greeting.uppercase()}")
println("""
    Hello
    "$greeting"
    """).trimIndent()
// trimIndent rimuove l'intentazione del codice dalle linee della stringa
```

Funzioni

- Una semplice funzione che accetta un parametro di tipo **String** e restituisce **Unit** (l'equivalente di **void** in Java)

```
fun printMessage(message: String): Unit {  
    println(message)  
}
```


Funzioni

- Una funzione che accetta un **secondo parametro** con un **valore di default**
- Il tipo di ritorno è omesso, perché il compilatore è in grado di inferire che si tratta di **Unit**

```
fun printMessageWithPrefix(message: String, prefix: String = "Info") {  
    println("[$prefix] $message")  
}
```

Funzioni

```
fun printMessageWithPrefix(message: String, prefix: String = "Info") {  
    println("[$prefix] $message")  
}
```

- Come possiamo invocare la funzione?
 1. Serve specificare quale parametro corrisponde a message e quale a prefix?
 2. Serve specificare prefix?
 3. I parametri in input devono essere ordinati?

Funzioni

```
fun printMessageWithPrefix(message: String, prefix: String = "Info") {  
    println("[$prefix] $message")  
}
```

- Come possiamo invocare la funzione?
 1. Serve specificare quale parametro corrisponde a message e quale a prefix?

No

```
printMessageWithPrefix("Hello", "Log")
```

2. Serve specificare prefix?
No, se il valore di default è quello desiderato

```
printMessageWithPrefix("Hello")
```

3. I parametri in input devono essere ordinati?
No, possiamo inserirli in qualunque ordine specificandone il nome

```
printMessageWithPrefix(prefix = "Log", message = "Hello")
```

Funzioni

- Funzione che restituisce un **Int**

```
fun sum(x: Int, y: Int): Int {  
    return x + y  
}
```

Funzioni

- Funzione che restituisce un **Int**

```
fun sum(x: Int, y: Int): Int {  
    return x + y  
}
```

- Può essere semplificata?
 - Il valore di ritorno è obbligatorio?
 - C'è altro che si può omettere?

Funzioni

- Funzione che restituisce un **Int**

```
fun sum(x: Int, y: Int): Int {  
    return x + y  
}
```

- Può essere semplificata?

- Il valore di ritorno è obbligatorio?

No, viene inferito dal compilatore

- C'è altro che si può omettere?

Se il corpo di una funzione è composto da una singola istruzione può essere specificato con = e senza graffe e return

```
fun sum(x: Int, y: Int) = x + y
```

Extension functions

- Kotlin permette di estendere classi o interfacce esistenti tramite le **extension**
- Esempio: **extension function** per calcolare la potenza di un intero (già presente nella standard library di Kotlin)

```
fun Int.pow(exp: Int): Int =  
    if (exp == 0) 1 else this * pow(exp - 1)  
  
fun main() {  
    println(2.pow(3)) // 8  
}
```

Infix functions

- Le funzioni con un solo parametro possono essere trasformate in **infix functions**
- Una infix function può essere richiamata come un operatore
- Esempio: trasformiamo l'extension function della slide precedente in una infix function

```
infix fun Int.pow(exp: Int): Int =  
    if (exp == 0) 1 else this * pow(exp - 1)  
  
fun main() {  
    println(2 pow 3) // 8  
}
```


Operator Functions

- Alcune funzioni possono essere promosse a operatori, tramite un nome predefinito e la keyword **operator**, consentendo di invocarle con il simbolo dell'operatore corrispondente

```
operator fun Int.times(str: String) = str.repeat(this)
println(2 * "Bye ") // Bye Bye

operator fun String.get(range: IntRange) = substring(range)
val str = "Always forgive your enemies; nothing annoys them so much."
println(str[0..14]) // Always forgive
```

<https://kotlinlang.org/docs/operator-overloading.html>

Funzioni con parametri vararg

- Un parametro **vararg** permette di passare un numero qualsiasi di argomenti, separandoli con delle virgole e trattandoli come un array all'interno del corpo della funzione
- Tramite lo spread operator ***** è possibile convertire un array in una lista di parametri vararg

```
fun printAll(vararg messages: String) {  
    for (m in messages) println(m)  
}  
printAll("Hello", "Hallo", "Salut", "Hola", "你好")
```

```
fun log(vararg entries: String) {  
    printAll(*entries)  
}  
log("Hello", "Hallo", "Salut", "Hola", "你好")
```

Null safety

- Kotlin offre supporto sintattico per la gestione dei tipi nullable, in modo da evitare le **NullPointerException** senza dover ricorrere agli **Option**
- Le variabili in Kotlin non consentono l'assegnazione del valore **null**
- A meno che non siano dichiarate come nullable

```
var neverNull: String = "This can't be null"  
var nullable: String? = "You can keep a null here"  
var inferredNonNull = "The compiler assumes non-null"
```

Null safety

- Come vengono gestite le variabili nullable dal compilatore?

Errore a
compile time

