Progettazione del Software

Programmazione in Java (1)

Domenico Fabio Savo

Dipartimento di Ingegneria Informatica, Automatica e Gestionale Antonio Ruberti

Sapienza Università di Roma

Le slide di questo corso sono il frutto di una rielaborazione di analogo materiale redatto da Marco Cadoli, Giuseppe De Giacomo, Maurizio Lenzerini e Domenico Lembo

Il linguaggio Java

Linguaggio Object Oriented recente (1996). Precursori: Smalltalk (fine anni ’70), C++ (inizio anni ’80). Sintatticamente simile al C++, ma con alcune importanti differenze. C# e Visual Basic .NET si ispirano ad esso in molti aspetti

Ha avuto successo perchè:

- librerie standard (in particolare, per Internet) native
- portabilità su varie piattaforme

Generalità su Java

Una caratteristica molto importante di Java quella è di essere indipendente dal sistema operativo in cui un programma viene compilato. Infatti, il compilatore Java produce un codice che non è il codice binario finale, ma un codice binario intermedio chiamato **bytecode** che è indipendente dal sistema operativo.

Una “macchina virtuale” chiamata **Java Virtual Machine** (JVM) si occupa di interpretare il **bytecode** e lo traduce in istruzioni macchina.

JVM per tutti i sistemi operativi sono distribuite gratuitamente.

Programmare il Java - di cosa abbiamo bisogno?

Lo strumento principale per poter sviluppare applicazioni in Java è il Java Developers Kit (JDK) che può essere scaricato dal sito www.java.com

JDK contiene tutto ciò che serve per poter creare sviluppare ed eseguire applicazioni in Java. In particolare:

- Il compilatore (javac) il cui compito è quello di trasformare il codice sorgente Java nel bytecode che sarà poi eseguito dalla macchine virtuale java (JVM)
- Diverse **librerie** standard – Java Core Appication Programming Interface (API)
- La macchina virtuale (JVM) che potrà essere lanciata attraverso il comando java

Compilare e lanciare un programma Java

Per scrivere un programma Java dobbiamo creare un file di testo contenente del codice Java. Tale file dovrà avere estensione **.java**.

Il file .java viene quindi passato al compilatore Java (programma chiamato javac).

javac xxx.java

Il compilatore Java restituirà un file **.class** con lo stesso nome del file .java dato in input. Il file **.class** contiene il bytecode per la JVM

Per eseguire il file **xxx.class** prodotto dal compilatore invochiamo la JVM nel seguente modo:

java xxx

Compilare e lanciare un programma Java

Editor ASCII → Codice sorgente → Compilatore javac → File bytecode (.class) → Interprete java → Programma in esecuzione

- Per lo sviluppo di applicazioni complesse in Java può tornare molto utile l'uso di un IDE (*Integrated Development Environment*), ovvero un ambiente di sviluppo che integra tutti i tool in un unico ambiente visuale.

- Alcuni esempi di IDE sono: Eclipse, NetBeans, IntelliJ, JCreator,...

Tipi di dato in Java

- Dobbiamo distinguere nettamente fra:

  1. variabili i cui valori sono di *tipi di dato di base (o primitivi)*, cioè `int`, `char`, `float`, `double`, `boolean`, e

  2. *oggetti*, cioè istanze delle *classi*.

- In particolare, la memoria per la loro rappresentazione viene, rispettivamente:

  1. *allocata* automaticamente, senza necessità di una esplicita richiesta mediante istruzione durante l'esecuzione del programma, ovvero

  2. *allocata* durante l'esecuzione del programma, a fronte della esecuzione di una opportuna istruzione (cioé con `new`).

Riferimenti e oggetti

Dobbiamo inoltre distinguere nettamente fra:

- *riferimenti* a oggetti, e
- oggetti veri e propri.

I primi sono di fatto assimilabili ai tipi di dato di base. Infatti, come questi, la memoria per la loro rappresentazione viene allocata automaticamente, senza necessità di una esplicita richiesta mediante istruzione durante l'esecuzione del programma

