

I cicli: esempi

Prof. Luca Trevisan

Corso di Fondamenti di Informatica
A.A. 2025/2026





DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Cicli annidati

Problema

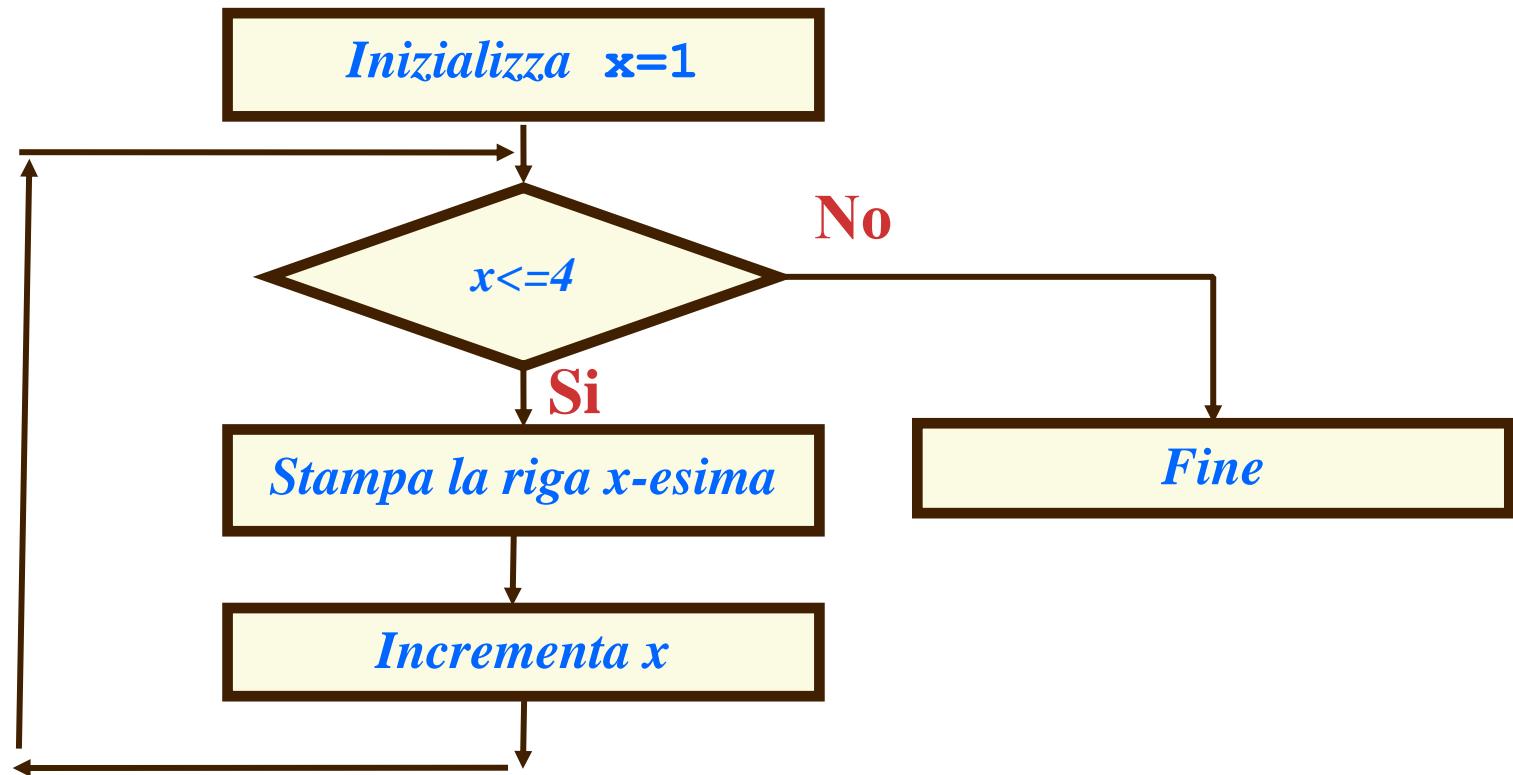
- Vogliamo stampare una tabella con i valori delle potenze x^y , per ogni valore di x tra 1 e 4 e per ogni valore di y tra 1 e 5

		y: 1 to 5				
		x →				
		1 ¹	1 ²	1 ³	1 ⁴	1 ⁵
1	2	2 ¹	2 ²	2 ³	2 ⁴	2 ⁵
3	4	3 ¹	3 ²	3 ³	3 ⁴	3 ⁵
4	5	4 ¹	4 ²	4 ³	4 ⁴	4 ⁵

- Scompongo il problema in due sottoproblemi
 - Stampare **4 righe**
 - Stampare il contenuto di una riga

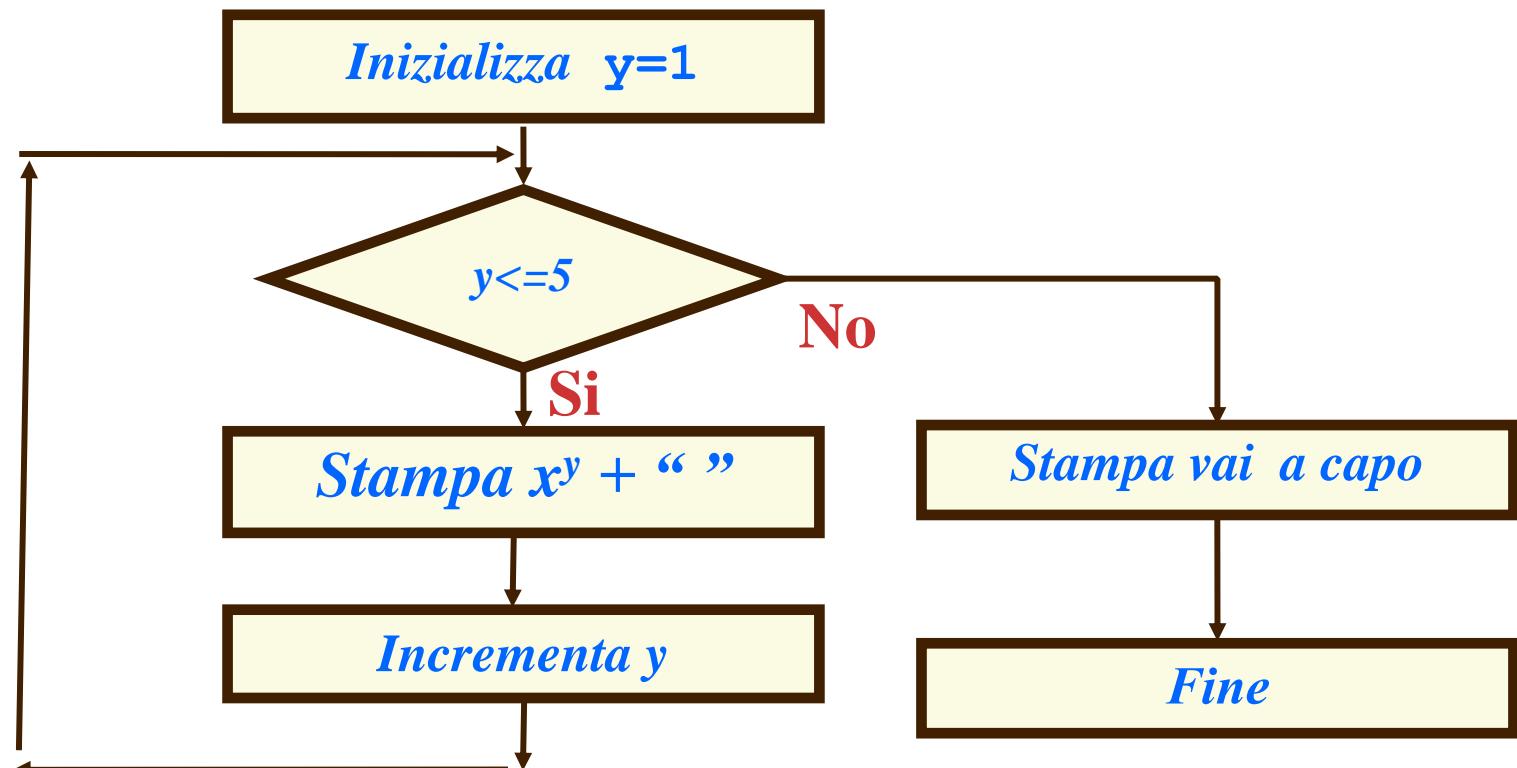
Stampare 4 righe: codice

```
for (int x = 1; x <= 4; x++)  
{    // stampa la riga x-esima della tabella  
    ...  
}
```



Stampare la riga x-esima: diagramma

- Per **stampare la riga x-esima**, bisogna calcolare e stampare i valori x^1, x^2, x^3, x^4 e x^5 , cosa che si puo' fare facilmente con un ciclo **for**



Stampare la riga x-esima: codice

- Per stampare la riga x-esima, bisogna calcolare e stampare i valori x^1, x^2, x^3, x^4 e x^5 , cosa che si può fare facilmente con un ciclo **for**

```
// stampa la riga x-esima della tabella
for (int y = 1; y <= 5; y++)
{
    int p = (int) (Math.pow(x, y));
    System.out.print(p+ ' ');
}
System.out.println(); // va a capo
```

- Ogni iterazione del ciclo stampa un valore, seguito da uno spazio bianco per separare valori successivi; al termine si va a capo



Soluzione

```
for (int x = 1; x <= 4; x++)
{
    // stampa la riga x-esima della tabella
    for (int y = 1; y <= 5; y++)
    {
        // stampa il valore y-esimo della riga x-esima
        int p = (int)(Math.pow(x,y));
        System.out.print(p+ ' ');
    }
    System.out.println(); // va a capo
}
```

- Problema: in output le colonne non sono allineate

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



Soluzione

- Incolonnare dati di lunghezza variabile è un problema frequente

```
final int COLUMN_WIDTH = 5;
for (int x = 1; x <= 4; x++)
{
    // stampa la riga x-esima della tabella
    for (int y = 1; y <= 5; y++)
    {
        // converte il valore in stringa
        String p = '' + (int)(Math.pow(x,y));
        // aggiunge gli spazi necessari
        while(p.length() < COLUMN_WIDTH)
        { p = ' ' + p;
        }
        System.out.print(p + ' ');
    }
    System.out.println();
}
```

Ora ci sono
tre cicli
Annidati!

