

Fondamenti di Informatica T2

Esercitazione Individuale:

Array multidimensionali & ADT Matrix

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ROBERTA CALEGARI

Prof. AMBRA MOLESINI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)



Agenda

- **FASE 1**

ripasso (veloce) array multidimensionali (matrici)

prima semplice esercitazione pratica con array multidimensionali

- **CASO DI STUDIO: matrici (libreria statica)**
- MA lavorare direttamente con gli array, esponendo direttamente la rappresentazione interna al cliente, può creare parecchi problemi..

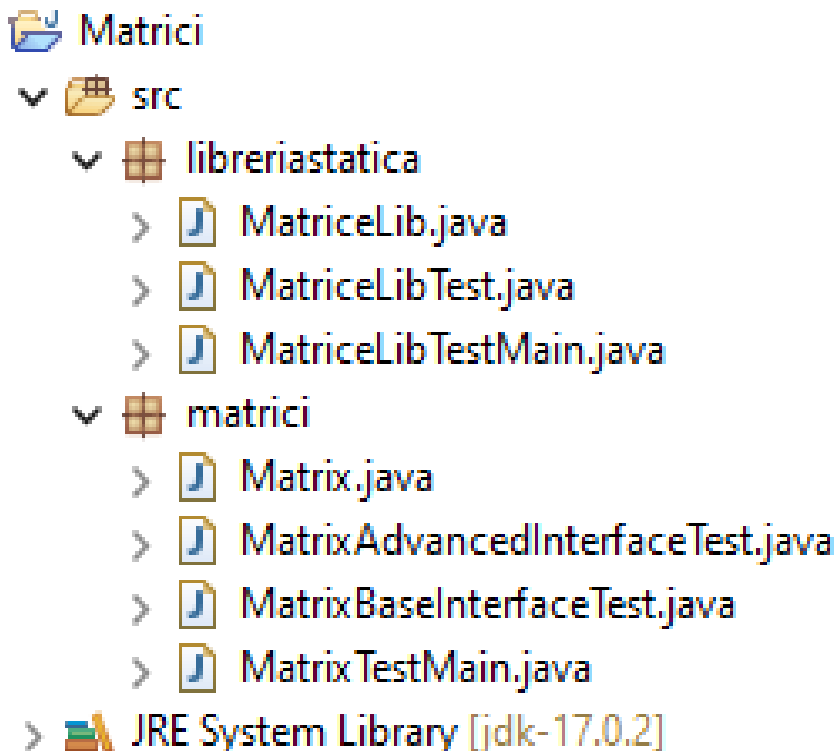
- **FASE 2 (revisione critica):**

progetto e realizzazione di un **nuovo ADT Matrix** che

- *incapsuli* al suo interno l'array (stato interno protetto, accesso mediato)
- esponga all'esterno un insieme di *metodi stabiliti da noi* (non da Java!)



Organizzazione Ambiente di Lavoro

- Utilizziamo due diversi package
 - Package **LibreriaStatica**
 - Contiene MatriceLib e le classi di test
 - Package **Matrici**
 - Contiene ADT Matrix e le classi di test
- 

Matrici Come Array Multidimensionali

Una **matrice** è un **array a più dimensioni** (array di array).

Il *tipo del riferimento* specifica il *numero di dimensioni*:

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

La creazione avviene quindi in **due passi**:

- prima si crea l'**array "esterno"**:

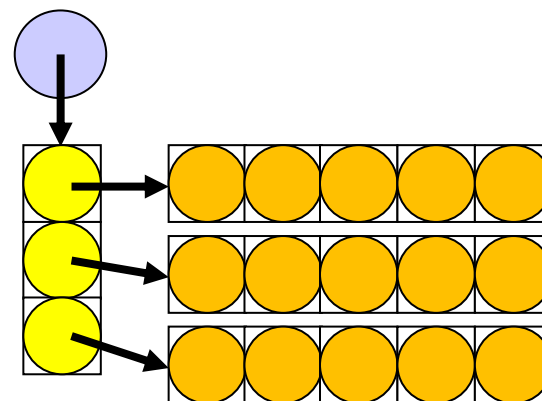
```
m = new double[3][];
```

- 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 Come Array Multidimensionali

Una **matrice** è un **array a più dimensioni** (array di array).

