Algebretta

Sezioni

La traccia

Scopo della prova è progettare e implementare una gerarchia di oggetti utili a rappresentare una *calcolatrice* per operazioni su *vettori* e *matrici quadrate* a valori interi.

I vettori

Una plausibile **interfaccia** per un *vettore* a valori interi potrebbe avere i seguenti metodi

```
int dim();

int val(final int i);

Vettore per(final int alpha);

Vettore più(final Vettore v);
```

[sorgente]

dove `dim` restituisce la dimensione del vettore (ossia il numero delle sue componenti), `val` restituisce il valore dell'i-esima componente del vettore, `più` restituisce la somma vettoriale tra il vettore corrente e quello dato (che è ovviamente possibile solo se i vettori sono *conformi*, ossia della stessa dimensione), mentre `per` restituisce il prodotto del vettore corrente per lo scalare `alpha`.

Scrivete la **classe** (concreta) `VettoreDenso` che la implementi memorizzando le componenti del vettore in un *array* di `int`. Tale classe deve avere un costruttore che prenda un *array* di interi come argomento (e costruisca il vettore avente come componenti tutti e soli gli elementi dell'array); infine, sovrascrivete il metodo `toString` in accordo a quanto mostrato negli esempi seguenti.

Le matrici

Prendendo spunto da quanto fatto per i vettori, definite ora la parte riguardante le **matrici**, limitatamente al caso di *matrici quadrate* a *valori interi*. Le operazioni che tali matrici devono consentire (oltre a quelle utili a conoscere le loro dimensioni e componenti) sono quelle utili a calcolare:

- il *prodotto per scalare*,
- la *somma matriciale*,
- il *prodotto matriciale*.

Osservate che, come è noto dall'algebra lineare, ci sono diverse matrici con proprietà particolari, ad esempio:

- la matrice *nulla* (che ha tutti i componenti pari a 0),
- le matrici *diagonali* (che hanno i componenti fuori dalla diagonale pari a 0),
- la matrice *identità* (che ha i componenti sulla diagonale principale pari 1 e tutti gli altri pari a 0).

Per tali matrici è sensato provvedere delle implementazioni specializzate.

Estensioni facoltative

Una volta implementate le operazioni precedenti, potete aggiungere l'operazione di *prodotto di una matrice per un vettore* osservando che, per realizzarla, potrebbe aver senso provvedere una implementazione specializzata del *vettore nullo* (che risulta, ad esempio, dal prodotto della matrice nulla per qualunque altro vettore).

Un'altra possibile estensione consiste nelle implementazioni specializzate al caso di matrici *sparse* (che abbiano cioè un numero di elementi diversi da 0 dell'ordine della loro dimensione).

Comportamento esibito

Scrivete un metodo statico `main` nella classe che ritenete più opportuna che legga dal flusso di ingresso una (rappresentazione testuale di una) serie di semplici operazioni tra matrici e vettori e ne stampi (qualora sia possibile) il risultato.

Le operazioni da considerare sono solo quelle tra *due soli operandi* (scalari, vettori e matrici), limitatamente ai caso di:

- prodotto scalare vettore,
- somma tra vettori,
- prodotto scalare matrice,
- somma tra matrici,
- prodotto tra matrici,
- prodotto tra matrice e vettore.

Somme e prodotti sono indicati rispettivamente da `+` e `*`, i vettori sono rappresentati come un elenco di interi separati da virgole e racchiusi tra parentesi tonde, mentre le matrici sono rappresentate, in generale, da un elenco di righe separate da punti e virgola e racchiuse tra parentesi quadre, dove ogni riga è data da un elenco di interi separati da virgole. Pertanto, la stringa `(1, 2, 3, 4, 5, 6, 7, 8, 9)` rappresenta il vettore

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

mentre la stringa `[1, 2, 3; 4, 5, 6; 7, 8, 9]` rappresenta la matrice

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

Alcune matrici hanno una rappresentazione particolare:

- la matrice nulla è rappresentata dalla lettera `Z` seguita dalla dimensione (racchiusa tra quadre),
- le matrici diagonali sono rappresentate dalla lettera `D` seguita dall'elenco delle componenti sulla diagonale principale (racchiuse tra quadre e separate da virgole),
- la matrice identità è rappresentata dalla lettera `I` seguita dalla dimensione (racchiusa tra quadre),

Decodifica dell'input

Estrarre operandi, operatore, scalari, vettori e matrici (comprese quelle con rappresentazione particolare) da ciascuna stringa non è affatto banale. Per questa ragione avete a disposizione una classe (statica) denominata `Parser`, che offre una serie di metodi (statici, dotati di commenti Javadoc) che consentono di estrarre le informazioni necessarie a costruire gli opportuni oggetti a partire dalla stringa corrispondente a una delle operazioni da trattare.

Esempio

Eseguendo `soluzione` e avendo

```
2 * (3, 4)
(5, 6) + (7, 8)
2 * [3, 4; 5, 6]
[3, 4; 5, 6] + [3, 4; 5, 6]
[3, 4; 5, 6] * (3, 5)
[3, 4; 5, 6] * (4, 6)
[3, 4; 5, 6] * [3, 4; 5, 6]
```

nel flusso d'ingresso, il programma emette

```
(6, 8)
(12, 14)
[6, 8; 10, 12]
[6, 8; 10, 12]
(29, 45)
(36, 56)
[29, 36; 45, 56]
```

nel flusso d'uscita. Eseguendo `soluzione` e avendo

```
2 * I[2]
2 * Z[2]
2 * D[3, 4]
I[2] * (3, 4)
Z[2] * (3, 4)
D[3, 4] * (3, 4)
```

nel flusso d'ingresso, il programma emette

```
[2, 0; 0, 2]
[0, 0; 0, 0]
[6, 0; 0, 8]
(3, 4)
(0, 0)
(9, 16)
```

nel flusso d'uscita.

Suggerimenti

Si ricorda che il prodotto matriciale tra M ed N è la matrice MN tale che $(MN)_{ij} = \sum_r M_{ir}N_{rj}$. Inoltre, il prodotto della matrice M con il vettore v è il vettore Mv tale che $(Mv)_i = \sum_r M_{ir}v_r$. Si noti che, in entrambi i casi, n è la dimensione delle matrici e dei vettori coinvolti e la somma è per r che va da 0 a n-1. Ovviamente, fatta eccezione per il prodotto per scalare, le altre operazioni sono possibili solo se le matrici, o la matrice e il vettore, hanno la stessa dimensione (ossia sono conformi).

La soluzione

I vettori

Iniziamo con una osservazione sull'interfaccia: prima di effettuare una operazione tra vettori o matrici è necessario sapere se due vettori sono *conformi*, o se un vettore è *conforme* ad una matrice; dato che tale informazione dipende solo dalla dimensione (che è una competenza espressa dalle interfacce), può aver senso aggiungere due metodi (sovraccaricati) di *default*

```
default boolean conforme(final Vettore v) {
  return dim() == v.dim();
}

default boolean conforme(final Matrice M) {
  return dim() == M.dim();
}
```

[sorgente]

Ora, la classe concreta da implementare è completamente specificata, non solo astrattamente, ma anche dal punto di vista della rappresentazione; resta solo da stabilire che (come è plausibile dato il tipo d'uso descritto dal progetto) il vettore sia *immutabile*. Date le

scelte fatte è immediato concludere che gli invarianti sono che l'array non sia un riferimento nullo e che contenga almeno un elemento e che tali invarianti possono essere controllati solo in costruzione

```
private final int[] val;

public VettoreDenso(final int dim) {
  if (dim <= 0) throw new IllegalArgumentException("La dimensione deve essere positiva.");
  val = new int[dim];
}

public VettoreDenso(final int[] val) {
  Objects.requireNonNull(val);
  if (val.length == 0)
    throw new IllegalArgumentException("Il vettore deve comprendere almeno un valore.");
  this.val = val.clone();  }
```

[sorgente]

Osservate l'uso del metodo `clone` che ha l'obiettivo di costruire una copia dell'array passato come argomento; questo accorgimento serve ad evitare che chi ha invocato li costruttore possa mantenere un riferimento all'array che costituisce la rappresentazione del vettore.

