

# Capitolo 7

## Vettori e array

**Cay S. Horstmann**  
**Concetti di informatica e fondamenti di Java**  
**quarta edizione**

# Obiettivi del capitolo

---

- Acquisire familiarità con l'utilizzo di array e vettori
- Studiare le classi involucro, la tecnica di *auto-impacchettamento* e il ciclo `for` generalizzato
- Apprendere gli algoritmi più comuni per gli array
- Capire come usare array bidimensionali
- Imparare a scegliere array o vettori nei vostri programmi
- Realizzare array riempiti solo in parte
- Apprendere il concetto di collaudo regressivo

# Array

---

- Array: sequenza di valori omogenei (cioè dello stesso tipo).
- Costruire un array:

```
new double[10]
```

- Memorizzare in una variabile il riferimento all'array.
- Il tipo di una variabile che fa riferimento a un array è il tipo dell'elemento.
- Dichiarazione di una variabile array

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

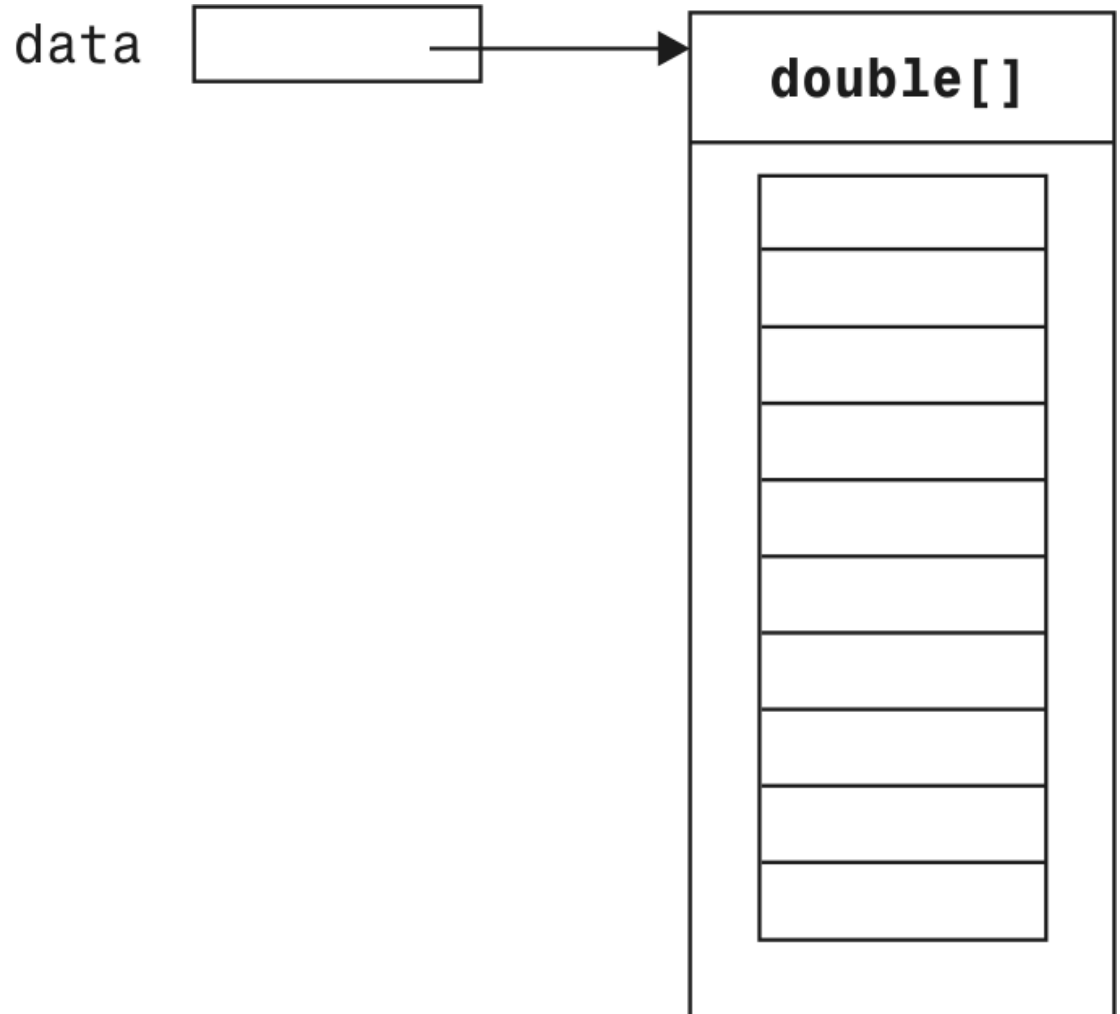
**Segue**

# Array

---

- Nel momento in cui viene creato l'array, tutti i suoi valori sono inizializzati al valore
  - 0 (per un array di numeri come `int[]` o `double[]`),
  - `false` (per un array `boolean[]`),
  - `null` (per un array di riferimenti a oggetti).

# Array

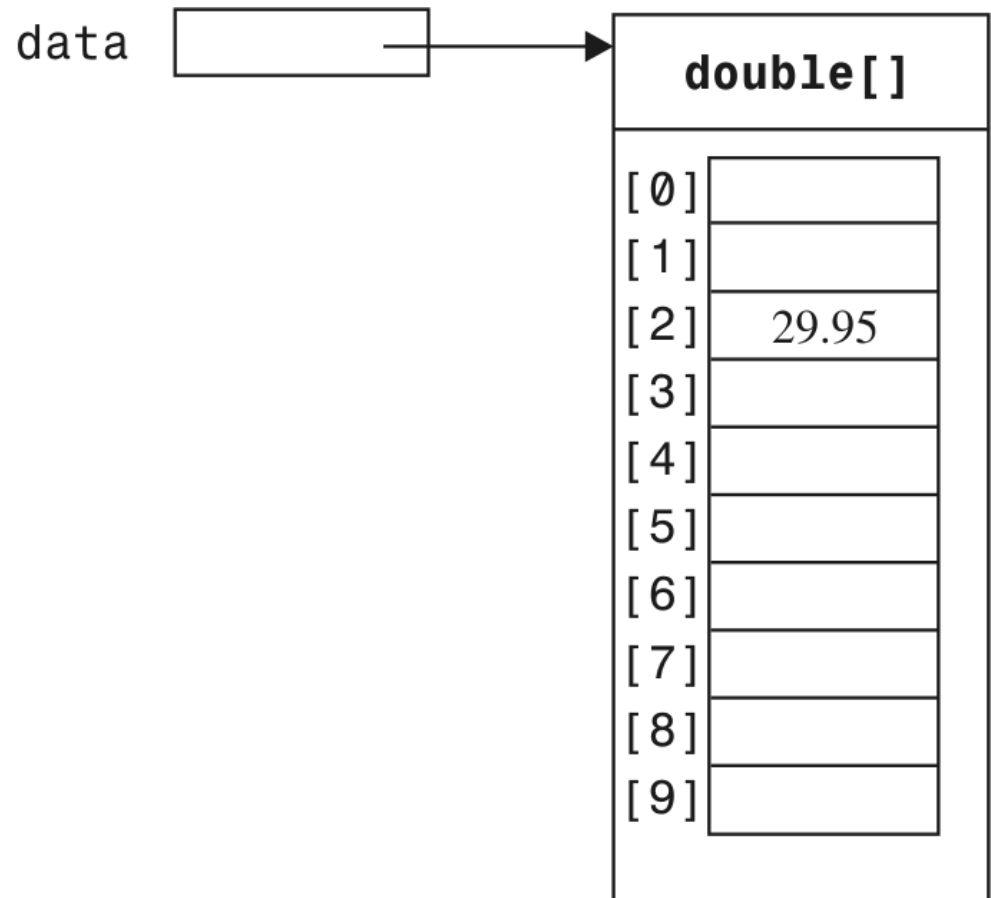


**Figura 1:**  
Un riferimento ad array  
e un array

# Array

- Usare [ ] per identificare un elemento di un array

```
data[2] = 29.95;
```



**Figura 2**  
Memorizzare un valore  
in un Array

# Array

- Si accede agli elementi di un array tramite un indice di tipo intero, usando la notazione a [i].

```
System.out.println("The value of this data item is " + data[2]);
```

- I valori per gli indici di un array vanno da 0 a  $\text{length} - 1$ .  
L'accesso a un elemento non esistente provoca il lancio di un'eccezione per errori di limiti.
- Per conoscere il numero di elementi di un array usare il campo `length`.
  - Es: `data.length`
- Gli array hanno un limite pesante: *la loro lunghezza è fissa*.

# Sintassi 7.1: Costruzione di array

---

```
new nomeTipo[lunghezza]
```

## **Esempio:**

```
new double[10]
```

## **Obiettivo:**

Costruire un array con un determinato numero di elementi



# Sintassi 7.2: Accesso a elementi di array

---

*referimentoAdArray[indice]*

**Esempio:**

data[2]

**Obiettivo:**

Accedere a un elemento di un array

# Vettori

- La classe `ArrayList` (vettore o lista sequenziale) gestisce oggetti disposti in sequenza.
- Un vettore può crescere e calare di dimensione in base alle necessità
- La classe `ArrayList` fornisce metodi per svolgere le operazioni più comuni, come l'inserimento e la rimozione di elementi
- La classe `ArrayList` è una classe generica: `ArrayList<T>` contiene oggetti di tipo `T`.

```
ArrayList<BankAccount> accounts = new ArrayList<BankAccount>();  
BankAccount account1 = new BankAccount(1001);  
accounts.add(account1);  
accounts.add(new BankAccount(1015));  
accounts.add(new BankAccount(1022));
```

- Il metodo `size()` restituisce la dimensione attuale del vettore

# Ispezionare gli elementi

- Per ispezionare gli oggetti contenuti nel vettore si usa il metodo `get` e **non** l'operatore `[ ]` (tranne che in Java 7)
- Come con gli array, i valori degli indici iniziano da 0
- Ad esempio, `accounts.get(2)` restituisce il conto bancario avente indice 2, cioè il terzo elemento del vettore:

```
BankAccount anAccount = accounts.get(2);  
    // fornisce il terzo elemento del vettore
```

- Accedere a un elemento non esistente è un errore.

```
int i = accounts.size();  
anAccount = accounts.get(i); // Errore  
// gli indici validi vanno da 0 a i-1
```

# Aggiungere elementi

- Per assegnare un nuovo valore a un elemento di un vettore già esistente si usa il metodo `set`:

```
BankAccount anAccount = new BankAccount(1729);  
accounts.set(2, anAccount);
```

- È possibile inserire un oggetto in una posizione intermedia all'interno di un vettore.

