Università di Torino – Facoltà di Scienze MFN
Corso di Studi in Informatica

Programmazione I - corso B
a.a. 2009-10

prof. Viviana Bono

Blocco 11 - Programmazione a oggetti

Programmazione a oggetti.

- oggetto: è una "entità" software (in memoria centrale, nella *heap*) costituita da un insieme di celle di memoria dette campi dell'oggetto, o variabili dell'oggetto, o variabili di istanza, e da un insieme di procedure dette metodi, con cui operare su tali celle;
- classe: è una specie di "stampo" per costruire oggetti aventi caratteristiche comuni, cioè è un pezzo di software contenente:
  - *la descrizione* dei campi che ogni oggetto della classe deve avere;
  - le istruzioni che compongono i metodi che permettono di agire sugli oggetti;
  - delle speciali procedure dette costruttori, aventi lo stesso nome della classe, che permettono di creare e inizializzare gli oggetti di quella classe.

- una classe può inoltre contenere:
  - campi statici, che sono celle di memoria proprie della classe e NON dei suoi oggetti;
  - metodi statici, che sono procedure della classe e NON dei suoi oggetti, e non possono quindi agire sui campi di un oggetto (ma solo sui campi statici)

Terminologia in uso **non** corretta

I campi, o variabili ***di*** istanza (o variabili ***dell'*** oggetto), vengono chiamati variabili istanza.

Si tratta di una **traduzione errata** dell'inglese instance variables:

istanza (di una classe) è sinonimo di oggetto (di una classe);

un campo è una variabile ***di*** un'istanza, cioè ***di*** un oggetto, **non è una variabile-oggetto**!

Anche variabili oggetto usato per le variabili di tipo riferimento-a-oggetti (di una data classe) è una terminologia non-standard e fuorviante.

Vedrete che le variabili di tipo riferimento-a-oggetto possono essere di tre generi:

- riferimento ad oggetto-array;
- riferimento ad oggetto di una data classe;
- riferimento ad un oggetto avente una data interfaccia.

Introduzione alla programmazione a oggetti in Java attraverso un esempio commentato

Modello di memoria centrale durante l'esecuzione di un programma Java

L'interprete Java (cioè il programma java.exe) mantiene 3 aree di memoria logicamente distinte:

- la method area o area delle classi: contiene le definizioni delle classi usate nel programma, cioè, per ciascuna classe:
  - le descrizioni dei campi degli oggetti (ma non i campi stessi), cioè i loro nomi, tipi, visibilità e eventualmente valori iniziali;
  - i campi statici o variabili di classe, cioè proprio i contenitori stessi;
  - i metodi, con le istruzioni che li compongono;
- lo heap (cioè *mucchio*): contiene gli oggetti che vengono via via creati durante l'esecuzione tramite l'istruzione new, cioè i loro campi di oggetto (detti anche variabili di istanza);
- lo stack o pila: contiene, durante ogni esecuzione di una procedura (metodo o *costruttore*), un frame contenente i parametri e le variabili locali di quella procedura.

file Conto.java

```
public class Conto {
  private static int numContiCreati = 0;
  private double saldo;
  private int numOperazioni;

  public Conto() {
    saldo = 0;
    numContiCreati++;
  }

  public Conto(double saldoIniziale) {
    saldo = saldoIniziale;
    numOperazioni = 0;
    numContiCreati++;
  }
  …
}
```


Definizione (semplice) di classe (ClassDeclaration)

visibilità (opzionale) nome

```
public class Conto  {
```

Definizioni costituenti il corpo della classe (ClassBodyDeclarations)

```
}
```

NB Secondo la nostra convenzione, la chiusa-graffa deve essere allineata con il primo carattere del costrutto sintattico (in questo caso la p di public); ciò è fatto automaticamente da TextPad.

Campi dell'oggetto, o variabili dell'oggetto o di istanza

```
public  class  Conto  {
  ...
  private double saldo;
  private int numOperazioni;
...
}
```

visibilità, tipo, e nome del contenitore (cioè del campo)

Ogni oggetto della classe **`Conto`** che sarà creato sullo heap (durante l'esecuzione di un programma utilizzante tale classe) sarà costituito da due campi di nome **`saldo`** e **`numOperazioni`**, capaci di contenere rispettivamente un double e un int, e non accessibili da procedure esterne alla classe **`Conto`**.

I campi di un oggetto sono detti anche

variabili dell'oggetto o variabili di istanza.

NB

Durante l'esecuzione ci saranno quindi tante variabili **`saldo`** e **`tasso`** quanti saranno gli oggetti creati (inizialmente nessuno).

Campi statici o variabili di classe

```
public class Conto {
  private static int numContiCreati = 0;
  private double saldo;
  private int numOperazioni;
  ...
}
```

quando la classe conto sarà caricata in memoria centrale nell'area delle classi, essa avrà al suo interno una cella **`numContiCreati`**, di cui esisterà un unico esemplare per tutta la classe, indipendentemente dal numero degli oggetti creati.

Nota L'inizializzazione a zero di numContiCreati è superflua: se assente, verrebbe fatta automaticamente da java.

Costruttori (I)

```
public class Conto {
  private static int numContiCreati;
  private double saldo;
  private int numOperazioni;

  public Conto() {
    saldo = 0;
    numOperazioni = 0;
    numContiCreati++;
  }
  ...
}
```

NOTA BENE
i costruttori non hanno tipo del risultato, nemmeno void.

costruttore, cioè procedura avente lo stesso nome della classe, che può venire invocata solo al momento della creazione di un oggetto; essa inizializzerà a zero i campi *saldo* e *numOperazioni* dell'oggetto creato, e incrementerà il campo statico *numConticreati*.

Nota Le inizializzazioni a zero dei campi dell'oggetto sono superflue: se assenti, verrebbero fatte automaticamente da java.

