Università di Roma "La Sapienza", Facoltà di Ingegneria

Corso di

Progettazione del Software

Corso di Laurea in Ingegneria Gestionale

Prof. Toni Mancini & Prof. Monica Scannapieco

AUTOV.Java.3

Nozioni Preliminari di Programmazione e Java

Versione del 3 gennaio 2007

Programmazione Orientata agli Oggetti

- Progettare un programma esteso richiede di strutturarlo in moduli quando più possibile indipendenti, legati fra loro attraverso relazioni definite in modo preciso. Una buona modularizzazione migliora la qualità del software(leggibilità, estendibilità, riusabilità...).

- Il processo di modularizzazione si appoggia ad una descrizione astratta della realtà d'interesse. Si può ricorrere a vari tipi di astrazione tra cui:

  - astrazione sulle operazioni (quali sono le operazioni rilevanti per l'applicazione)

  - astrazione sugli oggetti (quali sono le entità rilevanti per l'applicazione, e quali operazioni li manipolano)

- Java fornisce costrutti linguistici per supportare la programmazione orientata agli oggetti, cioè basata su un'astrazione sugli oggetti.

Programmazione Orientata agli Oggetti

- L'incapsulamento è un principio di modularizzazione che porta a raccogliere nello stesso modulo dati e operazioni relativi ad un insieme di oggetti. Java supporta l'incapsulamento attraverso il costrutto class.

- L'information hiding è un principio di modularizzazione secondo cui per utilizzare un modulo, è sufficiente conoscere la sua interfaccia verso l'esterno (servizi che offre, richiede), mentre i dettagli implementativi vanno nascosti. Java supporta l'Information Hiding attraverso il controllo dell'accesso ai membri delle classi ed alle classi stesse.

- Un oggetto del mondo reale può essere modellato specificandone *proprietà*(stato interno) e *comportamento*(operazioni).

- Java permette di gestire entità software (*oggetti*) dotati di una struttura dati che ne rappresenta lo stato, e un insieme di funzioni che implementano le operazioni disponibili.

- Una *classe* è una definizione di un tipo di oggetti che è possibile manipolare in un programma Java. Tutti gli oggetti appartenenti ad una data classe (si parla di *istanze*) sono descritti dalle stesse proprietà, e manipolabili tramite le stesse operazioni.

- Definire una classe significa definire le variabili che serviranno a mantenere lo stato interno delle sue istanze e i metodi che potranno essere utilizzati sulle sue istanze. Le variabili (o campi) e i metodi definiti internamente ad una classe, e cioè disponibili per gli oggetti della classe, si chiamano più genericamente membri. Un programma Java è costituito da un'insieme di definizioni di classi.

Dichiarazione di una classe

```
class UnaClasse{
    // ** membri ** (nome generico per indicare campi+metodi);

    // ** campi **
    //i campi non inizializzati subiscono un'inizializzazione di default.
    public int i;
    //Si può inizializzare un campo (anche di tipo riferimento) all'atto della dichiarazione
    public float f=0.5f;
    public boolean[] b=new boolean[5];

    // ** metodi **
    public boolean restituisciQualcosa(int x){
        int h=0  //variabile locale al metodo (non è un campo della classe)

        i++;     //I campi della classe hanno campo d'azione scope globale rispetto
                 //ai metodi della classe: sono visibili e modificabili da ogni metodo
    }

    public void faiQualcosAltro(){
        ...
    }
}
```

Nota: qualcuno usa il termine campo per indicare non solo le variabili ma membri di ogni tipo, (quindi distinguendo campi variabile e campi funzione)

Oggetti e variabili di tipo oggetto

```
public class ClasseMain{

    public static void main(String Args[]){
        //dichiarazione di una variabile di tipo UnaClasse
        UnaClasse o;

        //allocazione di un oggetto (un'istanza di UnaClasse) e assegnazione
        o=new UnaClasse();

        //accesso ai campi dell'oggetto
        o.f=0.5f;

        //accesso ai metodi dell'oggetto
        o.faiQualcosAltro()
    }
}
```

Gli oggetti in memoria

- Come gli array, gli oggetti sono porzioni di memoria allocata dinamicamente (sullo heap) contenente variabili il cui nome, tipo e valore iniziale sono definiti dalla dichiarazione dei campi nella classe.