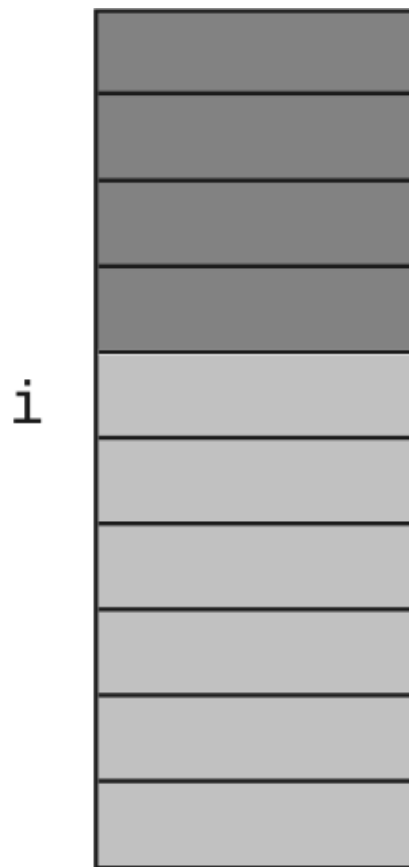
```
accounts.add(i, c)
```

- L'invocazione `accounts.add(i, c)` aggiunge l'oggetto `c` nella posizione `i` e sposta tutti gli elementi di una posizione, a partire dall'elemento attualmente in posizione `i` fino all'ultimo elemento presente nel vettore.

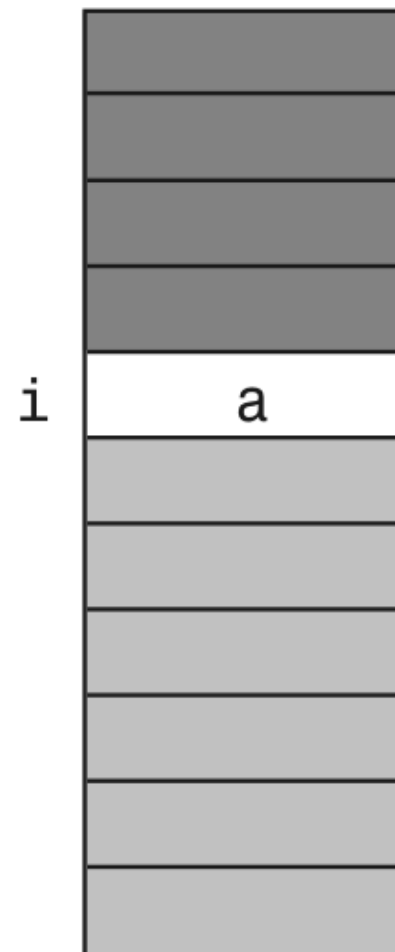
# Aggiungere elementi

**Figura 3**

Aggiungere un elemento  
in una posizione intermedia  
di un vettore



Prima



Dopo

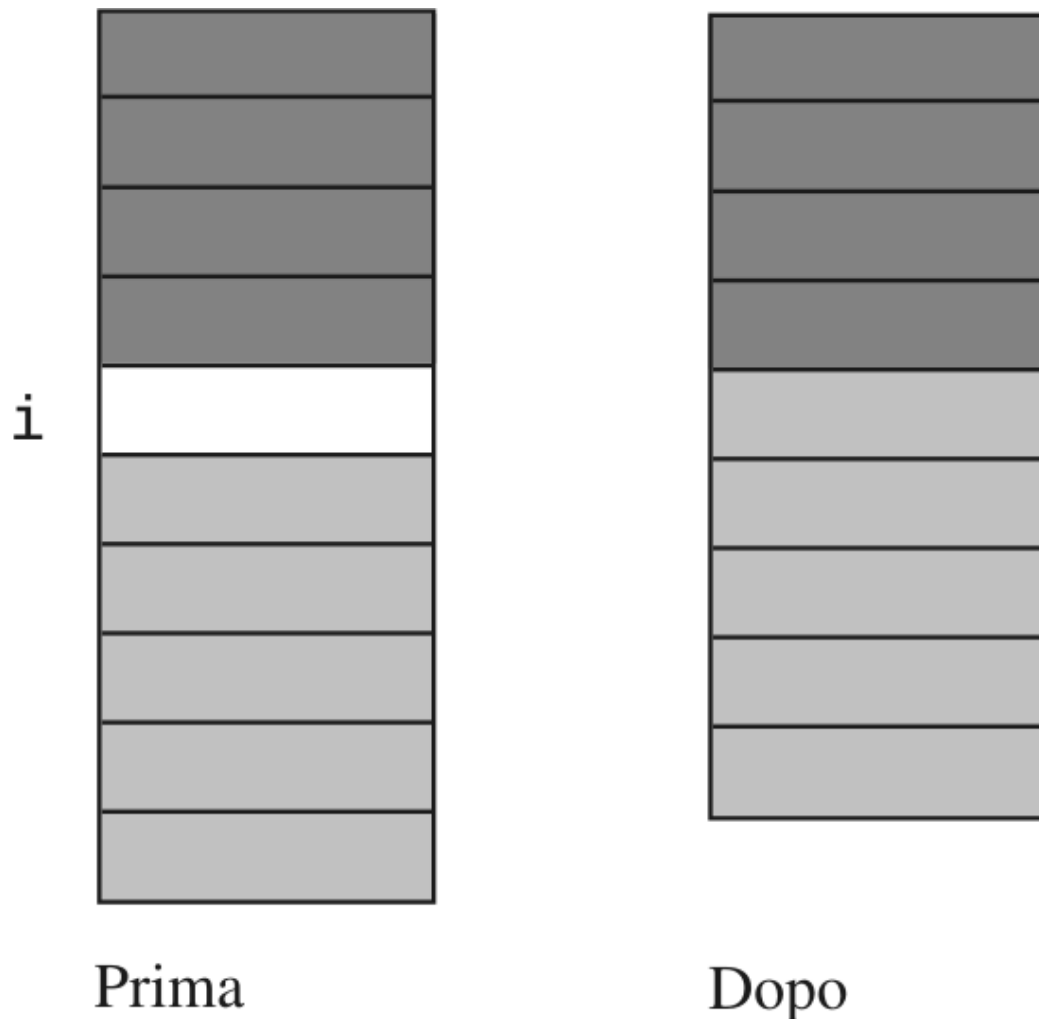
# Rimuovere elementi

## Figura 4

Rimuovere un elemento da una posizione intermedia di un vettore

L'invocazione

`accounts.remove(i)`  
elimina l'elemento che si trova in posizione  $i$ , sposta di una posizione all'indietro tutti gli elementi che si trovano dopo l'elemento rimosso e diminuisce di uno la dimensione del vettore.



# File: ArrayListTester.java

```
01: import java.util.ArrayList;
02:
03: /**
04:     Questo programma collauda la classe ArrayList.
05: */
06: public class ArrayListTester
07: {
08:     public static void main(String[] args)
09:     {
10:         ArrayList<BankAccount> accounts
11:             = new ArrayList<BankAccount>();
12:         accounts.add(new BankAccount(1001));
13:         accounts.add(new BankAccount(1015));
14:         accounts.add(new BankAccount(1729));
15:         accounts.add(1, new BankAccount(1008));
16:         accounts.remove(0);
```

**Segue**

# File: ArrayListTester.java

```
17:
18:     System.out.println("size=" + accounts.size());
19:     BankAccount first = accounts.get(0);
20:     System.out.println("first account number="
21:         + first.getAccountNumber());
22:     BankAccount last = accounts.get(accounts.size() - 1);
23:     System.out.println("last account number="
24:         + last.getAccountNumber());
25: }
26: }
```



# File: BankAccount.java

```
01: /**
02:     Un conto bancario ha un saldo che può essere modificato
03:     da depositi e prelievi.
04: */
05: public class BankAccount
06: {
07:     /**
08:         Costruisce un conto bancario con saldo uguale a zero.
09:         @param anAccountNumber il numero di questo conto bancario
10:     */
11:     public BankAccount(int anAccountNumber)
12:     {
13:         accountNumber = anAccountNumber;
14:         balance = 0;
15:     }
16:
```

**Segue**

# File: BankAccount.java

```
17:     /**
18:         Costruisce un conto bancario con un saldo assegnato.
19:         @param anAccountNumber il numero di questo conto bancario
20:         @param initialBalance il saldo iniziale
21:     */
22:     public BankAccount(int anAccountNumber, double initialBalance)
23:     {
24:         accountNumber = anAccountNumber;
25:         balance = initialBalance;
26:     }
27:
28:     /**
29:         Restituisce il numero di conto del conto bancario.
30:         @return il numero di conto
31:     */
32:     public int getAccountNumber()
33:     {
34:         return accountNumber;
35:     }
```

**Segue**

# File: BankAccount.java

```
36:
37:     /**
38:         Versa denaro nel conto bancario.
39:         @param amount l'importo da versare
40:     */
41:     public void deposit(double amount)
42:     {
43:         double newBalance = balance + amount;
44:         balance = newBalance;
45:     }
46:
47:     /**
48:         Preleva denaro dal conto bancario.
49:         @param amount l'importo da prelevare
50:     */
51:     public void withdraw(double amount)
52:     {
53:         double newBalance = balance - amount;
54:         balance = newBalance;
```