L'implementazione dei metodi prescritti dall'interfaccia è ovvia, se ne rimanda la presentazione alla sezione sulle estensioni (dato che l'aggiunta del vettore nullo rende in parte più sofisticati anche i metodi del vettore qui sviluppato).

Le matrici

L'interfaccia (e classe astratta)

Per iniziare, è opportuno sviluppare una *interfaccia* `Matrice` che esprima le competenze richieste per tutte le matrici

```
int dim();

int val(final int i, final int j);

Matrice per(final int alpha);

Matrice più(final Matrice B);

Matrice per(final Matrice B);
```

[sorgente]

Come nel caso dei vettori, può aver senso aggiungere dei metodi di *default* che consentano di valutare la conformità e (per comodità) se due coordinate siano valide.

```
default boolean conforme(final Vettore v) {
  return dim() == v.dim();
}

default boolean conforme(final Matrice M) {
  return dim() == M.dim();
}

default void requireValidIJ(final int i, final int j) {
  if (0 <= i && i < dim() && 0 <= j && j < dim()) return;
  throw new IndexOutOfBoundsException("Gli indici eccedono la dimensione della matrice.");
}
```

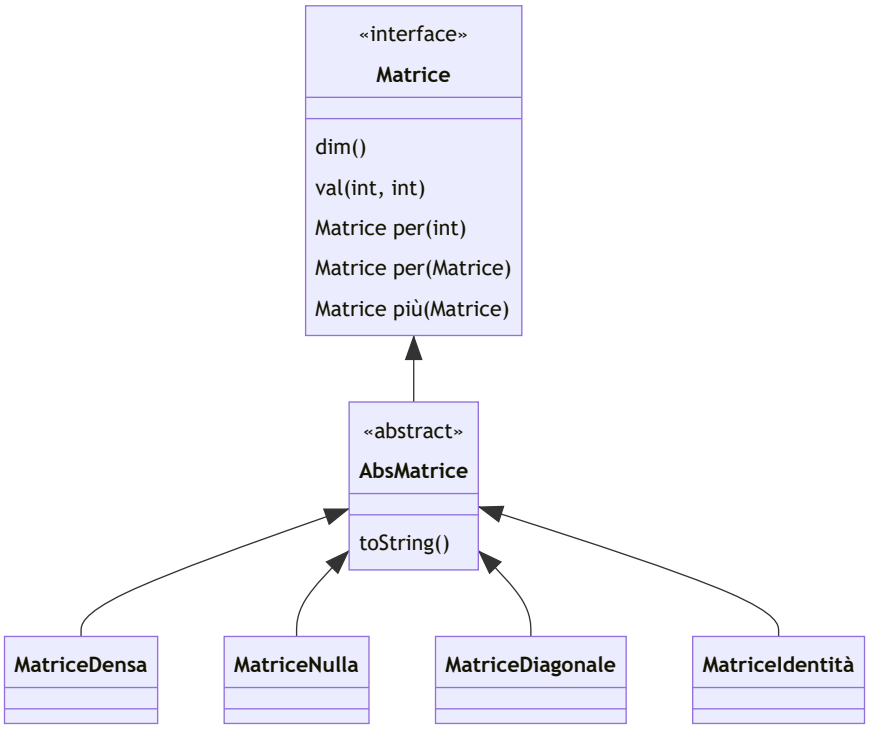
[sorgente]

Sarebbe utile sovrascrivere qui anche il metodo `toString` (che dipende solo dalle competenze nell'interfaccia), ma sfortunatamente non è possibile usare un metodo di default per farlo; a tale sopo può essere introdotta una *classe astratta* che implementi (parzialmente) l'interfaccia di fatto soltanto sovrascrivendo `toString`

```java
@Override
public String toString() {
  final StringBuilder sb = new StringBuilder();
  sb.append("[");
  for (int i = 0; i < dim(); i++) {
    for (int j = 0; j < dim(); j++) sb.append(val(i, j) + (j < dim() - 1 ? ", " : ""));
    if (i < dim() - 1) sb.append("; ");
  }
  sb.append("]");
  return sb.toString();
}
```

