Università di Roma "La Sapienza", Facoltà di Ingegneria

Corso di

Progettazione del Software

Anno Accademico 2005-2006

Corso di Laurea in Ingegneria Gestionale

Prof. Toni Mancini & Prof. Monica Scannapieco
tutor Diego Milano (milano@dis.uniroma1.it)

Parte J1: Nozioni Preliminari di Programmazione e Java

Riepilogo sugli Array

```
int[] mioArray; //dichiarazione

//allocazione della memoria e inizializzazione della variabile
mioArray=new int[10];

//assegnazione di un valore ad un elemento
mioArray[5]=23;

//accesso a componenti dell'array
int variabileIntera=mioArray[5];

//ciclo for sulle componenti:         //gli indici partono da zero
for(int i=0; i<mioArray.length;i++){  //si usa la variabile length
    System.out.println(mioArray[i]);
}
```

Inizializzatori e altri dettagli

```
int[] mioArray=new int[]{3,4,5}; //alloca automaticamente un array di
                                 //dimensione 3 e ne inizializza
                                 //gli elementi

//Nota: la seguente è un'inizializzazione valida
int[] tuoArray=new int[0];
```

Allocazione della memoria

Allocazione statica: viene *decisa* (la quantità di memoria necessaria) a tempo di *compilazione*. Viene effettuata prendendo memoria dall'area detta *stack*(pila).

Esempio: variabili locali in una funzione (vedremo altri esempi)

Allocazione dinamica: viene *decisa* a tempo di *esecuzione*. Viene effettuata prendendo memoria dall'area detta *heap*(catasta).

Esempio: creazione di un array tramite `new`.

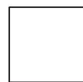
Evoluzione(run-time) della memoria

```
int variabileInt;
int[] variabileArray;
variabileInt=5;
variabileArray=new int[4];
variabileArray[2]=6;
```

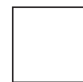
Notare che sono scatole vuote. Un tentativo di accedere al loro valore (prima di un assegnazione) viene rilevato a tempo di compilazione (variable might not have been initialized)

heap

stack



variabileInt



variabileArray

Evoluzione (run-time) della memoria

```
int variabileInt;
int[] variabileArray;
variabileInt=5;
variabileArray=new int[4];
variabileArray[2]=6;
```

heap

- -

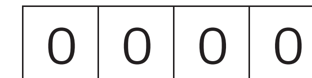
stack

5	
variabileInt	variabileArray

Evoluzione (run-time) della memoria

```
int variabileInt;
int[] variabileArray;
variabileInt=5;
variabileArray=new int[4];
variabileArray[2]=6;
```

