Department of Information Engineering

Array
Ricorsione

Department of Information Engineering

Array
(capitolo 7)

Nota: la sezione 7.7 del libro di testo non fa parte del programma d'esame

Memorizzare una serie di valori

- Problema: scrivere un programma che
  - legge da standard input una sequenza di **10** numeri double
  - chiede all'utente un numero intero **index**
  - visualizza il numero che nella sequenza occupava la **posizione indicata da index**
- Occorre memorizzare **tutti i valori** della sequenza
  - Potremmo usare **10** variabili diverse per memorizzarli
  - Poi selezionarli con una lunga sequenza di alternative.
  - E se i valori fossero **1000**? O un numero non noto?

- In **Java** (e in quasi tutti i linguaggi di programmazione) si usa un **array** (ovvero, sequenza ordinata) per memorizzare una sequenza di dati **dello stesso tipo**.

Costruire un array

- Un array in Java è un **oggetto**
- Come ogni oggetto, deve essere costruito con l'operatore **new**, dichiarando il **tipo di dati** che potrà contenere

```
new double[10];
```

- Il tipo di dati di un array può essere qualsiasi tipo di dati valido in Java
  - Uno dei tipi di **dati fondamentali** o una **classe**
  - Potremo avere quindi array di numeri **interi**, di numeri in **virgola mobile**, di **stringhe**, di **conti bancari**...
- Nella costruzione il tipo di dati è seguito da una **coppia di parentesi quadre** che contiene la **dimensione** dell'array, cioè il numero di elementi che potrà contenere

Riferimento a un array

- Come succede con la costruzione di ogni oggetto, l'operatore **new** restituisce un **riferimento** all'array appena creato, che può essere memorizzato in una **variabile oggetto** dello stesso tipo

```
double[] values = new double[10];
```

- **Attenzione**: nella definizione della variabile oggetto devono essere presenti le parentesi quadre, ma non deve essere indicata la dimensione dell'array; la variabile potrà riferirsi solo ad array di quel tipo, ma di qualunque dimensione

```
// si può fare in due passi
double[] values;
values = new double[10];
```

Utilizzare un array: indici

- Al momento della costruzione, gli elementi dell'array vengono inizializzati seguendo le **stesse regole** viste per le variabili di esemplare

```
double[] values = new double[10]; //gli elementi valgono 0
```

- **Accedere** in lettura a un elemento dell'array

```
double oneValue = values[3];
```

- **Modificare** un elemento dell'array

```
values[5] = 3.4;
```

- Il numero usato per accedere ad un elemento dell'array si chiama **indice**
  - Può assumere valori tra **0** (**incluso:** il **primo** elemento ha indice **0**) e la **dimensione** dell'array (**esclusa**: l'**ultimo** elemento ha indice **dimensione-1**)
  - Stesse convenzioni viste per i caratteri in una stringa

Utilizzare un array: indici

- L'indice di un elemento di un array può, in generale, essere un'**espressione** con valore intero

```
double[] values = new double[10];
int a = 4;
values[a + 2] = 3.2; // modifica il
                     // settimo elemento
```

- Cosa succede se si accede a un elemento dell'array con un indice **sbagliato** (maggiore o uguale alla dimensione, o negativo) ?
  - l'ambiente di esecuzione genera un'eccezione di tipo **ArrayIndexOutOfBoundsException**

La dimensione di un array

- Un array è un oggetto un po' strano…
  - non ha metodi pubblici
- L'unico elemento pubblico di un array è la **dimensione**
  - variabile di esemplare pubblica **length** (attenzione, **non è** un metodo!)

```
double[] values = new double[10];
int a = values.length; // a vale 10
```

  - La variabile di esemplare **length** pubblica non viola l'incapsulamento **perché** è dichiarata final, quindi **non può** essere modificata, può soltanto essere **ispezionata**

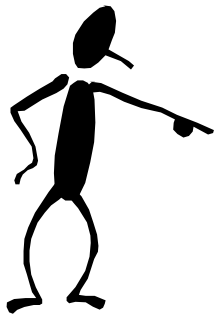
```
values.length = 15; // ERRORE IN COMPILAZIONE
```

- L'alternativa sarebbe stata fornire un metodo pubblico di accesso per accedere alla variabile privata
  - la soluzione scelta fornisce lo stesso livello di protezione dell'informazione ed è più veloce in esecuzione

Soluzione del problema iniziale

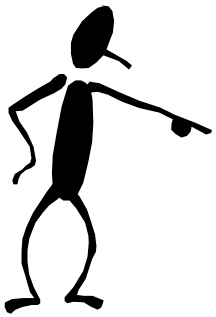
```
public class SelectValue
{  public static void main(String[] args)
   {  Scanner in = new Scanner(System.in);
      double[] values = new double[10];
      for (int i = 0; i < values.length; i++)
         values[i] = in.nextDouble();
      System.out.println("Inserisci un numero:");
      int index = in.nextInt();
      if (index < 0 || index >= values.length)
         System.out.println("Valore errato");
      else
         System.out.println(values[index]);
   }
}
```

Costruzione di un array



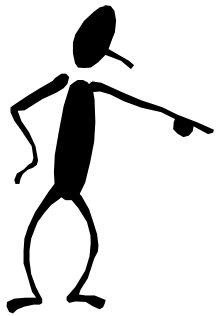
- Sintassi: `new NomeTipo[lunghezza]`
- Scopo: costruire un array per contenere dati del tipo ***NomeTipo***; la ***lunghezza*** indica il numero di dati che saranno contenuti nell'array
- Nota: ***NomeTipo*** può essere uno dei tipi fondamentali di Java o il nome di una classe
- Nota: i singoli elementi dell'array vengono inizializzati con le stesse regole delle variabili di esemplare
  - **0** (zero) per variabili numeriche e caratteri
  - **false** per variabili booleane
  - **null** per variabili oggetto

Riferimento a un array



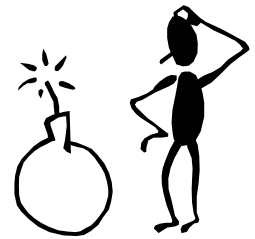
- Sintassi: `NomeTipo[] nomeRiferimento;`
- Scopo: definire la variabile ***nomeRiferimento*** come variabile oggetto che potrà contenere un riferimento a un array di dati di tipo ***NomeTipo***
- Le parentesi quadre **[ ]** sono necessarie e ***non*** devono contenere l'indicazione della dimensione dell'array
  - Sintassi alternativa: `NomeTipo nomeRiferimento[];`
  - **Sconsigliata**: la prima è preferibile perché tutta la dichiarazione del tipo è collocata in un unico punto

Accesso a un elemento di un array



- Sintassi: `riferimentoArray[indice]`
- Scopo: accedere all'elemento in posizione ***indice*** all'interno dell'array a cui ***riferimentoArray*** si riferisce, per conoscerne il valore o modificarlo
- Nota: il primo elemento dell'array ha indice 0, l'ultimo elemento ha indice (***dimensione*** - 1)
- Nota: se l'***indice*** non rispetta i vincoli, viene lanciata l'eccezione **ArrayIndexOutOfBoundsException**

Errori di limiti negli array

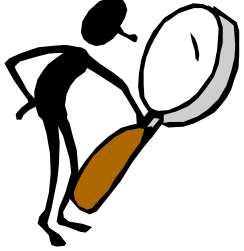


- Uno degli errori più comuni con gli array è l'utilizzo di un ***indice che non rispetta i vincoli***
  - il caso più comune è l'uso di un indice uguale alla dimensione dell'array, che è il primo indice non valido…

```
double[] values = new double[10];
values[10] = 2; // ERRORE IN ESECUZIONE
```

- Come abbiamo visto, l'ambiente runtime (cioè l'interprete Java) segnala questo errore con un'eccezione che arresta il programma