Un riferimento è un indirizzo di memoria

Inizializzazioni implicite per i campi delle classi

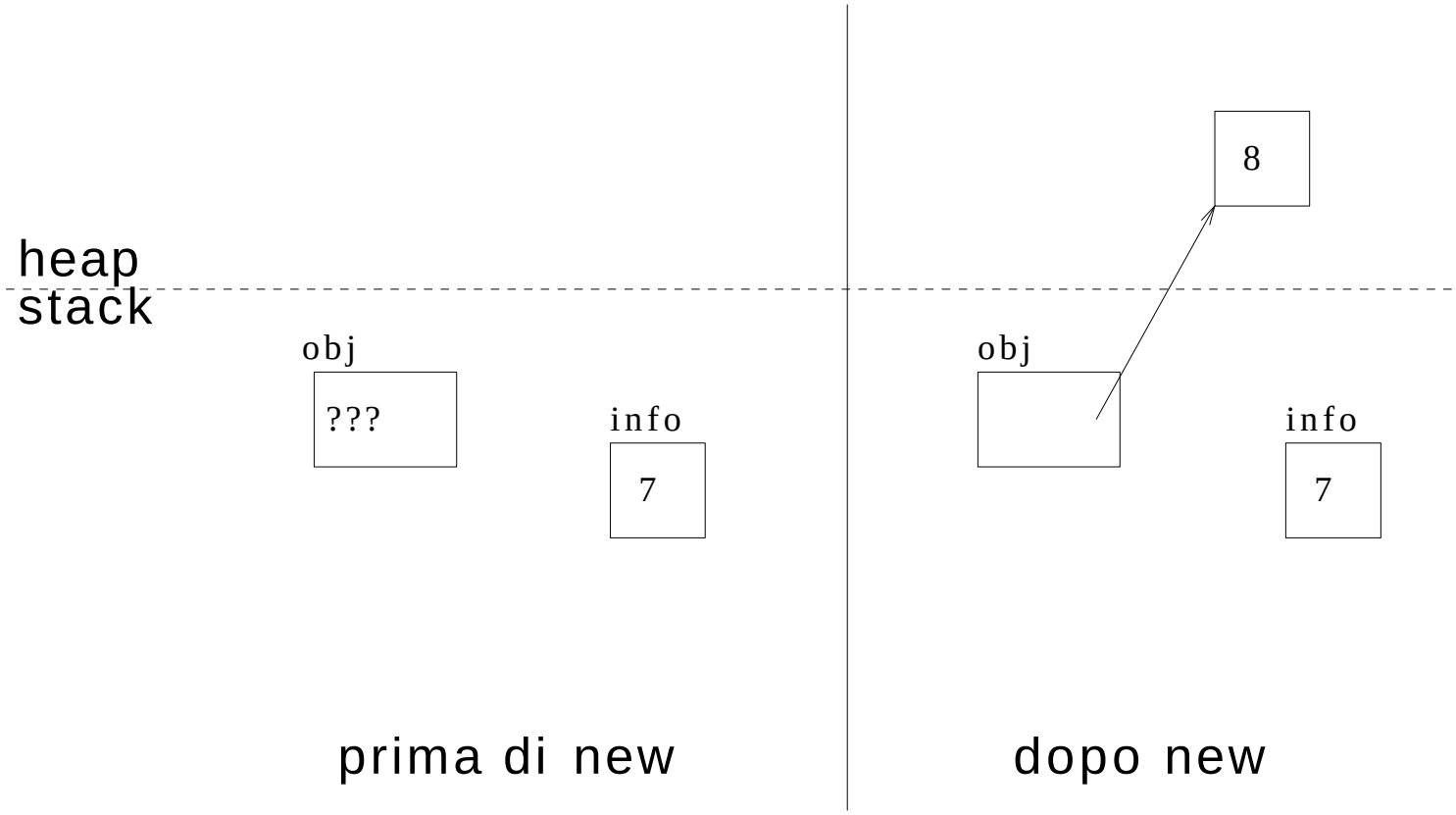
Un campo di tipo	Viene inizializzato implicitamente a	Note
int	0	
float, double	0.0	
char	'\0'	"null char"
boolean	false	
class C	null	il riferimento, non l'oggetto

Queste inizializzazioni avvengono automaticamente:

- per i campi dati di una classe;
- ma non per le variabili locali delle funzioni.

```
// File Esempio0.java
// Evidenzia la differenza fra valore di tipo base e oggetto

public class Esempio0 {
  public static void main(String[] args) {

    int info = 7; // dichiarazione di una variabile locale di tipo base;
                  // la memoria viene allocata SENZA esplicita richiesta
                  // mediante istruzione durante l'esecuzione del programma

    Integer obj; // Integer e' una classe: dichiarazione di un riferimento
                 // la memoria DEL SOLO RIFERIMENTO viene allocata SENZA esplicita
                 // richiesta mediante istruzione durante l'esecuzione del programma
                 // Adesempio:
                 //     System.out.println(obj);
                 //                        ^
                 // Variable obj may not have been initialized.

    obj = new Integer(8); // la memoria DELL'OGGETTO viene allocata durante l'esecuzione
                          // del programma mediante l'esecuzione dell'istruzione new
    System.out.println(obj);
  }
}
```

Evoluzione (run-time) dello stato della memoria



Allocazione della memoria

Allocazione statica: viene *decisa* a tempo di *compilazione*.

Viene effettuata prendendo memoria dall'area detta ***stack***.

Esempi: variabile locale in una funzione, campo dati `static` di una classe, . . .

