

# **Tipi di dato Fondamentali**

# Tipi numerici

---

- **int**: interi, senza parte frazionaria

```
1, -4, 0
```

- **double**: numeri in virgola mobile (precisione doppia)

```
0.5, -3.11111, 4.3E24, 1E-14
```

# Tipi numerici

---

- Una computazione su tipi numerici può causare overflow.

```
int n = 1000000;  
System.out.println(n * n); // stampa -727379968
```

- Java: 8 tipi primitivi, che includono quattro tipi interi e due in virgola mobile

# Tipi primitivi

---

Tipo	Range di valori	Dimensione
int	−2,147,483,648 . . . 2,147,483,647	4 bytes
byte	−128 . . . 127	1 byte
short	−32768 . . . 32767	2 bytes
long	9,223,372,036,854,775,808 . . . −9,223,372,036,854,775,807	8 bytes

*Continua...*

# Tipi primitivi

Tipo	Descrizione dei valori	Dim
<code>double</code>	Virgola mobile, precisione doppia. Range $\pm 10^{308}$ e 15 cifre decimali	8 bytes
<code>float</code>	Virgola mobile, precisione singola. Range $\pm 10^{38}$ e 7 cifre decimali	4 bytes
<code>char</code>	Caratteri nel sistema Unicode	2 bytes
<code>boolean</code>	<code>false</code> e <code>true</code>	1 byte

# Float e double

---

- **Attenzione agli errori derivanti dagli arrotondamenti**

```
double f = 4.35;  
System.out.println(100 * f); // stampa 434.99999999999994
```

- **Non è ammesso assegnare un double (o float) ad un intero ... non è C !**

```
double d = 13.75;  
int i = d; // Errore
```

*Continua...*

# Float e double

---

- **Cast:** per convertire un double ad un int è quindi necessario un cast

```
int i = (int) d; // OK
```

- **Cast equivale a troncamento.**
- **Math.round arrotondamento**
  - `static long round(double a)`
  - `static int round(float a)`

# Domande

---

- In quale caso il cast `(long) x` dà un risultato diverso da `Math.round(x)`?
- Quale istruzione utilizzereste per arrotondare `x:double` al valore `int` più vicino?



# Risposte

---

- Quando la parte frazionaria di  $x$  è  $\geq 0.5$
- `(int) Math.round(x)` se assumiano che  $x$  è minore di  $2 \cdot 10^9$

# Stringhe

---

- Le stringhe sono oggetti

- Stringhe costanti

```
"Hello, World!"
```

- Variabili di tipo stringa

```
String message = "Hello, World!";
```

- Lunghezza di una stringa

```
int n = message.length();
```

- La stringa vuota:

```
" "
```

# Concatenazione

---

- Utilizziamo l'operatore **+**:

```
String name = "Dave";  
String message = "Hello, " + name;  
// = "Hello, Dave"
```

- Se uno degli argomenti dell'operatore è una stringa, l'altro argomento viene automaticamente convertito

```
String a = "Agente";  
int n = 7;  
String bond = a + "00" + n; // bond = "Agente007"
```

# Concatenazione in output

---

- Utile per ridurre il numero di istruzioni  
`System.out.print:`

```
System.out.print("The total is ");  
System.out.println(total);
```

**è equivalente a**

```
System.out.println("The total is " + total);
```

# Conversioni

---

- **Da stringhe a numeri:**

```
int n = Integer.parseInt(str);  
double x = Double.parseDouble(x);
```

- **Da numeri a stringhe**

```
String str = "" + n;  
str = Integer.toString(n);
```

# Confronti

---

- **s1 == s2**
  - true se e solo se s1 e s2 sono riferimenti alla stessa stringa (come per tutti i tipi reference)
- **s1.equals(s2)**
  - true se e solo se s1 e s2 sono stringhe uguali (case sensitive)
- **s1.equalsIgnoreCase(s2)**
  - come sopra ma case insensitive
- **s1.compareTo(s2)**
  - < 0 se s1 precede s2 in ordine lessicografico
  - = 0 se s1 e s2 sono uguali
  - > 0 se s1 segue s2 in ordine lessicografico

*Continua...*

# Confronti

---

- **Attenzione**

```
"str" == "str"
```

**true**

```
new String("str") == new String("str")
```

**false**

```
"str" == new String("str")
```

**false**

- **Le chiamate a new creano sempre oggetti diversi, l'uso di costanti invece è ottimizzato**

# Stringhe e array di caratteri

---

- Le stringhe non sono array di caratteri
- Esistono conversioni
  - `String`  $\longrightarrow$  `char[]`

```
String s = "str"  
Char data[] = s.toCharArray()
```

- `char[]`  $\longrightarrow$  `String`

```
char data[] = { 's', 't', 'r' }  
String s = new String(data)
```



# Sottostringhe

- ```
String greeting = "Hello, World!";  
String sub = greeting.substring(0, 5); // sub = "Hello"
```
- **`s.substring(i, j)` restituisce la sottostringa di `s` dalla posizione `i` alla `j-1` (estremi inclusi)**
- **`s.substring(i)` restituisce la sottostringa di `s` da `i` (incluso) al termine di `s`**
- **`s.indexOf(s1)` restituisce l'indice della prima occorrenza di `s1` in `s`; `-1` se non ci sono occorrenze**
- **`s.indexOf(s1, i)` l'indice della prima occorrenza di `s1` in `s` da `i` (incluso); `-1` se non ci sono occorrenze**

