

# Lezione XIV

## Lu 29-Ott-2007

Confrontare oggetti

# Confronto di oggetti

- Abbiamo imparato a confrontare
  - *numeri interi* con gli operatori relazionali

```
if (n == 0 || n > 5)
```

- *numeri in virgola mobile*, confrontandoli con approssimazione

```
final double EPSILON = 1E-14;  
if (Math.abs(x - y) <=  
    EPSILON * Math.max(Math.abs(x), Math.abs(y)))
```

- *stringhe*, mediante i metodi equals() e compareTo()

```
if (s1.equals(s2))
```

```
if (s1.compareTo(s2) < 0)
```

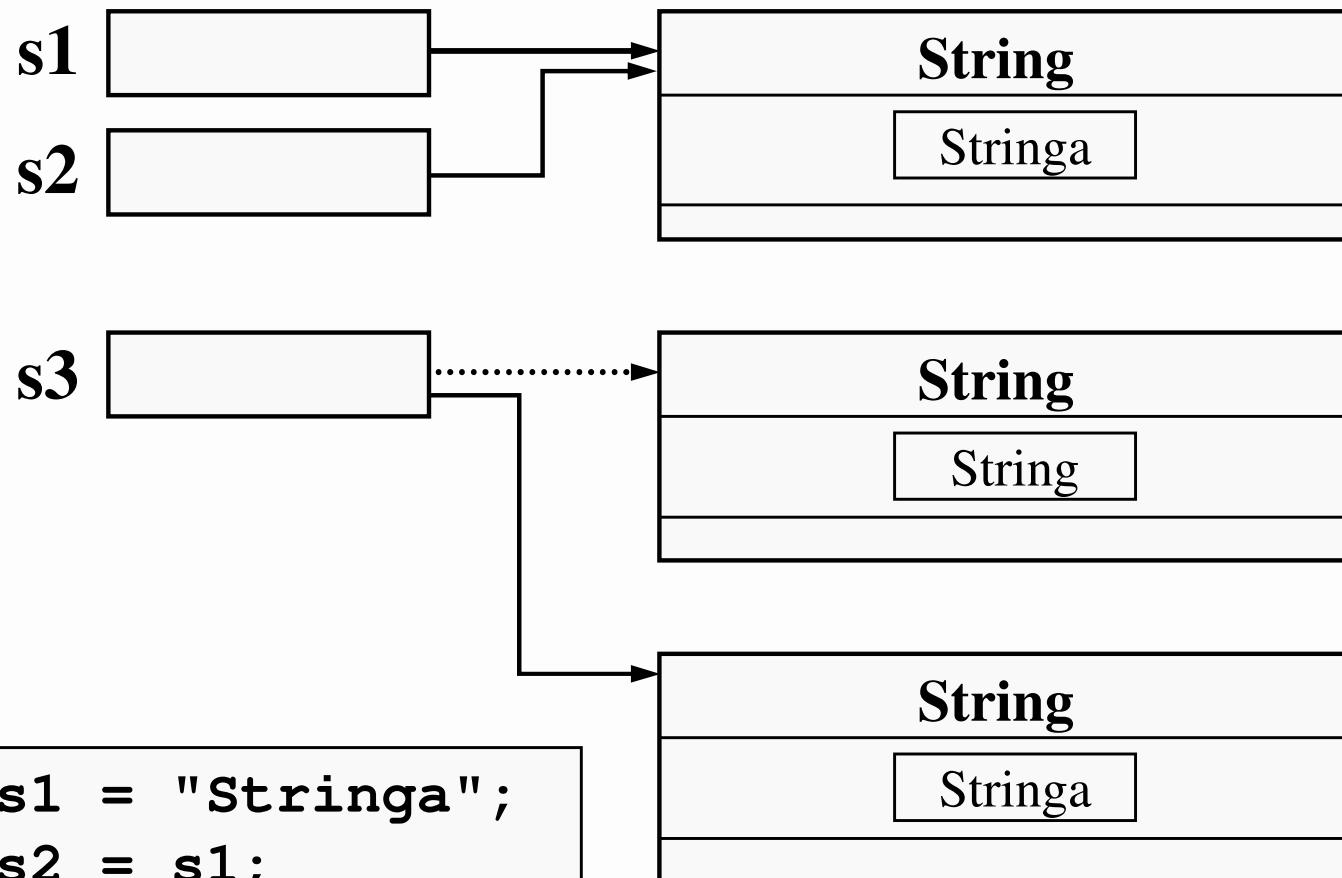
# Confronto di oggetti

- ❑ Confrontando *con l'operatore di uguaglianza* due riferimenti a oggetti si verifica se i riferimenti *puntano allo stesso oggetto*

```
String s1 = "Stringa";  
String s2 = s1;  
String s3 = "String";  
s3 = s3 + "a"; // s3 contiene "Stringa"
```

- ❑ Il confronto `s1 == s2` è *vero*, perché puntano allo stesso oggetto
- ❑ Il confronto `s1 == s3` è *falso*, perché puntano ad oggetti diversi, anche se tali oggetti hanno lo stesso contenuto (sono “identici”)

# Confronto di oggetti



```
String s1 = "Stringa";  
String s2 = s1;  
String s3 = "String";  
s3 = s3 + "a";
```

# Confronto di oggetti

- ❑ Confrontando invece *con il metodo equals()* due riferimenti a oggetti, si verifica se i riferimenti *puntano a oggetti con lo stesso contenuto*

```
String s1 = "Stringa";  
String s2 = s1;  
String s3 = "String";  
s3 = s3 + "a"; // s3 contiene "Stringa"
```

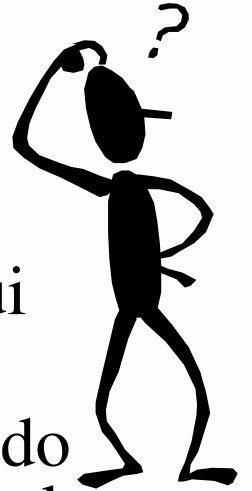
- ❑ Il confronto **s1.equals(s3)** è *vero*, perché puntano a due oggetti **String** identici

---

**Nota:** per verificare se un riferimento si riferisce a **null**, bisogna usare invece l'*operatore di uguaglianza* e non il metodo **equals()**

```
if (s == null)
```

# Confronto di oggetti



- ❑ Il metodo **equals()** può essere applicato a qualsiasi oggetto, perché è definito nella classe **Object**, da cui *derivano* tutte le classi
- ❑ È però compito di ciascuna classe *ridefinire* il metodo **equals()**, come fa la classe **String**, altrimenti il metodo **equals** di **Object** usa semplicemente l'operatore di uguaglianza
- ❑ Il metodo **equals()** di ciascuna classe deve fare il confronto delle caratteristiche (variabili di esemplare) degli oggetti di tale classe
- ❑ **Per il momento, usiamo equals() soltanto per oggetti delle classi della libreria standard**
  - **non facciamo confronti tra oggetti di classi definite da noi**

# Errori comuni

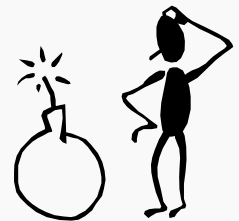
# Un punto e virgola di troppo

- ❑ Attenzione a non inserire, per errore, il carattere ';' dopo gli enunciati if, for e while

```
int n = 10;  
for (int i = 0; i < n; i++); // ahi!!!!  
    System.out.println (i + " ");
```

```
C:\javaExamples>javac TestSemicolon.java  
TestSemicolon.java:6: cannot find symbol  
symbol   : variable i  
location: class TestSemicolon  
    System.out.println(i + " ");
```

1 error



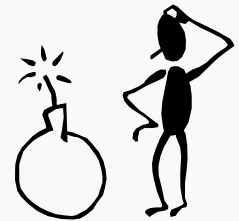


# Un punto e virgola di troppo

❑ In realta' il codice viene interpretato nel modo seguente

```
int n = 10;  
for (int i = 0; i < n; i++)  
    ; // enunciato vuoto: non fa niente  
System.out.println(i + " ");
```

Il ciclo significa:  
Per i da 0 a 9 non fare niente  
Poi stampa i



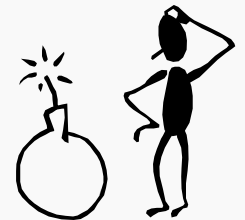
La variabile i non e' visibile  
nell'enunciato System.out.println(...) e  
quindi si ottiene un errore di  
compilazione

# Un punto e virgola di troppo

- ❑ Se definiamo il contatore i al di fuori del ciclo for:

```
int n = 10;  
int i;  
for (i = 0; i < n; i++); //ahi!!!  
    System.out.println (i + " ");
```

Il compilatore non segnala alcun errore, ma in esecuzione il frammento di codice stampa solo 10, ovvero il valore che assume i all'uscita del ciclo



# **Programmare le classi**

## **consigli utili**

# Esercizio: numeri complessi

- ❑ In Java non esiste un tipo di dati fondamentali per elaborare i numeri complessi
- ❑ Non e' fornita neppure una classe nella java platform API per questa funzione
- ❑ Scrivere la classe MyComplex che rappresenta un numero complesso (esercizio del lab. 4)
- ❑ L'interfaccia pubblica della classe è qui.

