# Kotlin (autore: Vittorio Albertoni)

## Premessa

Il linguaggio di programmazione Kotlin è stato inventato in casa JetBrains alcuni anni fa, nel 2011, per fare meglio e più facilmente tutto ciò che si può fare con il linguaggio Java.

Ricordo che JetBrains, in altri tempi chiamata Intellij, è la creatrice dell'ambiente di sviluppo integrato Intellij Idea, pensato per il linguaggio Java, disponibile liberamente con la sua edizione Community.
L'attuale ambiente di sviluppo integrato Android Studio di Google è basato su Intellij Idea e supporta il linguaggio Kotlin, che dal suo rilascio, nel 2012, è utilizzabile con Intellij Idea.

Nel tempo Google è arrivata a raccomandare Kotlin come il migliore linguaggio per programmare le app per Android, addirittura affermando che sarebbe assurdo continuare ad utilizzare Java a questo scopo, visto che c'è Kotlin.

Un'idea su come questo linguaggio sia più semplice di Java la possiamo avere dal seguente confronto.

Un piccolo programma che chiede il nome dell'utente per salutarlo, scritto in linguaggio Kotlin è così:

```
fun main()
{
   println("Come ti chiami?")
   var nome = readLine()
   println("Ciao $nome!")
}
```

e, scritto in linguaggio Java, è così:

```
import java.io.*;
class Saluto
{
    public static void main(String[] args) throws Exception
    {
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Come ti chiami?");
        String nome = input.readLine();
        System.out.println("Ciao " + nome + "!");
        System.exit(0);
    }
}
```

Mi pare evidente che, a partire dall'eliminazione dell'incubo di ricordarsi di mettere il punto e virgola alla fine delle righe, le semplificazioni sono parecchie e non di poco conto e ciò si riflette sicuramente in un miglioramento della facilità di apprendimento del linguaggio e in un alleggerimento nella scrittura dei programmi.

In questo manualetto mi propongo di descrivere come Kotlin possa essere utilizzato da un programmatore alle prime armi per fare cose non eccessivamente impegnative.

Per chi si appassioni e voglia fare di più suggerisco i seguenti indirizzi:
*https://www.html.it/guide/kotlin-guida-al-linguaggio/* in italiano, sempre a livello introduttivo;
*https://kotlinlang.org/api/latest/jvm/stdlib/index.html* dove abbiamo la bibbia di Kotlin in inglese.

Senza dimenticare l'ottimo testo in italiano di Massimo Carli - Kotlin, guida al nuovo linguaggio di Android e dello sviluppo mobile edito da Apogeo, fatto bene ma non proprio per principianti, e che, nell'edizione del maggio 2019, ha il difetto di riferirsi alla ormai superata versione 1.2 di Kotlin.

# Indice

# 1 Installazione

Tra i tanti modi di lavorare con Kotlin (utilizzando il già predisposto IDE Intellij Idea, con plugin per altri IDE, ecc.) scelgo, per intanto, quello più semplice ed istruttivo per un principiante che deve imparare: scrivere il programma con un qualsiasi editor di testo, compilarlo ed eseguirlo da terminale (quello che in Linux e Mac OS X si chiama Terminale e in Windows si chiama Prompt dei comandi).

Un editor di testo è sicuramente presente sul nostro computer (Gedit, Kate, ecc. per chi usa Linux, Blocco Note per chi usa Windows, TextEdit per chi usa Mac).

Per l'esecuzione del programma è necessario che sul computer sia presente Java, per l'esattezza la macchina virtuale Java (JRE, Java Runtime Environment) almeno nella versione 8: anche questo requisito è normalmente soddisfatto da quando acquistiamo il computer o installiamo il sistema operativo[1].
Per verificarlo basta digitare il comando `java -version` su terminale e vedere cosa succede.
Se Java 8 non è installato possiamo rimediare andando all'indirizzo *https://www.java.com/it/download/*, dove troviamo l'originale Java di Oracle. Chi usa Linux trova eventualmente la versione open source nell'installatore dei programmi.

Ciò che sicuramente dobbiamo installare è il compilatore Kotlin. La versione rilasciata lo scorso aprile 2019 la troviamo all'indirizzo *https://github.com/JetBrains/kotlin/releases/tag/v1.3.31*.[2].
Scorriamo la pagina e, nella zona Assets, clicchiamo sulla voce `kotlin-compiler-1.3.31.zip` per scaricare il file (per questo manuale occorre almeno la versione 1.3 che gira senza problemi su Windows da 7 in poi e Ubuntu da 16 in poi, il tutto con almeno Java 8).
Decomprimiamo il file in una directory ospite a nostra scelta e, in questa directory, verremo ad avere la sottodirectory `kotlinc` al cui interno, nella sottodirectory `bin`, si trovano gli eseguibili per compilare e per lanciare il programma compilato.
Inseriamo il percorso agli eseguibili nella variabile d'ambiente PATH in modo da poterli lanciare da qualsiasi directory[3].

# 2 Come funziona: il primo programma

Java è un linguaggio nato nel 1995, nemmeno tanto lontano nel tempo, ma molto deve alla cultura di linguaggi come C e C++, rispettivamente del 1972 e del 1983, e finisce per essere meno «moderno» rispetto a un linguaggio come Python, nato quattro anni prima nel 1991.

Kotlin modernizza Java, condividendone l'ambiente operativo e facendo salvo tutto il patrimonio che nel tempo ha arricchito Java, arrivando a potersi integrare con esso utilizzandone le librerie.

Accanto all'impostazione tutta tesa alla programmazione a oggetti di Java, nato nel periodo in cui era diventato «di moda» questo modo di programmare, Kotlin, per certi versi più ancora orientato agli oggetti rispetto a Java, riscopre la programmazione funzionale e la accosta,

[1] Ho trovato su Internet affermazioni, a volte apparentemente autorevoli, circa la necessità che il computer sia dotato anche dell'ambiente di sviluppo Java (JDK, Java Development Kit). Temo che questa esigenza non si ricolleghi tanto a Kotlin in sé ma al caso in cui utilizziamo Kotlin su IDE (Intellij Idea, Eclipse, NetBeans): in questo caso è l'IDE che ha bisogno di JDK. Posso comunque garantire che, utilizzando, come facciamo in questo manualetto, editor di testo e compilatore direttamente non c'è bisogno del JDK, almeno per compilare gli esempi di programma qui trattati. Nulla vieta, peraltro, che, in vista di dover affrontare programmazioni più impegnative e con maggiore integrazione con il linguaggio Java, sia opportuno installare il JDK.

[2] Kotlin si trova anche nei pacchetti snap. Pertanto chi usa una distro Linux che supporta snap (Debian, Ubuntu, Arch, Fedora, OpenSUSE) lo può installare con il comando a terminale `sudo snap install kotlin --classic`.

[3] Ogni sistema operativo ha i suoi modi per inserire un nuovo percorso nel PATH. In Linux dobbiamo aggiungere, con potere di root, l'istruzione `PATH=$PATH:/<directory_ospite>/kotlinc/bin` nel file `/etc/profile`. Stessa cosa si fa sul file `.profile` in Mac OS X. In entrambi i casi l'effetto dell'intervento si ha con il riavvio del sistema. In Windows dal Pannello di controllo si sceglie Sistema ▷ Impostazioni di sistema avanzate ▷ Variabili d'ambiente e si aggiunge l'indirizzo `C:\<directory_ospite>\kotlinc\bin` nella variabile PATH.
Se utilizziamo Snap, Kotlin si installa nella directory `snap` che è automaticamente aggiunta al PATH.

consentendo che una funzione possa essere utilizzata non necessariamente all'interno di una classe.

Il confronto che ho proposto in premessa tra i due piccoli programmi di saluto mi sembra eloquente al riguardo.

Ora vediamo come si fa a programmare in Kotlin.

Per non perdere l'abitudine lanciata da Brian Kernighan negli anni settanta del secolo scorso, quando ha presentato con Dennis Ritchie il linguaggio C creato da quest'ultimo, riduciamo la prima sperimentazione del linguaggio alla creazione di un piccolissimo programma che scriva il saluto «Hello world».