Il *tipo del riferimento* specifica il *numero di dimensioni*:

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

La creazione avviene quindi in **due passi**:

- 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];
```

NB: siamo noi a decidere
come interpretarle in termini
di *righe* e *colonne*

Possiamo interpretare questa
matrice come 3x5
(3 righe x 5 colonne)
o viceversa: basta essere
coerenti nel seguito

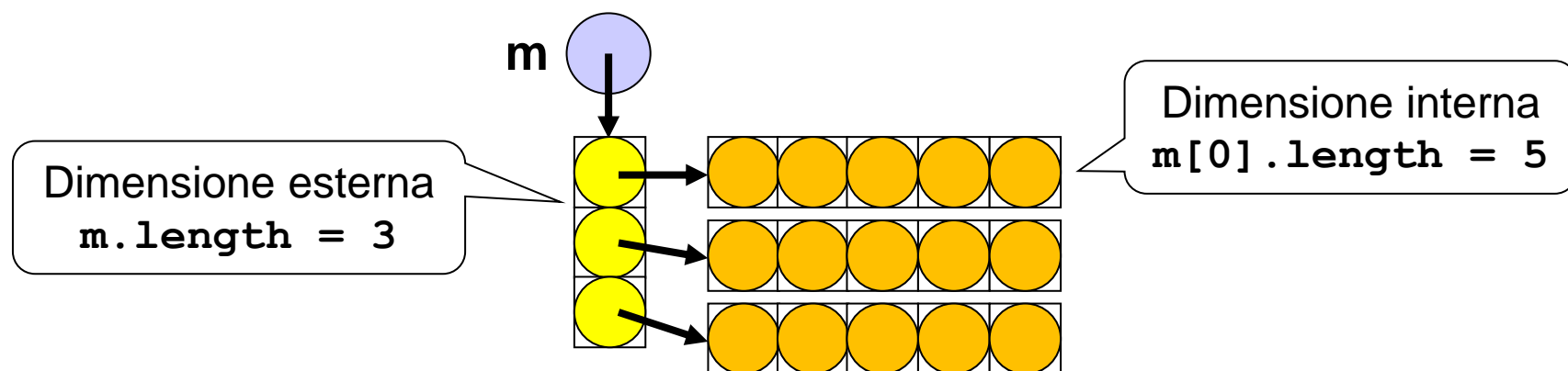
Array Multidimensionali in JAVA

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)



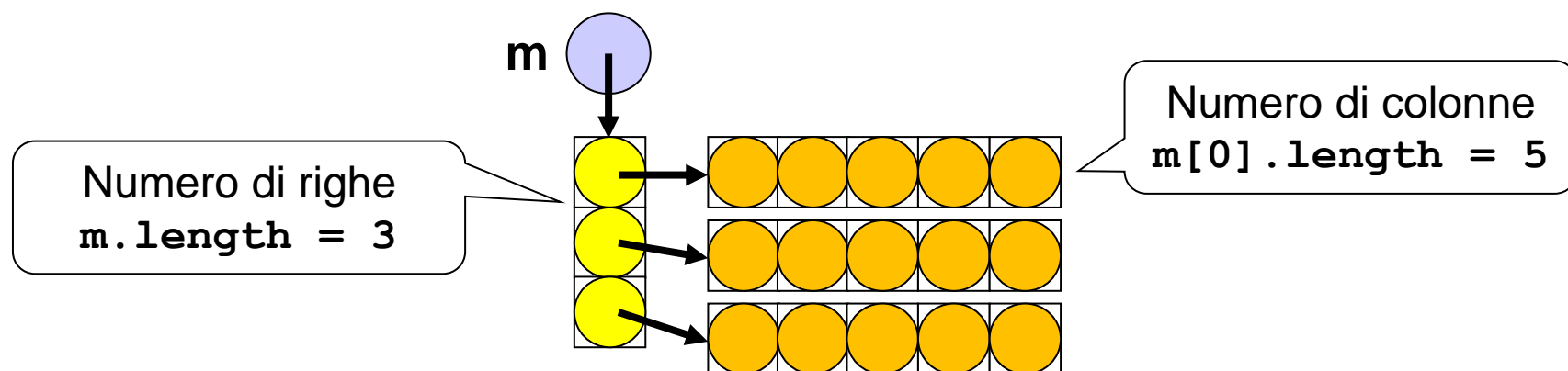
Array Multidimensionali in JAVA

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 *indice di colonna* dev'essere coerente con la precedente: qui,

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



Esempio: Somma di Matrici

