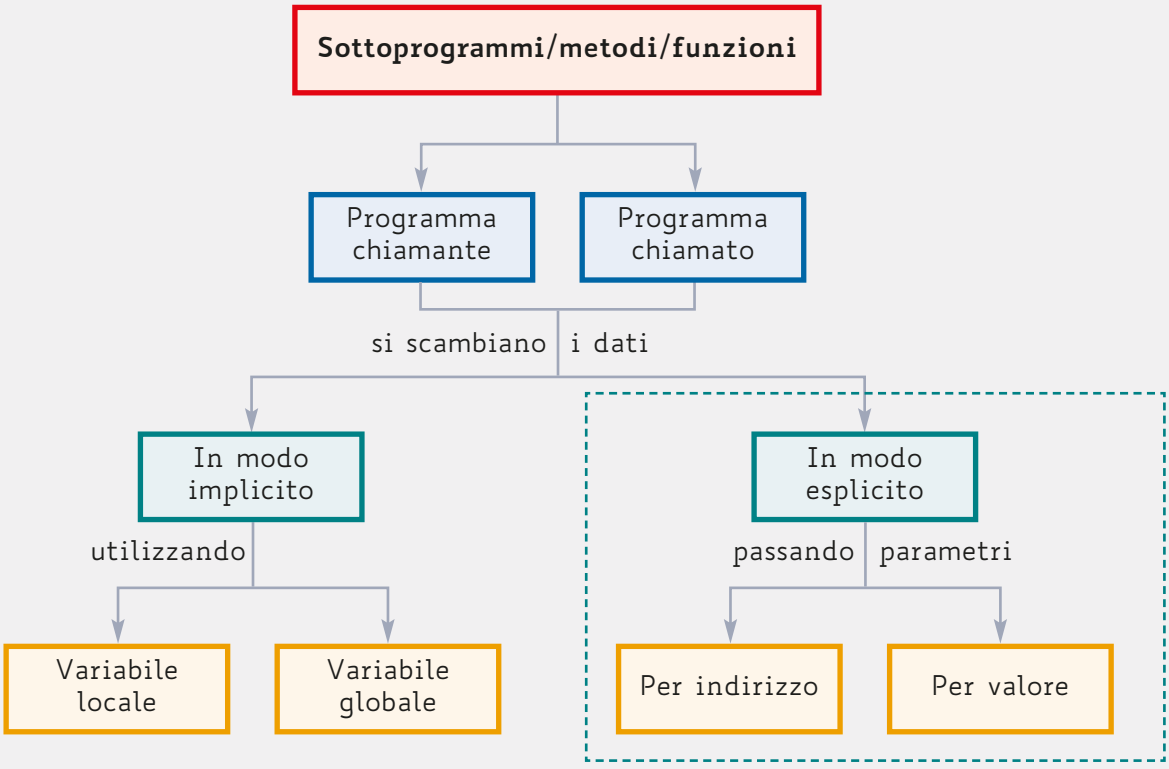
2 Lo scambio dei dati e le regole di visibilità

IN QUESTA LEZIONE IMPAREREMO...

- la differenza tra variabili locali e globali
- le modalità di scambio delle variabili

MAPPA CONCETTUALE

didattica inclusiva



Premessa: alcune doverose precisazioni (e anticipazioni)

Abbiamo finora utilizzato la parola metodo come sinonimo di funzione, ma a questo punto della trattazione è necessario aprire una parentesi per fare un po' di chiarezza e descrivere tutti i componenti del paradigma OOP per avvicinarsi all'uso corretto dei linguaggi che rientrano in questo paradigma.
Riprendiamo un qualunque programma scritto sinora, del quale riportiamo la struttura generale:

```
class NomeClasse
{
 // contenuto della classe
   <dichiarazione attributi>                    // variabili della classe o degli oggetti
   <dichiarazione metodi>                       // operazioni sulle variabili
   public static void main()
   {
   // istruzioni della classe
   // dichiarazione di variabili
   // creazione di oggetti
     ...
   }
}
```

La classe è l'elemento fondamentale del paradigma OOP e gli oggetti sono le istanze delle classi, cioè gli elementi creati a partire dalle classi che, dopo la loro creazione, vivono di vita propria, interagiscono tra loro utilizzando i metodi per scambiarsi informazione ed effettuare operazioni, fino a ottenere il risultato desiderato.

Un programma Java è quindi costituito da oggetti di vario tipo che interagiscono tra loro e viene avviato da una particolare classe che contiene il metodo main().

