Linguaggi

Corso M-Z - Laurea in Ingegneria Informatica
A.A. 2007-2008

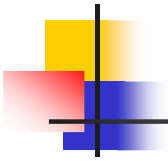
Alessandro Longheu
http://www.diit.unict.it/users/alongheu
alessandro.longheu@diit.unict.it

- lezione 04 -

Classi ed Oggetti in Java

Generalità

- Una classe Java modella l'insieme di tutti gli oggetti che condividono le stesse caratteristiche.
- Una classe Java è una entità **sintatticamente simile alle struct** del C, però non contiene solo i dati ma anche le funzioni che operano su di essi e ne specifica il livello di protezione rendendoli visibili o meno all'esterno della classe stessa
- Una classe Java è una entità dotata di una "doppia natura":
  - è un **componente software**, che in quanto tale può possedere propri dati e operazioni, opportunamente protetti
  - contiene anche la definizione di un **ADT**, cioè uno "stampo" per creare nuovi oggetti, anch'essi dotati di idonei meccanismi di protezione.



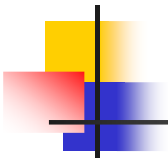
Definizione di una classe

- **La sintassi per definire una classe è:**

<modificatore> class <nomeclasse>
[extends <nomeclasse>]
[implements <nomeinterfaccia>]
[, implements <nomeinterfaccia>] *{*
<membri>
}

Esempio:

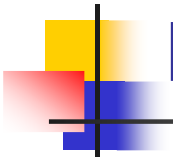
```
public class Body extends Star     {
   private String name;
   public String getName() { return name; }
   public setName(String n) { name = n; }
                                       }
```

Definizione di una classe

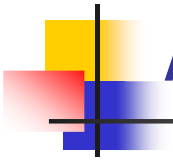
- La dichiarazione di una classe equivale alla creazione di un ADT, il cui nome coincide con il nome della classe
- È quindi possibile creare un **riferimento ad un oggetto** di tipo *Body* con l'istruzione <nome_tipo> <nome_variabile>, ad esempio:

  Body x; (analogo di int x per il tipo integer)
- **l'istruzione NON crea tuttavia ancora un oggetto**: non è ancora allocata memoria per contenerne dati e quant'altro; piuttosto crea un riferimento (il cui nome è x) per oggetti di tipo Body (un puntatore, inizialmente vuoto); la creazione effettiva dell'oggetto richiede altre istruzioni



Modificatori di Classe

- Il **modificatore di classe** definisce particolari proprietà della classe.
- Ne esistono diversi:
  - **Annotazioni**
  - **Public**
  - **Abstract**
  - **Final**
  - **Strict Floating Point**
- Non è permesso l'utilizzo contemporaneo di abstract e final
- L'ordine di utilizzo dei modificatori è irrilevante



Modificatori di Classe
Annotazioni

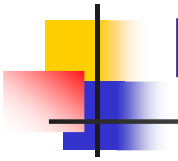
- Il codice solitamente è accompagnato da diverse **informazioni documentali**, come nome autore, copyright, numero versione ecc.
- Java offre un supporto per formalizzare queste informazioni, introducendo i **tipi annotazione**, con il fine di uniformare la tipologia di informazioni documentali e permetterne **l'elaborazione automatica**
- Esempio di **definizione di tipo annotazione**:

```
@interface ClassInfo {
   String createdBy();
   int revision();
}
```

- **L'applicazione del tipo annotazione** prima definito alla classe "prova" potrebbe essere la seguente:

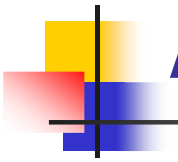
```
@ClassInfo ( createdBy = "Pinco Pallino", revision = 1.0 )
public class prova {
   …
}
```

Modificatori di Classe Public



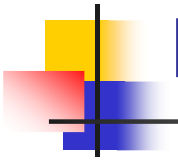
- Il modificatore **public** permette alla classe di essere accessibile da qualunque altra; se omesso, la classe risulterebbe accessibile solo dalle classi dello stesso package ("libreria")
- Molti strumenti di sviluppo richiedono che una classe public sia dichiarata all'interno di un file con lo stesso nome della classe, imponendo che esista una ed una sola classe pubblica in un dato file sorgente

Modificatori di Classe Abstract



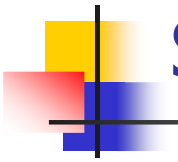
- **Abstract** indica che la classe è incompleta, ovvero che al suo interno viene fornita l'implementazione di alcuni ma non di tutti i suoi metodi
- poiché per istanziare (=usare) oggetti di una classe questa deve implementare tutti i suoi metodi (il comportamento deve essere definito), non è possibile istanziare oggetti di una classe astratta
- l'implementazione dei metodi "vuoti" di una classe astratta è fornita dalle sue sottoclassi (quelle cioè che ereditano dalla classe astratta); questo accade quando solo parte del comportamento è comune, e per alcuni metodi non ha invece senso dare una sola implementazione, che viene data nelle sottoclassi, probabilmente differenziandola per ogni sottoclasse

Modificatori di Classe Final

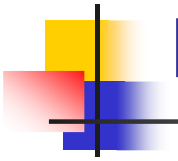


- Il modificatore **final** non permette l'estensione della classe, ovvero non può esistere un'altra classe che erediti da una final
- questo solitamente avviene quando, in fase di progetto, si considera definitiva e completa la definizione della classe, o quando per motivi di sicurezza si vuole la garanzia che nessuna delle sottoclassi possa cambiare qualche metodo tramite overriding
- final e abstract non possono essere utilizzati insieme

Modificatori di Classe Strict Floating Point

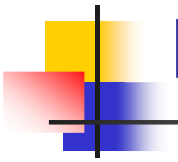


- Il modificatore **strictfp** (strict floating point) stabilisce la modalità di lavoro in virgola mobile
  - in modalità stretta, si applicano delle regole vincolanti che garantiscono risultati dei calcoli in virgola mobile sempre uguali (ripetibili) e identici su tutte le implementazioni della JVM
  - in modalità non stretta, i calcoli possono essere eseguiti con rappresentazioni in virgola mobile che potrebbero anche dare luogo a overflow o underflow, e che potrebbero non essere ripetibili o identici su diverse JVM; la modalità tuttavia potrebbe risultare più efficiente perché permette l'uso di rappresentazioni fornite dall'hardware della macchina (maggiore velocità, minore compatibilità)



Clausule Extends e Implements

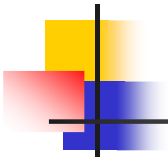
- Il **nome della classe** può essere una qualunque sequenza di caratteri ammessa da Java; solitamente è meglio usare nomi descrittivi (anche lunghi)
- la parola chiave **extends**, seguita dal nome di una classe X, indica che la classe che si sta definendo deve ereditare da (è una sottoclasse di) X
- la parola chiave **implements**, seguita dal nome di una interfaccia X (una classe con elenco di metodi senza alcuna implementazione), indica che la classe che si sta definendo fornirà le implementazioni per i metodi (vuoti) dell'interfaccia X; una data classe può implementare i metodi di un qualsiasi numero di interfacce