- L'allocazione avviene tramite l'operatore new, come per gli array.

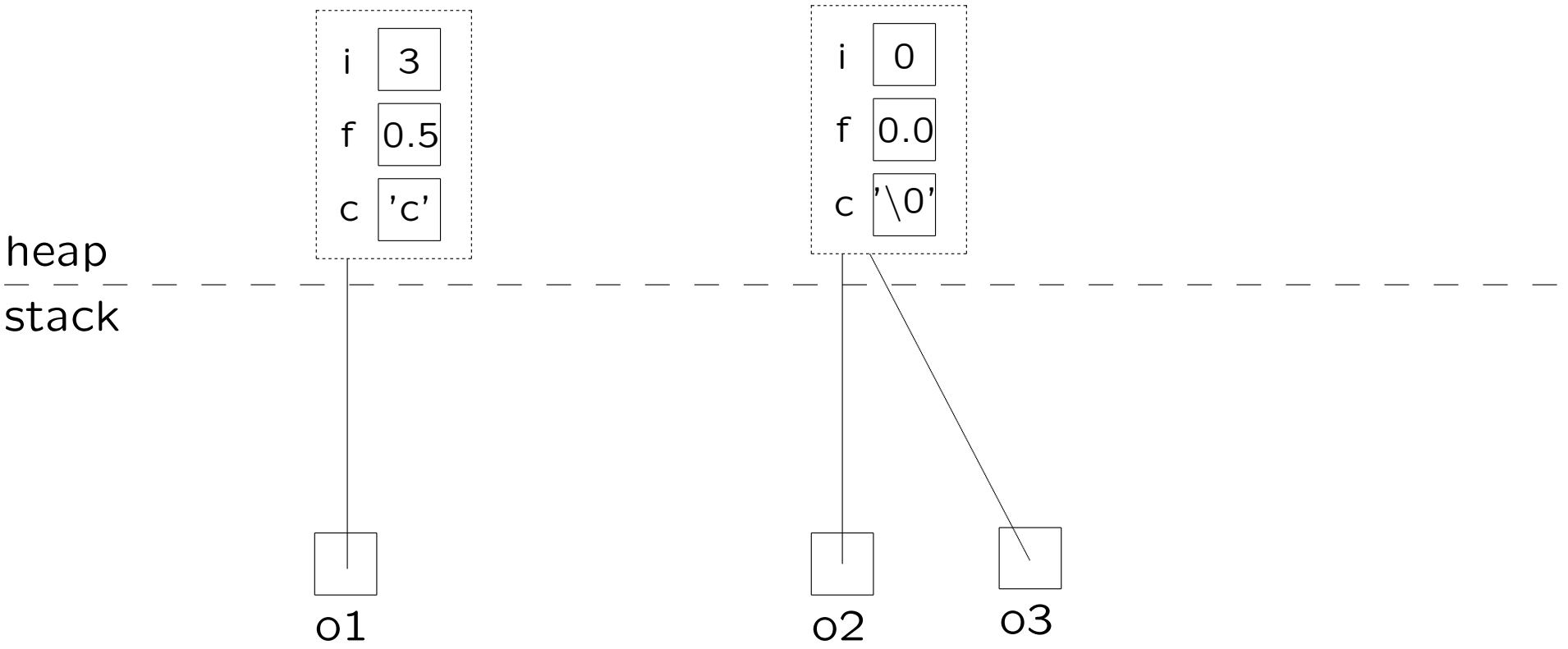
- Oggetti della stessa classe sono fra loro indipendenti: ciascuno può avere un differente stato interno (ovvero differenti valori per i campi).

