Introduzione alla Programmazione Orientata agli Oggetti

Slides per cortesia del Prof. Denti

Outline

- Dai linguaggi alle infrastrutture software
- Tipi di dato primitivi in Java e C#
- Un esempio concreto: il contatore
- Classi come componenti software: limiti
  - … verso un nuovo scenario: gli oggetti
  - … ancora un contatore …
- Classi come tipi di dato, Oggetti come istanze di classi
- Progettazione Incrementale: Delegazione ed Ereditarietà
- Polimorfismo

Introduzione: dai linguaggi alle infrastrutture software

Introduzione "bottom-up"

"PROGRAMMARE" … ?

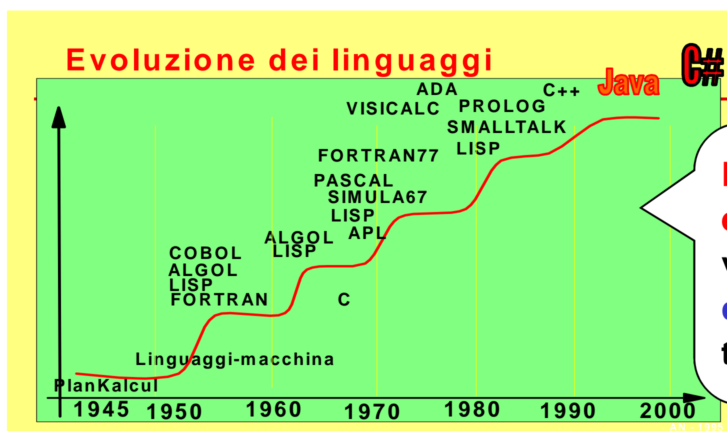
- In passato, "occuparsi di informatica" è stato a lungo sinonimo di "programmare computer"
- Fa pensare a una *attività poco stimolante*, dove le fasi creative – *analisi* e *progetto* – sono *già avvenute*
  - tipico della costruzione di *algoritmi* di medio/piccola dimensione, normalmente già noti e descritti in letteratura
- Anni '80: "crisi del software"
  l'insufficienza delle metodologie e degli strumenti nati per una *visione "algoritmica"* dell'informatica (programmazione *in-the-small)* rispetto alle problematiche di progettazione, sviluppo e manutenzione di *sistemi software complessi (progettazione in the large)*

PROGETTARE SISTEMI SOFTWARE

- Dunque, oggi la frase **"programmare un elaboratore"** può considerarsi un'espressione ***obsoleta***
- **Obiettivo:**
  saper progettare e costruire <u>sistemi</u> software **con il *rigore* e le *garanzie* che si richiedono a tutti gli altri prodotti dell'<u>ingegneria</u> – in primis *qualità* e *affidabilità***
- **MA**
  i linguaggi "classici" (C, Pascal,…) non offrono *strumenti mentali adatti* ad affrontare la progettazione di sistemi software complessi *(problema del "foglio bianco")*
  - **le metafore e i concetti da essi introdotti, nati per esprimere *algoritmi*, *non sono sufficienti* per aiutare a concepire la soluzione a un problema complesso.**

VERSO NUOVI LINGUAGGI

- **I modelli di progettazione e i linguaggi *<u>a oggetti</u>* sono nati per cercare di superare questo limite**
- **Essi introducono *<u>nuovi concetti</u>* e *<u>nuove metafore</u>* per il progetto e lo sviluppo di sistemi software complessi.**

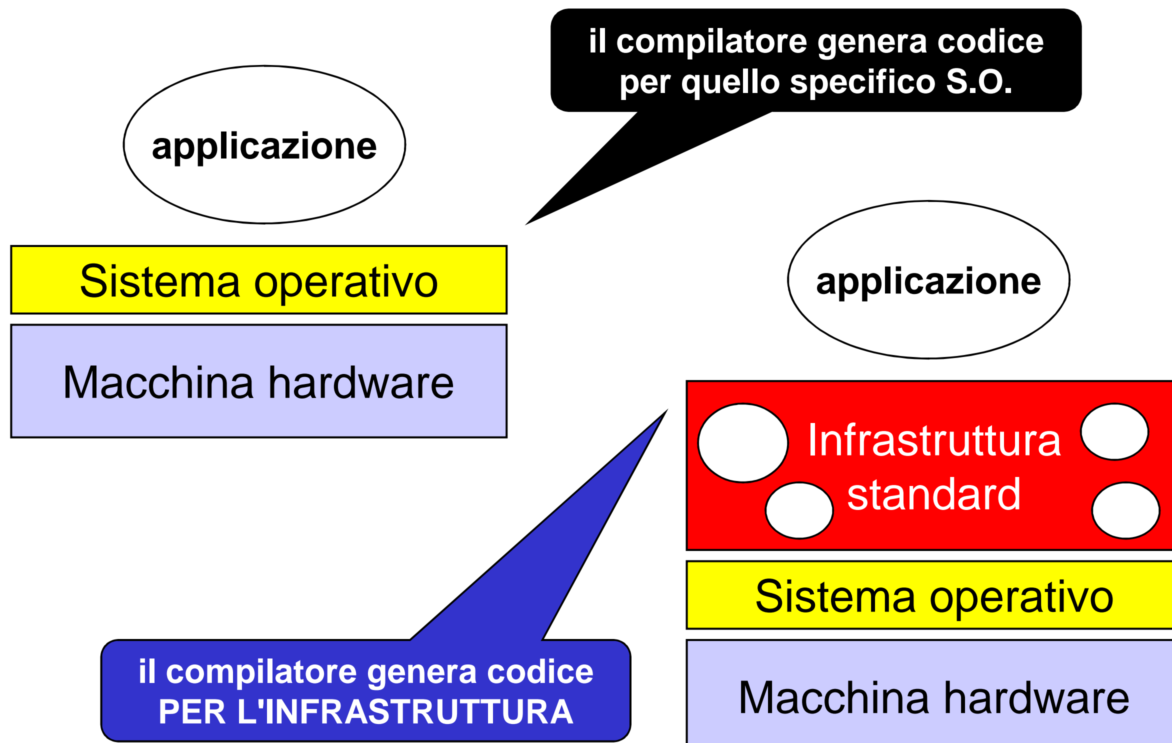


La diversità delle metafore e dei concetti introdotti dai vari linguaggi è la ragione di fondo per l'esistenza di tanti linguaggi.

DAI LINGUAGGI ALLE INFRASTRUTTURE

- MA *un buon linguaggio non basta*
  - L'esperienza (C++ docet..) insegna che poter creare propri componenti e oggetti è importante…
  - …ma per costruire facilmente e velocemente un SISTEMA SOFTWARE COMPLESSO ***non si può pensare di costruire tutto da zero***
  - non si può neanche pensare che ogni azienda/strumento offra ***le proprie*** librerie, fatte secondo il suo personale "gusto"
- occorre una *INFRASTRUTTURA STANDARD*
  - un ambiente che offra *componenti pronti standardizzati,* certamente presenti su ogni installazione
  - che offrano soluzioni pronte per tutti i principali problemi
  - ***uno strato di sicura affidabilità su cui contare e costruire***