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



Soluzione più elegante

- Incolonnare dati di lunghezza variabile è un problema frequente

```
final int COLUMN_WIDTH = 5;
for (int x = 1; x <= 4; x++)
{
    // stampa la riga x-esima della tabella
    for (int y = 1; y <= 5; y++)
    {
        // calcolo la potenza
        int p = (int) (Math.pow(x, y));
        System.out.printf("%" + COLUMN_WIDTH + "d ", p);
    }
    System.out.println();
}
```

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

Extra: le immagini





DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Zoom in





DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

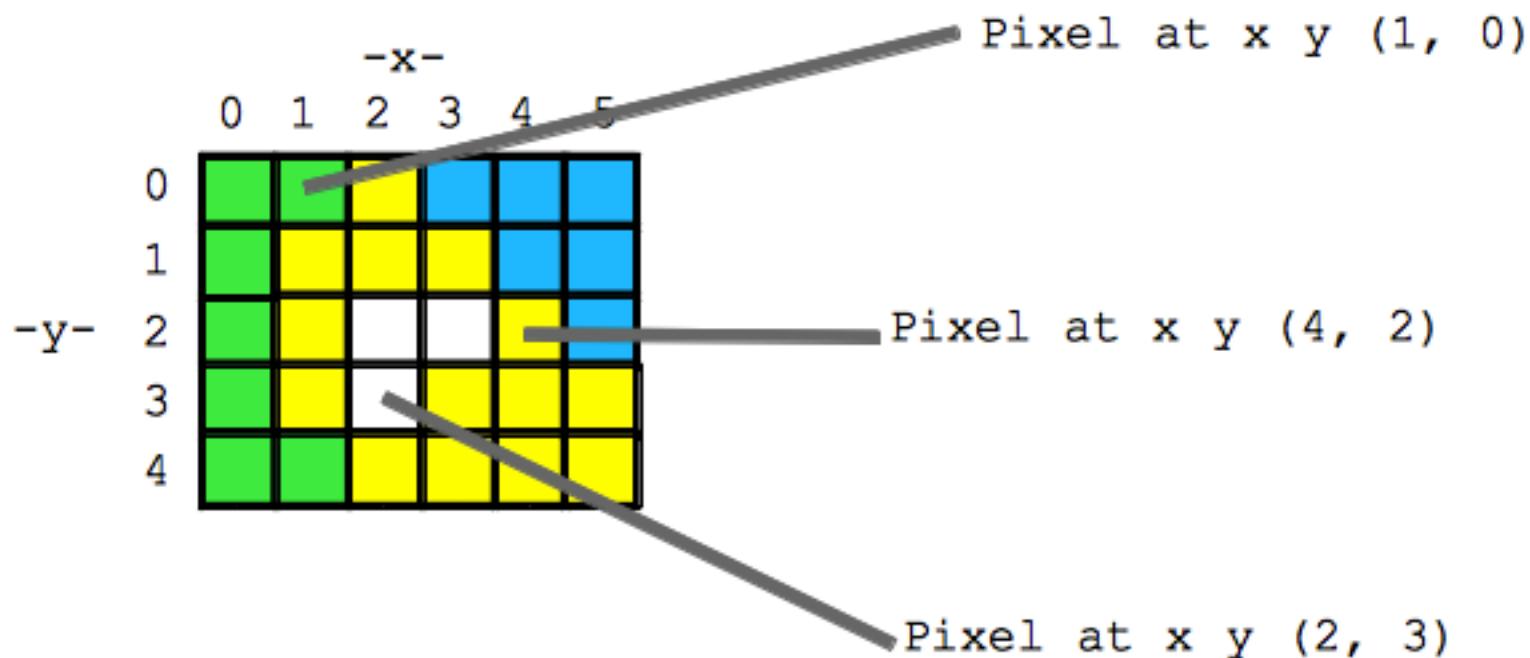
Zoom in





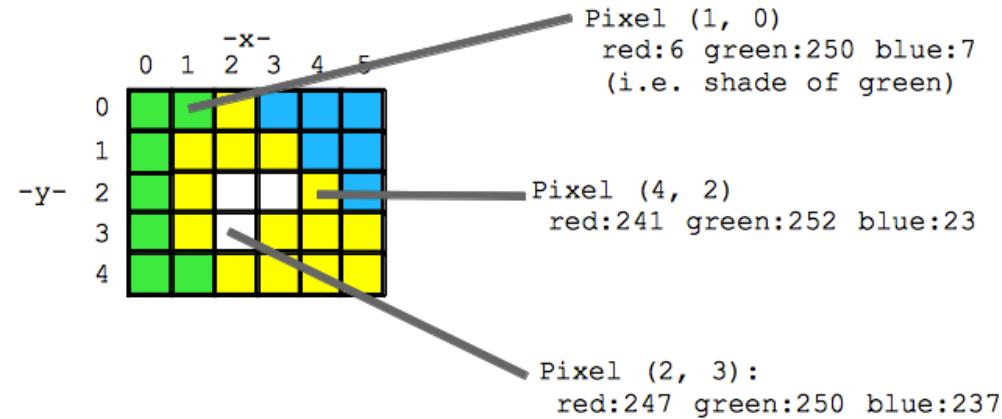
Come sono fatte le immagini

- ❑ Piccoli quadretti chiamati “pixel”
- ❑ Ogni pixel
 - Ha un colore associato
 - Ha una coordinata associata in un piano x/y
 - E’ molto piccolo (percezione di continuita’ per l’osservatore)
- ❑ Immagine: griglia di pixel, es 800x600
 - 800 pixel di larghezza, 600 di altezza
 - In tutto 480000 pixel (circa 0.5megapixel)
 - Nel vostro telefono decine di megapixel...



Come si rappresenta il colore?

- Schema RGB (Red Green Blue)
 - Ogni colore è definito da una particolare combinazione di pura luce rossa, verde e blu.
 - Ogni colore può assumere valore da 0 a 255
 - 0=nessuna luce, 255 massima luce
 - https://www.rapidtables.com/web/color/RGB_Color.html





DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Il problema del “ciclo e mezzo”



Il problema del “ciclo e mezzo”

- ❑ Un ciclo del tipo
 - “**fai** qualcosa, **verifica** una condizione, **fai** qualcos’altro e **ripeti** il ciclo se la condizione era vera”
- ❑ Non ha una struttura di controllo predefinita in Java e deve essere realizzata con un “trucco”, come quello di usare una variabile booleana, detta **variabile di controllo** del ciclo (la sentinella già vista!)
- ❑ Una struttura di questo tipo si chiama anche “**ciclo e mezzo**” o ciclo ridondante (perché c’è qualcosa di “aggiunto”, di innaturale...)

Il problema del “ciclo e mezzo”

- Situazione tipica: l’utente deve inserire un insieme di valori, la cui dimensione non è predefinita
 - Si realizza un ciclo while, dal quale si esce soltanto quando si verifica la condizione all’interno del ciclo

La lettera “Q” (Quit) è un **valore sentinella** che segnala che l’immissione dei dati è terminata

```
boolean done = false;
while (!done)
{ System.out.println("Valore?");
  String input = in.next();
  if (input.equalsIgnoreCase("Q"))
    done = true;
}
else{
  ... // elabora line
}
```



Esempio: il programma QEater

```
import java.util.Scanner;

public class QEater
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        boolean done = false; // variabile di controllo
        while(!done)
        {
            System.out.println("Voglio una Q");
            String input = in.nextLine();
            if (input.equals("Q"))
            {
                System.out.println("Grazie!");
                done = true;
            }
            else
            {
                System.out.println("Allora non hai capito... ");
            }
        }
        in.close();
    }
}
```



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Ciclo do



Ciclo *do*

- ❑ Capita spesso di dover **eseguire il corpo di un ciclo almeno una volta**, per poi ripeterne l'esecuzione se è verificata una particolare condizione

- ❑ Esempio tipico: leggere un valore in ingresso, ed eventualmente rileggerlo finché non viene introdotto un valore “valido”



Ciclo do

- Si può usare un ciclo **while** “innaturale”

```
// si usa un'inizializzazione "ingiustificata"
double rate = 0; // rende vera la prima verifica
while (rate <= 50)
{   System.out.println("Inserire il tasso:");
    rate = console.nextDouble();
}
```



Ciclo do

- Oppure si può ricopiare il corpo del ciclo (o una sua parte) prima del ciclo stesso

```
System.out.println("Inserire il tasso:");
double rate = console.nextDouble();
while (rate <= 50)
{
    System.out.println("Inserire il tasso:");
    rate = console.nextDouble();
}
```



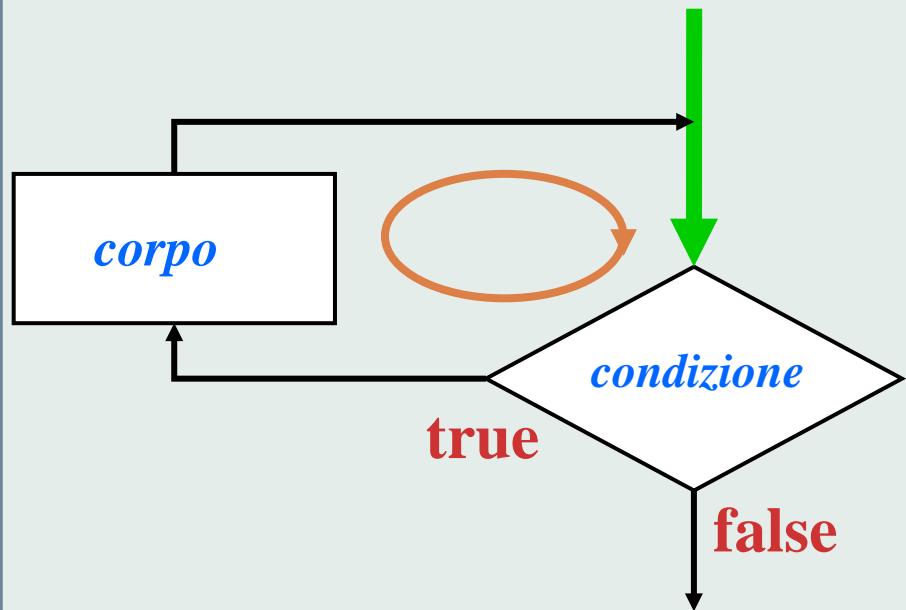
Ciclo do

- Ma per comodità e chiarezza esiste il ciclo **do**

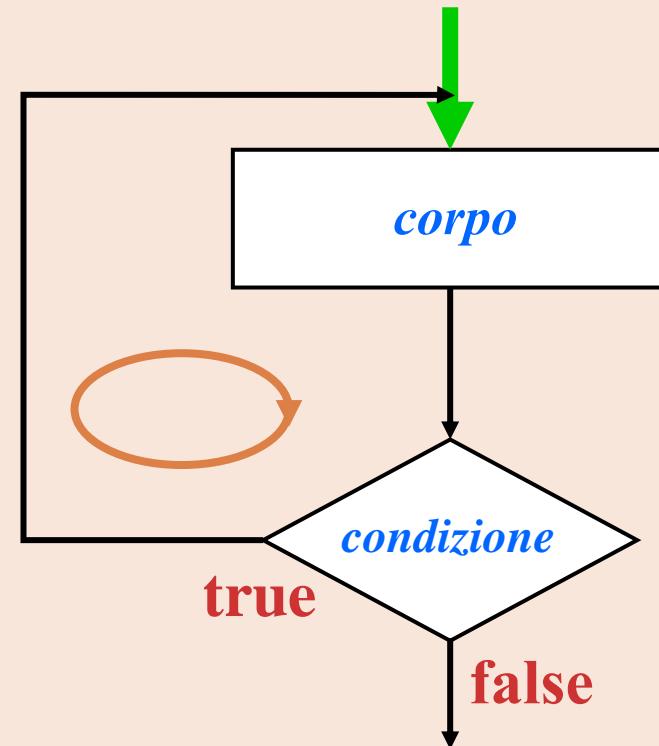
```
double rate; // manca valore iniziale ma OK !!!  
do  
{   System.out.println("Inserire il tasso:") ;  
    rate = console.nextDouble();  
} while (rate <= 50);  
// qui certamente a rate è stato assegnato un valore
```

Il ciclo *do* e il ciclo *while*

Ciclo *while*



Ciclo *do*





Variabile non inizializzata

- Ecco un caso in cui è ammessa la dichiarazione di variabile senza inizializzazione
 - Molto meglio che inizializzarla a un valore immotivato

```
double rate; // manca valore iniziale ma OK !!!  
do  
{   System.out.println("Inserire il tasso:") ;  
    rate = console.nextDouble();  
} while (rate <= 50);
```

- Perché non definiamo la variabile nel punto in cui viene utilizzata per la prima volta? Due motivi!

```
do  
{   System.out.println(...);  
    double rate = ...;  
} while (rate <= 50);
```

Variabile non inizializzata

```
do
{   System.out.println(...);
    double rate = ...;
} while (rate <= 50);
```

- Facendo così ci sono due problemi
 - Al termine dell'esecuzione del ciclo **do**, la variabile **rate** non è più visibile, perché è definita all'interno del suo corpo
 - Quindi il valore letto da tastiera non è utilizzabile
 - In realtà, **rate** non è visibile nemmeno nel punto in cui si valuta la condizione di uscita dal ciclo (**rate <= 0**), sempre perché è definita all'interno del corpo, quindi si ha errore di sintassi



Enunciato *do*

□ Sintassi:

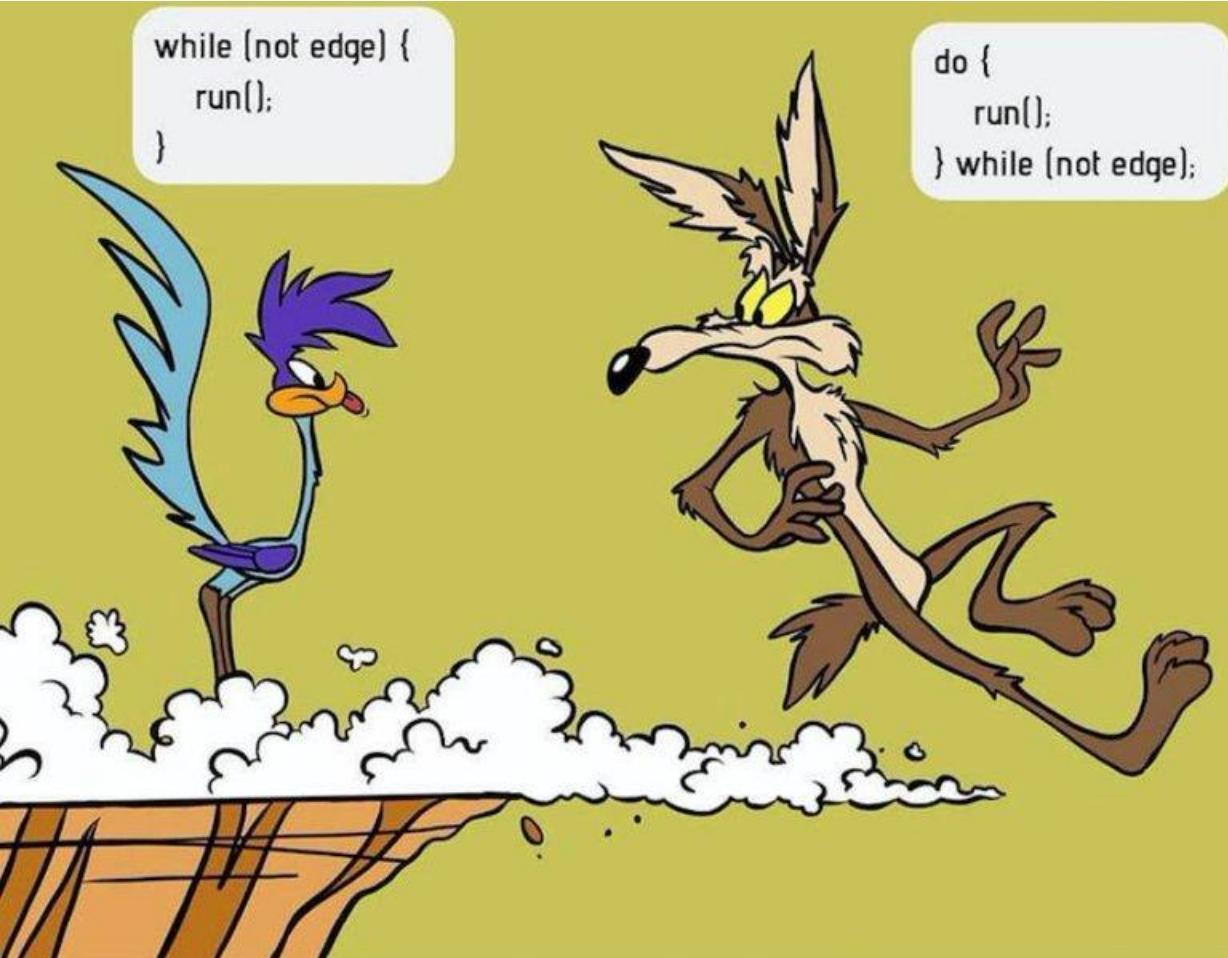
```
do
    enunciato
    while (condizione);
```

□ Scopo:

eseguire un **enunciato** una prima volta, poi eseguirlo nuovamente finché la **condizione** è **vera**

□ Nota: il **corpo** del ciclo **do** può essere un enunciato qualsiasi, quindi anche un **blocco di enunciati**

Raccomandazione: scegliere bene il tipo di ciclo che ci





DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

L'enunciato break

Un esempio di ciclo

- Problema: determinare se un particolare carattere è presente all'interno di una stringa

```
String s = "pippo";
char chToFind = 'i';
boolean found = false;
for (int i = 0; i < s.length(); i++)
{   char ch = s.charAt(i);
    if (ch == chToFind) {
        found = true;
    }
}
if (found) {
    System.out.println("Il carattere " + chToFind +
                       " si trova in " + s);
}
else{
    System.out.println("Il carattere " + chToFind +
                       " non si trova in " + s);
}
```

Funziona, ma, dopo aver eventualmente trovato ciò che cercava, **può darsi che esegua alcune iterazioni inutili**



L'enunciato break

- ***break*** ‘rompe’ il ciclo corrente
 - Se la condizione di uscita dal ciclo può essere verificata solo in un punto **INTERNO** al ciclo, si usa il comando *break*, dopo che la condizione è stata verificata, per **uscire** dal ciclo.

```
while (...)  
{  
    if (...)  
        break;  
}
```

```
for (...)  
{  
    if (...)  
        break;  
}
```



L'enunciato break

- Soluzione alternativa: usare l'enunciato **break**

```
String s = "pippo";
char chToFind = 'i';
boolean found = false;
for (int i = 0; i < s.length(); i++)
{   char ch = s.charAt(i);
    if (ch == chToFind)
    {
        found = true;
        break; // interrompe il ciclo, senza
    }           // completare l'iterazione in corso
}
if (found)
    ... // come prima
```

- L'enunciato **break** si può usare per **terminare un ciclo**
 - L'iterazione in corso **NON** viene completata, il ciclo termina "bruscamente"



L'enunciato break

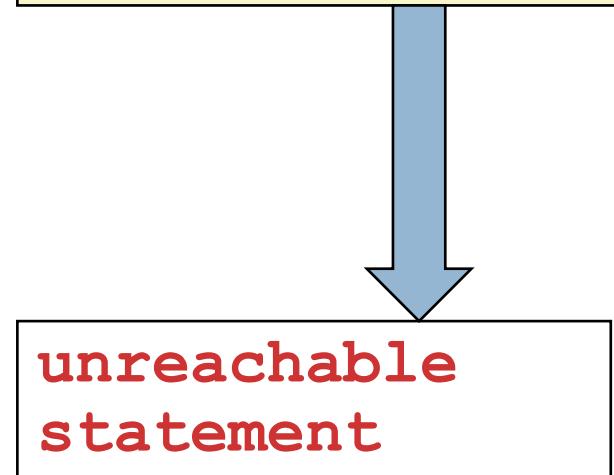
- Nel caso di cicli annidati, l'enunciato **break** provoca la terminazione del **ciclo più interno** tra quelli in cui si trova l'enunciato stesso

```
int i=0, j=0;
while (true)
{   System.out.println("i= "+i);
    while (true) {
        if(j==5) break;
        System.out.println("j= "+j);
        j++;
        ...
    }
    i++;
    if(i==3) break;
}
```

L'enunciato break

- L'esecuzione dell'enunciato **break** **dove** essere **condizionata**
 - In caso contrario, la porzione di corpo del ciclo che si trova dopo l'enunciato **break** (*enunciato2* nell'esempio) non viene **mai** eseguita
 - È certamente un errore logico, segnalato dal compilatore come “enunciato non raggiungibile” dal flusso d'esecuzione

```
while (...)  
{   enunciato1  
    break;  
    enunciato2  
}
```





Soluzione più efficiente

- Problema: determinare se un particolare carattere è presente all'interno di una stringa
- Per evitare iterazioni inutili, possiamo complicare la condizione di terminazione del ciclo

```
String s = "pippo";
char chToFind = 'i';
boolean found = false;
for (int i = 0; i < s.length() && !found; i++)
{   char ch = s.charAt(i);
    if (ch == chToFind) {
        found = true;
    }
} // questo ciclo può terminare per due motivi diversi
if (found){
    ... // come prima
```

- Anche l'algoritmo precedente era CORRETTO, ma questo può essere più veloce (valuteremo meglio più avanti)

L'enunciato break

- L'enunciato **break** NON è necessario, si può **sempre** scrivere codice **equivalente**, però è "comodo"

```
while (condA)
{
    { blocco 1 }

    if (condB) {
        break;
    }

    { blocco 2 }
}
```



```
boolean done = false;
while (condA && !done)
{
    { blocco 1 }

    if (condB) {
        done = true;
    }
    else
    { blocco 2 }
}
```

Enunciato *continue*

- Esiste anche l'enunciato ***continue*** e può essere usato all'interno dei cicli
 - provoca la terminazione dell'iterazione corrente e il passaggio all'iterazione successiva

```
while (condizione1)
{
    enunciato1
    if (condizione2)
        break;
    enunciato2
}
```

```
while (condizione1)
{
    enunciato1
    if (condizione2)
        continue;
    enunciato2
}
```



Esempio

```
int i=0;
while (i<10)
{
    if(i==3)
        {      i++;
            break;
        }
    System.out.print(i+" ");
    i++;
}
System.out.println();
```

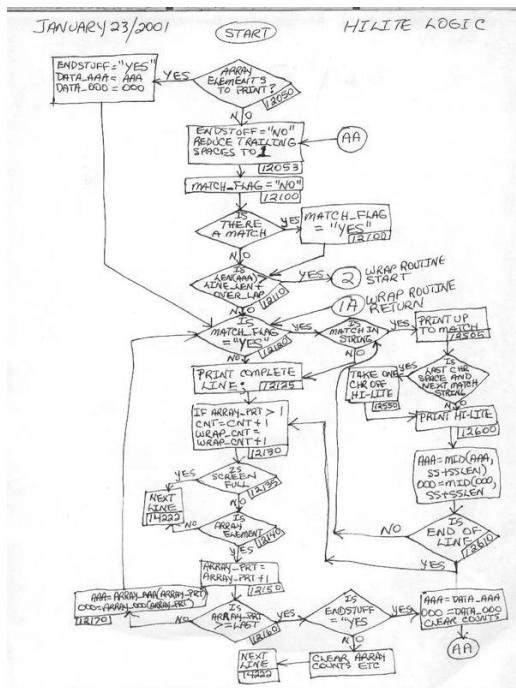
0,1,2,4,5,6,7,8,9

0,1,2

```
int i=0;
while (i<10)
{
    if(i==3)
        {      i++;
            continue;
        }
    System.out.print(i+" ");
    i++;
}
System.out.println();
```

Utilizzo di break e continue

L'uso di **break** e **continue** è non necessario e sconsigliabile perché contribuisce a creare **codice spaghetti** ovvero rappresentato da diagrammi di flusso pieni di linee, difficili da leggere e comprendere



The pasta theory of programming





Enunciato switch

- Una sequenza che confronti un'unica variabile intera con diverse alternative costanti può essere realizzata con un enunciato switch

```
int x;
int y;
. . .
if( x == 1)
    y = 1;
else if (x == 2)
    y = 4;
else if (x == 4)
    y = 16;
else
    y = 0;
```

```
int x;
int y;
. . .
switch (x)
{
    case 1: y = 1;
              break;
    case 2: y = 4;
              break;
    case 4: y = 16;
              break;
    default: y = 0;
              break;
}
```



Enunciato switch

- ❑ **Vantaggio:** non bisogna ripetere il nome della variabile da confrontare
- ❑ **Svantaggio:** si può usare solo se la variabile da confrontare è **intera** (byte, short, char, int) oppure una **stringa** (da Java 7)
- ❑ **Svantaggio:** non si può usare se uno dei valori da confrontare non è costante
- ❑ **Svantaggio:** ogni **case** deve terminare con un enunciato **break**, altrimenti viene eseguito anche il corpo del **case** successivo! Questo è fonte di molti errori...



