CORSO KOTLIN

Sommario

INTRODUZIONE

Kotlin è un linguaggio di programmazione ufficialmente supportato per la programmazione android. Questo linguaggio funziona dovunque funzioni JAVA, dal momento che viene eseguito sulla JVM, che è la stessa macchina su cui viene eseguito anche il codice JAVA.

Kotlin è un linguaggio orientato agli oggetti, ma allo stesso tempo utilizza la programmazione funzionale. Nella programmazione funzionale si possono utilizzare funzioni come variabili, posso salvarle all'interno delle variabili, posso ritornarle da altre funzioni e posso passarle come parametri in altre funzioni.

In Kotlin si possono dichiarare degli elementi come immutabili, ovvero che non possono essere modificati.

Tutto ciò consente a Kotlin di essere un linguaggio molto coinciso e permette di scrivere molto meno codice, rispetto a quello che avremmo scritto in JAVA.

Infine Kotlin ha la capacità di poter essere eseguito e di essere richiamato da/su classi JAVA.

VARIABILI

Le variabili sono i contenitori in cui andiamo a mettere i diversi tipi di dati. La keyword var è quella che ci permette di creare una variabile. Tale keyword a sua volta chiede altre informazioni per creare la variabile, ossia il nome, il tipo e il valore. Vediamo qualche esempio:

```
var numeroDiPersone: Int = 2000
var numeroDiPersone2: Byte =20
var numeroDiPersone3: Short = 200
var numeroDiPersone4:  Long = 2383848484339L

var PI: Double = 3.121537327468
var PI2: Float = 3.23629329f

var nome: String = "Ciao"

var carattere: Char = 'a'

var booleano: Boolean = true
```

In Kotlin i tipi int, short, float, char e così via, non sono considerati tipi primitivi come in JAVA, ma sono considerati comunque come oggetti.

Detto ciò, non è obbligatorio inserire il tipo della variabile, ma c'è un modo per farlo intuire automaticamente da Kotlin. Basta togliere i due punti con il tipo a seguire e, comunque, il codice continuerà a funzionare. Questo perché Kotlin riesce a dedurre il tipo della variabile semplicemente dal valore che gli andiamo ad assegnare:

```
var numeroDiPersone = 2000
var numeroDiPersone2 =20
var numeroDiPersone3 = 200
var numeroDiPersone4 = 2383848484339L

var PI = 3.121537327468
var PI2 = 3.23629329f

var nome = "Ciao"

var carattere = 'a'
var booleano = true
```

RIASSEGNAZIONE VARIABILI

Partiamo con una semplice assegnazione di un valore ad una variabile:

```
var moneta = 1000
```

Vogliamo in seguito cambiare quel valore dentro la variabile money. Il modo giusto per farlo è semplicemente riscrivendo solo il nome della variabile, assegnandole il nuovo valore, senza utilizzare la keyword var, perché non abbiamo bisogno di crearne una nuova dato che ce l'abbiamo già:

```
moneta = 800
```

Possiamo però assegnare ad una nuova variabile un'altra variabile, come segue:

```
var accountBanca = moneta
```

Se stampiamo il valore di accountBanca, vediamo a schermo l'ultimo valore aggiunto a moneta:

```
print ("Totale soldi nel suo conto: " + accountBanca) //Totale soldi nel suo conto: 800
```

OPERAZIONI CON VARIABILI

Vediamo un esempio facendo il calcolo della media:

```
fun main() {
    //Operazione di media: (8 + 9 + 4 + 6) / 4
    var voto1 = 8
    var voto2 = 9
    var voto3 = 4
    var voto4 = 6
    var media = (voto1 + voto2 + voto3 + voto4) / 4f //la f serve per avere risultato float, altrimenti arrotonda all'intero
    println("La media dei voti è: " + media) // 6.75
}
```

Nel prossimo esempio, analogo a quello precedente, vediamo come convertire una stringa in un valore numerico:

```
fun main() {
    var voto1 = "8"
    var voto2 = 9
    var voto3 = 4
    var voto4 = 6
    var media = (voto1.toInt() + voto2 + voto3 + voto4) / 4f
    println("La media dei voti è: " + media)
}
```

OPERATORI

Vediamo un esempio:

```
fun main() {
    // operatori classici: + - / *

    // modulo: % ci restituisce il resto di una divisione
    println(2 % 2)// 0
    println(3 % 2)// 1

    var number = 5
    number += 10 // corrisponde a 10 + 5 e questo vale anche per - / *
    println(number) //15

    number++ // incrementa di 1 la variabile
    println(number) //16
    number-- // decrementa di 1 la variabile
    println(number) //15
}
```

C'è da approfondire una casistica che riguarda gli operatori di incremento e decremento. Infatti è possibile scrivere tali operatori anche prima della variabile e questo ne cambia anche il comportamento. Vediamo un esempio con il decremento per capire la differenza:

```
var number2 = 3
println(number2--)// 3
println(number2)//2
```

In questo esempio usiamo il decremento dopo la variabile, ma la stampa a video sulla stessa riga dell'operatore restituisce lo stesso valore con cui abbiamo inizializzato la variabile. Solo sulla seconda stampa a video avremo effettivamente il decremento del valore. Questo perché, nel caso in cui l'operatore è scritto dopo la variabile, il compilatore andrà prima a controllare il contenuto della variabile e, solo sulla riga successiva, andrà effettivamente a decrementarlo. Viceversa succede se andiamo a scrivere l'operatore prima della variabile:

```
var number2 = 3
println(--number2)// 2
```

dove il decremento avviene direttamente sulla stessa riga dove ho l'operatore. Stessa cosa vale per l'incremento.

DIFFERENZA VAL E VAR

Entrambe le keyword vengono utilizzate per dichiarare una variabile. La differenza sta nel fatto che, come abbiamo visto, quando abbiamo una variabile con la keyword var, possiamo inizializzare più volte il valore che contiene, sostituendo quello precedente. Viceversa, una variabile con la keyword val viene considerata "immutabile", cioè una volta inizializzata ad un valore, quel valore non può essere più cambiato. Vediamo un esempio:

```
val voto1 = 10
voto1 = 9 //Val cannot be reassigned

var voto2 = 10
voto2 = 9
```

TRACCIA 1 PER ESERCITAZIONE

Per completare questo esercizio dovrai:

- Creare un algoritmo in grado di calcolare la tua età .
- Per fare questo dovrai poter inserire solo l'anno in cui sei nato.
- Non potrai scrivere a mano l'anno corrente ma dovrai prenderlo tramite delle classi Kotlin o Java (cerca su Internet come fare)

```
import java.text.SimpleDateFormat
import java.time.Year
import java.util.*

fun main(args: Array<String>) {
    print("Il mio anno di nascita è ")
    var annoNascita:Int = readLine()!!.toInt()
    // readLine() è usato per accettare la stringa
    // e ".toInt()" la converte da stringa a intero.
    //val sdf = SimpleDateFormat("dd/M/yyyy hh:mm:ss")
    val sdf = SimpleDateFormat("yyyy")
    val dataCorrente = sdf.format(Date())
    //System.out.println(" La data di oggi è: "+ currentDate)
    var eta:Int= dataCorrente.toInt() - annoNascita
    print("Ho $eta anni")
}
```

STRINGHE

Come sappiamo, le stringhe sono variabili che contengono del testo. In JAVA, per stampare a video un valore numerico con del testo affianco, è necessario concatenare il valore numerico dopo o prima del testo, come nell'esempio:

```
fun main() {
    val moneta = 5.34
    println("Il totale delle monete in mio possesso è " + moneta)
}
```

In Kotlin è possibile farlo in una maniera meno macchinosa grazie all'utilizzo del simbolo $, come nel prossimo esempio:

```
fun main() {
    val moneta = 5.34
    println("Sono in possesso di $moneta monete")
}
```

In questo modo è possibile richiamare una variabile direttamente all'interno della stringa. Il tipo della variabile da inserire nella stringa può essere qualsiasi, anche un carattere o un'altra stringa.

Possiamo anche eseguire delle operazioni all'interno della stringa, come nel seguente esempio:

```
fun main() {
    val moneta = 5.34
    val tasse = 2.20
    println("Sono in possesso di ${moneta - tasse} monete")
}
```

Kotlin mette a nostra disposizione quelle che sono chiamate Raw Strings, che sono stringhe racchiuse tra 3 doppie virgolette e sono utilizzate per indicare il percorso di un file, come nell'esempio seguente:

```
val path = """C:\cartella1\cartella2\file"""
```

Possiamo anche utilizzare le Raw Strings anche per creare un output ordinato, facendo in modo che venga fuori un testo allineato senza spazi o tab indesiderati, come nel seguente esempio:

```
val biografia = """Mi chiamo Stefano
                  |ho 27 anni
                  |faccio l'informatico""".trimMargin()
println(biografia)
```

L'output di tale codice è il seguente:

Mi chiamo Stefano
ho 27 anni
faccio l'informatico

Se al posto del simbolo |, che è utilizzato di default da Kotlin per allineare il testo, volessimo utilizzare un altro simbolo a nostra scelta, dobbiamo fare come nell'esempio seguente:

```
val biografia = """Mi chiamo Stefano
                  -ho 27 anni
                  -faccio l'informatico""".trimMargin(marginPrefix = "-")
println(biografia)
```

L'output di questo codice è uguale a quello precedente.

ARRAY

Un'array è un insieme di variabili. Ogni variabile occupa una posizione all'interno dell'array, contrassegnata da un indice. Si deve però prestare attenzione al fatto che l'indice in prima posizione parte da 0 e non da 1. Per dichiarare un array abbiamo bisogno della keyword val (perchè l'array nasce come struttura immutabile), seguita da un nome e un metodo arrayOf(), che ha come parametri un insieme di valori che saranno inseriti all'interno dell'array. Vediamo qualche esempio:

```
fun main() {
    val arrayInteri = arrayOf(1, 2, 3, 4)
    val arrayStringhe = arrayOf("Marco", "Anna", "Matilde", "Gianfranco")
    val arrayMisto = arrayOf("marco", 2, 5, 4.0, 8.0f, 's', false)
    //ciclo for per leggere ogni elemento dell'array di interi
    for (numero in arrayInteri){
        println(numero)
    }
    println()
    //ciclo for per leggere ogni elemento dell'array di stringhe
    for (nome in arrayStringhe){
        println(nome)
    }
    println()
    //ciclo for per leggere ogni elemento dell'array misto
    for (elemento in arrayMisto){
        println(elemento)
    }
}
```

È possibile tuttavia dichiarare un array, scrivendo per ogni singolo indice che valore inserire. Vediamo un esempio:

```
val arrayInteri2: Array<Int?> = arrayOfNulls(3)
arrayInteri2[0] = 1
arrayInteri2[1] = 2
arrayInteri2[2] = 3
//lettura di un singolo elemento dell'array all'indice indicato
println(arrayInteri2[1])
//ciclo for per leggere ogni elemento dell'array misto
for (elemento2 in arrayInteri2){
    println(elemento2)
}
```

CONDIZIONALI

IF ELSE STATEMENT

Vediamo un esempio di classico costrutto if:

```
fun main() {
    val totaleMonete = 0
    //val totaleMonete = 50
    //val totaleMonete = 10
    if (totaleMonete > 0 && totaleMonete <= 5){
        print("posso spendere pochi soldi")
    }else if (totaleMonete > 5 && totaleMonete <= 20){
        print("posso comprarmi un buon pasto")
    }else if(totaleMonete > 20) {
        print("sono ricco")
    }else if (totaleMonete == 0){
        print("sono povero")
    }
}
```

É possibile in Kotlin creare degli if else statement con un range di valori all'interno della condizione (con estremi compresi nell'intervallo), come nel seguente esempio:

```
fun main() {
    val totaleMonete = 7
    if (totaleMonete in 1..5){
        print("posso spendere pochi soldi")
    }else if (totaleMonete in 6..20){
        print("posso comprarmi un buon pasto")
    }else if(totaleMonete > 20) {
        print("sono ricco")
    }else if (totaleMonete == 0){
            print("sono povero")
    }
}
```

Una funzione simile ai due puntini è until, solo che non comprende nell'intervallo il secondo estremo, come nell'esempio:

```
fun main() {
    val totaleMonete = 4
    if (totaleMonete in 1 until 5){ // totaleMonete >=1 && totaleMonete < 5
        print("posso spendere pochi soldi")
    }else if (totaleMonete in 6..20){
        print("posso comprarmi un buon pasto")
    }else if(totaleMonete > 20) {
        print("sono ricco")
    }else if (totaleMonete == 0){
            print("sono povero")
    }
}
```

WHEN STATEMENT

Il costrutto when viene utilizzato come se fosse uno switch in altri linguaggi. Quello che si fa con when, si può fare anche con un if else, solo che il costrutto when è più pulito dal punto di vista della leggibilità del codice. Vediamo l'esempio precedente con questo costrutto:

```
fun main() {
    val totaleMonete = 4
    when(totaleMonete){
        in 0..5 -> println("posso spendere pochi soldi")
        in 6..20 -> println("posso comprarmi un buon pasto")
        else -> {println("sono ricco")}
    }
}
```

CICLI

Un ciclo è un blocco di codice che viene eseguito più volte, finché una determinata condizione non sarà verificata. I cicli possono essere definiti in vari modi e possono essere di vari tipi.

FOR

Il ciclo for è quello più utilizzato in assoluto. La variabile inserita all'interno della condizione non deve essere dichiarata, ma sarà il compilatore che inserirà direttamente un var dietro le quinte. Vediamo alcuni esempi:

```
fun main() {
    for(x in 1..10){
        print("$x ") //1 2 3 4 5 6 7 8 9 10
    }
}
```

```
fun main() {
    for(x in 1 until 10){
        print("$x ") //1 2 3 4 5 6 7 8 9
    }
}
```

```
fun main() {
    for(x in 15 downTo 1) {
        print("$x ") //15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
    }
}
```

```
for(x in 1..10 step 2){
   print("$x ") //1 3 5 7 9
}
```

Il for è utilizzato anche per poter accedere ad ogni elemento di una lista e di una maplist, ed eseguirvi del codice su di ognuno di esso. Vediamo degli esempi su queste strutture:

```
fun main() {
    val listaNomi= listOf<String>("Marco", "Anna", "Michele", "Alfredo")
    for(nome in listaNomi){
        print("$nome ")//Marco Anna Michele Alfredo
    }
    print("\n")
    val mapListNomi= mapOf(30 to "Marco", 20 to "Anna", 40 to "Michele", 50 to
"Michele")
    for((key, value) in mapListNomi){
        print("$key $value, ")//30 Marco, 20 Anna, 40 Michele, 50 Michele,


    }

}
```

WHILE

É importante sapere che tutto ciò che si può fare con il ciclo for, si può fare anche con il ciclo while. Il ciclo for in alcuni casi può essere più comodo da utilizzare rispetto il ciclo while, però ci sono altri casi in cui il while diventa più importante del for.