**Segue**

# File: BankAccount.java

```
55:     }
56:
57:     /**
58:      * Ispeziona il valore del saldo attuale del conto bancario.
59:      * @return il saldo attuale
60:      */
61:     public double getBalance()
62:     {
63:         return balance;
64:     }
65:
66:     private int accountNumber;
67:     private double balance;
68: }
```

# File: BankAccount.java

---

## Output:

```
Size: 3
```

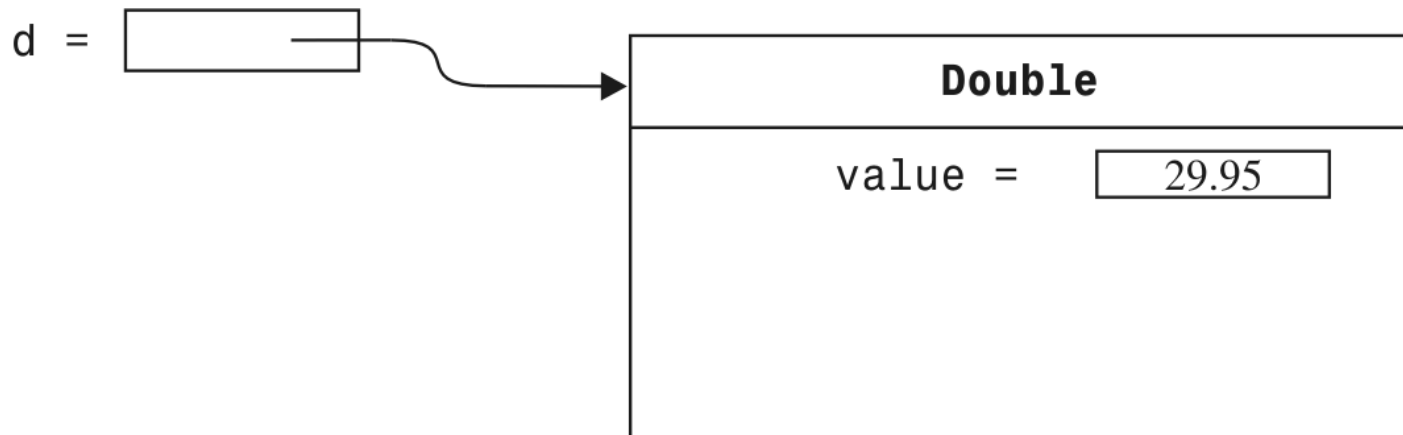
```
First account number: 1008
```

```
Last account number: 1729
```

# Involucri

- Non si possono inserire valori di tipo primitivo direttamente nei vettori
- Per poter manipolare valori di tipo primitivo come se fossero oggetti si usano le classi involucro.

```
ArrayList<Double> data = new ArrayList<Double>();  
data.add(29.95);  
double x = data.get(0);
```



**Figura 5**  
Esemplare  
di una classe involucro

# Involucri

- Esistono classi involucro per tutti gli otto tipi di dati primitivi

<b>Tipo primitivo</b>	<b>Classe involucro</b>
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

# Auto-impacchettamento

- A partire dalla versione 5.0 di Java, la conversione tra tipi primitivi e le corrispondenti classi involucro è automatica

```
Double d = 29.95; // auto-boxing
           // come se fosse Double d = new Double(29.95);
double x = d; // auto-unboxing
           // come se fosse double x = d.doubleValue();
```

**Segue**



# Auto-boxing

- Le conversioni automatiche funzionano anche all'interno di espressioni aritmetiche.
- L'enunciato

```
Double e = d + 1;
```

è valido e significa:

- Converti `d` in un valore di tipo `double`
- Aggiungi 1
- Impacchetta il risultato in un nuovo oggetto di tipo `Double`
- Memorizza in `e` il riferimento all'oggetto involucro appena creato

# Il ciclo `for` generalizzato

- Il ciclo `for` generalizzato scandisce tutti gli elementi di una raccolta:

```
double[] data = . . .;
double sum = 0;
for (double e : data) // si legge "per ogni e in data"
{
    sum = sum + e;
}
```

**Segue**

# Il ciclo `for` generalizzato

- Per scandire tutti gli elementi di un array non è obbligatorio utilizzare il ciclo `for` generalizzato: lo stesso ciclo può essere realizzato con un `for` normale e una variabile indice esplicita.

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

# Il ciclo `for` generalizzato

- Il ciclo `for` generalizzato può essere usato anche per ispezionare tutti gli elementi di un vettore.  
Ad esempio, il ciclo seguente calcola il saldo totale di tutti i conti bancari:

```
ArrayList<BankAccount> accounts = . . . ;
double sum = 0;
for (BankAccount a : accounts)
{
    sum = sum + a.getBalance();
}
```

- Il ciclo è equivalente a questo ciclo “normale”:

```
double sum = 0;
for (int i = 0; i < accounts.size(); i++)
{
    BankAccount a = accounts.get(i);
    sum = sum + a.getBalance();
}
```

# Sintassi 8.3: Il ciclo `for` generalizzato

```
for (Tipo variabile : raccolta)  
    enunciato
```

## **Esempio:**

```
for (double e : data)  
    sum = sum + e;
```

## **Obiettivo:**

Eseguire un ciclo avente un'iterazione per ogni elemento appartenente a una raccolta. All'inizio di ciascuna iterazione viene assegnato alla variabile l'elemento successivo della raccolta, poi viene eseguito l'enunciato.

# Semplici algoritmi per vettori

## Contare valori aventi determinate caratteristiche

- Per contare i valori aventi determinate caratteristiche e presenti in un vettore, ispezionare tutti gli elementi e contare quelli che rispondono ai requisiti, finché non si raggiunge la fine del vettore.

```
public class Bank
{
    public int count(double atLeast)
    {
        int matches = 0;
        for (BankAccount a : accounts)
        {
            if (a.getBalance() >= atLeast) matches++;
            // Trovato
        }
        return matches;
    }
    . . .
    private ArrayList<BankAccount> accounts;
}
```

# Semplici algoritmi per vettori

## Trovare un valore

- Per trovare un valore in un vettore occorre controllarne tutti gli elementi finché non si trova il valore cercato.

```
public class Bank
{
    public BankAccount find(int accountNumber)
    {
        for (BankAccount a : accounts)
        {
            if (a.getAccountNumber() == accountNumber)
                return a; // Trovato
        }
        return null; // non trovato nell'intero vettore
    }
    . . .
}
```

# Semplici algoritmi per vettori

## Trovare il valore massimo o minimo

- Per trovare il valore massimo (o minimo) in un vettore
  - inizializzare un candidato con l'elemento iniziale,
  - confrontare il candidato con gli elementi rimanenti
  - aggiornarlo se si trova un elemento maggiore (o minore).
- Esempio

```
BankAccount largestYet = accounts.get(0);
for (int i = 1; i < accounts.size(); i++)
{
    BankAccount a = accounts.get(i);
    if (a.getBalance() > largestYet.getBalance())
        largestYet = a;
}
return largestYet;
```



# Semplici algoritmi per vettori

## Trovare il massimo o il minimo

---

- Questo metodo funziona soltanto se il vettore contiene almeno un elemento: non ha senso cercare l'elemento di valore maggiore in un insieme vuoto.
- Se l'insieme è vuoto, restituisce `null`

```
if (accounts.size() == 0) return null;
BankAccount largestYet = accounts.get(0);
...
```

# File Bank.java

```
01: import java.util.ArrayList;
02:
03: /**
04:     Una banca contiene un insieme di conti bancari .
05: */
06: public class Bank
07: {
08:     /**
09:         Costruisce una banca priva di conti bancari.
10:     */
11:     public Bank()
12:     {
13:         accounts = new ArrayList<BankAccount>();
14:     }
15:
16:     /**
17:         Aggiunge un conto bancario a questa banca.
18:         @param a il conto da aggiungere
19:     */
```

**Segue**

# File Bank.java

```
20: public void addAccount (BankAccount a)
21: {
22:     accounts.add(a);
23: }
24:
25: /**
26:     Restituisce la somma dei saldi di tutti i conti della banca.
27:     @return la somma dei saldi
28: */
29: public double getTotalBalance()
30: {
31:     double total = 0;
32:     for (BankAccount a : accounts)
33:     {
34:         total = total + a.getBalance();
35:     }
36:     return total;
37: }
38:
```

**Segue**

# File Bank.java

```
39:     /**
40:         Conta il numero di conti bancari aventi saldo maggiore
41:         o uguale al valore indicato.
42:         @param atLeast il saldo minimo perché un conto venga conteggiato
43:         @return il numero di conti aventi saldo >= al saldo indicato
44:     */
45:     public int count(double atLeast)
46:     {
47:         int matches = 0;
48:         for (BankAccount a : accounts)
49:         {
50:             if (a.getBalance() >= atLeast) matches++; // trovato
51:         }
52:         return matches;
53:     }
54:
```

**Segue**

# File Bank.java

```
55:     /**
56:         Verifica se la banca contiene un conto con il numero indicato.
57:         @param accountNumber il numero di conto da cercare
58:         @return il conto con il numero indicato, oppure null se
59:                 tale conto non esiste
60:     */
61:     public BankAccount find(int accountNumber)
62:     {
63:         for (BankAccount a : accounts)
64:         {
65:             if (a.getAccountNumber() == accountNumber)
66:                 return a; // trovato
67:         }
68:         return null; // non trovato nell'intero vettore
69:     }
70:
```

**Segue**

# File Bank.java

```
71:     /**
72:         Restituisce il conto bancario avente il saldo maggiore.
73:         @return il conto con il saldo maggiore, oppure null se
74:             la banca non ha conti
75:     */
76:     public BankAccount getMaximum()
77:     {
78:         if (accounts.size() == 0) return null;
79:         BankAccount largestYet = accounts.get(0);
80:         for (int i = 1; i < accounts.size(); i++)
81:         {
82:             BankAccount a = accounts.get(i);
83:             if (a.getBalance() > largestYet.getBalance())
84:                 largestYet = a;
85:         }
86:         return largestYet;
87:     }
88:
89:     private ArrayList<BankAccount> accounts;
90: }
```