Il problema del calendario...

```
public class Calendario{  
  
    public static void main(String[] args) {  
  
        Scanner console = new Scanner(System.in);  
        System.out.println("Inserisci un numero da 1 a 12");  
        int n = console.nextInt();  
        String mese;  
        switch (n) {  
            case 1: mese = "Gennaio";  
            break;  
            case 2: mese = "Febbraio";  
            break;  
            // ... altri casi  
            default: mese = "Non e' un mese valido";  
        }  
        System.out.println(mese);  
        console.close();  
    }  
}
```



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Errori comuni con i cicli



Errori per scarto di uno

```
. . .
int years = 0; // o forse years = 1 ???

while (balance < 2 * initialBalance)
{           // o forse balance <= 2 * initialBalance
???
    double interest = balance * rate / 100;
    balance = balance + interest;
    years++;
}
System.out.println("L'investimento "
    + "raddoppia in " + years + " anni");
```



Errori per scarto di uno

□ Come evitare questi errori per scarto di uno?

- Provare con alcuni semplici casi di prova
- Investimento iniziale 100 euro, tasso 50%
 - Allora `years` deve essere inizializzato a 0
 - `years = 0; balance = 150, years =1;` `balance = 225, years = 2`
 - `years = 1; balance = 150, years =2;` `balance = 225, years = 3`
- Tasso di interesse 100%
 - Allora la condizione deve essere < e non <=
 - `balance = 200; years =1;`
 - al controllo successivo se <= ci sarebbe un'altra iterazione:



Definire bene i limiti

- `for (int i = 0; i !=n; i++)` E se n fosse negativo?
 - La condizione $i \neq n$ sarebbe sempre vera
 - Meglio $i < n$
- Imparare a contare le iterazioni

```
for (i = a; i < b; i+=c)  
//oppure  
for (i = a; i <= b; i+=c)
```

eseguito $(b-a)/c$ volte
eseguito $(b-a+1)/c$ volte

Esempio a=0, b =10, c=1

- nel primo caso il ciclo viene eseguito 10 volte ($a=0, 1, \dots, 9$)
- nel secondo caso il ciclo viene eseguito 11 volte ($a=0, 1, \dots, 9, 10$)



Definire bene i limiti

□ Due intestazioni:

```
for (i = 0; i <= s.length() - 1; i++)
//è corretta ma è meglio quella sotto

for (i = 0; i < s.length(); i++)
```

- Sono equivalenti
- Nella seconda il codice è più leggibile



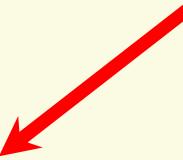
Punti e virgola mancanti...

```
for (years = 1; (balance = balance + balance * rate / 100)
      < targetBalance; years++)
//...;
System.out.println(years);
```

- In questo caso tutte le operazioni necessarie sono nell'intestazione del ciclo
 - Il corpo del ciclo deve essere vuoto
 - Ma abbiamo dimenticato il ";" dopo l'intestazione
 - Quindi l'istruzione di stampa viene considerata parte del corpo

... o di troppo!

```
sum = 0;  
int i;  
for (i = 1; i <= 10; i++) ;  
    sum = sum+i;  
System.out.println(sum);
```



- C'è un “;” di troppo!
 - il corpo del ciclo è vuoto
 - L'istruzione di aggiornamento di sum viene eseguita una sola volta, dopo l'uscita dal ciclo



Take home message

- ❑ Abbiamo visto come iterare un blocco di istruzioni
 - Ciclo while
 - Ciclo for
 - Ciclo do-while
- ❑ Abbiamo visto come utilizzarli per leggere un numero arbitrario di dati da standard input
- ❑ Abbiamo visto come interrompere il ciclo in punti arbitrari
 - break, continue
- ❑ Attenzione con le condizioni!



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Array

capitolo 7



Problema

- Scrivere un programma che:
 - legga dallo standard input una sequenza di dieci numeri in virgola mobile, uno per riga
 - chieda all'utente un numero intero **index** e visualizzi il numero che nella sequenza occupava la posizione indicata da **index**
- Occorre **memorizzare tutti i valori della sequenza**



Come memorizzarli?

□ Potremo:

- ❑ usare dieci variabili diverse per memorizzare i valori
 - `x1, x2, x3, x4, x5, x6, x7, x8, x9, x10`
- ❑ selezionarli con una lunga sequenza di alternative
 - `if (index == 1) System.out.println(x1);`
 - `...`
 - `else if (index==10) System.out.println(x10);`
- ❑ *Ma se i valori dovessero essere mille?*
- ❑ *O se il numero di valori dovesse essere chiesto all'utente come dato iniziale?*



Soluzione

- Ciò che veramente ci serve è ***un contenitore di dati di dimensione arbitraria***, una dimensione che possa essere ***decisa durante l'esecuzione del programma***
- Questo non si può fare con un semplice insieme di variabili: il numero di enunciati di dichiarazione di variabili deve essere noto al programmatore
- Inoltre, il contenitore **non** deve essere analogo a un insieme matematico: ci interessa ***che preservi le posizioni dei dati al suo interno***, come una stringa



© traveler 1116/iStockphoto.

Immaginate un treno di vagoni numerati per posizione, ciascuno dei quali contiene un carico dello stesso tipo, ma che puo' variare come quantita'



Le strutture dati

- Strumenti concettuali e concreti per organizzare l'informazione
 - Struttura dati astratta (Abstract Data Type): descrive i dati e le funzionalità che la struttura deve avere
 - Struttura dati: implementa le funzionalità indicate dalla ADT
- Algoritmi e Strutture Dati sono fondamentali per progettare soluzioni computazionali efficaci ed efficienti



Obiettivi

- Imparare a conoscere punti di forza e debolezza delle varie strutture dati
 - ▣ Capacità di scegliere la DS più adeguata ai dati e al tipo di elaborazione che dovrete eseguire
- Imparare a conoscere (alcuni) algoritmi per ricercare e ordinare i dati
 - ▣ Capacità di sfruttare gli algoritmi migliori in base al contesto in cui opererete per ottenere soluzioni più efficienti
- Implementare le strutture dati: capire cosa c'è sotto il “cofano”
 - ▣ Ottimizzare l'utilizzo o inventare qualcosa di nuovo!



Le sequenze (o liste) di dati

□ ADT Sequenza (o Lista)

- ▣ Lineare: ogni elemento ha un predecessore e un successore (a parte il primo e l'ultimo)
- ▣ Operazioni: inserimento, rimozione, ricerca

□ Strutture dati che la implementano:

- ▣ Array
- ▣ Lista concatenata



Array

- Lo strumento messo a disposizione dal linguaggio Java (e in molti altri linguaggi di programmazione) per memorizzare una sequenza di dati si chiama **array** (che significa “schiera” o “sequenza ordinata”)

- In Java, un **array** è **un oggetto** che rappresenta **una sequenza posizionale di dati omogenei** (cioè tutti dello stesso tipo), che vengono detti elementi o componenti dell’array



Sommario

- Come costruire un array
- Come utilizzare un array
- Come conoscere la lunghezza di un array
- Errori tipici
- Array come parametri



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
- 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
- **In Java, le parentesi quadre si usano solo per gli array**

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



Costruire un array

- 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**...

Costruire un array

- Si dice anche che l'array è costituito da celle o posizioni, nelle quali sono memorizzati i suoi elementi
 - **le posizioni sono identificate da numeri interi non negativi**
 - esiste il dato in prima posizione (la posizione 0), quello in seconda posizione (la posizione 1), ecc.
 - quindi l'array **NON** rappresenta un insieme matematico, anche perché i dati presenti nell'array possono essere replicati in diverse posizioni

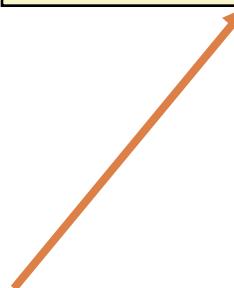
0	1	2	3	4	5	6	7
A	B	C	A	A	F	D	Z

Riferimento a un array

- Come avviene dopo la costruzione di qualunque 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];
```

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

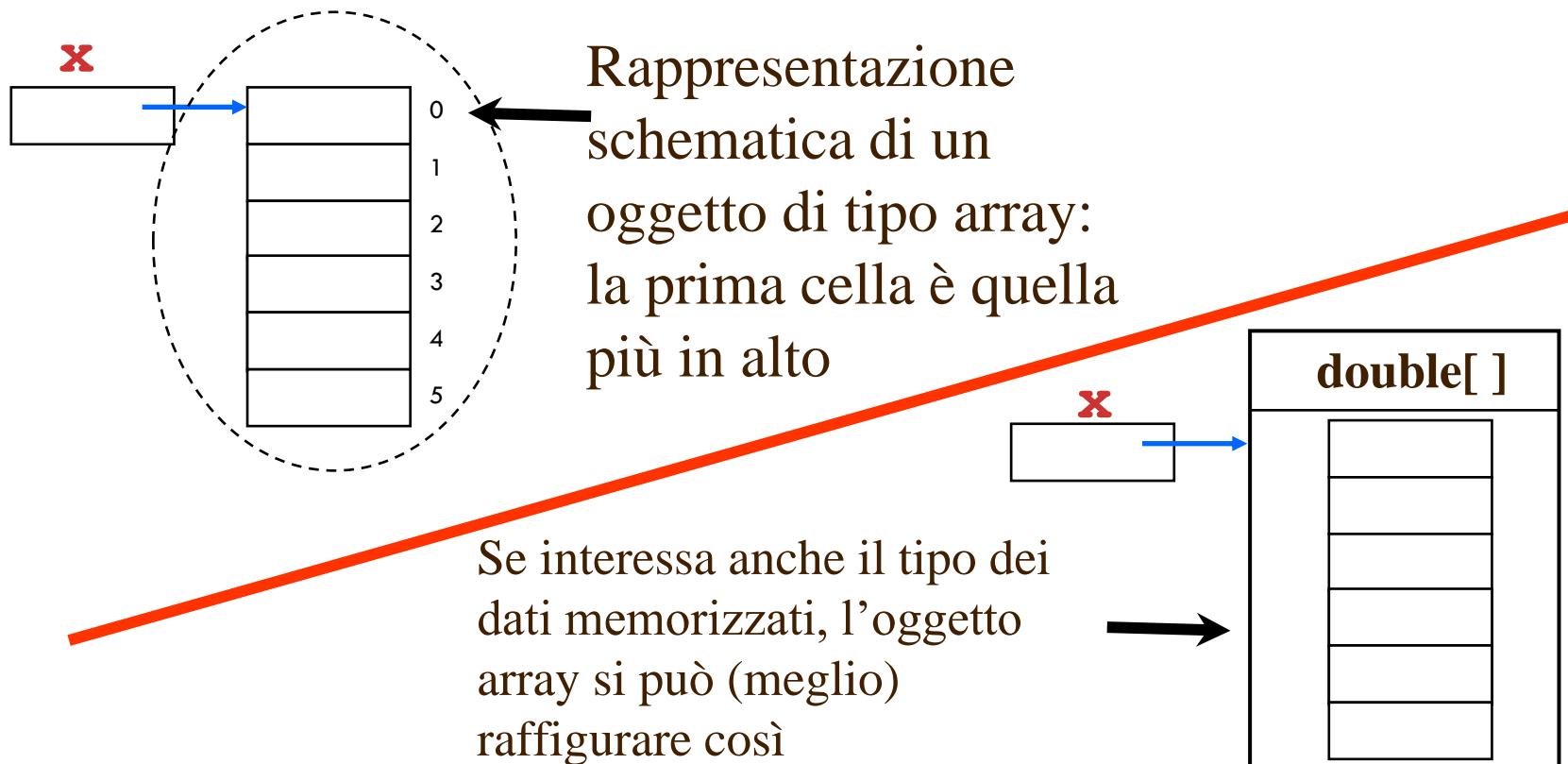


- **Attenzione:** nella definizione della variabile oggetto devono essere presenti le parentesi quadre, ma NON deve essere indicata la dimensione dell'array; la variabile potrà riferirsi solo ad array di quel tipo, ma **di qualunque dimensione**
 - Non esiste, quindi, il **tipo di dato** "array di dieci **double**", bensì il tipo di

Riferimento a un array

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

- Una variabile **x** che si riferisce a un array è
una variabile oggetto che contiene un riferimento all'oggetto array





DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Da qui venerdi' 8



Utilizzare un array

- Al momento della costruzione, tutti gli elementi dell'array vengono inizializzati a un valore, seguendo *le stesse regole viste per le variabili di esemplare prive di inizializzazione esplicita*

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

- *Tutti gli elementi sono inizializzati a 0 in questo caso*

- Si accede al contenuto di una cella dell'array indicando tra parentesi la posizione voluta
 - Accesso al contenuto della posizione i dell'array values
 - `values[i]`



Utilizzare un array

- Si può **leggere** il contenuto di una cella dell'array, per usarlo in un'espressione

