IL CONCETTO DI CLASSE

Una *CLASSE* riunisce le proprietà di:

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

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!***

CLASSI IN JAVA

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

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

CLASSI E OGGETTI IN JAVA

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

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

CLASSI COME ADT

Una classe <u>con solo la parte NON-STATICA</u> è una *pura definizione di ADT*

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

ESEMPIO: IL CONTATORE

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

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

Dati

Operazioni (comportamento)

Unico costrutto linguistico per dati e operazioni

ESEMPIO: LA CLASSE Counter

- **Ques… …oni *sue p…***
- **Forni… …che** potrà es… …ggetti

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

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

Dati

Operazioni (comportamento)

Unico costrutto linguistico per dati e operazioni

OGGETTI IN JAVA

- **Gli OGGETTI sono componenti "dinamici": *vengono creati "al volo"*, al momento dell'uso, tramite l'operatore `new`**
- **Sono creati *a immagine e somiglianza*** *(della parte non statica)* ***di una classe,* che ne descrive le *proprietà***
- **Su di essi è possibile invocare *le operazioni pubbliche* previste dalla classe**
- **Non occorre preoccuparsi della distruzione degli oggetti: Java ha un *garbage collector!***

CREAZIONE DI OGGETTI

Per creare un oggetto:

- **prima si definisce un *riferimento,* il cui tipo è *il nome della classe che fa da modello***
- ***poi si crea dinamicamente l'oggetto* tramite *l'operatore new*** *(simile a malloc in C)*

Esempio:

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


OGGETTI IN JAVA

Uso: stile a "*invio di messaggi*"

- **non una funzione con l'oggetto come parametro…**
- **…ma bensì *un oggetto su cui si invocano metodi***

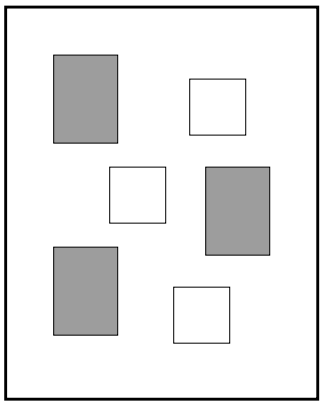
Ad esempio, se `c` è un `Counter`, un cliente potrà scrivere:

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

PROGRAMMI IN JAVA

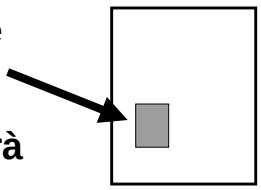
Un programma Java è *un insieme di classi e oggetti*

- **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

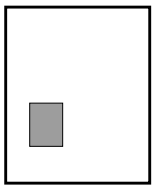
- **Il più semplice programma Java è dunque costituito da *una singola classe* operante come *singolo componente software***
- **Essa avrà quindi la sola parte statica**
- **Come minimo, tale parte dovrà definire *una singola funzione (statica): il `main`***



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)*

ESEMPIO COMPLETO

Programma fatto di due classi:

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

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!***

COSTRUZIONE DI OGGETTI

- **Molti errori nel software sono causati da *mancate inizializzazioni* di variabili**
- **Perciò i linguaggi a oggetti introducono il costruttore, un metodo particolare che *automatizza l'inizializzazione* degli oggetti**
  - **non viene *mai chiamato esplicitamente dall'utente***
  - **è invocato automaticamente dal sistema *ogni volta che si crea un nuovo oggetto* di quella classe**

COSTRUTTORI

Il costruttore:

- **ha un nome fisso, *uguale al nome della classe***
- **non ha tipo di ritorno, neppure `void`**
  - **il suo scopo infatti non è "calcolare qualcosa", ma *inizializzare un oggetto***
- **può *non essere unico***
  - **spesso vi sono *più costruttori*, con diverse liste di parametri**
  - **servono a inizializzare l'oggetto a partire da *situazioni diverse***

