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!***

CARATTERISTICHE di JAVA

- **Java è generalmente considerato un linguaggio orientato agli oggetti "puro"**
- **Tuttavia il suo modello si discosta per alcuni aspetti dalla visione "classica"**
- **Si tratta perlopiù di estensioni: nel corso del tempo il modello OOP si è arricchito di concetti nuovi e spesso molto importanti**
- **C'è anche qualche compromesso legato a motivi di efficienza**
- **La sintassi di Java è derivata da quella del C, la più diffusa, con alcune differenze significative che evidenzieremo nell'esposizione del linguaggio**

TIPI DI DATO PRIMITIVI IN JAVA (1)

- **Nel modello classico un linguaggio ad oggetti comprende solo oggetti e classi**
- **In Java, principalmente per motivi di efficienza, esistono anche i tipi primitivi (o predefiniti) che hanno un ruolo simile a quello che hanno in C. Sono cioè strutture dati slegate da qualunque comportamento**
- **La definizione dei tipi primitivi è però più precisa e non ci sono ambiguità:**
  - **Non esistono tipi la cui dimensione dipende dalla piattaforma (come int in C)**
  - **Interi, caratteri e booleani sono tipi ben distinti fra loro**

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) $-9\ 10^{18}$ ... $+9\ 10^{18}$**

  NB: le costanti long terminano con la lettera L

TIPI DI DATO PRIMITIVI IN JAVA

- **reali *(IEEE-754)***
  - **`float` (4 byte)** **- 10^{45} ... + 10^{38}**
    (6-7 cifre significative)
  - **`double` (8 byte)** **- 10^{328} ... + 10^{308}**
    (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`!**

VARIABILI e COSTANTI

- **In Java le variabili in Java vengono dichiarate come in C e possono essere inizializzate:**

  ```
  int n,m;
  float f = 5;
  boolean b = false;
  ```

- **Anche il loro uso è lo stesso:**

  ```
  n = 4;
  m = n * 3 +5:
  b = m > 6;
  ```

- **Esiste anche la possibilità di dichiarare delle costanti anteponendo alla dichiarazione la parola chiave final**

  ```
  final int n = 8;
  final boolean b = false;
  ```

CLASSI in JAVA

- **Nel modello "classico" le classi hanno due funzioni:**
  - **Definire una struttura e un comportamento**
  - **Creare gli oggetti (istanze)**
- **Sono quindi delle matrici, degli "stampini", che consentono di creare istanze fatte nello stesso modo ma con identità distinte.**
- **Svolgono nel contempo il ruolo di tipo e di strumenti di costruzione**
- **In Java le classi hanno anche un'altra capacità: possono fornire servizi indipendenti dalla creazione di istanze: E' infatti possibile definire dei metodi (parola chiave static) che possono essere invocati anche se non esiste alcuna istanza**

CLASSI in JAVA: VISIBILITÀ

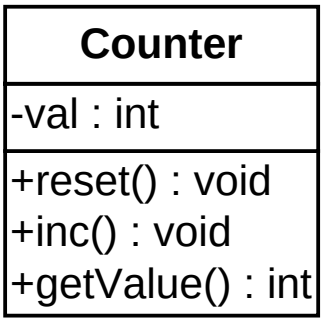
- **Nel modello "classico", in virtù dell' incapsulamento abbiamo i seguenti comportamenti:**
  - **Tutti gli attributi (variabili) sono invisibili all'esterno**
  - **Tutti i metodi sono visibili all'esterno**
- **Java introduce un meccanismo molto più flessibile: è possibile stabilire il livello di visibilità di ogni metodo e di ogni variabile usando le parole chiave private e public.**
- **Questa estensione consente di avere due comportamenti nuovi:**
  - **Metodi non visibili all'esterno (privati): molto utili per nascondere dettagli implementativi e migliorare l'incapsulamento**
  - **Variabili visibili all'esterno (pubbliche): pericolose e da evitare perché "rompono" l'incapsulamento**

ESEMPIO di CLASSE: Counter

- **Proviamo a costruire una semplice classe che rappresenta un contatore monodirezionale**
- **Come accade sempre nella programmazione ad oggetti si parte definendo il comportamento**
- **La nostra classe si chiamerà Counter (per convenzione i nomi di classe sono sempre in maiuscolo) e sarà in grado di:**
  - **Azzerare il valore del contatore: reset()**
  - **Incrementare il valore: inc()**
  - **Restituire il valore corrente: getValue()**
- **A livello implementativo useremo una variabile intera (val) per memorizzare il valore corrente del contatore**
- **In termini OOP: la classe Counter avrà uno stato costituito dalla variabile val e un comportamento definito dai tre metodi: reset(), inc() e getValue()**

ESEMPIO di CLASSE: Counter - Modello

- **Vediamo la rappresentazione UML (modello) della nostra classe**
- **I caratteri + e – davanti ai metodi e agli attributi indicano la visibilità:**
  - **- sta per private**
  - **+ sta per public**



Counter - Implementazione

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

- **Anche le classi hanno una visibilità**
- **Il campo val è definito come privato, mentre i metodi sono pubblici**
- **La sintassi per la definizione dei metodi e per la dichiarazione della variabile val è uguale a quella del C**
- **La variabile val può essere liberamente utilizzata dai metodi della classe**