Nel caso del ciclo while, la variabile da utilizzare all'interno della condizione ha bisogno di essere dichiarata e inizializzata. Il codice al suo interno verrà eseguito finché la condizione al suo interno risulta essere vera, altrimenti esce fuori. Vediamo un esempio:

```
fun main() {
    var x = 1
    while (x < 11){
        print("$x ")//1 2 3 4 5 6 7 8 9 10
        x++
    }
}
```

Vediamo un altro esempio in cui creiamo un loop infinito, in quanto la condizione sarà sempre vera e il codice verrà sempre rieseguito, quindi non riuscirà mai ad uscire dal ciclo:

```
fun main() {
    while(true){
        println ("sto eseguendo")
    }
}
```

DO WHILE

Il do while è molto simile al while, ma il do while esegue il codice almeno una volta. Mentre il while, prima di eseguire il blocco di codice, va a verificare la condizione almeno una volta, il do while esegue prima il blocco di codice e poi va a verificare se la condizione è vera, oppure no. Anche qui, il codice verrà rieseguito finché la condizione risulterà essere vera. L'utilizzo del do while in programmazione è molto raro, ma si utilizza soprattutto nei casi in cui si deve eseguire un blocco di codice e poi dobbiamo andare a controllarlo. Vediamo un esempio che ricrea la tabellina del 2:

```
fun main() {
    var num = 2
    var i = 1
    do{
        println("2 * $i = ${num * i}")
        i++
    }while (i < 11)
}
```

TRACCIA 3 PER ESERCITAZIONE

La traccia 2 verrà svolta quando si studieranno le MapList.

Per completare questo esercizio dovrai:

- utilizzare un ciclo per poter prendere i nomi dalla lista `names`
- dovrai successivamente sostituire lo spazio tra i nomi con un "_" e sistemare le lettere in modo che siano tutte minuscole. Il risultato dovrebbe essere con quello mostrato all'interno della lista `usernames`

val names = listOf("Marco Rossi", "Alfredo Andrei", "John Mayer", "Justin Biber")

//The output should be like this:
usernames = [marco_rossi, alfredo_andrei, john_mayer, justin_biber]

Per completare questo task dovrai cercare su internet come sostituire i caratteri all'interno delle stringhe con altri e come rendere tutte le lettere minuscole.

```
fun main() {
    val names = listOf("Marco Rossi", "Alfredo Andrei", "John Mayer", "Justin
Biber", "Maria Verdi")
    val namesArray: Array<String?> = arrayOfNulls(names.size)
    var x = 0
    var y = 0
    print("usernames = [")
    for(nome in names){
        val nomeMinuscolo = nome.lowercase().replace(" ","_")
        namesArray[x] = nomeMinuscolo
        x++
    }
    for (nomeMinuscolo in namesArray){
        if(y < namesArray.size-1) {
            print("$nomeMinuscolo, ")
        } else {
            print("$nomeMinuscolo")
        }
        y++
    }
    print("]")
}
```

BREAK E CONTINUE

Qualche volta abbiamo bisogno di bloccare un ciclo, ed è qui che entrano in gioco le keyword break e continue. Entrambe possono essere usate per cicli for e while.

Break serve per terminare un ciclo, mentre continue serve solo per saltare un'iterazione del ciclo. Vediamo un esempio in cui abbiamo un programma in cui andremo a vedere se il peso totale di un quantitativo di prodotti supera il peso massimo che può portare un camion di rifornimenti:

```
fun main() {

    val mapFood = mapOf(
        "banane" to 15,
        "materassi" to 24,
        "mangime per cani" to 42,
        "attrezzi da lavoro" to 120,
        "formaggi" to 5)

    var pesoCamion = 0
    val articoli = mutableListOf<String>()

    for ((tipoArticolo, pesoArticolo) in mapFood){
        println("Il peso totale del camion è $pesoCamion")
        if (pesoCamion >= 100){
            println("ciclo stoppato")
            break
        }
        else if (pesoCamion + pesoArticolo > 100) {
            println("ciclo saltato")
            continue /*salta iterazione,
            quindi non aggiunge l'ultimo articolo sul
            camion se il peso che sta portando il
            camion in quel momento + il peso del prossimo
            articolo supera il peso massimo (100)
            che può portare il camion */
        } else {
            println("aggiungi $pesoArticolo di $tipoArticolo")
            articoli.add(tipoArticolo)
            pesoCamion += pesoArticolo
        }
    }
}
```

Vediamo un altro esempio con il continue in cui dobbiamo avvisare l'utente di caricare sul camion solo frutta e non altri tipi di cibi:

```
fun main() {
    val cibi = arrayListOf<String>("mela", "toast", "arancia", "pizza", "pasta",
"kebab")
    val frutti = arrayListOf<String>("arancia", "mela", "strawberry", "pera")

    var contatoreDiFrutta = 0 // serve per contare quanti tipi di frutta stiamo
caricando sul camion

    for (cibo in cibi){
        if (cibo !in frutti){ //Se un cibo non è nella lista di frutti
            println("$cibo non è un frutto")
            continue
        }else{
            println("$cibo è un frutto")
        }
        contatoreDiFrutta++
    }
    println("Il totale dei frutti è $contatoreDiFrutta")
}
```

TRACCIA 4 PER ESERCITAZIONE

Per poter completare questo esercizio dovrai:

- scrivi un ciclo con la parola chiave `break` per creare una stringa `titleResult` che sia lunga esattamente 140 caratteri
- dovrai prendere i titoli dalla lista `headlines` e metterli dentro la stringa `titleResult` uno dopo l'altro con uno spazio tra un titolo e l'altro per dividere i vari titoli.
- l'output dovrà essere come mostrato sotto
- anche per questo esempio avrai bisogno di cercare su internet come poter tagliare le stringhe
- usa questo sito per verificare la lunghezza del tuo output (link)

```
var titleResult = ""

val headlines = listOf(
   "Ballata dell'usignolo e del serpente",
   "Tutti pronti per la scuola primaria!",
   "La prova preselettiva del Concorso Ordinario per le Scuole",
   "Se scorre il sangue",
   "Profezie: Che cosa ci riserva il futuro",
   "Una notte ho sognato New York",
   "Vado in prima. Attività, giochi, pregrafismi, lettere e numeri"
)


//The output should be like this:
//Ballata dell'usignolo e del serpente Tutti pronti per la scuola primaria! La prova preselettiva del Concorso Ordinario per le Scuole Se scor
```

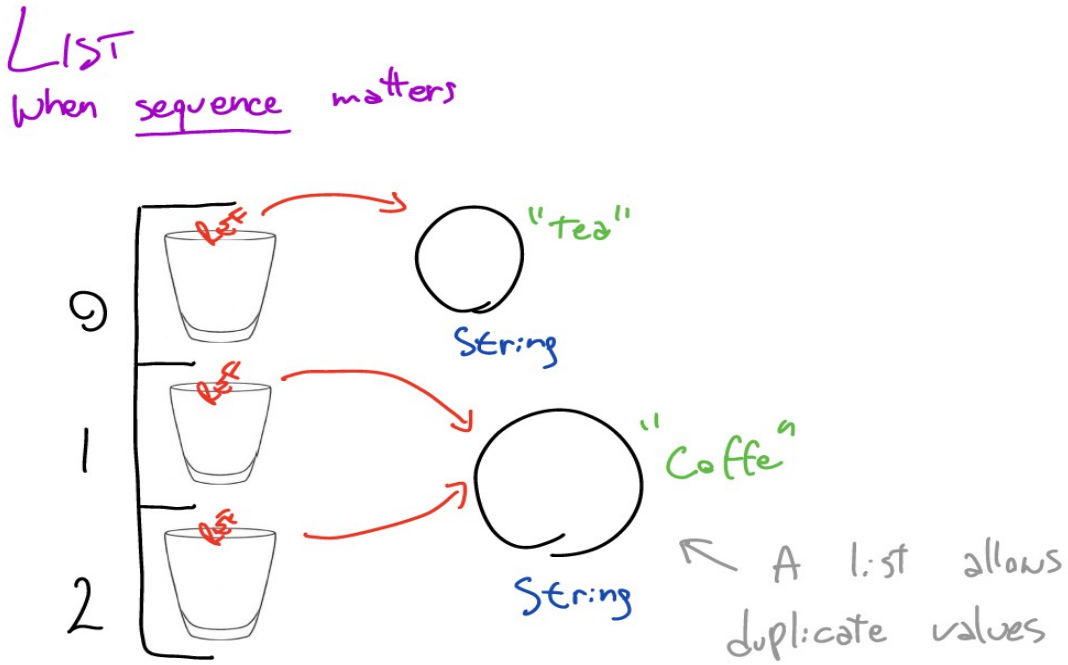
```
fun main() {
    var titleResult = ""

    val headlines = listOf(
        "Ballata dell'usignolo e del serpente",
        "Tutti pronti per la scuola primaria!",
        "La prova preselettiva del Concorso Ordinario per le Scuole",
        "Se scorre il sangue",
        "Profezie: Che cosa ci riserva il futuro",
        "Una notte ho sognato New York",
        "Vado in prima. Attività, giochi, pregrafismi, lettere e numeri"
    )
    for(titolo in headlines){
        if(titleResult.length > 140) {
            break
        }
        titleResult += titolo + " "
    }
    print(titleResult)
}
```

COLLEZIONI

Gli array sono le collezioni per eccellenza, che esistono dapprima delle collezioni che stiamo per vedere. Sono meno flessibili rispetto le list, set e map, ma sono comunque molto utilizzati.

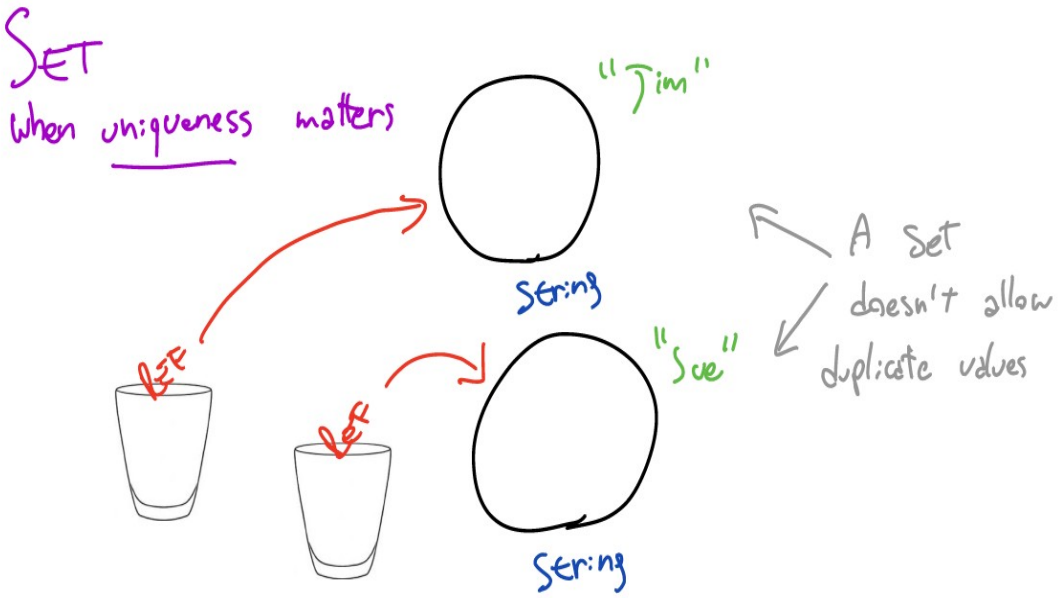
Le collezioni sono costrutti che possono contenere al loro interno un tot di elementi.

Le list in primis sono liste che contengono elementi contrassegnati da dei riferimenti, esattamente come gli array. É importante sapere che nelle list viene tenuto conto dell'ordine in cui inseriamo gli elementi al suo interno, ragion per cui le potremmo anche chiamare liste ordinate. Altra caratteristiche importante delle list è che più riferimenti possono puntare ad uno stesso valore, come nell'illustrazione che segue in cui due riferimenti della lista puntano a "coffe":

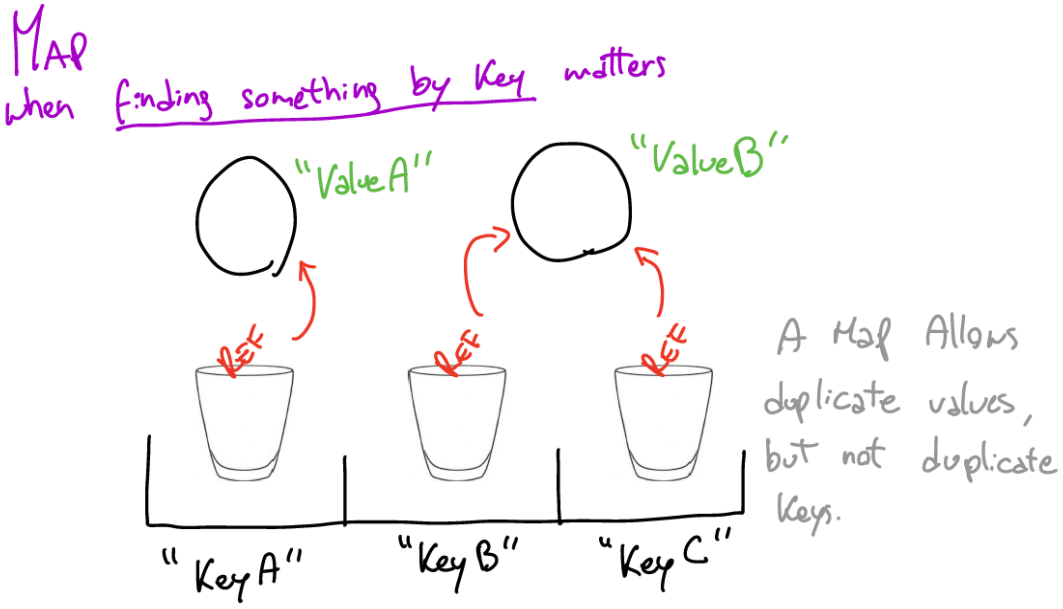


Più riferimenti all'interno di una lista possono puntare uno stesso valore perché all'interno di tale costrutto sono ammessi i duplicati. Una differenza che abbiamo tra list e array è che nelle list abbiamo molti più metodi che ci permettono di gestire il loro contenuto.