*Continua...*

# Lettura da input

---

- **System.in** (lo standard input) offre supporto minimo per la lettura
  - `read()`: lettura di una byte
- **La classe Scanner di Java 5.0** permette di leggere da input in modo più strutturato

```
Scanner in = new Scanner(System.in);  
System.out.print("Enter quantity: ");  
int quantity = in.nextInt();
```

*Continua...*

# Lettura da input

---

## Metodi utili della classe `Scanner`

- `nextDouble()` **legge un double**
- `nextLine()` **legge un linea**
  - la sequenza fino al primo newline
- `nextWord()` **legge una parola**
  - la sequenza fino al primo spazio/tab/newline

# File stringTester.java

```
import java.util.Scanner;
class stringTester {
    /**
     * Legge da input linee di testo ed estrae i campi delimitati da :
     *
     */
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String record, field;
        char delim = ':'; // il delimitatore
        int record_count = 1;
        while (in.hasNextLine()) {
            System.out.println("Record " + record_count++);
            record = in.nextLine();
            int begin, end, i; begin = 0;
            for (i=0; (end = record.indexOf(delim,begin)) >= 0; i++) {
                // end = indice della prossima occorrenza del delimiter
                field = record.substring(begin,end);
                begin = end + 1; // salta il delimitatore
                System.out.println("\tField" + i + ": " + field);
            }
            field = record.substring(begin); // l'ultimo campo
            System.out.println("\tField" + i + ": " + field);
        }
    }
}
```

# File stringTester.java

---

## Input

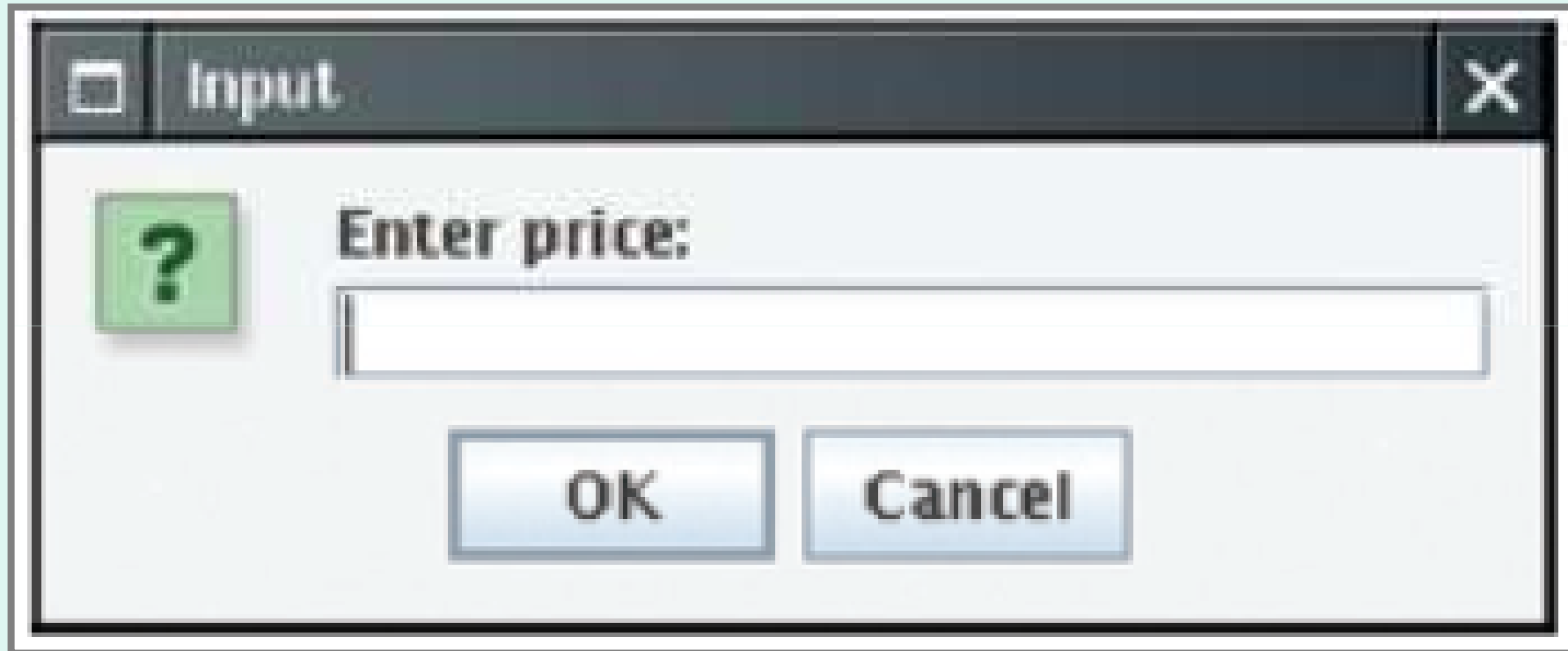
```
Gianni:Rossi:via Verdi 123:Milano:Italia  
Michele:Bugliesi:via Torino 155:Mestre:Italia:30173
```