CLASSI di SISTEMA

- **In C esiste una collezione di funzioni standard messe a disposizione dal sistema che prende il nome di libreria di sistema**
- **In Java, come nella maggior parte dei linguaggi ad oggetti, esiste invece una collezione di classi di sistema**
- **Abitualmente ci si riferisce all'insieme delle classi di sistema con il nome di framework**
- **Il framework di Java è molto ricco e comprende centinaia di classi che possono essere utilizzate per scrivere le proprie applicazioni**
- **Praticamente ogni applicazione Java fa uso di una o più classi di sistema**

Ancora sulle …. CLASSI

Una *classe Java* è una entità dotata di una *"doppia natura":*

- **contiene la definizione di un *tipo di dato astratto,* cioè uno "stampo" per *creare nuovi oggetti,*** anch'essi dotati di idonei **meccanismi di protezione**
- **Ma puo' fungere anche da *componente software,*** che in quanto tale può possedere propri *dati* e *operazioni,* opportunamente **protetti => metodi static**

UNA CLASSE PER I NUMERI PRIMI

- **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**

UNA CLASSE PER I NUMERI PRIMI

```
public class NumeriPrimi {
  private static int lastPrime = 0;
  private static boolean isPrime(int p) {
      … lo stesso codice usat
  public  static int n
      … lo stesso codice us
}
```

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

- **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

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*

VARIABILI e METODI di ISTANZA

- **Variabile di istanza:** esiste solo all'interno di un'istanza
- **Metodi di istanza:** metodi che trattano variabili di istanza

VARIABILI e METODI di CLASSE

- **variabili di classe** : variabili che esistono indipendentemente dall'istanza. Sono come variabili globali condivise da tutti gli oggetti. Per accedere alle variabili di classe si usa il nome stesso della classe, anziché il riferimento
- **metodi di classe:** tali metodi operano su variabili di classe (si comportano analogamente a funzioni del C)

PROGRAMMI in JAVA

- **Un programma Java è costituito da un insieme di classi**
- **Per convenzione deve esistere almeno una classe, definita come pubblica, che implementi un metodo static (cioè un metodo che può essere invocato anche in assenza di istanze) chiamato main()**
- **Questo metodo ha lo stesso ruolo della funzione main nei programmi C**
- **Il fatto che il metodo sia static consente al sistema di invocarlo alla partenza del programma, quando non esiste ancora nessuna istanza.**
- **Il più semplice programma Java è costituito quindi da una sola classe con le caratteristiche sopra descritte**

Esempio 1: Hello world!

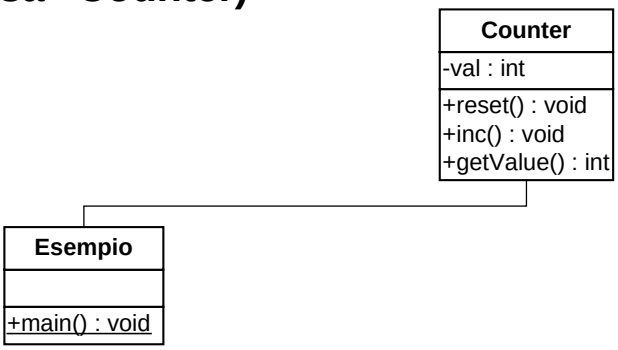
- **Scriviamo il programma Hello World in Java**

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

- **Per il momento ignoriamo il parametro di main() (vedremo più avanti il suo significato)**
- **System.out.println() è un metodo messo a disposizione da una delle classi di sistema (System.out) e consente di scrivere sul video.**

ESEMPIO 2: DESCRIZIONE e MODELLO

- **Proviamo a scrivere un'applicazione più complessa che fa uso di due classi: la classe che definisce il metodo main() e la classe Counter definita in precedenza.**
- **Nel metodo main() creeremo un'istanza di Counter e invocheremo alcuni metodi su di essa**
- **Il diagramma sottostante rappresenta la struttura dell'applicazione (la linea di collegamento ci dice che Esempio "usa" Counter)**



