Università di Roma "La Sapienza", Facoltà di Ingegneria

Corso di

Progettazione del Software

Anno Accademico 2003-2004

Corso di Laurea in Ingegneria Gestionale

Prof. Monica Scannapieco (A–L) & Prof. Giuseppe De Giacomo (M–Z)

Nozioni Preliminari di Programmazione e Java

Unità 0

Generalità su Java

- Linguaggio recente (1996); precursori: Smalltalk (fine anni '70), C++ (inizio anni '80). Fondamentalmente, simile al C++, ma con alcune importanti differenze. Esiste già C#!

- Ha avuto successo perché:
  - librerie standard (in particolare, per Internet) native
  - portabilità su varie piattaforme

- In questo corso:
  - *Java è lo strumento di realizzazione dei nostri progetti software.*
  - *Per fare ciò, faremo amplio uso delle caratteristiche orientate agli oggetti di Java.*

Ripasso di alcune nozioni fondamentali in Java

Argomenti che tratteremo in questa parte di corso:

1. Tipo di dato in Java
2. Riferimento e oggetti
3. Invocazione di funzioni (metodi)
4. Allocazione di variabili e di oggetti
5. campi static
6. Passaggio di parametri
7. Costruttori

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 (cioè 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

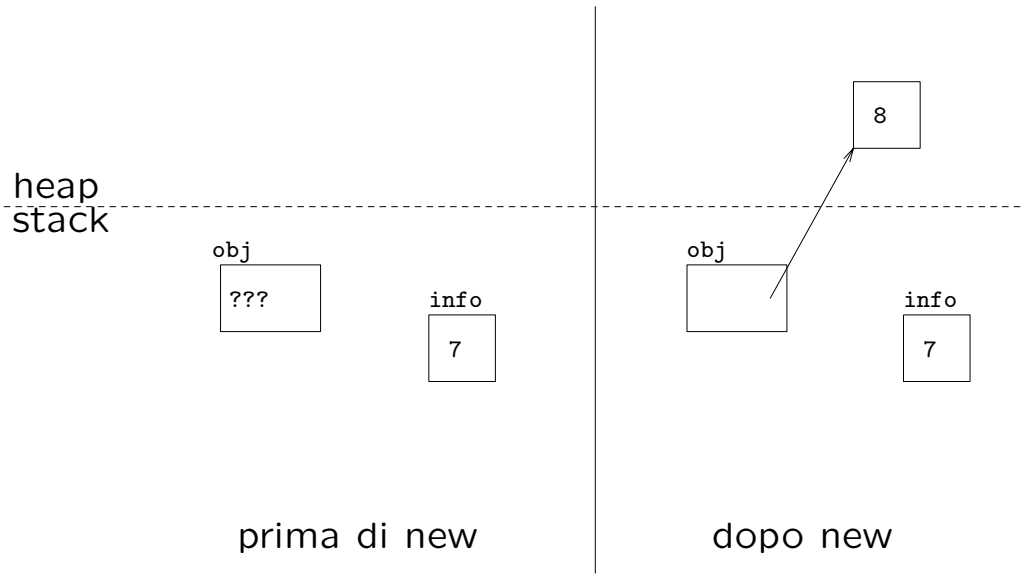
Inizializzazioni implicite per i campi delle classi

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 Unita0/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
                 // Ad esempio:
                 //    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*.