Costruttori (I): esempio di invocazione

```
public class ProvaConto {
  public static void main(String[] args) {
    Conto contoPaperone = new Conto();
  }
}
```

Inizializza campo saldo a 0

tipo e nome del contenitore

nel frame del metodo main vi sarà un contenitore in grado di contenere il riferimento a (cioè l'indirizzo di) un oggetto della classe Conto

crea nello heap un oggetto della classe Conto, con i campi come descritti nella definizione di classe, restituisce l'indirizzo di tale oggetto, e inoltre esegue su tale oggetto la procedura costruttore Conto()

l'assegnazione mette in contoPaperone l'indirizzo dell'oggetto

Costruttori (II)

```
public class Conto {
  private static int numContiCreati;
  private double saldo;
  private int numOperazioni;
  ...
  public Conto(double saldoIniziale) {
    saldo = saldoIniziale;
    numContiCreati++;
  }
  ...
}
```

un altro costruttore: in questo caso, esso ha bisogno di un parametro, per cui il chiamante deve fornire un valore (con il quale la procedura inizializzerà il campo *saldo* dell'oggetto).

Costruttori: esempio di invocazione (II)

```
public class ProvaConto {
  public static void main(String[] args) {
    Conto contoPaperone = new Conto(1000000);
  }
}
```

Inizializza campo saldo a 1000000

tipo e nome del contenitore

nel frame del metodo main vi sarà un contenitore in grado di contenere il riferimento a (cioè l'indirizzo di) un oggetto della classe Conto

crea nello heap un oggetto della classe Conto, con i campi come descritti nella definizione di classe, restituisce l'indirizzo di tale oggetto, e inoltre esegue su tale oggetto la procedura costruttore Conto(double saldoIniziale)

l'assegnazione mette in contoPaperone l'indirizzo dell'oggetto

Costruttori (I e II)

- Una classe può avere piú di un costruttore
- Conto ha due costruttori
- Entrambi si devono chiamare con il nome della della classe che li contiene, ovvero Conto
- Come si distinguono? Dal fatto che uno non ha parametri, invece l'altro si
- Il meccanismo che consente di dare nomi uguali a metodi (non solo costruttori) con parametri diversi (in numero, ma anche tipo) si chiama overloading

Costruttori

Importante: Le procedure costruttori possono essere invocate solo attraverso una *new*, al momento della creazione di un oggetto.

Un costruttore non può essere invocato su un oggetto preesistente per ri-inizializzarlo.

Nell'esempio precedente, **dopo l'istruzione:**

```
Conto contoPaperone = new Conto(1000000);
```

non è possibile scrivere:

```
contoPaperone.Conto(2000000);
```

È invece possibile scrivere:

```
contoPaperone = new Conto(2000000);
```

ma in tal modo si crea un secondo oggetto e si perde il riferimento al primo.

Riepilogo nozioni principali sui costruttori

Un costruttore è una procedura che ha lo stesso nome della classe in cui è definito e che non ha tipo del risultato, nemmeno void; esso può essere invocato soltanto attraverso l'espressione new, ad esempio:

new Conto(2000)

La valutazione (cioè l'elaborazione) di tale espressione ha come effetto la creazione dell'oggetto (che ci si può immaginare effettuata dalla parolina "new"), e poi l'esecuzione, su tale oggetto, del corpo del costruttore, che ne inizializza i campi. L'espressione, inoltre, restituisce un valore, e precisamente l'indirizzo dell'oggetto creato. Tale indirizzo, o riferimento, potrà essere messo in una opportuna variabile, tramite l'operazione di assegnazione; oppure passato "al volo" a un metodo, ecc.

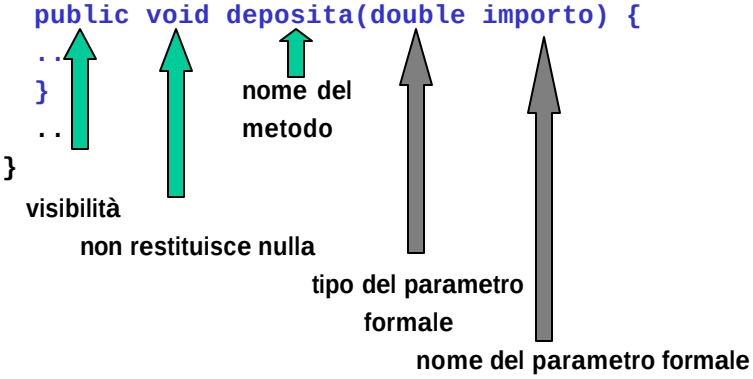
Un costruttore è quindi una sorta di metodo speciale, che non può essere invocato su un oggetto già esistente, ma soltanto all'atto della creazione di un oggetto.

Metodi della classe Conto

```
public class Conto {
  private static int numContiCreati;
  private double saldo;
  private int numOperazioni;
  ...
  public void deposita(double importo) {
    if(importo < 0)
      System.out.println("errore: valore < 0");
    else {
      saldo = saldo + importo;
      numOperazioni++;
    }
  }
  ...
}
```

Esaminiamo l'intestazione (header) di un metodo

```
public class Conto {
   ...
   public void deposita(double importo) {
   ..
   }
   ..
}
```



Il metodo *deposita* si aspetta un argomento di tipo double; esso dovrà quindi essere invocato passandogli un double, che il metodo automaticamente metterà nella cella *importo* del frame.

Segnatura di un metodo

La segnatura (o *signature* = firma) di un metodo è costituita da:

- il nome del metodo;
- il numero e i tipi (non i nomi) dei parametri formali.