Inizializzazione di un array



- Quando si assegnano i valori agli elementi di un array si può procedere così

```
int[] primes = new int[3];
primes[0] = 2;
primes[1] = 3;
primes[2] = 5;
```

- ma se si conoscono tutti gli elementi da inserire si può usare questa sintassi (**migliore**)

```
int[] primes = { 2, 3, 5};
```

- oppure (accettabile, ma **meno chiara**)

```
int[] primes = new int[] { 2, 3, 5};
```

Passare un array come parametro

- Spesso si scrivono metodi che ricevono array come parametri espliciti

```
public static double sum(double[] values)
{   if (values == null)
       throw new IllegalArgumentException();
    if (values.length == 0)
       return 0;
    double sum = 0;
    for (int i = 0; i < values.length; i++)
       sum = sum + values[i];
    return sum;
}
```

Usare array come valori di ritorno

- Un metodo può anche usare un array come **valore di ritorno**
  - Questo metodo restituisce un array contenente i dati dell'array **oldArray** e con lunghezza **newLength**

```
public static int[] resize(int[] oldArray, int newLength)
{  if (newLength < 0 || oldArray == null)
      throw new IllegalArgumentException();
   int[] newArray = new int[newLength];
   int n = oldArray.length;
   if (newLength < n)
      n = newLength;
   for (int i = 0; i < n; i++)
      newArray[i] = oldArray[i];
   return newArray;
}
```

```
Oppure:
int n = Math.min(
        oldArray.length,newLength);
```

```
int[] values = {1, 7, 4};
values = resize(values, 5);
values[4] = 9;
```

È tutto chiaro? …

1. Quali valori sono presenti nell'array dopo l'esecuzione delle istruzioni seguenti?

```
double[] data = new double[10];
for (int i = 0; i < data.length; i++)
   data[i] = i * i;
```

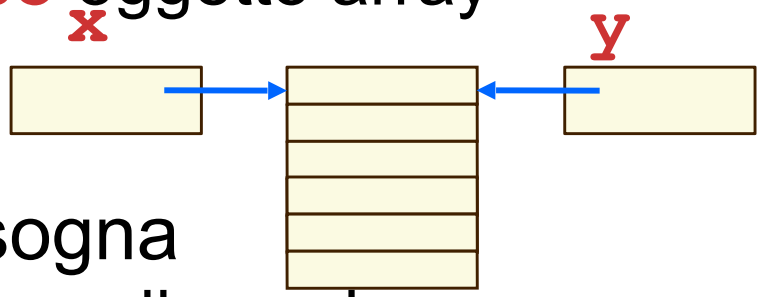
1. I seguenti enunciati sono corretti? Se sì, cosa visualizzano?
   a)
```
double[] a = new double[10];
System.out.println(a[0]);
```
   a)
```
double[] b = new double[10];
System.out.println(b[10]);
```
   a)
```
double[] c;
System.out.println(c[0]);
```

Copiare array

Copiare un array

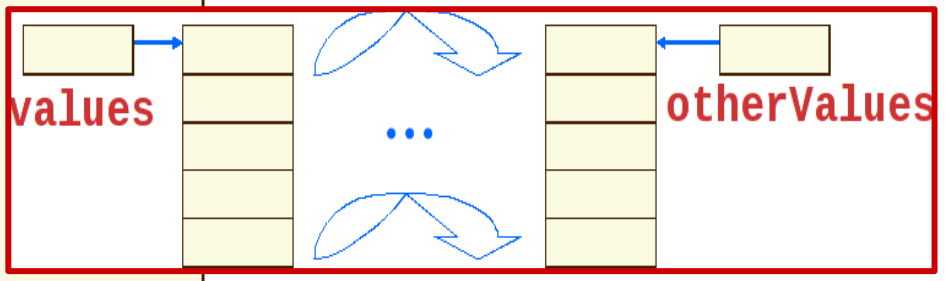
- Una variabile che si riferisce a un array è una variabile oggetto, quindi contiene un **riferimento** all'oggetto array
  - copiando il contenuto della variabile in un'altra **non si copia** l'array, si copia il riferimento allo **stesso** oggetto array

```
double[] x = new double[6];
double[] y = x;
```



- Per ottenere una **copia** dell'array, bisogna
  - creare un **nuovo array** degli stessi tipo e dimensione
  - copiare **ogni elemento** del primo array nel corrispondente elemento del secondo array

```
double[] values = new double[10];
... // inseriamo i dati nell'array values

double[] otherValues;
otherValues = new double[values.length];
for (int i = 0; i < values.length; i++)
   otherValues[i] = values[i];
```



Copiare un array

- Invece di usare un ciclo, è possibile (e ***più efficiente***) invocare il metodo statico **arraycopy** della classe **System** (nel pacchetto **java.lang**)

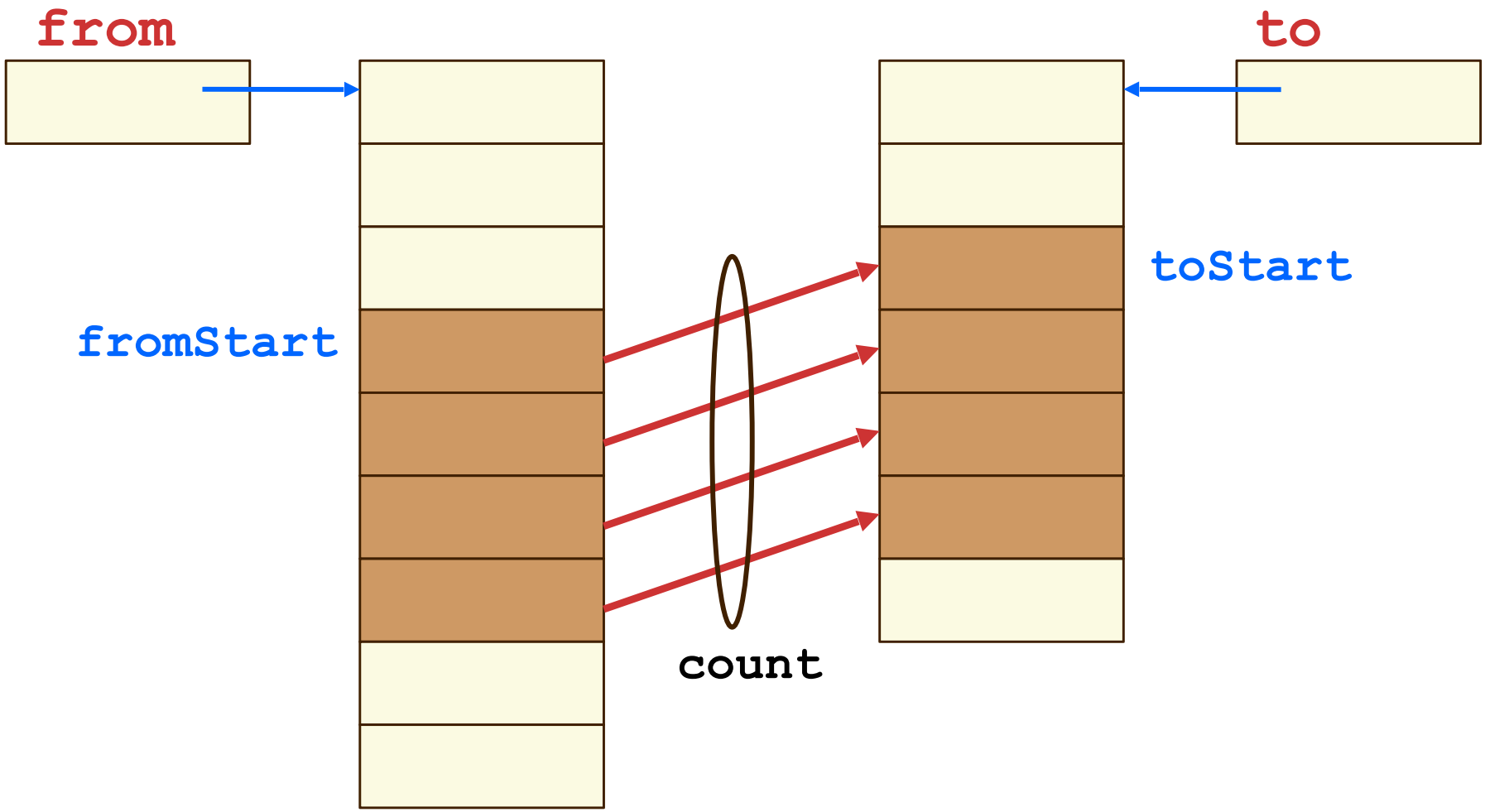
Attenzione
alla minuscola!

```
double[] values = new double[10];
... // inseriamo i dati nell'array

double[] otherValues = new double[values.length];
System.arraycopy(values, 0, otherValues, 0,
                 values.length);
```

