

SUPSI

Sottoprogrammi

Loris Grossi, Fabio Landoni, Andrea Baldassari

Contenuto realizzato in collaborazione con: T. Leidi, A.E. Rizzoli, S. Pedrazzini

Fondamenti di Informatica

Bachelor in Ingegneria Informatica

Obiettivo

Comprendere la dichiarazione e l'utilizzo delle procedure e delle funzioni all'interno di programmi Java.

Obiettivi della lezione:

- Conoscere il concetto di sottoprogramma, procedura e funzione.
- Conoscere la dichiarazione e l'utilizzo di procedure e funzioni.
- Conoscere la differenza tra parametri formali e parametri attuali.
- Conoscere l'uso della keyword `return` nei sottoprogrammi.
- Conoscere il concetto di sovraccaricamento dei sottoprogrammi.
- Conoscere la gestione della memoria nei sottoprogrammi e il passaggio dei parametri con copia del valore e con copia del riferimento.
- Conoscere come specificare parametri alla procedura `main`.
- Conoscere la dichiarazione e l'utilizzo delle variabili globali.
- Conoscere il ciclo di vita delle variabili e la loro visibilità.

Blocchi di codice

Un blocco è una struttura per **organizzare una sequenza d'istruzioni**.

```
{  
    int nuovoValore = 2;  
    nuovoValore++;  
    System.out.println(nuovoValore);  
}
```

Un blocco può essere **vuoto**.

In un blocco si possono **dichiarare variabili locali**.

Una variabile locale è visibile **solo all'interno del blocco in cui è dichiarata**. Essa è accessibile, utilizzando l'identificatore, dal momento in cui viene dichiarata e fino alla fine del blocco in cui essa è contenuta.

Istruzioni di selezione in Java

```
if (condizione1) {  
    sequenzaIstruzioni1;  
} else if (condizione2) {  
    sequenzaIstruzioni2;  
} else if (condizione3) {  
    sequenzaIstruzioni3;  
} else {  
    sequenzaIstruzioni4;  
}
```

```
switch (espressione) {  
case valore1:  
    sequenzaIstruzioni10;  
    break;  
case valore2:  
    sequenzaIstruzioni11;  
    break;  
case valore3:  
    sequenzaIstruzioni12;  
    break;  
default:  
    sequenzaIstruzioni13;  
    break;  
}
```

Istruzioni di ripetizione in Java

```
while (condizione1) {  
    sequenzaIstruzioni1;  
}
```

Eseguito **zero o più** volte.

```
do {  
    sequenzaIstruzioni2;  
} while (condizione2);
```

Eseguito **una o più** volte.

```
for (int i = valoreIniziale; i < valoreFinale; i++) {  
    sequenzaIstruzioni3;  
}
```

Eseguito un **numero fisso** di volte.

Riutilizzo di parti di codice

Vi confrontate con il seguente problema: avete una parte di codice che volete **riutilizzare più volte** all'interno del vostro programma. Come fate?

```
int x = 5;
int maxX = 100;
int roundsX = 20;
for (int i = 0; i < roundsX; i++) {
    if (x < maxX) {
        x += x;
    } else {
        break;
}
}
```

Esempio: volete eseguire il codice sopra per tre variabili `x`, `y` e `z` (invece che solo `x`). Alla fine volete visualizzare la somma dei valori delle tre variabili.

Soluzione copia/incolla: efficace?

```
int x = 5; int maxX = 100; int iterazioniX = 20;
for (int i = 0; i < iterazioniX; i++)
    if (x < maxX)
        x += x;
    else
        break;

int y = 4; int maxY = 80; int iterazioniY = 15;
for (int i = 0; i < iterazioniY; i++)
    if (y < maxY)
        y += y;
    else
        break;

int z = 6; int maxZ = 115; int iterazioniZ = 30;
for (int i = 0; i < iterazioniZ; i++)
    if (z < maxZ)
        z += z;
    else
        break;

System.out.println(x + y + z);
```

... ma se avessimo dovuto ripeterlo per cento variabili?

Soluzione con sottoprogramma

```
int x = 5;
int maxX = 100;
int iterazioniX = 20;
x = calcola(x, maxX, iterazioniX);

int y = 4;
int maxY = 80;
int iterazioniY = 15;
y = calcola(y, maxY, iterazioniY);

int z = 6;
int maxZ = 115;
int iterazioniZ = 30;
z = calcola(z, maxZ, iterazioniZ);

System.out.println(x + y + z);
```

Soluzione con sottoprogramma e un array

```
int[] valori = { 5, 4, 6 };
int[] max = { 100, 80, 115 };
int[] iterazioni = { 20, 15, 30 };

int tot = 0;
for (int i = 0; i < valori.length; i++) {
    tot += calcola(valori[i], max[i], iterazioni[i]);
}
System.out.println(tot);
```

Soluzione: la funzione in Java

```
private static int calcola(int p, int maxP, int iterazioniP) {  
    for (int i = 0; i < iterazioniP; i++) {  
        if (p < maxP) {  
            p += p;  
        } else {  
            return p;  
        }  
    }  
    return p;  
}
```

Sottoprogrammi

All'interno di una classe di Java è possibile specificare uno o più **sottoprogrammi (procedura o funzione)**.

In maniera simile ai sottoprogrammi predefiniti (disponibili nelle librerie di Java), questi sottoprogrammi sono **istruzioni** utilizzabili all'interno del codice che state sviluppando, ma che potete **specificare secondo le vostre necessità**.

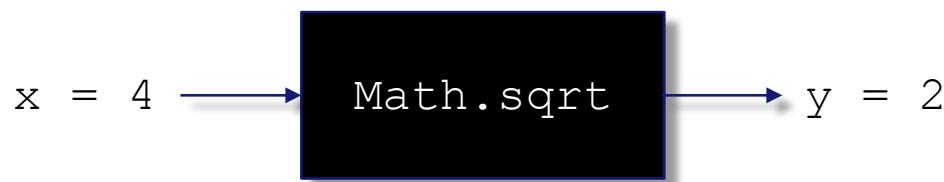


Sottoprogrammi come scatole nere

Un **sottoprogramma** è un insieme di istruzioni che vengono isolate per:

- facilitarne il **riutilizzo**,
- rendere il programma più **leggibile**,
- rendere il codice **facilmente manutenibile**.