Come avviene appunto nel linguaggio C, il programma parte eseguendo una funzione chiamata `main`, che non può mancare in nessun programma. Pertanto nel nostro editor di testo preferito scriviamo

```
fun main()
{
   println("Hello world")
}
```

Salviamo il file con un nome acconcio ed estensione `.kt`, per esempio, nel nostro caso, `primo_programma.kt`.

Ora compiliamo il nostro programma.
Apriamo il terminale, ci posizioniamo nella directory dove abbiamo salvato il file del programma e digitiamo il comando
`kotlinc primo_programma.kt`
seguito da INVIO.

Nella stessa directory generiamo così il file `Primo_programmaKt.class` che è il compilato in bytecode adatto alla Java Virtual Machine.

Lo possiamo eseguire da terminale con il comando
`kotlin Primo_programmaKt`
seguito da INVIO.
Sul terminale compare così la scritta `Hello world`.

Nonostante, come ho detto, si tratti di un bytecode adatto alla JVM, per eseguirlo così com'è occorre il comando `kotlin` e, pertanto, occorre avere installato Kotlin sul computer: ciò in quanto, per dare in pasto il nostro programma alla JVM, abbiamo bisogno di passare attraverso una libreria di runtime che fa parte del pacchetto Kotlin.

Possiamo tuttavia compilare il nostro programma includendo questa libreria di runtime e generando un file .jar che potrà essere eseguito, come qualsiasi programma Java, su qualsiasi computer e con qualsiasi sistema operativo che abbia installata la JVM, anche senza avere installato Kotlin.
Per generare lo jar eseguibile Java, che chiameremo `primo_programma.jar`, diamo il comando
`kotlinc primo_programma.kt -include-runtime -d primo_programma.jar`
seguito da INVIO.

Ora siamo in pieno tornati nel mondo Java. Kotlin ci è semplicemente servito per creare più facilmente un vero e proprio programma Java.
La sua esecuzione avviene con il solito comando
`java -jar primo_programma.jar`
seguito da INVIO.

Ma le ambizioni di Kotlin non finiscono qui.

Allo stesso indirizzo da cui abbiamo scaricato il compilatore Kotlin che stiamo usando troviamo infatti altri tre compilatori, chiamati kotlin-native-linux, kotlin-native-macos e kotlin-native-windows, utilizzando i quali possiamo compilare eseguibili rispettivamente per i sistemi operativi Linux, mac OS X e Windows, funzionanti anche senza che sul computer sia installato Java.

La tecnologia è ancora in uno stato appena fuori dallo sperimentale e, francamente, visto che ormai penso non esista computer su cui non sia presente Java - e ci vuole così poco a mettercela - mi pare che ci possiamo accontentare del vecchio file .jar per far circolare i nostri programmi.

Senza dimenticare che, almeno per ora, non esiste la cross-compilation in Kotlin. Per cui, se siamo su Linux possiamo generare un eseguibile nativo che gira solo su Linux, se siamo su Windows possiamo generare un eseguibile nativo che gira solo su Windows, ecc. e, per la circolarità del nostro programma, le cose addirittura peggiorano rispetto alla produzione dell'archivio eseguibile jar.

Il tutto, poi, non ha alcun riflesso su come si scrivono i programmi, che è l'obiettivo di questo manualetto.

* * *

A proposito di come si scrivono i programmi, una cosa molto utile per l'apprendimento del linguaggio e per il collaudo di costrutti prima di inserirli nel programma è la shell interattiva di cui Kotlin è dotato.
Penso che l'unico altro ambiente di sviluppo dotato di shell interattiva sia quello di Python.
La shell interattiva ci consente di scrivere un'istruzione, più o meno complessa, e vederne immediatamente l'effetto senza passare attraverso la compilazione e il lancio del programma.
Essa si apre scrivendo a terminale il comando `kotlinc` e dando INVIO e si presenta come nella seguente figura 1.

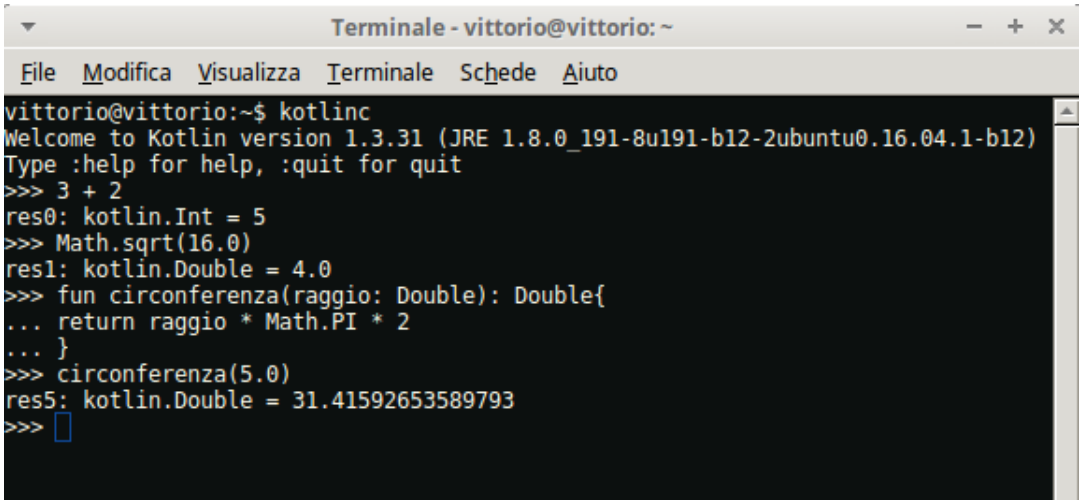



Figura 1: Shell interattiva di Kotlin

Vediamo proposta una semplice addizione con relativo risultato, il calcolo di una radice quadrata utilizzando l'apposita funzione predefinita nel linguaggio e la creazione di una funzione per calcolare la circonferenza, con il suo utilizzo. Tutto ciò avviene «al volo» senza passare attraverso la compilazione.

* * *

Ultima notazione introduttiva riguarda l'inserimento dei commenti nel listato del programma.
Esattamente come in Java, il commento su più righe è inserito tra i simboli `/*` e `*/` e il commento su una sola riga è preceduto dal simbolo `//`.

## 3 Tipi fondamentali

Fedele alla tradizione C, C++ e Java, Kotlin è un linguaggio fortemente tipizzato, a tipizzazione statica.

Nel prossimo capitolo vedremo come in Kotlin la gestione del tipo dei dati per costanti e variabili sia del tutto originale.

Per intanto immagazziniamo il concetto che Kotlin non ha, come praticamente tutti i linguaggi di programmazione, i così detti tipi primitivi (attributi con i quali si definisce il tipo di un valore) ma i tipi vengono definiti ricorrendo a classi.

Dal momento che in Java si è abituati a nominare le classi con iniziali maiuscole questo spiega come mai la definizione dei tipi in Kotlin avvenga ricorrendo a parole che cominciano con la lettera maiuscola.

### Tipi numerici

Le classi per definire i tipi numerici sono
**Byte** per interi con una dimensione di 8 bit
**Short** per interi con una dimensione di 16 bit
**Int** per interi con una dimensione di 32 bit
**Long** per interi con una dimensione di 64 bit
**Float** per decimali con una dimensione di 32 bit
**Double** per decimali con una dimensione di 64 bit

Per avere un'idea delle grandezze numeriche compatibili con i vari tipi di interi occorre elevare 2 al numero dei bit.

Per i decimali ricordare che il tipo float (precisione singola) gestisce fino a 7 cifre decimali significative e il double (doppia precisione) gestisce fino a 15 cifre decimali significative.

Con i tipi numerici possiamo compiere tutte le operazioni aritmetiche:
. somma attraverso l'operatore +
. sottrazione attraverso l'operatore -
. prodotto attraverso l'operatore *
. divisione attraverso l'operatore /
. resto intero attraverso l'operatore %

Per quanto riguarda la divisione, occorre ricordare che se sia il dividendo sia il divisore sono interi il risultato è un intero e l'eventuale parte decimale si perde.

