

*Array*¹

➤ **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

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

□ 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).

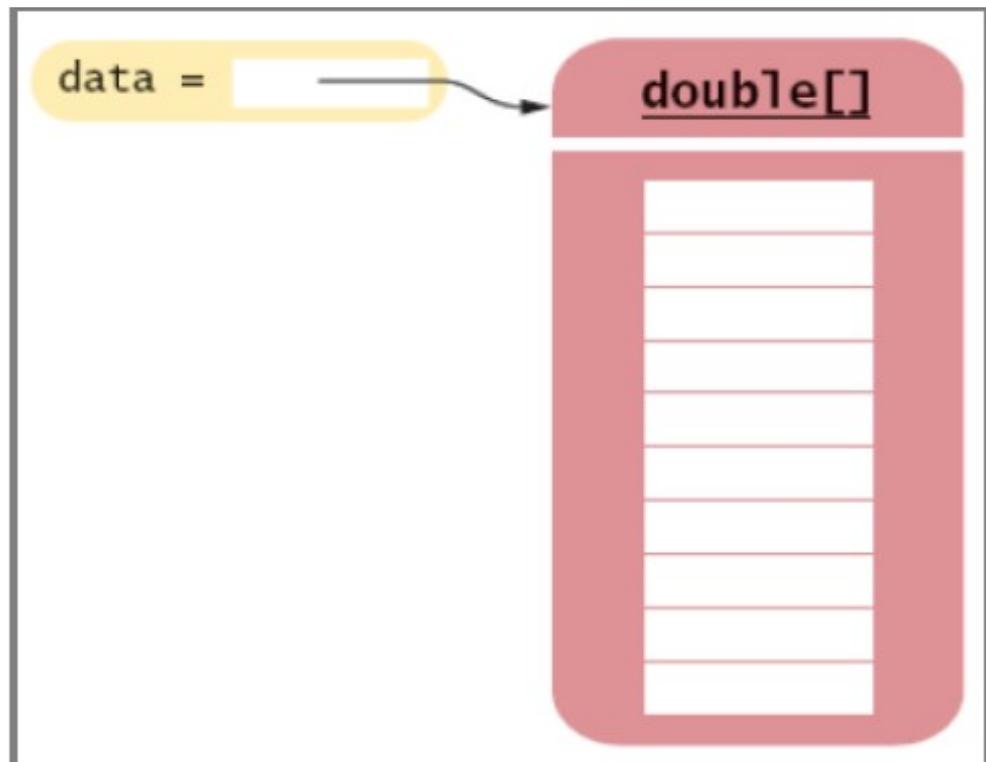


Figura 1:
Un riferimento
ad array e un array

¹ C. Horstmann, “Fondamenti di Programmazione e Java 2” cap.8 , 3a ed., Apogeo

- Usare [] per identificare un elemento di un array

`data[2] = 29.95;`

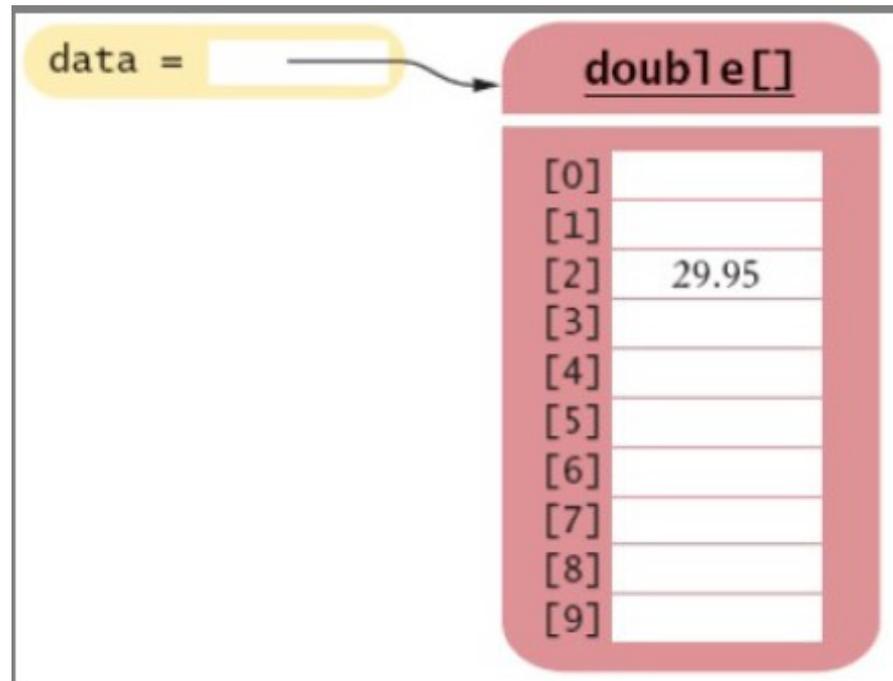


Figura 2
Memorizzare
un valore
in un Array

- Si accede agli elementi di un array tramite un indice di tipo intero, usando la notazione `nomeArray[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 **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**.
- Gli array hanno un limite pesante: *la loro lunghezza è fissa*.

