Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Array in C
vs
Array nei linguaggi a oggetti

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

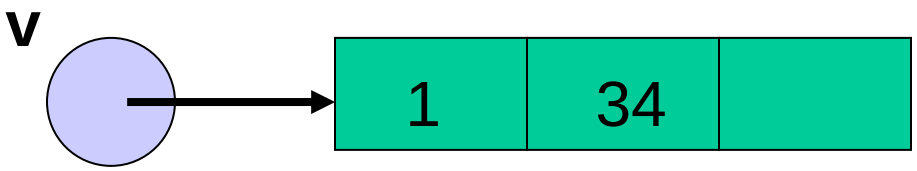
ARRAY IN C

In C, gli array sono.. *un'illusione ottica!*

- non esistono *veri array* come entità dotate di nome
- esistono solo *aree di memoria di cui è noto l'indirizzo iniziale*
- il *nome* è solo un sinonimo del *puntatore al primo elemento*

Ciò crea un *mix improprio* fra i concetti di *array* e *puntatore* che emerge in molti momenti, creando confusione**.**

Il nome NON è riferito all'array come tutt'uno, è solo un puntatore al primo elemento



ARRAY IN C: CONSEGUENZE

In conseguenza di ciò:

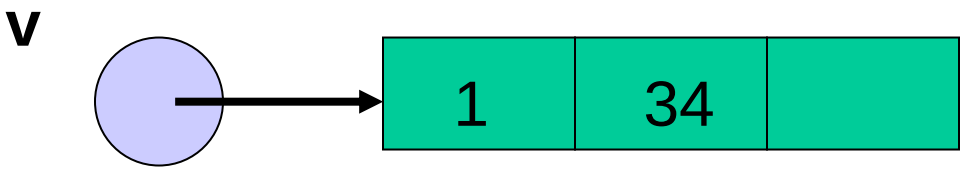
- un array viene passato a una funzione *per indirizzo*, quando tutti gli altri tipi di dati sono passati per valore
  - *mancanza di coerenza* nella gestione dei tipi
- assegnamenti fra array (come `v1 = v2`) sono *illegali*
  - per copiare un array in un altro bisogna *copiare ogni elemento*
- *non si può restituire un array* come risultato di una funzione (si deve restituire un puntatore al primo elemento)
- *non si può sapere quanti elementi contenga un array* passato come argomento, poiché l'unica informazione realmente trasferita è il suo indirizzo iniziale

Un costrutto "nato dal basso", linguisticamente mal definito

ARRAY IN C: CURIOSITÀ

- Il punto debole degli array C è che *il nome non denota tutta l'entità, ma solo una sua parte* (l'indirizzo iniziale)

```
int v[3];
```

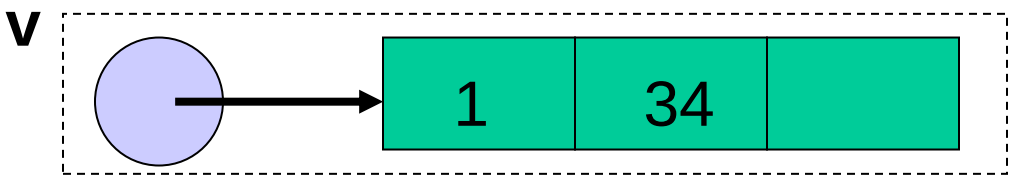


ARRAY IN C: CURIOSITÀ

- Il punto debole degli array C è che *il nome non denota tutta l'entità, ma solo una sua parte* (l'indirizzo iniziale)
- A riprova di ciò, *chiudendoli in una* ***`struct`****, cambia tutto!*
  - sebbene sia grande uguale, ora l'array passa per valore, è restituibile da una funzione e ammette l'assegnamento!

```
struct {
  int value[3];
} array3;
array3 v;
```

Il costrutto **`struct`** fornisce *l'involucro esterno* che manca, offrendo un *nome* che denoti il *tutto*



ARRAY IN C: CURIOSITÀ

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int v[3];
} array3;

void printArray(array3 a){
    for (int i=0; i<3; i++) printf("%d,", a.v[i]);
    printf("\n");
}

void main() {
    printf("Hello World\n");
    int v1[3] = {1,2,3}, v2[3];
    /*v2 = v1;*/
    array3 v3={v:{1,2,3}}, v4;
    v4 = v3;
    printArray(v4);
    printf("Bye bye World\n");
}
```

v1 e **v2** sono due array classici: l'assegnamento **v2=v1** è illegale

v3 e **v4** sono invece due struct: l'assegnamento **v4=v3** è legale!

input

```
Hello World
1,2,3,
Bye bye World


...Program finished with exit code 0
Press ENTER to exit console.
```

ARRAY: APPROFONDIMENTI

Per approfondire l'argomento:

- Esercitazione autonoma *«Array Java venendo dal C»*

COSA VORREMMO?

Una nuova nozione di array

- *pienamente integrata* nel paradigma a oggetti
- *totalmente separata* dal concetto di puntatore
- *uniforme* rispetto agli altri tipi del linguaggio

Un nuovo tipo-array rappresentato come *classe*

- una classe di nome *"Array of"* (..o quasi..)
- dotata di proprietà – la *lunghezza* (o dimensione)
- manipolabile tramite riferimenti come ogni altro oggetto, *con una semantica coerente*
  - possibilità di assegnare riferimenti per condividere oggetti
  - possibilità di restituire riferimenti come risultato di funzioni
  - …

COSA VORREMMO?

Una nuova nozione di array

- *pienamente integrata* nel paradigma a oggetti
- *totalmente separata* dal concetto di puntatore
- *uniforme* rispetto agli altri tipi del linguaggio

In Java e C#, al termine *Array* si è preferita la keyword **`[]`**

- comprensibile, per retro-compatibilità / abitudine rispetto al C
- in realtà, *molto inopportuna* rispetto alle altre collection (liste, alberi..)
- non a caso, approccio rivisto in linguaggi più recenti (Scala, Kotlin)

Esempi in Java e C#

- array di interi: non **`ArrayOfInt`** ma **`int[]`**
- array di **`Counter`**: non **`ArrayOfCounter`** ma **`Counter[]`**
- array di **`Frazione`**: non **`ArrayOfFrazione`** ma **`Frazione[]`**

COSA VORREMMO?

Una nuova nozione di array

- *pienamente integrata* nel paradigma a oggetti
- *totalmente separata* dal concetto di puntatore
- *uniforme* rispetto agli altri tipi del linguaggio

In Scala e Kotlin, si è invece scelta la keyword **`Array`**

- uniformità con le altre collection (liste, alberi..)
- coerenza con l'eliminazione dei tipi primitivi (*everything is an object*)

Esempi in Scala e Kotlin

- array di interi: Scala: **`Array[Int]`** Kotlin: **`Array<Int>`**
- array di **`Counter`**: Scala: **`Array[Counter]`** Kotlin: **`Array<Counter>`**
- array di **`Frazione`**: Scala: **`Array[Frazione]`** Kotlin: **`Array<Frazione>`**

NB: per ragioni di efficienza, su JVM Kotlin offre anche tipi di array specializzati per i tipi base: **`IntArray`**, **`LongArray`**, **`DoubleArray`**, ecc.

UN NUOVO CONCETTO DI ARRAY

- Gli array Java e C# sono *oggetti,* istanze di una *classe speciale* denotata da `[]`
- Quindi, come per ogni oggetto:
  1. prima si definisce un *riferimento*
  2. poi *si crea dinamicamente l'istanza*

```
int[] v;          // assomiglia a ArrayOfInt v
Counter[] w;      // assomiglia a ArrayOfCounter w
```

```
v = new int[3];
w = new Counter[6];
```

Sintassi idealmente equivalente a **`new int[](3)`**

che assomigliano a:

```
v = new ArrayOfInt(3);
w = new ArrayOfCounter(6);
```

UN NUOVO CONCETTO DI ARRAY

- Gli array Java e C# sono *oggetti,* istanze di una *classe speciale* denotata da `[]`
- per c
  lefinis
  a dir

È un riferimento: come tale, *non specifica la dimensione!*

In Java le parentesi quadre `[]` possono essere o dopo il nome (come in C) o di seguito al tipo

In C#, sempre di seguito al tipo

```
int[] v;          // assomiglia a ArrayOfInt v
Counter[] w;      // assomiglia a ArrayOfCounter w
```

```
v = new int[3];          // array di 3 int
w = new Counter[6];      // array di 6 Counter
```

```
Int(3);
Counter(6);
```

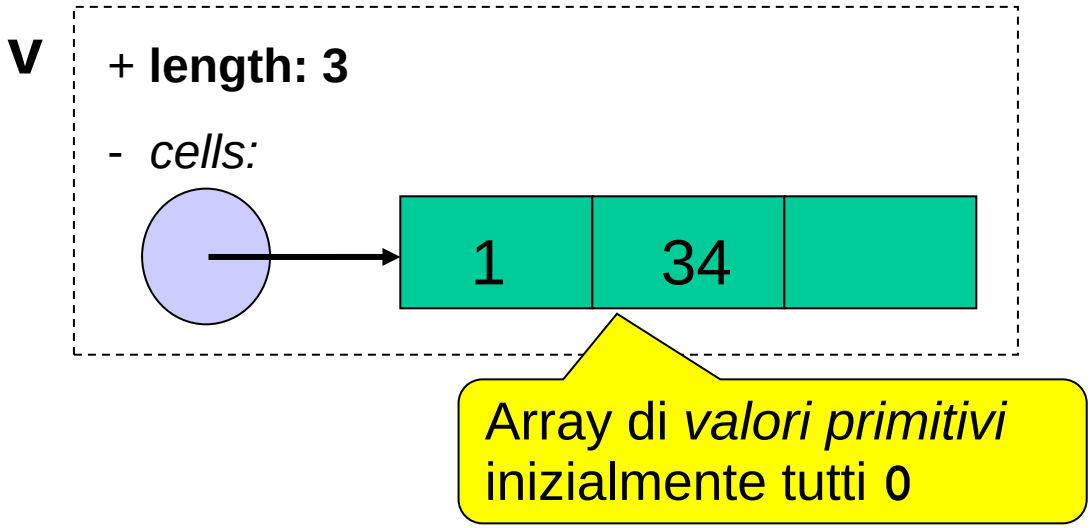
La dimensione effettiva *si specifica all'atto della creazione* dell'oggetto-array

ARRAY DI TIPI PRIMITIVI vs. ARRAY DI OGGETTI

Ogni elemento dell'array:

- è una *variabile primitiva*, se gli elementi sono di un *tipo primitivo* (`int`, `float`, `char`, ...)