# Invocare un costruttore da un altro costruttore

```
public MyComplex(double unRe, double unIm)
{
    re = unRe;
    im = unIm;
}
```

```
public MyComplex(double unRe) //inizializza il numero re+i0
{
    this(unRe, 0);
```

Invocazione del costruttore  
MyComplex(double unRe, double unIm)

```
public MyComplex() // inizializza il numero 0+i0
{
    this(0);
```

Invocazione del costruttore  
MyComplex(double unRe)

- ❑ La chiamata al costruttore this(...) deve essere il primo enunciato del costruttore
- ❑ Vantaggio: riutilizzo del codice!

# Assegnare il nome ai parametri dei costruttori

```
private double re; // variabile di esemplare
private double im; // variabile di esemplare
public MyComplex(double re, double im)
{
    re = re;
    im = im;
}
```

**ERRATO!**

???

re e im sono interpretate dal compilatore  
come variabili di esemplare o come  
parametri del metodo???



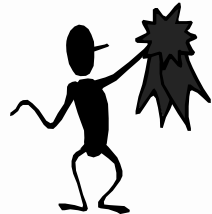
```
public MyComplex(double re, double im)
{
    this.re = re;
    this.im = im;
}
```

**OK**

Il parametro implicito `this` risolve  
l'ambiguità'

```
public MyComplex(double unRe, double unIm)
{
    re = unRe;
    im = unIm;
}
```

**OK**



Stile di programmazione migliore!

# I metodi

```
private double re; // variabile di esemplare
private double im; // variabile di esemplare public
...
/**
```

Calcola la somma  $a + z$  ( $z$  parametro implicito)

@return il numero complesso somma  $a + z$

```
*/
```

Restituisce dato di tipo MyComplex

↓

```
public MyComplex add(MyComplex z)
{
    MyComplex c = new MyComplex(re + z.re, im + z.im);
    return c; // c variabile di appoggio
}
```

Parametro di  
tipo MyComplex

OK

```
public MyComplex add(MyComplex z) // forma piu' coincisa
{
    return new MyComplex(re + z.re, im + z.im);
}
```

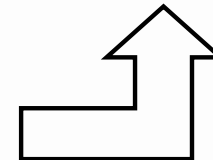


OK

# Uso delle variabili di esemplare

```
public class MyComplex
{
    // variabili di esemplare
    private double re; //parte reale
    private double im; //parte immaginaria
    ...
    public MyComplex add(MyComplex z)
    {
        return new Complex(re + z.re, im + z.im );
    }
}
```

`this.im + z.im`



❑ Solo nei metodi della classe stessa possiamo accedere alla variabile di esemplare **private double re** dell'oggetto **MyComplex z** con la notazione **z.re**



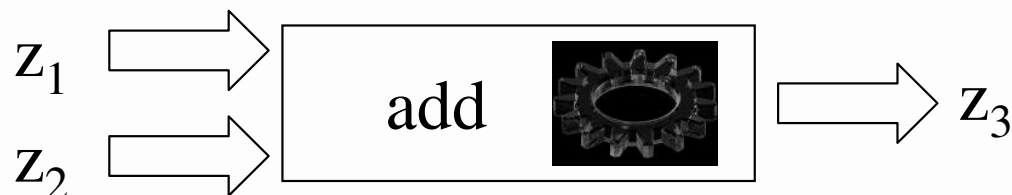
# MyComplex - la somma come metodo di esemplare

```
/**  
    somma due numeri complessi  
    @param z il secondo addendo  
    @return numero complesso pari alla somma di due  
            numeri complessi  
*/  
public MyComplex add(MyComplex z) {...}
```

```
...//z1 e z2 siano due riferimenti a esemplari  
...//della classe MyComplex  
MyComplex z3 = z1.add(z2);
```

parametro implicito

parametro esplicito



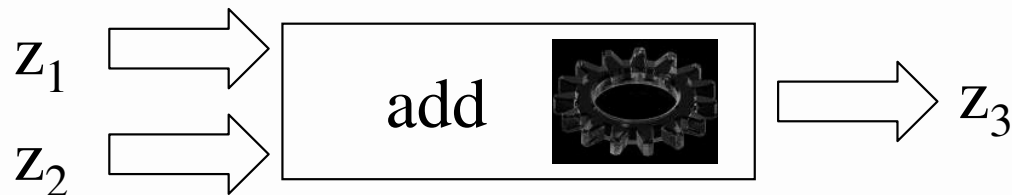
# MyComplex - la somma come metodo statico

```
/**
    somma due numeri complessi - metodo statico
    @param z1 il primo addendo
    @param z2 il secondo addendo
    @return la somma di due numeri complessi
 */
public static MyComplex add(MyComplex z1, MyComplex z2)
{...}
```

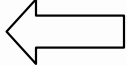
```
...//z1 e z2 siano due riferimenti a esemplari
...//della classe MyComplex
MyComplex z3 = MyComplex.add(z1, z2);
```

Nome della classe

parametri espliciti

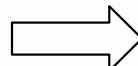


# Il metodo toString()

```
public String toString() 
{
    char sign = '+';
    if (im < 0)
        sign = '-';
    return re + " " + sign + " i" + Math.abs(im);
}
```

Restituisce una stringa

- ❑ Il metodo toString() che abbiamo realizzato restituisce una stringa con una descrizione testuale dello stato dell'oggetto

```
MyComplex z = new MyComplex(1,-2);
String s = z.toString();  "1 - i2"
```

- ❑ A che cosa serve? Ad esempio per stampare il valore di un oggetto

```
MyComplex z = new MyComplex (1,-2);
System.out.println(z.toString());
```

```
MyComplex z = new MyComplex (1,-2);
System.out.println(z);
```

"1 - i2"

# Metodi di accesso e modificatori

- ❑ I metodi della classe MyComplex realizzata accedono agli stati interni degli oggetti (variabili di esemplare) senza mai modificarli

```
// Classe MyComplex  
public MyComplex conj()  
{  
    return new MyComplex(re, -im);  
}
```

**METODO DI ACCESSO**

- ❑ Altre classi hanno metodi che modificano gli stati degli oggetti

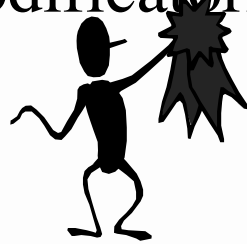
```
// Classe BankAccount  
public void deposit(double amount)  
{  
    balance += amount;  
}
```

**METODO MODIFICATORE**

# Metodi di Accesso e Modificatori

## ❑ Raccomandazione

- Ai metodi modificatori assegnare, generalmente, un valore di ritorno **void**



## ❑ Classi IMMUTABILI: si definiscono *immutabili* le classi che hanno solo metodi accessori

- la classe esempio `MyComplex` e' immutabile
- la classe `java.lang.String` e' immutabile

## ❑ I riferimenti agli oggetti delle classi immutabili possono essere distribuiti e usati senza timore che venga alterata l'informazione contenuta negli oggetti stessi

## ❑ Non cosi' per gli oggetti delle classi non immutabili:

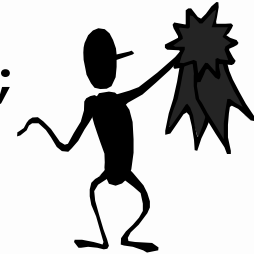
```
BankAccount mioConto = new BankAccount(1000);  
...  
mioConto.withdraw(1000);
```

# Metodi predicativi

- ❑ Metodi **predicativi** sono metodi che restituiscono un valore booleano
- ❑ Esempio: si scriva un metodo per confrontare due numeri complessi

```
public boolean equals(MyComplex z)
{
    return (re == z.re) && (im == z.im);
}
```

*Stile di programmazione migliore!*



```
public boolean equals(MyComplex z)
{
    if (re == z.re && im == z.im)
        return true;
    else
        return false;
}
```

OK

```
public boolean equals(MyComplex z)
{
    if (re == z.re && im == z.im)
        return true;
    return false;
}
```

OK

# Usare metodi privati

- ❑ Nelle classi possiamo programmare anche metodi privati
- ❑ Potranno essere invocati solo all'interno dei metodi della classe stessa
- ❑ Si fa quando vogliamo isolare una funzione precisa che, in genere, viene richiamata più volte nel codice
- ❑ Esempio: eseguiamo il confronto fra due numeri complessi ammettendo una tolleranza  $\epsilon$  sulle parti reale e immaginaria (sono numeri double!)

```
...
public boolean approxEquals(MyComplex z)
{
    return approx(re, z.re) && approx(im, z.im);
}

private static boolean approx(double x, double y)
{
    final double EPSILON = 1E-14; // tolleranza

    return Math.abs(x-y) <= EPSILON *
        Math.max(Math.abs(x), Math.abs(y));
}
```