Sintassi per la costruzione di un array: `new nomeTipo [lunghezza]`

Esempio:

```
new double[10]
```

Obiettivo:

Costruire un array con un **determinato** numero di elementi

Sintassi per accesso a elementi di array: `riferimentoAdArray[indice]`

Esempio:

```
data[2]
```

Obiettivo:

Accedere a un elemento in un array

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

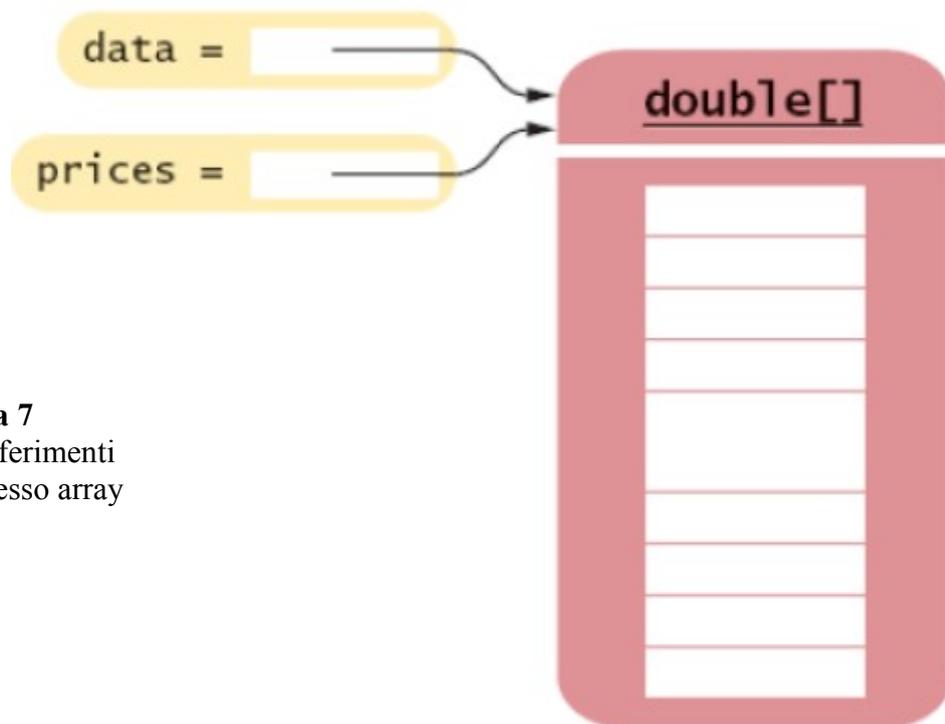


Figura 7
Due riferimenti
allo stesso array

Copiare array: clonare un array

□ Per copiare gli elementi di un array (**duplicare**) usate il **metodo clone**.

```
double[] data = new double[10];  
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 ad un altro.

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

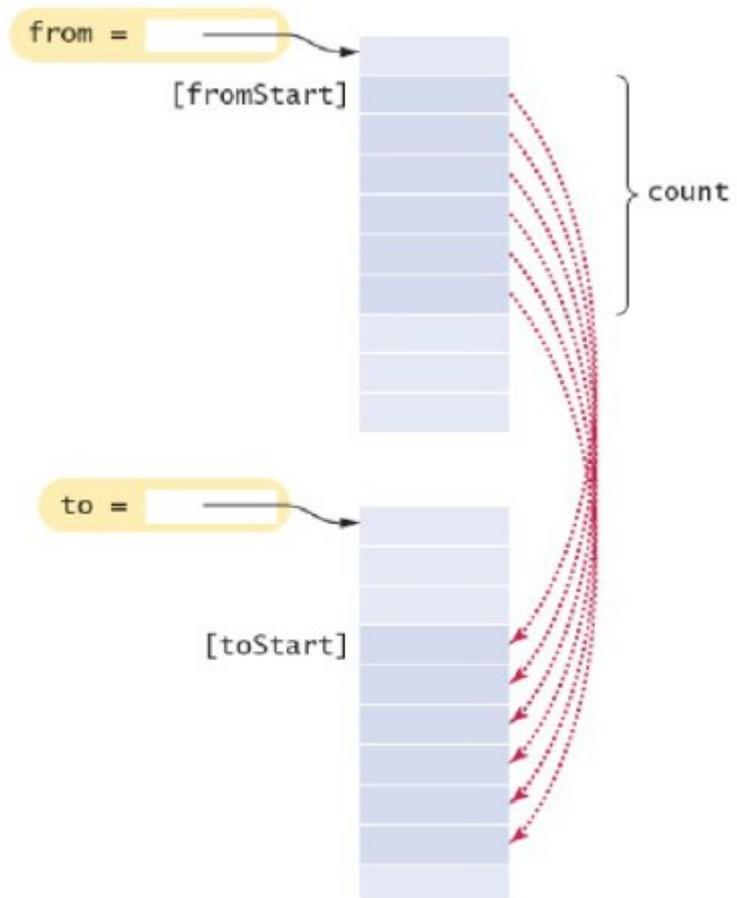


Figura 9
Il metodo