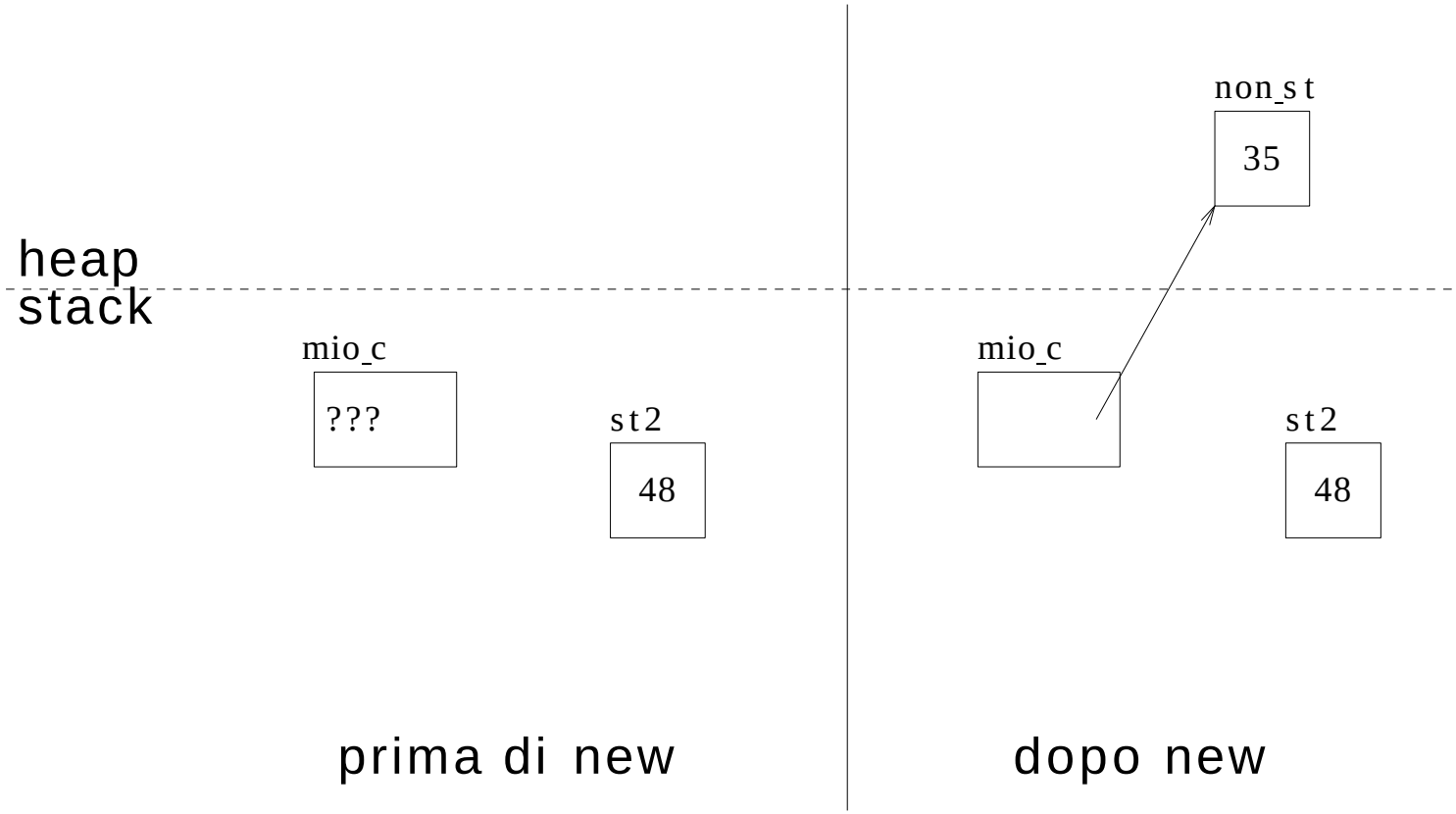
Allocazione dinamica: viene *decisa* a tempo di *esecuzione*.

Viene effettuata prendendo memoria dall'area detta ***heap***.

Esempio: creazione di un oggetto tramite `new`.

```
// File Esempio1.java
// Evidenzia la differenza fra allocazione statica e dinamica

class C {
  int non_st;
}

public class Esempio1 {
  public static void main(String[] args) {
    int st2;      // allocazione STATICA
    st2 = 48;     // OK: la memoria è stata già allocata
    C mio_c;      // allocazione STATICA (del solo riferimento)
    // mio_c.non_st=35;     // NO: la memoria non è stata ancora allocata
    mio_c = new C();        // allocazione DINAMICA dell'oggetto
    mio_c.non_st=35;        // OK: la memoria è stata già allocata
  }
}
```

Evoluzione (run-time) dello stato della memoria



Campi dati static

- In una classe C un qualsiasi campo dati s può essere dichiarato static.