- Il metodo **System.arraycopy** consente di copiare una porzione di un array in un altro array (grande almeno quanto la porzione che si vuol copiare)

System.arraycopy

```
System.arraycopy(from,fromStart,to,toStart,count);
```



Copiare un array

- È anche possibile usare il metodo **clone**

```
double[] otherValues = (double[]) values.clone();
```

- **Attenzione**: il metodo **clone** restituisce un riferimento di tipo **Object**
  - È necessario effettuare un **cast** per ottenere un riferimento del tipo desiderato
    - In questo caso **double[]**

Array riempiti solo in parte

Array riempiti solo in parte

- Riprendiamo un problema già visto, rendendolo un po' più complesso
- Scrivere un programma che
  - legge da standard input una sequenza di numeri int, uno per riga, **finché i dati non sono finiti**
  - chiede all'utente un numero intero **index** e visualizza il numero che nella sequenza occupava la posizione indicata da **index**
- La differenza rispetto al caso precedente è che ora **non sappiamo quanti saranno i dati** introdotti dall'utente

Array riempiti solo in parte

- **Problema**: se creiamo un array che contenga i numeri double, è necessario **indicarne la dimensione**, che è una sua proprietà **final**
  - gli array in Java **non possono crescere**!
- **Soluzione**: costruire un array di dimensioni **sufficientemente grandi** da poter accogliere una sequenza di dati di lunghezza "**ragionevole**", cioè tipica per il problema in esame
- **Nuovo Problema**: al termine dell'inserimento dei dati, in generale non tutto l'array conterrà dati validi
  - è necessario **tenere traccia** di quale sia **l'ultimo indice** nell'array che contiene **dati validi**

Array riempiti solo in parte

```
import java.util.Scanner;

public class SelectValue2
{  public static void main(String[] args)
   {
      final int ARRAY_LENGTH = 1000;
      int[] values = new int[ARRAY_LENGTH];
      Scanner in = new Scanner(System.in);
      int valuesSize = 0;
      boolean done = false;
      while (!done)
      {  String s = in.next();
         if (s.equalsIgnoreCase("Q"))
            done = true;
         else
         { values[valuesSize] = Integer.parseInt(s);
           valuesSize++;
         } // valuesSize è l'indice del primo dato non valido
      }
      System.out.println("Inserisci un numero:");
      int index = Integer.parseInt(in.next());
      if (index < 0 || index >= valuesSize)
         System.out.println("Valore errato");
      else
         System.out.println(values[index]);
   }
}
```

Array riempiti solo in parte

- **values.length** è il numero di valori ***memorizzabili***, **valuesSize** è il numero di valori ***memorizzati***
- Questa soluzione ha però ancora una debolezza
  - Se la ***previsione*** del programmatore sul numero massimo di dati inseriti dall'utente è sbagliata, il programma si arresta con un'eccezione di tipo **ArrayIndexOutOfBoundsException**
- Ci sono due possibili soluzioni

1. **impedire l'inserimento** di troppi dati, segnalando l'errore all'utente
2. **ingrandire l'array** quando ce n'è bisogno

Array riempiti solo in parte

- **Soluzione 1**: impedire l'inserimento di troppi dati

```
import java.util.Scanner;
public class SelectValue3
{
    ...
        final int ARRAY_LENGTH = 5;
        int[] values = new int[ARRAY_LENGTH];
        Scanner in = new Scanner(System.in);
        int valuesSize = 0;
        boolean done = false;
        while(!done)
        {   String s = in.nextLine();
            if (s.equalsIgnoreCase("Q"))
                done = true;
            else if (valuesSize == values.length)
            {   System.out.println("Troppi dati");
                done = true;
            }
                 else
                 {   values[valuesSize] = Integer.parseInt(s);
                     valuesSize++;
                 }
        }
...
```

Array riempiti solo in parte

- **Soluzione 2**: cambiare dimensione all'array
  - si parla di ***array dinamico***
  - Ma è ***impossibile*** modificare la dimensione di un array...

```
import java.util.Scanner;
public class SelectValue4
{...
        final int ARRAY_LENGTH = 5;
        int[] values = new int[ARRAY_LENGTH];
        Scanner in = new Scanner(System.in);
        int valuesSize = 0;
        boolean done = false;
        while (!done)
        {   String s = in.nextLine();
            if (s.equalsIgnoreCase("Q"))
                done = true;
            else
            {   if (valuesSize == values.length)
                    values = resize(values, valuesSize*2);
                values[valuesSize] = Integer.parseInt(s);
                valuesSize++;
            }
        }
...
```

Il metodo statico resize

- Restituisce un array di lunghezza **newLength** e contenente i dati dell'array **oldArray**
  - Crea un **nuovo** array più grande di quello "pieno" e copia in esso il contenuto del vecchio array
  - Useremo questo metodo **molto spesso**!

```
public static int[] resize(int[] oldArray, int newLength)
{  if (newLength < 0 || oldArray == null)
      throw new IllegalArgumentException();
   int[] newArray = new int[newLength];
   int n = oldArray.length;
   if (newLength < n)
      n = newLength;
   for (int i = 0; i < n; i++)
      newArray[i] = oldArray[i];
   return newArray;
}
```

```
int[] values = {1, 3, 7};
values = resize(values, 5);
```

Semplici algoritmi su array

La classe ArrayAlgs

- Costruiremo una classe **ArrayAlgs**
  - Sarà una "**classe di utilità**" (come la classe **Math**) che contiene una collezione di **metodi statici** che realizzano algoritmi per l'elaborazione di array
    - Per ora trattiamo array di numeri **interi**
    - Più avanti tratteremo array di **oggetti generici**

```
public class ArrayAlgs
{ ...
   public static int[] resize(int[] oldArray, int newLength)
   {  ... }

   public static  ...
}
```

```
//in un'altra classe i metodi verranno invocati cosi`
v = ArrayAlgs.resize(v,2*v.length);
```

Generare array di numeri casuali

- La classe **Math** ha il metodo **random( )** per generare sequenze di numeri pseudo-casuali
  - Una invocazione del metodo restituisce un numero **reale** pseudo-casuale nell'intervallo **[0, 1)**

```
double x =Math.random();
```

  - Per ottenere numeri **interi** casuali nell'intervallo **[a, b]**...

```
int n = (int)(a + (1+b-a)*Math.random());
```

- Usando **random** scriviamo nella classe **ArrayAlgs** un metodo che genera array di numeri interi casuali

```
public static int[] randomIntArray(int length, int n)
{
  int[] a = new int[length];
  for (int i = 0; i < a.length; i++)
    // a[i] e` un num intero casuale tra 0 e n-1 inclusi
    a[i] = (int) (n * Math.random());
  return a;
}
```