`System.arraycopy()`
per copiare

`arraycopy(Object src, int srcPos, Object dest, int destPos, int length)`

Inserire un elemento in un array

- Usate il metodo `System.arraycopy` per **inserire** un elemento in un array

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

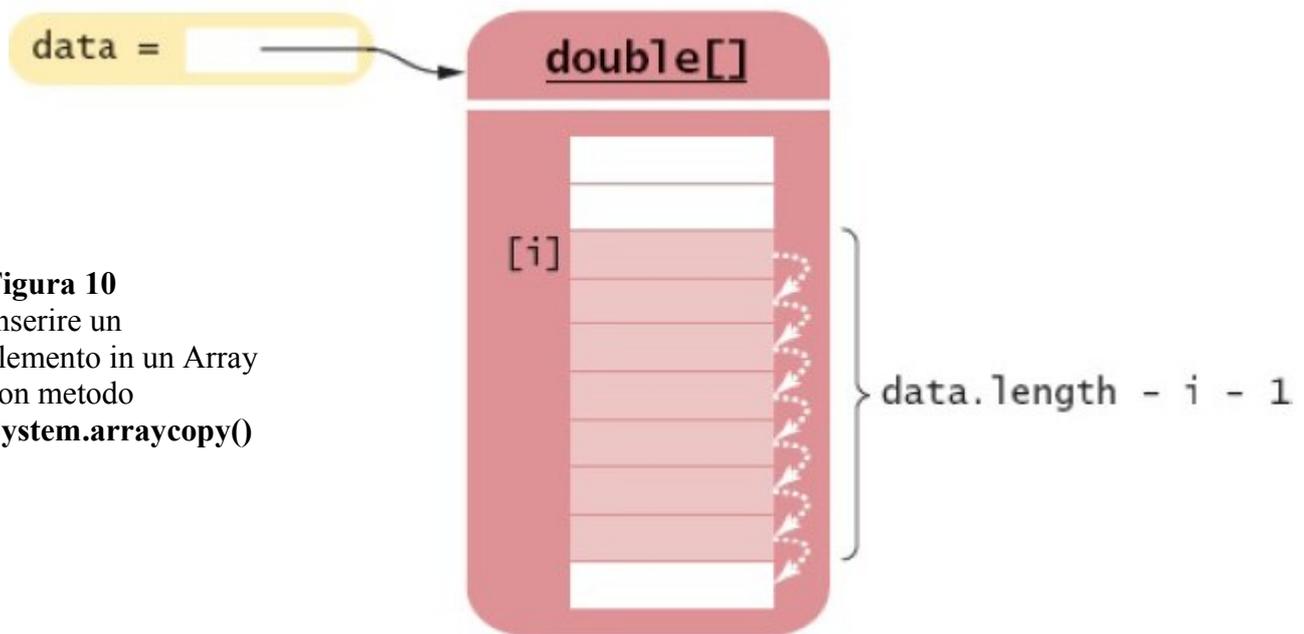


Figura 10
Inserire un
elemento in un Array
con metodo
`System.arraycopy()`

Rimuovere un elemento da un array

- Usate il metodo `System.arraycopy` per **eliminare** un elemento da un array

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

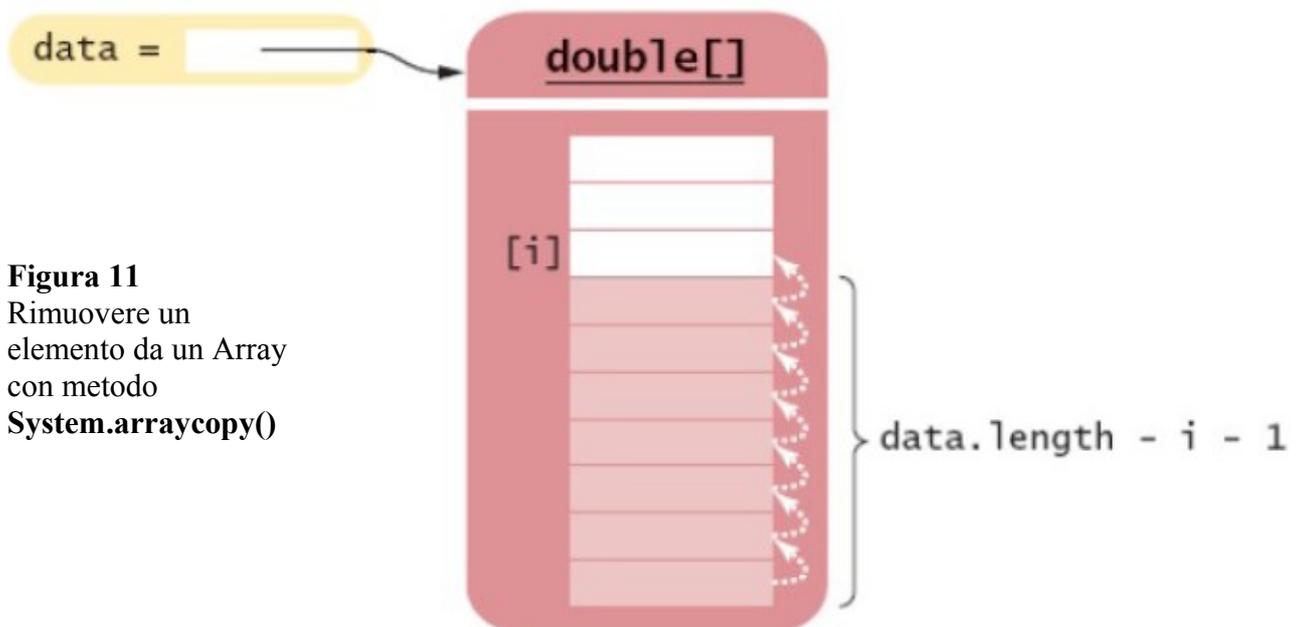


Figura 11
Rimuovere un
elemento da un Array
con metodo
`System.arraycopy()`

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. Create 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);
```