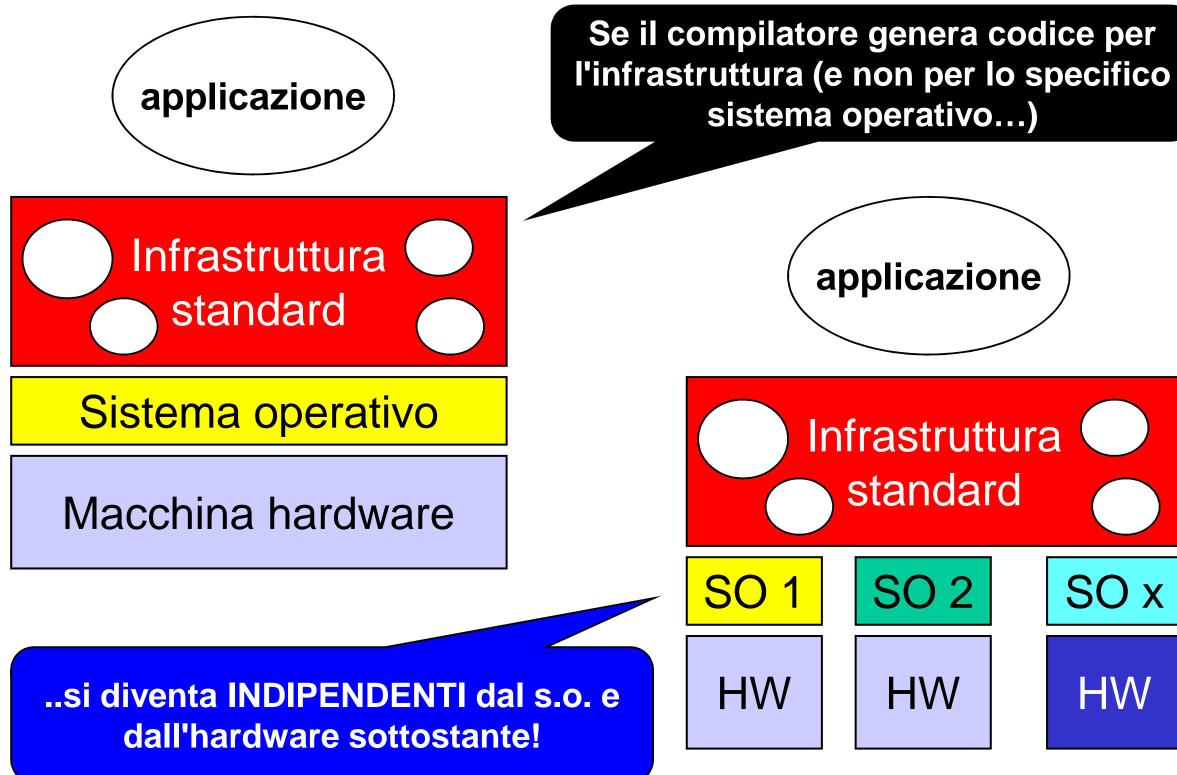
DAI LINGUAGGI ALLE INFRASTRUTTURE



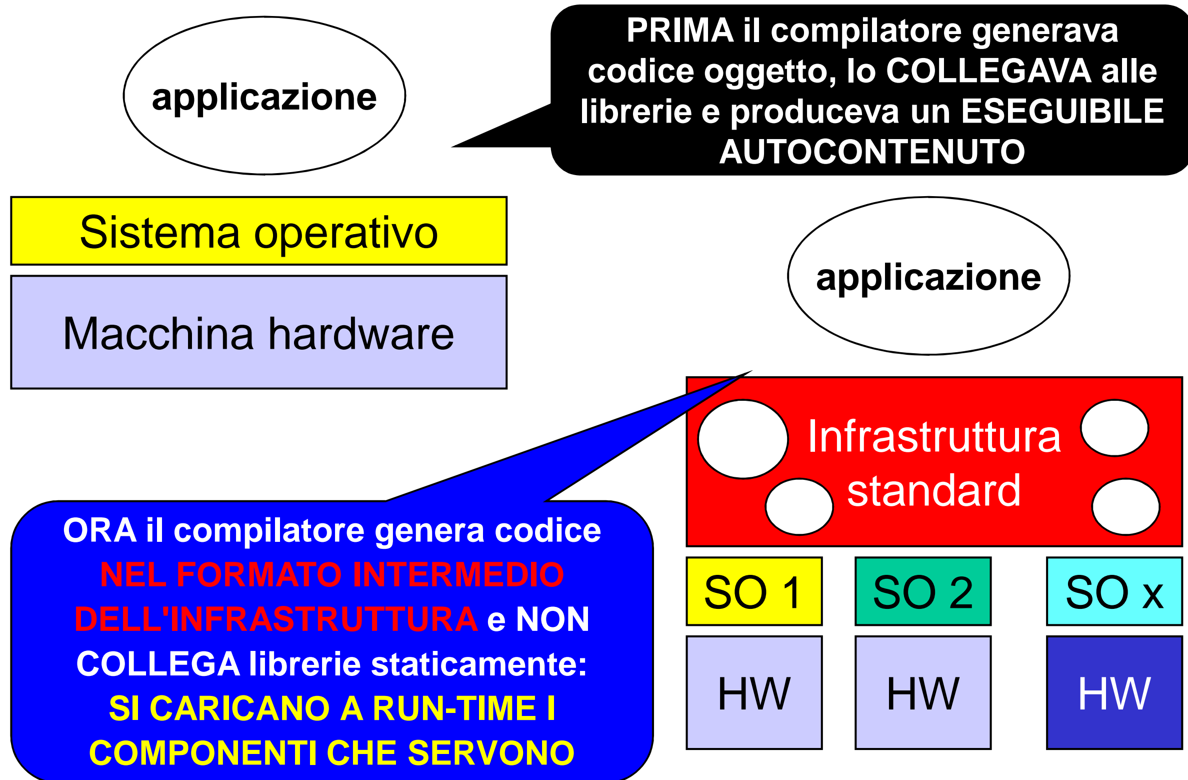
INFRASTRUTTURE INTER-PIATTAFORMA

- Poiché costruire software richiede tempo e risorse, si cerca di *massimizzare la resa* di tale investimento
- Perciò il software non dovrebbe essere specifico di una certa *piattaforma* hardware + software…
- … ma bensì *il più possibile portabile e inter-piattaforma*
- Per questo interessano *INFRASTRUTTURE PORTABILI e INTER-PIATTAFORMA*
  - infrastrutture installabili su diversi sistemi hardware, magari con diversi sistemi operativi e configurazioni software
  - capaci di offrire alle applicazione software costruite sopra di esse un *ambiente noto, ricco e dal funzionamento uniforme*

INFRASTRUTTURE INTER-PIATTAFORMA



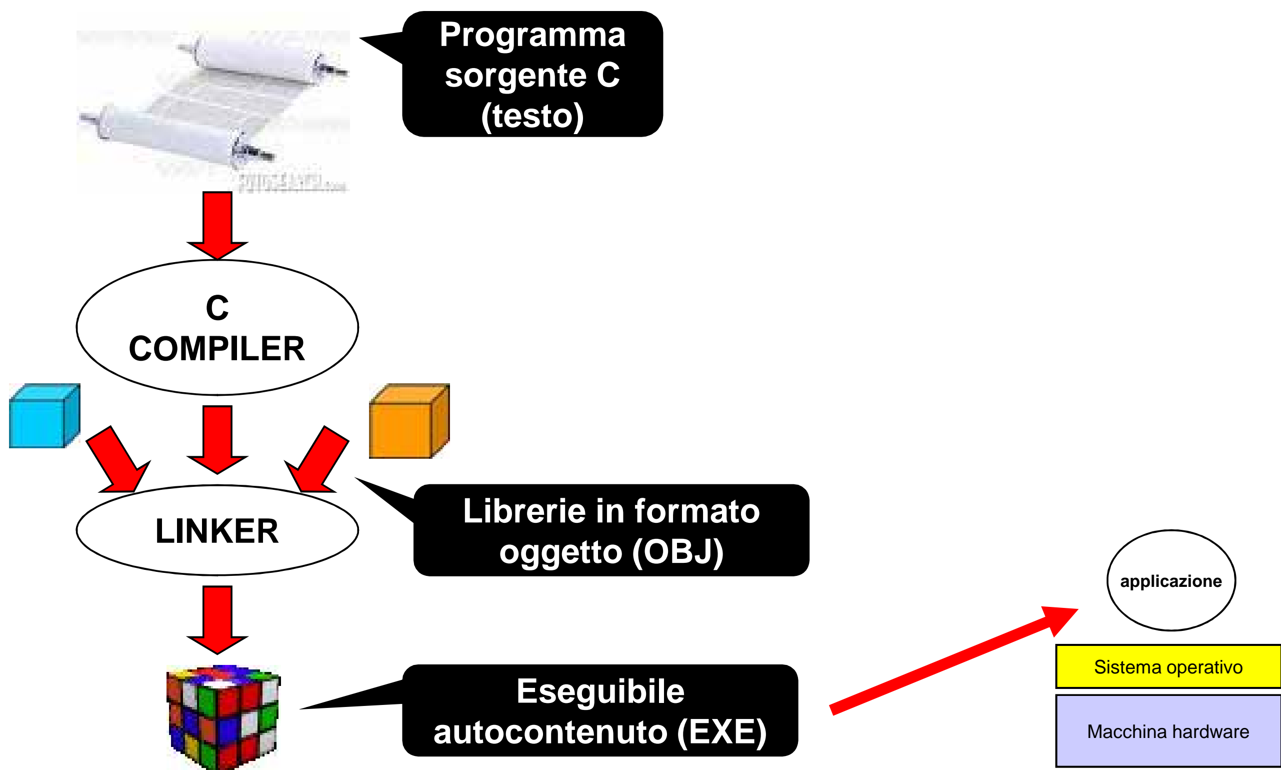
COMPILAZIONE ED ESECUZIONE



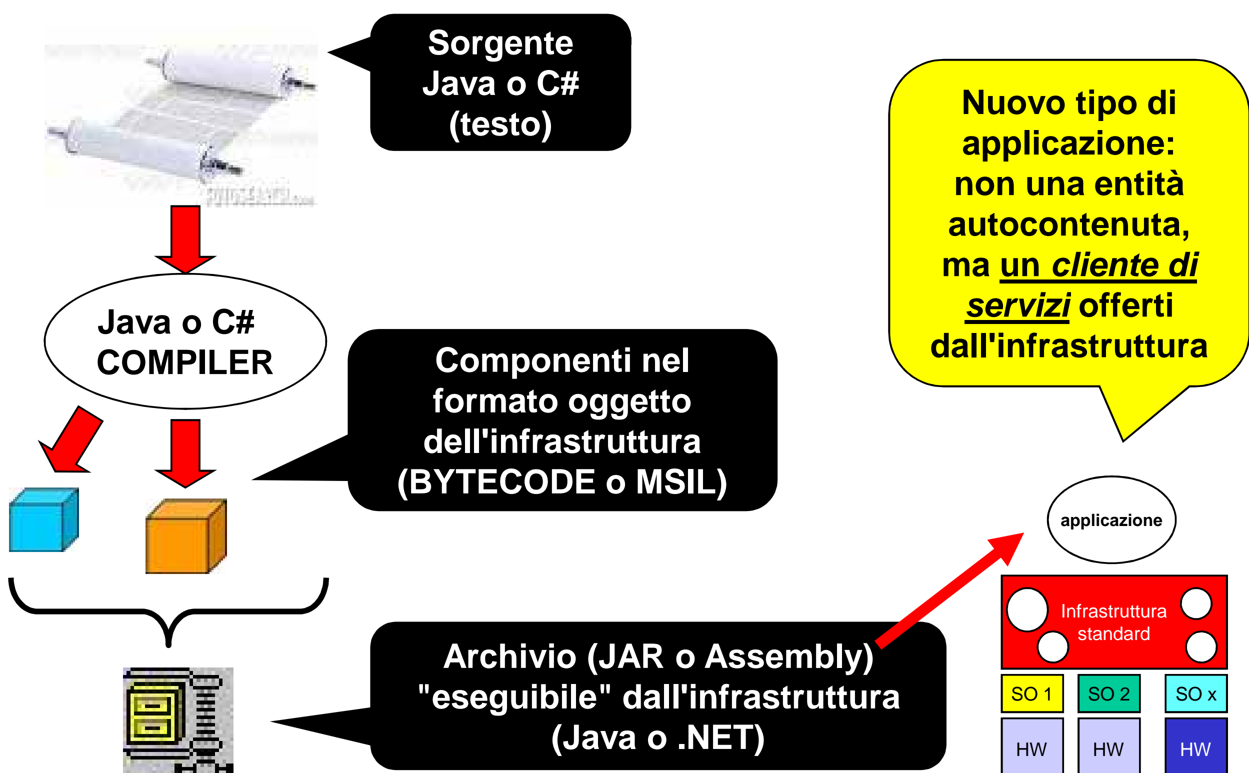
RIASSUMENDO

- **Cambia il processo di <u>costruzione</u> del software**
  - l'obiettivo non è più generare un programma per un certo processore e versione di sistema operativo..
  - ..ma generare qualcosa di MASSIMAMENTE RIUSABILE
- **Cambia il processo di <u>esecuzione</u> del software**
  - non si esegue più un "eseguibile" (EXE) prodotto apposta per un certo processore e sistema operativo..
  - ..ma si FA ESEGUIRE ALL'INFRASTRUTTURA un "prodotto" che non sarebbe eseguibile dal sistema operativo soltanto, poiché non è "autocontenuto": *un ESEGUIBILE PORTABILE*
- **Si paga un prezzo per questo passo extra**
- **Ma i vantaggi sono molto superiori al costo**
  - possibilità di scaricare ed eseguire applicazioni in rete
  - *"write once, run everywhere"*

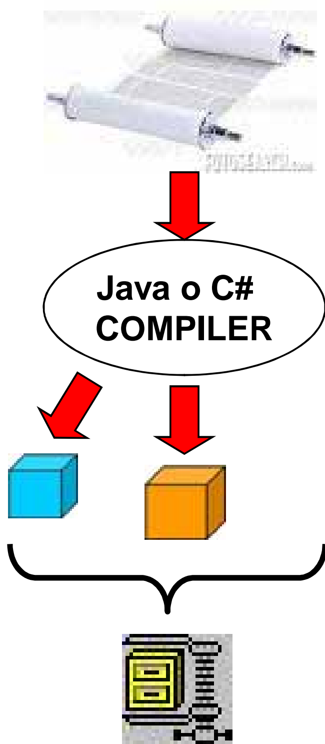
DAL "MONDO C" ...