ESEMPIO 2: IMPLEMENTAZIONE

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

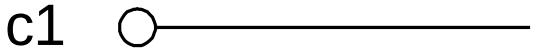
public class Esempio
{
    public static void main(String[] args)
    {
      int n;
      Counter c1;
      c1 = new Counter();
      c1.inc();
      n = c1.getValue();
      System.out.println(n);
    }
 }
```

PASSO 1: DICHIARAZIONE del RIFERIMENTO

- Java, come tutti i linguaggi ad oggetti, consente di dichiarare variabili che hanno come tipo una classe

  Counter c1;
- Queste variabili sono riferimenti ad oggetti (in qualche modo sono dei puntatori).

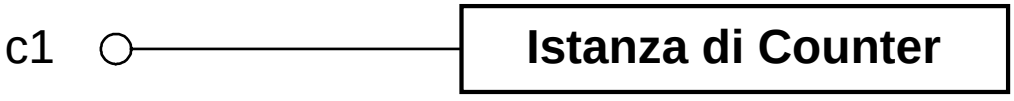
  Attenzione!
- La dichiarazione della variabile non implica la creazione dell'oggetto:
- la variabile c1 è a questo punto è solo un riferimento vuoto (un puntatore che non punta da nessuna parte)



PASSO 2: CREAZIONE dell'OGGETTO

- Per creare l'oggetto devo utilizzare un'istruzione apposita che fa uso della parola chiave new

  c1 = new Counter();
- A questo punto, e non prima, ho a disposizione un oggetto (un'istanza di Counter) e c1 è un riferimento a questa istanza



PASSO 3: USO dell'OGGETTO

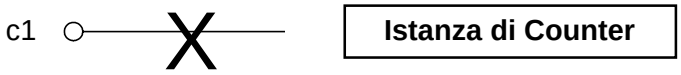
- A questo punto abbiamo un oggetto ed un riferimento a questo oggetto (la variabile c1)
- Possiamo utilizzare il riferimento per invocare i metodi pubblici dell'oggetto utilizzando la cosiddetta "notazione puntata":

  <nome variabile>.<nome metodo>
- Per esempio:

  c1.inc();
  n = c1.getValue();
- Dal momento che c1 è di "tipo" Counter, il compilatore, basandosi sulla dichiarazione della classe Counter può determinare quali sono i metodi invocabili (quelli dichiarati come pubblici)

PASSO 4: DISTRUZIONE dell'OGGETTO

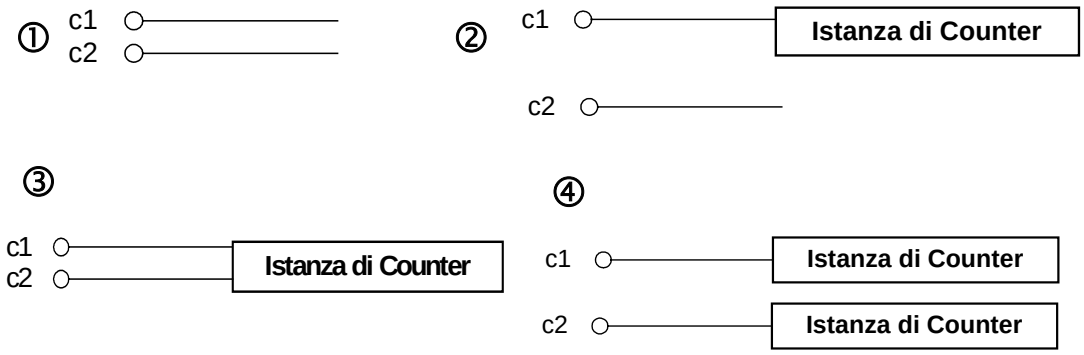
- Non occorre distruggere manualmente l'oggetto, in Java esiste un componente del sistema, chiamato garbage collector che distrugge automaticamente gli oggetti quando non servono più
- Come fa il garbage collector a capire quando un oggetto non serve più? Un oggetto non serve più quando non esistono più riferimenti ad esso
- c1 è una variabile locale del metodo main(): quando il meodo main() termina c1 non esiste più
- Quindi non esistono più riferimenti all'oggetto che abbiamo creato e il garbage collector può distruggerlo



RIFERIMENTI e OGGETTI

- Consideramo questa serie di istruzioni:
- ```
  1. Counter c1,c2;
  2. c1 = new Counter();
  3. c2 = c1;
  4. c2 = new Counter();
  ```

- Ecco l'effetto su variabili e gestione della memoria:



RIFERIMENTI e OGGETTI - ASSEGNAMENTO

💣 Attenzione!

- L'assegnamento c2=c1 non crea una copia dell'oggetto!
- Si hanno invece due riferimenti allo stesso oggetto!
- Questo perché le variabili che hanno come tipo una classe sono riferimenti agli oggetti: non contengono un valore ma l'indirizzo dell'oggetto
- Quindi un assegnamento copia l'indirizzo
- L'effetto è che abbiamo due variabili che contengono lo stesso indirizzo e quindi "puntano" allo stesso oggetto
  ```

RIFERIMENTI e OGGETTI - COPIA

- Ma allora come si fa a copiare un oggetto?
- Innanzitutto aggiungiamo alla classe un metodo che copia esplicitamente lo stato (nel nostro caso val)

```
public class Counter
{
  private int val;
  public void reset() { val = 0; }
  public void inc(){ val++; }
  public int getValue() { return val;}
  public void copy(Counter c2) { val = c2.val;}
}
```

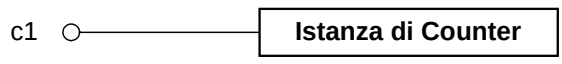
- Poi si procede così:

```
Counter c1, c2; /* Dichiariamo due variabili */
c1 = new Counter(); /* creiamo due istanze */
c2 = new Counter();
c2.copy(c1); /* copiamo lo stato di c1 in c2 */
```


RIFERIMENTI e OGGETTI – VALORE null

- A differenza dei puntatori del C non è possibile fare "pasticci" con i riferimenti agli oggetti
- Con un riferimento possiamo fare solo 4 cose:
  1. Dichiararlo: Counter c1, c2;
  2. Assegnargli un oggetto appena creato: c1 = **new** Counter();
  3. Assegnargli un altro riferimento: c2 = c1;
  4. Assegnargli il valore null: c1 = **null**;
- Assegnando il valore null (è una parola chiave di Java) il puntatore non punta più da nessuna parte

