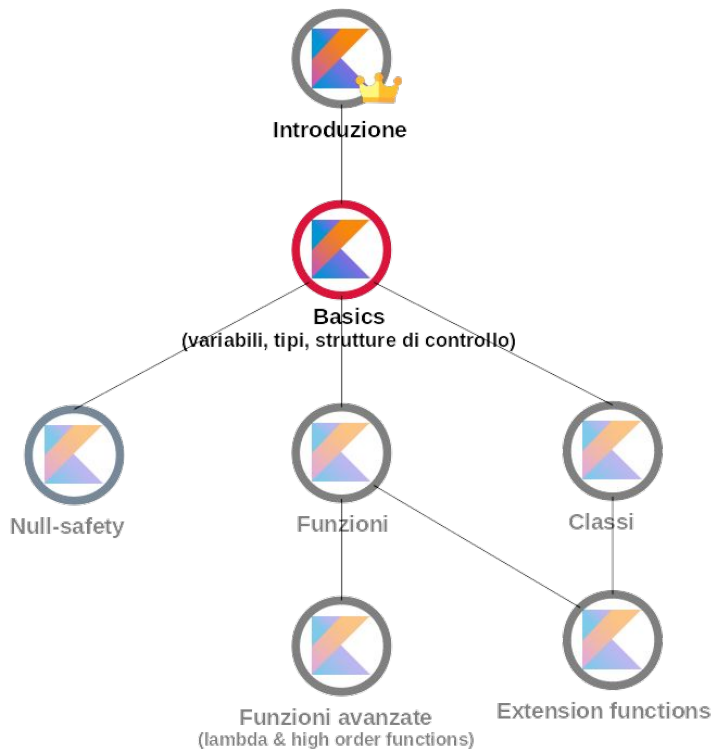


Kotlin



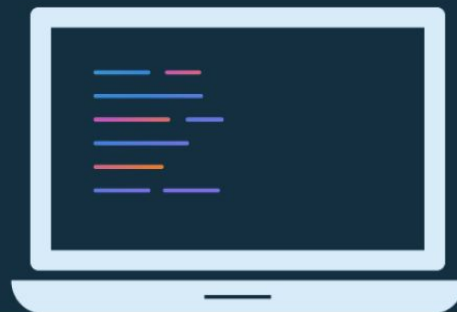


Lezione 1.1

Kotlin basics



[Algorithm - Muse](#)