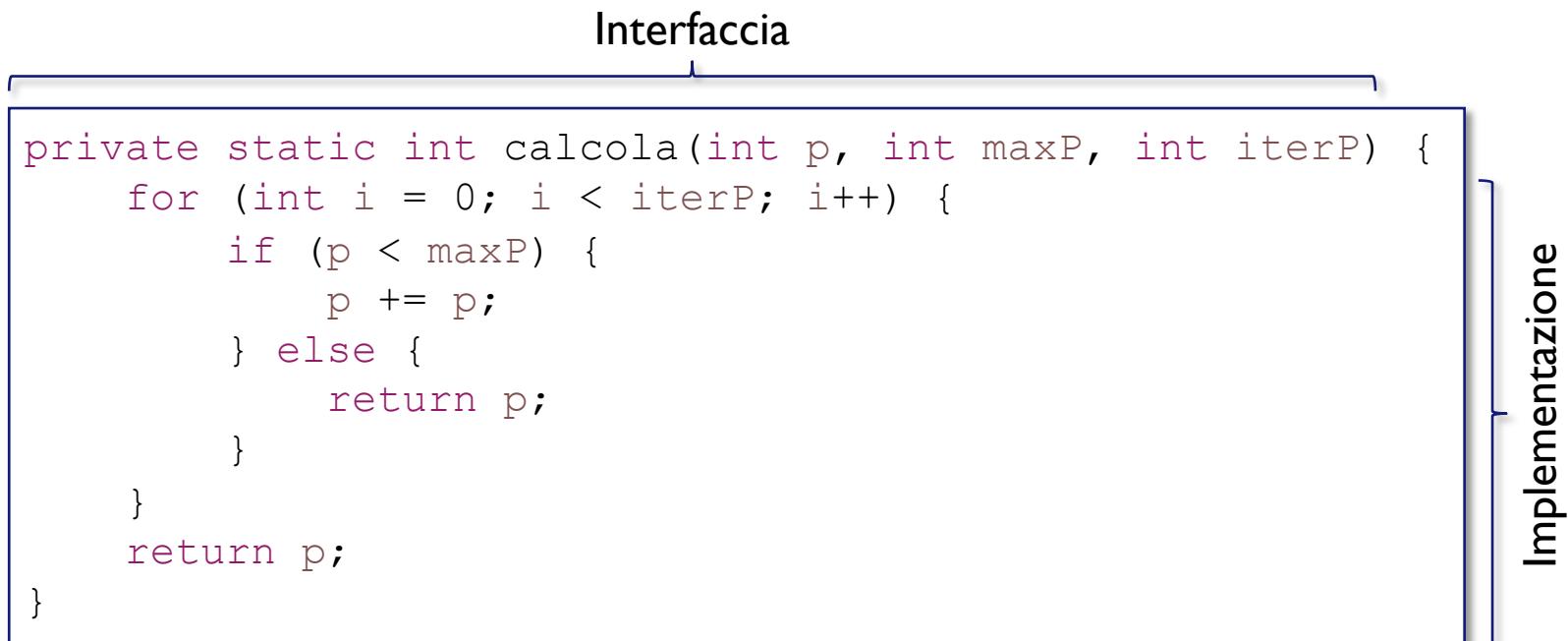
All'interno del codice che lo utilizza, un sottoprogramma è visto **come una scatola nera** perché non siamo interessati ai dettagli del suo funzionamento interno, ma agli effetti che produce.



Sottoprogrammi come scatole nere

Quindi, i sottoprogrammi sono dotati di:

- **Interfaccia di scambio dati:** contiene la descrizione dei dati utilizzati in ingresso e forniti in uscita dal sottoprogramma.
- **Implementazione:** contiene la descrizione dei dati locali e dell'algoritmo che viene eseguito dal sottoprogramma.



Le regole delle scatole nere

Regola 1:

L'**interfaccia** di una scatola nera deve essere **semplice, ben definita e di facile comprensione.**

Regola 2:

Per usare una scatola nera **non deve essere necessario conoscere alcun dettaglio** della sua implementazione.

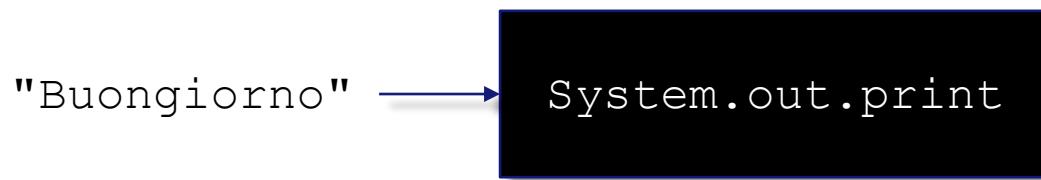
Regola 3:

Lo sviluppatore di una scatola nera **non deve conoscere nulla dei programmi** in cui questa potrà essere utilizzata.

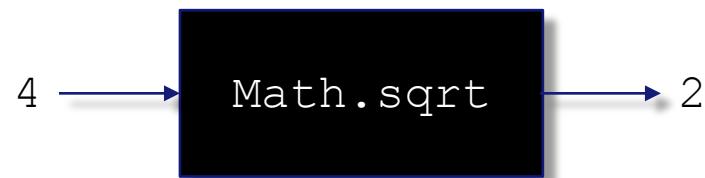
Sottoprogrammi: procedure e funzioni

I sottoprogrammi si dividono in:

- **Procedura**: sottoprogramma che effettua delle operazioni utilizzando gli eventuali argomenti d'ingresso ma **non restituisce un valore** in uscita.



- **Funzione**: sottoprogramma che effettua delle operazioni utilizzando gli eventuali argomenti d'ingresso e **restituisce un valore** in uscita.





Procedure e funzioni

Ogni procedura / funzione è un ***blocco di istruzioni che ha un nome*** e che ***può essere invocato*** all'interno di una qualsiasi espressione nel codice.

Le procedure / funzioni favoriscono la ***strutturazione e il riutilizzo del codice*** sorgente.

Procedure e funzioni: dichiarazione

In Java, per definire un sottoprogramma si utilizza la sintassi seguente:

```
private static tipoRitorno nomeSottoprogramma(listaParametri) {  
    istruzioni;  
}
```

Le **istruzioni** contenute nel blocco di codice sono il **corpo del sottoprogramma**.

Il **tipo di ritorno** viene specificato per le funzioni e rappresenta il **tipo di dato** del valore **restituito**. Per le procedure è sempre **void** (che significa niente).

La **lista dei parametri** rappresenta l'informazione che viene scambiata con il codice chiamante.