```
v = new int[3];
```

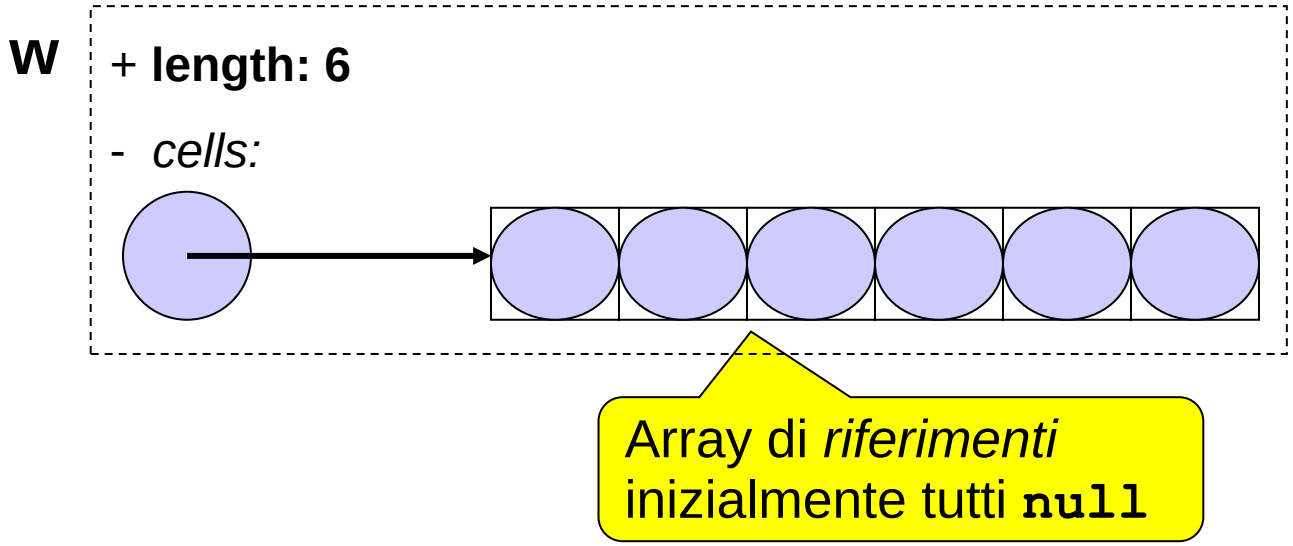


ARRAY DI TIPI PRIMITIVI vs. ARRAY DI OGGETTI

Ogni elemento dell'array:

- è un *riferimento a un (futuro) oggetto,* se gli elementi sono *(riferimenti a) oggetti*

```
w = new Counter[6];
```



ARRAY DI TIPI PRIMITIVI vs. ARRAY DI OGGETTI

Confronto

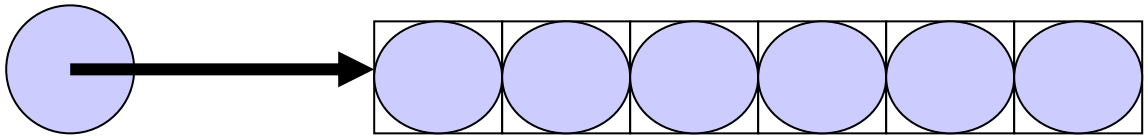
- Array di *tipi primitivi* (`int`, `float`, `char`, ...)

```
v = new int[3];
```

Singole variabili int, float, etc.

- Array di *tipi oggetto*

```
w = new Counter[6];
```



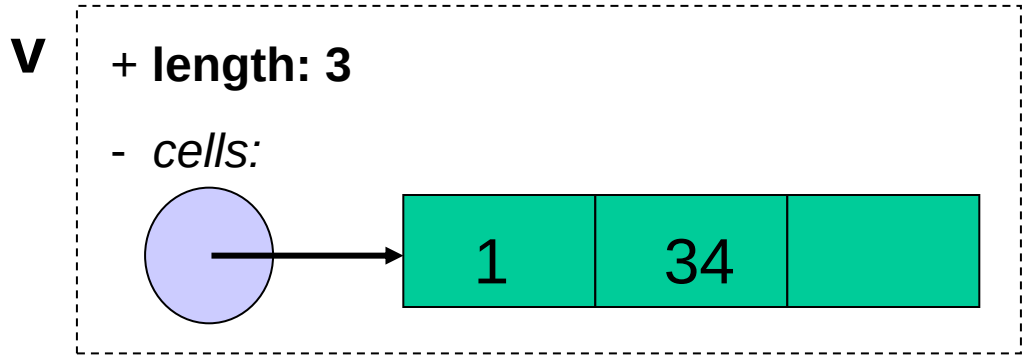
Singoli riferimenti (inizialmente **`null`**)

ARRAY DI TIPI PRIMITIVI

Quindi, in Java e C#:

- nel caso di tipi primitivi, ogni elemento dell'array è una *normale variabile*, *già usabile "così com'è":*

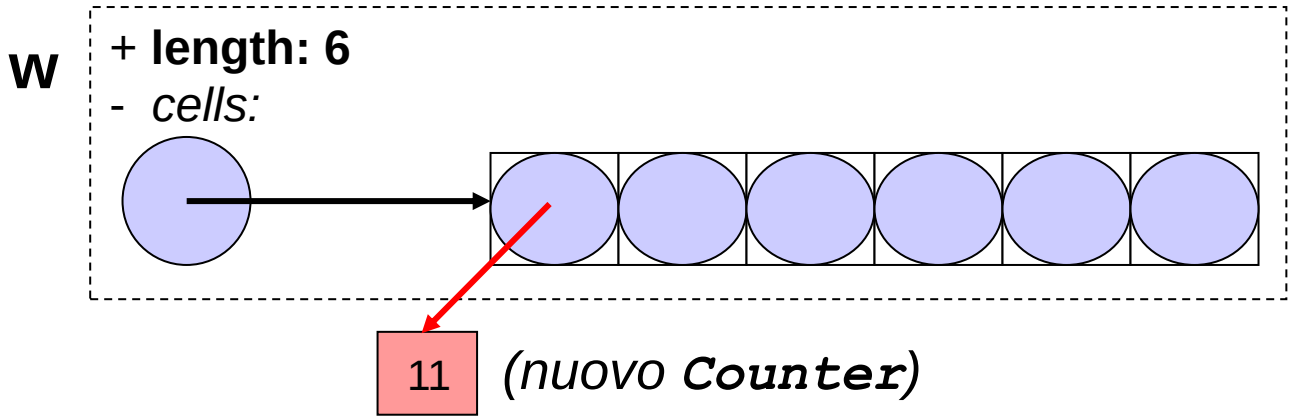
```
v = new int[3];
v[0] = 1; v[1] = 34;
```



ARRAY DI OGGETTI

- nel caso di tipi oggetto, invece, i singoli oggetti devono essere *creati uno ad uno* *(a meno che il riferimento non punti a un oggetto già esistente)*

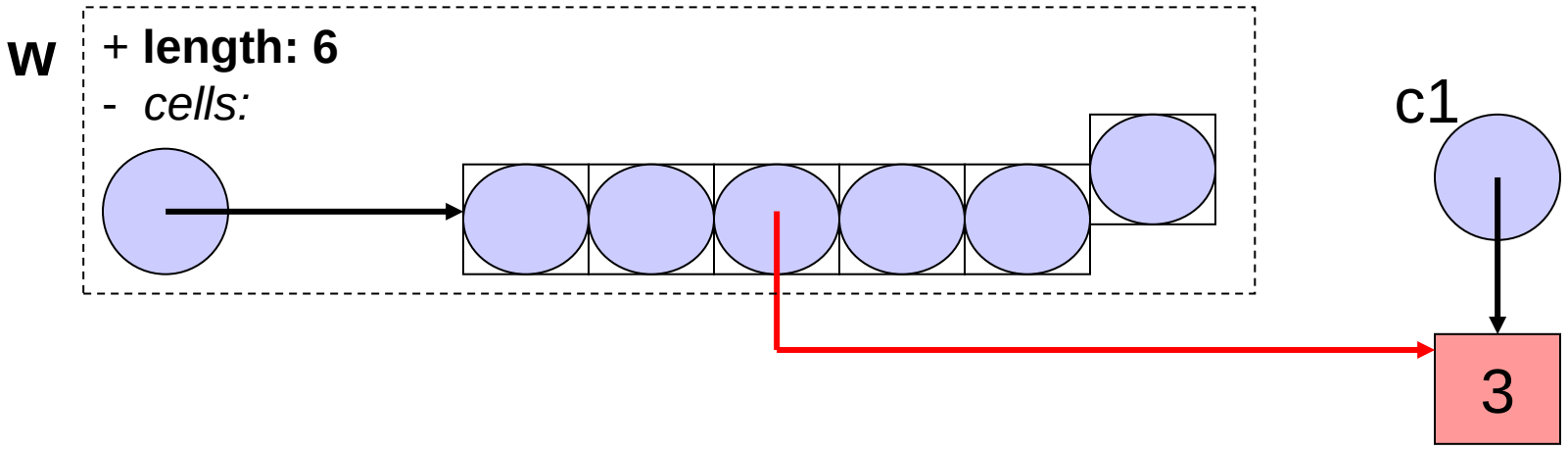
```
w = new Counter[6];
w[0] = new Counter(11);
```



ARRAY DI OGGETTI (segue)

- ovviamente, la creazione dell'oggetto non è necessaria se si voglia inserire nell'array un oggetto preesistente:

```
c1 = new Counter(3);
...
w = new Counter[6];
w[2] = c1;      // c1 è preesistente
```



LA PROPRIETÀ `length` (o `Length`)

- La proprietà **`length`** (**`Length`** in C#) rappresenta la lunghezza *(dimensione fisica)* dell'array:

  `v.length` vale 3

  `w.length` vale 6

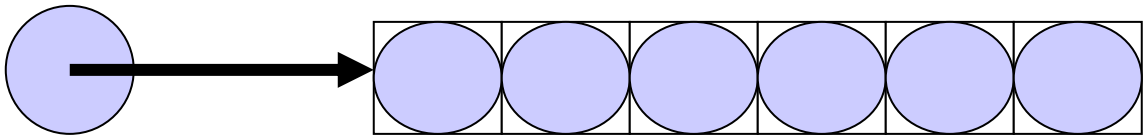
- Importante:
  - una volta creato, l'array ha lunghezza fissa: *non* può essere "allungato" a piacere
  - per tali necessità si usano le liste (interfaccia **`List`** e classi **`ArrayList`** e **`LinkedList`**)

ARRAY IN Scala & Kotlin

- Come già accennato, in Scala e Kotlin, la classe «array» si chiama proprio **`Array`**, non `[]` come in Java e C#

in Scala	*in Kotlin*
`Array[Int]`	`Array<Int>`
`Array[Frazione]`	`Array<Frazione>`

- Soprattutto, in Scala e Kotlin *scompare la differenza fra array di tipi primitivi e non*
  - per forza: non esistono più i tipi primitivi!
  - unico modello:



  - però, in Kotlin esistono anche **`IntArray`**, **`LongArray`**, **`DoubleArray`**.. che sotto sotto vengono mappati su array primitivi Java (per performance)

ARRAY IN Scala & Kotlin

- Accesso alle celle
  - Kotlin usa le [ ], come Java e C#
  - Scala invece usa le parentesi tonde ( ), perché le [ ] sono usate per la specifica di tipo

Scala	*Kotlin (Java, C#)*
`v(0) = 35;`	`v[0] = 35;`
`x = v(0);`	`x = v[0];`

- La proprietà «lunghezza» dell'array
  - Java e C# usano il nome **`length`**
  - Scala mantiene tale nome, ma aggiunge anche l'alias **`size`**
  - Kotlin la rinomina **`size`** per analogia con le altre collection

INIZIALIZZAZIONE DI ARRAY

- Anziché creare array «vuoti» (=valori di default) per poi riempirli coi contenuti desiderati, è possibile
  - in Kotlin, obbligatorio

  specificare direttamente il valore iniziale
  - in Java e C#, tramite *espressioni di inizializzazione* della forma **`{elemento1, elemento2, … , elementoN}`**
  - in Scala, tramite il *metodo* di creazione & inizializzazione **`Array(elemento1, elemento2, … , elementoN)`**
  - in Kotlin, tramite il *metodo* di creazione & inizializzazione **`arrayOf(elemento1, elemento2, … , elementoN)`**
- Ciò causa la creazione di un *nuovo* array *esattamente come se fosse stata usata la new:* è semplicemente uno shortcut

INIZIALIZZAZIONE DI ARRAY

- In Java e C#, all'atto della costruzione, gli array vengono inizializzati automaticamente con *valori di default*
  - `0` per array di tipi interi (int, long, short), `0.0F` per array di `float`, `0.0` per array di `double`, `false` per array di `boolean`, `'\0'` per array di `char`
  - `null` per array di tipi oggetto

```
jshell> char[] v2 = new char[4];
v2 ==> char[4] { '\000', '\000', '\000', '\000' }

jshell> boolean[] v3 = new boolean[4];
v3 ==> boolean[4] { false, false, false, false }

jshell> String[] q = new String[3];
q ==> String[3] { null, null, null }

jshell> int[] v = new int[3];
v ==> int[3] { 0, 0, 0 }
```

Java

C#

INIZIALIZZAZIONE DI ARRAY
ESEMPI in Java e C#

- Inizializzazione di array di valori primitivi:

```
int[] v1 = {2,3,4}, v2 = {2,3,5},
      v3 = {2,3},   v4 = {2,3,4};
```

Java C#

- Inizializzazione di array di oggetti:

```
Counter[] w1 = {
   new Counter(2),new Counter(3)};

Counter[] w2 = {
   new Counter(2),new Counter(3),new Counter(4)};
```

Java C#

NB: le espressioni di inizializzazione (come il loro nome indica) si possono usare *solo* al momento della definizione iniziale dell'array – non dopo, magari per passare argomenti

INIZIALIZZAZIONE DI ARRAY CONTROESEMPI in Java e C#

- Non si può passare un array costante «al volo» così:

```
void f(int[] v) { … }

f({2,3,4}); // NO, errato!
```

Java C#

```
void g(Frazione[] v) { … }

g({new Frazione(3,2), new Frazione(3,4)});
  // NO, errato!
```

Java C#

- Si può però costruire «al volo» un *nuovo* array, inizializzandolo nello stesso momento, in questo modo:

```
f( new int[]{2,3,4}); // OK

g( new Frazione[]{
      new Frazione(3,2), new Frazione(3,4)}); // OK
```

Java C#

INIZIALIZZAZIONE DI ARRAY
ESEMPI in Scala e Kotlin

- Scala

```
val v : Array[Int] = new Array[Int](3);
v(0) = 1; v(1) = 34;

val v : Array[Int] = Array(1,34);
```

Shortcut: `Array.ofDim(3)`

Inizializzazione implicita: tutti gli elementi sono posti al rispettivo default (per gli interi è il valore 0)

Alternativa: costruzione con contemporanea inizializza-zione (metodo statico Array)

- Kotlin (inizializzazione implicita *non* ammessa)

```
val v : Array<Int> = Array<Int>(3,{1});
v[0] = 1; v[1] = 34;

val v : Array<Int> = arrayOf(1,34);
```

Inizializzatore che pone tutti gli elementi a 1

Alternativa: costruzione con contemporanea inizializzazione (metodo statico arrayOf)

ARRAY: IL PROBLEMA `toString`

- Come qualunque classe, anche la classe "Array" (ossia, `[]`) ha una sua **`toString`** predefinita, che produce la "solita" stringa alfanumerica della forma *nomeclasse@identificativo*
  - in **`Counter`**, la **`toString`** predefinita produceva una stringa del tipo **`Counter@fc43de`**
  - analogamente, in "array" (ossia, `[]`) la **`toString`** produce una stringa del tipo **`[I@6659c656`**

```
jshell> int[] v = new int[3];
v ==> int[3] { 0, 0, 0 }

jshell> String s = v.toString();
s ==> "[I@5f4da5c3"

jshell> System.out.println(v)
[I@5f4da5c3
```

Usa la **`toString`** predefinita

- Poiché `[]` è una classe di sistema, non può essere modificata dall'utente → quella **`toString`** *non è personalizzabile* ☹

ARRAY: IL PROBLEMA `toString`

- Per ottenere una rappresentazione stringa del contenuto di un array, occorre quindi ricorrere ad altre soluzioni:
  1. scriversi un ciclo "classico" a mano
     → *prossimi esempi*
  2. sfruttare (se esiste) una qualche funzione di libreria
     → *ad es. quelle della libreria* ***`java.util.Arrays`*** *(v. oltre)*
  3. convertire (solo se utile) l'array in una struttura dati diversa, che abbia una **`toString`** migliore
     → *ad es. in certi casi, liste (come si vedrà in seguito)*

ESEMPIO 1 (1/5)

Stampare a video gli argomenti passati dalla linea di comando.

Serve un ciclo, non possiamo stamparlo "tutto insieme" con un'unica `println`

Il main riceve un *array di stringhe*

Java: `public static void main(String[] args)`

C#: `public static void Main(string[] args)`

Scala: `def main(args : Array[String]) : Unit`

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

- ogni elemento di **`args`** è un (riferimento a un) oggetto stringa
- a differenza del C, non c'è l'argomento extra **`argc`**, perché la dimensione dell'array è già presente nell'array, nella forma della proprietà **`length`** (in C#, **`Length`**; in Kotlin, **`size`**)

ESEMPIO 1 (2/5)

Java

```
public class EsempioMain {
 public static void main(String[] args){
   if (args.length == 0)
      System.out.println("Nessun argomento");
   else
      for (int i=0; i<args.length; i++)
        System.out.println("argomento " + i
                           + ": " + args[i]);
  }
}
```

L'operatore **+** concatena **`String`** *e valori di tipi primitivi,* che vengono automaticamente convertiti in **`String`** tramite il metodo statico **`String.valueOf`**

ESEMPIO 1 (3/5)

Java

```
public class EsempioMain {
  p
    else
       for (int i=0; i<args.length; i++)
         System.out.println("argomento " + i
                            + ": " + args[i]);
    }
}
```

Si possono definire variabili *in ogni punto del programma* (non solo all' inizio come in C): in particolare è possibile definire l'indice `i` *dentro* al ciclo `for`

Visibilità *limitata al corpo del ciclo*: fuori da esso, `i` risulta indefinita
→ *ottimizza hardware*

A differenza del C, `args[0]` è il primo argomento *(non il nome del programma)*

ESEMPIO 1 (4/5)

Versione C# *(in colore le differenze)*:

C#

```
public class EsempioMainCSharp{
 public static void Main(string[] args){
   if (args.Length==0)
    System.Console.WriteLine("Nessun argomento");
   else
    for (int i=0; i<args.Length; i++)
     System.Console.WriteLine("argomento " + i
                  + ": " + args[i]);
  }
}
```

ESEMPIO 1 (5/5)

Le versioni Scala & Kotlin sono analoghe, ma il ciclo for ha una sintassi innovativa

- Java C#

```
for (int i=0; i<10; i++) …
```

Controllo *diretto* dell'indice e del suo incremento → error-prone, bug a go-go

- Scala

```
for (i <- 0 to 10) …
```

- Kotlin

```
for (i in 0..10) …
```

Controllo *indiretto* dell'indice: si specifica solo l'intervallo, delegando la gestione dei dettagli

Lo sviluppo dell'esempio in Scala e Kotlin è lasciato per esercizio al lettore ☺

ESEMPIO 1: ESECUZIONE

Esecuzione:

```
C:> java EsempioMain   34 e 56 "io e te"
C:> EsempioMainCSharp  34 e 56 "io e te"
C:> scala  EsempioMain 34 e 56 "io e te"
C:> kotlin EsempioMain 34 e 56 "io e te"
```

Output:

```
argomento 0: 34
argomento 1: e
argomento 2: 56
argomento 3: io e te
```

ARRAY RESTITUITI DA FUNZIONI

- In C, un array *è* il solo tipo che *non* può essere restituito da una funzione
  - solito motivo: non esiste l'array in quanto tale
  - si può solo restituirne l'indirizzo iniziale
- Nei linguaggi OOP, gli array *possono essere restituiti* come risultato, come ogni altro oggetto

  ESEMPIO: una funzione Java che crei e restituisca una tabella (array) "opportunamente riempita"

  ```
  int[] creaTabella(int n)
  ```

ESEMPIO 2: `creaTabella`

- DUBBIO: è un metodo o una funzione statica?
  - per rispondere occorre valutare «a chi» verrebbe attribuita tale funzionalità, ossia «a chi» chiederemmo di farlo
- Ragionamento
  - un metodo viene invocato su uno specifico oggetto, che deve essere preventivamente creato (`c1.inc()`, `f2.getNum()`, etc.): in questo caso, chi sarebbe?
  - una funzione statica («di libreria») viene invece invocata semplicemente col suo nome assoluto (`Math.sin()`): non occorre creare nulla preventivamente, perché la classe già esiste
- Nel caso di `creaTabella`:
  - non c'è uno specifico oggetto «creatore di tabelle» a cui chiederlo, né appare opportuno introdurlo (sarebbe una mera «scatola»)
  - meglio una funzione *statica,* di libreria

ESEMPIO 2: `creaTabella`

IPOTESI: tabella dei *quadrati di N numeri interi*

- crea un array di interi della dimensione richiesta
- lo riempie con i quadrati dei primi N-1 valori
- lo restituisce al chiamante

SCELTA: funzione statica → invocata col nome assoluto (*`NomeClasse`*`.creaTabella`)
o anche solo relativo se il cliente (es. `main`) è nella stessa classe