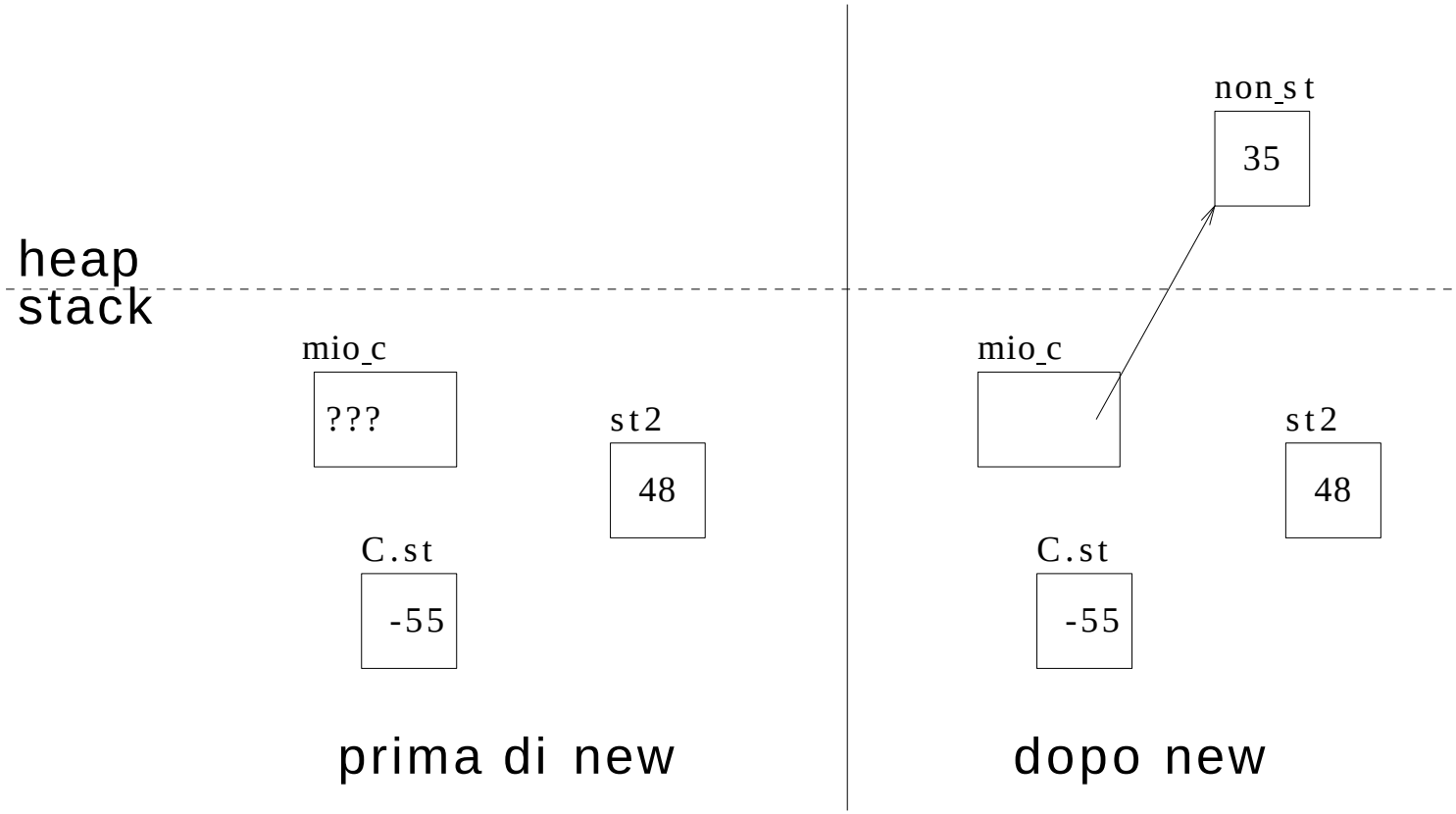
- Dichiarare s come static significa che s è un campo **relativo alla classe** C, non ai singoli oggetti di C.

- Pertanto, per un tale campo s esiste **una sola locazione di memoria**, che viene allocata **prima** che venga allocato qualsiasi oggetto della classe C.

- Viceversa, per ogni campo non static esiste una locazione di memoria **per ogni oggetto**, che viene allocata contestualmente a new.

```
// File Esempio1bis.java
// Evidenzia la differenza fra allocazione statica e dinamica

class C {
  static int st; // <-- NB: allocazione STATICA
  int non_st;
}

public class Esempio1 {
  public static void main(String[] args) {
    int st2;      // allocazione STATICA
    st2 = 48;     // OK: la memoria è stata già allocata
    C.st = -55;   // <--NB: OK: la memoria è stata già allocata
    C mio_c;      // allocazione STATICA (del solo riferimento)
    // mio_c.non_st=35;     // NO: la memoria non è stata ancora allocata
    mio_c = new C();        // allocazione DINAMICA  dell'oggetto
    mio_c.non_st=35;        // OK: la memoria è stata già allocata
  }
}
```

Evoluzione (run-time) dello stato della memoria



Funzioni in Java

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

- ogni funzione appartiene ad (*incapsulata* in) **una classe**

- esiste un'unità di programma principale (**main()**)

- le funzioni si distinguono in:
  - **static**: sono relative *alla classe*
  - **non static**: sono relative *agli oggetti della classe*

```
// File Esempio2.java
// Evidenzia la differenza fra funzioni statiche e non class C      {
  int x;
  void F() { System.out.println(x); }
  static void G() { System.out.println("Funzione G()"); }
  // static void H() { System.out.println(x); }
  //                                          ^
  // Can't make a static reference to nonstatic variable x in class C.
}

public class Esempio2 {
  public staticvoid  main(String[] args) {
     C c1 = new C();
     c1.x = -4;
     c1.F(); // invocazione di funzione NON STATIC
     C.G();  // invocazione di funzione STATIC
  }
}
```