Membri di una classe

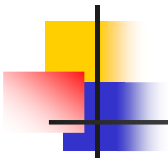
- I membri di una classe possono essere di tre tipi:
  - **attributi**
  - **metodi**
  - classi o interfacce **innestate** (all'interno della dichiarazione della classe)

Membri di una classe
Attributi

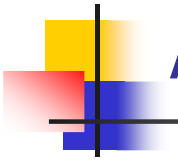


- Un **attributo** è un campo che contiene dati associati alla classe o ad uno specifico oggetto di quella classe; l'insieme dei valori degli attributi rappresenta lo **stato** di quella classe o di quell'oggetto
- la dichiarazione di un attributo segue la sintassi:
  <modificatore> <nometipo> <nomeattributo> <inizializzazione>
- il **modificatore** può essere:
  - **annotazione** (discorso analogo all'uso che se ne fa nelle classi)
  - **modificatore di accesso**
  - **static**
  - **final**
  - **transient**
  - **volatile**
- il **nome tipo** specifica il tipo della variabile (tipi base o riferimenti)
- il **nome attributo** è l'identificatore
- l'**inizializzazione** consiste in "= <valore>", valore che deve essere compatibile con il tipo scelto

Membri di una classe
Attributi – Modificatori di accesso

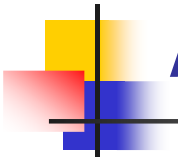


- I modificatori di accesso servono a stabilire la visibilità che si vuole conferire all'attributo, realizzando il system hiding
- tutti i membri di una classe sono visibili all'interno della classe stessa
- le altre classi possono o meno vedere un dato membro, in base al modificatore scelto; quattro sono i tipi disponibili:
  - **public**, che rende il membro accessibile in tutti i punti in cui lo è la classe di appartenenza
  - **protected**, che rende il membro accessibile, oltre che alla classe di appartenenza, alle sottoclassi della classe di appartenenza ed alle classi dello stesso package della classe di appartenenza
  - **private** rende il membro accessibile solo alla classe di appartenenza
  - in caso di assenza di modificatore, il membro è accessibile, oltre che alla classe di appartenenza, alle classi del suo stesso **package**
- i modificatori di accesso dei membri sono comunque subordinati alla visibilità della classe in cui si trovano, ad esempio se una classe non è public, le classi esterne non potranno comunque vedere un attributo dichiarato public



Membri di una classe
Attributi – Modificatori di accesso

- Esempi:
  - public int x=0, y=12;
  - private char a='b';
  - protected boolean z=false;
  - SomeClass x
  - public someOtherClass y;



Membri di una classe
Attributi static

- static viene utilizzato quando si desidera che l'attributo venga implementato in **unica istanza**, che sarà condivisa fra tutti gli oggetti di una stessa classe; un tale attributo viene detto campo statico o attributo della classe
- solitamente si utilizza un campo statico quando l'informazione che deve contenere non è associata ad ogni oggetto che sarà poi istanziato, ma piuttosto è **relativa a tutto l'insieme di tali istanze**, ad esempio un campo che deve totalizzare il numero di oggetti sinora creati

Membri di una classe
Attributi static

- per **referenziare un attributo statico** dall'esterno, è necessario utilizzare il nome della classe anteposto al nome del campo in notazione puntata, esempio

```
class Automobile {
    public static int numero_ruote=4;
}
class Esterna {
...
    int con_routa_di_scorta= Automobile.numero_ruote+ 1;
    Automobile x=new Automobile();
    System.out.println("ecco il numero"+x.numero_ruote);
}
```

  E' possibile anteporre al campo anche il nome di un qualsiasi oggetto di tipo Automobile (x nell'esempio sopra), il compilatore comunque farà riferimento alla variabile di classe; è però deprecabile utilizzare il nome di un oggetto (sembrerebbe che *numero_ruote* non sia statica)

Membri di una classe
Attributi static

- se gli accessi a membri statici sono tanti, il codice potrebbe diventare poco leggibile; è in tal caso possibile effettuare **l'importazione statica**, che consente di evitare di anteporre il nome della classe; la sintassi da utilizzare è

  import static <nomeclasse>.<nomemembro>;

- il comando posto all'inizio del sorgente java, consente in tutto il codice restante di utilizzare direttamente il solo nome del membro
- se nel comando di import si sostituisce il nome membro con *, si tratta di **importazione statica su richiesta**, e obbliga il compilatore, in caso nel codice si usi un membro non riconosciuto, ad esaminare la classe importata per verificare se essa contiene il membro statico cercato ed adottarlo in caso positivo

Membri di una classe
Attributi static

- **Esempio**:

```
class Automobile {
   public static int numero_ruote=4;
   public static boolean ESP=true;
}
import static Automobile.numero_ruote;
class Esterna {
...
   int totale_ruote= numero_ruote + 1;
   if Automobile.ESP { … }
...
}
```

- l'importazione del solo *numero_ruote* rende obbligatorio usare *Automobile* per *ESP*

Membri di una classe
Attributi static

- **Esempio**:

```
class Automobile {
   public static int numero_ruote=4;
   public static boolean ESP=true;
}

import static Automobile.*;
class Esterna {
...
   int totale_ruote= numero_ruote + 1;
   if ESP { … }
...
}
```

  - importazione statica su richiesta

Membri di una classe
Attributi static

- **Esempio**:

```
class Automobile {
   public static int numero_ruote=4;
   public static boolean ESP=true;
}
import static Automobile.*;
class Esterna {
... int numero_ruote=5;
   int k= numero_ruote + 1;
...
}
```

  - il *numero_ruote* che verrà usato è la variabile locale della classe *Esterna* (precedenza)

Membri di una classe
Attributi final

- La clausula **final** dichiara che il campo non può essere più modificato dopo la sua inizializzazione
- se il campo è di tipo base, viene detto **costante**
- il valore di un campo final potrebbe non essere noto al momento dell'inizializzazione; in tal caso viene denominato **blank final**;
- il valore di un campo blank final, comunque immutabile una volta fissato, deve essere fornito quando si inizializza la classe (se il membro è anche statico) o viene creato un oggetto di quella classe (se non è statico); tale inizializzazione viene effettuata dal metodo costruttore od eventualmente in un blocco di inizializzazione; il compilatore effettua controlli sulla fornitura di un valore, segnalandone la mancanza in fase di compilazione
- se il valore del campo final non è noto neanche al momento della creazione di un oggetto, non può essere final, anche se per il programmatore il suo valore sarà immutabile

Membri di una classe
Attributi transient e volatile

- la **serializzazione** è la trasformazione di un oggetto in un flusso di byte, operata per salvarne e/o trasmetterne il contenuto ad altri oggetti (invocazione remota di oggetti, o RMI)
- La clausula **transient** dichiara che il campo non deve essere serializzato, ad esempio perché non è fondamentale salvarne il valore o perché può essere ricalcolato a partire da altri campi serializzati
- quando invece più thread (parti di un processo che condividono risorse) devono accedere ad una stessa variabile, si può utilizzare la clausula **volatile**, che indica che qualsiasi scrittura effettuata su tale variabile (da un thread) deve sincronizzarsi con le letture successive su tale variabile (operate da altri thread), in modo da garantire che venga letto sempre il valore più recentemente scritto
- un campo volatile non può essere final