# Passaggio di parametri



# Accesso alle variabili di esemplare

```
// metodo di altra classe  
MyComplex a = new MyComplex(1,2);  
MyComplex b = new MyComplex(2,3);  
MyComplex c = a.add(b);  
...
```

```
// classe MyComplex  
public MyComplex add(MyComplex z)  
{  
    this  
    this.re  
    return new MyComplex(re + z.re, im + z.im);  
}
```

```
graph TD
    this[this] --> this_re[this.re]
    this --> this_im[this.im]
    this_re --> new_re[re]
    this_im --> new_im[im]
    z[MyComplex z] --> z_re[z.re]
    z --> z_im[z.im]
```

# Parametri formali ed effettivi

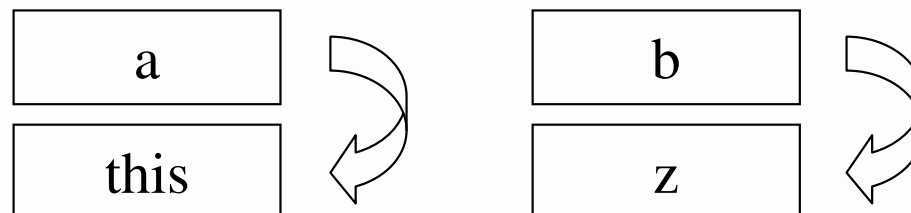
- ❑ I parametri espliciti che compaiono *nell'intestazione* dei metodi e il parametro implicito *this* (usati nella realizzazione dei metodi) si dicono *Parametri Formali* del metodo

```
public MyComplex add(MyComplex z)
{
    return new MyComplex(this.re + z.re, this.im + z.im);
}
```

- ❑ I parametri forniti *nell'invocazione* ai metodi si dicono *Parametri Effettivi* del metodo

```
MyComplex a = new MyComplex(1, 2);
MyComplex b = new MyComplex(2, 3);
MyComplex c = a.add(b);
```

- ❑ Al momento dell'esecuzione dell'invocazione del metodo, i **parametri effettivi** sono **copiati** nei **parametri formali**



# Modificare parametri numerici

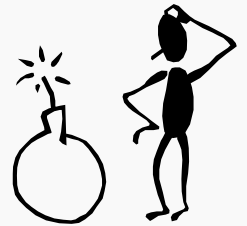
- ❑ Proviamo a scrivere un metodo **increment** che ha il compito di fornire un nuovo valore per una variabile di tipo numerico

```
// si usa con x = increment1(x)
public static int increment1(int index)
{   return index + 1;
} // è equivalente a x++
```

OK

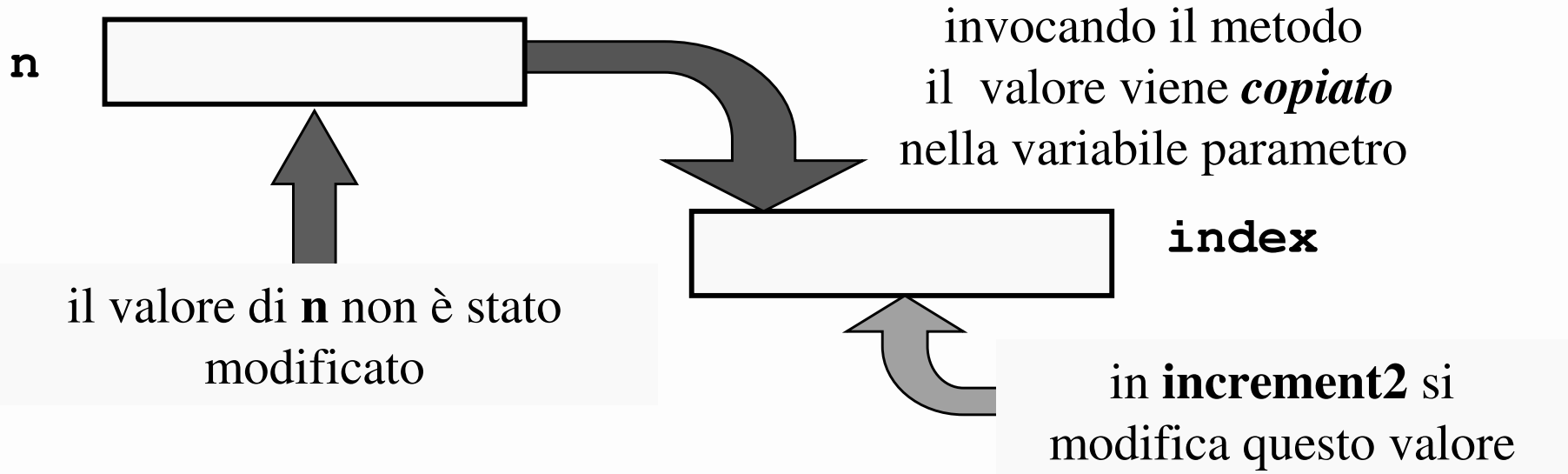
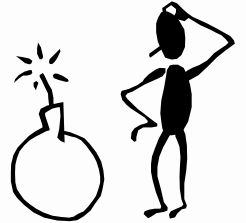
- ❑ Perché non abbiamo scritto semplicemente così?

```
// si usa con increment2(x)
public static void increment2(int index)
{   index = index + 1;
} // NON MODIFICA X!!!
```



# Modificare parametri numerici

```
// si usa con increment2(n)
public static void increment2(int index)
{   index = index + 1;
} // NON FA NIENTE !!
```



```
int n = 1;
increment2(n);
System.out.println("n =" + n);
```

⇒ **n = 1**

# Modificare parametri oggetto

- ❑ Un metodo può invece modificare lo *stato* di un *oggetto* passato come parametro (implicito o esplicito, **deposit()**...)

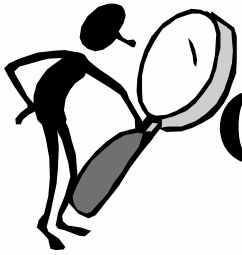
```
// classe BankAccount
// trasferisce denaro da un conto ad un altro
public void transfer(BankAccount to, double amount)
{
    withdraw(amount);    // ritira da un conto
    to.deposit(amount);   // deposita nell'altro conto
} // FUNZIONA !!
```

- ❑ ma non può modificare il *riferimento* contenuto nella variabile oggetto che ne costituisce il parametro effettivo

```
// NON FUNZIONA
public static void swapAccounts(BankAccount x,
                                BankAccount y)
{
    BankAccount temp = x;
    x = y;
    y = temp;
}
```

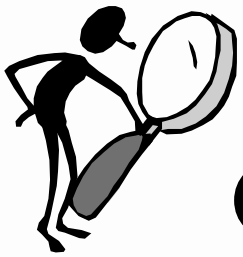
```
swapAccounts(a, b);
// nulla è successo alle
// variabili a e b
```





# Chiamate per valore e per riferimento

- ❑ In Java, il passaggio dei parametri è effettuato “per valore”, cioè il *valore* del *parametro effettivo* viene copiato nel *parametro formale*
  - *questo impedisce che il valore contenuto nella variabile che costituisce il parametro effettivo possa essere modificato*
- ❑ Altri linguaggi di programmazione (come C++) consentono di effettuare il passaggio dei parametri con altri meccanismi, ad esempio “per riferimento”, rendendo possibile la modifica dei parametri effettivi



# Chiamate per valore e per riferimento

- ❑ A volte si dice, *impropriamente*, che in Java *i numeri sono passati per valore e che gli oggetti sono passati per riferimento*
- ❑ In realtà, *tutte le variabili sono passate per valore*, ma
  - passando per valore una variabile oggetto, si passa una copia del riferimento in essa contenuto
  - l'effetto “pratico” del passaggio per valore di un riferimento a un oggetto è la possibilità di modificare lo stato dell'oggetto stesso, come avviene con il passaggio “per riferimento”

# **Lezione XV**

## **Ma 30-Ott-2007**

**Stili di programmazione  
sovrascrivere i parametri**



# Modifica di variabili parametro

- ❑ Le variabili parametro di un metodo sono variabili a tutti gli effetti e possono essere trattate come le altre variabili, quindi si puo' anche riassegnare alle variabili un valore

```
public void deposit(double amount)
{
    amount = balance + amount;
    ...
}
```

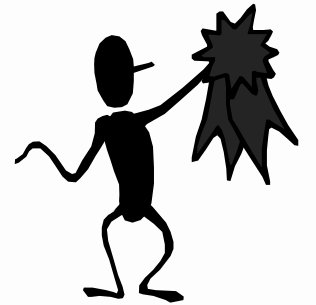


- ❑ Se nel codice successivo all'enunciato di assegnazione si usa la variabile **amount**, il suo valore non e' piu' quello del parametro effettivo passato al metodo!
- ❑ Questo e' fonte di molti errori, soprattutto in fase di manutenzione del codice
- ❑ Non modificate mai i valori dei parametri dei metodi!!!