La memoria allocata dinamicamente è inizializzata ad un valore di default



heap

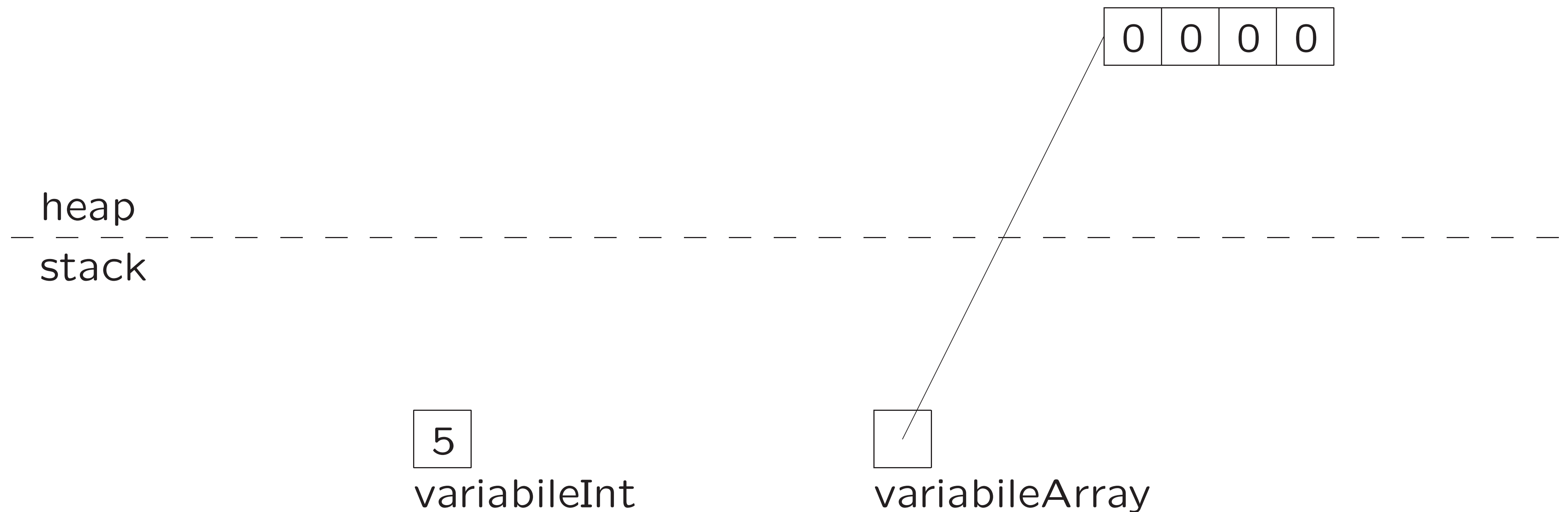
stack

5
variabileInt

variabileArray

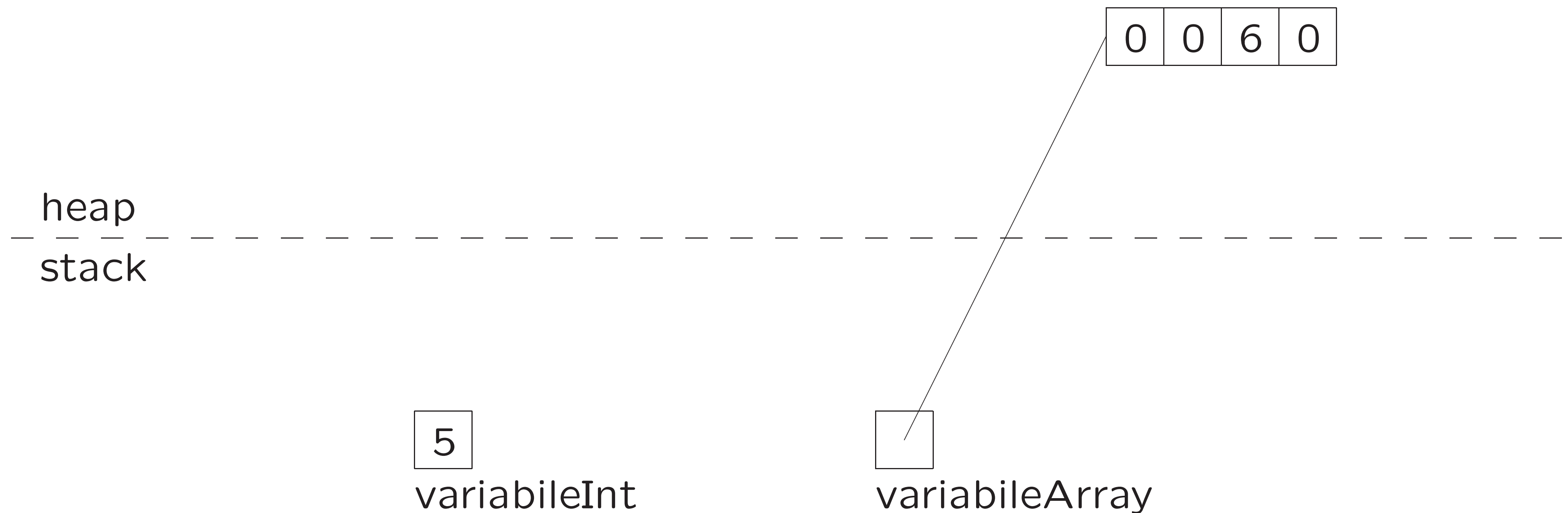
Evoluzione (run-time) della memoria

```
int variabileInt;
int[] variabileArray;
variabileInt=5;
variabileArray=new int[4];
variabileArray[2]=6;
```



Evoluzione (run-time) della memoria

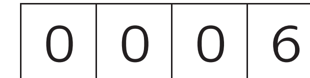
```
int variabileInt;
int[] variabileArray;
variabileInt=5;
variabileArray=new int[4];
variabileArray[2]=6;
```



Osservazione

Notare che new int[4]; è un espressione che restituisce un valore di tipo riferimento, e può essere impiegata come tale ovunque(compatibilmente con i vincoli sui tipi). Se presa da sola, è sintatticamente scorretta (not a statement), ma (new int[4])[3]=6; è un'istruzione valida.

Morale: allocare memoria sullo heap ed assegnarne il riferimento sono operazioni logicamente distinte.



heap

\- -

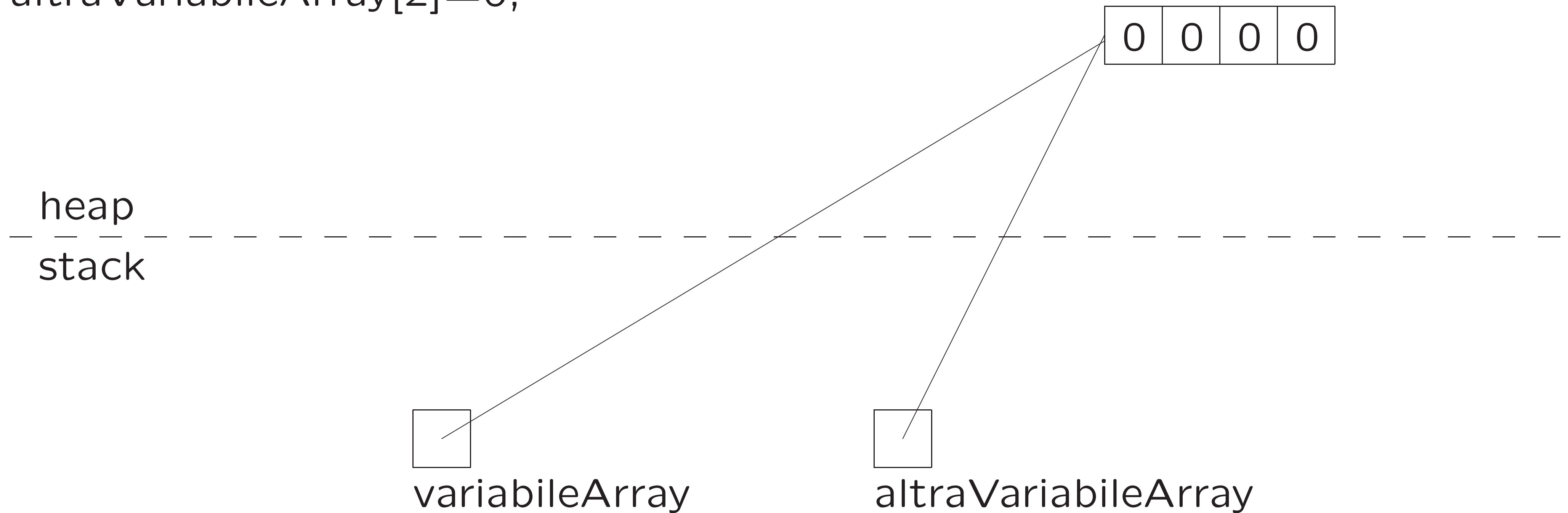
stack

Altra osservazione

```
int varInt=5;
int[] varArr=new int[varInt];
//la dimensione della memoria allocata dinamicamente
//è decisa a tempo di esecuzione
//quindi non è necessario che sia specificata tramite
//una costante!
```

