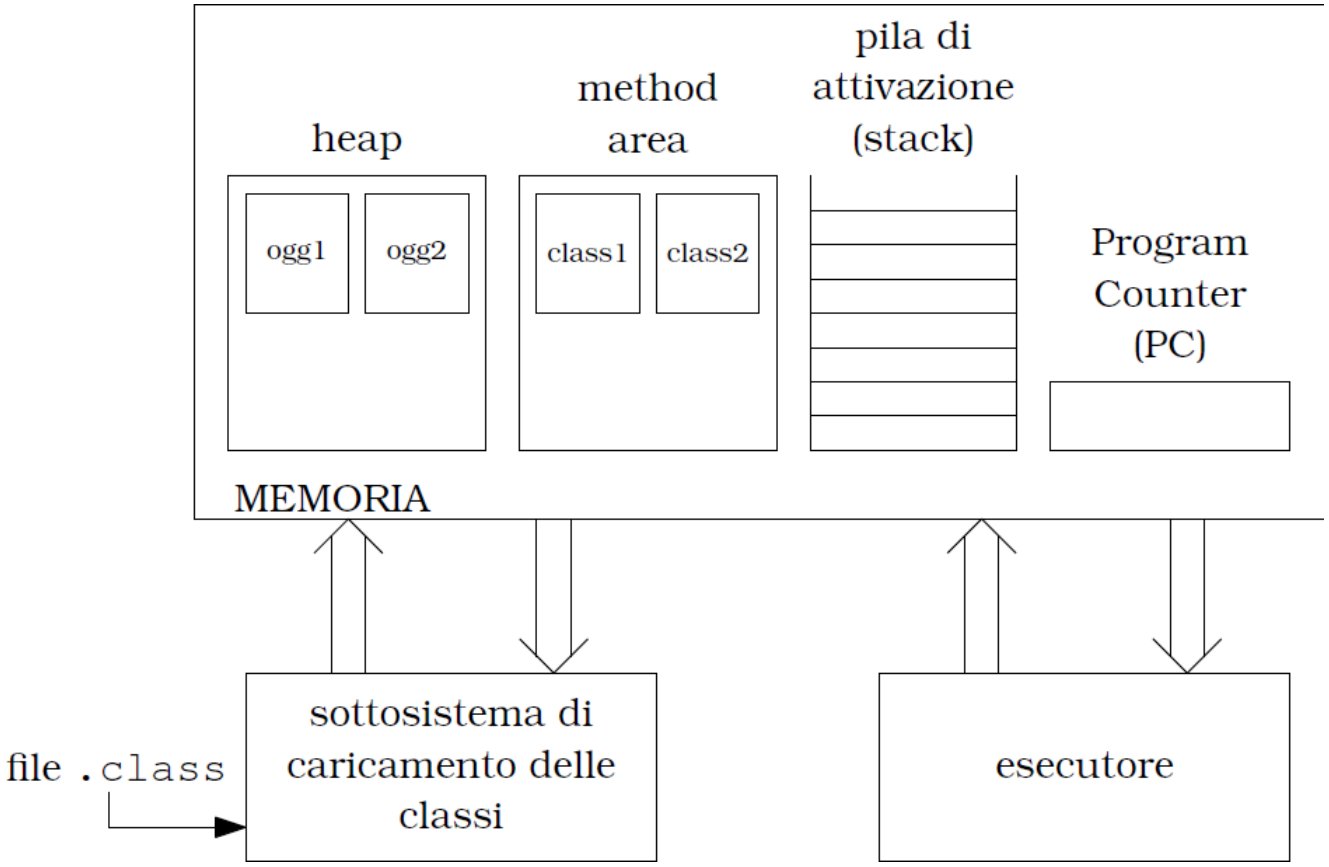
Gestione della memoria in Java

Emilio Di Giacomo e Walter Didimo

Gestione della memoria

- In questa lezione descriveremo un modello run-time (cioè a tempo di esecuzione) di gestione della memoria in Java
- vedremo come viene gestita, durante l'esecuzione di un programma, la memoria associata alle classi, agli oggetti e la memoria necessaria per l'esecuzione di costruttori e metodi
- Il modello che descriveremo è un modello semplificato ma sufficientemente descrittivo della gestione della memoria nella JVM

Architettura della JVM



Architettura della JVM

- Sottosistema per il caricamento delle classi (classloader subsytstem): si occupa di caricare in memoria il codice delle classi dai file *.class*
- Esecutore: esegue le istruzioni che compongono il programma
- Memoria: memorizza tutte le informazioni necessarie all'esecuzione del programma:
  - il codice delle classi, gli oggetti che vengono istanziati, le variabili che vengono definite, i risultati intermedi dei calcoli, ecc.

Architettura della JVM

- Quando serve memoria per uno scopo specifico viene riservata una porzione di memoria di dimensione opportuna per tale scopo
- Si dice che tale memoria è stata allocata.
- Si parla di deallocazione della memoria quando una porzione di memoria precedentemente allocata ritorna ad essere disponibile

Memoria nella JVM

- La memoria della JVM è suddivisa logicamente nelle seguenti aree:
  - Program Counter. È un singolo registro a 32 bit che memorizza l'indirizzo della prossima istruzione da eseguire
  - Heap. È la porzione della memoria in cui vengono memorizzati gli oggetti e gli array istanziati durante l'esecuzione del programma

Memoria nella JVM

- La memoria della JVM è suddivisa logicamente nelle seguenti aree:
  - Method area. È la porzione di memoria in cui vengono memorizzate le classi caricate durante l'esecuzione del programma
    - è concettualmente parte dello heap, ma gestita in maniera diversa
  - Pila di attivazione (in inglese stack). Viene utilizzata per allocare la memoria necessaria all'esecuzione di metodi e costruttori

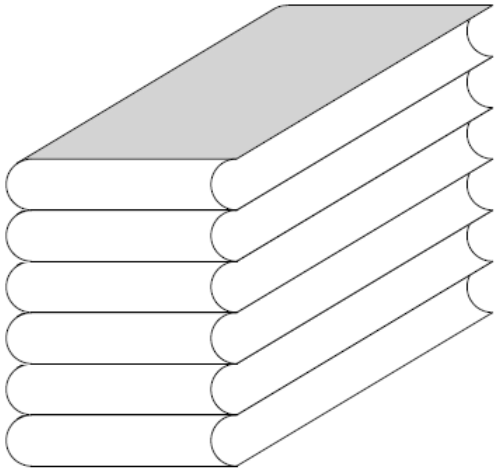
Pila e record di attivazione

- L’esecuzione dei metodi viene gestita mediante record di attivazione
- Un record di attivazione memorizza alcune informazioni relative ad una singola esecuzione di un metodo
- Ogni volta che un metodo viene eseguito si alloca per esso un record di attivazione
- Tale record viene poi deallocato quando l’esecuzione del metodo termina
- Ogni esecuzione di uno stesso metodo dà luogo alla allocazione di un diverso record di attivazione

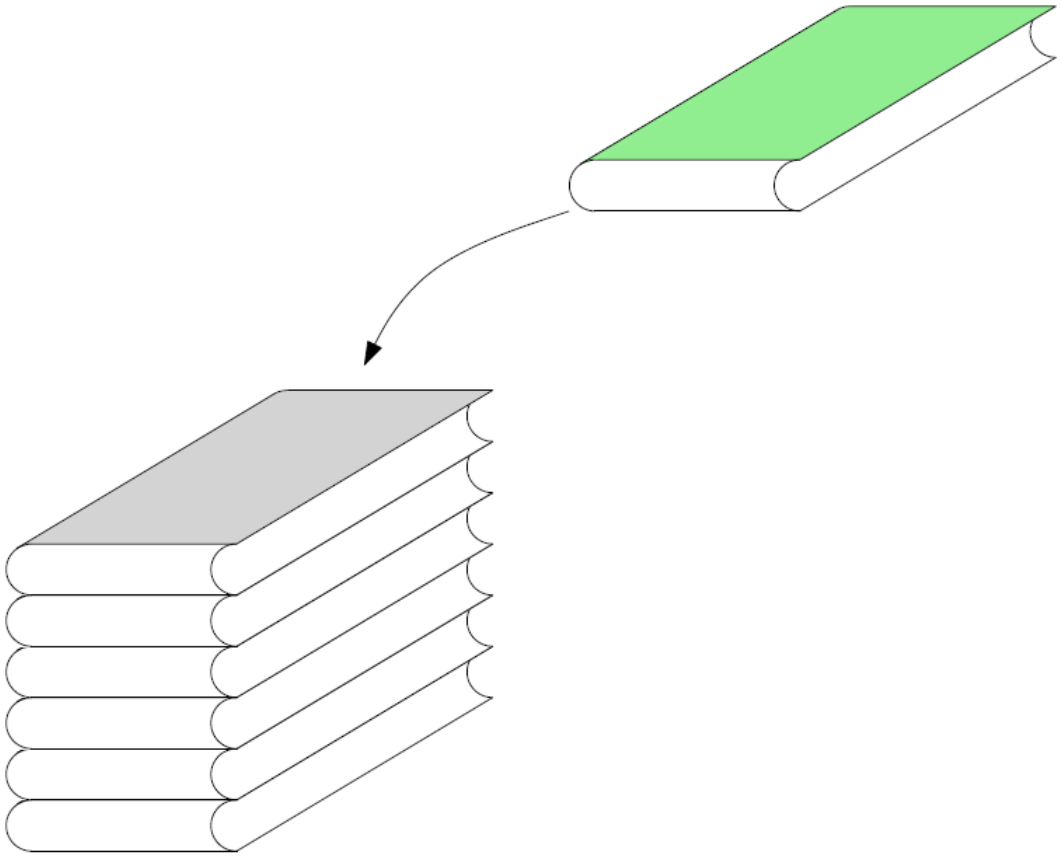
Pila e record di attivazione

- I record di attivazione vengono allocati all'interno di un'area di memoria gestita come una pila
- Per questo motivo tale area viene chiamata pila di attivazione
- Una pila è una struttura dati in cui gli elementi memorizzati vengono gestiti secondo la politica Last-In-First-Out (LIFO)
  - l'ultimo elemento inserito nella pila è il primo a uscire.
- In una pila quindi l'inserimento e la rimozione di elementi avviene sempre in cima alla pila

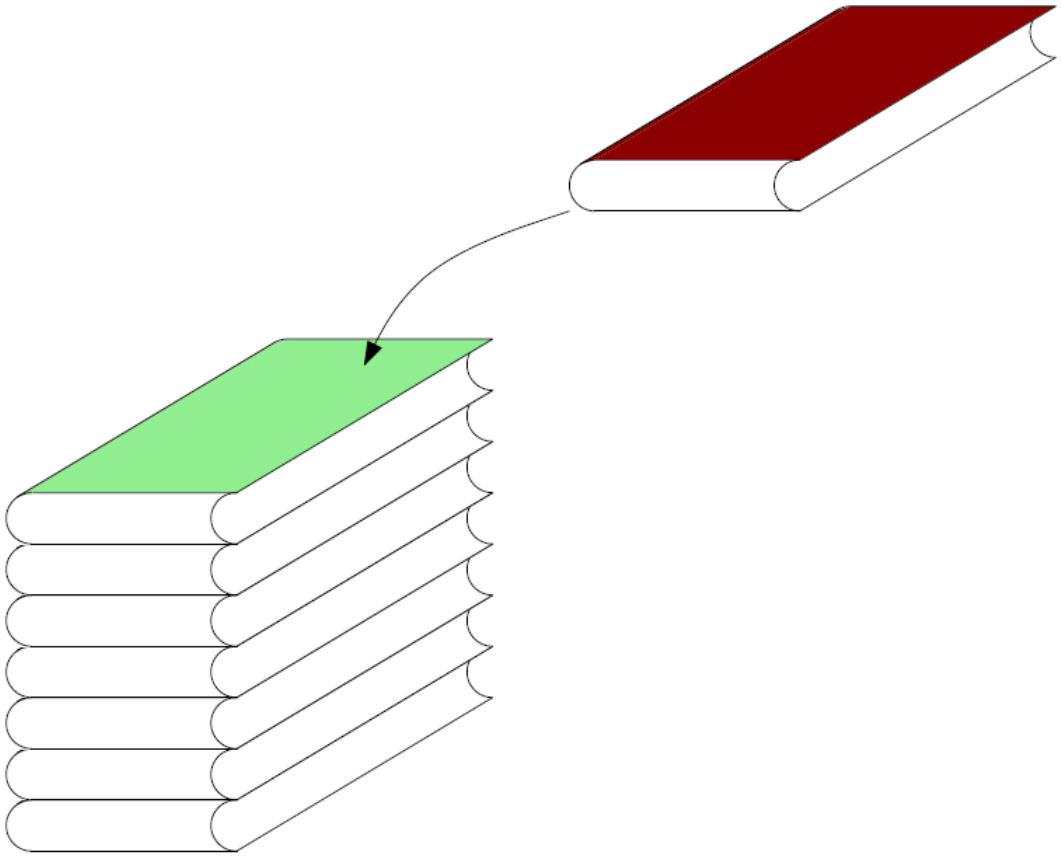
La struttura dati pila



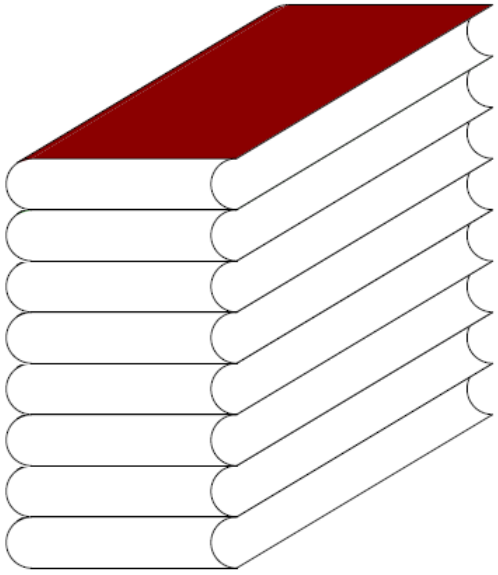
La struttura dati pila



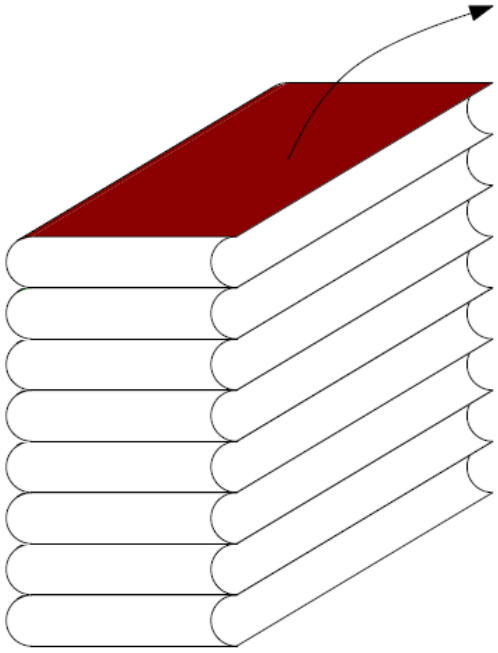
La struttura dati pila



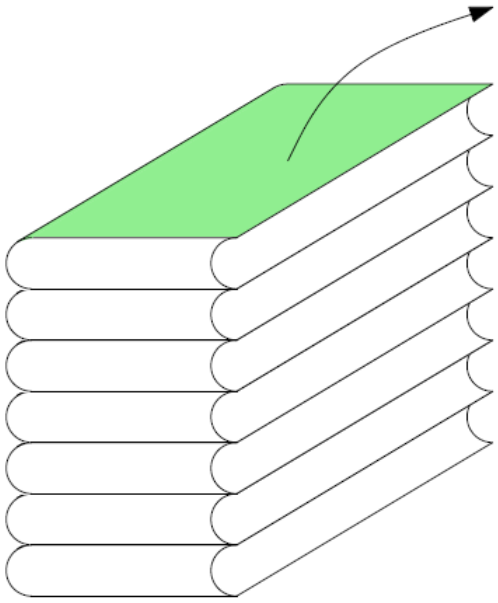
La struttura dati pila



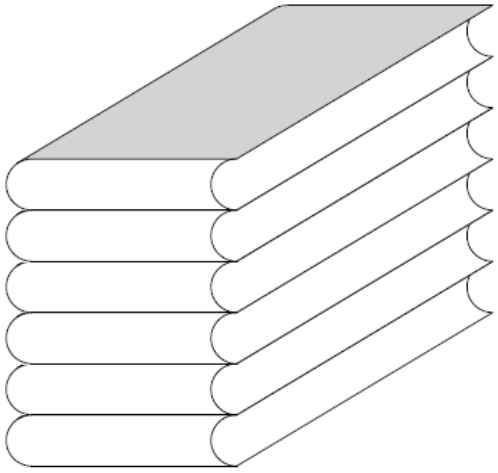
La struttura dati pila



La struttura dati pila



La struttura dati pila



Record e pila di attivazione

- Quando un metodo m_1 viene eseguito si alloca un record di attivazione r_1 che viene inserito in cima alla pila di attivazione
- Se durante l'esecuzione di m_1 si invoca un secondo metodo m_2, viene allocato un record di attivazione r_2 per m_2 e messo in cima alla pila di attivazione
- L'esecuzione di m_1 viene sospesa e il controllo passa a m_2
- In ogni momento soltanto il metodo il cui record di attivazione è in cima alla pila è in esecuzione
- Tutti gli altri metodi che si trovano nella pila sono in attesa

Record e pila di attivazione

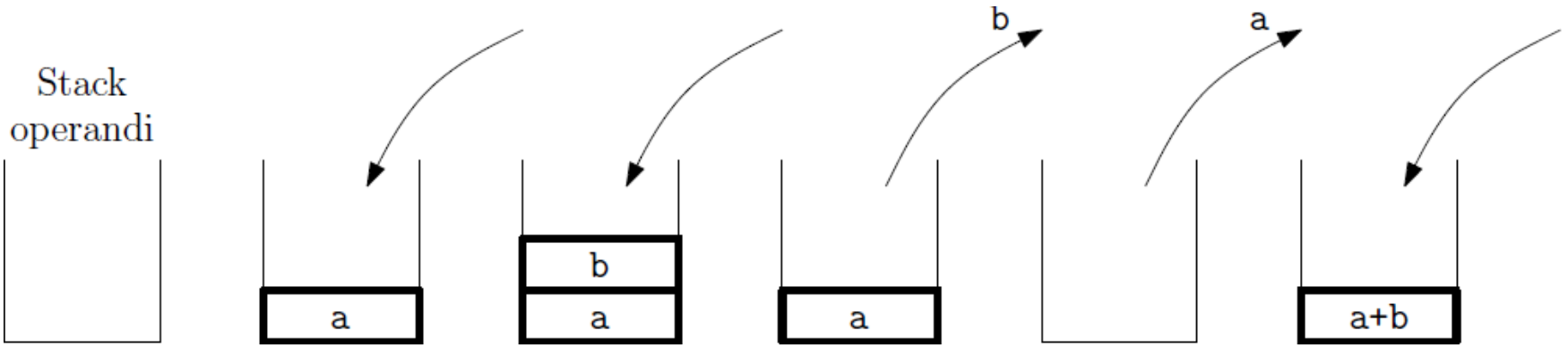
- Quando m_2 termina, il record r_2 viene deallocato
- Se m_2 restituisce un valore questo viene passato al record sottostante, cioè a r_1, che torna ad essere in cima alla pila
- L'esecuzione di m_1 riprende dal punto in cui si era interrotta
- Quando anche m_1 termina, anche r_1 viene deallocato
- Il controllo passa al metodo il cui record si trova sotto a r_1 nella pila
- Se non c'è nessun record al di sotto di r_1 l'esecuzione del programma termina

Record di attivazione

- Vediamo quali sono le informazioni memorizzate nei record di attivazione
- Un primo elemento è l'array delle variabili locali
  - contiene tutte le variabili locali del metodo inclusi
    - i parametri formali
    - un riferimento all'oggetto ricevente (cioè il riferimento *this*) se il metodo è di istanza
- Le variabili vengono denotate tramite un indice
- Il riferimento *this* (se c'è) si trova in posizione 0
- Dalla posizione 1 (o dalla posizione 0 se non c'è *this*) vengono memorizzati i parametri
- A seguire le variabili locali del metodo

Record di attivazione

- Il secondo elemento del record di attivazione è lo stack degli operandi
- In esso vengono memorizzati gli operandi e i risultati dei vari calcoli
- È gestito come una pila (da cui il nome)
- Es: *a+b*



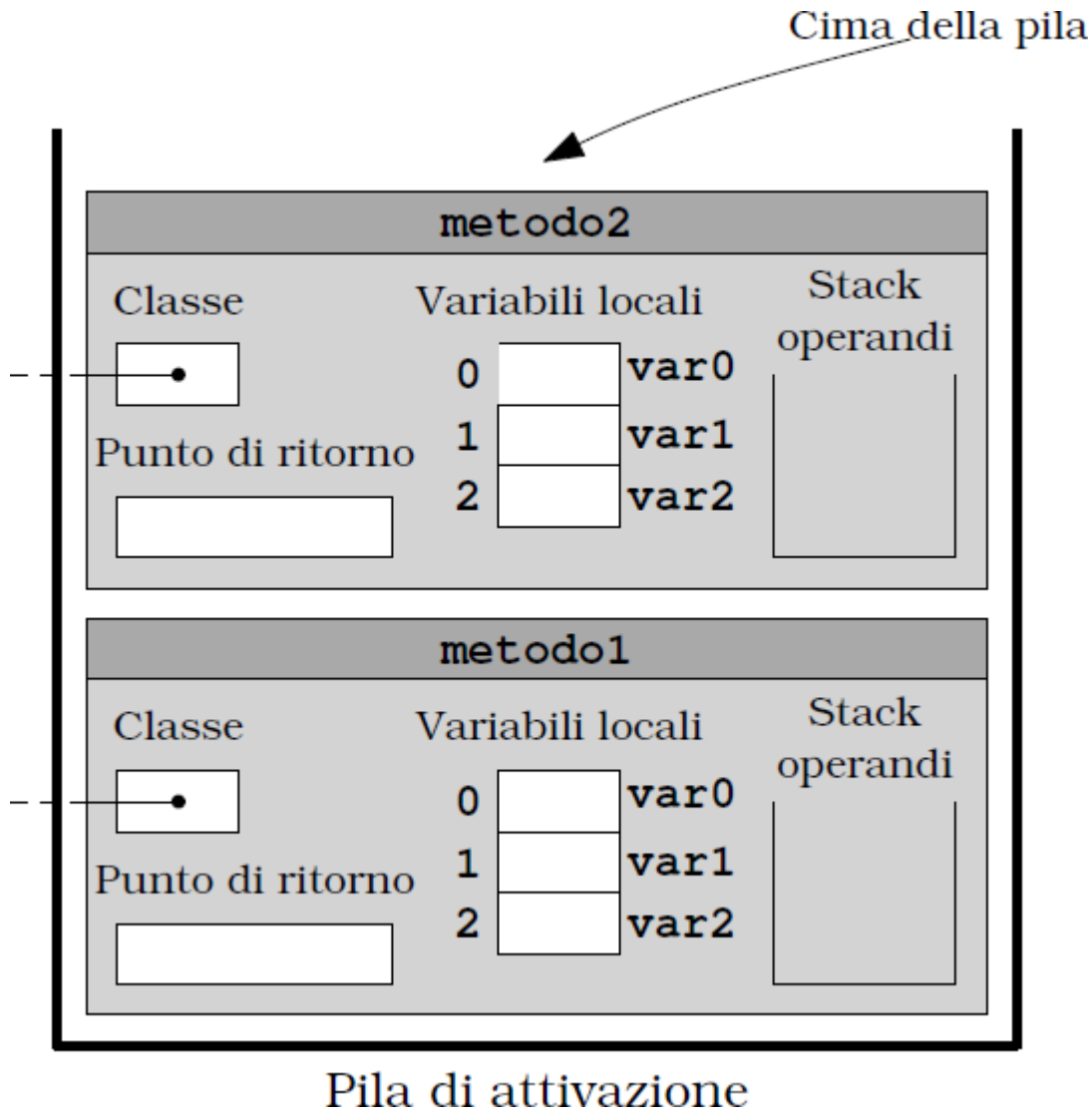
Record di attivazione

- Il terzo elemento di un record di attivazione è il campo classe
- Memorizza un riferimento alla classe di cui il metodo fa parte
  - più precisamente memorizza un riferimento alla run-time constant pool
  - una tabella contenente informazioni associate alla classe tra cui
    - nomi delle variabili
    - signature dei metodi utilizzati nella classe