DAL "MONDO C" AL "NUOVO MONDO"



IL MONDO DI JAVA e C#



- Il sorgente Java o C# è strutturato in una serie di componenti
- Ogni componente viene compilato in un *formato portabile, indipendente da hardware e sistema operativo*
- L'eseguibile può quindi essere compilato su una piattaforma *e funzionare su un'altra, purché dotata della stessa infrastruttura*
- Per consentire ciò, il formato "eseguibile" non è *"eseguibile"* in senso classico
  - un EXE di ".NET" non funziona da solo..
  - …e neanche un JAR di Java
- Occorre uno *STRATO-PONTE nell'infrastruttura,* che *interpreti* il formato portabile adattandolo alla specifica macchina

  → È l'unico strato *dipendente dalla piattaforma.*

IL RUNTIME NELL'INFRASTRUTTURA

- Lo strato infrastrutturale di JAVA va sotto il nome di *JRE (Java Runtime Environment)*
  - ogni utente che voglia ESEGUIRE applicazioni Java deve averlo installato sulla propria macchina
  - molto leggero, disponibile gratuitamente, esiste per tutti i principali sistemi operativi (Windows, Mac, Linux, SunOS…)
  - per SVILUPPARE applicazioni serve il *JDK (Java Development Kit),* che include il compilatore Java e tanti altri strumenti
- Lo strato infrastrutturale di C# va sotto il nome di *.NET FRAMEWORK*
  - ogni utente che voglia ESEGUIRE applicazioni C# deve averlo installato sulla propria macchina
  - disponibile gratuitamente per Windows XP
  - include già il compilatore a riga di comando e altri strumenti

JAVA e C#: IL LINGUAGGIO

- Java e C# sono linguaggi *progettati ex novo,* facendo tesoro delle esperienze (e degli errori) precedenti
- Sintatticamente simili a C e C++, ma *senza il requisito della piena compatibilità all'indietro*
- Obiettivo 1: eliminare meccanismi e costrutti linguistici *poco chiari*, *contorti* e *rischiosi* che causavano errori, sostituendoli con *nuovi meccanismi e costrutti evoluti*
- Obiettivo 2: intercettare a compile-time quanti più errori possibile: *"se si compila, quasi sicuramente è ok"*
  - sostituzione dei puntatori con *riferimenti*
  - dereferenziamento automatico
  - allocazione e deallocazione automatica della memoria (heap)
  - type system molto più stringente + controlli a run-time
  - …

JAVA e C#: L'INFRASTRUTTURA

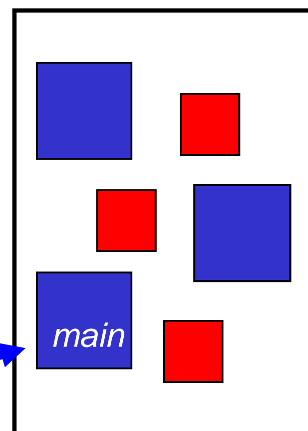
- A differenza di quanto accadeva in C e C++, Java e C# nascono dotati da una *infrastruttura standard portabile*
  - indipendenza dalla piattaforma hardware / software
  - write once, run everywhere
- L'infrastruttura offre servizi per coprire praticamente ogni esigenza del moderno sviluppo di software
  - supporto evoluto per la grafica (Swing, Java2D, 3D..)
  - programmazione a eventi
  - supporto di rete: URL, Socket, …
  - supporto per il multi-threading
  - interfacciamento con database (JDBC)
  - supporto per la sicurezza (cifratura, autenticazione..)
  - esecuzione remota (RMI)…

APPLICAZIONI JAVA e C#: ARCHITETTURA a RUN-TIME

Una applicazione Java o C# è strutturata come *un insieme di componenti,* di cui:

- alcuni sono ***statici***, ossia vengono caricati *prima* dell'inizio del programma ed *esistono per tutta la durata dello stesso*
- altri invece sono ***dinamici,*** ossia vengono *creati durante l'esecuzione* al momento del bisogno.

Poiché ogni applicazione deve avere un punto di partenza prestabilito, *uno dei componenti statici contiene il main*



APPLICAZIONI JAVA e C#: ARCHITETTURA a RUN-TIME

Una applicazione Java o C# è strutturata come *un insieme di componenti,* di cui:

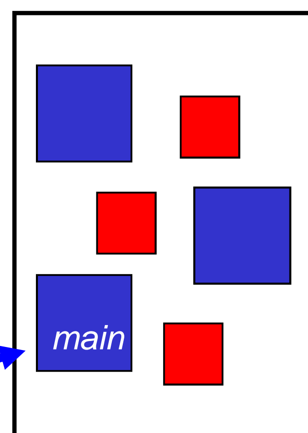
- alcuni sono ***statici***, os… …ricati *prima* dell'inizio del … …*stono per tutta la durata dello* …

CLASSI

- …no ***dinamici,*** ossia …*durante l'esecuzione* a… …del bisogno.

OGGETTI

Poiché ogni applicazione deve avere un punto di partenza prestabilito, *uno dei componenti statici contiene il main*

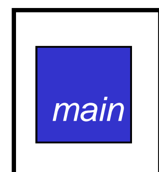


Classi come componenti software in Java e C#

IL MAIN dal C a JAVA / C#

La più semplice applicazione **Java o C# è quindi costituita da** ***una singola classe,*** **che definisce** ***soltanto il*** `main`

PRIMA DIFFERENZA RISPETTO AL C:
- **in C, il main è *semplicemente scritto in un file, non è racchiuso da alcun costrutto linguistico***
- **in Java e C#, *il main* dev'essere scritto dentro una classe *pubblica* ed essere esso stesso *pubblico (protezione)***



SECONDA DIFFERENZA RISPETTO AL C:
- **in C, il `main` può avere (argc/argv) o non avere argomenti**
- **in Java, il `main` ha sempre come argomento un *array di stringhe***
- **in C#, il `Main` può avere come argomento un *array di stringhe***

TERZA DIFFERENZA RISPETTO AL C:
- **in C, il `main` può avere tipo di ritorno `void` o `int`**
- **in Java, il `main` ha sempre tipo di ritorno `void` (NON `int`)**
- **in C#, il `Main` può avere tipo di ritorno `void` o `int`**

IL CASO PIÙ SEMPLICE (1/3)

```
// file MyProg.c
int main(int argc, char argv[]){
   ...
}
```

C

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

obbligatorio `void` — obbligatorio

JAVA

```
public class MyProg {
 public static void Main(string[] args){
   ...
 }
}
```

`void` o `int` — opzionale

C#

IL CASO PIÙ SEMPLICE (2/3)

```
// file MyProg.c
int main(int argc, char argv[]){
   int x=3, y=4; int z= x+y;
}
```

C

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

JAVA

```
public class MyProg {
 public static void Main(string[] args){
   int x=3, y=4; int z= x+y;
 }
}
```

C#

IL CASO PIÙ SEMPLICE (3/3)

COMPILAZIONE C

`C:> cc MyProg.c` *produce MyProg.exe*

L'EXE ottenuto è eseguibile sul sistema operativo

COMPILAZIONE JAVA

`C:> javac MyProg.java` *produce MyProg.class*

Il file ottenuto è eseguibile sull'infrastruttura Java

COMPILAZIONE C#

`C:> csc MyProg.cs` *produce MyProg.exe*

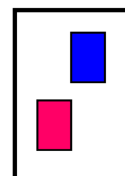
Il file ottenuto è un EXE eseguibile sull'infrastruttura .NET

Non sono la stessa cosa!

UN ESEMPIO UN PO' PIÙ COMPLESSO

Un programma costituito da due classi:

- la nostra `Esempio1`, che definisce il `main`
- una classe di sistema



Obiettivo:

- stampare a video la classica frase di benvenuto sfruttando il servizio di stampa fornito dal "sistema"

INVERSIONE DEL PUNTO DI VISTA

In C, l'enfasi è sull'operazione, l'entità che la svolge è un parametro:

```
fprintf(fout,"Hello!");
```

In JAVA e C#, l'enfasi è su chi svolge un *servizio*:

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

C# `System.Console.WriteLine("Hello!");`

NOTAZIONE PUNTATA

- In Java e C#, la notazione puntata non indica più solo la selezione di un campo dati, come nelle `struct` del C
- Indica anche *l'invocazione di una operazione* messa a disposizione da un componente, ossia un *metodo* per richiedere un certo *servizio*

IL CASO DI JAVA

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

Si richiede al componente `System.out` di svolgere il servizio `println` (`out` *è a sua volta un componente della classe di sistema* `System`)

IL CASO DI C#

```
System.Console.WriteLine("Hello!");
```

Si richiede al componente `System.Console` di svolgere il servizio `WriteLine` (`Console` *è una classe di sistema del namespace* `System`)

RIPRENDENDO L'ESEMPIO…

C

```
<include stdio.h>
int main(int argc, char argv[]){
   printf("%s", "Hello world!");
}
```

JAVA

```
public class Esempio1 {
 public static void main(String[] args){
    System.out.println("Hello world!");
 }
}
```

C#

```
public class Esempio1 {
 public static void Main(string[] args){
    System.Console.WriteLine("Hello world!");
 }
}
```

IL NUOVO ESEMPIO: COMPILAZIONE

COMPILAZIONE C

```
C:> cc Esempio1.c
```
produce Esempio1.exe

L'EXE ottenuto è eseguibile sul sistema operativo

COMPILAZIONE JAVA

```
C:> javac Esempio1.java
```
produce Esempio1.class

Il file ottenuto è eseguibile sull'infrastruttura Java

Non sono la stessa cosa!

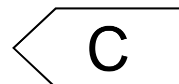
COMPILAZIONE C#

```
C:> csc Esempio1.cs
```
produce Esempio1.exe

Il file ottenuto è un EXE eseguibile sull'infrastruttura .NET

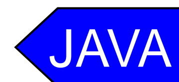
IL NUOVO ESEMPIO: ESECUZIONE

```
C:> Esempio alfa beta gamma
alfa
```



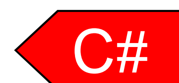
L'EXE ottenuto è eseguito direttamente sul sistema operativo

```
C:> java Esempio alfa beta gamma
alfa
```



Il file ottenuto è eseguito sull'infrastruttura Java
NOTA: occorre invocare esplicitamente l'interprete Java (strato-ponte)

```
C:> Esempio alfa beta gamma
alfa
```



Il file ottenuto è eseguito sull'infrastruttura .NET

NOTA: sembra uguale al primo.. *ma non funziona se sulla macchina non è installato il Microsoft .NET Framework*

JAVA: UNA CONVENZIONE

- <u>In Java</u> vi è *OBBLIGO di corrispondenza* fra:
  - *nome di una classe pubblica*
  - *nome del file in cui essa dev'essere definita*
- Una classe pubblica <u>deve</u> essere definita in un file *con lo stesso nome della classe* ed estensione `.java`
- Esempi

Essenziale *rispettare maiuscole / minuscole!*

classe `EsempioBase` → file `EsempioBase.java`
classe `Esempio0` → file `Esempio0.java`

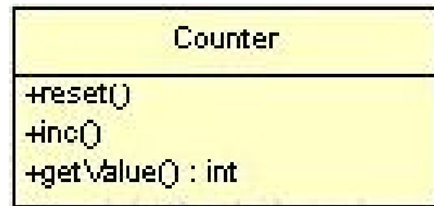
- <u>In C#</u> non esiste una regola *imperativa* analoga, tuttavia seguirla è *fortemente consigliato.*