ESEMPIO

La classe `Counter`

```
public class Counter {
  private int val;

  public Counter() { val = 1; }
  public Counter(int v) { val = v; }

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

Costruttore senza parametri

Costruttore con un parametro

ESEMPIO: UN CLIENTE

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

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

Qui scatta il costruttore/1 → **c2** inizializzato a 10

COSTRUTTORI - NOTE

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

RIUSABILITÀ

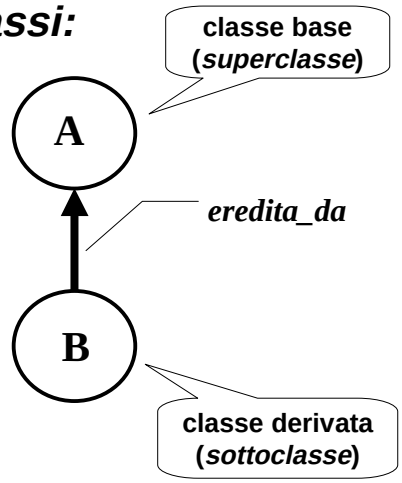
- **Si vuole riusare tutto ciò che può essere riusato (*componenti*, *codice*, *astrazioni*)**
- **Non è utile né opportuno modificare codice *già funzionante* e *corretto***
  - **il cui sviluppo ha richiesto tempo (anni-uomo)**
  - **ed è costato (molto) denaro**
- **Occorre disporre nel linguaggio di un modo per *progettare alle differenze*, procedendo *in modo incrementale***

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

- **Una *relazione tra classi:* si dice che la nuova classe B *eredita dalla* preesistente 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**

EREDITARIETÀ

Cosa si eredita?

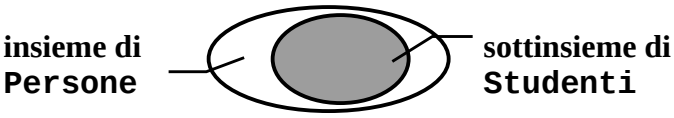
- **tutti i dati della classe base**
  - **anche quelli privati, a cui comunque la classe derivata non potrà accedere direttamente**
- **tutti i metodi...**
  - **anche quelli che la classe derivata non potrà usare direttamente**
- ***... tranne i costruttori*, perché sono specifici di quella particolare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

EREDITARIETÀ E COSTRUTTORI

- **Una classe derivata *non può prescindere dalla classe base*, perché ogni istanza della classe derivata *comprende* in sé, indirettamente, un oggetto della classe base**
- **Quindi, *ogni costruttore della classe derivata deve invocare un costruttore della classe base* affinché esso costruisca la "parte di oggetto" relativa alla classe base stessa:**

"ognuno deve costruire quello che gli compete"

UN ESEMPIO COMPLETO



- **Una classe `Persona`**
- **e una sottoclasse `Studente`**
  - ***<u>è aderente alla realtà</u>*, perché è vero nel mondo reale che tutti gli studenti sono persone**
  - **<u>compatibilità di tipo</u>: potremo usare uno studente (che è *anche* una persona) ovunque sia richiesta una generica persona**
    <u>ma non viceversa</u>: se serve uno studente, non si può accontentarsi di una generica persona!

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");
 }
}
```

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);
 }
}
```


LA CLASSE Studente

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

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**

CLASSI FINALI

- **Una classe finale (`final`) è una classe di cui si vuole *impedire a priori* che possano essere definite, un domani, delle sottoclassi**
- **Esempio:**

```
public final class TheLastCounter
   extends Counter {
   ...
}
```


OLTRE LE CLASSI

Possiamo anche avere classi e ereditarietà multipla

- **una classe** può derivare da **più di una classe esistente**

Nuova relazione tra classi

- implementazione difficile per possibili conflitti
- difficile gestione della gerarchia di ereditarietà delle classi, sia uso, sia supporto

OLTRE LE CLASSI

Può essere utile disporre di un *nuovo costrutto*

- **simile alla (parte non statica di una) classe,** nel senso di consentire la definizione del "modo di interagire" di un'entità...
- **… ma non tenuto a fornire *implementazioni*…**
- **… né legato alla *gerarchia di ereditarietà* delle classi, con i relativi vincoli**

INTERFACCE

INTERFACCE

Una *interfaccia* costituisce una *pura specifica di interfaccia*

- **contiene *solo dichiarazioni* di metodi**
- **ed eventualmente costanti**
- **ma non contiene né variabili né *definizioni* di metodi**

INTERFACCE

Praticamente, una *interfaccia*

- **è strutturalmente *analoga alla parte di interfaccia di una classe...***
- **ma è introdotta dalla parola chiave `interface` anziché `class`**
- **contiene solo *dichiarazioni* di metodi**

Esempio:

```
public interface Comparable {
  public int compareTo(Object x);
}
```


INTERFACCE E PROGETTO

Le interfacce inducono un diverso *modo di concepire il progetto*

- **prima si definiscono le interfacce delle entità che costituiscono il sistema**
  - **in questa fase si giocano *scelte di progetto (pulizia concettuale)***
- **poi si realizzeranno le classi che "implementeranno" tali interfacce**
  - **in questa fase entreranno in gioco *scelte implementative (efficienza ed efficacia)***

UN ESEMPIO di PURA SPECIFICA

Definizione dell'astrazione "Collezione"

- **Cosa si intende per "Collezione"?** ossia
- ***Come ci si aspetta di poter interagire*** **con un'entità qualificata come "Collezione"?**

Indipendentemente da qualsiasi scelta o aspetto implementativo, **una "Collezione" è tale perché**

- **è un contenitore → è possibile chiedersi se è vuota e quanti elementi contiene**
- **vi si possono aggiungere e togliere elementi**
- **è possibile chiedersi se un elemento è presente o no**
- **...**

UN ESEMPIO

Una "Collezione" è dunque una *qualsiasi entità che si conformi a questo "protocollo di accesso"*

- **è possibile chiedersi se è vuota**
- **è possibile chiedersi quanti elementi contiene**
- **vi si possono aggiungere e togliere elementi**
- **è possibile chiedersi se un elemento è presente o no**
- **...**

È possibile (e utile!) definire questo concetto *prima ancora di iniziare a pensare* come sarà realmente realizzata una "Collezione"!

un ESEMPIO di ASTRAZIONE PURA

Si definiscono così astrazioni di dato in termini di *comportamento osservabile*, ossia di

- **"cosa ci si aspetta" da esse**
- **"cosa si pretende che esse sappiano fare"**

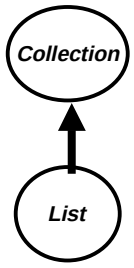
rinviando a tempi successivi la realizzazione pratica di ADT (classi) che rispettino questa specifica.

```
public interface Collection {
 public boolean add(Object x);
 public boolean contains(Object x);
 public boolean remove(Object x);
 public boolean isEmpty();
 public int size();
 ...
}
```


GERARCHIE DI INTERFACCE

Le interfacce possono dare luogo a *gerarchie*, *proprio come le classi:*

```
public interface List extends Collection {
 ...
}
```



La gerarchia delle interfacce:

- **è una gerarchia separata da quella delle classi**
- **è slegata dagli aspetti implementativi**
- **esprime le *relazioni concettuali* della realtà**
- **guida il progetto del *modello della realtà.***

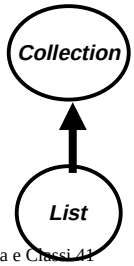
GERARCHIE: ESEMPIO

Come in ogni gerarchia, anche qui le interfacce derivate:

- **possono aggiungere nuove *dichiarazioni di metodi***
- **possono aggiungere nuove *costanti***
- ***non possono* eliminare nulla**

Significato: *"Ogni lista è anche una collezione"*

- **ogni lista può interagire col mondo come farebbe una collezione** *(magari in modo specializzato)***…**
- **… ma può avere *proprietà peculiari al concetto di lista*,** che non valgono per una "collezione" qualsiasi.

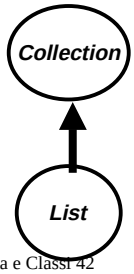


GERARCHIE: ESEMPIO

Ad esempio, una "Lista" ha un concetto di *sequenza,* di *ordine* fra i suoi elementi

- **esiste quindi un *primo elemento*, un secondo, …**
- **quando si aggiungono nuovi elementi bisogna dire *dove* aggiungerli** (ad esempio, in coda)
- **è possibile recuperare un elemento *a partire dalla sua posizione*** (il primo, il decimo,…)

```
public interface List extends Collection {
 public boolean add(int posizione, Object x);
 public Object get(int posizione);
 ...
}
```



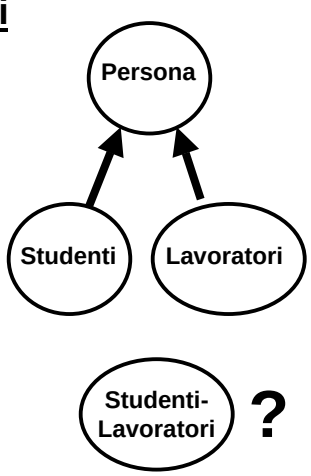
INTERFACCE: UN ALTRO ASPETTO

In Java, l'ereditarietà fra classi può essere solo *singola*

- **una classe può ereditare da *una sola superclasse***

Questo può essere limitativo in alcune circostanze:

- **se `Studenti` e `Lavoratori` estendono `Persona`...**
- ***...dove collocare la classe `StudentiLavoratori`?***



Uso Interfacce per ESTENSIONI

- **Le interfacce che introduciamo possono *servire a descrivere comportamenti condivisi***
  → **`Serializable`, `Runnable`, ...**
- **Una classe può dichiarare molte interfacce e differenziate, se ne implementa i metodi**
  Esempio

```
Public Class ActiveWindows implements
   Runnable, Serializable, ... {
}
```