Convertire array in stringhe

- Se cerchiamo di stampare un array sullo standard output non riusciamo a visualizzarne il contenuto ...

```
int[] a = {1,2,3};
System.out.println(a);
```

```
[I@10b62c9
```

- Scriviamo nella classe **ArrayAlgs** un metodo che **crea una stringa** contenente gli elementi di un array

```
public static String printArray(int[] v, int vSize)
{     String s = "[";
      for (int i = 0; i<vSize; i++)
          s = s + v[i] + " ";
      s = s + "\b]";
      return s;
}
```

```
int[] a = {1,2,3};
int aSize = a.length;
System.out.println(
        ArrayAlgs.printArray(a,aSize));
```

```
[1 2 3]
```

Eliminazione/inserimento di elementi in un array

Eliminare un elemento di un array

- **Primo algoritmo**: se **l'ordine** tra gli elementi dell'array **non** è importante (cioè se l'array realizza il concetto astratto di insieme), è sufficiente
  - **copiare l'ultimo elemento** dell'array nella posizione dell'elemento da eliminare
  - **ridimensionare** l'array (oppure usare la tecnica degli array riempiti soltanto in parte)

```
public static void remove(int[] v, int vSize, int index)
{
    v[index] = v[vSize - 1];
}
```

```
int[] a = {1,2,3,4,5};
int aSize = a.length;
ArrayAlgs.remove(a,aSize,1);
aSize--;
```

a diventa [1,5,3,4]

Eliminare un elemento di un array

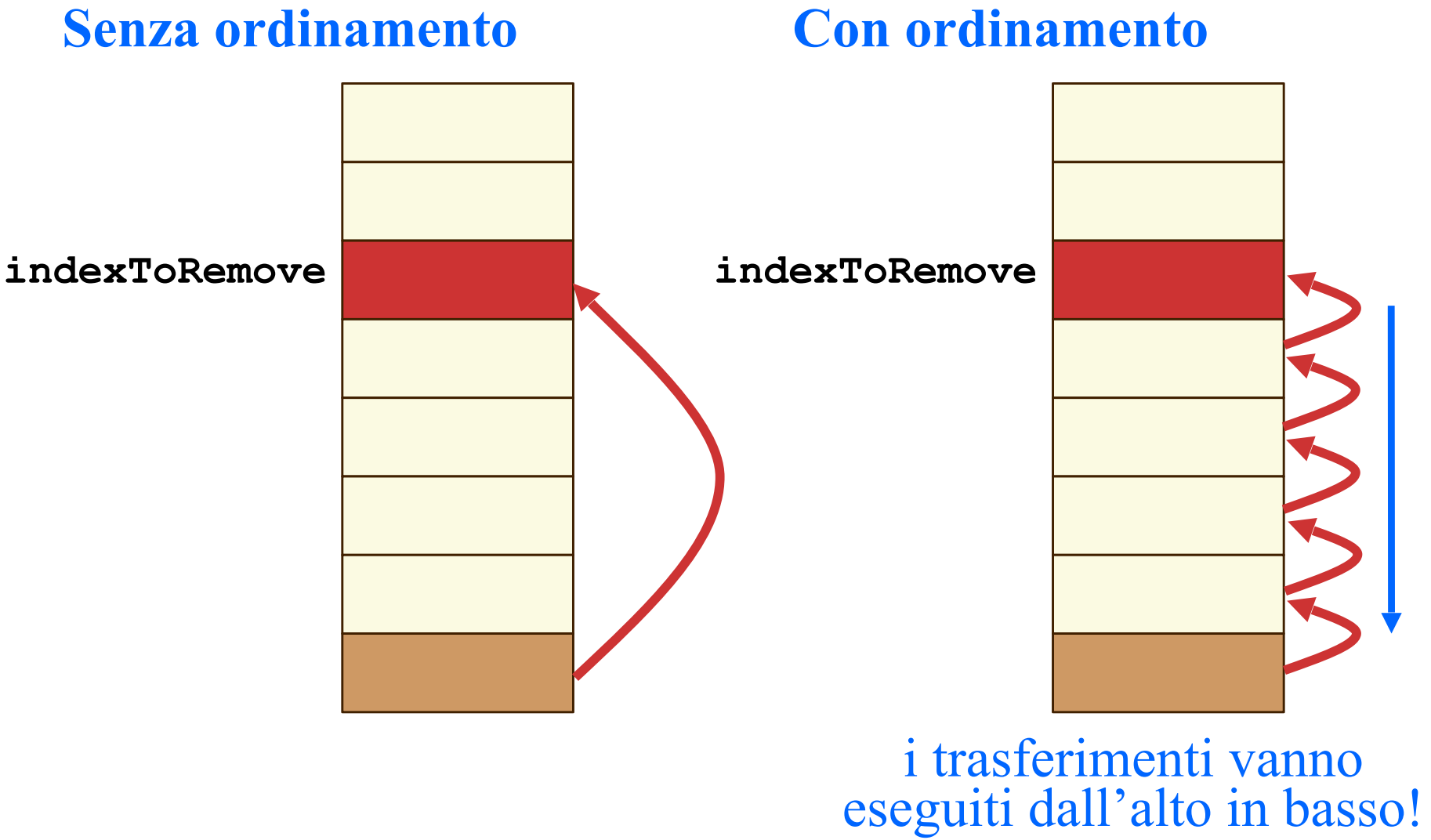
- ***Secondo algoritmo*** se **l'ordine** tra gli elementi dell'array deve essere mantenuto allora l'algoritmo è più complesso. Bisogna
  - **Spostare tutti gli elementi** dell'array successivi all'elemento da rimuovere nella posizione con indice immediatamente inferiore
  - **ridimensionare** l'array (oppure usare la tecnica degli array riempiti soltanto in parte)

```
public static void removeSorted(int[] v, int vSize, int index)
{     for (int i=index; i<vSize-1; i++)
         v[i] = v[i + 1];
}
```

```
int[] a = {1,2,3,4,5};
int aSize = a.length;
ArrayAlgs.removeSorted(a,aSize,1);
aSize--;
```

a diventa [1,3,4,5]

Eliminare un elemento di un array



Inserire un elemento in un array

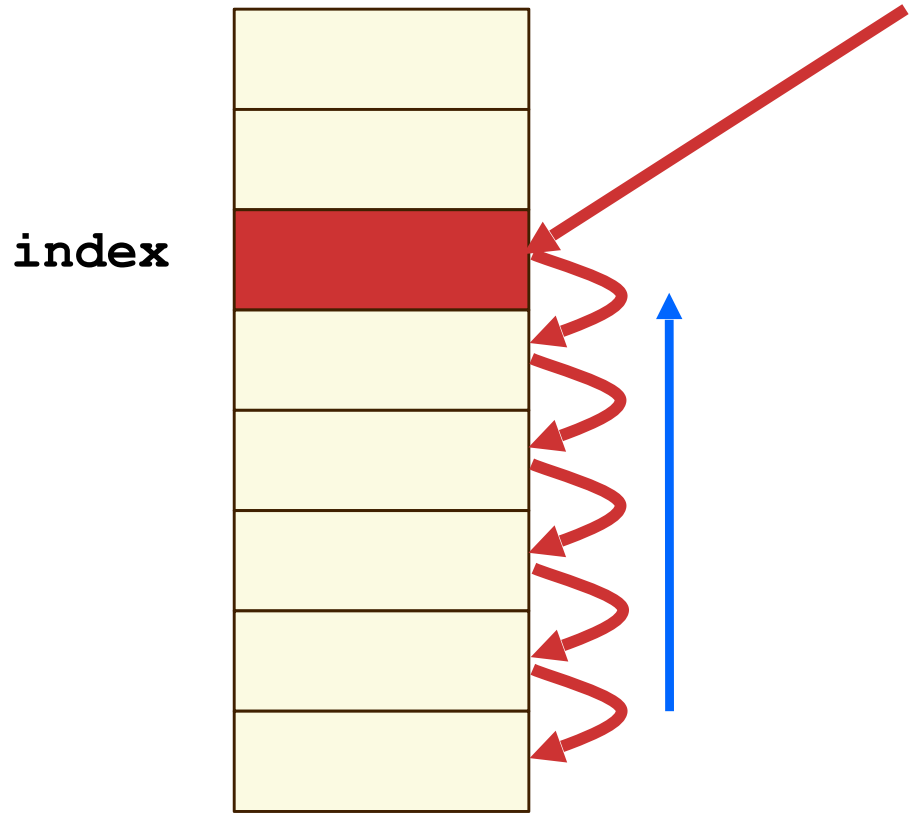
- **Algoritmo**: per inserire l'elemento nella posizione voluta, se non è la prima posizione libera, bisogna "**fargli spazio**". È necessario
  - **Ridimensionare** l'array (oppure usare la tecnica degli array riempiti soltanto in parte)
  - **Spostare tutti gli elementi** dell'array successivi alla posizione di inserimento nella posizione con indice immediatamente superiore (a partire dall'ultimo)

```
public static int[] insert(int[] v,int vSize, int index,int val)
{     if (vSize == v.length)  v = resize(v, 2*v.length);
      for (int i = vSize; i > index; i--)
          v[i] = v[i - 1];
      v[index] = val;
      return v;
}
```

```
int[] a = {1,2,3,4,5};
int aSize = a.length;
a = ArrayAlgs.insert(a,aSize,2,7);
aSize++;
```