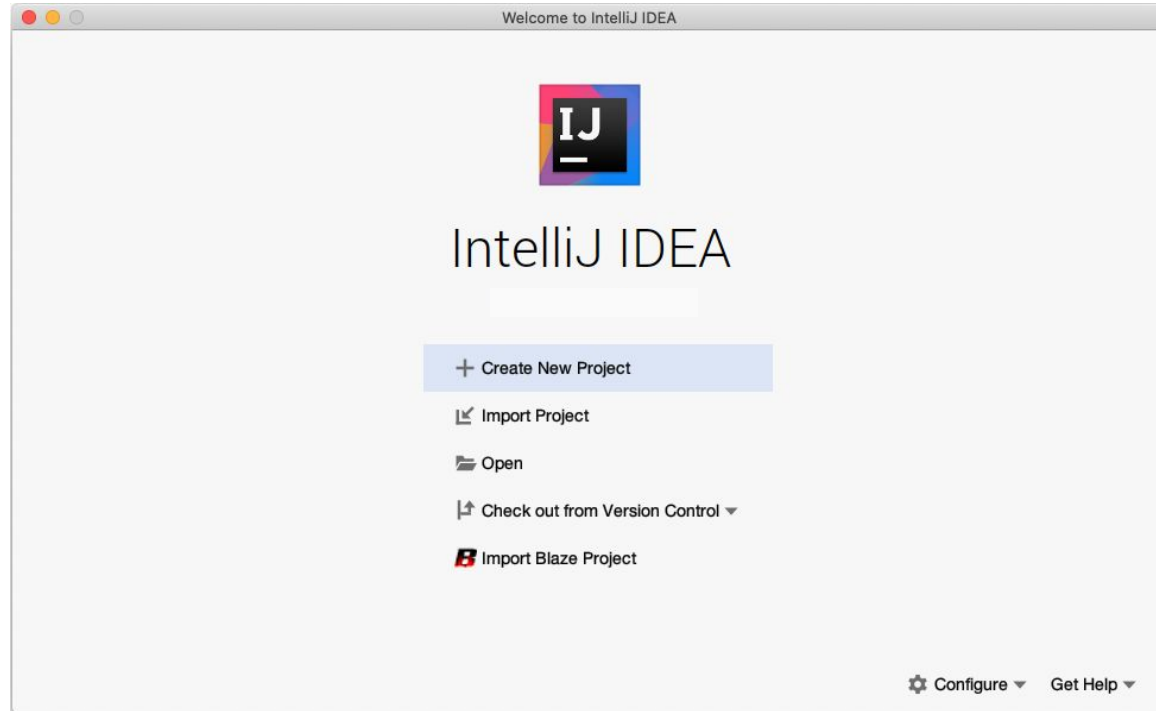
Questa lezione

Lezione 1.1 - Kotlin basics

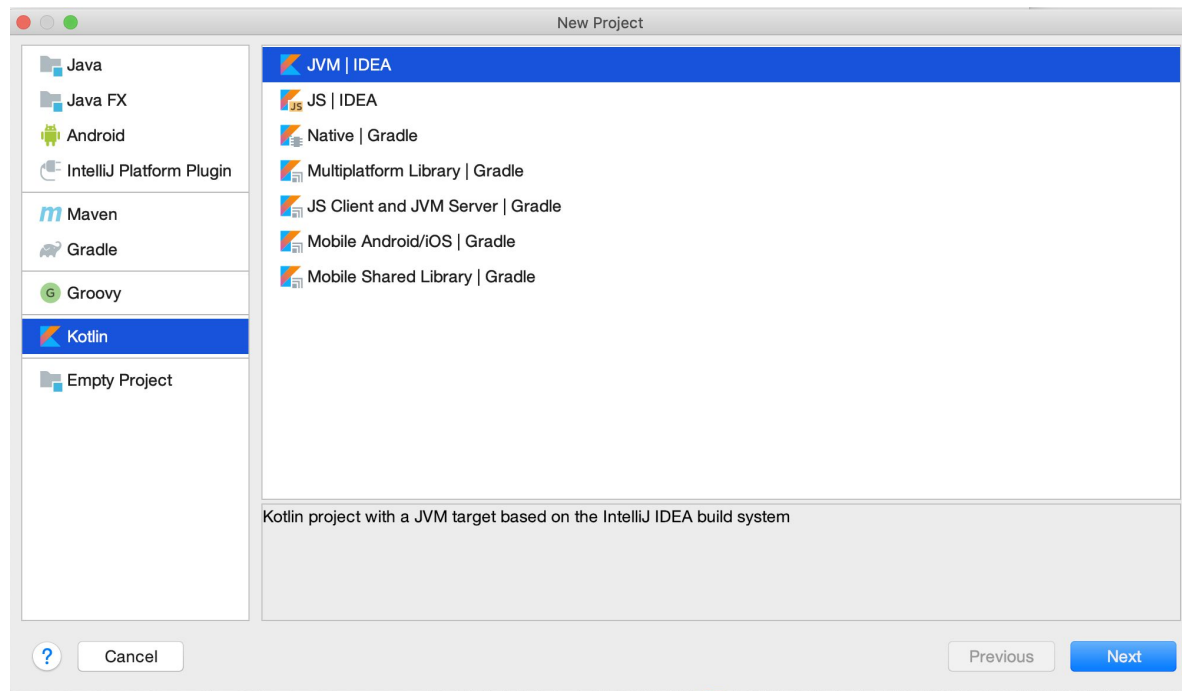
- Get started
- Operatori
- Data types
- Variabili
- Strutture condizionali
- Liste e array
- Null safety

Get started

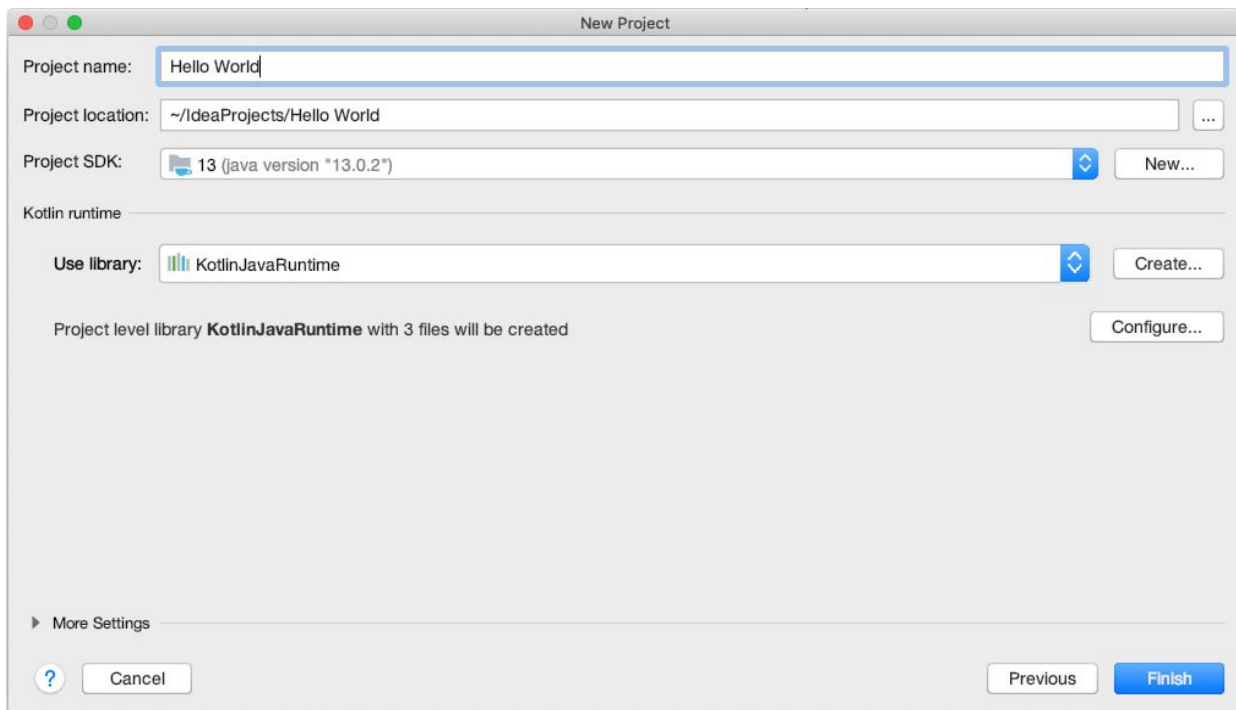
1) IntelliJ IDEA



Creare un nuovo progetto



Scegliere un nome

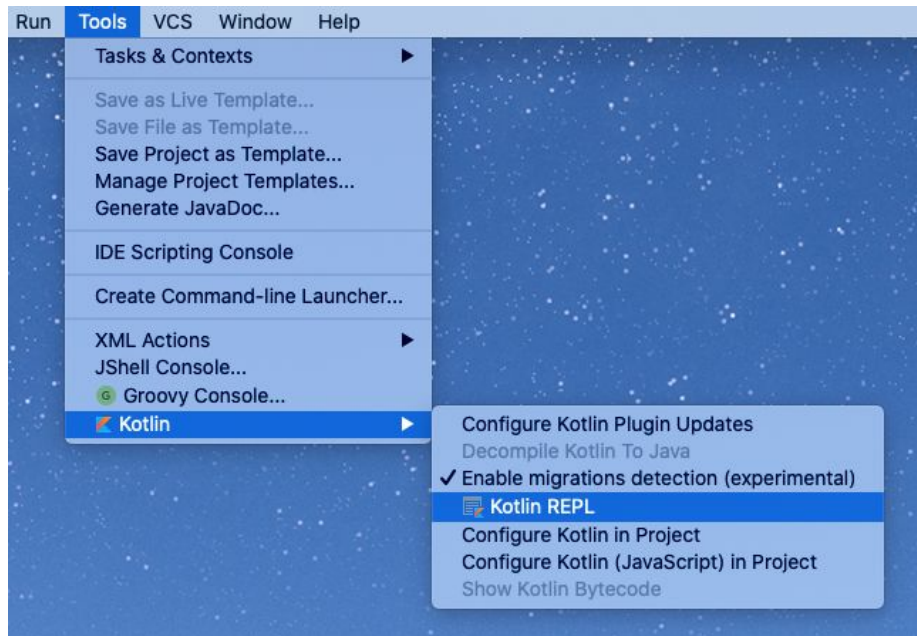


The screenshot shows the 'New Project' dialog box with the following fields and options:

- Project name:** Hello World
- Project location:** ~/IdeaProjects/Hello World
- Project SDK:** 13 (java version "13.0.2")
- Kotlin runtime:**
 - Use library:** KotlinJavaRuntime
 - Project level library **KotlinJavaRuntime** with 3 files will be created

Buttons at the bottom: ? (help), Cancel, Previous, and Finish.

Aprire REPL (Read-Eval-Print-Loop)



Creare una funzione printHello()

```
Run: Kotlin REPL (in module HelloKotlin) x
Welcome to Kotlin version 1.3.41 (JRE 11.0.2+9-LTS)
Type :help for help, :quit for quit

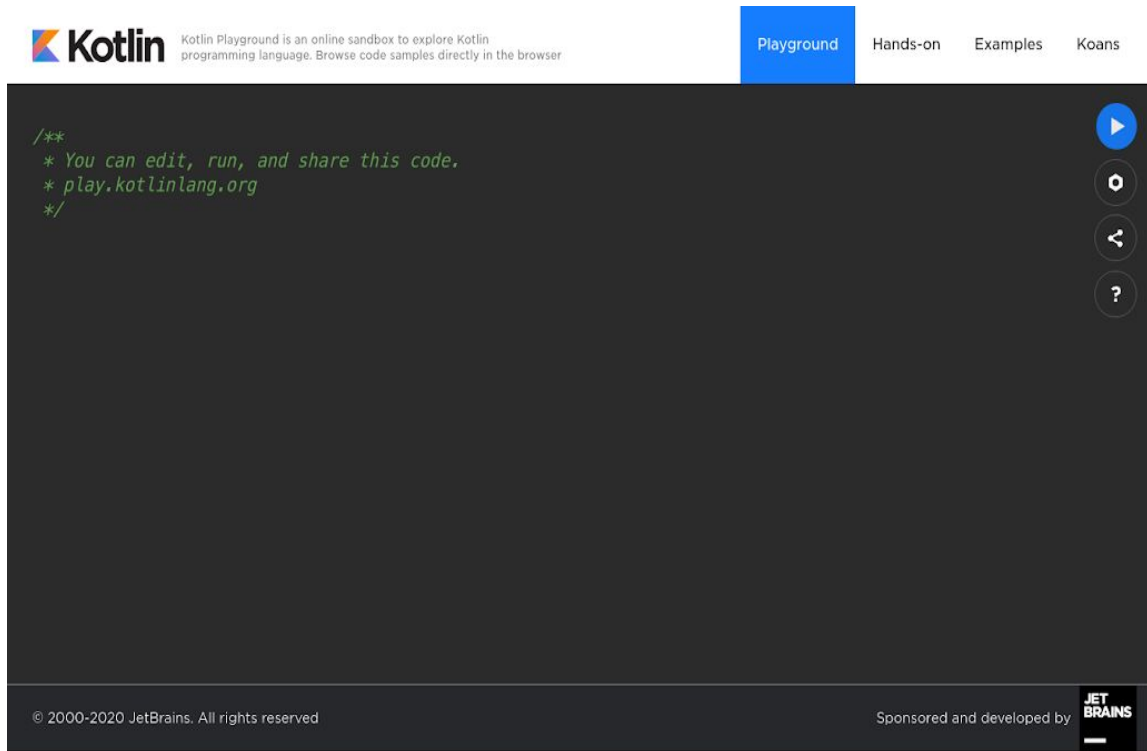
fun printHello() {
    println("Hello World")
}

printHello()
Hello World

<%=> to execute
```

Premere **Control+Enter**
(**Command+Enter** su
Mac) per eseguire

2) Kotlin playground



Operatori

Operatori

- Operatori matematici

+ - * / %

- Operatori di incremento e decremento

++ --

- Operatori di confronto

< <= > >=

- Operatori di assegnamento

=

- Operatori di uguaglianza

== !=

Operatori matematici con interi

$$1 + 1 \quad \Rightarrow \quad 2$$

$$53 - 3 \quad \Rightarrow \quad 50$$

$$50 / 10 \quad \Rightarrow \quad 5$$

$$9 \% 3 \quad \Rightarrow \quad 0$$

Operatori matematici con double

$1.0 / 2.0 \Rightarrow 0.5$

$2.0 * 3.5 \Rightarrow 7.0$

Operatori matematici

1+1

⇒ `kotlin.Int = 2`

1.0/2.0

⇒ `kotlin.Double = 0.5`

53-3

⇒ `kotlin.Int = 50`

2.0*3.5

⇒ `kotlin.Double = 7.0`

50/10

⇒ `kotlin.Int = 5`

Metodi per operatori numerici

Kotlin mantiene i numeri come dati primitivi, ma consente la chiamata a metodi sui numeri come se fossero oggetti.

```
2.times(3)
```

```
⇒ kotlin.Int = 6
```

```
3.5.plus(4)
```

```
⇒ kotlin.Double = 7.5
```

```
2.4.div(2)
```

```
⇒ kotlin.Double = 1.2
```


Data types

Integer types

Type	Bits	Note
Long	64	Da -2^{63} a $2^{63}-1$
Int	32	Da -2^{31} a $2^{31}-1$
Short	16	Da -32768 a 32767
Byte	8	Da -128 a 127