Procedure e funzioni: invocazione

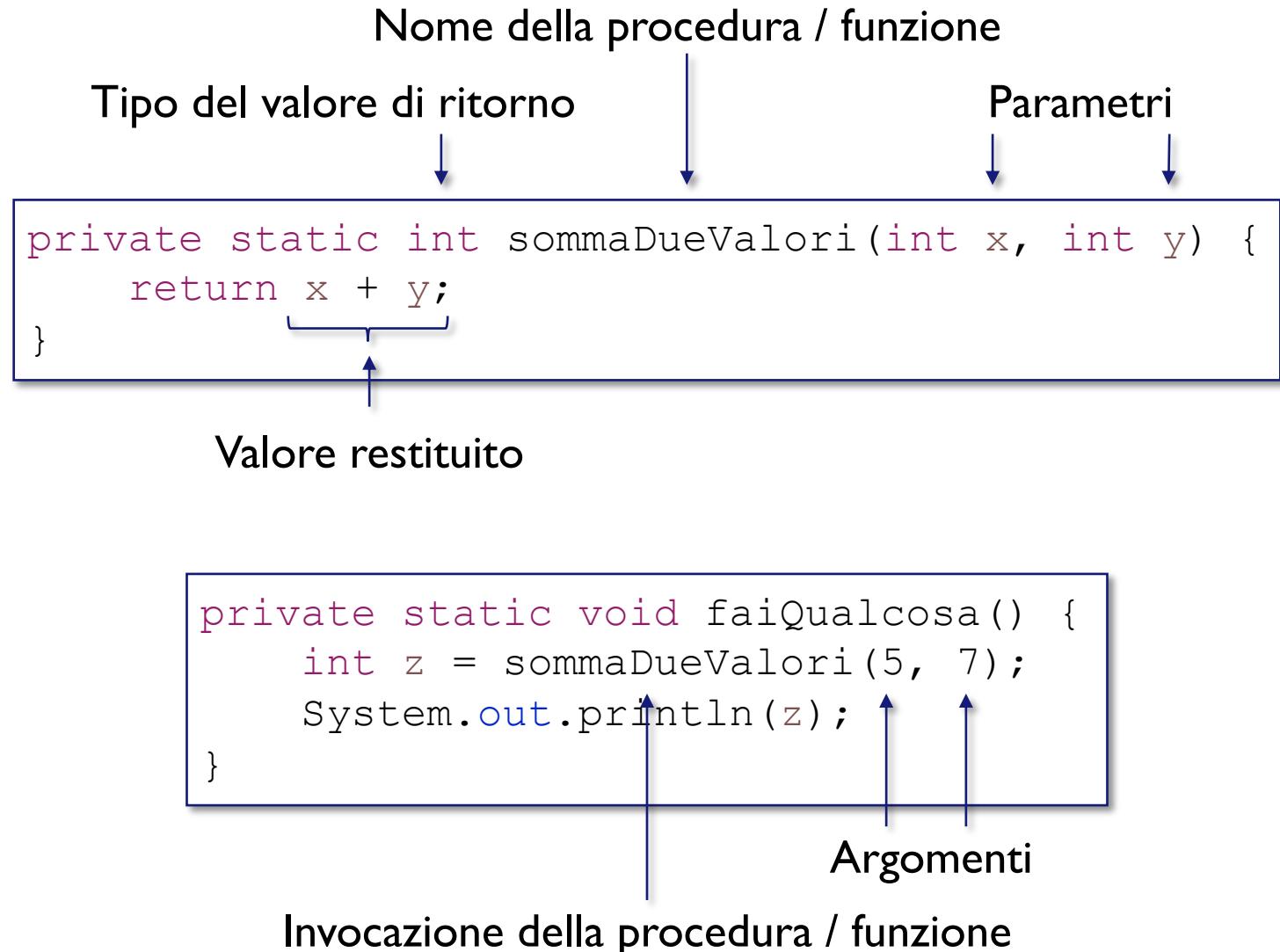
In Java, per **chiamare (invocare)** un sottoprogramma è sufficiente usare il **nome della procedura / funzione** seguito dagli **argomenti** della chiamata (lista di valori, separati da virgola, da assegnare ai parametri):

```
nomeSottoprogramma (listaArgomenti) ;
```

Nel caso di una **funzione** è possibile utilizzare il **valore restituito** dalla chiamata in un'**espressione**. Ad esempio, è possibile assegnare il valore ad una variabile.

```
int max = trovaMassimo(200, 438);
System.out.println("Massimo: " + max);
// Oppure, in modo analogo
System.out.println("Massimo: " + trovaMassimo(200, 438));
```

Procedure e funzioni: esempio



Parametri

I parametri servono a specificare le **informazioni che vengono scambiate** tra il sottoprogramma e il resto del programma.

I parametri definiscono:

- il **nome** (identificatore),
- il **tipo dei dati** che vengono passati al sottoprogramma.

Quando si **invoca** (chiama) un sottoprogramma, **si assegnano i valori ai parametri (che si comportano come variabili)**. Di conseguenza, i valori vengono trasmessi e possono essere utilizzati all'interno del sottoprogramma.

Parametri formali

I **parametri formali** sono quelli **usati nella dichiarazione del sottoprogramma**.

L'elenco dei parametri ne definisce:

- il **numero**,
- l'**identificatore** (il nome),
- il **tipo**.

La dichiarazione di un parametro formale è sostanzialmente analoga alla **dichiarazione di una variabile**.

Il parametro formale è **visibile**, tramite l'identificatore assegnatogli, **esclusivamente all'interno del sottoprogramma**.

Parametri attuali (o argomenti)

I **parametri attuali**, detti anche argomenti, sono quelli usati **al momento della chiamata al sottoprogramma**.

Il parametro attuale **è un valore**. Può quindi essere:

- un **letterale**,
- un **identificatore** di una variabile o di una costante,
- un'**espressione che viene valutata**. Sono comprese espressioni che contengono ulteriori chiamate ad altri sottoprogrammi.

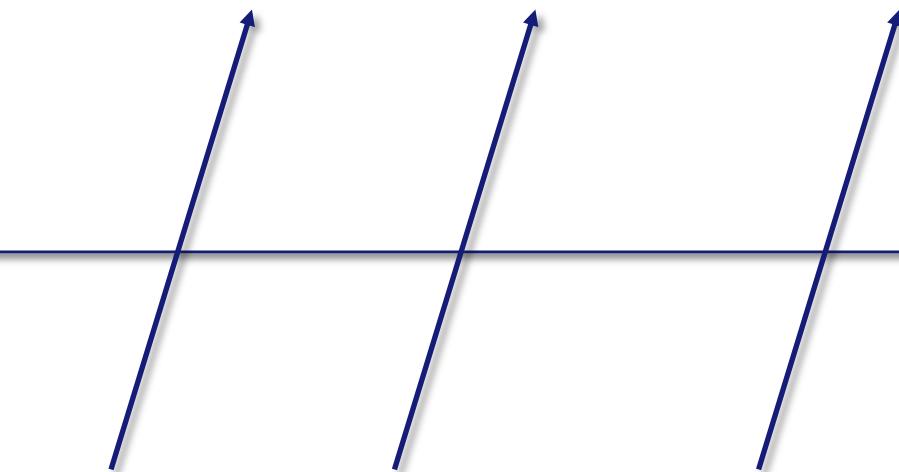
I **parametri attuali vanno specificati nello stesso ordine dei parametri formali**.

Il **tipo** dell'argomento deve essere **identico a quello specificato dal parametro formale**.

Parametri formali e attuali

Parametri formali

```
private static void faiQualcosa(int n, double x, boolean test) {  
    // Istruzioni  
    // da  
    // eseguire  
}
```



Parametri attuali (argomenti)



Dove collocare le procedure e funzioni?

Per ora, tutti i sottoprogrammi sviluppati vanno messi **all'interno della classe** del programma, ma **al di fuori della procedura main**.

Le procedure / funzioni possono essere dichiarate **sia prima che dopo la procedura main**.

```
public class IlMioProgramma {  
    private static int sommaDueValori(int x, int y) {  
        return x + y;  
    }  
  
    public static void main(String[] args) {  
        int i = 3;  
        int j = 4;  
        System.out.println(sommaDueValori(i, j));  
    }  
}
```

Procedure e funzioni

In Java si implementano procedure e funzioni **mediante la medesima sintassi**. L'unica differenza è che le procedure hanno **valore di ritorno void**.

Una funzione **non dovrebbe produrre alcun “side-effect”** sul resto dello stato del programma.

```
private static void mostraValore(int valore) {  
    System.out.println("Valore: " + valore);  
}
```

```
private static double calcolaArea(double raggio) {  
    return raggio * raggio * Math.PI;  
}
```

In Java e negli altri linguaggi orientati agli oggetti, procedure e funzioni vengono anche chiamate **metodi** (capiremo il perché nel corso del semestre).

Completamento di procedure e funzioni

Una procedura o funzione **termina** quando:

- vengono **eseguite tutte le istruzioni** al suo interno,
- viene eseguita l'istruzione **return**,
- viene lanciata un'eccezione (ne discuteremo il prossimo semestre).

L'istruzione return in una procedura

Se il tipo del valore di ritorno è **void** si tratta di una **procedura** (non restituisce alcun valore).