# Modifica di variabili parametro

- ❑ Meglio usate in alternativa una variabile locale in piu'

```
public void deposit(double amount)
{
    double tmpBalance = balance + amount;
    ...
    ...
}
```



# Scomposizione di stringhe

# Scomposizione di stringhe

- ❑ Una *sottostringa con caratteristiche sintattiche ben definite* (ad esempio, delimitata da spazi...) si chiama *token*
  - Es.: nella stringa “uno due tre” sono identificabili i token “uno”, “due”, “tre”
- ❑ Un problema comune nella programmazione e' la scomposizione di stringhe in token
- ❑ Per questa funzione è molto utile la classe **Scanner**, del package **java.util** che già' conosciamo per la lettura da standard input
- ❑ **Scanner** considera come delimitatori predefiniti gli spazi, i caratteri di tabulazione e i caratteri di “andata a capo”
  - Questi e altri caratteri sono detti *whitespaces* e sono riconosciuti dal metodo predicativo:  
*Character.isWhitespace(char c)*

# Scomposizione di stringhe

- ❑ Per usare **Scanner**, innanzitutto bisogna creare un oggetto della classe fornendo la stringa come parametro al costruttore

```
String line = "uno due tre";  
Scanner st = new Scanner(line);
```

- ❑ In generale non e' noto a priori il numero di token presenti nella stringa
- ❑ Successive invocazioni del metodo **next()** restituiscono successivi token

# Scomposizione di stringhe

- ❑ Il metodo `next ()` della classe `Scanner` lancia l'eccezione `java.util.NoSuchElementException` nel caso non ci siano piu' token nella stringa (non molto comodo!)

```
String line = "uno due tre";  
Scanner st = new Scanner(line);  
  
String token1 = st.next(); // "uno"  
String token2 = st.next(); // "due"  
String token3 = st.next(); // "tre"  
String token4 = st.next();
```

`java.util.NoSuchElementException`



# Scomposizione di stringhe

- ❑ Per questo prima di invocarlo si verifica la presenza di eventuali token per mezzo del *metodo predicativo* **hasNext ()**, che ritorna un dato di tipo boolean di valore **true** se e' presente nella stringa un token successivo non ancora estratto, **false** altrimenti.
- ❑ Il metodo **hasNext ()** e' molto comodo quando non conosciamo a priori il numero di token presenti nella stringa da scomporre

```
while (st.hasNext ())  
{   String token = st.next ();  
    ...// elabora token  
}
```

# Scomposizione di stringhe

- ❑ Alla prima invocazione, il metodo hasNext() riconosce che nella stringa e' presente un token successivo non ancora acquisito (il token **“uno”**) e quindi ritorna **true**



**“uno due tre”**

- ❑ Viene quindi eseguito il corpo del ciclo in cui il metodo next() acquisisce il token **“uno”**. Si noti che la stringa line rimane immutata.
- ❑ Alla successiva invocazione, il metodo hasNext() riconosce che nella stringa e' presente un token successivo non ancora acquisito (il token **“due”**) e quindi ritorna **true**



**“uno due tre”**



# Scomposizione di stringhe

- ❑ Viene eseguito nuovamente il corpo del ciclo e il metodo `next()` acquisisce il token **“due”**.
- ❑ Alla successiva invocazione, il metodo `hasNext()` riconosce che nella stringa e' presente un token successivo non ancora acquisito (il token **“tre”**) e quindi ritorna **true**



**“uno due tre”**

- ❑ Viene eseguito il corpo del ciclo e il metodo `next()` acquisisce il token **“tre”**.
- ❑ Alla successiva invocazione, il metodo `hasNext()` riconosce che nella stringa non e' presente un token successivo non ancora acquisito e quindi ritorna **false**



- ❑ Termina il ciclo **while**

**“uno due tre”**

# Metodi predicativi della classe Scanner nella lettura da standard input

- ❑ Quando il programmatore non sa *quanti saranno* i dati forniti in ingresso dall'utente, abbiamo imparato a usare i valori sentinella
- ❑ Nell'esempio il carattere 'Q' indica la fine della sequenza di dati

```
Scanner in = new Scanner(System.in);
int sum = 0;
boolean done = false;
while (!done)
{
    String line = in.next();
    if (line.equalsIgnoreCase("Q"))
        done = true;
    else
        sum += Integer.parseInt(line);
}
```

# Metodi predicativi della classe Scanner nella lettura da standard input

- ❑ Anche nella lettura da standard input e' utile usare i metodi predicativi della classe Scanner

```
while (st.hasNext())  
{   String token = st.next();  
    // elabora token  
}
```

- ❑ Il questo caso il metodo hasNext() interrompe l'esecuzione e attende l'immissione dei dati da standard input
- ❑ I dati a standard input sono acquisiti dal programma quando l'operatore preme il tasto invio
- ❑ **hasNext()** agisce sui dati in ingresso come spiegato precedentemente per la scomposizione di stringhe

# Metodi predicativi della classe Scanner nella lettura da standard input

- ❑ Esempio: sia il programma in attesa di dati da standard input in un frammento di codice come il seguente

```
while (st.hasNext())  
{   String token = st.next();  
    // elabora token  
}
```

- ❑ L'esecuzione del programma e' ferma nel metodo **hasNext()** che attende l'immissione dei dati
- ❑ Quando l'operatore inserisce dei dati e preme invio:

```
$ uno due tre <ENTER>
```

- ❑ La stringa **"uno due tre\n"** viene inviata allo standard input

# Metodi predicativi della classe Scanner nella lettura da standard input

❑ Al termine dell'elenco dei dati si può *comunicare al sistema operativo* che l'input da standard input destinato al programma in esecuzione è terminato

- in una finestra DOS/Windows bisogna digitare **Ctrl+C**
- in una *shell* di Unix bisogna digitare **Ctrl+D**

❑ All'introduzione di questi caratteri speciali il metodo **hasNext()**, la cui esecuzione era interrotta in attesa di immissione di dati, restituisce **false**

❑ Il ciclo **while** di lettura termina

# Esempio: conta parole

```
import java.util.Scanner;

public class WordCounter
{   public static void main(String[] args)
    {
        Scanner c = new Scanner(System.in);
        int count = 0;
        while (c.hasNext())
        {
            c.next();    // estrae il token!!!
            count++;
        }

        System.out.println(count + " parole");
        c.close();
    }
}
```

# Altri metodi predicativi

- ❑ Analogamente al metodo **hasNext()** nella classe `Scanner` sono definiti metodi predicativi per ciascun tipo fondamentale di dato, ad esempio
  - **hasNextInt()**
  - **hasNextDouble()**
  - **hasNextLong()**
  - ...
- ❑ E' definito anche il metodo **hasNextLine()** utile per leggere righe
- ❑ Lo useremo nei casi in cui vogliamo preservare la struttura per righe dei dati

# Java.util.StringTokenizer

- ❑ La classe `java.util.Scanner` è stata introdotta nella versione 1.5 di Java
- ❑ Nelle versioni di Java  $< 1.5$  la scomposizione in token di una stringa viene effettuata mediante la classe *`java.util.StringTokenizer`* che mette a disposizione i seguenti metodi
  - `boolean hasMoreTokens()`
  - `String nextToken()`

```
import java.util.StringTokenizer;
...
StringTokenizer st = new StringTokenizer(s);
while (st.hasMoreTokens())
{
    String token = st.nextToken();
    // elabora token
}
```



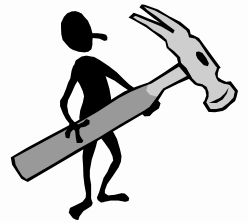
# Reindirizzamento di standard input e output

# Calcolare la somma di numeri

```
import java.util.Scanner;
public class Sum
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);

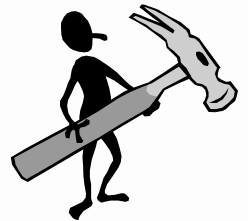
        double sum = 0;
        while (in.hasNextDouble())
        {
            sum = sum + in.nextDouble();
        }
        System.out.println("Somma: " + sum);
    }
}
```

# Redirezione di input e output



- ❑ Usando il precedente programma **Sum**, si inseriscono dei numeri da tastiera, che al termine non vengono memorizzati
  - *per sommare una serie di numeri, bisogna digitarli tutti, ma non ne rimane traccia!*
- ❑ Una soluzione “logica” sarebbe che *il programma leggesse i numeri da un file*
  - questo si può fare con la redirezione **dello standard input**, consentita da quasi tutti i sistemi operativi

# Redirezione di input e output

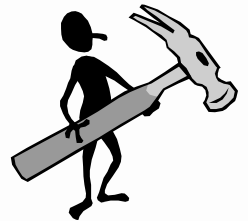


- ❑ La redirezione dello standard input, sia nei sistemi Unix sia nei sistemi MS Windows, si indica con il carattere **<** seguito dal **nome del file da cui ricevere l'input**

```
$ java Sum < input.txt
```

- ❑ Si dice che il file **input.txt** viene *collegato* all'input standard
- ❑ Il programma non ha bisogno di alcuna istruzione particolare, semplicemente **System.in** non sarà più collegato alla tastiera ma al file specificato

# Redirezione di input e output



- ❑ A volte è comoda anche la redirazione dell'output
  - ad esempio, quando il programma produce molte righe di output, che altrimenti scorrono velocemente sullo schermo senza poter essere lette

```
java Sum > output.txt
```

- ❑ Le due redirezioni possono anche essere combinate

```
java Sum < input.txt > output.txt
```

# Array

# Problema

- ❑ Scrivere un programma che
  - legge dallo standard input una sequenza di dieci numeri in virgola mobile, uno per riga
  - chiede all'utente un numero intero **index** e visualizza il numero che nella sequenza occupava la posizione indicata da **index**
- ❑ Occorre *memorizzare tutti i valori della sequenza*
- ❑ Potremmo usare dieci variabili diverse per memorizzare i valori, selezionati poi con una lunga sequenza di alternative, *ma se i valori dovessero essere mille?*

# Memorizzare una serie di valori

- ❑ Lo strumento messo a disposizione dal linguaggio Java per memorizzare una sequenza di dati si chiama *array* (che significa “sequenza ordinata”)
  - la struttura *array* esiste in quasi tutti i linguaggi di programmazione
- ❑ Un array in Java è *un oggetto* che realizza *una raccolta di dati che siano tutti dello stesso tipo*
- ❑ Potremo avere quindi array di *numeri interi*, array di numeri in virgola mobile, array di stringhe, array di conti bancari...