Un esempio concreto:
il contatore

UN ESEMPIO: UN "CONTATORE"

Un ***contatore*** può essere definito un componente caratterizzato da un valore (intero, variabile nel tempo) e tre *operazioni pubbliche:*

- `reset()` **per impostare il contatore a un valore iniziale noto (zero o altro)**
- `inc()` **per incrementare il valore attuale del contatore**
- `getValue()` **per recuperare il valore attuale del contatore sotto forma di numero intero**



Non importa come è realizzato *dentro*: importa il suo *COMPORTAMENTO OSSERVABILE*

UNA POSSIBILE REALIZZAZIONE IN C

- **Definire un *contatore tramite un file***
- **Stato del contatore *inaccessibile dall'esterno***
  ossia fuori dal file di definizione → keyword `static`
- **Accesso allo stato possibile solo attraverso apposite *funzioni (operazioni) pubbliche* :**

```
void reset(void);
void inc(void);
int  getValue(void);
```

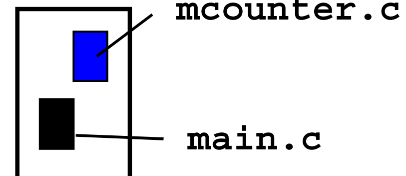
`mcounter.h`

- **Per USARE il contatore, si deve:**
  - **aggiungere questo modulo al progetto dell'applicazione**
  - **includere nel file cliente il corrispondente HEADER che dichiari le tre operazioni sopra citate.**

UN POSSIBILE CLIENTE

```
/* mymain.c */
#include <stdio.h>
#include "mcounter.h"
main() {
  reset(); inc();
  printf("%d", getValue() );
}
```

```
void reset(void);
void inc(void);
int getValue(void);
```



NOTE:

- **Non occorre creare esplicitamente il contatore**, perché esso **coincide col modulo** `mcounter.c` che fa parte del progetto
- Le operazioni non hanno parametri perché è implicito su quale contatore agiscono – l'unico presente!

UNA POSSIBILE REALIZZAZIONE IN C

mcounter.h

```
void reset(void);
void inc(void);
int  getValue(void);
```

mcounter.c

```
static int stato;
void reset(void){stato=0;}
void inc(void){stato++;}
int getValue(void){return stato;}
```

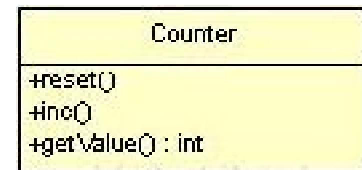
Stato *inaccessibile dall'esterno del file*

IL “CONTATORE” COME CLASSE

Lo stesso progetto di *contatore* può essere realizzato sotto forma di componente software (classe) Java o C#

- Lo stato sarà *protetto* dalla keyword `private`
- L'accesso allo stato sarà possibile solo attraverso le *funzioni (operazioni) pubbliche* :

```
void reset();
void inc();
int  getValue();
```



- Per USARE il contatore, basterà:
  - compilare la classe e tenerla nella stessa directory del cliente
  - senza necessità di includere alcun header nel file client

UNA POSSIBILE REALIZZAZIONE JAVA o C#

Contatore.java (o .cs)

```
public class Contatore {
  private static int stato;
  public static void reset(){stato=0;}
  public static void inc(){stato++;}
  public static int getValue(){return stato;}
}
```

Stato *inaccessibile dall'esterno della classe Contatore*

NOTE:

- il livello di protezione (privato/ pubblico) è espresso esplicitamente
- ora è un COSTRUTTO LINGUISTICO (la classe) a racchiudere in un'unica entità lo stato del contatore e le operazioni che agiscono su esso, *non più solo un contenitore fisico (il file)*
- è tutto statico, perché si tratta di un componente software che deve esistere per tutto il tempo di vita del programma.

UN POSSIBILE CLIENTE

MyMain.java (o .cs)

```
public class MyMain {
 public static void main(String[] args) {
   Contatore.reset(); Contatore.inc();
   System.out.println(Contatore.getValue());
 }
}
```

C#: string (minuscolo)

C#: System.Console.WriteLine

NOTE:

- Non occorre creare esplicitamente il contatore, perché esso coincide con la classe Contatore che è disponibile nel progetto
- Le operazioni non hanno parametri perché è implicito su quale contatore agiscono – l'unico presente, cioè Contatore medesima
- Si usa la "notazione puntata" per invocare funzioni pubbliche (METODI) del componente software Contatore

UNA POSSIBILE REALIZZAZIONE JAVA o C#

Contatore.java (o .cs)

```
public class Contatore {
  private static int stato;
  public static void reset(){stato=0;}
  public static void inc(){stato++;}
  public static int getValue(){return stato;}
}
```

Lo stato è privato e perciò *inaccessibile dall'esterno della classe Contatore*

Le signature delle funzioni sono *pubbliche* → accessibili da fuori

Il corpo delle funzioni è comunque inaccessibile da fuori

NOTE:

- il livello di protezione (privato/ pubblico) è espresso esplicitamente
- ora è un COSTRUTTO LINGUISTICO (la classe) a racchiudere in un'unica entità lo stato del contatore e le operazioni che agiscono su esso, *non più solo un contenitore fisico (il file)*
- è tutto statico, perché si tratta di un componente software che deve esistere per tutto il tempo di vita del programma.

COLLEGAMENTO STATICO vs. DINAMICO

In C:

- si compila ogni file sorgente
- *si collegano (link) i file oggetto*
- si crea un *eseguibile* che non contiene più riferimenti esterni
- *max efficienza…e max rigidità: ogni modifica comporta il rebuild dell'eseguibile*

In tale schema:

- il compilatore accetta l'uso delle entità esterne "con riserva di verifica"
- il linker verifica la presenza delle definizioni risolvendo i *riferimenti incrociati* fra i file

In Java o C#:

- non esistono dichiarazioni
- non esiste linker: si compilano le classi, ottenendo un *insieme di .class* o di *assembly*
- *si esegue la classe pubblica che contiene il `main`*
- *in caso di modifiche a una classe, basta ricompilare quella*

In questo nuovo schema:

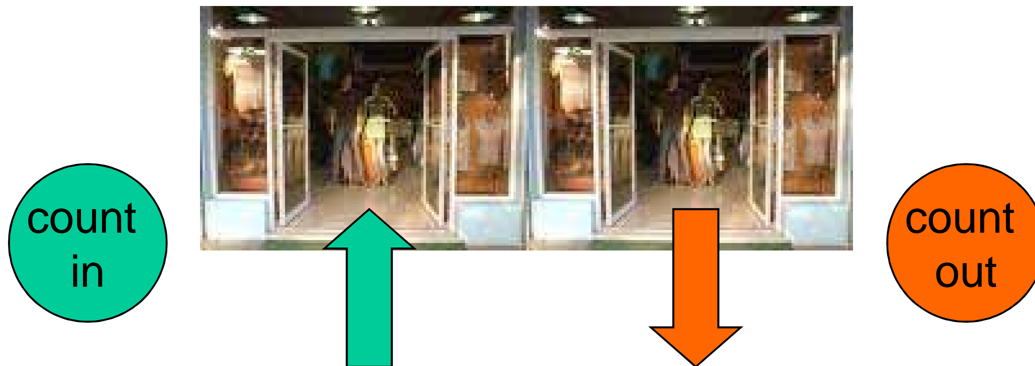
- il compilatore verifica subito il corretto uso delle altre classi (*sa dove trovarle nel file system*)
- le classi vengono *caricate e collegate al momento dell'uso*

Classi come componenti software: limiti

UNO SCENARIO PROBLEMATICO (1/2)

Si supponga di voler *contare le persone in entrata* e le *persone in uscita* da un museo o da un grande magazzino

- **A livello di principio, il problema è facile: *basta avere due contatori* attivati ciascuno da un sensore sulla porta**



UNO SCENARIO PROBLEMATICO (2/2)

Dunque, dovrebbe essere altrettanto semplice costruire il sistema software…

… MA *il nostro contatore è un pezzo unico!*

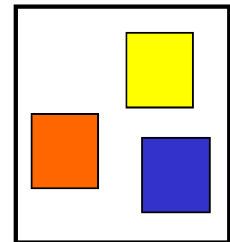
- ***Non possiamo averne due,* perché non possiamo includere due volte lo stesso componente nel progetto!**
  - **il C non consente che un modulo (file) sia presente due volte**
  - **.. e anche in Java e C# il nome di una classe dev'essere unico: la stessa classe non può comparire due volte nell' applicazione!**

…E ALLORA ??

CLASSI COME COMPONENTI SOFTWARE: LIMITI

- **Come componenti software, le classi sono certamente un passo avanti rispetto ai "file" (moduli) del C**
  - **protezione (parte pubblica vs. parte privata)**
  - **costrutto linguistico che delimita il componente**
  - **collegamento e caricamento dinamico**

- **Tuttavia, sono componenti *statici,* che *possono esistere* in un'applicazione solo *in copia unica***
  - **una classe o fa parte di un'applicazione, o non ne fa parte**
- **Questo spesso non basta:**
  - **può andare bene per LIBRERIE…**
  - **…o per componenti "SINGLETON"…**
  - **ma non per componenti "GENERAL PURPOSE", che spesso nei sistemi servono in *copie multiple.***

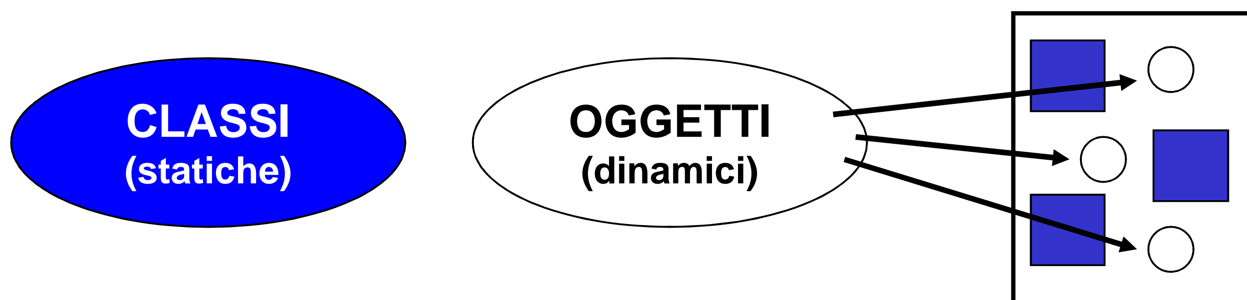


Verso un nuovo scenario

VERSO NUOVI COMPONENTI SOFTWARE

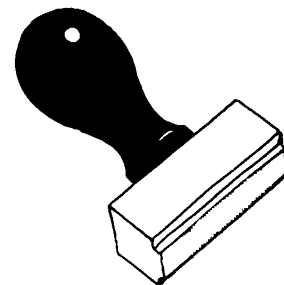
Dunque, una applicazione realistica difficilmente può essere fatta solo di classi, in quanto componenti statici.

