



Programmazione III

Prof.ssa Liliana Ardissono

Informazioni sull'insegnamento



Pagine web dell'insegnamento

- Sul sito laurea.informatica.unito.it (con programma, modalità d'esame, libri di testo):
https://laurea.informatica.unito.it/do/corsi.pl>Show?_id=5076
- Su I-learn (moodle, con material online, testi di esami passati, registrazioni delle lezioni ...):
<https://informatica.i-learn.unito.it/mod/folder/view.php?id=220910>



Orario delle lezioni

Orario del terzo anno laurea triennale:

- [https://unito.prod.up.cineca.it/calendarioPubblico/linkCal
endarId=612617b82db4bb0017172839](https://unito.prod.up.cineca.it/calendarioPubblico/linkCalendarId=612617b82db4bb0017172839)



Pagina su I-learn

LEZIONI TEORICHE (in orario: Progr III T – Programmazione III):

- unificate - docente Prof.ssa Liliana Ardissono

TURNI DI LABORATORIO

- **in orario:**
 - Progr III Lab1 – Programmazione III
 - Progr III Lab2 – Programmazione III
 - Progr III Lab3 – Programmazione III
- **Lab1: cognomi da A a D - Docente Prof.ssa Liliana Ardissono**
- **Lab2: cognomi da E a O - Docente Prof.ssa Liliana Ardissono**
- **Lab3: cognomi da P a Z – Docente Prof.ssa Liliana Ardissono**



Programma - I

Programmazione ad eventi in Java - programmare interfacce grafiche

- Sorgenti di eventi, gestori di eventi, event-driven programming
- Organizzazione e uso delle interfacce grafiche di Java (GUI)
- L'architettura Model-View-Controller (MVC) per lo sviluppo di applicazioni modulari



Programma - II

Programmazione Multithread

- Esecuzione concorrente di istruzioni
- I Thread in Java: ciclo di vita dei Thread
- Creazione e sincronizzazione di thread
- Estensione del modello della memoria in presenza di Thread
- Problemi di sincronizzazione e loro risoluzione mediante il linguaggio Java



Programma - III

Programmazione in rete in Java

- L'architettura client-server
- Uso di socket per la comunicazione tra applicazioni distribuite
- Polimorfismo e trasferimento di oggetti mediante Java
- Il modello di esecuzione distribuita di oggetti



Teoria + sperimentazione

Questo insegnamento ha un carattere sperimentale: per ogni concetto teorico spiegato a lezione ci saranno esercitazioni software volte ad applicarlo e assimilarlo

Durante il corso si svilupperà un progetto SW (in java) di medie dimensioni per «mettere insieme» tutti i concetti appresi. Il progetto SW deve essere sviluppato in gruppi di al più 3 persone e viene consegnato in sede d'esame (vedere i lucidi successivi)



Libro di testo

Programmare in Java 11/Ed. - MyLab

Paul J. Deitel - Harvey M. Deitel

ISBN: 9788891916211

Consiglio di utilizzare anche il vostro **libro di testo di Programmazione II**, che copre alcuni contenuti di questo insegnamento (per esempio, l'ereditarietà e il polimorfismo)



Frequenza alle lezioni

Non obbligatoria ma caldamente consigliata sia per le lezioni teoriche che per le esercitazioni - *il che significa seguire le lezioni di persona, se possibile, o scaricare con puntualità e studiare le registrazioni delle lezioni/esercitazioni*

Consiglio di mantenere il passo con le lezioni, indipendentemente dal fatto che siano di persona o registrate: studiare regolarmente è fondamentale per completare il corso con successo



Modalità d'esame

- L'esame si compone di una **prova teorica** e di una **discussione del progetto SW**

Le due prove possono essere svolte nell'ordine che si preferisce ma devono essere completate entrambe **entro l'appello che precede l'inizio della successiva edizione dell'insegnamento.** *In altre parole, avete un anno di tempo per completare l'esame a partire dall'inizio del corso*

Esame: prova teorica

Esame scritto su carta che include esercizi e domande teoriche sul programma dell'insegnamento. Viene valutata da un minimo di 0 ad un massimo di 30 (31 vale per la lode) e si considerano sufficienti i voti ≥ 18 .

Questa prova è volta a verificare che gli studenti e le studentesse abbiano acquisito la conoscenza fornita dall'insegnamento in termini di sapere e a verificare e allenare le abilità comunicative e l'autonomia di giudizio. Il voto ottenuto durante la prova teorica decade se si partecipa ad un'altra prova teorica.

Esame: prova teorica

La possibilità di **svolgere la prova online** dipende dalle disposizioni dell'Ateneo: <https://www.unito.it/ateneo/gli-speciali/coronavirus-aggiornamenti-la-comunita-universitaria/disposizioni-chi-studia-e>

E' necessario controllare periodicamente le disposizioni dell'Ateneo, che potrebbero variare nel tempo.

Chi ha diritto di partecipare online deve scriverlo nelle note quando si iscrive agli appelli. Se ci si ammala di COVID all'ultimo minuto, a iscrizioni scadute, mandare mail alla docente, tempestivamente, per informare della propria assenza o del fatto che si svolgerà la prova in remoto.



Esame: discussione del progetto SW

La discussione deve essere effettuata preferibilmente in unica soluzione, con tutti i membri del gruppo presenti. I gruppi di laboratorio devono includere al massimo 3 persone.

Il voto di laboratorio è un numero compreso tra 0 e 30 (31 vale per la lode), si considerano sufficienti i voti ≥ 18 . Questa prova è volta a verificare le competenze acquisite (saper fare) e a verificare e allenare le abilità comunicative e l'autonomia di giudizio.

La discussione del progetto SW si terrà preferenzialmente online, utilizzando la piattaforma Webex di Ateneo. Gli studenti e le studentesse potranno comunque richiedere di effettuare la prova in presenza inviando e-mail alla docente.

NB: non si richiede di consegnare i progetti di laboratorio prima dell'esame.



Esame: calcolo del voto finale

Il voto finale dell'esame viene determinato come media aritmetica semplice del voto della prova di teoria e del voto della discussione del progetto SW.

Cioé, **voto finale = (voto teoria + voto progetto SW)/2.**

Inoltre, secondo il regolamento di Ateneo, se prendete 30 in entrambe le prove, il voto finale sarà 30 e lode.

NB: i voti acquisiti durante ciascuna delle prove rimangono validi fino al termine della sessione d'esame che precede l'inizio del nuovo insegnamento. Quando si superano entrambe le prove, è necessario registrare il voto finale entro i limiti imposti dal Regolamento di Ateneo.

Ambiente di sviluppo per le attività di laboratorio



IDE per svolgere gli esercizi di programmazione e sviluppare il progetto SW (da fare prima dell'inizio delle esercitazioni in laboratorio):

- Scaricare sul proprio computer **IntelliJ Idea ULTIMATE** che offre licenza education.

IntelliJ ULTIMATE download:

<https://www.jetbrains.com/idea/download/>

Installare JAVA13 o successivo (far girare IntelliJ con Java13 o più recente) per compatibilità con JavaFX. Meglio se si installa Java20, attualmente disponibile.



Programmazione III

Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

Introduzione alla progettazione a oggetti



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND

Modellare la realtà - I

STATO
via1: verde
via2: rosso



STATO
motore: acceso
velocità: 0

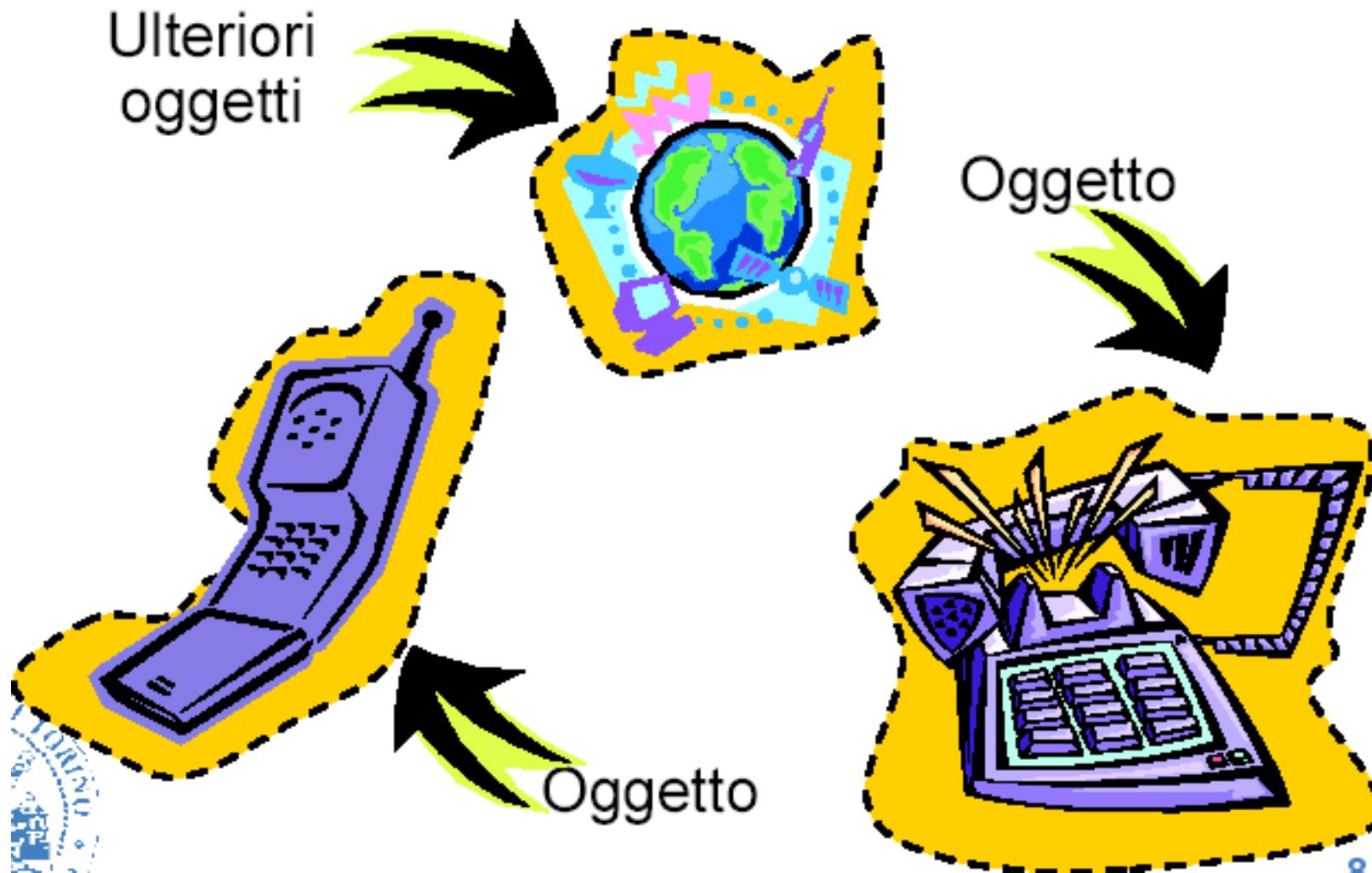
COMPORTAMENTO
Parti!
Frena!
Sterza!



Modellare la realtà - II

- Stato
 - L'insieme dei parametri caratteristici che contraddistinguono un oggetto in un dato istante
 - Modellato come insieme di **attributi**
- Comportamento
 - Descrive come si modifica lo stato a fronte degli stimoli provenienti dal mondo esterno
 - Modellato come insieme di **metodi**

Approccio nell'osservare il mondo





Oggetti e realtà

- Il mondo fisico è costituito da un insieme di oggetti variamente strutturati che interagiscono tra loro
- Ciascuno è dotato di:
 - Una propria **identità** (è riconoscibile)
 - Uno **stato** (ricorda la storia passata)
 - Un **comportamento** (reagisce a stimoli esterni in un modo prevedibile)
- Si può estendere la metafora al software
 - Ogni entità logica che deve essere manipolata può essere immaginata come un “oggetto”

Stato



- Ogni oggetto ha uno **stato**:
 - L'insieme dei parametri caratteristici che contraddistinguono un oggetto in un dato istante
- Composto da un gruppo di “attributi”
 - Ogni attributo modella un particolare aspetto dello stato
 - Può essere un valore elementare o un altro oggetto
- Implementato mediante un blocco di memoria
 - Contiene i valori degli attributi
- Principio fondamentale: **incapsulamento**
 - Lo stato “appartiene” all’oggetto
 - Un utente esterno non può manipolare direttamente lo stato di un oggetto

Comportamento



- Gli oggetti interagiscono a seguito di “richieste esterne”(invocazioni di metodi)
 - Dotate di eventuali parametri che ne specificano i dettagli
- Ogni oggetto sa reagire ad un ben determinato insieme di invocazioni di metodi
 - Costituiscono la sua interfaccia
- Ad ogni richiesta è associato un comportamento
 - Modifica dello stato
 - Invio di richieste verso altri oggetti
 - Comunicazione di informazioni (risultato)
- Implementato attraverso un blocco di codice (**metodo**)
- Principio fondamentale: **delega** - chi effettua la richiesta non vuole conoscere i dettagli di come la richiesta sia evasa

La programmazione orientata agli oggetti (secondo Alan Kay – Smalltalk)



- Ogni cosa è un *oggetto*
- Un programma è un insieme di *oggetti* che si “dicono l’un l’altro” che cosa fare invocando i metodi reciprocamente offerti
- Ogni *oggetto* può contenere riferimenti ad altri *oggetti*
- Ogni *oggetto* ha un tipo (*classe*), cioè ogni oggetto ha proprietà strutturate in campi definiti dalla *classe*
- Tutti gli *oggetti* di un determinato tipo possono rispondere alle stesse invocazioni di metodi

Tipo di dato astratto



- Definisce le operazioni fondamentali sui dati, ma non ne specifica l'implementazione

Per es. una *lista* (astratta) è una sequenza ordinata di dati

- le operazioni possibili sono:
 - lettura sequenziale
 - inserimento/rimozione di un elemento in posizione i-esima

Una **struttura-dati** è vista come l'insieme di operazioni (*servizi*) offerti



Il ruolo dell'astrazione

- Astrazioni procedurali
- Astrazioni dei dati

Obiettivo: trattare cose complesse come primitive e nascondere i dettagli

Domande:

- **Quanto è facile suddividere il sistema in moduli di astrazioni?**
- **Quanto è facile estendere il sistema?**

La programmazione orientata agli oggetti



Object-oriented design = progettazione (e sviluppo) orientato agli oggetti = costruzione di sistemi software visti come collezioni strutturate di (implementazioni di) **strutture-dati astratte** [B. Meyer, "Object-oriented software construction ", Prentice Hall, 1988, cap.4.8]

Programmazione: procedurale vs OO



- **Programmazione procedurale**
 - Organizzare il sistema intorno alle procedure che operano sui dati
- **Programmazione ad oggetti**
 - Organizzare il sistema intorno ad oggetti che interagiscono mediante invocazione di metodi
 - Un oggetto *incapsula* dati e operazioni

La programmazione procedurale

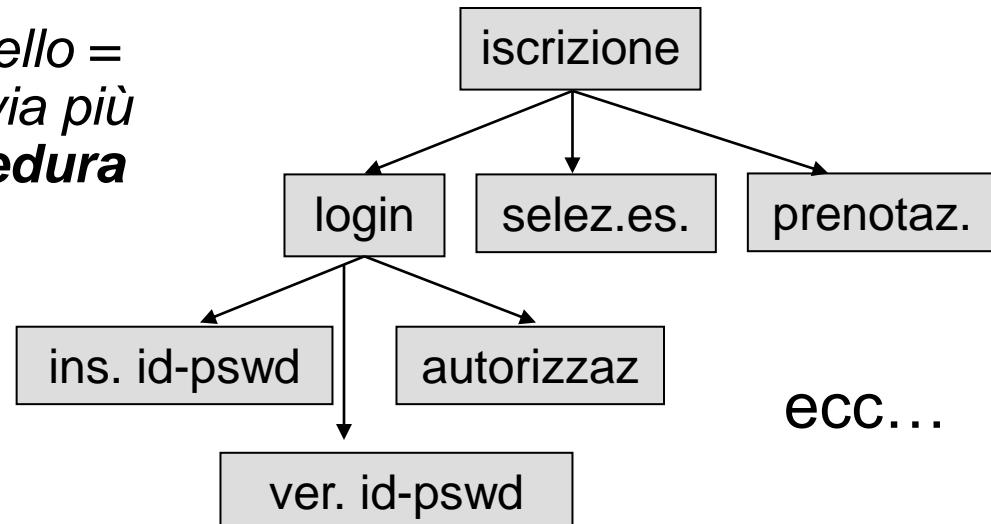


Metodo classico di software design = *top-down functional (structured) design* = scomposizione gerarchica funzionale (algoritmica)



paradigma procedurale: algoritmo = procedura = sequenza di passi per raggiungere il risultato

Per es. *iscrizione ad un appello* = scomposizione in passi via via più semplici, elementari = **procedura** per iscriversi ad un appello



PROGRAMMI = ALGORITMI + STRUTTURE-DATI

La programmazione orientata agli oggetti

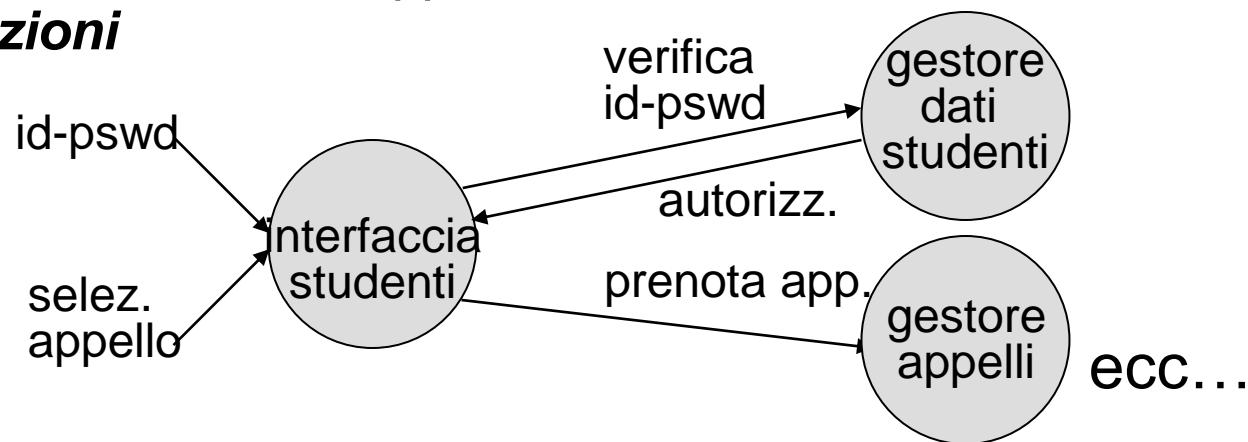


Metodo alternativo = **object-oriented design** = si parte dagli oggetti, non dalle funzionalità!



paradigma ad oggetti: oggetti che interagiscono tra loro mediante invocazione di metodi = collaborazione per raggiungere il risultato

Per es. iscrizione ad un appello = **entità coinvolte nell'attività e loro relazioni**

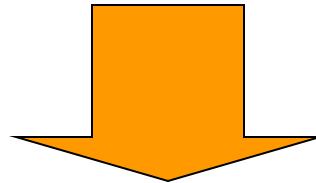


PROGRAMMI = OGGETTI (DATI + ALGORITMI) + COLLABORAZIONE (INTERFACCE)

Sviluppare un programma ad oggetti



- Un oggetto è un *fornitore di servizi*
- Un programma fornisce un servizio agli utenti e lo realizza utilizzando servizi di altri oggetti



- Obiettivo: produrre (o trovare librerie di oggetti già esistenti) l'insieme di oggetti che forniscono i servizi ideali per risolvere il problema

Progettare a oggetti



- Si parte dal testo delle specifiche
 - Si individuano i nomi e i verbi
- Tra i nomi, si individuano le possibili classi di oggetti
 - Con i relativi attributi
- Tra i verbi, si individuano metodi e relazioni
 - Di solito, i verbi di azione si modellano come metodi (“X apre Y”), quelli di stato come relazioni (“A si trova presso B”)
 - L’interazione tra due oggetti sottende l’esistenza di una relazione tra gli stessi

La programmazione orientata agli oggetti



Che cos'è un "oggetto"?

Passo 1: Distinguere classi e istanze

Passo 2: Distinguere interfaccia e implementazione

Passo 1: Distinguere classi e istanze

- Un'**istanza (oggetto)** è un'entità concreta, che esiste nel tempo (viene costruita e poi distrutta) e nello spazio (occupa memoria)
- Una **classe** è un'astrazione che rappresenta le proprietà comuni (struttura e comportamento) ad un insieme di oggetti concreti (istanze)

La programmazione orientata agli oggetti



Esempio: supponiamo di gestire una biblioteca, che contiene moltissimi libri

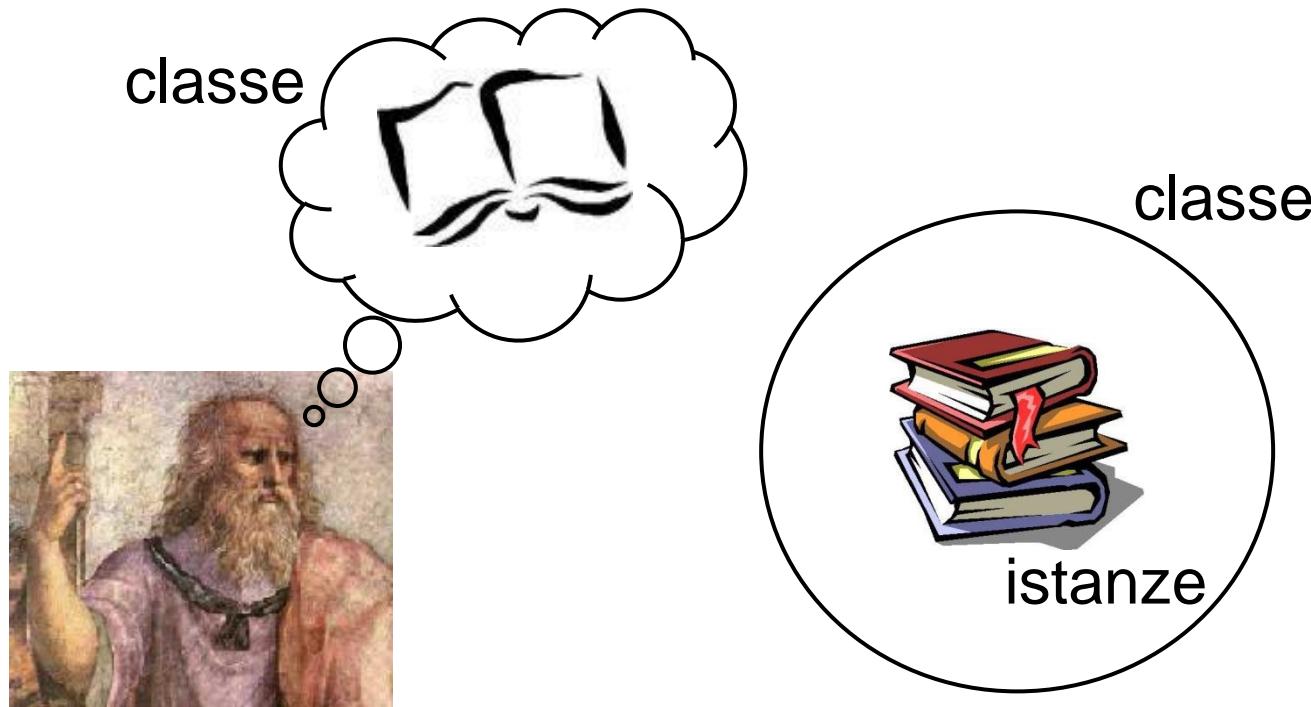
Una classe è...

- L'insieme di tutti i libri, la classe dei **libri**
- La proprietà *Libro(x)*, che definisce l'appartenenza all'insieme dei libri, ed è vera per tutti i libri (gli oggetti *x* che sono libri)
- L' "idea platonica" di libro, il prototipo ideale di libro, che esiste solo nel mondo delle idee; tutti i libri della nostra biblioteca "partecipano" dell'idea di libro, da momento che sono libri!
- Il concetto mentale di libro, che esiste solo nella nostra testa e di cui i libri del mondo sono degli esempi concreti
- ...

La programmazione orientata agli oggetti

Un'istanza (oggetto) è...

- Un singolo libro concreto (che può essere preso in prestito, restituito, distrutto, fotocopiato, ecc...)



La programmazione orientata agli oggetti

⇒ Una **classe** può essere vista come la definizione di un **tipo di dato astratto**

Per esempio, supponiamo che la biblioteca riceva un nuovo libro. Per prima cosa il bibliotecario deve classificarlo come libro (e non rivista, CD Rom, o altro), dichiarando quindi che l'oggetto appena arrivato è un libro.

Questo equivale a dichiarare che **il nuovo oggetto è di tipo "libro"** (cioè che alla domanda "cos'è questo?" rispondiamo "è un libro!")

Un tipo è un **modello** (un **template**) che definisce il **comportamento e la struttura** di un'insieme di istanze (oggetti).

Per esempio, il tipo "libro" definisce le operazioni che possono essere fatte sui libri (istanze): prestito, restituzione, ecc...

La programmazione orientata agli oggetti



⇒ Un'**istanza** è un oggetto concreto (di un certo tipo, cioè appartenente ad una certa classe), caratterizzato da:

- un'**identità**: possibilità di identificare univocamente l'oggetto
- uno **stato**: l'insieme dei valori dei suoi attributi, in un certo tempo t
- un **comportamento**: l'insieme delle operazioni (funzionalità) offerte dall'oggetto, cioè le cose che l'oggetto è in grado di fare

La programmazione orientata agli oggetti

Torniamo al nostro esempio:

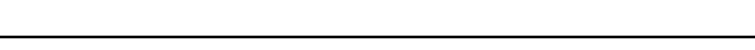
supponiamo di gestire una biblioteca, che contiene molti libri; nella biblioteca c'è un bibliotecario che classifica i nuovi libri, assegna i prestiti, ecc. e ci sono degli utenti che prendono in prestito i libri della biblioteca

Quali sono gli **oggetti** coinvolti nello scenario?

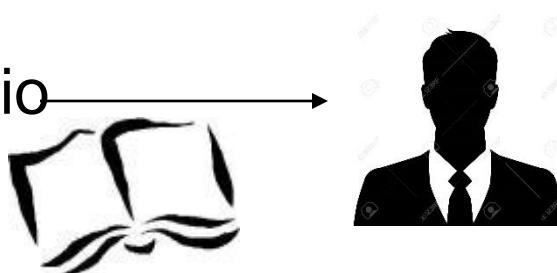
In particolare, quali sono le **classi** e quali le **istanze**?

Abbiamo bisogno dei seguenti **concetti (classi, tipi)**:

- la Biblioteca



- il Bibliotecario



- il Libro



- l'Utente



La programmazione orientata agli oggetti

Per ogni **concetto (classe, tipo)** di quali **proprietà (attributi, caratteristiche)** abbiamo bisogno per descriverlo in modo adeguato?



- per la Biblioteca:

- nome
 - indirizzo
 - orario apertura

- per il Bibliotecario:

- nome
 - turno

- per il Libro:

- autore
 - titolo
 - editore
 - collocazione

- per l'Utente

- nome
 - cognome
 - telefono



La programmazione orientata agli oggetti

Per ogni concetto (classe, tipo) di quante **istanze** (oggetti concreti) abbiamo bisogno?

- **una sola biblioteca** (un'istanza della classe Biblioteca), per la quale, per es:
 - nome = Biblioteca A. Gramsci
 - indirizzo = via Tizio 32, Roma
 - orario apertura = lun-sab 9:00-19:00
- **2 bibliotecari** (istanze della classe Bibliotecario), uno per il turno del mattino e uno per il turno del pomeriggio, per i quali, per es:



bibliotecario 1:

- nome = Paolo
- turno = mattino

bibliotecario 2:

- nome = Luca
- turno = pomeriggio

La programmazione orientata agli oggetti

- **un grande numero di libri** (istanze della classe Libro), per i quali, per es:

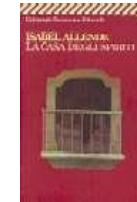
libro 1:

- autore = C.S. Horstmann
- titolo = Java 2
- editore = Apogeo
- collocazione = S21/L303



libro 2:

- autore = I. Allende
 - titolo = La casa degli spiriti
 - editore = Feltrinelli
 - collocazione = S13/L44
- ecc...



- **un certo numero di utenti** (istanze della classe Utente), per i quali, per es:

utente 1:

- nome = Maria
 - cognome = Bianchi
 - telefono = 011 1234567
- ecc...

La programmazione orientata agli oggetti

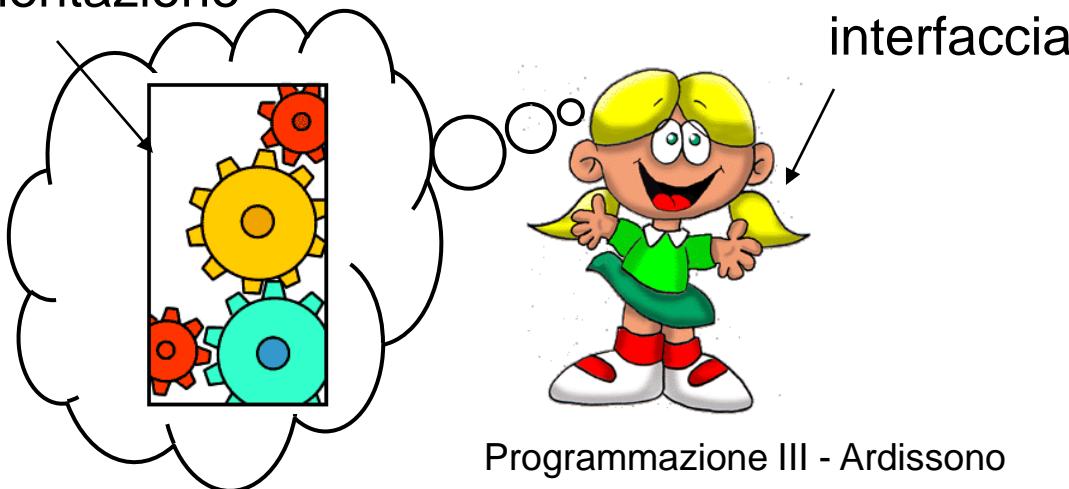


Passo 2: Distinguere interfaccia e implementazione

Quando definisco una **classe (tipo)** ne definisco:

- l'**interfaccia** = la “vista esterna” = l’insieme di operazioni che le sue istanze potranno fare
- l'**implementazione** = la “vista interna” = la definizione dei meccanismi che realizzano le operazioni definite nell’interfaccia

implementazione



La programmazione orientata agli oggetti

Torniamo al nostro **esempio** della biblioteca e consideriamo il Bibliotecario:

- quali **servizi (operazioni)** offre al pubblico?



prestito(libro)
restituzione(libro)
prenotazione(libro)

questi servizi
(operazioni) sono
accessibili al pubblico

= **interfaccia**

La programmazione orientata agli oggetti



Come li **implementa** (realizza)?

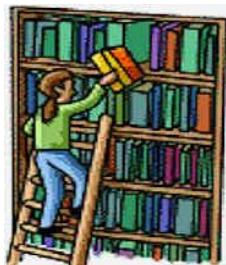
prestito(libro) →

procedura, per trovare il libro,

prenderlo, darlo all'utente,
registrare il

prestito sulla scheda, ...

} **non sono visibili
al pubblico**



= **implementazione**

La programmazione orientata agli oggetti

L'**interfaccia** definisce dunque il **comportamento** di un oggetto: nell'esempio, i servizi, cioè le operazioni di *prestito(libro)*, *restituzione(libro)*, *prenotazione(libro)* definiscono il comportamento dei bibliotecari (cioè di tutte le istanze della classe Bibliotecario: Paolo e Luca nell'esempio)

Come avviene l'**interazione** con un oggetto?

- avviene con un'**istanza** (con Paolo o Luca) e non con la classe (non si interagisce con il concetto di Bibliotecario!)
- **Invocando un metodo su** un'istanza, con il quale gli si chiede il servizio desiderato; nell'esempio di deve parlare o scrivere a Paolo o Luca per avere un libro in prestito, o per restituirlo, o prenotarlo

La programmazione orientata agli oggetti



1 ogni biblioteca può avere tanti (n) bibliotecari n



ogni bibliotecario può appartenere ad una sola biblioteca

| | |
|---|---|
| 1 | m |
| ogni biblioteca può avere tanti (n) libri | ogni libro può appartenere ad una sola biblioteca |



ogni biblioteca può avere tanti (n) utenti
ogni utente può essere iscritto a tante (m) biblioteche



La programmazione orientata agli oggetti

I principi fondamentali dell'*object-oriented*:

- **Astrazione**



Un'astrazione rappresenta le caratteristiche essenziali e distintive di un oggetto, dal punto di vista di chi lo guarda
[Grady Booch, *Object Oriented Design*, Benjamin/Cummings, 1991 p. 39]

La programmazione orientata agli oggetti



- **Incapsulamento** (*information hiding*)



L'incapsulamento (o *information hiding*) è il principio secondo cui la struttura interna, il funzionamento interno, di un oggetto **non deve essere visibile** dall'esterno

La programmazione orientata agli oggetti



⇒ ogni oggetto è costituito da 2 parti:

- l'*interfaccia* (vista “esterna”) → visibile
- l'*implementazione* (vista “interna”) → nascosta

L’incapsulamento (o *information hiding*) è il processo che **nasconde** quei dettagli, relativi al funzionamento di un oggetto, che non costituiscono le sue caratteristiche essenziali e distintive [BOOCH, p. 46]

La programmazione orientata agli oggetti



- ***Modularità***

La modularità consiste nella suddivisione di un sistema in una serie di **componenti indipendenti**, che interagiscono tra loro per ottenere il risultato desiderato

scelta dei moduli e
delle loro interazioni



definizione dell'architettura
del sistema

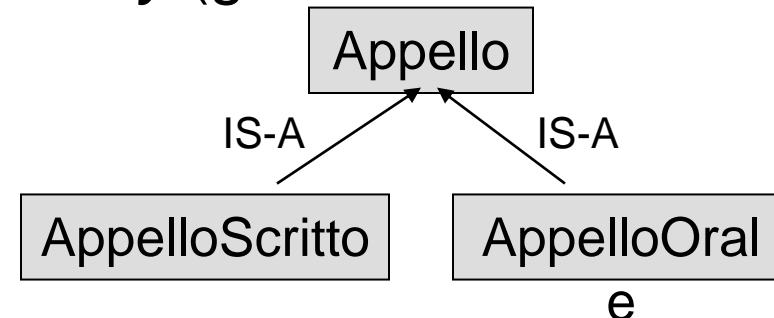
La programmazione orientata agli oggetti

Struttura gerarchica

In un sistema complesso, le due principali **gerarchie** sono:

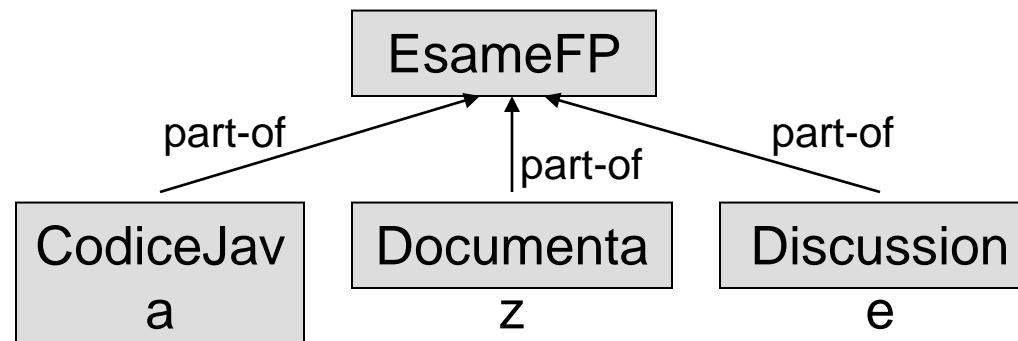
- *kind-of hierarchy* (gerarchia di **classi** e **sotto-classi**)

Per es.



- *part-of hierarchy* (gerarchia di **parti**)

Per es.





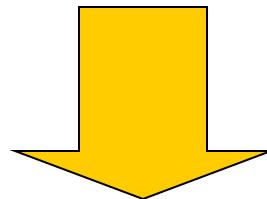
Vantaggi dell' approccio *object-oriented*

- Riuso
- Maggiore leggibilità
- Dimensioni ridotte
- Estensione e modifica più semplici
- Compatibilità
- Portabilità
- Manutenzione del software semplificata
- Migliore gestione del team di lavoro



Riuso

- Approccio procedurale:
 - occorreva conoscere tutto il software
- Approccio ad oggetti:
 - Occorre conoscere l'interfaccia delle classi ma non l'implementazione



- Molto più conveniente in termini di costi



Ringraziamenti

Grazie alla Prof.ssa Annamaria Goy del
Dipartimento di Informatica dell'Università di
Torino per aver redatto la prima versione di queste
slides.



Programmazione III

Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

JAVA – Ereditarietà – parte 1



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND



Ereditarietà 1

- Meccanismo per lo sviluppo incrementale di programmi
- Consente di estendere classi preesistenti aggiungendo o modificando componenti (variabili o metodi)



Ereditarietà 2

Data la classe

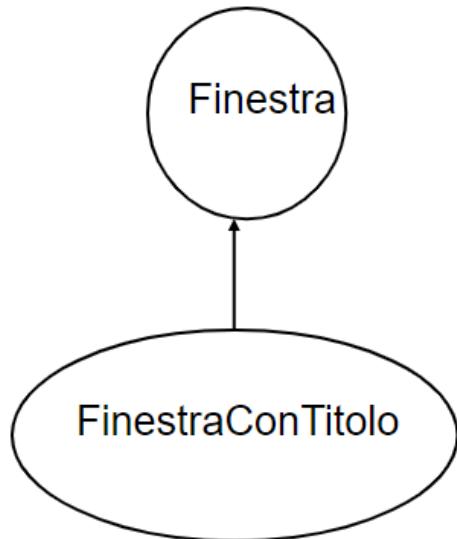
```
class Finestra {  
    Rettangolo r; ....  
    void disegnaCornice() {...}  
    void disegnaContenuto() {...}  
}
```

vogliamo estendere la finestra aggiungendo un titolo

```
class FinestraConTitolo extends Finestra {  
    String titolo;  
    void disegnaTitolo() {...}  
}
```



Sottoclassi



FinestraConTitolo è una **sottoclasse** di Finestra: gli oggetti di tipo FinestraConTitolo sono anche oggetti di tipo Finestra.

Una sottoclasse **eredita** tutte le variabili e i metodi della sopraclassse.



Ereditarietà 3

Gli oggetti della classe **Finestra** sono costituiti da

variabili

Rettangolo r;

metodi

void disegnaCornice() {...}

void disegnaContenuto() {...}

Gli oggetti della classe **FinestraConTitolo** sono costituiti da

variabili

Rettangolo r;

String titolo;

metodi

void disegnaCornice() {...}

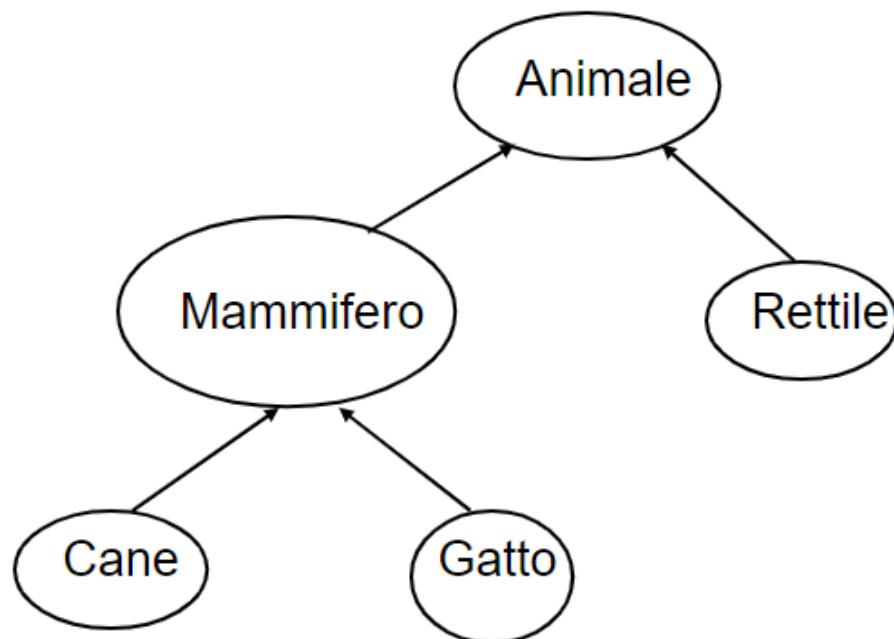
void disegnaContenuto() {...}

void disegnaTitolo() {...}



Tassonomie

Spesso l'ereditarietà è utilizzata per rappresentare tassonomie (classificazioni)





Controllo statico dei tipi 1

Java, come molti altri linguaggi, effettua un controllo dei tipi (**type checking**) **statico**.

Statico: fatto dal compilatore prima di iniziare l'esecuzione del programma. Identifica errori sintattici e mismatch di tipo tra variabili, guardando le loro dichiarazioni.

*(C'è poi il controllo di tipo **Dinamico**: fatto dall'interprete durante l'esecuzione (a runtime), che vedremo in seguito)*



Controllo statico dei tipi 2

```
Finestra f;  
FinestraConTitolo ft;
```

```
...
```

```
ft.disegnaCornice();  
f.disegnaCornice();  
ft.disegnaTitolo();  
f.disegnaTitolo(); // errore di compilazione
```

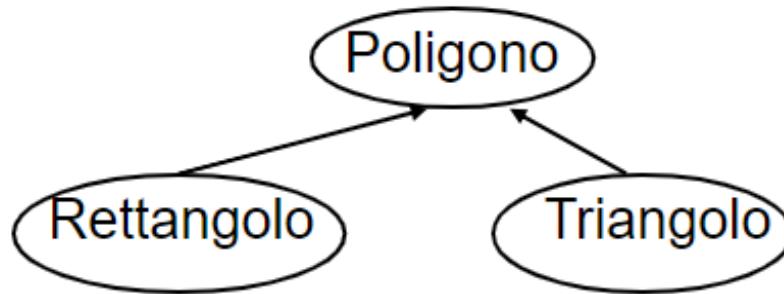
// f è una finestra e non ha il metodo *disegnaTitolo()*

Type checking statico: il compilatore controlla che per una variabile si chiami un metodo definito per la classe di quella variabile.



Sottotipi

Una sottoclasse può essere vista come l'implementazione di un **sottotipo**.



L'ereditarietà realizza una relazione **Is-a (è un)**.

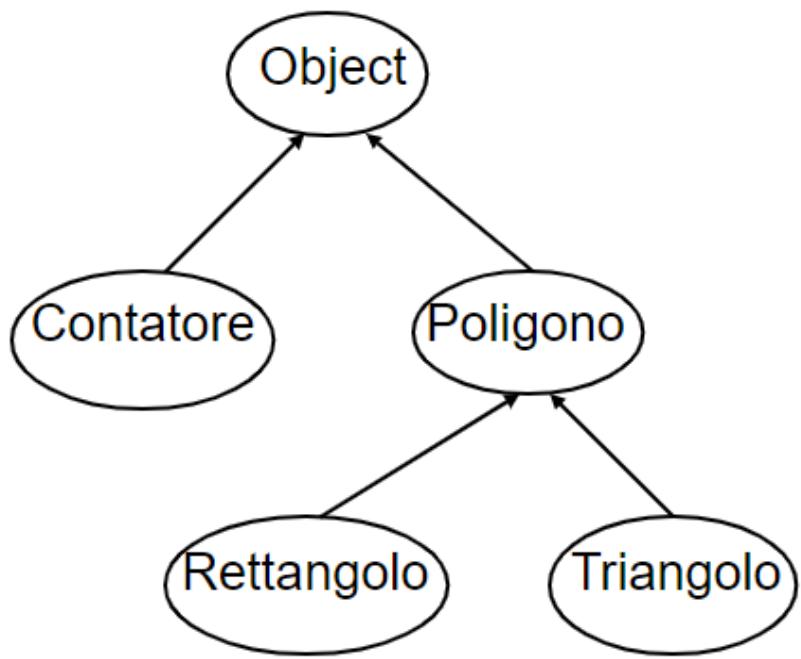
Un rettangolo **è un** poligono.

Un rettangolo ha tutte le operazioni di poligono
(eventualmente ridefinite) più altre operazioni specifiche.

Ereditarietà singola



Ogni sottoclasse ha una sola sopraclassse.
Struttura ad albero.



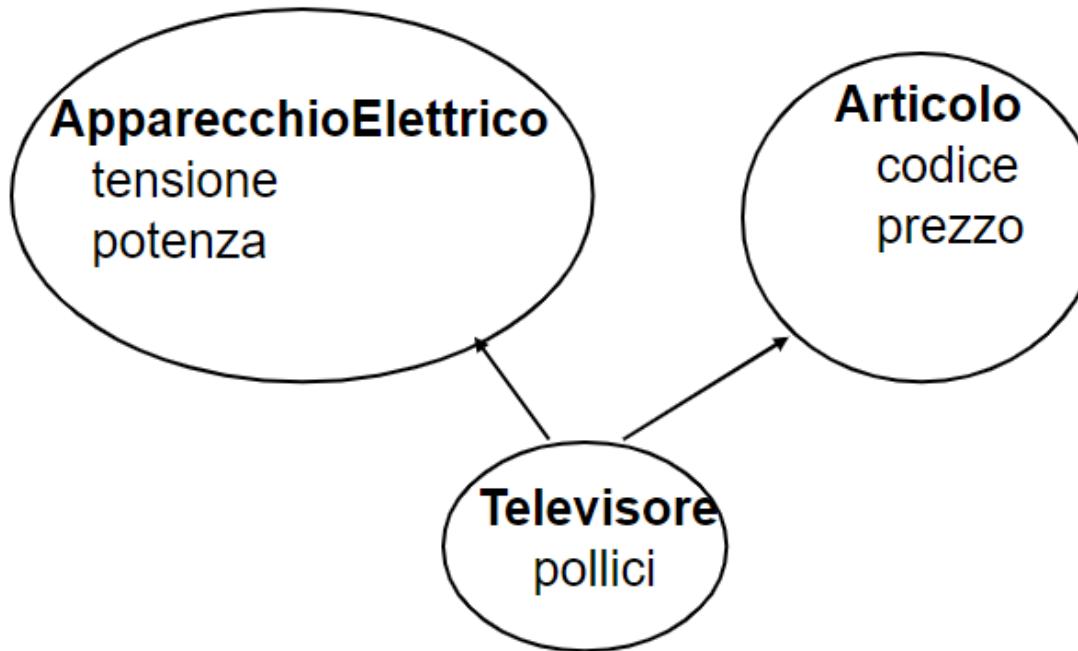
In Java la classe **Object** è la radice della gerarchia.