I set, invece, sono i costrutti dove non sono ammessi duplicati, ma conta che ogni elemento sia univoco. Se dovessimo provare a codice ad inserire un elemento più volte, il compilatore non ci darebbe errore, ma eliminerebbe per noi tutti i duplicati e ne lascerebbe solo uno. In più, in questo caso, non conta neanche l'ordine degli elementi, ma sono inseriti a caso. Ciò significa che gli elementi non sono contrassegnati ad indice.



I map sono costrutti dove ogni elemento è costituito dalla coppia chiave ed elemento. In linguaggio JAVA sono chiamati hash map. La logica è molto simile alla list, ma al posto di avere gli indici, abbiamo le chiavi che definiscono la posizione di un elemento. In questo caso, come per le list, sono ammessi i duplicati.



Ognuna di queste classi può essere immutabile o mutabile, ma di default sono immutabili. Per renderli mutabili, basta aggiungere a codice la keyword mutable prima di list, set e map. Nel caso di mutable, possiamo aggiungere o togliere elementi, viceversa non posso effettuare modifiche.

Vediamo un esempio per le list:

```
fun main() {
    val ml = mutableListOf(1, 3, 2)
    //Metodi utilizzabili per list mutable e immutable
    println(ml.contains(2))//stampa true, perchè la lista contiene questo valore

    println(ml.indexOf(1))//stampa 0, perchè ci restituisce la posizione
dell'elemento all'interno della lista.
    //Nel caso di duplicati, indexOf() restituisce la posizione del primo valore
che trova

    ml.sort()//stampa 1 2 3 4
    print("Gli elementi della lista sono: ")
    for (elemento in ml){
        print("$elemento ")//stampa 1 2 3 perchè sort ordina la lista in maniera
crescente.
        //Il metodo reverse(), invece, ordina la lista in maniera decrescente
        //Il metodo shuffle() mischia in modo random gli elementi della lista
    }
    println()

    //Metodi utilizzabili solo per list mutable
    ml.add(4)
    print("Gli elementi della lista sono: ")
    for (elemento in ml){
        print("$elemento ")//stampa "Gli elementi della lista sono: 1 3 2 4 "
perchè add() aggiunge un elemento
        //in coda alla lista. É possibile indicare ad add() anche la posizione
in cui aggiungere l'elemento
        //con la seguente sintassi: add(indice, elemento)
    }
    println()
    ml.remove(3)
    print("Gli elementi della lista sono: ")
    for (elemento in ml){
        print("$elemento ")//stampa "Gli elementi della lista sono: 1 2 4 "
perchè remove() toglie un elemento
        //dalla lista. É possibile utilizzare il metodo removeAt() per indicare
la posizione dell'elemento
        //da rimuovere, con la seguente sintassi: removeAt(indice)
    }
    println()
    ml.set(0, 8)
    print("Gli elementi della lista sono: ")
    for (elemento in ml){
        print("$elemento ")//stampa "Gli elementi della lista sono: 8 2 4 "
perchè set() sovrascrive un elemento
        //nella lista. La sintassi è set(indice elemento da sovrascrivere,
elemento nuovo)
    }
}
```

Vediamo un esempio per i set:

```
fun main() {
    val amici = mutableSetOf("Marco","Giulia","Giulia","Gianni","Gianni")

    println(amici.contains("Marco"))//true
    println(amici.add("Alessandro"))//true perchè add() aggiunge Alessandro alla
lista
    println(amici.add("Alessandro"))//false perchè add() non aggiunge Alessandro
alla lista, essendo un duplicato
    println(amici.remove("Alessandro"))//true perchè remove() rimuove dalla
lista il primo e unico Alessandro

    print("La lista di amici è ")
    for (amico in amici){
        print("$amico ")//La lista di amici è Marco Giulia Gianni
    }
}
```

Vediamo un esempio per i Map:

```
fun main() {

    val ricette = mutableMapOf("ricetta1" to "Pizza", "ricetta2" to "Thai",
"ricetta3" to "Sushi")

    println(ricette.getValue("ricetta1"))//Pizza

    println(ricette.values)//[Pizza, Thai, Sushi]

    for ((chiave, valore) in ricette) {
        println("chiave: $chiave, valore: $valore" )

/*chiave: ricetta1, valore: Pizza
chiave: ricetta2, valore: Thai
chiave: ricetta3, valore: Sushi*/
    }
    println()

    ricette.put("ricetta4", "Pasta")
    //ricette.remove("ricetta4")// remove() rimuove la chiave con il valore
corrispondente dalla lista
    //ricette.replace("ricetta4","Carne") replace() sovrascrive il valore alla
chiave corrispondente nella lista
    for ((chiave, valore) in ricette) {
        println("chiave: $chiave, valore: $valore" )

/*chiave: ricetta1, valore: Pizza
chiave: ricetta2, valore: Thai
chiave: ricetta3, valore: Sushi
chiave: ricetta4, valore:Pasta*/

    }
}
```

TRACCIA 2 PER ESERCITAZIONE

Per completare questo esercizio dovrai:

- Saper utilizzare i map (sezione 14)
- confrontare la lista `fruits` con `basketItems` (scritte sotto)
- sommare solo la quantità dei frutti nel carrello
- stampare quanti frutti ci sono nel carrello

val basketItems = mapOf("apples" to 4, "oranges" to 19, "kites" to 3, "sandwiches" to 8)
val fruits = listOf("apples", "oranges", "pears", "grapes", "bananas")

//The output should be like this:
//There are 23 fruits in the basket

```
fun main() {
    val basketItems = mapOf("apples" to 4, "oranges" to 19, "kites" to 3,
"sandwiches" to 8)
    val fruits = listOf("apples", "oranges", "pears", "grapes", "bananas")
    var contaFrutta = 0
    for((key,value) in basketItems){
        for(frutta in fruits){
            if(frutta == key){
                contaFrutta += value
            }
        }
    }
    println(contaFrutta)
}
```

FUNZIONI O METODI

Una funzione (o metodo) si utilizza per evitare di dover riscrivere più volte nella stessa classe un determinato blocco di codice. In questo modo, per eseguire quel blocco di codice basterà richiamare il metodo che lo contiene.

Per dichiarare una funzione si deve utilizzare la keyword fun, seguita da un nome, le parentesi tonde per i parametri e infine le graffe. Una volta dichiarata una funzione, la si può richiamare semplicemente utilizzando il suo nome e definendo i valori degli eventuali parametri all'interno delle parentesi tonde.

Se nella dichiarazione di una funzione abbiamo dei parametri, essi devono essere obbligatoriamente seguiti dal loro tipo, a differenza di come facciamo quando dichiariamo delle variabili normali.

Le dichiarazioni di funzioni devono essere sempre effettuate all'esterno della funzione main(), mentre la loro chiamata sempre all'interno.

Vediamo un esempio:

```
fun main() {
    eat("mela")//Stai mangiando una mela
}

fun eat(tipoDiCibo: String?){
    print("Stai mangiando una $tipoDiCibo")
}
```

A differenza di JAVA, in Kotlin è possibile impostare nei parametri di una funzione un valore di default, in modo tale che se nella chiamata non gli assegniamo nessun valore, il compilatore utilizza quello di default, altrimenti utilizza il valore che assegniamo nella chiamata. Vediamo un esempio:

```
fun main() {
    eat()//Stai mangiando una mela
}

fun eat(tipoDiCibo: String = mela){
    print("Stai mangiando una $tipoDiCibo")
}
```

TIPI DI RITORNO

Quando andiamo a ritornare un valore ad una funzione, abbiamo bisogno della keyword return. Le chiamate delle funzioni con il return, a differenza di quelle che non ritornano niente, hanno bisogno di essere assegnate ad una variabile, possibilmente dello stesso tipo del valore che restituisce il return.

Vediamo un esempio:

```
fun main() {
   val risultatoMoltiplicazione = moltiplicazione(5,5)
    print(risultatoMoltiplicazione)//25
}

fun moltiplicazione(a: Int, b: Int) : Int {
    var c = a * b
    return c
}
```

FUNZIONI ABBREVIATE

Il motivo per cui tali funzioni si chiamano abbreviate è per il semplice fatto che il loro corpo di riduce ad una sola riga di codice. Al posto delle parentesi graffe, hanno un = per indicare il loro corpo di codice e non riconoscono la keyword return. Vediamo un esempio:

```
fun main() {
    println(potenza(3))//9
    stampaMessaggio()//Hello World!
}

fun potenza(x: Int): Int = x * x

fun stampaMessaggio() = println("Hello World!")
```

TOP LEVEL FUNCTION

Le top level function sono state introdotte da Kotlin, ma hanno comunque un forte legame con il linguaggio JAVA. Queste funzioni vivono di vita propria, quindi non hanno bisogno delle classi per poter essere racchiuse.

In JAVA, per poter andare a creare una fuzione, la si deve racchiudere all'interno di una classe. Questo vale anche per il main(), che per essere eseguito in JAVA, ha bisogno comunque di una classe.

Ragion per cui, per evitare di dover creare delle classi apposite per dover eseguire una funzione, Kotlin ha introdotto queste top level function.

Qui di seguito possiamo vedere un esempio di top level function, ossia un esempio di funzione che non è all'interno di una classe, ma semplicemente in un file Kotlin:

```
1  package com.chikekotlin.projectx.utils
2
3  fun checkUserStatus(): String {
4      return "online"
5  }
```

Dal momento che, però, il codice Kotlin viene eseguito sulla JVM (Java Virtual Machine), tale codice viene trasformato da Kotlin a JAVA. Il codice scritto in Kotlin che abbiamo nell'esempio precedente, viene anch'esso trasformato in JAVA e si presenterà in questo modo dopo la compilazione:

```
1  /* Java */
2  package com.chikekotlin.projectx.utils
3
4  public class UserUtilsKt {
5
6      public static String checkUserStatus() {
7          return "online";
8      }
9  }
```

Quindi, la funzione di prima tradotta in JAVA verrà comunque inserita dal compilatore in una classe. Da intendersi che queste traduzioni in JAVA avvengono dietro le quinte, tutto all'oscuro dell'utente.

Quando andiamo a richiamare quel metodo in Kotlin, invochiamo semplicemente il nome della funzione. Ma anche in questa fase il compilatore effettuerà una traduzione in JAVA, invocando anche il nome della classe che contiene quel metodo, oltre che il metodo stesso, come nell'esempio seguente:

```
1  /* Java */
2  import com.chikekotlin.projectx.utils.UserUtilsKt
3
4  ...
5
6  UserUtilsKt.checkUserStatus()
```

Nel caso in cui in Kotlin volessimo cambiare il nome della classe creata dietro le quinte da JAVA, si può fare nel seguente modo:

Dopo aver cambiato in Kotlin il nome della classe, il nuovo nome verrà utilizzato da JAVA dietro le quinte nel momento in cui viene richiamata la funzione:

INTEROPERABILITY

L'interoperability è la capacità dei linguaggii Kotlin e JAVA (da JAVA 10 in poi) di potersi richiamare a vicenda le funzioni scritte nelle rispettive sintassi. Quindi, dal JAVA è possibile richiamare funzioni Kotlin ed eseguirle sempre in JAVA, e viceversa. Vediamo il seguente esempio:

```
fun main() {
    val PI = Math.PI //Math è una classe interamente scritta in JAVA
    Math.asin(5.5)
}
```

Quello che stiamo facendo in questo esempio è richiamare la costante PI e il metodo asin() presenti all'interno della classe Math. Con il tasto Ctrl su Intellij è possibile accedere alla classe Math e verificare ciò:

```
public final class Math {

    Don't let anyone instantiate this class.
    private Math() {}

    The double value that is closer than any other to e, the base of the natural logarithms.
    public static final double E = 2.7182818284590452354;

    The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.
    public static final double PI = 3.14159265358979323846;

    Returns the trigonometric sine of an angle. Special cases:
    • If the argument is NaN or an infinity, then the result is NaN.
    • If the argument is zero, then the result is a zero with the same sign as the argument.
    @Contract(pure = true)
    public static double asin(double a) { return StrictMath.asin(a); // default impl. delegates to StrictMath }

    Returns the arc cosine of a value; the returned angle is in the range 0.0 through pi. Special case:
    • If the argument is NaN or its absolute value is greater than 1, then the result is NaN.
    The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic.
    Params: a – the value whose arc cosine is to be returned.
    Returns: the arc cosine of the argument.
    @Contract(pure = true)
    public static double acos(double a) { return StrictMath.acos(a); // default impl. delegates to StrictMath }

    Returns the arc tangent of a value; the returned angle is in the range -pi/2 through pi/2. Special cases:
    • If the argument is NaN, then the result is NaN.
    • If the argument is zero, then the result is a zero with the same sign as the argument.
```

Possiamo notare che la classe Math ha anche tante altre variabili e metodi al suo interno, anch'esse tutte riutilizzabili in Kotlin.

EXTENSION FUNCTIONS

Le extension functions sono molto utili per andare a scrivere il minor numero di righe di codice possibile. Vediamo un esempio:

```
fun main() {
    val a = 5.potenza()
    println(a)//25

    "Hello world!".stampa()

}

fun Int.potenza() : Int {
    return this * this // Il this si riferisce al valore che sta nella chiamata
di tale funzione, prima del punto
}

fun String.stampa(){
    print(this)
}
```

INFIX FUNCTION

Anche le infix function sono molto utili per abbreviare il nostro codice e renderlo più pulito. Per dichiararle si utilizza la keyword infix prima di fun, mentre per il resto rimane tutto pressochè invariato.