- Le variabili di tipo oggetto sono variabili per riferimento, cioè contengono un riferimento ad una porzione di memoria allocata sullo heap (l'oggetto)

Esempio

```
public class ClasseMain{
    public static void main(String Args[]){
        int[] variabileArray=new int[10];
        C o1=new C();
        C o2=new C();
        C o3=o2;

        o1.i=3;
        o1.f=0.5f;
        o1.c='c';
    }
}

class C{
    public int i;
    public float f;
    public char c;

    public void metodoC{
        ...
    }
}
```

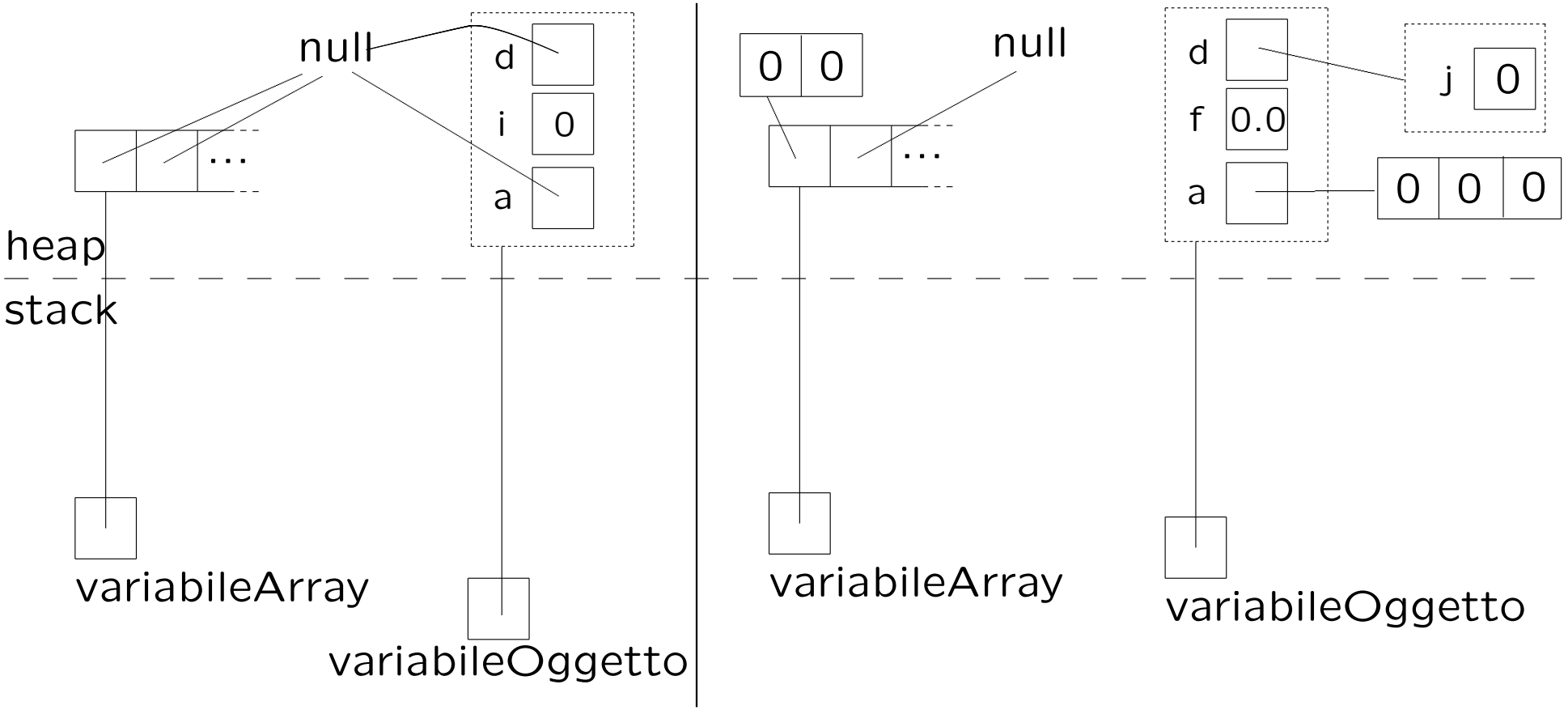
Comportamento in memoria



Oggetti con campi per riferimento

```
public class ClasseMain{
    public static void main(String Args[]){
        int[][] variabileArray=new int[10][];
        D variabileOggetto=new D();
        variabileArray[0]=new int[2];
        variabileOggetto.a=new int[3];
        variabileOggetto.d=new E();
    }
}
class D{
    public int i
    public int[] a;
    public E d;
    mioMetodo(){...};
}
class E{
    public int j;
}
```