Qualunque oggetto è un **Object**.

I tipi primitivi non sono **Object**.



Ereditarietà multipla

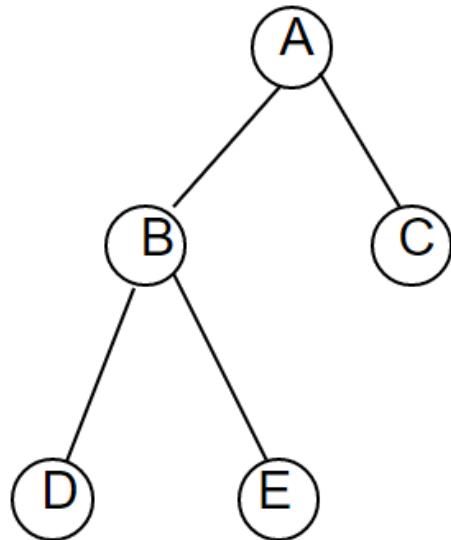


In Java l'ereditarietà multipla non è permessa.
L'ereditarietà multipla dà maggiore espressività ma
crea problemi di conflitti e duplicazioni.



Polimorfismo 1

POLIMORFISMO: proprietà di un oggetto di avere più di un tipo, in accordo alla relazione di ereditarietà.



D sottoclasse di B e di A

un oggetto D è un B ed è un A

un oggetto di tipo (classe) D è anche un oggetto di tipo (classe) B e anche un oggetto di tipo (classe) A

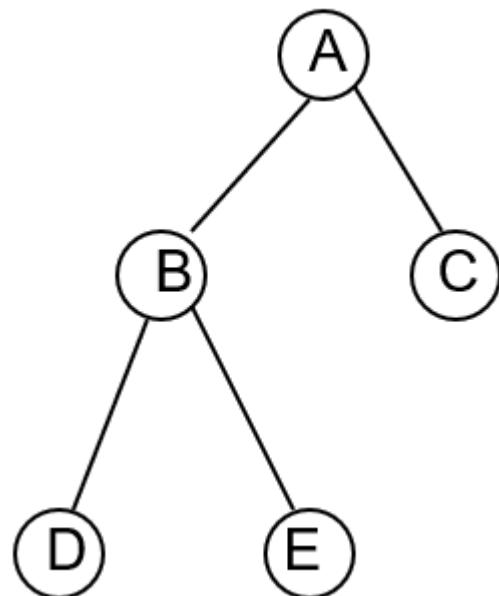


Polimorfismo 2

Questi assegnamenti sono tutti legali perché

- D extends B
- B extends A.

Quindi un oggetto di tipo D ha anche tipo B e A



```
A a; B b; D d;  
d = new D();  
b = new D();  
a = new D();  
a = b;
```



Polimorfismo 3

Sia A il tipo di x e B il tipo di expr . Allora, l'assegnamento

$X = \text{expr};$

è legale per il compilatore se:

- A uguale a B, oppure
- B sottoclasse (sottotipo) di A

Analogamente se x è un parametro formale di un metodo e expr è il parametro attuale (della chiamata).

Controllo statico.



Upcasting 1

```
A a; B b; D d;  
d = new D();
```

b = d; // d viene visto come se fosse un oggetto di tipo B

a = d; // d viene visto come se fosse un oggetto di tipo A

Upcasting: ci si muove da un tipo specifico ad uno più generico (da un tipo ad un suo “supertipo”).

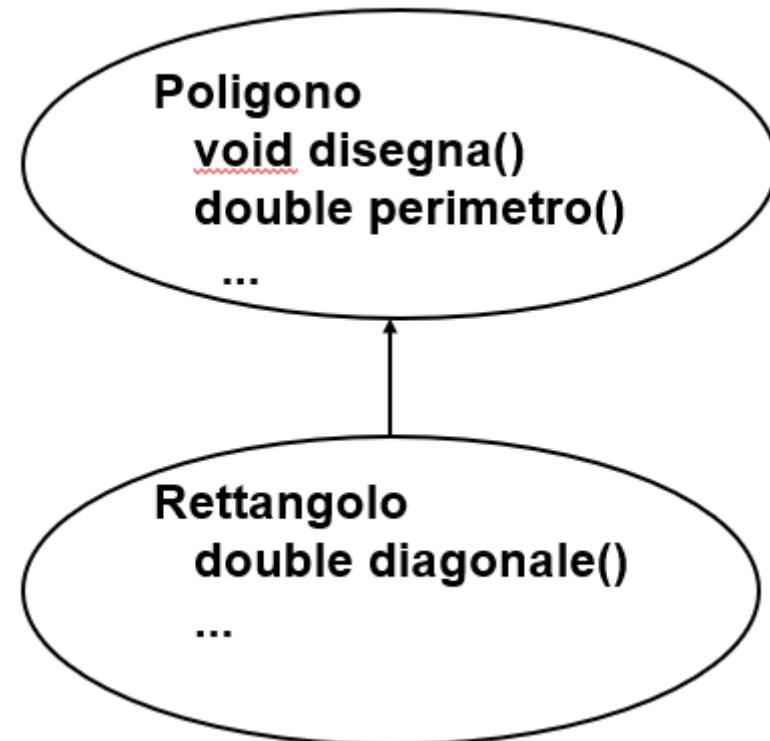
L'*upcasting* è sicuro per il type checking: dato che una sottoclasse eredita tutti i metodi delle sue sopraclasse, ogni messaggio che può essere inviato ad una sopraclasse può anche essere inviato alla sottoclasse senza il rischio di errori durante l'esecuzione.



Upcasting 2

Questo è corretto per il compilatore:

```
Poligono p;  
Rettangolo r;  
...  
p.disegna();  
r.disegna();  
p.perimetro();  
r.perimetro();
```

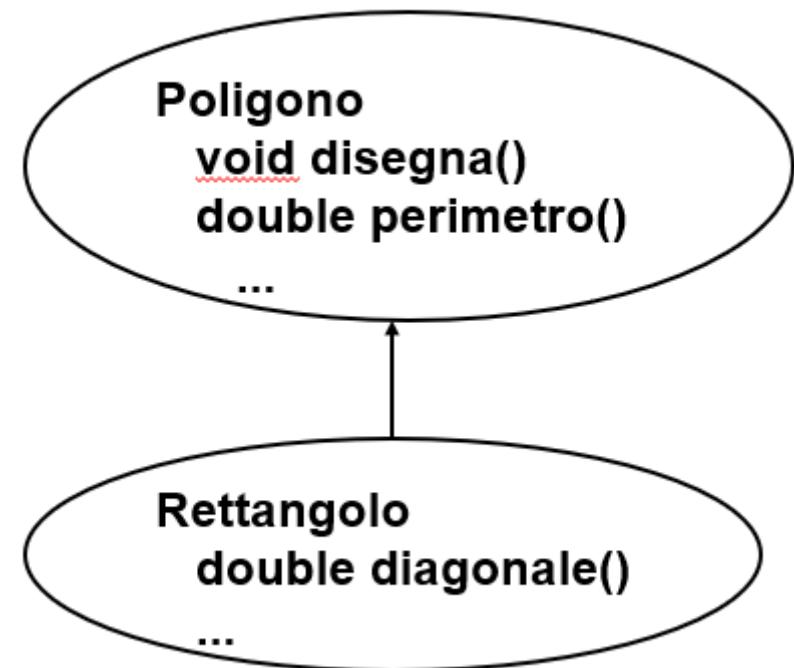




Upcasting 3

```
Poligono p;  
Rettangolo r;  
...  
r.diagonale();
```

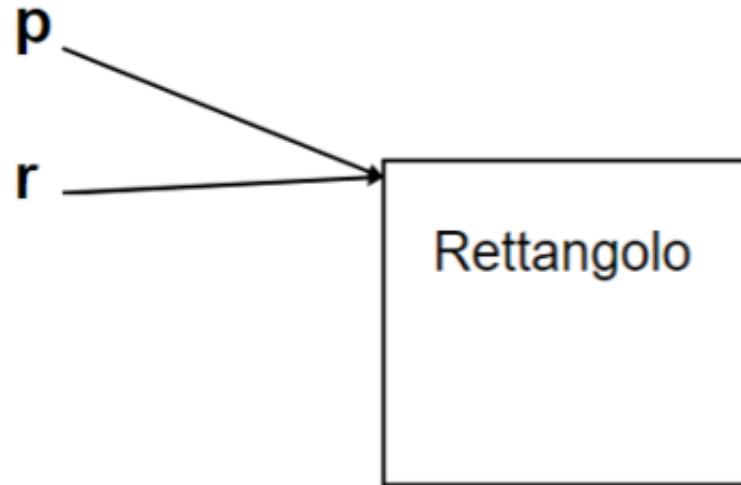
```
p.diagonale(); // errore di compilazione  
//p è di tipo Poligono e non  
// ha il metodo diagonale()
```





Upcasting 4

```
Poligono p;  
Rettangolo r;  
...  
p = r;  
r.diagonale();  
p.diagonale();
```

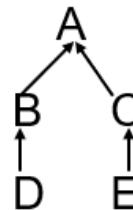


Il compilatore dà errore perché la variabile p ha tipo Poligono.

Tuttavia, se si facesse il controllo a runtime, **p.diagonale()** sarebbe corretto perché p è legata a un rettangolo, che possiede il metodo *diagonale()*.



Upcasting - esempio



```
class A { public String stampa() {return "sono un A";} }
class B extends A {public String stampa() {return "sono un B";} }
class C extends A {public String stampa() {return "sono un C";} }
class D extends B {public String stampa() {return "sono un D";} }
class E extends C {public String stampa() {return "sono un E";} }
```

```
public class UpcastingTest {
    public static void main(String[] args) {
        A a; B b;
        D d = new D(); System.out.println(d.stampa()); // stampa «sono un D»
        b = new D(); // oggetto di tipo D visto come se fosse di tipo B
        System.out.println(b.stampa()); // stampa «sono un D»
        a = new D(); System.out.println(a.stampa()); // idem per A
                                         // stampa «sono un D»
        a = b; // assegno a una variabile di tipo A un oggetto D (visto come A)
        System.out.println(a.stampa()); // stampa «sono un D»
        // b = a; errore: tipi incompatibili (cerco di assegnare un A ad un B)
    }
}
```



Overriding di metodi

Una sottoclasse può **ridefinire (OVERRIDING)** un metodo della sua sopraclassse. Esempio:

```
class Cella {  
    int contenuto=0;  
    int get() {return contenuto;}  
    void set(int n) {contenuto=n;}  
}
```

```
class CellaConMemoria extends Cella {  
    int backup=0;  
    void set(int n) {backup=contenuto;  
                    contenuto=n;}  
    void restore() {contenuto=backup;}  
}
```

Ereditarietà da Object (overriding di metodi)



```
class Complex {  
    double re,im;  
    ...  
}
```

```
Complex c = new Complex(1.5, 2.4);  
System.out.println(c); // c viene convertito in stringa con il metodo  
// toString() definito in Object.  
// Si ottiene (implementazione di Object):  
// Complex@.....
```

MA se si ridefinisce il metodo *toString()* Con la stampa si ottiene:

1.5 + i 2.4

```
class Complex {  
    ...  
    String toString() { return(re + " + i " + im); }  
}
```

Overriding: come estendere un metodo - I



Spesso un metodo di una sottoclasse definito per *overriding* non ridefinisce completamente il metodo con lo stesso nome della sua superclasse, ma lo estende soltanto. Esempio:

```
class Finestra {  
    Rettangolo r; ....  
    void disegnaCornice() {...}  
    void disegnaContenuto() {...}  
}  
  
class FinestraConTitolo extends Finestra {  
    String titolo;  
    void disegnaCornice() { ... disegna la cornice con il titolo ...}  
}
```

Il metodo *disegnaCornice()* di *FinestraConTitolo* estende il metodo *disegnaCornice()* di *Finestra* con il codice per disegnare il titolo.



Overriding: come estendere un metodo - II

Per non duplicare il codice, si può far riferimento ad un metodo della sopraclasse con lo stesso nome mediante la notazione **super**. Ridefiniamo il metodo `disegnaCornice()` in modo incrementale:

```
class FinestraConTitolo extends Finestra {  
    String titolo;  
    void disegnaCornice() {  
        super.disegnaCornice(); ←  
        ... codice aggiuntivo per disegnare il titolo...  
    }  
}
```

super.disegnaCornice() chiama il metodo **disegnaCornice()** della sopraclasse (che altrimenti non sarebbe visibile).



super

```
class FinestraConTitolo extends Finestra {  
    String titolo;  
    void disegnaCornice() {  
        super.disegnaCornice();  
        ... codice aggiuntivo per disegnare il titolo ...  
    }  
}
```

Cosa accadrebbe se non ci fosse **super**?



Visibilità dei membri delle classi

Ad ogni membro di una classe (variabile, metodo, ...) può essere associata uno *specificatore di accesso*:

- **private:** visibile solo dalla *classe stessa*
- *nessuna:* visibile da tutte le classi nello *stesso package*
- **protected:** visibile da tutte le classi nello *stesso package* e da tutte le *sottoclassi* (v. avanti)
- **public:** visibile da *tutti* (parte dell'interfaccia dell'oggetto)



Visibilità dei membri delle classi

Una sottoclasse non può accedere ai campi *privati* della sua superclasse

```
class Cella {  
    private int contenuto=0;  
    public int get() {return contenuto;}  
    public void set(int n) {contenuto=n;}  
}
```

```
class CellaConMemoria extends Cella {  
    private int backup=0;  
    public void set(int n) {backup=contenuto; // ERRORE  
                           contenuto=n;}  
    public void restore() {contenuto=backup;}  
}
```



Cella (2)

```
class Cella {  
    private int contenuto=0;  
    public int get() {return contenuto;}  
    public void set(int n) {contenuto=n;}  
}
```

```
class CellaConMemoria extends Cella {  
    private int backup=0;  
    public void set(int n) { backup = get();  
                           super.set(n); } ←  
    public void restore() { super.set(backup); }  
}
```

OK: Si accede al campo *contenuto* con i metodi *get()* e *set()*.



Super può essere usato per chiamare il costruttore della sopraclasse.

```
class Manager extends Employee {  
  
    public Manager(String n, double s, Day d) {  
        super(n,s,d);  
        secretaryName = "";  
    }  
    .....  
}
```

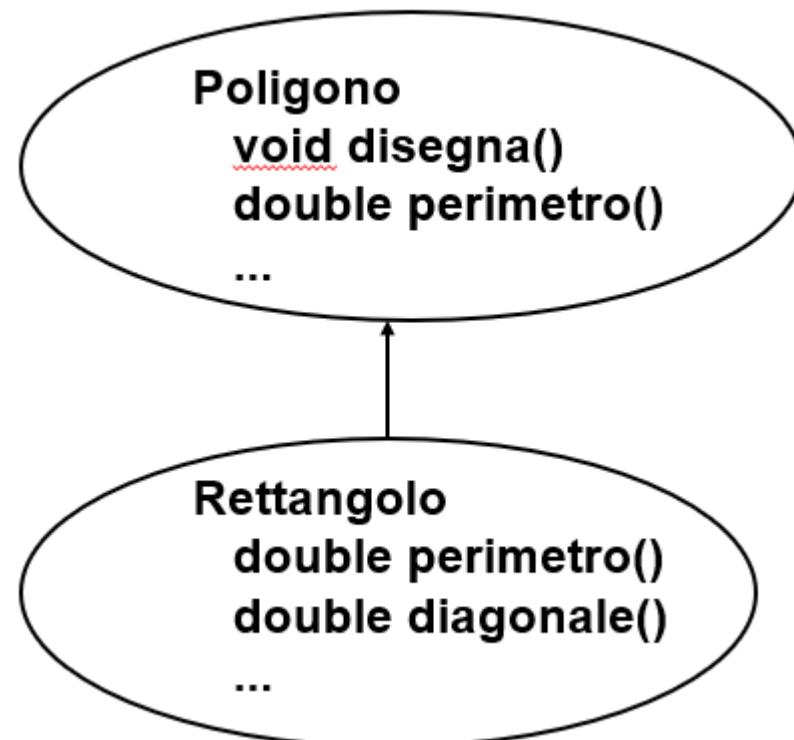


Upcasting 5

Questo è corretto per il compilatore, ma **quale metodo si esegue?**
Quello di **Poligono** o quello di **Rettangolo**?

```
Poligono p;  
Rettangolo r;  
...  
p = r;  
p.disegna();
```

```
p.perimetro();
```





Ringraziamenti

Grazie al Prof. Emerito Alberto Martelli del Dipartimento di Informatica dell'Università di Torino per aver redatto la prima versione di queste slides.



Programmazione III

Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

JAVA – Ereditarietà – parte 2



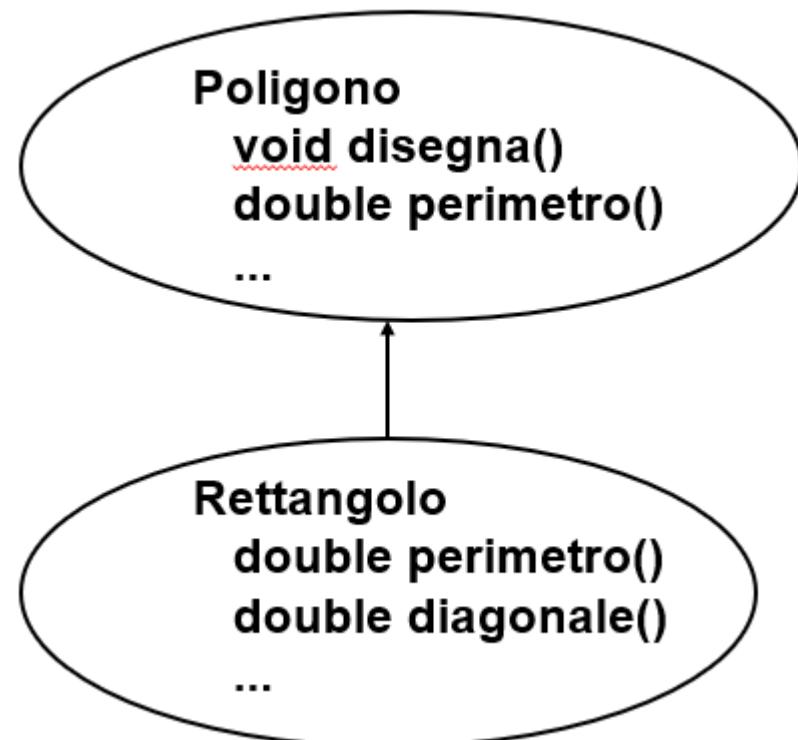
Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND



Upcasting 5

Questo codice è corretto per il compilatore, ma **quale metodo si esegue?**
Quello di **Poligono** o quello di **Rettangolo**?

```
Poligono p;  
Rettangolo r;  
...  
p = r;  
p.disegna();  
  
p.perimetro();
```





Binding dinamico – I

```
p.perimetro();
```

Si esegue il metodo *perimetro* dell'oggetto a cui *p* fa riferimento in quel momento.

```
Poligono p = new Poligono();  
Rettangolo r = new Rettangolo();
```

```
p.perimetro(); // si esegue il metodo perimetro() di Poligono
```

```
p = r;
```

```
p.perimetro(); // si esegue il metodo perimetro() di Rettangolo
```



Binding dinamico - II

In questo contesto:

Binding: legame fra il nome di un metodo in una invocazione e (codice del) metodo.

obj.m(): quale metodo **m** viene eseguito?

Nei linguaggi tradizionali le chiamate di procedura vengono risolte dal compilatore.

Nei linguaggi ad oggetti (tranne il C++) le chiamate di metodi sono risolte dinamicamente.

BINDING DINAMICO: la forma di un oggetto determina dinamicamente quale versione di un metodo applicare.



Binding dinamico - III

```
class Finestra {  
    Rettangolo r; ....  
    void disegnaCornice() {...}  
    void disegnaContenuto() {...}  
    void rinfresca() {  
        disegnaCornice(); disegnaContenuto();  
    }  
}  
  
class FinestraConTitolo extends Finestra {  
    String titolo;  
    void disegnaCornice() { ... disegna la cornice con il titolo ...}  
}
```

Nel main:

```
FinestraConTitolo ft;  
...  
ft.rinfresca(); ← chiama il metodo disegnaCornice() di FinestraConTitolo
```

Esempio - I



class A

```
{    void stampaTipo() {System.out.println("A");}  
    void stampa() {stampaTipo();}  
}
```

class B extends A

```
{    void stampaTipo() {System.out.println("B");}  
}
```

class EredMetodi {

```
    public static void main(String[] args)  
    {        B b = new B();  
            b.stampa(); // stampa "B"  
    }  
}
```

Esempio - II



Per le variabili di istanza non c'è il *binding dinamico* (esiste solo per i metodi)

```
class A
{
    String tipo = "A";
    void stampa() {System.out.println(tipo);}
}
// stampa() prende il valore di "tipo" locale (in A)
```

```
class B extends A
{
    String tipo = "B";
}
```

```
class EredVar {
    public static void main(String[] args)
    {
        B b = new B();
        System.out.println(b.tipo);          // stampa "B"
        b.stampa();                      // stampa "A"
    }
}
```



Esempio - III

```
public class Test {  
    public static void main(String[] args) {  
        B b = new B();  
        System.out.println(b.tipo);  
        b.stampaTipo();  
    }  
}
```

```
class A {  
    String tipo = "A";  
    String getTipo() {return tipo;}  
    void stampaTipo () {System.out.println(getTipo());} // STAMPA B B  
    //void stampaTipo() {System.out.println(tipo);} // STAMPA B A  
}
```

```
class B extends A {  
    String tipo = "B";  
    String getTipo() {return tipo;} // getTipo() prende il valore di "tipo" locale (in B)  
}
```

Esempio - IV



```
public class Test {  
    public static void main(String[] args) {  
        B b = new B();  
        System.out.println(b.tipo);  
        b.stampaTipo();  
    }  
}  
  
class A {  
    String tipo = "A";  
    String getTipo() {return tipo;}  
    //void stampaTipo () {System.out.println(getTipo());} // STAMPA B B  
    void stampaTipo() {System.out.println(tipo);}           // STAMPA B A  
    // stampaTipo() prende il valore di "tipo" locale (in A)  
}  
  
class B extends A {  
    String tipo = "B";  
    //String getTipo() {return tipo;}  
}
```



Classi "generiche"

Consideriamo il caso in cui non si dispone di classi parametriche (vedremo più avanti i tipi generici).

Come definire uno Stack «generico»?

```
class Stack {
    ...
    void push(Object x) {...};
    Object pop() {...};
}
```

Non si può specificare il tipo degli elementi della pila: possono essere oggetti qualunque.

```
Stack s; Linea l; Rettangolo r;
...
s.push(l); s.push(r);
```



Downcasting 1

```
class Stack {  
    ...  
    void push(Object x) {...};  
    Object pop() {...};  
}
```

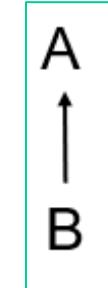
Supponiamo di sapere che sulla pila vengono messi solo rettangoli. Come possiamo utilizzare gli oggetti estratti dalla pila?

```
Stack s; Rettangolo r;  
...  
s.push(new Rettangolo());
```

r = s.pop(); ← Errore di compilazione. La pop() restituisce un *Object*, che è più generale di *Rettangolo*



Downcasting 2



Downcasting: ci si muove da un tipo più generale ad uno più specifico (da un tipo ad un sottotipo)

Se B è un sottotipo di A e se *espr* ha tipo A,

(B) *espr* ha tipo B

L'assegnamento

$B\ x = (B)\ espr$ è corretto per il compilatore

(B) *espr* può dare errore a *run time*, se l'oggetto ottenuto valutando *espr* non ha tipo B.



Downcasting 3

```
class Stack {  
    ...  
    void push(Object x) {...};  
    Object pop() {...};  
}  
  
...  
Stack s; Rettangolo r;  
  
...  
s.push(new Rettangolo());
```

r = (Rettangolo) s.pop(); // Accettato dal compilatore.

Controllo a run-time. Quando si esegue questa istruzione si controlla che l'oggetto restituito da *pop()* sia veramente un Rettangolo.



Riuso del software

La programmazione ad oggetti consente di utilizzare classi già esistenti per produrre nuovo software:

Uso

Un oggetto comunica con oggetti di altre classi

Contenimento (Part-of).

Si definiscono nuove classi i cui oggetti sono composti di oggetti di classi già esistenti.

Ereditarietà (Is-a).

Favorisce lo sviluppo incrementale, estendendo classi già esistenti.



Contenimento (Part-of)

```
class Automobile {  
    int lunghezza;  
    Motore motore;  
    Ruota[] ruote;  
    ... }
```

```
class motore {  
    int cilindrata;  
    ... }
```

```
class Ruota {  
    double pressione;  
    int diametro;  
    ... }
```

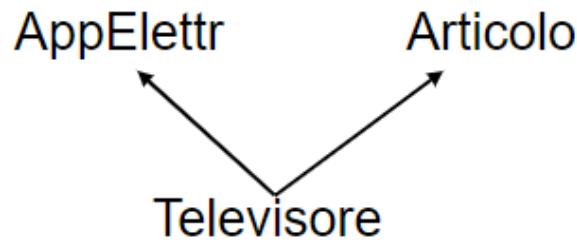
```
Automobile miaAuto = new Automobile();
```

...

```
miaAuto.motore.cilindrata;  
miaAuto.ruote[1].pressione;
```

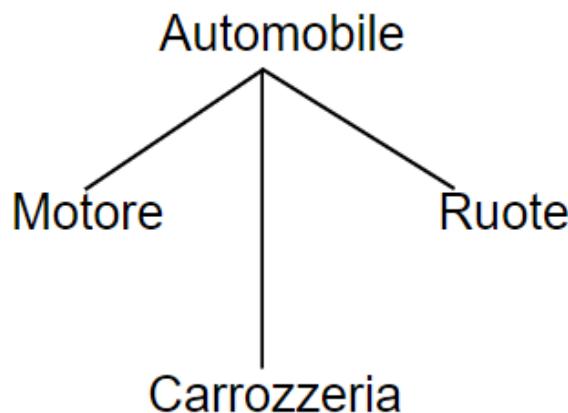


Ereditarietà vs contenimento



Televi~~sore~~ **è un** apparecchio elettrico e **è un** articolo.

La sottoclasse eredita il comportamento di altre classi.



L'automobile **ha un** motore, una carrozzeria, ...

Il Motore non eredita il comportamento dell'Automobile, ne fa parte (come componente)



Programmare con l'ereditarietà - I

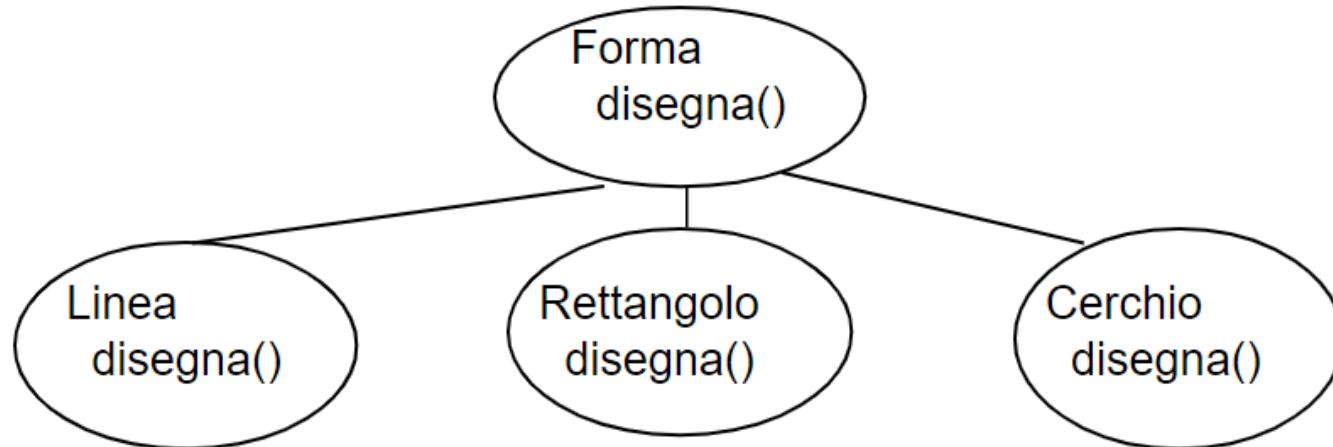
Si consideri una figura composta di diverse forme geometriche (linee, rettangoli, cerchi, ...).

Nella programmazione tradizionale, una procedura per disegnare la figura dovrebbe considerare tutti i casi:

```
void disegna(figura f) {  
    for ogni forma S in f  
        switch(S.genere)  
            case LINEA: disegnaLinea()  
            case RETTANGOLO: disegnaRettangolo()  
            case CERCHIO: disegnaCerchio()
```

Se si aggiunge la forma TRIANGOLO occorre modificare la procedura *disegna()*.

Programmare con l'ereditarietà – II



Con l'ereditarietà (binding dinamico) invece possiamo definire le classi che realizzano le diverse forme geometriche come sottoclassi di una sola classe **Forma**.

Programmare con l'ereditarietà – II



```
class Forma {  
    void disegna() {....}}
```

```
class Linea extends Forma {  
    void disegna() {....}}
```

```
class Rettangolo extends Forma {  
    void disegna() {....}}
```

```
class Cerchio extends Forma {  
    void disegna() {....}}
```

Ciascuna sottoclasse fa overriding del metodo disegna().

Programmare con l'ereditarietà – IV



```
class Figura {  
    Forma[] s; //s è una figura è implementata come un array di Forme  
  
    void disegnaFigura() {  
        for (int i=0; i<s.length; i++)  
            s[i].disegna();  
    }  
    ...
```

L'istruzione **s[i].disegna()** è corretta per il compilatore perché la classe **Forma** possiede il metodo **disegna()**, che è ridefinito da ogni sottoclasse.

Grazie al *binding dinamico*, quando si esegue il **for** verrà sempre eseguito il metodo **disegna()** della specifica forma geometrica.



Programmare con l'ereditarietà - V

Se si aggiunge la forma *Triangolo*, è sufficiente definire una nuova sottoclasse di *Forma* con il metodo *disegna()*:

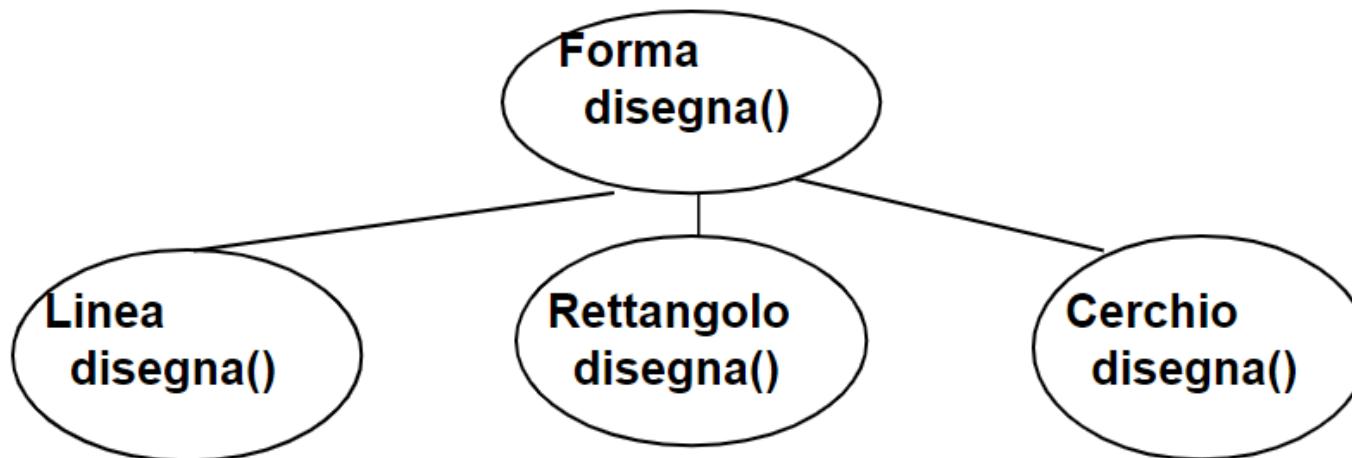
```
class Triangolo extends Forma {  
    ...  
    void disegna(){...} }
```

La classe *Figura* non viene modificata.

Il binding dinamico fa sì che, se la figura contiene un triangolo, venga chiamato il metodo per disegnare un triangolo.



Classi astratte - I



In un programma non ci si aspetta di chiamare il metodo *disegna()* della classe *Forma*, né di usare oggetti di questa classe.

La classe *Forma* serve solo per avere una **interfaccia comune**, contenente il metodo *disegna()*, per le varie forme specifiche. Quindi potremmo **NON** implementare il metodo *disegna* di *Forma*... di qui le classi astratte.



Classi astratte - II

Le classi astratte sono classi che includono almeno un metodo non implementato: si lascia l'implementazione alle sottoclassi che verranno definite.

I metodi non implementati sono dichiarati **abstract**.

Le classi astratte possono avere il costruttore per poter inizializzare i membri privati della classe.

Tuttavia, NON possono essere istanziate perché non hanno la definizione completa che serve per creare gli oggetti
→ la new non è permessa.



Classi astratte – esempio - I

```
abstract class Forma {  
    abstract void disegna(); ← Non c'è l'implementazione del metodo
```

```
}
```

```
class Linea extends Forma {  
    void disegna() { ..... }  
}
```

....

Forma f = new Forma(); ← Errore di compilazione. Non si possono
creare oggetti di una classe astratta.

Forma f = new Linea();

f.disegna();



Classi astratte – esempio - II

```
abstract class Forma {  
    private String nome;  
    public Forma(String nome) {  
        this.nome = nome;  
    }  
    public String getNome() {  
        return nome;  
    }  
    abstract public void disegna();  
}  
  
class Rettangolo extends Forma {  
    public Rettangolo() {  
        super("Rettangolo");  
    }  
    public void disegna() {  
        System.out.println("Disegno un rettangolo");  
    }  
}
```

Classi astratte – esempio - III

```
class Linea extends Forma {  
    public Linea() {  
        super("Linea");  
    }  
    public void disegna() {  
        System.out.println("Disegno una linea");  
    }  
}  
public class FormaApp {  
    public static void main(String[] args) {  
        Rettangolo r = new Rettangolo();  
        Linea l = new Linea();  
        r.disegna(); // stampa «Disegno un rettangolo»  
        l.disegna(); // stampa «Disegno una linea»  
    }  
}
```



Interface e classi astratte

Le **classi astratte** possono essere miste, ossia possono contenere anche metodi non astratti.

Con l'ereditarietà singola di Java, una classe può essere sottoclassificata solo di una classe astratta → **non si possono ereditare metodi da più sopraclassi**. Ma talvolta questo serve!

Le **interface** non sono soggette al vincolo della struttura gerarchica ad albero.

Una classe può implementare più di una interfaccia (qualche analogia con *ereditarietà multipla*).

Interface e Classi di libreria – esempio - I



Supponiamo di voler ordinare liste di oggetti di tipo Rettangolo.

Le librerie di Java forniscono l'interface **Comparable** che specifica la firma del metodo `compareTo()` per comparare l'oggetto su cui viene invocato con un oggetto passato per parametro.

`compareTo` restituisce un intero che indica quale dei due oggetti viene prima dell'altro secondo il criterio di ordinamento da applicare.

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

Le librerie di Java forniscono anche la classe **Arrays** che contiene un metodo `sort()` per ordinare un array di oggetti **Comparable**.

Interface e Classi di libreria – Esempio - II



Per confrontare due rettangoli, implementare l'interface:

```
class Rettangolo extends Poligono implements Comparable {  
    int altezza, larghezza;  
    ....  
    public int compareTo(Object ob) {  
        Rettangolo r = (Rettangolo)ob;  
        return altezza - r.altezza;  
    }  
}
```

e nel main dell'applicazione:

```
Rettangolo[] a = new Rettangolo[10];  
// ... Inizializzazione dell'array di rettangoli ...  
Arrays.sort(a);  
...
```

Interfacce multiple



```
interface A {  
    void metodA();  
}
```

```
interface B {  
    void metodB();  
}
```

```
class C implements A,B {  
    void metodA() {System.out.println("sono A");}  
    void metodB() {System.out.println("sono B");}  
  
    public static void main(String[] args) {  
        A a = new C();          //l'oggetto è visto come un A  
        B b = new C();          //l'oggetto è visto come un B  
        a.metodA();              // stampa A  
        b.metodB();              // stampa B  
        //a.metodB();            //errore: A non offre metodB()  
        ((C)a).metodoB(); // ma con downcast compilazione OK  
                           // stampa B  
    }  
}
```



Ringraziamenti

Grazie al Prof. Emerito Alberto Martelli del Dipartimento di Informatica dell'Università di Torino per aver redatto la prima versione di queste slides.



Programmazione III

Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

JAVA – Ereditarietà – parte 3



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND



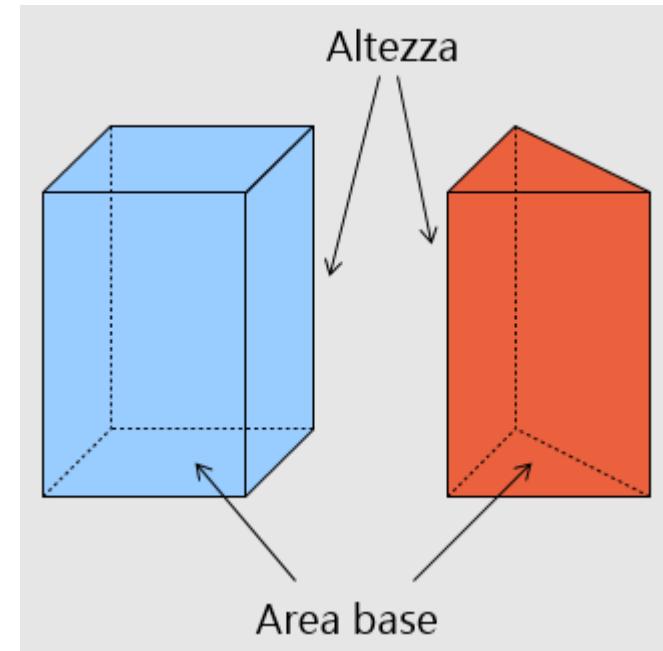
Verso i pattern architetturali: esempio - I

Prisma:

- Base, Altezza
- $\text{Volume} = \text{areaBase} * \text{altezza}$

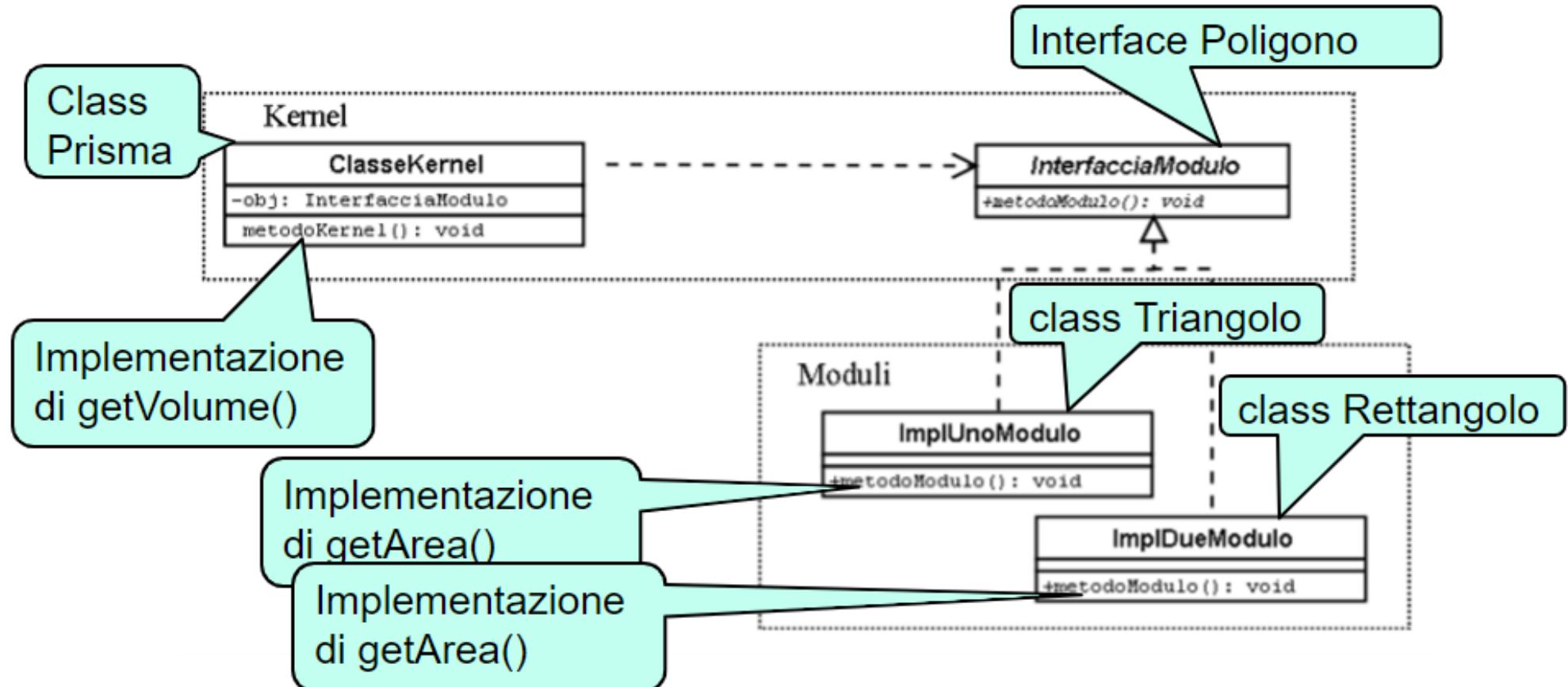
Io posso specificare Prisma astraendo dal poligono che viene usato come base:

- In Prisma, la base è un Polygon
- L'applicazione che crea un oggetto Prisma innesta il particolare poligono da usare (per esempio, un Triangle)
- Il poligono sa come calcolare la propria base





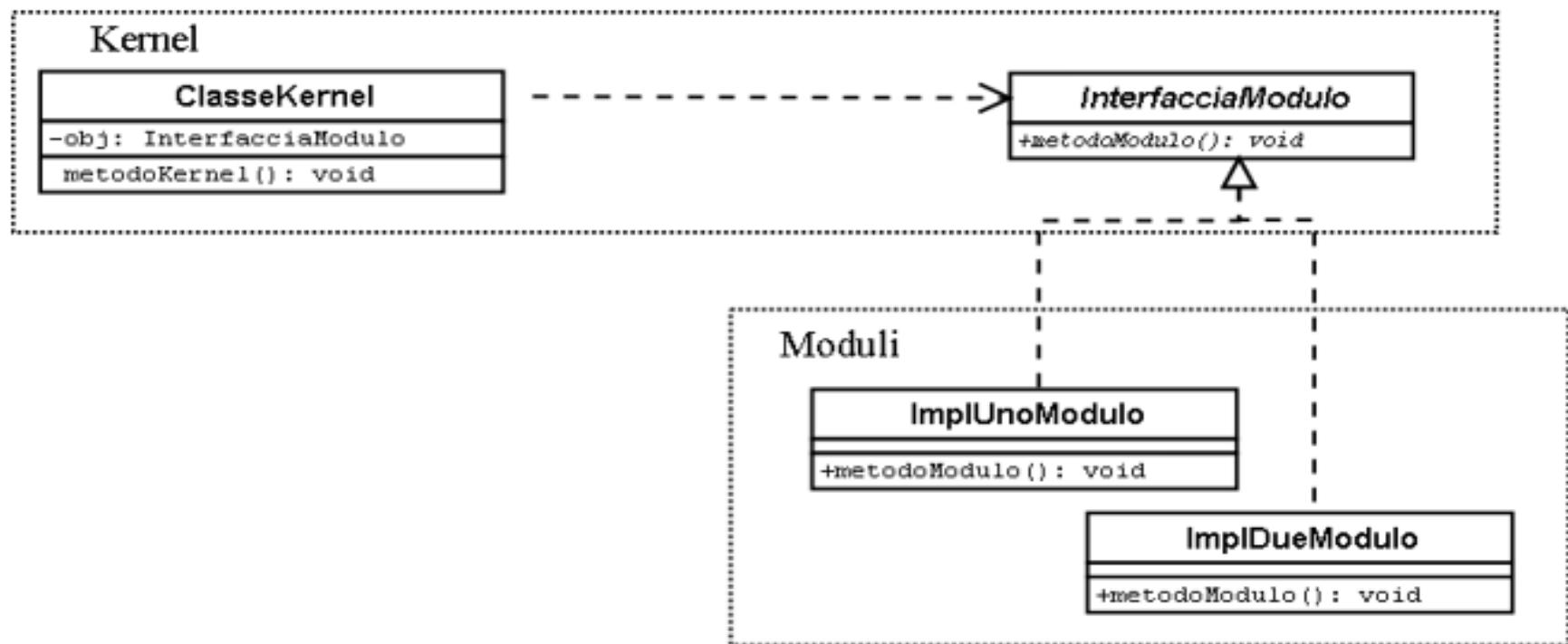
Verso i pattern architetturali: esempio - II