Servono *componenti software DINAMICI:*
- **di cui si possano avere *più copie…***
- **…che possano essere *create durante l'esecuzione* al momento del bisogno…**
- **…magari sulla base di un "template" prestabilito.**



IL CONCETTO DI OGGETTO

- **Un OGGETTO è un concettualmente un componente software dinamico, creato sulla base di un "modello"**
- **Tale "modello" assomiglia a un *timbro: definisce le caratteristiche degli oggetti creati a sua immagine.***



Dunque, gli oggetti creati a immagine e somiglianza di un certo "timbro" sono tutti *simili.*
In particolare, condividono:
- **la stessa *struttura interna***
- **le stesse *operazioni***
- **lo stesso *funzionamento***

ma ciascuno ha la propria *identità.*

TIPI DI OGGETTI

- **Un classico modo (anche se non l'unico) per procurarsi un "timbro" è *definire un TIPO DI DATO***
- **Così, facendo, gli oggetti diventano ISTANZE di quel tipo**
  - I linguaggi di programmazione che seguono questo approccio offrono costrutti per permettere di definire i propri tipi di dato…
  - … ma *L'ESPRESSIVITÀ PUÒ VARIARE MOLTO!*
    - alcuni linguaggi hanno costrutti che consentono di definire *solo campi-dati*, altri permettono di specificare *anche le operazioni* su tali dati
    - alcuni permettono di esprimere *forme di protezione* dei dati e/o delle operazioni, mentre altri non offrono tale possibilità
    - alcuni hanno un sistema di tipi "forte", altri più debole
    - alcuni permettono di esprimere anche *relazioni fra tipi*
    - alcuni permettono di definire *operatori* sui propri tipi
    - ecc ecc

TIPI DI OGGETTI

- **Un classico modo (anche se non l'unico) per procurarsi un "timbro" è *definire un TIPO DI DATO***
- **Così, facendo, gli oggetti diventano ISTANZE di quel tipo**
  - I linguaggi di programmazione che seguono questo approccio offrono costrutti per permettere di definire i propri tipi di dato…
  - … ma *L'ESPRESSIVITÀ PUÒ VARIARE MOLTO!*
    - alcuni linguaggi hanno costrutti che consentono di definire *solo campi-dati*, altri permettono di specificare *anche le operazioni* su tali dati
    - alcuni permettono di esprimere *forme di protezione* dei dati e/o delle operazioni, mentre altri non offrono tale possibilità
    - alcuni hanno un sistema di tipi "forte", altri più debole
    - alcuni permettono di esprimere anche *relazioni fra tipi*
    - alcuni permettono di definire *operatori* sui propri tipi
    - ecc ecc

C

Java, C#

Java, C#

C

Java, C#

C

Java, C#

C#

OGGETTI come ISTANZE DI TIPI

- Avendo a disposizione la definizione di un TIPO, si possono *ISTANZIARE* tanti *OGGETTI di quel tipo* quanti ne occorrono
  - si risolve il problema dei componenti software in copia unica
- In un sistema di tipi "ben fatto", la *struttura interna degli oggetti* dovrebbe essere *protetta* dall'esterno
  - i clienti dovrebbero poter accedere agli oggetti *SOLO TRAMITE LE OPERAZIONI PUBBLICHE APPOSITAMENTE FORNITE, mai tramite accesso diretto ai dati*
  - sacri principi: *ENCAPSULATION, INFORMATION HIDING*

Che forma assume tutto questo in C, Java, C# ?

TIPI DI DATO ASTRATTO (ADT)

Un *tipo di dato astratto (ADT)* definisce una categoria concettuale con le sue proprietà:

- *definizione di tipo* su un dominio D
- *insieme di operazioni ammissibili* su tale dominio.

Nel linguaggio C, gli ADT si definiscono tramite il costrutto linguistico `typedef`

- si possono creare tante entità di quel tipo quante ne servono
- tali entità sono espresse tramite VARIABILI di quel tipo

`typedef`, però, *non supporta l'incapsulamento*

- la struttura interna dell'ADT, pur lasciata concettualmente sullo sfondo, è in realtà perfettamente visibile e nota a tutti perché *i file header vanno inclusi ovunque l'ADT venga usato*
- non vi è dunque alcuna possibilità di *impedire usi errati* degli oggetti, perché l'accesso è *sostanzialmente libero!*

TIPI DI DATO ASTRATTO in C: BILANCIO

Definire gli ADT tramite `typedef` è possibile, ma:

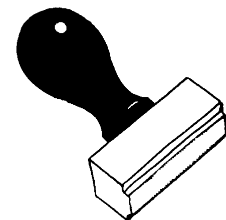
- non c'è unitarietà fra parte-dati (espressa da `typedef`) e parte-operazioni (scritte successivamente e altrove)
- non c'è protezione dall'uso improprio, perché tutti vedono `typedef` e dunque *tutti possono aggirarla*
- le signature delle operazioni fanno trasparire dettagli (puntatori…) che non dovrebbero entrare in gioco a questo livello.

CONCLUSIONE:

LIVELLO DI ESPRESSIVITÀ INADEGUATO

TIPI DI DATO in JAVA e C#

- Un OGGETTO è un concettualmente un componente software dinamico, creato sulla base di un "modello"
- Tale "modello" assomiglia a un *timbro: definisce le caratteristiche degli oggetti creati a sua immagine.*
- Un classico modo per procurarsi un "timbro" è *definire un TIPO DI DATO*



Nel linguaggi a oggetti come Java e C#, i tipi sono espressi tramite CLASSI

- lo stesso costrutto linguistico che già conosciamo e abbiamo usato *per definire componenti software statici…*
- …è usato qui per uno scopo diverso

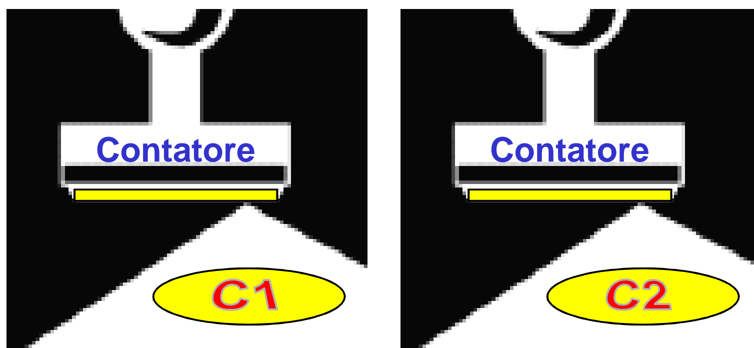
Classi come tipi di dato

Oggetti come istanze di classi

CLASSI come TIPI DI DATO

- Oltre a costituire un componente software (statico), **il costrutto linguistico `class`** può esprimere la **definizione di un *tipo di dato***
  - basta **OMETTERE LA keyword `static`** dalle definizioni di dati e funzioni.

- A quel punto, la classe può fungere da "timbro" per istanziare oggetti:



IL CONTATORE: dalla versione in C…

Riconsideriamo una delle realizzazioni del contatore in C:

contatore.h
```
typedef struct {int val;} contatore;
void reset(contatore *pc);
void inc(contatore *pc);
int  getValue(contatore c);
```

contatore.c
```
#include <contatore.h>
void reset(contatore *pc){ pc -> val=0; }
void inc(contatore *pc) { (pc -> val)++;}
int  getValue(contatore c){return c.val;}
```

include typedef

…alla versione JAVA e C#

In Java c C#, si uniscono nell'unico costrutto *CLASSE*

- la definizione di ADT
- le funzioni che su esso opereranno

specificando altresì il *livello di protezione* di ciascuna.

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

Dati

Operazioni

Nel corpo delle funzioni si usa direttamente il dato val definito sopra: *non occorre più passarlo come parametro*

…alla versione JAVA e C#

Counter.java (o .cs)

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

val è *privato* e come tale accessibile *solo alle operazioni definite nella classe*

Le signature delle funzioni sono *pubbliche* → accessibili da fuori

Il corpo delle funzioni è comunque inaccessibile da fuori

CLASSI in JAVA e C#

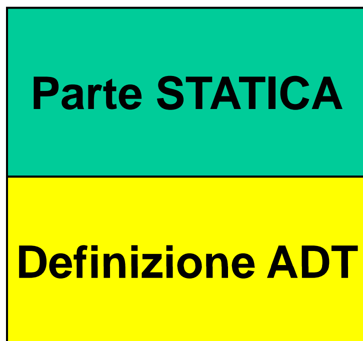
Una *CLASSE JAVA o C#* è dunque un costrutto linguistico che riunisce *ruoli svolti tradizionalmente da costrutti separati.*

- Con la parte statica consente di definire *componenti software* dotati di propri *dati* e *operazioni* (`static`)
  - ANALOGIA: i file usati come *moduli* del C (ma i file non sono un costrutto linguistico…!)
- Con la parte di definizione di tipo consente di definire *tipi di dato astratto* dotati di propri *dati* e *operazioni*
  - ANALOGIA: la `typedef...struct` del C

… il tutto con idonei *meccanismi di protezione*

- assenti in C per i tipi di dato astratto definiti tramite `typedef`
- ANALOGIA: l'uso della parola chiave *static* del C per rendere dati e funzioni invisibili fuori dal proprio file di definizione.

CLASSI in JAVA e C#



PARTE STATICA:
- Definisce un componente software
- *I dati e le operazioni della classe* intesa come componente software sono qualificati `static`

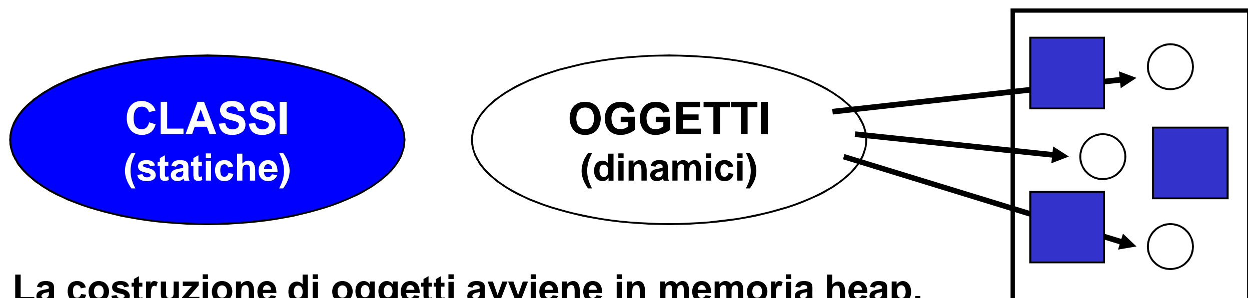
PARTE DI DEFINIZIONE DI TIPO:
- Definisce un tipo, un "timbro"
- usabile per *creare oggetti* fatti a immagine somiglianza del "timbro"

Spesso una classe Java ha una sola delle due parti, perché svolge uno solo di questi due ruoli.

OGGETTI come ISTANZE DI CLASSI

Gli *OGGETTI* sono *componenti software DINAMICI,* creati a immagine e somiglianza di una *CLASSE*

- si possono creare tutte le *istanze* che servono
- la creazione avviene al momento del bisogno, *durante l'esecuzione*



La costruzione di oggetti avviene in memoria heap, analogamente alle strutture dati dinamiche del C.