# File BankTester.java

```
01: /**
02:     Questo programma collauda la classe Bank.
03: */
04: public class BankTester
05: {
06:     public static void main(String[] args)
07:     {
08:         Bank firstBankOfJava = new Bank();
09:         firstBankOfJava.addAccount(new BankAccount(1001, 20000));
10:         firstBankOfJava.addAccount(new BankAccount(1015, 10000));
11:         firstBankOfJava.addAccount(new BankAccount(1729, 15000));
12:
13:         double threshold = 15000;
14:         int c = firstBankOfJava.count(threshold);
15:         System.out.println("Count: " + c);
        System.out.println("Expected: 2");
    }
}
```

**Segue**

# File BankTester.java

```
16:
17:     int accountNumber = 1015;
18:     BankAccount a = firstBankOfJava.find(accountNumber);
19:     if (a == null)
20:         System.out.println("No matching account");
21:     else
22:         System.out.println("Balance of matching account: "
23:             + a.getBalance());
24:     System.out.println("Expected: 10000");

25:     BankAccount max = firstBankOfJava.getMaximum();
26:     System.out.println("Account with largest balance: "
27:         + max.getAccountNumber()
28:     System.out.println("Expected: 1001");
29:     }
30: }
```

**Segue**



# File BankTester.java

---

## Visualizza

Count: 2

Expected: 2

Balance of matching account: 10000.0

Expected: 10000

Account with largest balance: 1001

Expected: 1001

# Array a due dimensioni

- Gli array bidimensionali rappresentano una tabella, una disposizione di elementi a due dimensioni. Si accede agli elementi di un array bidimensionale usando una coppia di indici, `a[i][j]`.
- Quando si costruisce un array bidimensionale, si deve specificare quante righe e quante colonne servono.

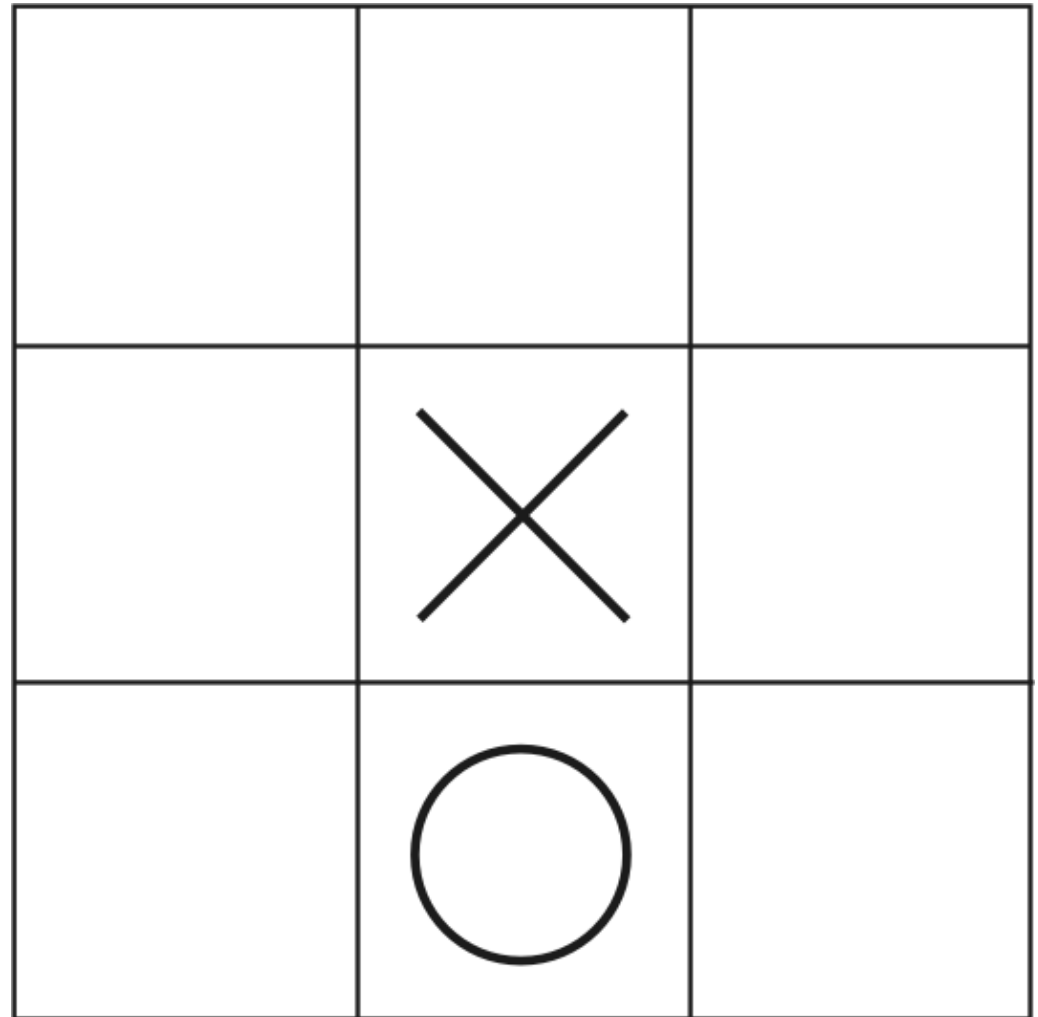
```
final int ROWS = 3;  
final int COLUMNS = 3;  
String[][] board = new String[ROWS][COLUMNS];
```

- Per accedere a un particolare elemento della matrice, si usano due indici tra parentesi quadre separate

```
board[i][j] = "x";
```

# Una scacchiera per il gioco “Tic-Tac-Toe”

**Figura 6**  
Una scacchiera per il gioco  
"Tic-Tac-Toe" (tris o filetto)



# Array a due dimensioni

- Quando si inseriscono o si cercano dati in un array bidimensionale, di solito si usano due cicli annidati.
- Per esempio, questa coppia di cicli assegna a tutti gli elementi dell'array una stringa contenente il solo carattere di spaziatura.

```
for (int i = 0; i < ROWS; i++)  
    for (int j = 0; j < COLUMNS; j++)  
        board[i][j] = " ";
```

# File TicTacToe.java

```
01: /**
02:     Una scacchiera 3x3 per il gioco tic-tac-toe.
03: */
04: public class TicTacToe
05: {
06:     /**
07:         Costruisce una scacchiera vuota.
08:     */
09:     public TicTacToe()
10:     {
11:         board = new String[ROWS][COLUMNS];
12:         // riempi di spazi
13:         for (int i = 0; i < ROWS; i++)
14:             for (int j = 0; j < COLUMNS; j++)
15:                 board[i][j] = " ";
16:     }
17:
```

**Segue**

# File TicTacToe.java

```
18:    /**
19:       Imposta un settore della scacchiera.
       Il settore deve essere libero.
20:       @param i l'indice di riga
21:       @param j l'indice di colonna
22:       @param player il giocatore ("x" o "o")
23:    */
24:    public void set(int i, int j, String player)
25:    {
26:        if (board[i][j].equals(" "))
27:            board[i][j] = player;
28:    }
29:
30:    /**
31:       Crea una rappresentazione della scacchiera in una stringa, ad esempio
32:       |x  o|
33:       |  x |
34:       |  o|
35:       @return la rappresentazione della stringa
36:    */
```

**Segue**

# File TicTacToe.java

```
37: public String toString()
38: {
39:     String r = "";
40:     for (int i = 0; i < ROWS; i++)
41:     {
42:         r = r + "|";
43:         for (int j = 0; j < COLUMNS; j++)
44:             r = r + board[i][j];
45:         r = r + "|\n";
46:     }
47:     return r;
48: }
49:
50: private String[][] board;
51: private static final int ROWS = 3;
52: private static final int COLUMNS = 3;
53: }
```

# File TicTacToeRunner.java

```
01: import java.util.Scanner;
02:
03: /**
04:     Questo programma esegue la classe TicTacToe
05:     chiedendo all'utente di selezionare posizioni sulla
06:     scacchiera e visualizzando il risultato.
07: */
08: public class TicTacToeRunner
09: {
10:     public static void main(String[] args)
11:     {
12:         Scanner in = new Scanner(System.in);
13:         String player = "x";
14:         TicTacToe game = new TicTacToe();
15:         boolean done = false;
16:         while (!done)
17:         {
```

**Segue**



# File TicTacToeRunner.java

```
18:         System.out.println(game.toString());
19:         System.out.print(
20:             "Row for " + player + " (-1 to exit): ");
21:         int row = in.nextInt();
22:         if (row < 0) done = true;
23:         else
24:         {
25:             System.out.print("Column for " + player + ": ");
26:             int column = in.nextInt();
27:             game.set(row, column, player);
28:             if (player.equals("x"))
29:                 player = "o";
30:             else
31:                 player = "x";
32:         }
33:     }
34: }
35: }
```

**Segue**

# File TicTacToeRunner.java

Visualizza

```
| |  
| |  
| |  
Row for x (-1 to exit): 1  
Column for x: 2  
  
| |  
| x|  
|  
Row for o (-1 to exit): 0  
Column for o: 0  
  
|o |  
| x|  
| |  
Row for x (-1 to exit): -1
```