Verso i pattern architetturali - I

Combinando l'uso di interface/classi che implementano le interface con le relazioni di contenimento si ottengono modelli di programmazione avanzati. In particolare:



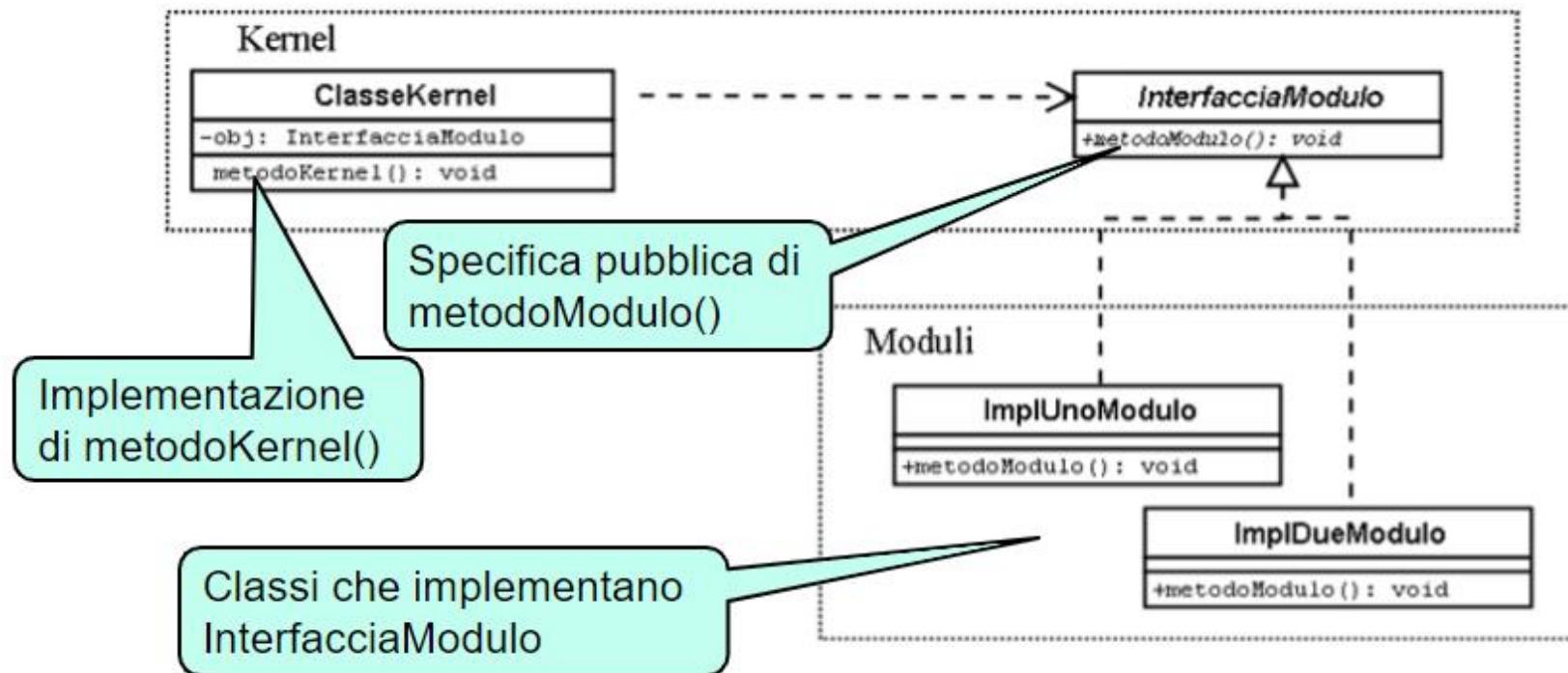
Questa imagine è tratta dai lucidi di presentazione di Programmazione III del Prof. Matteo Baldoni, Università di Torino

L. Ardissono - Ereditarietà



Verso i pattern architetturali - II

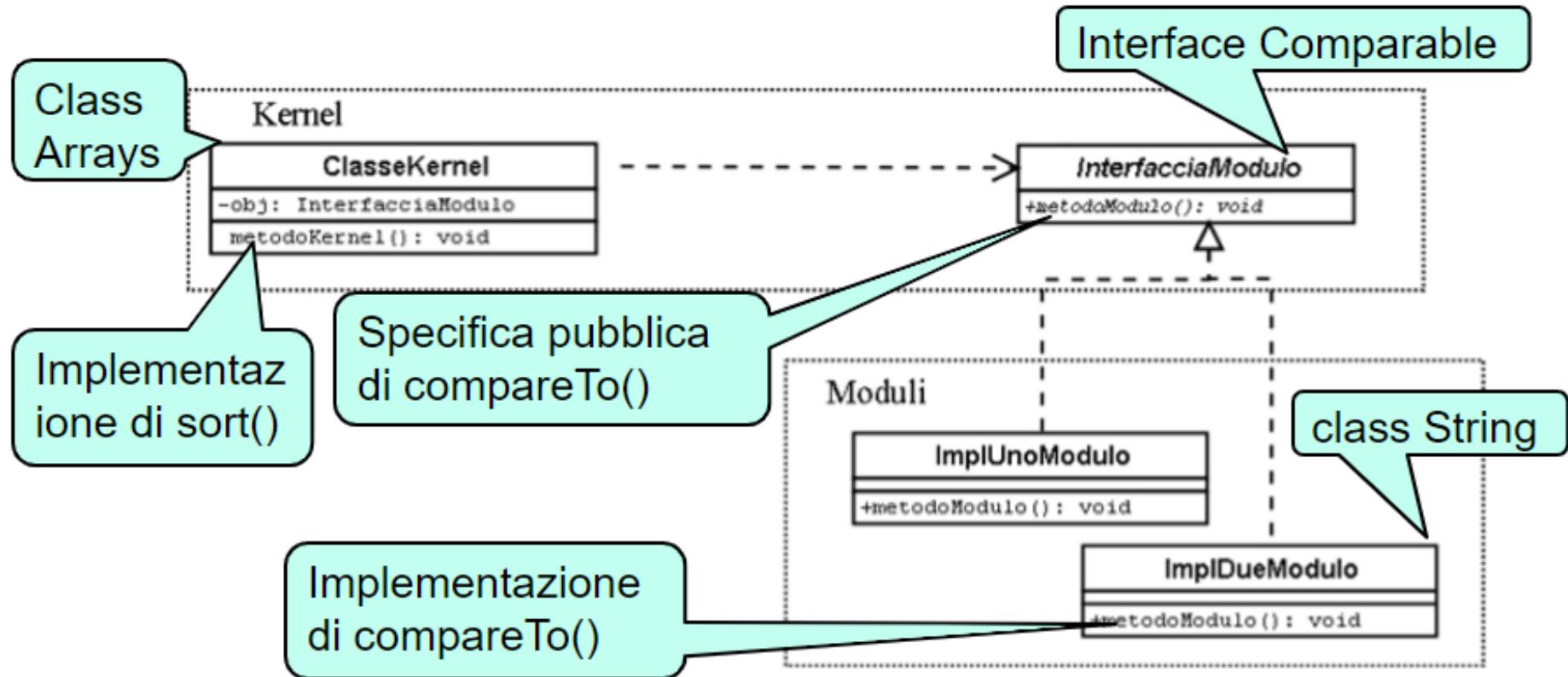
La classe ClasseKernel **usa** un componente obj di tipo InterfacciaModulo (in upcasting). Nel body del metodo metodoKernel(), la classe invoca metodoModulo() su obj per operare





Verso i pattern architetturali - III

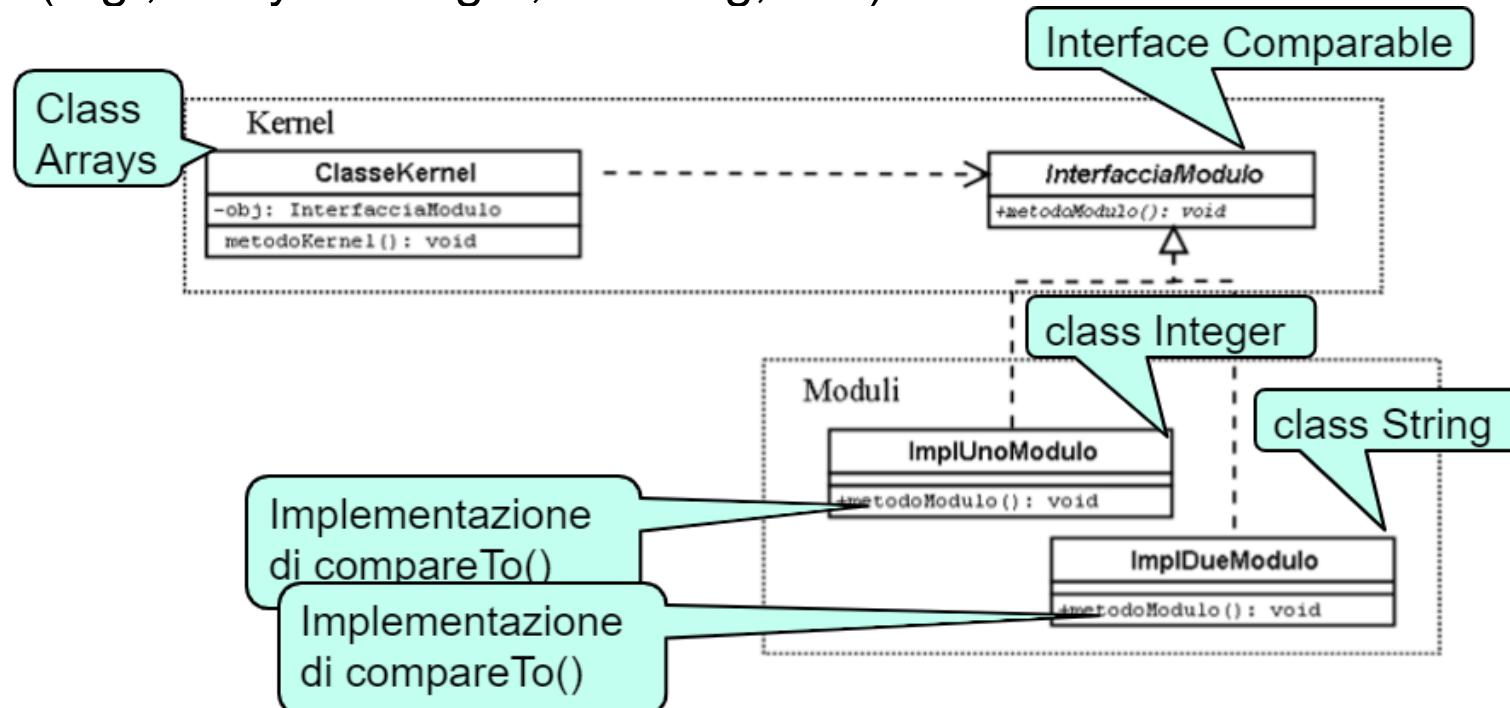
Esempio: l'applicazione usa la classe Arrays per ordinare un array di elementi
String[] myArray = new String[3]; //... inserimento di valori in myArray...
Arrays.sort(myArray);





Verso i pattern architetturali - IV

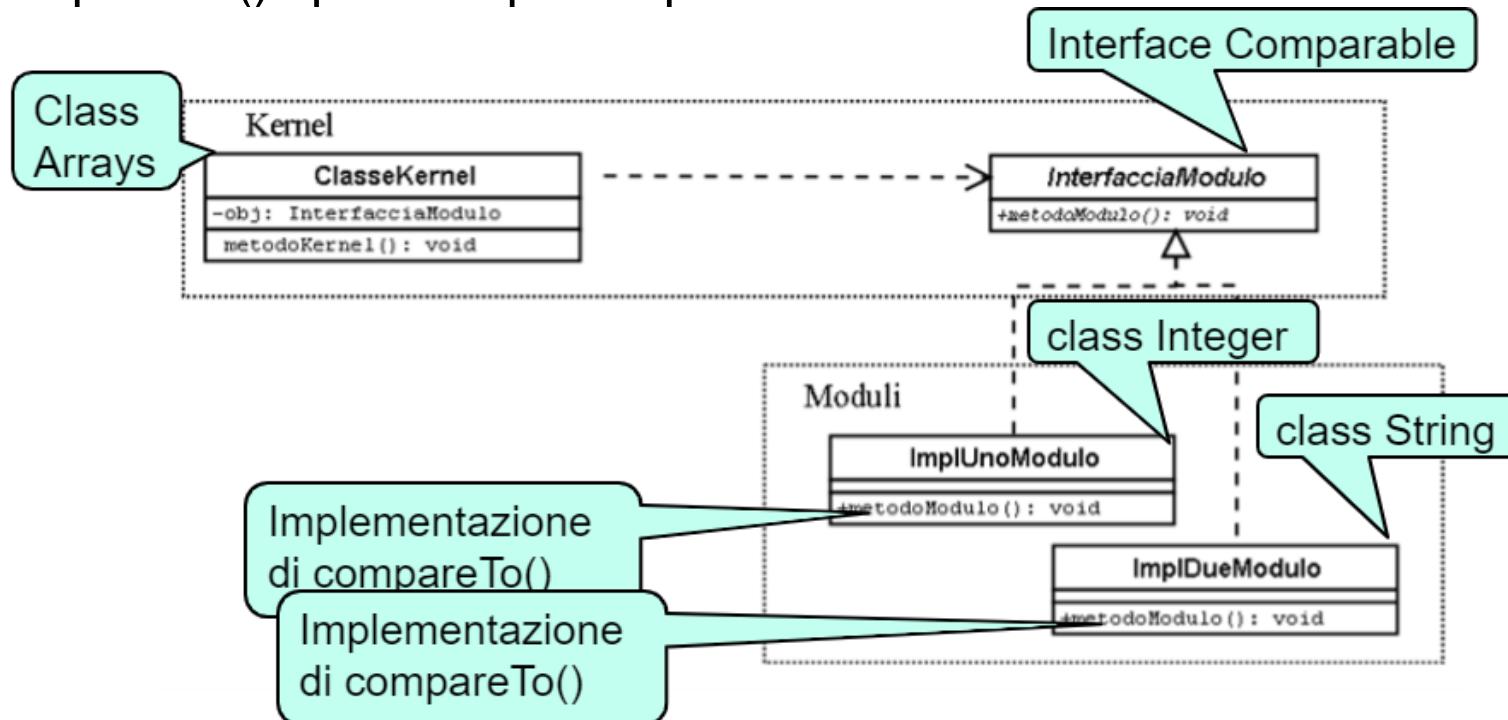
In una specifica applicazione, l'InterfacciaModulo (Comparable) viene implementata da una classe (String nel nostro esempio) che implementa il metodo compareTo(). L'applicazione potrebbe usare implementazioni differenti (e.g., array di Integer, di String, etc.)





Verso i pattern architetturali - V

Grazie al polimorfismo e al binding dinamico, a seconda di quale tipo di oggetti viene messo nell'array (Integer, String, ...), quando viene invocato Arrays.sort(myArray) verrà eseguita l'implementazione di compareTo() specifica per il tipo di dato

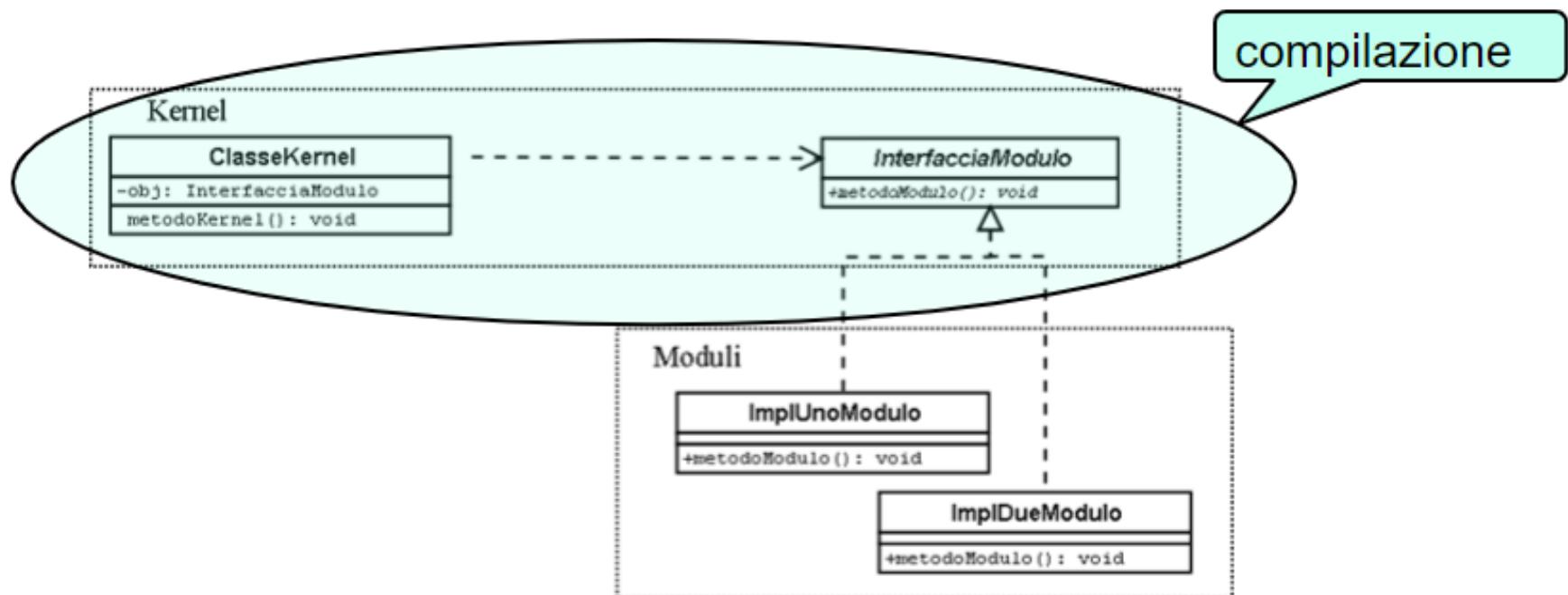




Verso i pattern architetturali - VI

Questa separazione tra interfacce e implementazioni permette la compilazione separata del codice:

- **ClasseKernel + InterfacciaModulo può essere compilata (come libreria) anche se non si specificano le implementazioni di InterfacciaModulo in una applicazione specifica**

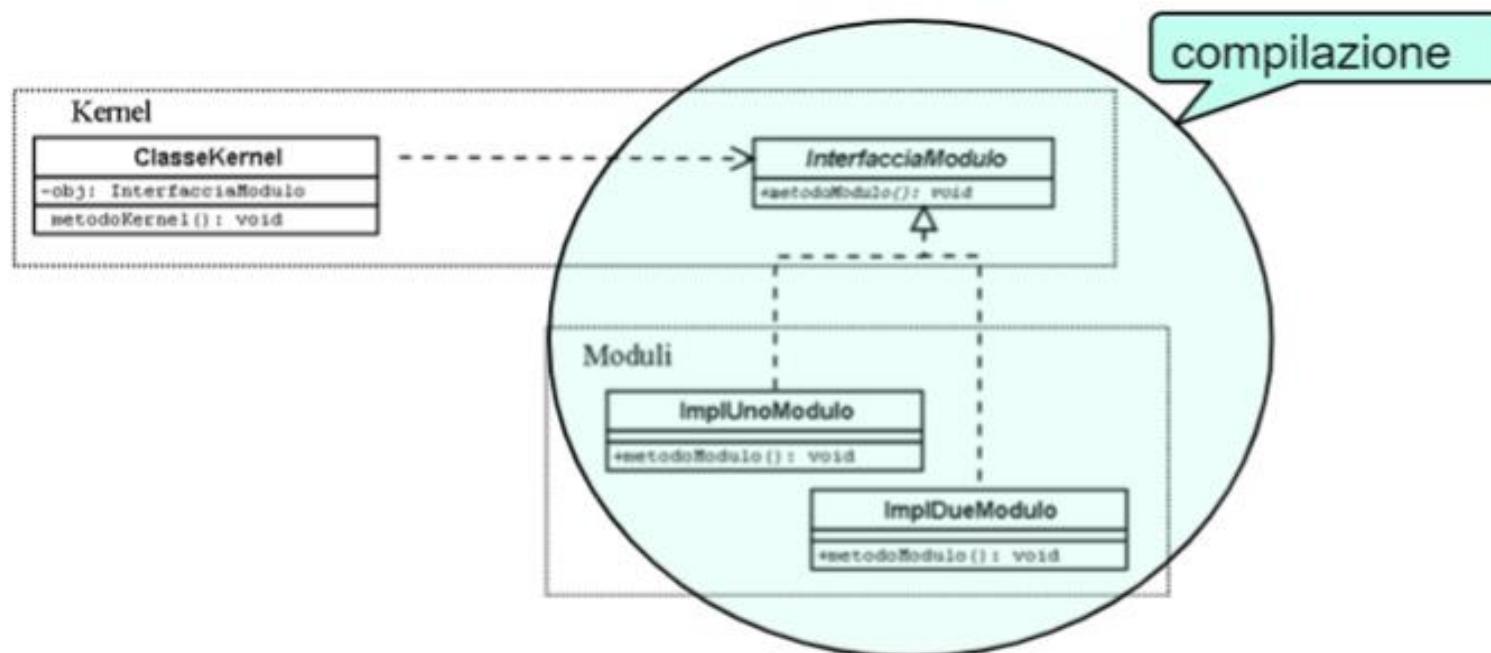




Verso i pattern architetturali - VII

Questa separazione tra interfacce e implementazioni permette la compilazione separata del codice:

- **ImplUnoModulo e ImplDueModulo possono essere compilate, in presenza di InterfacciaModulo, indipendentemente da ClasseKernel**



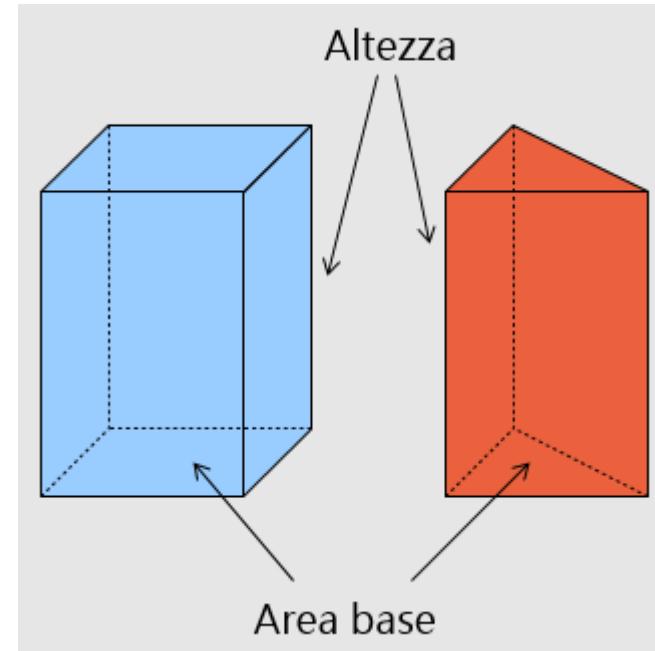
L. Ardissono - Ereditarietà



Esempio - I

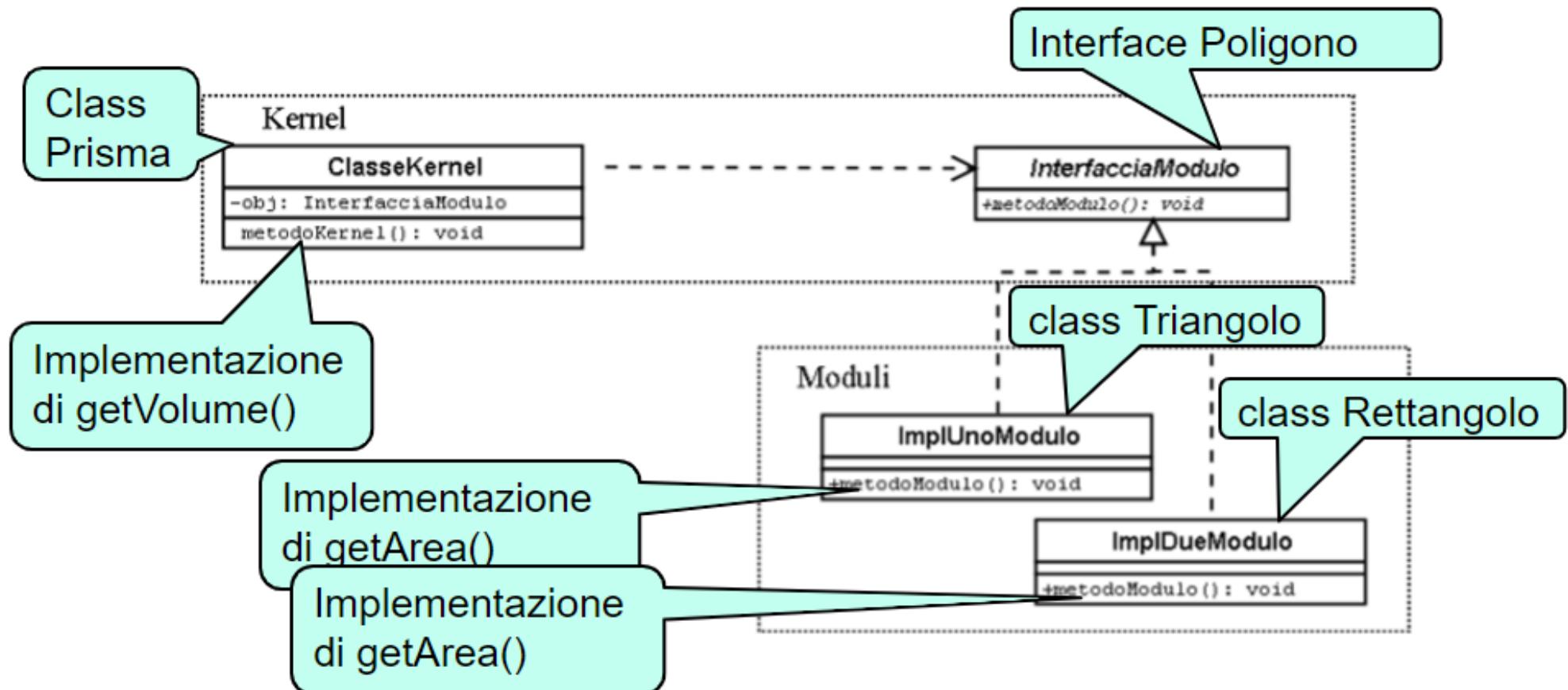
Definiamo i prismi come oggetti complessi composti da una base che è un poligono:

- **ClasseKernel: Prisma**
- **InterfacciaModulo: Poligono**
- **ImplementazioneModulo: Triangolo, Rettangolo**





Esempio - II





Esempio - III

```
interface Poligono {  
    public double getArea();  
}  
  
class Prisma {  
    private Poligono base;  
    private double altezza;  
  
    public Prisma(Poligono base, double altezza) {  
        this.base = base;  
        this.altezza = altezza;  
    }  
  
    public double getVolume() {  
        return base.getArea()*altezza;  
    }  
}
```



Esempio - IV

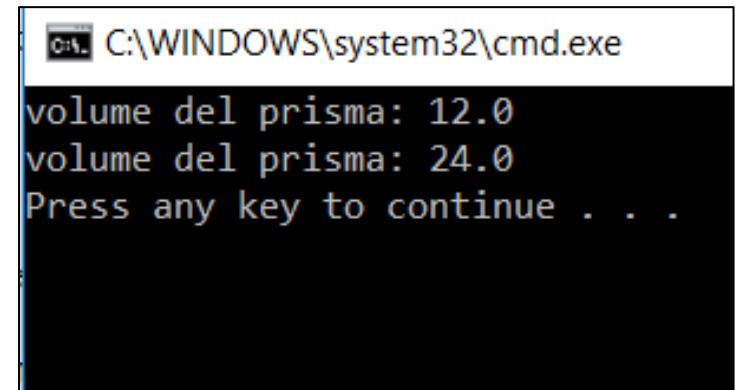
```
class Triangolo implements Poligono {  
    private double base;  
    private double altezza;  
    public Triangolo(double base, double altezza) {  
        this.base = base;  
        this.altezza = altezza;  
    }  
    public double getArea() { return base * altezza / 2;}  
}
```

```
class Rettangolo implements Poligono {  
    private double base;  
    private double altezza;  
    public Rettangolo(double base, double altezza) {  
        this.base = base;  
        this.altezza = altezza;  
    }  
    public double getArea() {return base * altezza;}  
}
```



Esempio - V

```
public class EreditarietaAdvanced {  
    public static void main(String args[]) {  
  
        Prisma p1 = new Prisma(new Triangolo(2, 3), 4);  
        System.out.println("volume del prisma: " + p1.getVolume());  
  
        Prisma p2 = new Prisma(new Rettangolo(2, 3), 4);  
        System.out.println("volume del prisma: " + p2.getVolume());  
    }  
}
```



C:\WINDOWS\system32\cmd.exe
volume del prisma: 12.0
volume del prisma: 24.0
Press any key to continue . . .



Programmazione III

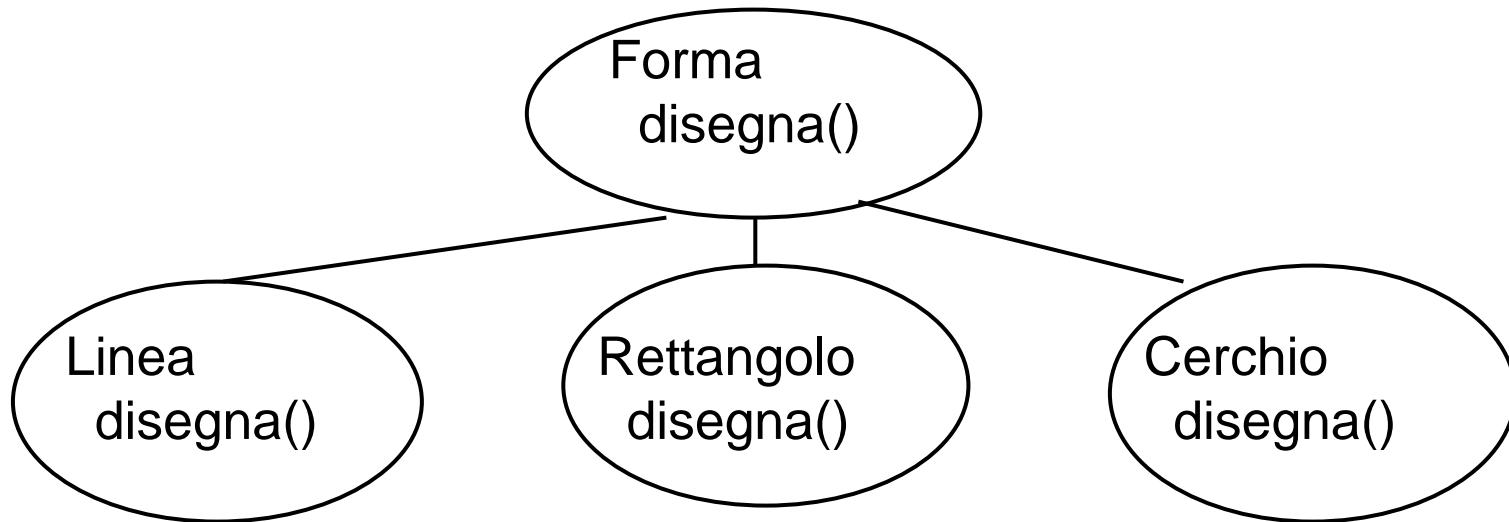
Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

**RunTime Type Identification (RTTI) e
Java Reflection
Controllo di tipo per downcast**



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND

Programmare con l'ereditarietà - I



```
interface Forma {  
    void disegna();  
}  
class Linea implements Forma {  
    void disegna() { ..... }  
}
```

...

```
Forma f = new Linea();  
f.disegna();
```

Programmare con l'ereditarietà - II



Si vuole eseguire un'operazione **op** particolare su oggetti di tipo Cerchio → downcast da Forma a Cerchio

Forma f;

...

(Cerchio)f.op()

Se f non è un cerchio, viene sollevata una eccezione a run-time.
Per prevenirla si può verificare il tipo di un oggetto a run-time:

Forma f;

...

if (f instanceof Cerchio) (Cerchio)f.op()

La notazione **instanceof** è statica. Deve essere specificato il nome del tipo (Cerchio, Triangolo, ecc.). Non abusare.



La classe **Class** (metaclasse)

In Java esiste la classe **Class**. Per ogni classe **C** usata in un programma, c'è un unico oggetto a run-time di tipo **Class** che rappresenta quella classe.

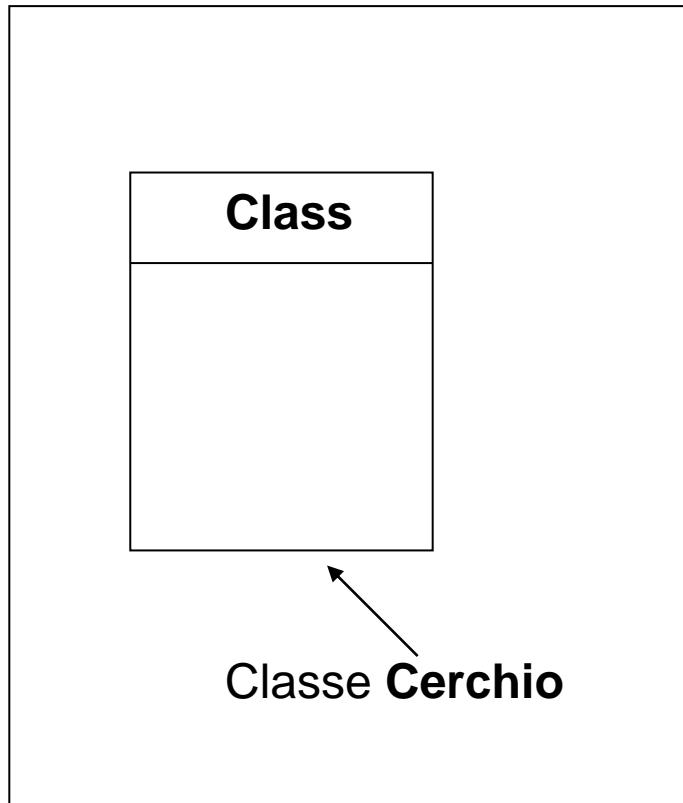
Esiste un oggetto Class per ogni tipo di dato: classi, tipi enumerativi, interfacce, annotazioni, array e tipi primitivi. L'oggetto Class serve per analizzare la classe (nome, membri, ...).

Quando un programma è in esecuzione, il sistema runtime di Java conserva la **RunTime Type Identification (RTTI)** di ogni oggetto. Per ogni oggetto o si mantiene il riferimento all'oggetto **Class** che rappresenta la classe di o.

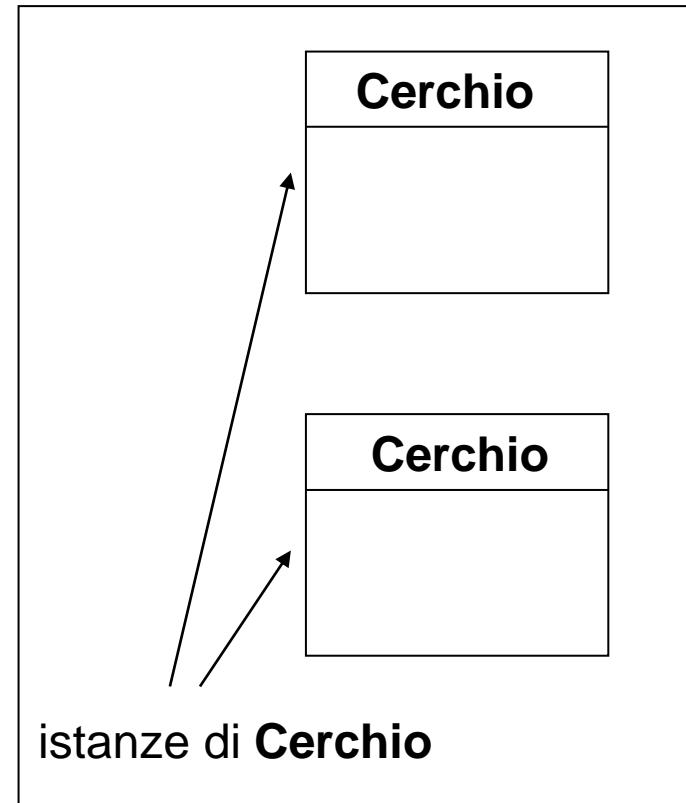
Classi e istanze



Area delle classi (memoria statica)



HEAP



Come rilevare dinamicamente il tipo di un oggetto? - I



Object offre il metodo **getClass()** che restituisce la classe dell'oggetto che lo esegue:

Forma f;

...

```
Class c = f.getClass();  
System.out.println(c.getName());
```

Class offre i seguenti metodi:

getName() restituisce il nome della classe come stringa

isInstance() - versione dinamica di **instanceof**:

Class c;

...

```
c.isInstance(f)
```

Come rilevare dinamicamente il tipo di un oggetto?



Altri metodi di **Class**

Class c = Class.forName("Cerchio");

Class c = Cerchio.class;

due modi per ottenere l'oggetto associato alla classe Cerchio;
o anche attraverso gli oggetti della classe:
Cerchio c = new Cerchio(); c.getClass();

c.getSuperclass(); restituisce la sopraclasse

c.newInstance(); crea un nuovo oggetto della classe **c**
(di tipo **Object**)

Come rilevare dinamicamente il tipo di un oggetto? Esempio

```
public class Esempio0 {  
    public static void main(String[] args) {  
        String f = "Ciao";  
        Class c = f.getClass();  
        System.out.println("Ciao e' istanza di " + c.getName() +  
                           "? " + c.isInstance(f));  
        ArrayList ar = new ArrayList();  
        System.out.println("L'ArrayList e' istanza di " +  
                           c.getName() + "? " + c.isInstance(ar));  
        System.out.println("La classe e' " + ar.getClass().getName());  
    }  
}
```



The screenshot shows a Windows Command Prompt window titled 'cmd.exe'. The output of the program is displayed:

```
C:\WINDOWS\system32\cmd.exe  
Ciao e' istanza di java.lang.String? true  
L'ArrayList e' istanza di java.lang.String? false  
La classe e' java.util.ArrayList  
Press any key to continue . . .
```

Gestione dell'oggetto Class



Così come un oggetto **Rettangolo** descrive le proprietà di un determinato oggetto di tipo Rettangolo, un oggetto **Class** descrive le proprietà di una determinata classe.

L'oggetto di tipo **Class** che rappresenta la classe **C** viene creato (**caricato**) dall'interprete, a partire dal file **C.class**, nel momento in cui la classe **C** è usata.

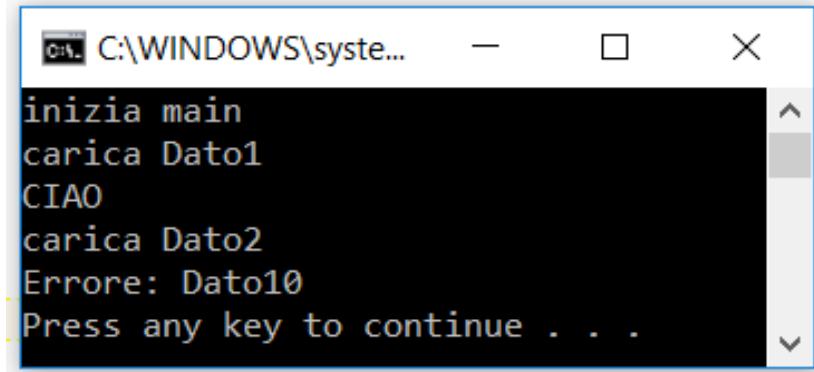
Se il file **C.class** non c'è, l'interprete lancia un'eccezione.

Vedere il prossimo esempio.

Class e errori a runtime – esempio – I

```
class Dato1 {  
    static {  
        System.out.println("carica Dato1");  
    }  
    public static void saluta() {System.out.println("CIAO");}  
}  
  
class Dato2 {  
    static {  
        System.out.println("carica Dato2");  
    }  
}
```

NB: non abbiamo definito la classe Dato10



```
C:\WINDOWS\system32  
inizia main  
carica Dato1  
CIAO  
carica Dato2  
Errore: Dato10  
Press any key to continue . . .
```



Class e errori a runtime – esempio - II

```
public class Esempio1 {  
  
    public static void main(String[] args) {  
        System.out.println("inizia main");  
        Dato1.saluta();  
        Dato1 d1 = new Dato1();  
        try {  
            Class c = Class.forName("Dato2");  
        } catch (ClassNotFoundException e) {System.out.println("Errore: " +  
                                         e.getMessage());}  
        try {  
            Class c = Class.forName("Dato10");  
        } catch (ClassNotFoundException e) {System.out.println("Errore: " +  
                                         e.getMessage());}  
    }  
}
```

Class e errori a runtime – esempio - III



Nell'esempio precedente, se la classe **Dato1** non è definita, si ottiene un errore di **compilazione** (type checking statico) alla linea

```
Dato1 d1 = new Dato1();
```

Viceversa, se **Dato10** non c'è, si verifica un errore a runtime e viene lanciata **l'eccezione**

ClassNotFoundException quando si esegue

```
Class c = Class.forName("Dato10");
```

Java Reflection (Riflessione) - I



La Reflection è un meccanismo molto potente fornito da Java per analizzare le funzionalità delle classi, ad esempio per ottenere a run-time informazioni su campi, metodi, costruttori, ...

Il package **java.lang.reflect** contiene le classi
Field, Method, Constructor

La classe **Class** contiene metodi come:
getFields, getMethods, getConstructors

La classe **Method** contiene i metodi:
getParameterTypes, invoke



Java Reflection (Riflessione) - II

Ad esempio la riflessione permette di leggere il nome di una classe e estrarre **dinamicamente** le informazioni su campi e metodi della classe.

La riflessione è usata in *JavaBeans*, l'architettura a componenti di Java, per analizzare dinamicamente le proprietà di nuovi componenti.

Vedere l'applicazione di esempio: ReflectionTest

Java Reflection (Riflessione) - III

```
public class ReflectionTest {  
    public static void main(String[] args) {  
        Scanner in = new Scanner(System.in);  
        System.out.println("Enter class name (e.g. java.util.Date): ");  
        String name = in.next();}  
        try { // print class name and superclass name (if != Object)  
            Class cl = Class.forName(name);  
            Class supercl = cl.getSuperclass();  
            System.out.print("class " + name);  
            if (supercl != null && supercl != Object.class) {  
                System.out.print(" extends " + supercl.getName()); }  
            System.out.print("\n{\n");  
            printConstructors(cl);  
            System.out.println();  
            printMethods(cl);  
            System.out.println();  
            printFields(cl);  
            System.out.println("}");  
        } catch (ClassNotFoundException e) {e.printStackTrace(); }  
        System.exit(0);  
    }
```



Java Reflection (Riflessione) - IV



```
public static void printConstructors(Class cl) {
    Constructor[] constructors = cl.getDeclaredConstructors();
    System.out.println("CONSTRUCTORS:");
    for (Constructor c : constructors) {
        String name = c.getName();
        System.out.print(" " + Modifier.toString(c.getModifiers()));
        System.out.print(" " + name + "(");

        // print parameter types
        Class[] paramTypes = c.getParameterTypes();
        for (int j = 0; j < paramTypes.length; j++) {
            if (j > 0) {
                System.out.print(", ");
            }
            System.out.print(paramTypes[j].getName());
        }
        System.out.println(");");
    }
}
```

Java Reflection (Riflessione) - V

```
public static void printMethods(Class cl) {  
    Method[] methods = cl.getDeclaredMethods();  
    System.out.println("METHODS:");  
    for (Method m : methods) {  
        Class retType = m.getReturnType();  
        String name = m.getName();  
  
        // print modifiers, return type and method name  
        System.out.print(" " + Modifier.toString(m.getModifiers()));  
        System.out.print(" " + retType.getName() + " " + name + "(");  
        // print parameter types  
        Class[] paramTypes = m.getParameterTypes();  
        for (int j = 0; j < paramTypes.length; j++) {  
            if (j > 0)  
                System.out.print(", ");  
            System.out.print(paramTypes[j].getName());  
        }  
        System.out.println(");");  
    }  
}
```



ReflectionTest – esecuzione (con java.util.Date)



```
C:\WINDOWS\system32\cmd.exe
Enter class name (e.g. java.util.Date):
java.util.Date
class java.util.Date
{
CONSTRUCTORS:
    public java.util.Date(java.lang.String);
    public java.util.Date(int, int, int, int, int, int);
    public java.util.Date(int, int, int, int, int);
    public java.util.Date();
    public java.util.Date(long);
    public java.util.Date(int, int, int);

METHODS:
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
    public int hashCode();
    public java.lang.Object clone();
    public int compareTo(java.util.Date);
    public volatile int compareTo(java.lang.Object);
    private void readObject(java.io.ObjectInputStream);
    private void writeObject(java.io.ObjectOutputStream);
    private final sun.util.calendar.BaseCalendar$Date normalize(sun.util.calendar.BaseCalendar$Date);
    private final sun.util.calendar.BaseCalendar$Date normalize();
    public static long parse(java.lang.String);
    public boolean after(java.util.Date);
    public boolean before(java.util.Date);
    public int getDate();
    public static java.util.Date from(java.time.Instant);
    public long getTime();
    public static long UTC(int, int, int, int, int, int);
    private static final java.lang.StringBuilder convertToAbbr(java.lang.StringBuilder, java.lang.String);
    private final sun.util.calendar.BaseCalendar$Date getCalendarDate();
    private static final sun.util.calendar.BaseCalendar getCalendarSystem(int);
    private static final sun.util.calendar.BaseCalendar getCalendarSystem(sun.util.calendar.BaseCalendar$Date);
    private static final sun.util.calendar.BaseCalendar getCalendarSystem(long);
    public int getDay();
    public int getHours();
    private static final synchronized sun.util.calendar.BaseCalendar getJulianCalendar();
    static final long getMillisOf(java.util.Date);
    public int getMinutes();
    public int getMonth();
    public int getSeconds();
    private final long getTimeImpl();
```

Verifica tipi per sviluppare metodi robusti - I

Nei metodi che utilizzano il downcast, la reflection permette di verificare il tipo degli oggetti prima di fare dei cast → prevenire eccezioni. Io consiglio di utilizzare tali controlli. Esempio:



```
public class Animale {
```

```
...
```

```
public int compareTo(Object o) {
```

```
    int ris = Integer.MIN_VALUE;
```

```
    if (o != null && o.getClass() == this.getClass()) {
```

// versione robusta: controlla null e tipo

```
        Animale a = (Animale) o;
```

```
        ris = nome.compareTo(a.nome);
```

```
}
```

```
return ris;
```

```
} //... continua
```

Verifica tipi per sviluppare metodi robusti - II



// ...

```
public boolean equals(Object o) {  
    boolean ris = false;  
    if (o != null && o.getClass() == this.getClass()) {  
        Animale a = (Animale) o;  
        ris = nome.equals(a.nome);  
    }  
    return ris;  
}  
} // fine classe Animale
```



Ringraziamenti

Grazie al Prof. Emerito Alberto Martelli del Dipartimento di Informatica dell'Università di Torino per aver redatto la prima versione di queste slides.



