

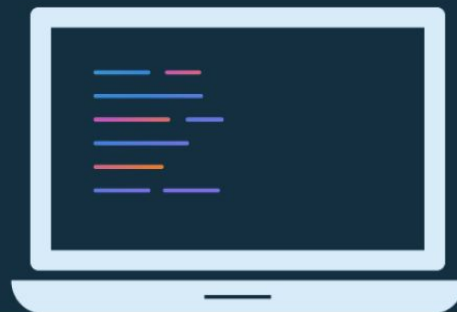


Lezione 1.2

Funzioni



[Functional - Thelonius Monk](#)



Questa lezione

Lezione 1.2 - Funzioni

- Programmi in Kotlin
- (Quasi) tutto ha un valore
- Funzioni in Kotlin
- Funzioni compatte
- Lambdas & higher-order functions
- List filters

Programmi in Kotlin

Setting up

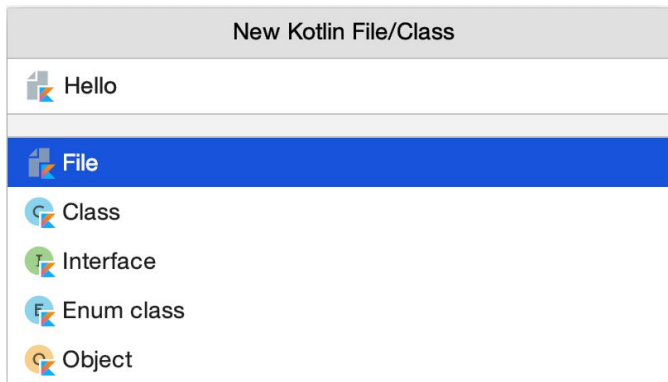
Prima di poter scrivere codice, occorre:

- Creare un file nel progetto
- Creare una funzione `main()`
- Passare degli argomenti al `main()` (opzionale)
- Lanciare il programma

Creare un file Kotlin

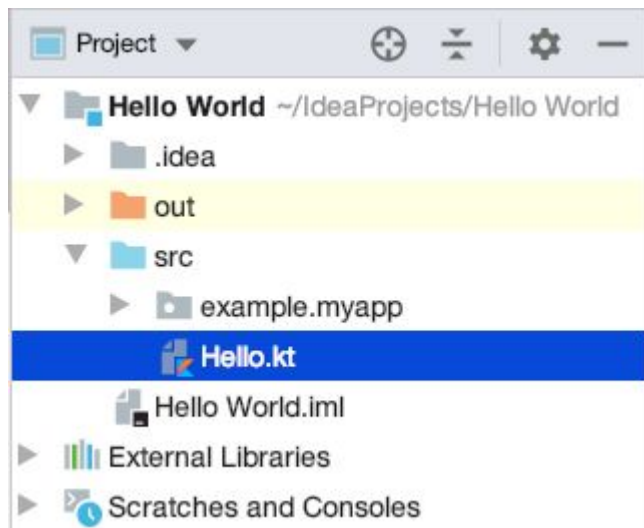
Nel pannello di progetto di IntelliJ IDEA, sotto **Hello World**, clicca con il destro la cartella `src`

- Seleziona **New > Kotlin File/Class**.
- Seleziona **File**, nomina il file `Hello`, e premi **Invio**



Creare un file Kotlin

Dovresti vedere un file chiamato `Hello.kt` nella cartella `src`



Creare una funzione `main()`

`main()` è l'entry point dell'esecuzione di un programma Kotlin


Nel file `Hello.kt`:

```
fun main(args: Array<String>) {  
    println("Hello, world!")  
}
```

Gli argomenti della funzione `main()` sono opzionali.

Eseguire il programma

Per eseguire il programma, clicca l'icona ▶ Run alla sinistra della funzione `main()`

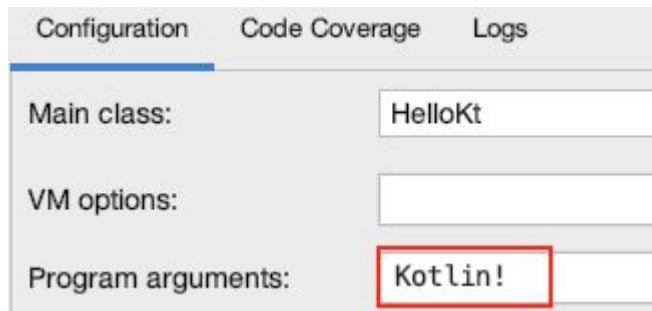
```
1 ▶  fun main(args: Array<String>) {  
2     println("Hello, world!")  
3 }  
4
```

IntelliJ IDEA esegue il programma e mostra i risultati nella console

```
HelloKt x  
/Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Home/bin/java  
Hello, world!  
  
Process finished with exit code 0
```

Passare argomenti al main()

Seleziona **Run > Edit Configurations** per aprire **Run/Debug Configurations**



Usare argomenti nel `main()`

Usa `args[0]` per accedere al primo argomento passato al `main()`.

```
fun main(args: Array<String>) {  
    println("Hello, ${args[0]}")  
}
```

⇒ Hello, Kotlin!