Membri di una classe
Attributi – Inizializzazione

- **l'inizializzazione** di un campo può essere effettuata all'atto della dichiarazione, esempio:

  double zero = 0.0;
  double sum = 4.5 +3.7;
  *double zero = sum*2;*
  *double someVal = sum + 2*Math.sqrt(zero)*
- Esistono comunque valori di **default automatici**:
  - boolean false
  - char \u0000
  - byte, short, int, long 0
  - float, double +0.0
  - riferimento null

Membri di una classe
Metodi – Dichiarazione

- Un **metodo** di una classe è un insieme di istruzioni in genere con il compito di manipolare lo stato degli oggetti della classe assegnata
- la dichiarazione di un metodo segue la sintassi:

  <modificatore> <tipiparametro>
  <tiporestituito> <nomemetodo> (<listaparametri>) <throws>
  {<corpo>}
- il **modificatore** può essere:
  - **annotazione** (discorso analogo a classi ed attributi)
  - **modificatore di accesso** (public, private, protected o package)
  - **abstract**
  - **static**
  - **final**
  - **synchonized**
  - **native**
  - **strict floating point**

A. Longheu – Linguaggi M-Z – Ing. Inf. 2007-2008

Membri di una classe
Metodi – Dichiarazione

- **annotazione** e **modificatore di accesso** sono gli stessi degli attributi
- **abstract** è relativo ad un metodo senza implementazione, usato in una classe astratta;
  - tale metodo è senza body, ossia alla fine della eventuale clausula throws termina con ";"
  - verrà comunque implementato in qualcuna delle sottoclassi
- un metodo **static** è semanticamente equivalente ad un attributo static, ossia è relativo all'intera classe e non ad ogni istanza; un siffatto metodo può solo usare attributi static (potrebbe comunque non usare affatto attributi) o altri metodi static della stessa classe
- un metodo **final** non potrà essere ridefinito all'interno delle sottoclassi (l'overriding non è ammesso)
- la parola chiave **synchronized** viene utilizzata quando il metodo è coinvolto in una race condition con altri metodi (thread concorrenti), ed occorre gestire l'accesso alla risorsa condivisa

Membri di una classe
Metodi – Dichiarazione

- un metodo è **native** quando la sua implementazione è fornita in un altro linguaggio di programmazione (ad esempio C o C++)
  - tali metodi si utilizzano quando il codice esiste già e/o quando occorre gestire direttamente particolari dispositivi hardware
  - nella dichiarazione di un metodo nativo il body viene sostituito da ";" come nei metodi abstract
  - i metodi nativi non possono essere né astratti né strictfp
  - l'uso di metodi nativi lede la portabilità e la sicurezza del codice, quindi il loro uso è vietato se il codice è scaricabile dalla rete (applet)
  - l'implementazione di un metodo nativo viene collegata al codice Java tramite opportune API; nel caso Java, esiste la Java Native Interface (JNI) per il C
- **strictfp** ha significato analogo all'uso che se ne fa nella classe; se una classe è strictfp, tutti i suoi metodi lo sono anche senza dichiarazione esplicita di ognuno
- un metodo astratto non può essere static, final, synchronized, native o strictfp

Membri di una classe
Metodi – Dichiarazione

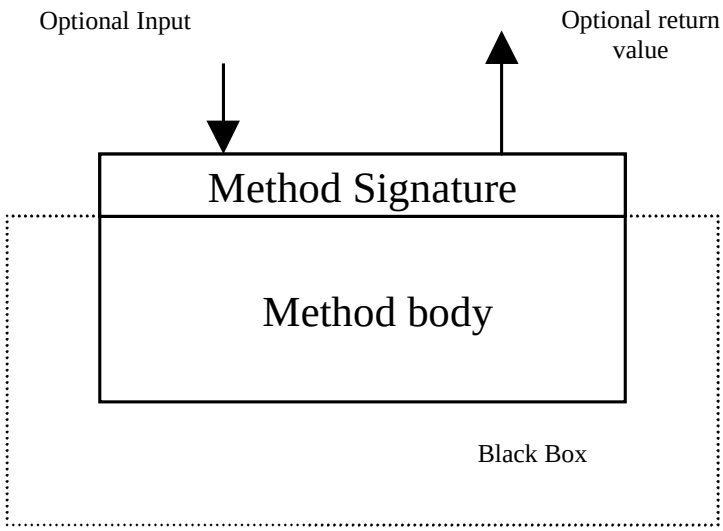
- I tipi parametro sono utilizzati per dichiarare **metodi generici**
- il **tipo restituito** da un metodo può essere sia primitivo (int, byte…) che un riferimento (un oggetto)
  - se il metodo non deve restituire nulla, va indicato *void* come in C
  - se il metodo deve restituire più di un risultato, allora il tipo restituito può essere un array, un oggetto (nei suoi attributi si porranno tutti i risultati desiderati), o infine il metodo potrebbe utilizzare qualcuno dei parametri di ingresso per scrivervi opportunamente i propri risultati
- **Nome metodo** e **lista parametri** sono la **signature** del metodo

Membri di una classe
Metodi – Dichiarazione

- la **lista dei parametri** è un elenco di indicatori <tipo> <nome>, ossia tipo del parametro e relativo nome (parametro formale)
  - la lista può essere anche vuota (ma la coppia di parentesi "()" è comunque richiesta)
  - l'ultimo parametro può essere dichiarato come sequenza non definita di parametri dello stesso tipo, ad esempio
    public SomeClass method(int x, OtherClass y, String… mess)
  - se è presente la sequenza, si possono passare parametri in numero variabile, motivo per cui il metodo che li prevede viene detto di tipo *varargs*
  - la sequenza viene gestita dal compilatore come un array
- la clausula **throws** serve per indicare che il metodo potrebbe sollevare eccezioni
- il **corpo** contiene le istruzioni che implementano il metodo

Membri di una classe
Metodi – Dichiarazione

Il corpo di un metodo è come una "black box" che contiene i dettagli dell'implementazione del metodo.