Assegnazione fra variabili per riferimento

```
int[] variabileArray;
int[] altraVariabileArray;
variabileArray=new int[4];
altraVariabileArray=variabileArray;
altraVariabileArray[2]=6;
```



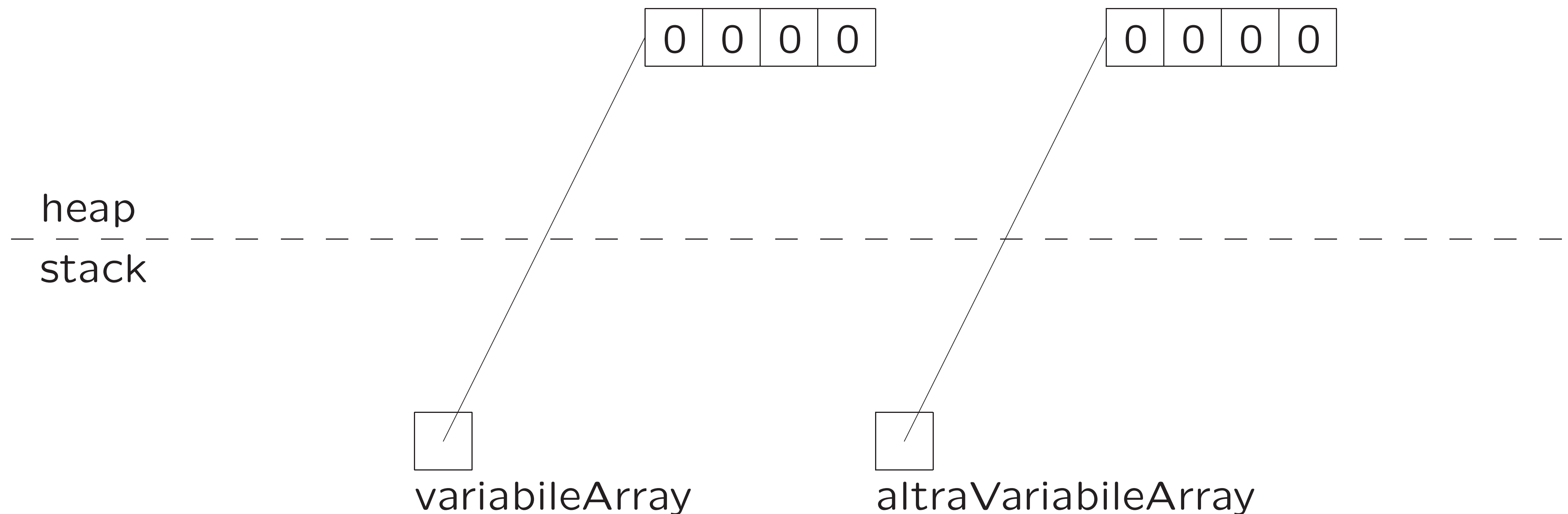
Quindi attenzione!

```
int varInt, varInt2;
varInt=5;
varInt2=varInt;
varInt2=10;
System.out.println(varInt); //stampa 5

int[] varArr, varArr2;
varArr=new int[4];
varArr[3]=5;
varArr2=varArr;
varArr2[3]=10;
System.out.println(varArr[3]); //stampa 10!
```

Che succede alla memoria dinamica...

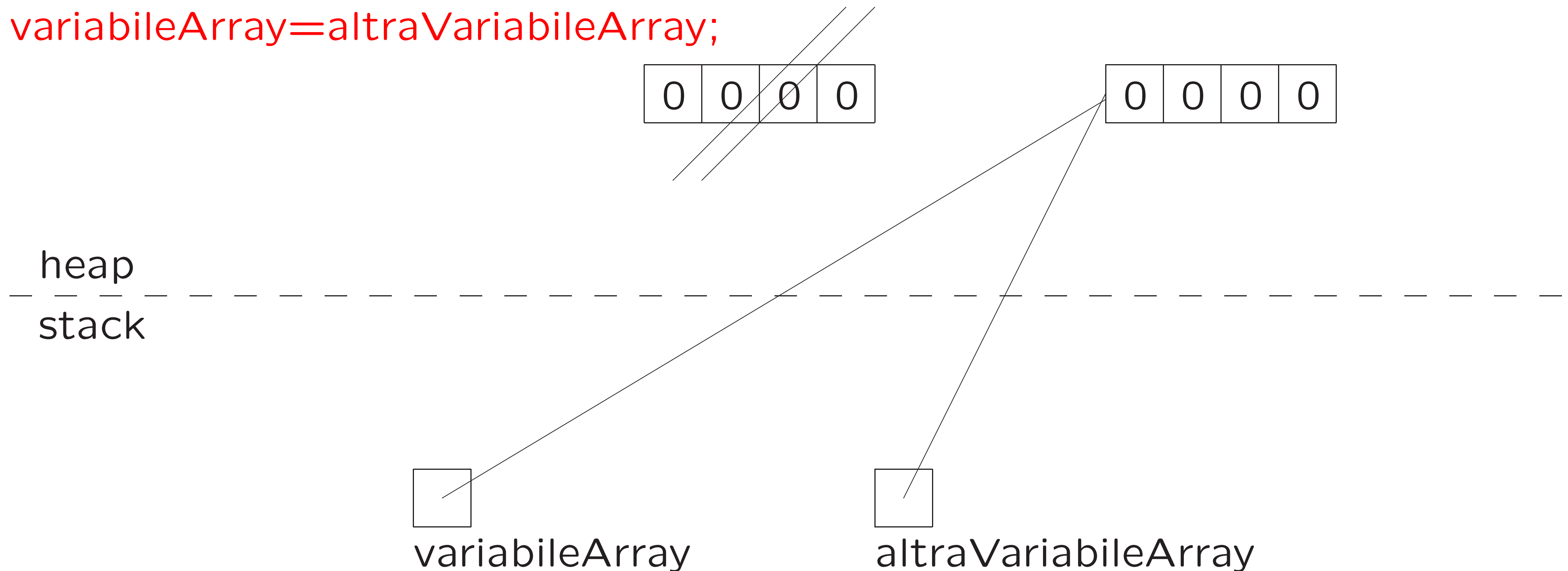
```
int[] variabileArray;
int[] altraVariabileArray;
variabileArray=new int[4];
altraVariabileArray=new int[4];
variabileArray=altraVariabileArray;
```