# Costruire un array

- ❑ Come ogni *oggetto*, un array 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**
  - un **riferimento** a un esemplare di una **classe**

e nella costruzione deve essere 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];  
values = new double[100];
```

# 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*

- **0** per le variabili numeriche
- **false** per le variabili di tipo boolean
- **null** per i riferimenti

- ❑ Per *accedere* a un elemento dell'array si usa

```
double[] values = new double[10];  
double oneValue = values[3];
```

- ❑ La stessa sintassi si usa per *modificare* un elemento dell'array

```
double[] values = new double[10];  
values[5] = 3.4;
```

# Utilizzare un array

```
double[] values = new double[10];  
double oneValue = values[3];  
values[5] = 3.4;
```

- ❑ Il numero intero 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)

# Utilizzare un array

- 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, 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 dell'incapsulamento...

# La dimensione di un array

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

- ❑ In realtà, **length** è una variabile pubblica ma è dichiarata **final**, quindi *non può essere modificata*, può soltanto essere ispezionata
- ❑ Questo paradigma è, in generale, considerato accettabile nell'OOP
- ❑ L'alternativa sarebbe stata fornire un metodo pubblico per accedere alla variabile privata
  - la soluzione 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 in = new Scanner(System.in);

    System.out.println("Inserisci 10" +
      " numeri, uno per riga");

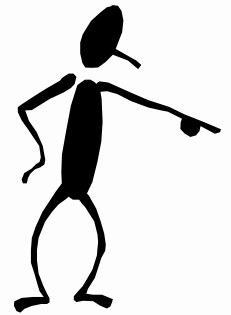
    double[] values = new double[10];
    for (int i = 0; i < values.length; i++)
      if (in.hasNextDouble())
        values[i] = in.nextDouble();

    System.out.print("Inserisci un indice: ");
    int index = 0;
    if (in.hasNextInt())
      index = in.nextInt();

    if (index < 0 || index >= values.length)
      System.out.println("Valore errato");
    else
      System.out.println("valore: " + 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

# Riferimento ad un array

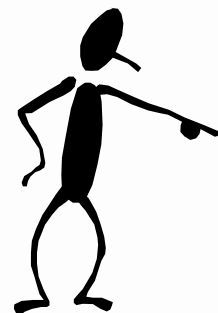
- ❑ Sintassi:



```
NomeTipo[] nomeRiferimento;
```

- ❑ Scopo: definire la variabile ***nomeRiferimento*** come variabile oggetto che potrà contenere un riferimento a un array di dati di tipo ***NomeTipo***
- ❑ Nota: ***NomeTipo*** può essere uno dei tipi fondamentali di Java o il nome di una classe
- ❑ Nota: le parentesi quadre [ ] sono necessarie e ***non*** devono contenere l'indicazione della dimensione dell'array

# Accesso a un elemento di un array



❑ Sintassi:

```
referimentoArray[indice]
```

- ❑ Scopo: accedere all'elemento in posizione *indice* all'interno dell'array a cui *referimentoArray* 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**

# Lezione XVI

## Me 31-Ott-2007

array  
continua

# 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...

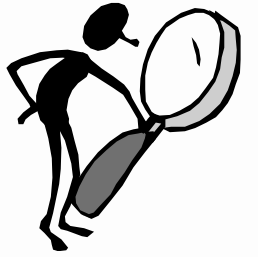
```
double[] values = new double[10];  
values[10] = 2; // ERRORE IN ESECUZIONE
```

```
double[] values = new double[10];  
for (int i = 0; i <= values.length; i++)  
{   int k = values[i]; // ERRORE IN ESECUZIONE  
}
```

- ❑ Come abbiamo visto, l'ambiente runtime segnala questo errore con un'eccezione che arresta il programma

ArrayIndexOutOfBoundsException

# 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*)

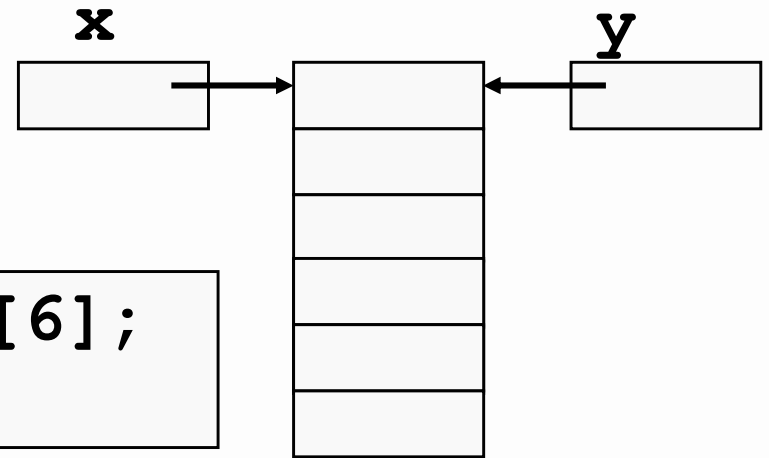
```
int[] primes = {2, 3, 5};
```

oppure (*accettabile, ma meno chiara*)

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

# Copiare un array

- ❑ Ricordando che 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*

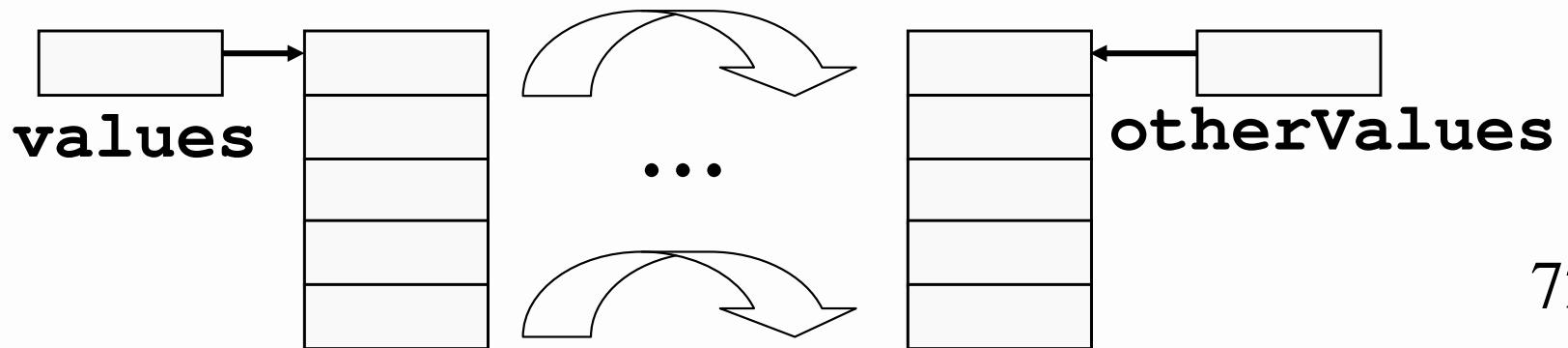


```
double[] x = new double[6];  
double[] y = x;
```

# Copiare un array

- ❑ Se si vuole ottenere *una copia dell'array*, 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[5];  
// inseriamo i dati nell'array  
...  
  
double[] otherValues = new double[values.length];  
for (int i = 0; i < values.length; i++)  
    otherValues[i] = values[i];
```





# Copiare un array

Attenzione alla minuscola!