# Copiare array:

## Copiare il riferimento a un array

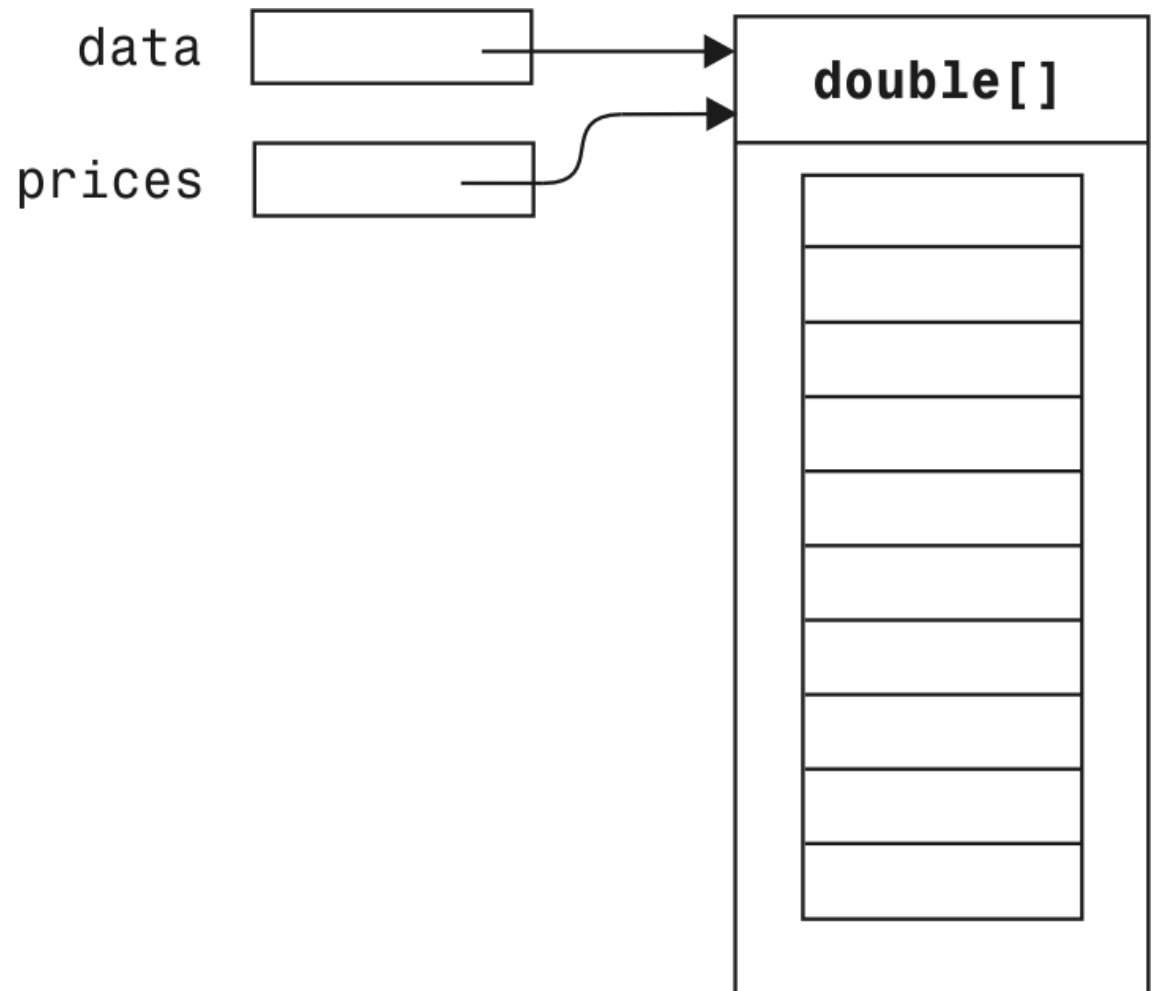
---

- Una variabile di tipo array memorizza un riferimento all'array. Copiando la variabile si ottiene un secondo riferimento al medesimo array.

```
double[] data = new double[10];  
    . . . // riempimento dell'array  
double[] prices = data;
```

***Segue***

# Copiare array: Copiare il riferimento a un array

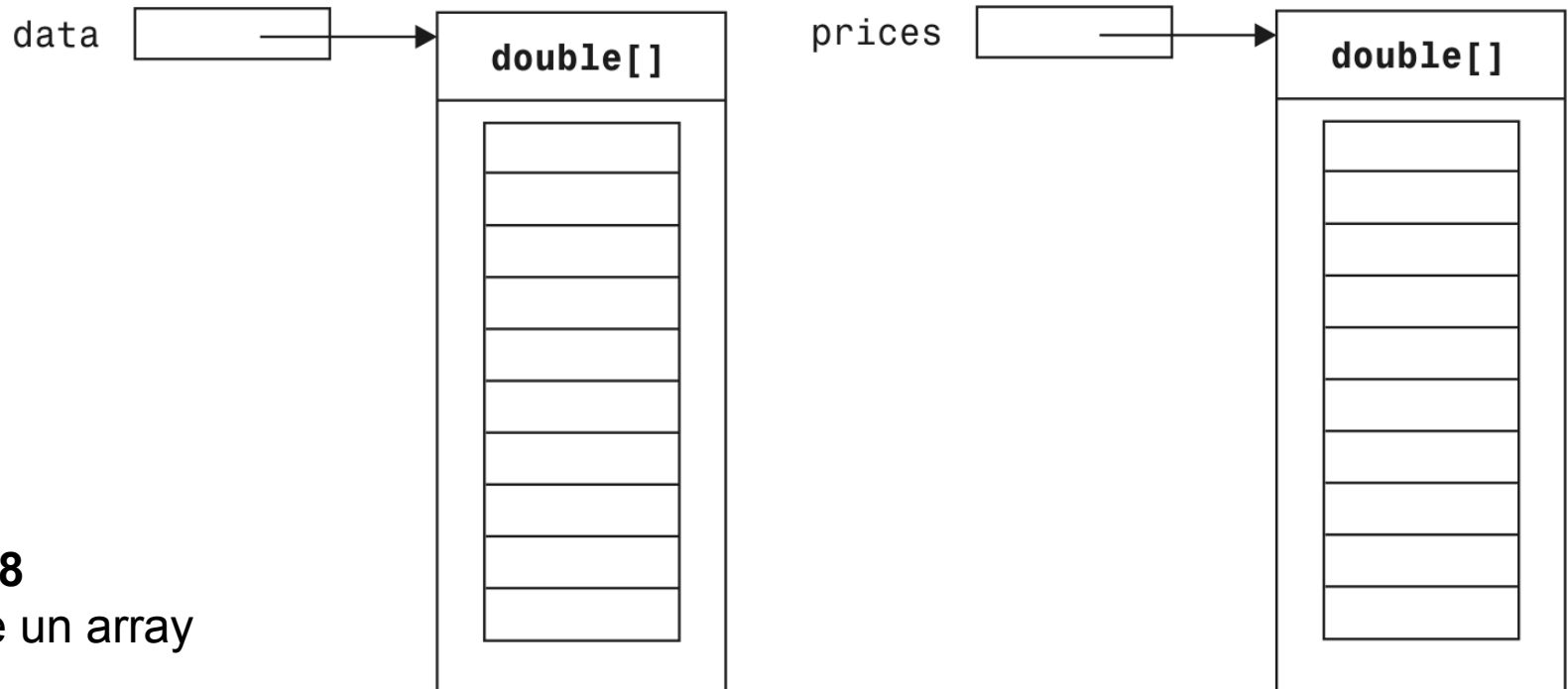


**Figura 7**  
Due riferimenti  
allo stesso array

# Copiare array: Clonare un array

- Per copiare gli elementi di un array usate il metodo clone.

```
double[] prices = (double[]) data.clone();
```

