# A.1 – Kotlin

E’ possibile provare il codice nel sito Kotlin Playground

<https://play.kotlinlang.org>

## Variabili

La keywork **val** identifica variabili *final*, ovvero read-only

val number: Int = 1

**var** indica invece variabili che possono essere riassegnate

var string: String = "hi!"

string = "hello!"

Kotlin capisce da solo il *Tipo* che si sta assegnando: viene quindi solitamente omesso nella scrittura a meno che non sia necessario per chiarificare il codice

val number = 1  
var string = "hi!"

Si può esplicitare il tipo di una variabile aggiungendolo dopo il nome:

val number: Int = 1  
var string: String = "hi!"

## Null-safety

Kotlin è un linguaggio *null-safe*, ovvero integra nella definizione dei tipi se possono essere null o no, utilizzando il punto interrogativo:

Graphical user interface, text

Description automatically generated

Di base si lavora con tipi *safe*, ovvero che non possono produrre null.

Nel caso di tipi nullabili, è possibile evitare di scrivere null check con IF ELSE, accedendo alla variabile tramite la nomenclatura con il punto interrogativo:



In questo esempio *lowercaseString* sarà anch’essa una String?, in quanto la funzione .lowercase() viene lanciata solo nel caso nullableString non sia null. Questo equivale a un check IF (null)

Per replicare IF(null) ELSE, è possibile utilizzare l’**Elvis Operator -** **?:**



In questo caso lowercaseString non sarà mai null, perché in caso nullableString sia null, verrà valorizzata a “default”

L’elvis operator si chiama così perché ricorda i capelli di elvis ?:)

## Classi, Data Classes

Le classi vengono dichiarate utilizzando la keywork *class*

class Person constructor(firstName: String) { /\* class content here \*/ }

Ma la keyword *constructor* del costruttore primario è omessa

class Person (firstName: String) { /\* class content here \*/ }

I parametri del costruttore possono essere utilizzati solo nelle assegnazioni delle *properties*. Assegnazioni e blocchi *init()* vengono eseguiti nell’ordine in cui sono scritti

class InitOrderDemo(name: String) {  
 val firstProperty = "First property: $name"  
  
 init {  
 *println*("init of $name")  
 }  
  
 val secondProperty = "Second property: ${name.length}"  
  
 init {  
 *println*("init of ${name.length}")  
 }  
}

Invece di assegnare manualmente i parametri alle *properties*, è possibile dichiararli come tali

class Person(  
 val firstName: String,  
 private val secretName: String,  
 var age: Int,   
) { /\*...\*/ }

E’ molto comune definire classi il cui unico scopo è mantenere dei dati. Kotlin ci fornisce le *data classes* che hanno delle facilitazioni, come funzioni di eguaglianza o di trasformazione in stringa leggibile. Affichè sia valida, una data class deve avere le proprietà dichiarate nel costruttore. Utilizzare una data class per definire i modelli è lo standard.

data class Person(  
 val firstName: String,   
 private val secretName: String,  
 var age: Int,   
) { /\*...\*/ }

## Funzioni, Costruttori

Le funzioni (metodi) hanno una struttura simile, utilizzando la keyword **fun**

Di base sono *public*, si può cambiare la visibilità con la keyword *private fun.*

fun performSum(first: Int, second: Int): Int {  
 return first + second  
}

val resultIsTen = performSum(3,7)

Possono essere dichiarate *inline*, ovvero il body può essere un’espressione. In questo caso il type è omesso.

fun performSum(first: Int, second: Int) = first + second

Quando sono membri di una classe, possono essere invocate con la struttura *dot notation*

MathHelper().performSum(3,7)