(Quasi) tutto ritorna un valore

(Quasi) tutto ritorna un valore

In Kotlin, quasi tutto è un'espressione e ritorna un valore (persino l'espressione `if`)

```
val temperature = 20
```

```
val isHot = if (temperature > 40) true else false
```

```
println(isHot)
```

```
⇒ false
```

Expression values

A volte il valore di ritorno è `kotlin.Unit`.

```
val isUnit = println("This is an expression")  
println(isUnit)
```

⇒ This is an expression
kotlin.Unit

Funzioni in Kotlin

Funzioni

- Un blocco di codice che esegue un compito specifico
- Divide un problema più grande in parti modulari più piccole
- Viene dichiarata usando la keyword `fun`
- Può avere argomenti, i quali possono assumere valori di default o nomi specifici

Parti di una funzione

Una semplice funzione che stampa a video "Hello World".

```
fun printHello() {  
    println("Hello World")  
}
```

```
printHello()
```

Funzioni che ritornano Unit

Se una funzione non ritorna alcun valore particolare, il suo tipo di ritorno è `Unit`.

```
fun printHello(name: String?): Unit {  
    println("Hi there!")  
}
```

`Unit` è un tipo con un solo possibile valore: `Unit`.

Funzioni che ritornano Unit

La dichiarazione del tipo di ritorno `Unit` è opzionale:

```
fun printHello(name: String?): Unit {  
    println("Hi there!")  
}
```

ed è equivalente a non indicarlo:

```
fun printHello(name: String?) {  
    println("Hi there!")  
}
```

Argomenti di funzione

Le funzioni possono avere:

- Default parameters
- Required parameters
- Named arguments

Default parameters

I valori di default forniscono un fallback se non viene passato alcun parametro:

```
fun drive(speed: String = "fast") {  
    println("driving $speed")  
}
```

Si usa "=" dopo il tipo per definire un parametro di default



drive() ⇒ driving fast

drive("slow") ⇒ driving slowly

drive(speed = "turtle-like") ⇒ driving turtle-like

Required parameters

Se nessun valore di default è specificato per un parametro, questo sarà obbligatorio (required).

Required parameters




```
fun tempToday(day: String, temp: Int) {  
    println("Today is $day and it's $temp degrees.")  
}
```

Default vs. required parameters

Una funzione può avere sia parametri di default che obbligatori:

```
fun reformat(str: String,  
            divideByCamelHumps: Boolean,  
            wordSeparator: Char,  
            normalizeCase: Boolean = true){
```



Ha un valore di default

Passaggio dei parametri obbligatori:

```
reformat("Today is a day like no other day", false, '_')
```

Named arguments

Per aumentare la leggibilità, si usano spesso argomenti con un nome (named) per i parametri obbligatori:

```
reformat(str, divideByCamelHumps = false, wordSeparator = '_')
```

Viene considerata una best practice specificare i default arguments dopo quelli posizionali, in modo che il chiamante possa indicare anche soltanto i primi e tralasciare i secondi.

Funzioni compatte

Single-expression functions

Le funzioni compatte, o single-expression functions, rendono il codice più conciso e leggibile:

```
fun double(x: Int): Int {  
    return x * 2  
}
```



Versione completa

```
fun double(x: Int): Int = x * 2
```



Versione compatta

vararg, tailrec

Numero variabile di argomenti

E' possibile definire che una funzione accetta un numero variabile di parametri di un certo tipo. In questo caso si marca il parametro con `vararg`

```
fun calcolaMediaVoti(matricola: String, vararg voti:Int): Double {  
    return voti.average()  
}
```

`calcolaMediaVoti("123456",18,30,28,25)`  **Elenco variabile di Int**

Numero variabile di argomenti

Se ho un array posso passarlo come parametro utilizzando l'operatore spread (*)

```
fun calcolaMediaVoti(matricola: String, vararg voti:Int): Double {  
    return voti.average()  
}
```

```
val voti = intArrayOf(18,30,28,25)  
calcolaMediaVoti("123456", *voti)
```

Ricorsione

Kotlin supporta uno stile di programmazione funzionale chiamata *tail recursion*: per alcuni algoritmi implementabili con un ciclo è possibile scrivere una versione ricorsiva efficiente senza rischi di stack overflow.

```
tailrec fun factorial(n: Long, accum: Long = 1): Long {  
    val soFar = n * accum  
    return if (n <= 1) {  
        soFar  
    } else {  
        factorial(n - 1, soFar)  
    }  
}
```

Ricorsione

Nota: questo non è possibile se la funzione non ha la chiamata ricorsiva come ultima istruzione, come nel caso seguente in cui l'ultima istruzione è la moltiplicazione tra n ed il risultato della chiamata

```
fun recursiveFactorial(n: Long) : Long {  
    return if (n <= 1) {  
        n  
    } else {  
        n * recursiveFactorial(n - 1)  
    }  
}
```