...se nessuno la punta?

```
int[] variabileArray;
int[] altraVariabileArray;
variabileArray=new int[4];
altraVariabileArray=new int[4];
variabileArray=altraVariabileArray;
```

Il Garbage Collector se ne accorge!



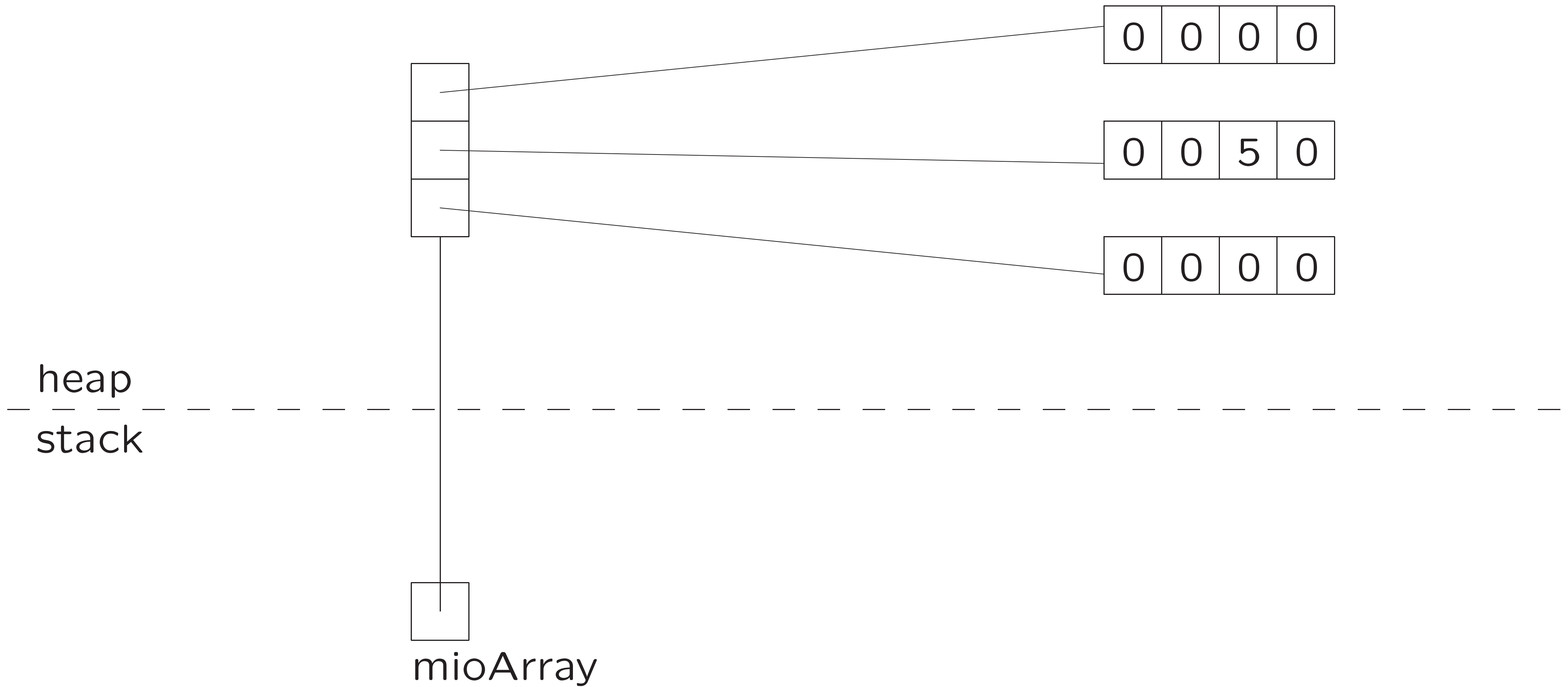
Array di array

- Gli elementi di un array sono normali variabili. Possono avere un tipo qualsiasi (lo stesso per tutte). In particolare possono essere a loro volta di un tipo per riferimento.

```
int[][] mioArray; //dichiara un array di array di int
mioArray=new int[3][4];
mioArray[1][2]=5;   //Notare che mioArray[1] è un array di int.
                     //quindi: int[] tuoArray=mioArray[1];

for(int i=0; i<mioArray.length;i++){ //length di mioArray
    for(int j=0; j<mioArray[i].length;j++){ //length dell'i-esimo array
        System.out.println(mioArray[i][j]); //contenuto in mioArray
    }
}
```

In memoria



E la memoria degli elementi?