Le funzioni possono essere dichiarate con argomenti di default, che possono essere omessi quando invocate

fun performSum(first: Int, second: Int = 7) = first + second  
val resultIsTen = performSum(3)

nel caso in cui sia necessario specificare quale dei due argomenti vogliamo passare, si utilizza l’invocazione con *named arguments*

fun performSum(first: Int = 3, second: Int = 7) = first + second  
val resultIsFive = performSum(second = 2)

se l’ultimo argomento è un *lambda*, può essere passato normalmente, o fuori in parentesi graffe. Quest’ultimo è lo standard.

fun performSum(first: Int = 3, second: Int = 7, logme: () -> Unit) =   
 first + second

val resultIsFive = performSum(second = 2) **{** *//* this is logme   
**}**

## Singleton

La keywork *object* permette di definire facilmente una classe con pattern *Singleton*:

object Car {  
 fun getCostByMaker(maker: String): Int {  
 // ...  
 }  
}

Car.getCostByMaker("BMV")

Questa viene definita *object declaration*. Ci sono altri utilizzi chiamati *object expression* che servono ad altri scopi.

<https://kotlinlang.org/docs/object-declarations.html>

## run, apply, let, also

Una particolarità di Kotlin è la sua capacità di definire dei *contesti temporanei* grazie alle Scope Functions*.* Estremamente utile per creare costrutti logici e raggruppare ripetizioni.   
Nell’esempio successivo, utilizziamo una variabile che viene poi invocata varie volte:

*A screenshot of a computer

Description automatically generated*

Ecco una versione più Kotlinosa

Graphical user interface, text, application, chat or text message

Description automatically generated

Non solo abbiamo risparmiato l’assegnazione di una variabile, ma il codice è ora schematizzato all’interno di un solo costrutto.

*run* e *let* ritornano entrambi il **risultato dell’ultima riga scritta nel lambda**.   
run identifica l’oggetto del contesto con *this*, mentre let con il parametro *it*

Text

Description automatically generated

*apply* e *also* invece ritornano **l’oggetto su cui si sta lavorando.**

**apply**utilizza *this*, mentre **also** utilizza *it*

In questo caso *mutableListOf*<Int>() crea una lista vuota di Integer:

Graphical user interface, text, chat or text message

Description automatically generated

La variabile *numbers* viene modificata aggiungendo 3 numeri, poi ordinati in ordine crescente. Verrà inoltre stampata.

## map()

Una interessante funzione riguardante gli *iterables* – oggetti come liste, array e mappe – è la funzione **.map()**

È una funziona che applica a tutti i membri dell’iterable la trasformazione dichiarata all’interno del lambda.

Questo permette di rimuovere strutture *for* e comprimere notevolmente il codice.

Graphical user interface

Description automatically generated with medium confidence

Nel nostro esempio a tutti gli elementi della lista *persons* viene applicato il trasform all’interno di map(): il risultato è una lista contenente i *firstname* portati a lowercase di tutti gli elementi di persons.

## Esercizio: Persona virtuale

Crea il seguente programma (puoi usare Kotlin Playground):

* Dichiara una classe Persona che include: id, nome, cognome, età, cittadinanza e hobbies.
* Nome e cognome e hobbies sono campi read-only, mentre età, cittadinanza sono campi variabili.
* cittadinanza è impostata di default a “Italiana”
* hobbies è una lista di stringhe privata e mutabile
* id è un double inizializzato con Random().nextDouble() del pacchetto java.util
* La classe ha una funzione che permette di aggiungere stringhe a hobby

Dopo aver instanziato la classe con i valori di default, fa in modo che questa venga:

1. stampata in un log come è stata creata nel costruttore
2. modificata cambiando età e cittadinanza
3. Aggiungi almeno 2 hobbies e ordinali per lunghezza della parola (basso prima).
4. stampata nuovamente dopo le modifiche

Una volta che l’oggetto è stato modificato e stampato correttamente:

1. Forma una lista popolata dall’oggetto e da 5 sue copie. Ogni copia deve avere un id diverso.
2. Estrai in una lista i 6 id dagli oggetti, e stampa la lista.