OGGETTI come ISTANZE DI CLASSI

- Gli OGGETTI sono componenti *dinamici, creati sul momento* tramite l'operatore `new`
  - agisce similmente alla `malloc` del C
- Strutturalmente, ogni oggetto *è composto dai dati specificati dalla sua classe*
  - in modo simile alle variabili struttura del C
- Su ogni oggetto si possono invocare *le operazioni pubbliche* previste dalla sua classe
- NOVITÀ: non occorre occuparsi della distruzione degli oggetti e della deallocazione della memoria, c'è il *garbage collector.*

UN MAIN CHE MANIPOLA CONTATORI

```
public class MyMain{
 public static void main(String[] args) {
    int v1, v2;
    Counter c1, c2;
    c1 = new Counter();
    c2 = new Counter();
    c1.reset(); c2.reset();
    c1.inc(); c1.inc(); c2.inc();
    v1 = c1.getValue(); v2 = c2.getValue();
    System.out.println(v1);
    System.out.println(v2);
 }
}
```

C#: `Main`

C#: `string`

RIFERIMENTI anziché puntatori

creazione oggetti in memoria dinamica

C#: adattare

deallocazione di memoria automatica, a cura del garbage collector

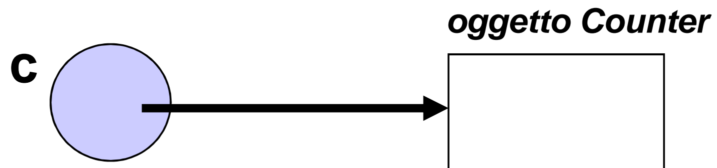
CREAZIONE DI OGGETTI

Per creare un oggetto:

- <u>prima</u> si definisce un *riferimento,* il cui tipo è *il nome della classe che fa da modello*
- *<u>poi</u> si crea dinamicamente l'oggetto* tramite *l'operatore `new`*

Esempio:

```
Counter c;
```




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

La distruzione degli oggetti invece è automatica

- il Garbage Collector elimina gli oggetti non più referenziati
- si eliminano i rischi relativi all'errata deallocazione della memoria

Costruzione di oggetti

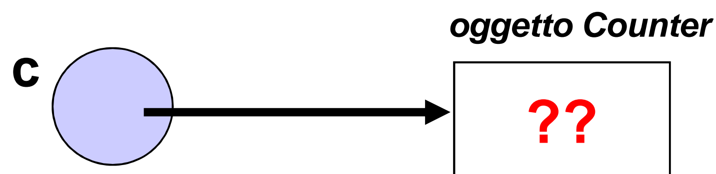
CREAZIONE DI OGGETTI vs. COSTRUZIONE DI OGGETTI

Finora, abbiamo CREATO oggetti:

- **definendo prima un *riferimento* del tipo opportuno**
- ***creando poi l'oggetto* tramite *l'operatore* `new`**

Esempio:

```
Counter c;
```




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

Tuttavia, l'oggetto così creato *non è stato inizializzato:* che valore ha questo `Counter`??

COSTRUZIONE DI OGGETTI

- **Più in generale, molti errori nel software sono causati tradizionalmente da *mancate inizializzazioni* di variabili.**
- **Per ovviare a questo problema, praticamente tutti i linguaggi a oggetti introducono il COSTRUTTORE: *un metodo particolare che automatizza l'inizializzazione***
  - **non viene *mai invocato esplicitamente dall'utente***
  - **è invocato automaticamente *ogni volta che si crea un nuovo oggetto* di una data classe: non si può evitare, né dimenticare!**
- **Per questo, *COSTRUIRE* è *più che CREARE***
  - **CREARE ricorda l'azione-base del "riservare memoria"**
  - **COSTRUIRE denota una attività più ampia, mirata a *"confezionare" un oggetto completo, pronto per l'uso.***

COSTRUTTORI

Il costruttore:

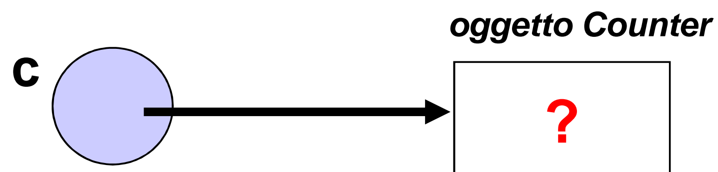
- ha un <u>nome fisso</u>, *uguale al nome della classe*
- non ha tipo di ritorno, neppure `void`
  - il suo scopo infatti non è “calcolare qualcosa”, ma <u>*inizializzare un oggetto*</u>
- può *non essere unico*
  - spesso vi sono *più costruttori*, che servono a inizializzare l’oggetto a partire da *situazioni diverse*
  - tali costruttori <u>si differenziano in base alla lista dei parametri</u>
- esiste sempre: in mancanza di una definizione esplicita, il compilatore inserisce un *construttore di default*
  - *è quello che* <u>*è scattato in automatico negli esempi precedenti*</u>
  - non fa quasi nulla (<u>non</u> inizializza i campi dati a zero!)
  - invoca soltanto un "costruttore predefinito"

COSTRUZIONE DI DEFAULT

Questo oggetto è stato

- *CREATO esplicitamente* da noi tramite `new`
- *COSTRUITO implicitamente* dal *costruttore di defaut*

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



- Il *costruttore di default* viene inserito dal compilatore <u>*solo se la classe non prevede esplicitamente alcun costruttore*</u>
- Poiché il costruttore di default inserito in automatico dal compilatore <u>non inizializza le variabili,</u> è *opportuno definirne sempre uno esplicitamente* che inizializzi i dati ai valori iniziali opportuni.

`Counter` CON COSTRUTTORI

```
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 di default personalizzato

Costruttore con un parametro

ESEMPIO DI COSTRUZIONE

```
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

COSTRUTTORI... PUBBLICI?

- Per poter istanziare oggetti, *deve esistere almeno un costruttore <u>pubblico</u>*
  - in assenza di costruttori pubblici, è impossibile istanziare oggetti di quella classe
  - il costruttore di default definito dal sistema ovviamente è *pubblico*
- È possibile definire costruttori non pubblici per scopi particolari
  - l'ereditarietà fa a volte uso di costruttori "protetti"
  - qualche volta si trovano classi con un costruttore privato *proprio per impedire che se ne creino istanze*

LA KEYWORD `this` (1)

- È possibile che un parametro di un metodo sia *omonimo* con il nome di un dato della classe?
- Ad esempio,se `Counter` contiene un campo `val`, può un costruttore avere un parametro di nome anch'esso `val`? E se sì, come si risolve questa ambiguità?

<u>RISPOSTA</u>:
sì, può esistere un parametro omonimo
L'ambiguità si risolve con la parola chiave `this`

```
public Counter(int val){ this.val = val; }
```

Si può usare nei costruttori e in ogni altro metodo dove occorra evitare ambiguità.

`this` <u>denota l'oggetto corrente</u>: di norma è sottintesa, ma può essere specificata se occorre

LA KEYWORD `this` (2)

La parola chiave `this` può anche servire per *richiamare un costruttore da un altro della stessa classe*

- la notazione `this()` invoca il costruttore di default
- la notazione `this(...)` invoca il costruttore *che corrisponde alla lista di parametri indicata*

Ciò risulta molto comodo per scrivere il codice di inizializzazione una volta sola:

- si scrive solo il costruttore del caso più generale *(con più parametri)*
- si implementano gli altri "rimpallando" l'azione sul primo, fornendo opportuni valori di default per i parametri non specificati dal cliente

`this` CON COSTRUTTORI – IN JAVA

```
public class Point {
 double x, y, z;
 public Point(double x, double y, double zz) {
  this.x = x; this.y = y; z = zz;
 }
 public Point(double x, double y) {
  this(x, y, 0); // richiama costruttore precedente
 }

 public Point(double x){
  this(x, 0); // richiama costruttore precedente
 }
}
```

Costruttore a 3 parametri (il caso più generale)

Costruttore a 2 parametri: richiama quello a 3 parametri!

Costruttore a 1 parametro: richiama quello a 2 parametri!

`this` CON COSTRUTTORI – IN C#

```
public class Point {
 double x, y, z;
 public Point(double x, double y, double zz) {
  this.x = x; this.y = y; z = zz;
 }
 public Point(double x, double y) : this(x, y, 0) {
   // corpo vuoto
 }
 public Point(double x) : this(x, 0) {
  // corpo vuoto
 }
}
```

Costruttore a 3 parametri (il caso più generale)

Costruttore a 2 parametri: richiama quello a 3 parametri

Costruttore a 1 parametro: richiama quello a 2 parametri

ESEMPIO

```
public class Esempio5 {
 public static void main(String[] args){
  Point p1 = new Point(3,2,1);
  Point p2 = new Point(4,5); // (4,5,0)
  Point p3 = new Point(7);   // (7,0,0)
  ...
 }
}
```

Il parametro z viene posto a 0 dal costruttore

I parametri y,z sono posti a 0 dal costruttore

Inizializzato da Point/1 che richiama Point/2 (dando come 2° parametro 0), che a sua volta richiama Point/3 (dando come 3° parametro 0)

Overloading di funzioni

OVERLOADING DI FUNZIONI

- **Il caso dei costruttori non è l'unico: Java e C# consento-no di *definire più funzioni con lo stesso nome* nella stessa classe**
  - **le funzioni "omonime" devono comunque *essere distinguibili tramite la lista dei parametri*, cioè tramite la loro *SIGNATURE***
  - ***ATTENZIONE: il tipo di ritorno da solo non basta a distinguere due funzioni che abbiano lista di parametri identica !***
- **La possibilità di definire funzioni omonime si chiama *overloading:* è utile per evitare una inutile proliferazione di nomi per operazioni "sostanzialmente identiche"**
  - **ESEMPIO: due metodi per incrementare il valore del contatore, rispettivamente di 1 o di K.**
  - ***Perché usare nomi diversi, visto che è concettualmente la stessa azione? Sono entrambi due forme di incremento!***

OVERLOADING: ESEMPIO

Un `Counter` con due metodi di incremento

```
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;}
}
```

Metodo `inc` senza parametri

Metodo `inc` con un parametro (intero)

PROGETTAZIONE INCREMENTALE: DELEGAZIONE ed EREDITARIETÀ

PROGETTAZIONE INCREMENTALE (1/2)

Spesso capita di aver bisogno di un componente *simile* a uno già esistente, ma *non identico*

- avevamo Orologio, ma lo volevamo con display
- abbiamo il contatore che conta in avanti, ma ne vorremmo uno che contasse anche indietro

Altre volte, *l'evoluzione dei requisiti* comporta una corrispondente *modifica dei componenti:*

- necessità di nuovi dati e/o nuovi comportamenti
- necessità di modificare il comportamento di metodi già presenti

Come evitare di dover riprogettare tutto da capo?