2. Memorizzare nella variabile array il riferimento al nuovo array

```
data = newData;
```

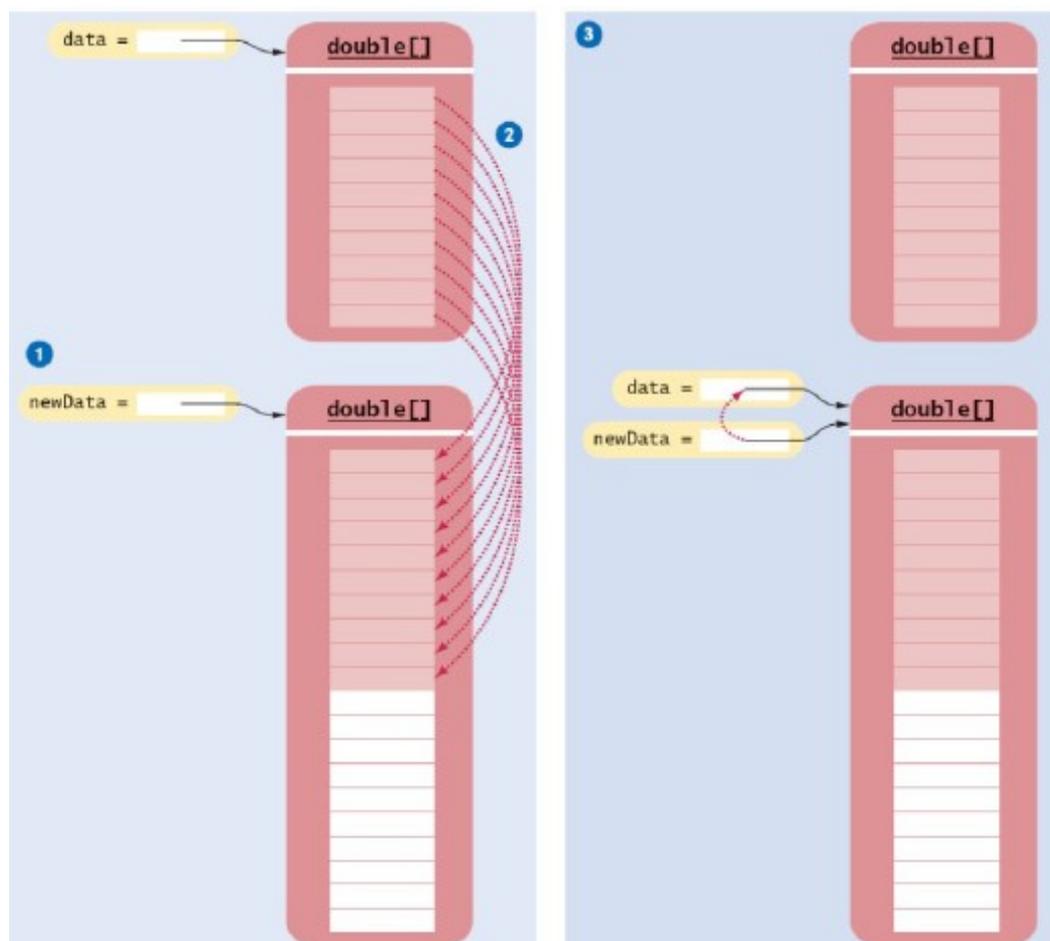


Figura 12
Far crescere un array
con metodo
`System.arraycopy()`

Trasformare array paralleli in array di oggetti

□ Una struttura dati è denominata “array paralleli” quando si usano *diversi array* per contenere i dati del problema e gli *elementi aventi lo stesso indice* nei diversi array sono tra loro *fortemente correlati*

- devono sempre contenere lo *stesso numero di elementi*
- in questo caso, rappresentano *diverse proprietà dello stesso oggetto*
- molte elaborazioni hanno bisogno di *utilizzare tutti gli array*, che devono quindi essere passati come parametri

Tutte le volte in cui il problema presenta una struttura dati del tipo “array paralleli”, si consiglia di *trasformarla in un array di oggetti*

- occorre realizzare la classe con cui costruire gli oggetti
- Risulta molto più facile scrivere il codice e, soprattutto, apportare modifiche