## Output

```
Record 1  
  Field 1: Gianni  
  Field 2: Rossi  
  Field 3: via Verdi 123  
  Field 4: Milano  
  Field 5: Italia  
Record 2  
...
```

# Input da un Dialog Box

---



# Input da un Dialog Box

---

- ```
String input = JOptionPane.showInputDialog(prompt)
```
- **Converti le stringhe in numeri se necessario:**  

```
int count = Integer.parseInt(input);
```
- **La conversione lancia una eccezione se l'utente non fornisce un numero**
- **Aggiungete `System.exit(0)` al metodo `main` di qualunque programma che usi `JOptionPane`**

# Domande

---

15. Perché non possiamo leggere input direttamente da `System.in`?
16. Supponiamo `s` sia un oggetto di tipo `Scanner` che estrae dati da `System.in`, e consideriamo la chiamata  
`String name = s.next();`  
Quale è il valore della variabile `name` se l'utente dà in input `Hasan M. Jamil`?



# Risposte

---

- 15. Possiamo, ma il tipo di `System.in` permette solo letture molto primitive (un byte alla volta)..
- 16. Il valore è `"Hasan"`.

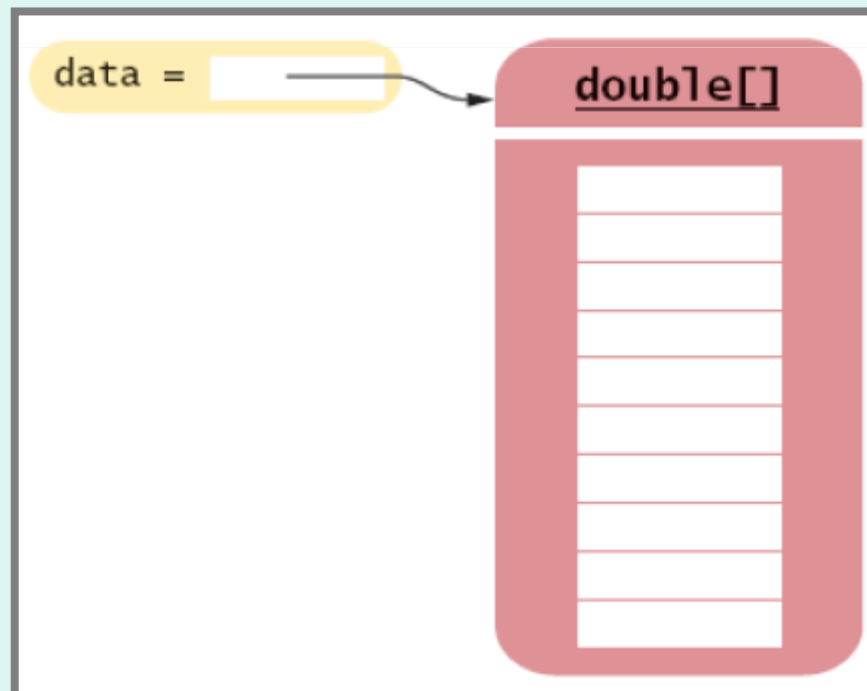
# **Arrays**

## **Array Lists**

# Arrays

- **Array: una sequenza di valori dello stesso tipo**

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



*Continua...*

# Arrays

---

- Al momento della allocazione, tutti i valori sono inizializzati ai valori di default del tipo base
  - numeri: 0
  - booleani: **false**
  - riferimenti: **null**

# Arrays

---

- **Accesso agli elementi mediante l'operatore [ ]**

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

- **Lunghezza dell'array: data.length**  
(N.B. non è un metodo)
- **Indici compresi tra 0 e data.length - 1**

```
data[10] = 29.95;// ERRORE: ArrayIndexOutOfBoundsException
```

# Domanda

---

- Quali elementi contiene l'array al termine della seguente inizializzazione?

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

# Risposta

---

- 0 1 4 9 16 25 36 49 64 81 **ma non 100**

# Domanda

---

- Cosa stampano i seguenti comandi. Ovvero, se causano un errore, quale errore causano? Specificate se si tratta di un errore di compilazione o di un errore run-time.

```
1.  double[] a = new double[10];  
    System.out.println(a[0]);  
2.  double[] b = new double[10];  
    System.out.println(b[10]);  
3.  double[] c;  
    System.out.println(c[0]);
```



# Risposta

---

- 2. 0
  3. errore run-time error: indice fuori range
  4. Errore compile-time: c non è inizializzato

# Array bidimensionali

---

- All'atto della costruzione specifichiamo il numero di righe e di colonne:

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

- Per accedere un elemento, utilizziamo i due indici: `a[i][j]`

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

# Copia di array

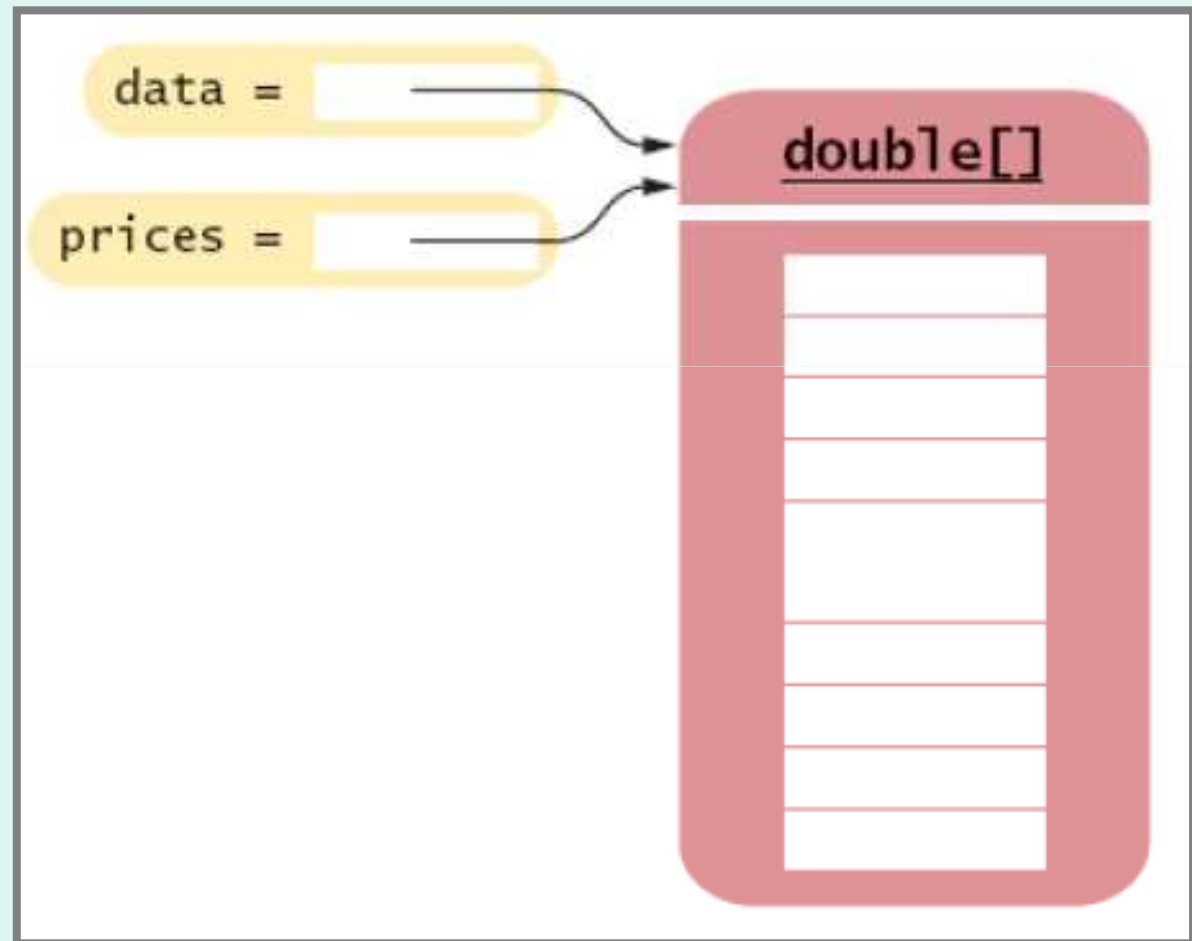
---

- **Gli array sono oggetti, quindi l'assegnamento tra array è un assegnamento di riferimenti**

```
double[] data = new double[10];  
// fill array . . .  
double[] prices = data;
```