```
fun main() {  
    var neverNull: String = "This can't be null"  
    1 neverNull = null  
  
    var nullable: String? = "You can keep a null here"  
    nullable = null  
  
    2 var inferredNonNull = "The compiler assumes non-null"  
    inferredNonNull = null  
  
    fun strLength(notNull: String): Int {  
        return notNull.length  
    }  
    strLength(neverNull)  
    3 strLength(nullable)  
}
```

- 1 ! Null can not be a value of a non-null type String
- 2 ! Null can not be a value of a non-null type String
- 3 ! Type mismatch: inferred type is Nothing? but String was expected

Null safety

- Come si lavora con variabili nullable?

- a) Sfruttando gli smart cast di Kotlin ①

- b) Con gli appositi operatori:

- ?. Safe-call ②
 - !! Non-null assertion
 - ?: Elvis ③

```
fun main() {  
    fun describeString(maybeString: String?) =  
        ① if (maybeString != null && maybeString.length > 0) {  
            "String of length ${maybeString.length}"  
        } else {  
            "Empty or null string"  
        }  
  
    fun describeString2(maybeString: String?) =  
        "String of length ${maybeString?.length ② ?: 0} ③"  
  
    println(describeString(null))           // Empty or null string  
    println(describeString(""))             // Empty or null string  
    println(describeString("hello"))        // String of length 5  
    println(describeString2(null))          // String of length 0  
    println(describeString2(""))            // String of length 0  
    println(describeString2("hello"))       // String of length 5  
}
```

Equality checks

- Kotlin utilizza `==` per il confronto strutturale e `===` per il confronto referenziale

```
val authors = setOf("Shakespeare", "Hemingway", "Twain")
val writers = setOf("Twain", "Shakespeare", "Hemingway")

println(authors == writers)    // true
println(authors === writers)  // false
```

Flussi di controllo

- When
- Loops
- Ranges
- Ternaries

Pattern matching

- Al posto dello switch, Kotlin ha una versione più flessibile e chiara: **when**
- Può essere utilizzato sia come statement che come espressione ed è un potente costrutto per effettuare pattern matching

Statement

```
fun whenStatement(obj: Any) {  
    when (obj) {  
        1 -> println("One")  
        "Hello" -> println("Greeting")  
        is Long -> println("Long")  
        !is String -> println("Not a string")  
        else -> println("Unknown")  
    }  
}
```

Espressione

```
fun whenAssignment(obj: Any): Any {  
    val result = when (obj) {  
        1 -> "one"  
        "Hello" -> 1  
        is Long -> false  
        else -> 42  
    }  
    return result  
}
```


Cicli – for, while, do-while

```
val cakes = listOf("carrot", "cheese", "chocolate")

for (cake in cakes) {
    println("Yummy, it's a $cake cake!")
}
```

```
fun eatACake() = println("Eat a Cake")
fun bakeACake() = println("Bake a Cake")

fun main(args: Array<String>) {
    var cakesEaten = 0
    var cakesBaked = 0
    while (cakesEaten < 5) {
        eatACake()
        cakesEaten ++
    }
    do {
        bakeACake()
        cakesBaked++
    } while (cakesBaked < cakesEaten)
}
```

Iteratori

- È possibile definire iteratori nelle classi implementando l'operatore **iterator**

```
class Animal(val name: String)

class Zoo(val animals: List<Animal>) {
    operator fun iterator(): Iterator<Animal> { // 1
        return animals.iterator() // 2
    }
}

fun main() {
    val zoo = Zoo(listOf(Animal("zebra"), Animal("lion")))

    for (animal in zoo) { // 3
        println("Watch out, it's a ${animal.name}")
    }
}
```

1. Definisce un operatore iterator: deve essere chiamato **iterator** e avere il modificatore **operator**.
2. Restituisce l'iterator, che deve implementare:
 - **next(): Animal**
 - **hasNext(): Boolean**
3. Cicla sugli animali nello zoo con l'iterator definito

Range

- Qual è l'output dei cicli?

1. 0..3
2. 0 until 3
3. 2..8 step 2
4. 3 downTo 0

```
for(i in 0..3) {           // 1
    print(i)
}
print(" ")

for(i in 0 until 3) {      // 2
    print(i)
}
print(" ")

for(i in 2..8 step 2) {    // 3
    print(i)
}
print(" ")

for (i in 3 downTo 0) {    // 4
    print(i)
}
print(" ")
```

Range

- Qual è l'output dei cicli?
 1. 0..3
0123 – equivale a **for(i=0; i<=3; i++)**
 2. 0 until 3
012 – equivale a **for(i=0; i<3; i++)**
 3. 2..8 step 2
2468 – incremento di 2, estremi compresi
 4. 3 downTo 0
3210 – ordine inverso

```
for(i in 0..3) {           // 1
    print(i)
}
print(" ")

for(i in 0 until 3) {      // 2
    print(i)
}
print(" ")

for(i in 2..8 step 2) {    // 3
    print(i)
}
print(" ")

for (i in 3 downTo 0) {    // 4
    print(i)
}
print(" ")
```

Range

- È possibile utilizzare i **range** anche per i **char**

```
for (c in 'a'..'d') {  
    print(c)  
}  
print(" ")  
  
for (c in 'z' downTo 's' step 2) {  
    print(c)  
}  
print(" ")  
  
// Output: abcd zxvt
```

Range

- O per controllare se un valore è compreso in un certo intervallo