```
static int[] creaTabella(int n){
   int[] v = new int[n];
   for (int i=0; i<v.length; i++)
      v[i] = i*i;
   return v;
}
```

Java

Allocato in heap
(sopravvive al termine della funzione)

C#: **`Length`**

Si restituisce un *nuovo array*
creato e riempito dalla funzione

ESEMPIO 2: UN MAIN DI PROVA

Java

```
// ipotesi: main nella stessa classe
public static void main(String[] args){
   int[] tab = creaTabella(4); // solo nome relativo
   for (int i=0; i<tab.length; i++)
       System.out.println(tab[i]);
}
```

C#

```
// ipotesi: main nella stessa classe
public static void Main(string[] args){
   int[] tab = CreaTabella(4); // solo nome relativo
   for (int i=0; i<tab.Length; i++)
       System.Console.WriteLine(tab[i]);
}
```

ESEMPIO 2: Scala & Kotlin

Poiché `creaTabella` e `main` sono nello stesso `object`, si può invocare la funzione col solo nome relativo

```scala
object Prova {

 def creaTabella(n : Int) : Array[Int] = {
   val v : Array[Int] = new Array[Int](n);
   for (i ← 0 to v.length-1) v(i) = i*i;
   return v;
 }

 def main(args: Array[String]) : Unit = {
  val tab : Array[Int] = creaTabella(4);
  for (i ← 0 to tab.length-1)
    println(tab(i));
 }

}
```

Scala

```
0
1
4
9
```

Osserva: il costruttore di Kotlin esige come secondo argomento un *blocco di inizializzazione*

```kotlin
fun creaTabella(n : Int) : Array<Int> {
     val v : Array<Int> = Array<Int>(n, {0});
     for (i in 0..v.size-1) v[i] = i*i;
     return v;
 }

fun main() {
    val tab : Array<Int> = creaTabella(4);
    for (i in 0..tab.size-1)
        println(tab[i]);
}
```

Kotlin

```
0
1
4
9
```

Qui si può invocare la funzione col solo nome relativo perché `creaTabella` e `main` sono entrambi a top level (→ nello stesso `package`)

VARIANTE: COSTRUTTO *FOR EACH*
iterare senza indici espliciti

Java

```
public static void main(String[] args){
   int[] tab = creaTabella(4);
   for (int x : tab) System.out.println(x);
}
```

Nuovo costrutto per iterare su insiemi (∀) *senza esplicitare indici*

C#

```
public stati    ain(string[] args){
   int[] tab =  reaTabella(4);
   foreach (int x in tab) System.Console.WriteLine(x);
}
```

Scala

```
def main(args: Array[String]) : Unit = {
 val tab : Array[Int] = creaTabella(4);
 for (x ← tab)
   println(x);
}
```

Kotlin

```
fun main() {
    val tab : Array<Int> = creaTabella(4);
    for (x in tab)
        println(x);
}
```

COSTRUTTO *FOR EACH* (1/3)

- Il costrutto *for each*
  - è l'analogo del simbolo matematico ∀ («per ogni»)
  - non richiede indici espliciti
  - a ogni iterazione *estrae una copia* dell'elemento

Java

```
public static void main(String[] args){
   int[] tab = creaTabella(4);
   for (int x : tab) System.out.println(x);
}
```

A ogni iterazione, x contiene **una copia del valore** dell'elemento tab[i]

COSTRUTTO *FOR EACH* (2/3)

- Il costrutto *for each*
  - è l'analogo del simbolo matematico ∀ («per ogni»)
  - non richiede indici espliciti
  - a ogni iterazione *estrae una copia* dell'elemento
- Pertanto:
  - si può usare per estrarre/leggere elementi da elaborare
  - *non* si può usare per *modificare* elementi
    - NB: nel caso di oggetti viene restituita una copia del riferimento, quindi si ha accesso all'oggetto originale, ma non si può comunque modificare il riferimento nell'array
  - *non* si può usare quando è indispensabile conoscere la *posizione* (indice) dell'elemento

COSTRUTTO *FOR EACH* (3/3)

Il *main* di poco fa è ok, perché estrae elementi:

```
for (int x : tab) System.out.println(x);
```

Sì

Invece, un ciclo *foreach* per raddoppiare gli elementi della tabella *non* funzionerebbe come previsto:

```
for (int x : tab) x = x*2;
```

No

A ogni iterazione, x contiene **una copia del valore** dell'elemento tab[i]
→ la tabella resterebbe **inalterata**

ESEMPIO 3 (1/6)

Scrivere una funzione che restituisca **`true`** se due array di interi sono uguali (cioè hanno tutti gli elementi corrispondentemente uguali).

Signature della funzione richiesta:

```
public static boolean idem(int[] a, int[] b){
   ...
}
```

C#: **`bool`**

Java ~C#

Perché statica?

- se fosse un metodo, dovrebbe appartenere alla classe **`int[]`**, che però non definiamo noi: **`[]`** è una classe di sistema
- perciò, non possiamo far altro che predisporre una funzione "di libreria" tradizionale.

ESEMPIO 3 (2/6)

```
public static boolean idem(int[] a, int[] b){
   if (a.length != b.length) return false;
   for (int i=0; i<a.length; i++){
       if (a[i] != b[i]) return false;
   }
   return true;
}
```

Java

~C#

Se la lunghezza è diversa, **a** e **b** sono certamente diversi.

Sono anche diversi se *un solo elemento* risulta diverso.

C#: **Length**

Collaudo

- creare tre array (uno di dimensione diversa dagli altri)
- i due della stessa dimensione devono avere contenuto diverso
- fare almeno tre prove

Ma usare un main come test è improprio..

- meglio sarebbe un vero test con assert… o con JUnit.. ☺

ESEMPIO 3 (3/6)

```
public static void main(String[] args){
   int[] v1 = {2,3,4}, v2 = {2,3,5},
         v3 = {2,3},   v4 = {2,3,4};
   System.out.println( idem( v1, v3 ) );
   System.out.println( idem( v1, v2 ) );
   System.out.println( idem( v1, v1 ) );
   System.out.println( idem( v1, v4 ) );
}
```

Java

~C#

Espressioni di inizializzazione di un array Java o C# con costanti

Lunghezza diversa

Contenuto diverso

Stesso array

Contenuto uguale

Output:

```
false
false
true
true
```

Le *espressioni di inizializzazione* causano la creazione di un oggetto array *esattamente come se fosse stata usata la new:* semplicemente, l'array ha già quel contenuto iniziale.

ESEMPIO 3 (4/6)

```scala
object Prova {

 def main(args: Array[String]) : Unit = {
  val v1 = Array(2,3,4); val v2 = Array(2,3,5);
  val v3 = Array(2,3);   val v4 = Array(2,3,4);
  println( idem( v1, v3 ) );
  println( idem( v1, v2 ) );
  println( idem( v1, v1 ) );
  println( idem( v1, v4 ) );
 }

 def idem(a: Array[Int], b : Array[Int]) : Boolean = {
    if (a.size ≠ b.size) return false;
    for (i ← 0 to a.size-1){
      if (a(i) ≠ b(i)) return false;
    }
    return true;
 }

}
```

Scala

```
false
false
true
true
```

In Scala, è il metodo **`Array`** (omonimo del costruttore e a numero variabile di argomenti) a provvedere anche all'inizializzazione

In Kotlin, l'inizializzazione è svolta dal metodo **`arrayOf`** (a numero variabile di argomenti) che *incapsula* il costruttore

```kotlin
fun main(args: Array<String>) : Unit {
 val v1 = arrayOf(2,3,4); val v2 = arrayOf(2,3,5);
 val v3 = arrayOf(2,3);   val v4 = arrayOf(2,3,4);
 println( idem( v1, v3 ) );
 println( idem( v1, v2 ) );
 println( idem( v1, v1 ) );
 println( idem( v1, v4 ) );
}

fun idem(a: Array<Int>, b : Array<Int>) : Boolean {
   if (a.size != b.size) return false;
   for (i in 0..a.size-1){
       if (a[i] != b[i]) return false;
   }
   return true;
}
```

Kotlin

```
false
false
true
true
```

ESEMPIO 3 (5/6)

Java

```
public static void main(String args[]){
    System.out.println(
        idem( new int[]{2,3,4}, new int[]{2,3} ));
    System.out.println(
        idem( new int[]{2,3,4}, new int[]{2,3,5} ));
     System.out.println(
        idem( new int[]{2,3,4}, new int[]{2,3,4} ));
}
```

Java e C# : inizializzazione “al volo” in fase di costruzione

Contenuto uguale, ma array distinti

Output:

```
false
false
true
```

ESEMPIO 3 (6/6)

```scala
object Prova {

 def main(args: Array[String]) : Unit = {
  println( idem( Array(2,3,4), Array(2,3) ) );
  println( idem( Array(2,3,4), Array(2,3,5) ) );
  println( idem( Array(2,3,4), Array(2,3,4) ) );
 }

 def idem(a: Array[Int], b : Array[Int]) : Boolean = {
    if (a.size ≠ b.size) return false;
    for (i ← 0 to a.size-1){
      if (a(i) ≠ b(i)) return false;
    }
    return true;
 }

}
```

Scala

```
false
false
true
```

In Scala e Kotlin non occorre alcuna sintassi speciale: il costruttore **`Array`** (risp. il metodo **`arrayOf`**) provvede anche all'inizializzazione e può essere usato ovunque.