*Continua...*

# Copia di array



# Copia di array – copia di elementi

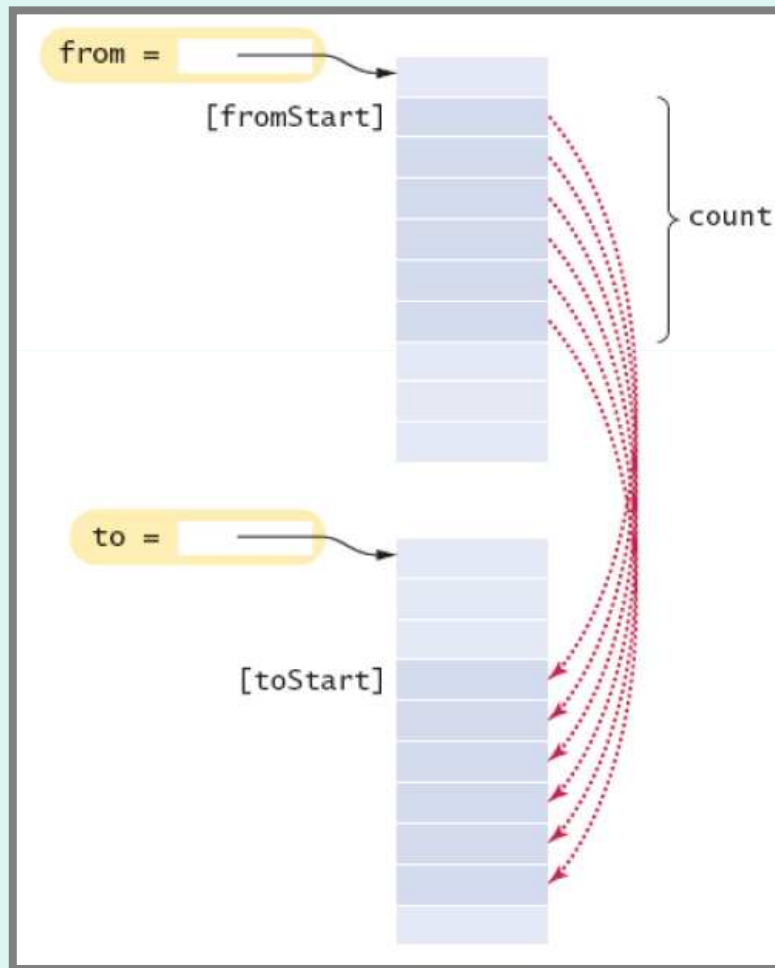
---

- Volendo duplicare gli elementi, il linguaggio fornisce metodi efficienti:

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

*Continua...*

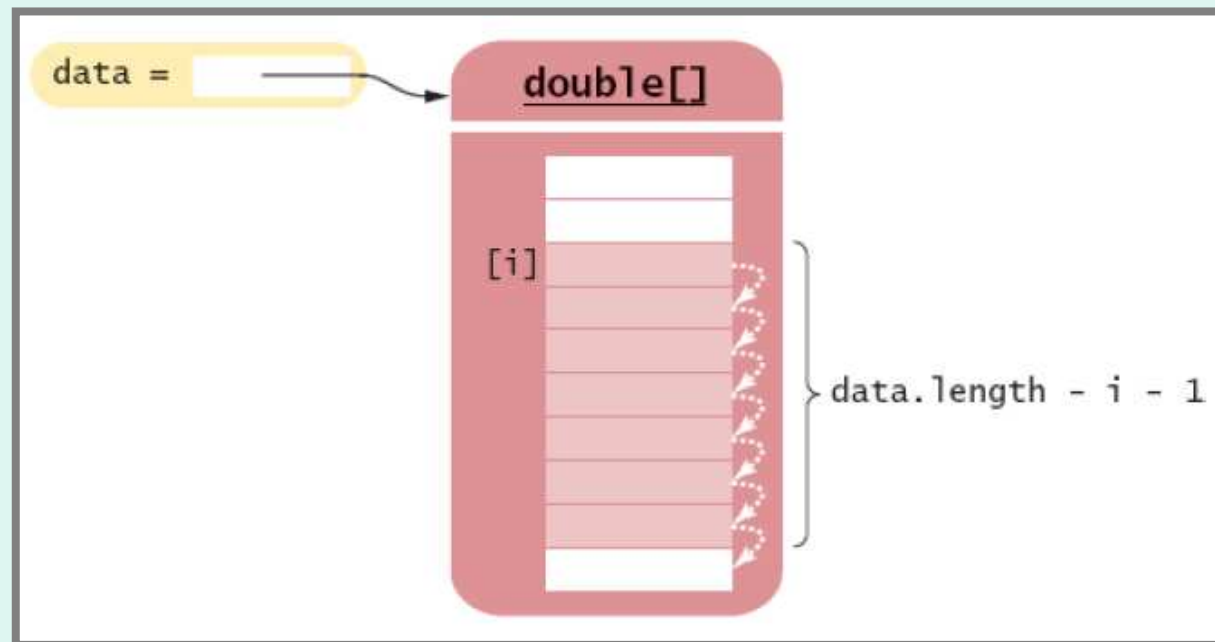
# Copia di array – copia di elementi



# Copia di elementi

- Array sorgente e destinazione possono coincidere

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



# ArrayLists

---

- La classe `ArrayList` gestisce una sequenza di oggetti
- A differenza degli array può variare in dimensione
- Fornisce metodi corrispondenti a molte operazioni comuni
  - inserzione, accesso e rimozione di elementi, ...

*Continua...*



# Array Lists

---

- La classe `ArrayList` è una classe parametrica (generica)
- `ArrayList<T>` contiene oggetti di tipo `T`:

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

- `size()`: il metodo che calcola il numero di elementi nella struttura

# Accesso agli elementi

---

- `get ( )`
- **Come per gli array**
  - gli indici iniziano da 0
  - errore se l'indice è fuori range
  - posizioni accessibili:  $0 \dots \text{size}() - 1$ .

```
BankAccount anAccount = accounts.get(2);  
    // il terzo elemento dalla arraylist
```