Floating-point e altri tipi numerici

Type	Bits	Note
Double	64	15 - 16 cifre decimali
Float	32	6 - 7 cifre decimali
Char	16	16-bit caratteri Unicode
Boolean	8	True o false. Le operazioni includono: - lazy disjunction, && - lazy conjunction, ! - negation

Tipi di operandi

I risultati delle operazioni mantengono il tipo degli operandi

`6*50`

`⇒ kotlin.Int = 300`

`1/2`

`⇒ kotlin.Int = 0`

`6.0*50.0`

`⇒ kotlin.Double = 300.0`

`1.0*2.0`

`⇒ kotlin.Double = 0.5`

`6.0*50`

`⇒ kotlin.Double = 300.0`

Type casting

Assegna un `Int` ad un `a Byte`

```
val i: Int = 6  
val b: Byte = i  
println(b)
```

⇒ error: type mismatch: inferred type is Int but Byte was expected

Converti un `Int` in un `Byte` con il casting

```
val i: Int = 6  
println(i.toByte())
```

⇒ 6

Underscore per numeri grandi

Usa gli underscore per rendere le costanti numeriche grandi più leggibili

```
val oneMillion = 1_000_000
```

```
val idNumber = 999_99_9999L
```

```
val hexBytes = 0xFF_EC_DE_5E
```

```
val bytes = 0b11010010_01101001_10010100_10010010
```

Stringhe

Le stringhe sono sequenze di caratteri racchiuse da doppi apici.

```
val s1 = "Hello world!"
```

Il testo di una stringa può contenere caratteri di escape.

```
val s2 = "Hello world!\n"
```

Oppure del testo qualsiasi se racchiuso da tre doppi apici ("")

```
val text = ""  
    var bikes = 50  
    ""
```

Concatenazione di stringhe: + o plus

```
val first = "Hello"
```

```
val second = " world"
```

```
val stringaFinale = first+second
```

oppure

```
val stringaFinale = first.plus(second)
```


Concatenazione di stringhe: StringBuilder

```
val builder = StringBuilder()  
    builder.append("Hello")  
        .append(" ")  
        .append("world")
```

Con StringBuilder non viene creato un nuovo oggetto String ogni volta che usiamo l'operatore di concatenazione, ma una sola volta alla fine. Utile se vanno concatenate molte stringhe tra loro.

String template

Una *template expression* inizia con un simbolo di dollaro (\$) e può contenere un singolo valore

```
val i = 10  
println("i = $i")  
=> i = 10
```

Oppure un'espressione all'interno di parentesi graffe:

```
val s = "abc"  
println("$s.length is ${s.length}")  
=> abc.length is 3
```

String template e concatenazione

```
val numberOfDogs = 3
```

```
val numberOfCats = 2
```

```
"I have $numberOfDogs dogs" + " and $numberOfCats cats"
```

```
=> I have 3 dogs and 2 cats
```

String template expressions

```
val numberOfShirts = 10
```

```
val numberOfPants = 5
```

```
"I have ${numberOfShirts + numberOfPants} items of clothing"
```

```
=> I have 15 items of clothing
```

Variabili

Variabili

- Potente meccanismo di inferenza di tipo (type inference)
 - Lascia che il compilatore inferisca il tipo
 - È possibile dichiarare il tipo esplicitamente
- Variabili mutable e immutable
 - Immutability non è obbligatoria, ma raccomandata

Kotlin è un linguaggio statically-typed. Il tipo è determinato a compile time e non può cambiare.

Specificare il tipo di una variabile

Colon Notation

```
var width: Int = 12
```

```
var length: Double = 2.5
```

Importante: una volta che il tipo è stato dichiarato esplicitamente o inferito dal compilatore, non può cambiare o viene generato un errore.

Variabili mutable (var) e immutable (val)

- Mutable (Changeable)

```
var score = 10
```

- Immutable (Unchangeable)

```
val name = "Jennifer"
```

Sebbene non strettamente obbligatorio, utilizzare variabili immutable è raccomandato nella maggior parte dei casi.

var e val: esempio

```
var count = 1
```

```
count = 2
```

```
val size = 1
```

```
size = 2
```

=> Error: val cannot be reassigned

Strutture condizionali

Control flow

Kotlin include diversi costrutti per implementare logica condizionale:

- Espressioni If/Else
- Espressioni When
- Cicli For
- Cicli While

Espressioni if/else

```
val numberOfCups = 30
```

```
val numberOfPlates = 50
```

```
if (numberOfCups > numberOfPlates) {  
    println("Too many cups!")  
} else {  
    println("Not enough cups!")  
}
```

=> Not enough cups!

Espressioni if con casi multipli

```
val guests = 30
if (guests == 0) {
    println("No guests")
} else if (guests < 20) {
    println("Small group of people")
} else {
    println("Large group of people!")
}
```

⇒ Large group of people!

Range

- Sono dei Data type che contengono un insieme di valori comparabili (ad esempio, gli interi da 1 a 100 inclusi)
- Sono intervalli chiusi, in cui si specifica il limite inferiore, due puntini ed il limite superiore (ad esempio, 1..100)
- Gli oggetti contenuti in un range possono essere mutable o immutable

Range nelle espressioni if/else

```
val numberOfStudents = 50
if (numberOfStudents in 1..100) {
    println(numberOfStudents)
}
```

=> 50

Nota: Non ci sono spazi attorno all'operatore di range (1..100)

Statement when

```
when (results) {  
    0 -> println("No results")  
    in 1..39 -> println("Got results!")  
    else -> println("That's a lot of results!")  
}
```

⇒ That's a lot of results!

Così come uno *statement* when, è possibile definire un'espressione when che fornisce un valore di ritorno



[Expression o statement?]