Programmazione III

Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

Tipi di dati generici (o parametrici) e collezioni



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND

Tipi generici: motivazioni - I



La specifica di tipi precisi nei metodi può portare a moltiplicare il codice (overloading di metodi quasi uguali) per trattare tutti i casi. Es: **per ogni tipo di dato devo definire un metodo printArray(). E' ridondante.**

```
public class SenzaGenerici {  
    public static void printArray(Integer[] ar) {  
        for (Integer element : ar) {System.out.print(element + ", ");}  
        System.out.println();  
    }  
    public static void printArray(Double[] ar) {  
        for (Double element : ar) {System.out.print(element + ", ");}  
        System.out.println();  
    }  
    public static void main(String[] args) {  
        Integer[] elenco1 = {1,2,3,4,5,6};  
        Double[] elenco2 = {1.1, 2.2, 3.3, 4.4};  
        printArray(elenco1);  
        printArray(elenco2);  
    }  
}
```

Tipi generici: motivazioni - II



I tipi generici sono stati introdotti per scrivere codice generico, applicabile a più tipi di dati (*reuse di codice*): **Il tipo E fa match con qualunque tipo non primitivo → basta un metodo per trattare ogni tipo di oggetto**

```
public static <E> void printArray(E[] ar) {  
    for (E element : ar)  
        System.out.print(element + ", ");  
    System.out.println();  
}  
  
public static void main(String[] args) {  
    Integer[] elenco1 = {1,2,3,4,5,6};  
    Double[] elenco2 = {1.1, 2.2, 3.3, 4.4};  
    String[] elenco3 = {"aa", "bb", "cc"};  
  
    printArray(elenco1);  
    printArray(elenco2);  
    printArray(elenco3);  
}
```

Tipi generici: motivazioni - III



Io potrei usare Object come tipo dei metodi. Tuttavia in quel caso dovrei fare esplicitamente il cast al tipo di oggetto atteso nell'invocazione del metodo:

```
public class SenzaGenericiObject {  
    public static void printArray(Object[] ar) {  
        for (Object element : ar)  
            System.out.print(element + ", ");  
        System.out.println();  
    }  
    public static Object getElement(Object[] ar, int index) {  
        Object ris = ar[index]; return ris; }  
    public static void main(String[] args) {  
        Integer[] elenco1 = {1,2,3,4,5,6};  
        Integer intero = (Integer)getElement(elenco1, 0); // senza il  
        // downcast non compila perché non si può assegnare un  
        // Object a una variabile di tipo Integer  
    }  
}
```

Tipi generici con Object – NB:



```
public static void main(String[] args) {  
    Integer[] elenco1 = {1,2,3,4,5,6};  
    Double[] elenco2 = {1.1, 2.2, 3.3, 4.4};  
    String[] elenco3 = {"aa", "bb", "cc"};  
    printArray(elenco1);  
    printArray(elenco2);  
    printArray(elenco3);  
    Integer intero = (Integer)getElement(elenco1, 0);  
    // intero = getElement(elenco1, 0); // errore, manca il downcast  
    System.out.println(intero);  
    System.out.println(getElement(elenco3, 0));
```

```
Object[] elencoMisto = {1, 2.2, "dd"};  
printArray(elencoMisto); // senza  
// controlli posso inserire elementi  
// eterogenei → fondamentale controllare  
// il tipo dei dati
```

}

```
C:\WINDOWS\system32\cmd.exe  
1, 2, 3, 4, 5, 6,  
1.1, 2.2, 3.3, 4.4,  
aa, bb, cc,  
1  
aa  
1, 2.2, dd,  
Premere un tasto per continuare . . .
```

Tipi generici: motivazioni - IV



Con i generici il compilatore inferisce il tipo degli oggetti in fase di **type checking statico**. Il compilatore verifica la compatibilità tra il tipo attuale e il tipo generico E, e sostituisce il tipo attuale a E, inserendo i cast (questa operazione si chiama **erasure**):

```
public class ConMetodiGenerici {  
    public static <E> void printArray(E[] ar) {  
        for (E element : ar)  
            System.out.print(element + ", ");  
        System.out.println(); }  
  
    public static <E> E getElement(E[] ar, int index) {  
        E ris = ar[index]; return ris; }  
  
    public static void main(String[] args) {  
        Integer[] elenco1 = {1,2,3,4,5,6};  
        Integer intero = getElement(elenco1, 0);  
        System.out.println(intero);  
        System.out.println(getElement(elenco3, 0)); }  
}
```

Il compilatore sa che verrà restituito un Integer e inserisce il cast nel bytecode
→ non devo scriverlo io

Tipi generici (o tipi parametrici) - I



I tipi di dati *generici* (o *parametrici*) sono classi e metodi che hanno un *parametro di tipo*.

Consideriamo una classe Point. Senza tipi generici si può solo definire:

```
ArrayList miaLista = new ArrayList();
miaLista.add(new Point(...));
Point e = (Point)miaLista.get(0);
```

Gli oggetti **ArrayList** possono contenere oggetti di qualunque tipo. Quindi, quando si estrae un oggetto da miaLista, è necessario fare il downcast. Tuttavia, di fronte a errori di programmazione non si può essere certi del tipo dell'oggetto che si estrae (il programmatore potrebbe inserire oggetti di tipo diverso).

Tipi generici (o tipi parametrici) - II



Invece, utilizzando i tipi generici si può scrivere:

```
ArrayList<Point> miaLista = new ArrayList<Point>();  
miaLista.add(new Point(...));  
Point e = miaLista.get(0);
```

`ArrayList<Point>` è un **tipo parametrico**, in cui è stato specificato il tipo argomento da applicare alla classe generica.

`miaLista` può contenere solo oggetti **Point (controllo a tempo di compilazione)** → verifica statica delle `add()` → si accettano solo `Point`.

Non è più necessario il cast nell'ultima istruzione: il compilatore sa che l'oggetto estratto è un `Point`.



Variabili di tipo - I

Nelle definizioni di metodi e classi generici, i tipi generici vengono chiamati **variabili di tipo** in quanto rappresentano tipi "formali", da abbinare ai tipi "attuali" specificati in sede di creazione degli oggetti, o di invocazione dei metodi:

```
public static <E> E getElement(E[] ar, int index)
```

→ **E** è la variabile di tipo, associata al tipo attuale nelle invocazioni del metodo. Es, in:

```
Integer[] elenco1 = {1,2,3,4,5,6}; \\ autoboxing  
Integer intero = getElement(elenco1, 0);
```

E prende valore "Integer"

Definizione di classi generiche - I



Oltre a usare classi generiche predefinite, come `ArrayList<E>`, è possibile definire delle nuove classi generiche. Vediamo un esempio di definizione di una *classe generica (o parametrica)* che realizza una coppia di oggetti di tipo `T`.

```
public class Pair<T> {  
    public Pair() {  
        first = null;  
        second = null; }  
  
    public Pair(T first, T second) {  
        this.first = first;  
        this.second = second; }  
  
    public T getFirst() { return first; }  
    public T getSecond() { return second; }  
  
    private T first;  
    private T second;  
}
```

Definizione di classi generiche - II



Per usare la classe `Pair<T>`:

```
Pair<String> coppia = new Pair<String>("AAA","BBB");
```

// oppure, usando la “diamond notation”:

```
// Pair<String> coppia = new Pair<>("AAA","BBB");
```

```
String s = coppia.getFirst();
```

NB: Non è possibile istanziare un parametro di tipo con un tipo primitivo. Le variabili di tipo sono solo riferimenti

`Pair<double>` non è accettato

`Pair<Double>` è corretto

Definizione di classi generiche – III



Altro esempio: Pila<T> usa una LinkedList<T> per mantenere la lista di elementi, ma la gestisce come uno stack con push() e pop()

```
public class Pila<T>
{
    private List<T> list = new LinkedList<T>();
    public boolean isEmpty() {return list.isEmpty();}
    public void push(T v) {list.addFirst(v);}
    public T pop() {return list.removeFirst();}
}

//... altra classe: applicazione che usa Pila<T>
public static void main(String[] args)
{
    Pila<String> stack = new Pila<String>();
    stack.push("a");
    stack.push("bcd");
    String s1 = stack.pop();
    String s2 = stack.pop();
    System.out.println(s1);
    System.out.println(s2);
}
```



Codice generico e macchina virtuale

La macchina virtuale non ha classi con parametri di tipo: tutti gli oggetti appartengono a classi ordinarie (senza parametri).

Il compilatore traduce le classi generiche, e le istruzioni che utilizzano oggetti di queste classi, in classi e istruzioni accettate dalla macchina virtuale.

Ogni volta che si definisce un *tipo generico*, il compilatore lo trasforma con una operazione in un **tipo grezzo (raw type)**. Il nome del tipo grezzo coincide con quello del tipo generico senza i parametri di tipo. Tutte le variabili di tipo che compaiono nella definizione della classe generica sono sostituite con **Object**.

Tipo grezzo (Raw type) - I



Ad esempio il tipo grezzo di `Pair<T>` è:

```
public class Pair {
```

```
    public Pair()
```

```
        { first = null; second = null; }
```

```
    public Pair(Object first, Object second)
```

```
        { this.first = first; this.second = second; }
```

```
    public Object getFirst()
```

```
        { return first; }
```

```
    public Object getSecond()
```

```
        { return second; }
```

```
    private Object first;
```

```
    private Object second;
```

```
}
```

Tipo grezzo (Raw type) - II



Tutte le classi parametriche diventano a runtime delle classi senza parametri. Infatti gli oggetti appartengono ad una sola classe. Es., consideriamo **Pair<T1, T2>** in cui usiamo 2 tipi parametrici per permettere di gestire coppie di elementi eterogenei:

```
public class PairApp {  
    public static void main(String[] args) {  
  
        Pair<String, Integer> p1= new Pair<>("Mario Rossi", 30);  
        Pair<Integer, Integer> p2= new Pair<>(25, 48);  
  
        System.out.println("Le due coppie sono istanze della stessa classe? " +  
                           (p1.getClass() == p2.getClass()));           // stampa true  
  
        System.out.println("A quale classe appartengono? " +  
                           p1.getClass() + "; " + p2.getClass());     // stampa Pair; Pair  
    }  
}
```

Tipo grezzo (Raw type) - III



Tuttavia questa è solo una questione implementativa. Dal punto di vista del programmatore è possibile usare le classi generiche.

È compito del compilatore tradurle in classi senza parametri, facendo tutti i controlli necessari per assicurarsi che non si verifichino errori di tipo.

È però da notare che la scelta implementativa ha degli effetti sulle operazioni consentite sulle classi generiche, come vedremo più avanti.

Per il momento, per es., si noti che una classe parametrica con parametro di tipo T non può utilizzare T nella dichiarazione di variabili statiche, o all'interno di un metodo statico o di codice di inizializzazione statico (solo gli oggetti hanno il tipo istanziato)

Tipo grezzo (Raw type) - esempio



```
public class PairWrong {  
    public static void main(String[] args) {  
    }  
}  
class Pair<T> {  
    public static T getPrimo() { return 1; }  
    public static T getSecondo() {return 2; }  
    public String toString() { return "1, 2"; }  
}
```

Errori di compilazione:

error: non-static type variable T cannot be referenced from a static context

public static T getPrimo() {

 ^

error: non-static type variable T cannot be referenced from a static context

public static T getSecondo() {

 ^

2 errors

Ereditarietà tra generici



Come con le normali classi Java, **anche con i generici possiamo definire relazioni di sottoclasse**. Es., estendo `Pair<T>` per avere coppie di elementi non ordinate (`<2,3>` è uguale a `<3,2>`)

```
class NonOrderedPair<T> extends Pair<T> {  
    public NonOrderedPair(T uno, T due) {  
        super(uno, due); }  
    public boolean equals(Object o) {  
        if (o==null) return false;  
        if (this.getClass()!=o.getClass()) return false;  
        return ((this.getPrimo().equals(coppia.getPrimo())) &&  
                this.getSecondo().equals(coppia.getSecondo())) ||  
               (this.getSecondo().equals(coppia.getPrimo())) &&  
               this.getPrimo().equals(coppia.getSecondo()));  
    }  
}
```

Overloading di metodi generici



Un metodo generico può essere **overloaded** da altri metodi generici o anche da metodi non generici. In caso di overloading il compilatore seleziona il metodo più specifico che fa match con l'invocazione. Es, dati: **Integer[] ar = {1,2,3}; C.printArray(ar); viene eseguito il secondo metodo printArray().**

```
public class C {  
    public static <E> void printArray(E[] ar) {  
        for (E element : ar) {  
            System.out.print(element + ", "); }  
        System.out.println(); }  
  
    public static void printArray(Integer[] ar) { ← ←  
        for (Integer element : ar) {  
            System.out.print("Speciale: " + element + ", "); }  
        System.out.println(); }  
}
```

Overloading vs. overriding di metodi - I

Attenzione ai metodi ereditati! Se faccio **overloading** di equals() gli oggetti Pair NON eseguono questo metodo quando utilizzati in upcasting come Object (es. Arrays.sort())



```
class Pair<T1, T2> {  
    private T1 primo; private T2 secondo;  
    public Pair(T1 uno, T2 due) { primo = uno; secondo = due; }  
    public T1 getPrimo() { return primo; }  
    public T2 getSecondo() { return secondo; }  
    public String toString() { // overrides toString() di Object  
        return "primo: " + primo.toString()+"\n"+ "secondo: "+secondo.toString(); }  
    public boolean equals(Pair<T1,T2> coppia) { // overloads equals()  
        // di Object  
        if (coppia==null) return false;  
        return (primo.equals(coppia.primo) &&  
                secondo.equals(coppia.secondo));  
    } }
```



Overloading vs. overriding di metodi - II

```
class Pair<T1, T2> {  
    private T1 primo; private T2 secondo;  
    public Pair(T1 uno, T2 due) {  
        primo = uno; secondo = due; }  
    ...  
    public String toString() { // overrides toString() di Object  
        return "primo: " + primo.toString() + "\n" +  
            "secondo: " + secondo.toString();  
    }  
    public boolean equals(Object o) { // OK!! overrides equals() di Object  
        if (o==null) return false;  
        if (this.getClass() != o.getClass()) return false;  
        Pair<T1, T2> coppia = (Pair<T1, T2>)o;  
        return (primo.equals(coppia.primo) &&  
            secondo.equals(coppia.secondo));  
    }  
}
```

Vincoli sui tipi parametrici - I



A volte non ha senso sostituire un tipo di riferimento qualunque ad un tipo parametrico in una classe generica. Es. se volessi aggiungere a `Pair<T>` il metodo `getMassimo()`, che restituisce il valore più alto in una coppia, dovrei imporre che il tipo sostituito a `T` implementi `Comparable`. Per farlo aggiungo a definizione di `Pair` la **restrizione di tipo su T**: dò upperbound al tipo parametrico di una classe

```
class Pair <T extends Comparable<T>> {  
    private T primo; private T secondo;  
public T getMassimo() {  
    if (primo.compareTo(secondo) >=0) return primo;  
    else return secondo; }  
// ... altri metodi di Pair  
}
```

Vincoli sui tipi parametrici - II



class Pair <T extends Comparable<T>>

- ⇒ Data la restrizione, il raw type con cui sostituire T è Comparable.
- ⇒ Il compilatore farà il controllo di tipo sui parametri attuali di costruttori e di invocazioni dei metodi, impedendo di creare Pair di oggetti che non implementino Comparable.

La restrizione di tipo può specificare al più una classe e n interfacce: **class C <T extends Classe0 & Interf₁ &...& Interf_n>**

NB: in questo caso:

- **il tipo grezzo (raw type) di T è Classe0**
- C può essere instanziata solo con elementi in cui T è Classe0 o un sottotipo di Classe0

Vincoli sui tipi parametrici – esempio – I



Consideriamo due classi: MioTipo e SottoTipo:

```
class MioTipo {  
    private String x;  
    public MioTipo(String x) { this.x = x; }  
    public String toString() { return x; }  
}  
  
class SottoTipo extends MioTipo {  
    public SottoTipo(String x) {  
        super(x);  
        //... altro codice }  
}
```



Vincoli sui tipi parametrici – esempio - II



Se impongo MioTipo come upper bound di T potrò creare oggetti Pair che contengono elementi MioTipo o SottoTipo, non altro:

```
class Pair <T extends MioTipo> {  
    // ... La definizione di Pair già data ...  
}  
  
public static void main(String[] args) {  
    Pair<MioTipo> coppia1 = // OK  
        new Pair<MioTipo>(new MioTipo("Ciao "), new MioTipo("mondo"));  
  
    Pair<SottoTipo> coppia2 = // OK  
        new Pair<SottoTipo>(new SottoTipo("Ciao "),  
            new SottoTipo("mondo2"));  
  
    Pair<String> coppiaWrong =  
        new Pair<String>("Ciao ", "mondo"); // NO!!  
}
```

Vincoli sui tipi parametrici – esempio - III



Per garantire la robustezza del software può essere necessario stringere i tipi generici utilizzati nei metodi di una classe/interface parametrica. Esempio:

```
public class Studente implements Comparable<Studente> {  
    private int matricola;  
    private String nome;  
    private String cognome;  
public int compareTo(Studente s) { // siamo certi che s sia Studente  
    // perché il compilatore richiede compareTo(Studente), non  
    // compareTo(Comparable)  
    if (this.cognome.compareTo(s.cognome)<0 ||  
        (this.cognome.equals(s.cognome) &&  
         this.nome.compareTo(s.nome)<0))  
        return -1;  
    else if (this.equals(s)) return 0;  
    else return 1;  
}
```



Collezioni - I

Java fornisce un insieme di classi che realizzano le strutture dati più utili (collezioni), come liste o insiemi.

Il package `java.util` che contiene le collezioni distingue fra *interfacce* e *implementazioni*.

Ad esempio **List<E>** è una **interface** che specifica le operazioni principali sulle liste.

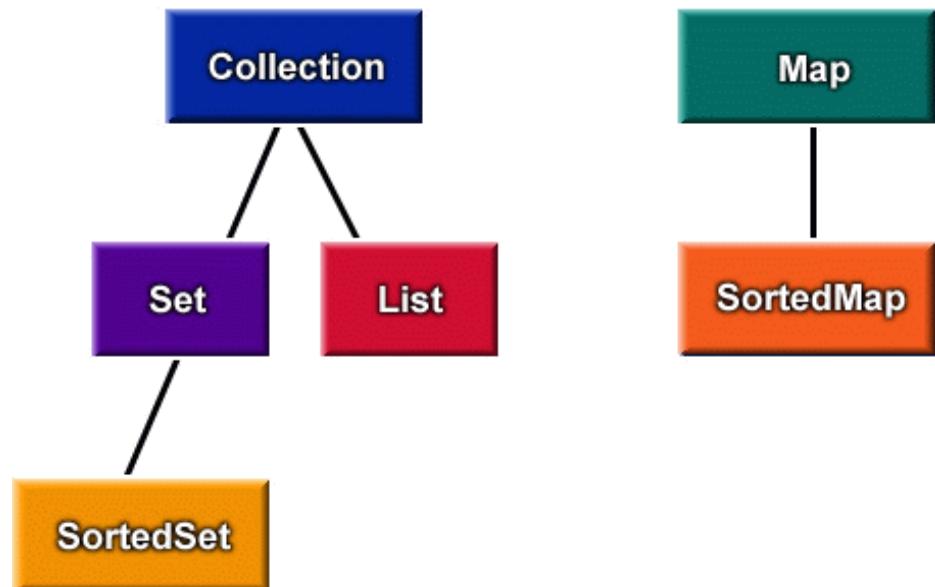
ArrayList<E> e **LinkedList<E>** sono due classi concrete che implementano l'interfaccia **List<E>** in modi diversi.



Strutture dati: Interfacce

Sono tutte Interface:

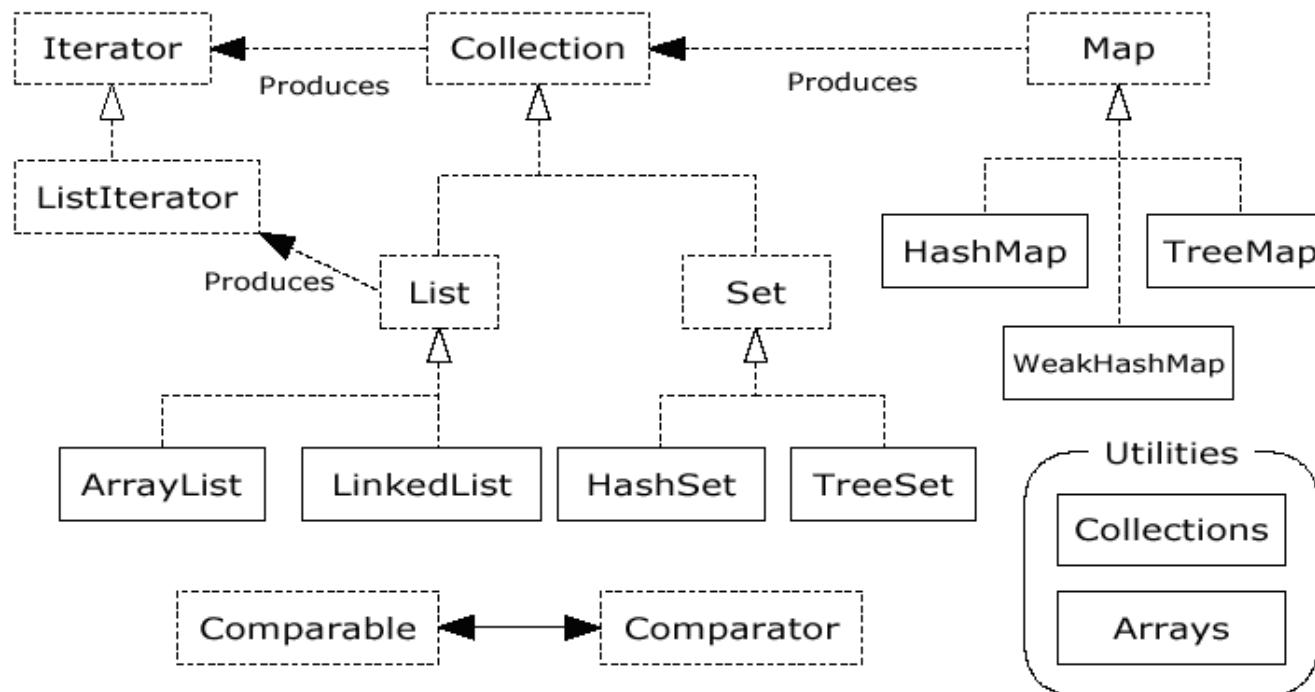
- **Collection:** un arbitrario gruppo di oggetti
- **List:** un gruppo di oggetti memorizzati in una data sequenza
- **Set:** un gruppo di oggetti senza duplicati
- **Map:** un gruppo di coppie (oggetti) chiave-valore



Strutture dati: implementazioni



Ogni interfaccia ha più implementazioni che possono essere scelte in modo indifferente a seconda delle esigenze (anche di efficienza)



Collezioni - II



L'interfaccia **Set<E>** specifica un insieme:

```
interface Set<E> {  
    boolean add(E o);  
    boolean contains(Object o);  
    boolean remove(Object o);  
    .....  
}
```

La classe **HashSet<E>** implementa l'interfaccia **Set<E>** mediante una tabella hash.

La classe **TreeSet<E>** implementa l'interfaccia **Set<E>** garantendo che gli elementi dell'insieme siano ordinati. Gli elementi dell'insieme devono implementare l'interfaccia **Comparable<T>**. Oltre ai metodi definiti in **Set<E>**, vengono forniti altri metodi come **first()** o **last()**.

Collezioni - III



Quando in un programma si utilizza un insieme, non è necessario sapere quale implementazione dell'insieme verrà utilizzata.

Conviene utilizzare la *classe concreta* solo quando si costruisce l'insieme, mentre si può utilizzare il *tipo dell'interfaccia* per specificare il riferimento alla collezione (upcasting):

```
Set<Point> s = new HashSet<Point>;
Point e = ...;
s.add(e);
...
...
```

In questo modo, se si decide di usare l'implementazione **TreeSet**, è sufficiente modificare la prima istruzione in cui si crea l'insieme, senza modificare il resto del programma.



Iteratori - I

L'interfaccia **Collection** prevede vari metodi:

```
public interface Collection<E> {  
    boolean add(E element);  
    Iterator<E> iterator();  
    ...  
}
```

Il metodo **iterator** restituisce un oggetto che implementa l'interfaccia **Iterator**:

```
public interface Iterator<E> {  
    E next();  
    boolean hasNext();  
    void remove();  
}
```



Iteratori - I

La chiamata ripetuta del metodo **next()** permette di scorrere gli elementi della collezione uno alla volta.

```
void printCollection(Collection<Object> c) {  
    Iterator<Object> iter = c.iterator();  
    while (iter.hasNext()) {  
        Object ob = iter.next();  
        System.out.println(ob);  
    }  
}
```

Per iterare sugli elementi della collezione si può usare anche una istruzione di tipo **for each**:

```
void printCollection(Collection<Object> c) {  
    for(Object ob : c)  
        System.out.println(ob);  
}
```

L'istruzione può essere usata con qualunque collezione che implementi l'interfaccia **Iterable**.



Auto-boxing

Le strutture come **Collection** possono contenere solo oggetti. Quindi per inserire un tipo primitivo (es. int) occorre convertirlo (**boxing**) nel corrispondente oggetto (es. Integer). Viceversa, quando si estraе un oggetto dalla collezione occorre riconvertirlo nel tipo primitivo (**unboxing**). Il compilatore fa automaticamente boxing e unboxing.

```
ArrayList<Integer> al = new ArrayList<Integer>();
```

```
al.add(0, new Integer(25));  
int x = (al.get(0)).intValue();
```

```
// oppure, grazie all'autoboxing:  
al.add(0, 25);  
int x = al.get(0);
```



Array e collezioni

È possibile specificare il tipo degli elementi di una collezione, es. `ArrayList<Point>`, esattamente come si fa con gli array: `Point[]`.

Tuttavia ci sono differenze:

1. non è possibile istanziare il parametro di tipo di una collezione con un tipo primitivo (i tipi parametrici sono riferimenti);
2. **gli oggetti array mantengono a runtime l'informazione sul tipo degli elementi, mentre gli oggetti collezione perdono l'informazione a runtime.**

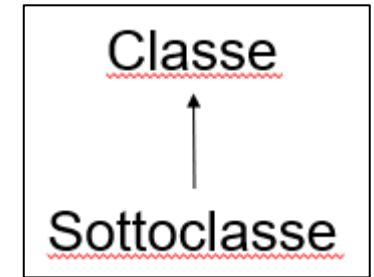
Vediamo le conseguenze del secondo punto.

Generici e sottotipi - I



Con gli array sono ammissibili:

1. Classe[] arrayClasse;
2. SottoClasse[] arraySottoclasse;
3. arraySottoclasse = new Sottoclasse[2];
4. arrayClasse = arraySottoclasse;
5. arrayClasse[0] = new Classe(...); // compila correttamente



Il compilatore accetta queste istruzioni, in particolare l'assegnazione nella linea 4 perché un array di **Sottoclasse** è una sottoclasse di array di **Classe**.

L'interprete dà errore a *runtime* eseguendo la linea 5, perché **arrayClasse** fa riferimento ad un array di oggetti di **Sottoclasse**, creato alla linea 3, a cui non è possibile assegnare oggetti **Classe**.

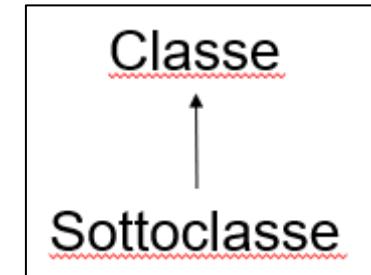
L'interprete può rilevare l'errore perché, quando viene creato un array, gli viene associata l'informazione sul tipo degli elementi.



Generici e sottotipi - II

Vediamo lo stesso programma con ArrayList al posto degli array:

1. `ArrayList<Classe> alClasse;`
2. `ArrayList<Sottoclasse> alSottoclasse;`
3. `alSottoclasse = new ArrayList<Sottoclasse>();`
4. `alClasse = alSottoclasse;`
5. `alClasse.add(new Classe(...));`



In questo caso **il compilatore dà errore** alla linea 4, dicendo che i tipi **ArrayList<Classe>** e **ArrayList<Sottoclasse>** sono incompatibili.

Se il compilatore accettasse questo programma l'interprete non sarebbe in grado di scoprire a runtime l'errore alla linea 5. Questo perché le collezioni non conservano a runtime l'informazione sul tipo degli elementi.

Generici e sottotipi - III



Data una classe parametrica **C<T>**, e due tipi **Tipo1** e **Tipo2**, tali che **Tipo2** è sottotipo di **Tipo1**, non esiste nessuna relazione tra i tipi **C<Tipo1>** e **C<Tipo2>**.

Es., anche se **SottoTipo** extends **MioTipo**, non è possibile assegnare un **Pair<SottoTipo>** a un **Pair<MioTipo>**, né viceversa (mentre sarebbe possibile assegnare un **SottoTipo[]** a un **MioTipo[]**).

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

```
    Pair<MioTipo> coppia3;
```

```
    Pair<SottoTipo> coppia4;
```

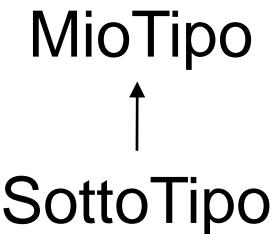
```
    coppia3 = coppia4; // error: incompatible types:
```

```
        // Pair<SottoTipo> cannot be converted to Pair<MioTipo>
```

```
    coppia4 = coppia3; // error: incompatible types:
```

```
        // Pair<MioTipo> cannot be converted to Pair<SottoTipo>
```

```
}
```



Generici e sottotipi - III



Il comportamento descritto prima è molto restrittivo. Ad esempio, il seguente metodo per stampare gli elementi di una collezione:

```
void printCollection(Collection<Object> c) {  
    for(Object ob : c)  
        System.out.println(ob);  
}
```

può essere usato solo per stampare gli elementi di una **Collection<Object>**. Il metodo non può avere come parametro, ad esempio, una **Collection<Integer>**, perché i due tipi sono incompatibili.



Il tipo jolly (wildcard) - I

Per definire una *Collection* di qualunque cosa si può usare la notazione **Collection<?>**, dove ? è la wildcard (jolly).

```
void printCollection(Collection<?> c) {  
    for(Object ob : c)  
        System.out.println(ob);  
}
```

È possibile chiamare **printCollection()** con un parametro **Collection<Integer>**, **Collection<Point>** o collezione di qualsiasi altro tipo: **l'upper bound di default di ? è Object.**



Il tipo jolly (wildcard) - II

Anche con il tipo jolly ci sono delle limitazioni.
Ad esempio in

```
Collection<?> c = new ArrayList<String>();  
c.add(new Object());
```

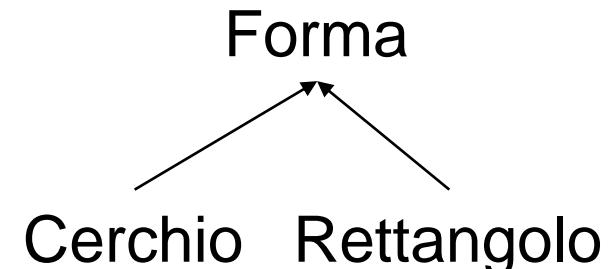
il compilatore accetta la prima istruzione, ma non la seconda. Questo perché non può garantire la correttezza del tipo dell'elemento che viene inserito nella collezione (di cui non conosce il tipo).

Limitazioni sul tipo jolly - I



Consideriamo la classe astratta:

```
public abstract class Forma {  
    public abstract void disegna();  
}
```



e varie sue sottoclassi come **Cerchio**, **Rettangolo**, ... che implementano il metodo **disegna**. Un metodo per disegnare le forme contenute in una lista potrebbe essere:

```
public void disegnaTutto(List<Forma> forme) {  
    for (Forma f: forme)  
        f.disegna();  
}
```



Limitazioni sul tipo jolly - II

La definizione

```
public void disegnaTutto(List<Forma> forme) {
```

è restrittiva perché il metodo **disegnaTutto** può essere applicato solo a una **List<Forma>** e non, per esempio, a una **List<Rettangolo>**.

D'altra parte

```
public void disegnaTutto(List<?> forme) {
```

sarebbe troppo generale perché il ? può essere legato a qualunque tipo, in particolare ad **Object** che non ha il metodo **disegna**.



Limitazioni sul tipo jolly - III

Per risolvere questo problema, il ? può essere limitato con la notazione

```
public void disegnaTutto(List<? extends Forma> forme) {  
    for (Forma f: forme)  
        f.disegna();  
}
```

in cui il tipo degli elementi della lista può essere una qualunque sottoclasse di **Forma**. Quindi certamente avrà il metodo disegna.

In questo caso il raw type che viene assegnato a List è Forma (il più generico tipo che gli elementi della collezione possono avere – **Forma è l'upper bound della wildcard**).



Uso scorretto delle wildcards

Voi non potete usare le wildcard nelle dichiarazioni delle variabili: il tipo dei dati deve essere definito in modo preciso.

```
public class WildCardTest {  
  
    public static void main(String[] args) {  
        ArrayList<?> lista1;  
        //lista1.add("Ciao"); // error: no suitable method found for add(String)  
        //lista1.add(new Object()); // idem, per Object  
  
        ArrayList<? extends Number> lista2;  
        //lista2.add(new Integer(2));  
        // error: no suitable method found for add(Integer)  
        Number num = 54;  
        // lista2.add(num); //error: no suitable method found for add(Number)  
    }  
}
```

Esempi di uso corretto delle wildcards



```
public class WildCardTest {
```

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

```
        ArrayList<Integer> numeri = new ArrayList<Integer>();
```

```
        numeri.add(54); numeri.add(23); // OK autoboxing
```

```
        System.out.println(sum(numeri)); // OK
```

```
        ArrayList<Number> numeri2 = new ArrayList<Number>();
```

```
        Number num = 54;
```

```
        numeri2.add(num); numeri2.add(num); // con upcasting OK!
```

```
        System.out.println(sum(numeri2));
```

```
}
```

```
    public static double sum(ArrayList< ? extends Number > lista) {
```

```
        double tot = 0;
```

```
        for (Number el : lista) {
```

```
            tot = tot+el.doubleValue(); }
```

```
        return tot;
```

```
}
```

Upcasting e tipi generici



```
public class PairApp {  
  
    public static void main(String[] args) {  
  
        Pair<MioTipo> coppia5 =  
  
            new Pair<>(new SottoTipo("Prova "),  
  
                        new SottoTipo("upcasting"));  
  
        Pair<MioTipo> coppia1 = coppia5;  
  
    }  
  
}
```

Usando upcasting non ci sono problemi di compilazione né in fase di esecuzione perché sia coppia1 che coppia5 sono viste come variabili di tipo Pair<MioTipo>



Programmazione III

Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

Classi innestate (e classi locali), Lambda expressions



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND

Classi e interfacce innestate



Le classi possono essere dichiarate:

- **all'interno di altre classi in qualità di membri (classi interne: possono essere statiche o di istanza)**
- **all'interno di blocchi di codice (classi interne locali).**

La definizione di tipi innestati è utile per:

- **Definire tipi strutturati e resi visibili all'interno di gruppi correlati logicamente**
- **Connettere in modo semplice e efficace oggetti correlati esplicitamente:** un tipo innestato è considerato parte del tipo in cui è racchiuso e con il quale condivide una relazione di fiducia per cui ognuno dei due può accedere a tutti i membri (variabili, metodi) dell'altro (anche a quelli privati)
- **Information hiding di tipi di dati** (classi all'interno di altre)

Classi interne – Inner class: esempio (classe di istanza) - I

```
class ClasseEsterna {  
    private int val;  
    private ClasseInterna ci;  
  
    public ClasseEsterna(int v) {  
        val = v; ci = new ClasseInterna(v); }  
  
    public int getVal() { return val; }  
  
    public ClasseInterna getCi() { return ci; } // ci serve per sperimentare...  
  
    public void setVal(int val) { // anche se viola information hiding  
        this.val = val;  
        ci.vallnterno = val; // la classe esterna accede ai  
        // componenti della classe interna  
    }  
}  
class ClasseInterna {  
    private int vallnterno;  
  
    public ClasseInterna(int v) { vallnterno = v; }  
    public int getVallnterno() { return vallnterno; }  
}
```



Classi interne – Inner classes (tipi innestati non statici)



Una classe innestata, dichiarata membro della classe che la racchiude, si comporta come una qualsiasi classe ma il suo nome e il suo grado di accessibilità dipendono dalla classe contenitore:

- **La visibilità della classe innestata è per default la stessa di quella del contenitore** (a meno che la classe inner venga dichiarata ***private***, nel qual caso è solo visibile all'interno del contenitore)
- **Il nome della classe innestata è così composto:**

NomeContenitore.NomeClasseInnestata

Classi interne – Inner class: esempio

class ClasseEsterna {

(classe di istanza) - I

private int val;

private ClasseInterna ci;

public ClasseEsterna(int v) {

 val = v; ci = new ClasseInterna(v); }

public int getVal() { return val; }

public ClasseInterna getCi() { return ci; } // ci serve per sperimentare...

public void setVal(int val) {

// anche se viola information hiding

 this.val = val;

ci.vallnterno = val; // la classe esterna accede ai
 // componenti della classe interna

}

class ClasseInterna {

 private int vallnterno;

 public ClasseInterna(int v) { vallnterno = v; }

 public int getVallnterno() { return vallnterno; }

}



Classi interne–Inner class: es. (classe di istanza) - II



```
public class Test1 {  
    public static void main(String[] args) {  
        ClasseEsterna ce = new ClasseEsterna(10);  
        System.out.println("Valore del dato di oggetto ClasseEsterna: " + ce.getVal());  
        ClasseEsterna.ClasseInterna ci = ce.getCi();  
        System.out.println("Valore del dato di oggetto ClasseInterna: " + ci.getVallnterno());  
        ce.setVal(30);  
        System.out.println("Valore del dato di oggetto ClasseInterna " +  
                           "dopo la modifica "+ ci.getVallnterno());  
    }  
}
```

The screenshot shows the 'Output' window of an IDE during the execution of the 'Progr3NestedClasses' program. The window title is 'Output - Progr3NestedClasses (run)'. The output text is as follows:

```
run:  
Valore del dato di oggetto ClasseEsterna: 10  
Valore del dato di oggetto ClasseInterna: 10  
Valore del dato di oggetto ClasseInterna dopo la modifica 30  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Classi interne – Inner classes - outerThis



Gli oggetti di una classe interna hanno un riferimento **outerThis** agli oggetti contenitori. Quindi essi possono accedere alle variabili di istanza dei contenitori, facendovi riferimento come se fossero locali. Es:

```
public class Test2 {  
    class Esterna {  
        private int val; Interna ci;  
        public Esterna(int v) {  
            val = v; ci = new Interna(); }  
        ... getter, setter, etc.  
  
        class Interna {  
            private int vallnterno;  
            public Interna() {  
                vallnterno = val; } // legge il valore di val definito nella classe esterna  
            public int getVallnterno() { return vallnterno; }  
        }}}
```

Esempio: BankAccount - I



Nella classe BankAccount, definita nel prossimo lucido, la classe Operation non ha motivo di essere dichiarata come normale classe (serve solo in BankAccount) → può essere una classe innestata: il vantaggio è che BankAccount può accedere direttamente a tutti i membri di Operation e viceversa, anche a quelli privati.

Inoltre definiamo Operation come classe inner **private** (interna locale) per cui visibile solo all'interno di BankAccount. In questo modo non esportiamo la struttura dati delle Operation sui conti correnti (incapsulamento).

Esempio: BankAccount - II



```
public class BankAccount {  
    private long number;  
    private long balance;  
    private Vector<Operation> history;  
    private class Operation {  
        private String op;  
        private long amount;  
        private Operation(String o, long q) {op = o; amount = q;}  
        public String toString() {  
            return number + ": " + op + " " + amount;}  
    }  
    public BankAccount(int n) {  
        number = n; history = new Vector<Operation>(); }  
    public void deposit(Person p, long amount) {  
        Operation op = new Operation("deposit", amount);  
        balance = balance+amount;  
        history.add(op);  
    }  
}
```

Tipi innestati statici (o classi interne statiche)



Se la classe innestata non ha bisogno di far riferimento ai membri della classe contenitrice ma si vogliono correlare i tipi (e magari rendere privata la dichiarazione della classe innestata) si può creare una classe interna statica.

Le classi statiche **non** mantengono il riferimento *outerThis*.

Es. in BankAccount la lista dei permessi forniti ai clienti potrebbe essere definita da una classe interna statica Permissions che specifica, per ogni persona, i diritti (deposito, prelevamento, etc.).



Esempio: BankAccount - III

```
public class BankAccount {  
    private long number;  
    private long balance;  
    private Vector<Permissions> grantedPerms;  
    private static class Permissions {  
        private Person pers;  
        private boolean canDeposit;  
        private boolean canWithdraw;  
        private Permissions(Person p, boolean d, boolean w) {  
            pers = p; canDeposit = d; canWithdraw = w;  
        }  
        public BankAccount(int n) {  
            number = n;  
            grantedPerms = new Vector<Permissions>();}  
        public void grantPermissions(Person p, boolean d, boolean w) {  
            Permissions perm = new Permissions(p, d, w);  
            grantedPerms.add(perm);  
        }  
    }  
}
```



Classi innestate in interfacce

Se noi vogliamo definire una Interface I corredata di implementazione di default possiamo innestare l'implementazione nella Interface come classe interna statica.

Le classi che implementano I possono

- estendere l'implementazione di default (→ usare direttamente l'implementazione di default), oppure
- modificare l'implementazione di default, eventualmente riutilizzandone delle parti.

Classi innestate in interfacce – esempio - I



```
public interface Message {  
    public String getText();  
    public String getDest();  
  
    static class MsgImpl { // Implementazione di default di Message  
        protected int destinatario;  
        protected String txt;  
  
        public MsgImpl(int destinatario, String testo) {  
            this.destinatario = destinatario;  
            txt = testo;  
        }  
        public String getText(){  
            return txt;  
        }  
        public String getDest() {  
            return Integer.toString(destinatario);  
        }  
    }  
}
```

Classi innestate in interfacce – esempio - II



```
public class SMSusaDefaultImpl extends Message.MsgImpl  
    implements Message {
```

```
public SMSusaDefaultImpl(int destinatario, String testo) {  
    super(destinatario, testo);  
}
```

SMSusaDefaultImpl usa l'implementazione di default di Message → non fornisce una sua implementazione dell'Interface Message

Classi innestate in interfacce – esempio - III

```
public class SMSusaComponentsImpl implements Message {  
    private Message.MsgImpl msg;  
  
    public SMSusaComponentsImpl(int dest, String txt) {  
        msg = new Message.MsgImpl(dest, txt);  
    }  
    public String getText() { return ">> " + msg.getText(); }  
  
    public String getDest() { return msg.getDest(); }  
}
```



SMSusaComponentsImpl incapsula un MsgImpl e usa i suoi metodi per reimplementare l'interface Message → io potrei aggiungere istruzioni ai metodi per modificare il comportamento rispetto a quello di default di MsgImpl

Classi e interfacce innestate anonime

Se non serve dare un nome alle classi innestate (perché usate in un solo punto del codice della classe contenitrice) le si può definire come anonime, per compattezza. Però *per questioni di leggibilità si consiglia di definire classi anonime solo se hanno poche linee di codice.*



Vediamo come esempio **un'implementazione dell'interfaccia Iteratore** che restituisce un iteratore su una collezione (qui implementata come array di Object).

NB: la classe interna è qui definita all'interno di un metodo

Classi e interfacce innestate – non anonime

```
interface Iteratore {  
    boolean hasNext();  
    Object next(); }  
class Collezione {  
    private Object[] array;  
    public Collezione(Object[] elenco) { array = elenco; }  
    public Iteratore getIteratore() {  
        class Iter implements Iteratore {  
            private int pos = 0;  
            public boolean hasNext() { return (pos < array.length); }  
            public Object next() {  
                if (pos < array.length) {  
                    pos++; return array[pos - 1];  
                } else return null;  
            }  
            return new Iter(); }  
    }  
}
```



La definizione di Iter, con nome, è verbosa (la classe non è usata altrove). Ma poiché restituisco un Iter, non può essere una classe privata

Classi e interfacce innestate - anonime