*Continua...*

# Nuovi elementi

---

- **set ( ) sovrascrive un valore esistente**

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

- **add ( ) aggiunge un nuovo valore, alla posizione i**

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

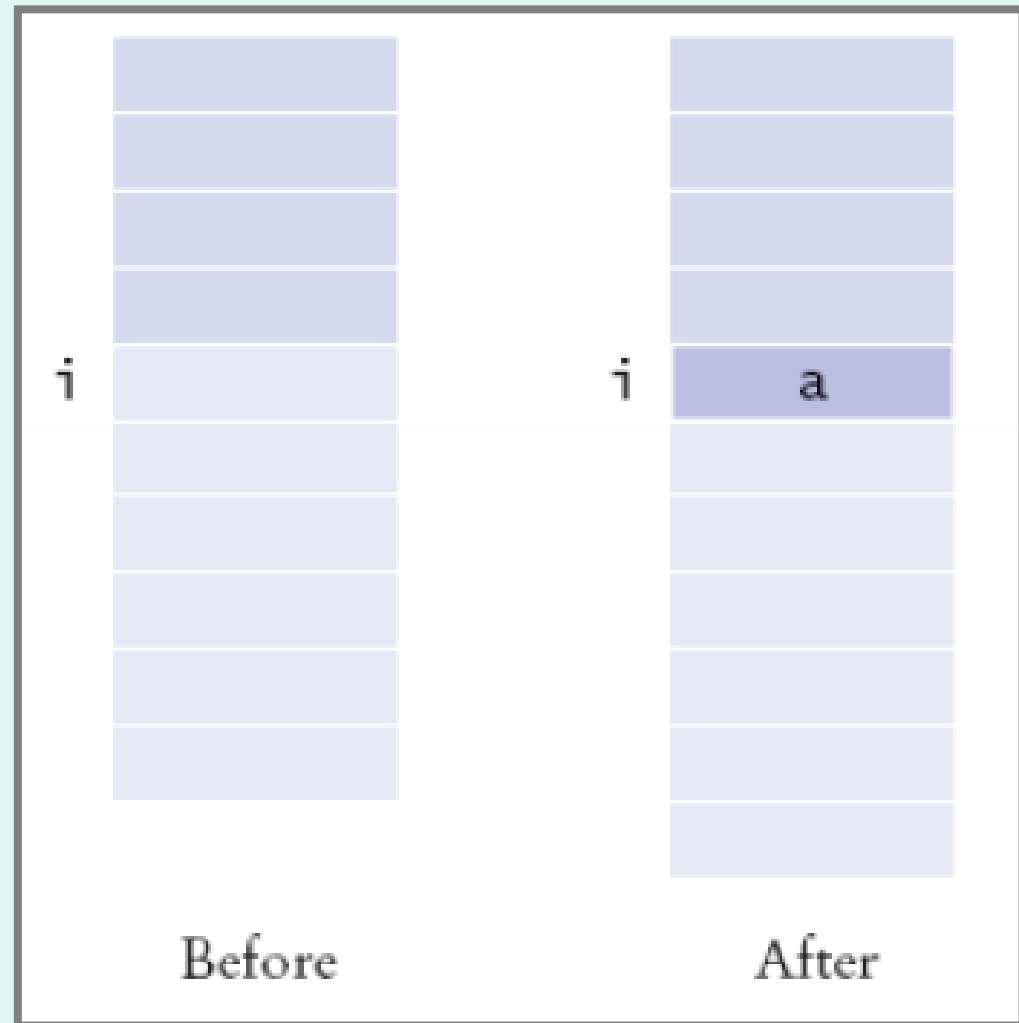
- **Oppure all'ultima posizione**

```
accounts.add(a)
```