Espressioni e statement sono due concetti diversi:

- Un'**espressione** è qualsiasi porzione di codice ritorni un valore, *<<a combination of one or more explicit values, constants, variables, operators and functions that the programming language interprets and computes to produce another value.>>*
 - `1+1`
 - `sumOf(1,2,3)` (ciascuna funzione in Kotlin ritorna qualcosa, incluso `println()`)
 - `if (a>b) true else false` (a differenza di Java)
 - ```
val color = when {
 relax -> GREEN
 studyTime -> YELLOW
 else -> BLUE}
```

# [Expression o statement?]

Espressioni e statement sono due concetti diversi:

- Un **statement** è qualsiasi porzione di codice non produca valori di ritorno, *<<the smallest standalone element of an imperative programming language that expresses some action to be carried out.>>*
  - Dichiarazioni di variabili, es: `val x = 1`
  - Assegnamento, es: `x = 20` (a differenza di Java)
  - Dichiarazione di classi locali, es: `class A{}`

# Espressione when

```
val risultato = when {
 bmi < 18.5 -> "Sottopeso"
 bmi < 25 -> "Normopeso"
 else -> "Sovrappeso"
}
```

# Cicli for

```
val pets = arrayOf("dog", "cat", "canary")
for (element in pets) {
 print(element + " ")
}
```

⇒ dog cat canary

Non è necessario definire una variabile per gestire l'iterazione

# Cicli for

In Kotlin è possibile iterare direttamente su stringhe, array, range:

```
for (c in "I like Pink Floyd") {
 print(c)
}
```

⇒ I like Pink Floyd

**Nota:** La variabile `c` viene automaticamente dichiarata `val`, dunque non può essere modificata all'interno del ciclo.

# Cicli for: element e index

```
for ((index, element) in pets.withIndex()) {
 println("Item at $index is $element\n")
}
```

⇒ Item at 0 is dog

Item at 1 is cat

Item at 2 is canary

# Cicli for: step size e range

```
for (i in 1..5) print(i)
```

⇒ 12345

```
for (i in 5 downTo 1) print(i)
```

⇒ 54321

```
for (i in 3..6 step 2) print(i)
```

⇒ 35

```
for (i in 'd'..'g') print (i)
```

⇒ defg

# Cicli while

```
var bicycles = 0
while (bicycles < 50) {
 bicycles++
}
println("$bicycles bicycles in the bicycle rack\n")
⇒ 50 bicycles in the bicycle rack
```

```
do {
 bicycles--
} while (bicycles > 50)
println("$bicycles bicycles in the bicycle rack\n")
⇒ 49 bicycles in the bicycle rack
```



# repeat loops

```
repeat(2) {
 print("Hello!")
}
```

⇒ Hello!Hello!

# Jump expressions

- `return`: ritorna dalla funzione in cui ci troviamo
- `break`: termina il ciclo più interno in cui ci troviamo
- `continue`: forza la prossima iterazione del ciclo più interno in cui ci troviamo

Per `break` e `continue` è possibile usare delle `label` per determinare il ciclo da interrompere o di cui forzare l'iterazione

# Uso delle label

```
myLabel@ for (i in 1..100) {
 for (j in 1..100) {
 if (...) break@myLabel
 }
}
```

L'uso di `break@myLabel` forza a terminare il ciclo più esterno

# Uso delle label

```
myLabel@ for (i in 1..100) {
 for (j in 1..100) {
 if (...) continue@myLabel
 }
}
```

Nell'esempio, l'uso di `continue@myLabel` forza ad iniziare una nuova iterazione del ciclo più esterno

# Array

# Array

- Gli array memorizzano oggetti multipli
- Gli elementi di un array possono essere acceduti programmaticamente tramite i loro indici
- Gli elementi di un array sono mutable
- La dimensione di un array è fissa

# Array usando arrayOf()

Un array di stringhe può essere creato tramite `arrayOf()`

```
val pets = arrayOf("dog", "cat", "canary")
```

```
println(pets.contentToString())
```

```
println(java.util.Arrays.toString(pets)) // alternativa usando librerie Java
```

```
⇒ [dog, cat, canary]
```

Se un array è definito `val`, non si può cambiare il riferimento, ma è possibile cambiarne il contenuto.

# Array con tipi singoli o mixati


Un array in Kotlin può contenere tipi diversi:

```
val mix = arrayOf("hats", 2)
```

Oppure un tipo solo (ad esempio interi):

```
val numbers = intArrayOf(1, 2, 3)
```

Esistono funzioni specifiche per istanziare array di altri tipi



In questo caso l'array è di tipo `IntArray` e a run-time verrà implementato come un array di tipi primitivi (cioè come `int []`)



# Combinare array

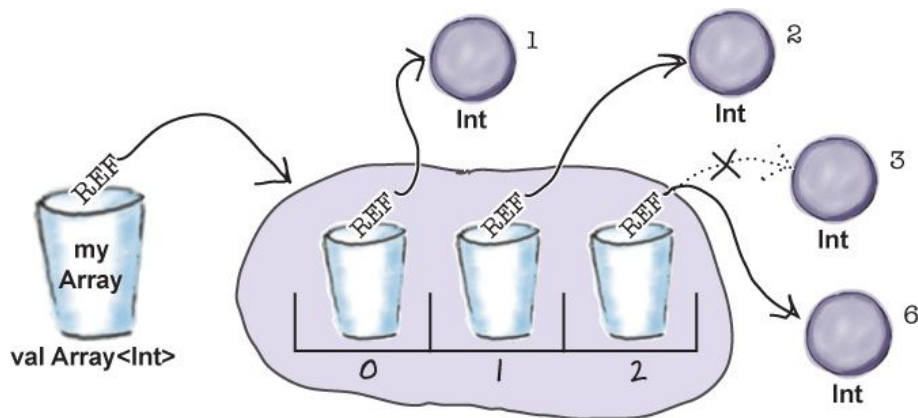
L'uso dell'operatore + consente di combinare due array:

```
val numbers = intArrayOf(1,2,3)
val numbers2 = intArrayOf(4,5,6)
val combined = numbers2 + numbers //oppure numbers2.plus(numbers)
println(Arrays.toString(combined)) //alternativa per la stampa, richiede di importare
 // la libreria java.util.Arrays

=> [4, 5, 6, 1, 2, 3]
```

# Modificare array

Se una variabile array è dichiarata `val` allora non potrà essere riassegnata, ma ciò non impedisce di modificare i suoi elementi.