Possiamo utilizzare numeri interi e numeri decimali con gli operatori visti sopra in espressioni finalizzate alla sola scrittura di un altro numero, come elemento letterale.
Se scriviamo `println(3.5)` otteniamo la stampa di 3.5.
Se scriviamo `println(7.0/2)` otteniamo sempre la stampa di 3.5.
Se scriviamo `println(1+2.5)` otteniamo ancora la stampa di 3.5.
Negli ultimi due casi, infatti, 7.0/2 e 1+2.5 erano soltanto modi diversi di scrivere il numero 3.5.
Notare come, per la divisione, almeno uno degli operandi sia stato scritto come numero decimale (7.0 e non 7) in modo da evitare la divisione tra interi (7/2 avrebbe dato il risultato 3).

### Tipi carattere

Le classi per definire i tipi carattere sono
**Char** per un carattere singolo scritto tra apici semplici o doppi con dimensione di 16 bit
**String** per una stringa di caratteri scritta tra doppi apici

La stringa è una successione di caratteri di lunghezza fissa usata per rappresentare testo.
Il suo contenuto si crea scrivendo i caratteri tra una coppia di doppi apici " al cui interno possiamo utilizzare i caratteri di escape, come il new line \n.
Introducendo e chiudendo la stringa con tre doppi apici possiamo scrivere la stringa su più righe e in questo caso non vengono accettati i caratteri di escape.

Più stringhe possono essere sommate attraverso l'operatore + e il risultato è una nuova stringa contenente i caratteri delle stringhe di partenza in successione senza spazi intermedi

(questi, se si vogliono nel risultato, vanno precedentemente inseriti come stringhe vuote o come caratteri vuoti alla fine delle stringhe interessate alla somma).

Possiamo contare i caratteri contenuti in una stringa utilizzando il metodo `length`, che ritorna un numero intero corrispondente al numero di caratteri contenuti nella stringa, contando anche gli eventuali spazi bianchi.
Se abbiamo una stringa `s` contenente la parola `Vittorio` con l'istruzione `println(s.length)` otteniamo la stampa del numero 8.

La posizione di ogni carattere è indicizzata partendo con l'indice 0 per il primo carattere a sinistra. Se alla stringa facciamo seguire l'indice tra parentesi quadre viene ritornato il carattere corrispondente all'indice. Sempre avendo la stringa `s` di prima, con l'istruzione `println(s[1])` otteniamo la stampa del carattere `i`, il secondo della stringa, corrispondente all'indice 1 partendo da zero.

### Tipo booleano

Con la classe **Boolean** rappresentiamo i valori logici `true` e `false`.

Con i valori booleani possiamo utilizzare i tre operatori logici
`&&` che sta per `and`,
`||` che sta per `or`,
`!` che sta per `not`.
Se scriviamo `println(true && true)` otteniamo la stampa di `true`,
Se scriviamo `println(true && false)` otteniamo la stampa di `false`,
Se scriviamo `println(true || true)` otteniamo la stampa di `true`,
Se scriviamo `println(true || false)` otteniamo la stampa di `true`,
Se scriviamo `println(!true)` otteniamo la stampa di `false`.

## 4 Variabili e costanti

Variabili e costanti, in Kotlin, sono oggetti e abbiamo già detto, nel precedente capitolo, che per definire il tipo del dato che esse contengono abbiamo bisogno di ricorrere a classi, non ad aggettivi come avviene nei linguaggi che hanno i tipi primitivi.

Tuttavia abbiamo anche detto che il programma che scriviamo in linguaggio Kotlin gira sulla Java Virtual Machine, che riconosce i tipi primitivi del linguaggio Java. Per fare che tutto funzioni, il tipo che la classe Kotlin assegna al dato è un tipo primitivo di Java.

Se scriviamo, compiliamo ed eseguiamo il seguente programmino

```
fun main()
{
    var a:  Double
    a = 32.78
    println(a.javaClass.name)
}
```

otteniamo il risultato
`double`.
La variabile `a` alla quale abbiamo assegnato il tipo con la classe `Double` (con l'iniziale maiuscola), se andiamo ad indagare di che tipo è, con l'istruzione argomento di `println()`, si rivela essere di tipo `double`, che, con la sua iniziale minuscola altro non è che un tipo primitivo di Java.

Una costante contiene un valore che non si può modificare nel corso del programma mentre una variabile contiene un valore che può essere modificato nel corso del programma.

Una costante si crea con la parola **val** mentre la variabile si crea con la parola **var**.

La sintassi canonica per creare costanti e variabili è

```
val <nome_costante>:  <tipo> = <valore>
var <nome_variabile>:  <tipo> = <valore>
```

Per esempio con `var nome: String = "Vittorio"` creiamo la variabile stringa identificata con `nome` contenente il valore `Vittorio`. Trattandosi di una variabile nulla vieta che, nel corso del programma, con l'istruzione `nome = "Maria"` il valore contenuto diventi `Maria`.
Con `val raggio: Int = 12` creiamo la costante numerica `raggio` contenente il valore `12`. Trattandosi di una costante, il valore 12 che contiene non può essere modificato nel corso del programma ed ogni tentativo di farlo genera un errore.

Esiste anche la sintassi abbreviata che ci solleva dalla fatica di indicare il tipo al momento della creazione:

```
val <nome_costante> = <valore>
var <nome_variabile> = <valore>
```

Il tipo viene assegnato automaticamente da Kotlin desumendolo da ciò che scriviamo al posto di <valore> (ricordarsi i doppi apici per le stringhe).

Di norma il valore va indicato al momento della creazione: Kotlin normalmente usa la tecnica delle costanti e delle variabili inizializzate. Comunque pretende che nel corso del programma avvenga l'inserimento di un valore, altrimenti si generano errori di compilazione. Per differire l'inserimento del valore, si creano costanti e variabili con

```
val <nome_costante>: <tipo>
var <nome_variabile>: <tipo>
```

sapendo che, prima o poi, sono d'obbligo le istruzioni

```
<nome_costante> = <valore>
<nome_variabile> = <valore>
```

altrimenti il programma non funziona.

Dal momento che, come ho già detto, le costanti e le variabili di Kotlin sono oggetti, esse sono dotate di vari metodi. Qui segnalo quelli che ritengo più utili.

**metodi delle costanti e delle variabili numeriche** .
`toString()` trasforma in tipo String il contenuto della costante o della variabile
`toByte()` trasforma in tipo Byte il contenuto della costante o della variabile
`toInt()` trasforma in tipo Int il contenuto della costante o della variabile
`toFloat()` trasforma in tipo Float il contenuto della costante o della variabile
`toDouble()` trasforma in tipo Double il contenuto della costante o della variabile
`compareTo(<altra_costante_o_variabile_numerica>)` instaura un confronto tra la costante/variabile numerica con altra costante/variabile numerica restituendo il valore 0 se sono uguali, un valore positivo se la prima è maggiore della seconda e un valore negativo se la prima è inferiore alla seconda.
Se, per esempio, abbiamo la variabile di tipo intero `x` contenente il valore `9` e vogliamo estrarne la radice quadrata con la funzione `sqrt` del package `Math` che accetta solo parametri di tipo double possiamo scrivere `Math.sqrt(x.toDouble())` e otteniamo il valore `3.0`. Se scrivessimo `Math.sqrt(x)` otterremmo errore.
Ricordare che il casting alla rovescia, da Double a Int, comporta la perdita della parte decimale senza arrotondamento e che tutto va fatto con un occhio alla logica: mi pare per esempio ovvio che il casting tra un double con sette cifre intere a un Byte non potrà mai funzionare.
Interessante notare che, dal momento che Kotlin riconosce il tipo adatto per quanto scriviamo sulla tastiera, i metodi che abbiamo visto si possono applicare direttamente a ciò che scriviamo. Per esempio se scriviamo `10.375.toInt()` otterremo `10`. Se scriviamo `10.toDouble()` otteniamo `10.0`, ecc.

**metodi delle costanti e delle variabili stringa** .
`length` come abbiamo già visto ritorna il numero dei caratteri della stringa
`get(<indice>)` ritorna il carattere corrispondente all'indice nella stringa con tipo Char
`compareTo(<altra_costante_o_variabile_stringa>)` instaura un confronto tra la costante/variabile stringa con altra costante/variabile stringa restituendo il valore 0 se sono di lunghezza

uguale, un valore positivo se la prima è più lunga della seconda e un valore negativo se la prima è più corta della seconda.