[sorgente]

Ora non manca che sviluppare le classi per una matrice densa e per quelle speciali; ad alto livello ci stiamo apprestando a sviluppare questa gerarchia



Si potrebbe essere tentati di ritenere la matrice identità come un sottotipo di matrice diagonale, questo ha senz'altro senso dal punto di vista algebrico, ma non da quello implementativo, come sarà più chiaro in seguito.

La matrice densa

Iniziamo dalla matrice densa, la cui rappresentazione sarà data da un array bidimensionale; anche in questo caso, essendo sensato che le matrici siano immutabili, gli invarianti sono che l'array non sia un riferimento a null, sia «quadrato» e di dimensione almeno 1 x 1 e possono essere controllati solo in costruzione.

```
  private final int[][] mat;

  private MatriceDensa(final int dim) {
    if (dim < 0) throw new IllegalArgumentException("La dimensione deve essere positiva.");
    mat = new int[dim][dim];
  }

  public MatriceDensa(final int[][] mat) {
    Objects.requireNonNull(mat);
    if (mat.length == 0) throw new IllegalArgumentException("La dimensione deve essere
positiva.");
    final int dim = mat.length;
    this.mat = new int[dim][dim];
    for (int i = 0; i < dim; i++) {
      if (mat[i].length != dim) throw new IllegalArgumentException("L'array deve essere
quadrato.");
      for (int j = 0; j < dim; j++) this.mat[i][j] = mat[i][j];
    }
  }

  public MatriceDensa(final Matrice A) {
    this(Objects.requireNonNull(A).dim());
    for (int i = 0; i < dim(); i++) for (int j = 0; j < dim(); j++) mat[i][j] = A.val(i, j);
  }
```

[sorgente]

Si osservi il «costruttore copia» che potrà risultare comodo per costruire matrici densa a partire da altre matrici (in particolare, quelle speciali).

Nel soddisfare l'interfaccia, osserviamo che alcuni metodi sono banali da implementare:

```
  @Override
  public int dim() {
    return mat.length;
  }

  @Override
  public int val(final int i, final int j) {
    requireValidIJ(i, j);
    return mat[i][j];
  }
```

[sorgente]

Nel caso delle operazioni, occorre prestare attenzione che in alcune circostanze può essere vantaggioso restituire matrici speciali, come ad esempio nel caso di moltiplicazione per lo scalare zero (evidenziato nella seguente porzione di codice):

```
  @Override
  public Matrice per(final int alpha) {
    if (alpha == 0) return new MatriceNulla(dim());
    final MatriceDensa N = new MatriceDensa(dim());
    for (int i = 0; i < dim(); i++) for (int j = 0; j < dim(); j++) N.mat[i][j] = alpha *
mat[i][j];
    return N;
  }
```

[sorgente]

o di somma con la matrice nulla, nel qual caso va restituita la matrice stessa (soluzione resa possibile dall'immutabilità):

```
  @Override
  public MatriceDensa più(final Matrice B) {
    Objects.requireNonNull(B);
    if (!conforme(B)) throw new IllegalArgumentException("Le matrici non sono conformi.");
    if (B instanceof MatriceNulla) return this;
    final MatriceDensa C = new MatriceDensa(dim());
    for (int i = 0; i < dim(); i++)
      for (int j = 0; j < dim(); j++) C.mat[i][j] = mat[i][j] + B.val(i, j);
    return C;
  }
```

[sorgente]

o di prodotto sia con la matrice nulla (che risulta nella matrice nulla), o con la matrice identità (che risulta la matrice stessa):

```java
@Override
public Matrice per(final Matrice B) {
  Objects.requireNonNull(B);
  if (!conforme(B)) throw new IllegalArgumentException("Le matrici non sono conformi.");
  if (B instanceof MatriceNulla) return B;
  if (B instanceof MatriceIdentità) return this;
  final MatriceDensa C = new MatriceDensa(dim());
  for (int i = 0; i < dim(); i++)
    for (int j = 0; j < dim(); j++)
      for (int k = 0; k < dim(); k++) C.mat[i][j] += mat[i][k] * B.val(k, j);
  return C;
}
```