```
double doubleVar = 5 + values[3];
```

- La stessa sintassi si usa per **scrivere** un valore in una cella dell'array, ponendola nella parte sinistra di un enunciato di assegnazione

```
values[9] = 3.4 + Math.sqrt(2);
```



Utilizzare un array

```
double[] values = new double[10];  
double oneValue = values[3]; // vale 0  
values[9] = 3.4;
```

- Il numero utilizzato per accedere a un particolare elemento dell'array si chiama *indice*
- L'indice può assumere un valore compreso tra **0 (*incluso*)** e la **dimensione** dell'array (*esclusa*), cioè segue le stesse convenzioni viste per le posizioni dei caratteri in una stringa
 - il primo elemento ha indice 0
 - l'ultimo elemento ha indice (*dimensione* - 1)
- **A tutti gli effetti sintattici e semantici, un elemento di un array equivale a una singola variabile dello stesso tipo**
 - **values [3]** è, quindi, una variabile di tipo **double**



Utilizzare un array

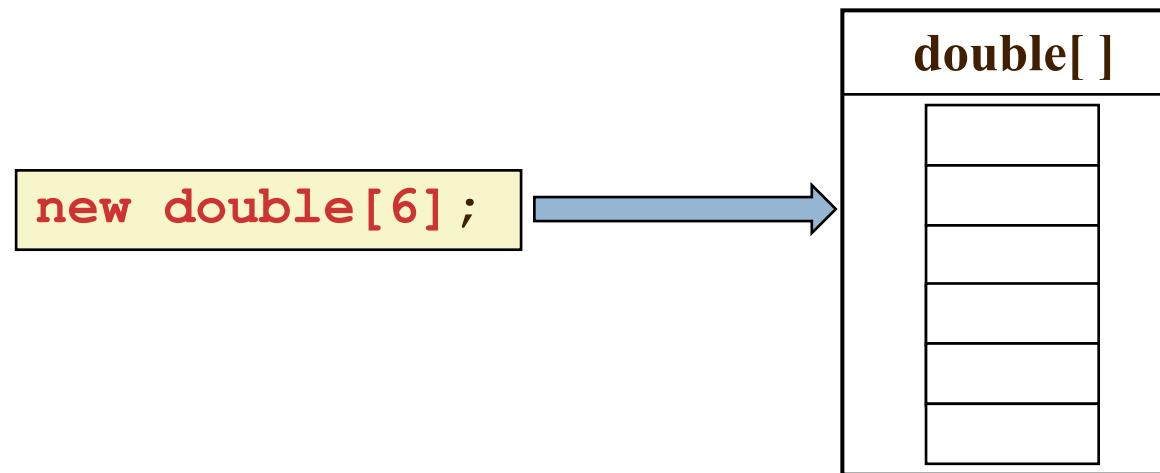
- L'indice di un elemento di un array può, più in generale, essere un'espressione di tipo **int**



Utilizzare un array

- Cosa succede se si accede a un elemento dell'array usando un indice sbagliato (maggiore o uguale alla dimensione, o negativo) ?
 - L'ambiente di esecuzione (cioè l'interprete) lancia un'eccezione di tipo **ArrayIndexOutOfBoundsException**
 - In modo simile a quanto abbiamo visto con le stringhe!

Gli array sono una struttura dati ad accesso casuale



- **Struttura ad accesso casuale**
 - Cioè **il tempo di accesso (in lettura o scrittura) a uno dei suoi elementi non deve dipendere dal valore dell'indice associato a tale elemento**
 - Questa proprietà ci sarà molto utile nella valutazione delle prestazioni degli algoritmi in merito al tempo di esecuzione



La dimensione di un array

- Un array è un oggetto un po' strano...
 - non ha metodi pubblici, né statici né di esemplare
- L'unico elemento pubblico di un oggetto di tipo array è la sua dimensione, a cui si accede attraverso la sua **variabile pubblica di esemplare length** (attenzione, non è un metodo!)

```
double[] values = new double[10];  
int a = values.length; // a vale 10
```
- Una variabile **pubblica** di esemplare sembrerebbe una violazione delle regole...



La dimensione di un array

- In realtà, **length** è una variabile pubblica ma è **final**, quindi ***non può essere modificata***, può soltanto essere ispezionata
 - Questo paradigma è, in generale, considerato accettabile nell'OOP

```
double[] values = new double[10];  
values.length = 15; // ERRORE IN COMPILAZIONE
```

- In alternativa si sarebbe potuto fornire un metodo pubblico per accedere alla variabile privata

```
double[] values = new double[10];  
int a = values.getLength(); // non è così!!
```

- la soluzione progettuale scelta è meno elegante ma fornisce lo stesso livello di protezione dell'informazione ed è più veloce in esecuzione

Soluzione del problema iniziale

```
import java.util.Scanner;
public class SelectValue
{ public static void main(String[] args)
{
    Scanner console = new Scanner(System.in);
    double[] values = new double[10];
    // usiamo values.length anziché ripetere il
    // numero 10, non usare "numeri magici" !!
    for (int i = 0; i < values.length; i++){
        values[i] = console.nextDouble();
    }

    System.out.println("Inserisci un numero:");
    int index = console.nextInt();
    if (index < 0 || index >= values.length){
        System.out.println("Valore errato");
    }
    else{
        System.out.println(values[index]);
    }
    console.close();
}
```



Variante

```
import java.util.Scanner;
public class SelectValue2
{ public static void main(String[] args)
{ Scanner console = new Scanner(System.in);
  System.out.println("Dimensione?");
  int dim = console.nextInt();
  double[] values = new double[dim];
  for (int i = 0; i < values.length; i++){
    System.out.println("Introduci un
valore");
    values[i] = console.nextDouble();
  }
  System.out.println("Inserisci un numero:");
  int index = console.nextInt();
  if (index < 0 || index >= values.length){
    System.out.println("Valore errato");
  }
  else{
    System.out.println(values[index]);
  }
  console.close();
}
```



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'interprete Java segnala questo errore con il lancio di un'eccezione

Errori di mancata inizializzazione

- Attenzione... quando creo un array di oggetti tutti i riferimenti sono null!

```
BankAccount[] accounts = new BankAccount[10];  
  
System.out.println("Saldo nel conto 0"+  
                    accounts[0].getBalance());
```

Exception in thread "main" java.lang.NullPointerException

- Prima di invocare metodi devo aver inizializzato le singole celle!

```
BankAccount[] accounts = new BankAccount[10];  
accounts[0] = new BankAccount(100);  
System.out.println("Saldo del cliente 0"  
                    +accounts[0].getBalance());
```



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

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

E' possibile la creazione di oggetto senza new, che è implicito



Passare un array come parametro

- Spesso si scrivono metodi **statici** che ricevono **array** come **parametri esplicativi**

```
public static double sum(double[] values)
{
    if (values.length == 0) {
        return 0; //questo if è inutile... perché?
    }
    double sum = 0;
    for (int i = 0; i < values.length; i++) {
        sum = sum + values[i];
    }
    return sum;
}
// è comodo che un array abbia length
// altrimenti dovrei passare 2 parametri
```

Passare un array come parametro

- Ricevendo una copia della variabile riferimento, nel metodo invocato **è anche possibile modificare gli elementi** dell'array: il metodo chiamante vedrà gli elementi dell'array

```
public static void incrementAll(double[] values)
{ // se l'array ha lunghezza zero, non fa nulla
  // senza protestare... Scelta ragionevole
  for (int i = 0; i < values.length; i++) {
    values[i]++; // si comporta come una variabile
  }
}
```

Estratto di codice dal metodo main:

```
double[] valori = {1,5,10,32};
incrementAll(valori);
for(int i=0; i<valori.length; i++){
    System.out.println(valori[i]);
}
//stampa: 2,6,11,33
```



Usare un array come valore restituito

- Un metodo può anche restituire un (riferimento a un) array

```
public class Util
{  public static String[] getDayNames()
    {  String[] days = {"Lunedì",
                       "Martedì", "Mercoledì",
                       "Giovedì", "Venerdì",
                       "Sabato", "Domenica"} ;

        return days;
    }
}
```

Attenzione: al termine dell'esecuzione del metodo, la variabile locale **days** "muore", ma l'oggetto array che è stato creato rimane in memoria!
Il suo indirizzo è assegnato alla variabile (**dayNames**) che, nell'invocante, raccoglie il valore restituito dal metodo

- Esempio

```
String[] dayNames = Util.getDayNames() ;
System.out.println(dayNames[1]) ;
```



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Sintassi sugli array: riassunto



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
 - ? Se ***lunghezza*** è negativa, viene lanciata l'eccezione **NegativeArraySizeException**



Costruzione di un array

`new NomeTipo [lunghezza]`

- Nota: ***NomeTipo*** può essere uno dei **tipi primitivi** di Java o il nome di **una classe**
- Nota: i singoli elementi dell'array vengono inizializzati con le stesse regole delle variabili di esemplare che siano prive di inizializzazione esplicita
 - **0** per variabili numeriche e caratteri, **false** per variabili booleans, **null** per variabili oggetto



Variabile 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, ciascuno dei quali sarà di tipo ***NomeTipo***
- Le parentesi quadre [] sono necessarie e **non** devono contenere l'indicazione della dimensione dell'array



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**



Argomenti sulla riga di comando

public static void main(**String[] args**)

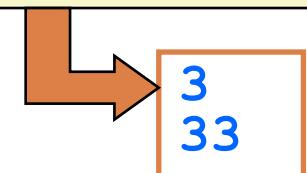
Argomenti sulla riga comandi

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

```
java Program 2 33 Hello
```

- Tali parametri vengono letti dall'interprete Java (che li riceve dal sistema operativo) e trasformati in un array di stringhe che costituisce il parametro del metodo

```
main
public class Program {
    public static void main(String[] args) {
        System.out.println(args.length);
        System.out.println(args[1]);
    } // qui sotto l'output del programma, se viene
} // eseguito con la riga di comando vista sopra
```





Argomenti sulla riga comandi

- Se tra gli argomenti che si vogliono inserire c'e' una stringa composta da piu' parole, e' necessario metterla tra doppi apici

```
MacBook-Pro-di-Cinzia:lezioni2023 cinzia$ java ProgramArgs 10 33 ciao come stai
Il numero di parametri inseriti e' 5
parametro 1 : 10
parametro 2 : 33
parametro 3 : ciao
parametro 4 : come
parametro 5 : stai
MacBook-Pro-di-Cinzia:lezioni2023 cinzia$ java ProgramArgs 10 33 "ciao come stai"
Il numero di parametri inseriti e' 3
parametro 1 : 10
parametro 2 : 33
parametro 3 : ciao come stai
MacBook-Pro-di-Cinzia:lezioni2023 cinzia$ █
```



Argomenti sulla riga comandi

- Le specifiche della JVM dicono che
 - la lunghezza dell'array corrisponde sempre al numero di parametri forniti sulla riga comandi
 - se non vengono forniti parametri sulla riga comandi, viene passato al metodo **main** un array di lunghezza zero (perfettamente valido)