Record di attivazione

- Oltre a quelle precedenti, un record di attivazione contiene altre informazioni necessarie alla gestione del record stesso. Ad es. informazioni per
  - gestire la terminazione corretta del metodo
  - generare eccezioni nel caso di terminazione errata.
- Non descriviamo tali informazioni
- Ci limitiamo a menzionare il punto di ritorno, cioè l'indirizzo dell'istruzione da cui deve continuare l'esecuzione una volta che il record viene deallocato
- Il punto di ritorno è sempre relativo al metodo il cui record si trova al di sotto del record corrente

Pila e record di attivazione



Esempio di esecuzione

- Illustriamo adesso il funzionamento della pila di attivazione mediante il seguente esempio

```
public class Esempio1{
   public static int metodo1(int a, int b){
        int m;
        m = a+b; // metodo1, 1
        return m; // metodo1, 2
   }
        ...
}
```

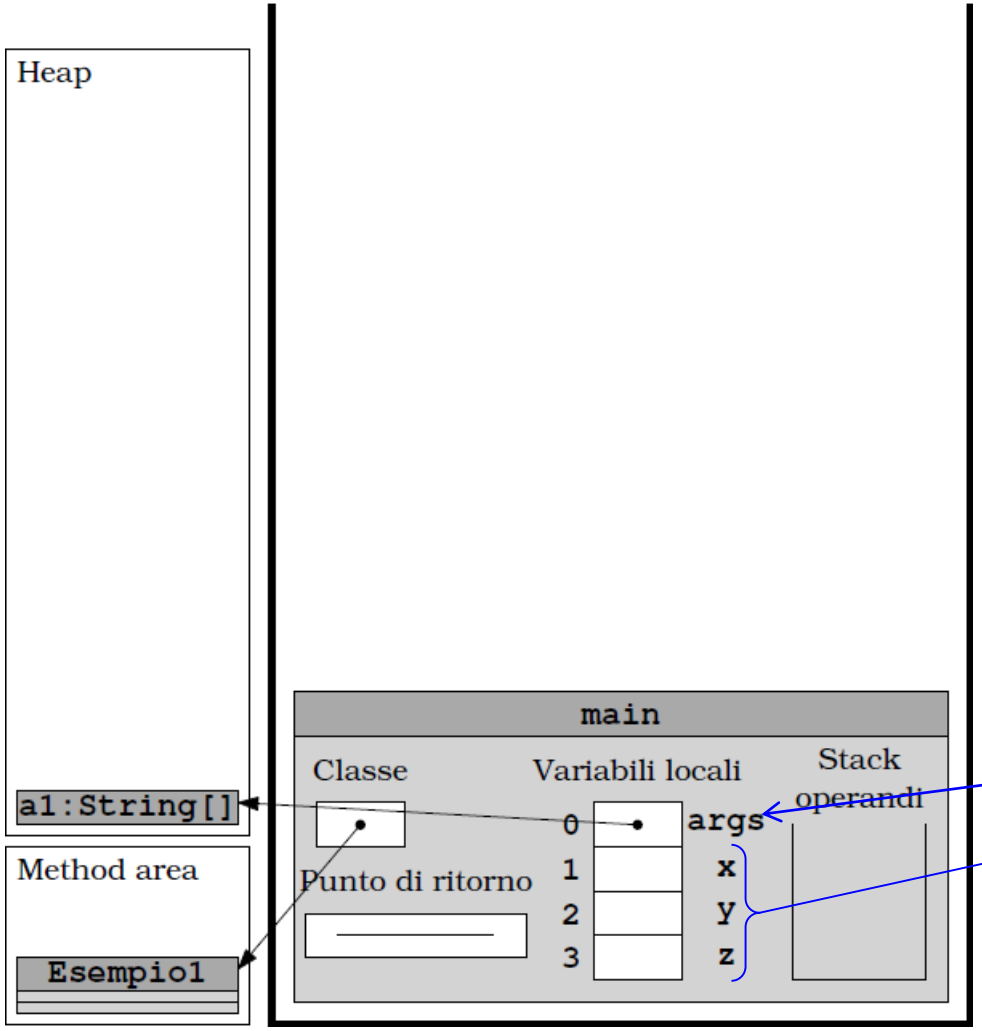
Esempio di esecuzione

```
public class Esempio1{
   ...

   public static int metodo2(int c){
       int i;
       i = metodo1(c, c); // metodo2, 1
       return i; // metodo2, 2
   }

   public static void main(String[] args){
       int x, y, z;
       x = 1; // main, 1
       y = metodo2(x); // main, 2
       z = metodo1(x, y); // main, 3
   }
}
```

Esempio

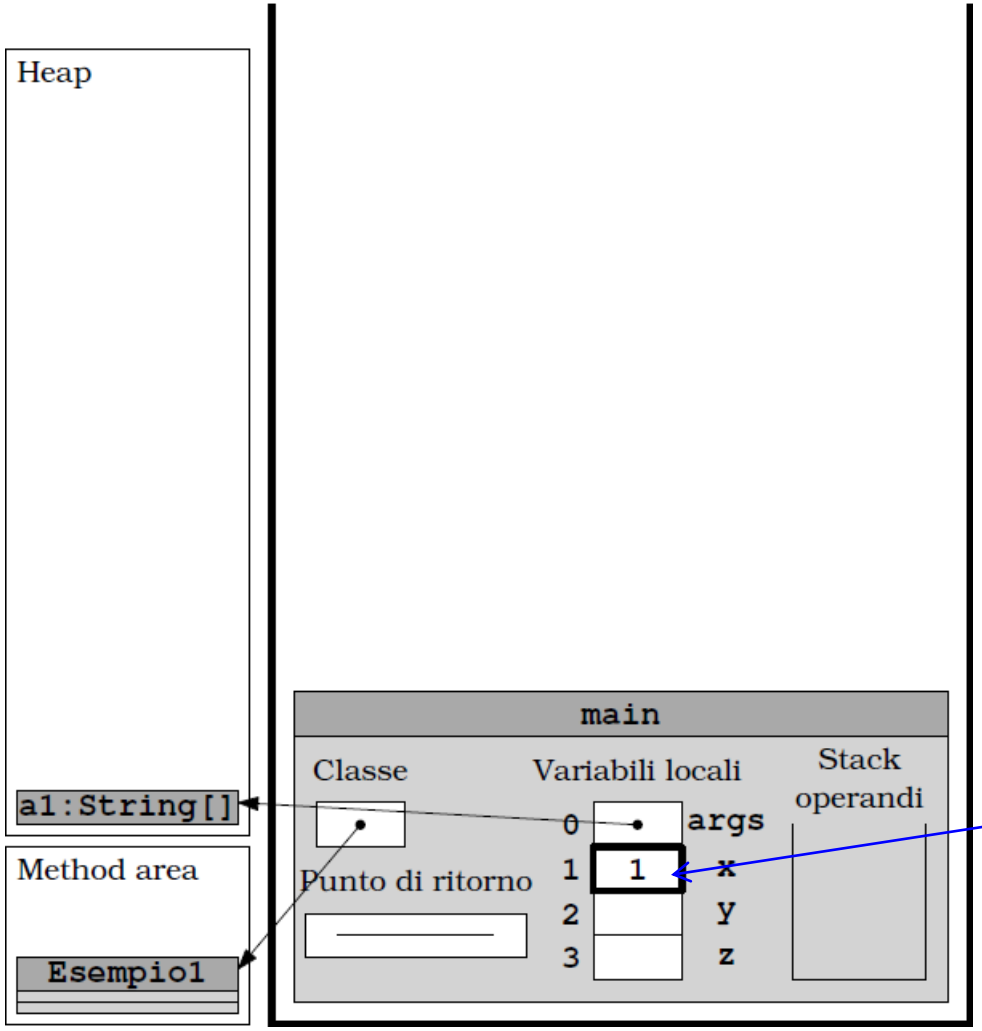
- All'avvio del programma la JVM carica la classe *Esempio1* nella method area.
- Poi crea nello heap un array di stringhe da passare come parametro al metodo *main(...)*
- Se il programma è stato lanciato senza parametri, tale array avrà dimensione 0
- Infine la JVM avvia l'esecuzione del metodo *main(...)* creando un record di attivazione per esso.

Esempio



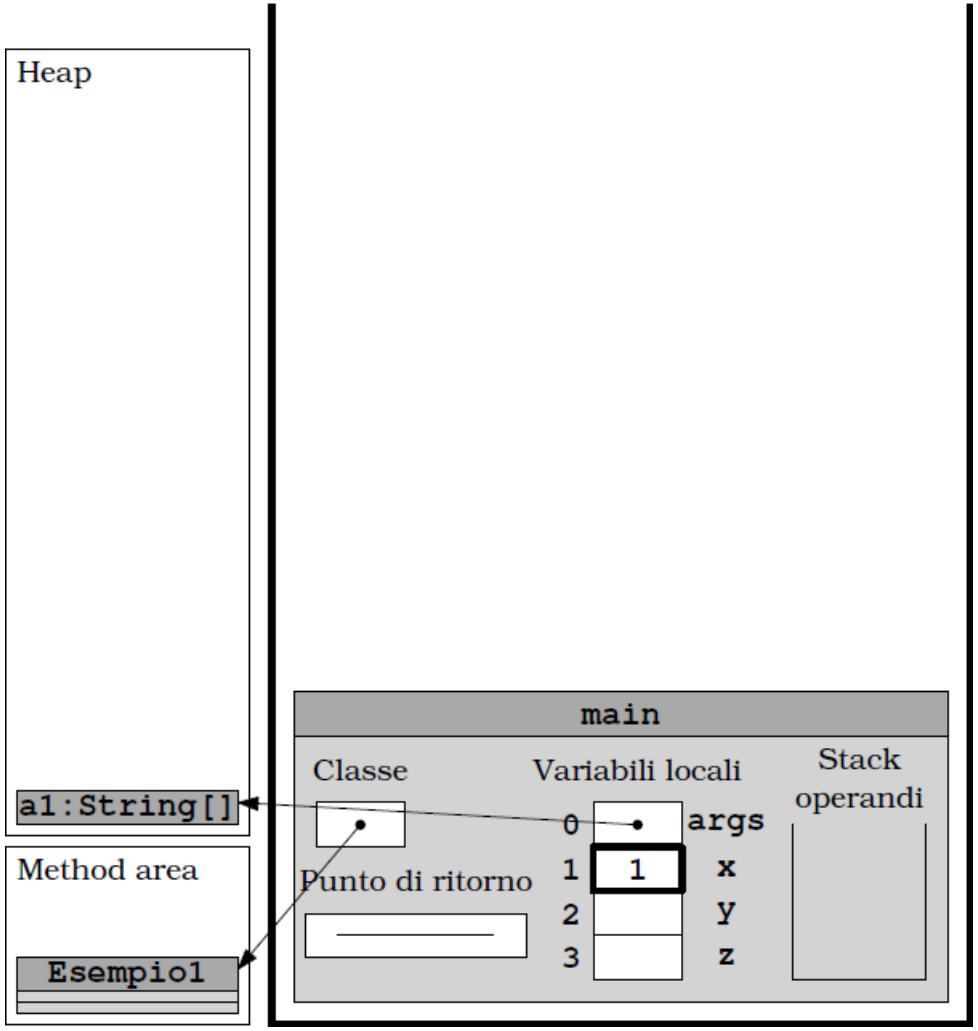

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

Esempio



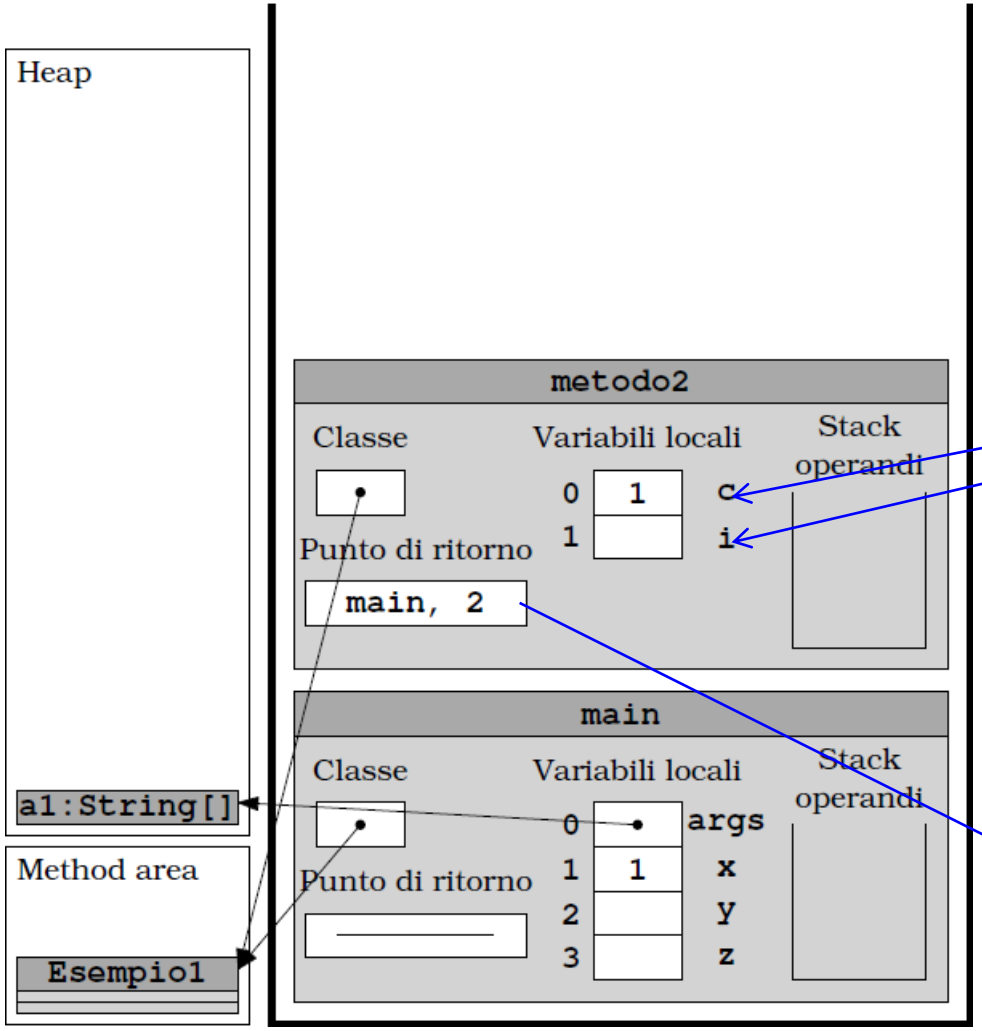

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

Esempio




```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

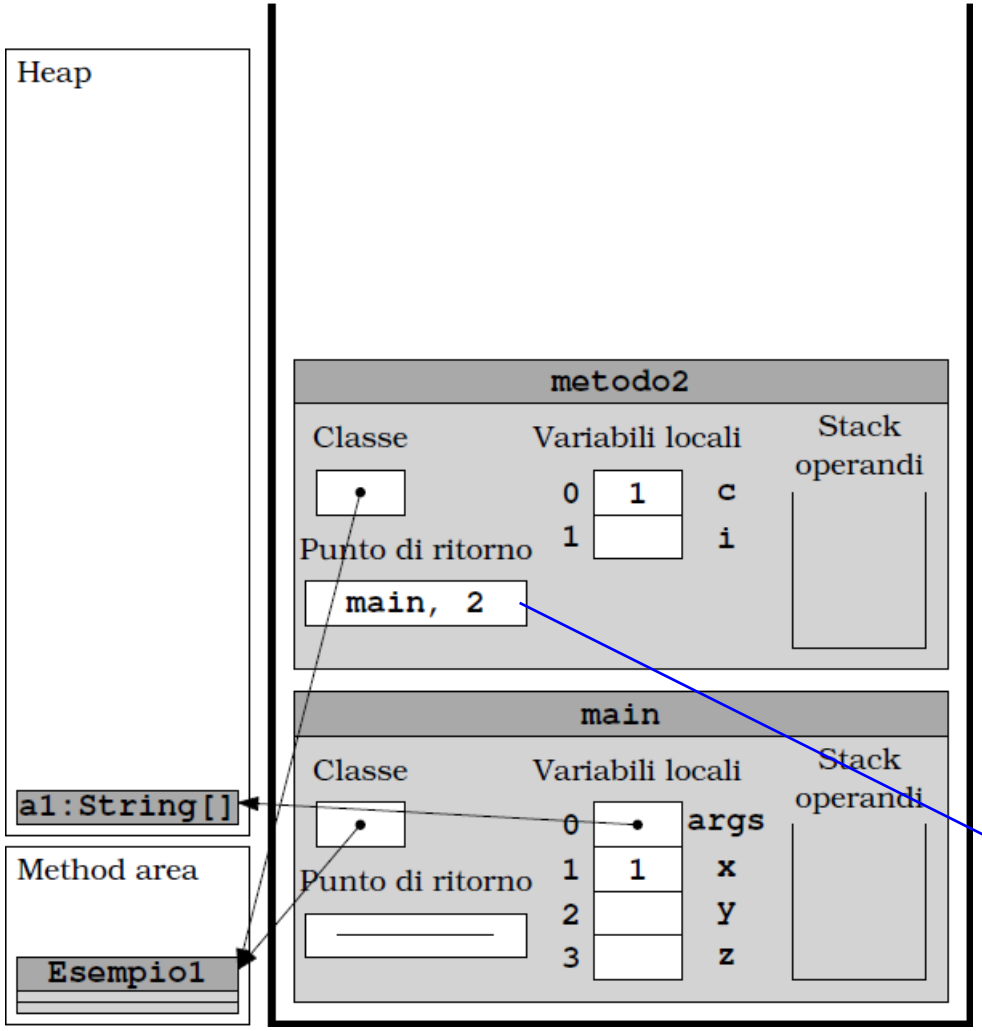
Esempio




```
int metodo2(int c){
   int i;
   i=metodo1(c,c); //metodo2,1
   return i; // metodo2, 2
}
```

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

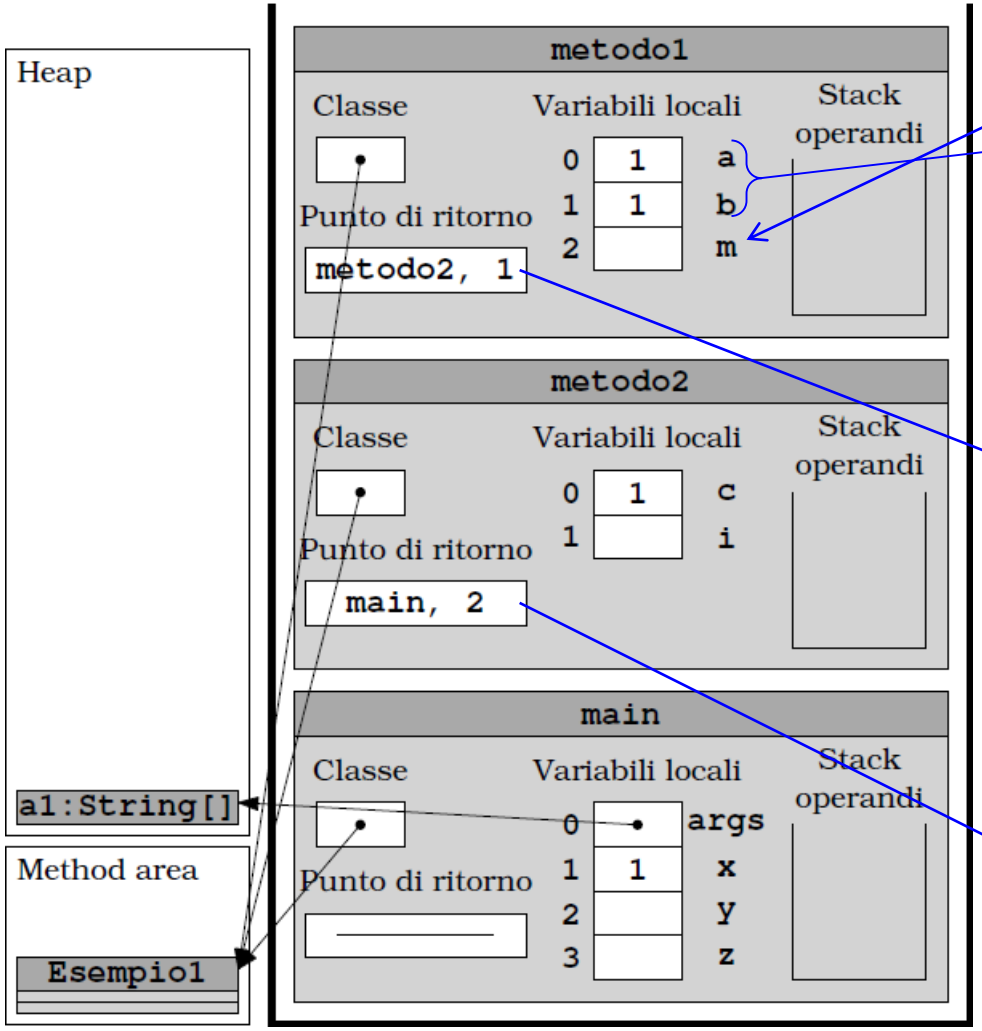
Esempio




```
int metodo2(int c){
   int i;
   i=metodo1(c,c); //metodo2,1
   return i; // metodo2, 2
}
```

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

Esempio

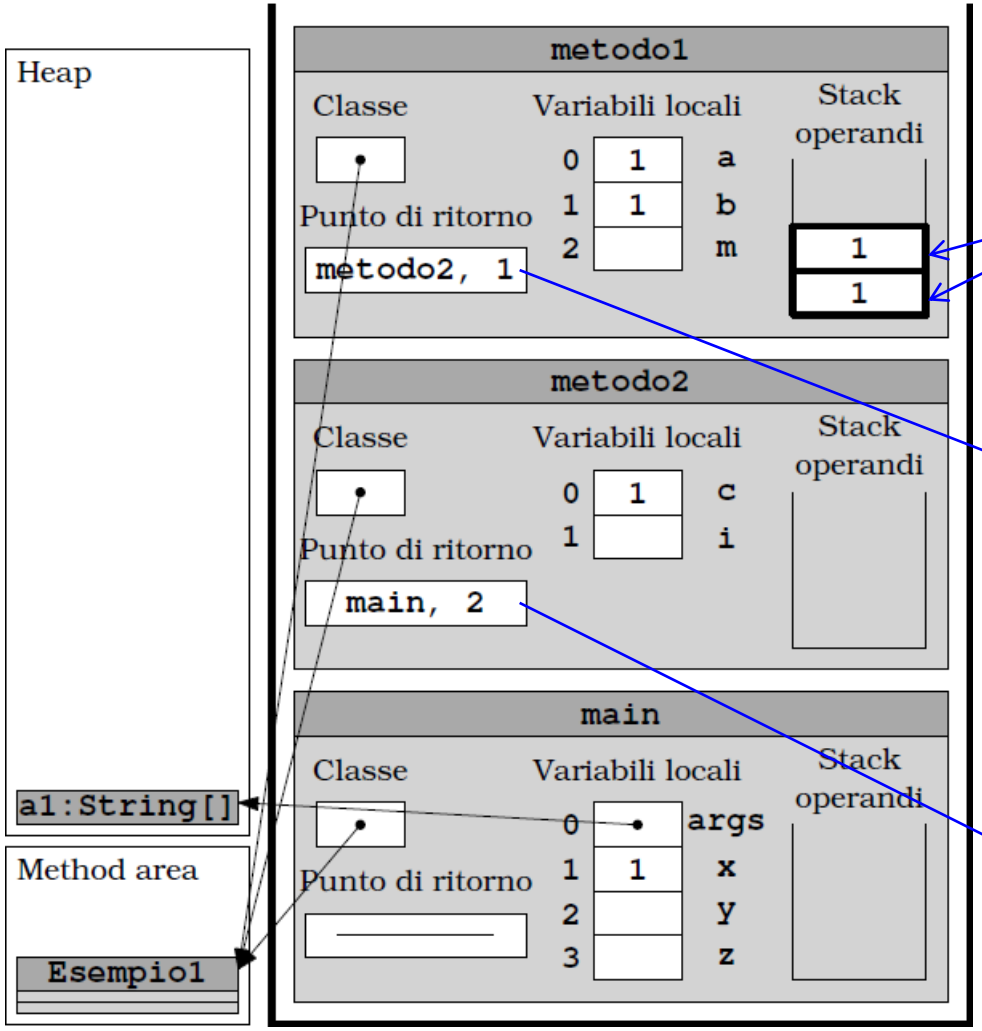