Per ogni programma esiste una sola classe che contiene il metodo main(): come vedremo, tutte le altre classi solo "semplicemente" composte da attributi e metodi.
Gli attributi non sono altro che "variabili che contengono i dati", cioè le caratteristiche che differenziano i singoli elementi della classe, cioè gli oggetti.

Possiamo dire che la classe è lo "stampino" con il quale vengono creati gli oggetti, che "nascono" tutti uguali tra loro e sui quali si può operare solo attraverso le operazioni descritte con i metodi.

ESEMPIO

CLASSE RETTANGOLO

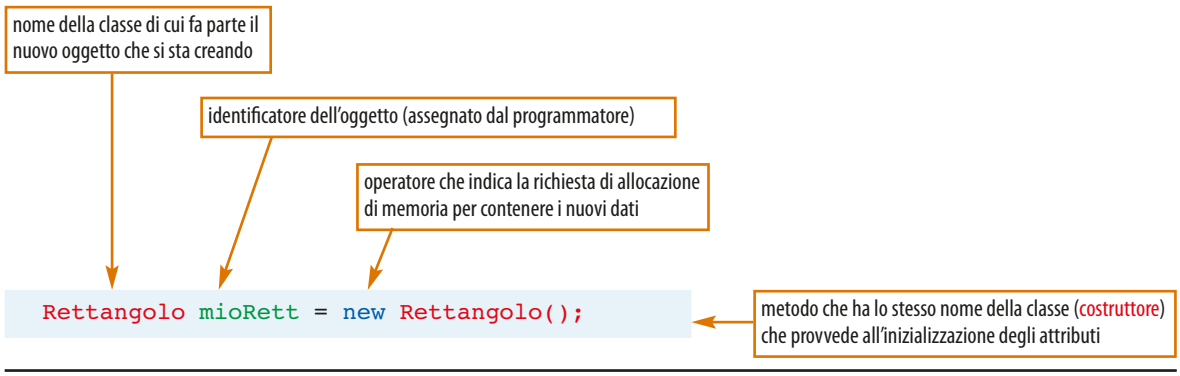
```
int base                    // attributi
int altezza
int perimetro()             // metodi
{
  return (base + altezza)*2
}
int area()
{
  return (base * altezza)
}
```

nome della classe

i dati dei singoli oggetti vengono memorizzati negli attributi, o variabili di istanza (anche chiamati campi)

le azioni/operazioni che possono essere effettuate sui dati sono definite nei metodi di istanza

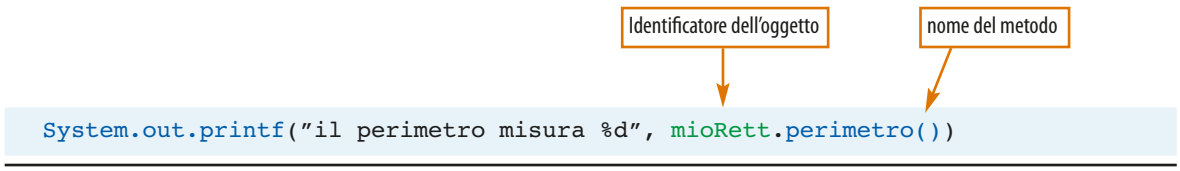
I singoli oggetti sono elementi della classe e vengono associati a una variabile che come tipo ha proprio il nome della classe: l'operazione che permette di creare un'istanza della classe, cioè un oggetto, ha la seguente struttura:



Alla creazione di un nuovo oggetto con la new() viene riservata nella memoria una spazio sufficiente a mantenere una "copia" degli attributi previsti dalla classe di appartenenza (attributi di istanza), mentre NON viene riservata memoria per i metodi, in quanto rimangono condivisi da tutti gli oggetti della classe, assieme a eventuali attributi comuni (attributi di classe).
Quando viene istanziato un oggetto, le sue variabili di istanza sono automaticamente inizializzate a valori di default, che dipendono dal tipo della variabile:

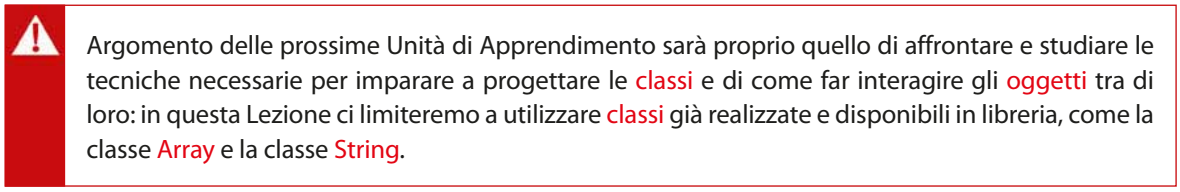
```
int : 0
boolean: false
String: null
```