```
public class Program {  
    public static void main(String[] args) {  
        if (args.length == 0){  
            System.out.println("Nessun parametro");  
        }  
    }  
}
```

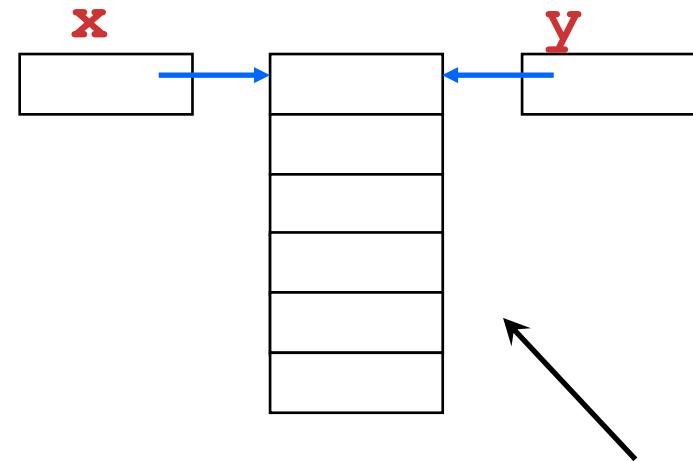


DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

"Copiare" o clonare un array

Copiare un array

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



Rappresentazione grafica "sintetica" dell'array

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



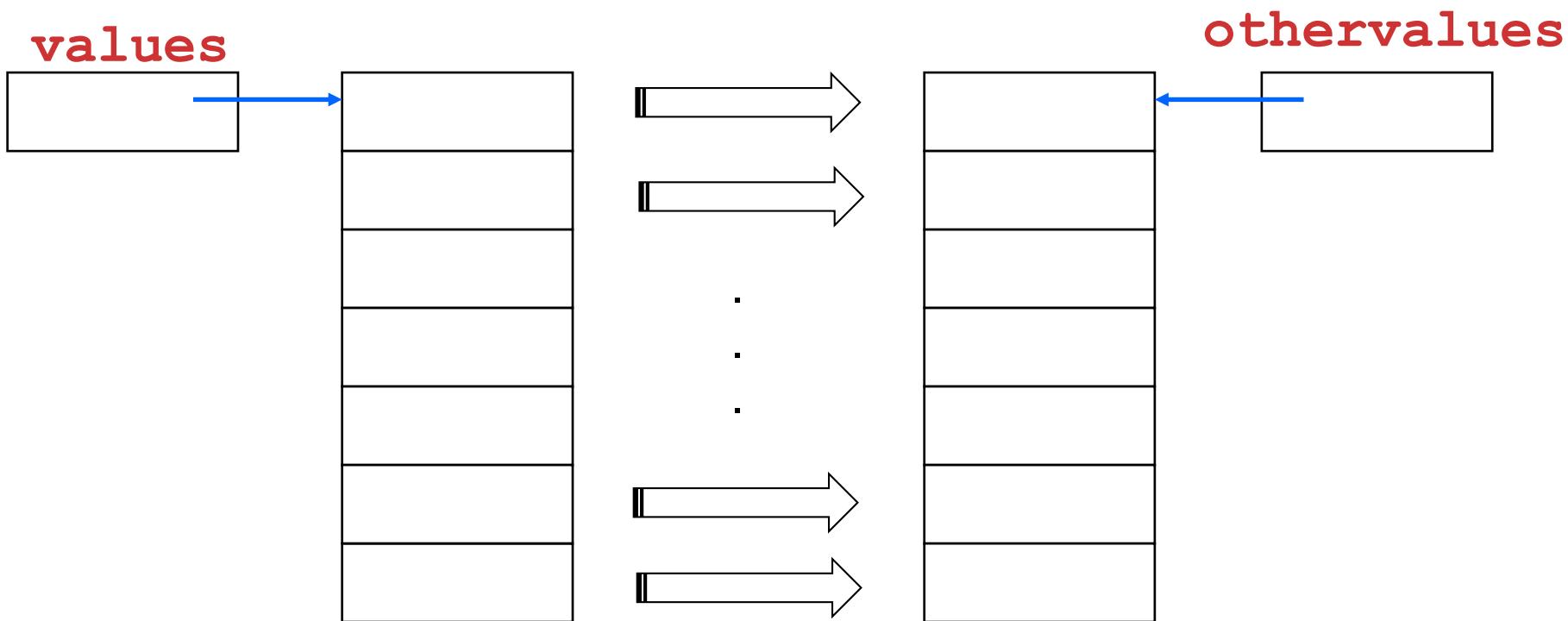
Clonare un array

- Se si vuole **clonare** un array, cioè ottenere **una copia dell'array** con identico contenuto, bisogna
 - **creare un nuovo array dello stesso tipo e con la stessa dimensione**
 - **copiare ogni elemento del primo array nel corrispondente elemento del secondo array**

```
double[] values = new double[10];  
// inseriamo i dati nell'array  
...  
double[] otherValues = new double[values.length];  
for (int i = 0; i < values.length; i++) {  
    otherValues[i] = values[i];  
}
```

Questa clonazione è
sempre possibile!!

Clonare un array



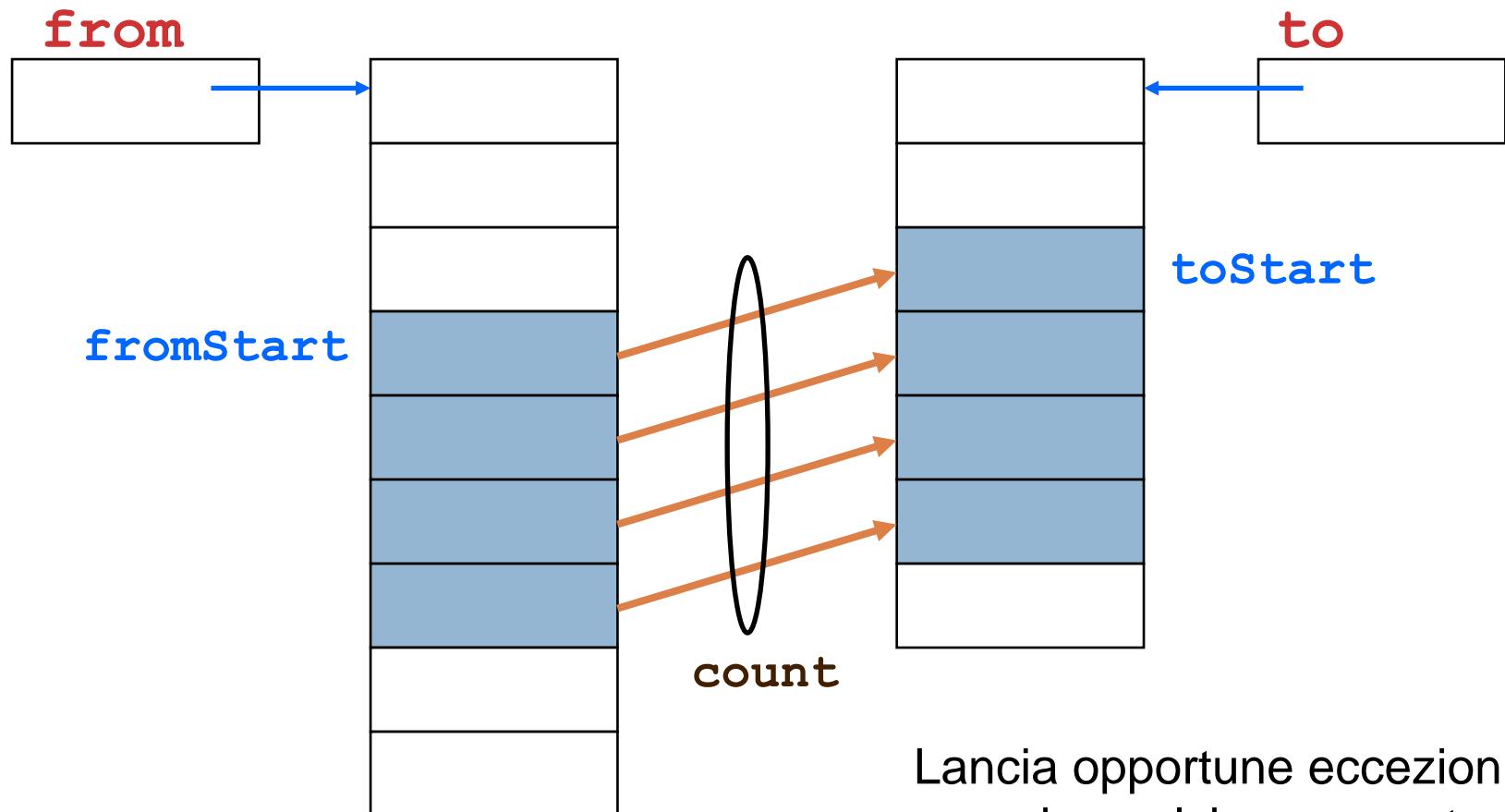
Attenzione alla
minuscola!

Clonare un array

- Invece di usare un ciclo, è possibile invocare il metodo statico **arraycopy** della classe **System**
- Il metodo **System.arraycopy** consente, più in generale, di copiare una **porzione** di un array in un altro array
 - Anche l'array di destinazione (quello in cui si copiano i dati) deve già esistere e deve essere grande a sufficienza
 - NB: si può usare all'esame!

Parametri di System.arraycopy

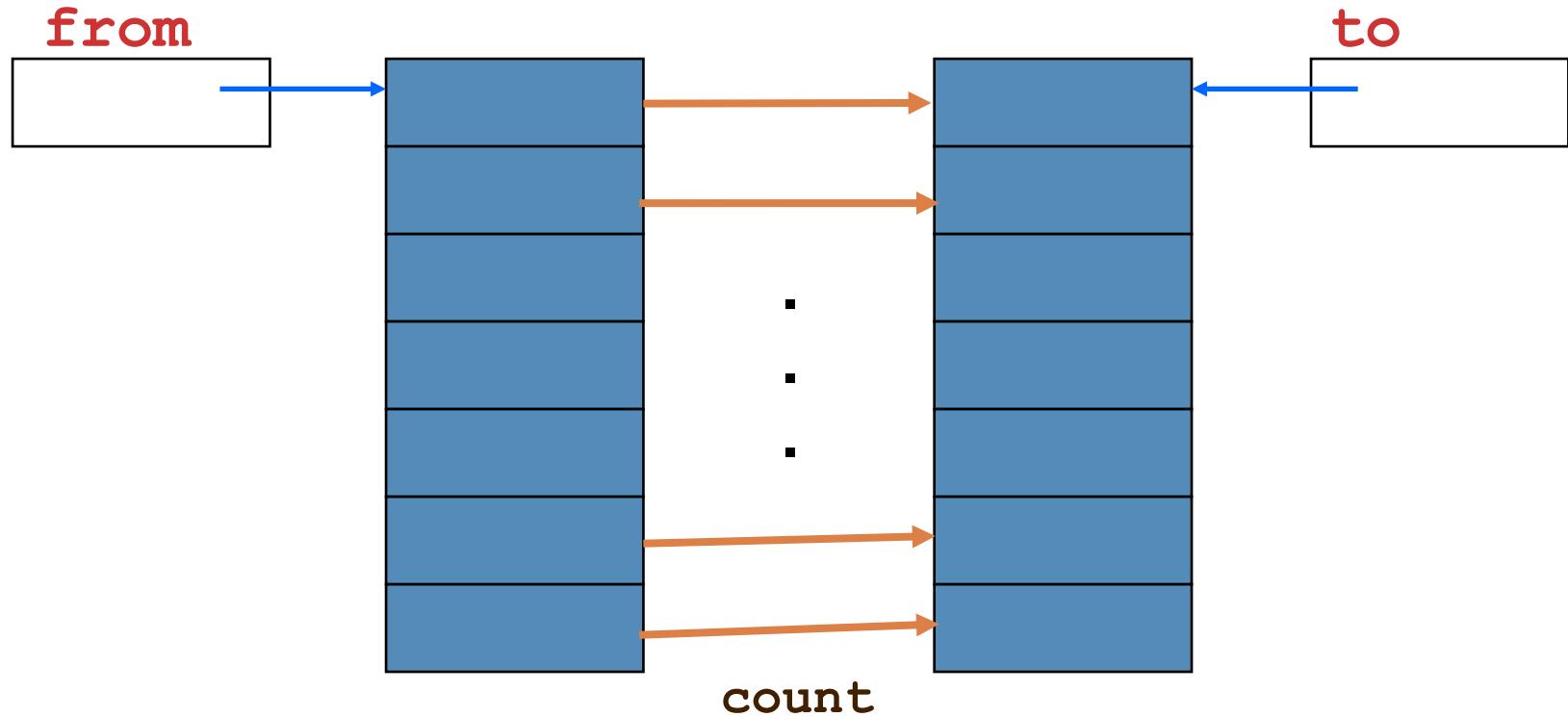
```
System.arraycopy(from, fromStart, to, toStart, count);
```



Lancia opportune eccezioni
quando qualche parametro è
sbagliato

Copiare un intero array con System.arraycopy