Le infix function ci permettono di trasformare una funzione in un operatore matematico, ma per poterle usare si devono seguire delle regole:

- La infix function deve essere obbligatoriamente dichiarata all'interno di una classe.
- La infix function deve avere un singolo parametro.
- La infix function non può avere valori di default.

Vediamo un esempio:

```
fun main() {
    val y = 10 aggiunge 20 // chiamata della infix function
    println(y)
}

infix fun Int.aggiunge(b:Int) : Int {
    return this + b
}
```

NULLS TYPES

Questo concetto introdotto da Kotlin va a salvare tanti dei nostri errori che svolgiamo nella scrittura di un codice.

Un errore che capita molto spesso nella programmazione JAVA è la NullPointerException, che viene fuori quando un oggetto è null. Detto questo, un oggetto può essere null, ma questo errore viene fuori quando su quell'oggetto null viene richiamato un metodo.

Partiamo da una classe di tipo Person:

```
var p = person()
```

In questo esempio stiamo salvando l'oggetto Person all'interno della variabile p, però tale variabile non contiene l'oggetto vero e proprio, ma contiene il suo riferimento. É un po' come se andassimo a copiare un link di una pagina web all'interno di una cartella e, quindi, nel momento in cui andiamo a cliccare su quel link, ci compare la pagina. Ma ciò non significa che la pagina web è stata salvata dentro la cartella, ma solo il collegamento per arrivare ad esso. Questo è ciò che succede in maniera del tutto similare all'interno di questa variabile p.

Detto ciò, in questo caso è impossibile avere un NullPointerException, perché stiamo andando proprio ad inizializzare la nostra variabile, quindi non stiamo dicendo che sarà uguale a null.

In Kotlin, nel caso in cui dovessimo avere una variabile che punta a null e su quella variabile venisse invocato un metodo, il compilatore è in grado di capire che quel metodo non potrà mai essere eseguito e si ferma alla sola lettura e stampa a video del valore null. É come se il compilatore svolgesse un controllo con un if, la cui condizione dice che se la variabile è diversa da null, allora può eseguire il metodo, altrimenti non fa nulla e fa solo la stampa a video del valore null. Vien da sé che in kotlin non avremo mai, quindi, una NullPointerException. Vediamo un esempio su questo concetto:

```
fun main() {
    var name: String? = null
    //name = "Marco"

    println(name?.length)/* stampa null con name commentato, altrimenti 5 con
name decommentato.
                         É come se facessi: if (name != null) {
                                                println(name.length)
                                            }else{
                                                println(null)
                                            }*/
}
```

In questo esempio, il punto interrogativo che abbiamo nella dichiarazione della variabile è obbligatorio quando alla variabile ci assegniamo il valore null. Con il punto interrogativo nella chiamata al metodo length stiamo indicando, invece, al compilatore di svolgere un controllo sul fatto che la variabile name sia null o meno. Ma vediamo anche l'esempio seguente:

```
fun main() {
    var name: String? = null
    println(name!!.length)//NullPointerException
}
```

In questo esempio abbiamo una NullPointerException, ma solo perché con i due punti esclamativi abbiamo detto che al compilatore che siamo sicuri che la variabile name non sia null, anche se effettivamente lo è.

ELVIS OPERATOR

Per capire a cosa serve l'operatore elvis, vediamo prima il seguente blocco di codice:

```
fun main() {
    var name: String? = null
    //name = "Marco"

    var check: Int = if (name != null){
        name.length
    }else{
        -1
    }

    println(check)//* stampa -1 con name commentato, altrimenti 5 con name
decommentato.
}
```

L'if nell'esempio permette di stampare un messaggio alternativo a null, che sarà -1 se la variabile dovesse puntare effettivamente a null.

L'elvis operator, che si scrive ?:, permette di ridurre l'if dell'esempio precedente in una sola riga di codice:

```
fun main() {
    var name: String? = null
    var check: Int = name?.length ?: -1
    println(check)// stampa -1 con name commentato, altrimenti 5 con name
decommentato.
}
```

LET

Finora abbiamo visto come gestire una variabile nel momento in cui su di essa dobbiamo eseguire un'operazione e non è null. Ma, sempre nel caso in cui tale variabile non sia null, ci potrebbero essere delle casistiche in cui su quella variabile devono essere eseguite più di una singola operazione, ed è qui che interviene la keyword let. Questo serve per evitare di scrivere più if annidati. Vediamo un esempio:

```
fun main() {
    var name: String? = null
    name = "Mario"
    name?.let {
        println("Il nome è $it")//Il nome è Mario
        println("La lunghezza della stringa $it è ${it.length}")//La lunghezza
della stringa Mario è 5
        println("Nome: $it Cognome: Rossi")//Nome: Mario Cognome: Rossi
    }
}
```

Nell'esempio, il blocco di codice tra le graffe viene eseguito solo se name non è null. Quando andiamo a richiamare name con la keyword let, il compilatore ci passa in automatico un parametro che si chiama it. Tale parametro serve per richiamare la variabile (in questo caso name), all'interno del blocco di codice tra le graffe.

BY LAZY

By lazy viene utilizzato per avere meno problemi per quanto riguarda la locazione di memoria. Vediamo un esempio per capire questo concetto:

```
val number: Float = 98.9f

fun main() {
    println("un po' di codice...")

    println("un altro po' di codice...")
}
```

In questo esempio, anche se non richiamiamo la variabile number all'interno del main(), essa comunque va ad occupare uno spazio nella memoria, sprecando una locazione. La variabile number acquisisce un senso ovviamente nel momento in cui viene utilizzata, cosa che in questo esempio non succede. Ma i problemi non finiscono qui:

```
val number: Float = 98.9f

fun main() {
    println("un po' di codice...")

    val result1 = number * 5 * 10

    val result2 = number * 5 * 10

    println("un altro po' di codice...")
}
```

In questo esempio, invece, la variabile number viene richiamata nel main(), ma viene anche reinizializzata dal compilatore ogni volta che su di essa viene eseguita un'operazione.

Per risolvere questi due problemi, interviene il by lazy, che è un inizializzatore:

```
val number: Float by lazy {
    98.9f
}

fun main() {
    println("un po' di codice...")

    val result1 = number * 5 * 10

    val result2 = number * 5 * 10

    println("un altro po' di codice...")
}
```

Facendo in questo modo evitiamo lo spreco di memoria, perché andrà a salvare il valore della variabile name nella cache, migliorando le performance del programma. In più, se andiamo a richiamare più volte number per eseguire delle operazioni su di essa, viene istanziata solo una volta, diversamente da prima che veniva istanziata ogni volta che era richiamata.

Quindi, il by lazy è stato creato per prevenire inizializzazioni non necessarie del nostro oggetto, salvandolo nella cache memory.

LATEINIT

Questo inizializzatore va anch'esso utilizzato nella dichiarazione di una variabile, come nel caso di by lazy. Il lateinit può essere utilizzato in sostituzione della dichiarazione di una variabile in cui abbiamo un'inizializzazione a null. Vediamo un esempio:

```
fun main() {

    val book1 = Book()//Richiamo la classe Book
    book1.title = "Title"//Richiamo la variabile title all'interno della classe
Book
    book1.stampaTitolo()//Richiamo il metodo stampaTitolo() all'interno della
classe Book

}

class Book{

    /*var title: String? = null*/ //Metodo classico per inizializzare variabile
a null
    lateinit var title: String //In sostituzione del metodo classico
    fun stampaTitolo(){
        println("Il titolo è $title")//Il titolo è Title
    }
}
```

Quando dichiariamo una variabile con lateinit dobbiamo considerare però due regole da rispettare:

- La variabile può essere dichiarata solo con var e non con val.
- Dopo la sua dichiarazione, la variabile deve essere obbligatoriamente definita (inizializzata) da qualche parte del codice.

LE CLASSI

Le classi sono l'argomento principale della programmazione Kotlin, essendo che è un linguaggio di programmazione ad oggetti, proprio come JAVA. Vediamo un esempio:

```
class Persona (var nome: String, var eta:Int, var altezza : Int) {
    fun stampaDati(){
        println("Nome : $nome, Età : $eta, Altezza: $altezza")
    }
}

fun main() {
    val mioAmico = Persona("Marco", 22, altezza = 180)//Primo oggetto di Persona
    mioAmico.stampaDati()//Nome : Marco, Età : 22, Altezza: 180


    val miaRagazza = Persona("Anna", 40, altezza = 160)//Secondo oggetto di
persona
    miaRagazza.stampaDati()//Nome : Anna, Età : 40, Altezza: 160

}
```

IL COSTRUTTORE

La parte delle parentesi tonde che vediamo dopo il nome della classe, si chiama costruttore. Il costruttore solitamente serve per andare a inserire i parametri di cui abbiamo bisogno dall'inizio della creazione della classe. Quindi, all'interno del costruttore andremo a mettere gli elementi chiave per poter in seguito costruire il nostro oggetto. Possiamo richiamare il costruttore dentro la sua classe con la keyword init, permettendoci di scrivere l'esempio precedente in un altro modo:

```
class Persona (var nome: String, var eta:Int, var altezza : Int) {
    fun stampaDati(){
        println("Nome : $nome, Età : $eta, Altezza: $altezza")
    }
    init {
        stampaDati()
    }
}

fun main() {
    val mioAmico = Persona("Marco", 22, altezza = 180)//Prima istanza di Persona
    //mioAmico.stampaDati()//Nome : Marco, Età : 22, Altezza: 180

    val miaRagazza = Persona("Anna", 40, altezza = 160)//Seconda istanza di
persona
    //miaRagazza.stampaDati()//Nome : Anna, Età : 40, Altezza: 160
}
```

GETTER E SETTER

Prima di partire con i getter e i setter, è bene sapere che il Kotlin implementa l'incapsulamento da solo, senza dovercene noi preoccupare. L'incapsulamento è un paradigma introdotto nei linguaggi di programmazione ad oggetti, che va a creare più sicurezza e stabilità all'interno del codice, migliorandone anche la manutenzione.

Prendendo come esempio il JAVA, per incapsulamento si intende il fatto di dichiarare private le nostre variabili, rendendole visibili solo nella classe in cui sono state dichiarate. Per questo motivo, le altre classi non possono accedere a queste variabili, creando sicurezza nel codice. Però in qualche modo dobbiamo comunque dover accedere a queste variabili da altre classi, ed è per questo che intervengono dei metodi di supporto, che si chiamano metodi getter e setter. Con il metodo set() andiamo ad impostare la nostra variabile, mentre con il get() la andiamo a prendere.

Visto che, però, in Kotlin non esiste l'incapsulamento, ci si pone la domanda sull'utilità di questi metodi. Questi metodi, come già anticipato, sono comunque importanti per la manutenzione del codice. In più, quando andiamo a settare o prendere una variabile, possiamo verificare delle condizioni, eseguire delle operazioni o effettuare delle modifiche.

Vediamo come utilizzare questi metodi utilizzando l'esempio del paragrafo precedente:

```
class GetterSetter (var nome: String, eta_param:Int, var altezza : Int) {
    var eta = eta_param
    set(value) {
        if (value > 0)
            field = value
        else
            println("L'età non può essere minore di 0")
    }
    get() {
        return field
    }
}

fun main() {
    val mioAmico = GetterSetter("Marco", 22, altezza = 180)
    mioAmico.eta = -5 //L'età non può essere minore di 0
}
```

Nell'esempio, alla variabile eta nel costruttore ci abbiamo aggiunto un _param, dopodichè su di essa viene eseguito un controllo sul valore che gli viene passato.

SECONDO COSTRUTTORE

Partiamo con il dire che le classi non necessitano obbligatoriamente di un costruttore, quindi potremmo anche fare a meno di inserire le parentesi tonde con i parametri dopo il nome della classe. Il compilatore poi, in fase di runtime, andrà comunque a creare un costruttore vuoto.

Ci possono essere, però, delle situazioni in cui abbiamo bisogno di più costruttori. Vediamo un esempio per capire questo concetto:

```
class Persona2 (val nome:String){
    constructor(nome:String, madre:Persona2):this(nome){
        //Corpo del codice
    }
}
```

Vedendo questo esempio, possiamo pensare ad un metodo main che stampa la variabile nome dal costruttore classico per sapere solo il nome di una persona, ed in seguito stampa le variabili del constructor (secondo costruttore) per sapere chi è la madre della persona presa dal costruttore classico. La keyword this sta ad indicare che il valore nella variabile nome del secondo costruttore è ricavata dalla variabile nome del costruttore classico.