Vedremo come personalizzare il metodo costruttore per inizializzare "a piacere" il valore degli attributi, in quanto gli oggetti si differenziano tra loro proprio per il loro valore.
Attraverso i metodi è quindi possibile operare i cambianti al valore degli attributi e su un oggetto è possibile richiamare (invocare) solo i metodi descritti all'interno della sua classe di appartenenza!
Un esempio di invocazione di un metodo su di un oggetto è il seguente:



La struttura <nomeOggetto>.<nomeMetodo> è nota come dot notation.

All'interno del metodo le diverse istruzioni, oltre che a modificare il valore di dati presenti negli attributi dell'oggetto sul quale sono invocati, possono anche creare nuovi oggetti e "collaborare" con questi inviando messaggi e ricevendo risposta sempre attraverso i metodi.



Argomento delle prossime Unità di Apprendimento sarà proprio quello di affrontare e studiare le tecniche necessarie per imparare a progettare le classi e di come far interagire gli oggetti tra di loro: in questa Lezione ci limiteremo a utilizzare classi già realizzate e disponibili in libreria, come la classe Array e la classe String.

Una puntualizzazione doverosa deve essere fatta per il metodo main(): questo metodo viene "attivato" automaticamente senza essere associato a un oggetto, ma semplicemente al caricamento della classe che lo contiene.

In un programma a oggetti deve essere presente una sola classe che contiene il metodo main().

Visibilità delle variabili e degli oggetti

Abbiamo visto come nella scrittura dei programmi, Java lascia molta libertà nella definizione delle variabili: riportiamo, per esempio, quattro codici alternativi di un semplice programma che calcola l'elevamento a potenza conoscendo la base e l'esponente.
In questi primi due codici effettuiamo direttamente il calcolo della potenza all'interno del main(), con un semplice ciclo a conteggio che utilizza un contatore *n* intero, definito in due "istanti differenti":
a all'inizio del metodo, assieme alle altre variabili;
b all'interno del ciclo, cioè solo al momento del suo utilizzo.

Nel primo caso la variabile può essere utilizzata in tutto il metodo, è cioè visibile e accessibile da tutte le istruzioni del main(), mentre nel secondo caso la sua visibilità si riduce alla sola istruzione nella quale è dichiarata: l'ambiente di visibilità è evidenziato in giallo nelle seguenti figure che riportano il codice del programma.

DICHIARAZIONE LOCALE AL METODO

```
public class Potenza1 {
  public static void main(String[] arg) {
    int n, base, esponente, potenza;
    potenza = 1;
    base = 2;
    esponente = 5;
    for (n = 1; n <= esponente; n++)
      potenza = potenza * base;
    System.out.println("La potenza e': " + potenza);
  }
}
```

DICHIARAZIONE LOCALE ALL'ISTRUZIONE

```
public class Potenza2 {
  public static void main(String[] arg) {
    int base, esponente, potenza;
    potenza = 1;
    base = 2;
    esponente = 5;
    for (int n = 1; n <= esponente; n++)
      potenza = potenza * base;
    System.out.println("La potenza e': " + potenza);
  }
}
```

Riscriviamo ora il programma introducendo la modularità e il riutilizzo, cioè definendo un metodo eleva() che effettua la specifica operazione e richiamandolo dal main() gli passa i parametri attuali sul quale deve operare: ne scriviamo due versioni, dove nuovamente definiamo il contatore *n* intero in due "istanti differenti":
a nel nuovo metodo eleva();
b all'interno della classe, fuori da ogni metodo.

DICHIARAZIONE LOCALE

```
public class Potenza3 {
  public static int eleva(int b, int esp) {
    int potenza = 1;
    for (int n = 1; n <= esp; n++)
      potenza = potenza * b;
    return potenza;
  }
  public static void main(String[] arg) {
    int base, esponente, potenza;
    potenza = 1;
    base = 2;
    esponente = 5;
    potenza = eleva(base, esponente);
    System.out.println("La potenza e': " + potenza);
  }
}
```

DICHIARAZIONE GLOBALE

```
public class Potenza4 {
  static int potenza = 1;
  public static void eleva(int b, int esp) {
    for (int n = 1; n <= esp; n++)
      potenza = potenza * b;
  }
  public static void main(String[] arg) {
    int base, esponente;
    base = 2;
    esponente = 5;
    eleva(base, esponente);
    System.out.println("La potenza e': " + potenza);
  }
}
```

Nel primo caso la variabile può essere utilizzata solo all'interno del metodo, è cioè visibile localmente, mentre nel secondo caso la sua visibilità è estesa a tutti i metodi della classe, è, quindi, una variabile condivisa da tutti i metodi: essa prende il nome di variabile di classe.