Per una procedura, ***I'istruzione return è opzionale*** e viene utilizzata qualora fosse necessario interromperne l'esecuzione.

```
private static void mostraValoreSePositivo(int valore) {  
    if (valore >= 0)  
        System.out.println(valore);  
}
```

```
private static void mostraValoreSePositivo(int valore) {  
    if (valore < 0)  
        return;  
    System.out.println(valore);  
}
```

L'istruzione `return` vs. l'istruzione `break`

Il comportamento dell'istruzione `return` è simile a quello dell'istruzione `break` ma, nel caso dell'istruzione **`return`**, l'esecuzione della procedura o funzione **termina in quel punto**.

```
private static void trovaCaratteriPrimaDi(String frase, char car) {  
    System.out.println("Frase: " + frase);  
    System.out.print("I caratteri prima di '" + car);  
    System.out.print("'" sono alla posizione: ");  
    for (int i = 0; i < frase.length(); i++) {  
        if (frase.charAt(i) == car) {  
            System.out.println();  
            return;  
        } else  
            System.out.print(i + " ");  
    }  
    System.out.println("Carattere non trovato!");  
}
```

L'istruzione return in una funzione

In una funzione, ***l'istruzione return è obbligatoria*** e viene utilizzata per ***restituire un valore*** di ritorno.

Anche nel caso delle funzioni, l'istruzione `return` interrompe l'esecuzione.

```
private static int leggiValorePositivo(Scanner scanner) {  
    System.out.print("Inserisci un valore positivo: ");  
    int x = scanner.nextInt();  
    if (x < 0) {  
        System.out.println("Valore negativo!");  
        return 0;  
    }  
    return x;  
}
```

Procedure e funzioni: esempio

```
import java.util.Scanner;

public class TriangoloRettangolo {
    private static double pitagora(double c1, double c2) {
        if (c1 <= 0 || c2 <= 0)
            return -1;
        return Math.sqrt(c1 * c1 + c2 * c2);
    }

    private static double perimetro(double c1, double c2, double i) {
        return c1 + c2 + i;
    }

    private static double area(double b, double h) {
        return b * h / 2.;
    }

    ...
}
```

Procedure e funzioni: esempio

```
...  
  
public static void main(String[] args) {  
    Scanner input = new Scanner(System.in);  
    System.out.print("Inserire i due cateti: ");  
    double cat1 = input.nextDouble();  
    double cat2 = input.nextDouble();  
    input.close();  
  
    double ipotenusa = pitagora(cat1, cat2);  
    if (ipotenusa == -1) {  
        System.out.println("Triangolo non valido!");  
        return;  
    }  
  
    System.out.println("Ipotenusa: " + ipotenusa);  
    System.out.println("Perimetro: " +  
                      perimetro(cat1, cat2, ipotenusa));  
    System.out.println("Area: " + area(cat1, cat2));  
}  
}
```

Visibilità delle variabili locali fra i blocchi

Se si specifica una variabile all'interno di un blocco di codice, la variabile sarà **visibile ed accessibile** da tutto il codice del blocco, a partire dalla **dichiarazione** della variabile e fino alla **parentesi di chiusura** di tale blocco. La variabile sarà **visibile anche nei blocchi annidati**.

```
int x = 0;
int y = 0;
while (y < 3) {
    x++;
    if (x == 100) {
        System.out.println("Cento!");
        x = 0;
        y++;
    }
}
```

Visibilità delle variabili locali fra le procedure e funzioni

Se si specifica una variabile all'interno di una procedura o funzione, questa variabile ***NON sarà automaticamente visibile all'interno delle procedure o funzioni chiamate dalla procedura stessa.*** Il valore della variabile deve essere scambiato utilizzando un parametro.

Esempio

```
private static int calcolaX(int x) {  
    int nuovoX = x + 1;  
    if (nuovoX == 100) {  
        System.out.println("Cento!");  
        nuovoX = 0;  
    }  
    return nuovoX;  
}
```

```
// Dichiarazione classe e main  
int x = 0;  
int y = 0;  
while (y < 3) {  
    x = calcolaX(x);  
    if (x % 100 == 0) {  
        y++;  
    }  
}  
// Resto del programma
```

Sovraccaricare procedure e funzioni

In Java è possibile **sovraccaricare (overloading)** le procedure e le funzioni.

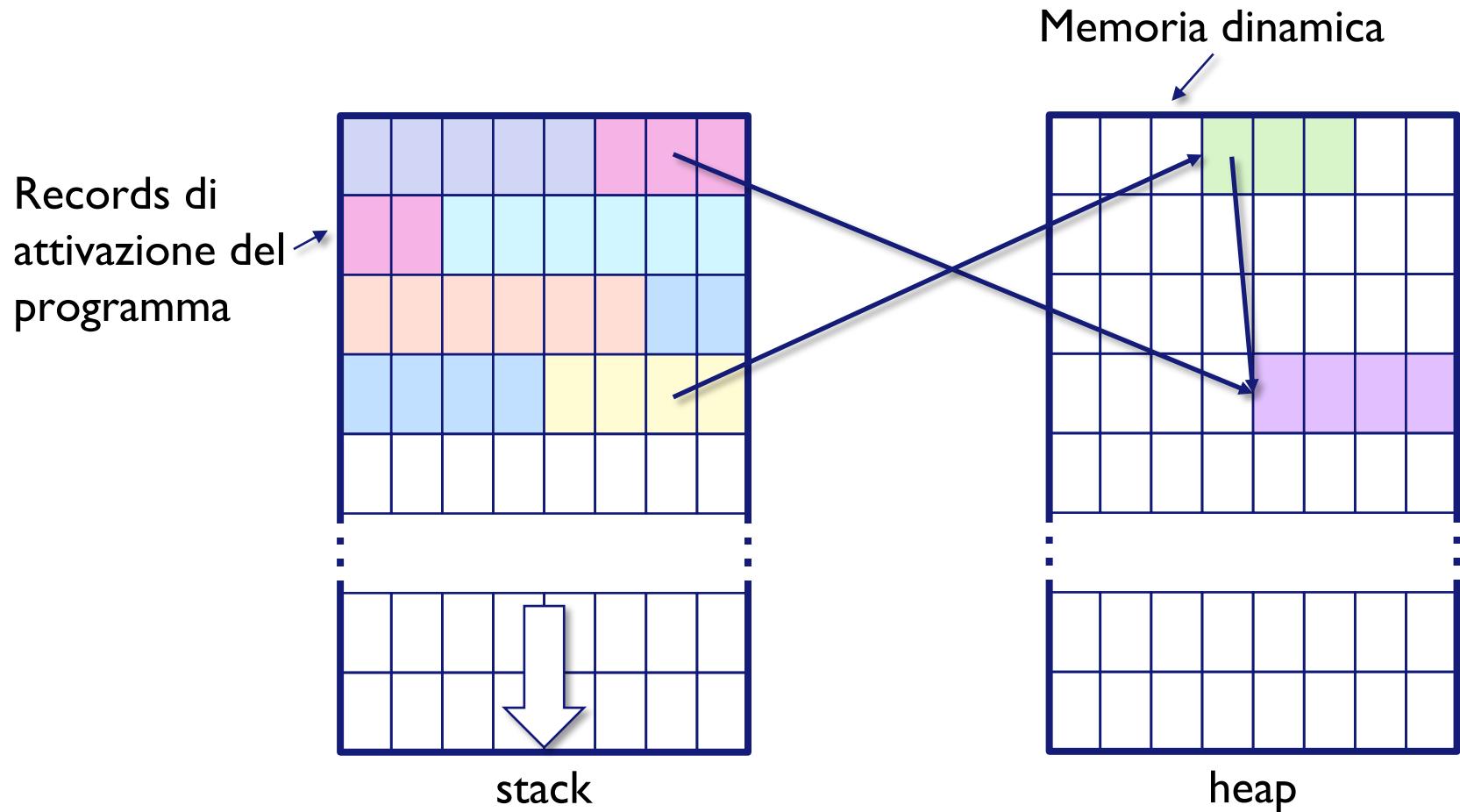
```
private static int area(int x1, int x2, int y1, int y2) {  
    // ...  
}  
  
private static int area(int base, int altezza) {  
    // ...  
}  
  
private static double area(double base, double altezza) {  
    // ...  
}
```

Possono esserci **procedure e funzioni con lo stesso nome** ma devono avere **il numero o il tipo dei parametri diversi**.

