

IL CONCETTO DI CLASSE

Una **CLASSE** riunisce le proprietà di:

- **componente software**: può essere dotata di suoi propri *dati / operazioni*
- **moduli**: riunisce dati e relative operazioni, fornendo idonei *meccanismi di protezione*
- **tipi di dato astratto**: può fungere da “stampo” per *creare nuovi oggetti*

Java e Classi 1

IL LINGUAGGIO JAVA

- È un linguaggio **totalmente a oggetti**: tranne i tipi primitivi di base (`int`, `float`, ...), **esistono solo classi e oggetti**
- È fortemente ispirato al C++, ma riprogettato **senza il requisito della piena compatibilità col C** (a cui però assomiglia...)
- Un programma è un insieme *di classi*
 - non esistono funzioni definite (come in C) a livello esterno, né variabili globali esterne
 - **anche il `main` è definito dentro a una classe!**

Java e Classi 2

AMBIENTI DI PROGRAMMAZIONE

È l'insieme dei programmi che consentono la scrittura, la verifica e l'esecuzione di nuovi programmi (**fasi di sviluppo**)

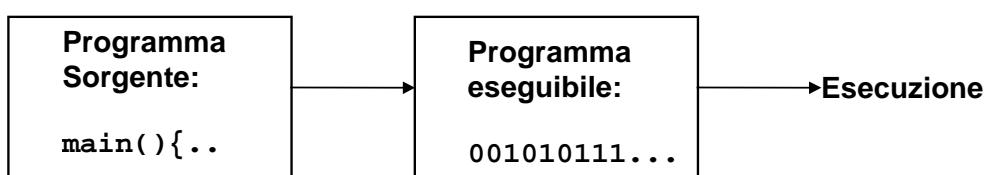
Sviluppo di un programma

- Affinché un programma scritto in un qualsiasi linguaggio di programmazione sia comprensibile (e quindi eseguibile) da un calcolatore, *occorre tradurlo* dal linguaggio originario al linguaggio della macchina

Questa operazione viene normalmente svolta da speciali strumenti, detti **traduttori**

Java e Classi 3

SVILUPPO DI PROGRAMMI

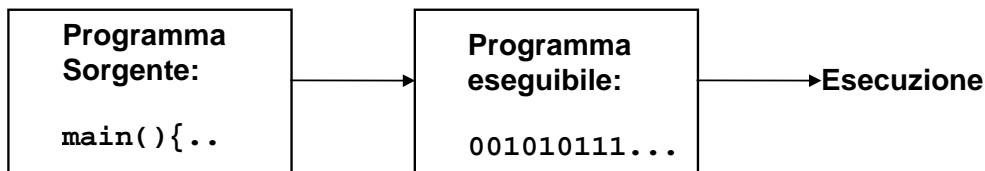


Due categorie di traduttori:

- i **Compilatori** traducono l'intero programma e producono il programma in linguaggio macchina
- gli **Interpreti** traducono ed eseguono immediatamente ogni singola istruzione del *programma sorgente*

Java e Classi 4

SVILUPPO DI PROGRAMMI (segue)



Quindi:

- **nel caso del compilatore**, lo schema precedente viene percorso *una volta sola* prima dell'esecuzione
- **nel caso dell'interprete**, lo schema viene invece attraversato *tante volte quante sono le istruzioni* che compongono il programma

Java e Classi 5

COMPILATORI E INTERPRETI

- I **compilatori** traducono automaticamente un programma dal linguaggio di alto livello a quello macchina (per un determinato elaboratore)
- Gli **interpreti** sono programmi capaci di eseguire direttamente un programma nel linguaggio scelto, istruzione per istruzione
- I programmi compilati sono in generale più efficienti di quelli interpretati

Java e Classi 6

AMBIENTI DI PROGRAMMAZIONE

I° CASO: COMPILAZIONE

- **Compilatore:** opera la **traduzione di un programma sorgente** (scritto in un linguaggio ad alto livello) **in un programma oggetto** direttamente eseguibile dal calcolatore
- **Linker:** (*collegatore*) nel caso in cui la costruzione del programma oggetto richieda l'unione di **più moduli** (compilati separatamente), il linker provvede a **collegarli** formando un unico **programma eseguibile**

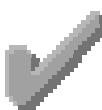
Java e Classi 7

AMBIENTI DI PROGRAMMAZIONE

II° CASO: INTERPRETAZIONE

- **Interprete:** traduce ed esegue direttamente **ciascuna istruzione** del **programma sorgente, istruzione per istruzione**

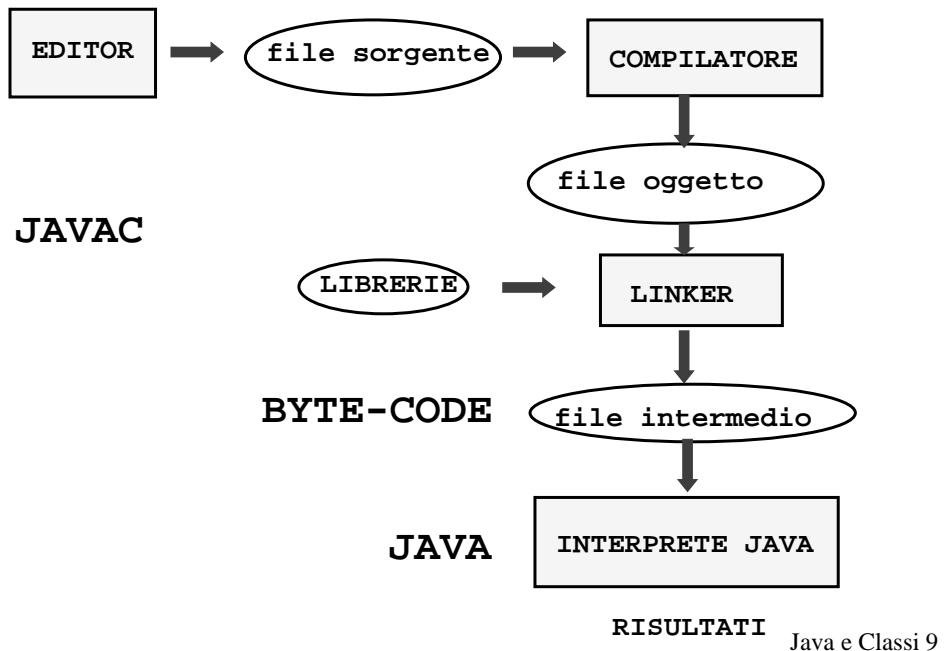
È generalmente in alternativa al compilatore
(raramente presenti entrambi)