```
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;  
}
```

Perché una funzione statica?

- di nuovo, perché, se fosse un metodo, dovrebbe essere nella classe “array di array di **double**”, che *non* definiamo noi
- perciò, non può che essere una funzione “classica”, di libreria, che in Java assume la forma statica



Esempio: Somma di Matrici

Mini-main di collaudo

```
public static void main(String[] args){  
    double[][] m = {{ 1, 0, 0 }, { 0, 1, 0 },  
                     { 0, 0, 1 }};  
    double[][] n = {{ 2, 1, 1 }, { 1, 1, 0 },  
                     { 0, 0, 1 }};  
    double[][] q = sommaMatrici(m,n);  
    assert(3.0 == q[0][0]); // fare il resto dei controlli  
    stampaMatrice(q);      // da fare  
}
```

Esempio: Somma di Matrici

Mini-main di collaudo

```
public static void main(String[] args) {
    double[][] m = {{ 1, 0, 0 }, { 0, 1, 0 },
                    { 0, 0, 1 }};
    double[][] n = {{ 2, 1, 1 }, { 1, 1, 0 },
                    { 0, 0, 1 }};
    double[][] q = sommaMatrici(m,n);
    assert(3.0 == q[0][0]); // fare il resto dei controlli
    stampaMatrice(q);      // da fare
}
```

Ma non sono
double? Perché
usiamo degli interi?

Il test di un numero double (o float) presenta alcuni problemi dovuti alla «precisione» del numero e facilmente portano al fallimento delle assert. Adottando il framework JUnit risolveremo la problematica..



Esercizio: Prodotto di Matrici

Analogo al precedente...

... MA *attenzione alle dimensioni delle matrici!*

Main di collaudo:

```
public static void main(String[] args){
    double[][] m = {{ 1, 0, 0 }, { 0, 1, 0 },
                    { 0, 0, 1 }};
    double[][] n = {{ 2, 1, 1 }, { 1, 1, 0 },
                    { 0, 0, 1 }};
    double[][] q = prodottoMatrici(m,n) ;
    assert(2.0 == q[0][0]) ;
    stampaMatrice(q) ;
}
```



Primo Step Esercitazione

- Implementare la classe **MatriceLib** con i relativi metodi statici
 - `public static double[][] sommaMatrici(double[][] a, double[][] b)`
 - `public static double[][] prodottoMatrici(double[][] a, double[][] b)`
 - `public static void stampaMatrice(double[][] a)`
- Nello startkit troverete già pronti **MatriceLibTest** per il collaudo della classe e **MatriceLibTestMain** per il collaudo della stampa

Tempo a disposizione: 30 minuti

Ricordatevi la direttiva **-ea** da dare alla JVM per far girare i test



Tutto Bene, dunque?

- **Non proprio:**

- la soluzione adottata **espone direttamente la rappresentazione interna**: tutti vedono che la matrice è *fatta con array bidimensionali*
- è un approccio *error-prone*, che non protegge i preziosi dati:
chi ha in mano quell'array può farci letteralmente quello che vuole
 - perché non prendere $a[i,j]$ e non moltiplicarlo per pigreco?!?
- esporre la rappresentazione interna **NON È MAI UNA BUONA IDEA**

- **Revisione critica:**

- anziché esporre e manipolare direttamente l'array, *incapsularlo in un opportuno ADT (**Matrix**) che protegga lo stato interno*
- l'accesso ai dati non è più diretto, ma **intenzionalmente mediato**: la classe **Matrix** esporrà all'esterno *un insieme di metodi stabiliti da noi* che consentiranno solo le manipolazioni sensate dei dati

Verso l'ADT "Matrix"



ADT MATRIX (1/8)

- **OBIETTIVO:** realizzare un componente Matrice (**Matrix**) come tipo di dato astratto (ADT)
- **COSA VOGLIAMO:** come minimo (interfaccia di base):
 - **Costruire** una nuova matrice partendo dai valori che la matrice dovrà contenere
 - **Ottenere** il numero di righe e colonne della matrice
 - **Ottenere** il valore dell'elemento della matrice individuato dal numero di riga e colonna
 - **Ottenere** una rappresentazione stringa della matrice
 - **Stabilire** se la matrice è quadrata




ADT MATRIX (2/8)

- **COSA VOGLIAMO in più (interfaccia estesa):**
 - **Sommare** la matrice con un'altra e restituire una nuova matrice che rappresenti il risultato
 - **Moltiplicare** la matrice con un'altra e restituire una nuova matrice che rappresenti il risultato
 - **Ottenere** una nuova matrice che rappresenti una sottomatrice della matrice corrente, partendo da una riga e colonna di partenza e dal numero di righe e colonne da considerare per l'estrazione della sottomatrice
 - **Ottenere** una nuova matrice che rappresenti il minore della matrice corrente, partendo dalla riga e colonna da rimuovere
 - **Ottenere** il determinante della matrice

ADT MATRIX (3/8)

E NATURALMENTE..

- Verificare il corretto funzionamento **di ogni singolo aspetto** del componente mediante opportuni test
 - Molti sono nello start kit, ma..
 - .. **alcuni dovreste farli VOI** 😊



Anche Matrix è
un oggetto
costante!

ADT MATRIX (4/8)

Vista generale completa

Matrix
- values: double ([][])
- calcDet() : double + det() : double + extractMinor(row :int, col :int) : Matrix + extractSubMatrix(startRow :int, startCol :int, rowCount :int, colCount :int) : Matrix + getCols() : int + getRows() : int + getValue(row :int, col :int) : double + isSquared() : boolean - Matrix(rows :int, cols :int) + Matrix(values :double[][]) + mul(m :Matrix) : Matrix - setValue(row :int, col :int, value :double) : void + sum(m :Matrix) : Matrix + toString() : String

ADT MATRIX (5/8)

Vista generale completa

Matrix	
-	values: double ([[]])
-	calcDet() : double
+	det() : double
+	extractMinor(row :int, col :int) : M
+	extractSubMatrix(startRow
+	getCols() : int
+	getRows() : int
+	getValue(row :int, col :int) : doub
+	isSquared() : boolean
-	Matrix(rows :int, cols :int)
+	Matrix(values :double[[]])
+	mul(m :Matrix) : Matrix
-	setValue(row :int, col :int, value :double) : void
+	sum(m :Matrix) : Matrix
+	toString() : String

PERCHÉ DUE COSTRUTTORI?

- Ipotesi: il cliente deve poter costruire solo matrici «piene», ben configurate: non devono esistere matrici «a metà»
- Il costruttore **pubblico** **Matrix/1** riceve perciò l'intero array bidimensionale con il contenuto

ADT MATRIX (6/8)

Vista generale completa

```
- values: double ([[ ]])