```kotlin
fun main(args: Array<String>) : Unit {
 println( idem( arrayOf(2,3,4), arrayOf(2,3) ) );
 println( idem( arrayOf(2,3,4), arrayOf(2,3,5) ) );
 println( idem( arrayOf(2,3,4), arrayOf(2,3,4) ) );
}

fun idem(a: Array<Int>, b : Array<Int>) : Boolean {
   if (a.size != b.size) return false;
   for (i in 0..a.size-1){
       if (a[i] != b[i]) return false;
   }
   return true;
}
```

Kotlin

```
false
false
true
```

ESEMPIO 4 (1/5)

E se volessimo una funzione analoga per due array di *oggetti*, ad esempio di **`Counter`**?

Signature della funzione richiesta:

```
public static boolean idem(Counter[] a, Counter[] b){
   ...
}
```

C#: **`bool`**

Java | ~C#

Deve ancora essere per forza statica?

- sì, perché se fosse un metodo dovrebbe essere nella classe "array di **`Counter`**", che *non* definiamo noi
- noi definiamo **`Counter`**, non "*Array di* ***`Counter`***": sono due cose diverse ("il singolare non è il plurale")

ESEMPIO 4 (2/5)

```
public static boolean idem(Counter[] a, Counter[] b){
   if (a.length != b.length) return false;
   for (int i=0; i<a.length; i++){
       if (!a[i].equals(b[i])) return false;
   }
   return true;
}
```

Java

~C#

Non si può più usare l'operatore `!=` dei tipi primitivi: occorre basarsi sul concetto di "uguale" proprio della classe ***`Counter`***

Adattare il collaudo

- creare due o più array di Counter…
- *ricordandosi di creare anche gli oggetti all'interno dell'array*
- anche qui è possibile creare gli oggetti o all'inizio o "al volo" (ma con una sintassi poco leggibile…)

ESEMPIO 4 (3/5)

- Per creare array di **`Counter`** già pre-inizializzati, si può riusare la notazione **`{…}`** già vista nel caso precedente
- Il contenuto sarà però un *elenco di (nuovi)* ***`Counter`***

Java

~C#

```
Counter[] w1 = {
   new Counter(2),new Counter(3)};

Counter[] w2 = {
   new Counter(2),new Counter(3),new Counter(4)};

...
```

- In alternativa, come nel caso precedente, per non dover introdurre nomi di array temporanei, si può ricorrere anche qui alla creazione con inizializzazione "on the fly"

ESEMPIO 4 (4/5)

```
System.out.println(idem( new Counter[]{
 new Counter(2),new Counter(3),new Counter(4)},
                          new Counter[]{
 new Counter(2),new Counter(3)} ));
System.out.println(idem( new Counter[]{
 new Counter(2),new Counter(3), new Counter(4)},
                        new Counter[]{
 new Counter(2),new Counter(3), new Counter(5)} ));
System.out.println(idem( new Counter[]{
 new Counter(2),new Counter(3), new Counter(4)},
                        new Counter[]{
 new Counter(2),new Counter(3), new Counter(4)} ));
```

Java

~C#

Output:

```
false
false
true
```

ESEMPIO 4 (5/5)

```scala
object Prova {

 def main(args: Array[String]) : Unit = {
  println( idem( Array(new Counter(2), new Counter(3), new Counter(4)), Array(new Counter(2), new Counter(3)) ) );
  println( idem( Array(new Counter(2), new Counter(3), new Counter(4)), Array(new Counter(2), new Counter(3), new Counter(5)) ) );
  println( idem( Array(new Counter(2), new Counter(3), new Counter(4)), Array(new Counter(2), new Counter(3), new Counter(4)) ) );
 }

 def idem(a: Array[Counter], b : Array[Counter]) : Boolean = {
    if (a.size ≠ b.size) return false;
    for (i ← 0 to a.size-1){
      if (!a(i).equals(b(i))) return false;
    }
    return true;
 }

}
```

Scala

```kotlin
fun main(args: Array<String>) : Unit {
   println( idem( arrayOf(Counter(2), Counter(3), Counter(4)), arrayOf(Counter(2), Counter(3)) ) );
   println( idem( arrayOf(Counter(2), Counter(3), Counter(4)), arrayOf(Counter(2), Counter(3), Counter(5)) ) );
   println( idem( arrayOf(Counter(2), Counter(3), Counter(4)), arrayOf(Counter(2), Counter(3), Counter(4)) ) );
}

fun idem(a: Array<Counter>, b : Array<Counter>) : Boolean {
   if (a.size != b.size) return false;
   for (i in 0..a.size-1){
       if (!a[i].equals(b[i])) return false;
   }
   return true;
}
```

Osserva: la sintassi concisa di Kotlin (senza keyword **new**) si rivela utile in queste situazioni

Kotlin

FUNZIONI DI UTILITÀ SUGLI ARRAY

- In Java e C#, le funzioni di utilità sugli array si trovano in librerie accessorie:
  - Java: classe **`java.util.Arrays`**
  - C#: classe ***`System.Array`***

Java

C#

- Sono disponibili funzioni statiche per:
  - copiatura: **`copyOf`** *(C#: **`Copy`**, **`CopyTo`**)*
  - ordinamento: **`sort`** *(C#: **`Sort`**)*
  - ricerca binaria: **`binarySearch`** *(C#: **`BinarySearch`**)*
  - test uguaglianza: **`equals`** *(C#: **`Equals`**, ma.. ☹)*
  - confronto lessicografico: **`compare`** *(solo Java)*
  - riempimento: **`fill`** *(solo Java)*
  - conversione in stringa: **`toString`** / **`deepToString`** *(solo Java)*
- In Scala e Kotlin, diventano metodi della classe **`Array`**

FUNZIONI DI UTILITÀ SUGLI ARRAY

- Più precisamente:

in Scala

- copiatura: `copyToArray`
- ordinamento: `sortBy` / `sorted`
- ricerca (binaria): `search` *(binaria se array ordinato)*
- test uguaglianza: `equals` / `sameElements` *(shallow / deep)*
- conversione in stringa: `mkString` *(shallow)*

in Kotlin

- copiatura: `copyOf`
- ricerche: `binarySearch` / `find` *(famiglia di metodi)*
- ordinamento: `sort` / `sortBy` / `sorted`
- test uguaglianza: `equals` / `contentDeepEquals` *(shallow / deep)*
- conversione in stringa: `joinToString` / `contentToString` *(shallow)*

ESERCIZIO: STAMPARE IL CONTENUTO DI UN ARRAY

- Java il modo più semplice per farlo è usare l'apposita funzione di libreria **`toString`** della classe **`Arrays`**
  - NB: non è il *metodo* **`toString`** della classe **`[]`**
  - È una funzione accessoria, statica, di libreria, che ha lo stesso nome solo perché concettualmente svolge lo stesso compito

```
jshell> int[] v = {1,2,3,4,5};
v ==> int[5] { 1, 2, 3, 4, 5 }

jshell> String s1 = v.toString();
s1 ==> "[I@621be5d1"

jshell> String s2 = Arrays.toString(v);
s2 ==> "[1, 2, 3, 4, 5]"

jshell> String[] w = {"hello", "ciao", "world", "mondo"};
w ==> String[4] { "hello", "ciao", "world", "mondo" }

jshell> String s3 = v.toString();
s3 ==> "[I@621be5d1"

jshell> String s4 = Arrays.toString(w);
s4 ==> "[hello, ciao, world, mondo]"
```

Metodo **`toString`** della classe **`[]`** (non personalizzabile né modificabile)

Funzione statica **`toString`** della classe **`Arrays`**

Metodo **`toString`** della classe **`[]`** (non personalizzabile né modificabile)

Funzione statica **`toString`** della classe **`Arrays`**

ESERCIZIO: STAMPARE IL CONTENUTO DI UN ARRAY

- Scala, la classe **`Array`** prevede uno speciale metodo **`mkString`**
  - NB: il metodo **`toString`** standard di **`Array`** funziona male, come in Java

```
scala> val v = Array("pippo", "pluto", "paperino")
val v: Array[String] = Array(pippo, pluto, paperino)

scala> val s1 = v.toString()
val s1: String = [Ljava.lang.String;@3a8640f7

scala> val s2 = v.mkString(",");
val s2: String = pippo,pluto,paperino

scala> val s2 = v.mkString("[", ",", "]" );
val s2: String = [pippo,pluto,paperino]
```

Metodo **`toString`** della classe **`Array`** (non personalizzabile né modificabile)

Metodo **`mkString`** della stessa classe **`Array`**

- Kotlin, la classe **`Array`** prevede uno metodo **`joinToString`**
  - NB: il metodo **`toString`** standard di **`Array`** funziona male, come in Java

```
>>> val v = arrayOf(1,2,3,4)
>>> val s1= v.toString() ; println(s1)
[Ljava.lang.Integer;@1174ff02
>>> val s2= v.joinToString(",") ; println(s2)
1,2,3,4
>>> val s3= v.joinToString(",", "[", "]") ; println(s3)
[1,2,3,4]
```

Metodo **`toString`** della classe **`Array`** (non modificabile)

Metodo **`joinToString`** della stessa classe **`Array`**

ESERCIZIO: STAMPARE IL CONTENUTO DI UN ARRAY

- Scala la classe **`Array`** prevede uno speciale metodo **`mkString`**
  - NB: il metodo **`toString`** standard di **`Array`** funziona male, come in Java

```
scala> val v = Array("pippo", "pluto", "paperino")
val v: Array[String] = Array(pippo, pluto, paperino)

scala> val s1 = v.toString()
val s1: String = [Ljava.lang.String;@3a8640f7

scala> val s2 = v.mkString(",");
val s2: String = pippo,pluto,paperino

scala> val s2 = v.mkString("[", ",", "]" );
val s2: String = [pippo,pluto,paperino]
```

- Kotlin in alternativa, la classe **`Array`** prevede anche uno metodo **`contentToString`** (meno personalizzabile, ma di uso più diretto)

```
val s5= v.contentToString()
println(s5)

[1, 2, 3, 4]
```

ESERCIZIO RIASSUNTIVO: OPERAZIONI SU ARRAY

1. Creare un array di interi inizialmente vuoto (in Kotlin: già pieno..)
2. Riempirlo di valori casuali (in Kotlin: da fare insieme alla creazione)
   - Java: classe `java.util.Random`, metodo `nextInt`
   - C#: classe `System.Random`, metodo `Next`
   - Scala: classe `scala.util.Random`, metodo `nextInt`
   - Kotlin: classe `kotlin.random.Random`, metodo `nextInt`
3. Ordinarlo in senso crescente
   - Java: classe `java.util.Arrays`, metodo `sort`
   - C#: classe `System.Array`, metodo `Sort`
   - Scala: classe `scala.util.Sorting`, metodo `quickSort`
   - Kotlin: metodo `sort` della classe Array
4. Cercare al suo interno un certo valore
   - Java: classe `java.util.Arrays`, metodo `binarySearch`
   - C#: classe `System.Array`, metodo `BinarySearch`
   - Scala: metodo `search` della classe Array
   - Kotlin: metodo `binarySearch` della classe Array

L'ESERCIZIO in Java

Java