Il valore di ritorno può essere diverso ma non lo deve essere necessariamente.

Gestione dinamica della memoria

Gestione dinamica della memoria: operatore *new* e *garbage collector*.



Gestione della memoria nei sottoprogrammi

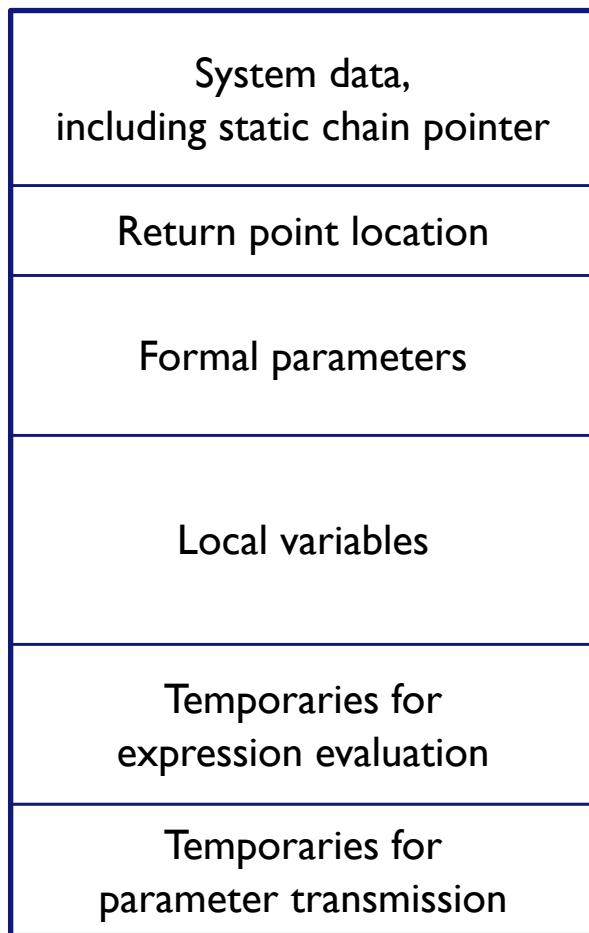
Le zone della memoria stack vengono riservate al momento dell'esecuzione dell'applicazione, sulla base delle **dichiarazioni dei parametri e delle variabili locali** nelle procedure e nelle funzioni.

Quando un sottoprogramma viene attivato (cioè chiamato) viene creato un record di attivazione che contiene:

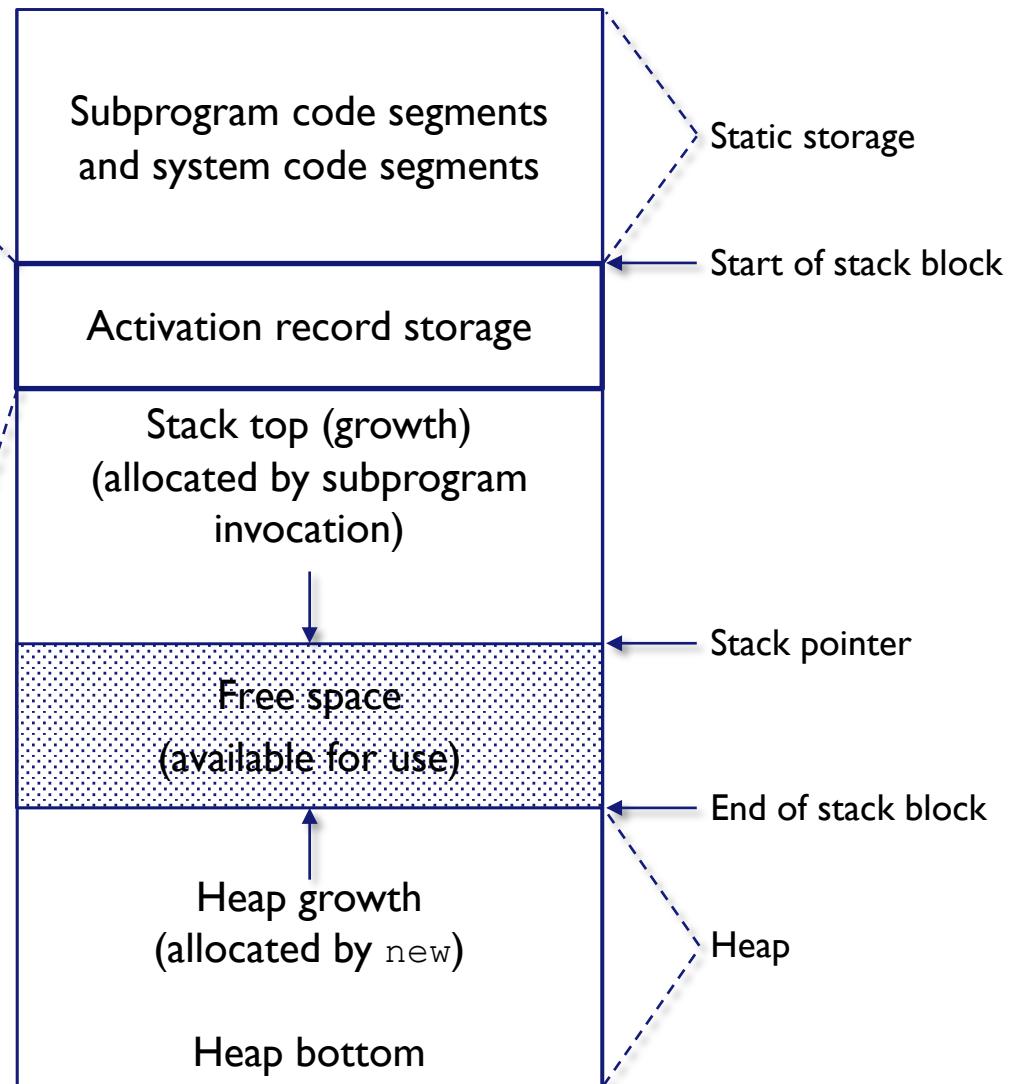
- **il punto di ritorno** nel codice chiamante,
- lo spazio per **i parametri**,
- lo spazio per **le variabili locali**.

Quindi, parametri e variabili locali vengono allocati (e sono accessibili) solo dal momento della chiamata al sottoprogramma.