Membri di una classe
Metodi – Esempio

```
public static int max(int num1, int num2)
 {
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```


Classi e membri
confronto modificatori

Caratteristica	Classe	Attributo	Metodo
Annotazioni	X	X	X
Public	X	X	X
Protected		X	X
Private		X	X
Static		X	X
Abstract	X		X
Final	X	X	X
Strictfp	X		X
Transient		X	
Volatile		X	
Synchronized			X
Native			X

Membri di una classe
Metodi – Invocazione

- Per invocare un metodo la forma è quella chiamata "**dot notation**" o notazione puntata

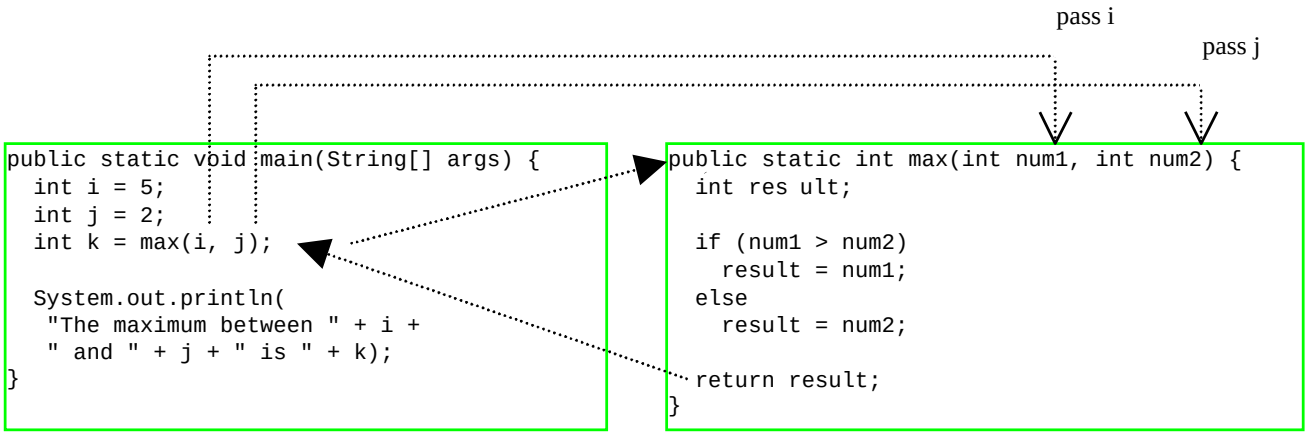
  nomeoggetto.nomemetodo(<lista_parametri_attuali>)
- se il metodo è un metodo di una istanza

  nomeclasse.nomemetodo(<lista_parametri_attuali>)
- se il metodo è un metodo della classe (statico)

Membri di una classe
Metodi – Invocazione

- l'invocazione di un metodo segue regole simili alla chiamata di una funzione in C:
  - il chiamante si interrompe, passa i parametri attuali, che prendono il posto di quelli formali, e l'esecuzione del metodo ha inizio
  - a differenza del C, i parametri attuali possono, entro certi limiti, avere un tipo diverso da quello dichiarato nel parametro formale, ad esempio un byte si può passare ad un parametro dichiarato intero
  - l'esecuzione si conclude, e il controllo ritorna al chiamante se si incontra un return (se il metodo deve restituire qualcosa), si raggiunge la fine (se il metodo non doveva tornare nulla) o si solleva un'eccezione non intercettata dal metodo

Membri di una classe
Metodi – Invocazione



Membri di una classe
Metodi – Invocazione

L'istruzione return è **sempre richiesta** per i metodi di tipi diverso da void:

```
public static int xMethod(int n) {
  if (n > 0) return 1;
  else if (n == 0) return 0;
  else if (n < 0) return –1;
}
```

Il metodo, benchè logicalmente corretto, provoca un errore di compilazione poichè il compilatore Java pensa che il metodo possa non tornare un valore. Occorre quindi rimuovere l'ultimo if.

Membri di una classe
Metodi – Invocazione

- Come il C, Java passa i parametri **per valore**. Finché parliamo di tipi primitivi non ci sono particolarità da notare, ma passare per valore un riferimento significa passare per riferimento l'oggetto puntato (che quindi potrebbe essere modificato).
- Un parametro di tipo primitivo viene copiato, e la funzione riceve la copia
- un riferimento viene pure copiato, la funzione riceve la copia, ma con ciò accede all'oggetto originale.

Membri di una classe
Metodi – Invocazione

- **Esempio di passaggio di un riferimento:**

```
class Prova {
    public int x;
    public static void main(String args[]) {
            Prova obj=new Prova();          obj.x=33;
            Prova pippo=new Prova();        pippo.x=334;
            System.out.println("val is "+obj.x);
            PassRef(obj);
            System.out.println("val is "+obj.x);
    }
    public static void PassRef(Prova xxx) {
            xxx.x=10;
            xxx=null;
    }
}
```

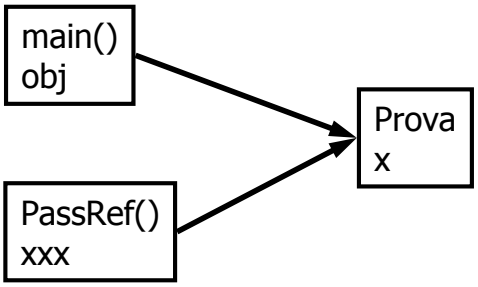
Membri di una classe
Metodi – Invocazione

- **Esempio di passaggio di un riferimento:**
- L'output prodotto dall'esempio è:

  val is 33

  val is 10

  Nonostante sia stato posto a null xxx, esso è solo un secondo puntatore, copia del parametro inviato, per cui obj (il primo puntatore) è rimasto inalterato ed accede ancora al valore modificato dal metodo

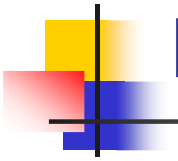


- I parametri possono anche essere dichiarati final, impedendo la loro modifica;
- nell'esempio precedente, xxx=null sarebbe stato proibito dal compilatore, ma xxx.x=10 sarebbe ancora ammesso (bisognava eventualmente proteggere l'attributo x con final)

Membri di una classe
Metodi – this

- la **parola chiave *this*** può essere utizzata all'interno dei metodi non statici ed è sostanzialmente un riferimento all'oggetto corrente su cui il metodo è stato invocato
- this viene utilizzato solitamente quando gli attributi sono mascherati da altre variabili locali omonime, ad esempio

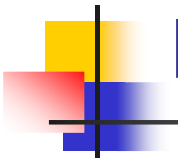
```
Class Prova {
    int x; int y;
public metodo (int x, int y) {
this.x=x;
this.y=y;
}
public static void main(String args[]) {
    Prova obj=new Prova();
    obj.metodo(3,12);
}
```



Membri di una classe
Metodi – main

- Il **metodo main** è, analogamente all'omonima funzione in C, il metodo invocato per primo nel caso sia presente
- la sintassi è

  public static void main(String args[]) { … }
- dove args è il vettore dei valori forniti sulla riga di comando
- il metodo main potrebbe non essere presente (applet)
- il metodo main può essere al massimo uno per ogni classe, ma una stessa applicazione, composta di più classi, può averne anche diversi, potendone fare partire uno alla volta.



Membri di una classe
Metodi – Overloading

- In Java è possibile definire più metodi con lo stesso nome, anche dentro alla stessa classe
- L'importante è che le funzioni "omonime" siano comunque distinguibili tramite la lista dei parametri, cioè tramite la loro signature
- Questa possibilità si chiama **overloading** ed è di grande utilità per catturare situazioni simili evitando una inutile proliferazione di nomi.
- i metodi con lo stesso nome devono differire per numero di parametri o per il tipo di almeno un parametro
- il tipo restituito deve invece essere sempre lo stesso

Membri di una classe
Metodi – Overloading

- l'overloading **è fonte di problemi** se uno dei metodi è varargs:

  public static void print (String title) { … }
  public static void print (String title, String… t1) { … }
  public static void print (String… title) { … }
- data l'invocazione print ("Hello"), il sistema userà il primo metodo, ma l'invocazione print("a","b") provocherà un errore di compilazione, non sapendo quale metodo utilizzare

Membri di una classe
Metodi – Overloading