```
public class SortAndSearchArray {
 ... // la mia funzione show
 public static void main(String[] args){
   int[] v = new int[20];
   java.util.Random gen = new java.util.Random();
   for (int i=0; i<v.length; i++)
        v[i] = gen.nextInt(30);
   show(v); // mia funzione che stampa tutto l'array
   java.util.Arrays.sort(v);
   show(v); // mia funzione che stampa tutto l'array
   int n = gen.nextInt(30);
   int pos = java.util.Arrays.binarySearch(v, n);
   System.out.println("cercato " + n +
                      ": trovato alla posizione " + pos);
 }
}
```

L'ESERCIZIO in C#

C#

```
public class SortAndSearchArray {
 ... // la mia funzione show
 public static void Main(string[] args){
   int[] v = new int[20];
   System.Random gen = new System.Random();
   for (int i=0; i<v.length; i++)
        v[i] = gen.Next(30);
   show(v); // mia funzione che stampa tutto l'array
   System.Array.Sort(v);
   show(v); // mia funzione che stampa tutto l'array
   int n = gen.Next(30);
   int pos = System.Array.BinarySearch(v, n);
   System.Console.WriteLine("cercato " + n +
                    ": trovato alla posizione " + pos);
 }
}
```

Java: ESECUZIONE

Output (caso di numero trovato):

```
3,12,5,8,2,11,4,6,3,26,19,27,23,20,7,8,26,10,9,19.
2,3,3,4,5,6,7,8,8,9,10,11,12,19,19,20,23,26,26,27.
cercato 7: trovato alla posizione 6
```

Java

Output (caso di numero non trovato):

```
12,15,19,5,25,4,8,17,28,0,23,6,21,13,19,20,15,1,7,25.
0,1,4,5,6,7,8,12,13,15,15,17,19,19,20,21,23,25,25,28.
cercato 26: trovato alla posizione -20
```

Java

Un valore negativo restituito da **`binarySearch`** indica che il numero richiesto non è stato trovato,
MA sarebbe da inserire prima della posizione 19
If not found, binarySearch returns -insertion point-1

L'ESERCIZIO in Scala e Kotlin

Kotlin

```
fun show(v : Array<Int>) : Unit {
    println(v.joinToString(","));
}

public fun main(args: Array<String>) {
    var v = Array<Int>(20, {kotlin.random.Random.nextInt(30)});
    show(v);
    v.sort()
    show(v);
    val n = kotlin.random.Random.nextInt(30)
    val pos = v.binarySearch(n);
    println("cercato " + n + ": trovato alla posi
}
```

```
5,29,14,25,8,1,19,20,19,22,6,7,28,24,3,20,7,
1,3,5,6,7,7,7,8,11,11,14,19,19,20,20,22,24,2
cercato 22: trovato alla posizione 15
```

Ricorda: in Kotlin il costruttore degli array esige un *inizializzatore*
→ incapsula l'invocazione del generatore di numeri casuali

Scala

```
def show(v : Array[Int]) : Unit = {
  println(v.mkString(","));
}

def main(args: Array[String]) : Unit = {

  var v = new Array[Int](20);
  val gen = new scala.util.Random();
  for (i ← 0 until v.length) v(i) = gen.nextInt(30);
  show(v);
  //scala.util.Sorting.quickSort(v); // se v è val
  v = v.sorted // se v è var
  show(v);

  val n = gen.nextInt(30);
  val pos = v.search(n);
  println("cercato " + n + ": trovato alla posizione " + pos);
}
```

```
24,11,2,26,18,18,12,21,2,26,5,14,11,11,10,25,1,2,1,19
1,1,2,2,2,5,10,11,11,11,12,14,18,18,19,21,24,25,26,26
cercato 27: trovato alla posizione InsertionPoint(20)
```

VARIANTE: ARRAY DI STRINGHE

Java

```
public class SortStringArray {
  ... // la mia funzione show
 public static void main(String[] args){
   show(args);
   java.util.Arrays.sort(args);
   show(args);
   int pos =
       java.util.Arrays.binarySearch(args,"Enrico");
   System.out.println("trovato alla posizione " + pos);
 }
}
```

C#, Scala, Kotlin: solite sostituzioni di nomi

```
java SortStringArray Antonio Vicky Giovanni Enrico
Antonio,Vicky,Giovanni,Enrico
Antonio,Enrico,Giovanni,Vicky
trovato alla posizione 1
```

QUALCHE ALTRO ESPERIMENTO

- Sperimentiamo le funzioni **`copyOf`** ed **`equals`**
  - **`Arrays.copyOf`** costruisce e restituisce *un nuovo array* contenente *i primi k elementi* dell'array ricevuto
  - NB: se k>length, aggiunge zeri in fondo

```
jshell> long[] v1 = {1, 7, -8, 22 }
v1 ==> long[4] { 1, 7, -8, 22 }

jshell> long[] v2 = java.util.Arrays.copyOf(v1, v1.length)
v2 ==> long[4] { 1, 7, -8, 22 }

jshell> long[] v3 = java.util.Arrays.copyOf(v1, v1.length-2)
v3 ==> long[2] { 1, 7 }

jshell> long[] v4 = java.util.Arrays.copyOf(v1, v1.length+2)
v4 ==> long[6] { 1, 7, -8, 22, 0, 0 }
```

Java

QUALCHE ALTRO ESPERIMENTO

- Sperimentiamo le funzioni **`copyOf`** ed **`equals`**
  - **`Arrays.equals`** verifica se *due array sono uguali*
  - NB: è una verifica *shallow*, ossia *superficiale*
    Ciò causerà problemi con gli array a più dimensioni (matrici)

```
jshell> int[] v1 = {3,4,5}
v1 ==> int[3] { 3, 4, 5 }

jshell> int[] v2 = {3,4,5}
v2 ==> int[3] { 3, 4, 5 }

jshell> int[] v3 = {3,4,6}
v3 ==> int[3] { 3, 4, 6 }

jshell> int[] v4 = {3,4}
v4 ==> int[2] { 3, 4 }

jshell> java.util.Arrays.equals(v1,v2)
$5 ==> true

jshell> java.util.Arrays.equals(v1,v3)
$6 ==> false
```

Java

Non usare il metodo **`equals`** della classe [ ] perché confronta solo i riferimenti ! ☹ ☹

```
jshell> v1.equals(v1)
$7 ==> true

jshell> v1.equals(v2)
$8 ==> false

jshell> java.util.Arrays.equals(v1,v2)
$9 ==> true
```

Java

La funzione statica equals della *libreria Arrays* invece *confronta il contenuto* ☺ (in modo shallow)

Quella di C# invece NO… ☹ C#

UN DETTAGLIO LINGUISTICO: DEFINIZIONE DI COSTANTI

- Disporre di *costanti simboliche* è spesso comodo, in particolare con gli array
- In Java, una *costante* si definisce qualificando **`final`** una variabile pre-inizializzata:

```
final int DIM = 8;
```

Java

DEVE esserci un valore: *se manca, errore di compilazione*

  - ogni tentativo di riassegnare la costante a un diverso valore sarà stroncato dal compilatore, *anche se il valore fosse identico:*

```
int final DIM = 8; // OK
    ...
DIM = 8;           // NO!
```

UN DETTAGLIO LINGUISTICO: DEFINIZIONE DI COSTANTI

- In C#, si usano i qualificatori **`const`** e **`readonly`**

```
const    int DIM = 8;
```

```
readonly int DIM = 8;
```

C#

  - nel primo caso, DIM viene sostituita *inline* dalla costante 8
  - nel secondo, DIM è a tutti gli effetti una variabile pre-inizializzata non modificabile, esattamente come in Java con **`final`**

- In Scala e Kotlin, basta usare il qualificatore **`val`**

```
final val DIM = 8;
```

Scala

```
const val DIM = 8;
```

Kotlin

  - con final (Scala) o const (Kotlin), si crea una costante statica
  - in Scala, la naming convention prevederebbe il Camel Case

Array multi-dimensionali
Matrici

MATRICI COME ARRAY MULTIDIMENSIONALI

- Una matrice è un *array a più dimensioni* (array di array)
  - sono possibili anche array a tre, quattro,.. *N* dimensioni
- Una matrice può essere creata:
  - o in PIÙ FASI (prima l'array più esterno, poi quello più interno, etc.)
  - o in UN'UNICA FASE (significato identico, ma sintassi più concisa)
- I vari linguaggi differiscono un po' sulla sintassi per farlo

  ESEMPI
  - in Java: `double[][] m = new double[…][…]`
  - in C#: `double[][] m = new double[…][…]`
    o anche `double[,] m = new double[…,…]` *(non identici)*
  - in Scala: `val m = Array.ofDim(…,…)`
  - in Kotlin: `val m = arrayOf(arrayOf(…))`

MATRICI in Java e C#

- In Java e C#, occorre inizialmente definire il *riferimento*
- Il *tipo* del riferimento specifica il *numero di dimensioni:*

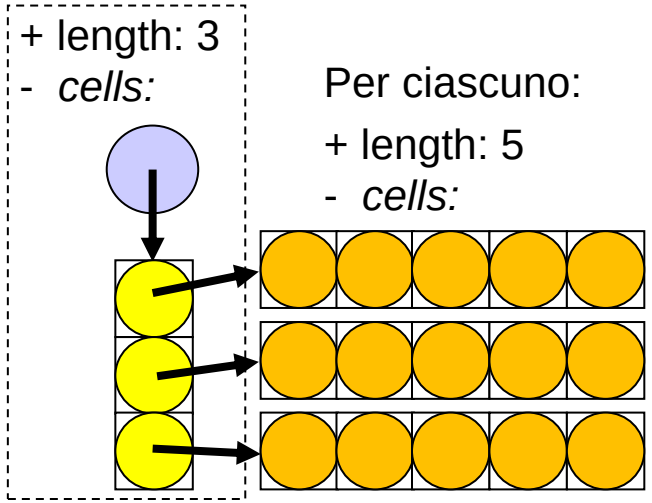
```
double[][] m;  // due dimensioni
```

- La creazione avviene quindi in DUE FASI:
  - prima si crea l'array "esterno":

```
m = new double[3][];
```

  - poi si creano gli array "interni":

```
m[0] = new double[5];
m[1] = new double[5];
m[2] = new double[5];
```



Potremmo definire anche matrici *irregolari,* con righe di lunghezza diversa le une dalle altre.

MATRICI in Java e C#

- In Java e C#, occorre inizialmente definire il *riferimento*
- Il *tipo* del riferimento specifica il *numero di dimensioni:*

```
double[][] m;  // due dimensioni
```

- La creazione avviene quindi in DUE FASI:
  - prima si crea l'array "esterno":

```
m = new double[3][];
```

  - poi si creano gli array "interni":