```
System.arraycopy(from, 0, to, 0, from.length);
```



Copro l'intero array “from”



Esempio

```
double[] values = new double[10];
// inseriamo i dati nell'array
...
double[] otherValues = new double[values.length];
System.arraycopy(values, 0, otherValues, 0, values.length);
```



Clonare un array

- E' anche possibile usare il metodo
 - **T[] Arrays.copyOf(T[] original, int newlength)**

```
double[] otherValues = Arrays.copyOf(values, values.length);
```

- **Se newlength == values.length**
 - Copio tutto l'array
- **Se newlength < values.length**
 - Tronco l'array alla dimensione fornita
- **Se newlength > values.length**
 - Copio tutto l'array e nelle celle in più vengono messi valori di default secondo le solite regole
- **NB: non si può usare all'esame!**



Clonare 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[]**
- NB: non si può usare all'esame!



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Array riempiti solo in parte



Array riempiti solo in parte

- Scrivere un programma che:
 - ▣ legga dallo standard input una sequenza di numeri in virgola mobile, uno per riga, ***finché i dati non sono finiti***
 - ad esempio, i dati terminano inserendo una riga vuota (non posso far chiudere lo standard input perché serve ancora...)
 - ▣ chieda all'utente un numero intero **index** e visualizzi il numero che nella sequenza occupava la posizione indicata da **index**
- La differenza rispetto al caso precedente è che ora, nel momento in cui il programma deve creare l'array, ***non si sa quanti saranno i dati*** introdotti dall'utente



Array riempiti solo in parte

- Problema: *per costruire un array è necessario indicarne la dimensione*, che è una sua proprietà final
 - *gli array in Java non possono crescere!*
- Possibile 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
 - Vedremo poi una soluzione migliore



Array riempiti solo in parte

- Nuovo problema: al termine dell'inserimento dei dati da parte dell'utente, in generale, non tutto l'array conterrà dati validi
 - ☒ *è necessario tenere traccia di quale sia l'indice dell'ultimo elemento dell'array che contiene dati validi*

```
Scanner c = new Scanner(System.in);
double[] values = new double[1000]; // oppure 10000 ?
int valuesSize = 0;
while (true) // sembra infinito... ma poi c'è break...
{   String s = c.nextLine(); // non usiamo nextDouble perché
                            // cerchiamo una riga vuota
    if (s.equals("")){ // oppure if (s.length() == 0)
        break; // riga vuota!
    }
    values[valuesSize] = Double.parseDouble(s);
    valuesSize++;
}
```

```
// valuesSize è l'indice del primo dato non valido
// nell'array e deve sempre essere non negativo e
// non maggiore della dimensione dell'array;
// è anche il numero di dati inseriti!!
// nota: qual è il suo valore iniziale? Osserva bene
```

```
System.out.println("Inserisci un numero:");
int index = Integer.parseInt(c.nextLine());
if (index < 0 || index >= valuesSize) {
    System.out.println("Valore errato");
}
else{
    System.out.println(values[index]);
}
```

Dimensione fisica vs dimensione logica

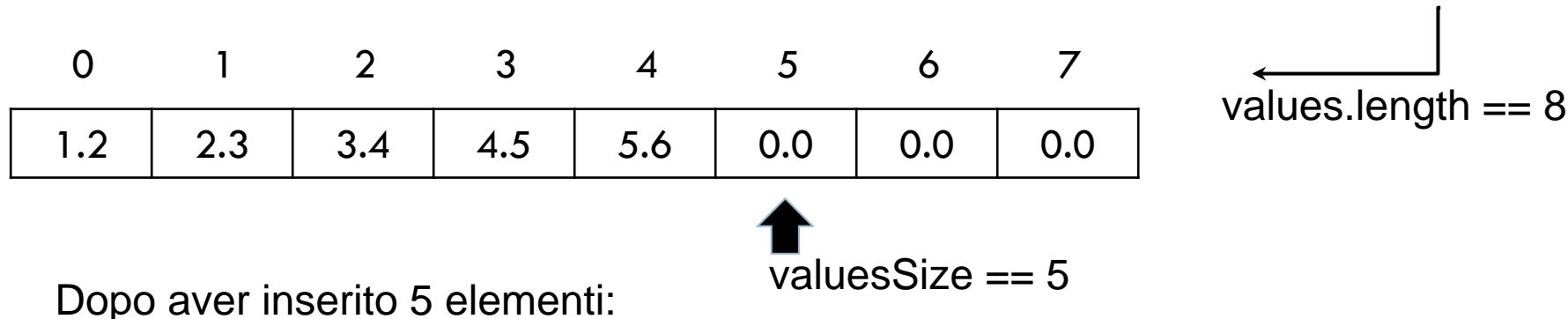
□ Dimensione fisica

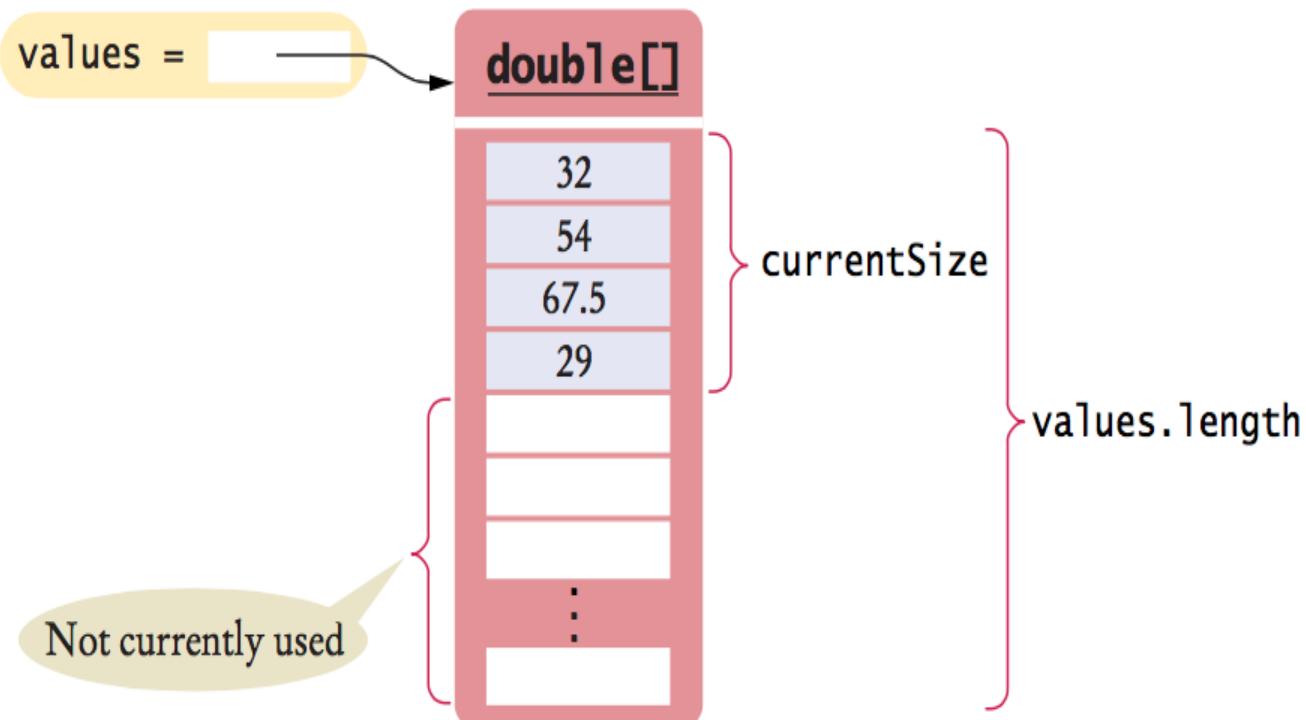
- **values.length** è il numero di valori *memorizzabili* ed è la dimensione **fisica** dell'array

□ Dimensione logica

- **valuesSize** è il numero di valori *memorizzati* ed è la dimensione **logica** dell'array

```
double[] values = new double[8];
```







Array riempiti solo in parte

- Si dice che un array utilizzato in questo modo è “**riempito solo in parte**”, anche se sarebbe meglio dire “**riempito in una sua porzione iniziale**” (eventualmente vuota o eventualmente coincidente con l’intero array)

- **valuesSize** è una variabile **esterna** all’array, ***non esistono oggetti primitivi di tipo “array riempito solo in parte”***

- L’array riempito solo in parte è, quindi, semplicemente **una modalità di utilizzo di un normale array**, a cui viene associata, dal punto di vista logico (ma non strutturale), una variabile di tipo **int**



Array riempiti solo in parte

- La soluzione presentata 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**

```
while (true)
{   String s = c.nextLine();
    if (s.equals("")) break;
    values[valuesSize] = Double.parseDouble(s);
    valuesSize++;
}
```

- Ci sono due possibili soluzioni
 - **impedire l'inserimento di troppi dati, segnalando l'errore all'utente**
 - **"ingrandire" l'array quando ce n'è bisogno**



Array riempiti solo in parte

```
// modifichiamo il ciclo di lettura
// per impedire l'inserimento di troppi dati
...
while(true)
{   String s = c.nextLine();
    if (s.equals(""))
        break;
    if (valuesSize == values.length) // è pieno
    {   System.out.println("Basta: troppi dati!");
        // eventualmente si può lanciare eccezione
        break;
    }
    values[valuesSize] = Double.parseDouble(s);
        // attenzione a dove si mette la verifica:
        // metterla qui sarebbe concettualmente errato
        // perché?

    valuesSize++;
}
...
```



Array riempiti solo in parte

```
// modifichiamo il ciclo di lettura
// per impedire l'inserimento di troppi dati
...
while(true)
{   String s = c.nextLine();
    if (s.equals(""))
        break;

    values[valuesSize] = Double.parseDouble(s);

    // ERRATO!!!
    if (valuesSize == values.length) // è pieno
    {   System.out.println("Basta: troppi dati!");
        // eventualmente si può lanciare eccezione
        break;
    }
    valuesSize++;
}

...
```



Cambiare dimensione a un array

- Abbiamo già visto come in Java sia ***impossibile*** aumentare (o diminuire) la dimensione di un array
- Ciò che si può fare è:
 - creare un nuovo array più grande di quello “ pieno ”
 - ad esempio il doppio, ma potrebbe bastare un elemento in più
 - copiarne il contenuto
 - “ abbandonare ” il vecchio array, usando poi quello nuovo



Cambiare dimensione a un array

- si parla di ***array dinamico***, sottintendendo che in realtà si tratta di una ***gestione dinamica dell'array***: non può esserci ambiguità perché **in Java non esistono array dinamici, cioè di dimensione modificabile**
- si dice anche che si ***ridimensiona l'array***, di nuovo senza ambiguità



Cambiare dimensione a un array