Struttura del record di attivazione



Activation record for one procedure



Memory organization during execution

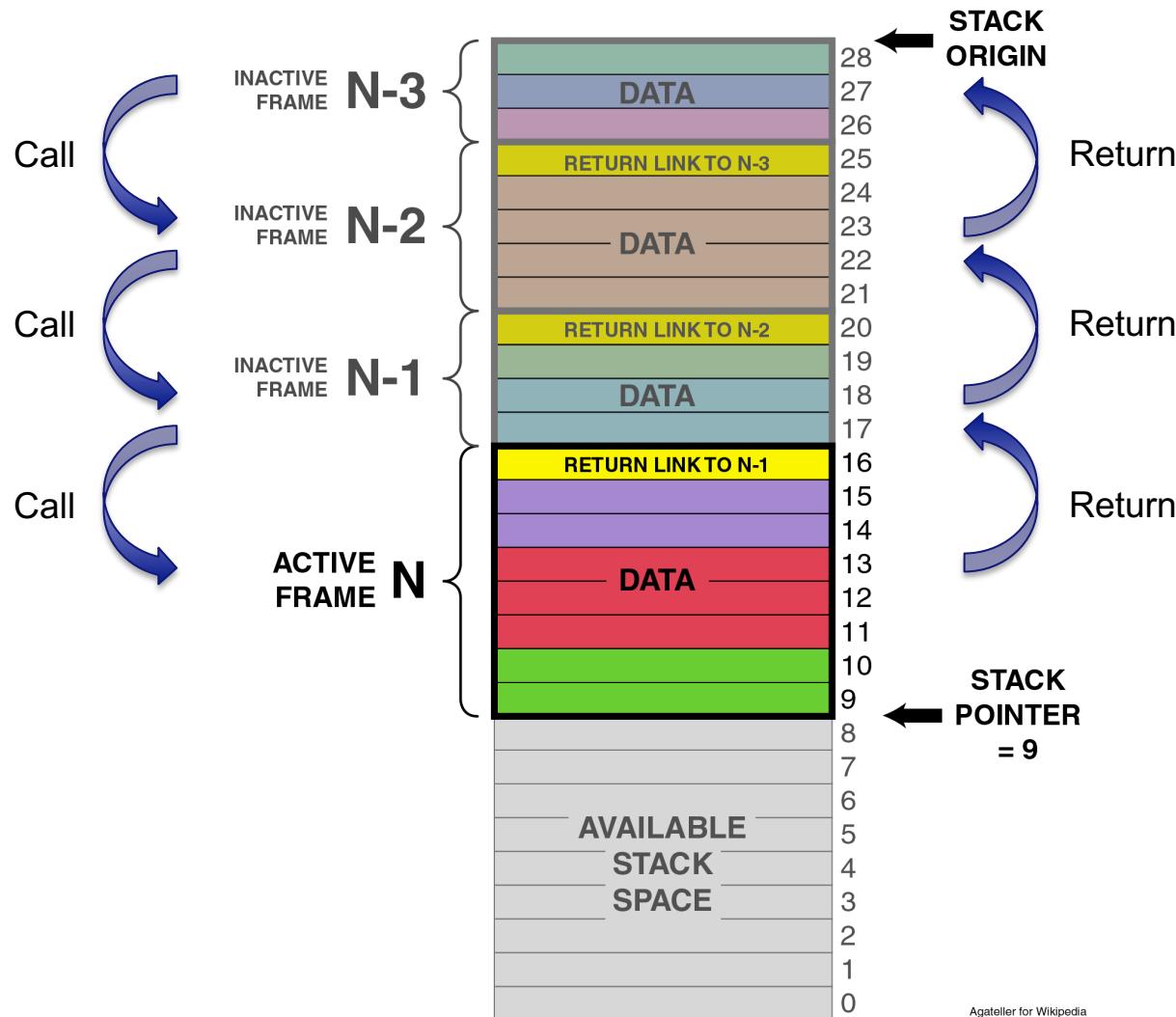
Come funzionano i record di attivazione?

Ogni chiamata a procedura o funzione richiede la creazione di un **nuovo record di attivazione**.

I records di attivazione possono essere **liberati** solo quando la rispettiva **procedura o funzione termina** l'esecuzione.

Ogni procedura o funzione può chiamare altre procedure o funzioni. Di conseguenza: i records di attivazione vengono organizzati in una struttura a pila (stack).

Come funzionano i record di attivazione?



Passaggio con copia del valore o del riferimento

Ogni volta che si invoca un sottoprogramma viene eseguito il **passaggio dei parametri**.

Ci sono **due principali alternative**:

- passaggio con copia del valore,
- passaggio con copia del riferimento.

Passaggio con copia del valore: nel record di attivazione del sottoprogramma viene riservato uno spazio di memoria corrispondente al tipo del parametro o variabile. In seguito, in questo spazio viene **copiato il valore dell'argomento**.

Passaggio con copia del riferimento: nel record di attivazione del sottoprogramma viene riservato lo spazio per un riferimento (indirizzo di memoria). In seguito, in questo spazio viene **copiato l'indirizzo della zona di memoria dove si trova l'argomento**.

Passaggio con copia del valore o del riferimento

In Java:

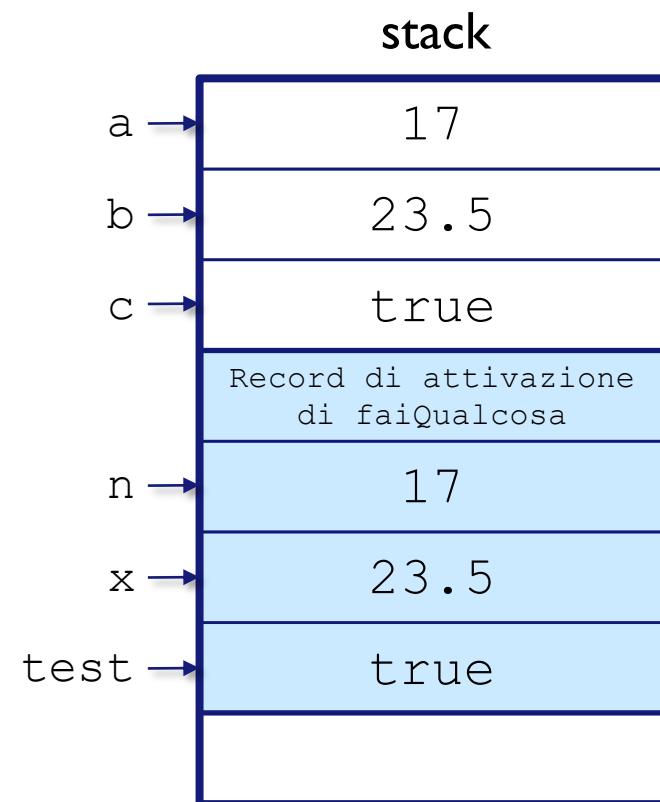
- i tipi di dato **primitivi** vengono sempre passati a procedure e funzioni con **copia del valore**,
- i tipi di dato **riferimento (arrays e oggetti)** vengono sempre passati con **copia del riferimento**.

Attenzione: in Java non è possibile fare un “pass by reference” come è possibile fare in C++. In Java viene sempre eseguita una copia (del valore o del riferimento).