**metodi delle costanti e delle variabili char** .
`toInt()` ritorna il codice ASCII del carattere.
Se abbiamo una stringa `s` contenente la frase «che bello!», con `var l = s.length` inseriamo il valore intero `10` nella variabile `l` (la conta comprende anche lo spazio bianco). Con `s.get(3)` non scriviamo nulla, in quanto all'indice `3` della stringa corrisponde uno spazio bianco. Con `s.get(3).toInt()` otteniamo `32` che è il codice ASCII dello spazio bianco.

# 5 Operatori

Gli operatori collegano tra loro operandi di varia natura in espressioni che forniscono un risultato.

## Operatori aritmetici

Li abbiamo già visti parlando dei tipi.

Mi limito a citare quelli di uso più comune per i principianti.

`+` somma, si applica a valori numerici e stringhe,
`-` differenza, si applica a valori numerici,
`*` prodotto, si applica a valori numerici,
`/` divisione, si applica a valori numerici,
`%` resto intero, si applica a valori numerici.

Ricordare sempre che la divisione tra interi restituisce un intero, ignorando la parte decimale. Per avere la parte decimale occorre che almeno uno degli operandi sia di tipo decimale (float o double).

Come si vede non c'è un semplice operatore per l'elevamento a potenza, operazione per la quale dobbiamo trovare altrove lo strumento.

## Operatori di confronto

Servono per confrontare due valori e il risultato che restituiscono è un valore booleano.

Sono i seguenti.

`==` uguale,
`!=` non uguale,
`<` minore,
`<=` minore o uguale,
`>` maggiore,
`>=` maggiore o uguale.

Gli operandi assoggettati al confronto devono avere lo stesso tipo.

## Operatori logici

Li abbiamo già visti parlando del tipo booleano, il risultato che forniscono è un valore booleano e sono i seguenti.

`&&` che sta per `and`,
`||` che sta per `or`,
`!` che sta per `not`.

# 6 Funzioni

La funzione, in altri contesti detta anche procedura o subroutine, è una sezione del programma in cui è racchiusa una serie di istruzioni da eseguire e che vengono eseguite ogni volta che la funzione è chiamata.

Nel Capitolo 2 abbiamo visto che in un programma Kotlin deve esistere almeno una funzione, la funzione `main`: è la funzione che viene automaticamente chiamata al lancio del programma.

Né basta questa sola funzione per far fare qualche cosa al programma: semplicemente per ottenere che il nostro programma scriva un salutino a terminale abbiamo dovuto chiamare un'altra funzione, la funzione `println`().

Le istruzioni contenute nella funzione, salvo che noi stessi la programmiamo, ci sono ignote: sappiamo solo che, chiamando la funzione e inserendo gli eventuali parametri richiesti, otteniamo un certo risultato. Come di fronte a una scatola nera.
Per scrivere Hello world abbiamo chiamato la funzione `println`, le abbiamo passato il parametro (la stringa `"Hello world"`) e ci siamo trovata la scritta a terminale, senza sapere minimamente quali istruzioni siano contenute nella funzione `println` per ottenere questo risultato.

Il linguaggio Kotlin ci mette a disposizione tantissime funzioni, raggruppate in packages, sia originarie del linguaggio sia appartenenti al sistema di sviluppo Java e nel seguito del manuale vedremo come utilizzarle.

Potremmo noi stessi scrivere funzioni e inserirle in packages in modo da averle a disposizione quando serve, ma nell'economia di questo manualetto per principianti direi che non è il caso di complicarci la vita per fare cose da professionisti.

Tuttavia, per semplificare la scrittura di un certo programma, potrebbe essere utile raggruppare alcune istruzioni all'interno del programma stesso, creando funzioni richiamabili, in modo da suddividere i compiti e da evitare di scrivere più volte le stesse cose.

La sintassi per dichiarare e scrivere una funzione è

```
fun <nome> (<nome_parametro>:  <tipo_parametro, ....):  <tipo_risultato>
{
   <istruzioni>
   return <risultato>
}
```

dove <nome> è il nome che intendiamo dare alla funzione, ed è il nome che useremo per richiamarla, <nome_parametro> e <tipo_parametro> sono nome e tipo del parametro (dato) da inserire quando la richiamiamo, e ce ne possono essere più di uno, separati da virgola, <tipo_risultato> è il tipo del risultato che intendiamo ottenere. Le <istruzioni> sono quelle relative alle elaborazioni da effettuare sui parametri per ottenere il risultato.

La sintassi per chiamare una funzione ed eseguirla è
`<nome_funzione> (<parametri>)`

Come esercizio esemplificativo supponiamo di voler scrivere un programma che calcoli Combinazioni e Disposizioni, semplici senza ripetizione, di n numeri presi k a k.
Le formule che dobbiamo utilizzare sono

$C(n,k) = \frac{n!}{(n-k)!k!}$ per le combinazioni,

$D(n,k) = \frac{n!}{(n-k)!}$ per le disposizioni,

nelle quali ricorre spesso il calcolo del fattoriale.
Il fatto che la necessità di questo calcolo ricorra spesso e comporti una non semplice scrittura di istruzioni per effettuarlo suggerisce di isolare queste istruzioni in una funzione dedicata al calcolo del fattoriale in modo da poter richiamare queste istruzioni in blocco quando serve.
La funzione per calcolare il fattoriale può essere scritta così:

```
fun fattoriale (n:  Int):  Long
{
    if (n == 0)
    {
       return 1
    }
    return n * fattoriale (n - 1)
}
```
E' un bell'esempio di formula ricorsiva (funzione che richiama sé stessa) che ci dimostra come il linguaggio Kotlin supporti la ricorsività.
Per scrivere questa funzione sono ricorso all'istruzione `if` che il lettore ancora non conosce. E' stato necessario per inserire nella funzione la convenzione per cui il fattoriale di 0 è 1.
Il programma per calcolare combinazioni e disposizioni di 10 numeri presi 3 a 3 diventa il seguente:
```
fun fattoriale(n:  Int):  Long
{
    if (n == 0)
    {
       return 1
    }
    return n * fattoriale (n - 1)
}
fun main()
{
    var n = 10
    var k = 3
    var c = fattoriale(n)/(fattoriale(n-k) * fattoriale(k))
    var d = fattoriale(n)/fattoriale(n-k)
    println("combinazioni:  $c")
    println("disposizioni:  $d")
}
```
Salviamo il nostro programma nel file calcolo_combinatorio.kt e lo compiliamo.
All'esecuzione del programma verrà scritto sul terminale il risultato in questi termini:
```
combinazioni:  120
disposizioni:  720
```

# 7 Classi

Nonostante le attenzioni per chi ama la programmazione funzionale, Kotlin supporta in pieno la programmazione a oggetti.

Abbiamo visto, anzi, che in Kotlin tutto è un oggetto: persino le variabili e le costanti lo sono.

Tanti più esperti di me raggiungibili in rete possono ricordare o spiegare al lettore che cosa si intende per programmazione a oggetti.

Io provo a farlo capire con un esempio.

Supponiamo di avere spesso bisogno di determinare area e circonferenza del cerchio di cui ci è noto il raggio.

Se vogliamo evitare di scrivere tutte le volte le necessarie formule nella funzione main, anche perché non è detto che ce le ricordiamo (a volte si ha a che fare con formule che più difficilmente di quelle per calcolare la circonferenza sono rimandabili a memoria) possiamo creare delle funzioni che le contengono, come abbiamo fatto nel precedente Capitolo per calcolare il fattoriale oppure possiamo ricorrere alla programmazione a oggetti. Se seguiamo quest'altra

strada scriviamo una Classe attraverso la quale creare un oggetto Cerchio, contenente funzioni (chiamate funzioni membro) attraverso le quali calcolare area e circonferenza.

Come si scrive una funzione lo abbiamo visto. Nel caso del nostro esempio la funzione per calcolare l'area del cerchio sarà scritta così:

```
fun area_cerchio(raggio: Double): Double
{
   return raggio * raggio * Math.PI
}
```

Nella funzione `main`, per calcolare l'area di un cerchio di raggio 3, scriveremo `area_cerchio(3.0)`[4].