Per i metodi non statici esiste anche una visibilità intermedia, cioè la variabile di istanza, visibile a ogni singola istanza della classe; a differenza della variabile di classe (static), che è unica per tutta la classe, della variabile di istanza viene generata una copia nell'area dati di ogni singolo oggetto che viene creato.

Passaggio dei parametri: modalità e osservazioni

Abbiamo visto la struttura di un metodo e ora facciamo un'osservazione sulla numerosità dei suoi attributi:
a in ingresso possiamo avere da 0 a *n* attributi;
b in uscita posiamo avene void, cioè nulla, oppure un attributo di ritorno.

Questa situazione si presenta sia nei linguaggi OOP sia nelle funzioni dei linguaggi imperativi.

Il fatto di avere un solo parametro d'uscita "sembrerebbe" limitare la funzionalità dei metodi ai casi in cui questi devono "ritornare" un unico risultato, e in parte questa affermazione è vera.

ESEMPIO

Supponiamo di voler scambiare il contenuto di due variabili: in questo caso anche se le operazioni sono molto semplici ci accorgeremmo di avere bisogno come parametri di ritorno entrambe le variabili, altrimenti solo il contenuto di una di loro verrebbe modificato!

I linguaggi di programmazione funzionali, per ovviare a questa limitazione, hanno introdotto un secondo meccanismo di passaggio dei parametri a una funzione, il passaggio per indirizzo, che permette di far operare direttamente la funzione sulle variabili del programma chiamante, passandogli come parametro non il loro valore, cioè una copia del dato, ma l'indirizzo della RAM nella quale è memorizzata la variabile e, quindi, mettendola in "comune" con la funzione.

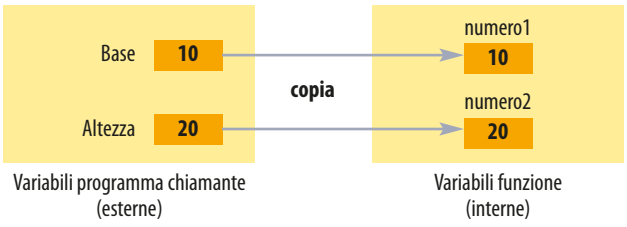
In Java, invece, il passaggio dei parametri avviene solo per valore, e quindi al metodo viene passato il valore della variabile che viene copiato in una variabile locale del metodo che, come sappiamo, "sparisce" alla fine dell'esecuzione del metodo stesso.

Quindi nessuna modifica nel metodo sui parametri formali cambierà i valori dei parametri attuali passati come argomenti: vediamo un breve descrizione che mostra le differenze tra le due modalità:
- per valore o copia (by value): si trasferisce il valore del parametro attuale;
- per riferimento (by reference): si trasferisce un riferimento (indirizzo di memoria) al parametro attuale.

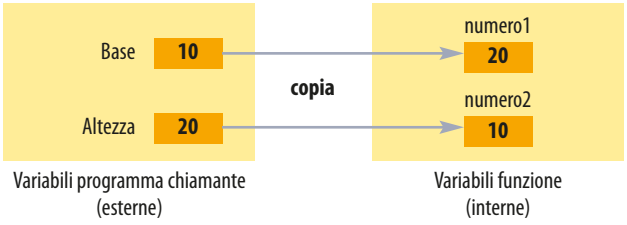
Passaggio per valore

Nel meccanismo di passaggio per valore, il programma chiamante comunica al programma chiamato il valore che è contenuto in quel momento in una sua variabile (parametro attuale).
Il programma chiamato copia tale valore all'interno di una variabile locale opportunamente predisposta (stabilita nell'elenco dei parametri formali).

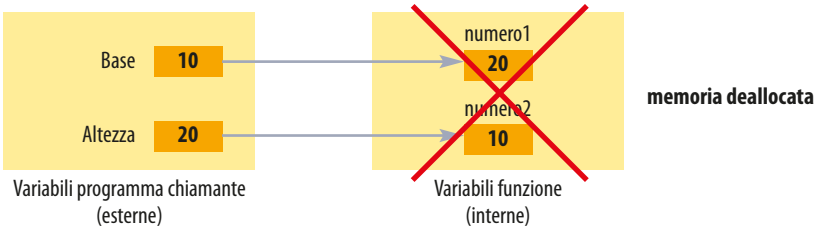
In questo momento sono quindi presenti in memoria due variabili con lo stesso valore!