a diventa [1,2,7,3,4,5]

Inserire un elemento in un array



i trasferimenti vanno
eseguiti dal basso in alto!

Trovare un valore in un array

- **Algoritmo**: la strategia più semplice è chiamata **ricerca lineare**. Bisogna
  - **scorrere** gli elementi dell'array **finché** l'elemento cercato non viene trovato oppure si raggiunge la fine dell'array
  - Nel caso in cui il valore cercato compaia più volte, questo algoritmo trova **soltanto** la prima occorrenza del valore e non le successive

```
public static int linearSearch(int[] v, int vSize, int value)
{
    for (int i = 0; i < vSize; i++)
        if (v[i] == value) return i; // trovato valore
    return -1;  // valore non trovato
}
```

```
int[] a = {1,2,3,4,5};
int aSize = a.length;
int i = ArrayAlgs.linearSearch(a,aSize,4);
```

i vale 3

Trovare il valore massimo

- **Algoritmo**: è necessario esaminare **l'intero array**. Bisogna
  - **Inizializzare** il valore candidato con il primo elemento
  - **Confrontare** il candidato con gli elementi rimanenti
  - **Aggiornare** il valore candidato se viene trovato un valore maggiore

```
public static int findMax(int[] v, int vSize)
{   int max = v[0];
    for (int i = 1; i < vSize; i++)
        if (v[i] > max)
            max = v[i];
    return max;
}
```

```
int[] a = {1,2,3,4,5};
int aSize = a.length;
int max = ArrayAlgs.findMax(a, aSize);
```

max vale 5

Trovare il valore minimo

- **Algoritmo** (del tutto analogo a quello per la ricerca del massimo): è necessario esaminare **l'intero array**. Bisogna
  - **Inizializzare** il valore candidato con il primo elemento
  - **Confrontare** il candidato con gli elementi rimanenti
  - **Aggiornare** il valore candidato se viene trovato un valore minore

```
public static int findMin(int[] v, int vSize)
{     int min = v[0];
      for (int i = 1; i < vSize; i++)
         if (v[i] < min)
              min = v[i];
      return min;
}
```

```
int[] a = {1,2,3,4,5};
int aSize = a.length;
int min = ArrayAlgs.findMin(a, aSize);
```

min vale 1

Array bidimensionali

Array bidimensionali

- Rivediamo un problema già esaminato
  - stampare una tabella con i valori delle potenze x^y, per ogni valore di **x** tra 1 e 4 e per ogni valore di **y** tra 1 e 5

```
1     1     1     1     1
2     4     8    16    32
3     9    27    81   243
4    16    64   256  1024
```

- e cerchiamo di risolverlo in modo più generale, scrivendo metodi che possano ***elaborare un'intera struttura di questo tipo***

Matrici

- Una struttura di questo tipo, con dati organizzati in righe e colonne, si dice ***matrice*** o array bidimensionale

1	1	1	1	1
2	4	8	16	32
3	9	27	81	243
4	16	64	256	1024

- Un elemento all'interno di una matrice è identificato da ***una coppia (ordinata) di indici***
  - un ***indice di riga***
  - un ***indice di colonna***
- In Java esistono gli **array bidimensionali**

Array bidimensionali in Java

- Dichiarazione di un array bidimensionale con elementi di tipo **int**

```
int[][] powers;
```

- Costruzione di array bidimensionale di **int** con **4** righe e **5** colonne

```
new int[4][5];
```

- Assegnazione di riferimento ad array bidimensionale

```
powers = new int[4][5];
```

- Accesso a un elemento di un array bidimensionale

```
powers[2][3] = 2;
```

Array bidimensionali in Java

- Ciascun indice deve essere
  - intero
  - maggiore o uguale a 0
  - minore della dimensione corrispondente
- Per conoscere il **valore delle due dimensioni**
  - il numero di **righe** è `powers.length;`
  - il numero di **colonne** è `powers[0].length;`
    (perché un array bidimensionale è in realtà **un array di array** e ogni array rappresenta una riga…)

```
import java.util.Scanner;

/**
    Programma che visualizza una tabella con i valori
    delle potenze "x alla y", con x e y che variano
    indipendentemente tra 1 ed un valore massimo
    assegnato dall'utente.
    I dati relativi a ciascun valore di x compaiono
    su una riga, con y crescente da sinistra
    a destra e x crescente dall'alto in basso.
*/
public class TableOfPowers
{  public static void main(String[] args)
   {  Scanner in = new Scanner(System.in);
      System.out.println(
         "Calcolo dei valori di x alla y");
      System.out.println("Valore massimo di x:");
      int maxX = in.nextInt();
      System.out.println("Valore massimo di y:");
      int maxY = in.nextInt();
      int maxValue =
            (int) Math.round(Math.pow(maxX, maxY));
      int columnWidth =
         1 + Integer.toString(maxValue).length();
      int[][] powers = generatePowers(maxX, maxY);
      printPowers(powers, columnWidth);
   }
                              //continua
```

```
                                                    //continua
/**
    Genera un array bidimensionale con i
    valori delle potenze di x alla y.
*/
private static int[][] generatePowers(int x,
                                      int y)
{  int[][] powers = new int[x][y];
   for (int i = 0; i < x; i++)
      for (int j = 0; j < y; j++)
         powers[i][j] =
            (int)Math.round(Math.pow(i + 1, j + 1));
   return powers;
}
                                                    //continua
```

- Notare l'utilizzo di metodi **private** per la scomposizione di un problema in sottoproblemi più semplici
  - in genere non serve preoccuparsi di pre-condizioni perché il metodo viene invocato da chi l'ha scritto

```
                                                    //continua
    /**
        Visualizza un array bidimensionale di
        numeri interi con colonne di larghezza
        fissa e valori allineati a destra.
    */
    private static void printPowers(int[][] v,
                                    int width)
    {  for (int i = 0; i < v.length; i++)
       {  for (int j = 0; j < v[i].length; j++)
          {  String s = Integer.toString(v[i][j]);
             while (s.length() < width)
                s = " " + s;
             System.out.print(s);
          }
          System.out.println();
       }
    }
}
```

Argomenti sulla riga dei comandi

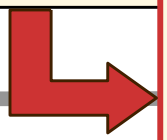
Argomenti sulla riga comandi

- Quando si esegue un programma Java, è possibile fornire dei parametri dopo il nome della classe che contiene il metodo **main**

```
java CommLineTester 2 33 Hello
```