```
int metodo1(int a, int b){
   int m;
   m = a+b; // metodo1, 1
   return m; // metodo1, 2
}
```

```
int metodo2(int c){
   int i;
   i=metodo1(c,c); //metodo2,1
   return i; // metodo2, 2
}
```

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

Esempio

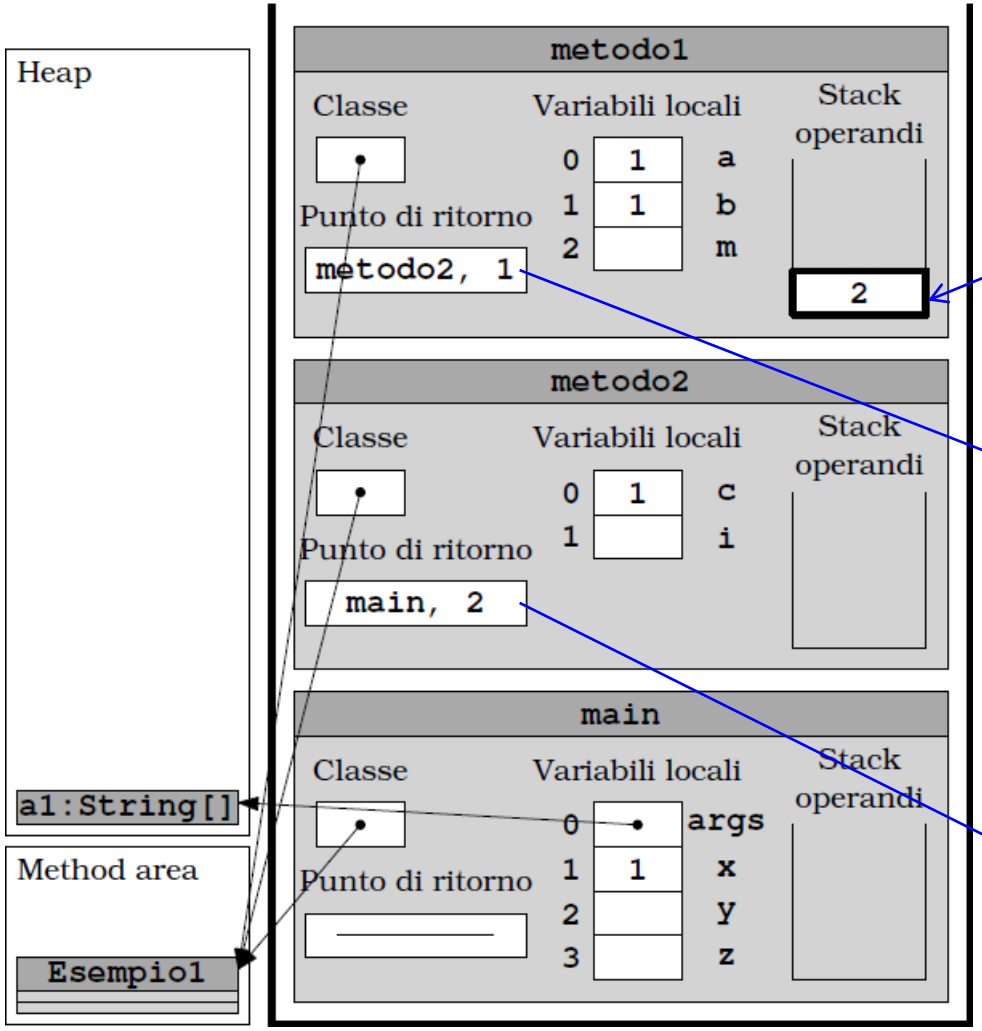



```
int metodo1(int a, int b){
   int m;
   m = a+b; // metodo1, 1
   return m; // metodo1, 2
}
```

```
int metodo2(int c){
   int i;
   i=metodo1(c,c); //metodo2,1
   return i; // metodo2, 2
}
```

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

Esempio

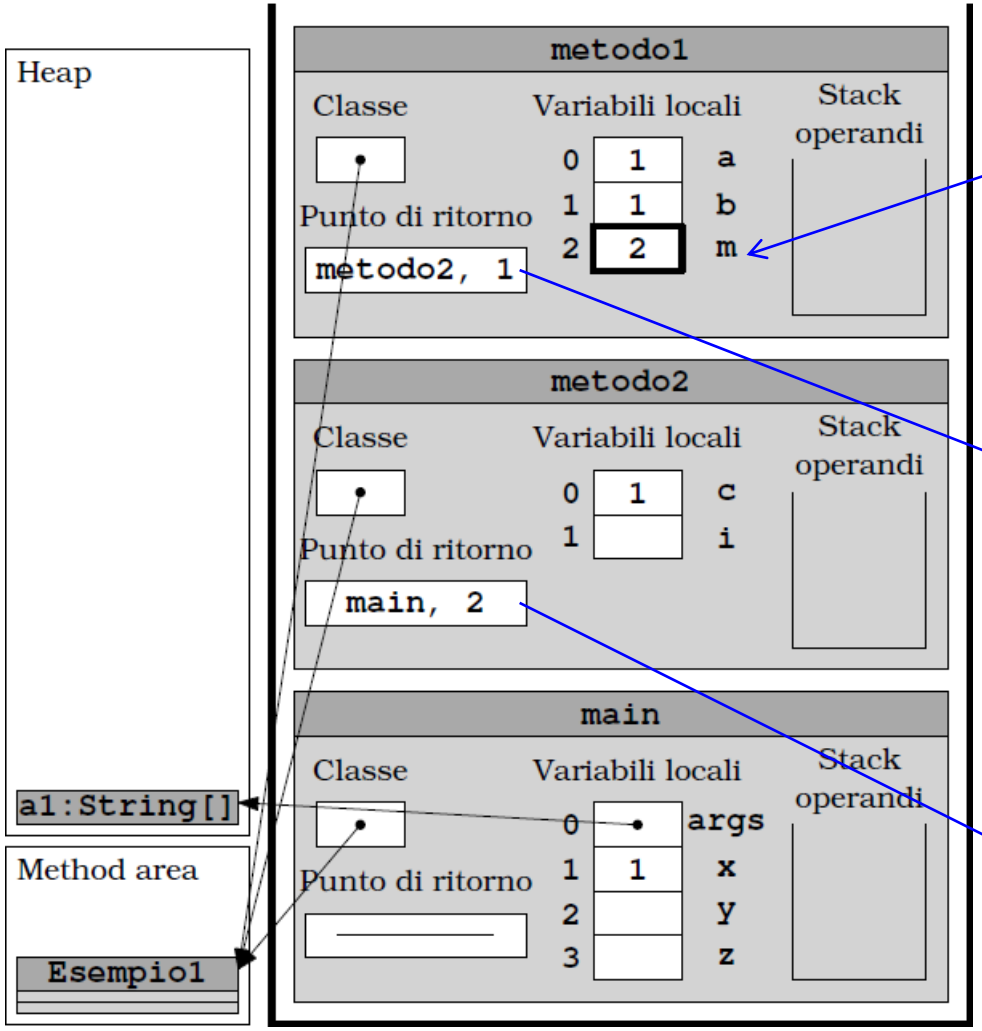



```
int metodo1(int a, int b){
   int m;
   m = a+b; // metodo1, 1
   return m; // metodo1, 2
}
```

```
int metodo2(int c){
   int i;
   i=metodo1(c,c); //metodo2,1
   return i; // metodo2, 2
}
```

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

Esempio

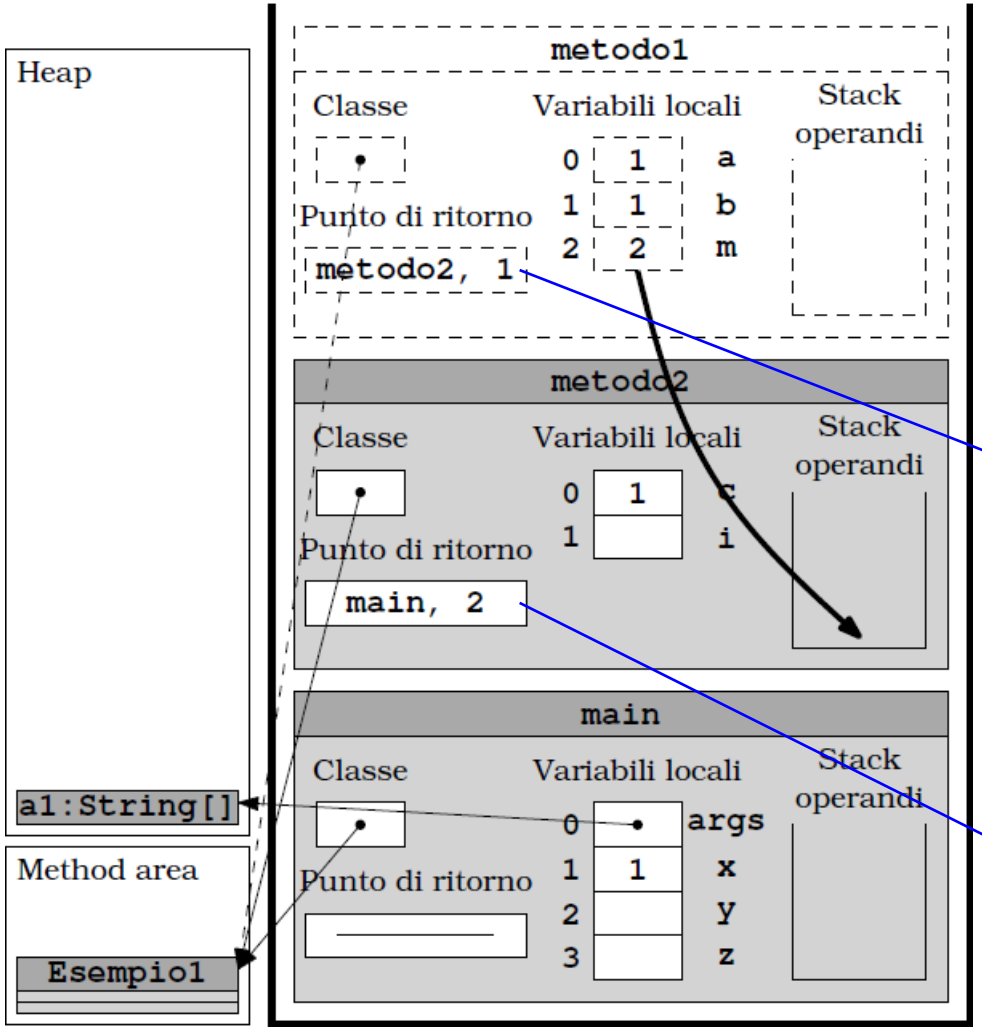



```
int metodo1(int a, int b){
   int m;
   m = a+b; // metodo1, 1
   return m; // metodo2, 2
}
```

```
int metodo2(int c){
   int i;
   i=metodo1(c,c); //metodo2,1
   return i; // metodo2, 2
}
```

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

Esempio

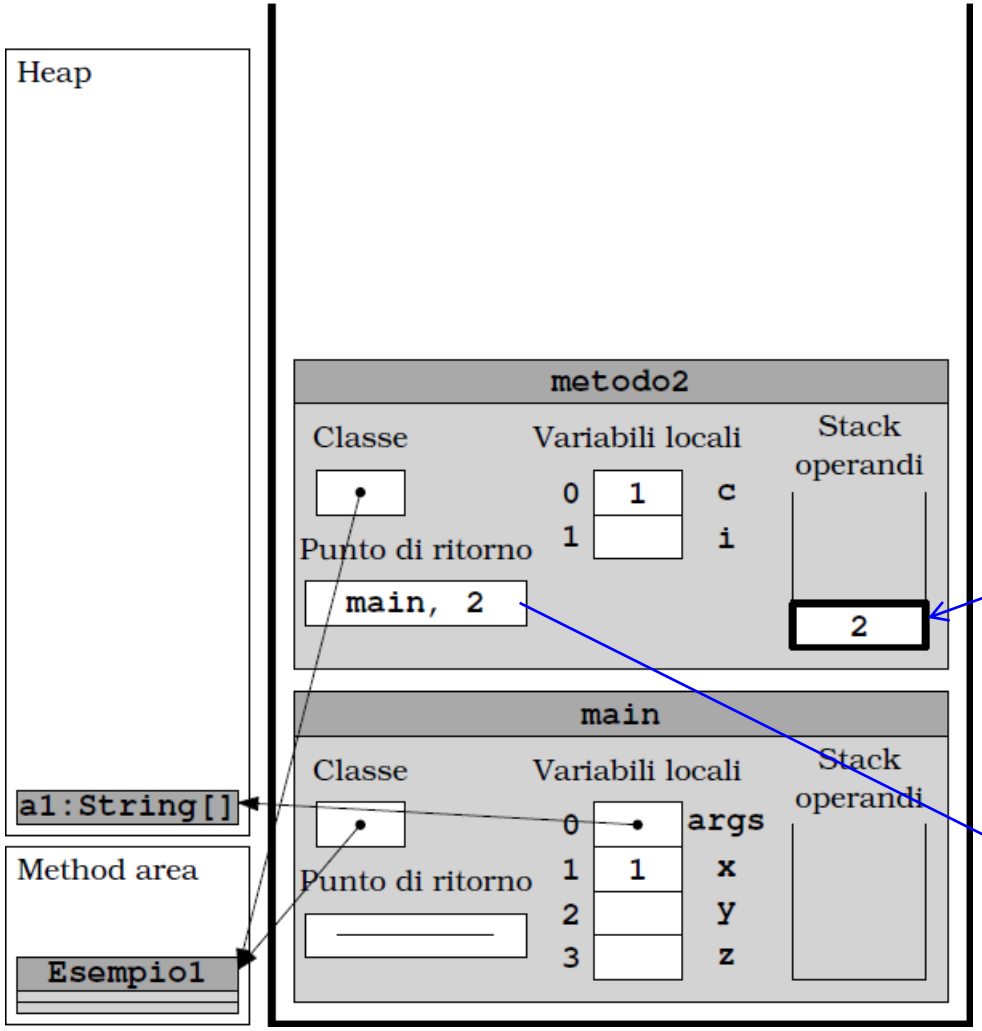



```
int metodo1(int a, int b){
   int m;
   m = a+b; // metodo1, 1
   return m; // metodo1, 2
}
```

```
int metodo2(int c){
   int i;
   i=metodo1(c,c); //metodo2,1
   return i; // metodo2, 2
}
```

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

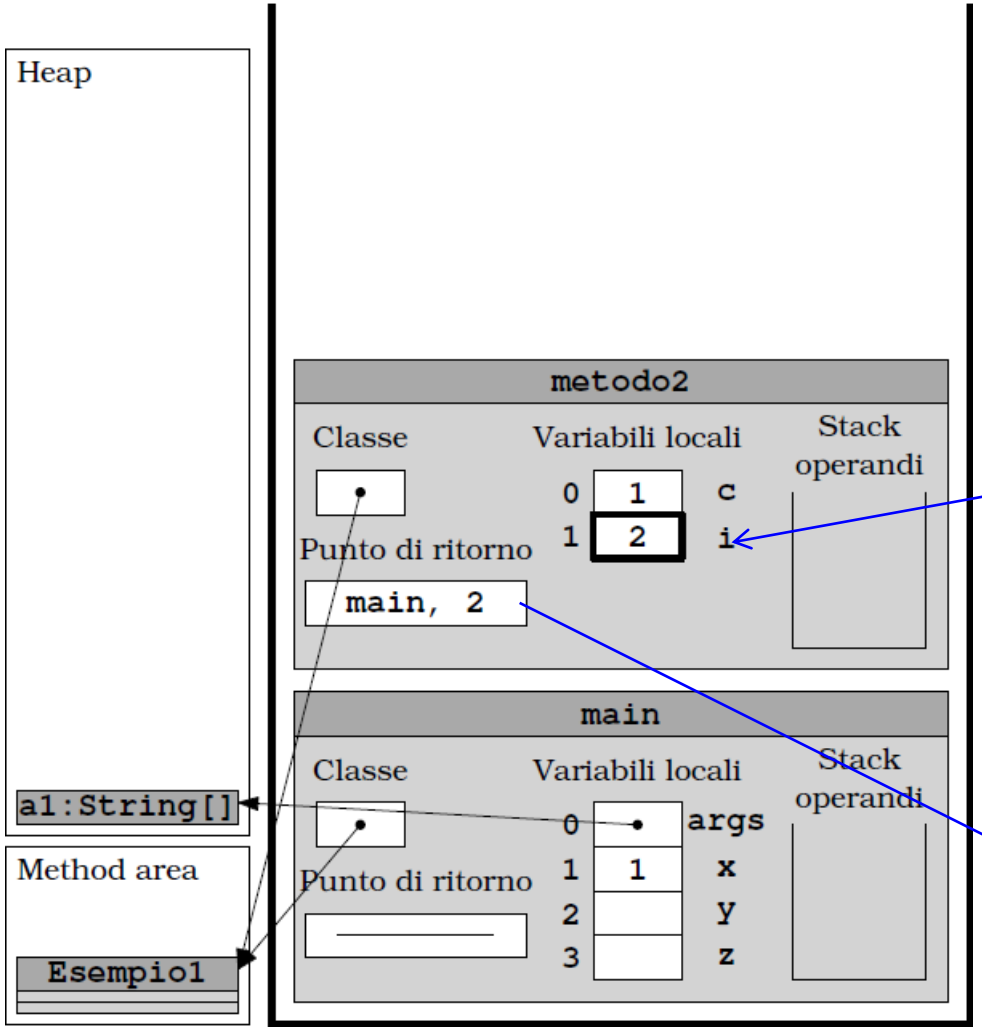
Esempio




```
int metodo2(int c){
   int i;
   i=metodo1(c,c); //metodo2,1
   return i; // metodo2, 2
}
```

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

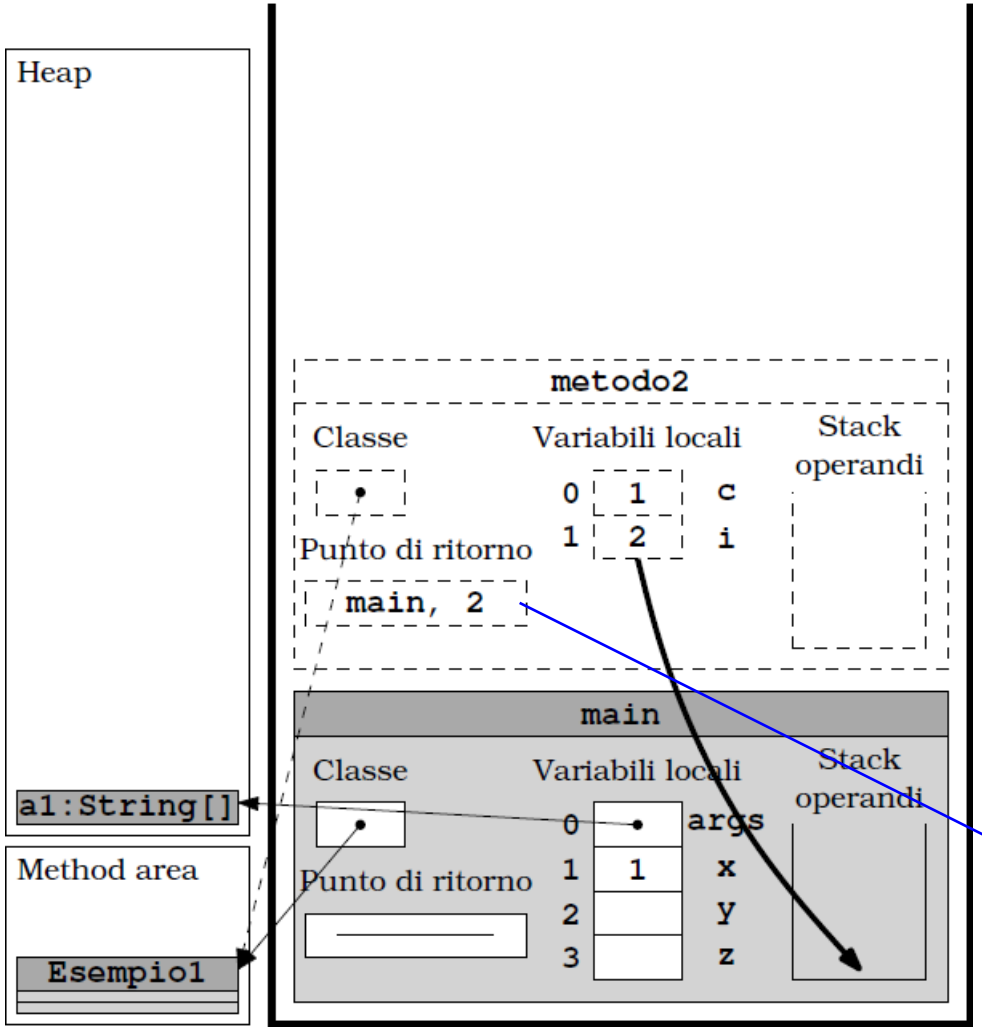
Esempio




```
int metodo2(int c){
   int i;
   i=metodo1(c,c); //metodo2,1
   return i; // metodo2, 2
}
```

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

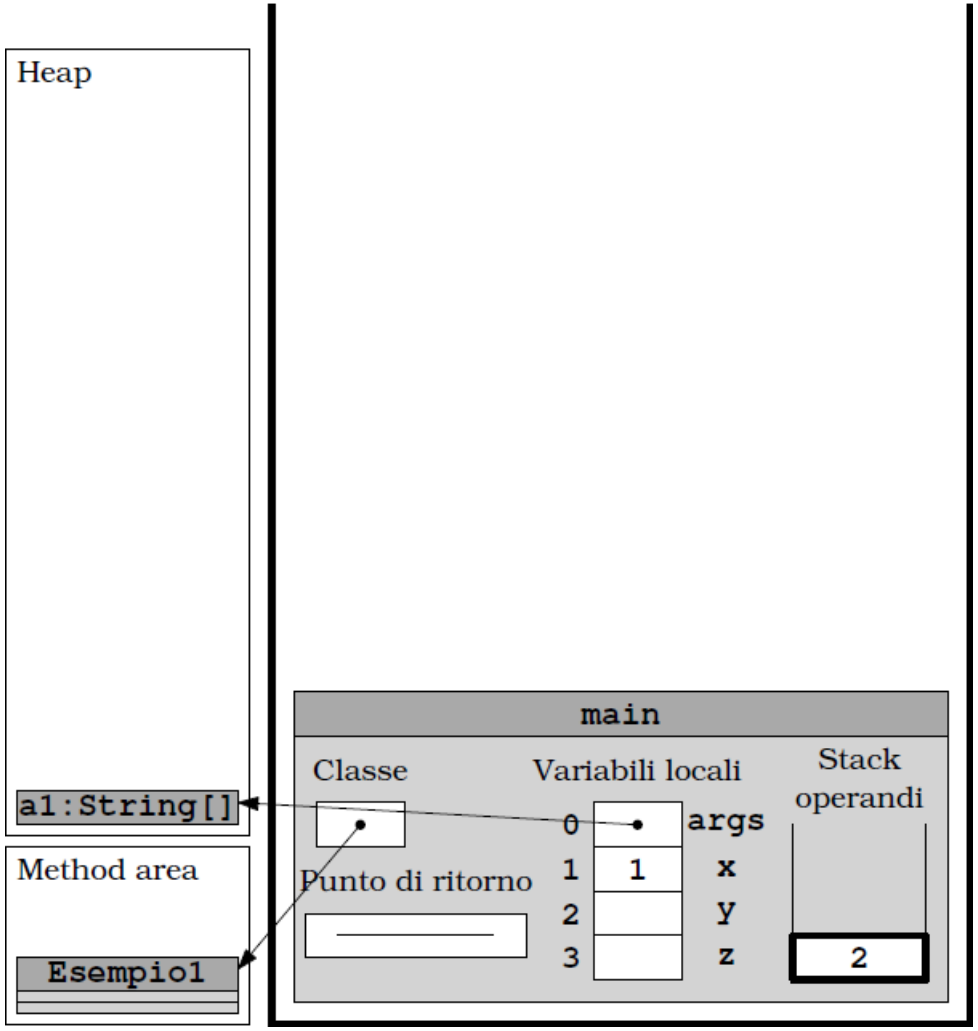
Esempio