Il programma chiamato può operare su tale valore, può anche modificarlo, ma tali modifiche riguarderanno solo la copia locale su cui sta lavorando (nel nostro esempio, scambia i valori).



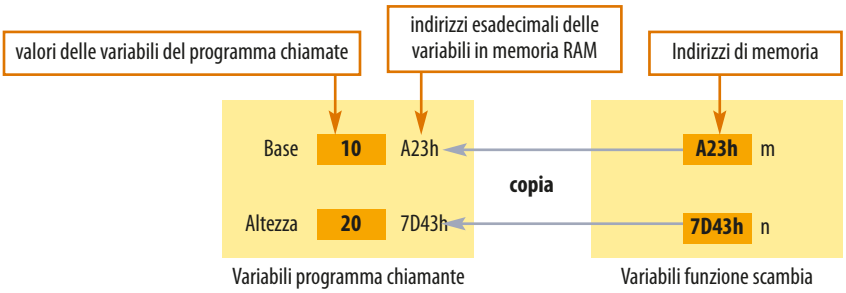
Terminato il sottoprogramma, il parametro formale viene deallocato e viene ripristinato il parametro attuale con il valore che esso conteneva prima della chiamata al sottoprogramma.



Alla ripresa dell'esecuzione del programma chiamante, non si sono riscontrati effetti dalla chiamata alla funzione: tutto è rimasto come prima della sua esecuzione.

Passaggio per riferimento

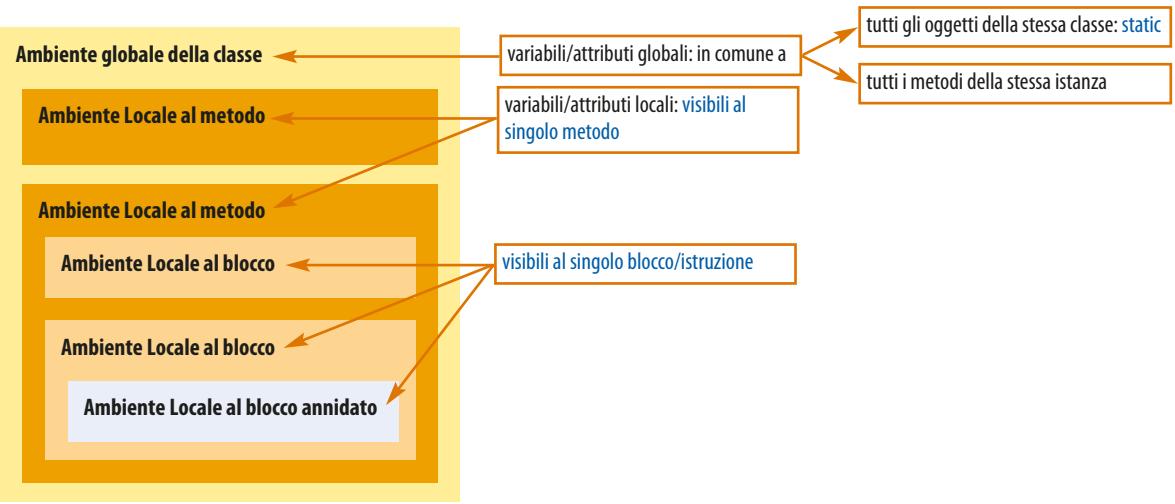
Il passaggio dei parametri è per riferimento (o per indirizzo) se il chiamante comunica al chiamato l'indirizzo di memoria di una determinata variabile (parametro attuale).



Il programma chiamato memorizza in una sua variabile locale l'indirizzo che riceve come parametro.

Le operazioni fatte nel sottoprogramma sono quindi eseguite nelle posizioni di memoria: ogni modifica effettuata tramite la variabile locale si ripercuoterà su quella del programma chiamante e, quindi, i dati nelle variabili risultano modificati anche dopo la terminazione del sottoprogramma.

Non essendo presente in Java questa modalità, la soluzione che noi adotteremo sarà quella di sfruttare le regole di visibilità delle variabili, che riportiamo schematicamente nella seguente figura:



La possibilità di avere aree di memoria condivise (ambiente condiviso) permette ai sottoprogrammi di comunicare in modo implicito: un metodo (produttore) può scrivere un dato in tale area e il metodo cliente lo può utilizzare.