Comportamento in memoria

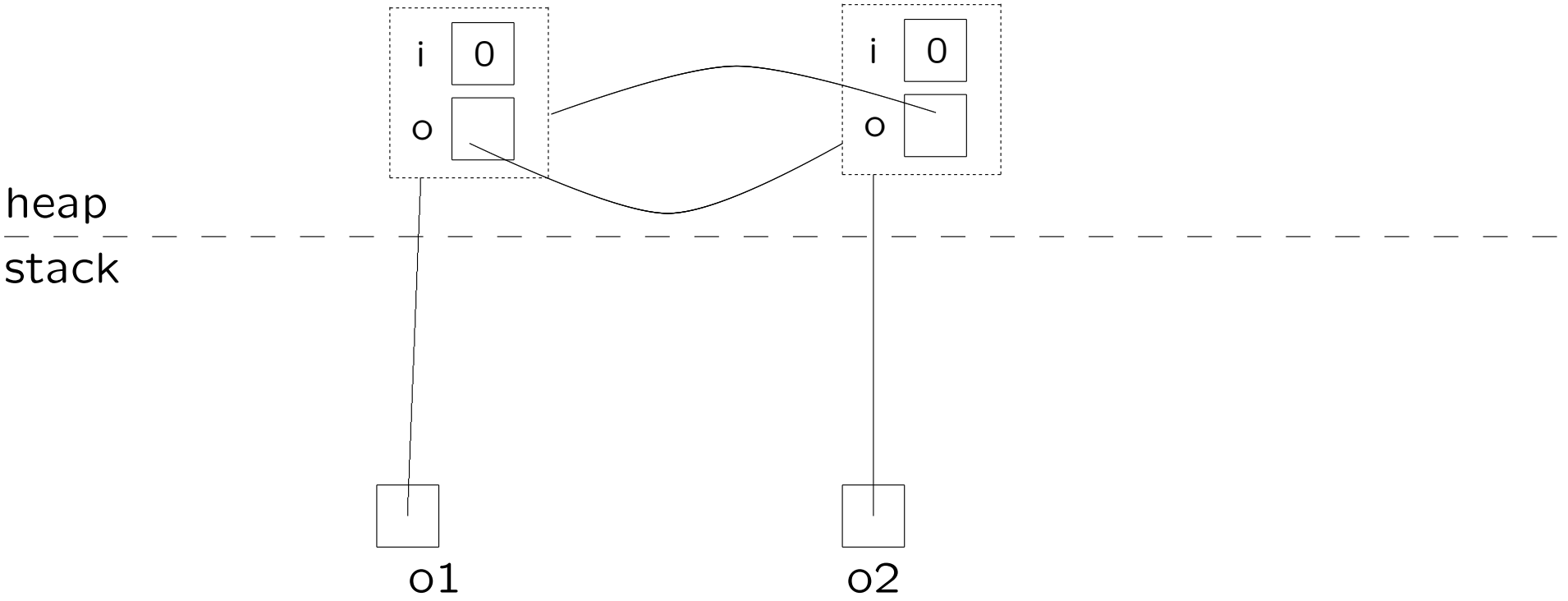


Osservazione

- le variabili membro di un oggetto possono essere di qualsiasi tipo, inclusi tipi oggetto, in particolare, possono essere dello STESSO tipo oggetto.

```
public class ClasseMain{
    public static void main(String Args[]){
        F o1=new F();
        F o2=new F();
        o1.a=o2;  //non c'è problema! Sono riferimenti, non scatole...
        o2.a=o1;  //quindi gli oggetti non si "contengono" a vicenda
                  //ma si "puntano" a vicenda
    }
}

class F{
    int i;
    public F o;
}
```