[sorgente]

La matrice nulla

Caso del tutto banale è quello della matrice nulla. La sua rappresentazione coincide esclusivamente con la sua dimensione, quindi l'invariante e i costruttori sono cosa ovvia:

```java
private final int dim;

public MatriceNulla(final int dim) {
  if (dim < 0) throw new IllegalArgumentException();
  this.dim = dim;
}
```

[sorgente]

Alcune competenze prescritte dall'interfaccia sono immediati da implementare:

```java
@Override
public int dim() {
  return dim;
}

@Override
public int val(final int i, final int j) {
  requireValidIJ(i, j);
  return 0;
}
```

[sorgente]

Nel caso delle operazioni, le proprietà algebriche delle matrici si riflettono in modo ovvio nel codice:

```java
@Override
public Matrice per(final int alpha) {
  return this;
}

@Override
public Matrice per(final Matrice B) {
  Objects.requireNonNull(B);
  if (!conforme(B)) throw new IllegalArgumentException("Le matrici non sono conformi.");
  return this;
}

@Override
public Matrice più(final Matrice B) {
  Objects.requireNonNull(B);
  if (!conforme(B)) throw new IllegalArgumentException("Le matrici non sono conformi.");
  return B;
}
```

[sorgente]

Unica accortezza è sollevare le necessarie eccezioni in ottemperanza alle specifiche dell'interfaccia.

La matrice diagonale

Nel caso della matrice diagonale, essendo sufficiente ricordare solo i valori lungo la diagonale, la rappresentazione è un array monodimensionale di interi (per cui valgono considerazioni analoghe alle precedenti per costruttori e invariante):

```
  private final int[] diagonale;

  public MatriceDiagonale(final int[] diagonale) {
    Objects.requireNonNull(diagonale);
    if (diagonale.length == 0)
      throw new IllegalArgumentException("La diagonale deve contenere almeno un valore.");
    this.diagonale = diagonale.clone();
  }
```

[sorgente]

Alcuni metodi prescritti dall'interfaccia hanno al solito implementazioni ovvie:

```
  @Override
  public int dim() {
    return diagonale.length;
  }

  @Override
  public int val(final int i, final int j) {
    requireValidIJ(i, j);
    return i == j ? diagonale[i] : 0;
  }
```

[sorgente]

Come per la matrice densa, nel caso delle operazioni, in alcuni circostanze può essere vantaggioso restituire matrici speciali, come ad esempio nel caso di moltiplicazione per lo scalare zero (evidenziato nella seguente porzione di codice):

```
  @Override
  public Matrice per(final int alpha) {
    if (alpha == 0) return new MatriceNulla(dim());
    int[] tmp = new int[diagonale.length];
    for (int i = 0; i < diagonale.length; i++) tmp[i] = alpha * diagonale[i];
    return new MatriceDiagonale(tmp);
  }
```

[sorgente]

o di somma con la matrice nulla, nel qual caso va ancora restituita la matrice stessa:

```
  @Override
  public Matrice più(final Matrice B) {
    Objects.requireNonNull(B);
    if (!conforme(B)) throw new IllegalArgumentException("Le matrici non sono conformi.");
    if (B instanceof MatriceNulla) return this;
    return new MatriceDensa(this).più(B);
  }
```

[sorgente]

o di prodotto sia con la matrice nulla (che risulta nella matrice nulla), o con la matrice identità (che risulta la matrice stessa):

```
  @Override
  public Matrice per(final Matrice B) {
    Objects.requireNonNull(B);
    if (!conforme(B)) throw new IllegalArgumentException("Le matrici non sono conformi.");
    if (B instanceof MatriceNulla) return B;
    if (B instanceof MatriceIdentità) return this;
    return new MatriceDensa(this).per(B);
  }
```