```
int metodo2(int c){
   int i;
   i=metodo1(c,c); //metodo2,1
   return i; // metodo2, 2
}
```

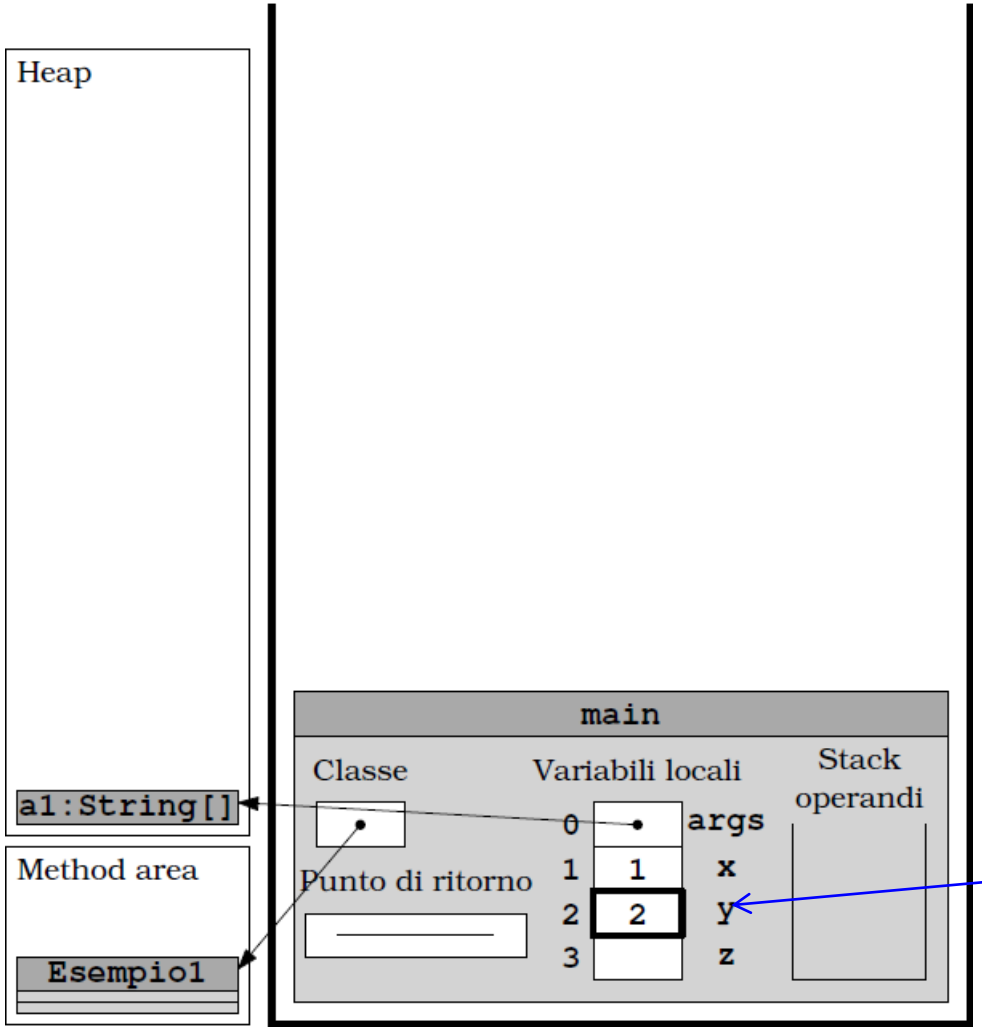
```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

Esempio



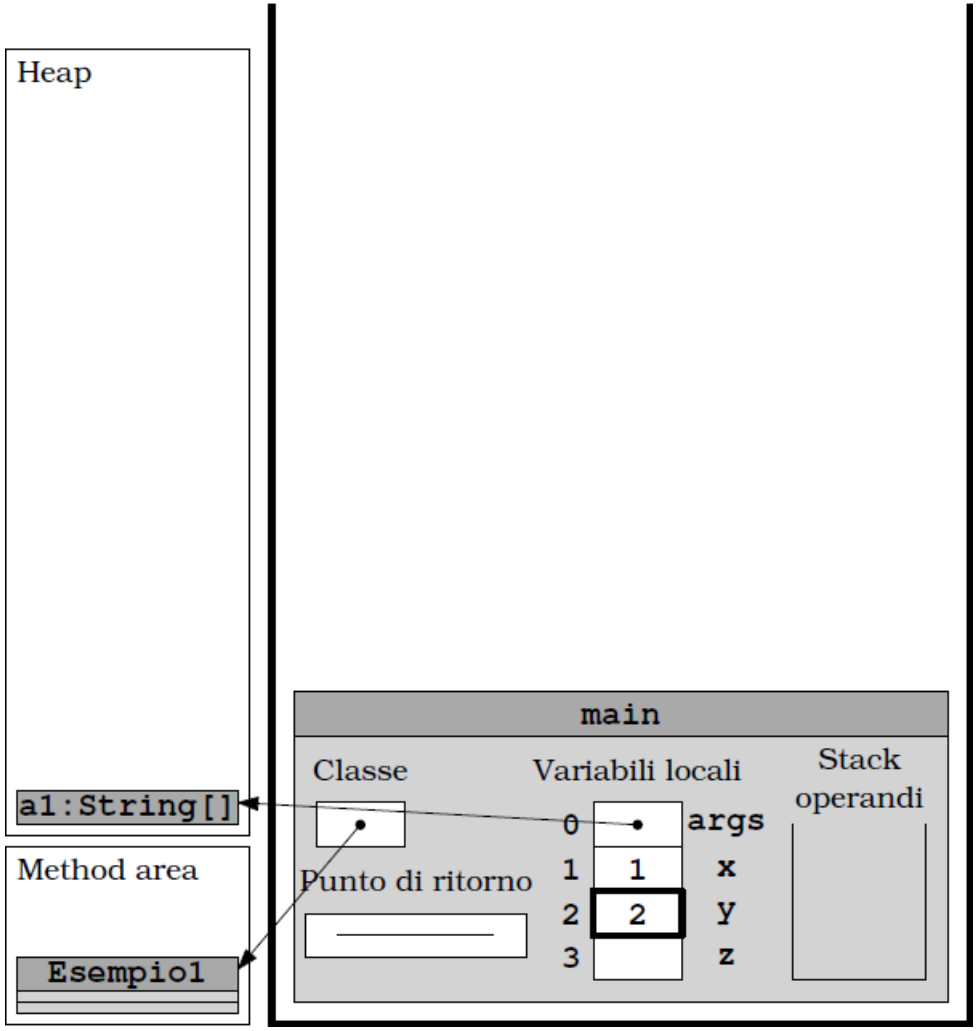

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

Esempio



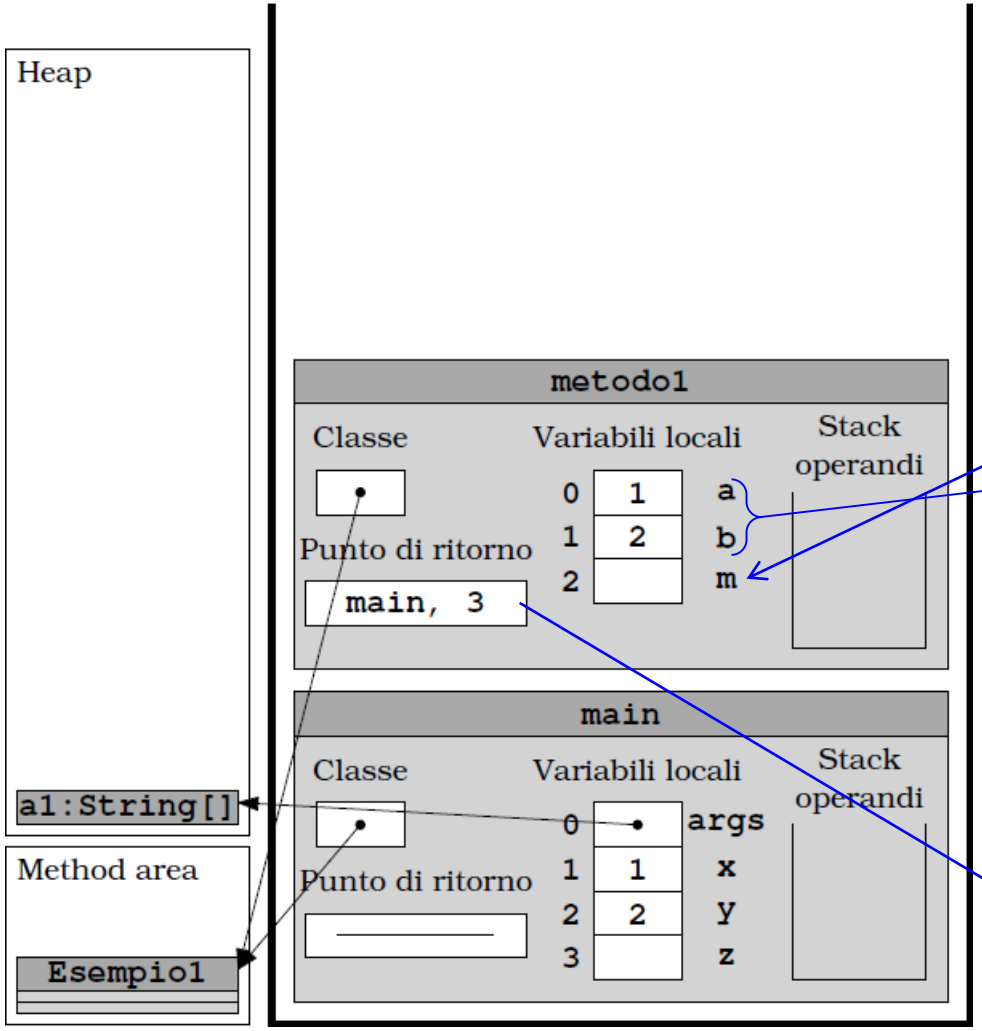

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

Esempio




```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

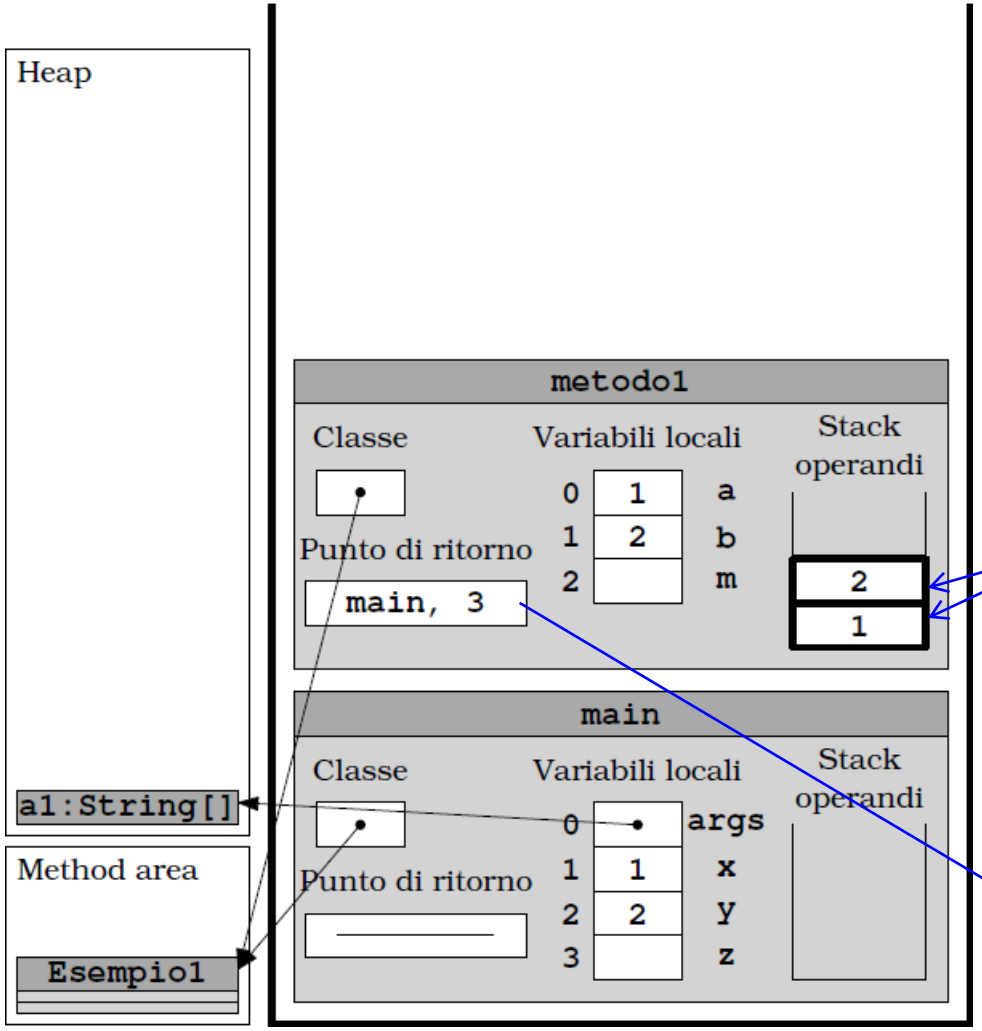
Esempio




```
int metodo1(int a, int b){
   int m;
   m = a+b; // metodo1, 1
   return m; // metodo1, 2
}
```

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

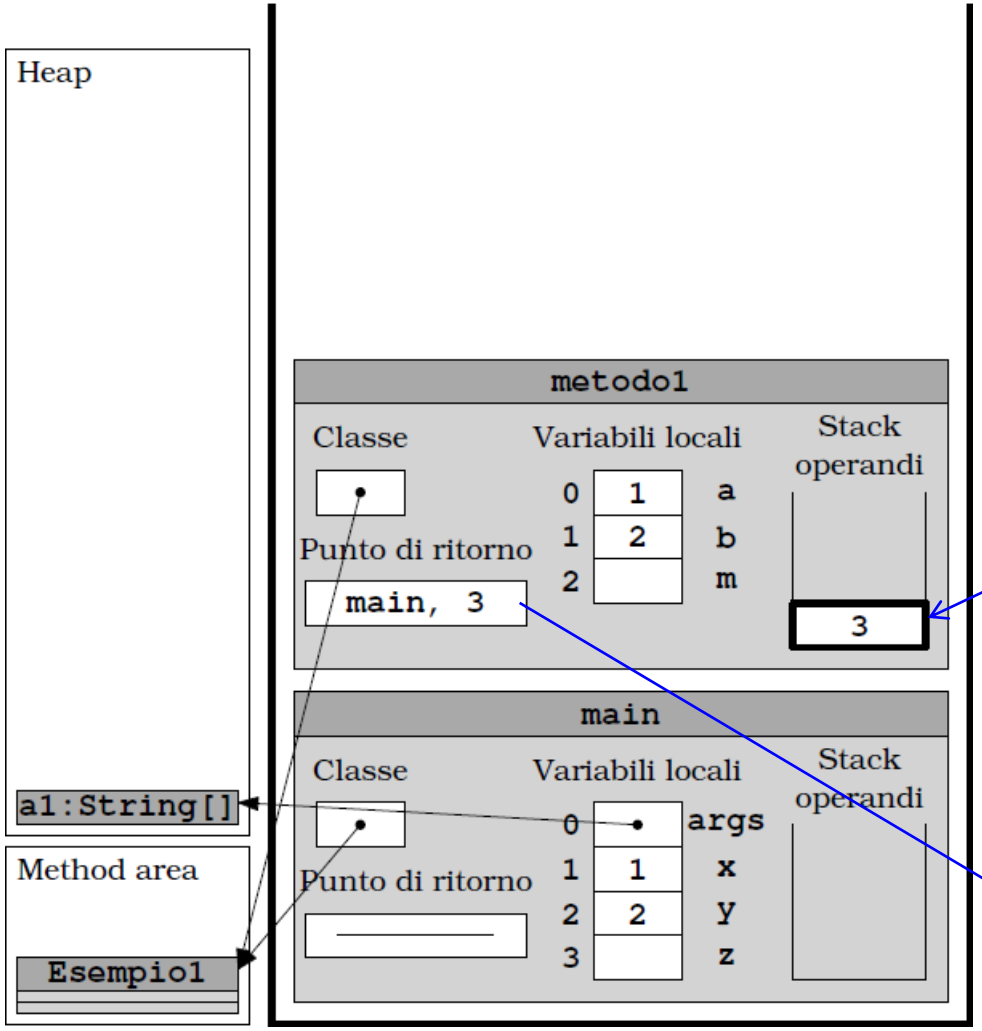
Esempio




```
int metodo1(int a, int b){
   int m;
   m = a+b; // metodo1, 1
   return m; // metodo1, 2
}
```

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

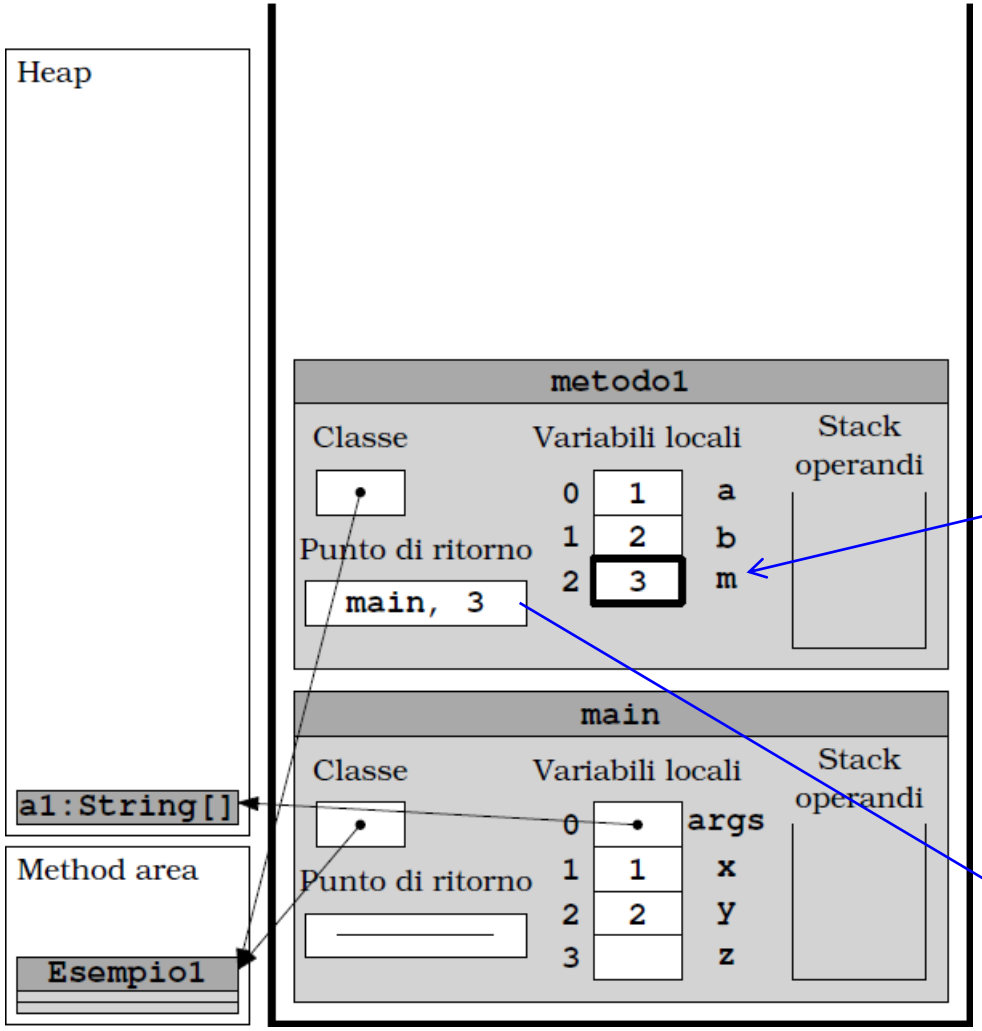
Esempio




```
int metodo1(int a, int b){
   int m;
   m = a+b; // metodo1, 1
   return m; // metodo1, 2
}
```

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

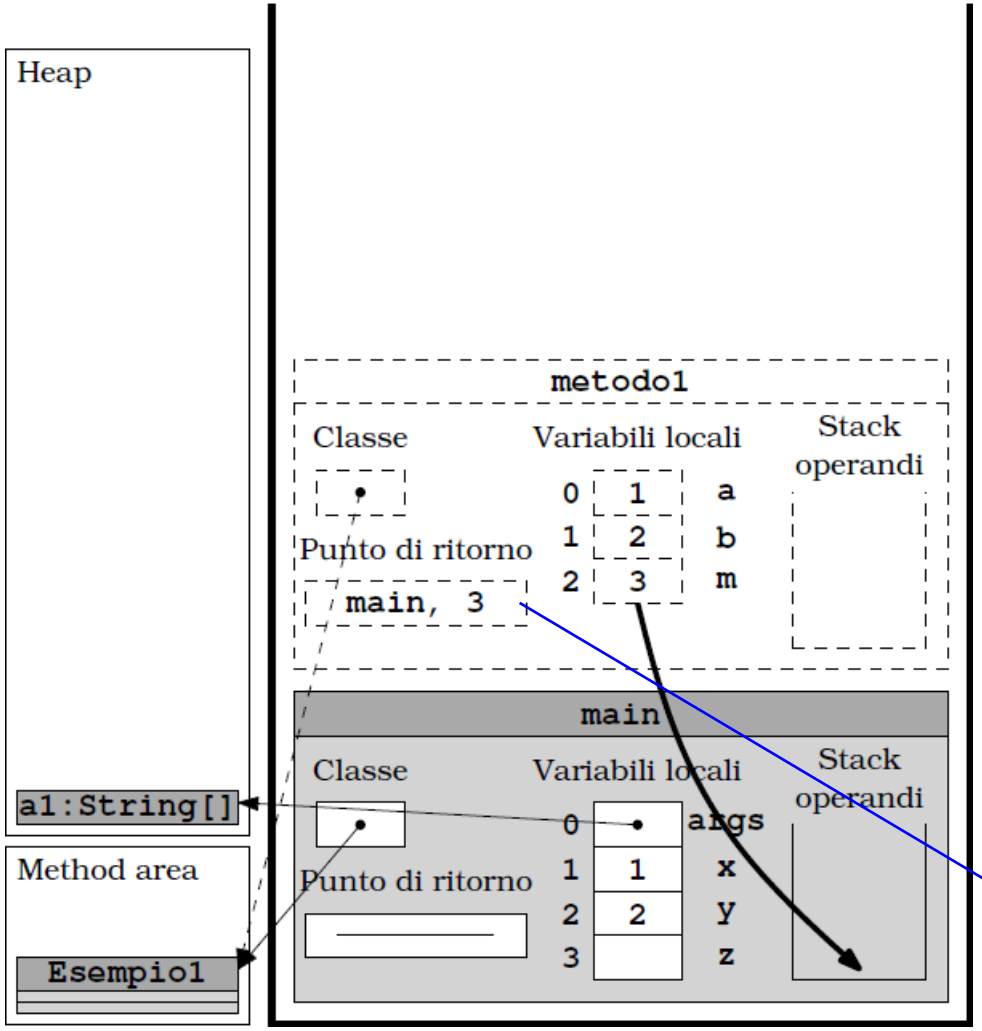
Esempio