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

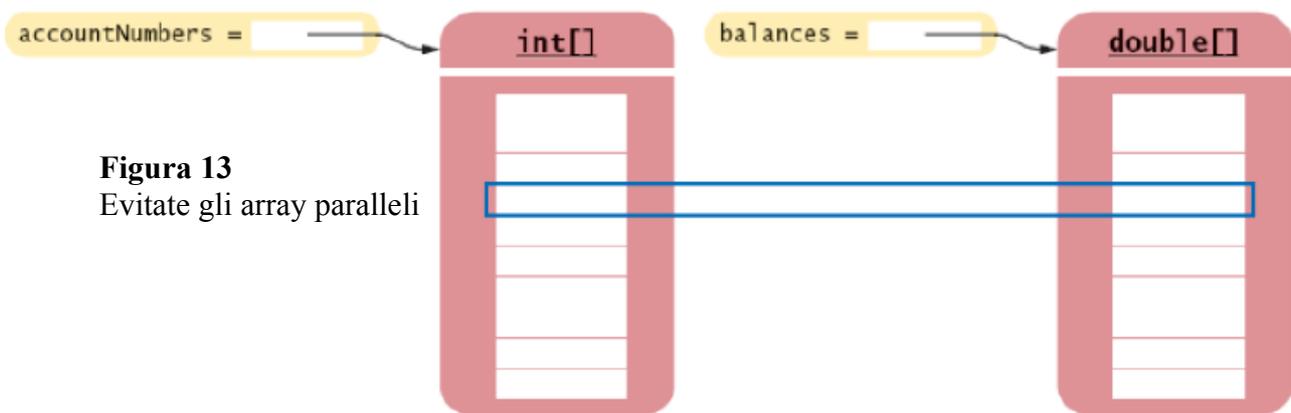


Figura 13
Evitate gli array paralleli

- Usare un **unico array di oggetti**

```
BankAccount[] accounts = new BankAccount [10];
```

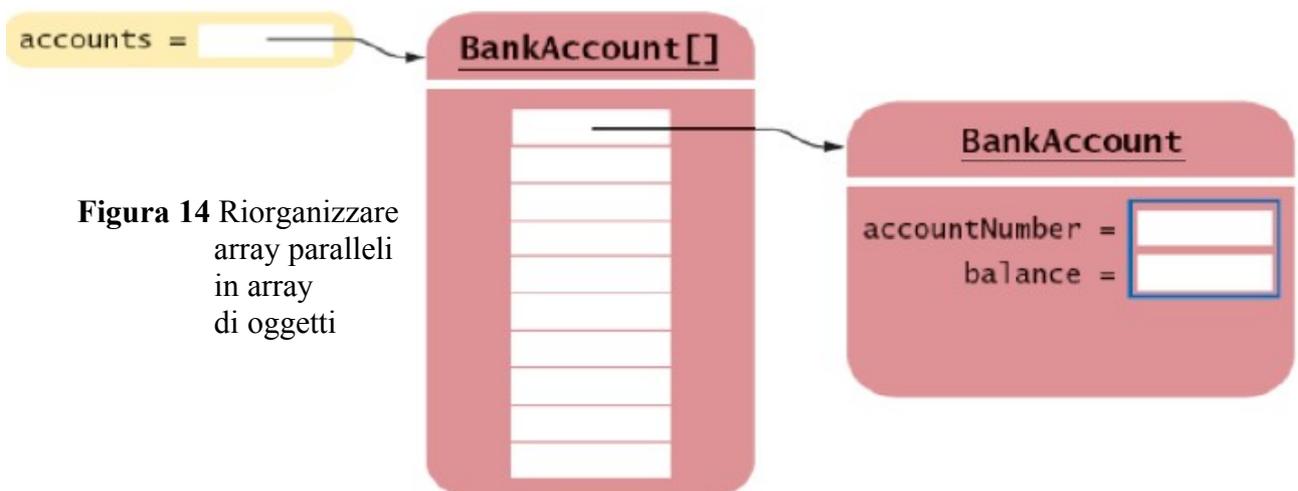


Figura 14 Riorganizzare array paralleli in array di oggetti

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 (con buono stile) 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;
```

- `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++;
```