Risolviamo il problema precedente di effettuare lo scambio tra due variabili sfruttando proprio la visibilità: dichiariamo num1 e num2 come variabili globali e utilizziamo proprio quelle sia nei metodi sia nel main(), senza effettuare alcun passaggio dei parametri.

```
import java.util.Scanner;
public class Scambia {
  static int num1, num2;  // area globale

  static void scambia() {
    int tempo;            // variabile locale
    tempo = num1;
    num1 = num2;
    num2 = tempo;
}
```

```
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    System.out.print("\nInserisci valore per num1 :");
    num1 = in.nextInt();
    System.out.print("Inserisci valore per num2 :");
    num2 = in.nextInt();
    scambia();       // chiamata al metodo senza parametri
    System.out.println("num1 contiene : " + num1);
    System.out.println("num2 contiene : " + num2);
  }
}
```

Un esempio completo con le variabili globali

Risolviamo adesso, a titolo di esempio, un problema su come gestire le variabili e scambiare i valori tra le funzioni utilizzando sia l'ambiente locale che l'ambiente globale.

✔ PROBLEMA SVOLTO PASSO PASSO

✔ Il problema

Calcola la temperatura media di due città prelevando alcune misurazioni nelle diverse ore del giorno: elabora una soluzione dove un sottoprogramma acquisisce i dati e li memorizza nelle variabili globali, mentre un secondo sottoprogramma esegue le medie e le visualizza sullo schermo.

✔ L'analisi e la strategia risolutiva

Non sapendo quante sono le letture, è necessario accumulare per ogni città le temperature inserite e contemporaneamente contare il numero di letture, in modo da poter calcolare la media al termine delle operazioni di input.
Predisponiamo quindi alcune variabili globali (tempera1, tempera2, letture1, letture2) e chiediamo all'utente, nella fase di input, di inserire la città desiderata (codificata con un intero compreso tra 0 e 2) e, successivamente, la lettura della temperatura; inserendo il numero 0 termina la fase di input.

✔ La codifica in linguaggio di programmazione

Linguaggio Java

Definizione della classe e il metodo leggiDati()

```
import java.util.Scanner;
public class Temperature{
  static double tempera1, tempera2;// area globale
  static int letture1, letture2;
  static void leggiDati(){      // senza parametri
    Scanner in = new Scanner(System.in);
    int citta;
    double dato;
    do{
      System.out.print("Digita il codice citta': ");
      citta = in.nextInt();
      if (citta != 0)    {
        System.out.print("inserisci temperatura: ");
        dato = in.nextDouble();
        switch (citta){
          case 1:
            tempera1 = tempera1 + dato;
            letture1 = letture1 + 1;    break;
          case 2:
            tempera2 = tempera2 + dato;
            letture2 = letture2 + 1;    break;
        }
      }
    }while(citta!=0);
  }
```

Il metodo elaboraDati() e il main()

```
  static void elaboraDati(){ //header senza parametri
    double media1, media2;
    media1 = tempera1 / letture1;
    media2 = tempera2 / letture2;
    System.out.printf("media 1 vale:%.2f%n", media1);
    System.out.printf("media 2 vale:%.2f%n", media2);
  }
    public static void main(String[] args){
    leggiDati();
    elaboraDati();
  }
}
```

✔ L'esecuzione del programma

```
Digita il codice citta': 1
inserisci temperatura: 17
Digita il codice citta': 2
inserisci temperatura: 24
Digita il codice citta': 1
inserisci temperatura: 25
Digita il codice citta': 0
media 1 vale:21,00
media 2 vale:24,00
```

Il codice sorgente di questi programmi lo trovi nel file Temperature.java.

L'utilizzo di aree condivise come meccanismo per la condivisione di dati tra processi è "funzionante", ma non è consigliabile per realizzare lo scambio di dati tra funzioni, sostanzialmente per i due motivi seguenti:

- non è possibile utilizzare più volte la funzione, dato che lavora solo su identificatori di variabili ben definite;
- non e possibile riutilizzare le funzioni in altri progetti, in quanto si è vincolati nell'avere la medesima area dati.

Per superare queste limitazioni bisogna fare in modo che le funzioni siano autonome, cioè siano dei black box che comunicano con il programma chiamante solo mediante una interfaccia ben definita: si realizzano funzioni autonome mediante la modalità esplicita di passaggio dei parametri.