Vediamo un altro esempio che rispetta la stessa logica di quello precedente:

```
class Studente (var nome: String){
    init {
        println("Il nome dello studente è $nome")//Il nome dello studente è
Giacomo
    }

    constructor(nome: String, matricola:Int):this(nome){
        println("La matricola di $nome è $matricola")//La matricola di Giacomo è
1111
    }
}

fun main(){
    val studente = Studente("Giacomo", 1111)
}
```

TRACCIA 5 PER ESERCITAZIONE

Per poter completare questo esercizio dovrai:

- creare un classe `Account` che dovrà contenere al suo interno un `pin` e un `balance` (quanti soldi abbiamo sul conto)
- la classe definirà un metodo `withdraw()` in cui andremo a chiedere quanti soldi vogliamo prelevare e andremo anche a verificare se il pin corrisponde a *46789*
- la classe definirà un metodo `deposit()` che ci permetterà di inserire soldi all'interno del nostro conto
- la classe definirà un metodo `check()`per verificare i soldi rimanenti nel nostro conto

fun withdraw(amount: Int, pin: Int)
fun deposit(amount: Int)
fun check() : Int

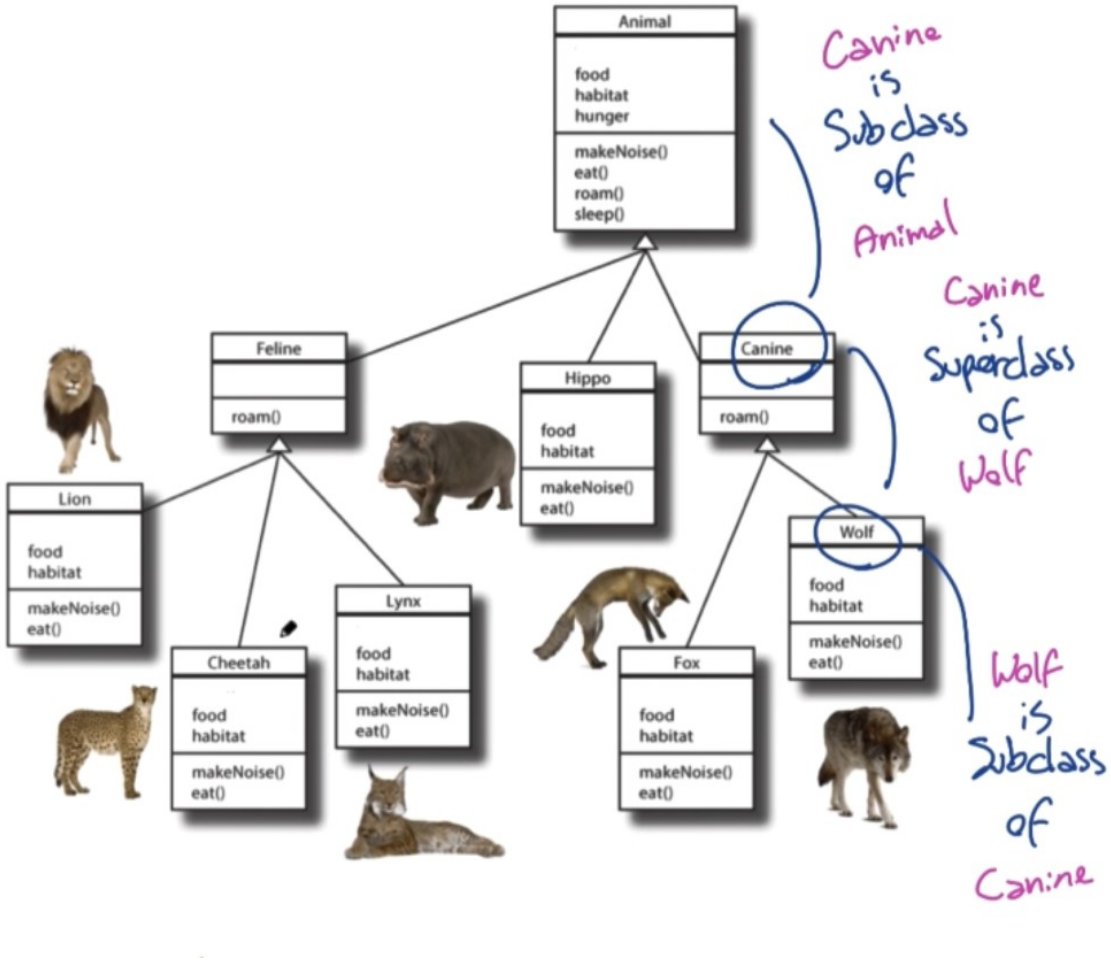
```
import kotlin.math.max

class Account (pin: Int, totSoldiConto: Int){
    var mioPin = pin
    var mioTotSoldiConto = totSoldiConto

    fun prelievo(totPrelievo: Int, pin: Int):Int{
        mioTotSoldiConto -= totPrelievo
        println("Ti sono rimasti nel conto $mioTotSoldiConto€")
        return mioTotSoldiConto
    }

    fun versamento(totVersamento: Int):Int{
        mioTotSoldiConto += totVersamento
        println("Ora hai nel conto $mioTotSoldiConto€")
        return mioTotSoldiConto
    }

    fun controlloTotaleSoldi() : Int {
        var totSoldiRimanenti = mioTotSoldiConto
        return totSoldiRimanenti
    }
}

fun main(){
    val mioAccount = Account(46789, 3000)
    var max = 0
    for(i in 0 .. max){
        println("""Scegli un'opzione:
            |1: Prelievo
            |2: Versamento
            |3: Totale soldi sul conto
            |4: Per uscire dal programma
        """.trimMargin())
        var opzione:String = readLine()!!.toString()
        when(opzione){
            "1" -> { print("Quanto vuoi prelevare?: ")
                  var totPrelievo:Int = readLine()!!.toInt()
```

```
                    if (totPrelievo > mioAccount.mioTotSoldiConto){
                        println("Non puoi permetterti questo prelievo!")
                    }else{
                        for (tentativo in 2 downTo 0){
                            print("Inserire pin: ")
                            var pin:Int = readLine()!!.toInt()
                            if (pin != mioAccount.mioPin){
                                if (tentativo > 0){
                                    println("Il pin inserito è sbagliato! Hai
$tentativo tentativi a disposizione")
                                    }else if(tentativo == 0) {
                                        println("Hai terminato i tentativi, torna
più tardi!")
                                        break
                                    }
                                }else{
                                    println("Hai inserito il pin corretto!")
                                    println("Prelievo effettuato")
                                    mioAccount.mioTotSoldiConto =
mioAccount.prelievo(totPrelievo, pin)
                                    break
                                }
                        }
                    }
            }
            "2" -> {print("Quanto vuoi versare: ")
                  var totVersamento:Int = readLine()!!.toInt()
                  mioAccount.mioTotSoldiConto =
mioAccount.versamento(totVersamento)
            }
            "3" -> {var totSoldiAccount = mioAccount.controlloTotaleSoldi()
                  println("Hai $totSoldiAccount € sul tuo conto")
            }
            "4" -> {break
            }
            else -> { println("Opzione non valida!")}
        }
        println()
        max++;
    }
}
```

EREDITARIETÁ: SUPERCLASSI E SOTTOCLASSI

Nei linguaggi di programmazione JAVA, Kotlin e Android, l'ereditarietà è molto usata.

Partiamo da un problema, per poi arrivare alla soluzione, in modo da capire al meglio il concetto di ereditarietà. Immaginiamo di voler andare a creare un'applicazione, che è in grado di salvare tutti gli animali presenti in natura al suo interno. Per fare questo, quindi, avremo tutta una serie di animali da dover classificare, come nel seguente esempio:



Nella nostra applicazione andremo a creare tutti gli animali, dopodiché si stabiliscono delle sopra-categorie e delle sotto-categorie. Ad esempio abbiamo:

- Il leone, il leopardo, la lince che appartengono alla famiglia dei felini.
- La volpe e il lupo che appartengono alla famiglia dei canini
- L'ippopotamo che è una famiglia a sé.

Ovviamente sia i felini che i canini appartengono alla famiglia degli animali.

Quindi, sostanzialmente la nostra app andrà a visualizzare tutti gli animali, suddivisi per sotto-categorie di felini e canini. Vien da sé che animali è la nostra categoria più in alto di tutte le altre, perché racchiude tutto quello che si può racchiudere riguardante l'argomento della nostra applicazione, comprese le sotto-categorie felini e canini.

Dal punto di vista della programmazione, possiamo dare le seguenti definizioni:

- Canino, Ippopotamo e Felino sono sottoclassi di Animale.
- Animale è superclasse di Canino, Ippopotamo e Felino
- Leone, Leopardo e Lince sono sottoclassi di Felino
- Felino è superclasse di Leone, Leopardo e Lince
- Volpe e Lupo sono sottoclassi di Canino
- Canino è superclasse di Volpe e Lupo

Tutti gli Animali hanno un determinato numero di proprietà, che verranno ereditate da tutte le sue sotto-categorie, comprese quelle di generazione superiore alla prima. In seguito possono essere implementate altre proprietà nello specifico di ogni sotto-categoria, che però non potevano essere implementate in Animal.

Per estendere in Kotlin quella che sarà la superclasse, si deve utilizzare la keyword open all'inizio della dichiarazione della classe stessa. La keyword open dovremo utilizzarla anche per i metodi della superclasse, perché poi verranno scaricati dalla superclasse alla sottoclasse (override). Nel nostro esempio, i metodi e le variabili della classe Animal passeranno a tutte le sue sottoclassi.

Vediamo un esempio scritto in codice:

```
open class Animale (val nome: String, var eta: Int){
    open fun stampaDati(){
        println("nome: $nome, età: $eta")
    }
}

class Cane(var velocita : Int, nome:String, eta: Int) : Animale(nome, eta){
    fun stampaInformazione(){
        stampaDati()
        println("velocità: $velocita")
    }
}

fun main() {
    val cane = Cane(40, "Fufi", 5)
    cane.stampaInformazione()//nome: Fufi, età: 5
                             //velocità: 40
}
```

In questo esempio andiamo a creare una classe Animale, in cui andiamo a dire nome ed età e andiamo a dichiarare un metodo stampaDati() con la keyword open, così come è stato per la classe Animale. Il metodo stampaDati() andrà a stampare il nome e l'età del nostro animale. In seguito, andiamo a definire la sottoclasse Cane, che estende la classe Animale. Nella classe Cane dichiaro un altro metodo stampaInformazione(), che chiama il metodo stampaDati(), avendo così di nuovo la stampa del nome e l'età dell'animale, più la sua velocità. Infine, nel main() ho istanziato un oggetto della classe Cane e, chiamando anche il suo metodo, otterrò la stampa a video di nome, età e velocità del cane istanziato.

Vediamo un altro esempio di ereditarietà con la classe Persona:

```
open class Persona3(var nome: String, var eta: Int) {

    open fun stampaDati(){
        println("nome: $nome, età: $eta")
    }

    open fun mangiare(){
        println("sto mangiando...")
    }

    open fun dormire(){
        println("sto dormendo...")
    }
}

class Calciatore(nome: String, eta: Int, var club: String) : Persona3(nome, eta)
{

    fun giocare(){
        println("sto giocando per il $club club...")
    }
}

class Insegnante(nome: String, eta: Int, var materia: String) : Persona3(nome,
eta){

    fun insegnare(){
        println("sto insegnando $materia ...")
    }
}

fun main(){
    val calciatore = Calciatore("Giammarco", 25, "Milan")
    calciatore.stampaDati()//nome: Giammarco, età: 25
    calciatore.giocare()//sto giocando per il Milan club...
    calciatore.mangiare()//sto mangiando...
    calciatore.dormire()//sto dormendo...

    val insegnante = Insegnante("Anna", 46, "Matematica")
    insegnante.stampaDati()//nome: Anna, età: 46
    insegnante.insegnare()//sto insegnando Matematica ...
    insegnante.mangiare()//sto mangiando...
    insegnante.dormire()//sto dormendo...
}
```

La logica è la stessa dell'esempio precedente.

Riprendendo l'esempio di prima con la classe Persona, vediamo in concetto di override:

```
open class Persona4(var nome: String, var eta: Int) {

    open fun stampaDati(){
        println("nome: $nome, età: $eta")
    }

    open fun mangiare(){
        println("sto mangiando...")
    }

    open fun dormire(){
        println("sto dormendo...")
    }
}

class Calciatore2(nome: String, eta: Int, var club: String) : Persona4(nome,
eta) {

    fun giocare() {
        println("sto giocando per il $club club...")
    }

    override fun mangiare() {
        super.mangiare()
        println("$nome sta mangiando")
    }

    override fun dormire() {
        super.dormire()
        println("$nome sta dormendo")
    }

}

class Insegnante2(nome: String, eta: Int, var materia: String) : Persona4(nome,
eta){

    fun insegnare(){
        println("sto insegnando $materia ...")
    }

    override fun mangiare() {
        super.mangiare()
        println("$nome sta mangiando")
    }

    override fun dormire() {
        super.dormire()
        println("$nome sta dormendo")
    }
}

fun main(){
    val calciatore = Calciatore2("Giammarco", 25, "Milan")
    calciatore.stampaDati()//nome: Giammarco, età: 25
    calciatore.giocare()//sto giocando per il Milan club...
```

```
    calciatore.mangiare()//sto mangiando...
                         //Giammarco sta mangiando
    calciatore.dormire()//sto dormendo...
                        //Giammarco sta dormendo

    val insegnante = Insegnante2("Anna", 46, "Matematica")
    insegnante.stampaDati()//nome: Anna, età: 46
    insegnante.insegnare()//sto insegnando Matematica ...
    insegnante.mangiare()//sto mangiando...
                         //Anna sta mangiando
    insegnante.dormire()//sto dormendo...
                        //Anna sta dormendo
}
```

Dall'esempio possiamo notare che possiamo fare l'override dei metodi dalla superclasse alle sottoclassi. Per override si intende semplicemente riscrivere un determinato metodo. Quando effettuiamo l'override dei metodi e ci compare la keyword super, tale keyword richiama la superclasse creando una sua istanza, come se stessimo facendo Persona.metodo(). Quando nel main() vado a richiamare i due metodi mangiare() e dormire() su cui ho fatto l'override, essi restituiranno due output a video, perché il compilatore va prima ad eseguirli nella superclasse e poi anche nella sottoclasse in cui sono stati reimplementati. Se non vogliamo che venga eseguito anche il corpo del metodo all'interno della superclasse, possiamo semplicemente rimuovere la keyword super e il compilatore eseguirà solo il corpo del metodo nella sottoclasse.

TRACCIA 6 PER ESERCITAZIONE

Per poter completare questo esercizio dovrai:

- Determina se il triangolo è equilatero, isoscele o scaleno
- Equilatero = ha tutti i lati della stessa lunghezza
- Isoscele = ha almeno 2 lati della stessa lunghezza
- Scaleno = ha tutti i lati differenti uno dall'altro
- Crea una classe per risolvere questo esercizio

```
class Triangolo(var lato1: Int, var lato2: Int, var lato3: Int) {

    fun stampaDati(){
        println("primo lato= $lato1 cm, secondo lato= $lato2 cm, terzo lato=
$lato3 cm")
    }

    fun controlloTipoTriangolo(){
        if(lato1 != lato2 && lato1 != lato3 && lato2 != lato3){
            println("Hai costruito un triangolo scaleno")
        }else if (lato1 == lato2 && lato1 == lato3 && lato2 == lato3){
            println("Hai costruito un triangolo equilatero")
        }else if (lato1 != lato2 && lato1 != lato3 && lato2 == lato3){
            println("Hai costruito un triangolo isoscele")
        }
    }
}

fun main() {
    print("Lunghezza base triangolo: ")
    var base:Int = readLine()!!.toInt()
    print("Lunghezza primo lato obliquo: ")
    var latoObliquo1:Int = readLine()!!.toInt()
    print("Lunghezza secondo lato obliquo: ")
    var latoObliquo2:Int = readLine()!!.toInt()
    var triangolo = Triangolo(base, latoObliquo1, latoObliquo2)
    println(triangolo.stampaDati())
    println(triangolo.controlloTipoTriangolo())
}
```

INTERFACCE, CLASSI ASTRATTE E DATA CLASS

MODIFICATORI DI VISIBILITÀ

I modificatori di visibilità servono per poter gestire l'accessibilità di variabili e metodi all'interno di una o più classi, o anche all'interno del package. Nel momento in cui non assegniamo nessun modificatore ad una variabile o ad un metodo, il compilatore assegna in automatico il modificatore public.