```
int metodo1(int a, int b){
   int m;
   m = a+b; // metodo1, 1
   return m; // metodo1, 2
}
```

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

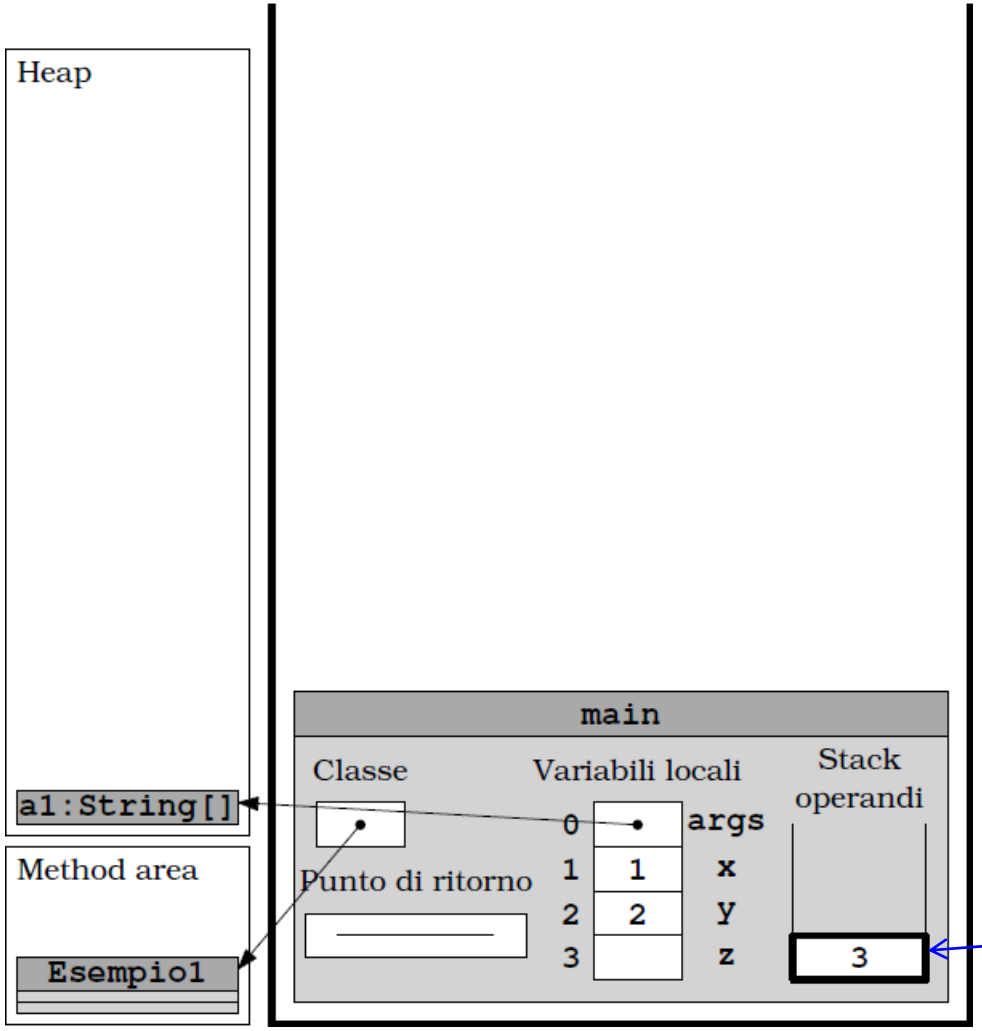
Esempio




```
int metodo1(int a, int b){
   int m;
   m = a+b; // metodo1, 1
   return m; // metodo1, 2
}
```

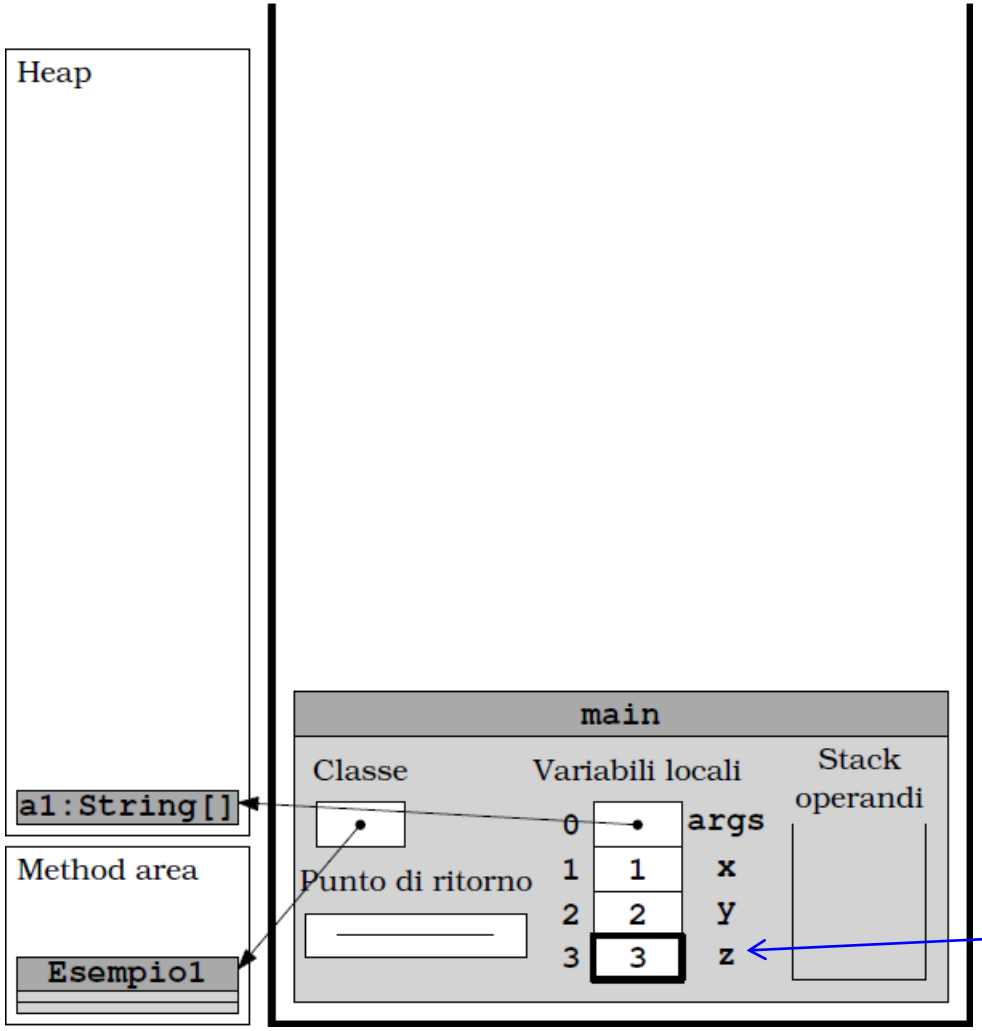
```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

Esempio



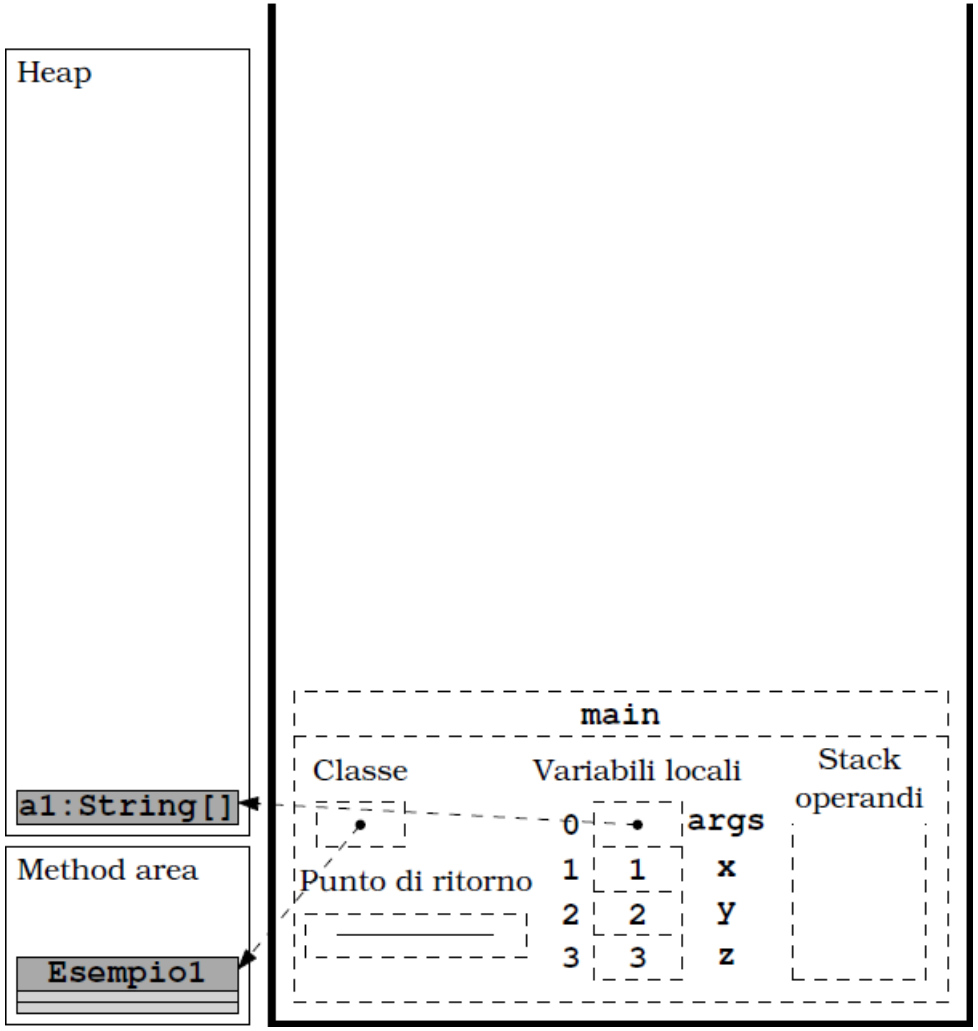

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

Esempio



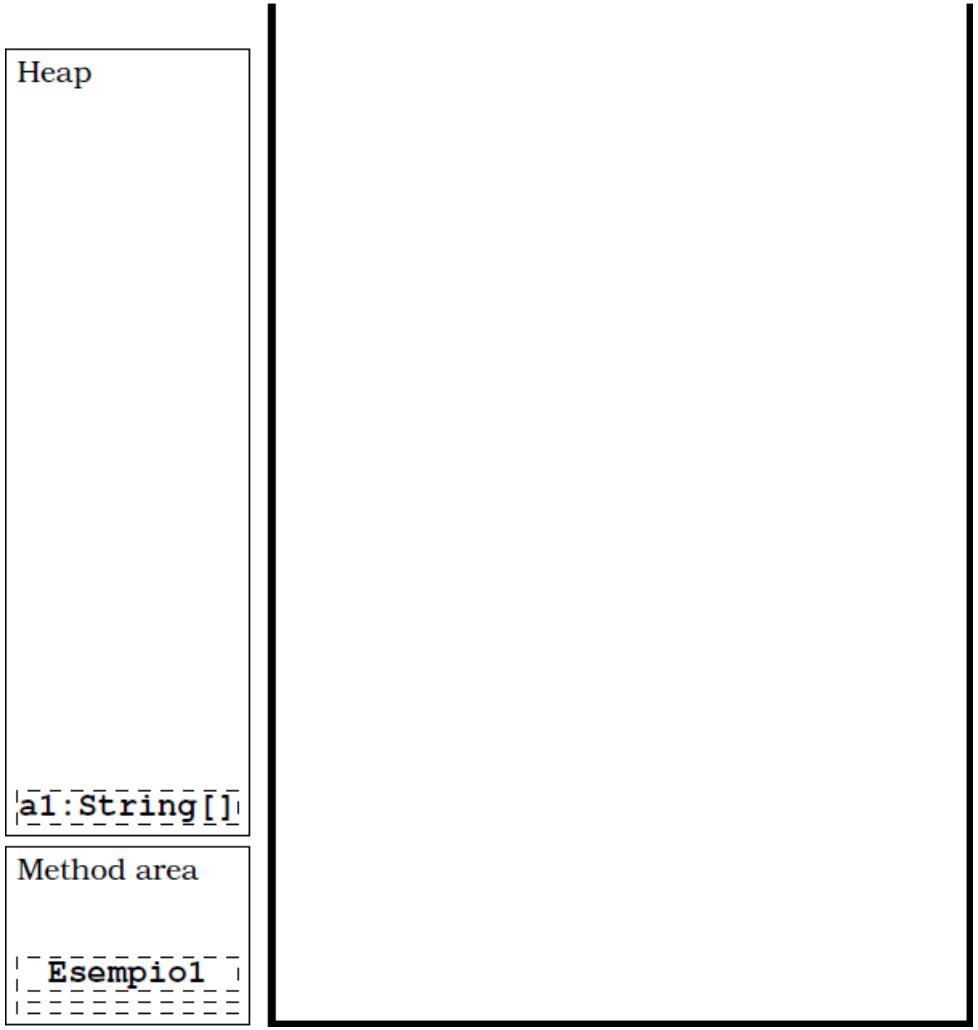

```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

Esempio




```
void main(String[] args){
   int x, y, z;
   x=1; // main, 1
   y=metodo2(x); // main, 2
   z=metodo1(x, y); // main, 3
}
```

Esempio



Lo heap

- Vediamo adesso alcuni dettagli sulla gestione dello heap
- Consideriamo parte dello heap anche la method area, cioè la porzione di memoria in cui vengono allocate le classi
- La parola heap significa “mucchio”, “ammasso”, “cumulo”
- Questo nome sta ad indicare la principale differenza rispetto alla pila di attivazione

Lo heap

- La pila di attivazione è una struttura ordinata a causa della politica LIFO
- In ogni momento esiste un unico elemento che può essere rimosso
- Nello heap invece i vari elementi vengono “ammucchiati” senza un ordine preciso
- In ogni momento è possibile rimuovere un qualunque elemento dal “mucchio”.

Allocazione di oggetti

- Quando viene creato un oggetto la JVM alloca per esso la giusta quantità di memoria all'interno dello heap
- Questa area di memoria è principalmente usata per allocare tutte le variabili di istanza dell'oggetto
- In aggiunta a queste vengono memorizzate altre informazioni necessarie alla gestione dell'oggetto, tra cui un riferimento alla classe cui appartiene (memorizzata nella method area).

Alocazione di array

- Gli array vengono gestiti in maniera simile agli altri oggetti
- Nell'area di memoria riservata ad un array vengono allocate le variabili corrispondenti ai suoi elementi
- Si ha inoltre un campo aggiuntivo che ne indica la dimensione
  - tale campo corrisponde in pratica al campo *length*

Deallocazione di oggetti

- A differenza di altri linguaggi di programmazione Java non prevede la deallocazione esplicita di un oggetto (o array)
- Mentre esiste un'istruzione per creare un oggetto non esiste un metodo per la "distruzione" di un oggetto, cioè per rilasciare la memoria allocata per un oggetto quando questa non serve più
- Quando l'oggetto non risulta più referenziato la memoria ad esso associata può essere rilasciata
- Java prevede un procedimento automatico, chiamato garbage collection, per il rilascio della memoria non più utilizzata

Caricamento di classi

- Una classe viene caricata dal classloader la prima volta che viene tirata in ballo in un programma
- Per essa viene allocata una porzione di memoria all'interno della method area
- In questa area di memoria vengono memorizzate alcune informazioni relative alla classe tra cui:
  - il nome,
  - il nome della sua super-classe diretta (se esiste)
  - se è una classe o una interface
  - il nome delle interface implementate
  - i modificatori della classe

Caricamento di classi

- Oltre alle informazioni precedenti, vengono memorizzati altri elementi:
  - informazioni sui campi definiti nella classe (nome, tipo e modificatori)
  - informazioni sui metodi definiti nella classe (nome, modificatori, tipo restituito e parametri)
  - la run-time constant pool di cui abbiamo già parlato
  - il codice bytecode di tutti i metodi
  - le variabili di classe

Deallocazione di classi

- La memoria allocata per le classi non viene deallocata per tutta la durata dell'esecuzione del programma
- Nella method area infatti non opera il meccanismo di garbage collection

Esempio di esecuzione

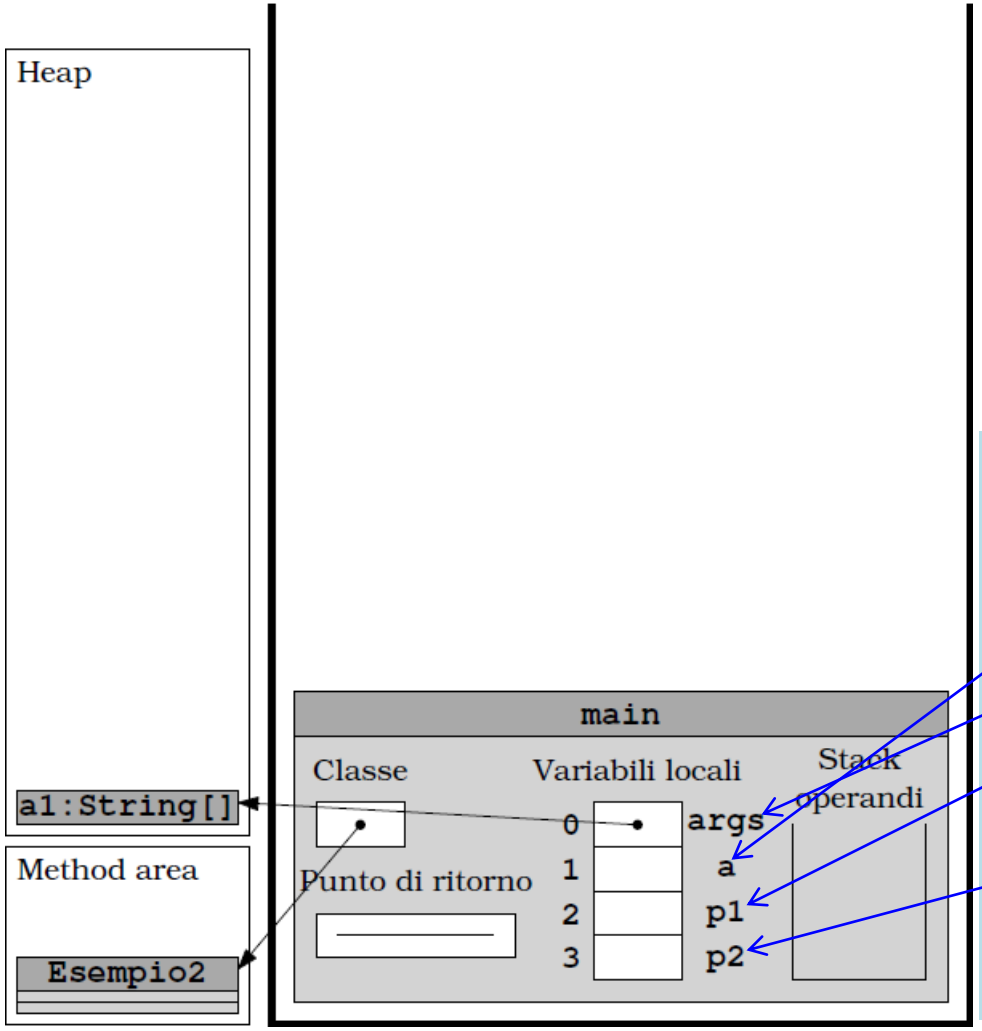
- Illustriamo adesso il funzionamento dello heap e della pila di attivazione mediante un esempio

```
public class Esempio2{
  public static void main(String[] args){
    int[] a = new int[2]; //main, 1
    a[0] = 1; //main, 2
    a[1] = 2; //main, 3
    Prova p1 = new Prova(3, a); //main, 4
    a[0] = 3; //main, 5
    a[1] = 4; //main, 6
    Prova p2 = new Prova(5, a); //main, 7
    a = null; //main, 8
    p1.metodo(p2); //main, 9
  }
}
```

Esempio di esecuzione

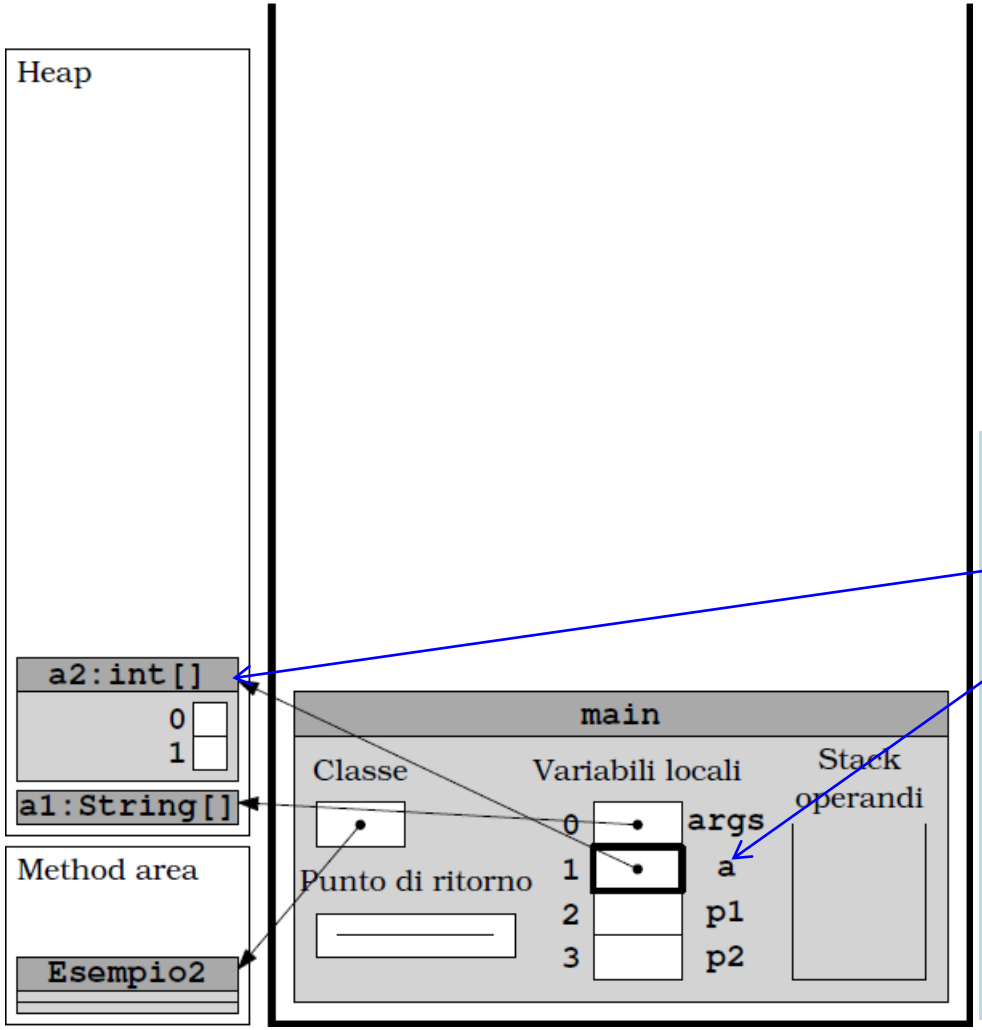
```
public class Prova{
   private int x;
   private int[] a;

   public Prova(int x, int[] a){
     this.x = x;
     this.a = new int[a.length];
     for (int i = 0; i<a.length; i++){
       this.a[i] = a[i];
     }
   }

   public void metodo(Prova p){
     if (this.a.length==p.a.length){
       for (int i = 0; i<a.length; i++){
         this.a[i] += p.a[i];
       }
     }
   }
}
```

Esempio



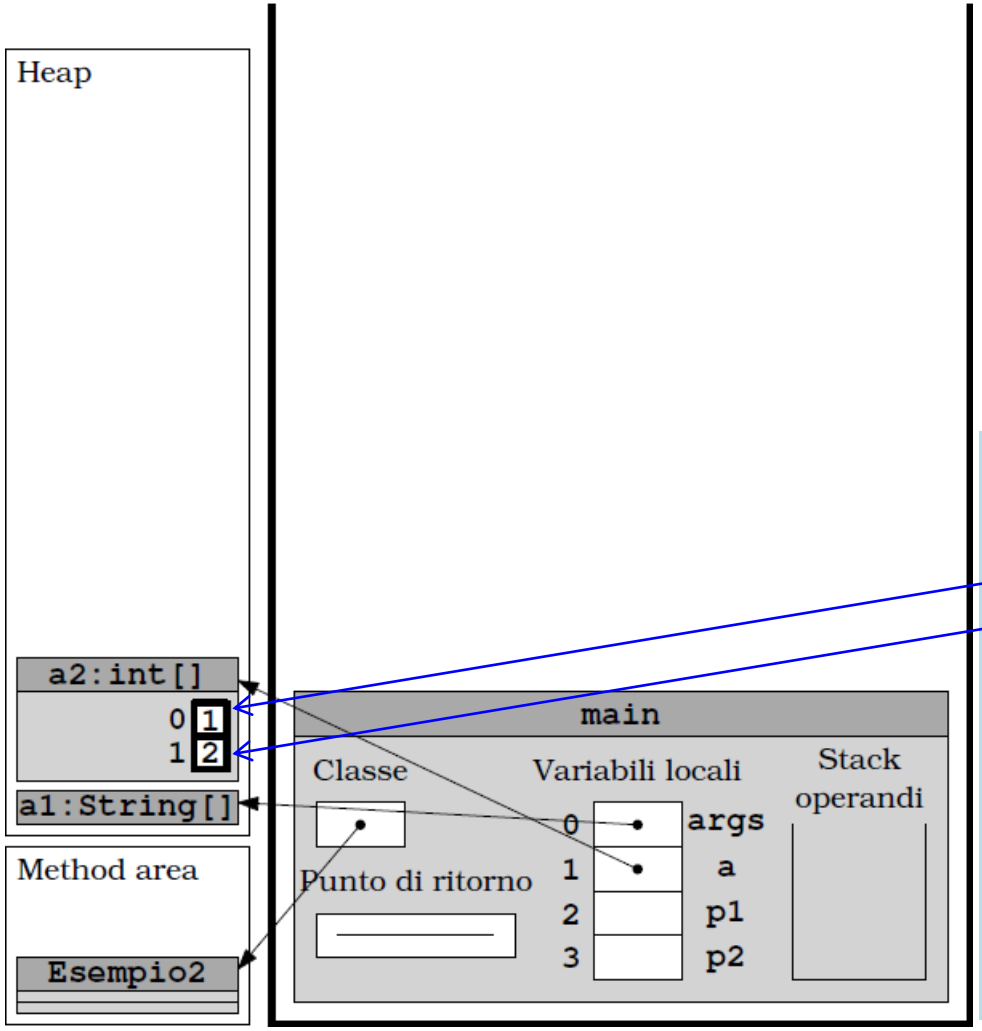