- Tali parametri vengono letti dall'interprete Java e trasformati in un array di stringhe che **costituisce il parametro del metodo main**

```
public class CommLineTester
{ public static void main(String[] args)
  { System.out.println("Passati " +args.length+ " parametri");
        for (int i=0; i<args.length; i++)
            System.out.println(args[i]);
    }
}
```



```
Passati 3 parametri
2
33
Hello
```

Argomenti sulla riga di comandi

- Uso tipico degli argomenti sulla riga di comandi
  - Specificare **opzioni** e **nomi di file** da leggere/scrivere

```
java LineNumberer -c HelloWorld.java HelloWorld.txt
```

- Per convenzione le stringhe che iniziano con un **trattino** sono considerate opzioni

```
...
for (int i = 0; i< args.length; i++)
{ String a = args[i];
  if (a.startsWith("-")) // è un'opzione
  {
    if (a.equals("-c")) useCommentDelimiters = true;
  }
  else if (inputFileName == null) inputFileName = a;
  else if (outputFileName == null) outputFileName = a;
}
...
```

Esercizio: Array paralleli (cfr. Suggerimenti per la programmazione 7.2)

Array paralleli

- Scriviamo un programma che riceve in ingresso un elenco di dati che rappresentano
  - i **cognomi** di un insieme di studenti
  - il **voto** della prova scritta
  - il **voto** della prova orale
- I dati di uno studente vengono inseriti in una riga separati da uno spazio
  - prima il cognome, poi il voto scritto, poi il voto orale
- I dati sono terminati da una **riga vuota**

Array paralleli

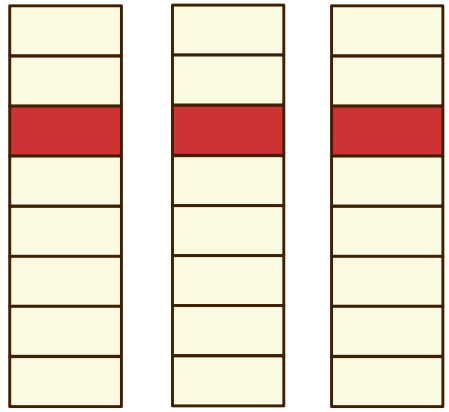
- Ora aggiungiamo le seguenti funzionalità
  - il programma chiede all'utente di inserire un comando per identificare l'elaborazione da svolgere
    - **Q** significa "**termina il programma**"
    - **S** significa "**visualizza la media dei voti di uno studente**"
  - Nel caso **S** il programma
    - chiede all'utente di **inserire il cognome** di uno studente
    - **Stampa il cognome** dello studente seguito dalla **media dei suoi voti**

```
import java.util.Scanner;

public class StudentManager
{  public static void main(String[] args)
   {  Scanner in = new Scanner(System.in);
      String[] names = new String[10];
      double[] wMarks = new double[10];
      double[] oMarks = new double[10];
      int count = 0; // array riempiti solo in parte
      boolean done = false;
      while (!done) //inserimento dati studenti
      {  String input = in.nextLine();
         if (input.length() == 0) done=true;
         else
         {
           Scanner t = new Scanner(input);
           if (count == names.length) //ridimensionamento array
           {  names = resizeString(names, count * 2);
              wMarks = resizeDouble(wMarks, count * 2);
              oMarks = resizeDouble(oMarks, count * 2);
           }
           names[count] = t.next();
           wMarks[count] = Double.parseDouble(t.next());
           oMarks[count] = Double.parseDouble(t.next());
           count++;
         }
      }
                                                          //continua
```

```
    done = false;                                          //continua
    while (!done)  //visualizzazione dati inseriti
    {
      System.out.println("Comando? (Q per uscire, S per vedere)");
      String command = in.nextLine();
      if (command.equalsIgnoreCase("Q"))
          done = true;
      else if (command.equalsIgnoreCase("S"))
      {
          System.out.println("Cognome?");
          String name = in.nextLine();
          printAverage(names, wMarks, oMarks,name, count); //NOTA:
          //non abbiamo gestito l'eccezione lanciata da printAverage
      }
      else
      {  System.out.println("Comando errato");
      }
    }
}

private static void printAverage(String[] names, double[] wMarks,
                                 double[] oMarks, String name, int count)
{  int i = findName(names, name, count);
   if (i == -1)  throw new IllegalArgumentException();
   else
   {  double avg = (wMarks[i] + oMarks[i]) / 2;
      System.out.println(name + " " + avg);
   }
}                                                          //continua
```

```
                                                                    //continua
   private static int findName(String[] names, String name, int count)
   {  for (int i = 0; i < count; i++)
      if (names[i].equals(name))
         return i;
      return -1;
   }

   private static String[] resizeString(String[] oldv, int newLength)
   {  if (newLength < 0 || oldv == null)
        throw new IllegalArgumentException();
      String[] newv = new String[newLength];
      int count = oldv.length;
      if (newLength < count) count = newLength;
      for (int i = 0; i < count; i++)
            newv[i] = oldv[i];
      return newv;
   }

   private static double[] resizeDouble(double[] oldv, int newLength)
   {  if (newLength < 0 || oldv == null)
        throw new IllegalArgumentException();
      double[] newv = new double[newLength];
      int count = oldv.length;
      if (newLength < count)  count = newLength;
      for (int i = 0; i < count; i++)
           newv[i] = oldv[i];
      return newv;
   }
}
```

Array paralleli



- L'esempio presentato usa una struttura dati denominata "**array paralleli**"
  - Si usano **diversi** array per contenere i dati del problema, ma questi sono tra loro **fortemente correlati**
  - In particolare, elementi aventi lo **stesso indice nei diversi array** sono tra loro correlati
    - In questo caso, rappresentano **diverse proprietà dello stesso studente**
  - I tre array devono sempre contenere lo **stesso numero** di elementi
  - Molte operazioni hanno bisogno di **usare tutti gli array**, che devono quindi essere passati come parametri
    - come nel caso di **printAverage**

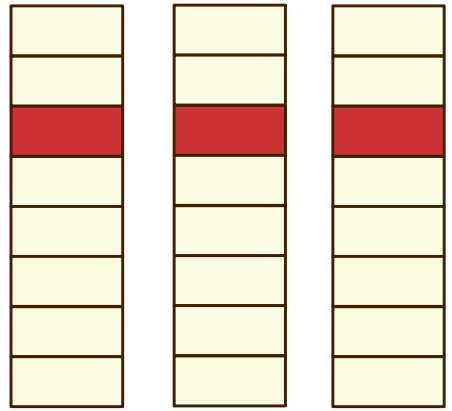
Array paralleli

- Array paralleli sono molto usate in linguaggi di programmazione **non OO**, ma presentano **numerosi svantaggi** che possono essere superati in Java
  - Le modifiche alle dimensioni di un array devono essere fatte contemporaneamente a tutti gli altri
  - I metodi che devono elaborare gli array devono avere una lunga lista di parametri espliciti
  - Non è semplice scrivere metodi che devono ritornare informazioni che comprendono **tutti** gli array
    - Per esempio, nel caso presentato non è semplice scrivere un metodo che realizzi la fase di input dei dati, perché tale metodo dovrebbe avere come **valore di ritorno i tre array**!