Vediamo ora di capire la differenza tra i vari modificatori di visibilità:

Access Modifiers	Within the class	Within the Same Package	Subclass	In Other Packages
PUBLIC	Access Allowed	Access Allowed	Access Allowed	Access Allowed
PROTECTED	Access Allowed	Access Allowed	Access allowed	Access Denied
PRIVATE	Access Allowed	Access Denied	Access Denied	Access Denied

Public è il modificatore che ci permette di accedere ai componenti di una classe da qualunque parte. Esso infatti è il modificatore di visibilità più aperto in assoluto.

Private è il contrario di public, cioè è il modificatore di visibilità più chiuso in assoluto. Quando dichiariamo il componente di una classe private, è possibile accedervi solo all'interno della classe stessa.

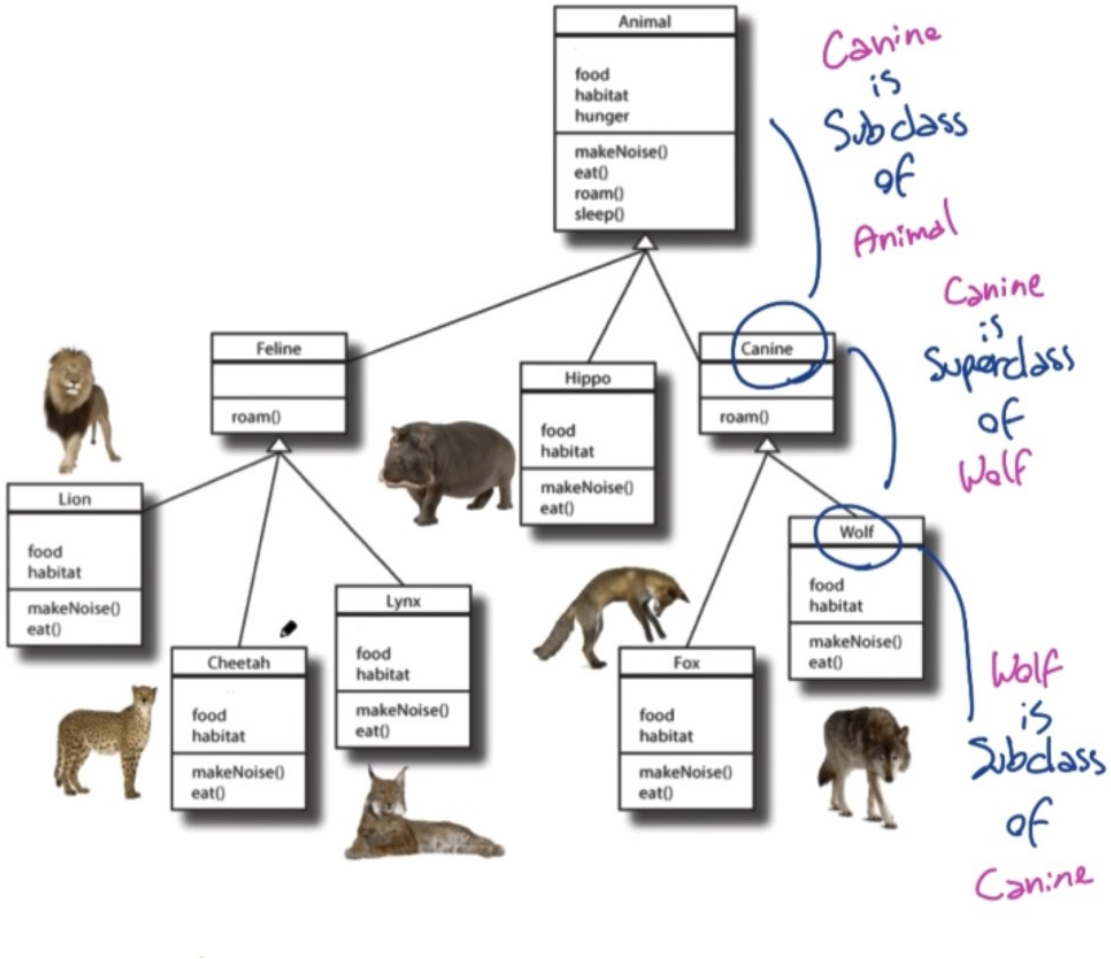
Protected è al secondo posto come vincoli di ristrettezza, perché permette di accedere al componente di una classe da qualunque parte, anche dalle sottoclassi, tranne nel momento in cui ci dovessimo trovare in un altro package.

Vediamo un esempio:

```
open class ModificatoriVisibilita {

    private var nome = "Giovanni"
    protected var cognome = "Rossi"
    var age = 19
}

class sottoclasseModificatoriVisibilita : ModificatoriVisibilita(){
    fun informazioni(){
        //super.nome //Cannot access 'nome': it is private in
'ModificatoriVisibilita'
        super.cognome
        super.age
    }
}

fun main() {

    val modificatoreVisibilita = ModificatoriVisibilita()
    modificatoreVisibilita.age
    //modificatoreVisibilita.nome //Cannot access 'nome': it is private in
'ModificatoriVisibilita'
    //modificatoreVisibilita.cognome //Cannot access 'cognome': it is protected
in 'ModificatoriVisibilita'

}
```

CLASSI ASTRATTE

Per capire cosa sono le classi astratte, possiamo riprendere un esempio già visto qualche paragrafo fa sugli animali:



Riprendiamo questo esempio per chiederci se abbia senso o meno avere un'istanza delle classi Animale, Canino e Felino. La risposta a questo quesito è che non è possibile avere istanze per queste classi, perché possiamo creare istanze solo di classi per cui esiste una loro rappresentazione concreta nella realtà. Quindi, nella vita reale possiamo avere l'esemplare di un leone, leopardo, lince, ippopotamo, volpe e lupo, ma non possiamo avere un esemplare denominato con il nome della loro categoria, in quanto è solo una rappresentazione astratta per poter classificare dei comportamenti e caratteristiche. Ragion per cui, è bene definire le classi Canino, Felino e Animale come classi astratte in fase di dichiarazione.

Come sappiamo, programmare bene vuol dire scrivere un codice robusto, che non permetta di creare cose che non si dovrebbero creare. Le classi astratte intervengono proprio per questo motivo, in quanto dichiarando una classe astratta in Kotlin con la keyword abstract, il compilatore ci restituirà errore nel momento in cui dovessimo andare a creare un'istanza per quella classe.

Vediamo un esempio di classe astratta:

```
abstract class Animale(var cibo : String){
    open fun mangiare(){
        println("sto mangiando $cibo")
    }
}

class Leone(var nome : String, cibo: String) : Animale(cibo){
    fun info(){
        println("Il mio nome è $nome il Re")
    }

    public override fun mangiare() {
        super.mangiare()
    }
}


fun main() {
    //var animale = Animale("Simba")//Cannot create an instance of an abstract
class
    var leone = Leone("Simba", "Cinghiali")
    leone.info()//Il mio nome è Simba il Re
    leone.mangiare()//sto mangiando Cinghiali
}
```

Nell'esempio possiamo vedere come non ci sia bisogno di scrivere la keyword open sulla dichiarazione della classe astratta Animale, perché il compilatore inserisce tale keyword in fase di esecuzione, in quanto si sa già che la classe astratta è fatta per poter essere estesa da altre classi. Questo però non vale anche per i metodi della classe astratta, che invece devono essere sempre accompagnati dalla keyword open per essere riutilizzati dalle sottoclassi.

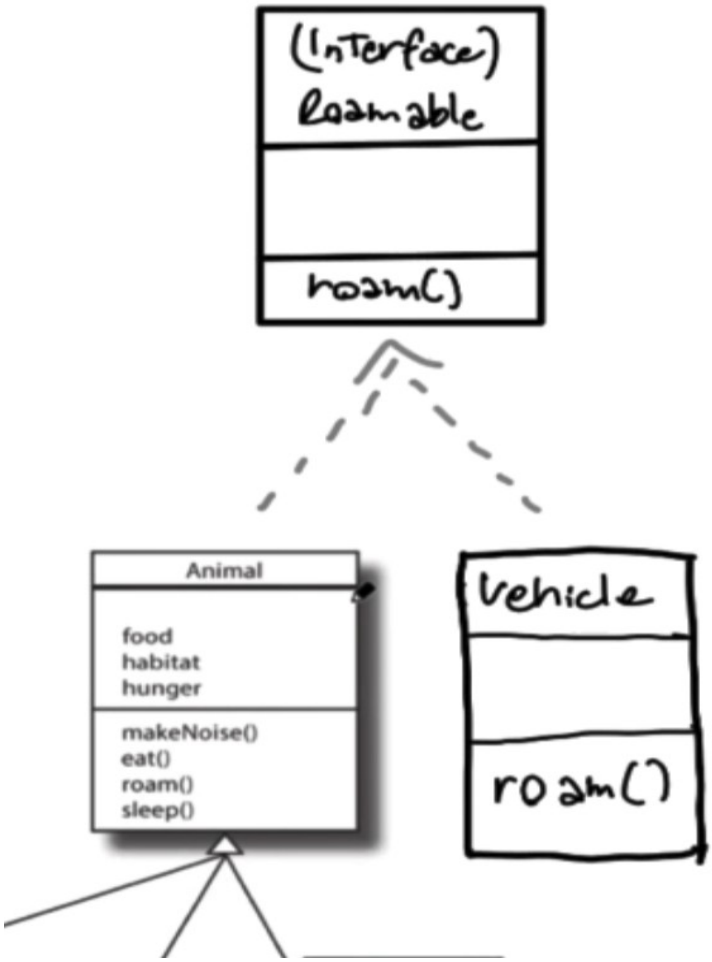
Quando abbiamo un progetto molto grande, andare ad utilizzare classi astratte e modificatori di visibilità garantisce il fatto di prevenire molti problemi che avremmo se non li utilizzassimo. Quindi, è importante stabilire tutto ciò ancor prima di andare a scrivere codice strada facendo.

Comunque non è detto che tutte le superclassi debbano essere per forza astratte, ma per capirlo basta porsi il quesito sul fatto che per tale classe abbia senso o meno creare un'istanza. Nel caso in cui non ci sia il senso di creare un'istanza per quella superclasse, la possiamo dichiarare astratta.

INTERFACCE

Le interfacce non sono delle vere e proprie classi, ma sono delle componenti a parte.

Le interfacce permettono di definire un comportamento (metodo) comune, al dì fuori della gerarchia della superclasse. Vediamo un esempio:



Nell'esempio possiamo vedere come le classi Animale e Veicolo siano del tutto scollegate tra di loro a livello di concetto, ma possono avere dei comportamenti in comune, come gironzolare().

Quindi, le interfacce ci permettono di andare a creare del codice robusto, creando un comportamento comune che possiamo andare a estendere in diverse classi.

Un altro concetto importante è che una classe può estendere (implementare) più interfacce, a differenza delle classi che possono estendere solo un'altra classe (possiamo avere una superclasse). Questo ha senso perché una classe può avere più comportamenti.

Quando andiamo ad estendere un'interfaccia, la classe che la estende deve obbligatoriamente anche estendere tutti i metodi di quell'interfaccia. Quindi se creo un'interfaccia con diversi metodi, nel momento in cui vado ad implementare quell'interfaccia all'interno di una classe, dovrò andare a implementare anche tutti i metodi che la classe ha ereditato dalla mia interfaccia.

Le interfacce, non essendo classi, non hanno costruttori e i metodi al loro interno possono non avere corpo.

Vediamo un esempio:

```
interface A{
    fun miaFunzione(){
        println("Sono la funzione dell'interfaccia A")
    }
}

interface B{

    fun miaFunzione(){
        println("Sono la funzione dell'interfaccia B")
    }
}

class X: A,B{
    override fun miaFunzione() {
        super<A>.miaFunzione()//super in questo caso serve per indicare a quale
interfaccia ci riferiamo
        super<B>.miaFunzione()

        //blocco di codice...
    }

}

fun main() {
    val x = X()
    x.miaFunzione()//Sono la funzione dell'interfaccia A
                   //Sono la funzione dell'interfaccia B
}
```

INTERFACCE E POLIMORFISMO

Nell'esempio di seguito vedremo come creare un array di elementi che sono concettualmente diversi tra di loro, ma che hanno una funzione in comune:

```
interface Gironzolando {

    fun gironzolare()//funzione astratta
}

class Macchina(val modello: String): Gironzolando{

    fun info(){
        println("Il modello della macchina è $modello")
    }
    override fun gironzolare() {
        println("La macchina sta gironzolando")
    }
}

abstract class Animal(val cibo: String): Gironzolando{

    open fun mangiare() {
        println("L'animale mangia $cibo")
    }

    override fun gironzolare() {
        println("L'animale sta gironzolando")
    }
}

class Lyon(val nome: String, cibo: String): Animal(cibo){

    fun info(){
        println("Il mio nome è $nome il Re")
    }

    override fun mangiare() {
        super.mangiare()
    }

    override fun gironzolare() {
        super.gironzolare()
    }
}

class Hyppo(val nome: String, cibo: String): Animal(cibo){

    fun info(){
        println("Il mio nome è $nome")
    }

    override fun mangiare() {
        super.mangiare()
    }

    override fun gironzolare() {
        super.gironzolare()
```

```
    }
}

fun main() {
    val gironzolanti = arrayOf(Hyppo("Samanta", "pesce"),
                               Lyon("Alex","carne"),
                               Macchina("Ford"))
    for (elemento in gironzolanti){
        elemento.gironzolare()//L'animale sta gironzolando
                              //L'animale sta gironzolando
                              //La macchina sta gironzolando
        if (elemento is Animal){//Con is chiedo se un oggetto è istanza di una
determinata classe
                                //Corrisponde all' instanceof di JAVA

            elemento.mangiare()//"L'animale mangia pesce" in riferimento
all'oggetto di Hyppo
                               //"L'animale mangia carne" in riferimento
all'oggetto di Lyon
        }
    }
}
```

FUNZIONI ENUM

Le funzioni enum sono delle interfacce utilizzate per creare un elenco di costanti, riguardanti una stessa categoria. Possiamo pensare come esempi i punti cardinali che non possono cambiare, come anche agli strumenti di una band.