```
class Collezione {  
    private Object[] array;  
    public Collezione(Object[] elenco) { array = elenco; }  
    public Iteratore getIteratore() {  
        return new Iteratore() {  
            private int pos = 0;  
            public boolean hasNext() {  
                return (pos < array.length); }  
            public Object next() {  
                if (pos < array.length) {  
                    pos++;  
                    return array[pos - 1];  
                } else return null;  
            } };  
    }}
```



Più sintetico del precedente. NB: **il risultato deve essere di tipo Iteratore** perché non c'è un nome di classe da usare nel return.

Lambda expressions



Le lambda expressions sono utili per implementare interface che offrono un solo metodo → non si vuole usare la sintassi verbosa delle classi anonime.

- Possono avere/non avere parametri;
- Possono restituire un risultato o essere di tipo void;

Tutto dipende dalla specifica del metodo
nell'interface.

Esempio: consideriamo l'Interface Comparator:

`public interface Comparator<T>`

che offre il metodo

`public int compare(T o1, T o2)`

Esempio con classe anonima innestata

```
import java.util.Comparator;  
public class EsempioSenzaLambda {  
    public static void main(String[] args) {  
        // con uso di classe anonima innestata  
        Comparator<String> c = new Comparator<String>() {  
            public int compare(String s1, String s2) {  
                return s1.compareTo(s2);  
            }  
        };  
        int ris = c.compare("ciao", "ciao");  
        System.out.println(ris);  
        System.out.println(c.compare("ciao1", "ciao2"));  
    }  
}
```



Esempio con uso di lambda expression – v0



```
import java.util.Comparator;
public class EsempioLambda0 {
    public static void main(String[] args) {
        Comparator<String> c = // con lambda, versione base
            (String s1, String s2) -> {return s1.compareTo(s2);};
        int ris = c.compare("ciao", "ciao");
        System.out.println(ris);
        System.out.println(c.compare("ciao1", "ciao2"));
    }
}
```

- Io ho eliminato la **new Comparator<String>** perché c ha tale tipo, quindi si inferisce automaticamente
- Io ho eliminato la **definizione del metodo compare()**, lascio solo il body

Notate la compattezza della definizione rispetto a usare la classe anonima innestata

Esempio con uso di lambda expression – v1

```
public class EsempioLambda1 {  
    public static void main(String[] args) {  
        // senza tipo dei parametri del metodo  
        Comparator<String> c =  
            (s1, s2) -> {return s1.compareTo(s2); };  
        int ris = c.compare("ciao", "ciao");  
        System.out.println(ris);  
        System.out.println(c.compare("ciao1", "ciao2"));  
    }  
}
```



Io posso omettere il tipo dei parametri del metodo perché è definito nell'interface Comparator e stretto dal tipo di c

Esempio con uso di lambda expression – v2

```
public class EsempioLambda2 {  
    public static void main(String[] args) {  
        // senza tipo dei parametri  
        Comparator<String> c =  
            // con body semplificato  
            (s1, s2) -> s1.compareTo(s2);  
        int ris = c.compare("ciao", "ciao");  
        System.out.println(ris);  
        System.out.println(c.compare("ciao1", "ciao2"));  
    }  
}
```

- Solo se il body del metodo implementato ha una sola istruzione e restituisce un valore (non è void) io posso omettere {} e anche il return
- NB: questo è sbagliato: (s1, s2) -> return 1;

Esempio con uso di lambda expression – v3

```
interface Prova {  
    public void stampa();  
}  
  
public class EsempioLambda3 {  
    public static void main(String[] args) {  
        Prova p = () -> {System.out.println("CIAO");};  
        p.stampa();  
    }  
}
```

- Come già specificato, questa lambda expression deve avere {} perché non restituisce un valore (non è una funzione)
- Se il metodo dell'interface non ha parametri io devo comunque specificare () nella lambda expression



Esempio con uso di lambda expression – v4

```
interface Prova {  
    public void stampa(String s);  
}
```



```
public class EsempioLambda4 {  
    public static void main(String[] args) {  
        Prova p =  
            messaggio -> {System.out.println(messaggio);}  
        p.stampa("CIAO!!");  
    }  
}
```

- Se il metodo dell'interface ha un solo parametro io posso omettere le () della dichiarazione del parametro
- NB: se il metodo ha più di un parametro le () della lista dei parametri non possono essere omesse

Le lambda expressions sono oggetti → posso passarle come parametri



```
interface Prova {  
    public void stampa(String s);  
}  
public class EsempioLambda5 {  
    public static void main(String[] args) {  
        Prova p = messaggio -> {System.out.println(messaggio);};  
        MyClass m = new MyClass();  
        m.metodo(p);  
    }  
}  
class MyClass {  
    void metodo(Prova p) {  
        System.out.println("Sto per richiamare il metodo stampa");  
        p.stampa("Metodo invocato!");  
    }  
}
```

Programmazione II (B)

a.a 2012/2013

Trattamento delle eccezioni

Matteo Baldoni

Dipartimento di Informatica
Università degli Studi di Torino
<http://www.di.unito.it/~baldoni>



Lezione di Programmazione II (B), a.a. 2012/2013, Corso di Studi in Informatica,
Università degli Studi di Torino by [Matteo Baldoni](#) is licensed under a
[Creative Commons Attribuzione-Non commerciale-Condividi allo stesso modo 2.5 Italia License](#).

Cos'è un'eccezione?

- Durante l'esecuzione di un programma possono verificarsi degli errori:
 - errori di programmazione
 - dati errati in ingresso
- *Interruzione dell'esecuzione!*

```
class ErroreIngresso {  
    public static void main (String[] args) {  
        String stringa;  
        int num;  
        stringa = Console.readLine();  
        num = Integer.parseInt(stringa);  
        System.out.println("Inserito: " + num);  
    }  
}
```

```
class ErroreProgrammazione {  
    static int[] a = new int[2];  
    static void p(int i, int val) {  
        a[i] = val;  
    }  
    public static void main (String[] args) {  
        p(0, 5);  
        p(2, 4);  
        p(1, 3);  
    }  
}
```

Eccezioni al “normale” flusso di esecuzione

quando l'utente inserisce ad es. “5a” invece di “5”

Eccezione: cosa fare?

- Una possibile soluzione:
il metodo potrebbe
restituire un valore che
indica il **successo** o il
fallimento
- Tale valore potrebbe essere
eventualmente utilizzato dal
chiamante per porre
rimedio alla situazione e
ripristinare l'esecuzione

```
class ErroreProgrammazione {  
    static int[] a = new int[2];  
    static bool p(int i, int val) {  
        if (a.length > i) {  
            a[i] = val;  
            return true;  
        }  
        else  
            return false;  
    }  
    public static void main (String[] args) {  
        ....  
        if (!p(2, 4))  
            System.out.println("Errore di indice!");  
        ....  
    }  
}
```

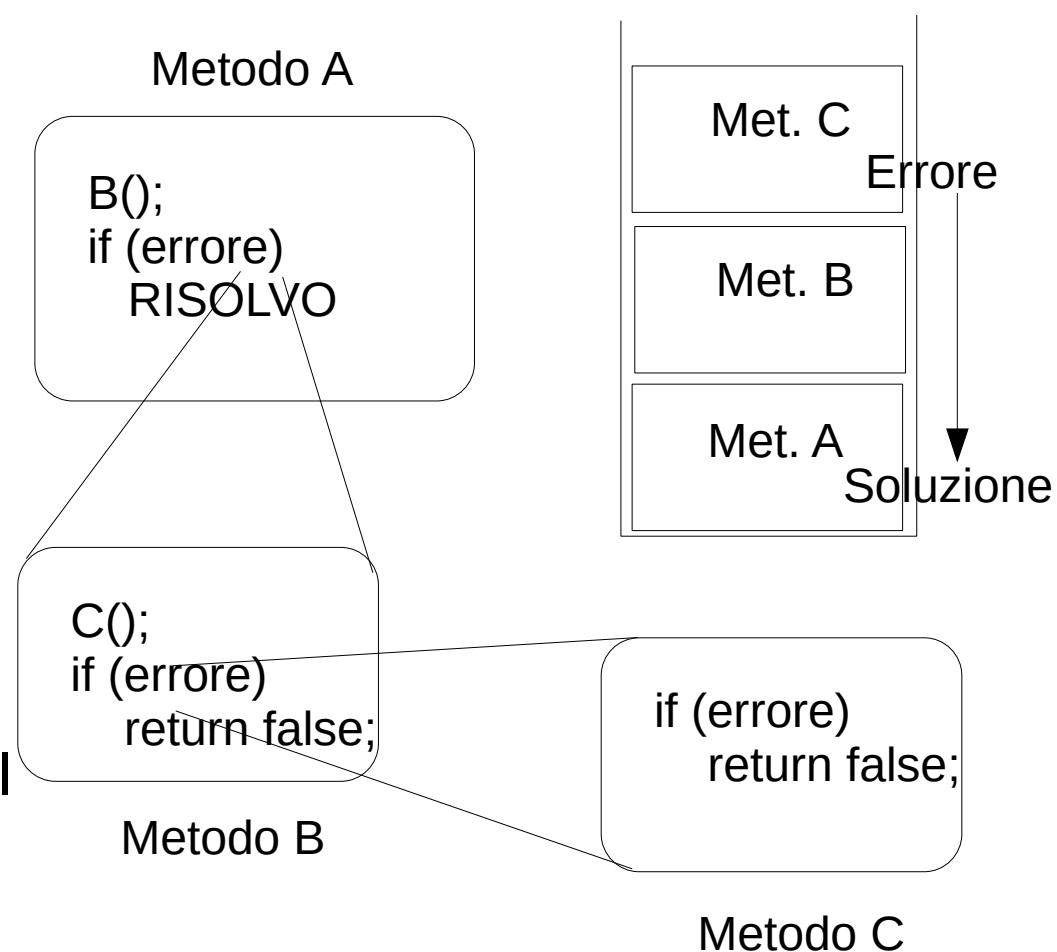
Problemi (#1, #2)

- È necessario controllare sempre il valore ricevuto: **il codice risulta meno leggibile**
- Non sempre è possibile restituire un valore di successo o fallimento, ad esempio quando **un metodo deve già restituire un valore di altro tipo**

```
class ErroreProgrammazione {  
    static int[] a = new int[2];  
    static bool p(int i, int val) {  
        if (a.length > i) {  
            a[i] = val;  
            return true;  
        }  
        else  
            return false;  
    }  
    public static void main (String[] args) {  
        if (!p(0, 5))  
            System.out.println("Errore di indice!");  
        if (!p(2, 4))  
            System.out.println("Errore di indice!");  
        if (!p(1, 3))  
            System.out.println("Errore di indice!");  
    }  
}
```

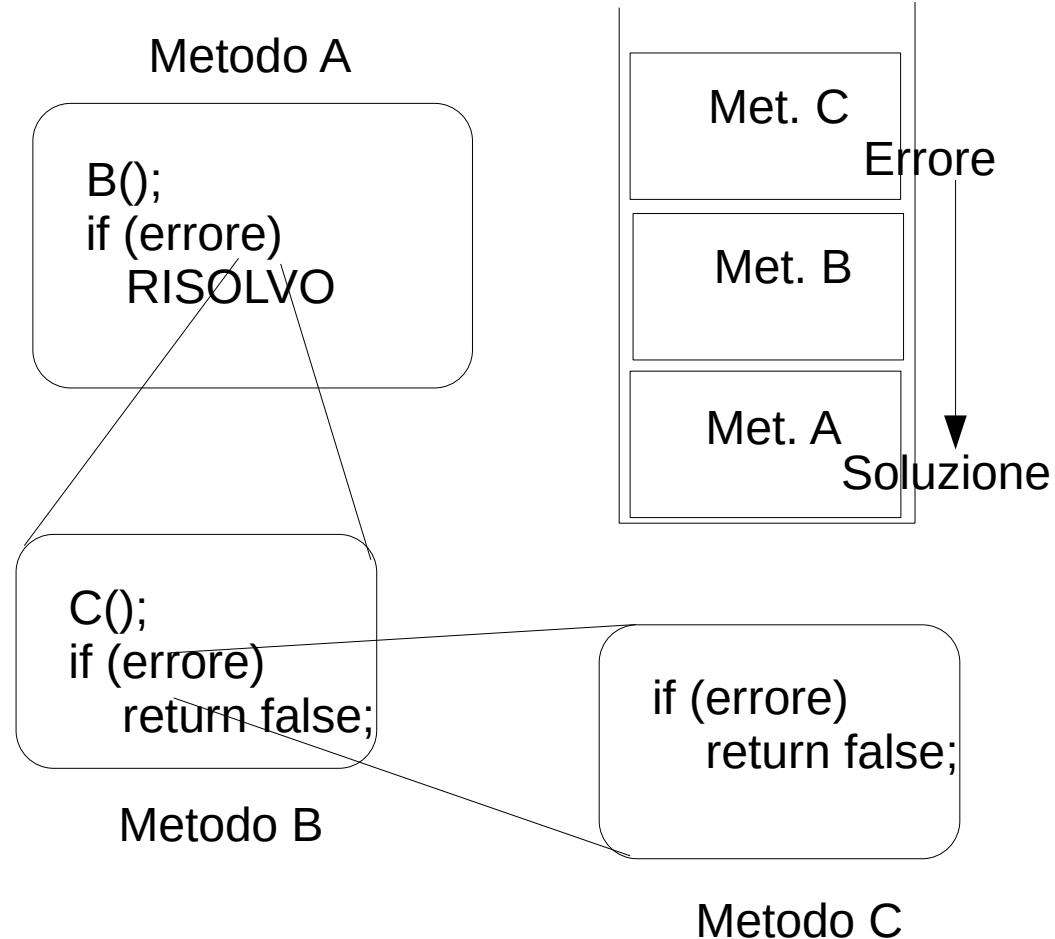
Problemi (#3)

- Non sempre si hanno sufficienti informazioni per “ripristinare” l'esecuzione lì dove l'eccezionalità si è verificata
- Potrebbe essere il caso che solo il chiamante del chiamante sia in grado di risolvere il problema
- o ancora peggio...
- Ad ogni livello dovremmo preoccuparci di restituire un valore di successo o fallimento al livello soprastante



Il meccanismo delle eccezioni

- Il **trattamento delle eccezioni** in Java offre una meccanismo **non affatto dai precedenti problemi** per affrontare condizioni “eccezionali”



throw, try e catch

- Il costrutto nel linguaggio Java per *lanciare* le eccezioni è **throw**
- Il costrutto per eseguire istruzioni che potrebbero *lanciare/sollevare* eccezioni e *catturarle* è **try ... catch**

potrebbero lanciare un'eccezione }

```
class ErroreProgrammazione {  
    static int[] a = new int[2];  
    static void p(int i, int val) {  
        if (a.length > i)  
            a[i] = val;  
        else  
            throw new  
                ArrayIndexOutOfBoundsException();  
    }  
    public static void main (String[] args) {  
        try{  
            p(0, 5);  
            p(2, 4);  
            p(1, 3);  
        }  
        catch (ArrayIndexOutOfBoundsException err) {  
            System.out.println("Errore di indice!");  
        }  
    }  
}
```

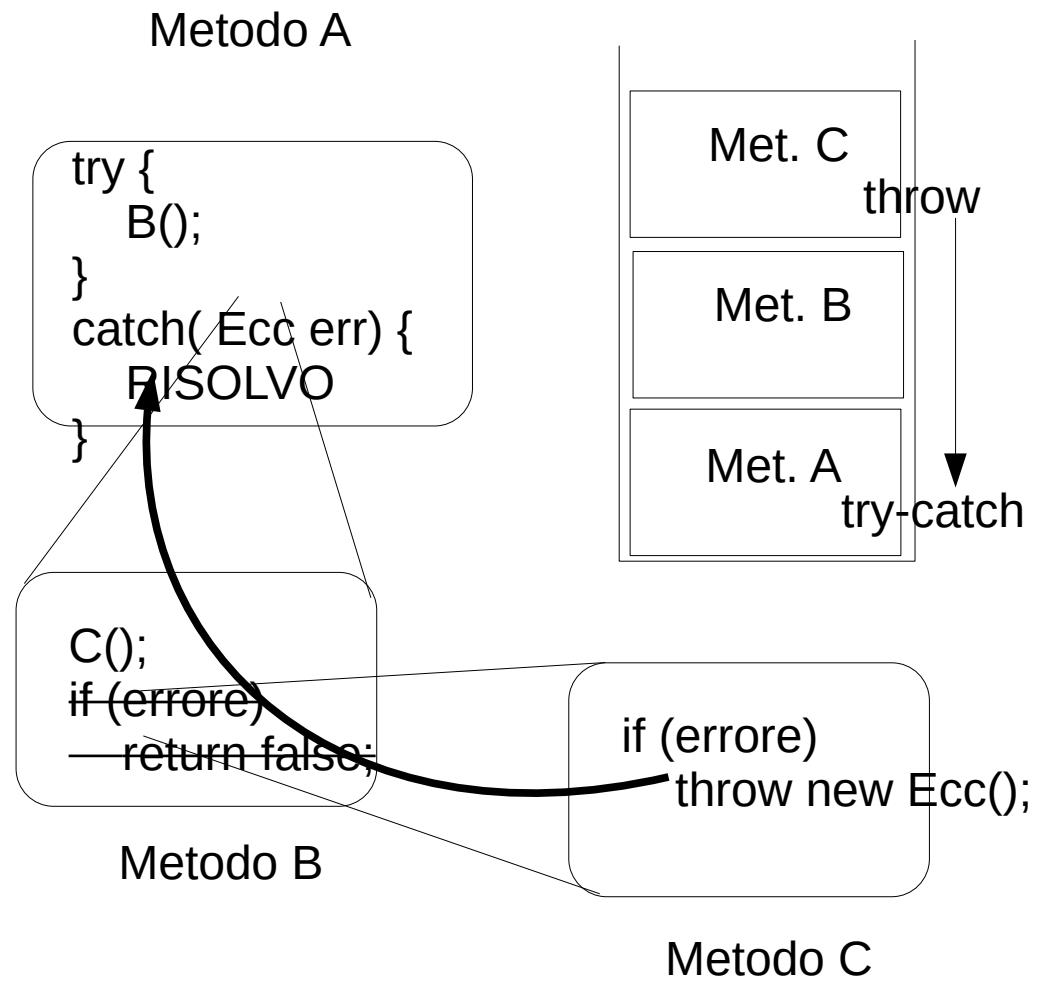
lancia un'eccezione

cattura questa eccezione

cosa fare se occorre un'eccezione

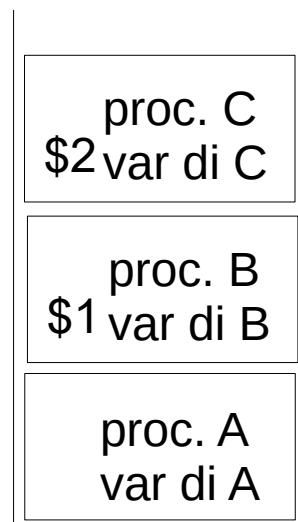
Il meccanismo delle eccezioni

- In caso di errore viene *lanciata una eccezione* (un oggetto) interrompendo il normale flusso di esecuzione
- Le eccezioni sono passate all'indietro da un metodo al suo chiamante fino a quando si trova *exception handler* che la *cattura* e la *gestisce* riprendendo l'esecuzione
- *Chi mi gestisce?* In una libreria i metodi sono utilizzati nei programmi più diversi non è possibile pensare ad una gestione locale



Eccezioni e stack dei record di attivazione

```
...  
void A() {  
    try {...  
        B(); /*$1/  
    } catch { ... }  
    ...  
    B();  
    ...  
}
```



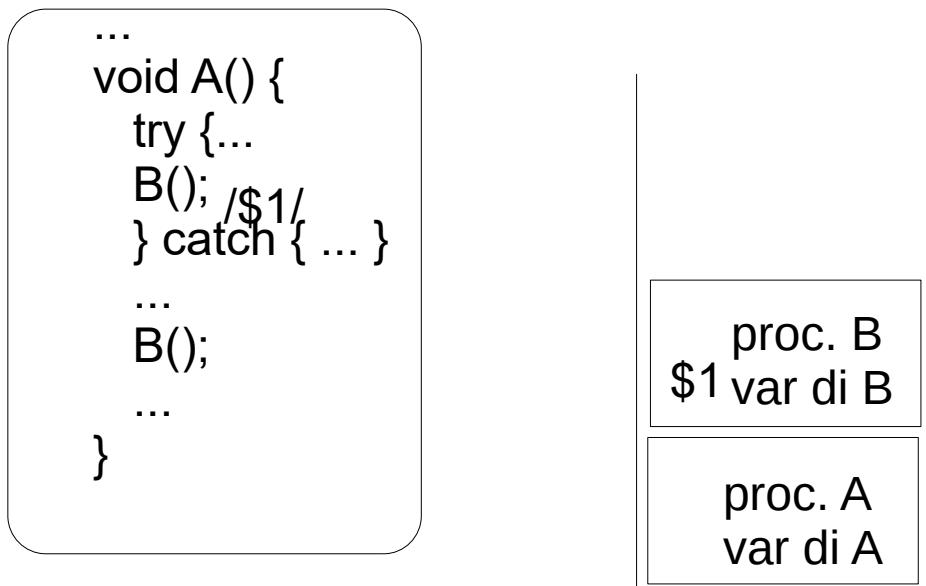
```
...  
void B() {  
    ...  
    C(); /*$2/  
    ...  
}
```

```
...  
void C() {  
    ...  
}
```

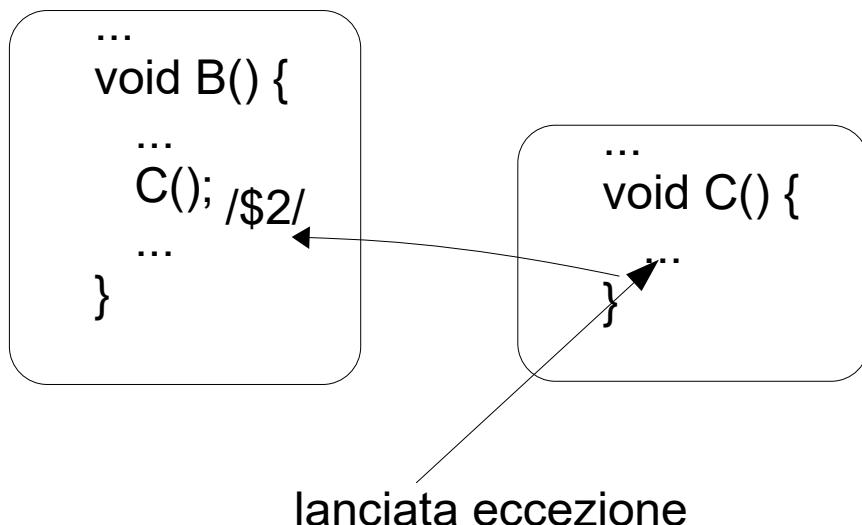
lanciata eccezione

- Le eccezioni sono passate all'indietro da un metodo al suo chiamante fino a quando si trova **exception handler** che la **cattura** e la **gestisce** riprendendo l'esecuzione

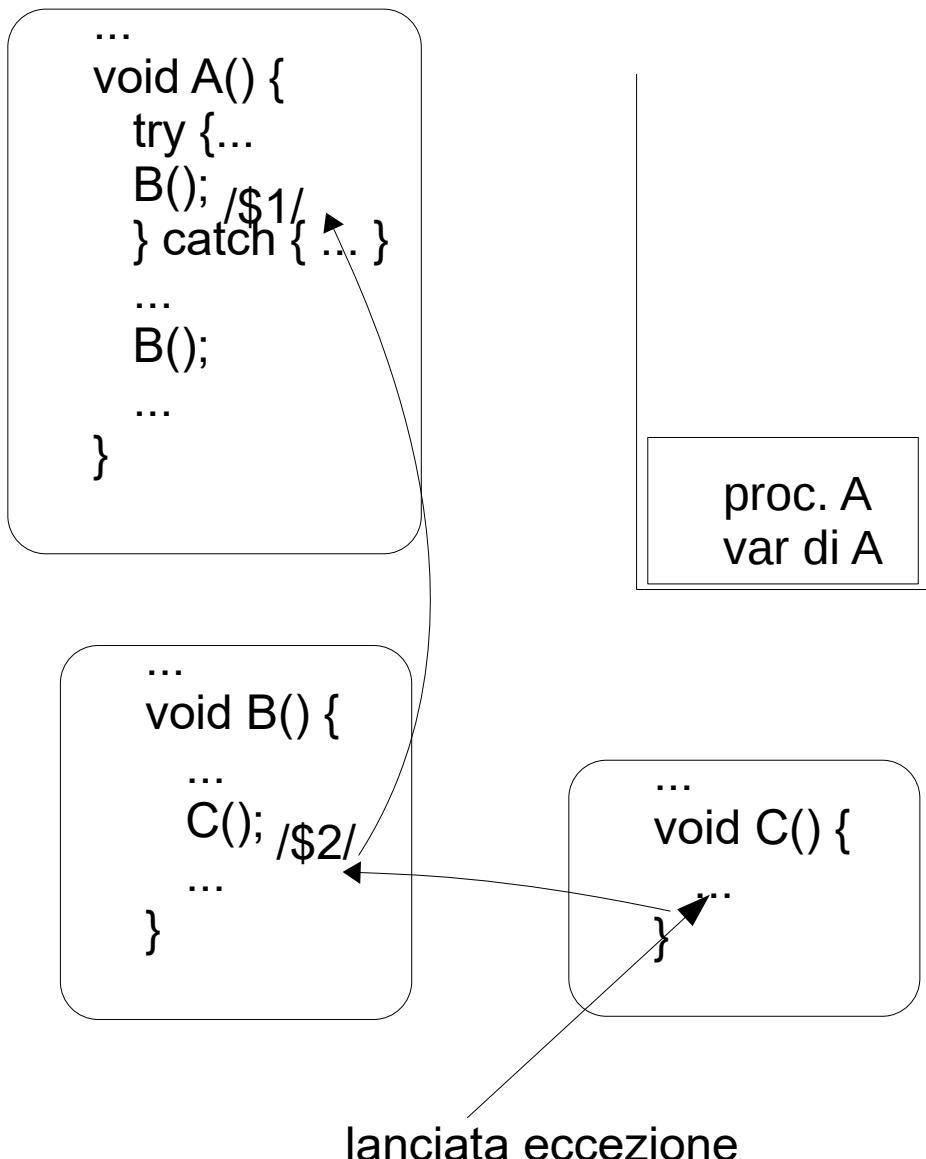
Eccezioni e stack dei record di attivazione



- Le eccezioni sono passate all'indietro da un metodo al suo chiamante fino a quando si trova **exception handler** che la **cattura** e la **gestisce** riprendendo l'esecuzione

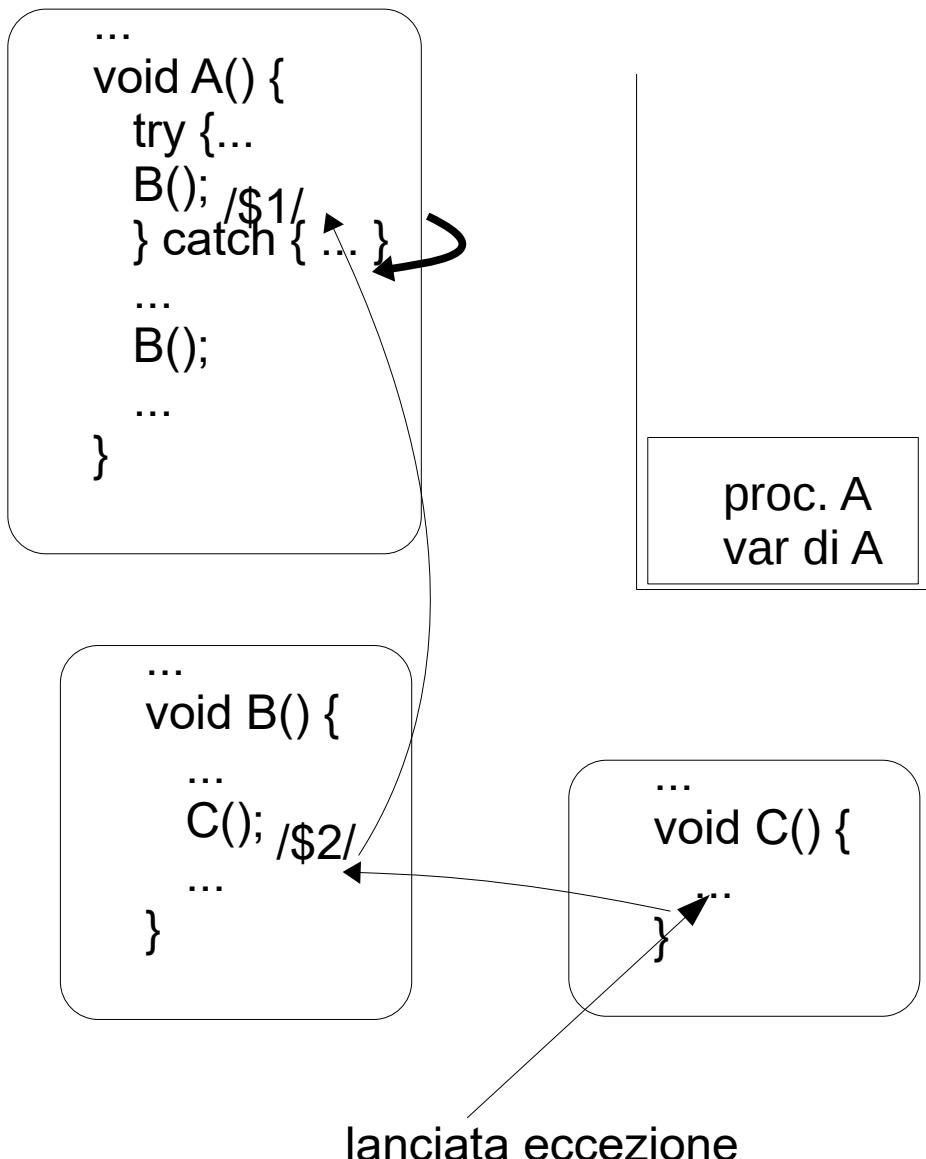


Eccezioni e stack dei record di attivazione



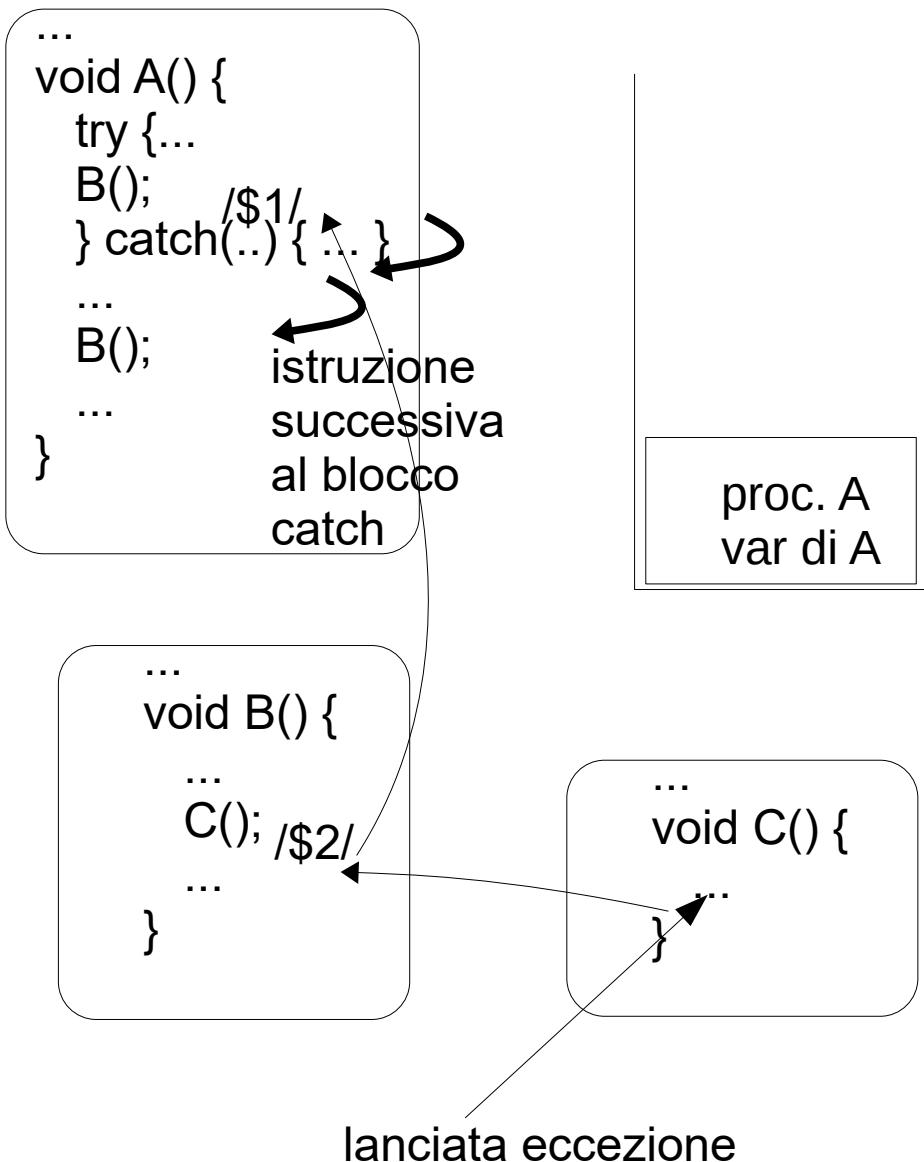
- Le eccezioni sono passate all'indietro da un metodo al suo chiamante fino a quando si trova **exception handler** che la **cattura** e la **gestisce** riprendendo l'esecuzione

Eccezioni e stack dei record di attivazione

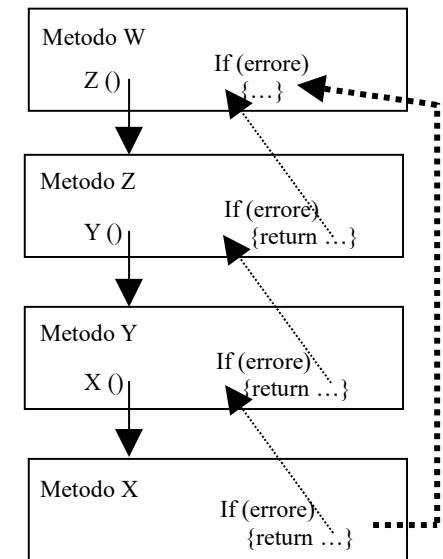


- Le eccezioni sono passate all'indietro da un metodo al suo chiamante fino a quando si trova **exception handler** che la **cattura** e la **gestisce** riprendendo l'esecuzione

Eccezioni e stack dei record di attivazione



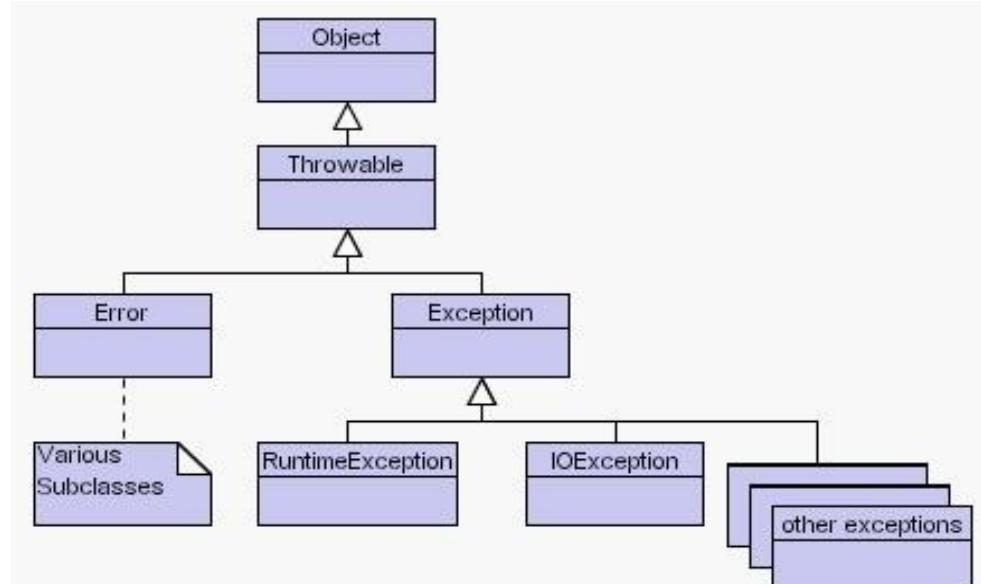
- Le eccezioni sono passate all'indietro da un metodo al suo chiamante fino a quando si trova **exception handler** che la **cattura** e la **gestisce** riprendendo l'esecuzione



Lanciare eccezioni

- Il costrutto **throw** permette di lanciare eccezioni
- Le eccezioni sono veri e propri oggetti il cui stato riporta le informazioni riguardanti la situazione in cui si è verificato errore (traccia dello stack di esecuzione)
- Gerarchia di eccezioni (il concetto di gerarchia di classi lo vedremo più avanti nel corso)

```
...
throw new xxxException();
...
```



Lanciare eccezioni

- Esempi di eccezioni presenti in Java:

- *IOException*
- *IllegalArgumentException*
- *NullPointerException*
- *NumberFormatException*
- *ArrayIndexOutOfBoundsException*

```
class ErroreIngresso {  
    public static void main (String[] args) {  
        String stringa;  
        int num;  
        stringa = Console.readLine();  
        num = Integer.parseInt(stringa);  
        System.out.println("Inserito: " + num);  
    }  
}
```

“*ArrayIndexOutOfBoundsException*”

```
class ErroreProgrammazione {  
    static int[] a = new int[2];  
    static void p(int i, int val) {  
        a[i] = val;  
    }  
    public static void main (String[] args) {  
        p(0, 5);  
        p(1, 3);  
        p(2, 4);  
    }  
}
```

“*NumberFormatException*”

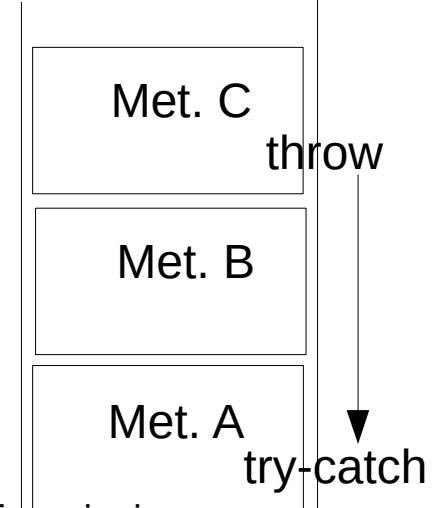
quando l'utente inserisce ad es. “5a” invece di “5”

Il gestore dell'eccezione

- Quando viene lanciata una eccezione, si interrompe l'esecuzione e *si cerca dinamicamente*, percorrendo all'indietro lo stack delle chiamate, *il primo gestore di eccezioni* che ha *una catch per il tipo dell'eccezione lanciata*, e l'esecuzione riprende da quel punto
- Se l'eccezione non viene catturata, l'interprete blocca l'esecuzione

```
...
try {
    istruzione;
    istruzione;
    istruzione;
}

} catch (Tipo err) {
    // gestisce eccezione
    // per il tipo specificato
}

...

```

try...catch: flusso di esecuzione

- Se all'interno del blocco try non si verifica alcuna eccezione

```
...  
try {  
    istruzione;  
    istruzione;  
    istruzione;  
    ...  
}  
} catch (Tipo err) {  
    // gestisce eccezione  
    // per il tipo specificato  
}  
...
```

try...catch: flusso di esecuzione

- Se all'interno del blocco try non si verifica alcuna eccezione
- L'esecuzione prosegue normalmente con l'istruzione che segue la try ... catch, **senza eseguire il codice della catch**

```
...  
try {  
    istruzione;  
    istruzione;  
    istruzione;  
    ...  
} catch (Tipo err) {  
    // gestisce eccezione  
    // per il tipo specificato  
}  
...
```

try...catch: flusso di esecuzione

- Se all'interno del blocco try si verifica una eccezione
- ***non viene eseguito il rimanente codice del blocco try***

```
...  
try {  
    istruzione;  
    istruzione; ← lancia eccezione  
di Tipo!  
    istruzione;  
    ...  
} catch (Tipo err) {  
    // gestisce eccezione  
    // per il tipo specificato  
}  
...
```

try...catch: flusso di esecuzione

- Se all'interno del blocco try si verifica una eccezione
- non viene eseguito il rimanente codice del blocco try
- Se l'eccezione sollevata è di tipo Tipo, si **esegue il codice del blocco catch**

```
...  
try {  
    istruzione;  
    istruzione;  
    istruzione;  
    ...  
} catch (Tipo err) {  
    // gestisce eccezione  
    // per il tipo specificato  
}  
...
```

lancia eccezione
di Tipo!

try...catch: flusso di esecuzione

- Se all'interno del blocco try si verifica una eccezione
- Non viene eseguito il rimanente codice del blocco try
- Se l'eccezione sollevata è di tipo Tipo, si esegue il codice del blocco catch
- e si **riprende l'esecuzione dall'istruzione che segue la try...catch**



try...catch: flusso di esecuzione

- Se all'interno del blocco try si verifica una eccezione
- Non viene eseguito il rimanente codice del blocco try
- Se l'eccezione sollevata non è di tipo Tipo, il metodo che contiene la try...catch termina immediatamente, il suo frame viene disallocato e **l'esecuzione viene passata indietro al suo chiamante**

```
...  
try {  
    istruzione;  
    istruzione;  
    istruzione;  
    ...  
} catch (Tipo err) {  
    // gestisce eccezione  
    // per il tipo specificato  
}  
...
```

lancia eccezione
di Tipo1!

Un esempio

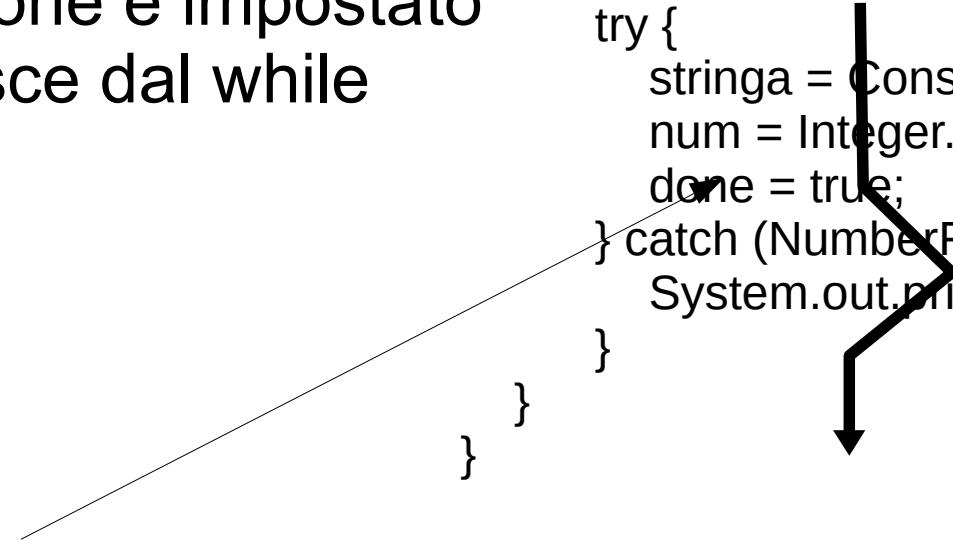
- L'esempio della lettura di un valore numerico
- Il metodo `Integer.parseInt` può lanciare l'eccezione `NumberFormatException` nel caso che la stringa passata come parametro non rappresenti un valore intero
- In caso di eccezione si desidera domandare all'utente di introdurre una nuova stringa

```
class ErroreIngresso {  
    public static void main (String[] args) {  
        String stringa;  
        int num = 0;  
        boolean done = false;  
  
        while (!done)  
            try {  
                stringa = Console.readLine();  
                num = Integer.parseInt(stringa);  
                done = true;  
            } catch (NumberFormatException err) {  
                System.out.println("Errore di immissione");  
            }  
    }  
}
```

Un esempio

- Se la stringa inserita **rappresenta un numero intero**, il blocco catch viene saltato
- e poiché done è impostato a true si esce dal while

```
class ErroreIngresso {  
    public static void main (String[] args) {  
        String stringa;  
        int num = 0;  
        boolean done = false;  
  
        while (!done)  
            try {  
                stringa = Console.readLine();  
                num = Integer.parseInt(stringa);  
                done = true;  
            } catch (NumberFormatException err) {  
                System.out.println("Errore di immissione");  
            }  
    }  
}
```



Viene eseguita solo se l'istruzione precedente non ha sollevato un'eccezione

Un esempio

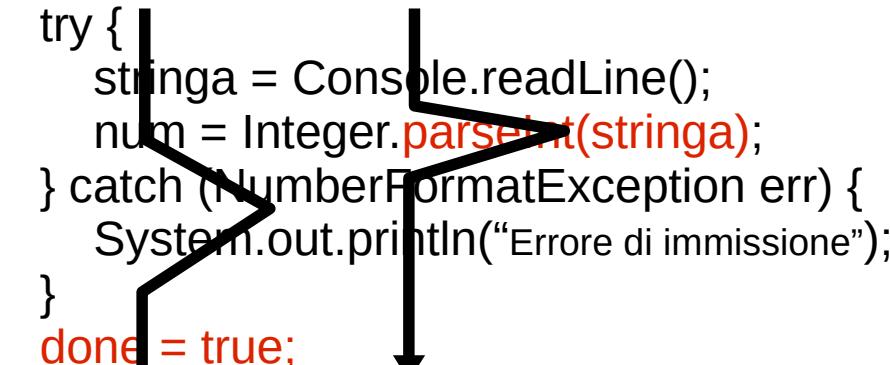
- Se, invece, la stringa inserita **non rappresenta un numero intero**, l'esecuzione è interrotta in corrispondenza all'invocazione del metodo Integer.parseInt
- e l'esecuzione prosegue all'interno del blocco della catch, lasciando a false la variabile done
- e quindi non si esce dal ciclo while

```
class ErroreIngresso {  
    public static void main (String[] args) {  
        String stringa;  
        int num = 0;  
        boolean done = false;  
  
        while (!done)  
            try {  
                stringa = Console.readLine();  
                num = Integer.parseInt(stringa);  
                done = true;  
            } catch (NumberFormatException err) {  
                System.out.println("Errore di immissione");  
            }  
    }  
}
```

Un esempio

- È fondamentale che l'istruzione “done = true” sia all'interno del blocco try
- Nell'esempio qui a fianco si esce dall'iterazione while in ogni caso, stringa che rappresenta un valore intero oppure no!
- ... con sicuri problemi in seguito

```
class ErroreIngresso {  
    public static void main (String[] args) {  
        String stringa;  
        int num = 0;  
        boolean done = false;  
  
        while (!done) {  
            try {  
                stringa = Console.readLine();  
                num = Integer.parseInt(stringa);  
            } catch (NumberFormatException err) {  
                System.out.println("Errore di immissione");  
            }  
            done = true;  
        }  
    }  
}
```



Exception handler generale

- Il blocco di finally è eseguito sempre, anche se non sono sollevate eccezioni
- Possono essere catturate eccezioni di più tipi diversi

```
try {  
    codice che può generare una eccezione  
} catch(Tipo1 id1) {  
    gestisce eccez. di Tipo1  
} catch(Tipo2 id2) {  
    gestisce eccez. di Tipo2  
}.....  
} finally {...}
```

Eccezioni controllate e non

- Eccezioni controllate (e.g. tutte le IOException)
 - associate a problemi non prevedibili, che capitano frequentemente durante esecuzione di programma bisogna specificare come gestirle (se no compilatore Java dà errore)
- Eccezioni non controllate (e.g. le RuntimeException, come ArrayIndexOutOfBoundsException)
 - associate a problemi dovuti a errori di programmazione, che dovrebbero essere prevenuti da programmatore non è necessario specificare come gestirle mediante gestore di eccezioni

Throws

- Se un metodo può lanciare un'eccezione e non la cattura, deve segnalarlo con *throws* (tranne che per RuntimeException).