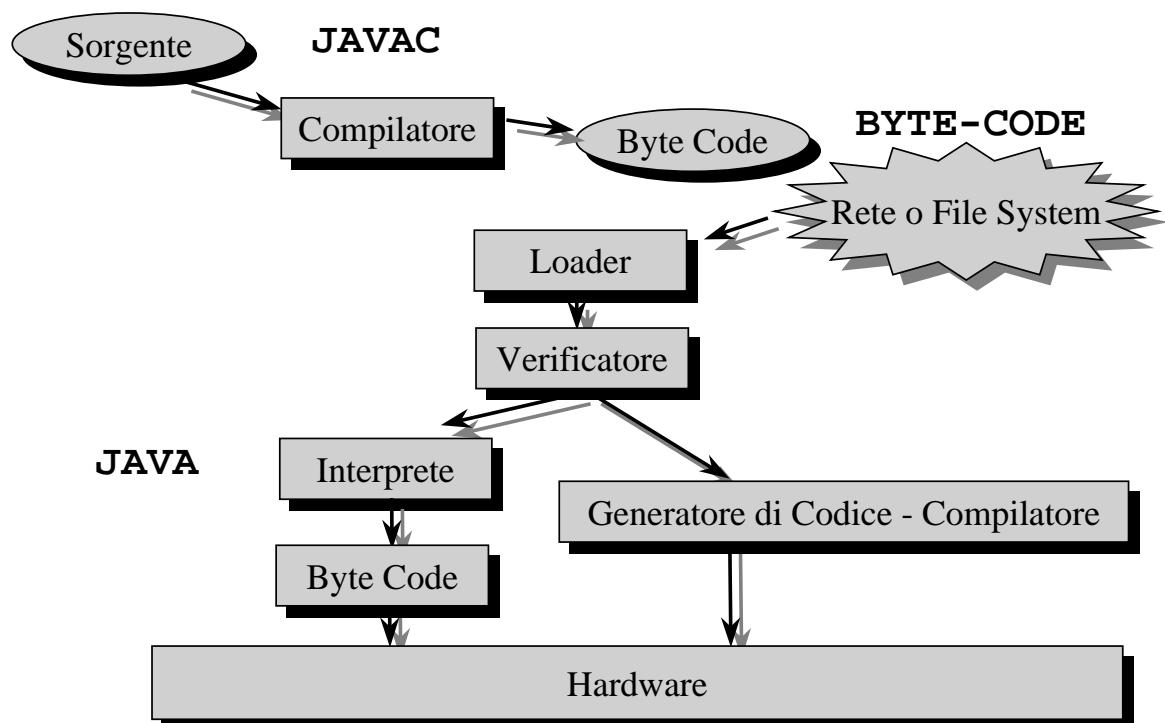
Traduzione ed esecuzione sono intercalate, e avvengono *istruzione per istruzione*

Java e Classi 8

APPROCCIO MISTO



APPROCCIO JAVA



LINGUAGGIO O ARCHITETTURA?

A differenza del C++, Java viene fornito con una notevole gerarchia di classi standard già pronte, che coprono quasi ogni esigenza

È un'architettura già pronta per l'uso!

- *Architettura indipendente dalla piattaforma*
- *Package grafici (AWT e Swing)*
- *Programmazione a eventi (molto evoluta!)*
- *Supporto di rete: URL, Socket, ...*
- *Supporto per il multi-threading*
- *Interfacciamento con database (JDBC)*
- *Supporto per la sicurezza (cifratura)*

Java e Classi 11

JAVA: L'INIZIO

- Nasce per applicazioni “embedded”
- Si diffonde attraverso il concetto di *applet* come piccola (?) applicazione da eseguire automaticamente in un browser Web
 - *grafica portatile ed eseguibile ovunque*
 - *modello di sicurezza “sandbox”*
- Può benissimo essere usato come linguaggio per costruire applicazioni
 - *anche non per Internet*
 - *anche non grafiche*

Java e Classi 12

JAVA: L'EVOLUZIONE

Oggi, molto orientato al *network computing*

- interazione con *oggetti remoti (RMI)*
- interazione con i *data base (JDBC)*
- interoperabilità con *CORBA*
- integrabilità attraverso *J2EE e Java Beans*
- *servlet* come schema flessibile per estendere un server Web

... e inoltre...

Java e Classi 13

JAVA: NON SOLO RETE

...

- applicazioni embedded (*JavaCard API*)
- dispositivi integrati (*JavaRing*)
- ispirazione per sistemi operativi (*JavaOS*)
- component technology (*JavaBeans*)
- ...

Java e Classi 14

JAVA: “LA SOLUZIONE” ?

- La tecnologia Java non è certo l'unica disponibile
- Non è detto che sia sempre la più adatta
- Però, permette di ottenere una soluzione *omogenea e uniforme* per lo sviluppo di *tutti gli aspetti* di un'applicazione

Java e Classi 15

CLASSI IN JAVA

Una **classe Java** è una entità *sintatticamente simile alle struct*

- però, contiene *non solo i dati...*
- .. ma anche *le funzioni che operano su quei dati*
- e ne specifica *il livello di protezione*
 - **pubblico:** visibile anche dall'esterno
 - **privato:** visibile solo entro la classe
 - ...