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



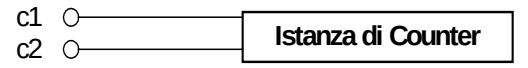

```
c1 = null;
```

RIFERIMENTI e OGGETTI – UGUAGLIANZA fra RIFERIMENTI

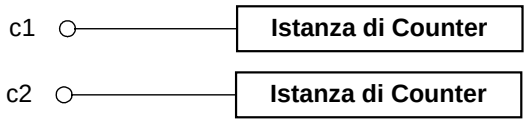
- Che significato può avere un'espressione come:

  c1 == c2

- c1 e c2 sono due riferimenti e quindi sono uguali se l'indirizzo che contengono è uguale
- Due riferimenti sono uguali se puntano allo stesso oggetto e non se gli oggetti a cui puntano sono uguali
- Quindi:



`c1 == c2 vale` **`true`**



`c1 == c2 vale` **`false`**

UGUAGLIANZA fra OGGETTI

- Come si fa se invece si vuole verificare se due oggetti sono uguali, cioè hanno lo stesso stato?
- Si fa come per la copia: si aggiunge a Counter il metodo equals()
- public class **Counter**
  {
  private int **val;**
  public void **reset() { val = 0; }**
  public void **inc(){ val++; }**
  public int **getValue() { return val; }**
  public void **copy(Counter c2) { val = c2.val };**
  public boolean **equals(Counter c2){**return **val == c2.val;}**
  }
- Poi si procede così:

```
Counter c1, c2; boolean b1,b2;
c1 = new Counter(); c1.inc();
c2 = new Counter();
b1 = c1.equals(c2); /* b1 vale false */
c1.copy(c2);
b2 = c1.equals(c2); /* b2 vale true */
```

COSTRUTTORI

- Riprendiamo in esame la creazione di un'istanza

  c1 = **new** Counter();
- Cosa ci fanno le parentesi dopo il nome della classe? Sembra quasi che stiamo invocando una funzione!
- In effetti è proprio così: ogni classe in Java ha almeno un metodo costruttore, che ha lo stesso nome della classe
- Il compito del costruttore è quello inizializzare la nuova istanza: assegnare un valore iniziale alle variabili, creare altri oggetti ecc.
- Quando usiamo l'operatore new, il sistema crea una nuova istanza e invoca su di essa il costruttore
- Un costruttore è un metodo che viene chiamato automaticamente dal sistema ogni volta che si crea un nuovo oggetto

CARATTERISTICHE dei COSTRUTTORI in JAVA

- Un costruttore ha lo stesso nome della classe
- Non ha tipo di ritorno, nemmeno void
- Ha una visibilità, comunemente public
- Vediamo la definizione del classe Counter con la definizione del costruttore:

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

COSTRUTTORI MULTIPLI - 1

- Java ammette l'esistenza di più costruttori che hanno lo stesso nome (quello della classe) ma si differenziano per il numero e il tipo dei parametri
- I costruttori con i parametri permettono di inizializzare un'istanza con valori passati dall'esterno

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

- In questo caso abbiamo la possibilità di creare un contatore con un valore iniziale prefissato

COSTRUTTORI MULTIPLI - 2

- Vediamo come si può operare in presenza di più di un costruttore:

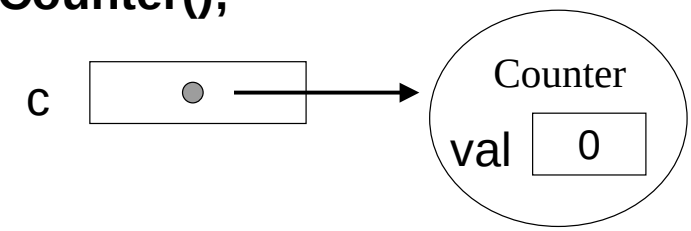
```
Counter c1, c2;
c1 = new Counter();
c2 = new Counter(5);
```

- Nel primo caso viene invocato il costruttore senza parametri (costruttore di default): il valore del contatore viene inizializzato con il valore 0
- Nel secondo caso viene invocato il costruttore che prevede un parametro e il valore iniziale del contatore è quindi 5
- Ogni classe ha un costruttore di default: se non viene definito esplicitamente il sistema ne crea automaticamente uno vuoto

E ora?