```
m[0] = new double[5];
m[1] = new double[5];
m[2] = new double[5];
```

Siamo noi a decidere come interpretarle in termini di *righe* e *colonne*

Possiamo interpretare questa matrice sia come 3x5 (3 righe x 5 colonne) sia come 5x3 (5 righe x 3 colonne): basta essere coerenti nel seguito.

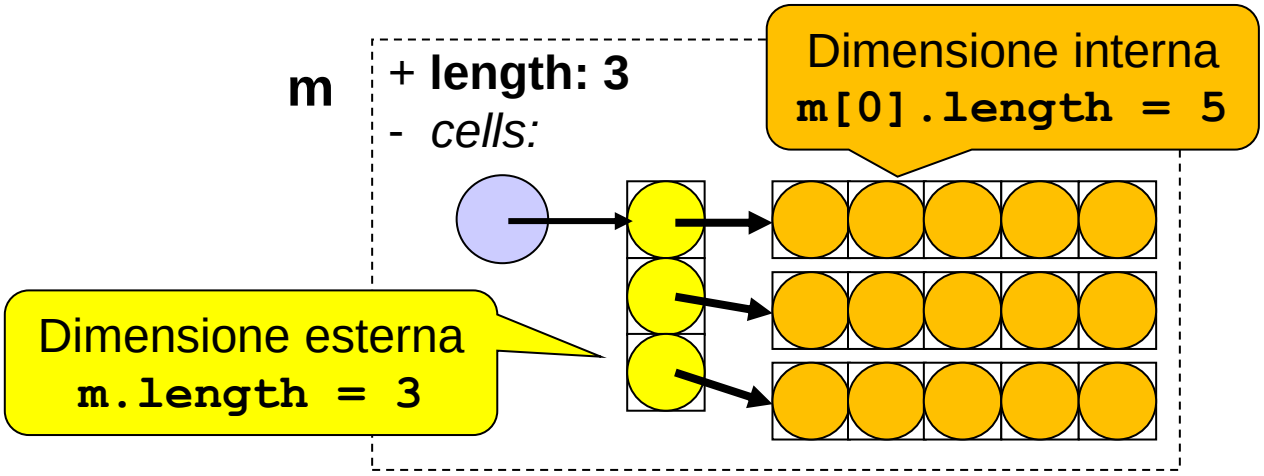
Di solito si intende la prima.

MATRICI REGOLARI in Java

- In Java, una matrice regolare si può creare più sinteticamente scrivendo:

```
m = new double[3][5];  // matrice 3 x 5
```

dove
  - **`m.length`** è la cardinalità della dimensione esterna
  - **`m[i].length`** è la cardinalità della dimensione interna
    *(per matrici regolari, **`i`** può essere una qualsiasi)*



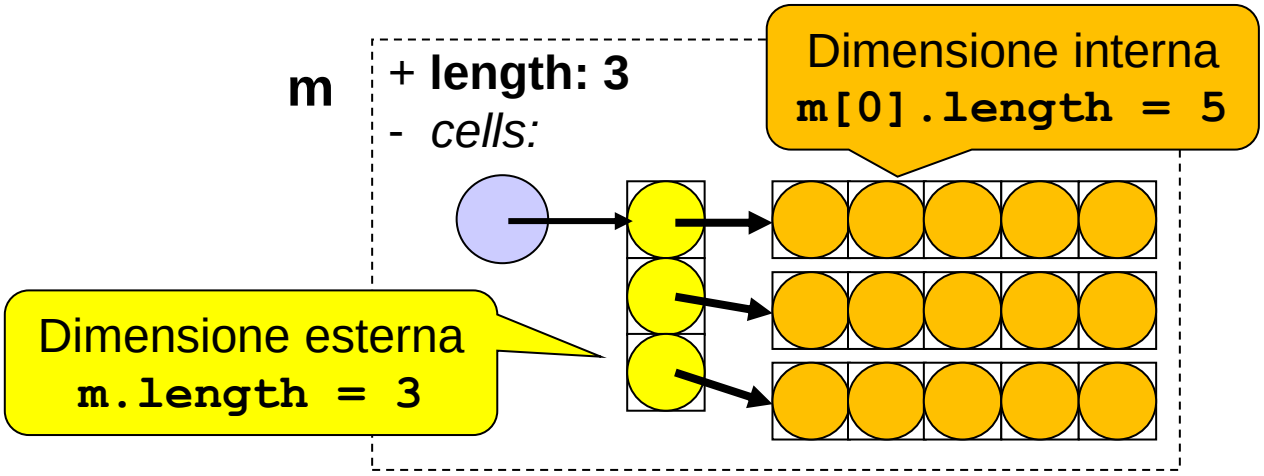
MATRICI REGOLARI in Java

- Per accedere alle celle si usano ovviamente *due indici:*

```
m[0][0] = 1.2;    m[1][0] = -2.6;    ...
```

la cui interpretazione in termini di *indice di riga* e *di colonna* deve essere *coerente* con la precedente:

- `m.length` è il numero di *righe* *(quindi, **3**)*
- `m[i].length` è il numero di *colonne* *(quindi, **5**)*



MATRICI REGOLARI in C#

- In C#, una matrice regolare si definisce sinteticamente scrivendo *(occhio alle virgole!)*:

```
double[,] m;                // due dimensioni
m = new double[3,5];        // matrice 3 x 5
```

  - Questo tipo di dato è *diverso dall'altro* e *incompatibile* con esso *anche a parità di dimensioni:*

`double[,]` ≠ **`double[][]`**

- Cambia anche l'approccio alle dimensioni:
  - **`m.Length`** è il numero totale di elementi della matrice
  - **`m.GetLength(i)`** restituisce la cardinalità della *dimensione i-esima* *(esterna: **`i`**=0; interna: **`i`**=1; etc.)*

MATRICI in Scala

- In Scala una matrice regolare si definisce tramite il metodo **`ofDim`** dell'oggetto singleton **`Array`**:

```
var m : Array[Array[…]] =
                    Array.ofDim(3,5);
```

  - è indispensabile specificare il tipo (altrimenti sarà di **`Nothing`**)
  - la matrice creata è vuota (numeri posti a 0, puntatori a **`null`**)

- Se si desidera anche inizializzare con costanti, occorre (come in Java) costruire le righe una ad una:

```
var m : = Array( Array(…), Array(…)), …);
```

  - non occorre più specificare il tipo (si può dedurre dal lato destro)
  - la matrice così definita potrebbe essere anche *irregolare*

MATRICI in Kotlin

- In Kotlin una matrice regolare si definisce costruendo gli array esterno e interno con le rispettive dimensioni:

```
var m : = Array(3) { DoubleArray(5) }
```

  - non occorre specificare il tipo (viene dedotto dal lato destro)
  - la matrice creata è vuota (numeri posti a 0.0) perché è di **`Double`**
  - si può usare il costruttore generico **`Array`** anche per l'array interno, ma in tal caso occorre specificare il relativo inizializzatore:
    `var m : = Array(3) { Array<Double>(5) { {2.2} }`

- Per inizializzare con costanti, anche qui si costruiscono le righe una ad una, tramite la famiglia di metodi *xx***`arrayOf`**:

```
var m : = arrayOf(doubleArrayof(…), …);
```

  - la matrice così definita potrebbe essere anche *irregolare*
  - si può usare **`arrayOf`** anche all'interno, come sopra

STAMPA DI MATRICI (1/2)

- Già sappiamo che il metodo **`toString`** è inadeguato
- Per gli array mono-dimensionali, lo abbiamo sostituito:
  - in Java, con la funzione statica **`Arrays.toString`**
  - in Scala, con il metodo **`mkString`** della classe **`Array`**
  - in Kotlin, con il metodo **`joinToString`** (o **`contentToString`**) della classe **`Array`**
- Tale approccio però *non si estende* a più dimensioni
  - poiché una matrice è in realtà un array di array, l'uso dei metodi precedenti risolverebbe il livello «esterno», ma riproporrebbe il problema nei livelli «interni»
  - in essi, infatti, verrebbe nuovamente richiamato in automatico il metodo **`toString`** di default

STAMPA DI MATRICI ESPERIMENTO (FALLITO)

Java

```
jshell> double[][] m = {{2,3,4},{3,4,5},{5,6,7}};
m ==> double[3][] { double[3] { 2.0, 3.0, 4.0 }, double ... ble[3] { 5.0, 6.0, 7.0 } }

jshell> System.out.println(Arrays.toString(m));
[[D@7ab2bfe1, [D@497470ed, [D@63c12fb0]
```

L'array esterno genera correttamente una stringa della forma **`[elem1, …, elemN]`** ma gli array interni li gestisce **`toString`**.. ☹

Scala

```
val v = Array(Array(1,2,3),Array(4,5,6),Array(7,8,9))
println(v)

println(v.mkString(","))
```

```
[[I@2d5cf243
[I@34bdf993,[I@7ca57417,[I@3e979d3c
```

Kotlin

```
val m = arrayOf(arrayOf(1,2,3),arrayOf(3,4,5), arrayOf(4,5,6))
val s1 = m.toString(); println(s1)
[[Ljava.lang.Integer;@6be865c1
val s2 = m.joinToString(",", "[", "]"); println(s2)
[[Ljava.lang.Integer;@68823b6b,[Ljava.lang.Integer;@4b4b02d,[Ljava.lang.Integer;@1dd01876]
val s3 = m.contentToString(); println(s3)
[[Ljava.lang.Integer;@68823b6b, [Ljava.lang.Integer;@4b4b02d, [Ljava.lang.Integer;@1dd01876]
```

Idem

STAMPA DI MATRICI (2/2)

- Già sappiamo che il metodo **`toString`** è inadeguato
- Per gli array mono-dimensionali, lo abbiamo sostituito:
  - in Java, con la funzione statica **`Arrays.toString`**
  - in Scala, con il metodo **`mkString`** della classe **`Array`**
  - in Kotlin, con il metodo **`joinToString`** (o **`contentToString`**) della classe **`Array`**
- Per array multi-dimensionali serve un approccio alternativo
  - in Java, con la funzione statica **`Arrays.deepToString`**
  - in Scala, si riusa il metodo **`mkString`** a 2+ livelli, con un approccio molto funzionale ed elegante (ma non semplicissimo..)
  - in Kotlin, con il metodo **`contentDeepToString`** di **`Array`**

STAMPA DI MATRICI ESPERIMENTO (RIUSCITO)