- In effetti non è necessario allocarla direttamente.

```
int[][] mioArray;
mioArray=new int[5][]; //alloca memoria solo per mioArray

//attenzione
//mioArray[3][5]=11;   E' sbagliato, la memoria per l'array mioArray[3]
//                     non è ancora stata allocata!

//Ora allochiamo la memoria per gli elementi
for(int i=0;i<mioArray.length;i++){
    mioArray[i]=new int[(i+1)*2];   //l'array finale contiene
}                                   //array di lunghezza 2, 4, 6, 8, 10
```

Inizializzatori

```
int[][] mioArray;
mioArray=new int[][]{   //di fatto e' un elenco di
{1,2},                  //inizializzatori lineari...
{3,4,5},
{6,7,8,9}
};          //notare il ;
```

Metodi in Java

- esiste un solo tipo di *unità di programma (eseguibile)*: il **metodo** (funzione)

- ogni metodo appartiene ad (è *incapsulata* in) **una classe**

- abbiamo visto il metodo `static void main(String Args[])`

- vediamo come si definiscono altri metodi

Metodi: Dichiarazione

```
static int somma(int o1, int o2){

//Notare:
//stiamo ancora usando la keyword static
//tipo di ritorno
//parametri formali

    int s;    //variabile locale al metodo
    s=o1+o2;

    return s; //restituzione di un valore
}
```

Il tipo di ritorno void

- denota un metodo che non restituisce alcun valore

- non è necessario (ma è consentito) che il metodo contenga la keyword *return*

- implementa uno stile di tipo procedurale

```
static void stampa(int[] array){
    for(int i=0;i<array.length;i++){
        System.out.println(array[i]);
    }
}
```

Metodi Sovraccarichi

- E' possibile dichiarare più metodi con lo stesso nome (Overloading)
- Devono distinguersi per numero e/o tipo degli argomenti
- Non basta (e non serve) che abbiano diverso tipo d ritorno

```
//overloading
static int[] sottoArray(int indiceInizio){...}
static int[] sottoArray(int indiceInizio, int indiceFine){...}
//errore
static int prodotto(int a,int b){...}
static long prodotto(int c, int d){...}
```

Metodi: Invocazione

```
int i=5;
i=somma(i, 4); //le espressioni i e 4 si chiamano "parametri attuali"
               //all'atto dell'invocazione viene effettuata l'assegnazione
               //dei parametri attuali a quelli formali o1 e o2
               //i parametri formali agiscono a tutti gli effetti come
               //variabili locali inizializzabili all'atto della chiamata.
```

Modello run-time dell'invocazione di funzioni

- Quando un metodo viene invocato, viene allocato **nello stack** un *record di attivazione*, che contiene le informazioni indispensabili per l'esecuzione.

- Fra queste informazioni, ci sono (anche);
  - le variabili locali del metodo
  - i parametri formali (che si comportano come variabili locali)

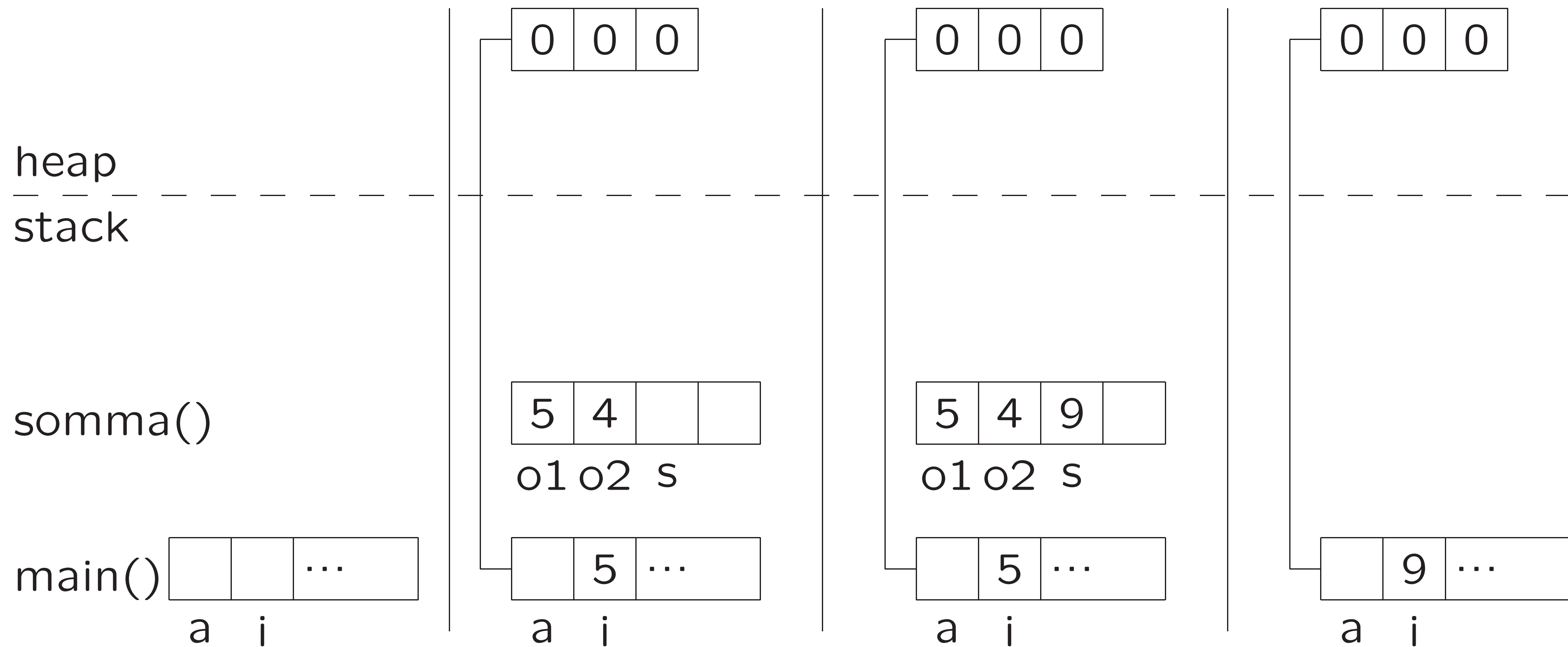
- Al termine dell'esecuzione della funzione, il record di attivazione viene deallocato.