```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

Esempio



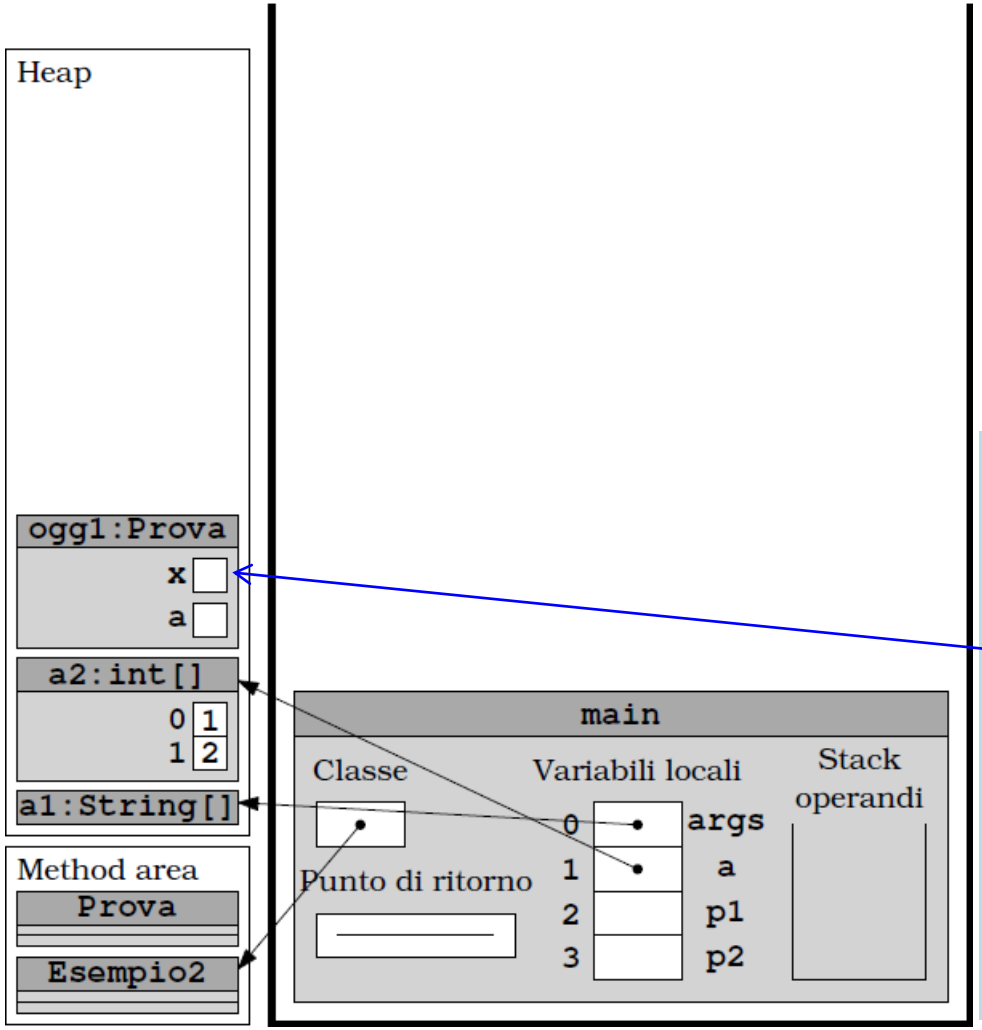

```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

Esempio



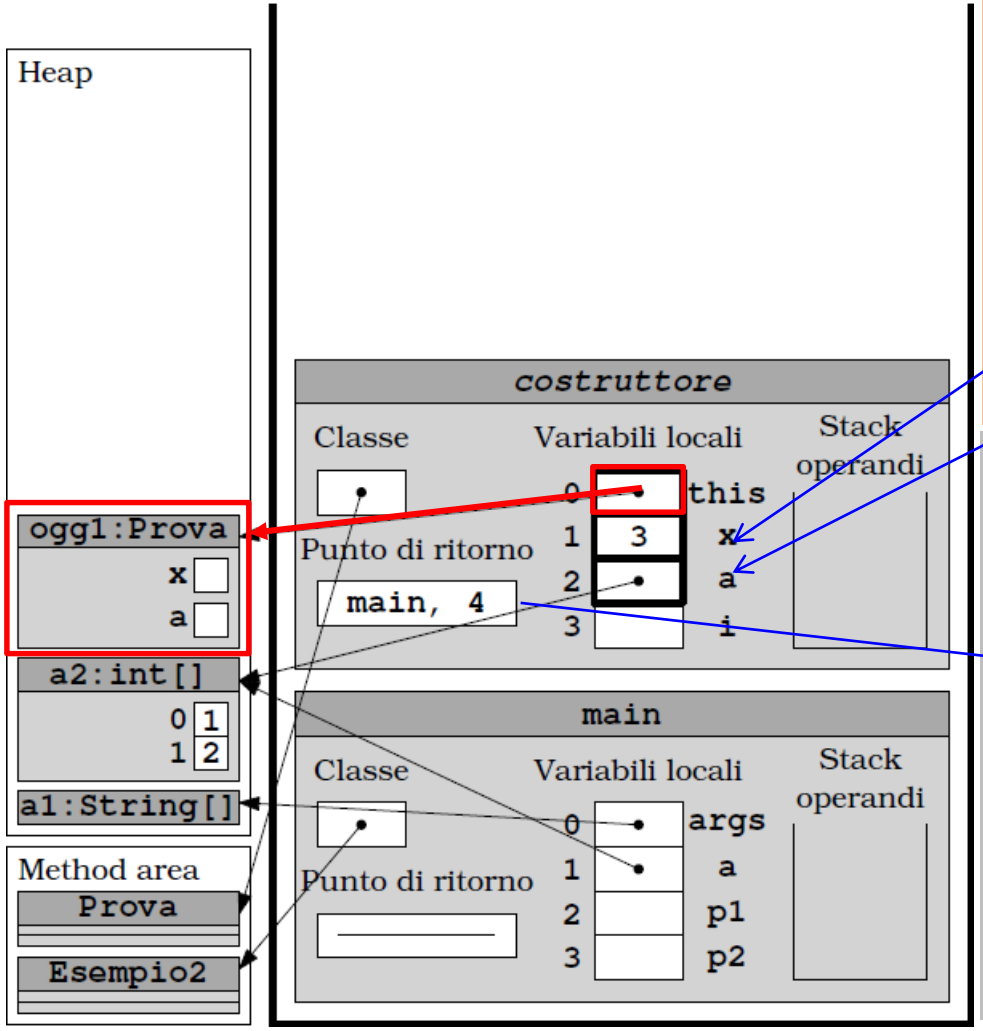

```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

Esempio




```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

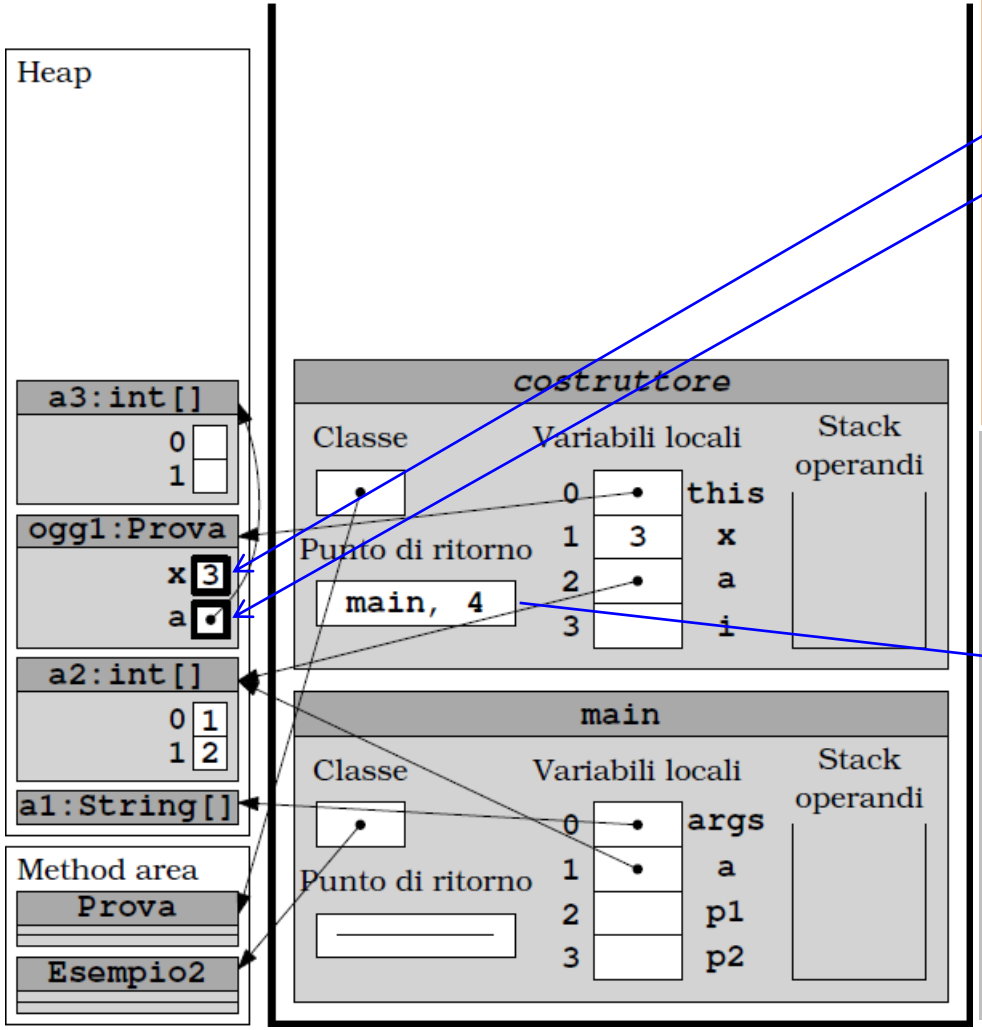
Esempio




```
public Prova(int x, int[] a){
  this.x = x;
  this.a = new int[a.length];
  for(int i=0;i<a.length;i++){
    this.a[i] = a[i];
  }
}
```

```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

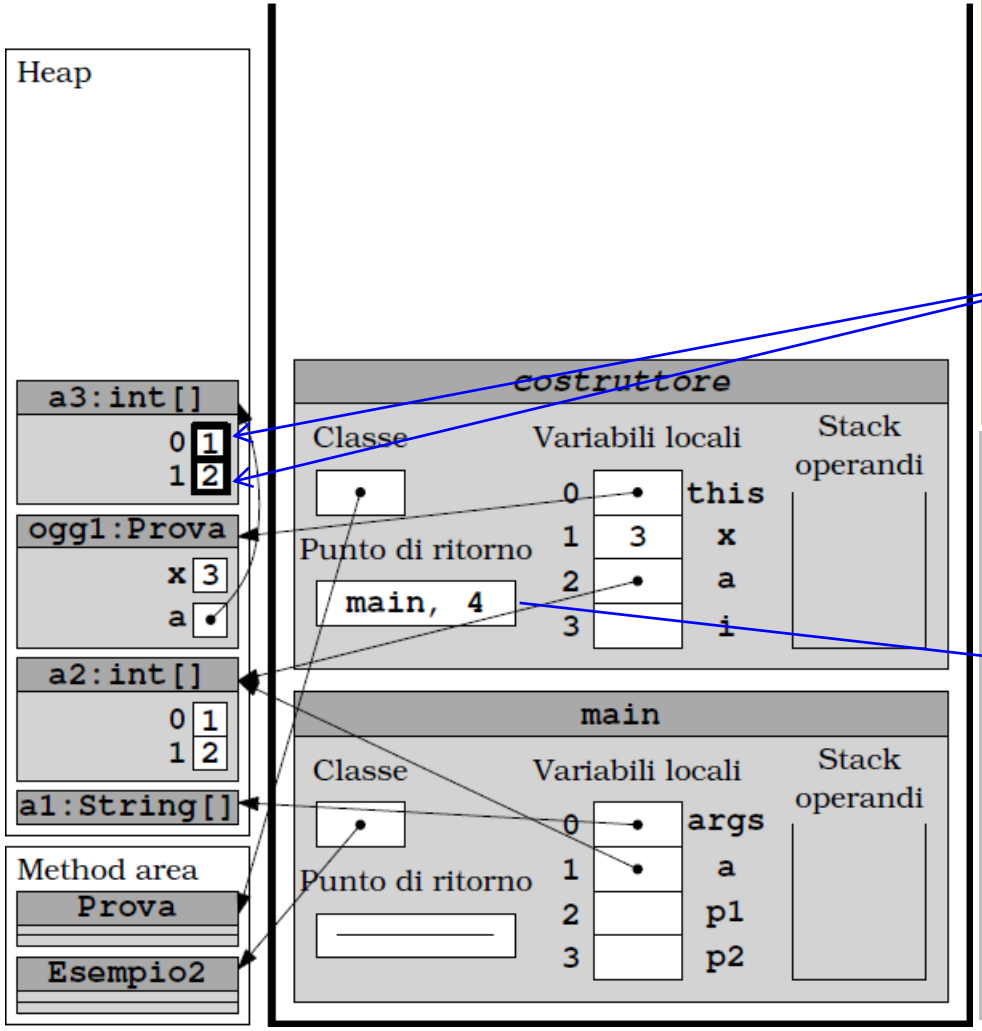
Esempio




```
public Prova(int x, int[] a){
   this.x = x;
   this.a = new int[a.length];
   for(int i=0;i<a.length;i++){
     this.a[i] = a[i];
   }
}
```

```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

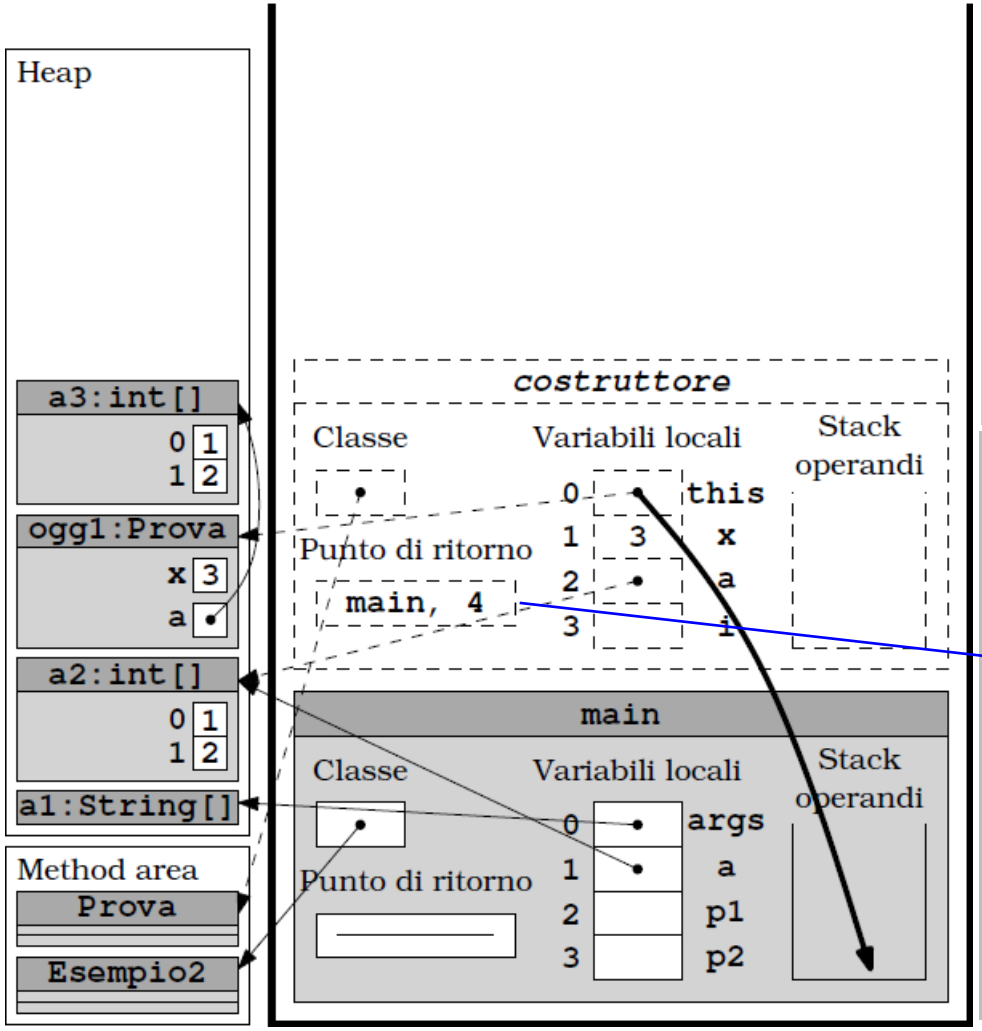
Esempio




```
public Prova(int x, int[] a){
  this.x = x;
  this.a = new int[a.length];
  for(int i=0;i<a.length;i++){
    this.a[i] = a[i];
  }
}
```

```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

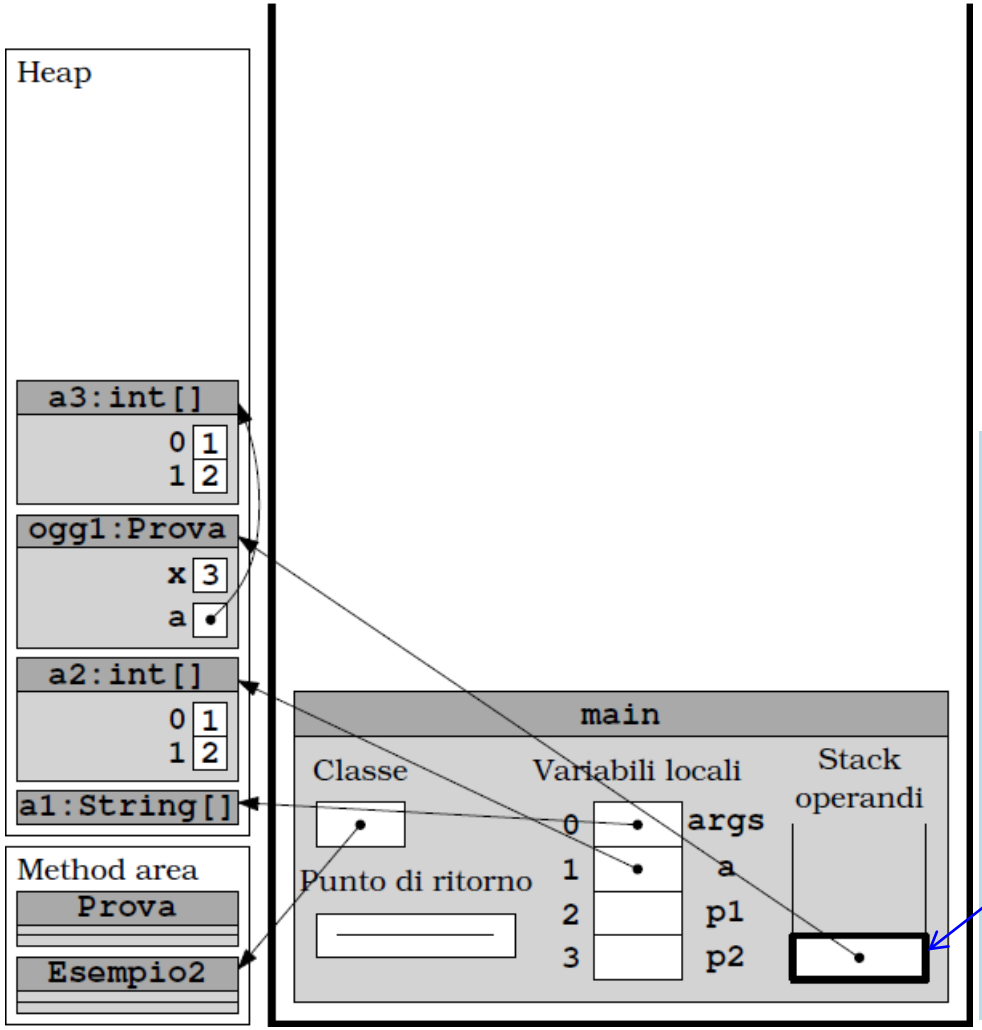
Esempio




```
public Prova(int x, int[] a){
  this.x = x;
  this.a = new int[a.length];
  for(int i=0;i<a.length;i++){
    this.a[i] = a[i];
  }
}
```