- calcDet() : double
+ det() : double
+ extractMinor(row :int, col :int) : Matrix
+ extractSubMatrix(startRow :int, startCol :int, endRow :int, endCol :int) : Matrix
+ getCols() : int
+ getRows() : int
+ getValue(row :int, col :int) : double
+ isSquared() : boolean
- Matrix(rows :int, cols :int)
+ Matrix(values :double[[ ]])
+ mul(m :Matrix) : Matrix
- setValue(row :int, col :int, value :double)
+ sum(m :Matrix) : Matrix
+ toString() : String
```

Il costruttore ausiliario NON è indispensabile

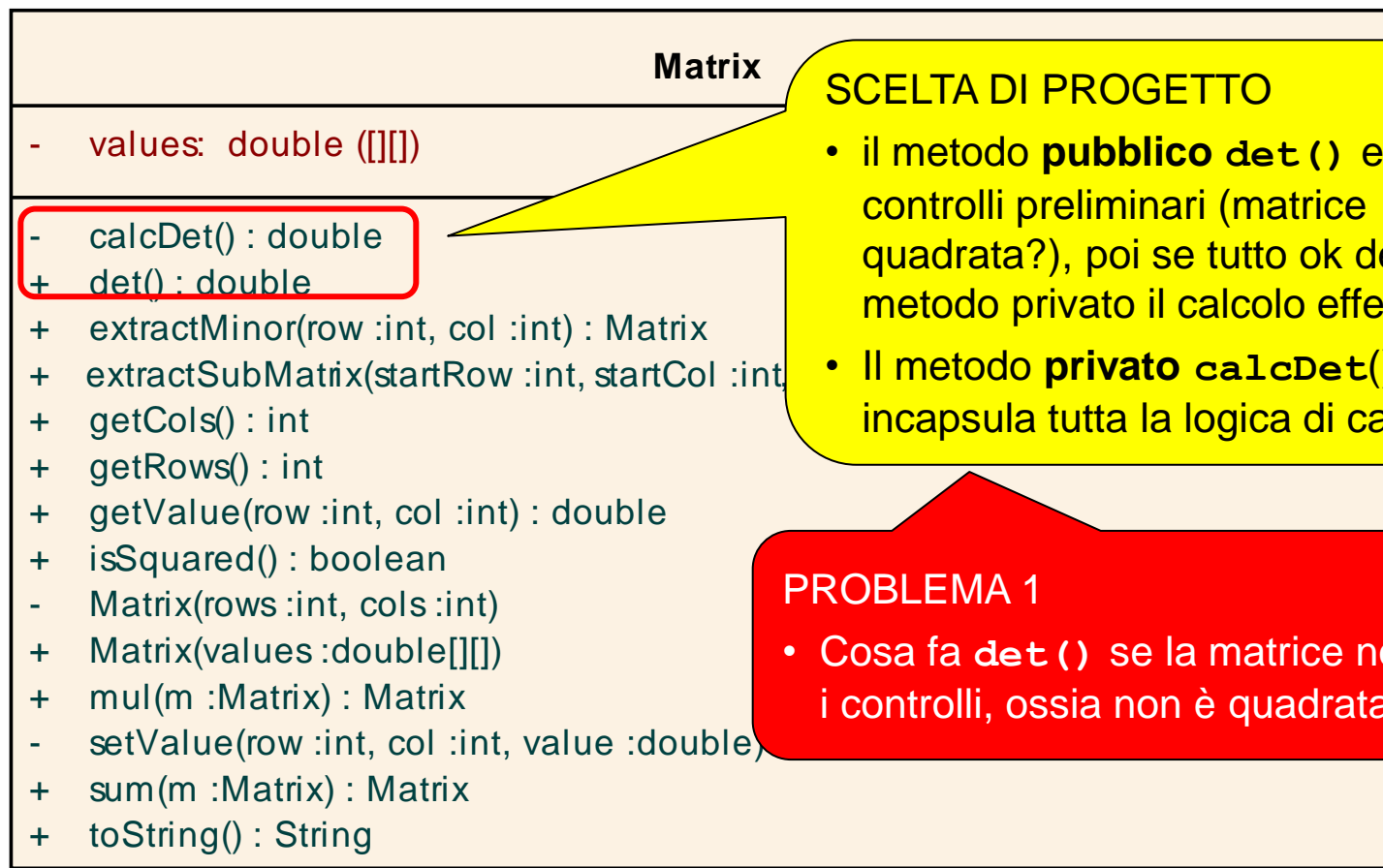
- Si sarebbe potuto anche parcheggiare gli elementi via via calcolati in un array bidimensionale temporaneo, e solo alla fine incapsularlo in una nuova matrice usando il costruttore principale

PERCHÉ DUE COSTRUTTORI?

- Tuttavia, i metodi che calcolano e producono nuove matrici (sum, mul) devono internamente calcolarne gli elementi uno ad uno
- E' quindi utile un costruttore **privato ausiliario** che costruisca una matrice vuota, riservato a tali metodi

ADT MATRIX (7/8)

Vista generale completa



SCELTA DI PROGETTO

- il metodo **pubblico** `det()` effettua i controlli preliminari (matrice quadrata?), poi se tutto ok delega al metodo privato il calcolo effettivo
- Il metodo **privato** `calcDet()` incapsula tutta la logica di calcolo

PROBLEMA 1

- Cosa fa `det()` se la matrice non passa i controlli, ossia non è quadrata?



SE IL RISULTATO NON C'È ?

- In parecchie situazioni ci si trova di fronte a **funzioni che dovrebbero restituire un valore che però non hanno**
- In questi casi
 - per i tipi-oggetto, si può **restituire null** (con tutti i rischi del caso)
 - MA **per i tipi primitivi?**