*Continua...*

# Nuovi elementi – add

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



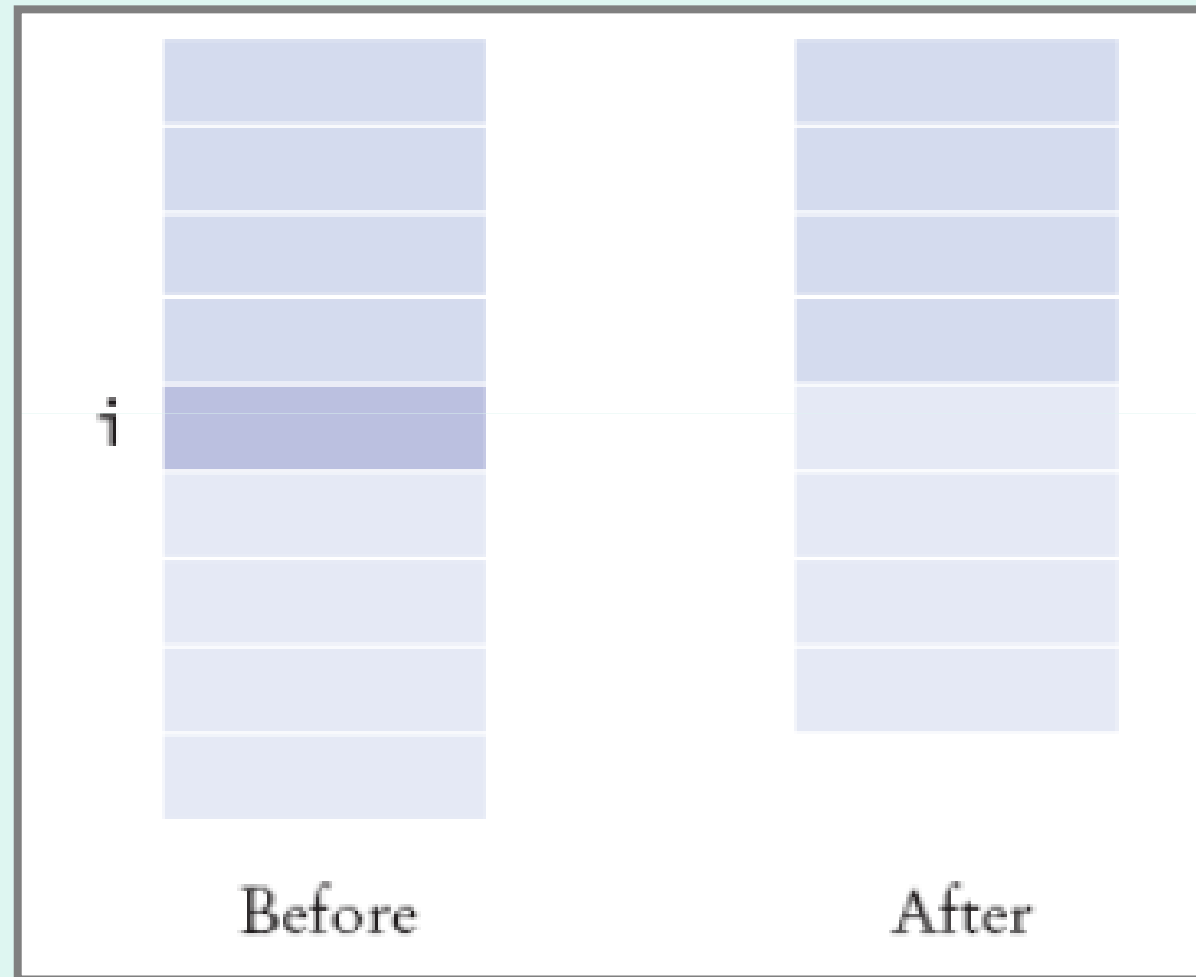
# Rimozione di elementi

---

- **`remove( )`** rimuove l'elemento all'indice **`i`**

```
accounts.remove(i)
```

# Rimozione di elementi



# Errori tipici

---

- **Accesso fuori dai bounds**

- indici legali 0..size()-1

```
int i = accounts.size();  
accounts.get(i);           // errore!  
accounts.set(i,anAccount); // errore!  
accounts.add(i+1,anAccount); // errore!
```

# Domanda

---

3. Quale è il contenuto della struttura `names` dopo le seguenti istruzioni?

```
ArrayList<String> names = new ArrayList<String>();  
names.add("A");  
names.add(0, "B");  
names.add("C");  
names.remove(1);
```