PROGETTAZIONE INCREMENTALE (2/2)

Finora, abbiamo solo due possibilità:

- *ricopiare manualmente il codice* della classe esistente e *cambiare quel che va cambiato*
- *creare un oggetto "composto"*
  - che incapsuli il componente esistente...
  - … gli "inoltri" le operazioni già previste…
  - … e crei, *sopra di esso*, le nuove operazioni richieste (eventualmente definendo nuovi dati)
  - *sempre che ciò sia possibile!*

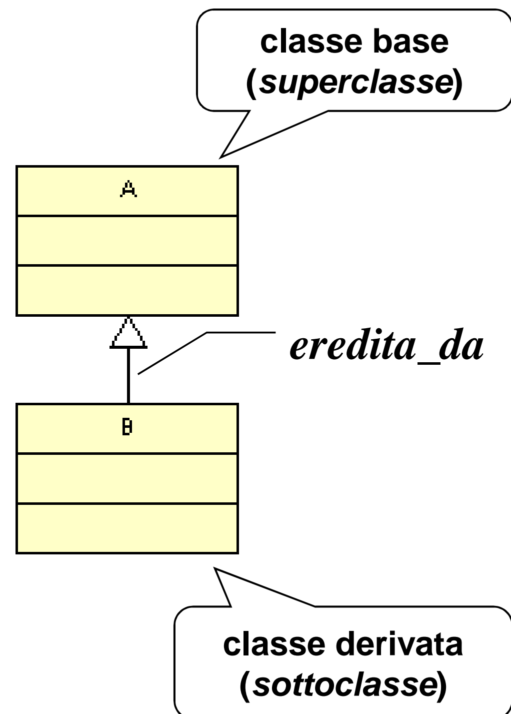
L'OBIETTIVO

- **Poter definire una nuova classe *a partire da una già esistente***
- **Bisognerà dire:**
  - ***quali dati*** **la nuova classe *ha in più* rispetto alla precedente**
  - ***quali metodi*** **la nuova classe *ha in più* rispetto alla precedente**
  - ***quali metodi*** **la nuova classe *modifica* rispetto alla precedente.**

EREDITARIETÀ

EREDITARIETÀ

Una *relazione tra classi:*
si dice che
la nuova classe B
eredita da
la pre-esistente
classe A



EREDITARIETÀ

- La nuova classe **ESTENDE** una classe già esistente
  - può aggiungere *nuovi dati o metodi*
  - può accedere ai dati ereditati *purché il livello di protezione lo consenta*
  - NON può eliminare dati o metodi.
- La classe derivata condivide ***la struttura e il comportamento*** (*per le parti non ridefinite*) della classe base

ESEMPIO

Dal contatore (solo in avanti) …

```
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;}
}
```

Attenzione alla protezione!

ESEMPIO

… al contatore avanti/indietro *(con decremento)*

JAVA

```
public class Counter2 extends Counter {
    public void dec() { val--; }
}
```

C#

```
public class Counter2 : Counter {
    public void dec() { val--; }
}
```

Questa nuova classe:

- **eredita da `Counter` il campo `val` (un `int`)**
- **eredita da `Counter` *tutti i metodi***
- **aggiunge a `Counter` il metodo `dec()`**

ESEMPIO

… al contatore avanti/indietro *(con decremento)*

```
public class Counter2 extends Counter {
        public void dec() { val--; }
}
```

JAVA

```
ounter {
l--; }
```

C#

Ma `val` era privato!!
ERRORE: nessuno può accedere a dati e metodi privati di qualcun altro!

al (un `int`)

- **eredita da `Counter` *tutti i metodi***
- **aggiunge a `Counter` il metodo `dec()`**

EREDITARIETÀ E PROTEZIONE

- **PROBLEMA:**
  il livello di protezione `private` impedisce a chiunque di accedere al dato, *anche a una classe derivata*
  - **va bene per dati "veramente privati"**
  - **ma è *troppo restrittivo* nella maggioranza dei casi**
- **Per sfruttare appieno l'ereditarietà occorre *rilassare un po' il livello di protezione***
  - **senza dover tornare per questo a `public`**
  - **senza dover scegliere per forza la visibilità di package: il concetto di package *non c'entra niente* con l'ereditarietà!**

LA QUALIFICA `protected`

Un dato o un metodo `protected`

- è come la visibilità di `package` (in C: `internal`) per chiunque non sia una classe derivata
- ma *consente libero accesso* a una classe derivata, indipendentemente dal package (namespace) in cui essa è definita.

Occorre dunque *cambiare la protezione del campo `val` nella classe `Counter`.*

ESEMPIO

Il contatore "riadattato"...

```
public class Counter {
  protected 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;}
}
```

Nuovo tipo di protezione.

ESEMPIO

… e il contatore con decremento:

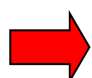
```
public class Counter2 extends Counter {
   public void dec() { val--; }
}
```

Ora funziona !

UNA RIFLESSIONE

La qualifica `protected`:

- **rende accessibile un campo *a tutte le sottoclassi*, presenti *e future***
- **costituisce perciò un *permesso di accesso "indiscriminato"*, valido per *ogni possibile sottoclasse che possa in futuro essere definita*, senza possibilità di distinzione.**



I membri `protected` *sono citati* nella documentazione prodotta da Javadoc (a differenza dei membri qualificati *privati* o con visibilità *package*).

EREDITARIETÀ

Cosa si eredita?

- tutti i dati della classe base
  - anche quelli privati, a cui comunque la classe derivata non potrà accedere direttamente
- tutti i metodi...
  - anche quelli che la classe derivata non potrà usare direttamente
- *... tranne i costruttori*, perché sono specifici di quella particolare classe.

EREDITARIETÀ E COSTRUTTORI

- Una classe derivata *non può prescindere dalla classe base*, perché ogni istanza della classe derivata *comprende in sé*, indiretta-mente, un oggetto della classe base.
- Quindi, *ogni costruttore della classe derivata si appoggia a un costruttore della classe base* affinché esso costruisca la "parte di oggetto" relativa alla classe base stessa:

"ognuno deve costruire ciò che gli compete"

EREDITARIETÀ E COSTRUTTORI

Perché ogni costruttore della classe derivata si deve appoggiare a un costruttore della classe base?

- solo il costruttore della classe base può sapere come inizializzare i dati ereditati in modo corretto
- solo il costruttore della classe base può garantire l'inizializzazione dei dati privati, a cui la classe derivata non potrebbe accedere direttamente
- è inutile duplicare nella sottoclasse tutto il codice necessario per inizializzare i dati ereditati, che è già stato scritto.

Ma cosa vuol dire "appoggiarsi" a un costruttore della classe base?

- il costruttore della classe derivata *INVOCA AUTOMATICA-MENTE un opportuno costruttore della classe-base*

EREDITARIETÀ E COSTRUTTORI

- ***Ma come può un costruttore della classe derivata invocare un costruttore della classe base?***
  I costruttori non si possono chiamare direttamente!
- **Occorre un modo per dire al costruttore della classe derivata di *rivolgersi al "piano di sopra"***
  - In JAVA, si usa a questo scopo la keyword `super`
  - In C#, si usa per analogo scopo la keyword `base`
- `super` e `base` si usano in modo analogo alla keyword `this`
- `this(…)` richiama un altro costruttore della stessa classe
- `super(…)` / `base(…)` richiamano quello della classe base

ESEMPIO IN JAVA

Il contatore con decremento:

Costruttore di default *generato automatica-mente* in assenza di altri costruttori

```
public class Coun      extends Counter {
  public void dec() { val--; }
  public Counter2() { super(); }
  public Counter2(int v) { super(v); }
}
```

L'espressione `super(...)` invoca il costruttore della classe base <u>che corrisponde come numero e tipo di parametri</u> alla lista di argomenti fornita.

ESEMPIO IN C#

Il contatore con decremento:

Costruttore di default *generato automatica-mente* in assenza di altri costruttori

```
public class Coun      : Counter {
  public void dec() { val--; }
  public Counter2() : base() {}
  public Counter2(int v) : base(v) {}
}
```

L'espressione `base(...)` invoca il costruttore della classe base <u>che corrisponde come numero e tipo di parametri</u> alla lista di argomenti fornita.

EREDITARIETÀ E COSTRUTTORI

E se non indichiamo alcuna chiamata a ***`super(...)/base(...)`*** *?*

- **Il compilatore inserisce automaticamente una chiamata al *costruttore di default* della classe base aggiungendo `super()` o `base()`**
- **In questo caso il costruttore dei default della classe base deve esistere, altrimenti si ha ERRORE.**

EREDITARIETÀ E COSTRUTTORI

RICORDARE: il sistema genera automaticamente il costruttore di default *solo se noi non definiamo alcun costruttore*.
Se c'è *anche solo una definizione di costruttore data da noi*, il sistema assume che noi sappiamo il fatto nostro e non genera più il costruttore di default automatico.

te
ella
se()
della
classe base deve esistere, ...nenti si ha
ERRORE.

`super` e `base`: RIASSUNTO

La parola chiave *`super`* o *`base`*

- nella forma `super(...)` o `base(…)`,
  invoca un costruttore della classe base
- nella forma `super.val` o `base.val`,
  consente di accedere al campo `val` della classe base
  (sempre che esso non sia privato)
- nella forma `super.metodo()` o `base.metodo()`
  invoca il metodo `metodo()` della classe base
  (sempre che esso non sia privato)

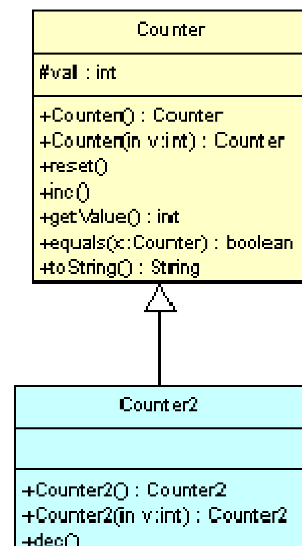
COSTRUTTORI e PROTEZIONE

- **Di norma, i costruttori sono `public`**
  - ciò è necessario *affinché chiunque possa istanziare oggetti* di quella classe
  - in particolare, è sempre pubblico il costruttore di default generato automaticamente
- **Possono però esistere classi con tutti i costrut-tori *non pubblici* (tipicamente, *protected*)**
  - lo si fa per *impedire di* creare oggetti di quella classe al di fuori di "casi controllati"
  - caso tipico: una classe pensata per fungere SOLO da *classe base per altre* da definirsi in futuro (definirà probabilmente solo costruttori *protected*)

Polimorfismo

EREDITARIETÀ: RIFLESSIONI

- Se una classe eredita da un'altra, *la classe derivata mantiene l'interfaccia di accesso della classe base*
  - Naturalmente può *estenderla*, aggiungendo nuovi metodi
    - Quindi, *ogni* `Counter2` *è anche un (tipo particolare di)* `Counter` *!*
    - Ergo, un `Counter2` può essere usato *al posto di un* `Counter` ***in modo trasparente al cliente***