Modello run-time dell'invocazione di funzioni

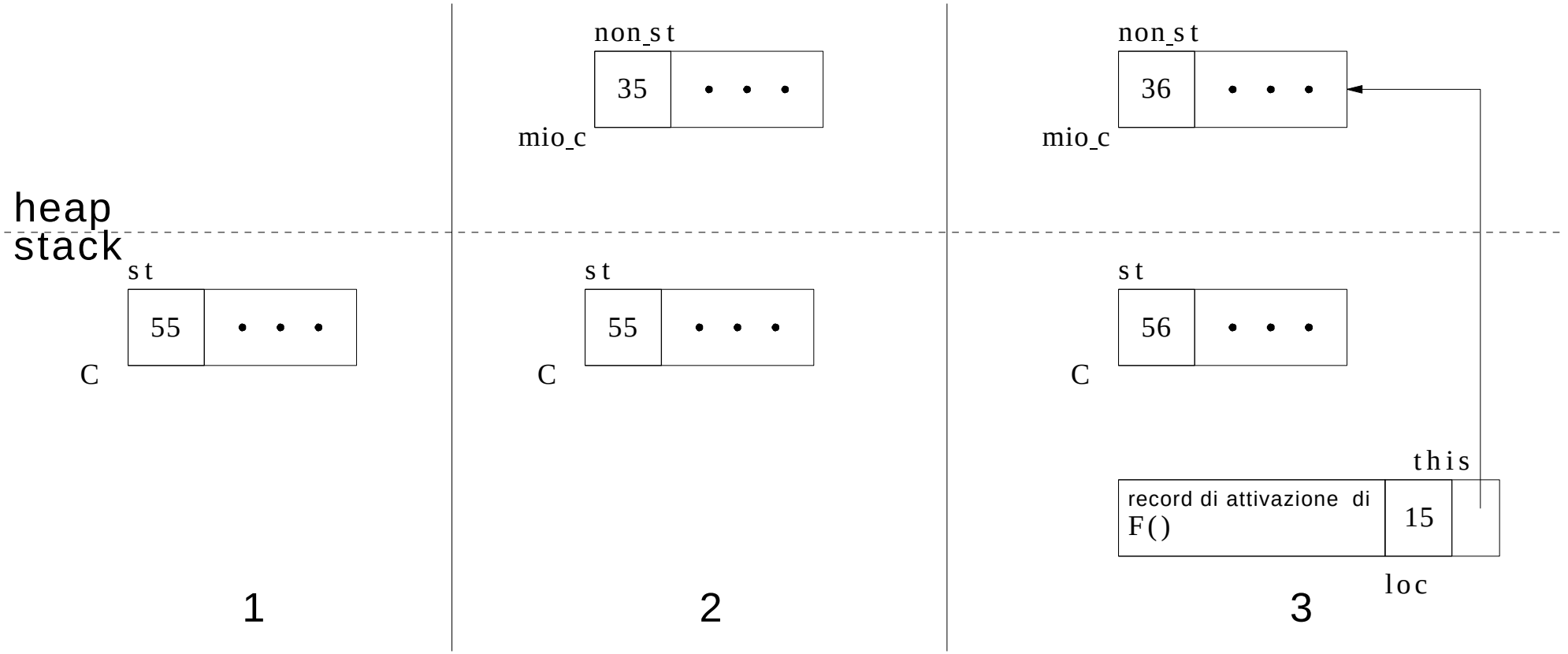
- Quando una funzione viene invocata viene allocato nello **stack** un record di attivazione che contiene le informazioni indispensabili per l'esecuzione.

- Fra queste informazioni, ci sono:
  - le variabili locali,
  - un *riferimento*, o *puntatore*, (il cui nome è `this`) all'oggetto di invocazione.

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

```
// File Esempio3.java
// Evidenzia il comportamento run-time di una chiamata di funzione
class C {
  static int st;
  int non_st;
  voidF()  {
    int loc = 15;
    non_st++;
    st++;
    System.out.println(st + "  " + non_st + "  " + loc);  }
}
public class Esempio3 {
  public static void main(String[] args) {
    C.st = 55;                                    // 1
    C mio_c;
    // mio_c.F();  // NO: la memoria non è stata ancora allocata
    mio_c = new C();
    mio_c.non_st = 35;                            // 2
    mio_c.F();      // OK: stampa 56 36 15        // 3
  } }
```

Evoluzione (run-time) dello stato della memoria



Esercizio 1: stack e heap

Progettare due classi:

Punto: per la rappresentazione di un punto nello spazio tridimensionale, come aggregato di tre valori di tipo reale;

Segmento: per la rappresentazione di un segmento nello spazio tridimensionale, come aggregato di due punti.

Esercizio 1 (cont.)

Scrivere una funzione `main()` in cui vengono creati:

- due oggetti della classe `Punto`, corrispondenti alle coordinate $\langle 1, 2, 4 \rangle$ e $\langle 2, 3, 7 \rangle$, rispettivamente;