# Risposta

---

3. contiene le stringhe "B" e "C" alle posizioni 0 e 1

# Wrappers

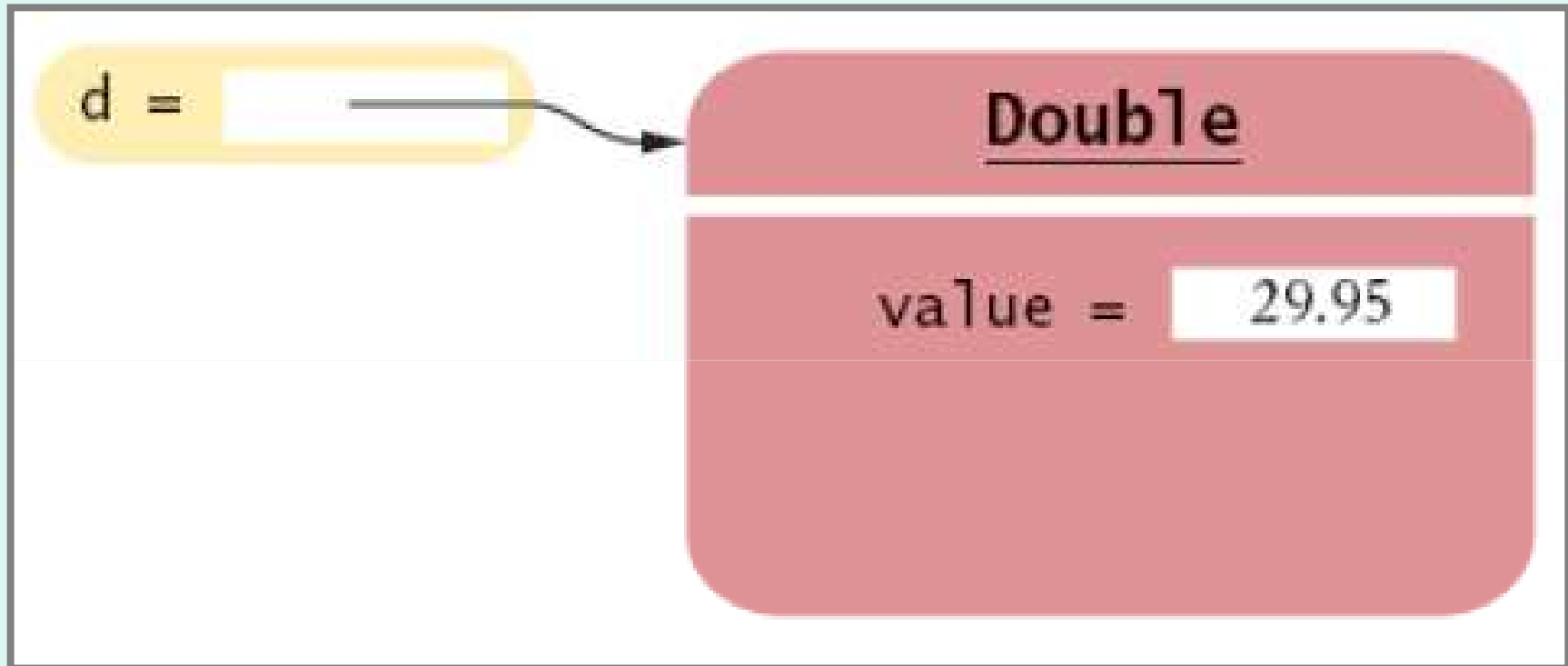
---

- A differenza degli array, le arraylists possono solo contenere elementi di tipo reference
- È necessario quindi utilizzare le cosiddette classi “wrapper” che permettono di convertire valori primitivi in corrispondenti valori di tipo riferimento:

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

*Continua...*

# Wrappers



# Wrappers

- Ci sono classi wrapper per ciascuno degli otto tipi primitivi

Primitive Type	Wrapper Class
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

# Auto-boxing

---

- Da Java 5.0, la conversione tra tipi primitivi e i corrispondenti tipi wrapper è automatica.

```
Double d = 29.95; // auto-boxing; equivale a
                  // Double d = new Double(29.95);

double x = d;      // auto-unboxing; equivale a
                  // double x = d.doubleValue();
```

*Continua...*

# Auto-boxing

---

- Auto-boxing opera anche all'interno delle espressioni aritmetiche

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

## Valutazione:

- auto-unbox `d` in un `double`
- aggiungi `1`
- auto-box il risultato in un nuovo `Double`
- Memorizza il riferimento nel wrapper `e`

# Auto-boxing

---

## Attenzione!

- **== è definito diversamente sui tipi primitivi e sul loro corrispettivo boxed**
- **==**
  - su interi significa uguaglianza di valori
  - su integer significa identità di riferimenti
- **Evitiamo l'uso di == sulle classi wrapper**

# Domanda

---

- Supponiamo che `data` sia un `ArrayList<Double>` di dimensione  $> 0$ .  
Come facciamo ad incrementare l'elemento all'indice 0?



# Risposta

---

- `data.set(0, data.get(0) + 1);`

# Array e *for* loops

---

- La soluzione tradizionale

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

# “for each”: un for generalizzato

---

- **Itera su tutti gli elementi di una collezione**
  - ad esempio sugli elementi di una array

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

*Continua...*

# “for each”

---

- Si applica nello stesso modo alle **ArrayLists**:

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

# “for each”

---

- **Equivalente al seguente loop tradizionale:**

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

# Sintassi

---

```
for (Type variable : collection)  
    statement
```

- **Esegue il corpo del ciclo su ciascun elemento della collezione**
- **La variabile è assegnata ad ogni ciclo all'elemento successivo**

# **“for each” – limitazioni**

---

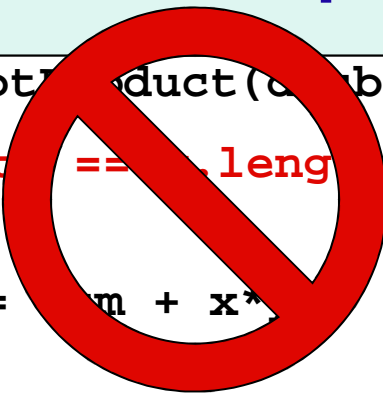
- **La sintassi è deliberatamente semplice**
- **Si applica solo ai casi più semplici di gestione di collezioni.**
- **Spesso è necessario utilizzare la sintassi tradizionale**

*Continua...*

# “for each” – limitazioni

- Non utilizzabile per scorrere due strutture all'interno dello stesso loop

```
public static double dotProduct(double[] u, double[] v)
{
    // assumiamo u.length == v.length;
    double sum = 0,
    for (x:u, y:v) sum = sum + x*y;
}
```



```
public static double dotProduct(double[] u, double[] v)
{
    // assumiamo u.length == v.length;
    double sum = 0,
    for (int i=0; i<u.length; i++) sum = sum + u[i]*v[i];
    return sum;
}
```

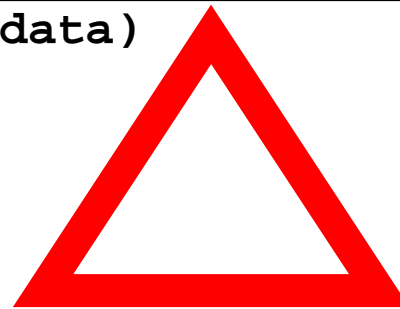
..



# “for each” – limitazioni

- Non sempre utilizzabile per inizializzazioni

```
public static void init(double[] data)
{
    int i = 0;
    for (x:data) { x = i*i; i++; }
}
```



```
public static void init(double[] data)
{
    for (int i = 0; i<data.length; i++)
        data[i] = i*i;
}
```

*Continua...*

# “for each” – limitazioni

---

- Iterazione su array bidimensionali
- tipicamente utilizziamo due cicli innestati: anche qui, il “foreach” non aiuta

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