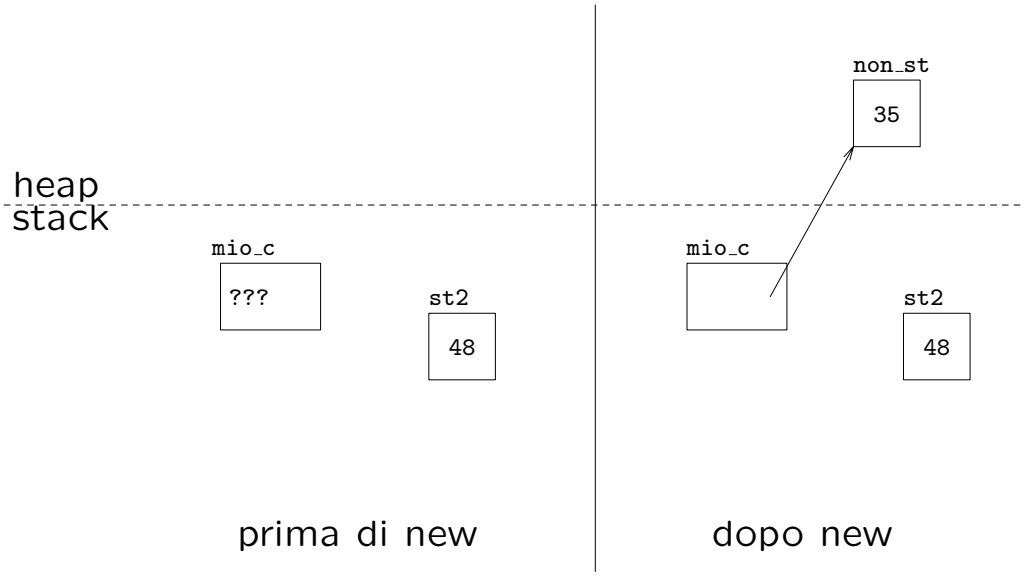
Esempi: variabile locale in una funzione, campo dati `static` di una classe, ...