Comportamento in memoria



Invocazione di metodi della classe

```
public class ClasseMain{
    public static void main(String Args[]){
        Rettangolo r1=new Rettangolo();
        r1.b=5.0f;
        r1.a=6.0f;

        float p=r1.calcolaPerimetro(); //invocazione del metodo
    }
}

class Rettangolo{
    //variabili membro
    public float b, a; //base e altezza

    //metodi
    public float calcolaPerimetro(){
        float f=2*(b+a);
        return f;
    }

    public float calcolaArea(){
        return 2*(b+a);
    }
}
```

Osservazione

- Per i metodi invocati tramite un oggetto, il record di attivazione allocato sullo stack contiene anche la variabile *this*, che punta all'oggetto di invocazione ed è implicitamente impiegata nell'accesso ai membri della classe dall'interno del metodo.

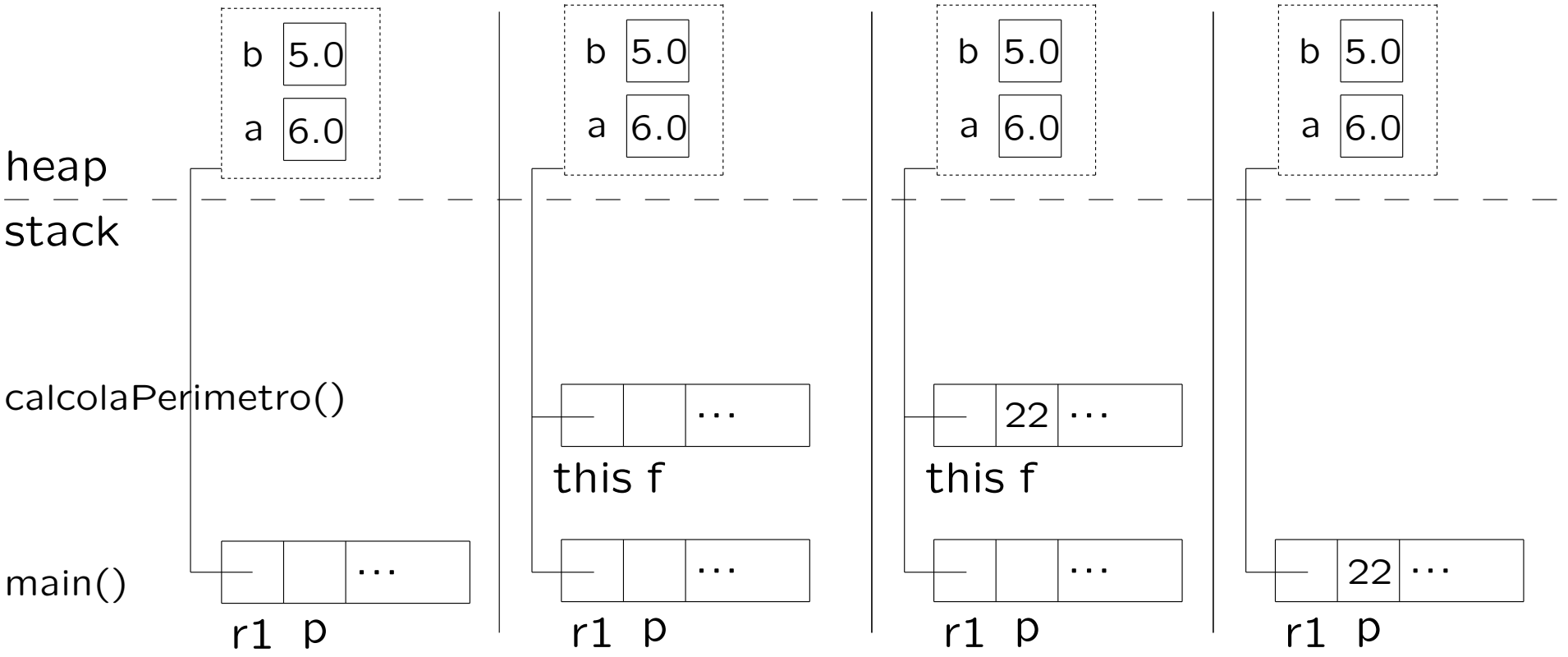
- tutte le considerazioni fatte a proposito di passaggio e restituzione di riferimenti valgono anche per oggetti:

  - I metodi non possono modificare il riferimento contenuto in una variabile oggetto passata come parametro attuale.