Se vogliamo seguire i dettami della programmazione a oggetti dobbiamo scrivere la classe Cerchio, dotarla delle funzioni membro per calcolare area e circonferenza, utilizzarla nella funzione `main` per costruire un oggetto cerchio del raggio desiderato e fare i nostri calcoli richiamando le funzioni membro di questo oggetto.

La sintassi per scrivere la classe è la seguente

```
class <nome_della_classe>(var <nome_parametro>: <tipo_parametro>, ...)
{
   <funzioni_membro>
}
```

Nel nostro caso scriveremo

```
class Cerchio(var raggio: Double)
{
    fun area(): Double
    {
        return raggio * raggio * Math.PI
    }
    fun circonferenza(): Double
    {
        return raggio * 2 * Math.PI
    }
}
```

Questa classe può essere utilizzata nella funzione main per calcolare area e circonferenza del cerchio in questo modo:

```
fun main()
{
   var mioCerchio = Cerchio(3.0)
   var area = mioCerchio.area()
   var circonferenza = mioCerchio.circonferenza()
   println("area: $area")
   println("circonferenza: $circonferenza")
}
```

Innanzi tutto abbiamo costruito l'oggetto mioCerchio come istanza della classe Cerchio con il parametro raggio uguale a 3.
Poi abbiamo calcolato area e circonferenza richiamando le funzioni membro dell'oggetto mioCerchio (con la classica sintassi della programmazione per oggetti indicando il nome dell'oggetto seguito da un punto e poi dal nome della funzione membro).
Infine abbiamo dato istruzioni per la stampa a terminale dei risultati che sono:

```
area: 28.274333882308138
circonferenza: 18.84955592153876
```

[4]Notare l'inserimento del parametro 3 scritto 3.0 come double. Per come abbiamo costruito la funzione, infatti, il tipo del parametro raggio è Double. Se scrivessimo 3 genereremmo errore in quanto passeremmo un intero dove è atteso un double.

# 8 Packages

Il package è una raccolta di funzioni e/o di classi, una libreria.

Il linguaggio Kotlin ce ne mette a disposizione parecchi propri e tutti quelli del sistema di sviluppo Java, per fare praticamente tutto ciò che si può immaginare di programmare.

Anche le funzioni e le classi che creiamo noi potremmo inserirle in package, ma non è un lavoro da dilettanti, soprattutto quando si tratta di utilizzarli.
Ai principianti che vogliano conservare librerie prodotte da loro consiglio di tenerle memorizzate in un file di testo e copiare la parte che interessa nel file dove si scrive il programma, lavorando come fatto negli esempi dei capitoli precedenti: in questo modo creiamo pseudo-packages artigianali senza complicazioni.

I package propri di Kotlin non disponibili per default e quelli del sistema Java, se servono, occorre importarli con l'istruzione `import` come prima istruzione del file di programma, prima della funzione main e di quant'altro.

L'importazione di tutto ciò che contiene il package si fa con la sintassi

```
import <nome_pacchetto>.*
```

Se ci basta importare una sola classe possiamo farlo con la sintassi

```
import <nome_pacchetto>.<nome_classe>
```

Per tutto ciò che può fare un dilettante Kotlin mette a disposizione per default tutti i package propri che servono e, visto che stiamo usando il compilatore per la Java Virtual Machine, il pacchetto java.lang; per cui normalmente non dovremmo avere nulla da importare.

Infatti, per default, abbiamo a disposizione le seguenti funzioni di base con le quali possiamo fare praticamente tutto se lavoriamo a terminale senza interfacce grafiche.

### println

Questa funzione scrive l'output a terminale usando una formattazione di default.

Gli elementi da scrivere, che passiamo come parametri, possono essere nomi di variabili (e ne verrà scritto il valore), letterali numerici o espressioni numeriche (e ne verrà scritto il risultato), stringhe (parole o frasi racchiuse tra doppi apici).

Più parametri si inseriscono separati dal simbolo `+`.

Per iniettare il valore di una variabile in una stringa si può inserire nella stringa il nome della variabile preceduto dal simbolo `$`.

`println` inserisce automaticamente un new line e, dopo avere scritto, va a capo.
Esempio:
Supponendo di avere una variabile `a` contenente il valore `12.5`, l'istruzione

```
println("3 moltiplicato 4 fa" + 3*4 + " e la variabile a vale" + a)
```

scrive a terminale quanto segue

```
3 moltiplicato 4 fa 12 e la variabile a vale 12.5
>>>
```

Lo stesso risultato si sarebbe ottenuto scrivendo

```
println("3 moltiplicato 4 fa" + 3*4 + " e la variabile a vale $a")
```

Se vogliamo applicare formattazioni particolari dobbiamo ricorrere alle direttive di formattazione, utilizzate come segnaposto in stringhe e, dopo la stringa, elencare i nomi delle variabili, nello stesso ordine dei segnaposti, come parametri del metodo `format` dell'oggetto stringa che abbiamo scritto.

Le direttive di formattazione si aprono con il simbolo % cui fanno seguito, per citare quelli di più ricorrente uso, i seguenti altri simboli:
`s` ad indicare l'inserimento di una stringa,
`d` ad indicare l'inserimento di un numero intero non particolarmente formattato,
`f` ad indicare l'inserimento di un numero decimale non particolarmente formattato,
`<n>d` con n numero intero, ad indicare l'inserimento di un intero allineato a destra in un campo di n caratteri,

`.<n>f` con n numero intero, ad indicare l'inserimento di un decimale con una parte decimale di n caratteri (l'ultimo dei quali arrotondato).
Esempio:
Supponiamo di avere tre variabili: a intera 78, b intera 3456 e c decimale 786,89567.
Se scriviamo l'istruzione

```
println("2 moltiplicato 3.5 fa %.2f e la variabile c vale %.3f".format(2*3.5, c))
```

otteniamo

```
2 moltiplicato 3.5 fa 7,00 e la variabile c vale 786,896
>>>
```

Notare che l'istruzione `format` è inserita con il punto che la separa dall'oggetto stringa di cui è metodo; il risultato della moltiplicazione 2 * 3,5 compare con i due decimali richiesti e il valore della variabile c è scritto con i tre decimali richiesti, con arrotondamento.
Se scriviamo l'istruzione

```
println("la variabile a vale%6d\nla variabile b vale%6d".format(a, b))
```

otteniamo

```
la variabile a vale    78
la variabile b vale  3456
>>>
```

con i due interi allineati a destra in campi di sei caratteri, come richiesto.

#### print

Fa tutto quello che fa `println` ma non inserisce il new line e non va a capo quando ha scritto. Per andare a capo occorre inserire il new line `\n` come stringa.

Non andando automaticamente a capo non è adatto per essere usato nella shell interattiva in quanto non compare subito ciò che ha scritto ma compare alla prossima istruzione con un a capo, mischiando le due scritte.

#### readLine

Legge l'input dalla tastiera come stringa e lo colloca in una variabile o in una costante.

La sintassi è

```
var <nome_variabile> = readLine()!!
val <nome_variabile> = readLine()!!
```

I due punti esclamativi sono necessari se, come avviene quando sulla tastiera si inseriscono cifre per un input numerico, è poi necessario il casting in variabili o costanti numeriche per poterne fare uso in calcoli.

Il casting si fa richiamando i relativi metodi `toByte()`, `toShort()`, `toInt()`, `toLong()`, `toFloat()` o `toDouble()` della variabile o costante di tipo stringa creata da `readLine()`.

### Package Math

Contiene costanti e funzioni matematiche che ci possono servire per qualsiasi calcolo possiamo immaginare. Esse si richiamano con la sintassi

```
Math.<costante_o_funzione>
```

Esaminiamole per raggruppamenti.

#### Costanti

Abbiamo le due più utilizzate costanti matematiche `E` (numero e, base dei logaritmi naturali), `PI` (pi greco).

**Radici e potenze**

`pow(<base>, <esponente>)` ritorna in double la base elevata all'esponente indicato, espressi in double,
`exp(x)` ritorna in double il numero e elevato a x, espresso in double,
`sqrt(x)` ritorna in double la radice quadrata di x, espresso in double,
Per radici ennesime dobbiamo ricorrere alla funzione `pow` indicando come esponente il reciproco dell'indice di radice, dato che $\sqrt[n]{x} = x^{\frac{1}{n}}$. Attenzione ad indicare nel rapporto 1/n il numero n come double.