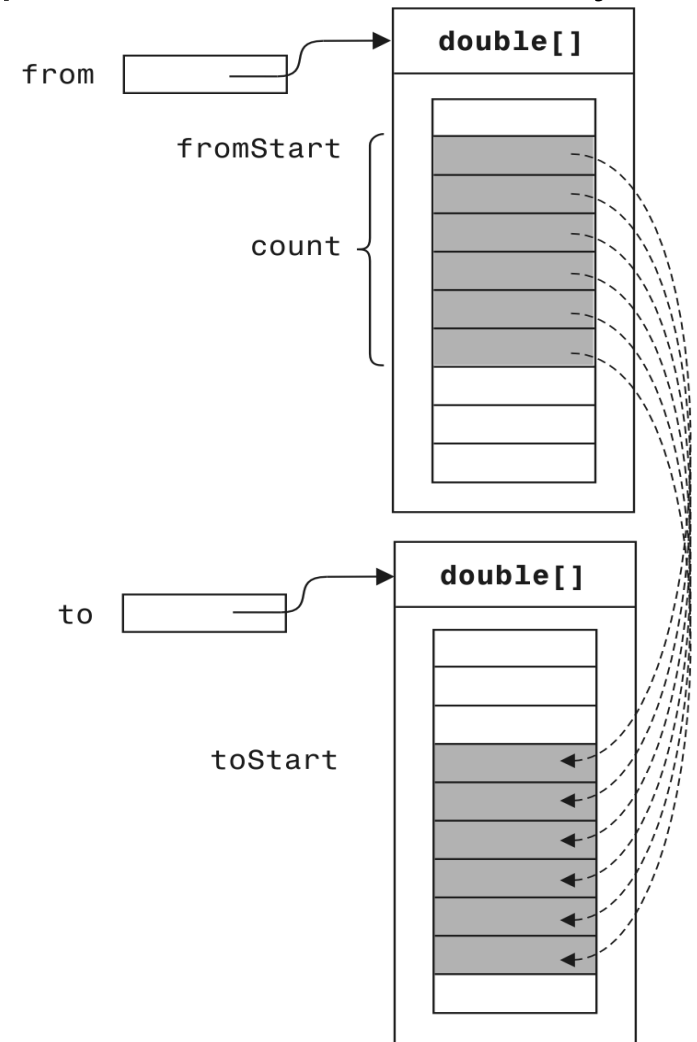


**Figura 8**  
Clonare un array

# Copiare array: Copiare gli elementi di un array

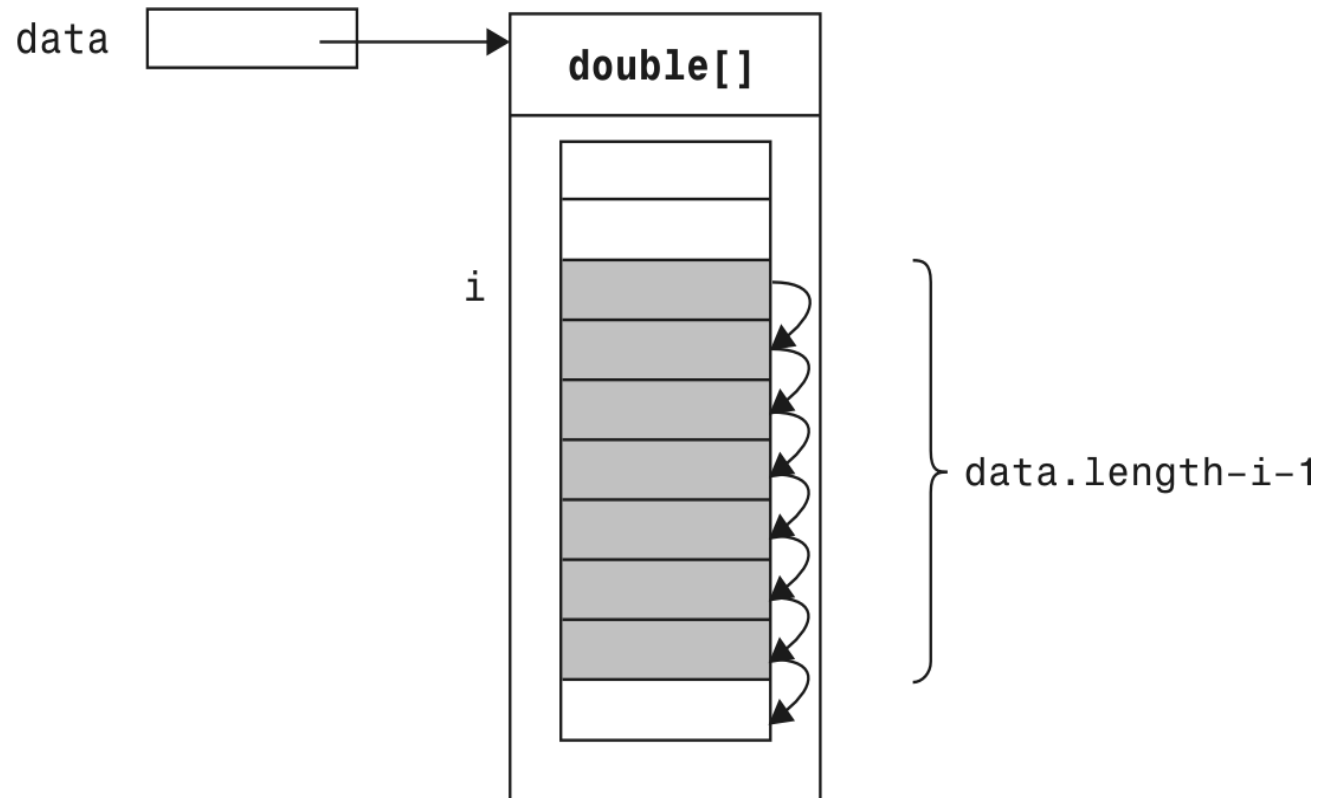
- Usate il metodo `System.arraycopy` per copiare elementi da un array a un altro.

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



**Figura 9**  
Il metodo  
`System.arraycopy`

# Inserire un elemento in un array

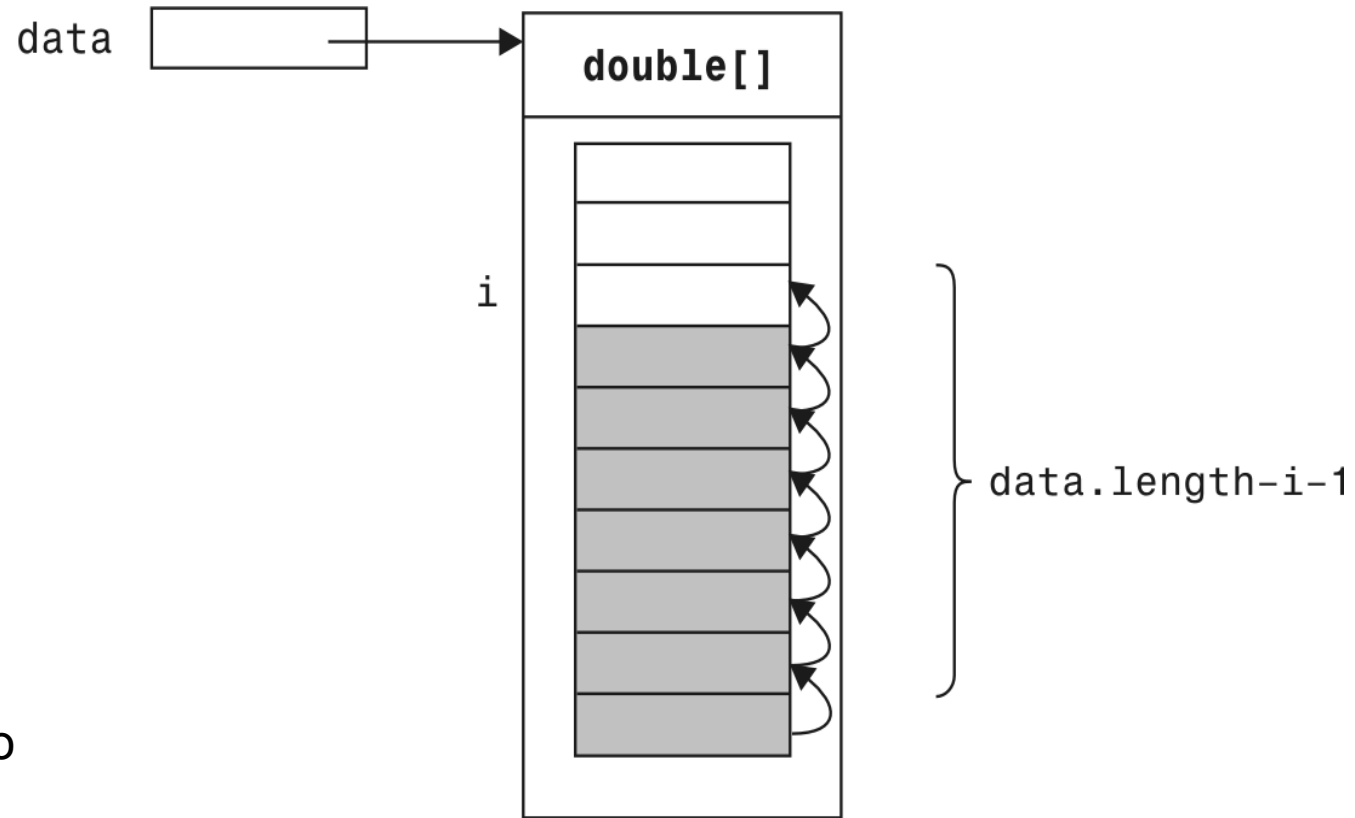


**Figura 10**

Inserire un nuovo elemento in un array

```
System.arraycopy(data, i, data, i + 1, data.length - i - 1);  
data[i] = x;
```

# Rimuovere un elemento da un array



**Figura 11**

Rimuovere un elemento  
da un array

```
System.arraycopy(data, i + 1, data, i, data.length - i - 1);
```



# Far crescere un array

- Il metodo `System.arraycopy` viene anche utilizzato per far crescere di dimensione un array che non ha più spazio, seguendo queste fasi operative:

1. Creare un nuovo array, di dimensione maggiore

```
double[] newData = new double[2 * data.length];
```

2. Copiare tutti gli elementi nel nuovo array

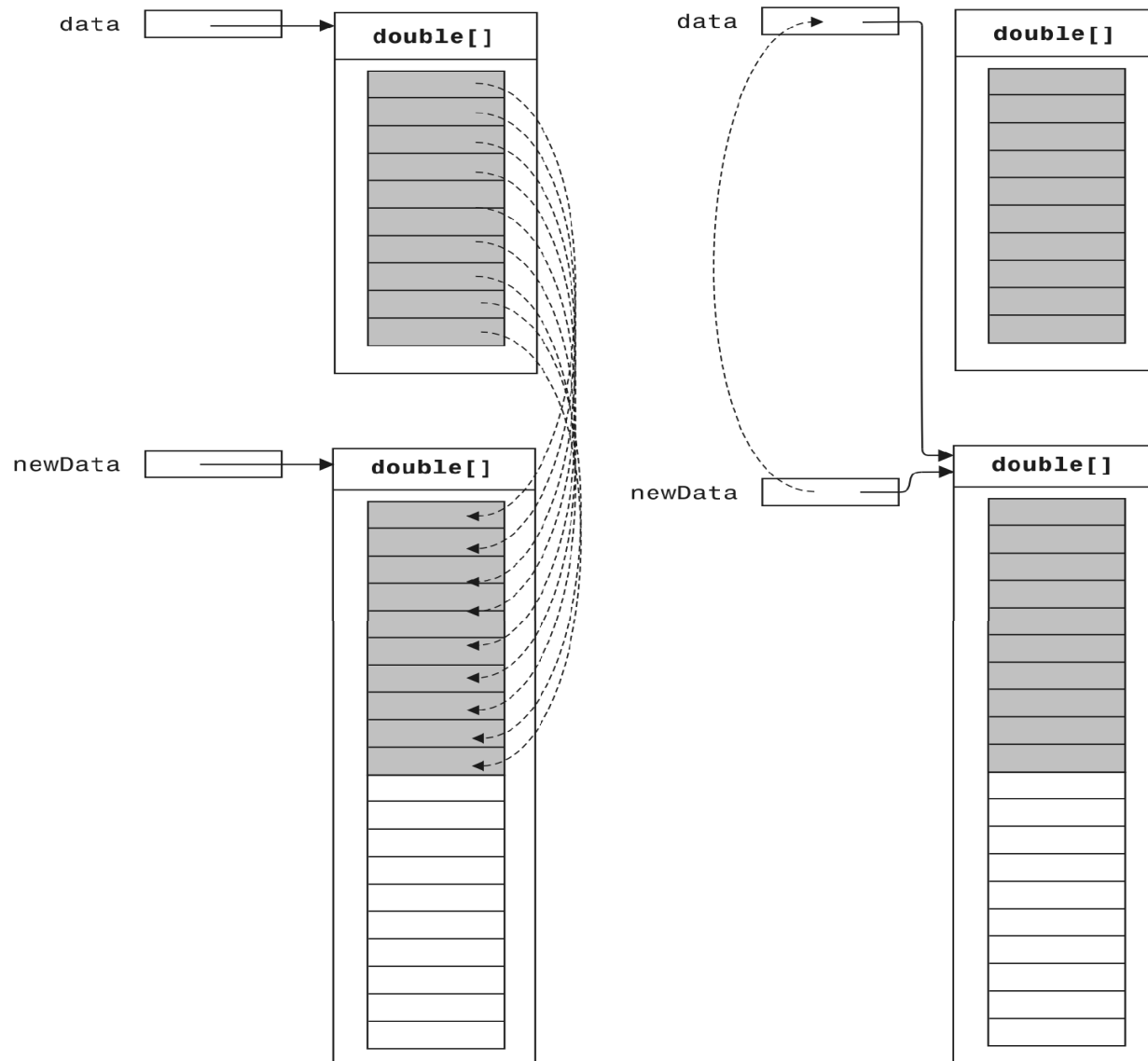
```
System.arraycopy(data, 0, newData, 0, data.length);
```