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

Esempio: creazione di un oggetto tramite `new`.


```
// File Unita0/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

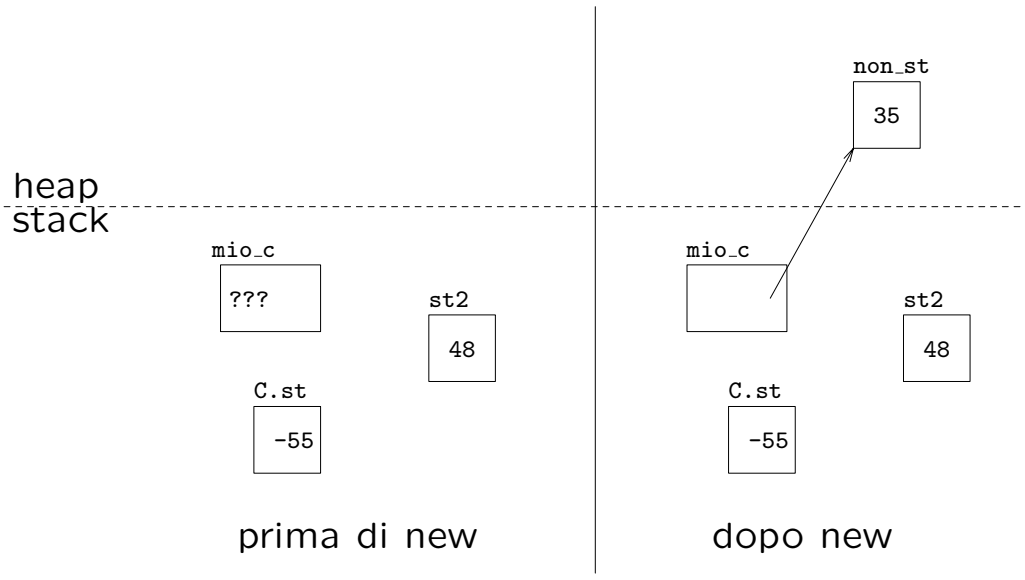


Campi dati static

- 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 Unita0/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 (è *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 Unita0/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 static void 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

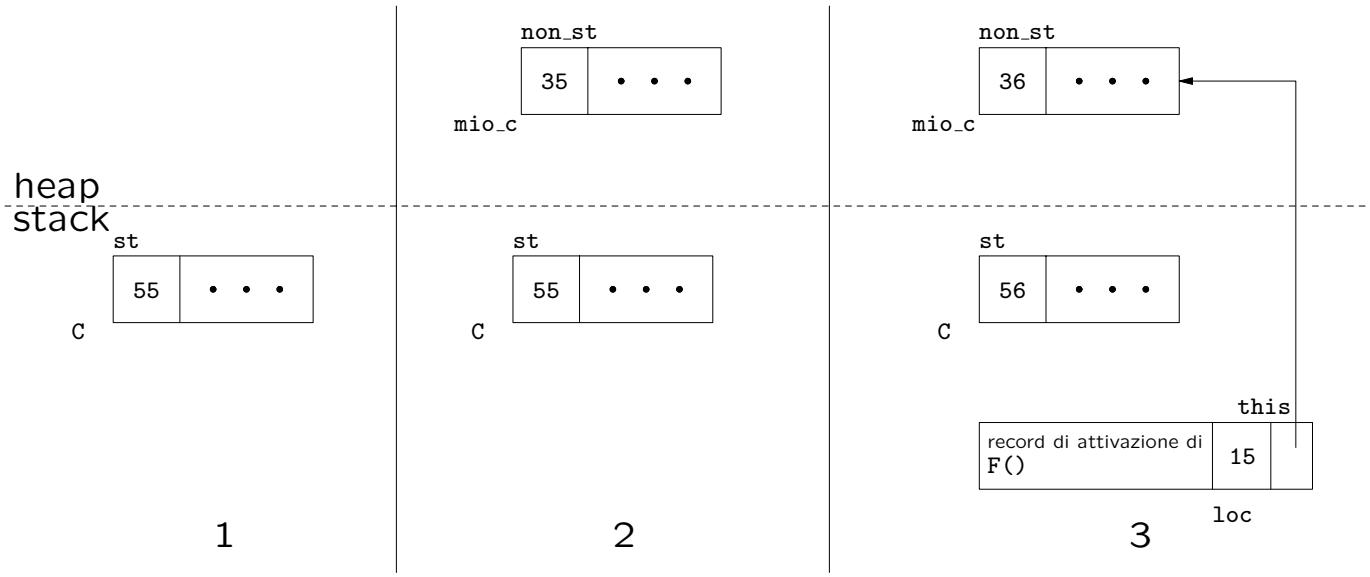
- Quando una funzione viene invocata, viene allocato **nello stack** un *record di attivazione*, che contiene le informazioni indispensabili per l'esecuzione.

- 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 Unita0/Esempio3.java
// Evidenzia il comportamento run-time di una chiamata di funzione
class C {
  static int st;
  int non_st;
  void F() {
    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.

Comunicazione fra unità di programma

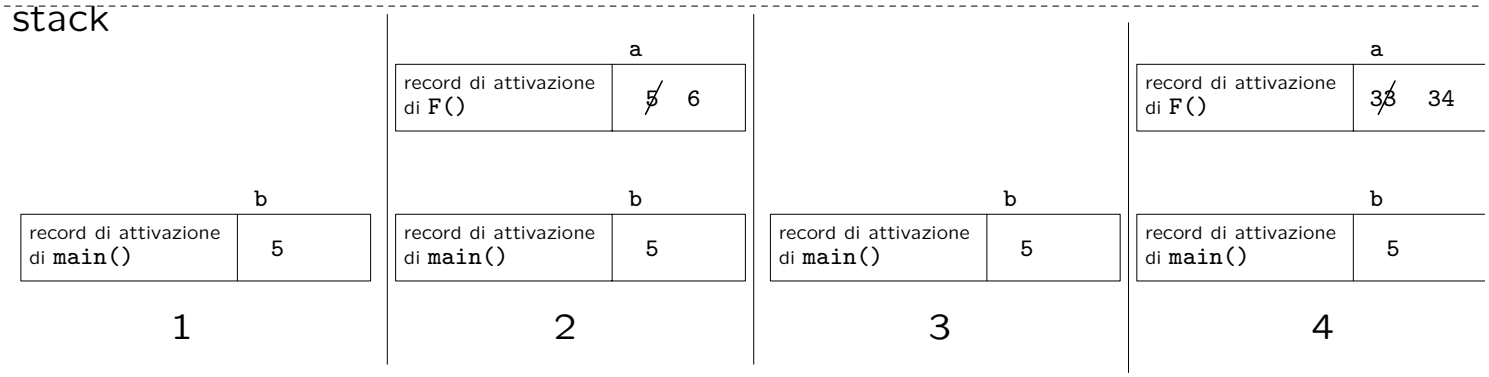
- Il passaggio di parametri ad una funzione è **solamente per valore**.
- Ciò significa che:
  1. il **parametro attuale** può essere un'espressione qualsiasi (costante, variabile, espressione non atomica);
  2. viene effettuata una **copia** del valore del parametro attuale nella locazione di memoria 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 passato appartiene ad un **tipo base** (`int`).