Counter c;

c = new Counter();



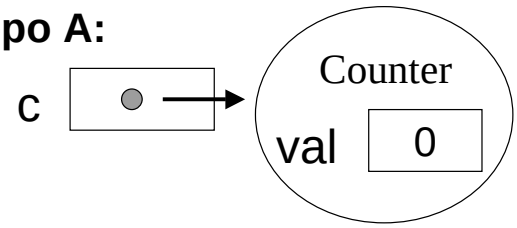
new: chiama il costruttore della classe per generare un oggetto e assegnargli uno stato iniziale, ritorna un indirizzo di memoria al nuovo oggetto che viene memorizzato in una variabile riferimento di tipo compatibile

Val è una variabile di istanza (esiste solo all'interno di un'istanza)

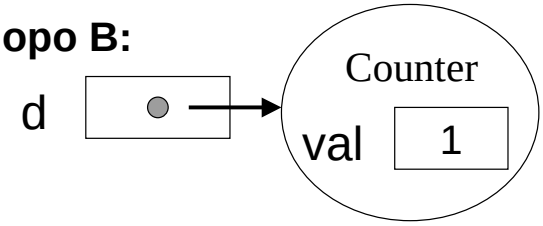
E ora (2.)?

```
Counter c, d;
c = new Counter();  A
d = new Counter;    B
c.inc;              C
```

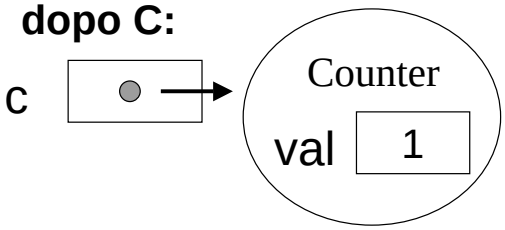
dopo A:

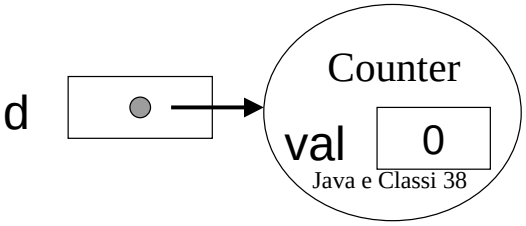


dopo B:



dopo C:





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!***

OVERLOADING DI FUNZIONI

- **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**
- **Es. overloading: costruttori**

OVERLOADING DI FUNZIONI

Riprendiamo la nostra classe Counter e definiamo una seconda versione del metodo inc():

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

Vediamo un esempio di uso:

```
Counter c1;
c1 = new Counter();
c1.inc();  /* Viene invocata la prima versione */
c2.inc(3); /* Viene invocata la seconda versione */
```


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 il metodo riceve la copia**
- ***un riferimento*** **viene *pure copiato*, il metodo riceve la copia, ma la copia punta all'oggetto originale!**

Ovvero:

- **I tipi primitivi vengono passati SEMPRE per valore**
- **Gli oggetti vengono passati SEMPRE per riferimento**

PASSAGGIO di TIPI PRIMITIVI

```
public class Tester {
…….
public void myMethod(int one, float two)
    {
          one =25;
          two= 35.4;          C
    }
}

public class Programma {
    public static void main(String args[]) {
          Tester tester;
          int a, b;
          tester = new Tester();
          a=10;
          b=20;                 A
          tester.myMethod(a,b);       B
          System.out.println("a = " + a);
}
```

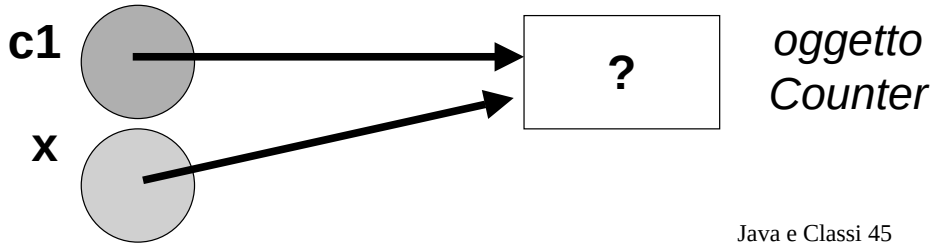
PASSAGGIO DI OGGETTI

Esempio:

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

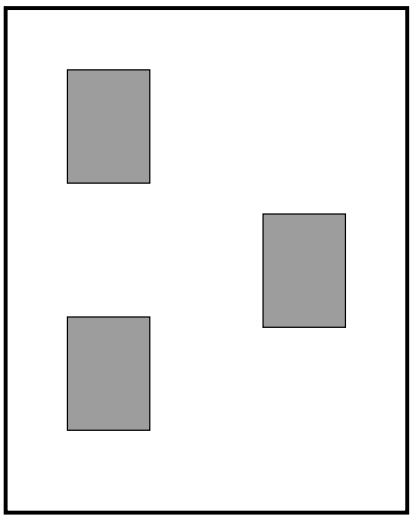
Il cliente:

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



RIPRENDIAMO …… PROGRAMMI IN JAVA

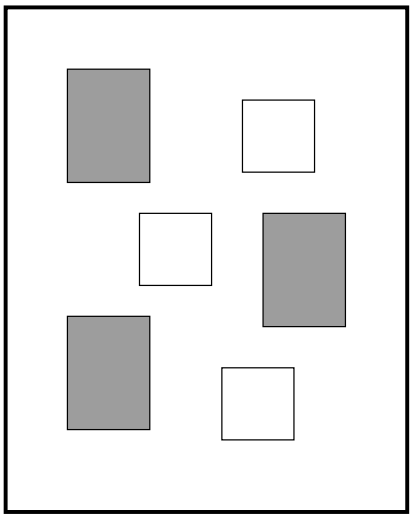
Un programma Java è *un insieme di classi e oggetti*