3. Memorizzare nella variabile array il riferimento al nuovo array

```
data = newData;
```

# Far crescere un array

**Figura 12**  
Far crescere un array



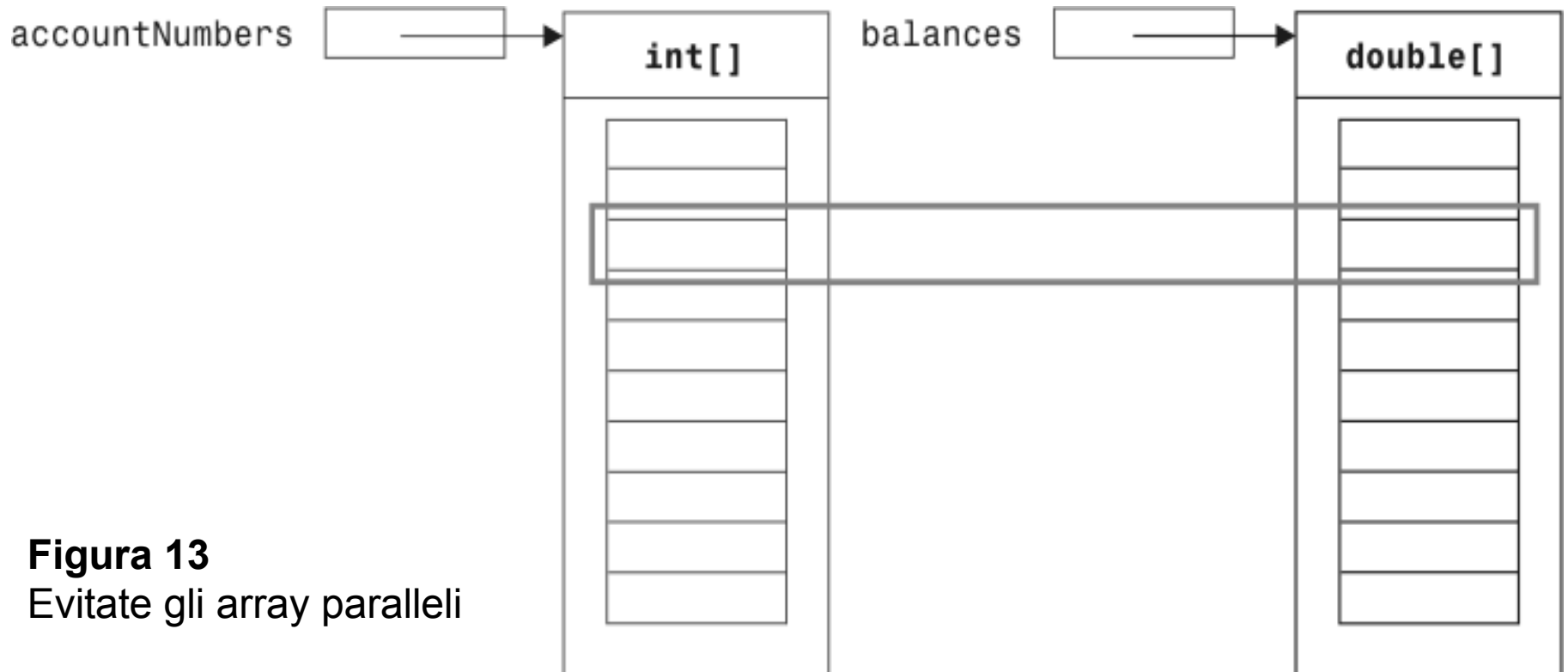
# Far crescere un array

---

- In alternativa a `.clone()` ed al metodo `System.arraycopy` si può utilizzare
  - `Arrays.copyOf`
  - `Arrays.copyOfRange`

# Trasformare array paralleli in array di oggetti

```
// non fate così  
int[] accountNumbers;  
double[] balances;
```

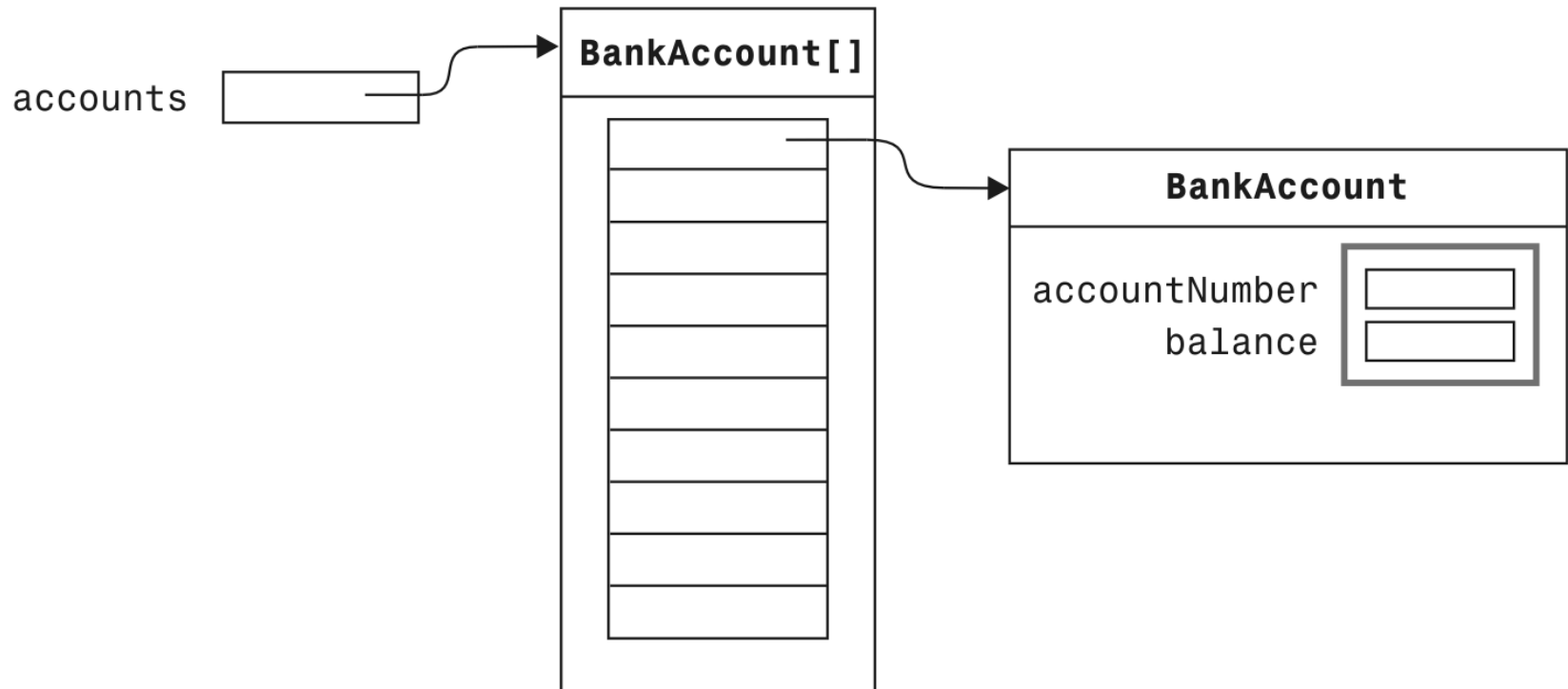


**Figura 13**  
Evitate gli array paralleli

# Trasformare array paralleli in array di oggetti

- Evitare di usare array paralleli trasformandoli in array di oggetti. Usare un unico array di oggetti

```
BankAccount[] = accounts;
```



**Figura 14** Riorganizzare array paralleli in array di oggetti

# Array riempiti solo in parte

- La dimensione dell'array va impostata prima di sapere quanti sono gli elementi di cui si ha bisogno e non può più essere modificata.
- Si può creare un array che sia sicuramente più grande del numero massimo possibile di voci e poi riempirlo solo parzialmente.
- Usare una variabile complementare che dica quanti elementi dell'array sono realmente utilizzati.
- Assegnare sempre a tale variabile complementare un nome ottenuto aggiungendo il suffisso `Size` al nome dell'array.