```
class Generica {  
    void m(String s) throws ClassNotFoundException {  
        Class c = Class.forName(s);  
        .....  
    }  
}
```

Throws

- Se lo si desidera è possibile vedere l'eccezione come un valore alternativo a quello usuale restituito dal metodo quando occorre una qualche situazione eccezionale

return di un valore

o.metodo(...)

throw di una
eccezione

Suggerimenti di uso

- Suggerimenti per l'uso delle eccezioni (da Core Java)
- La gestione delle eccezioni non è concepita per sostituire un semplice test
- Le eccezioni non devono essere gestite in modo troppo frammentario
- Si faccia buon uso della gerarchia delle eccezioni: meglio individuare una sottoclasse appropriata di RuntimeException o crearne una nuova
- Non si devono ridurre al silenzio le eccezioni
~~try{...} catch(Exception e){}~~
- Quando si cattura un errore, è preferibile essere severi piuttosto che indulgenti
- Propagare le eccezioni non è da biasimare: spesso conviene passare una eccezione piuttosto che gestirla subito

Definizione di eccezioni

- È possibile definirsi una propria eccezione: questa potrà essere utilizzata mediante new MiaEccezione come una qualsiasi altra eccezione nota a Java e descritta nella sua documentazione

```
class MiaEccezione extends Exception  
{ ...}
```

```
class MiaEccezione extends RuntimeException  
{ ...}S
```

Un esempio

```
class Ecc1 extends RuntimeException {}  
class Ecc2 extends RuntimeException {}  
class Ecc3 extends Exception {}  
  
public class EsempioEccezioni {  
    public static void p(int i) throws Ecc3 {  
        System.out.println("istr0");  
        try {  
            if (i == 0) throw new Ecc1();  
            i--;  
            System.out.println("istr1");  
            if (i == 0) throw new Ecc2();  
            i--;  
            System.out.println("istr2");  
            if (i == 0) throw new Ecc3();  
            i--;  
            System.out.println("istr3");  
        } catch(Ecc2 e) {  
            System.out.println("istr4");  
        }  
        System.out.println("istr5");  
    }  
}
```

[...]

```
public static void main(String[] args) throws Ecc3 {  
    int i = UtilLeggiTastiera.leggiInteroPositivo  
        DaTastiera("Immetti un valore intero [0..]: ");  
    System.out.println("istr6");  
    try {  
        p(i);  
    } catch(Ecc1 e) {  
        System.out.println("istr7");  
    } catch(Ecc2 e) {  
        System.out.println("istr8");  
    }  
    System.out.println("istr9");  
}
```

Cosa succede per gli input 0, 1, 2 e 3?

Un esempio

```
class Ecc1 extends RuntimeException {}  
class Ecc2 extends RuntimeException {}  
class Ecc3 extends Exception {}  
  
public class EsempioEccezioni {  
    public static void p(int i) throws Ecc3 {  
        System.out.println("istr0");  
        try {  
            if (i == 0) throw new Ecc1();  
            i--;  
            System.out.println("istr1");  
            if (i == 0) throw new Ecc2();  
            i--;  
            System.out.println("istr2");  
            if (i == 0) throw new Ecc3();  
            i--;  
            System.out.println("istr3");  
        } catch(Ecc2 e) {  
            System.out.println("istr4");  
        }  
        System.out.println("istr5");  
    }  
}
```

```
[...]  
public static void main(String[] args) throws Ecc3 {  
    int i = UtilLeggiTastiera.leggiInteroPositivo  
    DaTastiera("Immetti un valore intero [0..]: ");  
    System.out.println("istr6");  
    try {  
        p(i);  
    } catch(Ecc1 e) {  
        System.out.println("istr7");  
    } catch(Ecc2 e) {  
        System.out.println("istr8");  
    }  
    System.out.println("istr9");  
}
```

Input: 0

Un esempio

```
class Ecc1 extends RuntimeException {}  
class Ecc2 extends RuntimeException {}  
class Ecc3 extends Exception {}
```

```
public class EsempioEccezioni {  
    public static void p(int i) throws Ecc3 {  
        System.out.println("istr0");  
        try {  
            if (i == 0) throw new Ecc1();  
            i--;  
            System.out.println("istr1");  
            if (i == 0) throw new Ecc2();  
            i--;  
            System.out.println("istr2");  
            if (i == 0) throw new Ecc3();  
            i--;  
            System.out.println("istr3");  
        } catch(Ecc2 e) {  
            System.out.println("istr4");  
        }  
        System.out.println("istr5");  
    }  
}
```

```
[...]  
public static void main(String[] args) throws Ecc3 {  
    int i = UtilLeggiTastiera.leggiInteroPositivo  
    DaTastiera("Immetti un valore intero [0..]: ");  
    System.out.println("istr6");  
    try {  
        p(i);  
    } catch(Ecc1 e) {  
        System.out.println("istr7");  
    } catch(Ecc2 e) {  
        System.out.println("istr8");  
    }  
    System.out.println("istr9");  
}
```

Input: 0

Un esempio

```
class Ecc1 extends RuntimeException {}  
class Ecc2 extends RuntimeException {}  
class Ecc3 extends Exception {}
```

```
public class EsempioEccezioni {  
    public static void p(int i) throws Ecc3 {  
        System.out.println("istr0");  
        try {  
            if (i == 0) throw new Ecc1();  
            i--;  
            System.out.println("istr1");  
            if (i == 0) throw new Ecc2();  
            i--;  
            System.out.println("istr2");  
            if (i == 0) throw new Ecc3();  
            i--;  
            System.out.println("istr3");  
        } catch(Ecc2 e) {  
            System.out.println("istr4");  
        }  
        System.out.println("istr5");  
    }  
}
```

```
[...]  
public static void main(String[] args) throws Ecc3 {  
    int i = UtilLeggiTastiera.leggiInteroPositivo  
    DaTastiera("Immetti un valore intero [0..]: ");  
    System.out.println("istr6");  
    try {  
        p(i);  
    } catch(Ecc1 e) {  
        System.out.println("istr7");  
    } catch(Ecc2 e) {  
        System.out.println("istr8");  
    }  
    System.out.println("istr9");  
}  
}  
  
Input: 0
```

Un esempio

```
class Ecc1 extends RuntimeException {}  
class Ecc2 extends RuntimeException {}  
class Ecc3 extends Exception {}
```

```
public class EsempioEccezioni {  
    public static void p(int i) throws Ecc3 {  
        System.out.println("istr0");  
        try {  
            if (i == 0) throw new Ecc1();  
            i--;  
            System.out.println("istr1");  
            if (i == 0) throw new Ecc2();  
            i--;  
            System.out.println("istr2");  
            if (i == 0) throw new Ecc3();  
            i--;  
            System.out.println("istr3");  
        } catch(Ecc2 e) {  
            System.out.println("istr4");  
        }  
        System.out.println("istr5");  
    }  
}
```

```
[...]  
public static void main(String[] args) throws Ecc3 {  
    int i = UtilLeggiTastiera.leggiInteroPositivo  
    DaTastiera("Immetti un valore intero [0..]: ");  
    System.out.println("istr6");  
    try {  
        p(i);  
    } catch(Ecc1 e) {  
        System.out.println("istr7");  
    } catch(Ecc2 e) {  
        System.out.println("istr8");  
    }  
    System.out.println("istr9");  
}
```

Input: 0

Un esempio

```
class Ecc1 extends RuntimeException {}  
class Ecc2 extends RuntimeException {}  
class Ecc3 extends Exception {}  
  
public class EsempioEccezioni {  
    public static void p(int i) throws Ecc3 {  
        System.out.println("istr0");  
        try {  
            if (i == 0) throw new Ecc1();  
            i--;  
            System.out.println("istr1");  
            if (i == 0) throw new Ecc2();  
            i--;  
            System.out.println("istr2");  
            if (i == 0) throw new Ecc3();  
            i--;  
            System.out.println("istr3");  
        } catch(Ecc2 e) {  
            System.out.println("istr4");  
        }  
        System.out.println("istr5");  
    }  
}
```

```
[...]  
public static void main(String[] args) throws Ecc3 {  
    int i = UtilLeggiTastiera.leggiInteroPositivo  
    DaTastiera("Immetti un valore intero [0..]: ");  
    System.out.println("istr6");  
    try {  
        p(i);  
    } catch(Ecc1 e) {  
        System.out.println("istr7");  
    } catch(Ecc2 e) {  
        System.out.println("istr8");  
    }  
    System.out.println("istr9");  
}
```

Input: 1

Un esempio

```
class Ecc1 extends RuntimeException {}  
class Ecc2 extends RuntimeException {}  
class Ecc3 extends Exception {}
```

```
public class EsempioEccezioni {  
    public static void p(int i) throws Ecc3 {
```

i: 0

```
        System.out.println("istr0");  
        try {  
            if (i == 0) throw new Ecc1();  
            i--;  
            System.out.println("istr1");  
            if (i == 0) throw new Ecc2();  
            i--;  
            System.out.println("istr2");  
            if (i == 0) throw new Ecc3();  
            i--;  
            System.out.println("istr3");  
        } catch(Ecc2 e) {  
            System.out.println("istr4");  
        }  
        System.out.println("istr5");  
    }
```

[...]

1 == 0

0 == 0

Input: 1

i: 1

p(1)

```
[...]  
public static void main(String[] args) throws Ecc3 {  
    int i = UtilLeggiTastiera.leggiInteroPositivo  
    DaTastiera.a("Immetti un valore intero [0..]: ");  
    System.out.println("istr6");  
    try {  
        p(i);  
    } catch(Ecc1 e) {  
        System.out.println("istr7");  
    } catch(Ecc2 e) {  
        System.out.println("istr8");  
    }  
    System.out.println("istr9");  
}
```

Un esempio

```
class Ecc1 extends RuntimeException {}  
class Ecc2 extends RuntimeException {}  
class Ecc3 extends Exception {}
```

```
public class EsempioEccezioni {  
    public static void p(int i) throws Ecc3 {
```

i: 0

```
        System.out.println("istr0");  
        try {  
            if (i == 0) throw new Ecc1();  
            i--;  
            System.out.println("istr1");  
            if (i == 0) throw new Ecc2();  
            i--;  
            System.out.println("istr2");  
            if (i == 0) throw new Ecc3();  
            i--;  
            System.out.println("istr3");  
        } catch(Ecc2 e) {  
            System.out.println("istr4");  
        }  
        System.out.println("istr5");  
    }
```

1 == 0

0 == 0

```
[...]  
    public static void main(String[] args) throws Ecc3 {  
        int i = UtilLeggiTastiera.leggiInteroPositivo  
        DaTastiera.a("Immetti un valore intero [0..]: ");  
        System.out.println("istr6");  
        try {  
            p(i);  
        } catch(Ecc1 e) {  
            System.out.println("istr7");  
        } catch(Ecc2 e) {  
            System.out.println("istr8");  
        }  
        System.out.println("istr9");  
    }
```

Input: 1

Un esempio

```
class Ecc1 extends RuntimeException {}  
class Ecc2 extends RuntimeException {}  
class Ecc3 extends Exception {}
```

```
public class EsempioEccezioni {  
    public static void p(int i) throws Ecc3 {
```

i: 0

```
        System.out.println("istr0");  
        try {  
            if (i == 0) throw new Ecc1();  
            i--;  
            System.out.println("istr1");  
            if (i == 0) throw new Ecc2();  
            i--;  
            System.out.println("istr2");  
            if (i == 0) throw new Ecc3();  
            i--;  
            System.out.println("istr3");  
        } catch(Ecc2 e) {  
            System.out.println("istr4");  
        }  
        System.out.println("istr5");  
    }
```

1 == 0

p(1)

0 == 0

Input: 1

```
[...]  
    public static void main(String[] args) throws Ecc3 {  
        int i = UtilLeggiTastiera.leggiInteroPositivo  
        DaTastiera.a("Immetti un valore intero [0..]: ");  
        System.out.println("istr6");  
        try {  
            p(i);  
        } catch(Ecc1 e) {  
            System.out.println("istr7");  
        } catch(Ecc2 e) {  
            System.out.println("istr8");  
        }  
        System.out.println("istr9");  
    }
```

Un esempio

```
class Ecc1 extends RuntimeException {}  
class Ecc2 extends RuntimeException {}  
class Ecc3 extends Exception {}
```

```
public class EsempioEccezioni {  
    public static void p(int i) throws Ecc3 {
```

i: 0

```
        System.out.println("istr0");  
        try {
```

```
            if (i == 0) throw new Ecc1();  
            i--;
```

```
            System.out.println("istr1");  
            if (i == 0) throw new Ecc2();  
            i--;
```

```
            System.out.println("istr2");  
            if (i == 0) throw new Ecc3();  
            i--;
```

```
            System.out.println("istr3");  
        } catch(Ecc2 e) {
```

```
            System.out.println("istr4");  
        }
```

```
        System.out.println("istr5");  
    }
```

[...]

1 == 0

0 == 0

i: 1

p(1)

```
[...]  
public static void main(String[] args) throws Ecc3 {  
    int i = UtilLeggiTastiera.leggiInteroPositivo  
    DaTastiera.a("Immetti un valore intero [0..]: ");  
    System.out.println("istr6");  
    try {  
        p(i);  
    } catch(Ecc1 e) {  
        System.out.println("istr7");  
    } catch(Ecc2 e) {  
        System.out.println("istr8");  
    }  
    System.out.println("istr9");  
}
```

Input: 1

Un esempio

```
class Ecc1 extends RuntimeException {}  
class Ecc2 extends RuntimeException {}  
class Ecc3 extends Exception {}  
  
public class EsempioEccezioni {  
    public static void p(int i) throws Ecc3 {  
        System.out.println("istr0");  
        try {  
            if (i == 0) throw new Ecc1();  
            i--;  
            System.out.println("istr1");  
            if (i == 0) throw new Ecc2();  
            i--;  
            System.out.println("istr2");  
            if (i == 0) throw new Ecc3();  
            i--;  
            System.out.println("istr3");  
        } catch(Ecc2 e) {  
            System.out.println("istr4");  
        }  
        System.out.println("istr5");  
    }  
}
```

```
[...]  
public static void main(String[] args) throws Ecc3 {  
    int i = UtilLeggiTastiera.leggiInteroPositivo  
        DaTastiera("Immetti un valore intero [0..]: ");  
    System.out.println("istr6");  
    try {  
        p(i);  
    } catch(Ecc1 e) {  
        System.out.println("istr7");  
    } catch(Ecc2 e) {  
        System.out.println("istr8");  
    }  
    System.out.println("istr9");  
}
```

Input: 2

Un esempio

```
class Ecc1 extends RuntimeException {}  
class Ecc2 extends RuntimeException {}  
class Ecc3 extends Exception {}
```

```
public class EsempioEccezioni {  
    public static void p(int i) throws Ecc3 {
```

i: 1

```
        System.out.println("istr0");  
        try {
```

```
            if (i == 0) throw new Ecc1();
```

```
            i--;
```

```
            System.out.println("istr1");
```

```
            if (i == 0) throw new Ecc2();
```

```
            i--;
```

```
            System.out.println("istr2");
```

```
            if (i == 0) throw new Ecc3();
```

```
            i--;
```

```
            System.out.println("istr3");
```

```
        } catch(Ecc2 e) {
```

```
            System.out.println("istr4");
```

```
}
```

```
        System.out.println("istr5");
```

```
}
```

```
[...]
```

2 == 0

1 == 0

0 == 0

[...]

```
public static void main(String[] args) throws Ecc3 {
```

```
    int i = UtilLeggiTastiera.leggiInteroPositivo
```

```
    DaTastiera.a("Immetti un valore intero [0..]: ");
```

```
    System.out.println("istr6");
```

```
    try {
```

```
        p(i);
```

```
    } catch(Ecc1 e) {
```

```
        System.out.println("istr7");
```

```
    } catch(Ecc2 e) {
```

```
        System.out.println("istr8");
```

```
}
```

```
    System.out.println("istr9");
```

```
}
```

i: 2

p(2)

Input: 2

Un esempio

```
class Ecc1 extends RuntimeException {}  
class Ecc2 extends RuntimeException {}  
class Ecc3 extends Exception {}
```

```
public class EsempioEccezioni {  
    public static void p(int i) throws Ecc3 {  
        System.out.println("istr0");  
        try {
```

i: 1

i: 0

2 == 0

1 == 0

0 == 0

```
            if (i == 0) throw new Ecc1();  
            i--;  
            System.out.println("istr1");  
            if (i == 0) throw new Ecc2();  
            i--;  
            System.out.println("istr2");  
            if (i == 0) throw new Ecc3();  
            i--;  
            System.out.println("istr3");  
        } catch(Ecc2 e) {  
            System.out.println("istr4");  
        }  
        System.out.println("istr5");  
    }
```

[...]

Input: 2

```
[...]  
    public static void main(String[] args) throws Ecc3 {  
        int i = UtilLeggiTastiera.leggiInteroPositivo  
            DaTastiera("Immetti un valore intero [0..]: ");  
        System.out.println("istr6");  
        try {  
            p(i);  
        } catch(Ecc1 e) {  
            System.out.println("istr7");  
        } catch(Ecc2 e) {  
            System.out.println("istr8");  
        }  
        System.out.println("istr9");  
    }
```

Un esempio

```
class Ecc1 extends RuntimeException {}  
class Ecc2 extends RuntimeException {}  
class Ecc3 extends Exception {}
```

```
public class EsempioEccezioni {  
    public static void p(int i) throws Ecc3 {  
        System.out.println("istr0");  
        try {  
            if (i == 0) throw new Ecc1();  
            i--;  
            System.out.println("istr1");  
            if (i == 0) throw new Ecc2();  
            i--;  
            System.out.println("istr2");  
            if (i == 0) throw new Ecc3();  
            i--;  
            System.out.println("istr3");  
        } catch(Ecc2 e) {  
            System.out.println("istr4");  
        }  
        System.out.println("istr5");  
    }  
}
```

i: 1

i: 0

2 == 0

1 == 0

0 == 0

i: 2

p(2)

```
[...]  
public static void main(String[] args) throws Ecc3 {  
    int i = UtilLeggiTastiera.leggiInteroPositivo();  
    DaTastiera.a("Immetti un valore intero [0..].");  
    System.out.println("istr6");  
    try {  
        p(i);  
    } catch(Ecc1 e) {  
        System.out.println("istr7");  
    } catch(Ecc2 e) {  
        System.out.println("istr8");  
    }  
    System.out.println("istr9");  
}
```

Input: 2

Un esempio

```
class Ecc1 extends RuntimeException {}  
class Ecc2 extends RuntimeException {}  
class Ecc3 extends Exception {}  
  
public class EsempioEccezioni {  
    public static void p(int i) throws Ecc3 {  
        System.out.println("istr0");  
        try {  
            if (i == 0) throw new Ecc1();  
            i--;  
            System.out.println("istr1");  
            if (i == 0) throw new Ecc2();  
            i--;  
            System.out.println("istr2");  
            if (i == 0) throw new Ecc3();  
            i--;  
            System.out.println("istr3");  
        } catch(Ecc2 e) {  
            System.out.println("istr4");  
        }  
        System.out.println("istr5");  
    }  
}
```

```
[...]  
public static void main(String[] args) throws Ecc3 {  
    int i = UtilLeggiTastiera.leggiInteroPositivo  
    DaTastiera("Immetti un valore intero [0..]: ");  
    System.out.println("istr6");  
    try {  
        p(i);  
    } catch(Ecc1 e) {  
        System.out.println("istr7");  
    } catch(Ecc2 e) {  
        System.out.println("istr8");  
    }  
    System.out.println("istr9");  
}
```

Input: 3

Un esempio

```
class Ecc1 extends RuntimeException {}  
class Ecc2 extends RuntimeException {}  
class Ecc3 extends Exception {}
```

```
public class EsempioEccezioni {  
    public static void p(int i) throws Ecc3 {  
        System.out.println("istr0");
```

i: 2

```
        try {  
            if (i == 0) throw new Ecc1();
```

3 == 0

i: 1

```
            i--;  
            System.out.println("istr1");  
            if (i == 0) throw new Ecc2();
```

p(3)

i: 0

```
            i--;  
            System.out.println("istr2");  
            if (i == 0) throw new Ecc3();
```

2 == 0

}

```
        } catch(Ecc2 e) {  
            System.out.println("istr4");
```

Input: 3

}

```
        System.out.println("istr5");
```

}

[...]

```
[...]  
public static void main(String[] args) throws Ecc3 {  
    int i = UtilLeggiTastiera.leggiInteroPositivo  
    DaTastiera.a("Immetti un valore intero [0..]: ");  
    System.out.println("istr6");  
    try {  
        p(i);  
    } catch(Ecc1 e) {  
        System.out.println("istr7");  
    } catch(Ecc2 e) {  
        System.out.println("istr8");  
    }  
    System.out.println("istr9");  
}
```

Un esempio

```
class Ecc1 extends RuntimeException {}  
class Ecc2 extends RuntimeException {}  
class Ecc3 extends Exception {}
```

```
public class EsempioEccezioni {  
    public static void p(int i) throws Ecc3 {  
        System.out.println("istr0");  
        try {
```

i: 2

```
            if (i == 0) throw new Ecc1();
```

3 == 0

i: 1

```
            i--;  
            System.out.println("istr1");
```

```
            if (i == 0) throw new Ecc2();
```

```
            i--;
```

```
            System.out.println("istr2");
```

2 == 0

i: 0

```
            if (i == 0) throw new Ecc3();
```

```
            i--;
```

```
            System.out.println("istr3");
```

1 == 0

}

```
        } catch(Ecc2 e) {
```

```
            System.out.println("istr4");
```

}

```
        System.out.println("istr5");
```

}

[...]

[...]

```
    } catch(Ecc1 e) {  
        System.out.println("istr7");  
    } catch(Ecc2 e) {  
        System.out.println("istr8");  
    }  
    System.out.println("istr9");  
}
```

Input: 3

Un esempio

```
class Ecc1 extends RuntimeException {}  
class Ecc2 extends RuntimeException {}  
class Ecc3 extends Exception {}
```

```
public class EsempioEccezioni {  
    public static void p(int i) throws Ecc3 {  
        System.out.println("istr0");  
        try {
```

i: 2

```
            if (i == 0) throw new Ecc1();
```

3 == 0

i: 1

```
            i--;  
            System.out.println("istr1");  
            if (i == 0) throw new Ecc2();
```

```
            i--;
```

i: 0

```
            System.out.println("istr2");  
            if (i == 0) throw new Ecc3();
```

```
            i--;
```

2 == 0

```
            System.out.println("istr3");  
        } catch(Ecc2 e) {
```

```
            System.out.println("istr4");
```

1 == 0

```
        }  
        System.out.println("istr5");
```

}

[...]

```
[...]  
public static void main(String[] args) throws Ecc3 {  
    int i = UtilLeggiTastiera.leggiInteroPositivo  
    DaTastiera.a("Immetti un valore intero [0..]: ");  
    System.out.println("istr6");  
    try {  
        p(i);  
    } catch(Ecc1 e) {  
        System.out.println("istr7");  
    } catch(Ecc2 e) {  
        System.out.println("istr8");  
    }  
    System.out.println("istr9");  
}
```

Input: 3



Programmazione III

Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

**Breve ripasso della gestione della memoria nella
macchina virtuale Java (JVM)**

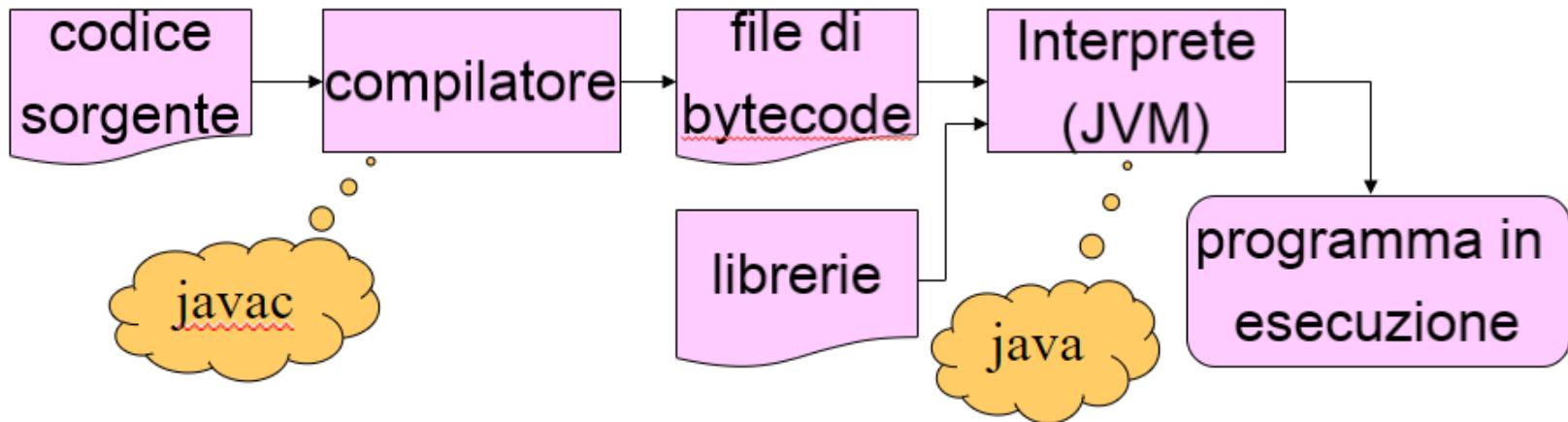


Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND

Richiami a Java



- Compilazione programmi scritti in Java
 - Dato codice sorgente: `MiaClasse.java`
 - Fornisce bytecode (eseguibile su macchine fisiche diverse, es: windows e unix) – per esempio, `MiaClasse.class`



Memoria di JVM - I



- Organizzata in 3 parti principali
 - **Memoria statica**
 - mantiene costanti e variabili statiche (variabili di tipo semplice e riferimenti a oggetti)
 - mantiene il codice delle classi
 - **Stack**
 - mantiene record di attivazione di metodi; per ciascun record di attivazione, mantiene variabili dei metodi (variabili di tipo semplice e riferimenti a oggetti)
 - gestito come una **pila** (LIFO)
 - **Heap**
 - mantiene i dati creati dinamicamente (oggetti)
 - quando i dati non sono più indirizzati dal programma il **Garbage Collector** libera la memoria da essi occupata per reciclarla. Il Garbage Collector è un programma SW che agisce in background durante l'esecuzione dei programmi affinché il programmatore non debba preoccuparsi di rilasciare le aree di memoria dismesse in modo esplicito



Classi e Metodi (statici)

```
public class Esempio {  
    public static void saluti (int n, int m) { // n, m: parametri formali  
        for (int i = 0; i < m; i++) {  
            for (int j = 0; j < n; j++)  
                System.out.print("Ciao! ");  
            System.out.println();  
        }  
    }  
    public static void main (String[] args) { // tipo di output: void  
        saluti (5,3); // anche Esempio.saluti(5, 3);  
        // 5, 3: parametri attuali  
    } //fine main  
} //fine classe
```

Esecuzione: Ciao! Ciao! Ciao! Ciao! Ciao!

Ciao! Ciao! Ciao! Ciao! Ciao!

Ciao! Ciao! Ciao! Ciao! Ciao!



Esecuzione di metodi – record di attivazione - I

Record di attivazione (o frame) di un metodo: contiene i dati necessari per gestire l'esecuzione di un metodo. L'allocazione di un record di attivazione nello stack avviene al momento in cui il metodo viene invocato.

Il record di attivazione di un metodo contiene:

- I parametri formali, inizializzati con i valori dei parametri attuali
- Le variabili locali del metodo
- Il risultato di ritorno per raccogliere il risultato dei metodi invocati dal metodo
- L'indirizzo di ritorno per effettuare correttamente il rientro dall'esecuzione di metodi richiamati dal metodo

Stack – allocazione del record di attivazione di un metodo



Avviene al momento in cui il metodo viene invocato

- Si alloca spazio al top dello stack
- NB: le variabili locali e i parametri di un metodo esistono solo durante l'esecuzione del metodo stesso (nel periodo temporale in cui il suo record di attivazione sta sullo stack)

Esecuzione di metodi – record di attivazione II situazione dello stack a inizio esecuzione

```
public class Doppio {          /* restituisce il doppio del valore in input */
    public static int raddoppia(int i) {
        int k = i * 2;
        return k; // qui si restituisce il risultato al chiamante
    }
    public static void main (String[] args) { STACK
        int x = 3;
        int y = raddoppia(x);
        int z = raddoppia(y);
        System.out.println (z);
    }
}
```



| <i>main</i> | |
|-------------|------|
| args | null |
| x | ? |
| y | ? |
| z | ? |
| risultato | ? |
| ritorno | ? |

Metodi – esecuzione - I



```
public class Doppio {  
    public static int raddoppia(int i) {  
        int k = i * 2;  
        return k;  
    }  
}
```

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

```
    int x = 3;
```

- ➡ ① int y = raddoppia(x);
- ② int z = raddoppia(y);
- System.out.println (z);
- }
- }

| <i>main</i> | |
|-------------|------|
| args | null |
| x | 3 |
| y | ? |
| z | ? |
| risultato | ? |
| ritorno | ? |

Metodi – esecuzione - II



```
public class Doppio {  
    public static int raddoppia(int i) {  
        int k = i * 2;  
        return k;  
    }  
}
```

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

- ① int y = raddoppia(x);
 - ② int z = raddoppia(y);
 - System.out.println (z);
- ```
}
}
```

STACK

| <i>raddoppia</i> |   |
|------------------|---|
| i                | 3 |
| k                | ? |
| risultato        | ? |
| ritorno          | ? |

| <i>main</i> |      |
|-------------|------|
| args        | null |
| x           | 3    |
| y           | ?    |
| z           | ?    |
| risultato   | ?    |
| ritorno     | ❶    |

# Metodi – esecuzione - III



```
public class Doppio {
 public static int raddoppia(int i) {
 int k = i * 2;
 return k;
 }
}
```

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

- ① int y = raddoppia(x);
  - ② int z = raddoppia(y);
  - System.out.println (z);
- ```
}  
}
```

STACK

| <i>raddoppia</i> | |
|------------------|---|
| i | 3 |
| k | 6 |
| risultato | ? |
| ritorno | ? |

| <i>main</i> | |
|-------------|------|
| args | null |
| x | 3 |
| y | ? |
| z | ? |
| risultato | ? |
| ritorno | ❶ |

Metodi – esecuzione - IV



```
public class Doppio {  
    public static int raddoppia(int i) {  
        int k = i * 2;  
        return k;  
    }  
}
```

```
public static void main (String[] args) {  
    int x = 3;  
    ➔ ① int y = raddoppia(x);  
    ② int z = raddoppia(y);  
    System.out.println (z);  
}  
}
```

STACK

| <i>main</i> | |
|-------------|------|
| args | null |
| x | 3 |
| y | ? |
| z | ? |
| risultato | 6 |
| ritorno | ① |

Metodi – esecuzione - V



```
public class Doppio {  
    public static int raddoppia(int i) {  
        int k = i * 2;  
        return k;  
    }  
}
```

```
public static void main (String[] args) {  
    int x = 3;  
    ❶ int y = raddoppia(x);  
    ➔ ❷ int z = raddoppia(y);  
    System.out.println (z);  
}  
}
```

STACK

| main | |
|-----------|------|
| args | null |
| x | 3 |
| y | 6 |
| z | ? |
| risultato | 6 |
| ritorno | ❸ |

Metodi – esecuzione - VI



```
public class Doppio {  
    public static int raddoppia(int i) {  
        int k = i * 2;  
        return k;  
    }  
}
```

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

- ① int y = raddoppia(x);
 - ② int z = raddoppia(y);
 - System.out.println (z);
- ```
}
}
```

STACK

| <i>raddoppia</i> |   |
|------------------|---|
| i                | 6 |
| k                | ? |
| risultato        | ? |
| ritorno          | ? |

| <i>main</i> |      |
|-------------|------|
| args        | null |
| x           | 3    |
| y           | 6    |
| z           | ?    |
| risultato   | 6    |
| ritorno     | ❷    |

# Metodi – esecuzione - VII



```
public class Doppio {
 public static int raddoppia(int i) {
 int k = i * 2;
 return k;
 }

 public static void main (String[] args) {
 int x = 3;

 ① int y = raddoppia(x);
 ② int z = raddoppia(y);
 System.out.println (z);
 }
}
```

## STACK

| <i>raddoppia</i> |    |
|------------------|----|
| i                | 6  |
| k                | 12 |
| risultato        | ?  |
| ritorno          | ?  |

| <i>main</i> |      |
|-------------|------|
| args        | null |
| x           | 3    |
| y           | 6    |
| z           | ?    |
| risultato   | 6    |
| ritorno     | ②    |

# Metodi – esecuzione - VIII



```
public class Doppio {
 public static int raddoppia(int i) {
 int k = i * 2;
 return k;
 }
 public static void main (String[] args) {
 int x = 3;
 ① int y = raddoppia(x);
 ➔ ② int z = raddoppia(y);
 System.out.println (z);
 }
}
```

| main         |      |
|--------------|------|
| args         | null |
| x            | 3    |
| y            | 6    |
| z            | ?    |
| risultato 12 |      |
| ritorno      |      |

# Metodi – esecuzione - IX



```
public class Doppio {
 public static int raddoppia(int i) {
 int k = i * 2;
 return k;
 }
 public static void main (String[] args) {
 int x = 3;
 ① int y = raddoppia(x);
 ② int z = raddoppia(y);
 ➔ System.out.println (z);
 }
}
```

| main      |      |
|-----------|------|
| args      | null |
| x         | 3    |
| y         | 6    |
| z         | 12   |
| risultato | 12   |
| ritorno   | ②    |

# Metodi – passaggio di parametri di tipo oggetto - I

**Riferimenti come parametri:** si possono passare riferimenti ad oggetti, come gli array, come parametro (l'indirizzo viene copiato nel parametro, come valore). In tal caso, il metodo opera direttamente sull'elemento a cui il riferimento punta (l'array) e lo può modificare. Esempio:

```
public class ProvaParametri {

 public static void modifica(int[] b) {
 for (int i = 0; i < b.length; i++) {
 b[i] = i+3; } }

 public static visualizza(int[] b) {
 for (int i = 0; i < b.length; i++) {
 System.out.println(b[i]); } }
}
```



# Metodi – passaggio di parametri oggetto - II

```
public static void main (String[] args) {
 int[] a = new int [5];
 for (int i = 0; i<a.length; i++)
 a[i] = 0;
modifica(a); // qui il metodo modifica l'array
 // a passato come parametro
visualizza(a); // qui il metodo accede
 // all'array per visualizzare dati
}
}
```



| <i>esecuzione</i> |
|-------------------|
| 3                 |
| 4                 |
| 5                 |
| 6                 |
| 7                 |

# Metodi – passaggio di parametri oggetto - III

```
public class ProvaParametri {

 public static void modifica(int[] b) {
 for (int i = 0; i < b.length; i++) {
 b[i] = i+3; } }

 public static visualizza(int[] b) {
 for (int i = 0; i < b.length; i++) {
 System.out.println(b[i]); } }

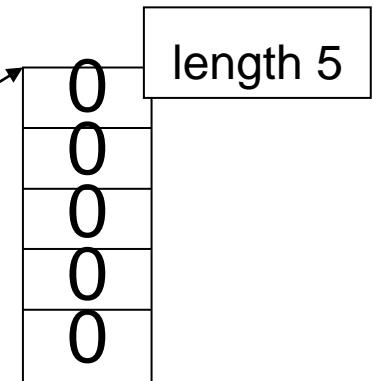
 public static void main (String[] args) {
 int[] a = new int [5];
 for (int i = 0; i<a.length; i++)
 a[i] = 0;
 modifica(a);
 visualizza(a); }
}
```



STACK

| main      |      |
|-----------|------|
| args      | null |
| i         | 5    |
| a         |      |
| risultato | ?    |
| ritorno   | ?    |

HEAP



# Metodi – passaggio di parametri oggetto - IV



```
public class ProvaParametri {

 public static void modifica(int[] b) {
 for (int i = 0; i < b.length; i++) {
 b[i] = i+3; } }

 public static visualizza(int[] b) {
 for (int i = 0; i < b.length; i++) {
 System.out.println(b[i]); } }

 public static void main (String[] args) {
 int[] a = new int [5];
 for (int i = 0; i<a.length; i++)
 a[i] = 0;

 ① modifica(a);
 visualizza(a); }
}
```

## STACK

|                 |   |
|-----------------|---|
| <i>modifica</i> |   |
| b               |   |
| i               | 3 |
| risultato       | ? |
| ritorno         | ? |

## HEAP

|             |      |
|-------------|------|
| <i>main</i> |      |
| args        | null |
| i           | 5    |
| a           |      |
| risultato   | ?    |
| ritorno     | OE   |

length 5

|   |
|---|
| 3 |
| 4 |
| 5 |
| 0 |
| 0 |

# Metodi – passaggio di parametri oggetto - V

```
public class ProvaParametri {

 public static void modifica(int[] b) {
 for (int i = 0; i < b.length; i++) {
 b[i] = i+3; } }

 public static visualizza(int[] b) {
 for (int i = 0; i < b.length; i++) {
 System.out.println(b[i]); } }

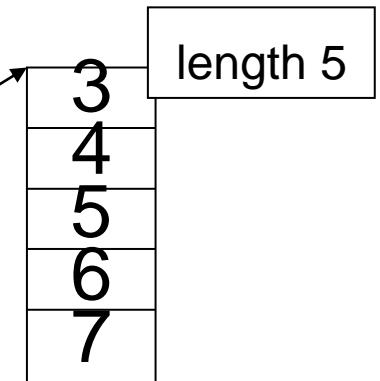
 public static void main (String[] args) {
 int[] a = new int [5];
 for (int i = 0; i<a.length; i++)
 a[i] = 0;
 modifica(a);
 visualizza(a); }
}
```



STACK

| main      |      |
|-----------|------|
| args      | null |
| i         | 5    |
| a         |      |
| risultato | ?    |
| ritorno   | OE   |

HEAP



# Variabili statiche e di istanza



- **Variabili di istanza:** servono per memorizzare lo stato degli oggetti – ogni oggetto ha la sua copia della variabile di istanza nella heap (nell'area di memoria dedicata all'oggetto)
- **Variabili di classe, o statiche (static):** c'e' una unica copia della variabile per la classe ed e' condivisa tra tutti gli oggetti.

Consideriamo i punti nello spazio cartesiano, rappresentati in una applicazione dalla classe Point. Ogni oggetto Point ha le proprie coordinate, che devono essere associate solo a quel punto. Invece, se considero per esempio il numero di punti (istanze) create a partire dalla classe Point, questa deve essere una variabile static per permettere di incrementare il suo valore correttamente ogni volta che l'applicazione crea un nuovo oggetto Point. Vedere prossimo esempio.

numPoints: 3

Point: (125, 34)

Point: (-30, 45)

Point: (20, -10)

# Esempio – stack e heap - I



```
public class Point {
 private static int numPoints = 0; // numPoints memorizza quanti punti
 // sono stati creati. La dichiaro private per evitare violazioni del suo
 private int x; // valore (information hiding)
 private int y;
 public Point(int x, int y) {
 this.x = x; // notare il this per indicare la variabile di istanza
 this.y = y; // e distinguerla dal parametro (omonimo) del metodo
 numPoints++; // memorizzo che c'e' una nuova istanza di punto
 // Non si può scrivere this.numPoints!! E'una variabile statica!!!
 }
 public static getNumPoints() { return numPoints; }
 public int getX() { return x; }
 public int getY() { return y; }
 public boolean equals(Point pt) {
 return x==pt.x && y==pt.y; }
}
```

# Esempio – stack e heap - II



```
public class PointApp {
 public static void main(String[] args) {
 Point p = new Point(3, 4);
 Point q = new Point(2, 5);

 System.out.println("Ho creato n. " +
 Point.getNumPoints() + " oggetti Point!");

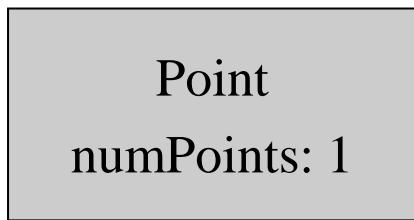
 p.equals(q); ← Invoca il metodo equals() di Point
 sull'oggetto a cui fa riferimento p.
 Si passa q come parametro attuale
 del metodo equals()
 }
}
```

# Esempio – stack e heap - III

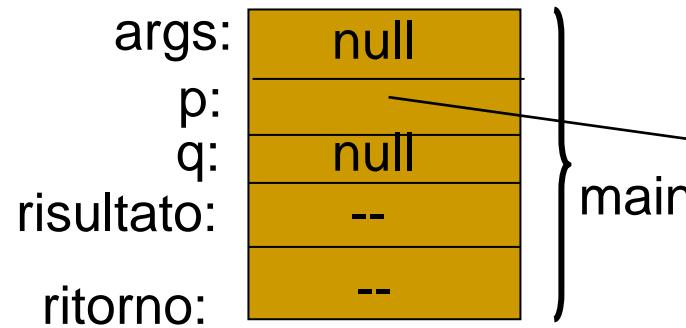


Subito dopo la creazione del primo oggetto Point:  
Point p = new Point(3,4);

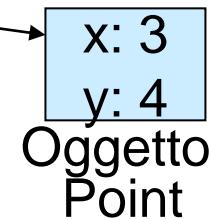
## Memoria statica



## Stack



## Heap



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

```
 Point p = new Point(3, 4);
 Point q = new Point(2, 5);
 System.out.println("Ho creato n. " +
 Point.getNumPoints() + " oggetti Point!");
 $ p.equals(q);
}
```

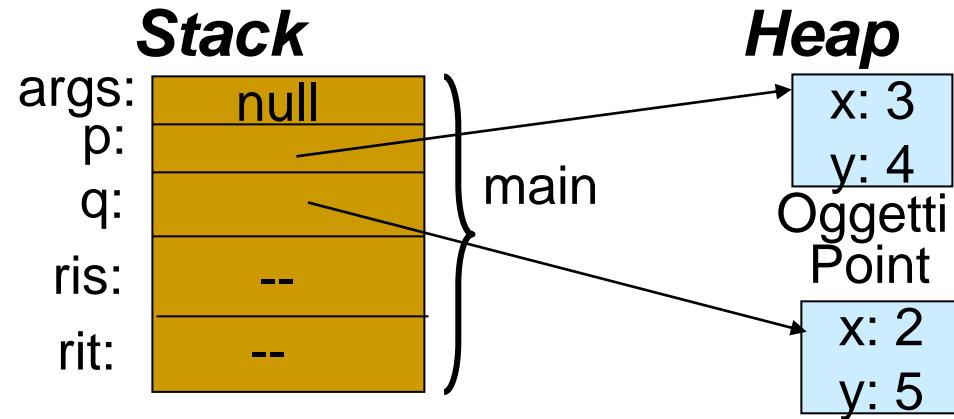
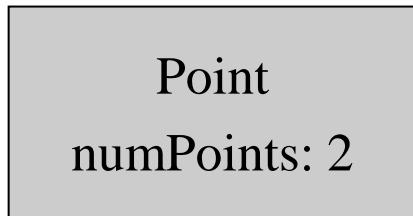
# Esempio – stack e heap - IV



Subito dopo la creazione del secondo oggetto Point:

```
Point q = new Point(2,5);
```

## Memoria statica



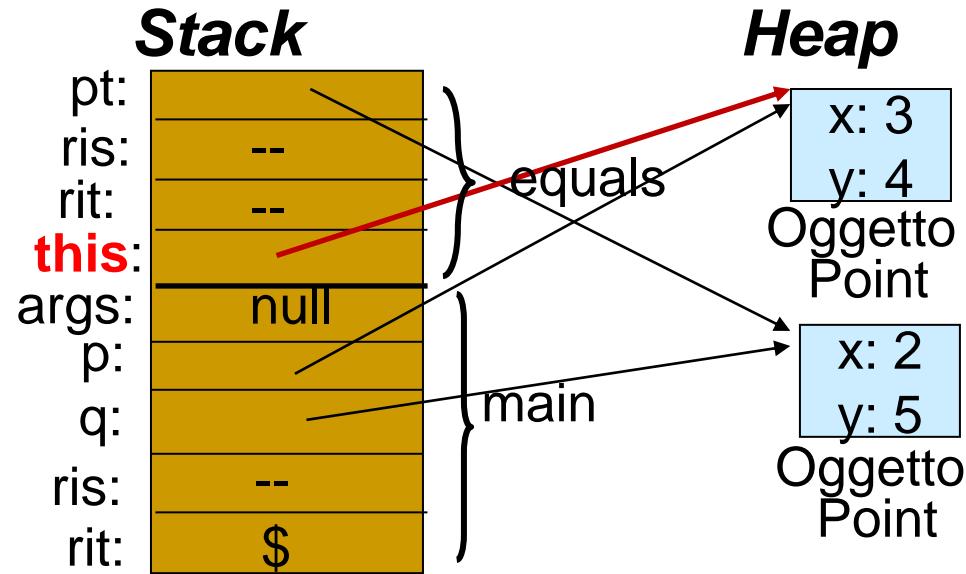
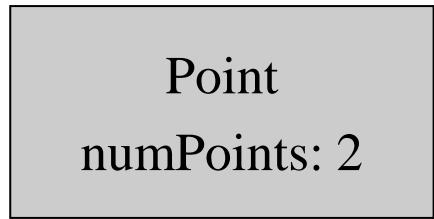
```
public static void main(String[] args) {
 Point p = new Point(3, 4);
 Point q = new Point(2, 5);
 System.out.println("Ho creato n. " +
 Point.getNumPoints() + " oggetti Point!");
 $ p.equals(q);
}
```



# Esempio – stack e heap - V

Al momento della valutazione dell'argomento di  
System.out.println(p.equals(q)):

## Memoria statica