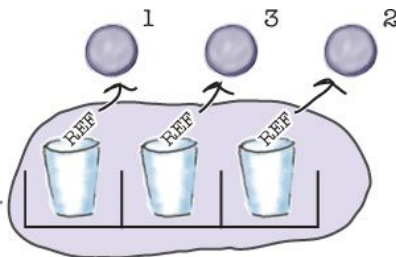
*The array itself can still be updated, even though the variable is declared using `val`.*

# Metodi utili



**Make an array:**

```
var array = arrayOf(1, 3, 2)
```



**Make an array initialized with nulls:**

```
var nullArray: Array<String?> = arrayOfNulls(2)
```

Creates an array of size 2 initialized with null values. It's like saying: `arrayOf(null, null)`



**Find out the size of the array:**

```
val size = array.size
```

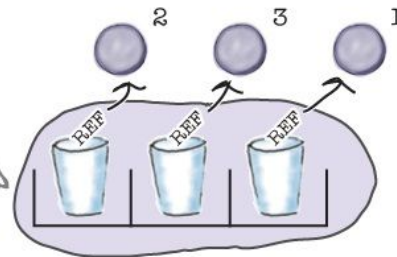
← array has space for three items, so its size is 3.



**Reverse the order of the items in the array:**

```
array.reverse()
```

← Flips the order of the items in the array.



**Find out if it contains something:**

```
val isIn = array.contains(1)
```

← array contains 1, so this returns true.

# Metodi utili



Calculate the sum of its items (if they're numeric):

`val sum = array.sum()` ← This returns *b* as  $2 + 3 + 1 = 6$ .



Calculate the average of its items (if they're numeric):

`val average = array.average()` ← This returns a *Double*—in this case,  $(2 + 3 + 1)/3 = 2.0$ .



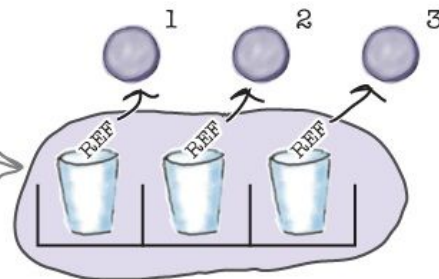
Find out the minimum or maximum item (works for numbers, Strings, Chars and Booleans):

`array.min()` } `min()` returns 1, as this is the lowest value in the  
`array.max()` } `array.max()` returns 3 as this is the highest.



Sort the array in a natural order (works for numbers, Strings, Chars and Booleans):

`array.sort()` ← Changes the order of the items in array so they go from the lowest value to the highest, or from false to true.



# Collections

# Collections in Kotlin

Le Collections sono utilizzate per memorizzare un numero variabile di oggetti (elementi o items) dello stesso tipo. Le principali sono:

- List
- Set
- Map

La Kotlin Standard Library offre interfacce, classi e funzioni generiche per creare, popolare e gestire collections di qualsiasi tipo.

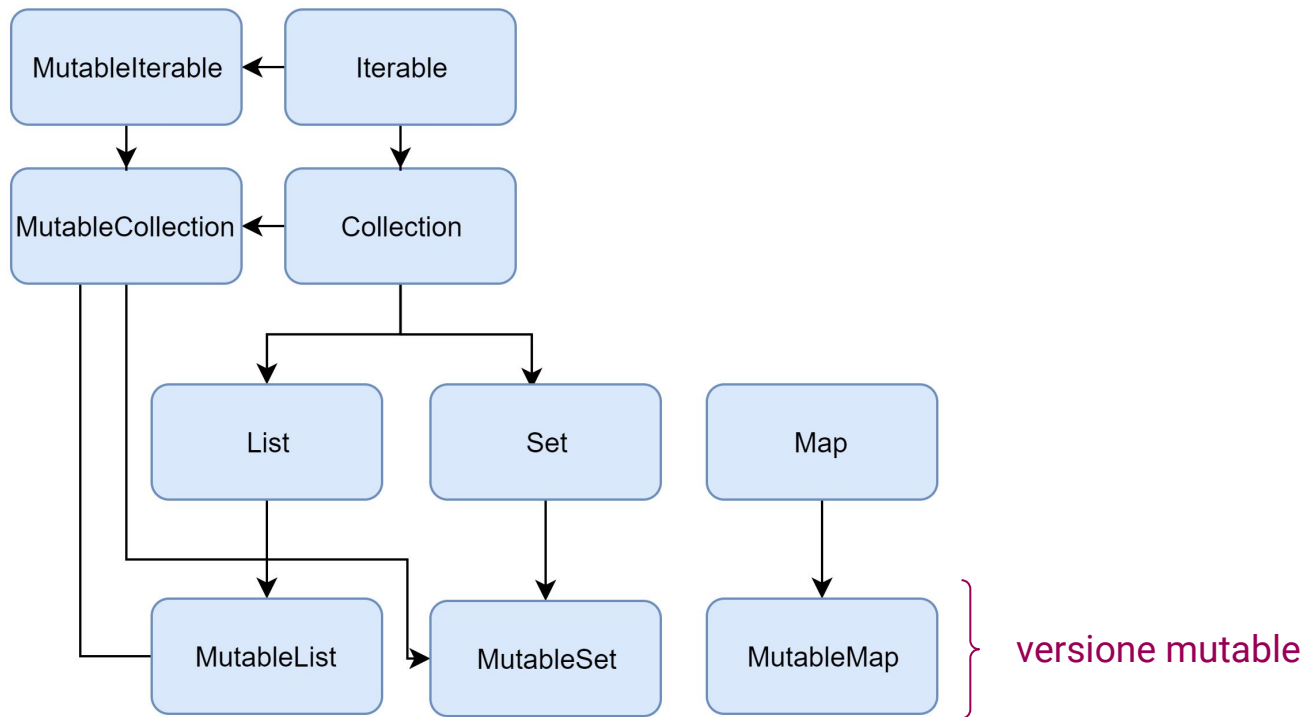
# Collections in Kotlin

Esistono due varianti per ciascun tipo di collection:

- *read-only*, in cui è possibile solo accedere agli elementi
- *mutable*, che estende le prime con metodi per aggiungere, modificare, rimuovere elementi

Attenzione: non stiamo parlando di `var` vs `val`, ma di collection che possono essere modificate oppure no