**Funzioni trigonometriche**

Tutte le funzioni trigonometriche hanno l'argomento espresso in double radianti e ritornano un valore double. Le inverse hanno l'argomento in doule e ritornano un valore in double radianti.

Abbiamo le funzioni `sin(x)`, `cos(x)`, `tan(x)` e le funzioni inverse `asin(x)`, `acos(x)`, `atan(x)`.

Abbiamo pure le seguenti funzioni di conversione
`toDegrees(x)` converte radianti in gradi, il tutto espresso in double,
`toRadians(x)` converte gradi in radianti, il tutto espresso in double.

**Funzioni iperboliche**

Tutte le funzioni iperboliche hanno l'argomento espresso in double e ritornano valori in double.

Abbiamo le funzioni `sinh(x)`, `cosh(x)`, `tanh(x)`.

**Funzioni varie**

Tutte queste funzioni operano su tipi double.
abs(x) ritorna il valore assoluto di x,
ceil(x) ritorna, in float64, il più piccolo intero maggiore o uguale a x,
floor(x) ritorna, in float64, il più grande intero minore o uguale a x,
round(x) ritorna il numero intero più prossimo a x,
log(x) ritorna il logaritmo naturale di x,
log10(x) ritorna il logaritmo decimale di x.

# 9 Interattività con l'utente

Negli esempi che abbiamo visto finora abbiamo scritto programmi che contenevano già i dati da elaborare e, una volta lanciati, fornivano il risultato. Se ci accontentassimo di questo, per calcolare le combinazioni e le disposizioni di 18 numeri presi 4 a 4, dovremmo prendere il programma che abbiamo scritto nel Capitolo 6 per calcolare combinazioni e disposizioni di 10 numeri presi 3 a 3, ricopiarlo sostituendo al valore della variabile n il numero 18 e al valore della variabile k il numero 4, ricompilarlo e rilanciarlo.

Ma con quanto abbiamo visto nel precedente Capitolo 8 possiamo ora fare meglio e modificare il programma in modo che, una volta lanciato, chieda all'utente per quali valori vuole eseguire i calcoli in modo che il programma diventi finalizzato a calcolare combinazioni e disposizioni su numeri qualsiasi.

Forti di quanto abbiamo appreso basta che modifichiamo il programma scritto nel Capitolo 6 in questo modo

```
fun fattoriale (n:  Int) :Long
{
    if (n == 0)
    {
        return 1
    }
```

```
    return n * fattoriale (n-1)
}
fun main()
{
    println("Quanti elementi utilizzi?")
    var n = readLine()!!.toInt()
    println("Con quanti elementi formi i raggruppamenti?")
    var k = readLine()!!.toInt()
    var c = fattoriale(n)/(fattoriale(n-k) * fattoriale(k))
    var d = fattoriale(n)/fattoriale(n-k)
    println("combinazioni:  $c")
    println("disposizioni:  $d")
}
```

Lanciando questo programma saremo richiesti di indicare il numero di elementi utilizzati (n), il numero di elementi per i raggruppamenti (k) e in un attimo vedremo il risultato. Purtroppo, a causa del pesante calcolo dei fattoriali implicati nelle formule, avremo risultati sballati se il numero di elementi (n) supera la ventina[5].

# 10 Strutture di controllo

Come tutti i linguaggi di programmazione, Kotlin ci offre il modo di controllare l'esecuzione del programma attraverso istruzioni per condizionare l'esecuzione di un blocco al verificarsi di certe condizioni oppure per la ripetizione di blocchi.

### if

L'istruzione `if`, chiamata istruzione di esecuzione condizionale (in inglese if è il nostro se), ci dà modo di assoggettare l'esecuzione di un blocco di istruzioni al verificarsi di una determinata condizione: se la condizione è vera viene eseguito il blocco di istruzioni, altrimenti si prosegue l'esecuzione del programma saltando il blocco stesso.

La sintassi è la seguente

```
if (<condizione>)
{
   <istruzioni>
}
```

dove <condizione> è una qualsiasi espressione che relaziona due valori attraverso operatori di confronto: se la condizione si verifica vengono eseguite le istruzioni contenute tra le parentesi graffe, altrimenti si passa oltre.

L'istruzione `if` si presta anche all'esecuzione condizionale a due rami. Per ottenere questo dobbiamo abbinarla all'istruzione `else` con questa sintassi

```
if (<condizione>)
{
   <istruzioni>
}
else
{
   <istruzioni>
}
```

[5]Quello dello stack overflow, cioè del non riuscire a trattare numeri al di là di certe dimensioni, è un grande limite di tutti i linguaggi compilati (Pascal, C, C++, ADA) che si è trascinato anche nel linguaggio semi-compilato Java da cui deriva Kotlin. Per la verità nel pacchetto java.math esistono classi come BigInteger e BigDecimal che alzano i limiti. Se vogliamo essere tranquilli e poter usare grandi numeri senza alcuna complicazione è comunque meglio che ricorriamo al linguaggio Python, dove è tutto più semplice.

In questo caso se la condizione si verifica vengono eseguite le istruzioni contenute nel primo blocco altrimenti vengono eseguite quelle contenute nel secondo blocco dopo else (altrimenti). In ogni caso proseguendo poi nell'esecuzione del resto del programma.

Possiamo infine gestire l'esecuzione condizionale a più rami abbinando a `if` l'istruzione `else if` con questa sintassi

```
if (<condizione>)
{
   <istruzioni>
}
else if (<condizione>)
{
   <istruzioni>
}
else if (<condizione>)
{
   <istruzioni>
}
else if (<condizione>)
{
   <istruzioni>
}
.....
```

e proseguire quanto vogliamo.

Il seguente programma

```
fun main()
{
    println("Inserisci un numero")
    var x = readLine()!!.toInt()
    if (x < 10)
    {
        println("Ci sei dentro")
    }
    println("Ciao")
}
```

se inseriamo un numero minore di 10 stampa la frase «Ci sei dentro» e poi il saluto «Ciao»; se inseriamo 10 o un numero superiore stampa semplicemente il saluto «Ciao».

Quest'altro programma

```
fun main()
{
    println("Inserisci un numero")
    var x = readLine()!!.toInt()
    if (x < 10)
    {
        println("Ci sei dentro")
    }
    else
    {
        println("Sei fuori")
    }
    println("Ciao")
}
```

se inseriamo un numero minore di 10 stampa la frase «Ci sei dentro» e poi il saluto «Ciao»; se inseriamo 10 o un numero superiore stampa la frase «Sei fuori» e poi il saluto «Ciao».

Quest'altro ancora

```
fun main()
{
    println("Inserisci un numero")
    var x = readLine()!!.toInt()
    if (x < 10)
    {
        println("Ci sei dentro")
    }
    else if (x >= 10 && x <= 12)
    {
        println("Sei appena fuori")
    }
    else
    {
        println("Sei completamente fuori")
    }
    println("Ciao")
}
```

se inseriamo un numero minore di 10 stampa la frase «Ci sei dentro» e poi il saluto «Ciao»; se inseriamo 10 o 11 o 12 stampa la frase «Sei appena fuori» e poi il saluto «Ciao»; se inseriamo un numero superiore a 12 stampa la frase «Sei completamente fuori» e poi il saluto «Ciao».

**when**

Il costrutto `when` può semplificare ciò che possiamo fare con la serie di else e else if.

Con questo costrutto l'ultimo esempio che ho fatto può essere scritto così

```
fun main()
{
    println("Inserisci un numero")
    val numero = readLine()!!.toInt()
    when (numero)
    {
        in 0..9 -> println("Ci sei dentro")
        in 10..12 -> println("Sei appena fuori")
        else -> println("Sei completamente fuori")
    }
    println("Ciao")
}
```

Come si vede, si abbinano direttamente istruzioni semplici ai valori che può assumere una determinata variabile, che può anche essere una stringa, come vediamo nel seguente esempio.

```
fun main()
{
    println("Inserisci il colore del semaforo con lettera minuscola")
    val semaforo = readLine()
    when (semaforo)
    {
        "rosso" -> println("Fermati")
        "verde" -> println("Passa tranquillo")
        "giallo" -> println("Rallenta perché dovrai fermarti")
        else -> println("Dovevi inserire uno dei colori del semaforo in minuscolo")

    }
}
```

In questo caso, a seconda del colore che scriviamo, abbiamo la risposta che ci indica come comportarci e se inseriamo una parola che non indica un colore del semaforo o se non usiamo l'iniziale minuscola come richiesto abbiamo l'invito a comportarci come si deve.