```
public static void F(int a) {
  // a e' il PARAMETRO FORMALE
  a++;
  System.out.println("a: " + a);
}

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
}
```


Evoluzione (run-time) dello stato della memoria



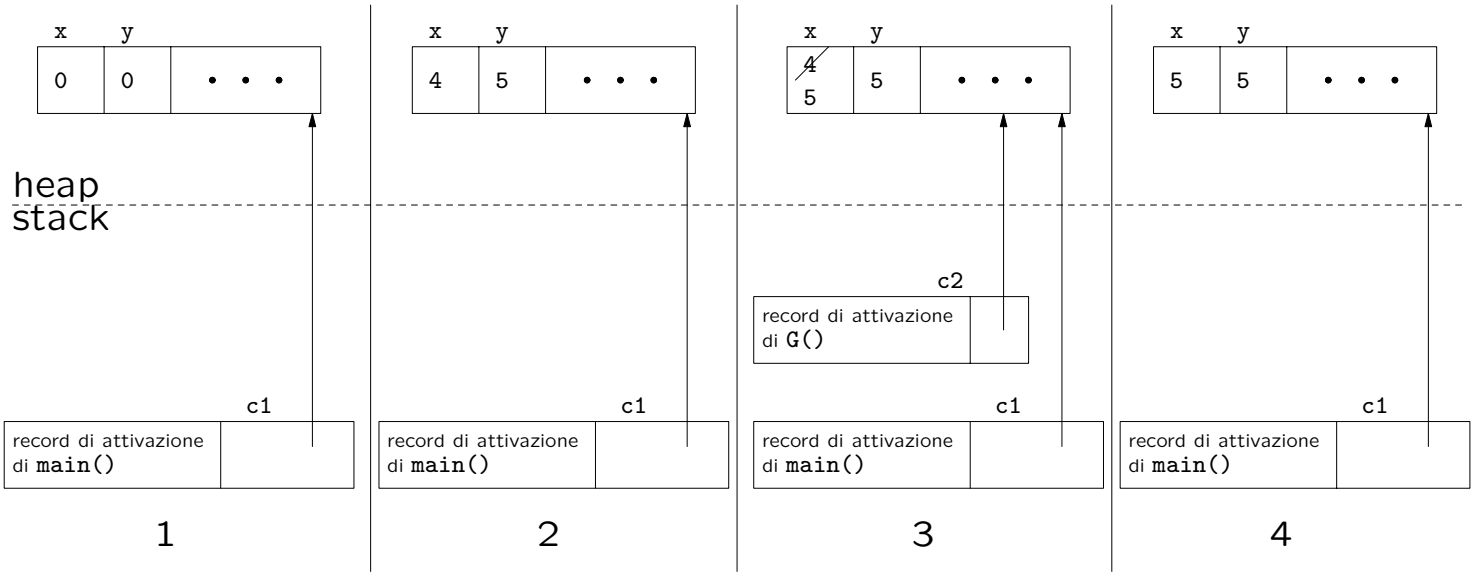
Comunicazione fra unità di programma (cont.)

Esempio: 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) {
    C c1 = 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()`, che restituisce un valore di tipo `double` corrispondente alla lunghezza del segmento;

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

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

4. `meta()`, 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.

Esercizio 3: cosa fa questo programma?