```
val x = 2
if (x in 1..5) {
    print("x is in range from 1 to 5")
}
println()

if (x !in 6..10) {
    print("x is not in range from 6 to 10")
}

// Output:
// x is in range from 1 to 5
// x is not in range from 6 to 10
```

Ternaries

- Kotlin non offre alcun operatore ternario
- Ma l'if può essere usato come espressione

```
fun max(a: Int, b: Int) = if (a > b) a else b
```

- Ed è più potente di un operatore ternario (anche se in molti casi è preferibile ricorrere a **when**)

```
fun sign(n: Int) =  
    if (n > 0) "+"  
    else if (n < 0) "-"  
    else "0"
```

```
fun sign(n: Int) = when {  
    n > 0 -> "+"  
    n < 0 -> "-"  
    else -> "0"  
}
```

Classi

- La dichiarazione di una classe è composta dal **nome** della classe, dalla sua **intestazione** (che specifica i parametri generici, il costruttore, ecc...) e dal corpo
- Sia l'intestazione che il corpo sono facoltativi e, se la classe non ha un corpo, le parentesi graffe possono essere omesse.

```
class Customer
class Contact(val id: Int, var email: String)
fun main() {
    val customer = Customer()
    val contact = Contact(1, "mary@gmail.com")
    println(contact.id)
    contact.email = "jane@gmail.com"
}
```

Un costruttore predefinito non parametrizzato viene creato automaticamente da Kotlin.

Dichiara una classe con due proprietà: **id** immutabile e **email** mutabile, e un costruttore con due parametri **id** e **email**

Non c'è la keyword **new** per la creazione di Una nuova istanza della classe

Naming conventions

- Sostanzialmente le stesse convenzioni di Java
- I nomi dei package vanno interamente in minuscolo e senza underscore
- I nomi delle **classi** e degli **object** (che vediamo a breve) usano il **PascalCase**

```
open class DeclarationProcessor { /*...*/ }  
object EmptyDeclarationProcessor : DeclarationProcessor() { /*...*/ }
```

- I nomi di **funzioni**, **proprietà**, **variabili locali** e oggetti mutabili usano il **camelCase**

```
fun processDeclarations() { /*...*/ }  
var declarationCount = 1
```

Naming conventions

- I nomi delle costanti usano lo **SCREAMING_SNAKE_CASE**

```
const val MAX_COUNT = 8
val USER_NAME_FIELD = "UserName"
```

- I nomi delle **proprietà** che contengono riferimenti a oggetti **singleton** possono usare lo stesso stile di denominazione delle dichiarazioni degli oggetti

```
val PersonComparator: Comparator<Person> = /*...*/
```

- Se una classe ha **due proprietà concettualmente uguali**, ma una fa parte di un'API pubblica e l'altra è un dettaglio di implementazione, allora è bene prefissare la proprietà privata, detta anche **backing property**, con un underscore

```
class C {
    private val _elementList = mutableListOf<Element>()

    val elementList: List<Element>
        get() = _elementList
}
```

Ereditarietà

- Kotlin supporta pienamente il tradizionale meccanismo di **ereditarietà** object-oriented di Java

```
open class Dog {                // 1
    open fun sayHello() {        // 2
        println("wow wow!")
    }
}

class Yorkshire : Dog() {        // 3
    override fun sayHello() {    // 4
        println("wif wif!")
    }
}

fun main() {
    val dog: Dog = Yorkshire()
    dog.sayHello()
}
```

1. Le **classi** Kotlin sono **final** di default. Se si desidera consentire l'ereditarietà della classe, contrassegnare la classe con il modificatore **open**
2. Anche i **metodi** Kotlin sono **final** di default. Con il modificatore **open** se ne consente l'override
3. Una classe eredita una superclasse quando si specifica : **SuperclassName()** dopo il suo nome. Le parentesi vuote indicano l'invocazione del costruttore predefinito della superclasse
4. L'override di metodi o attributi richiede il modificatore **override**

Ereditarietà

- È possibile fornire parametri al costruttore della superclasse

```
open class Tiger(val origin: String) {
    fun sayHello() {
        println("A tiger from $origin says: grrhhh!")
    }
}

class SiberianTiger : Tiger("Siberia")

fun main() {
    val tiger: Tiger = SiberianTiger()
    tiger.sayHello()
}
```

- È possibile passare gli argomenti del costruttore alla superclasse

```
open class Lion(val name: String, val origin: String) {
    fun sayHello() {
        println("$name, the lion from $origin says: graoh!")
    }
}

class Asiatic(name: String)
    : Lion(name = name, origin = "India")

fun main() {
    val lion: Lion = Asiatic("Rufo")
    lion.sayHello()
}
```

Special classes

- Data classes
- Enum
- Sealed
- Object keyword

Data classes

- Le **data classes** semplificano la creazione di classi utilizzate per modellare dati
- Una data class implementa automaticamente i metodi:
 - **equals()** e **hashCode()**
 - **toString()**, nella forma "**User(name=John, age=42)**"
 - **componentN()** per il destructuring
 - **copy()**
- Il costruttore di una data class deve avere almeno un parametro
- Tutti i parametri del costruttore devono essere contrassegnati come **val** o **var** e diventano quindi proprietà della classe

```
data class User(val id: Int, val username: String)
```

Data classes

- Il compilatore utilizza solo le proprietà definite all'interno del costruttore per le funzioni generate automaticamente
- Per escludere una proprietà dalle implementazioni generate, dichiararla all'interno del corpo della classe

```
data class Person(val name: String) {  
    var age: Int = 0  
}
```

Solo **name** verrà considerato in **toString()** e negli altri metodi auto-implementati

- È comunque possibile fare override dei metodi auto-implementati per fornire un'implementazione custom

Enum classes

- Sono utilizzate per modellare tipi che rappresentano un **set finito** di valori distinti (es. stati, modalità...)