- Per tamponare almeno parzialmente il problema, **le due classi di libreria Float e Double** incorporano *servizi di utilità* rispettivamente per i tipi primitivi **float** o **double**
- In particolare definiscono **la costante NaN** (*Not a Number*)

SE IL RISULTATO NON C'È ?

- Le due classi di libreria `Float` e `Double` sono "librerie matematiche" per i tipi primitivi `float` o `double`, rispettivamente.
- Fra le altre cose, definiscono **tre costanti pubbliche**

MAX_VALUE	A constant holding the largest positive finite value of type float, $(2 \cdot 2^{23}) \cdot 2^{127}$.
MIN_VALUE	A constant holding the smallest positive nonzero value of type float, 2^{-149} .
NaN	A constant holding a Not-a-Number (NaN) value of type float.
NEGATIVE_INFINITY	A constant holding the negative infinity of type float.
POSITIVE_INFINITY	A constant holding the positive infinity of type float.

- NaN** è il valore restituito dalle *forme indeterminate* ($\infty - \infty$ o ∞ / ∞), o da *funzioni che non riescono a computare il risultato*.

ADT MATRIX (8/8)

Quindi, se il determinante non si può calcolare perché la matrice non è quadrata, possiamo **restituire NaN**

Matrix	
-	values: double ([][])
-	calcDet() : double
+	det() : double
+	extractMinor(row :int, col :int) : Matrix
+	extractSubMatrix(startRow :int, startCol :int, rowCount :int, colCount :int)
+	getCols() : int
+	getRows() : int
+	getValue(row :int, col :int) : double
+	isSquared()
-	Matrix(row :int, col :int)
+	Matrix(values : double [])
+	mul(m : Matrix) : Matrix
-	setValue(row :int, col :int, value : double)
+	sum(m : Matrix) : Matrix
+	toString() : String