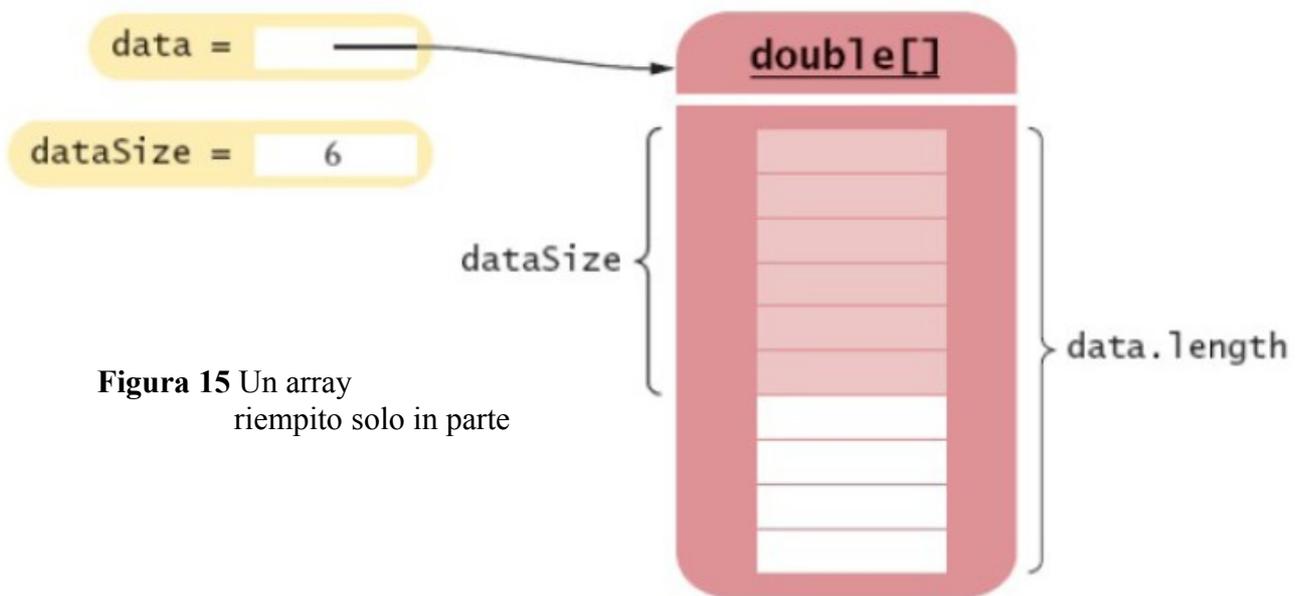


Figura 15 Un array riempito solo in parte

Uno dei primi worm di Internet

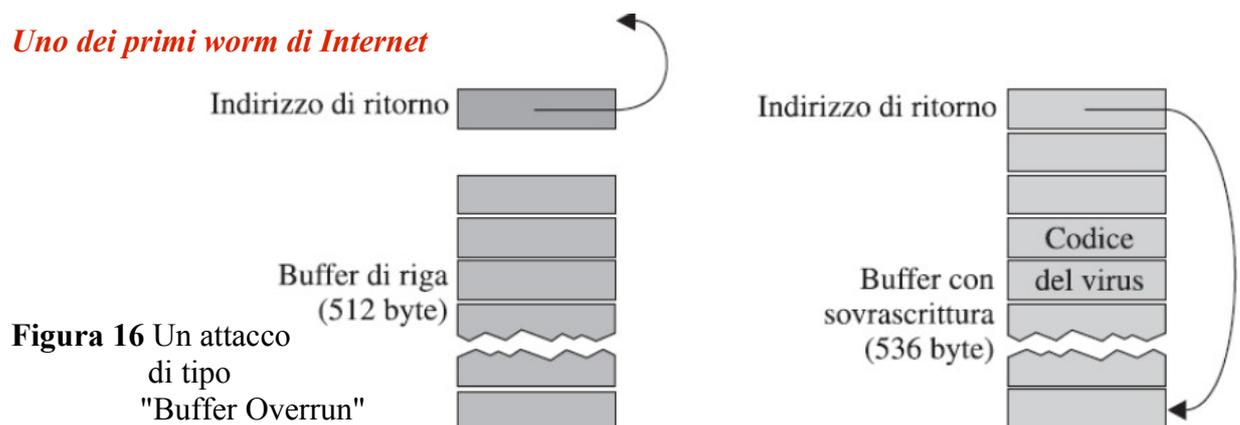


Figura 16 Un attacco di tipo "Buffer Overrun"

Vettori: array dinamici

- Un *array dinamico* o *vettore* può crescere e calare di dimensione in base alle necessità
 - La classe **ArrayList** (vettore o **lista sequenziale**) gestisce oggetti disposti in sequenza.
 - 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>();  
  
accounts.add(new BankAccount(1001));  
accounts.add(new BankAccount(1015));  
accounts.add(new BankAccount(1022));
```

Ispezionare gli elementi

- Per ispezionare gli oggetti contenuti nel vettore si usa il **metodo get** e **non** l'operatore `[]`
- 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);  
// estrae il terzo elemento della lista
```

- Accedere a un elemento non esistente è un errore.
- L'errore di limiti più frequente è il seguente:

```
int i = accounts.size();  
anAccount = accounts.get(i); // Error  
// valori possibili dell'indice 0..i-1
```

NB:

Anche la classe **Vector** come `ArrayList` (più recente) permette di istanziare **contenitori** di tipi poliformici “**estendibili**” e “**accorciabili**” **dinamicamente**.

Prima di JDK5 potevano contenere solo oggetti `Object`, in JDK5 (con l'introduzione di classi di tipi generici² o **Generics**) sono parametrici rispetto al tipo degli oggetti contenuti:

- **Vector** per costruire array estensibile, generico nel tipo `T` degli elementi (una sequenza di oggetti tipizzati solo in fase di definizione) con metodi sincronizzati (*synchronized*: senza accesso a risorse finché esse sono impegnate da altri metodi)
- **ArrayList** per costruire lista estensibile, generica nel tipo `T` degli elementi con metodi non sincronizzati (programmazione concorrente).

² Dopo gli opportuni controlli di consistenza tra tipi, il compilatore **cancella** (Erasure) tutte le informazioni relative ai tipi parametrici, e **sostituisce** (Translation) le relative occorrenze con il tipo di oggetto specifico.