Esempio

```
class Esempio{
    public static void main (String Args[]){
        int[] a=new int[3];
        int i=5;
        i=somma(i,4);
    }

    public static int somma(int o1, int o2){
        int s;
        s=o1+o2;
        return s;
    }
}
```

Evoluzione (run-time) dello stato della memoria



Comunicazione fra unità di programma

- Il passaggio di parametri ad una funzione è **solamente per valore**.

- Ciò significa che:

1. il **parametro attuale** può essere un'espressione qualsiasi (costante, variabile, espressione non atomica);

2. viene effettuata una **copia** del valore del parametro attuale nella locazione di memoria corrispondente al **parametro formale** che si trova nel record di attivazione della funzione chiamata;

3. tale locazione viene ovviamente perduta al termine dell'esecuzione della funzione, quando il record di attivazione corrispondente viene deallocato.

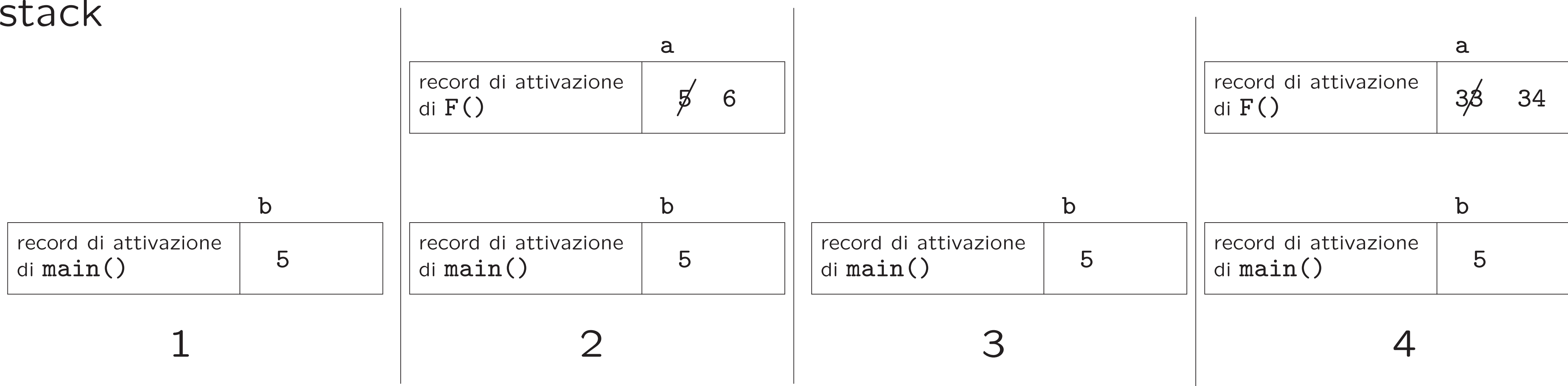
Comunicazione fra unità di programma (cont.)

Esempio: argomento di un **tipo base**

```
public static void main(String[] args) {
  int b = 5;                          // 1
  F(b);                               // 2 --  b e’ il PARAMETRO ATTUALE
  System.out.println("b: " + b);      // 3
  F(33);                              // 4 -- 33 e’ il PARAMETRO ATTUALE
}

public static void F(int a) {
  a++;                               //modifica il parametro formale
  System.out.println("a: " + a);
}
```

Evoluzione (run-time) dello stato della memoria



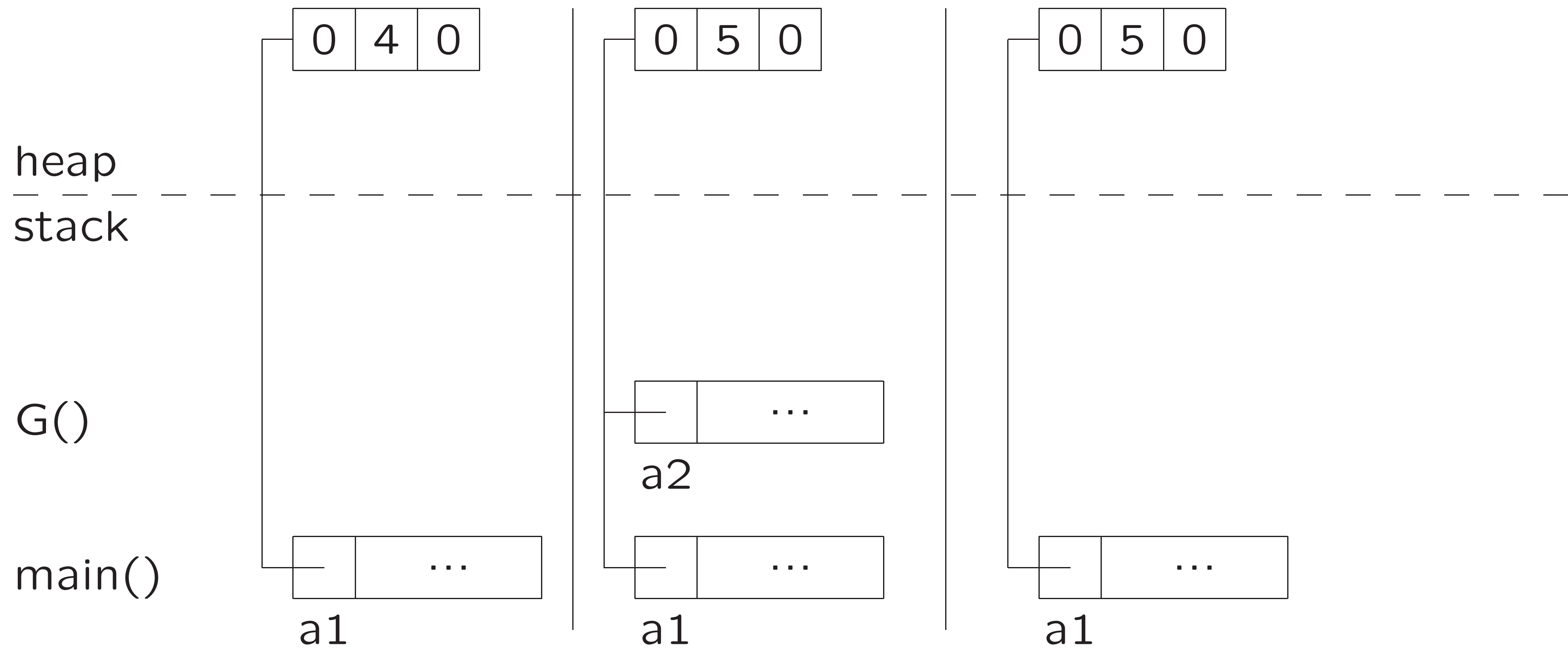
Comunicazione fra unità di programma (cont.)