```
// File Unita0/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 = new Integer(i.intValue());
    i = j;
    j = temp;
  }

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

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 né `int` né `Integer` per rappresentare gli interi.

Classi: qualificatori dei campi dati

Esistono tre tipi di qualificazione per i campi dati:

```
class C {
  int x;
  static int 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**, purché 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.

Soluzioni degli esercizi dell'Unità 0

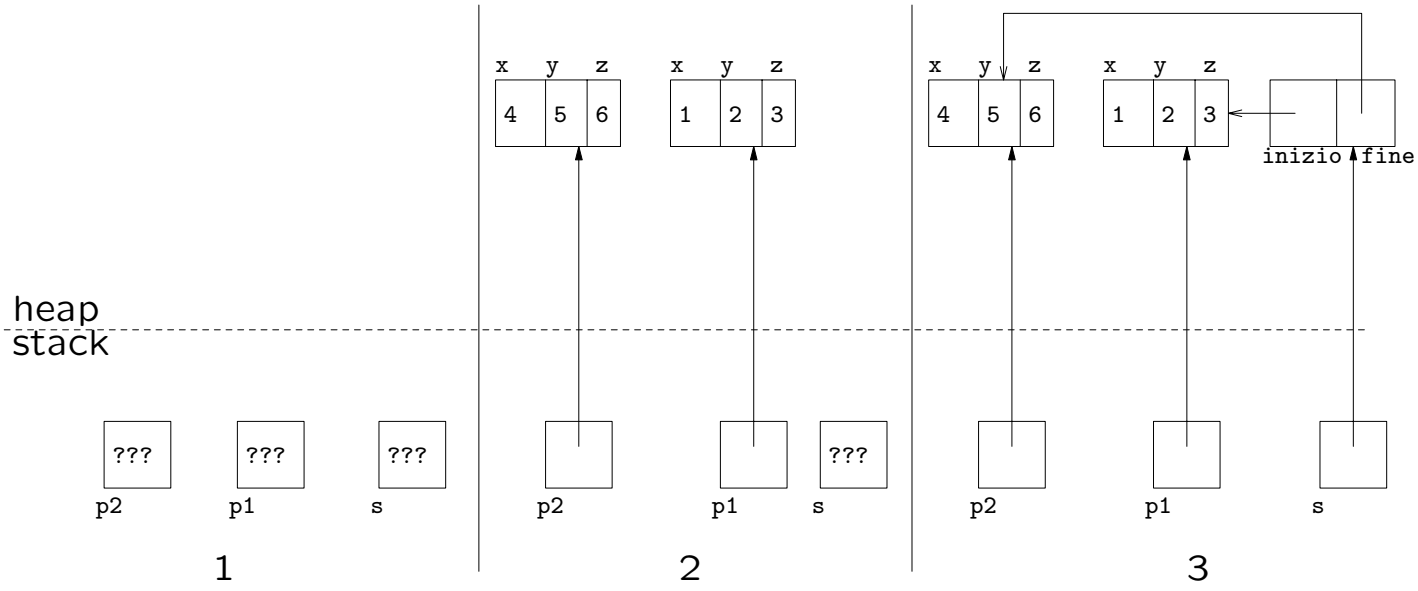
Soluzione esercizio 1

```
// File Unita0/Esercizio1.java
class Punto {
  float x, y, z;
}
class Segmento {
  Punto inizio, fine;
}
public class Esercizio1 {
  public static void main(String[] args) {
    Punto p1, p2;
    Segmento s;                          // 1
    p1 = new Punto();
    p2 = new Punto();
    p1.x = 1; p1.y = 2; p1.z = 4;
    p2.x = 2; p2.y = 3; p2.z = 7;        // 2
    s = new Segmento();
```



```
    s.inizio = p1; s.fine = p2;          // 3
  }
}
```

Esercizio 1: evoluzione dello stato della memoria



Soluzione esercizio 2