Java e Classi 16

CLASSI IN JAVA

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**
- ma contiene anche la definizione di un **tipo di dato astratto**, cioè uno "stampo" per **creare nuovi oggetti**, anch'essi dotati di idonei meccanismi di **protezione**

Java e Classi 17

CLASSI IN JAVA

- 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)** ("*schema 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 "schema"

Java e Classi 18

IL CONCETTO DI CLASSE



Una classe è un *componente software*: può avere propri *dati (STATICI)* e proprie *operazioni (STATICHE)*

Una classe contiene però anche la *definizione di un ADT*, usabile come "schema" per creare poi *nuovi oggetti (parte NON statica)*

Java e Classi 19

IL CONCETTO DI CLASSE

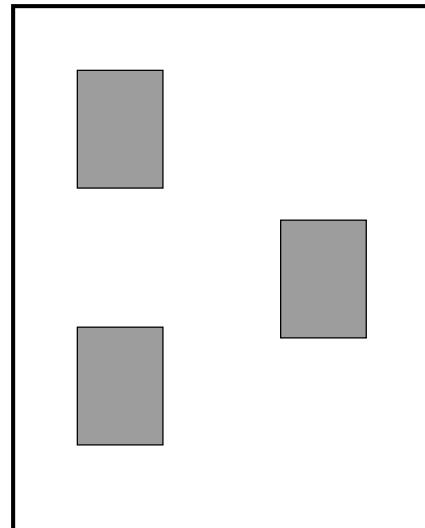
- **Se c'è solo la parte STATICÀ:**
 - la classe opera solo come componente software
 - contiene dati e funzioni, come un modulo
 - con in più la possibilità di definire l'appropriato *livello di protezione*
 - caso tipico: *library di funzioni*
- **Se c'è solo la parte NON STATICÀ:**
 - la classe definisce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

Java e Classi 20

PROGRAMMI IN JAVA

Un programma Java è *un insieme di classi e oggetti*

- Le classi sono componenti *statici*, che esistono già all'inizio del programma.

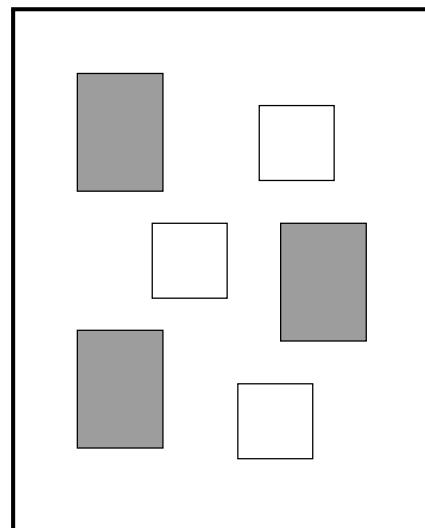


Java e Classi 21

PROGRAMMI IN JAVA

Un programma Java è *un insieme di classi e oggetti*

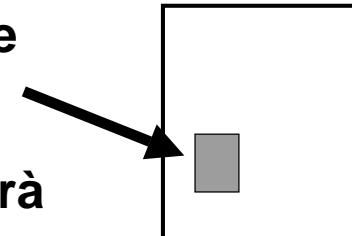
- Le classi sono componenti *statici*, che esistono già all'inizio del programma
- Gli oggetti sono invece componenti *dinamici*, che vengono creati dinamicamente al momento del bisogno



Java e Classi 22

IL PIÙ SEMPLICE PROGRAMMA

- Il più semplice programma Java è dunque costituito da *una singola classe* operante come *singolo componente software*
- Essa avrà quindi la sola parte statica
- Come minimo, tale parte dovrà definire *una singola funzione (statica)*: *il main*

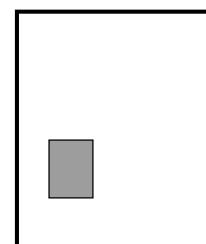


Java e Classi 23

IL MAIN IN JAVA

Il main in Java è una funzione pubblica con la seguente interfaccia obbligatoria:

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



- Deve essere dichiarato **public, static, void**
- Non può avere valore di ritorno (è void)
- Deve sempre prevedere gli argomenti dalla linea di comando, *anche se non vengono usati*, sotto forma di *array di String* (*il primo non è il nome del programma*)

Java e Classi 24

PROGRAMMI IN JAVA

Prima differenza rispetto al C:

- il `main` deve sempre dichiarare l'array di stringhe `args`, anche se non lo usa (ovviamente può anche non chiamarlo `args`...)
- il `main` non è più una funzione a sé stante: è definito dentro a una classe pubblica, ed è a sua volta pubblico
- In effetti, in Java *non esiste nulla* che non sia definito dentro una qualche classe!

Java e Classi 25

CLASSI IN JAVA

Convenzioni rispettate dai componenti esistenti:

- il nome di una classe ha sempre *l'iniziale maiuscola* (es. Esempio)
 - se il nome è composto di più parole concatenate, ognuna ha l'iniziale maiuscola (es. DispositivoCheConta)
 - non si usano trattini di sottolineatura
- i nomi dei singoli campi (dati e funzioni) iniziano invece per *minuscola*

Java e Classi 26

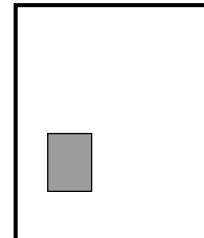
ESEMPIO BASE

Un programma costituito da una singola classe EsempioBase che definisce il main

La classe che contiene il main dev'essere **pubblica**

```
public class EsempioBase {  
    public static void main(  
        String args[]){  
        int x = 3, y = 4;  
        int z = x + y;  
    }  
}
```