Ad esempio, la segnatura del metodo `deposita` precedente è:

deposita(double)

Se si definisce un metodo

```
public void trasferisci(double importo, Conto conto) {
  ...// trasferisce un importo ad un altro conto
}
```

la sua segnatura è:

trasferisci(double, Conto)

Metodi: esempio di invocazione

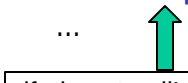
- **Importante!** Per poter scrivere l'invocazione di un metodo, non c'è bisogno di conoscere il corpo del metodo, cioè non c'è bisogno di sapere che cosa il metodo fa: basta conoscere la sua segnatura, cioè il suo nome e quanti argomenti vuole, e di che tipi.
- Non serve sapere i nomi dei parametri formali: essi sono di uso interno del metodo, e non interessano al chiamante.

```
public class ProvaConto {
  public static void main(String[] args) {
    Conto contoPaperone = new Conto(1000000);
    contoPaperone.deposita(240.61);
  }
}
```

Per poter invocare deposita non c'è bisogno di sapere che il suo parametro si chiama *importo*; basta sapere che è un double, e che 240.61 è appunto un double.

Metodi: esempio di invocazione

```
public class provaConto {
  public static void main(String[] args) {
    Conto contoPaperone = new Conto(1000);
    contoPaperone.deposita(240.61);
    ...
```



riferimento all'oggetto per cui il metodo viene invocato

argomento passato al metodo

il metodo main chiama il metodo *deposita* per l'oggetto il cui indirizzo è memorizzato nella variabile *contoPaperone*;

sullo stack, sotto al *frame* di *main* si impila un frame per *deposita*, che è un contenitore composto di:

- una cella, di nome predefinito *this*, in cui viene automaticamente messo l'indirizzo dell'oggetto su cui agire;
- una cella di nome *importo* in cui viene automaticamente messo l'argomento 240.61.

Uso di this (cioè "questo")

Il nome predefinito this può essere usato all'interno dei metodi e dei costruttori; ad esempio, nel costruttore:

```
public Conto(double saldoIniz) {
  this.saldo = saldoIniz;// equivale a saldo = saldoIniz;
  numContiCreati++;
}
```

Anzi: se il parametro formale viene chiamato anch'esso *saldo*, e in tal modo maschera all'interno della procedura il campo *saldo* dell' oggetto, l'unico modo di accedervi è proprio attraverso *this*.

Esempio:

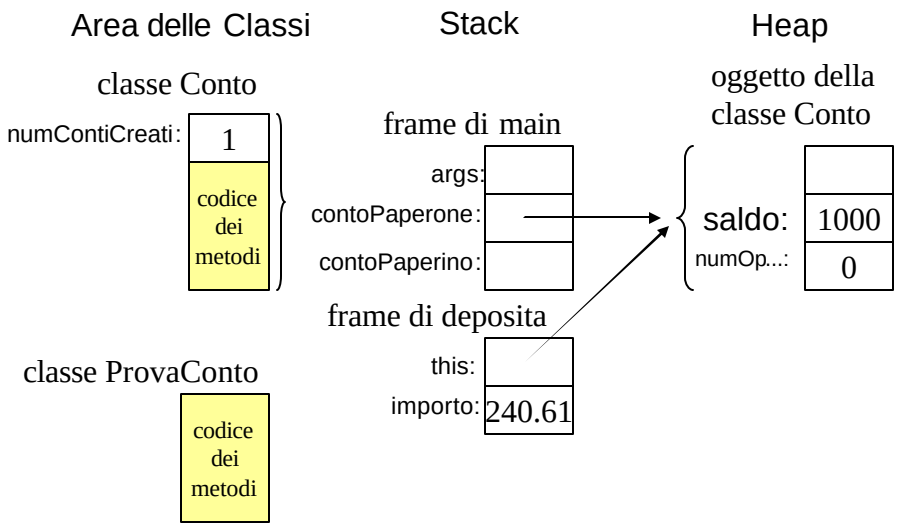
```
public Conto(double saldo) {
  this.saldo = saldo;//
  numContiCreati++;
}
```

NB L'istruzione saldo = saldo ricopierebbe semplicemente il valore del parametro in se stesso, e non avrebbe quindi alcun effetto.

Il file provaConto

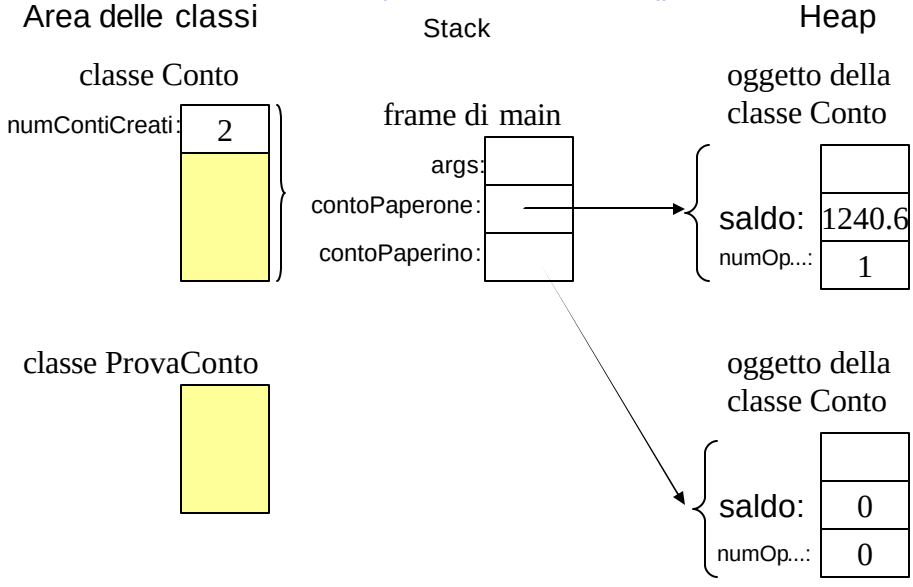
```
public class provaConto {
  public static void main(String[] args) {
    Conto contoPaperone = new Conto(1000);
    Conto contoPaperino;
    contoPaperone.deposita(240.61);
    contoPaperino = new Conto();
  }
```

Stato della memoria all'istante dell'invocazione di *deposita*



Stato della memoria dopo l'esecuzione dell'istruz. successiva:

```
contoPaperino = new Conto();
```



Corpo di un metodo

Le istruzioni che compongono il corpo di un metodo vengono eseguite ogni volta che il metodo viene invocato.