```
// File Unita0/Esercizio2.java
// Esercizio: passaggio e restituzione di argomenti

class Punto {
  float x, y, z;
}

class Segmento {
  Punto inizio, fine;
}

public class Esercizio2 {
  static double lunghezza(Segmento s) {
    return Math.sqrt( Math.pow((s.fine.x - s.inizio.x),2) +
                      Math.pow((s.fine.y - s.inizio.y),2) +
                      Math.pow((s.fine.z - s.inizio.z),2) ) ;
```

```
}

static void inizioInOrigine(Segmento s) {
  s.inizio.x = s.inizio.y =  s.inizio.z = 0;
}

static Punto mediano(Segmento s) {
  Punto med = new Punto();
  med.x = (s.fine.x + s.inizio.x) / 2;
  med.y = (s.fine.y + s.inizio.y) / 2;
  med.z = (s.fine.z + s.inizio.z) / 2;
  return med;
}

static Segmento meta(Segmento s) {
  Segmento met = new Segmento();
  met.inizio = s.inizio;
  met.fine = mediano(s);
```

```
  return met;
}

public static void main(String[] args) {
  Segmento s1 = new Segmento();
  s1.inizio = new Punto();
  s1.fine = new Punto();
  s1.inizio.x = s1.inizio.y = s1.inizio.z = 4;
  s1.fine.x = s1.fine.y = s1.fine.z = 10;
  System.out.println("Lunghezza di s1: " + lunghezza(s1));
  inizioInOrigine(s1);
  System.out.println("s1.inizio.x: " + s1.inizio.x + ", s1.inizio.y: " +
                     s1.inizio.y + ", s1.inizio.z: " + s1.inizio.z);
  Punto m = mediano(s1);
  System.out.println("m.x: " + m.x + ", m.y: " + m.y + ", m.z: " + m.z);
  Segmento s2 = meta(s1);
  System.out.println("s2.inizio.x: " + s2.inizio.x + ", s2.inizio.y: " +
                     s2.inizio.y + ", s2.inizio.z: " + s2.inizio.z);
```

```
    System.out.println("s2.fine.x: " + s2.fine.x + ", s2.fine.y: " +
                       s2.fine.y + ", s2.fine.z: " + s2.fine.z);
  }
}
```

Soluzione esercizio 3

- Il programma stampa:

  ```
  p: 5 s: 7 o_p: 50 o_s: 70
  ```

- Poiché in Java il passaggio di parametri è sempre per valore, non vi è alcun effetto sui parametri attuali nelle due chiamate.

- Possiamo dire che le due funzioni `mistero1()` e `mistero2()` sono completamente inutili, in quanto:

  – non restituiscono alcun valore;

  – non alterano, tramite side-effect, i loro argomenti.

Soluzione esercizio 4

```
// File Unita0/Esercizio4.java
// Esercizio: side-effect
class MioIntero {
  int dato;
}
class Esercizio4 {
  static void scambia (MioIntero i, MioIntero j) {
    int temp = i.dato;
    i.dato = j.dato;
    j.dato = temp;
  }
  public static void main (String[] args) {
    MioIntero p = new MioIntero(), s = new MioIntero();
    p.dato = 5; s.dato = -4;
    System.out.println(p.dato + " " + s.dato);
    scambia(p,s);
```



```
    System.out.println(p.dato + " " + s.dato);
  }
}
```

Soluzione esercizio 5

```
// File Unita0/Esercizio5.java
class Punto {
  Punto(float a, float b, float c) { x = a; y = b; z = c; }
  Punto() { } // punto origine degli assi
  float x, y, z;
}
class Segmento {
  Segmento(Punto i, Punto f) { inizio = i; fine = f; }
  Punto inizio, fine;
}
public class Esercizio5 {
  public static void main(String[] args) {
    Punto p1 = new Punto(1,2,4);
    Punto p2 = new Punto(2,3,7);
    Segmento s = new Segmento(p1,p2);
  }
}
```