```
public class AmbiguousOverloading {
  public static void main(String[] args) {
    System.out.println(max(1, 2));
  }
public static double max
(int num1, double num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }
public static double max
(double num1, int num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }
}
```

Talvolta possono essere presenti due o più funzioni che corrispondono ad una invocazione di un metodo. Queste sono chiamate **invocazioni ambigue** e provocano un errore di compilazione

Membri di una classe
Metodi – Controllo di accesso

- Spesso un metodo viene introdotto per **gestire un attributo**, ovvero per leggerlo o modificarlo non direttamente ma solo tramite opportuni metodi
- Esempio

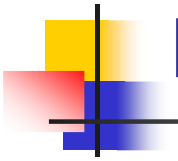
```
Public class Prova {
      public int x;
      …
      }
public class external {
   Prova pippo=new Prova();
   pippo.x=13;
   System.out.println("x is "+pippo.x);
}
```

in questo caso l'accesso ad x è diretto

Membri di una classe
Metodi – Controllo di accesso

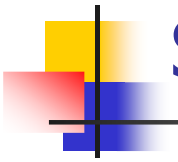
```
Public class Prova {
      private int x;
      …
      public int getx() {      return x;        }
      public void setx(int y) {       this.x=y;       }
      }
public class external {
   Prova pippo=new Prova();
   pippo.x=13; // NON FUNZIONA
   pippo.setx(13); // QUESTO SI
   System.out.println("x is "+pippo.x);    //KO
   System.out.println("x is "+pippo.getx());       //OK    }
```

- in questo caso l'accesso ad x può avvenire solo tramite i due metodi, politica frequente in Java (le librerie adottano largamente questo approccio)



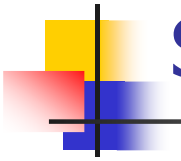
Membri di una classe
Metodi – Controllo di accesso

- i metodi possono implementare anche controlli sofisticati per settare o restituire l'attributo privato x solo sotto certe condizioni (avere x pubblico invece non permette questo controllo fine grained)
- l'alternativa è impostare una variabile come final, però:
  - così diventa a sola lettura, cosa che potrebbe non essere desiderata
  - si confondono immutabilità ed accessibilità, concetti ortogonali



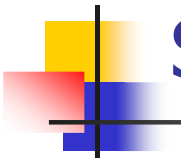
Membri di una classe
Scope Attributi nei metodi

- **Scope (ambiente)**: la parte di programma in cui una varibile può essere referenziata
- Una variabile **locale** è definita all'interno di un metodo
- Una variabile locale deve essere sempre dichiarata prima di essere utilizzata
- Lo scope di una variabile locale inizia quando la variabile viene dichiarata e continua fino alla fine del blocco che contiene la variabile.
- Noi possiamo dichiarare più variabili locali con lo stesso nome in differenti blocchi non-nesting (non annidati, di pari livello)



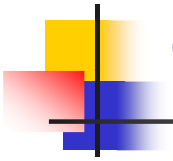
Membri di una classe
Scope Attributi

```
public static void correctMethod() {
  int x = 1;
  int y = 1;
  for (int i = 1; i < 10; i++) {
    x += i;
  }
  // i viene ridichiarata
  for (int i = 1; i < 10; i++) {
    y += i;
  }
}
```

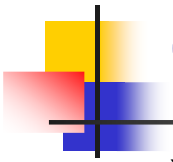
Membri di una classe
Scope Attributi

```
public static void incorrectMethod() {
  int x = 1;
  int y = 1;
  for (int i = 1; i < 10; i++) {
    int x = 0;
    x += i;
  }
}
```



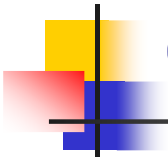
Classi

- Riassumendo, una classe contiene attributi e metodi
- La parte della classe che realizza il concetto di componente software si chiama **parte statica**
  - contiene i dati e le funzioni che sono propri della classe in quanto componente software autonomo
- L'altra parte della classe, che contiene la definizione di un tipo di dato astratto (ADT) ("stampo per oggetti"), è la **parte non-statica**
  - contiene i dati e le funzioni che saranno propri degli oggetti che verranno creati successivamente sulla base di questo "stampo".
- spesso una classe ha una sola delle due parti



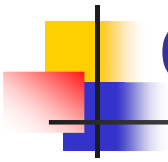
Classi

- Se c'è solo la parte STATICA:
  - la classe opera solo come componente software
  - contiene dati e funzioni, come un modulo
  - con in più la possibilità di definire l'appropriato livello di protezione
  - caso tipico: librerie di funzioni
- Se c'è solo la parte NON STATICA:
  - la classe definisce semplicemente un ADT
  - specifica la struttura interna di un tipo di dato, come una typedef applicata a una struct
  - con in più la possibilità di specificare dentro la struct anche le funzioni che operano su tali dati.



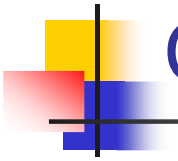
Classi – collegamento statico

- Nei linguaggi "classici":
  - si compila ogni file sorgente
  - si collegano i file oggetto così ottenuti
- In tale schema:
  - ogni file sorgente dichiara tutto ciò che usa
  - il compilatore ne accetta l'uso "condizionato"
  - il linker verifica la presenza delle definizioni risolvendo i riferimenti incrociati fra i file
  - l'eseguibile è "autocontenuto" (non contiene più riferimenti a entità esterne)
- massima efficienza e velocità perché l'eseguibile è già pronto
- Ma scarsa flessibilità perché tutto ciò che si usa deve essere dichiarato a priori
- Poco adatto ad ambienti con elevata dinamicità come internet



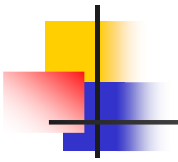
Classi – collegamento dinamico

- Non esistono dichiarazioni
  - si compila ogni file sorgente, e si esegue la classe pubblica che contiene il main
- In questo schema:
  - il compilatore accetta l'uso di altre classi perché può verificarne esistenza e interfaccia in quanto sa dove trovarle nel file system
  - le classi usate vengono caricate dall'esecutore al momento dell'uso



Classi – collegamento dinamico

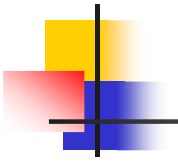
- ogni classe è compilata in un file .class
- il formato dei file .class ("bytecode") non è direttamente eseguibile: è un formato portabile, inter-piattaforma
- per eseguirlo occorre un interprete Java
  - è l'unico strato dipendente dalla piattaforma
- in questo modo si ottiene vera portabilità:
  - un file .class compilato su una piattaforma può funzionare su qualunque altra
- Si perde un po' in efficienza perché c'è di mezzo un interprete
- Si guadagna però la possibilità di scaricare ed eseguire dalla rete e l'indipendenza dall'hardware



Esempio di riferimento

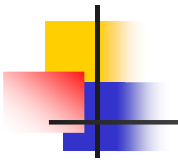
```
public class Counter {
   private int val;
   public void reset() { val = 0; }
   public void inc() { val++; }
   public int getValue() {
        return val;
   }
}
```

- Questa classe non contiene dati o funzioni sue proprie (statiche)
- Fornisce solo la definizione di un ADT che potrà essere usata poi per istanziare oggetti



Oggetti in Java

- Gli OGGETTI sono componenti "dinamici":
  - vengono creati "on the fly", al momento dell'uso, tramite l'operatore *new*
  - Sono creati a immagine e somiglianza (della parte non statica) di una classe, che ne descrive le proprietà
  - Su di essi è possibile invocare le operazioni pubbliche previste dalla classe (metodi)
  - Non occorre preoccuparsi della distruzione degli oggetti: Java ha un garbage collector che, a differenza del C in cui è richiesta la liberazione esplicita, entra automaticamente in funzione per recuperare la memoria occupata da oggetti non più referenziabili

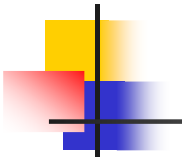


Oggetti in Java

- Uso: stile a " invio di messaggi" non una funzione con l'oggetto come parametro, ma un oggetto su cui si invocano metodi
- Ad esempio, se c è un Counter, un cliente potrà scrivere:

```
c.reset();
c.inc(); c.inc();
int x = c.getValue();
```

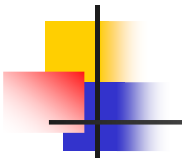
- Cambia il punto di vista: non operazioni con parametri, ma oggetti che svolgono servizi.



Oggetti in Java

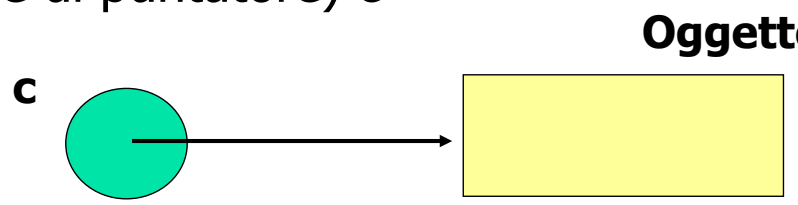
- Per creare un oggetto:
  - prima si definisce un riferimento, il cui tipo è il nome della classe che fa da modello poi si crea dinamicamente l'oggetto tramite l'operatore new (simile a malloc)
  - se non c'è abbastanza memoria, viene sollevata un'opportuna eccezione
- Esempio:

```
Counter c;              // def del riferimento
c = new Counter();      // creazione oggetto
```

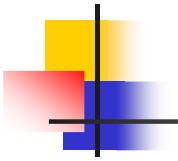
Oggetti in Java

- La frase *Counter c;* non definisce una variabile Counter, ma solo un riferimento a Counter (una specie di puntatore) c



- L'oggetto Counter viene poi creato dinamicamente, quando opportuno, con **new**

```
c = new Counter();
```

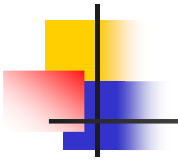


Oggetti in Java

- Un **riferimento** è come un puntatore, ma viene dereferenziato automaticamente, senza bisogno di * o altri operatori
- L'oggetto referenziato è quindi direttamente accessibile con la notazione puntata, senza bisogno di dereferencing esplicito:

```
c.inc(); x = c.getValue();
```

- Si conserva l'espressività dei puntatori, ma controllandone e semplificandone l'uso.



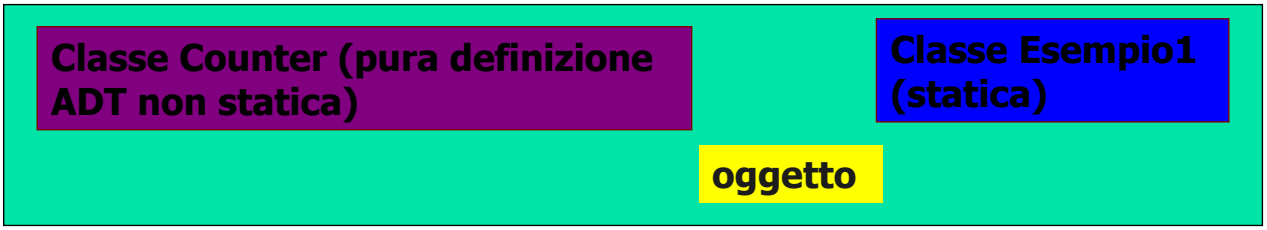
Oggetti in Java
Riferimenti vs Puntatori

- **Puntatore (C)**
- contiene l'indirizzo di una variabile (ricavabile con &)…
- e permette di manipolarlo in qualsiasi modo
  - incluso spostarsi altrove (aritmetica dei puntatori)
- richiede dereferencing esplicito
  - operatore * (o [ ])
  - rischio di errore
- possibilità di invadere aree non proprie!
- Strumento potente ma pericoloso.

- **Riferimento (Java)**
- contiene l'indirizzo di un oggetto…
- … ma non consente di vedere né di manipolare tale indirizzo!
  - – niente aritmetica dei puntatori
- ha il dereferencing automatico
  - niente più operatore * (o [ ])
  - niente più rischio di errore
- Impossibile invadere aree non proprie!
- Mantiene la potenza dei puntatori disciplinandone l'uso.

Oggetti in Java - Esempio

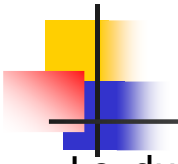
- Programma fatto di due classi:
  - una che fa da componente software, e ha come compito quello di definire il main (solo parte statica)
  - l'altra invece implementa il tipo Counter (solo parte non-statica)
- A run-time, nasce un oggetto:
  - lo crea "al volo" il main, quando vuole, tramite new, a immagine e somiglianza della classe Counter



Oggetti in Java - Esempio

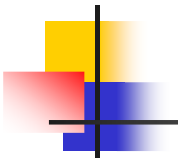
```
public class Esempio1 {
   public static void main(String v[]) {
        Counter c = new Counter();
        c.reset();
        c.inc(); c.inc();
        System.out.println(c.getValue());
   }
}
```

Il main crea un nuovo oggetto Counter e poi lo usa per nome, con la notazione puntata senza bisogno di dereferenziarlo esplicitamente



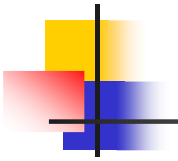
Oggetti in Java - Esempio

- Le due classi devono essere scritte in due file distinti, di nome, rispettivamente:
  - Esempio1.java (contiene la classe Esempio1)
  - Counter.java (contiene la classe Counter)
- Ciò è necessario perché entrambe le classi sono pubbliche: in un file .java può infatti esserci una sola classe pubblica
  - ma possono essercene altre non pubbliche
- I due file devono essere nella stessa directory in modo che Esempio possa vedere l'ADT contatore definito da Counter; mettendole nella stessa directory, appartengono allo stesso package, anche se implicitamente
- Per compilare:
  - javac Esempio1.java Counter.java



Oggetti in Java - Esempio

- La compilazione dei file produce due file .class, di nome, rispettivamente:
  - Esempio1.class
  - Counter.class
- Per eseguire il programma basta invocare l'interprete con il nome di quella classe (pubblica) che contiene il main
  - java Esempio1

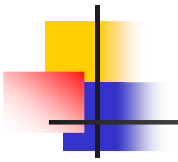


Oggetti in Java - Esempio

- Se la classe Counter non fosse stata pubblica, le due classi avrebbero potuto essere scritte nel medesimo file .java

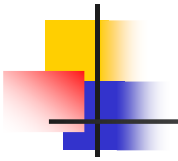
```
public class Esempio2 {
//Importante: l'ordine delle classi nel file è irrilevante,
//non esiste un concetto
//di dichiarazione che preceda l'uso
}
class Counter {
...
}
```

- nome del file = quello della classe pubblica (Esempio2.java)



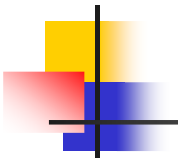
Oggetti in Java - Esempio

- Se la classe Counter non fosse stata pubblica, le due classi avrebbero potuto essere scritte nel medesimo file .java ma compilandole si sarebbero comunque ottenuti due file .class:
  - Esempio2.class
  - Counter.class
- In Java, c'è sempre un file .class per ogni singola classe compilata
  - ogni file .class rappresenta quella classe
  - non può inglobare più classi



Oggetti in Java - Riferimenti

- In C si possono definire, per ciascun tipo:
  - sia variabili (es. int x;)
  - sia puntatori (es. int *px;)
- In Java, invece, è il linguaggio a imporre le sue scelte:
  - variabili per i tipi primitivi (es. int x;)
  - riferimenti per gli oggetti (es. Counter c;)



Oggetti in Java - Riferimenti

- Cosa si può fare con i riferimenti?
  - Definirli:

  Counter c;
  - Assegnare loro la costante null:

  c = null;

  Questo riferimento ora non punta a nulla.
  - Le due cose insieme: Counter c2 = null;

  Definizione con inizializzazione a null

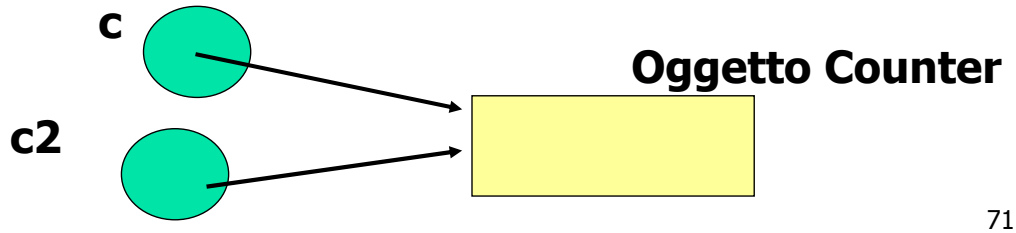
Oggetti in Java - Riferimenti

- Cosa si può fare con i riferimenti?
  - Usarli per creare nuovi oggetti:

  c = new Counter();
  - Assegnarli uno all'altro:

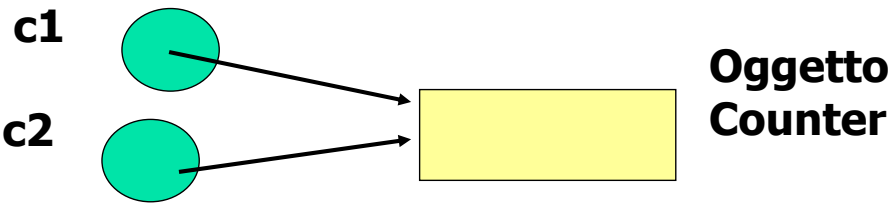
  Counter c2 = c;

  In tal caso, l'oggetto referenziato è condiviso



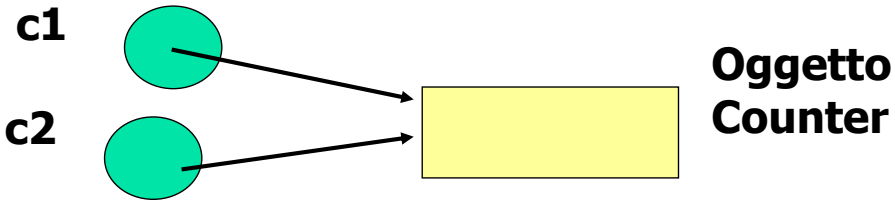
Oggetti in Java - Riferimenti