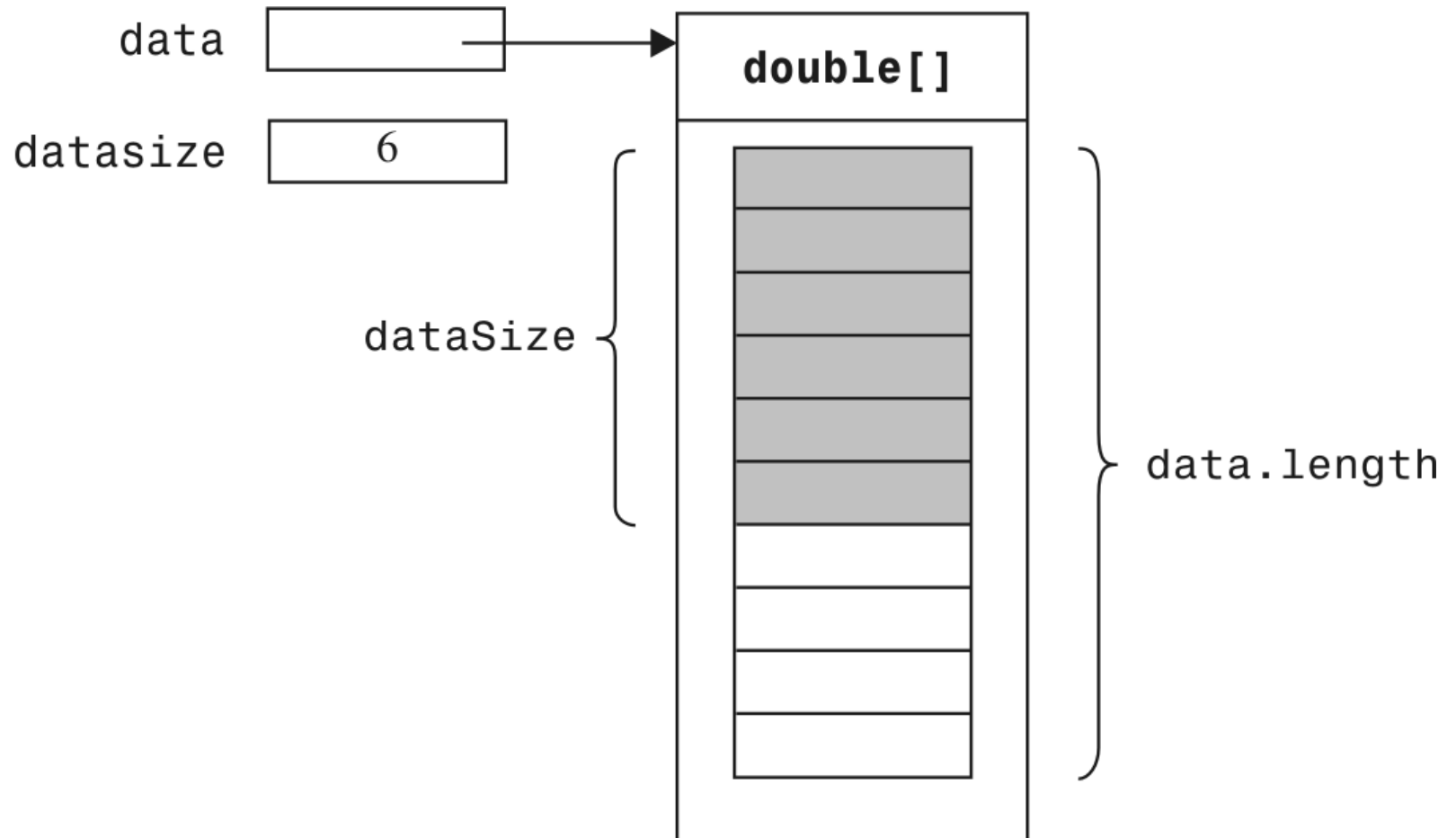
```
final int DATA_LENGTH = 100;  
double[] data = new double[DATA_LENGTH];  
int dataSize = 0;
```

# Array riempiti solo in parte

- `data.length` è la capacità dell'array `data`, mentre `dataSize` è la dimensione reale dell'array (Figura 15).  
Continuando ad aggiungere elementi all'array, bisogna incrementare di pari passo la variabile dimensione.

```
data[dataSize] = x;  
dataSize++;
```

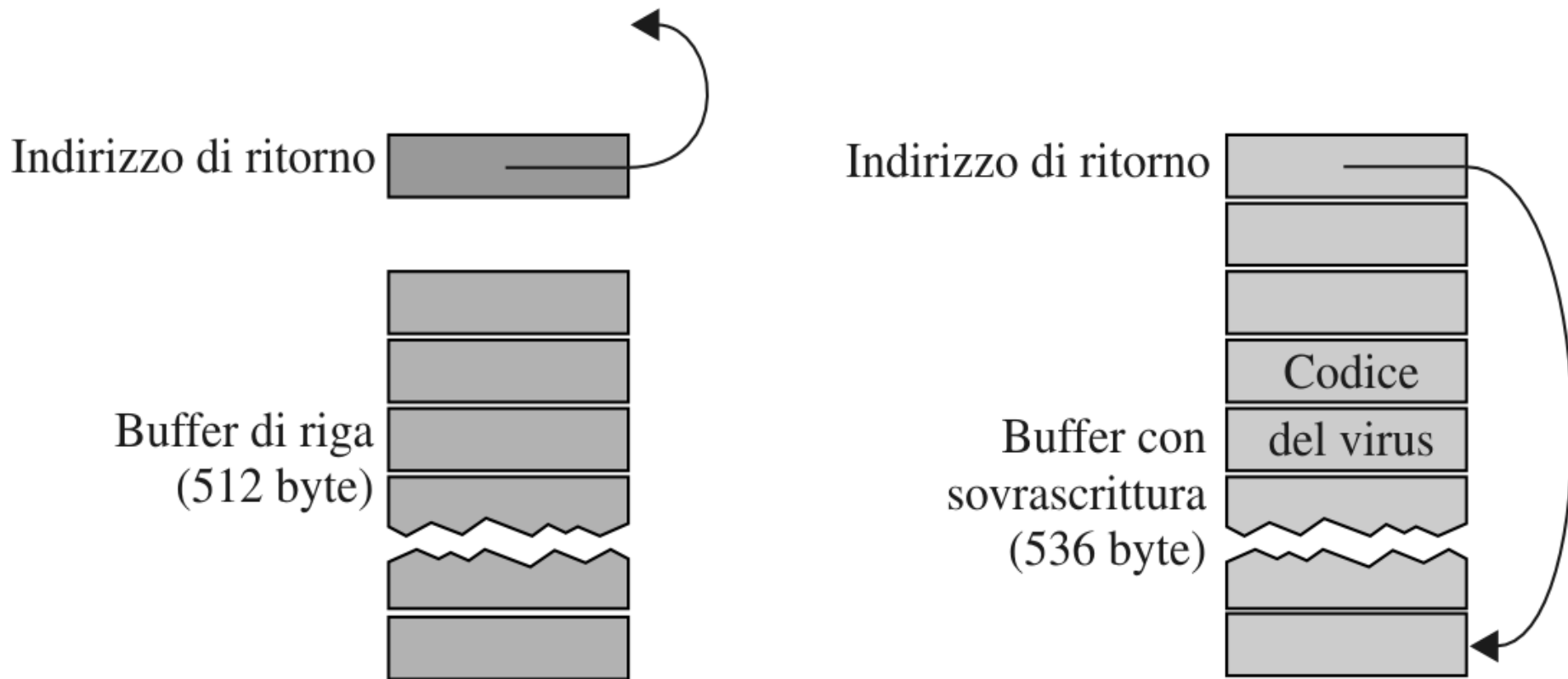
# Array riempiti solo in parte



**Figura 15** Un array riempito solo in parte



# Uno dei primi *worm* di Internet



**Figura 16** Un attacco di tipo "Buffer Overrun"

# Collaudo regressivo

---

- Salvare i casi di prova
- Usare i casi di prova salvati per collaudare la versione successiva del programma
- Un *pacchetto di prova* è un insieme di prove da ripetere per il collaudo
- *Ciclicità*: fenomeno per cui un errore già corretto riappare in una versione successiva
- Il collaudo regressivo prevede l'esecuzione ripetuta di prove già eseguite in precedenza, per essere certi che guasti noti delle versioni precedenti non compaiano nelle nuove versioni del programma

# File BankTester.java

```
01: import java.util.Scanner;
02:
03: /**
04:     This program tests the Bank class.
05: */
06: public class BankTester
07: {
08:     public static void main(String[] args)
09:     {
10:         Bank firstBankOfJava = new Bank();
11:         firstBankOfJava.addAccount(new BankAccount(1001, 20000));
12:         firstBankOfJava.addAccount(new BankAccount(1015, 10000));
13:         firstBankOfJava.addAccount(new BankAccount(1729, 15000));
14:
15:         Scanner in = new Scanner(System.in);
16:
17:         double threshold = in.nextDouble();
18:         int c = firstBankOfJava.count(threshold);
19:         System.out.println("Count: " + c);
20:         int expectedCount = in.nextInt();
21:         System.out.println("Expected: " + expectedCount);
22:
```

# File BankTester.java

```
23:     int accountNumber = in.nextInt();
24:     BankAccount a = firstBankOfJava.find(accountNumber);
25:     if (a == null)
26:         System.out.println("No matching account");
27:     else
28:     {
29:         System.out.println("Balance of matching account: " +
a.getBalance());
30:         int matchingBalance = in.nextLine();
31:         System.out.println("Expected: " + matchingBalance);
32:     }
33: }
34: }
```

# Redirezione del flusso di ingresso

- Memorizzare i valori in ingresso in un file

## **File input1.txt**

15000

2

1015

10000

- Scrivete questo comando in una finestra di shell:

```
java BankTester < input1.txt
```

- Visualizza

```
Count: 2
```

```
Expected: 2
```

```
Balance of matching account: 10000.0
```

```
Expected: 10000
```

- E' possibile redigere anche il flusso in uscita

```
java BankTester < input1.txt > output1.txt
```

# Metodi con numero variabile di parametri

- E' possibile definire metodi che accettano in input un numero variabile di parametri, che vengono memorizzati in un array
- Con restrizioni: una sola variabile di questo tipo, e deve essere l'ultima

```
7     public static double average( double... numbers )
8     {
9         double total = 0.0; // initialize total
10
11         // calculate total using the enhanced for statement
12         for ( double d : numbers )
13             total += d;
14
15         return total / numbers.length;
16     } // end method average
```

Fonte Deitel.com