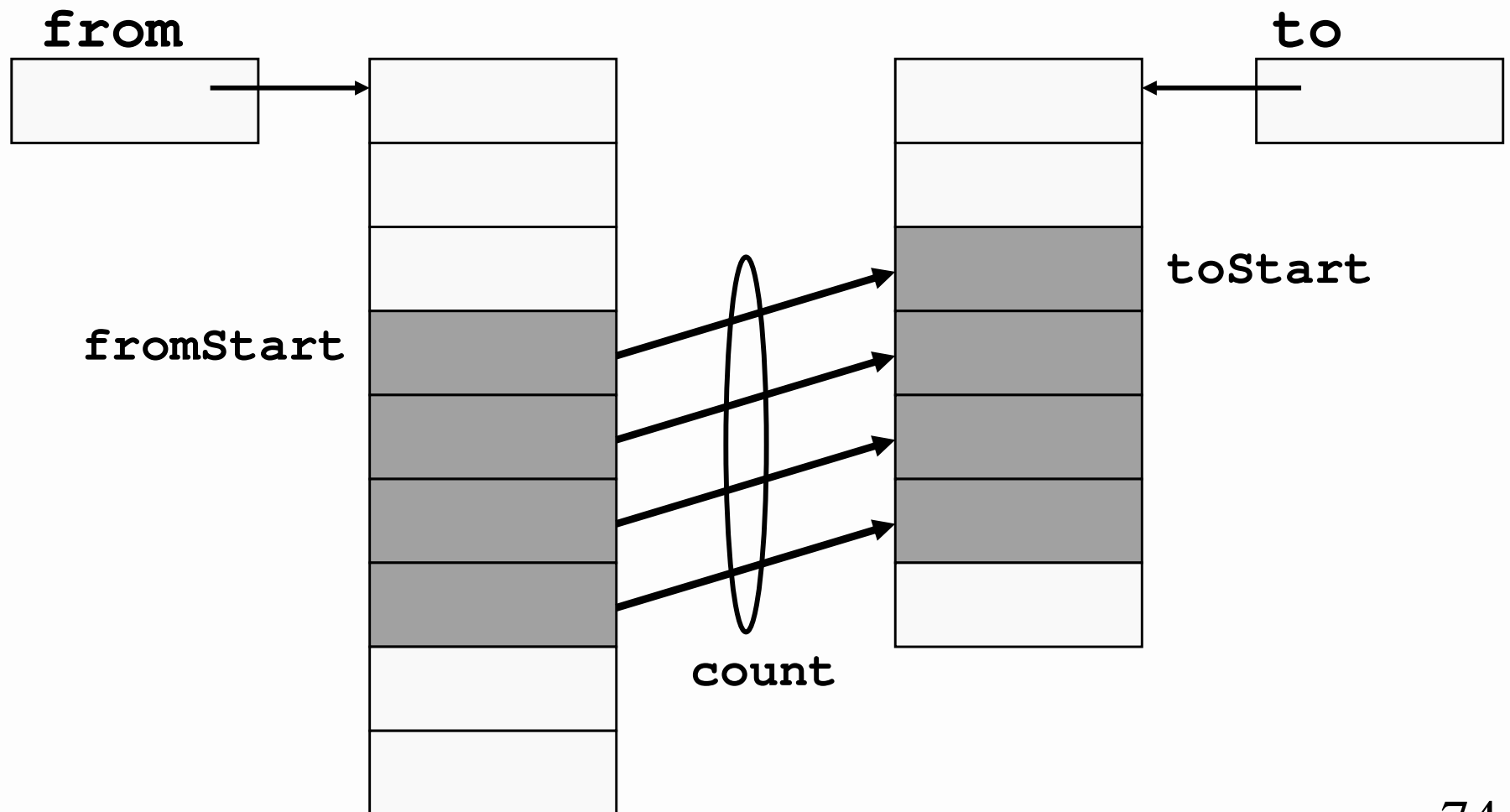
- ❑ Invece di usare un ciclo, è possibile (e *più efficiente*) invocare il metodo statico **arraycopy()** della classe **System** (nel pacchetto **java.lang**)

```
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 un 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) ;
```



# Passare un array come parametro

- ❑ Spesso si scrivono metodi che ricevono array come parametri espliciti

```
private static double sum(double[] values)
{
    double sum = 0;
    for (int i = 0; i < values.length; i++)
    {
        double e = values[i];
        sum = sum + e;
    }
    return sum;
}
```

# Array come dato di ritorno

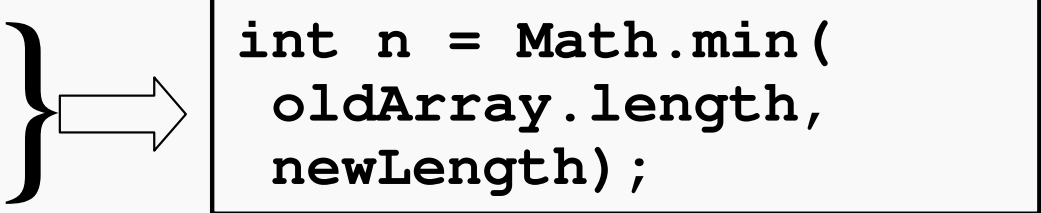
- ❑ Un metodo può anche usare un array come valore di ritorno

```
private static double[] resize(double[] oldArray, int
    newLength)
{
    double[] newArray = new double[newLength];

    int n = oldArray.length;
    if (newLength < n)
        n = newLength;

    for (int i = 0; i < n; i++)
        newArray[i] = oldArray[i];

    return newArray;
}
```



```
double[] values = {1, 2.3, 4.5};
values = resize(values, 5);
values[3] = 5.2;
// nella stessa classe
```

1	2.3	4.5
---	-----	-----

1	2.3	4.5	5.2	0
---	-----	-----	-----	---

# Ciclo for generalizzato

- ❑ Spesso l'elaborazione richiede di scandire tutti gli elementi di un array
- ❑ Esempio: somma degli elementi di un array di numeri

```
private static double sum(double[] values)
{
    double sum = 0;
    for (int i = 0; i < values.length; i++)
    {
        double e = values[i];
        sum = sum + e;
    }
    return sum;
}
```

# Ciclo for generalizzato

❑ In questo caso si puo' usare il ciclo **for generalizzato**

```
private static double sum(double[] values)
{
    double sum = 0;
    for (double e: values)
        sum = sum + e;
    return sum;
}
```

```
private static double accountSum(BankAccount[] v)
{
    double sum = 0;
    for (BankAccount e: v)
        sum = sum + e.getBalance();
    return sum;
}
```

# Ciclo for generalizzato

- ❑ Il ciclo for generalizzato va usato quando si vogliono scandire tutti gli elementi dell'array nell'ordine dal primo all'ultimo
- ❑ Se invece si vuole scandire solo un sottoinsieme
  - ad esempio non si parte dal primo elemento
- ❑ oppure si vuole scandire in ordine inverso, **si deve usare un ciclo ordinario**

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

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

```
System.out.println("***ORDINE DIRETTO ***");  
for (int e : data)  
    System.out.print(e + " ");
```

\$ 0 1 2 3 4 5 6 7 8 9

```
System.out.println("\n***ORDINE INVERSO ***");  
for (int i = data.length - 1; i >= 0; i--)  
    System.out.print(data[i] + " ");
```

\$ 9 8 7 6 5 4 3 2 1 0

**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 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)
  - 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

- ❑ Come risolvere il problema?
  - *per costruire un array è necessario indicarne la dimensione*, che è una sua proprietà **final**
    - *gli array in Java non possono crescere!*
- ❑ Una possibile soluzione consiste nel costruire un array di *dimensioni sufficientemente grandi* da poter accogliere una sequenza di dati di lunghezza “ragionevole”, cioè tipica per il problema in esame
- ❑ 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’ultimo indice nell’array che contiene dati validi*

# Array riempiti solo in parte

```
final int ARRAY_LENGTH = 1000;
final String END_OF_DATA = ""; //Sentinella

double[] values = new double[ARRAY_LENGTH];
int valuesSize = 0;

Scanner in = new Scanner(System.in);

while (in.hasNextLine()) // un numero per riga
{
    String token = in.nextLine();
    if (token.equals(END_OF_DATA))
        break;

    values[valuesSize] = Double.parseDouble(token);
    valuesSize++;
}
... // continua
```

# Array riempiti solo in parte

```
...
/*
    valuesSize è l'indice del primo dato
    non valido
*/
System.out.print("Inserisci un indice: ");

int index = in.nextInt();

if (index < 0 || index >= valuesSize)
    System.out.println("Valore errato");
else
    System.out.println("v = " + values[index]);
```

# Array riempiti solo in parte

- ❑ **values.length** è il numero di valori *memorizzabili*, **valuesSize** è il numero di valori *memorizzati*
- ❑ 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**
- ❑ 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

```
// impedisce l'inserimento di troppi dati
...
Scanner in = new Scanner(System.in);

while (in.hasNextLine())    // un numero per riga
{
    String token = in.nextLine();
    if (token.equals(END_OF_DATA))
        break;
    if (valuesSize >= values.length)
    {
        System.out.println("Troppi dati");
        break;
    }
    values[valuesSize] = Double.parseDouble(token);
    valuesSize++;
}
...
```

# Cambiare dimensione a un array


- ❑ Abbiamo già visto come 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), copiarne il contenuto e abbandonarlo, usando poi quello nuovo (si parla di *array dinamico*)

Raddoppiare la dimensione!