Java e Classi 27



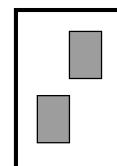
ESEMPIO 0

Un programma costituito da due classi:

- la nostra Esempio0, che definisce il main
- la classe di sistema System

```
public class Esempio0 {  
    public static void main(  
        String args[]){  
        System.out.println("Hello!");  
    }  
}
```

Stampa a video la classica frase di benvenuto



Java e Classi 28

ESEMPIO 0

Stile a “*invio di messaggi*”:

- non più chiamate di funzioni *con parametri* che rappresentano i dati su cui operare
(ma che siano quelli lo sa solo l'utente...)...
- ...ma componenti su cui vengono invocate operazioni a essi pertinenti

Notazione puntata:

```
System.out.println("Hello!");
```

Il messaggio `println("Hello!")` è inviato all'oggetto `out` che è un dato (statico) presente nella classe `System`

CLASSI E FILE

- In Java esiste una *ben precisa corrispondenza* fra
 - nome di una classe pubblica
 - nome del file in cui essa dev'essere definita
- Una classe pubblica deve essere definita in un file *con lo stesso nome della classe* ed estensione `.java`
- **Esempi**
classe `EsempioBase` → file `EsempioBase.java`
classe `Esempio0` → file `Esempio0.java`

CLASSI E FILE

- In Java esiste una *ben precisa corrispondenza* fra
 - nome di una classe pubblica
 - ! Essenziale:
 - poter usare *nomi di file lunghi*
 - rispettare maiuscole/minuscole
- Un file finita classe in un file finita classe ed estensione . java
- **Esempi**
classe EsempioBase → file EsempioBase.java
classe Esempio0 → file Esempio0.java

Java e Classi 31

IL Java Development Kit (JDK)

Il JDK della Sun Microsystems è l'insieme di strumenti di sviluppo che funge da “*riferimento ufficiale*” del linguaggio Java

- *non è un ambiente grafico integrato:*
è solo un insieme di strumenti da usare dalla linea di comando
- *non è particolarmente veloce ed efficiente* (non sostituisce strumenti commerciali)
- **però funziona, è gratuito ed esiste per tutte le piattaforme** (Win32, Linux, Solaris, Mac..)

Java e Classi 32

... E OLTRE

Esistono molti strumenti tesi a migliorare il JDK, e/o a renderne più semplice l'uso

- ***editor con “syntax highlighting”***
 - TextTool, WinEdt, JPad, e tanti altri
- ***ambienti integrati freeware che, pur usando “sotto” il JDK, ne consentono l’uso in modo interattivo e in ambiente grafico***
 - FreeBuilder, Forte, Jasupremo, etc...
- ***ambienti integrati commerciali, dotati di compilatori propri e debugger***
 - Jbuilder, Codewarrior, VisualAge for Java, ...

Java e Classi 33

COMPILAZIONE ED ESECUZIONE

Usando il JDK della Sun:

- ***Compilazione:***
`javac Esempio0.java`
(produce `Esempio0.class`)
- ***Esecuzione:***
`java Esempio0`

**Non esiste una fase di link esplicita:
Java adotta il collegamento dinamico**

Java e Classi 34

COLLEGAMENTO STATICO...

Nei linguaggi “classici”:

- si compila ogni file sorgente
- **si collegano i file oggetto così ottenuti**

In questo 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)**

Java e Classi 35

COLLEGAMENTO STATICO...

Nei linguaggi “classici”:

- si compila ogni file sorgente
- **si collegano i file oggetto così ottenuti**

Massima efficienza e velocità,
perché l'eseguibile è “già pronto”

In questo schema...

- ...ma scarsa flessibilità**, perché tutto ciò che si usa deve essere **dichiarato a priori**
- risolvendo i *riferimenti incrociati* fra i file
- **L'eseguibile è “autocontenuto” (non contiene più riferimenti a entità esterne)**

Poco adatto ad *ambienti a elevata dinamicità come Internet*

Java e Classi 36

.. E COLLEGAMENTO DINAMICO

In Java

- **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 verificare l'esistenza e l'interfaccia** in quanto **sa dove trovarle nel file system**
- **le classi usate vengono caricate dall'esecutore solo al momento dell'uso**

Java e Classi 37

ESECUZIONE E PORTABILITÀ

In Java,

- **ogni classe è compilata in un file .class**
- **il formato dei file .class (“bytecode”)** non è direttamente eseguibile: è **un formato portatile, 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!!!

Java e Classi 38

ESECUZIONE E PORTABILITÀ

In Java,

- **ogni classe è compilata in un file .class**
- **il formato di esecuzione non è diretto** *Si perde un po' in efficienza (c'è di mezzo un interprete)...*
- **formato portabile, inter-piattaforma**
..ma si guadagna *molto di più:*
 - possibilità di scaricare ed eseguire codice dalla rete
 - indipendenza dall'hardware
 - “*write once, run everywhere*”

Java e Classi 39

LA DOCUMENTAZIONE

- È noto che un buon programma dovrebbe essere ben documentato..
- *ma l'esperienza insegna che quasi mai ciò viene fatto!*
 - “*non c'è tempo*”, “*ci si penserà poi*”...
 - ... e alla fine la documentazione non c'è!
- Java prende atto che la gente *non scrive* documentazione...
- ..e quindi fornisce uno strumento per *produrla automaticamente* a partire dai *commenti* scritti nel programma: *javadoc*

Java e Classi 40

L'ESEMPIO... COMPLETATO

```
/** File Esempio0.java
 * Applicazione Java da linea di comando
 * Stampa la classica frase di benvenuto
@author Enrico Denti
@version 1.0, 5/4/98
*/
public class Esempio0 {
    public static void main(String args[]){
        System.out.println("Hello!");
    }
}
```