Esempio: argomento di un tipo per riferimento (es. array)

```
public static void main(String[] args) {
  int[] a1 = new int[3];
  a1[1] = 4
  G(a1);
  System.out.println(a[1]);
}

public static void G(int[] a2) {
  a2[1]++;                        // SIDE-EFFECT
}
```

Evoluzione (run-time) dello stato della memoria



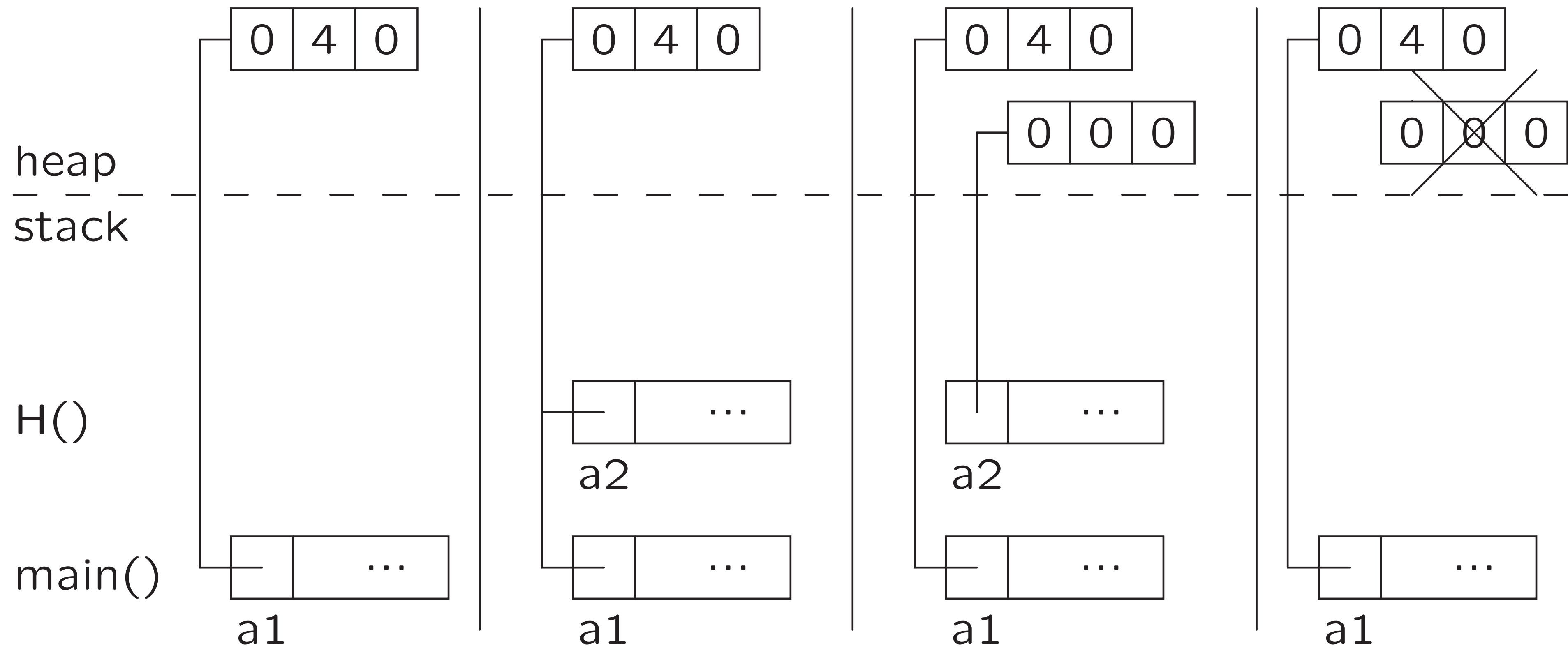
L’array cambia, il riferimento no!