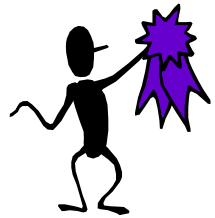
Array paralleli in OOP

- Le tecniche di ***OOP*** consentono di gestire molto più efficacemente le strutture dati di tipo “array paralleli”
  - Definire una classe che contenga tutte le informazioni relative ad “una fetta” degli array, cioè raccolga tutte le informazioni presenti nei diversi array in relazione ad un certo indice
  - Costruire un array di oggetti di questa classe

```
public class Student
{   private String name;
    private double wMark;
    private double oMark;

    ...
}
```



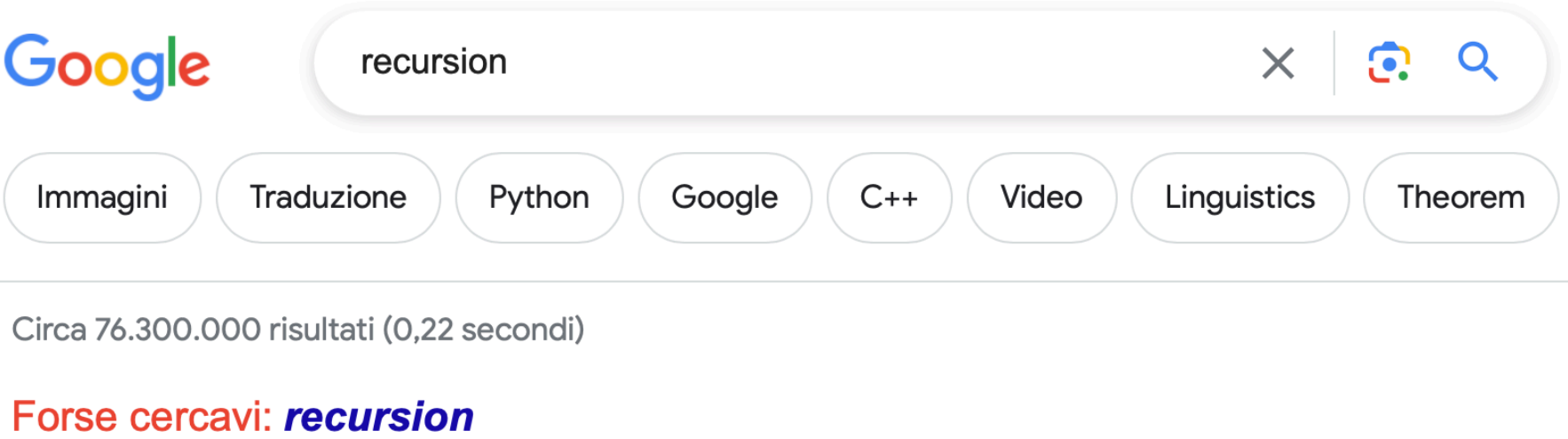
Non usare array paralleli



- Tutte le volte in cui il problema presenta una struttura dati del tipo “array paralleli”, si consiglia di **trasformarla in un array di oggetti**
  - occorre realizzare la classe con cui costruire gli oggetti
- È molto più facile scrivere il codice e, soprattutto, modificarlo
- Immaginiamo di introdurre un’altra caratteristica per gli studenti (per esempio il numero di matricola)
  - nel caso degli array paralleli è necessario ***modificare le firme di tutti i metodi***, per introdurre il nuovo array
  - nel caso dell’array di oggetti **Student**, basta modificare la classe **Student**

Department of Information Engineering

Ricorsione (capitolo 12)



Il calcolo del fattoriale

- La funzione **fattoriale**, molto usata nel calcolo combinatorio, è così definita

$$n! = \begin{cases} 1 \text{ se } n = 0 \\ n(n-1)! \text{ se } > 0 \end{cases}$$

- dove **n** è un numero intero non negativo

Il calcolo del fattoriale

- Vediamo di capire cosa significa…

  $0! = 1$

  $1! = 1(1-1)! = 1 \cdot 0! = 1 \cdot 1 = 1$

  $2! = 2(2-1)! = 2 \cdot 1! = 2 \cdot 1 = 2$

  $3! = 3(3-1)! = 3 \cdot 2! = 3 \cdot 2 \cdot 1 = 6$

  $4! = 4(4-1)! = 4 \cdot 3! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$

  $5! = 5(5-1)! = 5 \cdot 4! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$

- Quindi, per ogni **n** intero positivo, il fattoriale di **n** è il **prodotto dei primi n numeri interi positivi**

Il calcolo del fattoriale

- Scriviamo un metodo statico per calcolare il fattoriale

```
public static int factorial(int n)
{   if (n < 0)
       throw new IllegalArgumentException();

    if (n == 0)
       return 1;

    int p = 1;
    for (int i = 2; i <= n; i++)
       p = p * i;
    return p;
}
```

Il calcolo del fattoriale

- Fin qui, nulla di nuovo… però abbiamo dovuto fare un'analisi matematica della definizione per scrivere l'algoritmo
- Realizzando direttamente la definizione, sarebbe stato più naturale scrivere

```
public static int factorial(int n)
{   if (n < 0)
       throw new IllegalArgumentException();
    if (n == 0)
       return 1;
    return n * factorial(n - 1);
}
```

Il calcolo del fattoriale

```
public static int factorial(int n)
{   if (n < 0)
       throw new IllegalArgumentException();
    if (n == 0)
       return 1;
    return n * factorial(n - 1);
}
```

- Si potrebbe pensare: “Non è possibile **invocare un metodo** mentre si esegue **il metodo stesso**!”
- Invece, come è facile verificare scrivendo un programma che usi **factorial**, questo è lecito in Java
  - così come è lecito in quasi tutti i linguaggi di programmazione

Invocazione di un metodo ricorsivo

Invocare un metodo ricorsivo

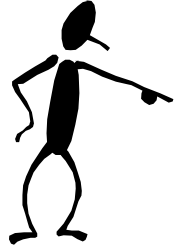
- **Invocare un metodo** mentre si esegue **lo stesso metodo** è un paradigma di programmazione che si chiama **ricorsione**
  - Un metodo che ne faccia uso è un **metodo ricorsivo**
  - La ricorsione è uno strumento **potente** per realizzare alcuni algoritmi, ma è anche **fonte di molti errori** di difficile diagnosi
- Come **funziona** una invocazione ricorsiva di un metodo?
  - In una invocazione ricorsiva, la JVM esegue le stesse azioni eseguite nell'invocazione di un metodo qualsiasi:
    - **sospende** l'esecuzione del metodo invocante
    - **esegue il metodo** invocato fino alla sua terminazione
    - **riprende l'esecuzione** del metodo invocante dal punto in cui era stata sospesa

Invocare un metodo ricorsivo

- Invochiamo **factorial(3)** per calcolare 3!
  - **factorial(3)** invoca **factorial(2)**
    - **factorial(2)** invoca **factorial(1)**
      - **factorial(1)** invoca **factorial(0)**
        - **factorial(0) restituisce 1**
      - **factorial(1)** restituisce 1
    - **factorial(2)** restituisce 2
  - **factorial(3)** restituisce 6
- Si crea quindi una **pila di metodi "in attesa"**, che si allunga e che poi si accorcia fino a estinguersi

```
public static int factorial(int n)
{   if (n < 0) throw new IllegalArgumentException();
    if (n == 0) return 1;
    return n * factorial(n - 1); }
```

Algoritmi ricorsivi



- **Ricorsione**: tecnica di programmazione che sfrutta l'idea di suddividere un problema da risolvere in sottoproblemi simili a quello originale, ma più semplici.

- Un algoritmo ricorsivo per la risoluzione di un dato problema deve essere definito nel modo seguente:
  - prima si definisce come risolvere direttamente dei problemi analoghi a quello di partenza, ma di dimensione **"sufficientemente piccola"** (detti **casi base**);
  - poi (**passo ricorsivo**) si definisce come ottenere la soluzione del problema di partenza combinando la soluzione di uno o più sottoproblemi di **"dimensione inferiore"**.