Passaggio di parametri con copia del valore

```
private static void faiQualcosa(int n, double x, boolean test) {  
    // Istruzioni da eseguire  
}
```

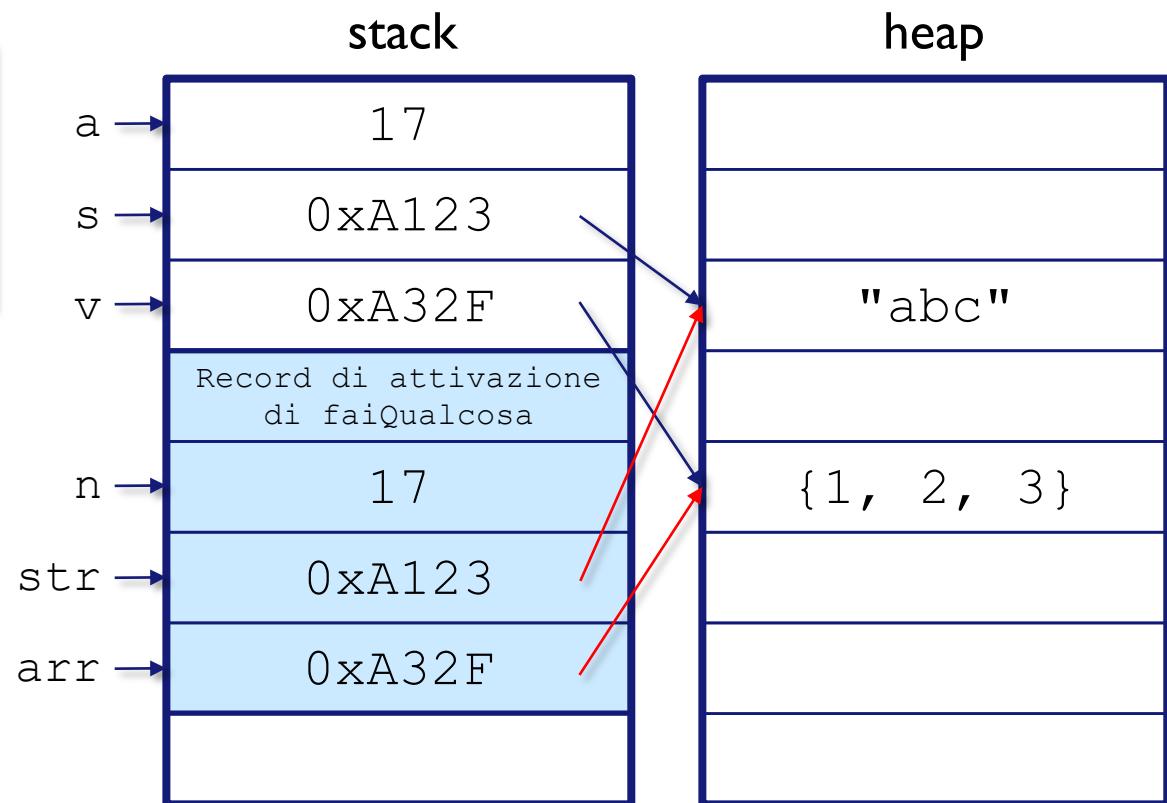
```
int a = 17;  
double b = 23.5;  
boolean c = true;  
faiQualcosa(a, b, c);
```



Passaggio di parametri con copia del riferimento

```
private static void faiQualcosa(int n, String str, int[] arr) {  
    // Istruzioni da eseguire  
}
```

```
int a = 17;  
String s = "abc";  
int[] v = { 1, 2, 3 };  
faiQualcosa(a, s, v);
```



Copia del valore o copia del riferimento: conseguenze

Parametri passati con **copia del valore**: se all'interno della procedura o funzione si modifica il valore del parametro, dopo la chiamata della procedura o funzione, **l'argomento conterrà il valore che aveva prima della chiamata.**

Parametri passati con **copia del riferimento**: se all'interno della procedura o funzione si modifica il parametro, dopo la chiamata della procedura o funzione, **l'argomento conterrà le modifiche fatte all'interno della procedura o funzione.**

Copia del valore o copia del riferimento: conseguenze

```
import java.util.Arrays;

public class PassaggioParametri {
    private static int eseguiLavoro(int valore, char[] arrC) {
        valore = 666;
        arrC[0] = 'Z';
        return 77;
    }

    public static void main(String[] args) {
        int valInt = 15;
        int valRet = 0;
        char[] arrayChar = { 'A', 'B', 'C' };

        System.out.println("Prima dell'invocazione:");
        System.out.println("\tValInt: " + valInt + ", ValRet: " + valRet);
        System.out.println("\tArray: " + Arrays.toString(arrayChar));

        valRet = eseguiLavoro(valInt, arrayChar);

        System.out.println("\nDopo l'invocazione:");
        System.out.println("\tValInt: " + valInt + ", ValRet: " + valRet);
        System.out.println("\tArray: " + Arrays.toString(arrayChar));
    }
}
```



Passaggio parametri con copia del valore: esempio

```
public class EsempioCopiaValore {  
    private static void routine(int p) {  
        // Modifica il parametro  
        p = 10;  
        System.out.println("In routine, p = " + p);  
    }  
  
    public static void main(String[] args) {  
        int x = 3;  
  
        // Invoca routine() con x come argomento  
        routine(x);  
  
        // Mostra x per vedere se il valore è cambiato  
        System.out.println("Dopo l'invocazione, x = " + x);  
    }  
}
```

Passaggio parametri con copia del riferimento: esempio

```
public class EsempioCopiaRiferimento {  
    private static void spostaCerchio(int[] cerchio, int deltaX, int deltaY) {  
        // Sposta il centro del cerchio  
        cerchio[0] += deltaX;  
        cerchio[1] += deltaY;  
        System.out.println("x: " + cerchio[0] + " y: " + cerchio[1]);  
  
        // Assegna una nuova referenza a cerchio  
        // (nessun effetto fuori dalla procedura)  
        cerchio = new int[2];  
        System.out.println("x: " + cerchio[0] + " y: " + cerchio[1]);  
    }  
  
    public static void main(String[] args) {  
        int[] ilMioCerchio = { 25, 50 };  
        System.out.println("x: " + ilMioCerchio[0] + " y: " + ilMioCerchio[1]);  
        spostaCerchio(ilMioCerchio, 23, 56);  
        System.out.println("x: " + ilMioCerchio[0] + " y: " + ilMioCerchio[1]);  
    }  
}
```

Passaggio parametri con copia del riferimento: esempio 2

```
public class EsempioCopiaRiferimentoStringa {  
    private static void modificaStringa(String frase, String aggiunta) {  
        System.out.println("Frase: " + frase);  
        frase += aggiunta;  
        System.out.println("Frase dopo la modifica: " + frase);  
    }  
  
    public static void main(String[] args) {  
        String frase = "Benvenuti";  
        System.out.println("Frase: " + frase);  
        modificaStringa(frase, " a tutti");  
        System.out.println("Frase: " + frase);  
    }  
}
```

I parametri della procedura main

I parametri della procedura `main` permettono di specificare degli **argomenti che vengono passati al momento del lancio del programma**. Questo permette di specificare, all'avvio dell'applicazione, delle informazioni di configurazione.

Questi argomenti sono **sempre di tipo String** e sono passati al `main` sottoforma di array.

```
public class TestArgomentiMain {  
    public static void main(String[] args) {  
        System.out.print("Argomenti: ");  
        for (String arg : args)  
            System.out.print(arg + " ");  
    }  
}
```

```
$ java TestArgomentiMain 100 1.25 Prova  
Argomenti: 100 1.25 Prova
```

Variabili locali vs. variabili globali

Ogni variabile dichiarata all'interno di una **procedura o funzione** è considerata **variabile locale** ed esiste dal momento in cui viene dichiarata fino alla fine del blocco di codice in cui è contenuta. **Lo stesso discorso vale per i parametri** della procedura o funzione.