```
enum class State {  
    IDLE, RUNNING, FINISHED  
}  
  
fun main() {  
    val state = State.RUNNING  
    val message = when (state) {  
        State.IDLE -> "It's idle"  
        State.RUNNING -> "It's running"  
        State.FINISHED -> "It's finished"  
    }  
    println(message)  
}
```

Si accede alla costante tramite il nome della classe

Non serve l'**else** nel **when**, poiché il compilatore sa che stiamo già gestendo tutte le casistiche possibili per l'enum

Come convenzione, sia **SCREAMING_SNAKE_CASE** che **PascalCase** sono accettabili per i valori dell'enum

Enum classes

- Come le altre classi, anche gli enum possono accettare parametri nel costruttore e contenere funzioni

```
enum class Color(val rgb: Int) {  
    RED(0xFF0000),  
    GREEN(0x00FF00),  
    BLUE(0x0000FF),  
    YELLOW(0xFFFF00);  
  
    fun containsRed() = (this.rgb and 0xFF0000 != 0)  
}  
  
fun main() {  
    val red = Color.RED  
    println(red) // RED  
    println(red.containsRed()) // true  
    println(Color.BLUE.containsRed()) // false  
    println(Color.YELLOW.containsRed()) // true  
}
```

Sealed classes

- Una classe **sealed** è considerata **abstract** e può avere sottoclassi solo all'interno dello stesso package in cui è dichiarata

```
sealed class Mammal(val name: String)

class Cat(val catName: String) : Mammal(catName)
class Human(val humanName: String, val job: String) : Mammal(humanName)

fun greetMammal(mammal: Mammal): String {
    when (mammal) {
        is Human -> return "Hello ${mammal.name}; You're working as a ${mammal.job}"
        is Cat -> return "Hello ${mammal.name}"
    }
}

fun main() = println(greetMammal(Cat("Snowy")))
```

Viene eseguito uno smartcast da **Mammal** a **Cat** (o **Human**)

Anche in questo caso l'**else** non è necessario, poiché tutte le possibili sottoclassi sono già gestite. Sarebbe servito con una superclasse non sealed

Properties (introduzione)

- Kotlin offre varie funzionalità per la definizione delle proprietà di una classe
- Alcuni esempi (non esaustivi):

- Proprietà semplice

```
var height: Int = 2
```

- Proprietà read-only con getter

```
val area get() = this.side * this.side
```

Properties (introduzione)

– Proprietà con setter privato

```
class Counter {  
    var count: Int = 0  
    private set  
    fun inc() = count++  
    fun dec() = count--  
}
```

– Proprietà con backing field

```
var rating: Int? = null  
get() {  
    if (field == 5) {  
        println("This is an amazing book!")  
    }  
    return field  
}  
set(value) {  
    if (value != null && value !in 1..5) {  
        throw IllegalArgumentException()  
    }  
    field = value  
}
```

Delegates

- Kotlin supporta il pattern **delegate** a livello di linguaggio tramite la keyword **by**
- Esempio: supponiamo di voler creare una proprietà lazy, che non viene inizializzata alla creazione di un oggetto, ma solo al primo accesso alla proprietà stessa

Delegates

- Potremmo implementarla così:

```
class LazyProperty(val initializer: () -> Int) {  
    var value: Int? = null  
    val lazy: Int  
        get() {  
            if (value == null) value = initializer()  
            return value!!  
        }  
}
```

Delegates

- O potremmo usare un delegate per rendere la funzionalità riutilizzabile:

```
class MyLazy<T>(val initializer: () -> T) {  
    var instance: T? = null  
    operator fun getValue(thisRef: Any?, prop: KProperty<*>): T {  
        if (instance == null) instance = initializer()  
        return instance!!  
    }  
}  
  
class LazyProperty(val initializer: () -> Int) {  
    val lazyValue by MyLazy(initializer)  
}
```

Delegates

- In realtà questa funzionalità è già fornita dalla standard library di Kotlin tramite la funzione **lazy**
- **Nota:** i delegates sono largamente utilizzati nella programmazione Android con Jetpack Compose!