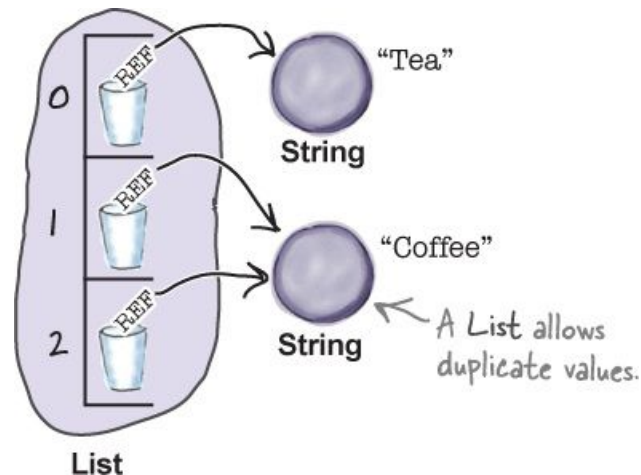
# Collections in Kotlin





# Liste

- Le liste sono collezioni ordinate di elementi
- Gli elementi di una lista possono essere acceduti programmaticamente per mezzo dei loro indici
- Gli elementi possono ripetersi
- Viene implementata di default come `ArrayList` (un array ridimensionabile)



Una frase è un esempio di lista: è un gruppo ordinato di parole che possono ripetersi.

# Immutable list usando listOf()

Dichiarazione di una lista usando `listOf()` e stampa a video:

```
val instruments = listOf("trumpet", "piano", "violin")
println(instruments)
println(instruments.size)
println(instruments[1])
println(instruments.get(2))
```

```
⇒ [trumpet, piano, violin]
 3
 piano
 violin
```

# Mutable list usando mutableListOf()

Una lista può essere definita mutable usando `mutableListOf()`

```
val myList = mutableListOf("trumpet", "piano", "violin")
myList.remove("violin")
```

⇒ `kotlin.Boolean = true`

Se una lista è dichiarata `val`, ciò che non può cambiare è l'oggetto a cui si riferisce la variabile, mentre il suo contenuto può farlo.

# List

Alcuni metodi utili:

- `add(elem: E) : Boolean` // aggiungere un elemento
- `addAll(col: Collection<E>): Boolean` // aggiunge una collection alla lista
- `set(ind: Int, elem: E)` // sostituisce l'elemento ind-esimo con elem
- `remove(elem: E)` // rimuove l'elemento indicato
- `removeAt(ind: Int)` // rimuove un elemento dalla posizione indicata
- `clear()` // svuota la lista
- `lastIndex()` //ritorna l'indice dell'ultimo elemento
- `subList(from:Int, to:Int)` //ritorna la mutableList nel range indicato
- `indexOf(elem: E)` // ritorna l'indice del primo elemento uguale a elem

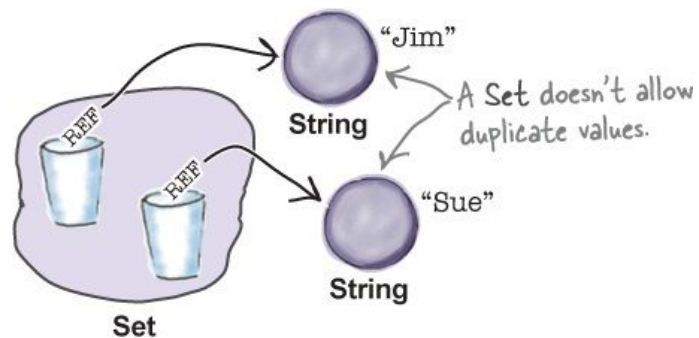
# Uguaglianza tra liste

Due liste sono uguali (==) se hanno la stessa dimensione ed elementi strutturalmente equivalenti nelle stesse posizioni:

```
val bob = Person("Bob", 31)
val people = listOf(Person("Adam", 20), bob, bob)
val people2 = listOf(Person("Adam", 20), Person("Bob", 31), bob)
println(people == people2)
bob.age = 32
println(people == people2)
⇒ true
 false
```

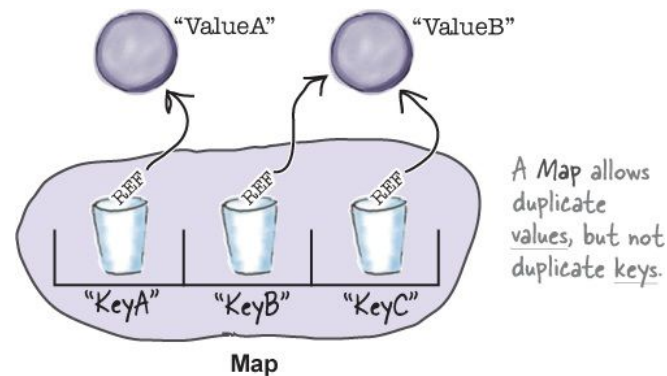
# Set

- Set viene usato per memorizzare elementi unici senza duplicazioni (anche il null può comparire una sola volta)
- Due set sono uguali se hanno la stessa dimensione e gli stessi elementi (in qualsiasi ordine)
- Set vs MutableSet
- L'implementazione di default è quello del `LinkedHashSet` che mantiene l'ordine degli elementi (opzionalmente si può usare `HashSet` che non mantiene un ordine ma è più efficiente)



# Map

- Sebbene non erediti dalla classe Collection, è comunque un tipo particolare di collection, che memorizza coppie key-value, dove le key sono uniche
- Map vs MutableMap
- Una Map viene implementata come `LinkedHashMap`, che mantiene l'ordine di inserimento degli elementi, mentre `HashMap` non lo fa



# Map: esempio

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2)

numbersMap.put("three", 3)

numbersMap["one"] = 11

println(numbersMap)

println("All keys: ${numbersMap.keys}")

println("All values: ${numbersMap.values}")