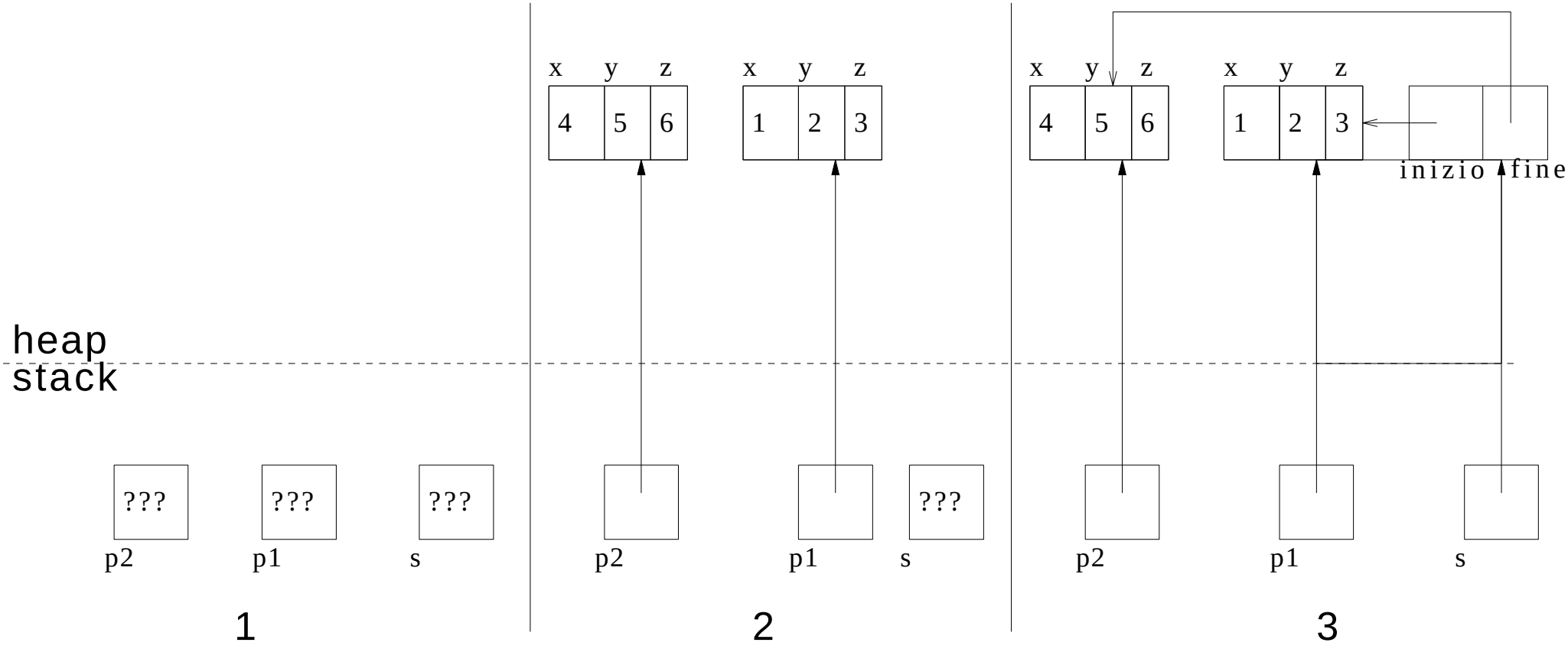
- un oggetto della classe `Segmento`, che unisce i due punti suddetti.

Raffigurare l'evoluzione dello stato della memoria, distinguendo fra stack e heap.

Soluzione esercizio 1

Soluzione esercizio 1

Esercizio 1: evoluzione stato della memoria



Comunicazione fra unità di programma

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

- Ciò significa che:

  1. il **parametron 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;

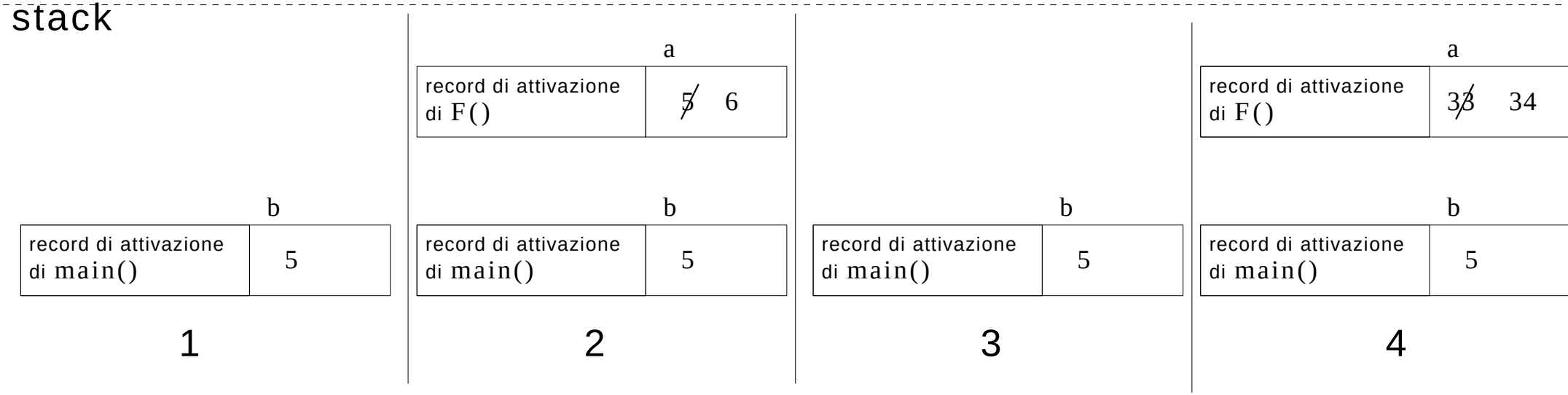
  3. tale locazione viene ovviamente perduta al termine dell'esecuzione della funzione, quando il record di attivazione corrispondente viene deallocato.

Comunicazione fra unità di programma (cont.)