  - I metodi possono modificare l'oggetto puntato dal parametro attuale (side effect). Per evitarlo bisogna fare una copia esplicita.

  - i metodi possono restituire riferimenti a oggetti (eventualmente creati durante l'esecuzione del metodo).

Modello run-time dell'invocazione



Shadowing di variabili membro

Un metodo può dichiarare una variabile locale o parametro formale con lo stesso nome di un campo della classe. In questo caso, nel metodo o blocco il nome denota la variabile dichiarata localmente.

```
class Corda{
    //variabili membro
    public float spessore;
    public float lunghezza;

    //metodi
    public void cambiaLunghezza(int lunghezza){ //parametro formale che fa shadowing
        this.lunghezza=lunghezza;               //per disambiguare si usa this
    }

    public float ridSpessoreInTrazione(int[] coeff){
        float spessoreIniziale=spessore;
        float spessore=0;                       //variabile locale che fa shadowing
        for(int i=0;i<coeff.length;i++){
            spessore=spessore-spessoreIniziale/coeff[i];
        }
        return spessore;
    }
}
```

Lo shadowing, soprattutto da parte di variabili locali, non è molto utile, rende il codice poco leggibile e andrebbe evitato.

Visibilità dei membri di una classe

Java supporta l'information hiding permettendo di specificare diversi livelli di accesso ai membri della classe (sia campi che metodi)

- membri publici (modificatore d'accesso *public*)
- membri privati (modificatore d'accesso *private*)
- membri protetti (modificatore d'accesso *protected*)
- package (nessun modificatore d'accesso)

Per ora ci preoccupiamo solo di membri pubblici e privati. Un membro privato è accessibile solo dai metodi della classe. Un membro pubblico è accessibile anche dai metodi di altre classi.

Due implementazioni - un'interfaccia

```
class MioOrario{
    private long millisec;
    public void impostaOrario(int o, int m, int s){
        millisec=(o*3600+m*60+s)*1000;
    }
    public boolean piùTardiDi(MioOrario o1){
        return millisec>o1.millisec;
    }
}
class TuoOrario{
    private int ora, minuti, secondi;
    public void impostaOrario(int o, int m, int s){
        ora=o;
        ...
    }
    public boolean piùTardiDi(TuoOrario o1){
        if(ora>o1.ora)return true;
        etc...
    }
}
```

Osservazione: le altre classi possono interagire con questa solo attraverso i metodi pubblici, e non hanno accesso diretto ai dettagli implementativi. E' realizzata l'Information Hiding.

Classi: costruttori

Un costruttore è un metodo che:

- Ha lo stesso nome della classe, e gestisce la creazione di un oggetto della classe.
- Non ha tipo di ritorno (nemmeno void!)
- Può essere sovraccaricato.
- Viene invocato automaticamente durante l'allocazione di un oggetto tramite new. La sintassi è *new NomeClasse(parametri)*