Torniamo, ad esempio, all'istante dell'invocazione

```
contoPaperone.deposita(240.6);
```

all'interno del metodo main, e quindi alla situazione della memoria rappresentata nella diapositiva 28:

negl'istanti successivi saranno eseguite le istruzioni costituenti il corpo del metodo `void deposita(double importo)` cioè

```
if(importo < 0)
  System.out.println("errore: valore < 0");
else
  saldo = saldo + importo;
```

Il valore contenuto nel contenitore *importo* (che sta nel frame di *deposita*) viene inviato all'ALU, che esegue l'operazione di confronto con 0 e restituisce il risultato *true* o *false*; se *true* viene eseguita l'istruzione *println* ecc.; se *false* vengono eseguite le istruzioni:

```
saldo = saldo + importo; numOperazioni++;
```


Esecuzione dell'istruzione `saldo = saldo + importo;`

```
public void deposita(double importo) {
  if(importo < 0)
    System.out.println("errore: valore < 0");
  else {
    saldo = saldo + importo;
    numOperazioni++;
  }
}
```

l'interprete java va a cercare un contenitore di nome *saldo*:

poiché nel frame di *deposita* non vi è nessun contenitore di tal nome, va a cercarlo dentro l'oggetto il cui riferimento gli è stato passato come parametro implicito *this*, cioè in questo caso l'oggetto il cui indirizzo è contenuto in *contoPaperone*;

prende quindi il contenuto della cella *saldo* di tale oggetto, prende inoltre il contenuto della cella *importo*, che invece è un contenitore locale di *deposita* che sta nel suo frame, ordina alla ALU di fare la somma dei due valori, e ne rimette il risultato nella cella *saldo*.

Il ruolo di this come parametro implicito

Il metodo di oggetto (della classe Conto) della forma (semplificata):

```
void deposita(double importo) {
  saldo += importo;
  numOperazioni++;
}
```

è realizzato da Java (compilatore ed esecutore) ***come se fosse***:

```
static void deposita(Object this, double importo) {
  this.saldo += importo;
  this.numOperazioni++;
}
```

Corrispondentemente, l'invocazione di tale metodo su un oggetto, ad es.:

```
contoPaperone.deposita(250);
```

è realizzato da Java ***come se fosse***:

```
Conto.deposita(contoPaperone, 250);
```

this è cioè un parametro **IMPLICITO** di ogni metodo non-statico, e l'oggetto su cui il metodo viene invocato è l'argomento passato in tale parametro.

Parliamo di metodi statici…

I metodi qualificati **`static`** non ricevono il parametro implicito **`this`**, e quindi non possono accederne ai campi (degli oggetti), o tanto meno modificarli; possono accedere soltanto (eventualmente per modificarli) ai campi statici.

Essi possono (e anzi, nella buona programmazione, *devono*) essere invocati *su un nome di classe invece che su un oggetto*.

Insomma, i metodi statici sono, per così dire, metodi propri della classe, e non degli oggetti di tale classe.

Esempio: un metodo che semplicemente restituisca oppure azzeri il contatore dei conti creati, che è un campo statico, può (e anzi, nella buona programmazione, *deve*) essere dichiarato *static*.

Esempi di metodi statici

```
public static int numContiCreati() {
  return numContiCreati;
}

public static void resetNumContiCreati() {
  numContiCreati = 0;
}
```


Oggetti: assegnazione, passaggio parametri, uguaglianza

In Java un oggetto (sia oggetto di una classe sia array) non è mai manipolabile direttamente, ma soltanto attraverso un riferimento ad esso (cioè il suo indirizzo); non esistono in Java espressioni di tipo oggetto, ma soltanto di tipo riferimento-a-oggetto, cioè in un programma Java non scriviamo mai "un oggetto", ma soltanto un "riferimento ad un oggetto".

Da qui derivano delle conseguenze…

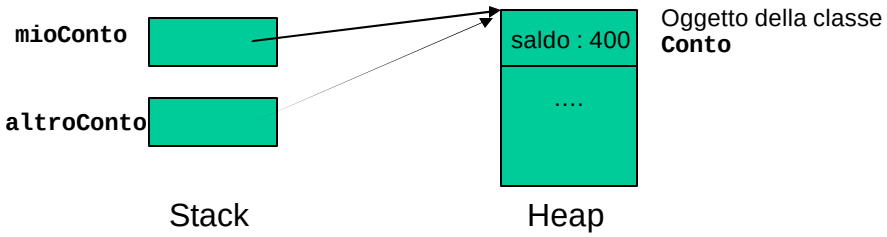
Oggetti: assegnazione (I)

L'assegnazione di variabili di tipo riferimento-a-oggetto non duplica l'oggetto, ma semplicemente copia il puntatore all'oggetto (cioè il suo indirizzo).

Esempio:

```
Conto mioConto = new Conto();
Conto altroConto;
…
altroConto = mioConto;
```


Oggetti: assegnazione (II)



L'istruzione **`altroConto = mioConto;`**

NON duplica l'oggetto, bensì copia nella cella **`altroConto`** l'indirizzo contenuto in **`mioConto`**; le due celle puntano allora allo stesso oggetto, quindi modificando **`altroConto`** si modifica anche **`mioConto`** e viceversa. [Lo stesso vale per gli array, che sono anch'essi degli oggetti.]