```
var value by remember { mutableStateOf("") }
```

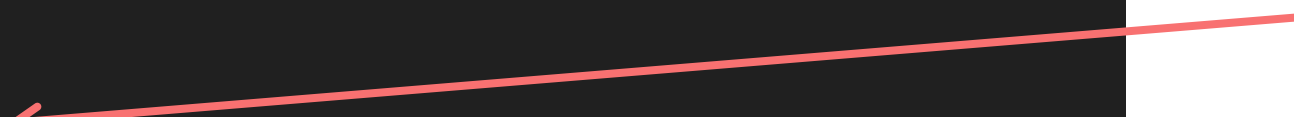

object keyword

- Identifica un tipo di dato con una **singola implementazione**
- Similmente al pattern **singleton**, garantisce che venga creata una sola istanza di una certa classe, anche se più thread tentano di crearla

object expression

```
fun rentPrice(standardDays: Int, festivityDays: Int, specialDays: Int): Unit {  
    val dayRates = object {  
        var standard: Int = 30 * standardDays  
        var festivity: Int = 50 * festivityDays  
        var special: Int = 100 * specialDays  
    }  
  
    val total = dayRates.standard + dayRates.festivity + dayRates.special  
  
    print("Total price: $$total")  
}  
  
fun main() {  
    rentPrice(10, 2, 1)  
}
```

L'object viene creato
a questo punto



object keyword

- Identifica un tipo di dato con una **singola implementazione**
- Similmente al pattern **singleton**, garantisce che venga creata una sola istanza di una certa classe, anche se più thread tentano di crearla

object expression

```
fun rentPrice(standardDays: Int, festivityDays: Int, specialDays: Int): Unit {  
    val dayRates = object {  
        var standard: Int = 30 * standardDays  
        var festivity: Int = 50 * festivityDays  
        var special: Int = 100 * specialDays  
    }  
  
    val total = dayRates.standard + dayRates.festivity + dayRates.special  
  
    print("Total price: $$total")  
}  
  
fun main() {  
    rentPrice(10, 2, 1)  
}
```

object declaration

```
object DoAuth {  
    fun takeParams(username: String, password: String) {  
        println("input Auth parameters = $username:$password")  
    }  
}  
  
fun main(){  
    DoAuth.takeParams("foo", "qwerty")  
}
```

Companion objects

- La dichiarazione di un object all'interno di una classe ha un caso speciale, quello del **companion object**
- Il funzionamento è simile a quello dei **metodi statici** in Java: è possibile richiamare i membri del companion object utilizzando il nome della classe

```
class BigBen {  
    companion object Bonger {  
        fun getBongs(nTimes: Int) {  
            for (i in 1 .. nTimes) {  
                print("BONG ")  
            }  
        }  
    }  
}
```

```
fun main() {  
    BigBen.getBongs(12)  
}
```

Higher-order functions

- Funzione che accetta un'altra **funzione come parametro** e/o **restituisce una funzione**

```
fun calculate(x: Int, y: Int, operation: (Int, Int) -> Int): Int {  
    return operation(x, y)  
}  
  
fun sum(x: Int, y: Int) = x + y  
  
fun main() {  
    val sumResult = calculate(4, 5, ::sum)  
    val mulResult = calculate(4, 5) { a, b -> a * b }  
    println("sumResult $sumResult, mulResult $mulResult")  
}
```

Sintassi per fare riferimento a una funzione per nome

Sintassi lambda

Lambda

- Sintassi alternative per la creazione della stessa funzione **upperCase**

```
val upperCase1: (String) -> String = { str: String -> str.uppercase() }  
val upperCase2: (String) -> String = { str -> str.uppercase() }  
val upperCase3 = { str: String -> str.uppercase() }  
// val upperCase4 = { str -> str.uppercase() }  
val upperCase5: (String) -> String = { it.uppercase() }  
val upperCase6: (String) -> String = String::uppercase  
  
println(upperCase1("hello"))  
println(upperCase2("hello"))  
println(upperCase3("hello"))  
println(upperCase5("hello"))  
println(upperCase6("hello"))
```

Perchè la n° 4
è errata?

Lambda

- Sintassi alternative per la creazione della stessa funzione **upperCase**

```
val upperCase1: (String) -> String = { str: String -> str.uppercase() }  
val upperCase2: (String) -> String = { str -> str.uppercase() }  
val upperCase3 = { str: String -> str.uppercase() }  
// val upperCase4 = { str -> str.uppercase() }  
val upperCase5: (String) -> String = { it.uppercase() }  
val upperCase6: (String) -> String = String::uppercase  
  
println(upperCase1("hello"))  
println(upperCase2("hello"))  
println(upperCase3("hello"))  
println(upperCase5("hello"))  
println(upperCase6("hello"))
```

Perché la n° 4
è errata?

Perché il compilatore
non può conoscere il
tipo del parametro **str**

Lambda

- Sintassi alternative per la creazione della stessa funzione **upperCase**

```
val upperCase1: (String) -> String = { str: String -> str.uppercase() }  
val upperCase2: (String) -> String = { str -> str.uppercase() }  
val upperCase3 = { str: String -> str.uppercase() }  
// val upperCase4 = { str -> str.uppercase() }  
val upperCase5: (String) -> String = { it.uppercase() }  
val upperCase6: (String) -> String = String::uppercase  
  
println(upperCase1("hello"))  
println(upperCase2("hello"))  
println(upperCase3("hello"))  
println(upperCase5("hello"))  
println(upperCase6("hello"))
```

Per le lambda con un singolo parametro, non è necessario nominarlo esplicitamente. Si può invece utilizzare la variabile implicita **it**

Trailing lambda

- Se una lambda è l'ultimo parametro di una funzione, allora è possibile piazzarla fuori dalle parentesi

```
someList.getOrElse(1) {  
    println("Missing item")  
    42  
}
```

return è implicito

- Inoltre, se la lambda è l'unico parametro, è possibile omettere interamente le parentesi

```
someList.filter { it < 0 }
```


Collections

- List
- Set
- Map

List

- Collezione **ordinata** di elementi
 - **List**: read-only
 - **MutableList**: read-write

Perchè l'errore?