PROGRAMMI IN JAVA

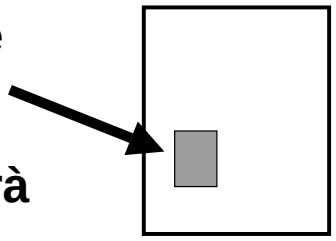
Un programma Java è *un insieme di classi e oggetti*

- **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***



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`***



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)*

⇒ *Tutto è un oggetto anche un'applicazione*

⇒ *Ogni classe puo' avere un metodo main(): solo il metodo main verrà eseguito all'avvio dell'applicazione*

PROGRAMMI IN JAVA

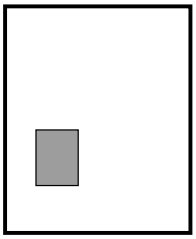
Prima differenza rispetto al C:

- **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!**

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;
  }
}
```

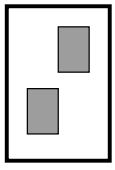

ESEMPIO 0

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



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`

ESEMPIO: UN CLIENTE

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

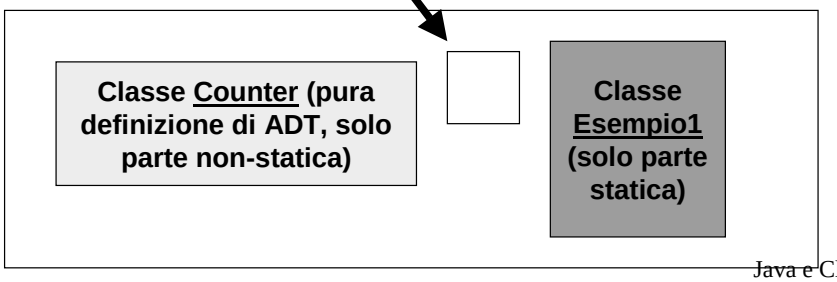
Qui scatta il costruttore/0 → **c1** inizializzato a 1

Qui scatta il costruttore/1 → **c2**

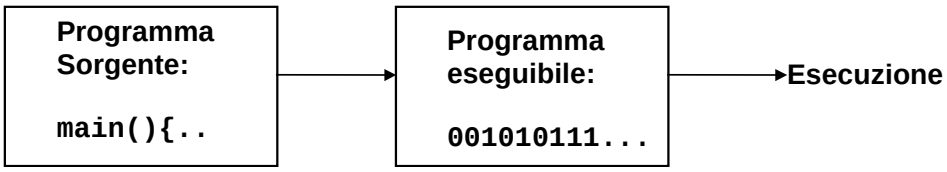
ESEMPIO COMPLETO

A run-time, nasce un oggetto:

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



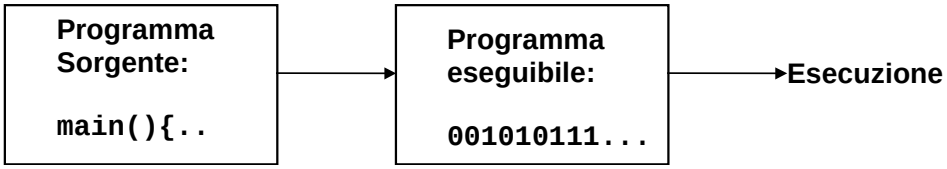
SVILUPPO DI PROGRAMMI



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*

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

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

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*

AMBIENTI DI PROGRAMMAZIONE

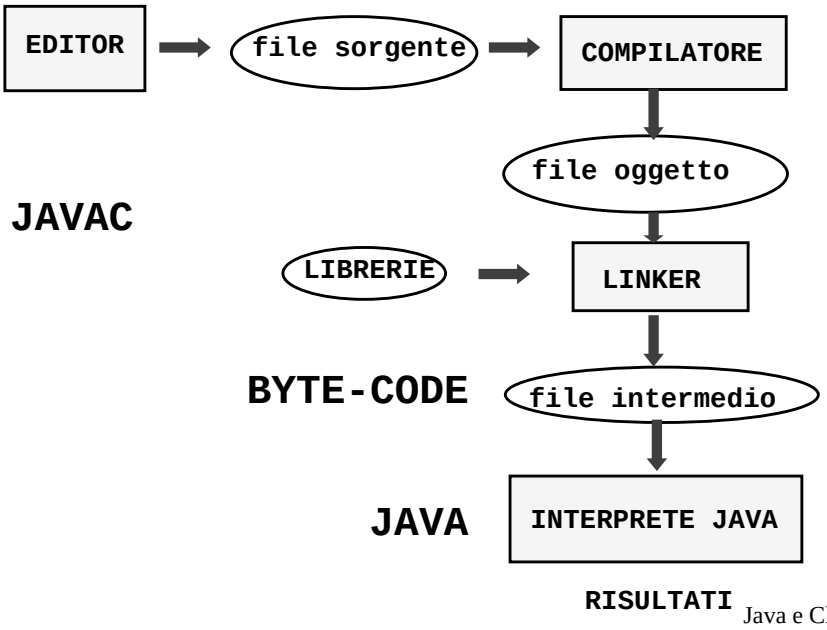
II° CASO: INTERPRETAZIONE

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

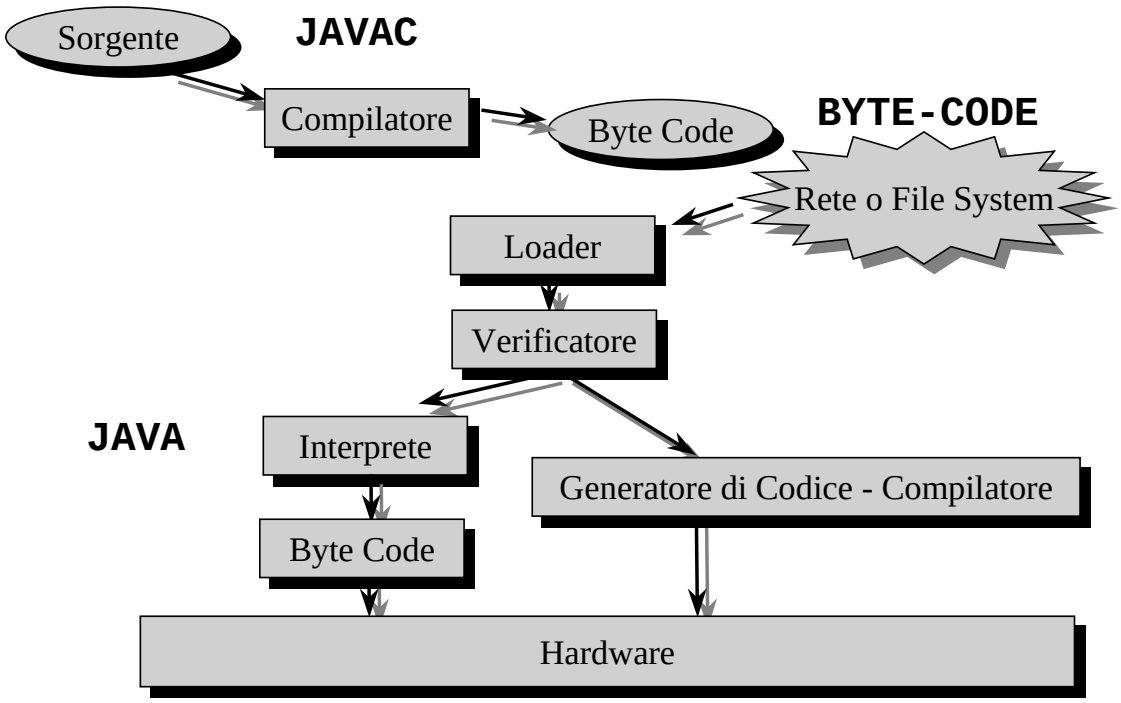
  È generalmente in alternativa al compilatore (raramente presenti entrambi)

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

APPROCCIO MISTO



APPROCCIO JAVA



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***

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***
  - ***[illegible] definita***