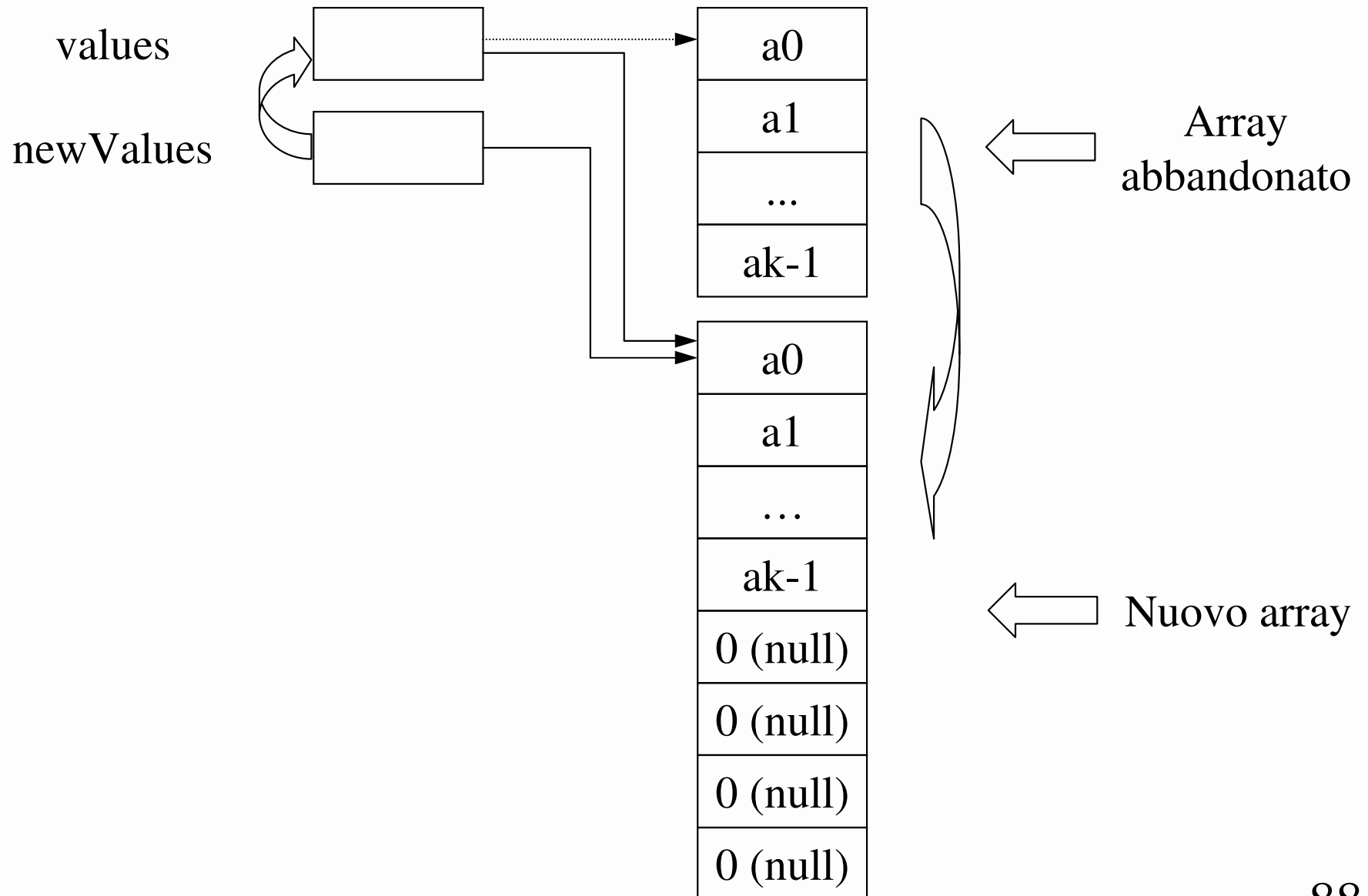
```
if (valuesSize >= values.length)
{
    //definiamo un nuovo array di dim. doppia
    double[] newValues = new double[2 * values.length];

    // ricopiamo gli elementi nel nuovo array
    for (int i = 0; i < values.length; i++)
        newValues[i] = values[i];

    //assegniamo a value il riferimento a newValue
    values = newValues;
    // valuesSize non cambia
}
```



# Cambiare dimensione a un array





# Garbage Collector

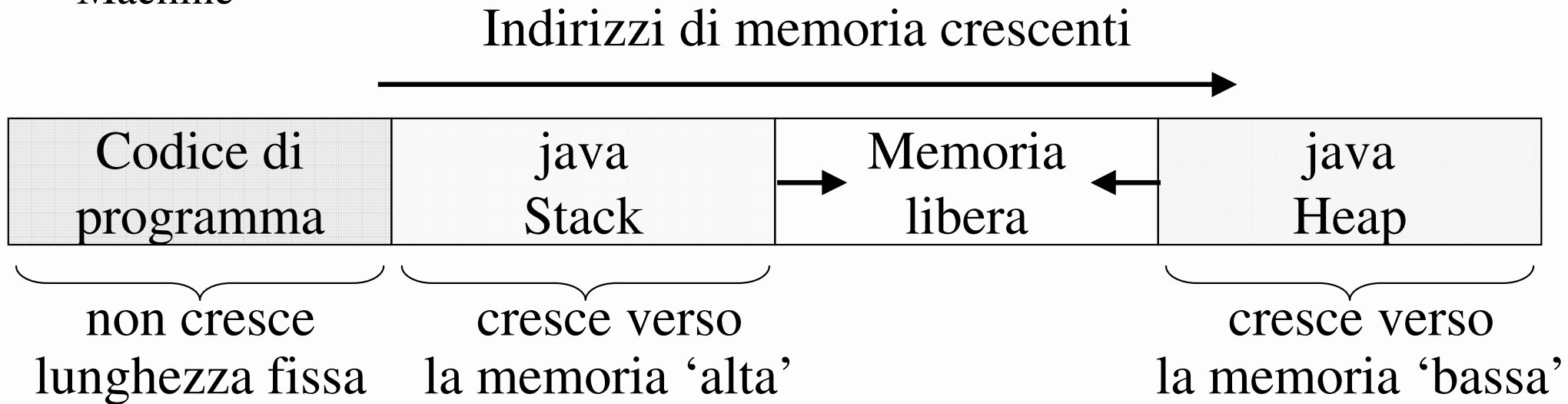
- ❑ *Raccoglitore di rifiuti*
- ❑ Che cosa succede all'array '*abbandonato*'?
- ❑ Se un programma abbandona molti dati (ad esempio cambiando spesso le dimensioni di grandi array puo' esaurire la memoria a disposizione?)
- ❑ JVM (Java Virtual Machine) provvede a effettuare *automaticamente* la *gestione 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 piu' 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*
- ❑ 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**

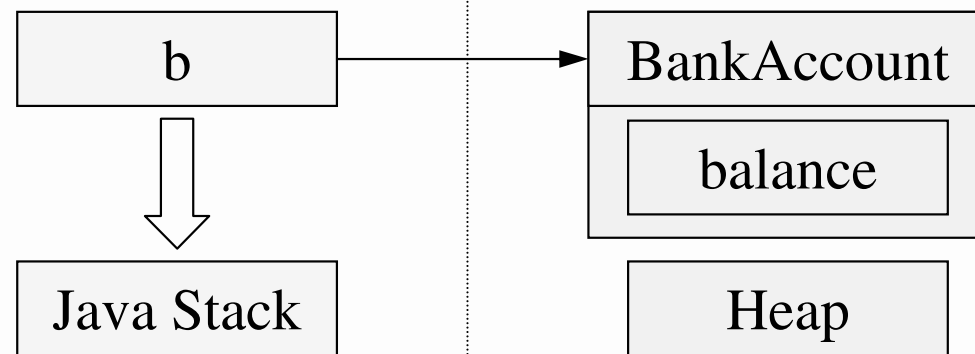
# Allocazione della memoria in Java

- ❑ Schema della disposizione degli indirizzi di memoria nella Java Virtual Machine



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

```
BankAccount b = new BankAccount(1000);
```



# Semplici algoritmi degli array


# Trovare un valore particolare

- ❑ Un tipico algoritmo consiste nella *ricerca all'interno dell'array di (almeno) un elemento avente determinate caratteristiche*

```
double[] values = {1, 2.3, 4.5, 5.6};
double target = 3; // valore da cercare

int index = 0;
boolean found = false;
while (index < values.length && !found)
{
    if (values[index] == target)
        found = true;
    else
        index++;
}
if (found)
    System.out.println(index);
```

valuesSize; /\*se riempito  
solo in parte \*/



- ❑ Trova soltanto il primo valore che soddisfa la richiesta
- ❑ La variabile **found** è necessaria perché la ricerca potrebbe avere esito negativo

# Trovare il valore minimo

- ❑ Un altro algoritmo tipico consiste nel *trovare l'elemento con il valore minore (o maggiore) tra quelli presenti nell'array*

```
double[] values = {1, 2.3, 4.5, 5.6};
```

```
double min = values[0];
```


```
for (int i = 1; i < values.length; i++)
```

```
    if (values[i] < min)
```

```
        min = values[i];
```

```
System.out.println(min);
```

valuesSize; //se riempito  
//solo in parte



- ❑ In questo caso bisogna *esaminare sempre l'intero array*

# Eliminare un elemento

- L'eliminazione di un elemento da un array richiede due algoritmi diversi
  - *se l'ordine tra gli elementi dell'array non è importante* (cioè se l'array realizza il concetto astratto di *insieme*), allora *è sufficiente copiare l'ultimo elemento dell'array nella posizione dell'elemento da eliminare e ridimensionare l'array* (oppure usare la tecnica degli array riempiti soltanto in parte)

```
double[] values = {1, 2.3, 4.5, 5.6};  
int indexToRemove = 0;  
values[indexToRemove] =  
    values[values.length - 1];  
values = resize(values, values.length - 1);
```