Esempio: argomento passato appartiene ad un **tipo base** (int).

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

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

Evoluzione (run-time) dello stato della memoria



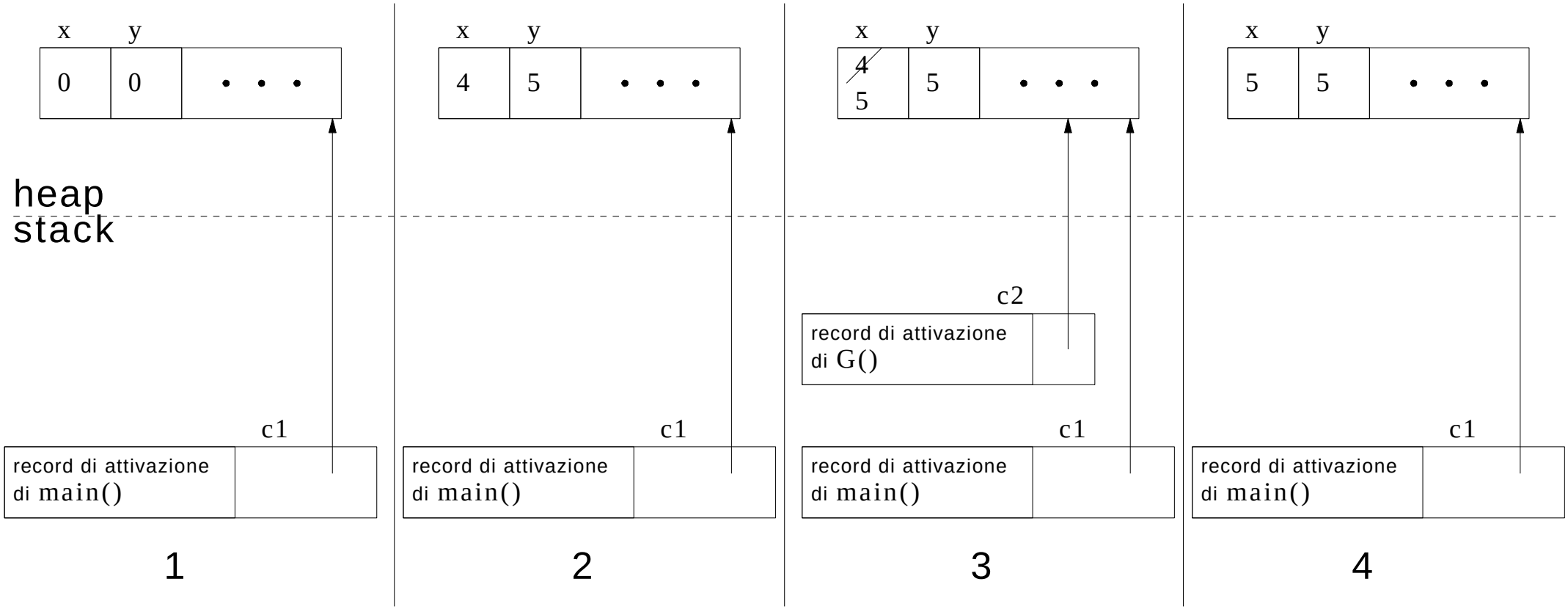
Comunicazione fra unità di programma (cont.)

Esempio: l'argomento passato appartiene ad una **classe** (C).

```
class C{ int
  x, y;
}
// ...
  public static void G(C c2) {
    c2.x++;
  }

  public static void main(String[] args) {
    Cc1 = new C();                                                    // 1
    c1.x = 4; c1.y = 5;                                               // 2
    G(c1);      // SIDE-EFFECT                                        // 3
    System.out.println("c1.x: " + c1.x + ", c1.y: " + c1.y);         // 4
  }
}
```

Nota: l'oggetto cambia, **il riferimento** no!

Evoluzione (run-time) dello stato della memoria



Riassunto passaggio argomenti

	argomento TIPO BASE	argomento CLASSE
viene copiato	il valore	il riferimento, non l'oggetto
può cambiare	niente	oggetto
non può cambiare	—	riferimento

Nota: se la funzione cliente vuole essere assolutamente sicura di non alterare l'oggetto passatogli tramite riferimento, deve:

1. **farsene una copia** mediante `clone()` e lavorare sulla copia *(vedi dopo)*.
2. sapere che l'oggetto passato tramite riferimento **non dispone di metodi che fanno side-effect** *(vedi dopo)*.

Restituzione da parte di una funzione

Anche la restituzione da parte di una funzione è **solamente per valore**.

```
  public static C H(C c3) {
    c3.x++;
    return c3;  // RESTITUZIONE PER VALORE
  }
// ...
    System.out.println(H(c1).x);
```

Pertanto, tutte le considerazioni sul passaggio di argomenti valgono anche per la restituzione. Ad esempio:

- 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 dell'oggetto**.