```
val systemUsers: MutableList<Int> = mutableListOf(1, 2, 3)
val sudoers: List<Int> = systemUsers

fun addSystemUser(newUser: Int) {
    systemUsers.add(newUser)
}

fun getSysSudoers(): List<Int> {
    return sudoers
}

fun main() {
    addSystemUser(4)
    println("Tot sudoers: ${getSysSudoers().size}")
    getSysSudoers().forEach {
        i -> println("Some useful info on user $i")
    }
    // getSysSudoers().add(5) // <- Error!
}
```

List

- Collezione **ordinata** di elementi
 - **List**: read-only
 - **MutableList**: read-write

Perché l'errore?

Perché
getSysSudoers()
restituisce una **List**
immutabile

```
val systemUsers: MutableList<Int> = mutableListOf(1, 2, 3)
val sudoers: List<Int> = systemUsers

fun addSystemUser(newUser: Int) {
    systemUsers.add(newUser)
}

fun getSysSudoers(): List<Int> {
    return sudoers
}

fun main() {
    addSystemUser(4)
    println("Tot sudoers: ${getSysSudoers().size}")
    getSysSudoers().forEach {
        i -> println("Some useful info on user $i")
    }
    // getSysSudoers().add(5) // <- Error!
}
```

Set

- Collezione **non ordinata** di elementi **senza duplicati**:
 - **Set**: read-only
 - **MutableSet**: read-write

```
val openIssues: MutableSet<String> = mutableSetOf("uniqueDescr1", "uniqueDescr2", "uniqueDescr3")

fun addIssue(uniqueDesc: String): Boolean {
    return openIssues.add(uniqueDesc)
}

fun getStatusLog(isAdded: Boolean): String {
    return if (isAdded) "registered correctly." else "marked as duplicate and rejected."
}

fun main() {
    val aNewIssue: String = "uniqueDescr4"
    val anIssueAlreadyIn: String = "uniqueDescr2"

    println("Issue $aNewIssue ${getStatusLog(addIssue(aNewIssue))}")
    println("Issue $anIssueAlreadyIn ${getStatusLog(addIssue(anIssueAlreadyIn))}")
}
```

Map

- Collezione di coppie chiave-valore
 - Map
 - MutableMap

Output?

```
const val POINTS_X_PASS: Int = 15

val EZPassAccounts: MutableMap<Int, Int> =
    mutableMapOf(1 to 100, 2 to 100, 3 to 100)
val EZPassReport: Map<Int, Int> = EZPassAccounts

fun updatePointsCredit(accountId: Int) {
    if (EZPassAccounts.containsKey(accountId)) {
        println("Updating $accountId...")
        EZPassAccounts[accountId] =
            EZPassAccounts.getValue(accountId) + POINTS_X_PASS
    } else {
        println("Trying to update a non-existing account (id: $accountId)")
    }
}

fun accountsReport() {
    println("EZ-Pass report:")
    EZPassReport.forEach {
        k, v -> println("ID $k: credit $v")
    }
}

fun main() {
    accountsReport()
    updatePointsCredit(1)
    updatePointsCredit(1)
    updatePointsCredit(5)
    accountsReport()
}
```

Map

- Collezione di coppie chiave-valore
 - Map
 - MutableMap

Output?

```
EZ-Pass report:
ID 1: credit 100
ID 2: credit 100
ID 3: credit 100
Updating 1...
Updating 1...
Trying to update a non-existing account (id: 5)
EZ-Pass report:
ID 1: credit 130
ID 2: credit 100
ID 3: credit 100
```

```
const val POINTS_X_PASS: Int = 15
val EZPassAccounts: MutableMap<Int, Int> =
    mutableMapOf(1 to 100, 2 to 100, 3 to 100)
val EZPassReport: Map<Int, Int> = EZPassAccounts

fun updatePointsCredit(accountId: Int) {
    if (EZPassAccounts.containsKey(accountId)) {
        println("Updating $accountId...")
        EZPassAccounts[accountId] =
            EZPassAccounts.getValue(accountId) + POINTS_X_PASS
    } else {
        println("Trying to update a non-existing account (id: $accountId)")
    }
}

fun accountsReport() {
    println("EZ-Pass report:")
    EZPassReport.forEach {
        k, v -> println("ID $k: credit $v")
    }
}

fun main() {
    accountsReport()
    updatePointsCredit(1)
    updatePointsCredit(1)
    updatePointsCredit(5)
    accountsReport()
}
```

Collection methods

- **filter**

```
val negatives = numbers.filter { it < 0 }
```

- **map**

```
val doubled = numbers.map { x -> x * 2 }
```

- **count**

```
val evenCount = numbers.count { it % 2 == 0 }
```

- **getOrElse**: safe access, tramite indice o chiave, all'elemento della collezione