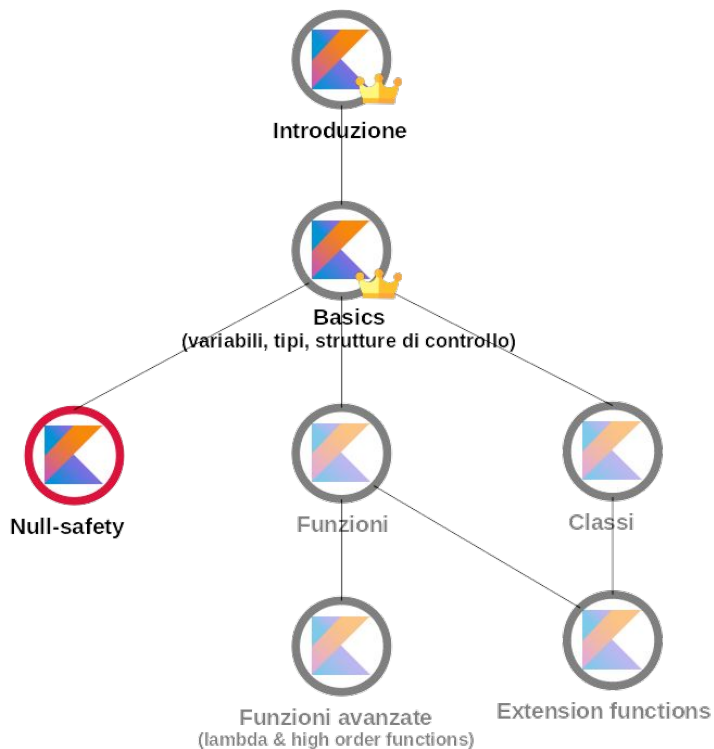
if ("key2" in numbersMap) println("Value by key \"key2\": ${numbersMap["key2"]}")

if (1 in numbersMap.values) println("The value 1 is in the map")

if (numbersMap.containsValue(1)) println("The value 1 is in the map")
```



# Kotlin



# Null safety

# Null safety

- Molti linguaggi, come il Java, consentono ad una variabile di assumere il valore `null`.
- Se il null viene trattato come un valore normale, possono verificarsi problemi critici:
  - In Java viene lanciata una `NullPointerException` (o NPE)
  - In C un *null pointer* porta ad un crash del software o del S.O.

# Null safety



*<<I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (**ALGOL W**). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.>>*

Tony Hoare

<https://youtu.be/YYkOWzrO3xg?t=1651>

# Null safety

Due opzioni:

1. Non consentire mai valori *null*, e al contrario prevedere un valore speciale che indichi la mancanza di valori (impossibile per Kotlin, perché deve essere interoperabile con Java)
2. Di default, i tipi non possono essere mai nulli (cioè sono *non-nullable*). Consentire un meccanismo per dichiarare esplicitamente dei tipi che potrebbero contenere valori nulli (cioè *nullable*)



Il miglior compromesso per Kotlin

# Null safety



# Null safety

- In Kotlin, le variabili non possono essere nulle per definizione
- E' possibile assegnare esplicitamente un valore null ad una variabile tramite il safe-call operator "?"
- E' possibile testare la presenza di valori nulli con l'operatore elvis "? :"
- E' possibile consentire null-pointer exceptions tramite l'operatore "!!"

# Le variabili non possono essere null

In Kotlin, di default non è possibile assegnare `null` ad una variabile.

Proviamo a dichiarare una variabile `Int` e assegnarle il valore `null`

```
var numberOfBooks: Int = null
```

⇒ error: null can not be a value of a non-null type Int



# Safe call operator

Dichiariamo la variabile di tipo `Int?` come “nullable”:

```
var numberOfBooks: Int? = null
```

Il *safe call operator* (?) dopo il tipo indica che una variabile potrebbe essere `null`. In questo modo stiamo specificando un tipo diverso.

In generale, è meglio non settare mai una variabile a `null` perché potrebbe comportare conseguenze indesiderate.

# Safe call operator

Se una variabile è nullable, non è possibile dereferenziarla (cioè accedere all'oggetto a cui punta) direttamente.

```
val s1: String = "abc"
val s2: String? = s1
```

```
println(s2.length) // non compila
```

Prima del dereferenziamento è necessario verificare che non sia nulla.

# Testare valori null

Controlliamo se la variabile `numberOfBooks` non è `null`. Poi decrementiamo la variabile:

```
var numberOfBooks = 6
if (numberOfBooks != null) {
 numberOfBooks = numberOfBooks.dec()
}
```

Si può fare la stessa cosa in modo più conciso usando il safe call operator:

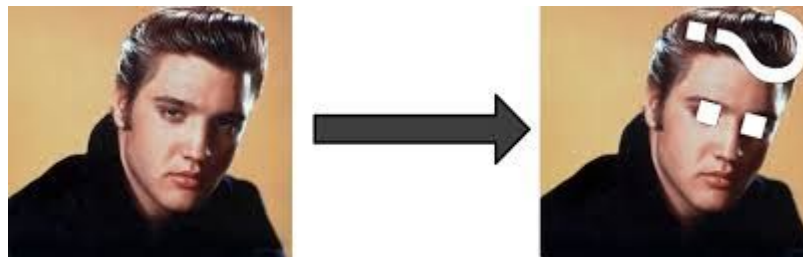
```
var numberOfBooks = 6
numberOfBooks = numberOfBooks?.dec()
```

# L'operatore Elvis

Alcuni linguaggi hanno un *null coalescing operator*, che testa se una variabile è nulla, e in caso produce un valore convenzionale.

In Kotlin, l'operatore (`? :`) consente di determinare un valore alternativo valido nel caso in cui la variabile sia nulla:

```
numberOfBooks = numberOfBooks?.dec() ?: 0
```



# Non-Null assertion (!!)

Se siamo certi che una variabile non sarà nulla, possiamo usare una *non-null assertion (!!)* per forzare l'interpretazione della variabile come non nullable (una sorta di cast). A questo punto è possibile eseguire qualsiasi metodo su di essa.

```
val len = s!!.length
```



*throws NullPointerException if s is null*

**Warning:** Dato che !! lancia un'eccezione nel caso la variabile sia nulla, sarebbe preferibile usarla di rado.

# Null safety in breve