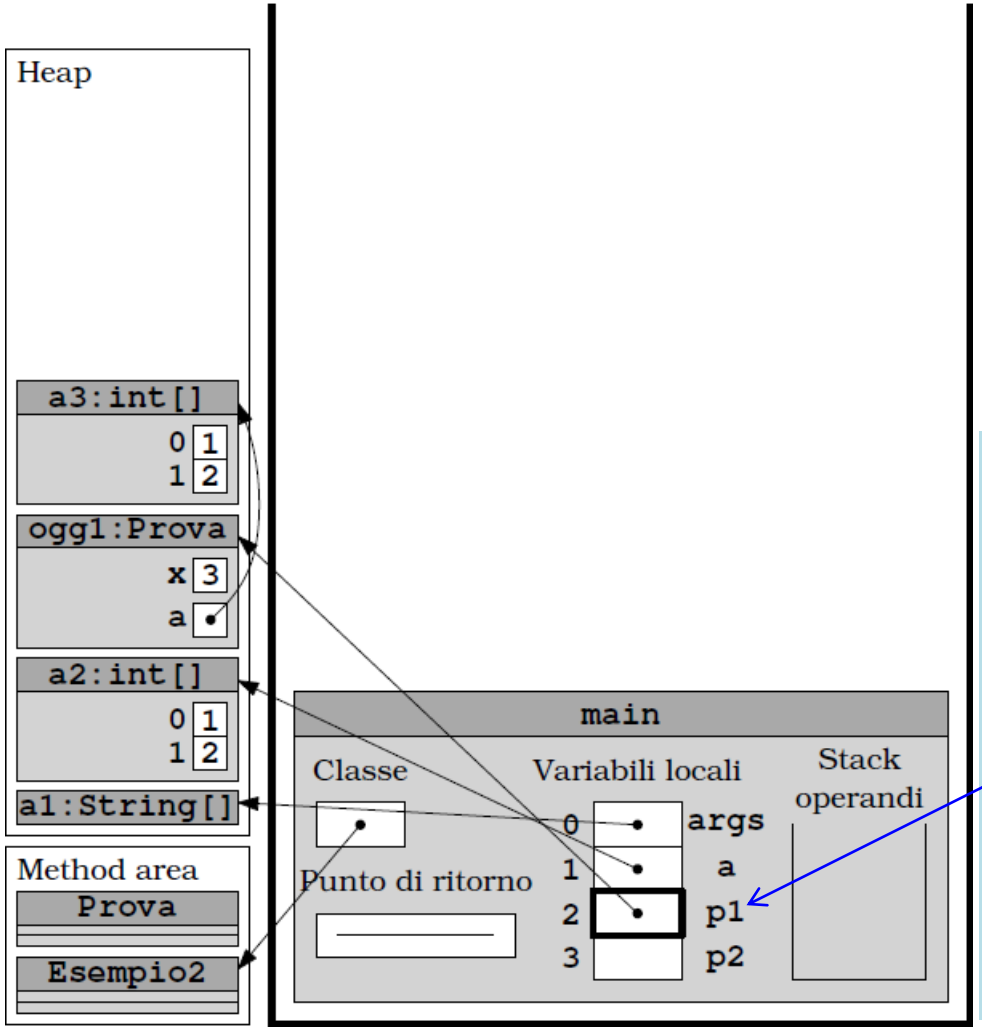
```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

Esempio



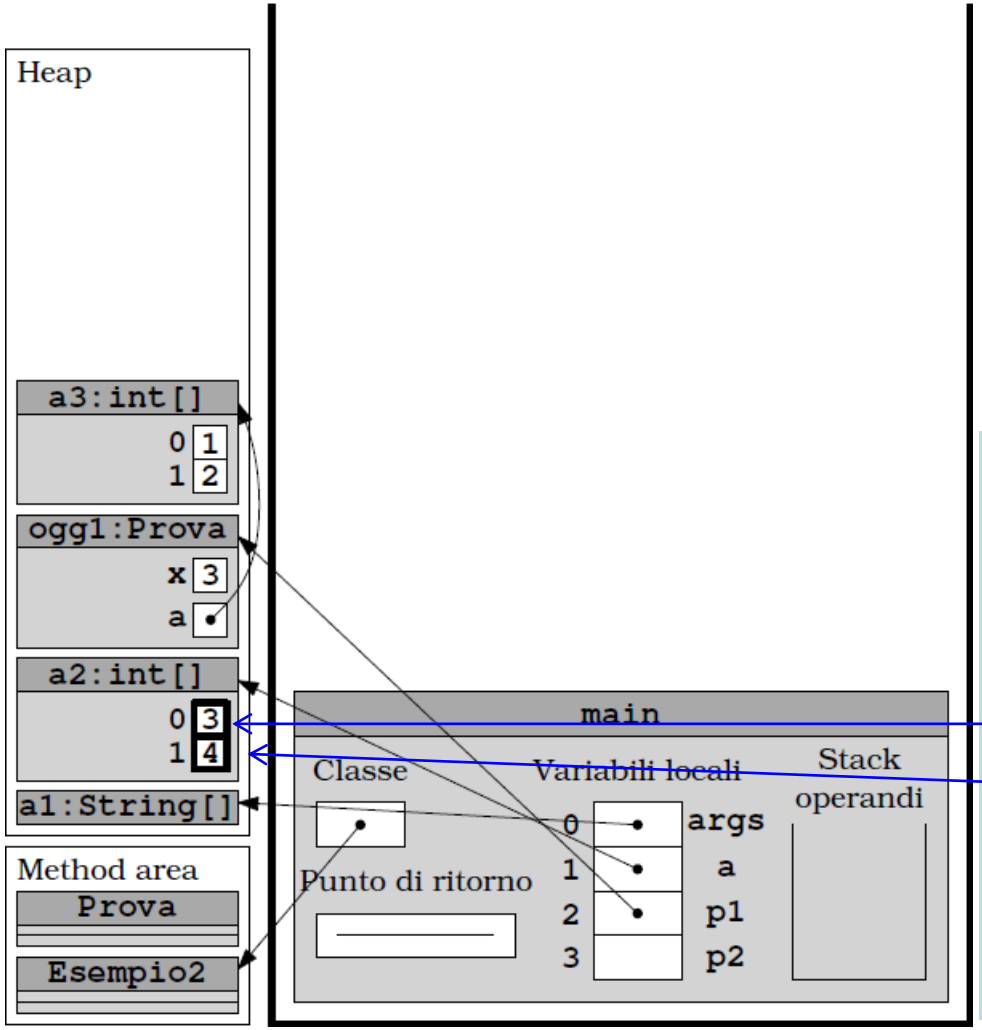

```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

Esempio



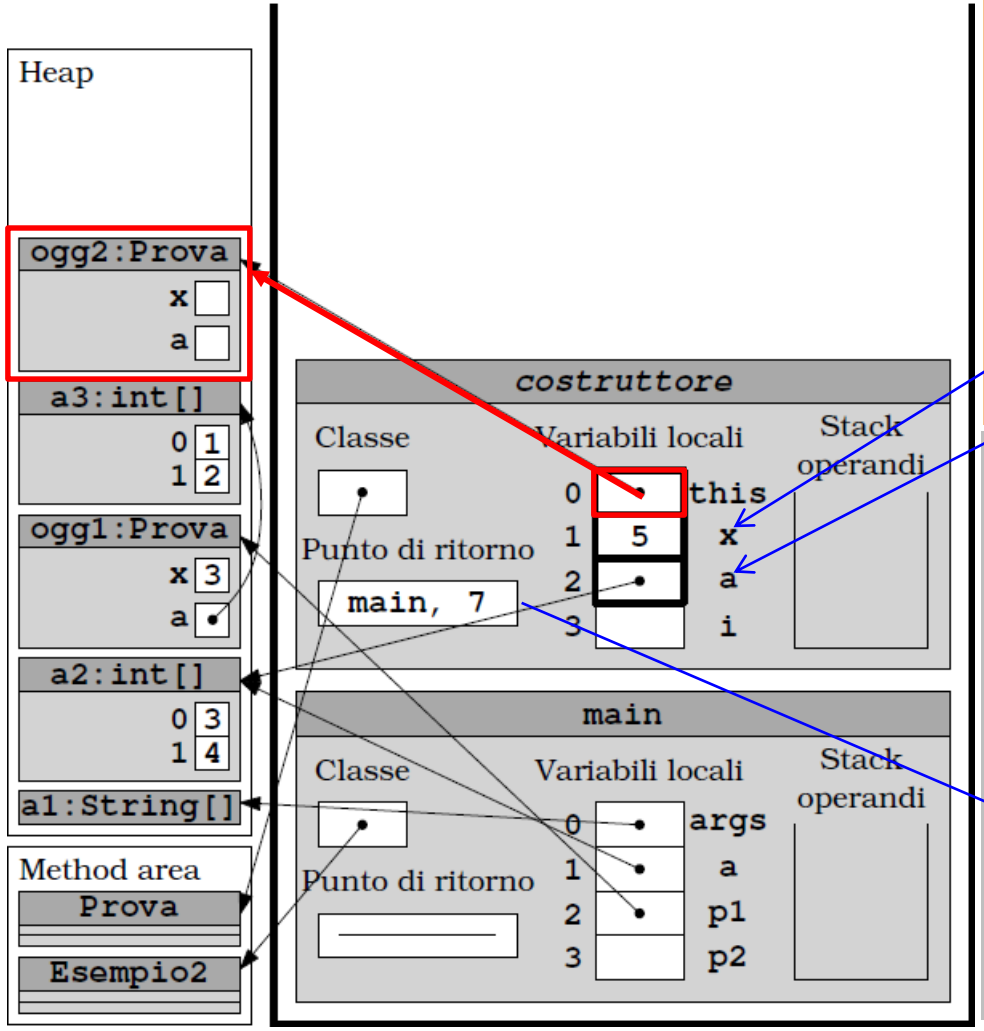

```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

Esempio




```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

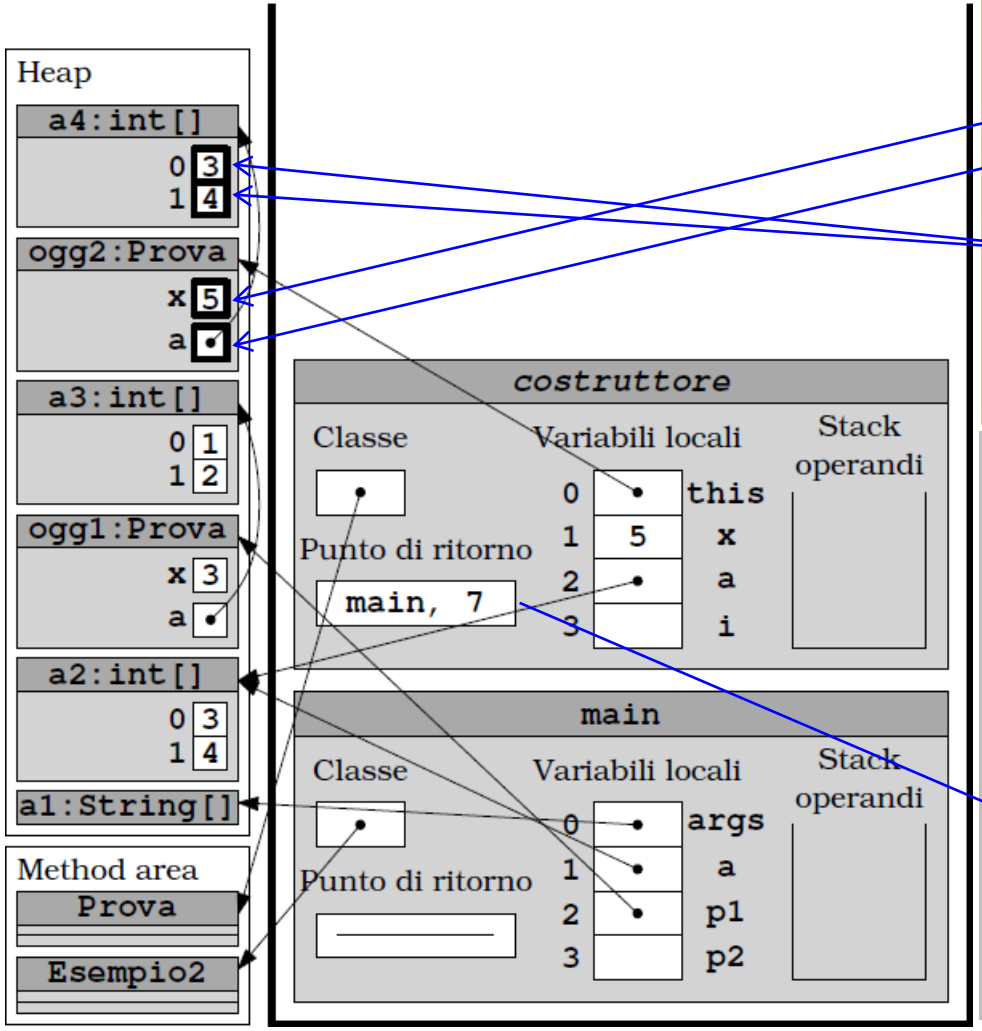
Esempio




```
public Prova(int x, int[] a){
  this.x = x;
  this.a = new int[a.length];
  for(int i=0;i<a.length;i++){
    this.a[i] = a[i];
  }
}
```

```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

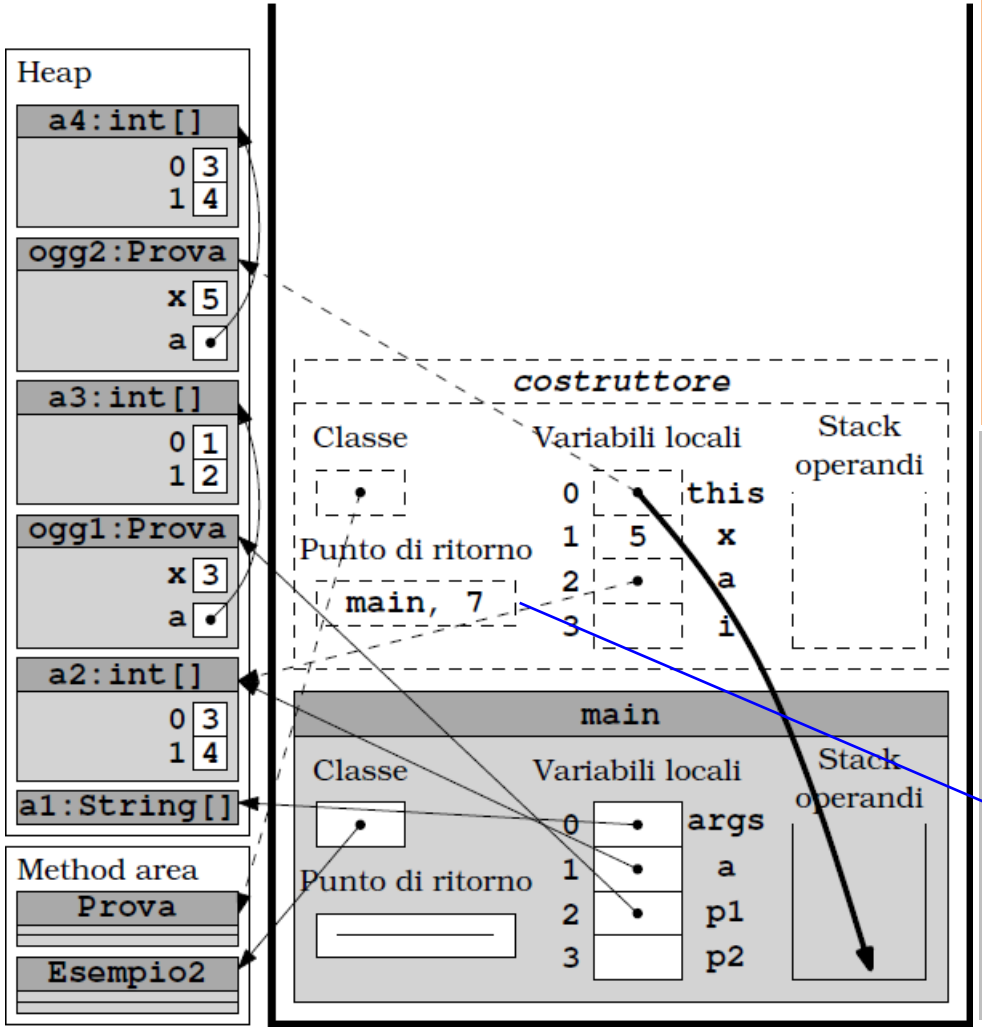
Esempio




```
public Prova(int x, int[] a){
   this.x = x;
   this.a = new int[a.length];
   for(int i=0;i<a.length;i++){
      this.a[i] = a[i];
   }
}
```

```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

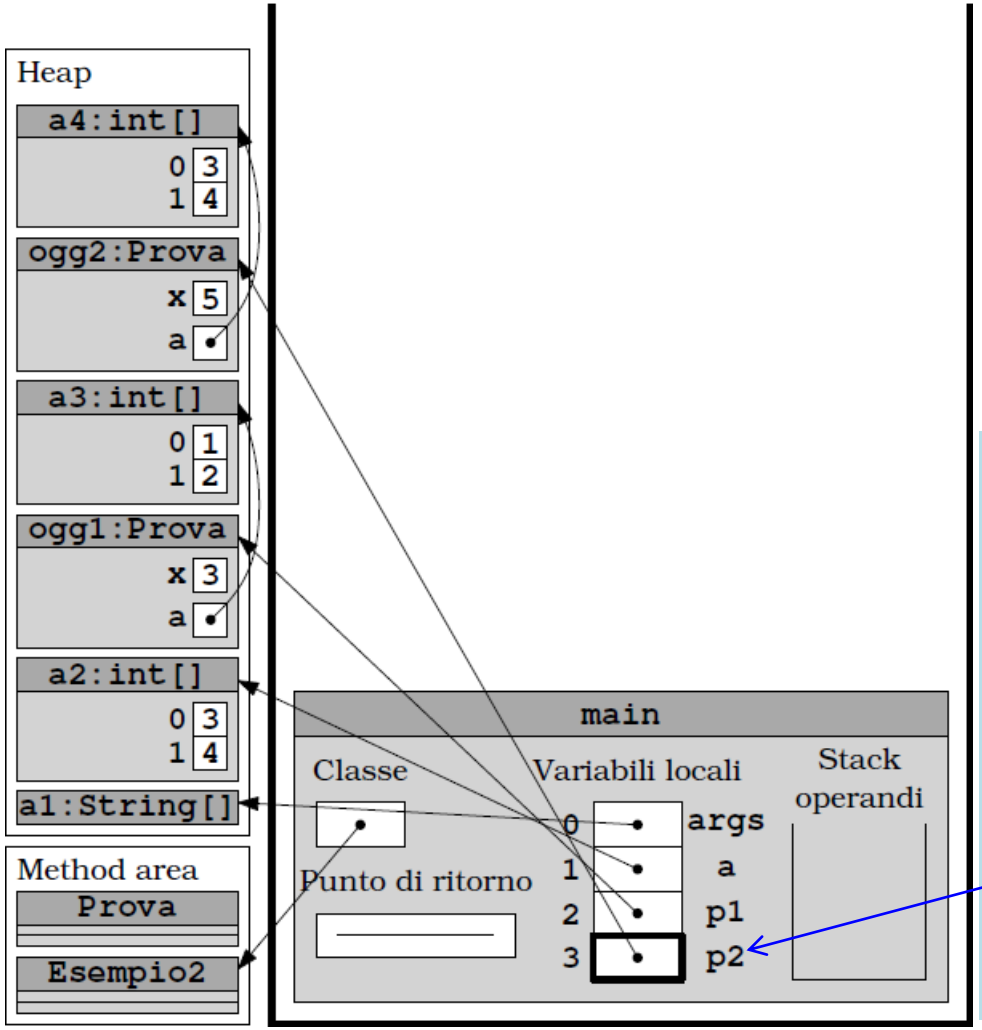
Esempio




```
public Prova(int x, int[] a){
  this.x = x;
  this.a = new int[a.length];
  for(int i=0;i<a.length;i++){
    this.a[i] = a[i];
  }
}
```

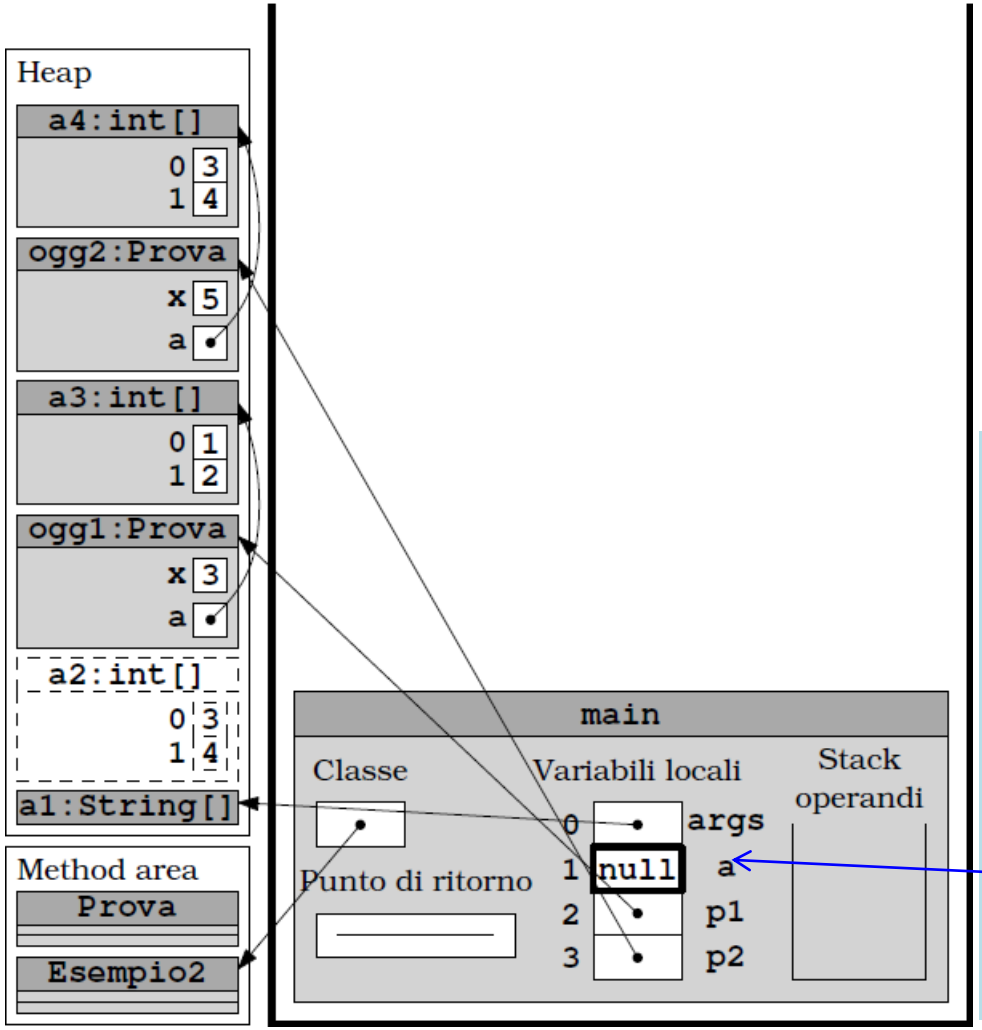
```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

Esempio



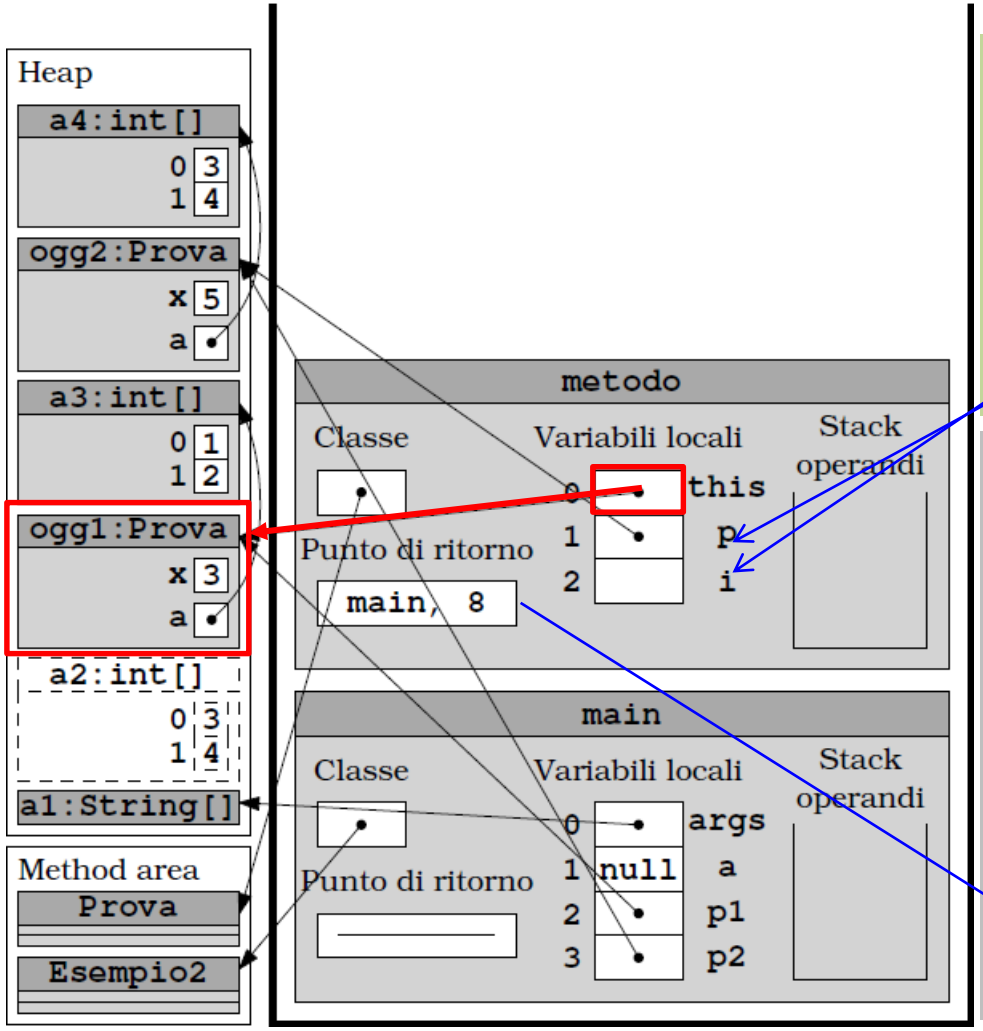