La ricorsione: caso base

- **Prima regola**
  - Un algoritmo ricorsivo deve fornire la soluzione del problema **in almeno un caso particolare**, senza ricorrere a una chiamata ricorsiva
  - tale caso si chiama **caso base** della ricorsione
- Nel nostro esempio, il caso base è

```
if (n == 0)
    return 1;
```

- A volte ci sono **più casi base**, non è necessario che sia unico

La ricorsione: passo ricorsivo

- **Seconda regola**
  - Un algoritmo ricorsivo deve effettuare la chiamata ricorsiva **dopo aver semplificato** il problema

- Nel nostro esempio, per il calcolo del fattoriale di **n** si invoca la funzione ricorsivamente per conoscere il fattoriale di **n-1**, cioè per risolvere un problema più semplice

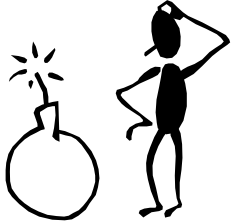
```
if (n > 0)
    return n * factorial(n - 1);
```

- Il concetto di "problema più semplice" varia di volta in volta: in generale, bisogna **avvicinarsi a un caso base**

La ricorsione: un algoritmo?

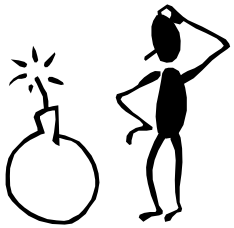
- Queste due regole sono fondamentali per dimostrare che la soluzione ricorsiva di un problema sia un ***algoritmo***
  - in particolare, che termini in un numero **finito** di passi
- Si potrebbe pensare che le chiamate ricorsive si susseguano una dopo l'altra, **all'infinito**. Invece se:
  - Ad ogni invocazione il problema diventa sempre **più semplice** e si avvicina al caso base
  - E la soluzione del caso base **non** richiede ricorsione
- allora certamente la soluzione viene calcolata in un **numero finito** di passi, per quanto complesso possa essere il problema

Ricorsione infinita



- Quanto detto ci suggerisce che non tutti i metodi ricorsivi realizzano algoritmi
  - ***se manca il caso base***, il metodo ricorsivo continua ad invocare se stesso all'infinito
  - ***se il problema non viene semplificato a ogni invocazione ricorsiva***, il metodo ricorsivo continua ad invocare se stesso all'infinito
- Dato che la lista dei metodi "in attesa" si allunga indefinitamente
  - l'ambiente runtime **esaurisce la memoria** disponibile per tenere traccia di questa lista
  - E il programma **termina con un errore**

Ricorsione infinita



- Un programma che presenta ricorsione infinita:

```
public class InfiniteRecursion
{  public static void main(String[] args)
   {  main(args);
   }
}
```

- Il programma terminerà con la segnalazione dell'eccezione **StackOverflowError**
  - Ricordiamo che il **runtime stack** ("pila") è la struttura di memoria all'interno dell'interprete Java che gestisce le invocazioni di metodi in attesa

Eliminare la ricorsione

La ricorsione in coda

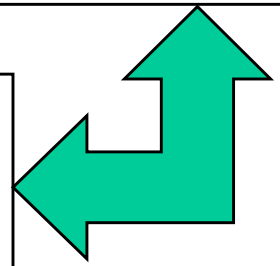
- Esistono diversi tipi di ricorsione
- Il modo visto fino a ora si chiama **ricorsione in coda** (***tail recursion***)
  - il metodo ricorsivo esegue ***una sola invocazione ricorsiva*** e tale invocazione è l'**ultima azione** del metodo

```
public void tail(...)
{   ...
    tail(...);
}
```

Eliminare la ricorsione in coda

- La ricorsione in coda può sempre essere agevolmente ***eliminata***, trasformando il metodo ricorsivo in un metodo che usa un ***ciclo***

```
public int factorial(int n)
{   if (n == 0) return 1;
    return n * factorial(n - 1);
}
```



```
public int factorial(int n)
{   int f = 1;
    while (n > 0)
    {   f = f * n;
        n--;
    }
    return f;
}
```

Eliminare la ricorsione in coda

- Allora, a cosa serve la ricorsione in coda?
- Non è **necessaria**, però in alcuni casi rende il codice più leggibile
- È **utile** quando la soluzione del problema è esplicitamente ricorsiva (per esempio nel calcolo della funzione fattoriale)
- In ogni caso, la ricorsione in coda è ***meno efficiente*** del ciclo equivalente, perché il sistema deve gestire le invocazioni sospese

Eliminare la ricorsione

- Se la ricorsione non è in coda, non è facile eliminarla (cioè scrivere codice non ricorsivo equivalente), però si può dimostrare che **ciò è sempre possibile**
  - **deve** essere così, perché il processore esegue istruzioni in sequenza e non può tenere istruzioni in attesa
  - In Java l'interprete si fa carico di eliminare la ricorsione (usando il **runtime stack**)
  - In un linguaggio compilato il compilatore trasforma il codice ricorsivo in codice macchina non ricorsivo

Ricorsione multipla e problemi di efficienza

La ricorsione multipla

- Si parla di **ricorsione multipla** quando un metodo invoca se stesso **più volte** durante la propria esecuzione
  - la ricorsione multipla è ancora più difficile da eliminare, ma è sempre possibile
- **Esempio**: il calcolo dei **numeri di Fibonacci**

$$F_1 = 1,$$
$$F_2 = 1,$$
$$F_n = F_{n-1} + F_{n-2} \text{ (per ogni n>2)}$$

La classe FibTester

```
public class FibTester
{   private static int invcount = 0;    // variabile statica
    public static void main(String[] args)
    {   int n = 0;
        if (args.length != 1)
        {   System.out.println("Uso: $java FibTester <numero>");
            System.exit(1); }
        try
        {   n = Integer.parseInt(args[0]); }
        catch(NumberFormatException e)
        {   System.out.println("Inserire un intero!");
            System.exit(1); }
        System.out.println("*** Collaudo iterativeFib ***");
        for (int i = 1; i <= n; i++)
        {   long f = iterativeFib(i);
            System.out.println("iterativeFib(" +i+ ") = " + f);}
        System.out.println("\n*** Collaudo recursiveFib ***");
        for (int i = 1; i <= n; i++)
        {  invcount = 0;
           long f = recursiveFib(i);
           System.out.println("recursiveFib(" +i+ ") = " + f);
           System.out.println(invcount+" invocazioni"); }
    }                                          //continua
```

La classe FibTester

```
    public static long recursiveFib(int n)          //continua
    {   if (n < 1) throw new IllegalArgumentException();
        System.out.println("Inizio recursiveFib(" + n +")");
        invcount++;
        long f;
        if (n < 3) f = 1;
        else f = recursiveFib(n-1) + recursiveFib(n-2);
        System.out.println("Uscita recursiveFib(" + n + ")");
        return f;
    }
    public static long iterativeFib(int n)
    {   if (n < 1) throw new IllegalArgumentException();
        long f = 1;
        if (n >= 3)
        {  long fib1 = 1;
           long fib2 = 1;
           for (int i = 3; i<=n; i++)
           {   f = fib1 + fib2;
               fib2 = fib1;
               fib1 = f;   }
        }
        return f;
    }
}
```

La ricorsione multipla

- Il metodo **fib** realizza una **ricorsione multipla**
- La ricorsione multipla va usata con molta attenzione, perché può portare a programmi **molto inefficienti**
- Eseguendo il calcolo dei numeri di Fibonacci di ordine crescente
  - Si nota che il tempo di elaborazione cresce **molto rapidamente**
  - Sono necessarie quasi **3 milioni di invocazioni** per calcolare **Fib(31)** !!!!
  - più avanti quantificheremo questo problema
- Invece una **soluzione iterativa** risulta molto più efficiente

Albero delle invocazioni di fib

- Visualizziamo lo schema delle invocazioni di **fib** in una struttura ad **albero**
  - **Albero di ricorsione** dell'algoritmo ricorsivo
- Lo stesso valore viene calcolato **più volte**
  - **Molto** inefficiente