Vediamo ora un esempio con il codice:

```
enum class Band(val strumento : String){
    MARCO("chitarra"){
        override fun tipoDiMusica() = "rock"
                    },
    GIANNI("basso"){
        override fun tipoDiMusica() = "metal"
                  },
    LAURA("voce"){
        override fun tipoDiMusica() = "classico"
    };

    open fun tipoDiMusica() = "pop"
}

fun main(){
    println(Band.MARCO)//MARCO
    println(Band.MARCO.strumento)//chitarra
    println(Band.MARCO.tipoDiMusica())//rock
    println(Band.GIANNI)//GIANNI
    println(Band.GIANNI.strumento)//basso
    println(Band.GIANNI.tipoDiMusica())//metal
    println(Band.LAURA)//LAURA
    println(Band.LAURA.strumento)//voce
    println(Band.LAURA.tipoDiMusica())//classico
}
```

Nell'esempio, MARCO, GIANNI E MIRCO non sono assolutamente delle istanze, ma sono delle variabili contenenti un valore costante. Tali variabili devono essere sempre chiamate con nomi in lettera maiuscola, regola che vale anche in generale quando una variabile deve contenere una costante.

DATA CLASS

Le data class sono classi simili alle classi normali, ma che ci mettono a disposizione dei metodi in più e sono, la maggior parte delle volte, sprovviste di un corpo. Vediamo un esempio per capire la loro utilità:

```
data class Macchina_(var proprietario: String, val modello: String, var
annoRilascio: Int)

class Umano(var name: String, var eta: Int)

data class Camion(var proprietario: String = "Stefano",
                  val modello: String = "Toyota",
                  var annoRilascio: Int = 2001)

fun main() {

    val umano1 = Umano("Marco",20)
    val umano2 = Umano("Marco",20)
    println(umano1 == umano2)//false
    println(umano1)//interfacce_e_classi_astratte.Umano@29453f44
    println(umano2)//interfacce_e_classi_astratte.Umano@5cad8086

    val umano3 = Umano("Marco",20)
    val umano4 = umano3
    println(umano3 == umano4)//true
    println(umano3)//interfacce_e_classi_astratte.Umano@6e0be858
    println(umano4)//interfacce_e_classi_astratte.Umano@6e0be858

    val macchina1 = Macchina_("Marco","Ford", 2012)
    val macchina2 = Macchina_("Marco","Ford", 2012)
    println(macchina1 == macchina2)//true
    println(macchina1)//Macchina_(proprietario=Marco, modello=Ford,
annoRilascio=2012)
    println(macchina2)//Macchina_(proprietario=Marco, modello=Ford,
annoRilascio=2012)

    var macchina3 = Macchina_(proprietario = "Marco", annoRilascio = 2012,
modello = "Ford")
    println(macchina3)//Macchina_(proprietario=Marco, modello=Ford,
annoRilascio=2012)

    var macchina4 = macchina3.copy(modello = "Ferrari")
    println(macchina4)//Macchina_(proprietario=Marco, modello=Ferrari,
annoRilascio=2012)
}
```

Nell'esempio abbiamo una data class Macchina e una classe normale Umano. Iniziando con l'istanziare due oggetti della classe Umano, che saranno del tutto identici per quanto riguarda i valori al loro interno, possiamo notare come comparando il loro riferimenti con un'uguaglianza, il compilatore ci restituisca comunque false. Il motivo di ciò è che il compilatore assegna dietro le quinte un id diverso ad ogni oggetto istanziato in memoria, per cui se facciamo l'uguaglianza tra due riferimenti contenenti rispettivamente i due oggetti, stiamo paragonando i loro id che saranno per forza di cose diversi. Per ottenere il true da un'uguaglianza tra due riferimenti, dobbiamo istanziare un solo oggetto, come nel caso di umano3, e poi assegnare il primo riferimento al secondo, come nel caso di umano4. Solo in questo modo, siamo sicuri che due riferimenti hanno lo stesso oggetto. Questo che stiamo facendo ora, corrisponde a fare equals() in JAVA. Nel caso degli oggetti della data class, invece, possiamo notare che, istanziando due oggetti con lo

stesso contenuto e facendo l'uguaglianza tra i loro due riferimenti, la loro uguaglianza restituisce true. Questo perché il compilatore non fa l'uguaglianza tra gli id di oggetti che derivano da una data class, ma solo dei loro contenuti.

Le data class ci permettono anche di mischiare i parametri di un oggetto, come nel caso di macchina3, senza dover obbligatoriamente seguire l'ordine dei parametri che stanno nel costruttore della classe.

La data class introduce anche il metodo copy() per copiare un oggetto da un altro. Questo però non nega il fatto che si possono effettuare delle modifiche sull'oggetto copiato.

La data class ci permette anche di definire dei valori di default ai parametri del suo costruttore, come nel caso della data class Camion.

TRACCIA 7 PER ESERCITAZIONE

Per poter completare questo esercizio dovrai:

- cerca di utilizzare tutti i componenti che abbiamo visto all'interno di questa sezione: Interfacce, Ereditarietà, Classi Astratte e Enumerazioni.
- creare una superclasse `Polygon`
- creare una sottoclasse `Square` (dovrai gestire anche il rettangolo dal momento che ha 4 lati anche lui)
- ogni classe dovrà definire le funzioni: `perimeter` e `area` . Dovremmo essere quindi in grado di calcolare il perimetro e l'area delle nostre figure geometriche
- ogni classe dovrà anche definire un metodo `shape`. Questa funzione può essere richiamata su qualsiasi figura e dovrà essere in grado di capire di che tipo di poligono si tratta.

```
interface CalcoloAreaPerimetro {

    fun calcoloArea()
    fun calcoloPerimetro()
}

abstract class Poligono (val lato : Double): CalcoloAreaPerimetro{

    override fun calcoloArea() {
    }

    override fun calcoloPerimetro() {
    }
}

class Quadrato(lato: Double) : Poligono(lato){

    override fun calcoloArea() {
        super.calcoloArea()
        var area = lato * 4
        println("L'area del quadrato è $area")
    }

    override fun calcoloPerimetro() {
        super.calcoloPerimetro()
        var perimetro = lato * lato
        println("Il perimetro del quadrato è $perimetro")
    }
}

class Rettangolo(lato: Double, val altezza: Double) : Poligono(lato){

    override fun calcoloArea() {
        super.calcoloArea()
        var area = lato * altezza
        println("L'area del rettangolo è $area")
    }
```

```
    override fun calcoloPerimetro() {
        super.calcoloPerimetro()
        var perimetro = lato + lato + altezza + altezza
        println("Il perimetro del rettangolo è $perimetro")
    }
}

fun main() {
    var numero = 0

    fun ControlloNumeroNegativo(numero: Double): Boolean{
        var numeroNegativo = false
        if(numero < 0){
            println("Il numero che stai inserendo non può essere negativo")
            numeroNegativo = true
        }else{
            numeroNegativo = false
        }
        return numeroNegativo
    }

    var max = 0
    for(i in 0 .. max){
        println("""Voglio area e perimetro di: SELEZIONA UN' OPZIONE
            |1: Quadrato
            |2: Rettangolo
            |3: Per uscire dal programma
        """.trimMargin())
        var opzione:String = readLine()!!.toString()
        when(opzione){
            "1" -> { print("Indicare la lunghezza del lato del quadrato: ")
                var lato:Double = readLine()!!.toDouble()
                var quadrato = Quadrato(lato)
                var numeroNegativo = ControlloNumeroNegativo(lato)
                if (numeroNegativo){
                    println("Prova di nuovo!")
                }else{
                    quadrato.calcoloArea()
                    quadrato.calcoloPerimetro()
                }

            }
            "2" -> {print("Indicare la lunghezza della base del rettangolo: ")
                var base:Double = readLine()!!.toDouble()
                var rettangolo= Rettangolo(base, 0.0)
                var numeroNegativo = ControlloNumeroNegativo(base)
                if (numeroNegativo) {
                    println("Prova di nuovo!")
                }else{
                    print("Indicare la lunghezza dell'altezza del rettangolo: ")
                    var altezza:Double = readLine()!!.toDouble()
                    rettangolo = Rettangolo(base, altezza)
                    var numeroNegativo2 = ControlloNumeroNegativo(altezza)
                    if (numeroNegativo2) {
                        println("Prova di nuovo!")
                    }else{
                        rettangolo.calcoloArea()
```

```
                        rettangolo.calcoloPerimetro()
                    }
                }
            }
            "3" -> {break
            }
            else -> { println("Opzione non valida!")}
        }
        println()
        max++;
    }
}
```

OBJECT

Gli object in Kotlin sono la stessa cosa dei singleton in JAVA. Per capire cosa sono i singleton, immaginiamo di avere un'applicazione che registra l'audio e, come sappiamo, tale applicazione può usare il microfono del dispositivo su cui è installato solo se non è occupato già da un'altra applicazione. Possiamo capire, quindi, che è possibile avere solo un'istanza della classe Registratore avendo un solo microfono a disposizione, perché nel caso in cui andassimo a creare più istanze, può essere che il nostro metodo registra() venga richiamato più volte simultaneamente, causando problemi o bug.

Ragion per cui, se nella fase di creazione di una classe ci accorgiamo che essa non ha bisogno di più di una istanza, allora possiamo utilizzare gli object. La classe object non può avere costruttore e, allo stesso tempo, non possiamo creare sue istanze, perché sarà il compilatore in automatico a creare l'unica istanza che può avere. Vediamo un esempio in cui andiamo a creare il logo di una compagnia:

```
import java.time.Year

object DirittiRiservatiCompagnia{
    val annoCorrente = Year.now().value

    fun getTagLine() = "Computer S.R.C. Company"
    fun getCopyrightLine() = "Copyright \u00A9$annoCorrente Our Company. All
rights reserved"
}

fun main() {
    println(DirittiRiservatiCompagnia.getTagLine())
    println(DirittiRiservatiCompagnia.getCopyrightLine())
}
```

COMPANION OBJECT

Possiamo dire che l'object è una sorta di estensione del companion object. Le companion object sono la stessa cosa del modificatore static in JAVA. Quando in JAVA definiamo una variabile o un metodo statico, vuol dire che tali membri sono associati alla classe e non alla istanza. Ragion per cui, un membro statico può essere utilizzato anche se non esiste nessuna istanza della classe che li contiene, ma basta richiamare la classe stessa. La stessa logica vale per tutti i metodi e variabili dichiarate all'interno delle graffe della companion object. Per richiamare un metodo "statico" in Kotlin, quindi non ci sarà bisogno di istanziare un oggetto, ma basterà richiamare la classe con il suo metodo a seguire. Per capire quando usare la companion object, basta chiedersi se è necessario creare un oggetto per il metodo che vogliamo definire. Nel caso in cui non sia necessario, allora usiamo le companion object. Ultima cosa da notare è che tutti i membri all'interno della companion object posso interagire tra di loro, ma non possono interagire con i membri all'esterno. Vediamo un esempio:

```
class QualcheClasse private constructor(val string: String) {

    companion object{
        var variabileStatica = 6 //variabile statica

        fun assegnaStringa(str: String) = QualcheClasse(str)

        fun lettereMaiuscoleMinuscole(str: String, lowerCase: Boolean) :
QualcheClasse{
            return if (lowerCase){
                QualcheClasse(str.uppercase())
            }else{
                QualcheClasse(str.lowercase())
            }
        }
    }
}

fun main() {
    val classe1 = QualcheClasse.assegnaStringa("questa è una stringa")
    val classe2 = QualcheClasse.lettereMaiuscoleMinuscole("Questa É Una clAsse",
true)
    val classe3 = QualcheClasse.lettereMaiuscoleMinuscole("Questa É Una clAsse",
false)
    println(classe1.string)//questa è una stringa
    println(classe2.string)//QUESTA É UNA CLASSE
    println(classe3.string)//questa é una classe
    println(QualcheClasse.variabileStatica)//6
}
```

Nell'esempio abbiamo il costruttore di tipo private per evitare di poter creare istanze della classe QualcheClasse.

OBJECT EXPRESSION

Un'altra cosa che gli object possono fare è quella di poter assegnare una classe anonima (cioè senza nome, ma chiamata semplicemente object) ad una variabile. Vediamo un esempio per capire questo concetto:

```
fun main() {
    val coordinate = object {
        var x = 10
        var y = 10

    }
    println(coordinate.x)//10
}
```

La classe object nell'esempio è comunque una classe singleton, però anonima proprio perché non ha nome. Per stampare qualcosa all'interno di object, come nel caso del singleton, possiamo semplicemente richiamare la classe (che in questo caso sarà il riferimento che contiene object) con il nome del membro che lo segue, senza dover istanziare alcun oggetto.

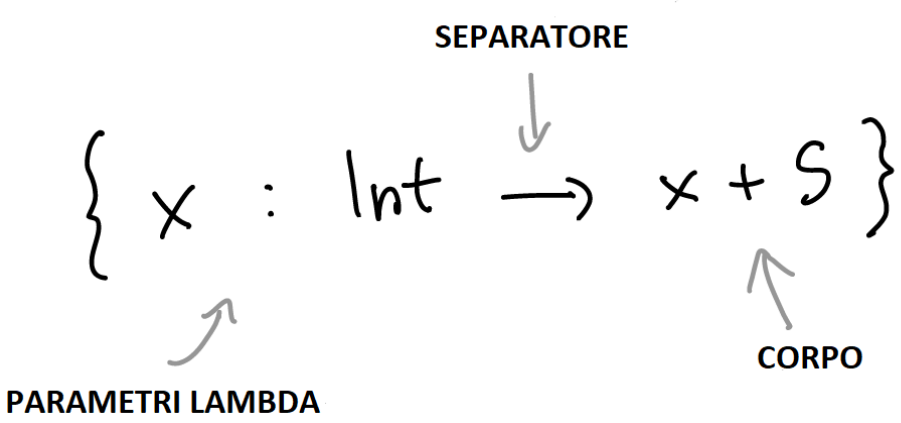
Possiamo usare anche la classe anonima come estensione della superclasse:

```
open class Persona{
    fun mangiare() = println("sto mangiando")

    open fun pregare() = println("sto pregando")
}

fun main() {

    val persona = object : Persona() {
        override fun pregare() {
            println("non sto pregando")
        }
    }


    persona.mangiare()//sto mangiando
    persona.pregare()//non sto pregando

}
```

Da questo esempio possiamo subito vedere come dalla classe anonima abbiamo accesso ai metodi della superclasse Persona.