```
public class Esempio3 {
        public static void main(String[] args){
                Counter c1 = new Counter();
                c1.reset(); c1.inc();
                System.out.println("c1 = " + c1.getValue());
                Counter c2 = c1;
                c2.inc();
                System.out.println("c1 = " + c1.getValue());
                System.out.println("c2 = " + c2.getValue());
        }
}
```



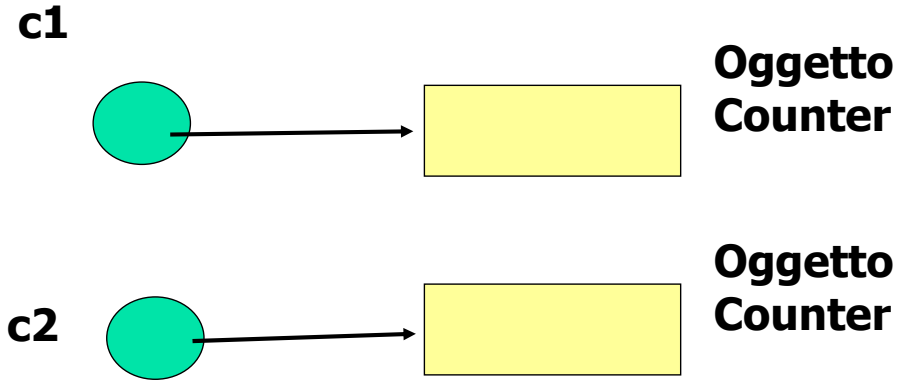
Oggetti in Java
Confronto Riferimenti

Quale significato per c1==c2?
- c1 e c2 sono due riferimenti
  uguali se puntano allo stesso oggetto
- qui, c1==c2 è true



Oggetti in Java
Confronto Riferimenti

E se si creano due oggetti identici?
Counter c1 = new Counter();
Counter c2 = new Counter();
c1.reset(); c2.reset();
Il contenuto non conta: c1==c2 è false

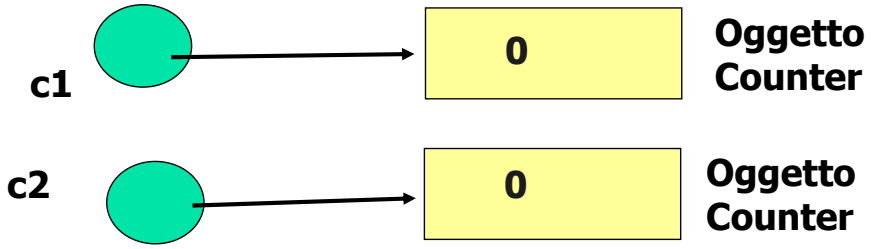


Oggetti in Java
Confronto Riferimenti

Per verificare l'uguaglianza fra i valori di due oggetti si usa il metodo equals:

```
Counter c1 = new Counter();
Counter c2 = new Counter();
c1.reset(); c2.reset();
```

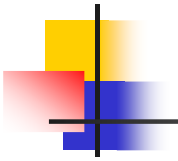
Contenuto uguale: c1.equals(c2) è true purché la classe Counter definisca il suo concetto di "uguaglianza"



Oggetti in Java
Confronto Riferimenti

La classe Counter con equals()

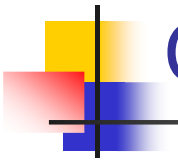
```
public class Counter {
        private int val;
        public boolean equals(Counter x){
                // UNA IMPLEMENTAZIONE
                return (val==x.val);      }
        public static boolean equals(Counter pippo, Counter pluto){
                // UN'ALTRA IMPLEMENTAZIONE
                return (pippo.val==pluto.val); }
        public void reset() { val = 0; }
        public void inc() { val++; }
        public int getValue() { return val;}
}
public class Esterna {
        public static void main (String args[]){
        Counter c1=new Counter(), c2=new Counter();
        if (c1.equals(c2)) {…}
        if (Counter.equals(c1,c2)) {…}   }}
```



Oggetti in Java
Confronto Riferimenti

- Consideriamo uguali due Counter se e solo se hanno identico valore Ma ogni altro criterio (sensato) sarebbe stato egualmente lecito! Ad esempio, per contatori modulari modulo N si potrebbe porre

  val == x.val % N
- Quali conseguenze sull'incapsulamento?
- x.val è un campo privato dell'oggetto x, che non è l'oggetto corrente su cui il metodo equals sta operando, è un altro oggetto
- In Java l'incapsulamento è a livello di classe: i metodi di una classe hanno libero accesso a tutti i campi (anche privati) di tutte le istanze di tale classe.
- Se volessimo comunque rispettare l'incapsulamento dell'istanza x, potremmo scrivere:
- return (val == x.getValue());



Oggetti in Java
Costruzione di oggetti

- Molti errori nel software sono causati da mancate inizializzazioni di variabili
- Perciò i linguaggi a oggetti introducono il costruttore, un metodo particolare che automatizza l'inizializzazione degli oggetto
  - è invocato automaticamente dal sistema ogni volta che si crea un nuovo oggetto di quella classe, subito dopo avere assegnato i valori iniziali di default agli attributi dell'istanza
  - può anche essere invocato esplicitamente

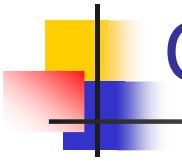
Oggetti in Java
Costruzione di oggetti

- Il **costruttore**:
  - è un metodo che ha un nome fisso, uguale al nome della classe
  - non ha tipo di ritorno, neppure void; il suo scopo infatti non è "calcolare qualcosa", ma inizializzare un oggetto
  - può non essere unico; spesso vi sono più costruttori (overloading), con diverse liste di parametri e servono a inizializzare l'oggetto a partire da situazioni diverse
- È anche possibile definire dei **blocchi di inizializzazione**, ossia un blocco di istruzioni racchiuso da {} e situato all'interno della dichiarazione di una classe ma fuori da ogni membro;
  - le istruzioni sono eseguite all'inizio della chiamata di un qualsiasi costruttore della classe; i blocchi di inizializzazione sono infatti usati per raccogliere istruzioni comuni a tutti i costruttori
  - I blocchi possono contenere comunque qualsiasi istruzione, pertanto occorre evitare il loro utilizzo se non strettamente indispensabile
  - I blocchi di inizializzazione statica, circondati da *static {}*, sono analoghi a quelli standard, ma può solo riferirsi ai membri statici della classe

Oggetti in Java
Costruzione di oggetti

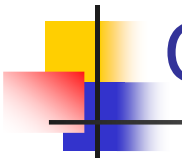
La classe Counter

```
public class Counter {
        private int val;
        public Counter() { val = 1; }
        public Counter(int v) { val = v; }
        public void reset() { val = 0; }
        public void inc() { val++; }
        public int getValue() { return val;}
        public boolean equals(Counter x) ...
}
```



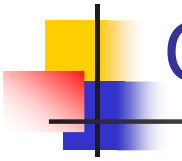
Oggetti in Java
Costruzione di oggetti

```
public class Esempio4 {
        public static void main(String[] args){
        Counter c1 = new Counter();
        c1.inc();
        Counter c2 = new Counter(10);
        c2.inc();
        System.out.println(c1.getValue()); // 2
        System.out.println(c2.getValue()); // 11
        }
}
```

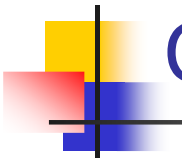
Oggetti in Java
Costruzione di oggetti

- Il costruttore senza parametri si chiama **costruttore di default**
  - viene usato per inizializzare oggetti quando non si specificano valori iniziali
  - esiste sempre: se non lo definiamo noi, ne aggiunge uno il sistema
  - però, il costruttore di default definito dal sistema non fa nulla: quindi, è opportuno definirlo sempre.



Oggetti in Java
Costruzione di oggetti

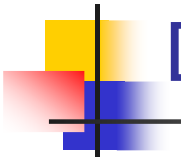
- Una classe destinata a fungere da stampo per oggetti deve definire almeno un costruttore pubblico
  - in assenza di costruttori pubblici, oggetti di tale classe non potrebbero essere costruiti
  - a volte si trovano classi con un costruttore privato proprio per impedire che se ne creino istanze!
  - il costruttore di default definito dal sistema è pubblico
- È possibile definire costruttori non pubblici per scopi particolari



Oggetti in Java
Costruzione di oggetti

Esempio

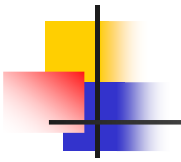
```
public class Counter {
        private int val;
        public Counter() { val = 1; }
        public Counter(int v) { val = v; }
        public void reset() { val = 0; }
        public void inc() { val++; }
        public void inc(int k) { val += k; }
        public int getValue() { return val;}
}
```



Oggetti in Java
Distruzione di oggetti

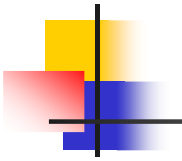
- Quando si cessa di utilizzare un oggetto, si smette di farvi riferimento:
  - perché si pongono a null tutti i suoi puntatori
  - perché si esce dal codice in cui era visibile il riferimento (ad esempio da dentro un metodo)
- In tali situazioni, il processo di garbage collection, avviato automaticamente da Java, recupera lo spazio, risolvendo i problemi connessi alla manualità di rilascio delle variabili (dangling pointers)
- il GC è un processo però autonomo; non è detto che entri in funzione immediatamente, anzi di solito parte solo quando arrivano nuove richieste e/o la memoria sta per esaurirsi; potrebbe rappresentare un problema per programmi time-critical

A. Longheu – Linguaggi M-Z – Ing. Inf. 2007-2008



Oggetti in Java
Distruzione di oggetti

- Esistono diverse teorie per la progettazione di un GC efficiente ed efficace
- un modello semplice è il mark and sweep, che prevede due fasi:
  - nella prima, si determina l'insieme degli oggetti raggiungibili direttamente, che formeranno un insieme di radici (root)
  - successivamente, il GC marca tutti gli oggetti raggiungibili dalle radici, e recursivamente analizza e marca gli oggetti raggiungibili da quelli così ottenuti; alla fine, gli oggetti non marcati sono quelli non più raggiungibili, e saranno rimossi dalla memoria
  - il modello è elementare, nella realtà i meccanismi sono più complessi



Oggetti in Java
Distruzione di oggetti

- Java permette, entro certi limiti, di interagire con il proprio GC:
  - è possibile implementare il metodo finalize, che viene invocato dal GC quando un oggetto non è più raggiungibile, e permette all'utente di eseguire azioni, ad esempil il rilascio di risorse diverse dalla memoria (file o connessioni aperte)
  - le classi runtime e System prevedono metodi quali gc(), per chiedere l'esecuzione del gc (anche se questo non è una garanzia che verrà in corrispondenza ed istantaneamente liberata memoria) e runfinalization() che richiede alla JVM di eseguire i finalizzatori degli oggetti irraggiungibili, per rilasciare le risorse