```
val list = listOf(0, 10, 20)  
println(list.getOrElse(1) { 42 })
```

```
val map = mutableMapOf("x" to 3)  
println(map.getOrElse("x") { 1 })
```

Collection methods

- **any, all, none**: controllano la presenza di elementi nella collection
- **find, findLast**: trovano un certo elemento nella collection
- **first, last**: restituiscono il primo/ultimo elemento
- **sorted**: crea una copia ordinata della collezione
- E altri...

Sequences

- Similmente agli Stream di Java, consentono di fare operazioni in maniera lazy su una collezione

```
fun findMostExpensiveProductBy(customer: Customer): Product? =  
    customer  
        .orders  
        .asSequence()  
        .filter { it.isDelivered }  
        .flatMap { it.products }  
        .maxByOrNull { it.price }
```

Scope functions

- Funzioni con in grado di eseguire un blocco di codice nel contesto di un oggetto.
 - **let**
 - **run**
 - **with**
 - **apply**
 - **also**

let

- La funzione **let** può essere utilizzata per scoping e null-checks
- Quando viene chiamata su un oggetto, esegue il blocco di codice dato e restituisce il risultato della sua ultima espressione
- L'oggetto è accessibile all'interno del blocco tramite **it** (o un nome personalizzato passato alla lambda)

```
fun printNonNull(str: String?) {  
    println("Printing \"$str\":")  
  
    str?.let {  
        print("\t")  
        customPrint(it)  
        println()  
    }  
}  
  
printNonNull(null)  
printNonNull("my string")
```

```
Printing "null":  
Printing "my string":  
    MY STRING
```

run

- Il funzionamento è analogo a quello di **let**, ma in **run** si accede all'oggetto tramite **this** (implicito). Questo è utile quando si vogliono chiamare i metodi dell'oggetto, invece di passarlo come parametro

```
fun getNullableLength(ns: String?) {  
    println("for \"$ns\":")  
    ns?.run {  
        println("\tis empty? " + isEmpty())  
        println("\tlength = $length")  
        length  
    }  
}  
getNullableLength(null)  
getNullableLength("")  
getNullableLength("some string with Kotlin")
```

```
for "null":  
for "":  
    is empty? true  
    length = 0  
for "some string with Kotlin":  
    is empty? false  
    length = 23
```

with

- A differenza di **let** e **run**, **with** non è una extension function
- Può accedere ai membri del suo parametro in modo conciso, omettendo il nome dell'istanza

```
class Configuration(var host: String, var port: Int)

fun main() {
    val configuration = Configuration(host = "127.0.0.1", port = 9000)

    with(configuration) { println("$host:$port") }

    // instead of:
    println("${configuration.host}:${configuration.port}")
}
```

```
127.0.0.1:9000
127.0.0.1:9000
```

apply

- Esegue un blocco di codice su un oggetto e restituisce l'oggetto stesso
- All'interno del blocco, l'oggetto è referenziato con **this** (implicito)
- Utile in fase di inizializzazione di un oggetto

```
data class Person(var name: String, var age: Int = 0, var about: String = "")  
  
fun main() {  
    val jake = Person("Jake")  
    val stringDescription = jake.apply {  
        age = 30  
        about = "Android developer"  
    }.toString()  
    println(stringDescription)  
}
```

Person(name=Jake, age=30, about=Android developer)

also

- Funziona come **apply**, ma all'interno della funzione l'oggetto è referenziato con **it**, quindi è più facile passarlo come parametro
- Utile per **incorporare azioni aggiuntive**, come il logging, all'interno di una catena di chiamate a funzione

```
data class Person(var name: String, var age: Int = 0, var about: String = "")

fun writeCreationLog(p: Person) {
    println("A new person ${p.name} was created.")
}

fun main() {
    val jake = Person("Jake", 30, "Android developer")
        .also {
            writeCreationLog(it)
        }
}
```

A new person Jake was created.

Error handling

- I meccanismi di gestione degli errori offerti da Kotlin sono essenzialmente gli stessi di Java
- Con un'importante differenza: **Kotlin non supporta le checked exceptions** (clausola **throws** nella signature dei metodi Java)

Generici

- I generici sono un meccanismo diventato standard nei linguaggi moderni
- Le classi e le funzioni generiche aumentano la riusabilità del codice incapsulando la logica comune e indipendente da un particolare tipo generico

Generic class

```
class MutableStack<E>(vararg items: E) {  
    private val elements = items.toMutableList()  
  
    fun push(element: E) = elements.add(element)  
    fun peek(): E = elements.last()  
    fun pop(): E = elements.removeAt(elements.size - 1)  
    fun isEmpty() = elements.isEmpty()  
    fun size() = elements.size  
  
    override fun toString() =  
        "MutableStack(${elements.joinToString()})"  
}
```

Generici

- I generici sono un meccanismo diventato standard nei linguaggi moderni
- Le classi e le funzioni generiche aumentano la riusabilità del codice incapsulando la logica comune e indipendente da un particolare tipo generico

Generic class