Esempio: argomento di un tipo per riferimento (es. array)

```
public static void main(String[] args) {
  int[] a1 = new int[3];
  a1[1] = 4
  H(a1);
  System.out.println(a[1]);
}

public static void H(int[] a2) {
  a2=new int[3];
}
```

Evoluzione (run-time) dello stato della memoria



Restituzione da parte di una funzione

Anche la restituzione da parte di una funzione è **solamente per valore**. Pertanto, tutte le considerazioni sul passaggio di argomenti valgono anche per la restituzione:

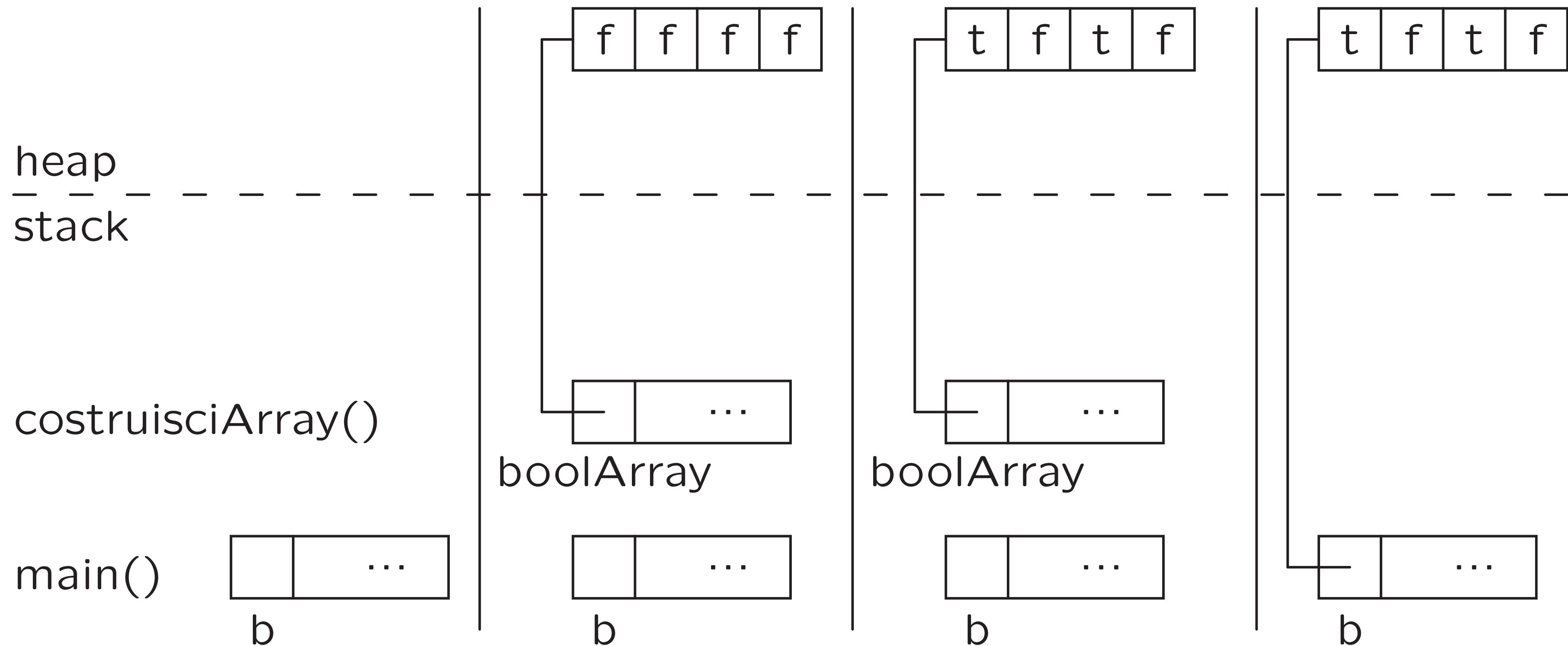
- se il tipo restituito è un tipo base, viene fatta una copia
- se il tipo restituito è una classe, viene fatta una copia del riferimento, **ma non della memoria dinamica a cui esso punta**

Nota: Restituzione di riferimenti

```
public static void main(String[] args) {
  boolean[] b;
  b=costruisciArray();
  for(int i=0;i<b.length;i++){
      System.out.println(b[i]);
  }
}

public static boolean[] costruisciArray() {
  boolean[] boolArray=new boolean[4]; //tutti inizializzati a false!
  for(int i=0;i<boolArray.length;i=i+2){
      boolArray[i]=true;
  }
}
```

Evoluzione (run-time) dello stato della memoria



Visibilità delle variabili locali ad un metodo

- I parametri formali e le variabii locali esistono localmente ad un record di attivazione. Quindi:
  - Non sono visibili (non esistono) all'esterno del metodo
  - Possono avere lo stesso nome di variabili definite in un altro metodo
  - vedremo in seguito altre regole di visibilità...

Esempio

```
public static void main(String[] args) {
    int a=5;
    int b=7;
    func(a+b);
}

public static void func(int a){
    System.out.println(a)   //QUESTA a è il parametro formale, e vale 12
    int b;
    //System.out.println(b) //errore! QUESTA b non è stata inizializzata
    int b=a;
}
```

Invocazioni Annidate

```
public static void main(String[] args) {
    int[] a1=new int[]{1,2,3,4,5};
    stampaArray(a1);
}

public static void stampaArray(int[] b) {
    for(int i=0;i<b.length;b++){
        stampaValore(i,b[i]);
    }
}

public static void stampaValore(int i, int v) {
    System.out.println("il campo dell'array di indice "+i+" vale "+ v);
}
```

Funzioni ricorsive

```
public static void main(String[] args) {
    int[] a1=new int[]{1,2,3};
    stampaArrayRic(0,a1);
}

public static void stampaArrayRic(int indice,int[] array) {
    if(indice<array.length){
        System.out.println(array[indice]);
        stampaArrayRic(indice+1,array);
        System.out.println("io ho finito!");
    }
    else{
        System.out.println("Aargh, non posso stampare!!");
    }
}
```

Evoluzione (run-time) dello stato della memoria