Oggetti: passaggio parametri (I)

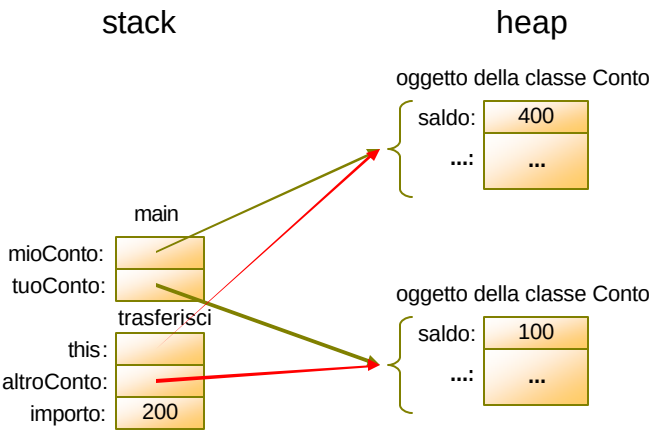
Nel passaggio di un parametro di tipo riferimento-a-oggetto, ciò che viene passato non è una copia dell'oggetto, ma il suo indirizzo (quindi il passaggio parametri dei tipi non primitivi è sempre automaticamente per riferimento). Nello stesso modo, un metodo il cui tipo del risultato è un tipo riferimento-a-oggetto restituisce l'indirizzo di un oggetto. Esempio:

```
void traferisci (Conto altroConto, int importo){
   saldo = saldo – importo;
   altroConto.saldo = altroConto.saldo +
  importo;
}
…
Conto mioConto = new Conto(400);//siamo nel main
Conto tuoConto = new Conto(100);
mioConto.trasferisci(tuoConto, 200);
```

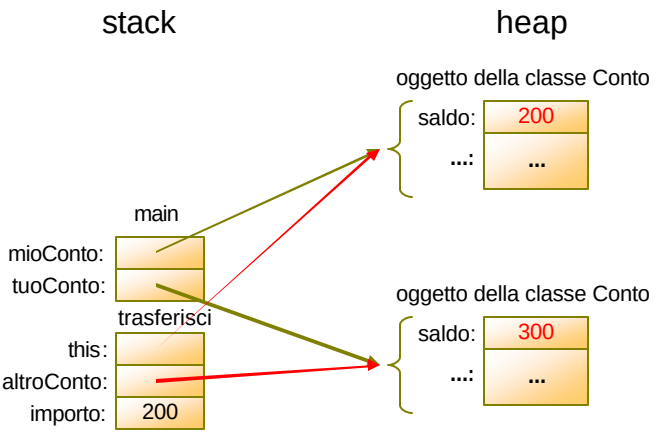

Oggetti: passaggio parametri (II)

Nel frame del metodo **trasferisci**, la cella-parametro-formale di nome **`altroConto`** contiene l'indirizzo dell'oggetto puntato da **`tuoConto`**; il metodo quindi modifica proprio tale oggetto…

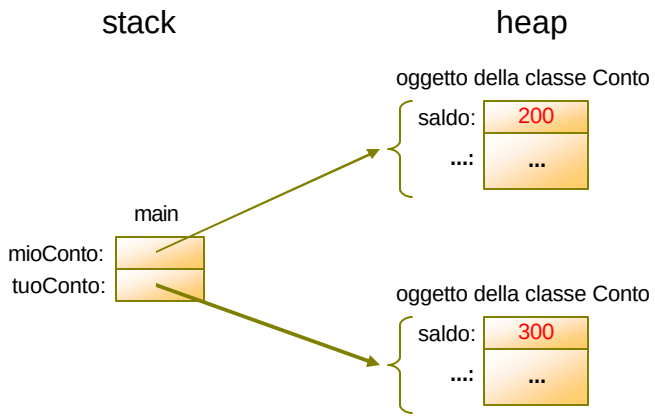
Passaggio parametri: rappresentazione della memoria (I)



Passaggio parametri: rappresentazione della memoria (II)



Passaggio parametri: rappresentazione della memoria (III)



Al termine dell'esecuzione del metodo trasferisci il frame di trasferisci viene deallocato e riprende l'esecuzione del metodo chiamante (in questo caso il main); gli oggetti i cui indirizzi erano stati passati come argomenti a trasferisci (sia l'argomento implicito mioConto che l'argomento esplicito tuoConto) risultano però permanentemente modificati.

Oggetti: uguaglianza (I)

L'operatore di uguaglianza == confronta sempre valori; quindi, se è applicato a due riferimenti a oggetti (di una classe oppure oggetti-array), confronta gli indirizzi di tali oggetti e non i loro contenuti.

Pertanto, l'operatore di uguaglianza applicato a due celle di tipo rifermento a oggetto restitusce true soltanto se le due celle puntano allo **stesso** oggetto (o se entrambe contengono il valore speciale null); in tutti gli altri casi restituiscono false.

Insomma: l'operatore == fra oggetti ne verifica l'identità, non l'uguaglianza!

Oggetti: uguaglianza (II)

Esempio:

```
Conto mioConto = new Conto(400);
Conto tuoConto = new Conto(400);
Conto suoConto = mioConto;
System.out.println(mioConto == tuoConto); // risultato: false
System.out.println(mioConto == suoConto); // risultato: true
```

Per testare l'uguaglianza fra due oggetti di una classe occorre definire nella classe un metodo convenzionalmente chiamato equals (cioè "è uguale a", 3ª persona sing. del verbo to equal); tale metodo deve prendere come argomento un oggetto e confrontarne i contenuti con quelli dell'oggetto this. Ad esempio, se nella classe Conto definiamo correttamente un tale metodo, avremo poi:

```
System.out.println(mioConto.equals(tuoConto)); //
```
risultato: true

Variabili locali e variabili (o campi) statici

- variabili locali:
  - sono le variabili dichiarate all'interno di un metodo (anche del metodo main); esse esistono, al pari dei parametri formali, solo durante l'attivazione del metodo stesso, e sono delle celle nel suo record di attivazione (o frame). Al di fuori del corpo del metodo esse non sono né visibili né definite (es. visti a lezione).
- campi statici (simili alle variabili globali in altri linguaggi):
  - sono le variabili dichiarate fuori dei metodi, ma all'interno di una classe (**ricorda**: in Java tutto deve essere all'interno di una classe), e qualificate `static`; esse esistono durante tutta l'esecuzione del programma, e sono visibili dall'interno dei metodi della stessa classe (es. `numContiCreati`); a certe condizioni (che vedrete piú avanti) esse sono visibili anche dall'interno di metodi di altre classi.