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; }
}
```

Membri Static

- Fino adesso abbiamo visto che campi e metodi definiti in una classe sono legati agli oggetti che sono istanze della classe.

- E' possibile definire anche membri che sono relativi alla classe nel suo complesso, preponendo alla dichiarazione della varibile o metodo la keyword `static`.

- Una variabile static esiste indipendentemente dall'esistenza di istanze della classe, ed è condivisa da tutte le istanze della classe.

Membri Static (cont.)

- Un metodo static non è legato ad un particolare oggetto. Per invocarlo si usa il nome della classe.

- Visto che non ha alcun oggetto a cui fare riferimento (a meno che non ne usi uno passatogli o creato in modo esplicito), un metodo static può modificare solo campi della classe che siano stati dichiarati a loro volta static, ed invocare solo altri metodi static.

- Viceversa, da un metodo non static si possono invocare metodi static, e manipolare variabili static. In quest'ultimo caso, è bene ricordare che le modifiche saranno visibili a tutti gli oggetti della classe.

Allocazione di Variabili Static

- Come abbiamo visto, per ogni campo non-static esiste una locazione di memoria che viene allocata all'atto della creazione dell'oggetto tramite `new`. Quindi a variabili non-static di oggetti diversi corrispondono locazioni di memoria distinte.

- I campi static invece non sono legati a un oggetto. Per un campo static esiste **una sola locazione di memoria**, che viene allocata **prima** che venga allocato qualsiasi oggetto della classe.

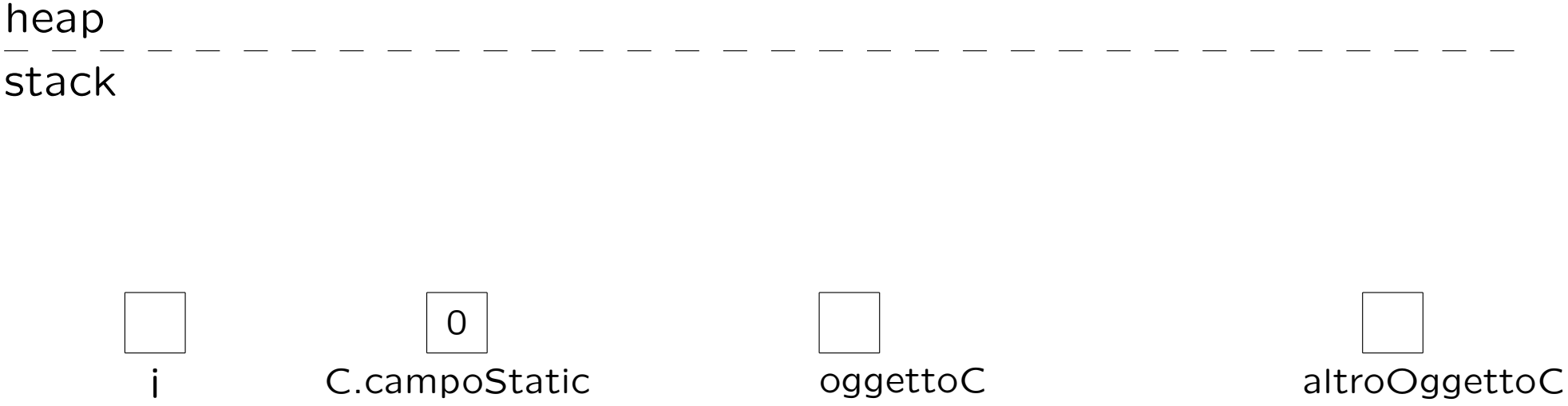
Esempio

```
class C{
  public int campoIstanza;
  public static int campoStatic;

  public void metodoIstanza(){
    System.out.println(campoIstanza);   //ok
    System.out.println(campoStatic);    //ok
    metodoStatic(); //ok
  }