```
jshell> double[][] m = {{2,3,4},{3,4,5},{5,6,7}};
m ==> double[3][] { double[3] { 2.0, 3.0, 4.0 }, double ... ble[3] { 5.0, 6.0, 7.0 } }

jshell> System.out.println(Arrays.toString(m));
[[D@7ab2bfe1, [D@497470ed, [D@63c12fb0]

jshell> System.out.println(Arrays.deepToString(m));
[[2.0, 3.0, 4.0], [3.0, 4.0, 5.0], [5.0, 6.0, 7.0]]
```

La funzione **deepToString** gestisce correttamente anche i livelli interni ☺

Java

```
val v = Array(Array(1,2,3),Array(4,5,6),Array(7,8,9))
println(v)

println(v.mkString(","))

println(v.map(_.mkString("[", ",", "]")).mkString("[", ",", "]"))

println(v.map(_.mkString(" ")).mkString("\n"))
```

Scala

```
[[I@2d5cf243
[I@34bdf993,[I@7ca57417,[I@3e979d3c

[[1,2,3],[4,5,6],[7,8,9]]

1 2 3
4 5 6
7 8 9
```

```
val m = arrayOf(arrayOf(1,2,3),arrayOf(3,4,5), arrayOf(4,5,6))
val s1 = m.toString(); println(s1)
[[Ljava.lang.Integer;@6be865c1
val s2 = m.joinToString(",", "[", "]"); println(s2)
[[Ljava.lang.Integer;@68823b6b,[Ljava.lang.Integer;@4b4b02d,[Ljava.lang.Integer;@1dd01876]
val s3 = m.contentToString(); println(s3)
[[Ljava.lang.Integer;@68823b6b, [Ljava.lang.Integer;@4b4b02d, [Ljava.lang.Integer;@1dd01876]
val s4 = m.contentDeepToString(); println(s4)
[[1, 2, 3], [3, 4, 5], [4, 5, 6]]
```

Kotlin

Anche qui, **contentDeepToString** gestisce correttamente i livelli interni

ESEMPIO: SOMMA DI MATRICI

Java

```
public static double[][] sommaMatrici(
                    double[][] a, double[][] b) {
  double[][] c = new double[a.length][a[0].length];
  for (int i=0; i < a.length; i++)
         for (int j=0; j < a[0].length; j++)
             c[i][j] = a[i][j] + b[i][j];
  return c;
}
```

- Java: perché **`sommaMatrici`** è una funzione statica?
  - per il solito motivo: se fosse un metodo, dovrebbe appartenere alla classe **`double[][]`**, che però *non* definiamo noi
  - perciò, può essere solo una funzione "classica" di libreria, che in Java e C# assume la forma statica (in Scala e Kotlin, dove non c'è la keyord **`static`**, andrebbe comunque posta in un **`object`** singleton)

ESEMPIO: SOMMA DI MATRICI

C#

```
public static double[,] sommaMatrici(
                        double[,] a, double[,] b) {
  double[,] c = new double[a.GetLength(0),a.GetLength(1)];
  for (int i=0; i < a.GetLength(0); i++)
          for (int j=0; j < a.GetLength(1); j++)
              c[i,j] = a[i,j] + b[i,j];
  return c;
}
```

- C#: perché una funzione statica?
  - stesse motivazioni rispetto a Java

ESEMPIO: SOMMA DI MATRICI

- Scala la funzione va posta in un *oggetto singleton*
  - non può essere metodo di una classe, per le stesse ragioni per cui non può esserlo in Java e C#
  - in Scala non esiste più la keyword **`static`**, perché sussunta da **`object`** → le parti «statiche» vanno poste in un **`object`** singleton
- Kotlin la funzione va posta o a top-level (come il main) o in un *oggetto singleton* (come in Scala)
  - non può essere metodo di una classe, per le stesse ragioni per cui non può esserlo in Java, C# e Scala
  - in Kotlin, le funzioni possono essere a top-level (cioè fuori da una classe o da un **`object`**, a livello di package) oppure, come in Scala, in un **`object`**

ESEMPIO: SOMMA DI MATRICI

Mini-main di collaudo:

Java

```
public static void main(String[] args){
  double[][] m = {{ 1.2, 2.3, 2.3 },
                  { 7.4, 5.1, 9.8 } };
  double[][] n = {{ 5.0, 4.0, 1.3 },
                  { 1.2, 0.3, 3.2 } };
  double[][] q = sommaMatrici(m,n);
  stampaMatrice(q);                    // da fare
}
```

C#

```
...
 double[,]  m = {{ 1.2, 2.3, 2.3 },
                 { 7.4, 5.1, 9.8 } };
...
```

ESEMPIO: SOMMA DI MATRICI

```
def sommaMatrici(a: Array[Array[Double]], b: Array[Array[Double]]) : Array[Array[Double]]
  var c : Array[Array[Double]] = Array.ofDim(a.size, a(0).size)
  for (i ← 0 until a.size)
      for (j ← 0 until a(0).size)
          c(i)(j) = a(i)(j) + b(i)(j);
  return c;
}

def main(args: Array[String]) : Unit = {
  var m = Array( Array( 1.2, 2.3, 2.3 ),
                 Array( 7.4, 5.1, 9.8 ) );
  var n = Array( Array( 5.0, 4.0, 1.3 ),
                 Array( 1.2, 0.3, 3.2 ) );
  var q = sommaMatrici(m,n);
  stampaMatrice(q);
}

def stampaMatrice(m: Array[Array[Double]]) : Unit = {
  //for (i in 0..m.size-1) println(m[i].mkString("\t"));
  for (i ← 0 until m.size) {
    for (j ← 0 until m(0).size) print("%5.2f\t".format(m(i)(j)));
       println();
   }
}
```

In Scala è possibile collassare due cicli singoli in un *ciclo doppio*: ad esempio, **`for(i<-0 to 2; j<-0 to 3){…}`**

```
 6.20    6.30    3.60
 8.60    5.40   13.00
```

Scala

ESEMPIO: SOMMA DI MATRICI

```kotlin
fun sommaMatrici(a: Array<DoubleArray>, b: Array<DoubleArray>) : Array<DoubleArray> {
    var c  = Array(a.size) { DoubleArray(a[0].size) }
    for (i in 0..a.size-1)
        for (j in 0..a[0].size-1)
            c[i][j] = a[i][j] + b[i][j];
    return c;
}

public fun main(args: Array<String>) {
   var m = arrayOf( doubleArrayOf( 1.2, 2.3, 2.3 ),
                    doubleArrayOf( 7.4, 5.1, 9.8 ) );
   var n = arrayOf( doubleArrayOf( 5.0, 4.0, 1.3 ),
                    doubleArrayOf( 1.2, 0.3, 3.2 ) );
   var q = sommaMatrici(m,n);
   stampaMatrice(q);
}

fun stampaMatrice(m: Array<DoubleArray>) : Unit {
    //for (i in 0..m.size-1) println(m[i].joinToString("\t"));
    for (i in 0..m.size-1)  {
        for (j in 0..m[0].size-1) print("%5.2f\t".format(m[i][j]));
         println();
     }
}
```

```
 6.20    6.30    3.60
 8.60    5.40   13.00
```

Kotlin

ESERCIZIO: PRODOTTO DI MATRICI

- Analogo al precedente…
  … MA *attenzione alle dimensioni delle matrici!*

È lasciato per esercizio al lettore ☺

Mini-main di collaudo:

Java

```
public static void main(String[] args){

  double[][] m = {{ 1.4, 4.3, 5.3 },        // 2 x 3
                  { 7.4, 5.0, 4.8 } };
  double[][] n = {{ 5.0, 3.0},              // 3 x 2
                  { 1.2, 0.6},
                  { 0.5, 8.1} };
  double[][] q = prodottoMatrici(m,n);      // 2 x 2
  stampaMatrice(q);
}
```

UN PROBLEMINO CON `equals` UGUAGLIANZA SUPERFICIALE

- In Java, parlando della libreria **`Arrays`**, dicemmo che la funzione statica **`equals`** agiva in modo *shallow*
  - ora capiremo cosa significa (purtroppo…)
- Sappiamo che **`Arrays.equals`** verifica se *due array* sono uguali: ma… *cosa fa con due matrici?*
- Il problema è che le «matrici» sono in realtà *array di array*
  - non esistono «veramente» array a più dimensioni..!
- Ergo, **`Arrays.equals`** verifica *l'array che vede,* che è *quello esterno:* il risultato è un check *superficiale (shallow)*
  - conseguenza: le cose NON vanno come pensiamo!

UN PROBLEMINO CON `equals` UGUAGLIANZA SUPERFICIALE

- Facciamo un esperimento:

```
jshell> int[][] m1 = { {1,1}, {2,3}}
m1 ==> int[2][] { int[2] { 1, 1 }, int[2] { 2, 3 } }

jshell> int[][] m2 = { {1,1}, {2,3}}
m2 ==> int[2][] { int[2] { 1, 1 }, int[2] { 2, 3 } }

jshell> java.util.Arrays.equals(m1,m2)
$12 ==> false
```

- Sorpresa: le due matrici risultano diverse!
  - MOTIVO: m1 e m2 sono in realtà array di **`int[]`** e **`equals`** lavora solo sul primo livello, quello esterno
  - Perciò, risponde **`true`** solo se gli array interni *coincidono* (uguaglianza di riferimenti), mentre qui sono *oggetti distinti* (seppur con contenuto identico) → la risposta è **`false`**

DA `equals` A `deepEquals`

- Per risolvere il problema, occorre fare una verifica *profonda*, non superficiale → metodo Java **`deepEquals`**

```
jshell> int[][] m1 = { {1,1}, {2,3}}
m1 ==> int[2][] { int[2] { 1, 1 }, int[2] { 2, 3 } }

jshell> int[][] m2 = { {1,1}, {2,3}}
m2 ==> int[2][] { int[2] { 1, 1 }, int[2] { 2, 3 } }

jshell> java.util.Arrays.equals(m1,m2)
$12 ==> false

jshell> java.util.Arrays.deepEquals(m1,m2)
$13 ==> true
```

- Ora sì che le due matrici risultano «uguali»!
  - **`equals`** lavora solo in superficie («shallow»)
    → risponde **`true`** solo se gli array interni *coincidono*
  - **`deepEquals`** lavora in profondità («deep»)
    → risponde **`true`** se gli array interni hanno *egual contenuto*

E NEGLI ALTRI LINGUAGGI…?

- C# *non* offre metodi per la verifica deep
  - esistono alcune librerie di terze parti per casi specifici
- Scala *non* offre metodi per la verifica deep
  - a dire il vero c'era (si chiamava **`deep`**), ma è stato rimosso in Scala 2.13 (pare per motivi di prestazioni)
- In Kotlin c'è un metodo **`contentDeepEquals`** analogo al metodo **`deepEquals`** di Java