Informazioni di documentazione

Java e Classi 41

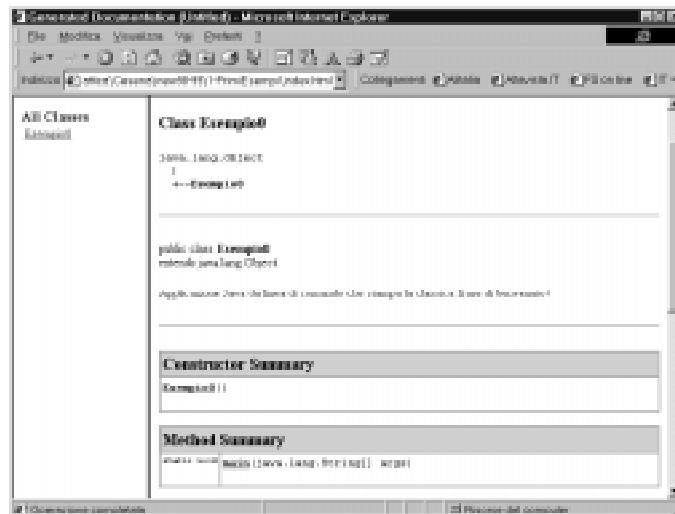
L'ESEMPIO... COMPLETATO

Per produrre la relativa documentazione:
javadoc Esempio0.java

Produce una serie di file HTML



Si consulti la documentazione di javadoc per i dettagli.



Java e Classi 42

TIPI DI DATO PRIMITIVI IN JAVA

- **caratteri**
 - **char** (2 byte) codifica UNICODE
 - coincide con ASCII sui primi 127 caratteri
 - e con ANSI / ASCII sui primi 255 caratteri
 - *costanti char anche in forma '\u2122'*
- **interi (con segno)**
 - **byte** (1 byte) -128 ... +127
 - **short** (2 byte) -32768 ... +32767
 - **int** (4 byte) -2.147.483.648 ... 2.147.483.647
 - **long** (8 byte) -9 10¹⁸ ... +9 10¹⁸

NB: le costanti long terminano con la lettera L

Java e Classi 43

TIPI DI DATO PRIMITIVI IN JAVA

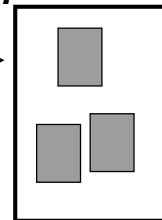
- **reali (IEEE-754)**
 - **float** (4 byte) -10⁴⁵ ... +10³⁸
(6-7 cifre significative)
 - **double** (8 byte) -10³²⁸ ... +10³⁰⁸
(14-15 cifre significative)
- **boolean**
 - **boolean** (1 bit) **false e true**
 - tipo autonomo **totalmente disaccoppiato dagli interi**: non si convertono boolean in interi e viceversa, *neanche con un cast*
 - tutte le espressioni relazionali e logiche danno come risultato un boolean, non più un int!

Java e Classi 44

UN ESEMPIO CON TRE CLASSI

- Un programma su tre classi, tutte usate come **componenti software** (*solo parte statica*):
 - Una classe **Esempio** con il **main** →
 - Le classi di sistema **Math** e **System** →
- **Chi è Math ?**
 - **Math** è, di fatto, la libreria matematica
 - **comprende solo costanti e funzioni statiche:**
 - costanti: **E, PI**
 - funzioni: **abs(), asin(), acos(), atan(), min(), max(), exp(), log(), pow(), sin(), cos(), tan()...**

Java e Classi 45



UN ESEMPIO CON TRE CLASSI

- Il nome di una classe (**Math** o **System**) definisce uno *spazio di nomi*
- Per *usare* una funzione o una costante definita dentro di esse occorre specificarne il *nome completo*, mediante la *notazione puntata*

Esempio:

```
public class EsempioMath {  
    public static void main(String args[]){  
        double x = Math.sin(Math.PI/3);  
        System.out.println(x);  
    }  
}
```

Java e Classi 46

UN ESEMPIO CON TRE CLASSI

- Il nome di una classe (`Math` o `System`) definisce uno *spazio di nomi*
- Per usare una funzione o un metodo bisogna specificare chi fornisce un certo servizio
 - Inoltre, è immediato riconoscere chi fornisce un certo servizio

In questo modo si evitano *conflitti di nome (name clash)*

```
public class EsempioMath {  
    public static void main(String args[]){  
        double x = Math.sin(Math.PI/3);  
        System.out.println(x);  
    }  
}
```

Java e Classi 47

UNA CLASSE PER I NUMERI PRIMI

- Tutti gli esempi fatti con oggetti
- Un componente che a ogni invocazione restituisce il successivo numero di una sequenza (es. numeri primi)
 - In C realizzato con un modulo
 - Ora lo possiamo realizzare con (la parte statica di) una classe
- Possiamo anche *garantire l'incapsulamento*
 - In C avevamo usato una variabile static, che come tale è automaticamente protetta
 - Ora possiamo specificare esplicitamente cosa debba essere privato e cosa invece pubblico

Java e Classi 48

UNA CLASSE PER I NUMERI PRIMI

```
public class NumeriPrimi {  
    private static int lastPrime = 0;  
    private static boolean isPrime(int p) {  
        ... lo stesso codice usato in C  
    }  
    public static int nextPrime() {  
        ... lo stesso codice usato in C  
    }  
}
```

• È un puro **componente software** (*ha solo la parte statica*)

• Il dato **lastPrime** (un intero) e la funzione **isPrime** sono **privati** e come tali *invisibile a chiunque fuori dalla classe*

• La funzione **nextPrime()** è invece **pubblica** e come tale *usabile da chiunque*, dentro e fuori dalla classe

Provare a definire un'altra classe
EsempioPrimi che definisca un
main che usi nextPrime()

Java e Classi 49

UNA CLASSE PER I NUMERI PRIMI

Seconda differenza rispetto al C:

- una funzione *senza parametri* viene definita *senza la parola-chiave void*
 - **NON così...**
`public static int nextPrime(void) { ... }`
 - ... **MA così:**
`public static int nextPrime(){ ... }`
- la parola-chiave **void** viene *ancora usata*, ma *solo per il tipo di ritorno delle procedure*

Java e Classi 50

CLASSI E OGGETTI IN JAVA

Esclusi i tipi primitivi, *in Java esistono solo:*

- *classi*
 - componenti software che possono avere i loro dati e le loro funzioni (parte statica)
 - ma anche fare da “schema” per costruire oggetti (parte non-statica)
- *oggetti*
 - entità dinamiche costruite al momento del bisogno secondo lo "stampo" fornito dalla parte "Definizione ADT" di una classe

Java e Classi 51

CLASSI COME ADT

Una classe con solo la parte NON-STATICÀ è una *pura definizione di ADT*

- È simile a una struct + typedef del C...
- ... *ma riunisce dati e comportamento (funzioni) in un unico costrutto linguistico*
- Ha solo *variabili e funzioni non-statiche*
- Definisce un tipo, che potrà essere usato per *creare (istanziare) oggetti*

Java e Classi 52

ESEMPIO: IL CONTATORE

- **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**

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

Dati

Operazioni (comportamento)

Unico costrutto linguistico per dati e operazioni

Java e Classi 53

ESEMPIO: LA CLASSE Counter

- **Questo è un esempio di incapsulamento**
- **Il campo val è *privato*: può essere acceduto solo dalle *operazioni definite nella medesima classe* (reset, inc, getValue), e nessun altro! Si garantisce l'incapsulamento**

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

Dati

Operazioni (comportamento)

Unico costrutto linguistico per dati e operazioni

Java e Classi 54

OGGETTI IN JAVA

- Gli OGGETTI sono componenti “dinamici”: *vengono creati “al volo”, 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
- Non occorre preoccuparsi della distruzione degli oggetti: Java ha un *garbage collector!*

Java e Classi 55

OGGETTI IN JAVA

Uso: stile a “*invio di messaggi*”

- non una funzione con l’oggetto come parametro...
- ...ma bensì *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();
```

Java e Classi 56

CREAZIONE DI OGGETTI

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 in C)*

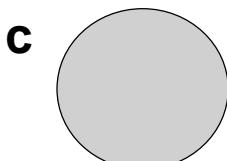
Esempio:

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

Java e Classi 57

RIFERIMENTI A OGGETTI

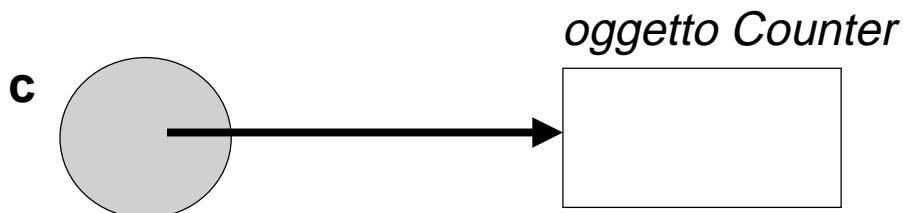
- La frase `Counter c;`
*non definisce una variabile Counter,
ma solo un riferimento a Counter
(una specie di puntatore)*



Java e Classi 58

RIFERIMENTI A OGGETTI

- La frase `Counter c;`
*non definisce una variabile Counter,
ma solo un riferimento a Counter*



- L'oggetto `Counter` viene poi creato dinamicamente, quando opportuno, con `new`
`c = new Counter();`

Java e Classi 59

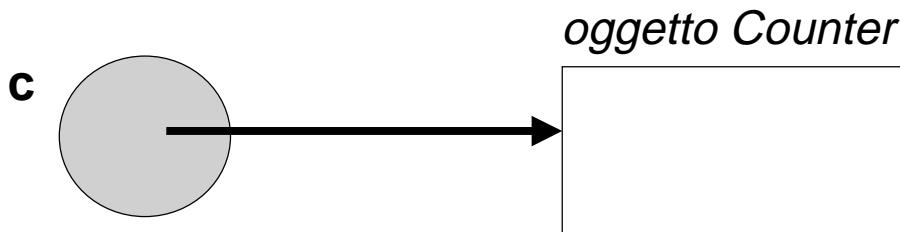
RIFERIMENTI A OGGETTI

- 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 dereferencing esplicito:
`c.inc(); x = c.getValue();`
- Si conserva l'espressività dei puntatori, ma *controllandone e semplificandone l'uso*

Java e Classi 60

RIFERIMENTI vs. PUNTATORI

A livello fisico, *un riferimento è di fatto un puntatore...*



...ma rispetto ad esso è *un'astrazione di più alto livello, che riduce i pericoli legati all'abuso (o all'uso errato) dei puntatori e dei relativi meccanismi*

Java e Classi 61

RIFERIMENTI vs. PUNTATORI

Puntatore (C)

- contiene l'indirizzo di una qualsiasi variabile (ricavabile con &)...
- ... e permette di manipolarlo in qualsiasi modo
 - incluso spostarsi altrove (aritmetica dei puntatori)
- richiede *dereferencing esplicito*
 - operatore * (o [])
 - rischio di errore
- possibile 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

Java e Classi 62

CREAZIONE DI OGGETTI

Per creare un oggetto:

La frase `Counter c;` definisce un *riferimento* a un (futuro) oggetto di classe `Counter`

In *riferimento*, il cui tipo *serve che fa da modello* per creare dinamicamente l'oggetto

Esempio:

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

// creazione oggetto

L'oggetto di tipo `Counter` viene però *creato dinamicamente* solo in un secondo momento, quando serve, mediante l'operatore `new`

Java e Classi 63

ESEMPIO COMPLETO

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)

Classe Counter (pura definizione di ADT, solo parte non-statica)

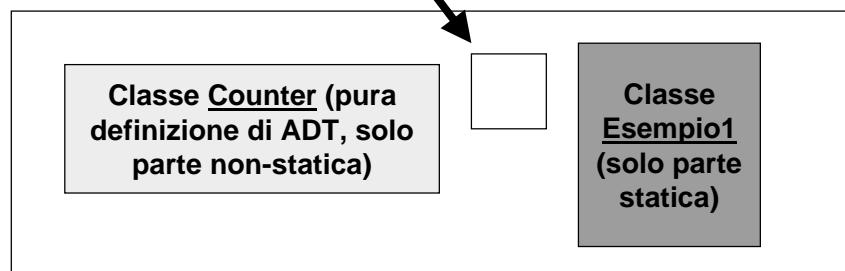
Classe Esempio1 (solo parte statica)

Java e Classi 64

ESEMPIO COMPLETO

A run-time, nasce un oggetto:

- *Io crea "al volo" il main, quando vuole, tramite new...*
- *...a immagine e somiglianza della classe Counter*



Java e Classi 65

ESEMPIO COMPLETO