Java: campi e metodi statici

Adesso possiamo capire meglio come funzionano (almeno in parte…) alcuni metodi "magici" utilizzati sin qui:

```
Math.round(…)
```

1) invocazione del metodo **`round(…)`** …

2) sulla classe **`Math`**

```
public static void main(String[] args){
```

`main` è un metodo di classe! (infatti lo dichiariamo *static*)

```
System.out.println(output);
```

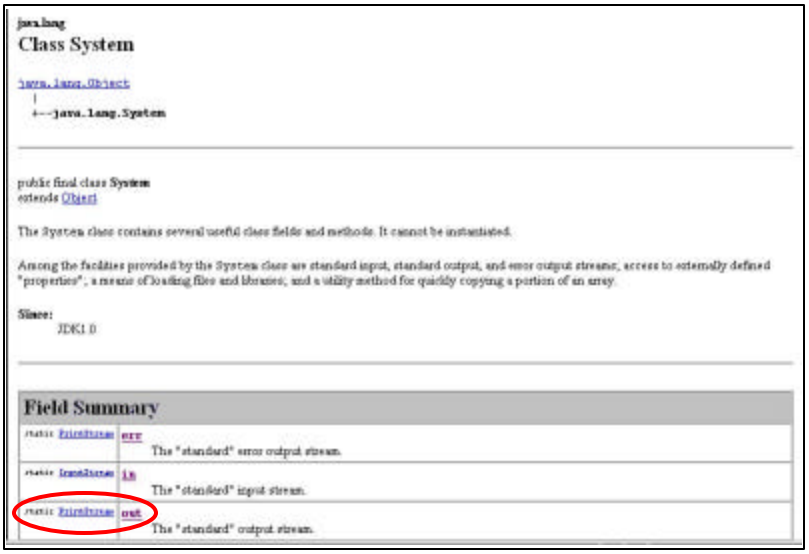
1) invocazione del metodo **`println(...)`** …

2) sulla variabile statica (di classe) **`out`** …

3) sulla classe **`System`**

Infatti…

Java: campi e metodi statici



java.lang
Class System

java.lang.Object
|
+--java.lang.System

public final class System
extends Object

The System class contains several useful class fields and methods. It cannot be instantiated.

Among the facilities provided by the System class are standard input, standard output, and error output streams; access to externally defined "properties"; a means of loading files and libraries; and a utility method for quickly copying a portion of an array.

Since:
JDK1.0

Field Summary

static PrintStream	err The "standard" error output stream.
static InputStream	in The "standard" input stream.
static PrintStream	out The "standard" output stream.

Altre particolarità dei costruttori: costruttore di default

Se in una classe non viene definito alcun costruttore, l'interprete java definisce automaticamente un costruttore di default privo di parametri e con corpo vuoto.

Ad esempio, nella seguente definizione di classe:

```
class Punto {
  private int x;
  private int y;

  int getX {return x;}
  int getY {return y;}
  void sposta(int dx, int dy) {x += dx; y += dy;}
}
```

è come se ci fosse il costruttore

```
  Punto() {}
```

Grazie ad esso, è quindi possibile creare un oggetto della classe:

```
Punto mioPunto = new Punto();
```

I campi x e y non vengono inizializzati dal costruttore (che non fa nulla); essi tuttavia vengono automaticamente inizializzati a zero dall'interprete java.

Costruttori espliciti e costruttore di default

Se invece in una classe viene definito almeno un costruttore esplicito (non importa se con parametri o senza), il costruttore di default non viene più automaticamente inserito. Ad esempio, data la definizione di classe:

```
class Punto {
  private int x;
  private int y;

  Punto(int x, int y) {this.x = x; this.y = y;}

  int getX {return x;}
  int getY {return y;}
}
```

non è più possbile creare un oggetto con l'istruzione:

```
Punto mioPunto = new Punto();
```

come nell'esempio precedente.

Perché ciò sia ancora possibile, il programmatore deve definire esplicitamente nella classe un costruttore senza argomenti, ad esempio:

```
  Punto() {}
```

Il parametro del **main**

```
public static void main(String[] args)
```

Anche quando viene lanciato il comando (*)

```
java NomeFile
```

viene invocato un metodo, il **main** (che necessariamente deve essere contenuto nella classe **NomeFile**).

È possibile passare al **main** dei dati, che verranno memorizzati nell'array di stringhe **args** che è il suo parametro di tipo **String[]** e che possono essere usati nel suo body.

(*): naturalmente dopo aver compilato con:

```
javac NomeFile.java
```


Java: argomenti di main

```
public class TestArgs {
  public static void main( String[] args ){
    if (args.length == 0) {
      System.out.println("Non mi hai dato
  argomenti...");
    }
    for (int i=0; i<args.length; i++) {
      System.out.println(args[i]);
    }
  }
}
```

se l'array *args* è vuoto...

array di stringhe

ciclo sull'array *args* e per ogni elemento **args[i]** lo stampo

L'array **args** viene riempito con le **stringhe scritte dall'utente** al momento dell'invocazione dell'interprete sulla classe; per es:

```
> java TestArgs pippo pluto ...
⇒ args[0] = "pippo"
   args[1] = "pluto"
   ...
```

Attenzione!!! Se lo lanciate da *TextPad* non potete passargli argomenti!!!

`public` e `private`

- Le due parole-chiave **`public`** e **`private`** fanno parte dei *modficatori* (*modifier*) di Java
- Anche **`static`** è un modificatore
- Ce ne sono altri, che incontrerete durante il Corso di Studi
- Ogni cosa dichiarata **`public`** è accessibile da qualsiasi programma, interno od esterno
- Ogni cosa dichiarata **`private`** è accessibile soltanto nella classe che contiene la dichiarazione.