```
public double det() {
    return isSquared() ? calcDet() : Double.NaN;
}
```

Metodo **pubblico** det ()

- effettua i controlli preliminari (matrice quadrata?)
- se tutto ok delega al metodo privato il calcolo effettivo

Metodo **privato** calcDet()

- incapsula la logica di calcolo

Se la matrice **NON** è quadrata

- si segnala errore restituendo NaN



Interfaccia di Base (1)

- **Stato:**
 - scegliamo un **array bidimensionale** di **double**
- **Due costruttori:**
 - **uno privato, a uso interno**
(due argomenti: numero di righe, numero di colonne)
 - **uno pubblico**
(un solo argomento: un array bidimensionale di **double** contenente tutti gli elementi che compongono la matrice)
 - **REQUISITO DI CONSISTENZA:** l'array ricevuto come argomento deve essere **ricopiato su quello interno valore per valore**, perché se si copiasse solo il riferimento qualcuno da fuori potrebbe cambiare lo stato interno dell'oggetto, violando l'incapsulamento → **pericoloso**
 - **MA** ciò è chiaramente oneroso: *si potrebbe evitare se la struttura dati fosse imm modificabile...*



Interfaccia di Base (2)

- **Metodi *accessor*** per ottenere informazioni sul componente:
 - ottenere il numero di righe o colonne (**getRows** / **getCols**)
 - ottenere il valore di un particolare elemento della matrice (**getValue**)
 - **non vogliamo** un metodo per impostare il valore di un elemento, perché anche la matrice, come una stringa, dev'essere un oggetto costante
 - una volta costruita, non deve essere alterata
 - potrà però esserci utile un metodo **privato setValue** a uso interno...
- **Altri metodi** per:
 - ottenere la rappresentazione in stringa del componente (**toString**)
 - stabilire se la matrice è quadrata (**isSquared**)



Dritta sui Costruttori

- **Chi crea l'array bidimensionale** che conterrà i valori?
 - i costruttori!
- **In particolare:**
 - l'array bidimensionale va allocato in entrambi i casi
 - nel caso di **costruttore privato** con due argomenti interi (righe, colonne), creare un array bidimensionale grande quanto indicato dai parametri
 - nel caso di **costruttore pubblico** con un argomento array bidimensionale, creare un array bidimensionale grande come quello ricevuto

Interfaccia Estesa (1)

ULTERIORI METODI per

- ottenere una nuova matrice che rappresenti il **minore** della matrice (**quadrata**) corrente (**extractMinor**). In particolare, dati in ingresso gli indici di una riga e di una colonna, il metodo deve restituire una matrice:
 1. identica a quella corrente, *TRANNE per le riga e colonna specificate*
 2. la riga e la colonna indicate in ingresso devono essere *soppresse*
- calcolare il **determinante** (**det**) della matrice sapendo che, per una **matrice quadrata $n \times n$** , esso è definito come:

$$\det(A_k) = \sum_{i=1}^n (-1)^{i+k} a_{i,k} \det(A_{i,k})$$

- $a_{i,k}$ è l'elemento di coordinate i,k
- $A_{i,k}$ è il minore ottenuto sopprimendo la i -esima e la k -esima colonna (ottenibile tramite **extractMinor**)



Interfaccia Estesa (2)

ULTERIORI METODI per

- estrarre dalla matrice corrente una nuova matrice che rappresenti una sua **sottomatrice** (**extractSubMatrix**). A tal fine, dovranno essere passati i seguenti parametri di ingresso:
 1. gli indici di riga e colonna da cui partire per l'estrazione della sottomatrice
 2. Il numero di righe e colonne che comporranno la sottomatrice
- ottenere una nuova matrice **somma** (**sum**) della matrice corrente e di una matrice passata come parametro di ingresso
- ottenere una nuova matrice **prodotto** (**mul**) della matrice corrente e di una matrice passata come parametro di ingresso

Interfaccia Estesa (3)

PRECONDIZIONI

- Attenzione alla correttezza dei parametri in ingresso!
- Che fare se:
 - al metodo **extractMinor** vengono passati *indici di riga/colonna «esterni» alla matrice corrente* oppure se la matrice *non è quadrata*?
 - al metodo **extractSubMatrix** vengono passati *indici di riga o colonna «esterni» alla matrice corrente* o vengono richieste *più righe/colonne di quante disponibili*?
 - ai metodi **sum/mul** vengono passate *matrici non compatibili* con l'operazione?

Non avendo ancora gli strumenti adatti, per ora ci accontenteremo di restituire `null` ...
...ben sapendo però che è insoddisfacente perché non impedisce la prosecuzione e non fa capire cosa sia successo.

Test Interfaccia Estesa

- I test sono nello start kit!



I test verificano il funzionamento di tutti i metodi

- sia nel caso di **parametri corretti**
- sia nei casi di **parametri errati**
- Se pensate che manchi qualche test... può essere 😊
Fatelo voi!



Secondo Step Esercitazione

- Implementare la classe **Matrix** con i relativi metodi
- Nello Startkit troverete già pronti **MatrixTestMain** contenente il main di collaudo, **MatrixBaseInterfaceTest** e **MatrixAdvancedInterfaceTest** per collaudare rispettivamente l'interfaccia base e quella avanzata

Tempo a disposizione: 1 ora e 30 minuti

Ricordatevi la direttiva **-ea** da dare alla JVM per far girare i test