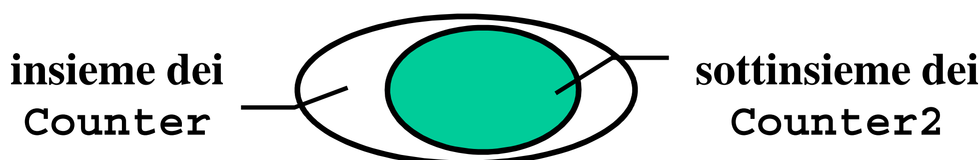


EREDITARIETÀ: RIFLESSIONI

Dire che

- *ogni `Counter2` è anche un tipo particolare di `Counter`*
- **un oggetto `Counter2` può essere usato** *ovunque sia atteso un oggetto `Counter`*

significa dire che l'insieme dei `Counter2` è ***un sottoinsieme*** **dell'insieme dei `Counter`**



EREDITARIETÀ: CONSEGUENZE

Poter usare un `Counter2` al posto di un `Counter` significa che:

- se `c` è un riferimento a `Counter`, si deve poterlo usare per referenziare un'istanza di `Counter2`
- se una funzione si aspetta come parametro un `Counter`, si deve poterle passare un'istanza di `Counter2`
- se una funzione dichiara di restituire un `Counter`, può in realtà restituire un'istanza di `Counter2`

Ovviamente, non è vero il viceversa:

- un `Counter` NON È un tipo particolare di `Counter2`

EREDITARIETÀ: CONSEGUENZE

Poter usare un `Counter2` al posto di un `Counter` significa che:

```
Counter c = new Counter2(11);
Counter2 c2 = new Counter2(); c = c2;
```

```
void f(Counter x){ ... }
...
f(c2); // c2 è un'istanza di Counter2
```

```
Counter getNewCounter(int v){}
    return new Counter2(v);
}
```

```
Counter c = new Counter2(11);
Counter2 c2 = c; // NO, ERRATO!!!
```

ESEMPIO

```
public class Esempio6 {
 public static void main(String args[]) {
  Counter  c1 = new Counter(10);
  Counter2 c2 = new Counter2(20);
  c1=c2;        // OK: c2 è anche un Counter
  // c2=c1;     // NO: c1 è solo un Counter
 }
}
```

OK perché `c2` **è un** `Counter2`, ***quindi anche implicitamente un*** `Counter` (e `c1` è un `Counter`)

NO, perché `c2` **è un** `Counter2` ***e come tale esige un suo "pari",*** **mentre** `c1` **è "solo" un** `Counter`

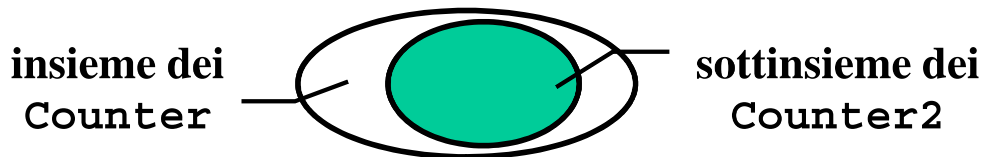
CONSEGUENZE CONCETTUALI

Dire che

l'insieme dei `Counter2` è un sottoinsieme dell'insieme dei `Counter`

induce *una classificazione del mondo*

(aderente alla realtà...?)



Per questo l'ereditarietà è più di un semplice *"riuso di codice": riusa l'astrazione*

CLASSIFICAZIONE DEL "MONDO"

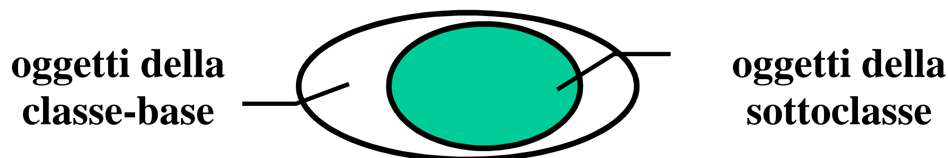
Questa classificazione può essere o meno *aderente alla realtà:*

- **se è aderente alla realtà,**
  - **→ rappresenta bene la situazione**
  - **→ è un buon modello del mondo**
- **se invece *nega la realtà,***
  - **→ non è un buon modello del mondo**
  - **→ può produrre assurdità e inconsistenze**

Come si riconosce una *buona* classificazione?

EREDITARIETÀ: CONSEGUENZE

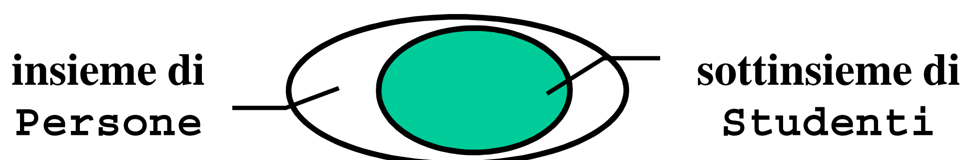
Una sottoclasse deve *delimitare un sottoinsieme* della classe base, altrimenti rischia di modellare la realtà *al contrario.*



Esempi

- **è vero che ogni Studente è *anche* una Persona**
  → Studente può derivare da Persona
- **non è vero che ogni Reale *sia anche* un Intero**
  → Reale non dovrebbe derivare da Intero

ESEMPIO: Persone e Studenti

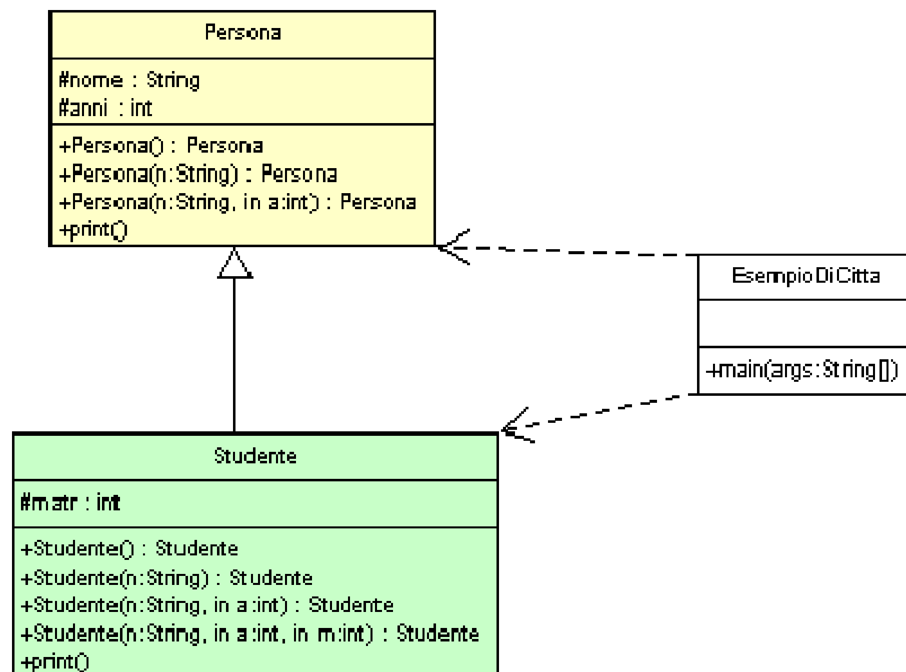


Una classe `Persona`

e una sottoclasse `Studente`

- ***è aderente alla realtà*, perché è vero nel mondo reale che *tutti gli studenti sono persone***
- **compatibilità di tipo: potremo usare uno studente (che è *anche* una persona) ovunque sia richiesta una generica persona**
  ma non viceversa: se serve uno studente, non ci si può accontentare di una generica persona.

Persone e Studenti: MODELLO



LA CLASSE `Persona`

```
public class Persona {
 protected String nome;
 protected int anni;
 public Persona() {
     nome = "sconosciuto"; anni = 0; }
 public Persona(String n) {
     nome = n; anni = 0; }
 public Persona(String n, int a) {
     nome=n; anni=a; }
 public void print() {
  System.out.print(”Mi chiamo " + nome);
  System.out.println(" e ho " +anni+ "anni");
 }
}
```

Hanno senso tutti questi costruttori?

C#: minime modifiche nei metodi di stampa

LA CLASSE Studente

```
public class Studente extends Persona {
 protected int matr;
 public Studente() {
     super(); matr = 9999; }
 public Studente(String n) {
     super(n); matr = 8888; }
 public Studente(String n, int a) {
     super(n,a); matr=7777; }
 public Studente(String n, int a, int m) {
     super(n,a); matr=m; }
 public void print() {
  super.print();
  System.out.println("Matricola = " + matr);
 }
}
```

C#: : al posto di extends

Ridefinisce il metodo print di Persona

C#: base(…) al posto di super(…)

LA CLASSE Studente

Ridefinisce il metodo void print()

- sovrascrive quello ereditato da Persona
- è una versione specializzata per Studente che però *riusa quello di Persona* (super), estendendolo per stampare la matricola.


```
public class Studente extends Persona {
...
 public Studente(String n, int a, int m) {
     super(n,a); matr=m; }
 public void print() {
  super.print();
  System.out.println("Matricola = " + matr);
 }
}
```

UN MAIN DI PROVA

```
public class EsempioDiCitta {
 public static void main(String args[]){
  Persona  p = new Persona("John",21);
  Studente s = new Studente("Tom",33);
  p.print();  // stampa nome ed età
  s.print();  // stampa nome, età, matricola
 }
}
```

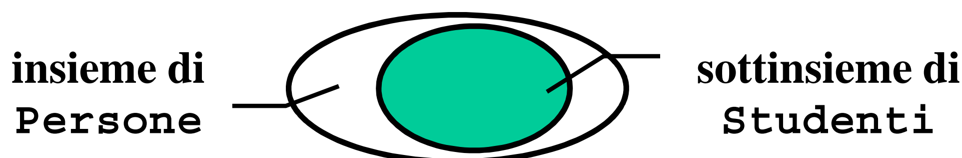
OUTPUT:

```
Mi chiamo John e ho 21 anni
Mi chiamo Tom e ho 33 anni
```

print di Persona

print di Studente

UNA NUOVA QUESTIONE



Poiché `Studente` eredita da `Persona`, possiamo usare uno Studente ovunque sia richiesta una Persona.

Ciò equivale infatti a dire che il tipo `Studente` è <u>compatibile</u> col tipo `Persona`, esattamente come `float` è compatibile con `double`.

Ergo, se `s` è uno `Studente` e `p` una `Persona`, la frase:

```
p = s
```

è LECITA e *non comporta perdita di informazione.*

MA ALLORA...

Se è lecito scrivere:

```
Persona  p = new Persona("John",21);
Studente s = new Studente("Tom",33);
p = s;
```

COSA SUCCEDE ADESSO invocando metodi su p ??

```
p.print();  // COSA STAMPA ???
```

Infatti, `p` è un riferimento a `Persona`,
ma gli è stato assegnato un oggetto `Studente`!

COSA è "GIUSTO" che accada?
QUALE `print` deve scattare?

Considerazioni finali

- La programmazione ad oggetti nasce come risposta ai limiti di alcuni linguaggi per programmare sistemi “complessi”
- Permette di “raccogliere” in un unico elemento software (la classe) i dati ed i metodi per accedervi, fornendo un potente meccanismo di *astrazione*
  - Incapsulamento
  - Ereditarietà
  - Overloading dei metodi
  - Polimorfismo