```
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!

Java e Classi 66

ESEMPIO: COSTRUZIONE

- 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
- Per compilare: NB: l'ordine non importa
`javac Esempio1.java Counter.java`

Java e Classi 67

ESEMPIO: COSTRUZIONE

- Queste due classi devono essere scritte *in due file distinti*, di nome, rispettivamente:
 - Esempio1.java (contiene la classe Esempio1)
 - Counter.java (contiene la classe Counter)
- Anche separatamente, ma nell'ordine:
`javac Counter.java`
`javac Esempio1.java`
La classe Counter deve infatti già esistere quando si compila la classe Esempio1
- Per compilare:
`javac Esempio1.java Counter.java`

Java e Classi 68

ESEMPIO: ESECUZIONE

- La compilazione di quei due 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
```

Java e Classi 69

ESEMPIO: UNA VARIANTE

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

```
public class Esempio2 {  
    ...  
}  
  
class Counter {  
    ...  
}
```

Importante: l'ordine delle classi nel file è irrilevante, non esiste un concetto di dichiarazione che deve precedere l'uso!

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

Java e Classi 70

ESEMPIO: UNA VARIANTE

- 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

Java e Classi 71

RIFERIMENTI A OGGETTI

- In C si possono definire, per ciascun tipo:
 - sia variabili (es. int x;)
 - sia puntatori (es. int *x;)
- 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;)

Java e Classi 72

RIFERIMENTI A OGGETTI

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

Java e Classi 73

RIFERIMENTI A OGGETTI

Cosa si può fare con i riferimenti?

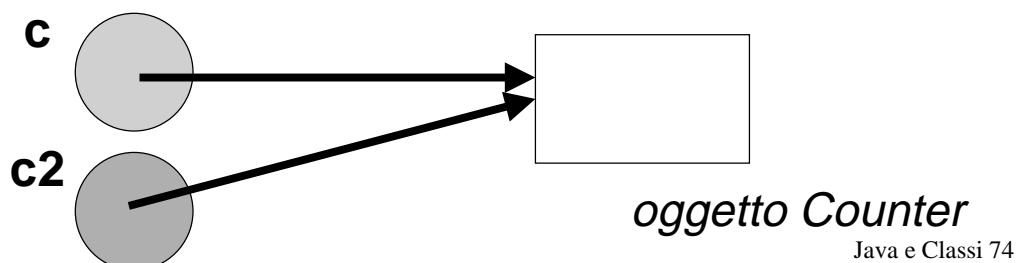
- **Usarli per creare nuovi oggetti:**

`c = new Counter();`

- **Assegnarli uno all'altro:**

`Counter c2 = c;`

In tal caso, l'oggetto referenziato è condiviso!



Java e Classi 74

ESEMPIO

```
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; Ora c2 coincide con c1!  
        c2.inc(); Quindi, se si incrementa c2 ...  
        System.out.println("c1 = " + c1.getValue());  
        System.out.println("c2 = " + c2.getValue());  
    }  
}
```

... risultano incrementati entrambi!

Java e Classi 75

ESEMPIO