```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

Esempio




```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

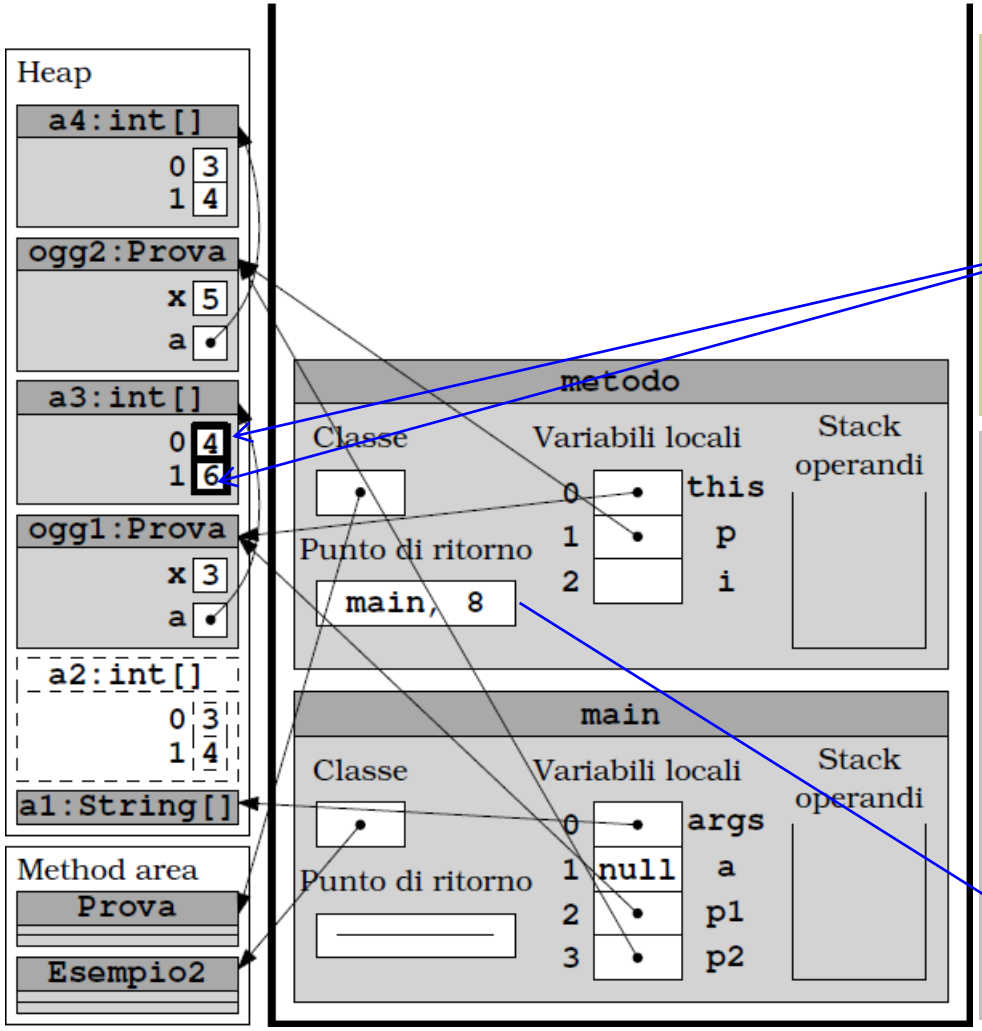
Esempio




```
public void metodo(Prova p){
  if (this.a.length==p.a.length){
    for (int i=0;i<a.length;i++){
      this.a[i] += p.a[i];
    }
  }
}
```

```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

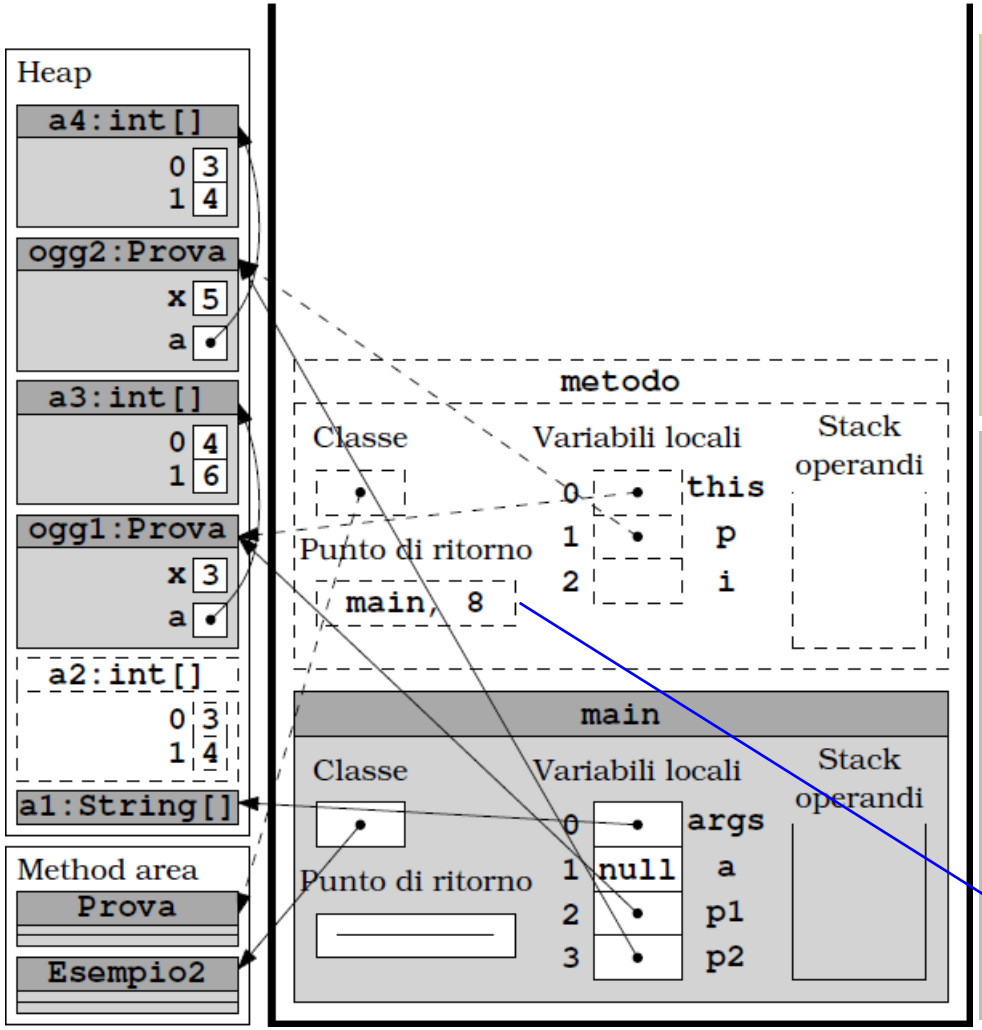
Esempio




```
public void metodo(Prova p){
  if (this.a.length==p.a.length){
    for (int i=0;i<a.length;i++){
      this.a[i] += p.a[i];
    }
  }
}
```

```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

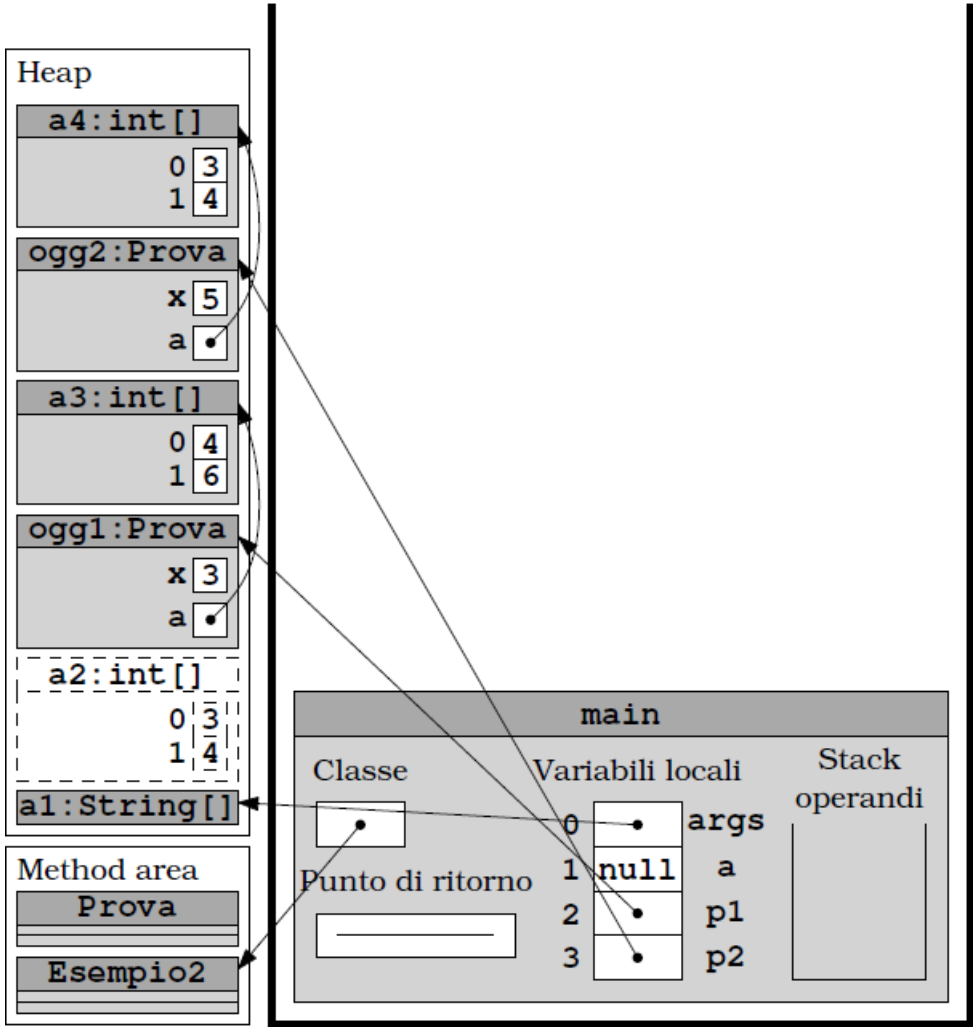
Esempio




```
public void metodo(Prova p){
  if (this.a.length==p.a.length){
    for (int i=0;i<a.length;i++){
      this.a[i] += p.a[i];
    }
  }
}
```

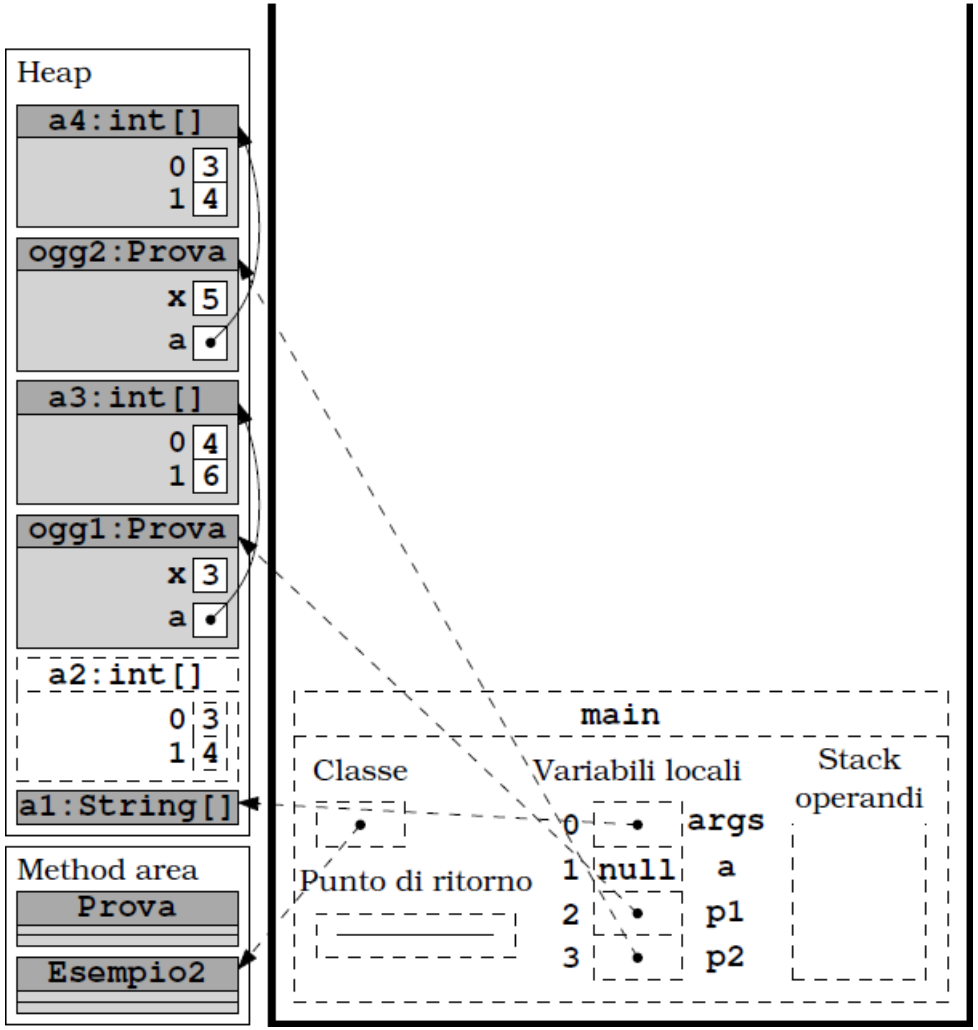
```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

Esempio

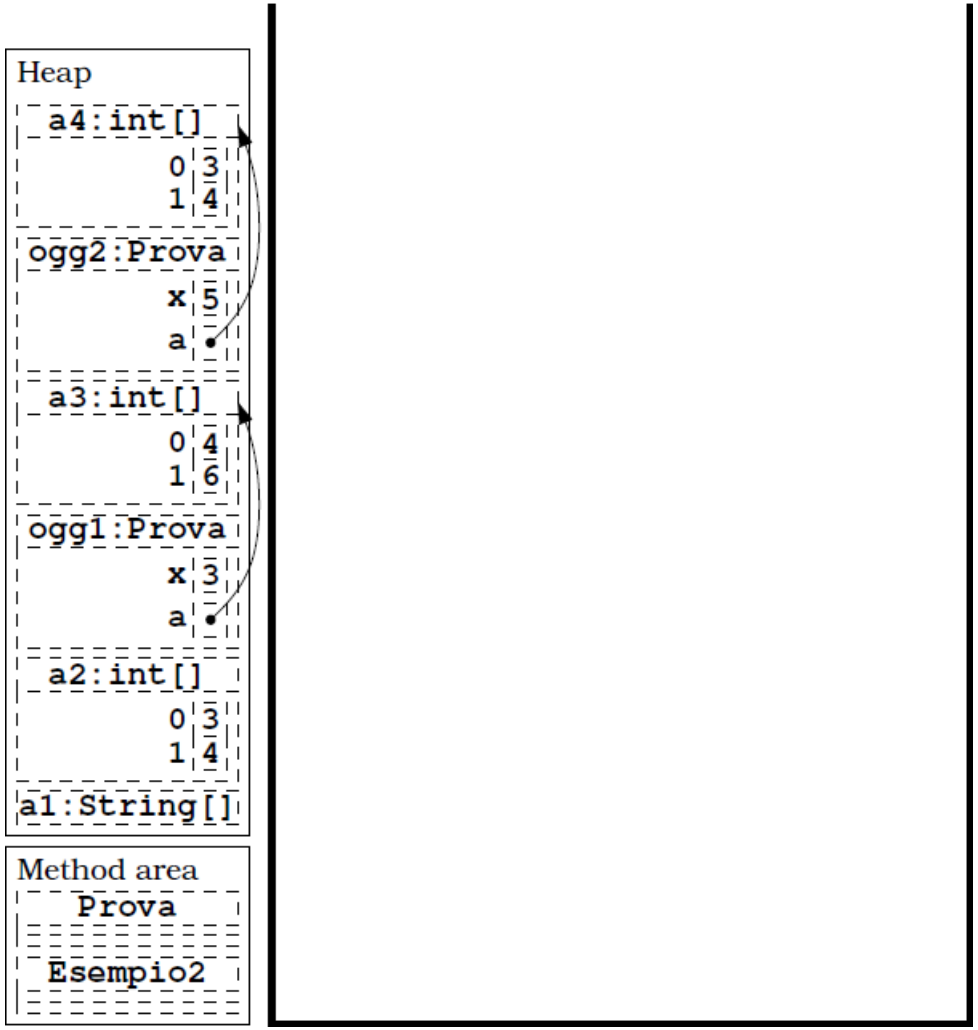



```
void main(String[] args){
  int[] a=new int[2];//main, 1
  a[0]=1; //main, 2
  a[1]=2; //main, 3
  Prova p1=new Prova(3,a);//main,4
  a[0]=3; //main, 5
  a[1]=4; //main, 6
  Prova p2=new Prova(5,a);//main,7
  a=null; //main, 8
  p1.metodo(p2); //main, 9
}
```

Esempio


Heap
a4:int[]
0 3
1 4
ogg2:Prova
x 5
a
a3:int[]
0 4
1 6
ogg1:Prova
x 3
a
a2:int[]
0 3
1 4
a1:String[]
Method area
Prova
Esempio2
main
Classe
Variabili locali
Stack operandi
Punto di ritorno
0
1
2
3
args
a
p1
p2
null

Esempio


Heap
a4:int[]
0 3
1 4
ogg2:Prova
x 5
a
a3:int[]
0 4
1 6
ogg1:Prova
x 3
a
a2:int[]
0 3
1 4
a1:String[]
Method area
Prova
Esempio2

Vita di variabili, classi e oggetti

- In base a quanto visto, i diversi tipi di variabili (di classe, di istanza, locali) vengono allocati in zone diverse della memoria (method area, heap, pila di attivazione) e hanno cicli di vita diversi
- Il ciclo di vita di un elemento (classe, oggetto o variabile) è il periodo di tempo in cui l'elemento risulta disponibile per essere utilizzato nell'ambito del programma

Variabili di classe

- Il ciclo di vita di una variabile di classe coincide con il ciclo di vita della classe cui appartiene
- Tale ciclo inizia quando la classe viene referenziata la prima volta
- Finisce quando il programma termina
- La variabile è infatti allocata all'interno dell'area di memoria riservata alla classe di cui fa parte

Variabili di istanza

- Una variabile di istanza è allocata nello spazio di memoria riservato all'oggetto a cui la variabile appartiene e quindi il suo ciclo di vita coincide con il ciclo di vita dell'oggetto in questione
- Tale ciclo inizia quando l'oggetto viene creato mediante l'operatore *new*
- Termina nel momento in cui l'oggetto non è più referenziato

Variabili locali

- Una variabile locale viene allocata nell'ambito del record di attivazione del metodo in cui la variabile è definita
- Il ciclo di vita di una variabile locale coincide quindi con il periodo in cui il corrispondente record di attivazione risulta allocato nella pila di attivazione
- Quando viene avviata l'esecuzione del metodo la variabile viene allocata e quando l'esecuzione termina la variabile viene deallocata

Commenti

- Capire come sono gestiti i diversi tipi di variabili a tempo di esecuzione permette di comprendere meglio alcune delle differenze che esistono tra di essi
- Ad esempio, una variabile di classe esiste in un'unica copia:
  - infatti essa è allocata all'interno dell'area di memoria associata alla classe
- Una variabile di istanza esiste invece in tante copie quanti sono gli oggetti creati:
  - per ogni oggetto viene creata una copia nello heap
  - le diverse copie sono indipendenti

Commenti

- Per una variabile locale esiste una copia per ogni attivazione
- Negli esempi visti, c'è al più una copia di ciascuna variabile locale in ogni istante
- Questo perché una volta che un metodo viene attivato, non ci sono nuove attivazioni prima della sua terminazione
  - Vedremo che, quando si usa la ricorsione, un metodo viene attivato più volte prima che le precedenti attivazioni terminino
  - Quindi più copie di una stessa variabile locale esistono contemporaneamente

Commenti

- Un’altra differenza tra i diversi tipi di variabili riguarda la loro visibilità o scope
- Le variabili di istanza e di classe sono visibili all’interno di tutti i metodi della classe cui appartengono
- Le variabili locali invece sono visibili solo all’interno del metodo in cui sono definite

Commenti

- Inoltre, una modifica fatta ad una variabile di istanza o di classe all'interno di un metodo permane anche dopo che il metodo è terminato
- le modifiche fatte alle variabili locali risultano invece perdute nel momento in cui il metodo termina

Commenti

- Il motivo è che le variabili di classe e di istanza sono allocate al di fuori dei record di attivazione dei metodi
  - il loro ciclo di vita continua anche dopo la terminazione di tali metodi
- Le variabili locali invece esistono solo nell'ambito di una esecuzione di un metodo
  - una volta che l'esecuzione del metodo termina le variabili locali non esistono più
- Successive attivazioni dello stesso metodo creeranno nuove copie della stessa variabile indipendenti dalle copie create in precedenza

Gestione della memoria in C

Gestione della memoria in C

- Molto di quanto detto riguardo la gestione della memoria in Java vale anche per il C
- In particolare anche in C esistono la pila di attivazione (o stack) e lo heap

La pila di attivazione in C

- Come in Java la pila di attivazione viene utilizzata per gestire le diverse attivazioni delle varie funzioni
- Ogni volta che invochiamo una funzione viene creato un record di attivazione che contiene varie informazioni, in particolare:
  - le variabili locali, inclusi i parametri formali
  - il punto di ritorno
- Il record creato viene messo in cima alla pila

La pila di attivazione in C

- Al termine della funzione il record viene rimosso e il controllo passa alla funzione associata al record sottostante, alla quale viene anche restituito il valore ritornato dalla funzione terminata

Lo heap in C

- Abbiamo visto che in Java lo heap viene utilizzato per memorizzare gli oggeti e le classi
- Poiché né le classi né gli oggetti esistono in C, a che cosa serve lo heap in C?

Lo heap in C

- Lo heap viene utilizzato per gestire l'allocazione dinamica della memoria
- Ogni volta che allochiamo memoria dinamicamente (usando la malloc o funzioni simili), la memoria richiesta viene allocata nello heap

Lo heap in Java e in C

- In effetti la gestione dinamica della memoria in C e la creazione di oggetti in Java sono molto simili tra loro
- In entrambi i casi:
  - la memoria viene allocata nello heap
  - si fa riferimento a tale memoria tramite un riferimento/puntatore
  - la memoria rimane allocata al termine dell'esecuzione del metodo/funzione in cui avviene l'allocazione

Alcune differenze

- La memoria allocata nello heap
  - in C deve essere esplicitamente rilasciata
  - in Java viene automaticamente recuperata dal garbage collector
- Array e oggetti/struct
  - in C possono essere allocati nello stack (come variabili locali) o nello heap (con la malloc)
  - in Java vengono sempre allocati nello heap
- Nota per il C: poiché lo stack ha dimensioni limitate, quando si devono gestire array o struct di grandi dimensioni è bene allocarli nello heap