[sorgente]

La matrice identità

Come nel caso della matrice nulla, anche per l'identità è sufficiente ricordare la sola dimensione, quindi rappresentazione, invariante e costruttori sono:

```
  private final int dim;

  public MatriceIdentità(final int dim) {
    if (dim < 0) throw new IllegalArgumentException("La dimensoine dev'essere positiva.");
    this.dim = dim;
  }
```

[sorgente]

Al solito, alcune competenze prescritte dall'interfaccia sono immediate da implementare:

```
  @Override
  public int dim() {
    return dim;
  }

  @Override
  public int val(final int i, final int j) {
    requireValidIJ(i, j);
    return i == j ? 1 : 0;
  }
```

[sorgente]

Il prodotto per scalare e la somma trattano al solito in modo speciale il caso dello zero:

```
  @Override
  public Matrice per(final int alpha) {
    if (alpha == 0) return new MatriceNulla(dim());
    int[] tmp = new int[dim];
    for (int i = 0; i < dim; i++) tmp[i] = alpha;
    return new MatriceDiagonale(tmp);
  }
```

[sorgente]

```
  @Override
  public Matrice più(final Matrice B) {
    Objects.requireNonNull(B);
    if (!conforme(B)) throw new IllegalArgumentException("Le matrici non sono conformi.");
    if (B instanceof MatriceNulla) return this;
    return new MatriceDdensa(this).più(B);
  }
```

[sorgente]

mentre il caso del prodotto è elementare per via delle proprietà algebriche:

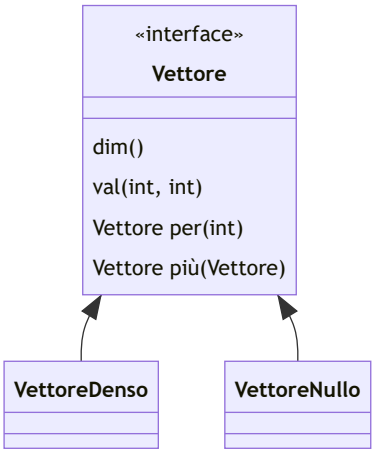
```
  @Override
  public Matrice per(final Matrice B) {
    Objects.requireNonNull(B);
    if (!conforme(B)) throw new IllegalArgumentException("Le matrici non sono conformi.");
    return B;
  }
```

[sorgente]

Le estensioni

Il vettore nullo

Nell'implementazione dei prodotti matrice vettore, avendo a che fare con la matrice nulla, può risultare utile avere una implementazione del vettore nullo. Questo completa la gerarchia relativa ai vettori che diventa:



Iniziamo con la rappresentazione che, come nel caso della matrice nulla, è data soltanto dalla dimensione:

```
private final int dim;

public VettoreNullo(final int dim) {
  if (dim <= 0) throw new IllegalArgumentException("La dimensione dev'essere positiva.");
  this.dim = dim;
}
```

[sorgente]

Le competenze più immediate da implementare sono:

```
@Override
public int dim() {
  return dim;
}

@Override
public int val(int i) {
  if (i < 0 || i >= dim)
    throw new IndexOutOfBoundsException("L'indice eccede la dimensoine del vettore.");
  return 0;
}
```

[sorgente]

Le operazioni sono anch'esse semplici, date le proprietà algebriche:

```
@Override
public VettoreNullo per(int alpha) {
  return this;
}

@Override
public Vettore più(Vettore v) {
  Objects.requireNonNull(v);
  if (!conforme(v)) throw new IllegalArgumentException();
  return v;
}
```

[sorgente]

La moltiplicazione matrice vettore

Per aggiungere la funzionalità della moltiplicazione tra matrice e vettore è sufficiente aggiungere una competenza all'interfaccia

```
Vettore per(final Vettore v);
```