### while

Si usa quando si vuole ripetere l'esecuzione di una istruzione o di un blocco di istruzioni fino a quando rimane vera (true) una determinata condizione booleana.

E' la tipica istruzione che consente di ripetere l'esecuzione un numero definito di volte utilizzando un contatore.

La sintassi per l'uso di questa istruzione è

```
var contatore = 1
while (contatore <= n)
{
   <istruzioni>
   contatore = contatore + 1
}
```

La variabile che ho battezzato `contatore` per richiamarne la natura, viene inizializzata con il valore 1. Ad ogni esecuzione delle istruzioni viene aumentata di 1 e, una volta raggiunto un valore n voluto, si ferma l'esecuzione. In genere viene nominata semplicemente `i`.

Questo piccolo programma

```
fun main()
{
    var i = 1
    while (i <= 3)
    {
        println("Ciao, Vittorio")
        i = i + 1
    }
}
```

scrive la frase Ciao, Vittorio per tre volte.

Quest'altro

```
fun main()
{
    var i = 1
    while (i <= 6)
    {
        println(i)
        i = i + 1
    }
}
```

elenca i numeri da 1 a 6.

In entrambi gli esempi il blocco da ripetere contiene una sola istruzione oltre a quella di aumentare di 1 il contatore ma, ovviamente, può contenerne quante vogliamo.

### for

Si usa quando si vuole ripetere l'esecuzione di una istruzione o di un blocco di istruzioni ogniqualvolta è vera (true) una determinata condizione booleana, verificata attraversando una qualsiasi struttura (stringa, range di valori, ecc.)

La sintassi per l'uso di questa istruzione è

```
<definizione di una struttura>
for (<variabile> in <struttura>)
{
   <istruzioni>
}
```

Se definiamo la struttura come un range di numeri interi (per esempio da 1 a 5) con l'istruzione for possiamo fare ciò che si fa con while.
Per scrivere tre volte il saluto «Ciao, Vittorio!» possiamo fare così:

```
fun main()
{
    var r = 1..3
    for (x in r)
    {
        println("Ciao, Vittorio")
    }
}
```

La struttura potrebbe essere una stringa e potremmo collegare l'esecuzione di un'istruzione o di un blocco di istruzioni alla presenza di un certo carattere o tipo di carattere contenuto nella stringa, come nei seguenti esempi.

Il seguente programma

```
fun main()
{
    var s = "Vittorio"
    for (x in s)
    {
        if (x.toString() == "t")
        println("trovato t")
    }
}
```

scrive due volte la frase «trovato t» avendo trovato due volte la lettera `t` nella stringa `Vittorio`.

Quest'altro programma

```
fun main()
{
    var s = "Vittorio abita a Milano"
    for (x in s)
    {
        if (x.isUpperCase())
        println(x)
    }
}
```

stampa le lettere V e M che sono le uniche lettere maiuscole contenute nella stringa.

## 11 Integrazione con Java

Fin qui abbiamo utilizzato strumenti (classi e funzioni) propri di Kotlin.

Ho ripetutamente detto, tuttavia, che Kotlin ha a disposizione tutte le librerie Java, cioè i packages che contengono le classi Java.

Importando una classe Java possiamo utilizzarla per costruire un oggetto in Kotlin e utilizzarne le funzioni membro per fare i nostri programmi con il linguaggio Kotlin.

Per esempio non mi risulta che Kotlin abbia una funzione per generare numeri casuali.

Dato che nel package java.util esiste la classe Random con cui possiamo costruire un oggetto generatore di numeri casuali possiamo importarla ed utilizzarla a questo fine.

Poniamo, per esempio, che vogliamo generare una serie di 3 numeri casuali compresi tra 0 e 10.
Lo possiamo fare con questo programmino

```
import java.util.Random
fun main()
{
    var generatore = Random()
```

```
    var limite = 10
    var contatore = 1
    while(contatore <= 3)
    {
        println(generatore.nextInt(limite))
        contatore = contatore + 1
    }
}
```

Con l'istruzione `import` importiamo la classe Random del package java.util, poi nella funzione `main()` la utilizziamo per costruire il generatore con la sintassi di Kotlin e, sempre con la sintassi di Kotlin, facciamo tutto il resto, salvo quando richiamiamo la funzione generatrice (`nextInt()`) che è definita nel linguaggio Java della classe Random.

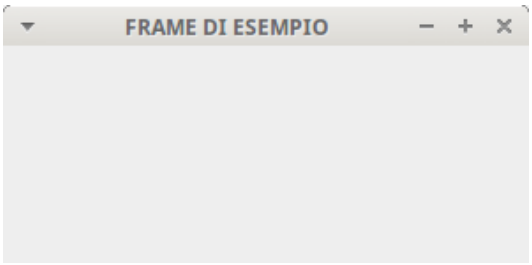
Ma le buone notizie non finiscono qui.

Tra le librerie Java ci sono infatti quelle per la grafica, la vecchia java.awt, la gloriosa javax.swing e la moderna javafx, più complicata e che richiede una installazione a parte.
Con queste librerie abbiamo la possibilità di costruire programmi dotati di GUI (Graphical User Interface), più apprezzati da chi preferisce usare il mouse insieme alla tastiera e non ama lavorare sul terminale.

Il seguente programma

```
import javax.swing.JFrame
fun main()
{
    var mioFrame = JFrame("FRAME DI ESEMPIO")
    mioFrame.setBounds(200, 100, 300, 150)
    mioFrame.setVisible(true)
}
```

genera questa finestrella




Dopo aver importato la classe JFrame del package javax.swing l'abbiamo utilizzata per creare l'oggetto mioFrame dandogli il titolo «FRAME DI ESEMPIO».
Poi, usando il metodo `setBounds` dell'oggetto `mioFrame`, gli abbiamo dato una posizione (le prime due cifre 200 e 100 indicano le coordinate della posizione dell'angolo in alto a sinistra del frame sullo schermo) e una dimensione (le altre due cifre 300 e 150 indicano rispettivamente la larghezza e l'altezza del frame in pixel).
Infine abbiamo fatto in modo che il frame si veda utilizzando il metodo `setVisible` con parametro `true`.
Anche in questo caso abbiamo l'integrazione tra la sintassi del linguaggio Kotlin e la sintassi del linguaggio Java con cui utilizzare i metodi dell'oggetto creato.

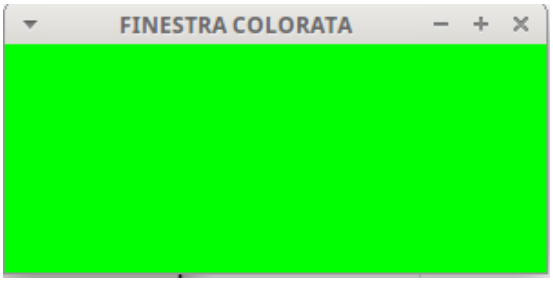
Aggiungiamo ora una piccola complicazione: coloriamo la finestra.

L'unico modo per colorare un Frame Java è quello di aggiugere al Frame un Panel che lo copra tutto e colorare quest'ultimo.
Possiamo raggiungere l'obiettivo con questo programma

```
import java.awt.Color
import javax.swing.JFrame
import javax.swing.JPanel
fun main()
```

```
{
    val pannello = JPanel()
    pannello.setBackground(Color.green)
    val mioFrame = JFrame("FINESTRA COLORATA")
    mioFrame.setBounds(200, 200, 300, 150)
    mioFrame.getContentPane().add(pannello)
    mioFrame.setVisible(true)
}
```

che genera il seguente risultato




Rispetto a prima abbiamo dovuto importare dalle librerie Java anche la classe Color del package awt e la classe JPanel del package swing. Abbiamo costruito il pannello, l'abbiamo colorato. Poi abbiamo costruito la finestra, l'abbiamo dimensionata e vi abbiamo inserito il pannello colorato.