LAMBDA

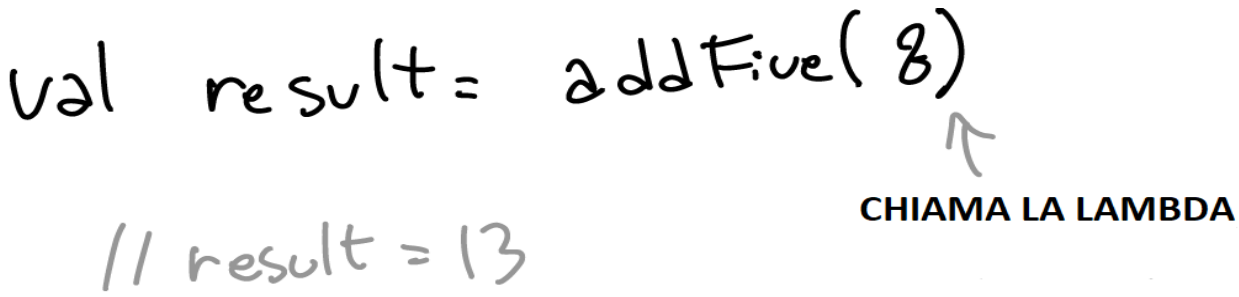
Possiamo vedere le lambda come delle funzioni anonime, un po come gli object che erano però delle classi anonime. Un esempio di espressione lambda la possiamo vedere di seguito:



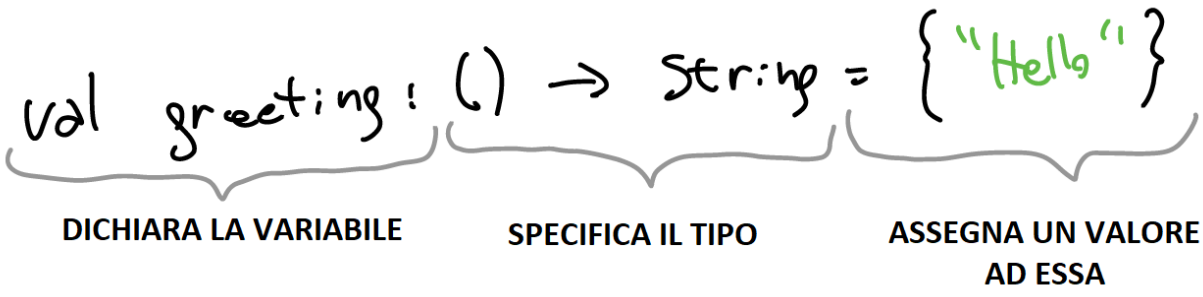
Un'espressione lambda è sempre delimitata da due parentesi graffe, in cui all'interno ci troviamo uno o più parametri che useremo per richiamare l'espressione stessa all'esterno, il corpo e un separatore tra corpo e parametri.

Possiamo andare ad associare l'espressione lambda ad una variabile:

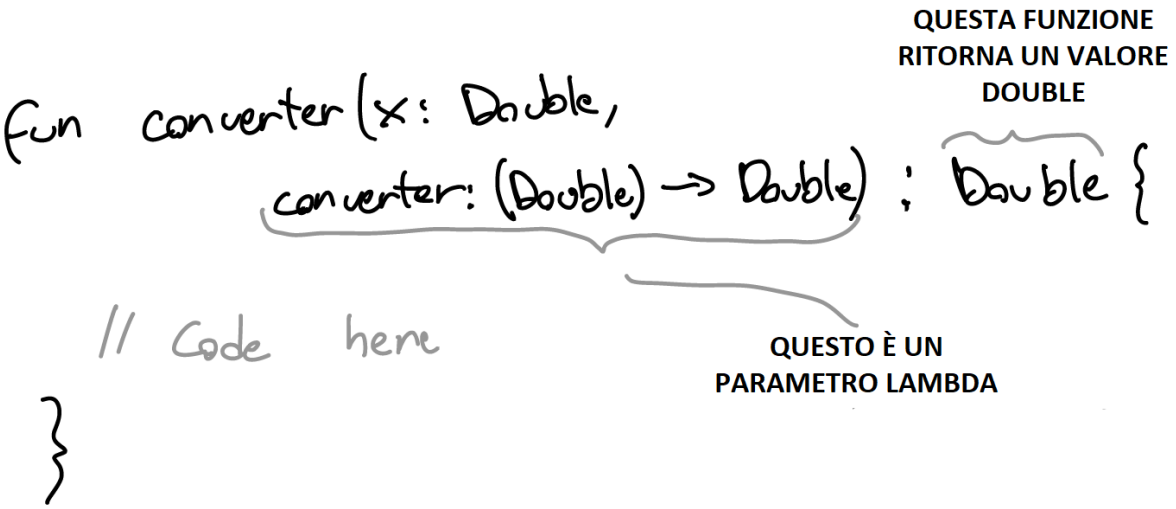
Dopodiché, possiamo utilizzare sempre quella variabile per richiamare l'espressione lambda che contiene:



È possibile anche scrivere un'espressione lambda senza parametri, ma solo con il corpo che ci ritorna un valore:



Possiamo utilizzare le lambda anche come parametro di un metodo vero e proprio. È un po come avere un metodo dentro un altro metodo:



Per richiamare la funzione precedente, faremo in questo modo:

Vediamo un primo esempio di codice sulle espressioni lambda che conta il numero di caratteri in una parola mediante un metodo già implementato dagli sviluppatori di Kotlin, ossia count():

```
fun main() {
    val numeroDiLettere = "Mississippi".count()
    println(numeroDiLettere)//11

    val numeroDiLettereSMinuscole = "Mississippi".count{ letter ->
        letter == 's'}
    println(numeroDiLettereSMinuscole)//4
}
```

Quando l'espressione lambda è composta da un solo parametro, è possibile utilizzare la keyword it in sostituzione del codice di prima, nel seguente modo:

```
fun main() {
    val numeroDiLettere = "Mississippi".count()
    println(numeroDiLettere)//11

    val numeroDiLettereSMinuscole = "Mississippi".count{ it == 's'}
    println(numeroDiLettereSMinuscole)//4
}
```

La keyword it, come possiamo vedere, riduce la quantità di codice da scrivere.

Vediamo altri esempi di utilizzi base di lambda:

```
fun main() {

    val print = { println("Hello")}//espressione lambda senza parametri
    print()

    val printString: (String) -> Unit = {println("Hello $it")}//espressione
lambda con un parametro
    printString("Mattia")//Hello Mattia

    val aggiungiCinque: (Int) -> Int = {it + 5}//espressione lambda con un
parametro
    //val aggiungiCinque= { number : Int -> number + 5 }//equivale alla lambda
precedente
    println(aggiungiCinque(8))//13

    val calcolare = { x : Double, y : Double -> x + y}////espressione lambda con
due parametri
    println(calcolare(5.0, 16.0))//21.0
}
```

LAMBDA E FUNZIONI

Le lambda possono essere usate come parametri di funzioni è anche come loro valore di ritorno. Vediamo un esempio in cui sono presenti entrambi i casi:

```
fun main() {
    fun converteDaCentigradoAFahrenheit( numeroDaConvertire : Int, convertitore:
(Int) -> Double) : Double{
        val risultato = convertitore(numeroDaConvertire)
        println("$numeroDaConvertire è stato convertito in $risultato")
        return risultato
    }

    converteDaCentigradoAFahrenheit(5, {it * 1.8 + 35})//5 è stato convertito in
44.0

    fun getConversione(string : String): (Double) -> Double{

        when(string){
            "DaCentigradoAFahrenheit" -> return {it * 1.8 + 32}
            "DaKgALibbre" -> return {it * 2.204623}
            else -> return {it}
        }
    }
    val numero = getConversione("DaKgALibbre")(85.0)
    println(numero)//187.39295500000003
}
```

SCOPING FUNCTION

Sono delle espressioni lambda che ci mette a disposizione Kotlin per poter scrivere meno righe di codice. Le scoping function sono utilizzate molto sulle stringhe (T.also{ } e T.let{ }) e sugli oggetti da istanziare (T.apply{ }, T.run{ }, with(T) { }), per evitare di dover richiamare tante volte il nome del riferimento della classe. Vediamo un esempio:

```
class Persona{
    var nome : String = "Marco"
    var numeroTelefono : String = "123456789"
    fun displayInfo() = println("\nNome: $nome\nNumero di telefono:
$numeroTelefono")
}


fun main() {

    //Metodo classico per istanziare un oggetto
    val amico = Persona()
    amico.nome = "Clara"
    amico.numeroTelefono = "987654321"
    amico.displayInfo()//Nome: Clara
                       //Numero di telefono: 987654321

    //Metodo con T.run{ } per istanziare un oggetto
    Persona().run{ //Per gestire i tipi null: Persona()?.run{
        nome = "Clara"
        numeroTelefono = "987654321"
        displayInfo()//Nome: Clara
                     //Numero di telefono: 987654321
    }

    //Metodo con with(T) { } per istanziare oggetto, ma sconsigliato per gestire
i tipi null
    with(Persona()){
        nome = "Clara" //Per gestire i tipi null: this?.nome = "Clara"
        numeroTelefono = "987654321" //Per gestire i tipi null:
this?.numeroTelefono = "987654321"
        displayInfo()//Nome: Clara
                     //Numero di telefono: 987654321
    }

    //Metodo con T.apply{ } per istanziare oggetto e anche il più utilizzato
    val persona = Persona().apply{
        nome = "Clara" //Nome: Clara
        numeroTelefono = "987654321" //Numero di telefono: 987654321
    }
    persona.displayInfo()

    //Metodo con T.let{ } per gestire le stringhe
    val string = "abc"
    string.let {
        println("\nLa stringa originale è $it")//La stringa originale è abc
        it.reversed()
    }.let {
        println("La stringa invertita è $it")//La stringa invertita è cba
    }/* let considera sempre lo stato della stringa precedente,
        per cui il secondo let prende la stringa invertita*/
```

```
    //Metodo con T.also{ } per gestire le stringhe
    string.also {
        println("\nLa stringa originale è $it")//La stringa originale è abc
        it.reversed()
    }.also {
        println("La stringa invertita è $it")//La stringa invertita è abc
    }/* also considera sempre lo stato della stringa principale,
        per cui il secondo also non prende la stringa invertita*/

}
```

GENERIC TYPE

I generic type li abbiamo già visti con le collezioni. È il tipo racchiuso tra le parentesi angolari nella dichiarazione delle collezioni e rappresenta il tipo di dato che le collezioni stesse possono contenere.

Usando i generic type delle collezioni, è possibile utilizzare un singola funzione che opera su più collezioni, anche se hanno il generic type diverso. Vediamo un esempio:

```
fun main() {

    val nomeArray = arrayListOf<String>("Marco", "Giulia", "John", "Giuseppe")
    stampaCollezione(nomeArray)//MarcoGiuliaJohnGiuseppe
    nomeArray.stampaCollezione2()//MarcoGiuliaJohnGiuseppe

    val numeroArray = arrayListOf<Int>(1,2,3,4,5)
    stampaCollezione(numeroArray)//12345
    numeroArray.stampaCollezione2()//12345
}

//Funzione classica
fun <T> stampaCollezione(collezione: ArrayList<T>) {
    for (item in collezione) {
        print(item)
    }
}

//Extension function
fun <T> ArrayList<T>.stampaCollezione2(){
    for (item in this){
        print(item)
    }
}
```

COVARIANCE

Siamo sempre nei generic type, ma per capire bene a cosa servono le covariance dobbiamo vedere un primo esempio:

```
fun main() {
    val listaDiShort: List<Short> = listOf(1, 2, 3, 4, 5)
    converteDaShortAInt(listaDiShort)//1 2 3 4 5
}

fun converteDaShortAInt(collezione: List<Number>){
    for(numero in collezione){
        print("${numero.toInt()} ")
    }
}
```

Nell'esempio sono partito con il creare una funzione che andrà a convertire una lista di elementi di tipo Short in Int. Anche se il generic type della lista inserita come parametro nella dichiarazione della funzione è di tipo Number, quando inserisco nella chiamata alla funzione una lista di tipo Short, il programma compila tranquillamente. Questo succede perché Number è la superclasse di Short, quindi è possibile inserire delle sottoclassi di un generic type nei parametri di una chiamata alla funzione.

Vediamo però cosa succede se dichiaro le liste Mutable:

```
fun main() {
    val listaDiShort: MutableList<Short> = mutableListOf(1, 2, 3, 4, 5)
    converteDaShortAInt(listaDiShort)//Type mismatch: inferred type is
MutableList<Short> but MutableList<Number> was expected
}

fun converteDaShortAInt(collezione: MutableList<Number>){
    for(numero in collezione){
        print("${numero.toInt()} ")
    }
}
```

In questo caso avremo un errore sul fatto che il generic type Number non è compatibile con il generic type Short, non permettendoci quindi di inserire una sottoclasse in una superclasse e creando un' incoerenza con ciò che è stato detto precedentemente. É qui che entrano in gioco le covariance, specificando nel nostro caso che le mutable list non sono covariance, mentre le liste normali lo sono. Definendo un costrutto covariance, otteniamo delle funzionalità in più, a discapito però di qualche svantaggio. Nel caso delle liste normali, ma delle collezioni in generale, quando andiamo a definire il suo contenuto in fase di dichiarazione, esso poi non potrà essere più modificato e questo fa parte degli svantaggi di un costrutto covariance. Ma il vantaggio che otteniamo in cambio è che, in fase di chiamata alla funzione, se inseriamo come parametro una collezione cui generic type è sottoclasse di quello della collezione inserita nella dichiarazione della funzione stessa, non abbiamo errore di compilazione. Nel caso di collezioni mutabili, invece, non essendo covariance, è possibile cambiare il loro contenuto anche dopo la loro dichiarazione, ma una volta inserite come parametro d'ingresso in una funzione, non ne possiamo cambiare il generic type.