```
public class Esempio3 {  
    Novità di Java: le definizioni di  
variabile possono comparire  
ovunque nel programma, non  
più solo all'inizio.  
    public static void main(String[] args){  
        Counter c1 = new Counter();  
        System.out.println("c1 = " + c1.getValue());  
        Counter c2 = c1; Ora c2 coincide con c1!  
        c2.inc(); Quindi, se si incrementa c2 ...  
        System.out.println("c1 = " + c1.getValue());  
        System.out.println("c2 = " + c2.getValue());  
    }  
}
```

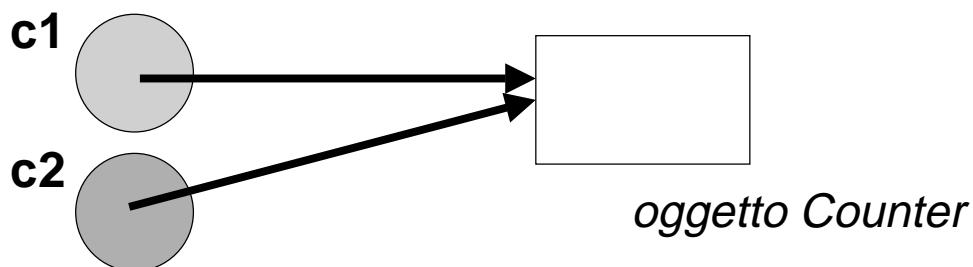
... risultano incrementati entrambi!

Java e Classi 76

UGUAGLIANZA FRA OGGETTI

Quale significato per $c1==c2$?

- $c1$ e $c2$ sono due riferimenti
→ uguali se puntano allo stesso oggetto



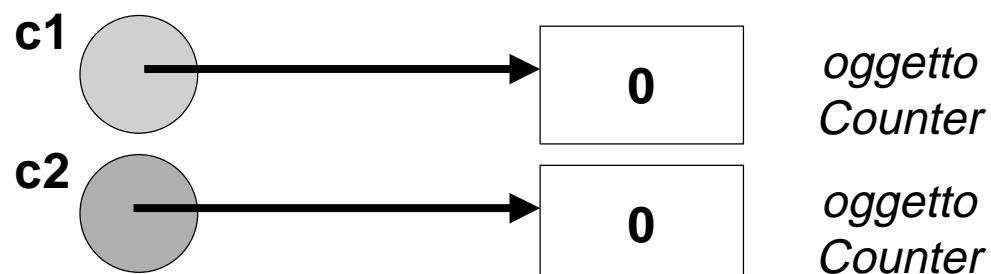
- qui, $c1==c2$ è true

Java e Classi 77

UGUAGLIANZA FRA OGGETTI

E se creo due oggetti identici?

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



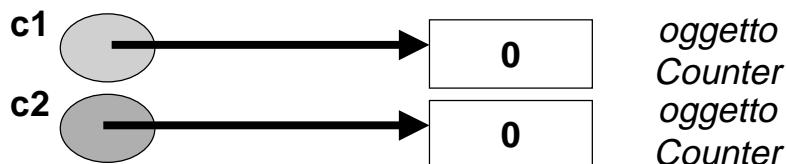
- il contenuto non conta: $c1==c2$ è false !

Java e Classi 78

UGUAGLIANZA FRA OGGETTI

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"

Java e Classi 79

UGUAGLIANZA FRA OGGETTI

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

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

Per impostazione predefinita,
equals() controlla se i
riferimenti sono uguali,
quindi dà lo stesso risultato
di `c1==c2`

Però, mentre `c1==c2` darà
sempre quel risultato, il
comportamento di equals()
possiamo ridefinirlo noi.

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

Java e Classi 80

UGUAGLIANZA FRA OGGETTI

La classe Counter con equals()

```
public class Counter {  
    private int val;  
  
    public boolean equals(Counter x){  
        return (val==x.val);  
    }  
    public void setVal(int v) { val = v; }  
    public int getVal() { return val; }  
}
```

Consideriamo uguali due Counter se e solo se hanno identico valore

Ma ogni altro criterio (sensato) sarebbe stato egualmente lecito!!

Java e Classi 81

PASSAGGIO DEI PARAMETRI

- Come il C, Java passa i parametri alle funzioni *per valore*...
- ... e finché parliamo di *tipi primitivi* non ci sono particolarità da notare...
- ... ma *passare per valore un riferimento* significa passare per riferimento l'oggetto puntato!

PASSAGGIO DEI PARAMETRI

Quindi:

- *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!*

Java e Classi 83

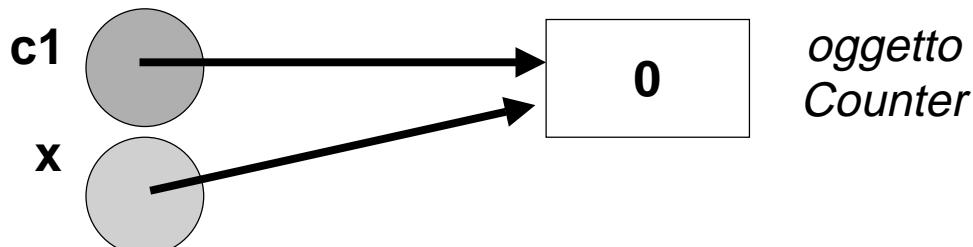
PASSAGGIO DEI PARAMETRI

Esempio:

```
void f(Counter x) { ... }
```

Il cliente:

```
Counter c1 = new Counter();
f(c1);
```



Java e Classi 84

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 oggetti
 - non viene *mai chiamato esplicitamente dall'utente*
 - è invocato automaticamente dal sistema *ogni volta che si crea un nuovo oggetto* di quella classe

Java e Classi 85

COSTRUTTORI

Il costruttore:

- 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**, con diverse liste di parametri
 - servono a inizializzare l'oggetto a partire da *situazioni diverse*

Java e Classi 86

ESEMPIO

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) ...  
}
```

Costruttore senza parametri

Costruttore con un parametro

Java e Classi 87

ESEMPIO: UN CLIENTE

```
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  
    }  
}
```

Qui scatta il costruttore/0
→ c1 inizializzato a 1

Qui scatta il costruttore/1 → c2 inizializzato a 10

Java e Classi 88

COSTRUTTORE DI DEFAULT

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!*

Java e Classi 89

COSTRUTTORI - NOTE

- Una classe destinata a fungere da schema per oggetti *dove definire almeno un costruttore pubblico*
 - in assenza di costruttori pubblici, oggetti di tale classe *non* potrebbero essere costruiti
 - il costruttore di default definito dal sistema è *pubblico*
- È possibile definire costruttori non pubblici per scopi particolari

Java e Classi 90

COSTANTI

- In Java, un simbolo di variabile dichiarato **final** denota una **costante**

```
final int DIM = 8;
```

- Deve obbligatoriamente essere *inizializzata*
- Questo è *il solo modo di definire costanti*
 - infatti, *non esiste preprocessore*
 - *non esiste #define*
 - *non esiste la parola chiave const*
- **Convenzione: nome tutto maiuscolo**

Java e Classi 91

OVERLOADING DI FUNZIONI

- Il caso dei costruttori non è l'unico: in Java è possibile *definire più funzioni con lo stesso nome*, anche dentro alla stessa classe
- L'importante è che le funzioni "omonime" siano comunque distinguibili tramite la lista dei parametri
- Questa possibilità si chiama *overloading* ed è di grande utilità per catturare situazioni simili senza far proliferare nomi inutilmente

Java e Classi 92