  public static void metodoStatic(){
    System.out.println(campoStatic);    //ok
    //System.out.println(campoIstanza); //No, il campo è legato a un oggetto
    //metodoIstanza();                  //No, il metodo è legato a un oggetto
  }
}
```

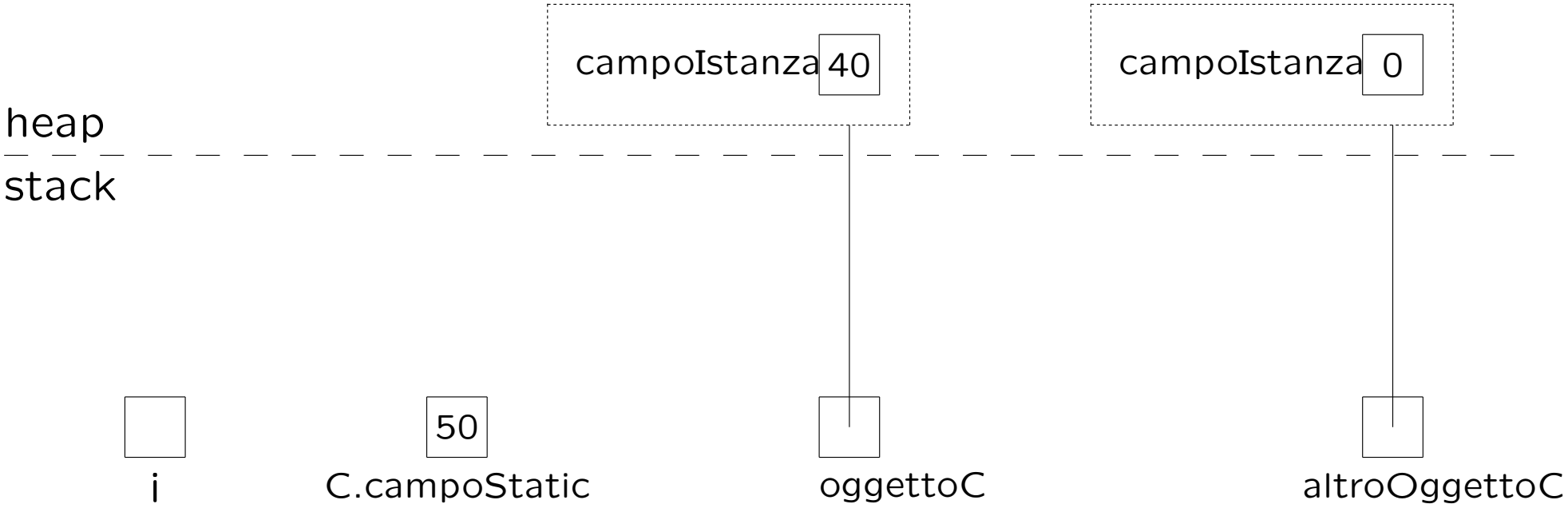
Esempio (cont.)

```
public static void main(String[] args) {
  int i;      //variabili locali (allocazione statica/no valore default)
  C oggettoC, altroOggettoC;

  //System.out.println(a);      //No, non ancora assegnato un valore
  //oggettoC.campoIstanza = 30; //No: oggetto non ancora allocato.
  //oggettoC.metodoIstanza();

  System.out.println(C.campoStatic); //ok (allocazione statica con valore di default)
  C.metodoStatic();                  //ok

  //allocazione dinamica degli oggetti
  oggettoC=new C(); altroOggettoC=new C();
  oggettoC.campoIstanza = 40;  //ok
  oggettoC.campoStatic=50;     //ok
  oggettoC.metodoIstanza();    //ok
  oggettoC.metodoStatic();     //ok

  System.out.println(altroOggettoC.campoIstanza); //stampa 0
  System.out.println(altroOggettoC.campoStatic);  //stampa 50:campoStatic è condiviso!

  //Notare:
  //C.campoIstanza=20; //No, il campo è legato a un oggetto
  //C.metodoIstanza(); //No, il metodo istanza è legato a un oggetto
}
```

Prima del new



Dopo il new


campoIstanza
40
campoIstanza
0
heap
stack
50
i
C.campoStatic
oggettoC
altroOggettoC

Variabili Final

- La keyword `final` può essere premessa alla dichiarazione di una variabile per indicare che il valore della variabile, una volta assegnato, non può più essere modificato. In pratica, serve a dichiarare che una certa variabile è una costante.

- si può usare sia per i campi delle classi che per le variabili locali ai metodi. La differenza è che nel caso delle variabili membro la variabile deve essere necessariamente inizializzata al momento della dichiarazione.

- Si possono combinare le keyword final e static per indicare costanti note a tutte le istanze di una certa classe.

- final si può usare anche per i metodi, **ma ha diverso significato***(lo vedrete in seguito)*;