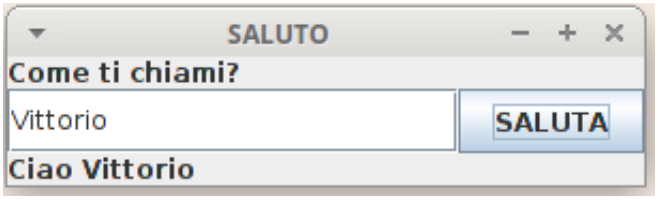
Notiamo come abbia inserito gli oggetti creati, nel primo esempio in variabili (var) e nel secondo esempio in costanti (val): si può fare indifferentemente in un modo o nell'altro.

Visto che siamo partiti con un esempio in cui si chiedeva il nome dell'utente per salutarlo con un programmino a riga di comando vediamo come la stessa cosa si possa fare con una GUI, sempre usando il linguaggio Kotlin, questa volta, però, integrato con le librerie Java per la grafica.

Il listato del programma è questo

```
import java.awt.BorderLayout
import javax.swing.JButton
import javax.swing.JFrame
import javax.swing.JLabel
import javax.swing.JTextField
fun main()
{
    val richiesta = JLabel("Come ti chiami?")
    val risposta = JTextField()
    val pulsante = JButton("SALUTA")
    val saluto = JLabel(".")
    val frame = JFrame("SALUTO")
    frame.getContentPane().add(richiesta, BorderLayout.NORTH)
    frame.getContentPane().add(risposta, BorderLayout.CENTER)
    frame.getContentPane().add(pulsante, BorderLayout.EAST)
    frame.getContentPane().add(saluto, BorderLayout.SOUTH)
    frame.setBounds(200,200, 300, 150)
    pulsante.addActionListener
    {
        var nome = risposta.getText()
        saluto.text ="Ciao $nome"
    }
    frame.setVisible(true)
}
```

e la GUI generata è questa




La GUI si apre con vuote le finestrelle per il nome e per il saluto. Inserito il nome e premuto il pulsante SALUTA viene scritto il saluto in basso.

Come si vede, le cose si complicano, soprattutto per la necessità di conoscere le librerie Java e il modo di utilizzarle in Kotlin.

## 12 Utilità di un IDE

Per programmare in Java l'utilità di avvalerci di un Integrated Development Environment la riscontriamo su due fronti:
. la code completion,
. la possibilità di costruire l'interfaccia grafica con un designer.

I tre classici IDE per programmare in Java sono, in ordine di anzianità, NetBeans, Intellij Idea e Eclipse.

Per programmare in Kotlin possiamo utilizzare gli stessi IDE, poi vediamo con quali integrazioni, sapendo tuttavia che l'unica utilità che rimane è la code completion in quanto i designer scrivono codice Java: peraltro Eclipse non ci dà nemmeno la code completion e la sua utilità si riduce a quella di un comune editor di testo.
La code completion, soprattutto se si lavora con grafica, è molto utile per alleggerire le difficoltà create dalla mancanza del designer.

Il miglior IDE per programmare in Kotlin è, ovviamente, Intellij Idea, predisposto ad utilizzare il linguaggio Kotlin da quando questo è nato.
NetBeans, a partire dalla versione 8.2, ci offre Kotlin nei suoi plugin.
Eclipse si offre il plugin nell'Eclipse Marketplace.

A seconda dell'anzianità dell'IDE può accadere che la predisposizione o il plugin che installiamo non supportino la versione 1.3 di Kotlin ma supportino ancora la 1.2: non dimentichiamo che la versione 1.3 è stata rilasciata a fine ottobre 2018.
Tra la versione 1.2 e la versione 1.3 l'unica differenza di rilievo, per una programmazione di non eccessivo impegno, sta nella semplificazione apportata nella scrittura dell'istruzione di creazione della funzione `main`.
Nella versione 1.2 la sintassi è

```
fun main(args:  Array<String>)
```

mentre nella versione 1.3 la sintassi è semplicemente

```
fun main()
```

Se vogliamo utilizzare Kotlin per sviluppare applicazioni Android non possiamo che avvalerci di Intellij Idea oppure di Android Studio che su di esso è basato.

*  *  *

I programmi Kotlin che produciamo con l'IDE, se l'unica nostra risorsa per programmare Kotlin è l'IDE, possiamo eseguirli richiamandoli nello stesso IDE e da lì lanciarli. Salvo avere installato anche il compilatore e lanciarli con il comando `kotlin` da terminale.

Conviene pertanto che produciamo il file java eseguibile .jar con l'IDE.
Indirizzo chi non sappia come fare al fascicoletto PDF «java_android» allegato al mio articolo «Importante riconoscimento per OpenJDK» dell'aprile 2016, archiviato in Programmazione sul mio blog *www.vittal.it*.

*  *  *

Un'ultima considerazione, per chi possa essere interessato, riguarda una particolarità di Intellij Idea: questo IDE è dotato di un designer per la GUI in Java (purtroppo molto più difficile da utilizzare rispetto a quello di NetBeans) ed ha la possibilità di convertire codice Java in codice Kotlin.

La circostanza ci offre l'opportunità di disegnare la GUI con il designer e di utilizzarla in Kotlin convertendo in linguaggio Kotlin la funzione main che utilizza la GUI in Java: praticamente si costruisce una funzione main in linguaggio Kotlin che utilizza la classe Java che genera la GUI.

Dal momento che la classe Java che genera la GUI, dotata del metodo main in linguaggio Java, è già un programma inseribile in un eseguibile .jar, il trascinare l'utilizzo di questa classe in un metodo main di Kotlin per arrivare allo stesso risultato potrebbe apparire un esercizio inutile. A meno che il programma Kotlin non abbia la sola finalità di utilizzare la classe Java per la GUI ma abbia anche altre finalità.

Per chi voglia fare l'inutile esercizio e per chi abbia in mente cose più complesse accenno al modo di procedere.

In Intellij Idea si crea un nuovo progetto con FILE ▷ NEW ▷ PROJECT, scegliendo JAVA e cliccando sull'opzione KOTLIN/JVM della zona ADDITIONAL LIBRARIES AND FRAMEWORKS. Nella successiva finestra si dà un nome al progetto e si clicca su NEXT.

Il progetto viene creato e se lo apriamo nella finestra di navigazione sulla sinistra vediamo che contiene la cartella SRC. Cliccando destro su questa scegliamo NEW nel menu che compare e con questo sistema creiamo prima un KOTLIN FILE/CLASS e poi un GUI FORM dando loro nomi appropriati.

Appena creato il GUI Form vediamo comparire nella cartella SRC il file Kotlin con estensione `.kt` e i due file della GUI, uno per il disegno e uno per il codice, e ci ritroviamo aperto quello per il disegno con predisposta nel FORM la componente di base contenitore JPANEL.

Disegnamo la GUI avvalendoci del designer, non dimenticando di battezzare la componente JPANEL selezionandola nella finestra COMPONENT TREE e inserendo un nome acconcio (ad esempio «pannello», senza usare molta fantasia) nella finestrella VALUE della PROPERTY FIELD NAME.

Una volta disegnata la GUI apriamo il file di codice Java generato dal designer.
Ci posizioniamo appena prima della parentesi graffa di chiusura della public class del form, clicchiamo destro, scegliamo GENERATE ▷ FORM MAIN.

Siccome il designer ha generato il pannello come private, dobbiamo renderlo public cliccando destro appena dopo la parentesi graffa di chiusura della funzione main appena creata e scegliendo GENERATE ▷ GETTER e poi il pannello nell'elenco che compare.

A questo punto apriamo il file con estensione `.kt`, creiamo la funzione `main()` e all'interno delle parentesi graffe copiamo quanto c'è tra le parentesi graffe della funzione main che avevamo creato nel file `.java` del designer, non dimenticando di dare il nostro assenso alla richiesta di conversione che ci viene posta.

Compiliamo il file `.kt` ed abbiamo il nostro programma Kotlin dotato di GUI.