```
public static void main(String[] args) {
 Point p = new Point(3, 4);
 Point q = new Point(2, 5);
 System.out.println("Ho creato n. " +
 Point.getNumPoints() + " oggetti Point!");

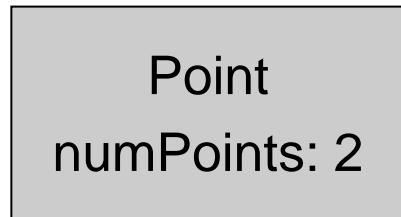
 $ p.equals(q);
}
```



# Esempio – stack e heap - VI

Una volta terminato p.equals(q);

## Memoria statica



## Stack

args:  
p:  
q:  
ris:  
rit:

null  
\_\_\_\_\_  
false  
\$

## Heap

} main

Oggetto  
Point

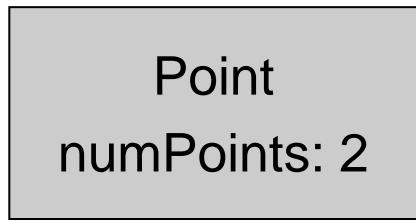
Oggetto  
Point

# Esempio – stack e heap - VII

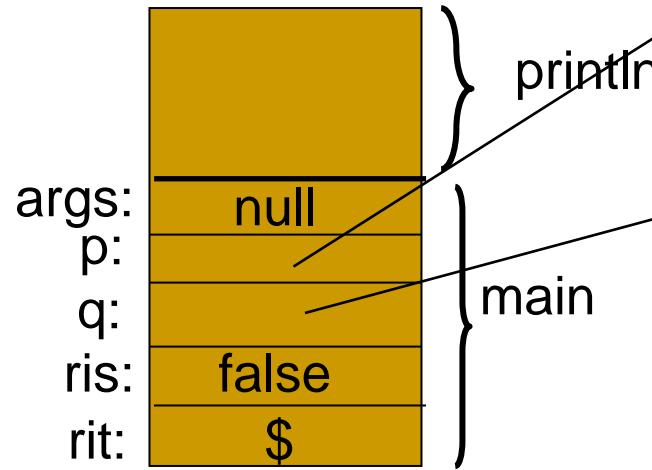


NB: Durante la System.out.println();

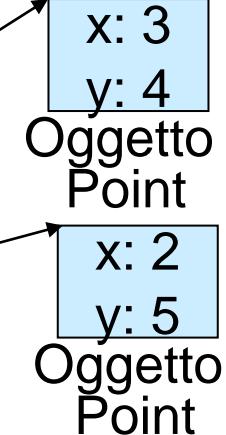
## Memoria statica



## Stack



## Heap



```
public static void main(String[] args) {
 Point p = new Point(3, 4);
 Point q = new Point(2, 5);
 System.out.println("Ho creato n. " +
 Point.getNumPoints() + " oggetti Point!");
 $ p.equals(q);
}
```

# Esempio – stack e heap - VIII

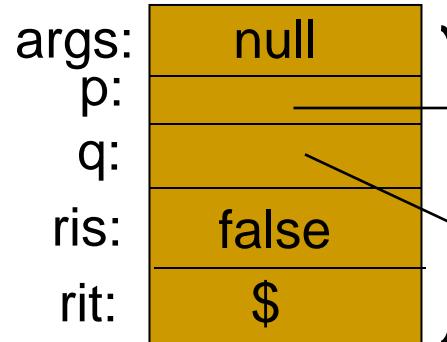


Come sarebbe la memoria se numPoints fosse una variabile di istanza?

## Memoria statica

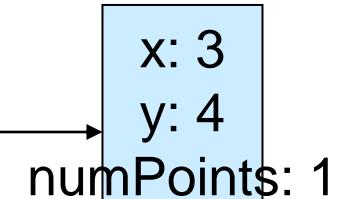


## Stack

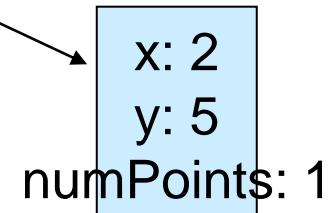


main

## Heap



Oggetto Point



Oggetto Point



# Programmazione III

Prof.ssa Liliana Ardissono  
Dipartimento di Informatica  
Università di Torino

## Input/Output

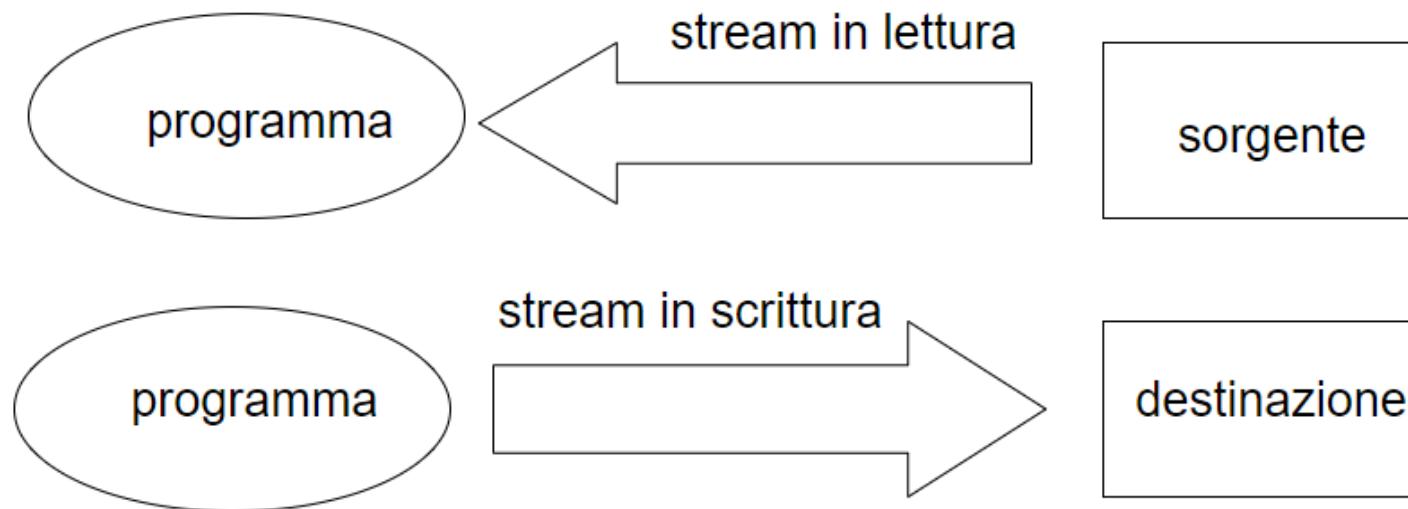


Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND



Le operazioni di I/O avvengono attraverso **stream**

**Flussi o streams**: sono successioni di informazioni in forma elementare, principalmente sequenze di bytes. I files sono un caso particolare di flussi.



Sorgente e destinazione possono essere di vario genere (file, connessioni di rete, ...) ma vengono gestite essenzialmente nello stesso modo.



Java fornisce molte classi per I/O (circa 60).

Tutti i file sono memorizzati come sequenze di bit (0/1). Tuttavia in alcune situazioni si interpreta il contenuto di un file come sequenza di caratteri (file di testo), in altre come sequenza di byte (file binari).

Divise in due gerarchie:

- basate su caratteri Unicode (2 byte) (derivate dalle classi astratte **Reader** e **Writer**)
- basate su byte (derivate dalle classi astratte **InputStream** e **OutputStream**)



Tutti i file sono memorizzati come sequenze di bit (0/1). Nei file di testo la sequenza di bit viene vista come sequenza di caratteri UNICODE, mentre in quelli binari come sequenza di byte. Per es. consideriamo il carattere 5 e il numero 5

Dato il valore (numero) 5:

- Rappresentazione del valore 5 in un file di testo:
  - 00000000 00110101 (corrisponde al 5 UNICODE)
  - Esempio: Laura ha 5 anni
- Rappresentazione del valore 5 in un file binario:
  - 101
  - Esempio:  $5+5=10$



Le classi di I/O possono essere classificate in base allo scopo:

- tipo di sorgente/destinazione
- tipo di elaborazione sui dati

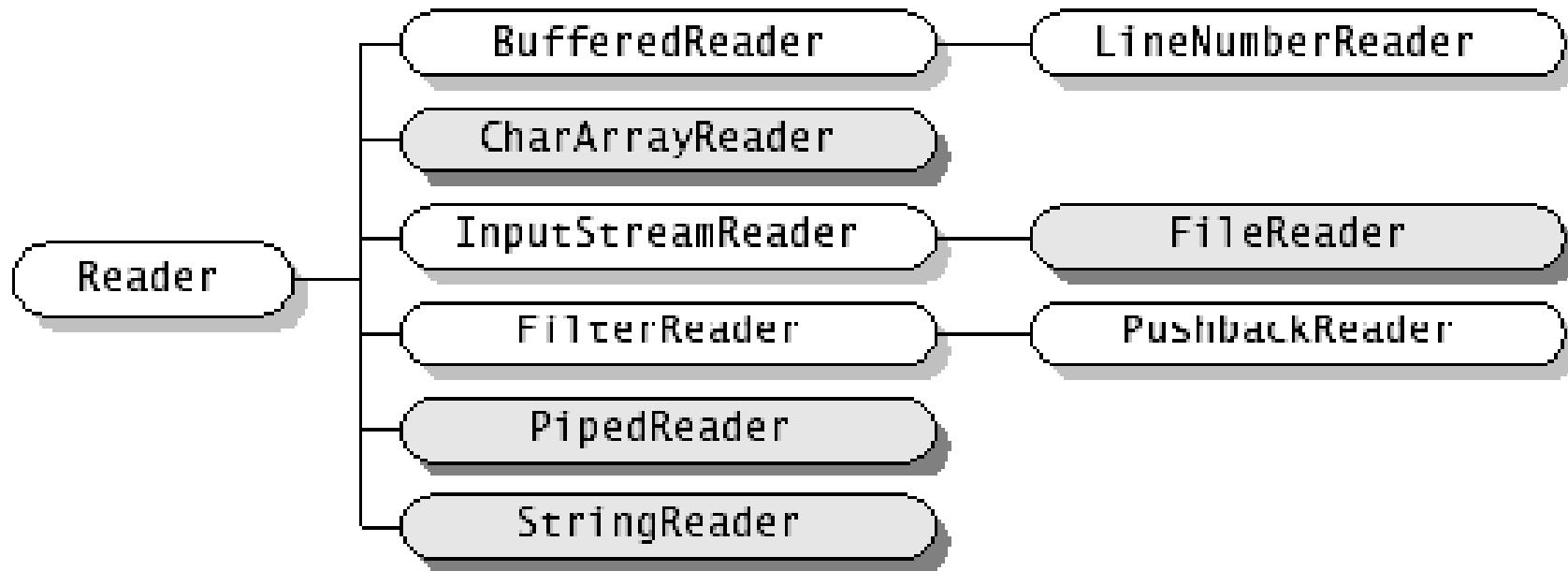
**Sorgente o destinazione** possono essere di vario tipo

- memoria (array o stringhe),
- *file*
- *pipe* (per collegare due programmi - thread),
- connessioni via socket o internet.

# Input a caratteri: Reader è una classe astratta.

La classe **Reader** ha metodi per leggere un carattere per volta:

**abstract int read()**



In grigio gli *stream* che leggono da una sorgente.

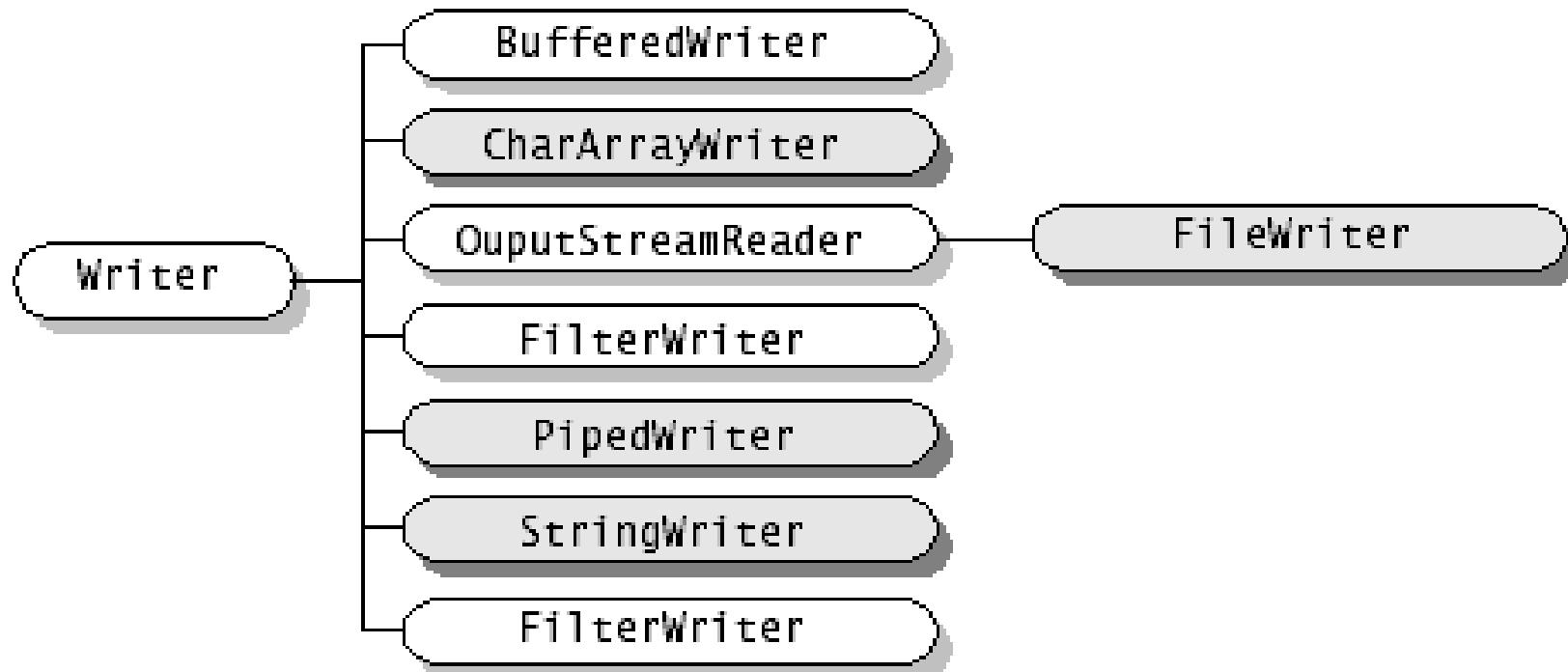
In bianco quelli che eseguono qualche tipo di elaborazione.



# Output a caratteri: Writer è una classe astratta.

Writer può scrivere un carattere per volta:

```
abstract void write(int c)
```





# Uso dei buffer per leggere e scrivere dati su file

Per ottimizzare i tempi di accesso ad una risorsa è opportuno leggere e scrivere molti caratteri in una sola volta e memorizzarli in un buffer, in modo che la prossima lettura/scrittura prelevi/scriva il carattere dal buffer anziché richiedere un nuovo accesso alla risorsa

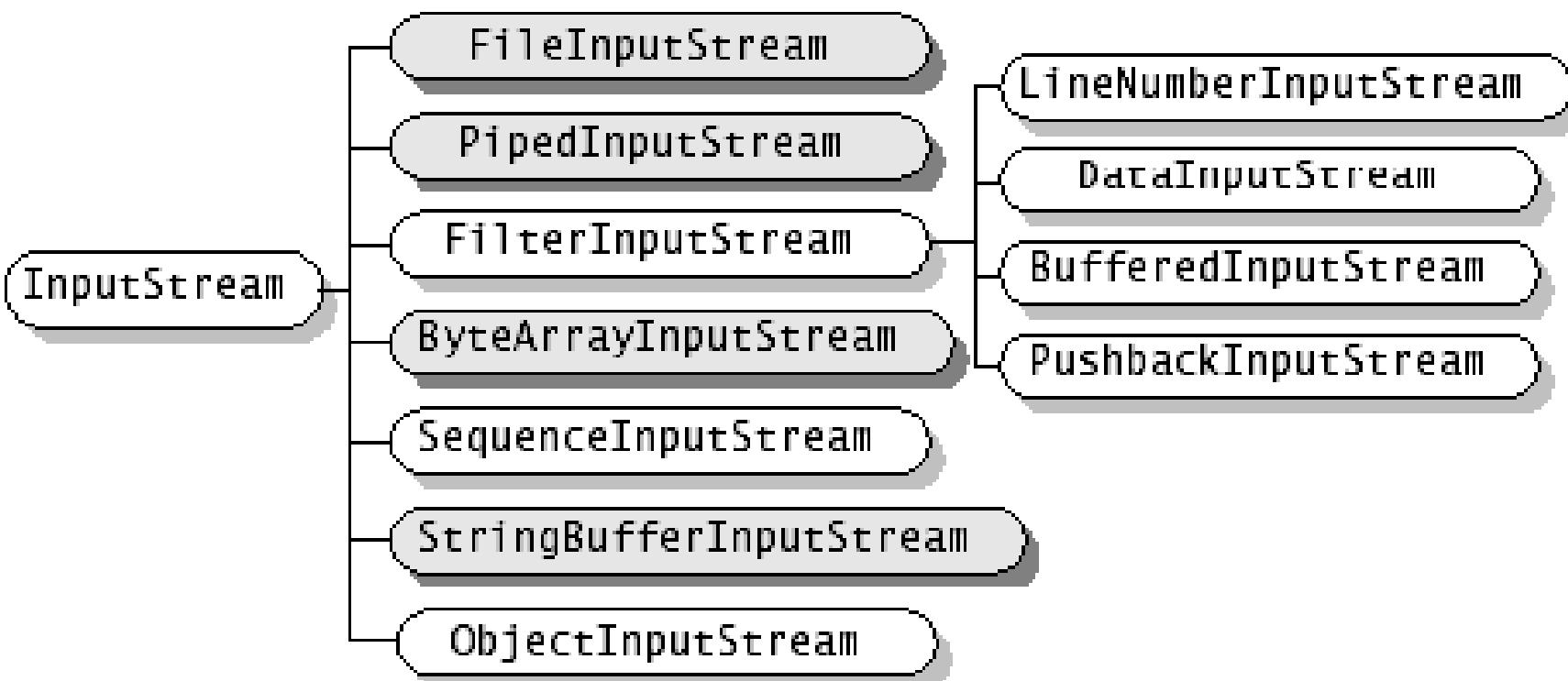
→ introdotte le classi **BufferedReader** e **BufferedWriter** per usare buffer in lettura e scrittura (e analogamente esistono classi di buffering per la lettura/scrittura di byte)



# Input a byte: InputStream è una classe astratta.

InputStream può leggere un byte per volta:

**abstract void read(int c)**

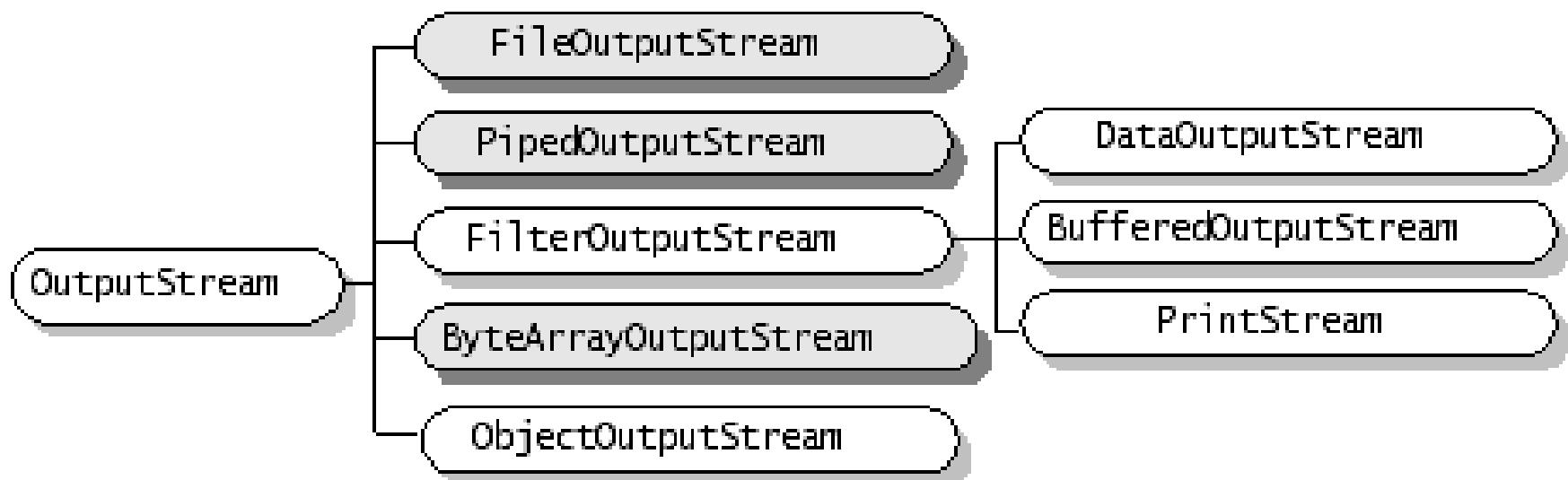




# Output a byte: OutputStream è una classe astratta.

OutputStream può scrivere un byte per volta:

```
abstract void write(int c)
```





# Flussi standard

I flussi standard (tastiera e video) per motivi storici sono flussi di byte:

- **System.in** ha tipo `InputStream`
- **System.out** ha tipo `PrintStream`, che è sottotipo di `OutputStream`
- **System.err** ha tipo `PrintStream`

# Input e output da file (a caratteri)



La classe **FileReader** può essere usata per leggere da file, specificando il nome del file nel costruttore:

```
FileReader in = new FileReader("esempio.txt");
```

Analogamente per scrivere:

```
FileWriter out = new FileWriter("copia.txt");
```

**FileReader** e **FileWriter** sono sottoclassi di **Reader** e **Writer**.  
**FileReader** converte da uno stream di byte contenuti nel file in uno stream di caratteri, e viceversa per **FileWriter**.

```
int c = in.read();
out.write(c);
```

# Lettura e scrittura di stream di dati



Il processo di lettura di dati da uno stream può essere sintetizzato come segue:

apri lo stream

while (ci sono dati da leggere nello stream)

    leggi un dato dallo stream

chiudi lo stream

Il processo di scrittura è schematizzato nel modo seguente:

apri lo stream

while (ci sono dati da scrivere)

    scrivi il dato nello stream

chiudi lo stream

# Esempio: programma che copia un file in un altro file leggendo e scrivendo carattere per carattere.



```
public class Copy {
 public static void main(String[] args) throws IOException {

 FileReader in = new FileReader("esempio.txt");
 FileWriter out = new FileWriter("copia.txt");
 int c;

 while ((c = in.read()) != -1) // -1 è l'EOF
 out.write(c);

 in.close();
 out.close();
 }
}
```

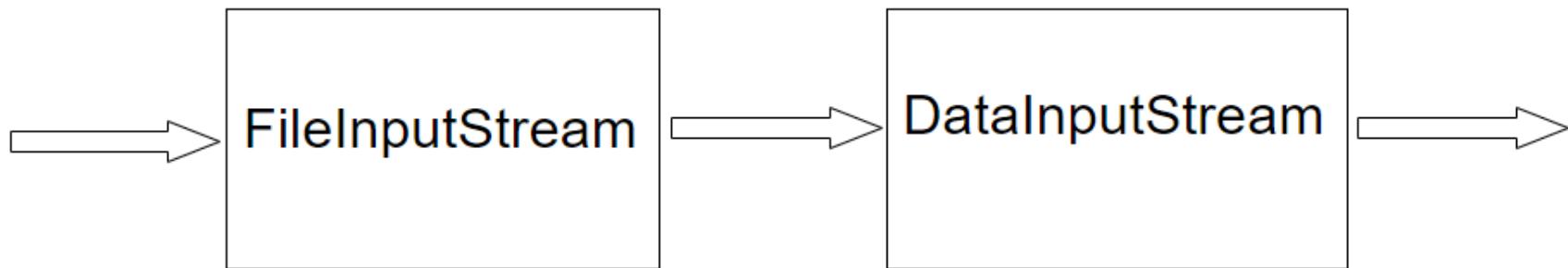


Gli *stream* che specificano il tipo di sorgente, come ad es. **FileInputStream**, non hanno metodi per leggere tipi primitivi (numeri, ...) - leggono solo byte o caratteri.

Altri *stream*, derivati da **FilterInputStream**, possono assemblare i byte in tipi di dati. Es. **DataInputStream** (vd. poi)

Analogamente per l'output.

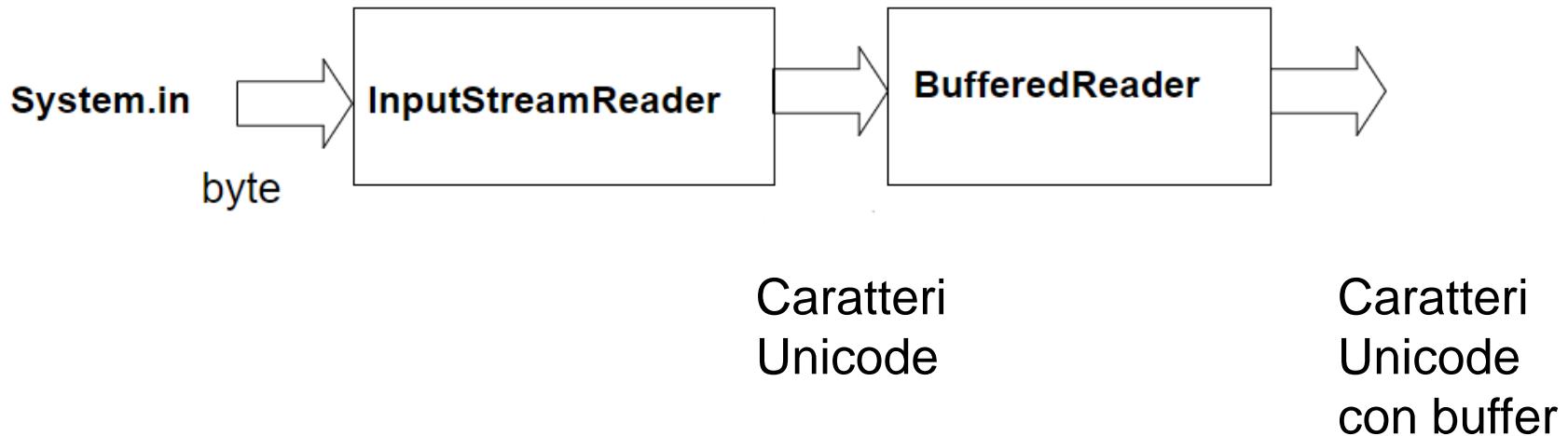
Gli *stream* possono essere composti per manipolare i dati:





# Composizione di stream:

Lo «standard input» (System.in) ha tipo InputStream → si legge usando un InputStreamReader.



```
BufferedReader in = new BufferedReader(
 new InputStreamReader(System.in));
```



# Composizione di stream

```
BufferedReader in = new BufferedReader(
 new InputStreamReader(System.in));
```

La classe **InputStreamReader** trasforma uno stream di input che contiene byte in un *reader* che emette caratteri Unicode.

Componendo **BufferedReader** con **InputStreamReader** si ottiene uno stream di input che usa un buffer:

```
DataInputStream in = new DataInputStream(
 new BufferedInputStream(
 new FileInputStream("impiegati.dat")))
```



# Esempio:

```
public class CopiaFileCaratteriConBuffer {

 public static void main(String[] args) throws IOException {
 BufferedReader in =
 new BufferedReader(new FileReader("esempio.txt"));
 BufferedWriter out =
 new BufferedWriter(new FileWriter("copia.txt"));
 int c;
 while ((c = in.read()) != -1)
 out.write(c);
 out.flush();
 in.close();
 out.close();
 }
}
```



# I/O di tipi di dati primitivi

Le classi **DataInputStream** e **DataOutputStream** forniscono metodi per leggere e scrivere dati primitivi

I dati vengono codificati in formato binario e non sono leggibili come testi.

```
DataOutputStream dout = new DataOutputStream(
 new FileOutputStream("prova.dat"));
dout.writeInt(250);
dout.writeDouble(3.14);
dout.writeChar('a');
dout.close();
DataInputStream din = new DataInputStream(
 new FileInputStream("prova.dat"));
System.out.println(din.readInt());
System.out.println(din.readDouble());
System.out.println(din.readChar());
```



# I/O di oggetti

Con **ObjectInputStream** e **ObjectOutputStream** è possibile leggere o scrivere qualunque **oggetto** purché la sua classe implementi l'interfaccia **Serializable**.

```
ObjectOutputStream out = new ObjectOutputStream(
 new FileOutputStream("impiegati.dat"));
Impiegato rossi = new Impiegato(...);
out.writeObject(rossi);
ObjectInputStream in = new ObjectInputStream(
 new FileInputStream("impiegati.dat"));
Impiegato imp = (Impiegato)in.readObject();

class Impiegato implements Serializable {...}
```



# Interface Serializable

L'interfaccia **Serializable**, predefinita, non contiene nessun metodo.

Serve da marcatore di classi di oggetti che possono essere trasmessi negli stream. ObjectOutputStream non emette alcun oggetto che non sia Serializable. Gli oggetti Serializable vengono salvati con tanto di informazioni sulla definizione della loro classe, per permetterne la ricostruzione (deserializzazione).

Un oggetto Serializable non può contenere variabili di istanza che non lo siano, oppure le deve marcare come ***transient*** per indicare che non sono serializzabili e quindi verranno ignorate nel processo di Serializzazione. Tutti i tipi primitivi sono serializable per default.

*Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:*

*private void writeObject(java.io.ObjectOutputStream out) throws IOException*

*private void readObject(java.io.ObjectInputStream in) throws IOException,*

*ClassNotFoundException;*

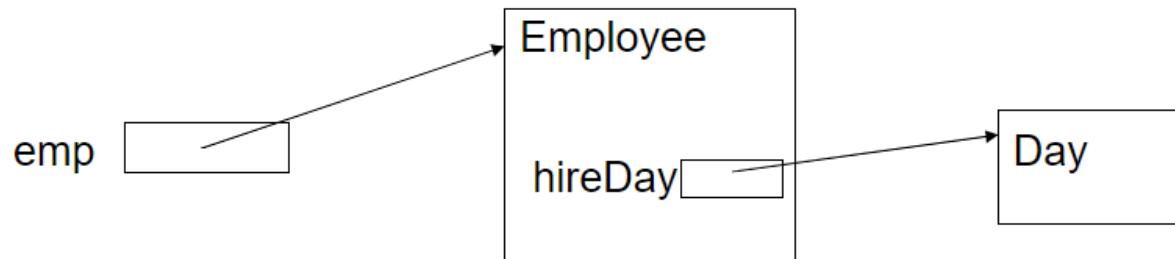
*private void readObjectNoData() throws ObjectStreamException;*

Input Output



Cosa succede se si scrive un oggetto che contiene riferimenti ad altri oggetti?

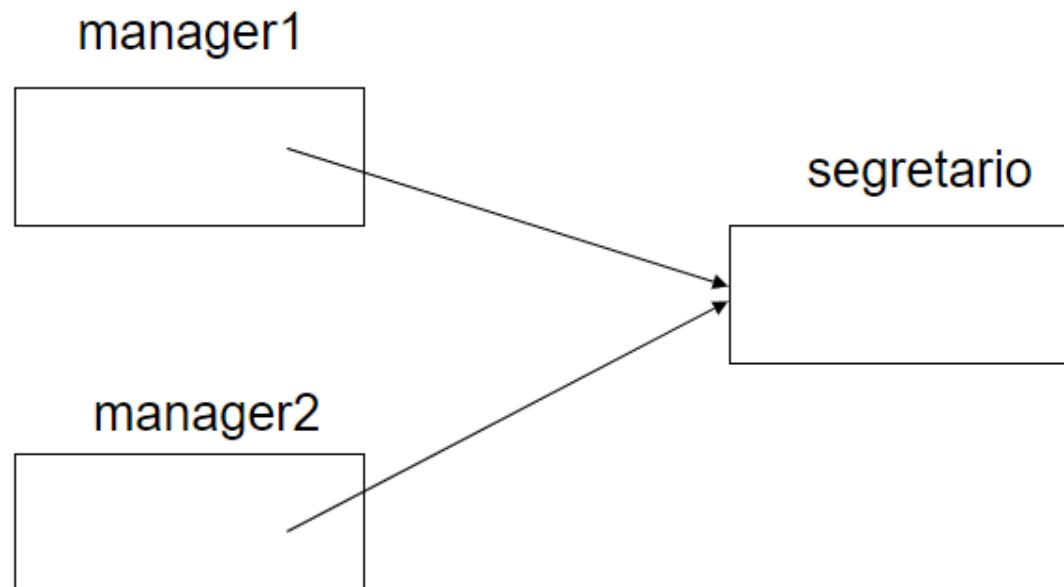
Vengono scritti anche tutti gli altri oggetti raggiungibili.



Se si esegue `out.writeObject(emp)` vengono scritti in `out` sia l'oggetto **Employee** che l'oggetto **Day**.



Se un oggetto è condiviso, ne viene scritta solo una copia.





# File

**File** rappresenta i file come oggetti (nascondendo dettagli del file system). Si può passare un file come parametro al costruttore di **FileReader** per creare un lettore che lavora sul file corrispondente:

```
File inputFile = new File("esempio.txt");
FileReader in = new FileReader(inputFile);
```

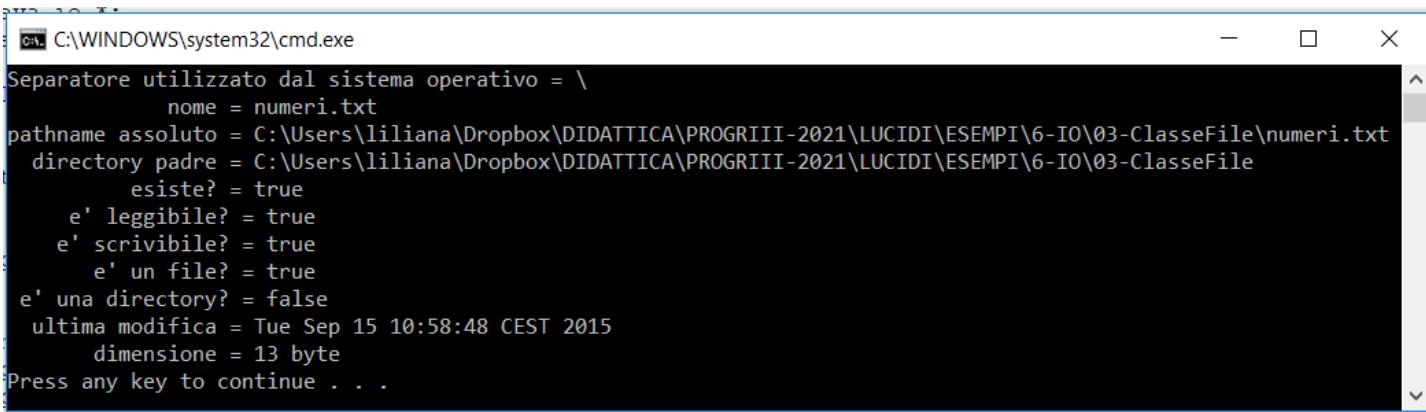
La classe **File** può anche essere usata per gestire directory.



# File: gestione di file

```
public class TestFile {

 public static void main(String[] args) throws IOException {
 PrintStream out = System.out;
 out.println("Separatore utilizzato dal sistema operativo = " + File.separator);
 File file = new File(args[0]);
 out.println(" nome = " + file.getName());
 out.println("pathname assoluto = " + file.getAbsolutePath());
 out.println(" directory padre = " +
 (new File(file.getAbsolutePath()).getParent()));
 out.println(" esiste? = " + file.exists());
 out.println(" e' leggibile? = " + file.canRead());
 out.println(" e' scrivibile? = " + file.canWrite());
 out.println(" e' un file? = " + file.isFile());
 out.println(" e' una directory? = " + file.isDirectory());
 out.println(" ultima modifica = " +
 (new Date(file.lastModified())).toString());
 out.println(" dimensione = " + file.length() + " byte");
 }
}
```



```
C:\WINDOWS\system32\cmd.exe
Separatore utilizzato dal sistema operativo = \
 nome = numeri.txt
pathname assoluto = C:\Users\liliana\Dropbox\ DIDATTICA\PROGRIII-2021\ LUCIDI\ESEMPI\6-IO\03-ClasseFile\numeri.txt
 directory padre = C:\Users\liliana\Dropbox\ DIDATTICA\PROGRIII-2021\ LUCIDI\ESEMPI\6-IO\03-ClasseFile
 esiste? = true
 e' leggibile? = true
 e' scrivibile? = true
 e' un file? = true
 e' una directory? = false
 ultima modifica = Tue Sep 15 10:58:48 CEST 2015
 dimensione = 13 byte
Press any key to continue . . .
```

# File: gestione di directory



```
public class ListaDir {
 public static void main(String[] args) throws IOException {
 PrintStream out = System.out; //predisposizione del canale di output
 File f = null;
 if (args.length == 0) //se non ci sono argomenti consideriamo la directory corrente
 f = new File(".");
 else
 f = new File(args[0]);

 if (f.isFile()) //se il nome specificato e'...
 //...quello di un file allora ne stampa nome e dimensione
 out.println("File: " + f.getAbsolutePath() + ", " + f.length() + " byte");
 else { //...quello di una directory allora...
 out.println("Directory: " + f.getAbsolutePath());
 String[] lista = f.list(); //...preleva la lista dei file...
 for(int i=0; i < lista.length; i++) { //...stampa i dati di ognuno
 File tmp = new File(f.getAbsolutePath(), lista[i]);
 if (tmp.isFile())
 out.println(" file: " + tmp.getName() + " " + tmp.length() + " byte");
 else
 out.println(" dir.: " + tmp.getName());
 } } } }
```

The screenshot shows a Windows Command Prompt window titled 'cmd.exe'. The command 'dir' is run in the directory 'C:\Users\liliana\Dropbox\ DIDATTICA\PROGRIII-2021\ LUCIDI\ ESEMPI\6-I0\03-ClasseFile\.'. The output lists several files and a directory:

```
C:\WINDOWS\system32\cmd.exe
Directory: C:\Users\liliana\Dropbox\ DIDATTICA\PROGRIII-2021\ LUCIDI\ ESEMPI\6-I0\03-ClasseFile\

file: ListaDir.class 1313 byte
file: ListaDir.java 1073 byte
file: numeri.txt 13 byte
dir.: sottoCartella
file: TestFile.class 1744 byte
file: TestFile.java 1039 byte
file: tp20710e.BAT 168 byte
Press any key to continue . . .
```



# Classi Java per I/O da file – da JDK 1.5

- Per file testuali
  - classi Scanner e File per leggere da file
  - classe PrintWriter per scrivere su file
- Per file binari
  - classe FileInputStream per leggere da file
  - classe FileOutputStream per scrivere su file



# Lettura da file testuale - I

Per aprire un file da applicazione java e leggerne i dati

- **Scanner scf = new Scanner(new File(path+“/info.txt”));**
- Scanner s = new Scanner(new File(“C:\\documenti\\info.txt”));

Lo Scanner mi permette di specificare file con path assoluto (notare i doppi backslash)



# Lettura da file testuale - II

- **Scanner scf = new Scanner(new File(path+“/info.txt”));**
    - Oggetto **File** permette di aprire un file senza scrivere operazioni a livello di sistema operativo
    - Oggetto **Scanner** opera sul file passato come parametro e permette di leggere contenuto (come da standard input)
  - File è nel package `java.io`
  - Scanner è nel package `java.util`
- importare i package se si usano queste classi



# Input/Output da file - eccezioni

Leggere dati da file richiede la gestione di eccezioni

- **FileNotFoundException**: il file specificato non esiste (nome di file sbagliato, path errato, file cancellato per errore, ...)
- **InputMismatchException**: dati malformattati - non corrispondono a ciò che applicazione si aspetta (es: si aspetta token int e trova boolean)

⇒ operazioni da eseguire in **try** e **catch** di queste eccezioni

**Oppure si catturi IOException**: eccezione + generale che si può catturare quando si manipolano file (consigliata in quanto più generica delle altre).



# Esempio: visualizzo linee di file - I

```
public class TestScannerPrintFile {
 public static void main(String[] args) {
 String path = «....path....»;
 try {
 Scanner scf = new Scanner(new File(path+“/info.txt”));
 while (scf.hasNextLine()) { //Verifico che ci sia una
 // linea di file da leggere con l’oggetto Scanner

 String riga = scf.nextLine(); // leggo la linea

 System.out.println("RIGA: "+riga); }
 scf.close(); // chiudo l’oggetto Scanner (meglio in un blocco finally)
 } catch (IOException e) {...gestione eccezione ...}
 }
}
```



# Classe Scanner

L'oggetto Scanner impostato su un file permette di analizzare il contenuto di una linea del file se si sa quali tipi di dati contiene

- Es: se il file contiene una sequenza di numeri interi posso estrarre tali numeri (token) a 1 a 1 con `scf.nextInt()`
- L'estrazione di token si basa su definizione di **token** e di **simbolo delimitatore** (ciò che separa 1 token dal successivo). Il delimitatore di default è lo spazio vuoto.

Quando si chiude lo scanner (`scf.close()`) si rilascia la risorsa su cui lavorava (file, String, etc.)



# Token e delimitatori - I

- Il delimitatore di base è lo spazio vuoto (uno o + spazi consecutivi sono un solo delimitatore)
  - Es: sequenza di numeri interi (5 token):  
123 24 76 4 534
  - Es: sequenza di stringhe (3 token):  
ciao a tutti
- Per leggere i token presenti in un file bisogna leggere il file, linea per linea, ed estrarre i token a 1 a 1 da ciascuna linea



# Token e delimitatori - esempio

```
public class TestScanToken {
 public static void main (String[] args) {
 try {
 Scanner scf = new Scanner(new File("dati1.txt"));

 while (scf.hasNext())
 System.out.print(scf.nextInt() + " ");
 scf.close();
 } catch (IOException e) {e.printStackTrace();}
 }
}
```

```
3425 342 7777 2 34 12 897 44
222 2222 333 4444 55555
File dati1.txt
```



# Token e delimitatori - II

- Si può configurare un oggetto Scanner affinché usi un delimitatore diverso da quello base specificandolo attraverso espressioni regolari
- Es: **1 fish 2 fish red fish blue fish**, voglio trattare **fish** come delimitatore

```
String tmp = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(tmp).useDelimiter("\s*fish\s*");
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());
s.close();
```

Nelle espressioni regolari, \s rappresenta lo spazio vuoto  
→ \s\* significa 0 o più spazi vuoti



# Scrittura su file testuale - I

- **PrintWriter p = new PrintWriter(new File(*nomeFile*));**
- **Es: PrintWriter p = new PrintWriter(new File(“risultati.txt”));**
- Se nella cartella corrente esiste già un file di nome *nomeFile*, viene aperto tale file.  
Altrimenti, viene creato un nuovo file con il nome specificato.
- L’oggetto PrintWriter **p** è ora pronto per scrivere sul file specificato



# Scrittura su file testuale - II

- **PrintWriter p = new PrintWriter(new File("risultati.txt"));**
- PrintWriter offre metodi **print(String s)** e **println(String s)** per stampare stringhe passate come parametro  
**p.println(23); p.println(29); ...**
- NB: la scrittura avviene su buffer di output, per ragioni di efficienza ⇒ per scrivere effettivamente sul file bisogna svuotare il buffer: **p.flush()**

```
23
29
File risultati.txt
```

NB: i numeri sono stati convertiti in String ...

# Scrittura su file testuale - esempio