Esercizio 2: passaggio e restituzione

Con riferimento alla classe Segmento vista in precedenza, scrivere le seguenti funzioni esterne ad essa, tutte con un argomento di tale classe:

1. **lunghezza(Segmento s)**, che restituisce un valore di tipo double corrispon- dente alla lunghezza del segmento;

2. **inizioInOrigine(Segmento s)**, che modifica l'argomento ponendo il punto di inizio nel punto di origine (cioè di coordinate $\langle 0, 0, 0 \rangle$);

3. **mediano(Segmento s)**, che restituisce un riferimento al punto (di classe Punto) mediano del segmento;

4. **meta(Segmento s)**, che restituisce un riferimento ad un segmento (di classe Segmento) i cui estremi sono, rispettivamente, l'inizio e il mediano del segmento passato come argomento.

Soluzione esercizio 2 (1)

Soluzione esercizio 2 (2)

Soluzione esercizio 2 (3)

Esercizio 3: cosa fa questo programma?

```
// File Esercizio3.java
public class Esercizio3 {
  static void mistero1(int i, int j) {
    int temp =i;
    i =j;
    j =temp;
  }

  static void mistero2(Integer i, Integer j) {
    Integer temp =newInteger(i.intValue()); //viene creato un'istanza della classe
                                            //Integer inizializzata la valore i.intValue()
    i =j;
    j =temp;
  }

  public static void main(String[] args) {
    int p=5, s =7;
    mistero1(p,s);
    Integer o_p =newInteger(50), o_s =newInteger(70);
    mistero2(o_p,o_s);
    System.out.println("p: " +p +" s: " +s +" o_p: " +o_p +" o_s: " +o_s);
  }
}
```

Soluzione esercizio 3

Esercizio 4: side-effect

Scrivere un'unità di programma che riceve, come parametri di input, due locazioni di tipo intero e scambia il loro contenuto.

Suggerimento: non utilizzare nè **int** nè **Integer** per rappresentare gli interi.

Soluzione esercizio 4

Classi: qualificatori dei campi dati

Esistono tre tipi di qualificazione per i campi dati:

```
class C {
  int x;
  staticint y;
  final int z=  12;
}
```

static: campo relativo alla classe, non all'oggetto;

esiste anche per campi funzione, con lo stesso significato;

Classi: qualificatori dei campi dati (cont.)

final: campo costante, deve essere inizializzato;

esiste anche per campi funzione, **ma ha diverso significato** *(vedi dopo)*;

nessuna: campo relativo all'oggetto;

può essere inizializzato, altrimenti riceve un valore di default:

- `0` (tipi base numerici);
- `\0` (tipo char)
- `false` (tipo base `boolean`);
- `null` (riferimenti a oggetti).

Classi: overloading di funzioni

È ammesso l'*overloading* (dall'inglese, sovraccarico) di funzioni.

È possibile definire nella stessa classe più funzioni **con lo stesso nome**, purchè differiscano nel numero e/o nel tipo dei parametri formali.

Non è invece possibile definire due funzioni con lo stesso nome e stesso numero e tipo di argomenti ma diverso tipo di ritorno.

```
class C {
  int x;
  void F() { x++; }                    //  OK
  void F(int i) { x = i; }             //  OK
  void F(int i, int j) { x = i * j; }  //  OK
  //  int F() { return x; }            //  NO
  //      ^
  // Methods can't be redefined with a different return type
}
```

Classi: costruttori

Un costruttore è una funzione che:

- si chiama con lo stesso nome della classe;
- gestisce la nascita di un oggetto;
- viene invocata con **new**;
- (come le altre) può essere sovraccaricata.

Costruttori: esempio di definizione e uso

```
class C {
  int x,y;
  C(int p) { x = p; }
  C(int p, int s) { x = p; y = s; }
}
// ..
  public static void main(String[] args) {
    C c1 = new C(4);      // viene scelto il costruttore AD UN ARGOMENTO
    System.out.println("c1.x: " +      c1.x+",     c1.y:"  +  c1.y);
    C c2 = new C(7,8);   // viene scelto il costruttore A DUE ARGOMENTI
    System.out.println("c2.x: " + c2.x +", c2.y:  " + c2.y);
  }
}
```

Costruttore senza argomenti

- Per le classi che **non hanno** dichiarazioni di costruttori viene invocato il cosiddetto *costruttore standard*.

- Il costruttore standard esiste per tutte le classi e non modifica l'inizializzazione ai **valori di default** dei campi dati *(in pratica non fa nulla)*.

- Il costruttore standard viene automaticamente **inibito** dal compilatore a fronte della dichiarazione di **un qualsiasi** costruttore da parte del programmatore.

- In quest'ultimo caso, **può** essere dichiarato esplicitamente un costruttore senza argomenti.

Classi: costruttore senza argomenti (esempio)

```
class C {//   HA  il  costr. Senza argomenti  int
      x, y;
}

class C1 { //  NON HA il costr. senza argomenti
   int x, y;
  C1(int p, int s) { x = p; y = s; }
}

class C2 { // HA il costr. senza argomenti
  int x, y;
  C2() { x = 0; y = 0; }
  C2(int p, int s) { x = p; y = s; }
}
```

Esercizio 5: costruttori

Equipaggiare le classi Punto e Segmento con opportuni costruttori.

Utilizzare i costruttori per creare:

- due oggetti della classe Punto, corrispondenti alle coordinate $\langle 1, 2, 4 \rangle$ e $\langle 2, 3, 7 \rangle$, rispettivamente;

- un oggetto della classe Segmento, che unisce i due punti suddetti.

Soluzione esercizio 5