```
if (valuesSize == values.length) // è pieno
{   double[] newValues = new double[values.length * 2];
    for (int i = 0; i < values.length; i++) {
        newValues[i] = values[i];
    }
    values = newValues;
// valuesSize ovviamente non cambia;
// values non punta più al vecchio
// array, che viene abbandonato
}
```

Perché *** 2** e non **+ 1**?
Per motivi di **efficienza**,
come **vedremo**.
Naturalmente funziona
anche con + 1.

Il metodo statico *resize*

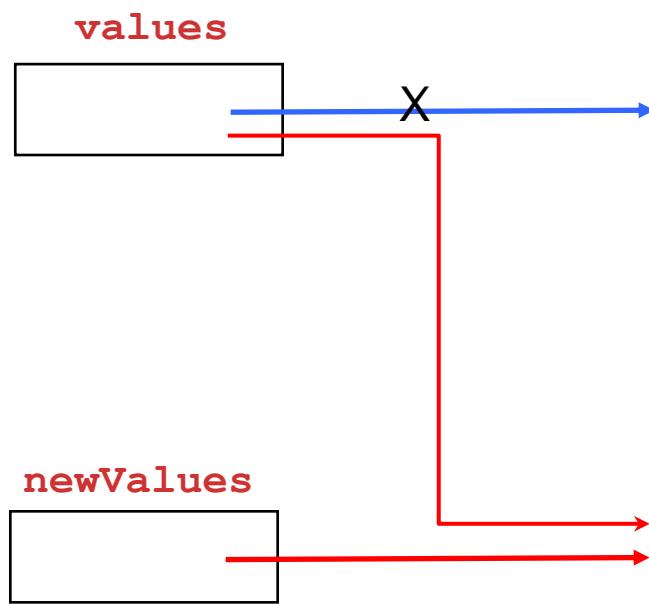
- Possiamo progettare un metodo di utilità che restituisca un array “ingrandito” a partire da quello che viene fornito

```
public class ArrayUtil
{
    public static double[] resize(double[] oldArray, int
newLength)
    { if (newLength < oldArray.length) {
        // gestire la situazione come piu' opportuno
    }
        double[] newArray = new double[newLength];
        for (int i = 0; i < oldArray.length; i++) {
            newArray[i] = oldArray[i];
        }
        return newArray;
    } // la "coda" del nuovo array rimane riempita
      // con i valori predefiniti
}
```

Si usa cosi'

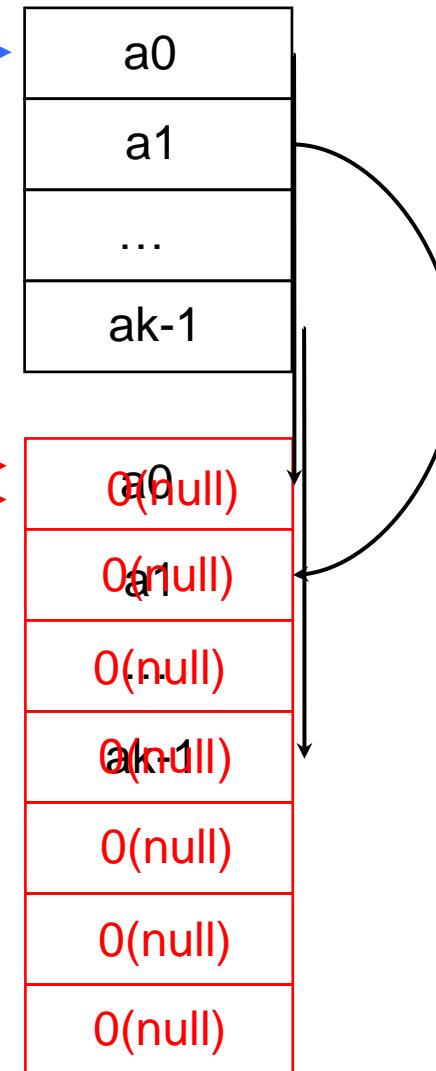
```
double[] values = {1, 2.3, 4.5};
values = ArrayUtil.resize(values, 5);
values[4] = 5.2;
```

Visualizzazione di resize



```
values = ArrayUtil.resize(values, 7);
```

Questa istruzione crea la linea rossa
(riferimento in memoria) tra `values`
e il nuovo array



Il vecchio array viene
“abbandonato”

nuovo array
restituito da resize

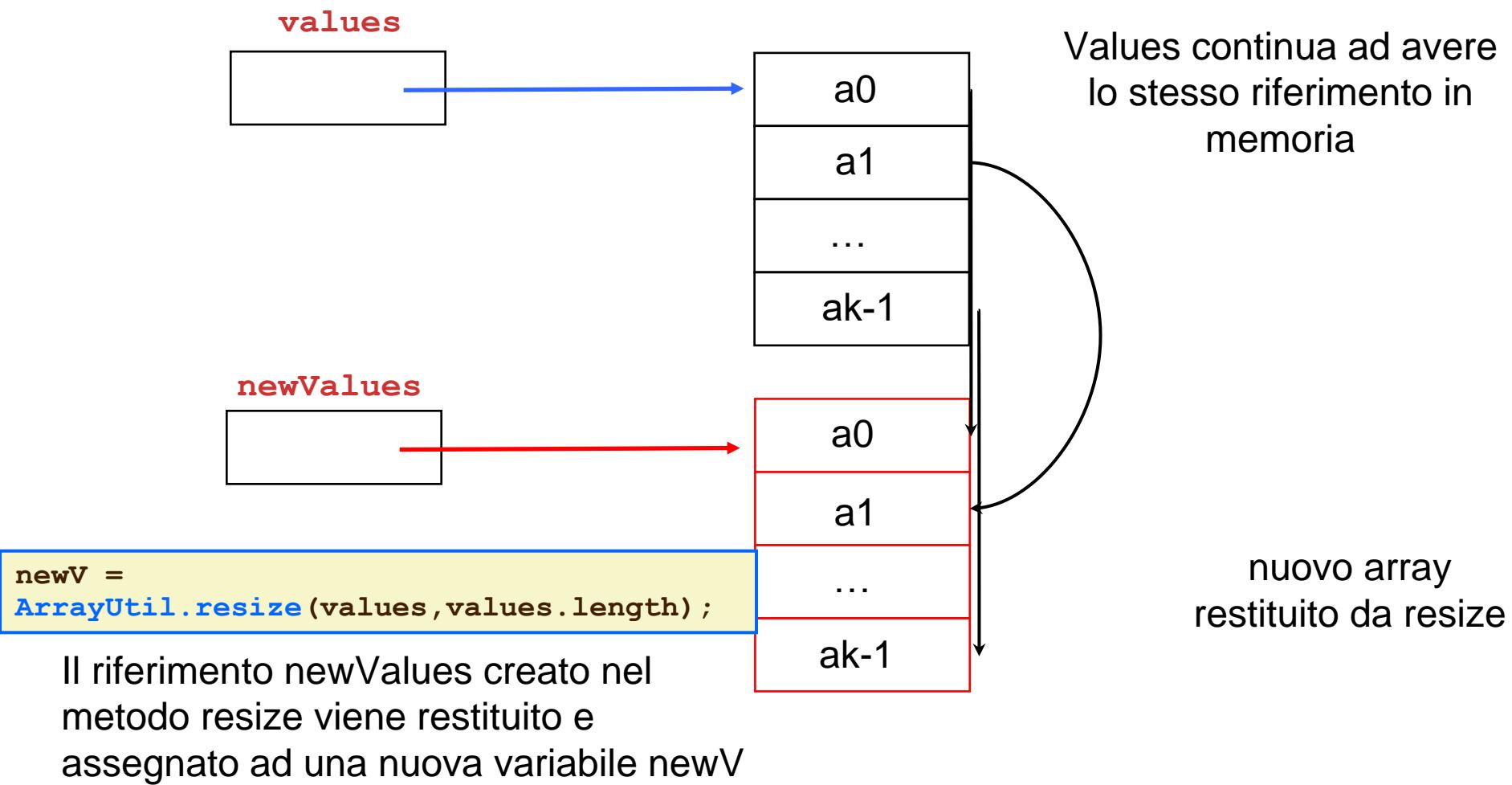


Cambiare dimensione a un array

- Lo stesso metodo può essere usato per **clonare** un intero array; in seguito, è consentito utilizzare sia il nuovo sia il vecchio

```
double[] values = {1, 2.3, 4.5};  
double[] newV = ArrayUtil.resize(values, values.length);  
  
newV[2] = 5.2;  
  
if (values[2] == 4.5) System.out.println("OK");  
  
if (newV[2] == 5.2) System.out.println("OK");
```

Resize





Garbage collector

- Che cosa succede all'array 'abbandonato'?
 - JVM (Java Virtual Machine) provvede a effettuare automaticamente la gestione della memoria (**garbage collection**) durante l'esecuzione di un programma
- Viene considerata memoria libera (quindi riutilizzabile) la memoria eventualmente occupata da oggetti che non abbiano più un riferimento nel programma



Allocazione della memoria in Java

- A ciascun programma java al momento dell'esecuzione viene assegnata un'area di memoria
- Una parte della memoria serve per **memorizzare il codice**; quest'area è statica, ovvero non modifica le sue dimensioni durante l'esecuzione del programma
- In un'area dinamica (ovvero che modifica la sua dimensione durante l'esecuzione) detta **Java Stack** vengono memorizzati **i parametri e le variabili locali dei metodi**



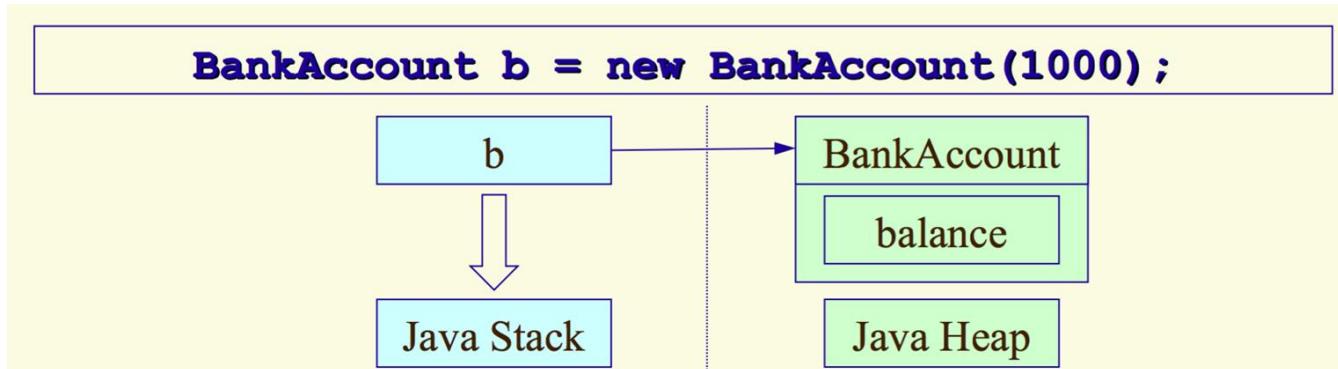
Allocazione della memoria in Java

- Durante l'esecuzione dei metodi di un programma vengono creati dinamicamente oggetti (allocazione dinamica) usando lo speciale operatore new:
 - BankAccount acct = new BankAccount();
 - crea dinamicamente un oggetto di classe BankAccount
- Per l'**allocazione dinamica di oggetti** Java usa un'area di memoria denominata **Java Heap**

Disposizione degli indirizzi di memoria nella JVM



- Le variabili parametro e locali sono memorizzate nel ***java Stack o runtime stack***
- Gli oggetti sono costruiti dall’operatore ***new*** nel ***java Heap***





Array riempito in parte come parametro

- Un metodo che voglia elaborare un array riempito solo in parte deve dichiarare due parametri:
 - l'array
 - la sua dimensione logica, perché quest'ultima non può essere dedotta ispezionando l'array, come invece avviene per la dimensione fisica
 - NB: se passo come parametro v.length elaboro array pieni!

```
public static double sum(double[] v, int vSize)
{ if (vSize > v.length) {
        // qui lanceremo una eccezione!
    }
    double sum = 0;
    for (int i = 0; i < vSize; i++)
        sum = sum + v[i];
    return sum;
}
```



Take home message

- Array: contenitori per dati omogenei
 - ▣ Array di tipi primitivi
 - ▣ Array di oggetti
- Dimensione di un array
 - ▣ Fisica: definita al momento della dichiarazione
 - ▣ Logica: si usa una variabile per tener conto del riempimento dell'array
 - utilizzo come array riempito “in parte”
- Modificare la dimensione di un array
 - ▣ Di fatto si crea un nuovo array e si copiano i dati
 - ▣ Se attribuisco il nuovo riferimento al “vecchio” array è “come se” avessi aumentato la dimensione dell'array