Aggiungere elementi

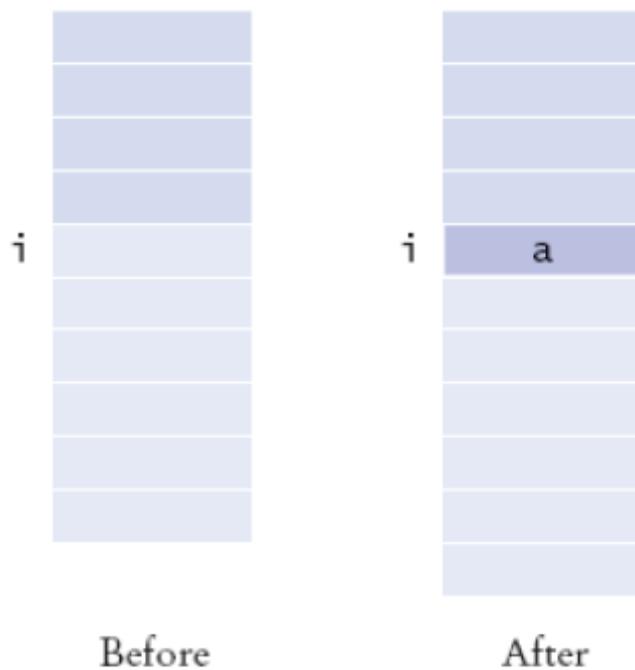
- Per **assegnare** un nuovo valore a un elemento esistente di un vettore si usa il **metodo set**:

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

- È possibile **inserire** un oggetto in una posizione intermedia all'interno di un vettore con il **metodo add**.

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

Figura 3
Aggiungere un elemento
in una posizione
intermedia
di un vettore: `accounts.add (i, a)`



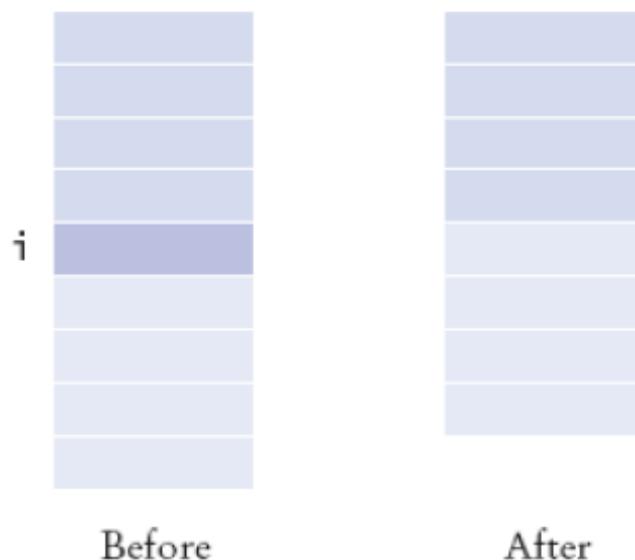
allunga di 1 la (parte utilizzata del) l'array

Rimuovere elementi

- Con il metodo **remove** si può **eliminare** l'elemento che si trova in una data posizione, sposta di una posizione all'indietro tutti gli elementi che si trovano dopo l'elemento rimosso e diminuisce di uno la dimensione del vettore.

Figura 4
Rimuovere un elemento
da una posizione
intermedia di un vettore:

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



Involucri

□ 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);
```

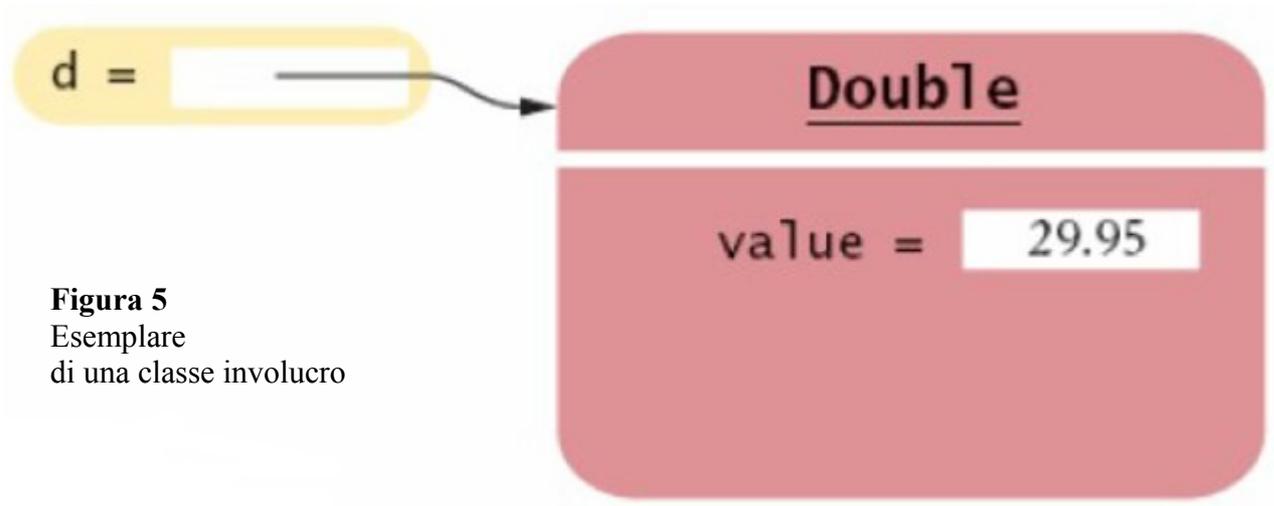


Figura 5
Esemplare
di una classe involucro

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

Tipo primitivo	Classe involucro
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, gli oggetti involucro vengono automaticamente “tolti dalla scatola” per generare valori di tipo primitivo, mediante un processo denominato “**auto-unboxing**”