[sorgente]

che poi sarà implementata in modo semplice nei vari tipi di matrice, a partire dalla densa:

```
@Override
public Vettore per(final Vettore v) {
  Objects.requireNonNull(v);
  if (!conforme(v))
    throw new IllegalArgumentException("Il vettore e la matrice non sono conformi.");
  if (v instanceof VettoreNullo) return v;
  final int[] temp = new int[mat.length];
  for (int i = 0; i < mat.length; i++)
    for (int j = 0; j < mat.length; j++) temp[i] += mat[i][j] * v.val(j);
  return new VettoreDenso(temp);
}
```

[sorgente]

alla nulla:

```
@Override
public VettoreNullo per(final Vettore v) {
  Objects.requireNonNull(v);
  if (!conforme(v))
    throw new IllegalArgumentException("Il vettore e la matrice non sono conformi.");
  return new VettoreNullo(dim);
}
```

[sorgente]

per passare alla diagonale:

```
@Override
public Vettore per(final Vettore v) {
  Objects.requireNonNull(v);
  if (!conforme(v))
    throw new IllegalArgumentException("Il vettore e la matrice non sono conformi.");
  if (v instanceof VettoreNullo) return v;
  final int[] temp = new int[diagonale.length];
  for (int i = 0; i < diagonale.length; i++) temp[i] = diagonale[i] * v.val(i);
  return new VettoreDenso(temp);
}
```

[sorgente]

e infine all'identità:

```
@Override
public Vettore per(final Vettore v) {
  Objects.requireNonNull(v);
  if (!conforme(v))
    throw new IllegalArgumentException("Il vettore e la matrice non sono conformi.");
  return v;
}
```

[sorgente]

Si osservi il codice evidenziato che tratta i casi speciali dovuti al vettore nullo (che sono rilevanti solo per la matrice densa e diagonale).

L'uso di `instanceof`

In generale, differenziare il comportamento all'interno del codice di un metodo tramite l'uso di `instanceof` (o di strategie analoghe) è un segno di cattiva progettazione ad oggetti: il modo più indicato, nella programmazione orientata agli oggetti, per gestire il polimorfismo è infatti l'uso della sovrascrittura e del sovraccaricamento dei metodi.

Va però osservato che, per via del meccanismo di dispatching, operando su istanze di tipo apparente `Matrice` e `Vettore`, potrebbe capitare che non venga di fatto mai eseguito il codice «ottimizzato»: la scelta della segnatura del metodo da eseguire effettua la ricerca sul tipo apparente dell'istanza su cui è invocato e del parametro passato!

Per ovviare a questo limite, nel caso di una gerarchia come la presente che comprende pochi tipi e che difficilmente è soggetta ad ulteriore espansione (non ci sono molte altre matrici «speciali»), l'uso di `instanceof` può costituire una ragionevole ottimizzazione (anche se certamente non necessaria ai fini della correttezza).

La classe di test

Nella scrittura è possibile avvalersi della classe di utilità `Parser` che offre i seguenti metodi statici (commentati nel Javadoc della classe stessa).

```
public static String[] partiOperazione(final String linea);
public static boolean èMatrice(final String operando);
public static char tipoMatrice(final String operando);
public static int[][] valoriMatrice(final String operando);
public static boolean èVettore(final String operando);
public static int[] valoriVettore(final String operando);
public static boolean èScalare(final String operando);
public static int valoreScalare(final String operando);
```

Tali metodi possono essere utilizzati, in un ciclo che consumi l'input per righe, per suddividere ciascuna riga nelle parti dell'operazione (operandi e operatore) e trattare adeguatamente addizioni e moltiplicazioni (gestendo in tal caso l'eventualità che l'operando di sinistra sia, o meno, scalare):

```
try (final Scanner s = new Scanner(System.in)) {
  while (s.hasNextLine()) {
    final String[] lor = Parser.partiOperazione(s.nextLine());
    final char op = lor[1].charAt(0);
    final String left = lor[0], right = lor[2];
    if (op == '+') {
      ...
    } else { // op == '*', altrimenti partiOperazione solleva eccezione
      if (Parser.èScalare(left)) {
        ...
      } else if (Parser.èMatrice(left)) {
        ...
      }
    }
  }
```