```
class MutableStack<E>(vararg items: E) {  
    private val elements = items.toMutableList()  
  
    fun push(element: E) = elements.add(element)  
    fun peek(): E = elements.last()  
    fun pop(): E = elements.removeAt(elements.size - 1)  
    fun isEmpty() = elements.isEmpty()  
    fun size() = elements.size  
  
    override fun toString() =  
        "MutableStack(${elements.joinToString()})"  
}
```

Generic function

```
fun <E> mutableStackOf(vararg elements: E) =  
    MutableStack(*elements)  
  
fun main() {  
    val stack = mutableStackOf(0.62, 3.14, 2.7)  
    println(stack)  
}
```

Problemi di Java che Kotlin resolve

- Null references
- Raw types
- Non-invariant arrays
- Lack of proper function types
- Use-site variance without wildcards
- Checked exceptions

Feature di Java che Kotlin non fornisce

- Checked exceptions
- Primitive types
- Static class members
- Wildcard-types
- Ternaries

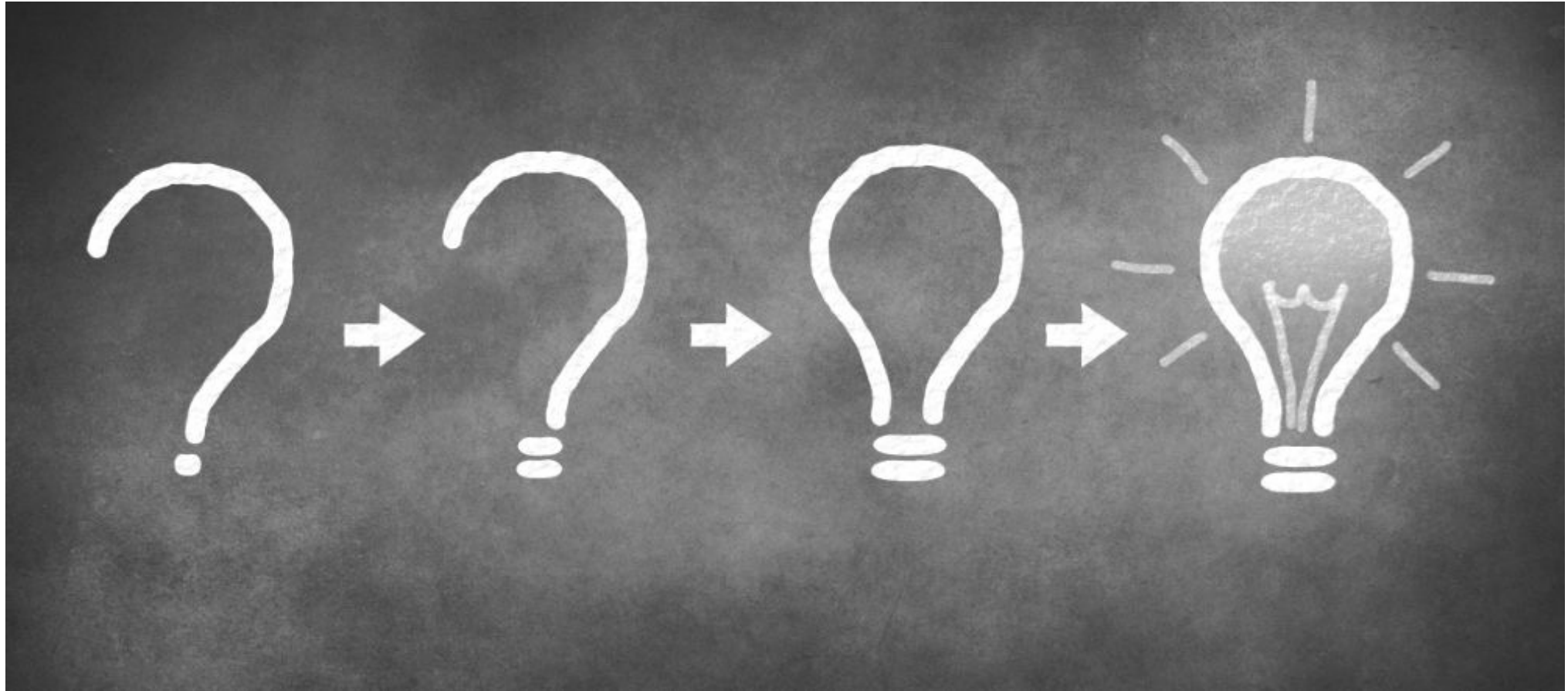
Feature aggiuntive di Kotlin rispetto a Java

- Performant custom control structures with lambda expressions and inline functions
- Extension functions
- Null-safety
- Smart casts
- String templates
- Properties
- Primary constructors
- First-class delegation
- Type inference for variable and property types
- Singletons
- Declaration-site variance and type projections
- Range expressions
- Operator overloading
- Companion objects
- Data classes
- Separate interfaces for read-only and mutable collections
- Coroutines

Learn Kotlin

<https://play.kotlinlang.org/koans/overview>

Domande?



Riferimenti

- Kotlin:
 - <https://kotlinlang.org/>
 - <https://play.kotlinlang.org/byExample/overview>
- Documentazione Android Developers
 - <https://developer.android.com/>