```
Double d = 29.95; // auto-boxing
                // invece di Double d = new Double(29.95);
                // da primitivo nell'oggetto wrapper
```

```
double x = d; // auto-unboxing
              // invece di double x = d.doubleValue();
              // da oggetto wrapper in primitivo
```

Auto-boxing

□ Le conversioni automatiche funzionano anche all'interno di espressioni aritmetiche (infatti gli operatori possono trattare le classi wrapper come tipi base).

□ L'enunciato è valido e significa:

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

- 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

□ E' lecita la sintassi:

```
Integer i = 5;
i++;           // senza estrazione del campo con metodo
```

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 può leggere questo loop come
                    // "for each e in data" "per ogni e in data"
{
    sum = sum + e;
}
```

□ 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** 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();
}
```

- Rende più intuitivo e corto da scrivere il codice per scandire collezioni (array, liste etc..)

```
int v[]={1, 2, 3, 4, 5};
int s = 0;
for (int i: v) { s += i; } // scansione di array di interi
.....
```

```
Vector <Integer> vettore = new Vector<Integer>();
for(int i=0;i<10;i++) { // for normale
    vettore.add(i); // autoboxing ... invece di Integer(i)
}
for(Integer oggetto:vettore) { // ciclo for-each usato per scandire gli elementi
    System.out.println(oggetto); // unboxing ... invece di oggetto.intValue()
}
```

Sintassi del 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

Varargs



□ Con la versione JDK 1.5 (comunemente chiamata Tiger) è possibile definire metodi e/o costruttori che ricevano un **numero arbitrario di argomenti**.

□ L'unica limitazione è che gli argomenti devono essere dello stesso tipo.

```
public void metodoVar(int intero, String...argomenti)
```

□ Il compilatore tratterà gli argomenti variabili come se fossero un array *metodo equivalente considerato dal compilatore*:

```
public void metodoVar(int intero, String[]argomenti)
```

per tale motivo **non è possibile** fare l'overload del metodo utilizzando un array dello stesso tipo del parametro varargs → ci sarebbe un **errore** in compilazione in quanto il compilatore troverebbe una duplice definizione con la stessa segnatura **public void metodoVar(int intero, String[]argomenti)**

□ Si può fare l'overloading di un metodo che ha come parametri varargs:

```
public void metodoVar(int intero, String...argomenti)
```

```
public void metodoVar(int intero, Object...argomenti)
```

□ Esempio d'uso di un **numero arbitrario** di parametri, **ciclo for generalizzato** e **auto-boxing**

```
public class varTest {  
  
    void testArg(Object ... var) { // numero arbitrario di parametri  
                                // oggetti di qualsiasi tipo  
        for( Objects : var ) // ciclo for-each  
            System.out.println( s );  
    }  
    public static void main(String[] args) {  
  
        varTest x = new varTest();  
        x.testArg( "ciao", "io", "sono", "Matteo", 10, 20 ); // argomenti  
                                                            // di tipo String  
        System.out.println( ); // e auto-boxing  
                                // in Integer  
  
        Double num1 = 44.3, num2 = 66.8; // auto-boxing  
  
        x.testArg( num1, num2 ); // argomenti di tipo Double  
        System.out.println( );  
    }  
}
```

nb: Solo l'ultimo argomento può essere un vararg

Semplici algoritmi per vettori (array dinamici)

Contare

□ 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 {  
  
    /**  
     * Conta il numero di conti bancari aventi saldo maggiore o uguale al valore indicato.  
     * @param atLeast il saldo minimo perché un conto venga conteggiato  
     * @return il numero di conti aventi saldo maggiore o uguale al valore indicato.  
     */  
  
    public int count (double atLeast) {  
  
        int matches = 0;  
        for (BankAccount a : accounts) {  
  
            if (a.getBalance() >= atLeast) matches++; // trovato  
        }  
        return matches;  
    }  
    ...  
    private ArrayList<BankAccount> accounts;  
}
```

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 {  
  
    /**  
     * Verifica se la banca contiene un conto con il numero indicato.  
     * @param accountNumber il numero di conto da cercare  
     * @return il conto con il numero indicato, oppure null se tale conto non esiste  
     */  
  
    public BankAccount find (int accountNumber) {  
  
        for (BankAccount a : accounts) {  
  
            if (a.getAccountNumber() == accountNumber) // trovato  
                return a;  
        }  
        return null; // nessuna occorrenza in tutta l'array list  
    }  
    ...  
}
```

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);
        ...

public class Bank {

    /**
     Restituisce il conto bancario avente il saldo maggiore.
     @return il conto con il saldo maggiore, oppure null se la banca non ha conti
     */

    public BankAccount getMaximum() {

        if (accounts.size() == 0)
            return null;
        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;
    }
    ...
}
```