- **Un[illegible] definita in un[illegible] *classe* ed estensione `.java`**
- **Esempi**
  classe `EsempioBase` → file `EsempioBase.java`
  classe `Esempio0` → file `Esempio0.java`

Essenziale:
- **poter usare *nomi di file lunghi***
- ***rispettare maiuscole/minuscole***

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..)

... E OLTRE

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

- ***editor con "syntax highlightling"***
  - 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, ...

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*

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)

COLLEGAMENTO STATICO...

Nei linguaggi "classici":

- si compila ogni file sorgente
- ***si collegano***

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)

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

...ma scarsa flessibilità, perché tutto ciò che si usa deve essere *dichiarato a priori*

Poco adatto ad *ambienti a elevata dinamicità* come Internet

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

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 portabile, inter-piattaforma***
- **per eseguirlo occorre un *interprete Java***
  - **è l'unico strato *dipendente dalla piattaforma***
- **in questo modo si ottiene *vera portabilità:* un file `.class` compilato su una piattaforma *può funzionare su qualunque altra!!!***

ESECUZIONE E PORTABILITÀ

In Java,

- **ogni classe è compilata in un file `.class`**
- **il formato d**
  non è dirett
  formato portabile, inter-piattafo

Si perde un po' in efficienza (c'è di mezzo un interprete)...

..ma si guadagna *molto di più:*

- **possibilità di scaricare ed eseguire codice dalla rete**
- **indipendenza dall'hardware**
- ***"write once, run everywhere"***

terpre ava
la piatta orma
era portabilità:
u una piattafor-
unque altra!!!

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`***

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

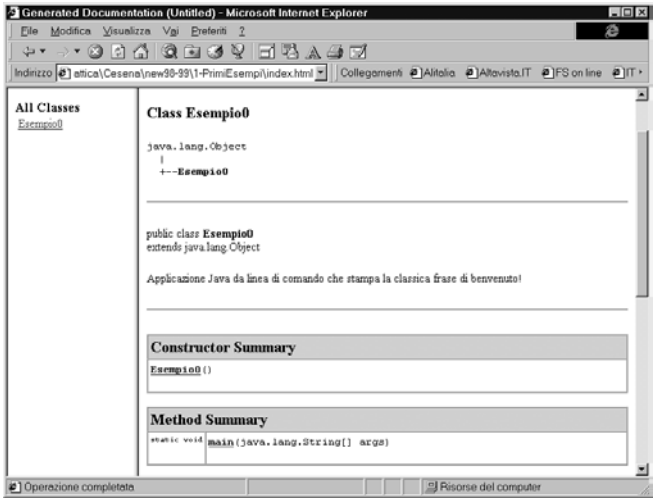
L'ESEMPIO... COMPLETATO

Per produrre la relativa documentazione:

```
javadoc Esempio0.java
```

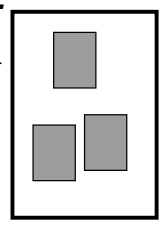
Produce una serie di file HTML

Si consulti la documentazione di javadoc per i dettagli.



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()`…**

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);
   }
}
```

UN ESEMPIO CON TRE CLASSI

- **Il nome di una classe (`Math` o `System`) definisce uno *spazio di nomi***
- **Per *usare* una funzione o ta der sp *co otazione puntata***

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

Inoltre, è immediato riconoscere *chi fornisce un certo servizio*

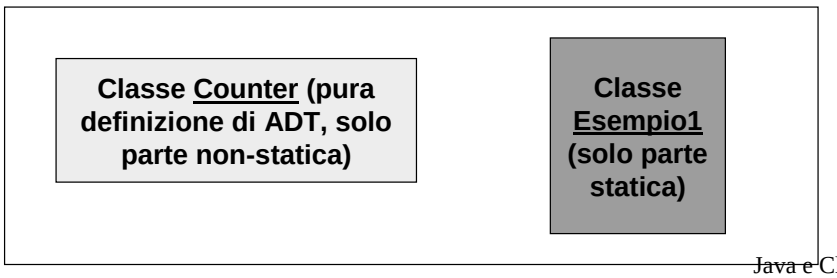
Ese

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


ESEMPIO COMPLETO

Programma fatto di <u>due classi</u>:

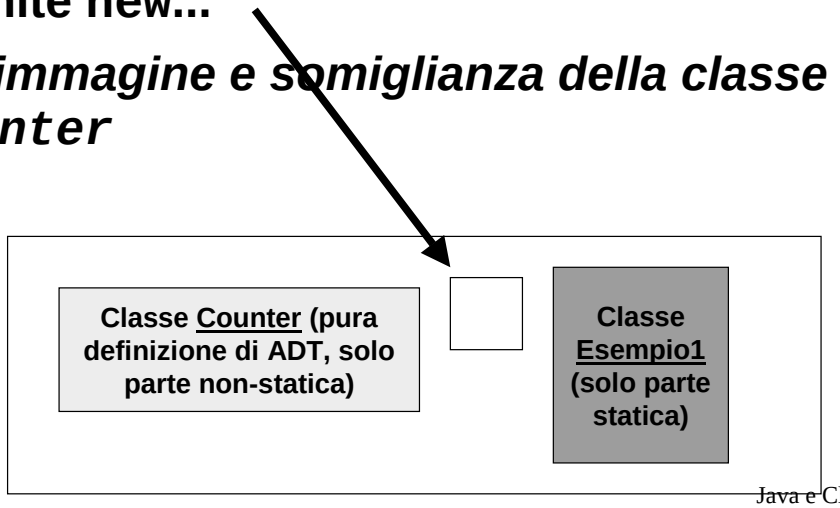
- **una che fa da componente software, e ha come compito quello di *definire il main*** *(solo parte statica)*
- ***l'altra invece implementa il tipo `Counter`*** *(solo parte non-statica)*



ESEMPIO COMPLETO

A run-time, nasce un oggetto:

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



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!***

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`

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`

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
```


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`**)

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**

FASE 1: Definizione dei Requisiti

Si progetti un sistema software per la gestione per conto della segreteria di facoltà dei:

• Dati relativi ai corsi di laurea (studenti iscritti, statistiche studenti, …)

• Dati relativi agli studenti (iscrizione degli studenti, gestione degli esami superati, …)

FASE 2: Identificazione delle Entità

Il sistema ha a che fare con entità concrete del mondo reale che devono essere modellate e rappresentate all'interno del sistema. In particolare:

- Studenti (con dati anagrafici, curriculum)
- Esami (con denominazione, docente)
- Corso di Laurea (con manifesti degli studi, studenti iscritti, …)

…altre entità di utilità per il sistema:

- Contatori (per contare iscritti, numero di esami dati,…)
- Date (ad es. date esami, date iscrizioni, …)

FASE 3: Identificazione degli Attributi e del Comportamento

Esame, Studente, Contatore => Classi

Classe Esami:

attributi: nome corso, promosso/bocciato, data superamento, voto

metodi: lettura nome esame e voto preso, servizi per dare un voto positivo (modificando gli attributi)

Classe Studente:

attributi: nome corso, promosso/bocciato, data superamento, voto

metodi: lettura nome esame e voto preso, servizi per dare un voto positivo (modificando gli attributi)

FASE 3: Identificazione degli Attributi e del Comportamento

Classe Corso di Laurea:

attributi: nome corso di laurea, lista studenti iscritti, ..

metodi: per iscrivere un nuovo studente, registrare un esame dato da uno studente, fare statistiche sugli studenti e sul loro curriculum, ….

……