# Eliminare un elemento

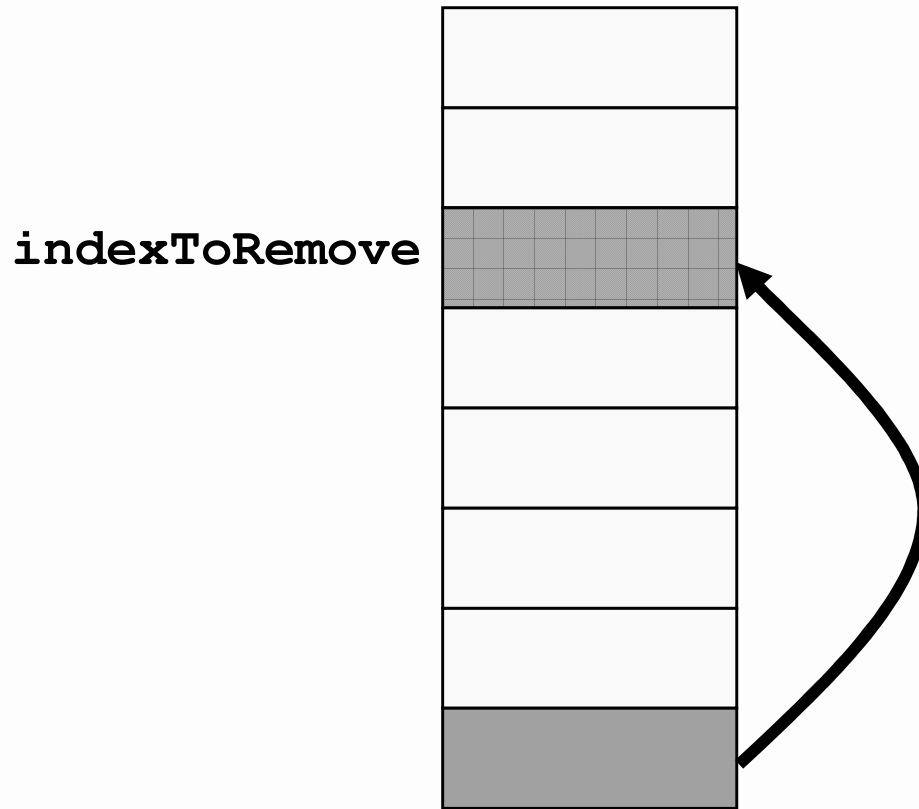
- ❑ *Se l'ordine tra gli elementi deve, invece, essere mantenuto*, l'algoritmo è più complesso
  - *tutti gli elementi il cui indice è maggiore dell'indice dell'elemento da rimuovere devono essere spostati nella posizione con indice immediatamente inferiore*, per poi ridimensionare l'array (oppure usare la tecnica degli array riempiti soltanto in parte)

```
double[] val = {1, 2.3, 4.5, 5.6};  
  
int indexToRemove = 0;  
for (int i = indexToRemove; i < val.length - 1; i++)  
    val[i] = val[i + 1];  
  
val = resize(val, val.length - 1);
```

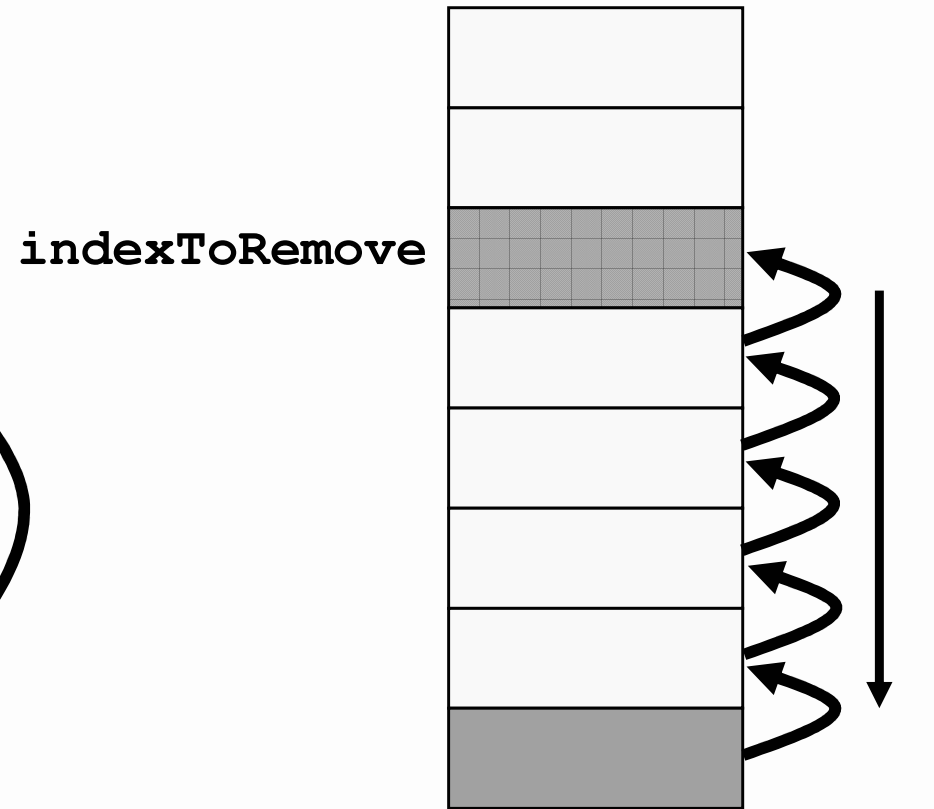


# Eliminare un elemento

Senza ordinamento



Con ordinamento



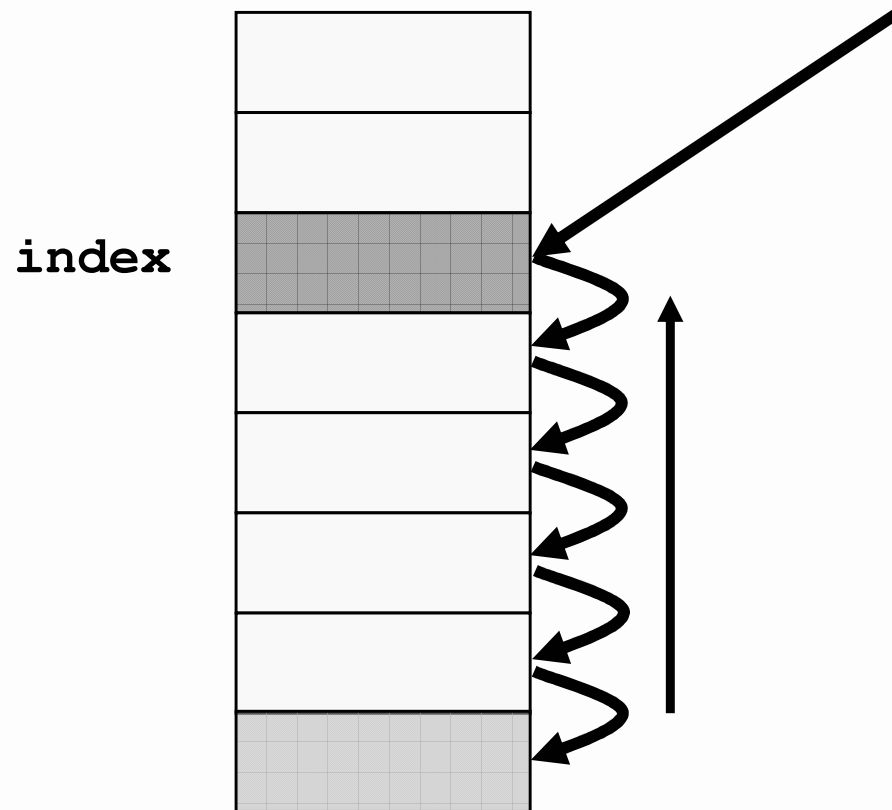
i trasferimenti vanno  
eseguiti dall'alto in basso!

# Inserire un elemento

- ❑ Per inserire un elemento in un array nella posizione voluta, se questa non è la prima posizione libera, bisogna “fargli spazio”
  - *tutti gli elementi il cui indice è maggiore dell'indice della posizione voluta devono essere spostati nella posizione con indice immediatamente superiore, a partire dall'ultimo elemento dell'array*

```
double[] val = {1, 2.3, 4.5, 5.6};  
  
val = resize(val, val.length + 1);  
  
int index = 2;  
for (int i = val.length - 1; i > index; i--)  
    val[i] = val[i-1];  
val[index] = 5.4;
```

# Inserire un elemento



i trasferimenti vanno  
eseguiti dal basso in alto!