NB Le due definizioni suddette sono un po' imprecise, ma sufficienti al momento

- Di solito: campi **`private`** + metodi accessori **`public`** (set e get)

Uso dei modificatori: esempio (I)

```
public class Studente {
// campi
 private String nome;
 private String cognome;
 private String dataNascita;
 private Integer matricola;
// costruttori
 public Studente() {
  nome = "";
  cognome = "";
  dataNascita = "";
  matricola = new Integer(0);
 }
 public Studente(String _nome, String _cognome,
    String _dataNascita, Integer _matricola) {
  nome = _nome;
  cognome = _cognome;
  dataNascita = _dataNascita;
  matricola = _matricola;
 }
  …
```

Uso dei modificatori: esempio (II)

```
  …
// metodi set/get per i campi
  public void setNome(String n){
      nome = n;
  }
  public String getNome(){
      return nome;
  }
  public void setCognome(String c){
      cognome = c;
  }
  public String getCognome(){
      return cognome;
  }
  public void setDataNascita(String dn){
      dataNascita = dn;
  }
  public String getDataNascita(){
      return dataNascita;
  }
  public void setMatricola(Integer m){
      matricola = m;
  }
  public Integer getMatricola(){
      return matricola;
  }
}
```


Come definire una classe in modo che di essa non si possano creare oggetti

Se una classe è costituita solo di metodi e campi statici, quindi priva di campi e metodi di istanza, come la classe ArrayUtil sviluppata nelle esercitazioni, oppure come la classe predefinita Math, non ha senso creare oggetti di una tale classe, che non contengono alcun dato.

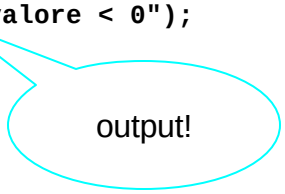
Per fare in modo che il compilatore vieti la creazione di tali oggetti, basta definire nella classe un costruttore (e uno solo) e dichiararlo privato. Ad esempio:

```
public class ArrayUtil {
  static Scanner input = new Scanner(System.in);

  private ArrayUtil() {}

  static ...
}
```

In tal modo il costruttore pubblico di default non viene inserito dall'interprete java, ma d'altra parte il costruttore definito dal programmatore – essendo privato – non può essere usato fuori della classe. L' espressione `new ArrayUtil()` genera dunque (fuori della classe stessa) un errore di compilazione.

Qualche osservazione sull'input/output…

```
public class Conto {
  private static int numContiCreati;
  private double saldo;
  private int numOperazioni;
  ...
  public void deposita(double importo) {
    if(importo < 0)
      System.out.println("errore: valore < 0");
    else {
      saldo = saldo + importo;
      numOperazioni++;
    }
  }
  ...
}
```

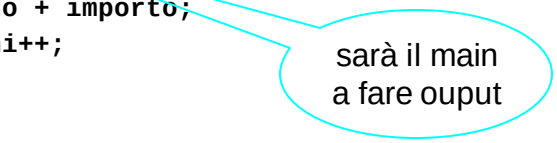


Qualche osservazione sull'input/output…

- Attenzione: in generale, l'input/output deve essere tenuto distinto dall'algoritmo vero e proprio. Una procedura che realizza un algoritmo, o che comunque effettua un'elaborazione di dati, di solito è bene che non faccia operazioni di input/output, ma soltanto passaggio parametri e restituzione di risultato.
- L'input/output deve essere effettuato, nel caso dei primi semplici esercizi di programmazione, dal main. Ovvero, **`deposita`** dovrebbe restituire un **`boolean`**, che viene testato dal main… (vedremo in lab.)
- Nel caso di programmi complesse l'input/output dovrà essere realizzato da procedure appositamente definite (metodi di input/output). Nelle applicazioni reali tali procedure devono di solito realizzare una vera e propria interfaccia "a finestre" con l'utente.

Qualche osservazione sull'input/output…

```
public class Conto {
  private static int numContiCreati;
  private double saldo;
  private int numOperazioni;
  ...
  public boolean deposita(double importo) {
    if(importo < 0)
      return false;
    else {
      saldo = saldo + importo;
      numOperazioni++;
      return true;
    }
  }
  ...
}
```



Norme di stile

- È buona norma stilistica, dopo aver scritto tutte le definizioni dei campi in righe successive (un campo per riga), lasciare una riga bianca prima del primo metodo, e poi separare un metodo dall'altro per mezzo di una riga bianca.
- È buona norma stilistica, anche se non è obbligatorio, mettere i campi statici prima dei campi di oggetto (cioè non statici), e analogamente mettere i metodi statici prima degli altri metodi.
- Infine, dopo aver scritto tutte le definizioni di metodi, naturalmente si chiude il corpo della classe con una *chiusa graffa*.

Norme di stile

Alcuni testi, nello scrivere gli esempi di classi, usano la brutta convenzione, tra l'altro contraria alle convenzioni ufficiali dei programmatori Java, di mettere i campi al fondo della classe, dopo i metodi; ad es.:

```
class Conto {
  ...
  void deposita(double importo) {
  ...
  }

  boolean preleva(double importo) {
  ...
  }

  ...
  private double saldo;
  private int numOperazioni;
}
```

da non fare!!

ATTENZIONE!

SEGUIRE LA CONVENZIONE UFFICIALE JAVA !

(vedi: http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html)

L'ordine in cui devono comparire le varie parti è il seguente:

1. campi statici (prima i pubblici, poi i privati)
2. campi di oggetto (o di istanza) (prima i pubblici, poi i privati)
3. costruttori
4. metodi; essi devono essere raggruppati per funzionalità, e non in base al fatto che siano pubblici o privati. Ad es., se un metodo pubblico utilizza un metodo ausiliario privato, i due metodi andranno messi vicini l

Lo scopo di tutto questo è quello di facilitare la lettura e la comprensione del programma!!!!!!!!!!