Se si vogliono immagazzinare valori in **variabili che esistono per tutta la durata del programma** si possono utilizzare le **variabili globali**.

```
private static tipoDiDato nomeDellaVariabile;
```

In Java, le **variabili globali** vanno **dichiarate al di fuori** (ad esempio prima) **delle procedure e funzioni** del programma. Sono **accessibili da ovunque** all'interno del programma.

Variabili globali: esempi

```
private static String mioNome = "Nome non presente";  
private static int altezza;  
private static boolean isGiallo = false;  
private static int[] valori = new int[2];
```

Inizializzazione

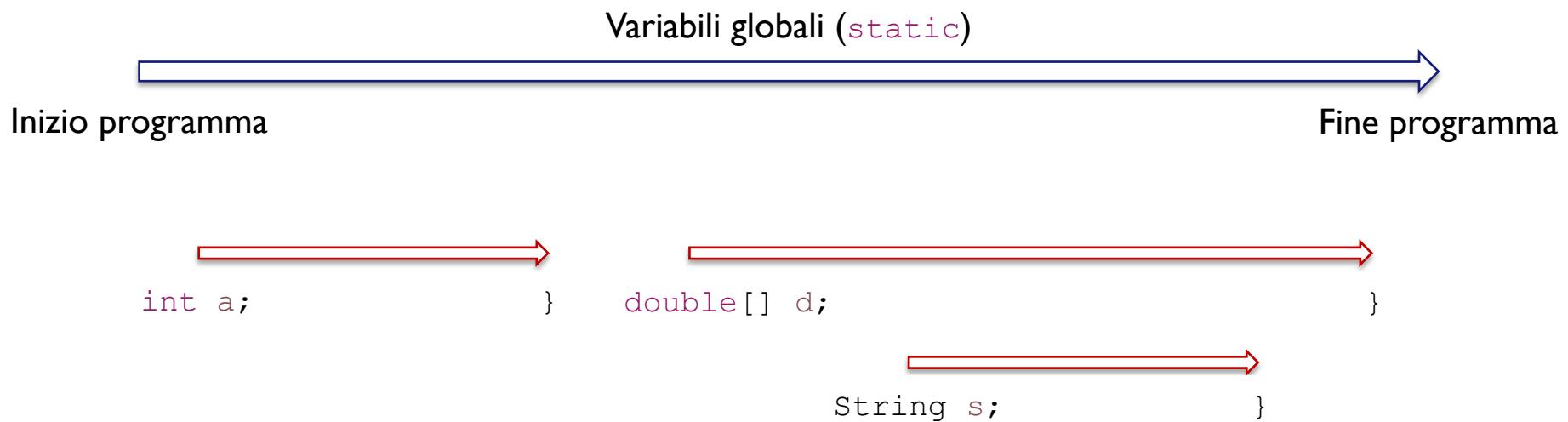
Variabili globali: esempio

```
public class UsoVariabiliGlobali {  
    private static int x = 0;  
  
    private static void sottrai10() {  
        System.out.println("x = " + x);  
        x -= 10;  
        System.out.println("x = " + x);  
    }  
  
    public static void main(String[] args) {  
        System.out.println("x = " + x);  
        x++;  
        System.out.println("x = " + x);  
        sottrai10();  
        System.out.println("x = " + x);  
        x++;  
        System.out.println("x = " + x);  
    }  
}
```

Il ciclo di vita delle variabili

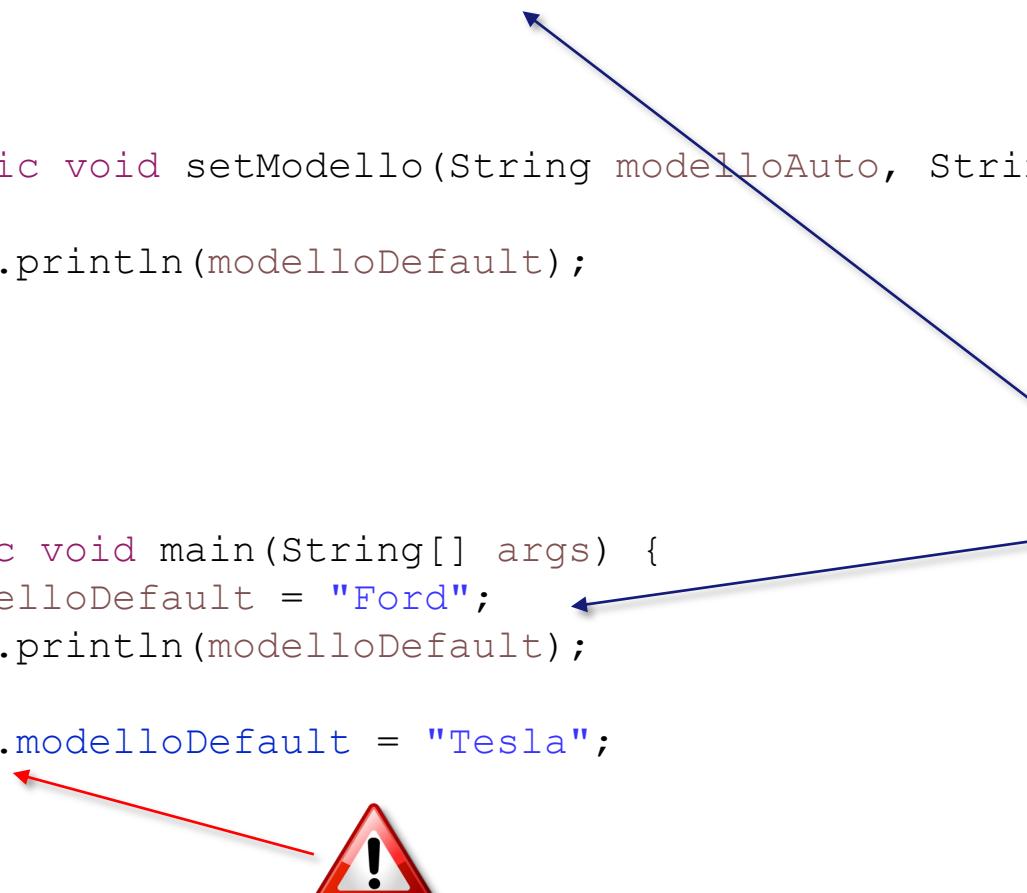
Le **variabili locali** e i **parametri** esistono dall'istante in cui vengono dichiarate fino alla fine del blocco in cui sono contenute.

Le **variabili globali** esistono durante tutta l'esecuzione del programma.



Visibilità

```
public class Automobile {  
  
    private static String modelloDefault = "Aston Martin";  
  
    private static void setModello(String modelloAuto, String modelloDefault) {  
        ...  
        System.out.println(modelloDefault);  
        ...  
    }  
  
    public static void main(String[] args) {  
        String modelloDefault = "Ford";  
        System.out.println(modelloDefault);  
        ...  
        Automobile.modelloDefault = "Tesla";  
        ...  
    }  
}
```



Le variabili locali e i parametri possono **rendere invisibili** le variabili globali.

Riepilogo

- Riutilizzo del codice
- Procedure e funzioni
- Struttura di un sottoprogramma
- Dichiarazione e invocazione di un sottoprogramma
- Parametri formali e attuali (o argomenti)
- L'istruzione `return`
- Visibilità delle variabili tra i sottoprogrammi
- Sovraccaricare delle procedure o funzioni
- Gestione della memoria nei sottoprogrammi
- Record di attivazione
- Passaggio con copia del valore o con copia del riferimento
- Parametri della procedura `main`
- Variabili globali e visibilità
- Ciclo di vita delle variabili