Altre convenzioni Java

• niente spazi fra nome del metodo e parentesi aperta:

`void deposita(double...)` NON **`void deposita (double...)`**

• saltare una riga per separare un metodo dall'altro

• stile di indentazione raccomandato:

```
 void deposita(double...) {
    istruz;
    ...
 }
```

nota: mettere uno spazio fra ")" e "{"

Fra le poche convenzioni Java che non seguiremo vi è quella del rientro impostato a 4: un rientro di 2 o al massimo di 3 sembra migliore. In ogni caso rientri superiori a 4 sono severamente vietati!

Differenze rispetto alle convenzioni

Altre convenzioni che sono adottate spesso nei nostri lucidi e programmi e che sono leggermente differenti da quelle ufficiali Java sono le seguenti:

```
if(BoolExpr) {
  istruzThen1;
  ...
}
else {
  istruzElse1;
  ...
}
```

```
if(BoolExpr1) {
  istruz1_1;
  ...
}
else if(BoolExpr2) {
  istruz2_1;
  ...
}
else if(BoolExpr3) {
  istruz3_1;
  ...
}
...
else {
  istruzN_1;
  ...
}
```

Documentare il codice

- Il codice va documentato: per capirlo, *debuggarlo* e riusarlo (anche da parte del programmatore che l'ha scritto…)
- Java permette di documentare il codice tramite commenti:
  - `// questo sta su una sola riga`
  - `/* questo commento e` su piu'`
    `righe`
    `*/`
- Il secondo formato è usato per documentare i metodi, ad esempio:

```
/* Questo metodo cancella il numero di matricola
   indicato (parametro) dalla lista degli iscritti
   (variabile d'istanza 'listaStudenti')
*/
 public void cancellaPrenotaz(Integer matricola) {
   ...
 }
```

- Utilizzeremo anche JavaDoc, che genera gli APIs per il *nostro* codice

Qualche nota tecnica sulla heap (I)

- E` una sequenza di celle di memoria.
- Quando un metodo richiede la creazione di un nuovo oggetto (*new*), java deve trovare dentro questa sequenza lo spazio libero necessario per l'oggetto, ad es.:
  - vettori: lo spazio necessario deve essere costituito da una sequenza di celle adiacenti
  - vettore di 5 interi: java dovrà trovare una quantità di spazio libero (in numero di bytes) sufficiente per memorizzare 5 interi (40 bytes)
- Viene dato al metodo che ha richiesto la creazione qualcosa che gli permetta di accedere allo spazio riservato; un *riferimento* all'oggetto.
- Il termine *riferimento* è piuttosto astratto: questo è fatto di proposito per evitare di legarsi ad una particolare *implementazione*.

Qualche nota tecnica sulla heap (II)

- Se può aiutare, si può pensare al riferimento ad un vettore come all'indirizzo della prima cella di memoria riservata per il vettore, o all'indirizzo del primo campo di un oggetto (Java non dice come deve essere realizzato un riferimento, dice solo come funziona)
- Sebbene ogni dato elementare (come *int*) abbia una dimensione predefinita, che è diversa a seconda del tipo (ad esempio, un *int* occupa 4 bytes, un *char* 2 bytes, un intero *long* 8 bytes, ecc.), evitiamo di considerare questo problema, parlando, in generale di "celle" di memoria. Questo è impreciso, ma semplifica notevolmente la presentazione.

Qualche nota tecnica sulla heap (III)

- Il riferimento viene inserito in una locazione di memoria, di cui il metodo conosce la posizione.
- Il metodo usa un'espressione come **`mioVettore[3]:`** dove viene in realtà messo il riferimento?
  - Esso viene posto all'interno dello stack.

Qualche nota tecnica sullo stack (I)

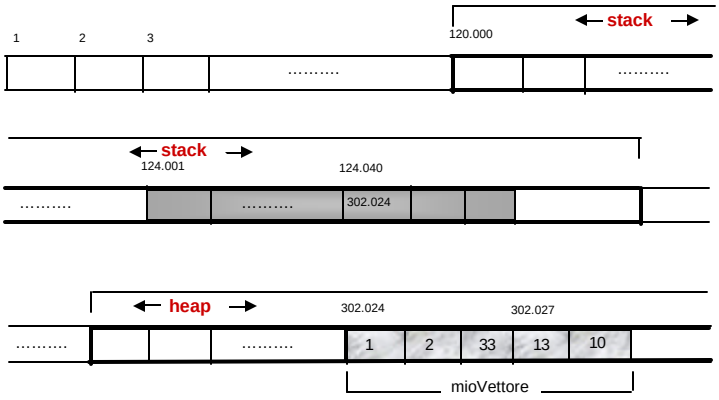
- Ogni volta che inizia l'esecuzione di un metodo, sullo stack viene riservata una quantità di spazio (*record di attivazione*, o blocco) pari a quella (determinata dal compilatore) necessaria per i parametri e le variabili locali del metodo.
- Quando il metodo inizia l'esecuzione, è nota la posizione iniziale del blocco sullo stack ad esso associato (ad esempio, il blocco inizierà dalla cella 124.001) e, per ogni parametro o variabile del metodo, è nota la sua posizione all'interno del blocco

Qualche nota tecnica sullo stack (II): esempio

Per leggere il valore di **`mioVettore[3]:`**

- si prende la posizione del blocco sullo stack associato al metodo che si sta eseguendo (124.001)
- ad essa si somma la posizione del riferimento all'interno del blocco meno 1 (124.001+40-1)
- in questa cella (124.040) ci sarà il riferimento al vettore (ad es. 302.024 se il riferimento è costituito da un indirizzo); a questo punto
  - se ogni intero occupa una cella, e poiché le celle del vettore sono adiacenti, è immediato determinare la posizione della cella voluta: 302.024+3 = 302.027. Si noti che qui non è necessario togliere 1, proprio perché la numerazione degli elementi del vettore parte da 0.

Qualche nota tecnica sullo stack (III): esempio



Qualche nota tecnica sullo stack (III): esempio