```
public static void main (String[] args) {
 PrintWriter p = null;
 try {
 p = new PrintWriter(new File("numeri.txt"));
 for (int i=0; i<5; i++) {
 int num = Lettura.leggiIntero("digita numero: ");
 p.println(num);
 p.flush();
 }
 } catch(IOException | RuntimeException e) {
 System.out.println(e.getMessage());
 // NB: metto la chiusura del file in finally per essere certa che
 // se ci sono problemi venga chiuso il file (che conterrà
 // quanto inserito fino a prima dell'eccezione).
 finally {if (p!=null) //NB: se fallisce la new, p è null
 p.close();
 }
}
```

*NB: chiudere il PrintWriter a termine scrittura.*



# Ringraziamenti

Grazie al Prof. Emerito Alberto Martelli e  
al Prof. Matteo Baldoni del Dipartimento  
di Informatica dell'Università di Torino per  
aver prodotto parte del material  
presentato in queste slides.



# Programmazione III

Prof.ssa Liliana Ardissono  
Dipartimento di Informatica  
Università di Torino

**Interfacce Utente Grafiche (GUI) – parte 1  
(basi con Java SWING)**



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND



# PROGRAMMAZIONE GRAFICA

Molti programmi interagiscono con l'utente attraverso una interfaccia grafica

**GUI** - Graphical User Interface

Java fornisce diverse librerie di classi per realizzare GUI.



Nelle prime versioni di Java (1.0, 1.1) era fornita la libreria

## **AWT** (Abstract Window Toolkit)

per realizzare la portabilità, la gestione dei componenti grafici era delegata ai toolkit nativi delle varie piattaforme (Windows, Unix, iOS, ...)

Successivamente è stata fornita la libreria **SWING**, che fa un uso molto ridotto dei toolkit nativi.

I componenti sono *dipinti* in finestre vuote.

In ogni caso, programmi Java che usano SWING, devono spesso usare anche classi AWT.



Il componente di più alto livello di una interfaccia grafica è una finestra, realizzata dalla classe **JFrame**.

Tutte le classi i cui nomi iniziano con J appartengono alla libreria javax.swing.

Gli altri componenti al livello top sono:

## **JApplet e JDialog**

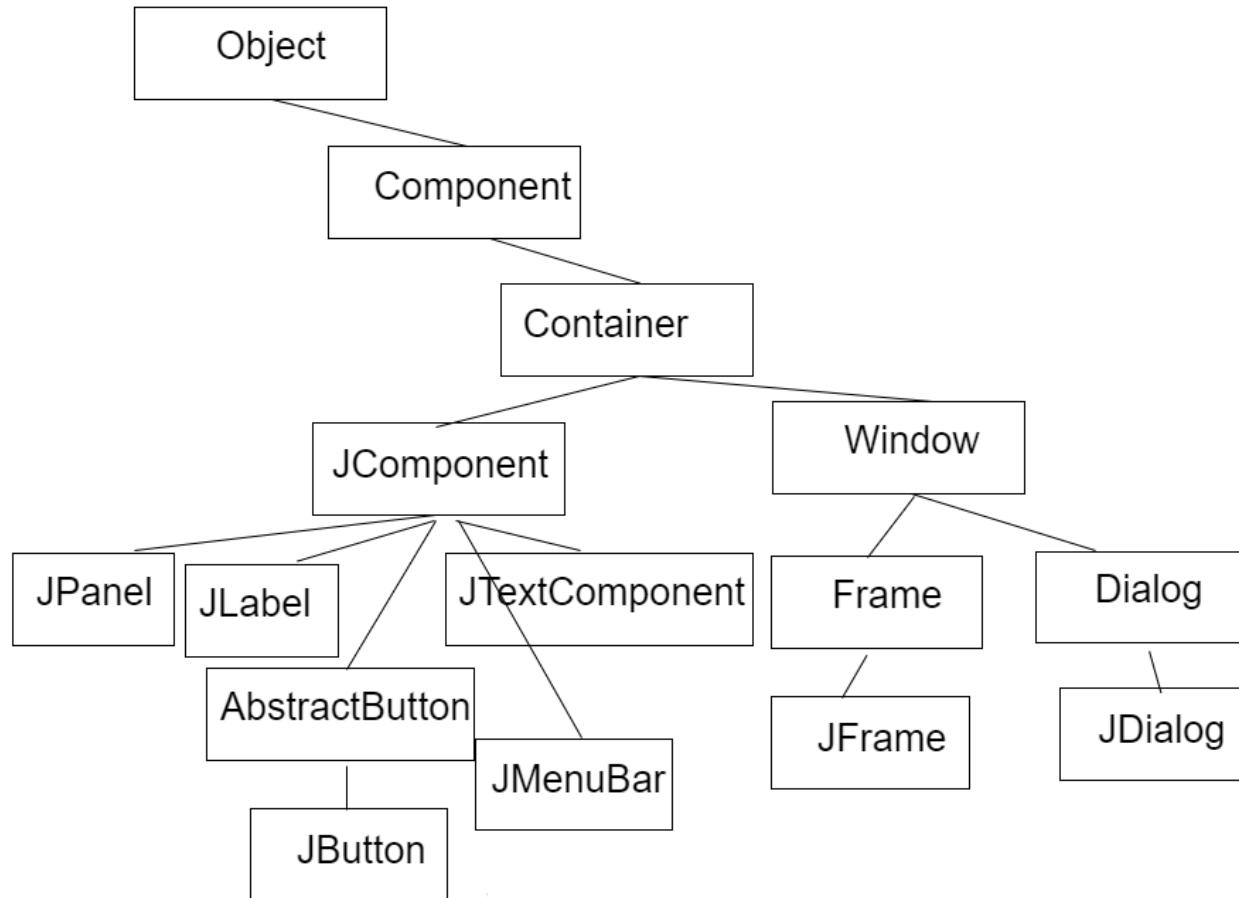
I *frame* sono dei contenitori, in cui si possono inserire altri componenti (pulsanti, testo, ...) o in cui si può disegnare.

Altri contenitori sono:

## *JPanel*

## *Container*

# Porzione della gerarchia delle classi di interfaccia grafica java



# Esempio: HelloWorldSwing - I

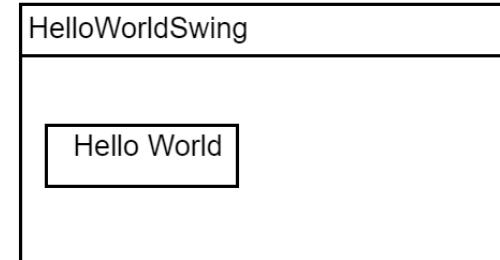


Un **JFrame** contiene un *pannello del contenuto* (che è un **Container**) in cui si possono inserire i componenti grafici.

I componenti grafici vengono inseriti nel contenitore secondo un *layout* (disposizione) predefinito, che dipende dal tipo del contenitore.

Il programmatore può impostare il *layout* da utilizzare tramite opportuni comandi, che trascuriamo.

Noi inseriamo la scritta *Hello World* in un componente **JLabel**, etichetta con testo, che viene inserito nel "content pane" del **JFrame**.



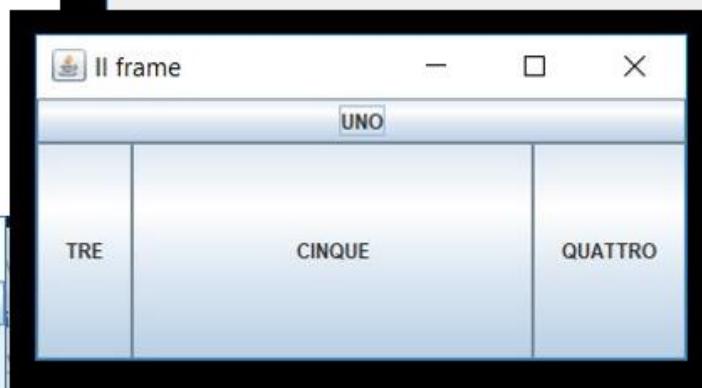


Esempi di applicazioni con layout differenti (su I-Learn):  
gli elementi sono visualizzati nel pannello in modi diversi.

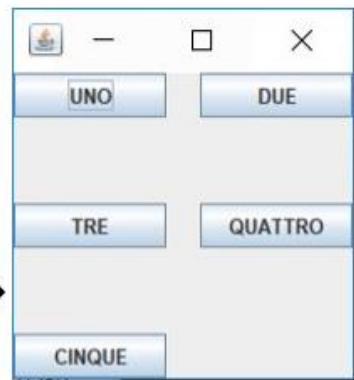
FlowLayoutApp →



BorderLayoutApp →



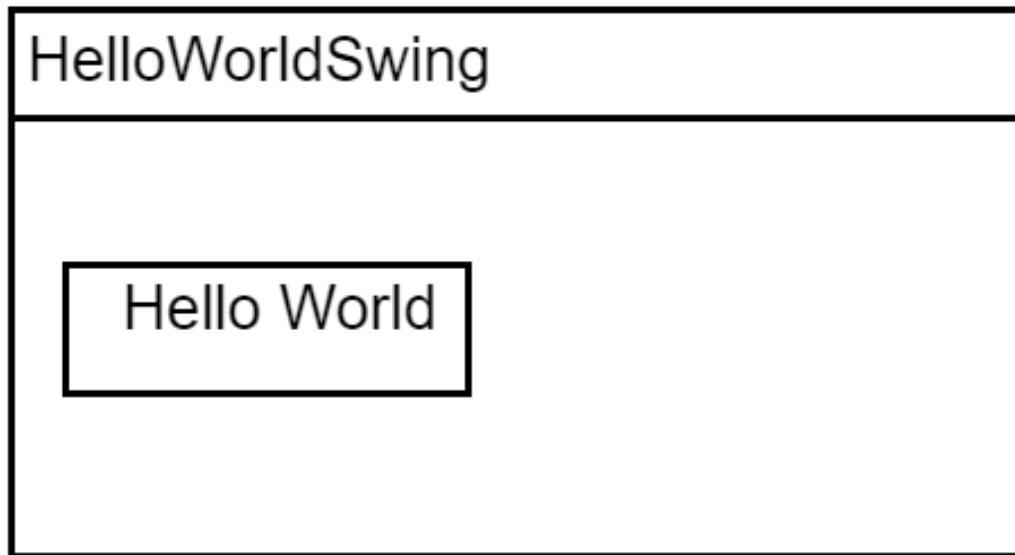
GridLayoutApp →



# Esempio: HelloWorldSwing - II



```
JFrame frame = new JFrame("HelloWorldSwing");
JLabel label = new JLabel("Hello World");
frame.add(label);
```



# Esempio: HelloWorldSwing - III



```
public class HelloWorldSwing {
 public static void main(String[] args) {

 JFrame frame = new JFrame("HelloWorldSwing");

 final JLabel label = new JLabel("Hello World");

 frame.add(label);

 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 // quando si chiude la finestra
 // termina l'esecuzione dell'applicazione

 frame.pack(); // fissa la dimensione della finestra in base al contenuto
 frame.setVisible(true); // mette la finestra visibile
 }
}
```

# Definizione di classi di finestre specifiche



Per permettere la creazione di finestre multiple, e per organizzare il codice in modo modulare, si può creare una classe che estende JFrame, e che nel suo costruttore ha tutti gli elementi del tipo di finestra desiderato:

```
class MyFrame extends JFrame {
 public MyFrame(String s) {
 super(s);
 setSize(400, 200);
 add(new JLabel("ciao"));
 ...
 }
}
```



# Esempio: Beeper - I

```
public class Beeper extends JFrame {
```

```
 JButton button;
```

```
 JPanel panel;
```

```
Beeper() {
```

```
 button = new JButton("Click Me");
```

```
 panel = new JPanel();
```

```
 panel.add(button);
```

```
 add(panel);
```

```
}
```



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

```
 Beeper beep = new Beeper();
```

```
 beep.pack();
```

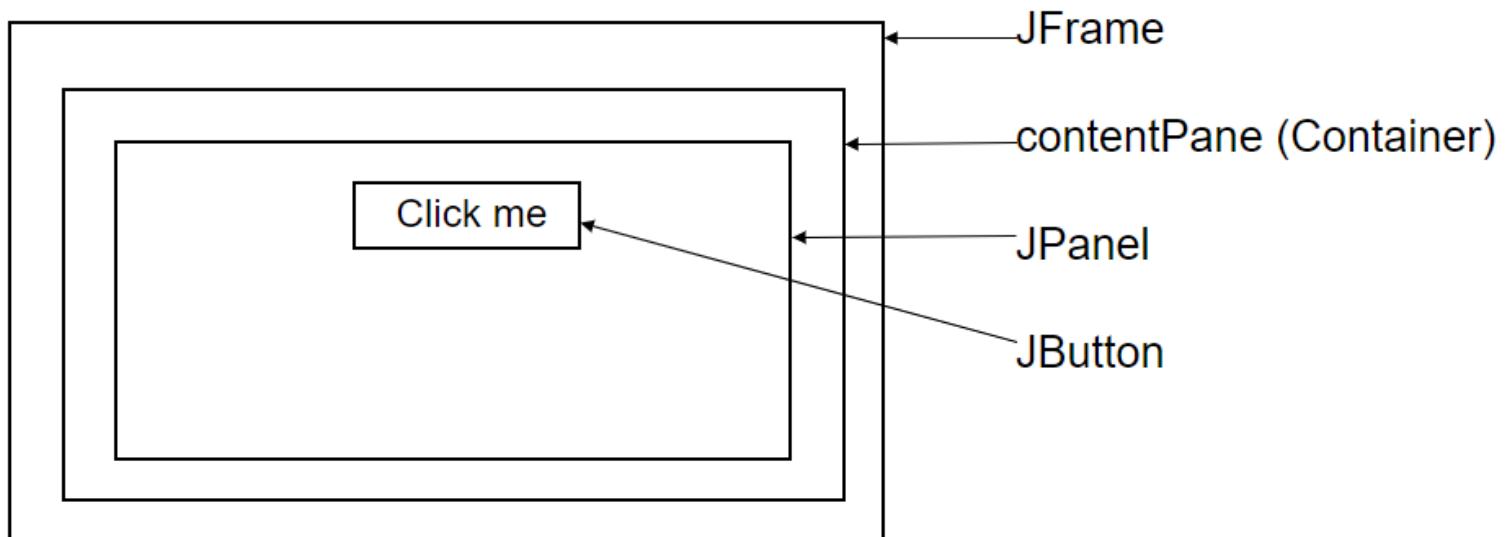
```
 beep.setVisible(true);
```

```
}
```

```
}
```

# Beeper – struttura della finestra

```
button = new JButton("Click Me");
panel = new JPanel();
panel.add(button);
frame.add(panel);
```





# Event-driven programming

**Come far sì che, quando si preme il bottone «Click me» nella finestra di Beeper, si senta beep?**

Quando si fa click con il mouse sul bottone, viene generato un evento "bottone premuto". Questo evento deve attivare l'azione di fare *beep*.

Per fare questo si usa la programmazione guidata dagli eventi: **event-driven programming**.

Questa tecnica è adottata da molti linguaggi usati per realizzare interfacce o per programmare browser (JavaScript, ...).



# Programmazione guidata dagli eventi (event-driven programming)

I programmi tradizionali hanno un comportamento funzionale: ricevono un input, eseguono la propria computazione e restituiscono un risultato.

Normalmente questi programmi seguono il proprio flusso di controllo e raramente possono contenere punti di diramazione che si basano su input dell'utente.

In molti casi invece, es. interfacce grafiche, un programma deve avere un **comportamento reattivo**: ogni volta che l'utente genera un evento, il programma deve reagire all'evento eseguendo una azione opportuna.



# Programmazione guidata dagli eventi (event-driven programming)

Un programma basato su questa metodologia consiste di un insieme di procedure (**event handlers**), ciascuna delle quali specifica cosa fare quando si verifica un certo tipo di evento.

Il programma deve contenere un event-handler per ogni tipo di evento da gestire. Quando l'evento si verifica, verrà eseguito l'event-handler associato.

→ Il flusso di controllo con cui il programma viene eseguito non è determinato a priori, ma dipende dall'ordine con cui gli eventi si verificano. Il programma termina quando si verifica un evento che ne richiede la terminazione.

# Gli eventi in un'interfaccia grafica



In Java **gli eventi** sono oggetti derivati dalla classe **EventObject**.

Tipi di eventi:

- **semantic**i, che fanno riferimento a quello che l'utente fa su componenti "virtuali" dell'interfaccia (premere un pulsante, selezionare la voce di un menu, ...)
- **low-level**, ossia eventi fisici relativi al mouse o alla tastiera (tasto premuto, tasto rilasciato, mouse trascinato, ...)

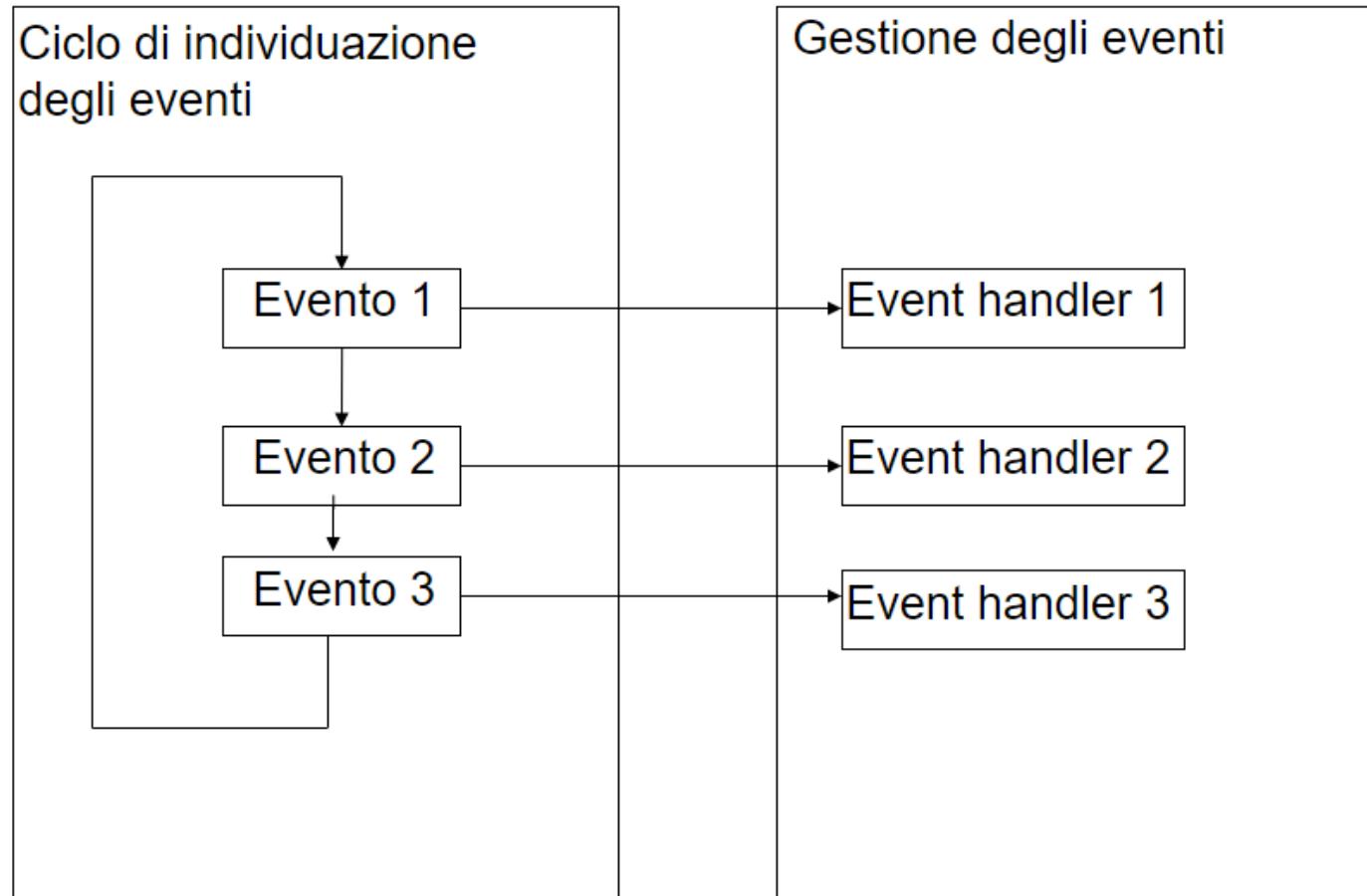
Le **sorgenti** degli eventi sono i diversi componenti dell'interfaccia, come JButton, JTextField, Component, Window, ...

# Organizzazione di un'applicazione event-driven

1. L'applicazione crea gli *event-handlers*.
2. Poi, l'applicazione deve **registrare** gli *event handlers* presso la sorgente degli eventi. Questo significa legare ogni *event handler* a un tipo di evento che riguarda la specifica sorgente (componente della GUI).

Durante l'esecuzione dell'applicazione, la sorgente degli eventi esegue un ciclo degli eventi, per scoprire se qualche evento si verifica. Quando si verifica un evento, la sorgente degli eventi invoca l'*event handler (listener)* associato, se esiste.

# Schema di programma guidato dagli eventi

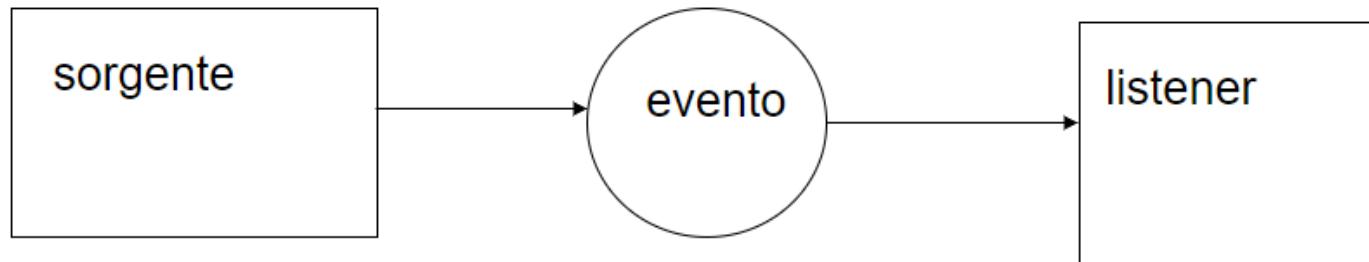


*Il ciclo degli eventi è un concetto astratto: gli eventi potrebbero essere segnalati da un meccanismo di interrupt, senza bisogno che il programma esegua continuamente il ciclo.*



# Gestione degli eventi

Gli eventi sono gestiti con un meccanismo di *delega*.



La sorgente, quando genera un evento, passa un **oggetto** che descrive l'evento ad un "listener" che gestisce l'evento.

Il *listener* deve essere "registrato" presso la sorgente per poter ricevere l'evento.

Il passaggio dell'evento causa l'invocazione di un metodo del *listener* associato a quel tipo di evento.



# Event handlers (listeners)

In Java un *event-handler*, chiamato **listener**, è un'istanza di una classe che contiene dei metodi per gestire gli eventi.

Per ogni tipo di evento è definita una interfaccia che il *listener* relativo deve implementare (ogni *listener* può gestire eventi di un certo tipo). Es:

- ActionListener (per eventi da bottoni)
- MouseListener (eventi del mouse)
- MouseMotionListener (spostamenti del mouse)
- WindowListener (eventi dovuti ad azioni su finestra - JFrame)
- ...

# Esempio: eventi generati dai bottoni (Jbutton)



I bottoni generano un solo tipo di evento, quando vengono schiacciati: l'**ActionEvent**.

Il rispettivo *listener* deve implementare l'interfaccia

```
interface ActionListener {
 void actionPerformed(ActionEvent e);
}
```

Per registrare l'**ActionListener** nel bottone, si usa il metodo della classe **JButton**

```
void addActionListener(ActionListener l)
```

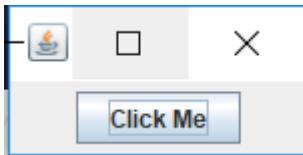


Per gestire un **ActionEvent** generato da un bottone, si deve:

- definire una classe che implementa l'interfaccia **ActionListener**, con il relativo metodo **actionPerformed()**;
- creare un'istanza di questa classe;
- *registrarla* presso il bottone, eseguendo il metodo **addActionListener()** del bottone stesso.

Ogni volta che si preme il bottone, questo invoca automaticamente il metodo **actionPerformed()** del listener inviandogli (passando come parametro attuale) l'evento.

È possibile registrare più listener nello stesso componente grafico.



# Esempio: Beeper - II

```
public class Beeper extends JFrame { //con classe interna non anonima
 JButton button;
 JPanel panel;

 Beeper() {
 button = new JButton("Click Me");
 panel = new JPanel();
 panel.add(button);
 add(panel); pack();
 button.addActionListener(new BeepListener());
 }

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

class BeepListener implements ActionListener {
 public void actionPerformed(ActionEvent e) {
 Toolkit.getDefaultToolkit().beep();
 }
}
```

## Versione alternativa, con classe anonima o lambda expression

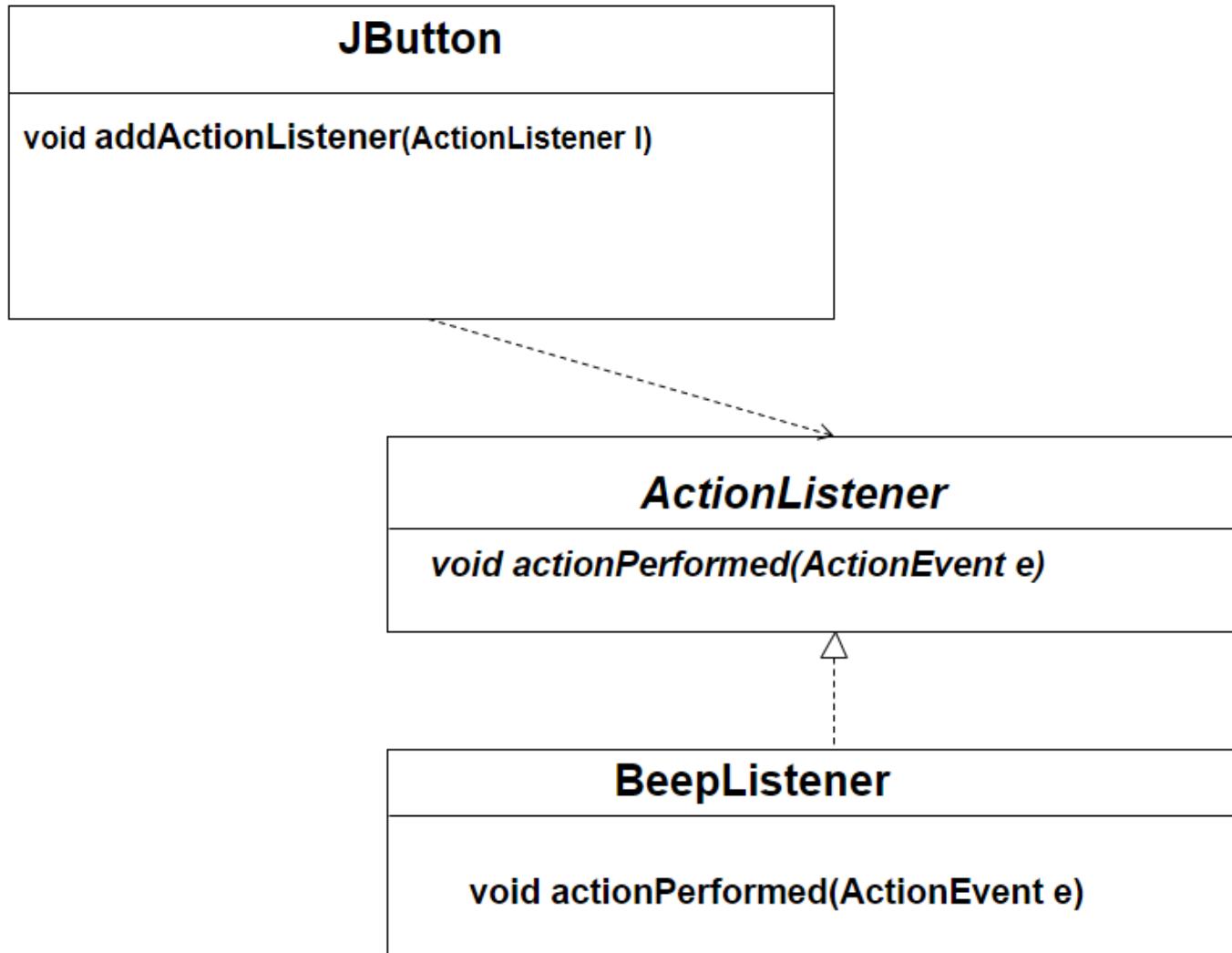
```
public class Beeper6Lambda extends JFrame
 private JButton button; private JPanel panel;

 public Beeper6Lambda() {
 panel = new JPanel(); button = new JButton("Click Me");
 ActionListener listener = event -> {Toolkit.getDefaultToolkit().beep();
 System.out.println("BEEP!");};

 button.addActionListener(listener); // con lambda expression
 /*button.addActionListener(new ActionListener() { // classe anonima
 public void actionPerformed(ActionEvent e) {
 Toolkit.getDefaultToolkit().beep();
 System.out.println("BEEP!"); } });*/
 panel.add(button); add(panel); pack();
 }

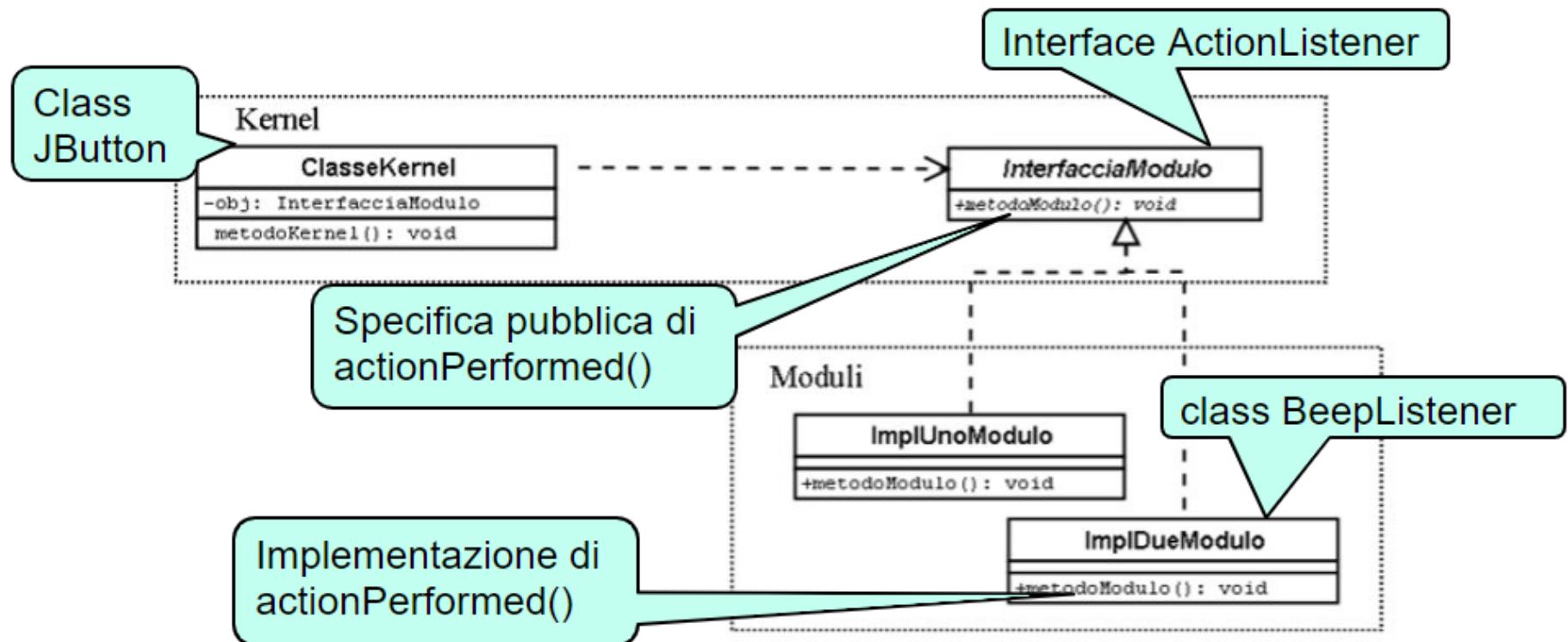
 public static void main(String[] args) {
 Beeper6Lambda beep = new Beeper6Lambda(); beep.setVisible(true);
 }
}
```

# Gerarchia delle classi





# Ma è lo schema Kernel-Modulo!



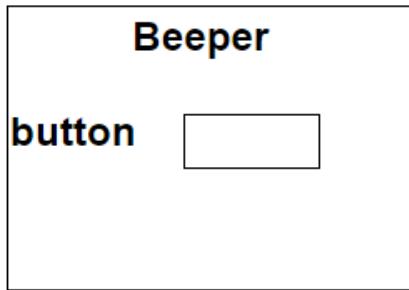
# Simulazione della memoria JVM



A runtime

Beeper beep = new Beeper(); \\ istruzione del main()

Oggetti creati nello HEAP

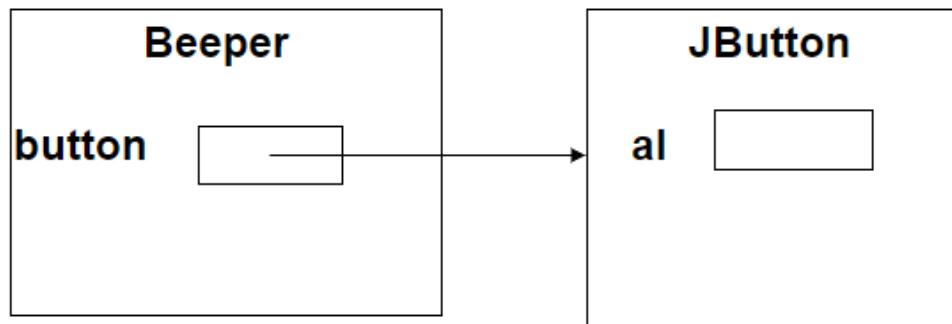


# A runtime



`JButton button = new JButton("Click Me"); \\ nel costruttore di Beeper`

Oggetti creati nello HEAP



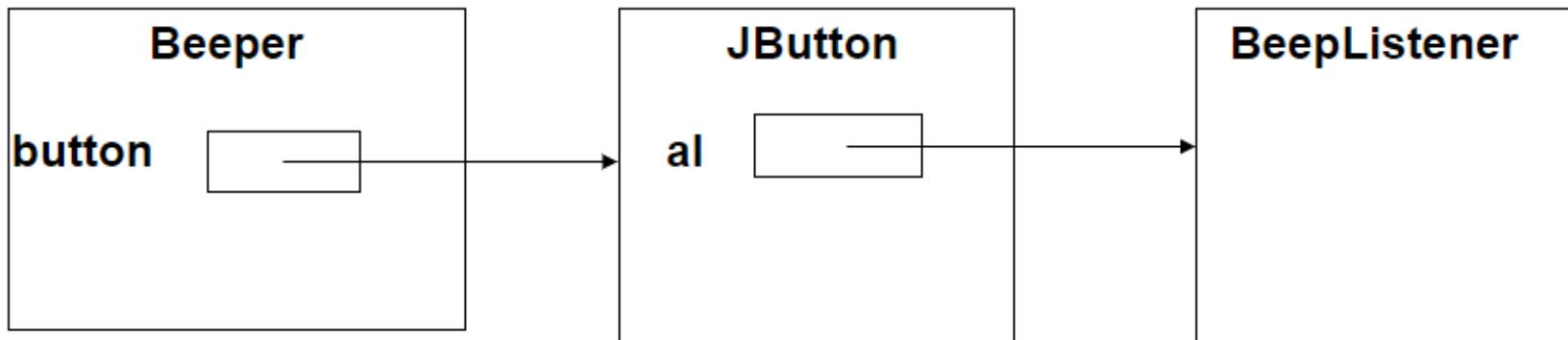
Il campo **al** di **JButton** contiene un riferimento alla lista degli **ActionListener** del bottone.

# A runtime



```
JButton button = new JButton("Click Me");
button.addActionListener(new BeepListener());
```

Oggetti creati nello HEAP

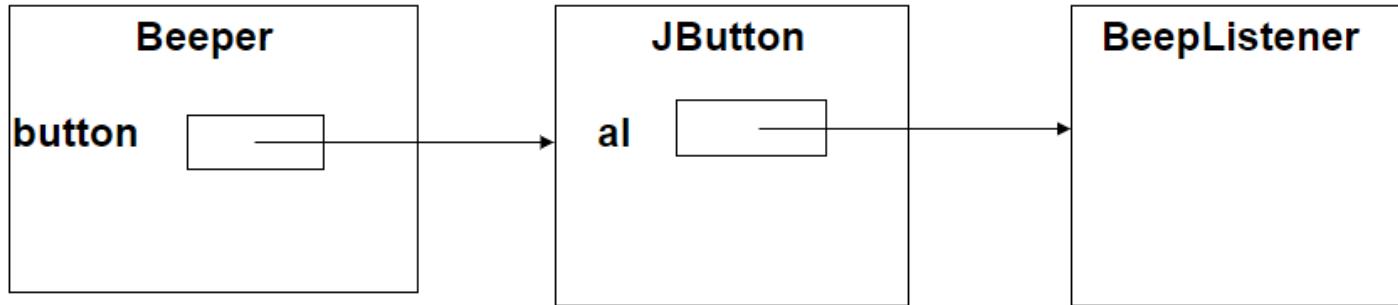


Il campo **al** di **JButton** contiene un riferimento alla lista degli **ActionListener** del bottone.



## A runtime

```
JButton button = new JButton("Click Me");
button.addActionListener(new BeepListener());
```



Il bottone, quando viene premuto, crea un oggetto **ActionEvent** e lo passa al **BeepListener** chiamandone il metodo **actionPerformed()**

**al.actionPerformed(new ActionEvent());**  
**(in JButton)**

# Beeper: implementazione alternativa - I



```
public class Beeper extends JFrame implements ActionListener {

 JButton button;
 JPanel panel;

 Beeper() {
 button = new JButton("Click Me");
 panel = new JPanel();
 panel.add(button);
 add(panel); pack();
 button.addActionListener(this);
 }

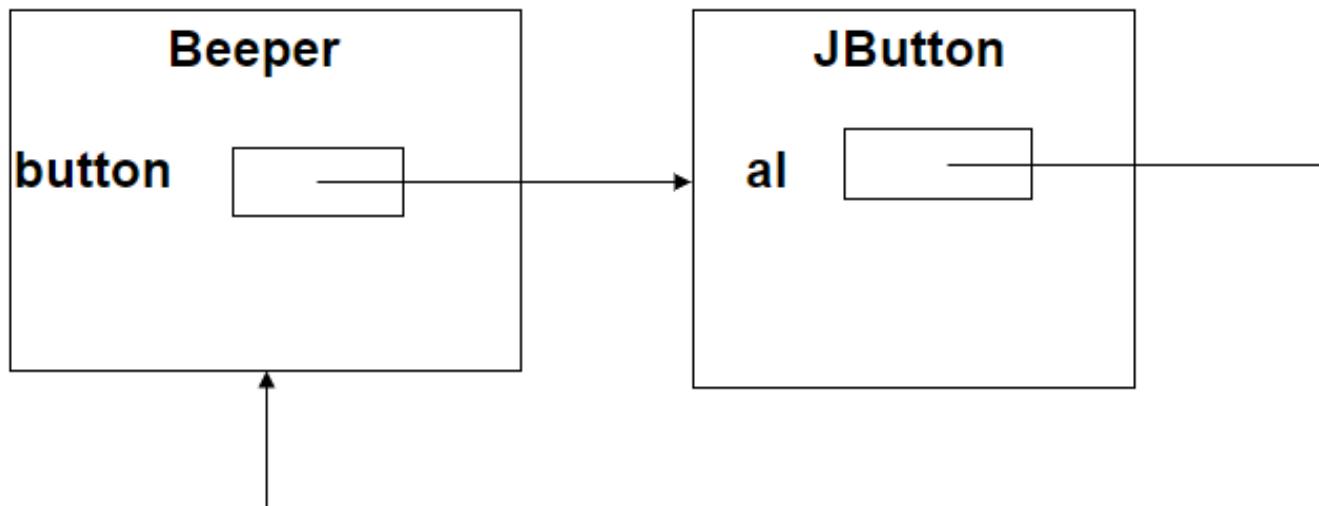
 public void actionPerformed(ActionEvent e) {
 Toolkit.getDefaultToolkit().beep();
 }
}
```

# Beeper: implementazione alternativa - II



L'*ActionListener* è implementato direttamente dal *JFrame*:

```
public class Beeper extends JFrame implements
ActionListener {
```



```
button.addActionListener(this);
```



# Tipi di eventi, eventHandlers e loro metodi

In generale, i nomi delle classi e delle operazioni relative agli eventi seguono un *pattern* comune.

Se **C** è una classe (bottone, finestra, ...), i cui oggetti possono generare eventi di tipo **xxx**, ci sarà:

una classe **xxxEvent** che implementa gli eventi;

una *interface* **XXXListener** con uno o più metodi per gestire l'evento;

i metodi **addXXXListener()** o **removeXXXListener()** nella classe **C**.

# Classi filtro (Adapters) - I



Le interfacce di molti tipi di listener specificano un lungo elenco di metodi per gestire i vari tipi di evento che possono essere lanciati dal corrispondente tipo di sorgente. Es:

- **MouseListener:** mouseExited(MouseEvent),  
mousePressed(MouseEvent),  
mouseReleased(MouseEvent),  
mouseEntered(MouseEvent)
- **WindowListener:** windowClosing(WindowEvent),  
windowOpened(WindowEvent),  
windowIconified(WindowEvent),  
windowDeiconified(WindowEvent),  
windowClosed(WindowEvent), ...



# Classi filtro (Adapters) - II

Il pattern di implementazione dell'interfaccia richiederebbe che il listener che voi sviluppate implementi tutti i suoi metodi. Questo potrebbe non essere rilevante per voi (magari vi interessa gestire un solo tipo di evento).

- Sono state introdotte le classi filtro, o adapters, che offrono le **implementazioni di default delle interfacce dei listener** (con metodi che non fanno nulla).
- Invece di implementare l'interfaccia del listener, quando non si è interessati a gestire tutti i suoi eventi si può estendere la classe adapter del listener e fare overriding dei soli metodi di gestione di eventi che ci servono

## Esempio: gestione di eventi delle finestre (JFrame)



Quando si preme il pulsante di chiusura di una finestra, viene generato un **WindowEvent** che deve essere opportunamente gestito.

Ad esempio, con l'istruzione:

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

quando si specifica che quando si chiude la finestra deve terminare l'esecuzione del programma.

Però, se in chiusura di finestra volessimo fare anche altre operazioni, questa istruzione non sarebbe sufficiente →  
Servirebbe un listener con opportuno metodo di gestione dell'evento.



L'evento di chiusura della finestra può essere gestito come qualunque altro evento.

Un **JFrame** genera un **WindowEvent** ogni volta che la finestra cambia stato: aperta, chiusa, ridotta a icona, ...

L'interfaccia **WindowListener** deve gestire tutti i possibili cambiamenti di stato della finestra. Per questo specifica 7 metodi:

    windowActivated(WindowEvent e)

    windowClosing(WindowEvent e)

    ecc.

Se a noi interessa solo il metodo **windowClosing()**, per implementare correttamente l'interfaccia dovremmo comunque definire anche gli altri 6 metodi.



# Implementazioni di default delle interface!

Java fornisce la classe **WindowAdapter**, che implementa l'interfaccia **WindowListener** con i 7 metodi che non fanno nulla (body vuoto).

Noi dovremo solo estendere questa classe ridefinendo i metodi che ci interessano. Nel nostro caso solo **windowClosing()**.

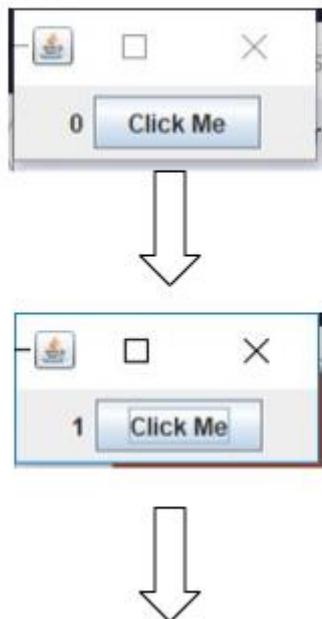
- Overriding dei metodi che vogliamo personalizzare.
- Il polimorfismo fa eseguire i metodi da noi scritti anziché quelli vuoti dell'implementazione di default.



```
public class Beeper extends JFrame {
 ...
 Beeper() {
 JButton button ...
 button.addActionListener(new ActionListener {
 public void actionPerformed(ActionEvent e) {
 Toolkit.getDefaultToolkit().beep();
 }
 });
 }
 addWindowListener(new WindowAdapter {
 public void windowClosing(WindowEvent e) {
 System.exit(0);
 //termina l'esecuzione del programma
 }
 });
}
}
```

# Beeper 1: non solo BEEP, ma anche accesso a variabili del pannello contenitore!

Supponiamo di voler contare il numero di volte che l'utente clicca il bottone Click Me → **il Listener del bottone deve gestire un contatore, da visualizzare nell'interfaccia utente.**



# Esempio1: Beeper2 (il panel implementa l'ActionListener)

```
public class Beeper2 extends JFrame implements ActionListener {
```

```
 private JButton button = new JButton("Click Me");
```

```
 private JPanel panel = new JPanel();
```

```
 private JLabel display = new JLabel("0");
```

```
 private int i = 0; // i è il contatore dei click
```

```
Beeper2() {
```

```
 panel.add(display); panel.add(button); add(panel);
```

```
 button.addActionListener(this);
```

```
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
}
```

```
public void actionPerformed(ActionEvent e) {
```

```
 Toolkit.getDefaultToolkit().beep();
```

```
 i++;
```

```
 display.setText(Integer.toString(i));
```

```
}
```

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

```
 Beeper2 beep = new Beeper2(); beep.pack(); beep.setVisible(true);
```

```
}
```

```
}
```



# Esempio2: ActionListener come nested class anonima



```
public class BeeperNestedAnonima extends JFrame {
```

```
 private JButton button = new JButton("Click Me");
```

```
 private JPanel panel = new JPanel();
```

```
 private JLabel display = new JLabel("0");
```

```
 private int i = 0;
```

```
 BeeperNestedAnonima() {
```

```
 panel.add(display); panel.add(button); add(panel);
```

```
 button.addActionListener(new ActionListener() {
```

```
 public void actionPerformed(ActionEvent e) {
```

```
 Toolkit.getDefaultToolkit().beep();
```

```
 i++; // accede a i perché nested class
```

```
 display.setText(Integer.toString(i)); }
```

```
 });
```

```
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
 pack(); setVisible(true);
```

```
}
```

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

```
 BeeperNestedAnonima beep = new BeeperNestedAnonima();
```

```
}
```



Programmazione III - Ardissono

# Esempio3: ActionListener come lambda expression



```
public class BeeperLambda extends JFrame {

 private JButton button = new JButton("Click Me");
 private JPanel panel = new JPanel();
 private JLabel display = new JLabel("0");
 private int i = 0;

 BeeperLambda() {
 panel.add(display); panel.add(button); add(panel);
 ActionListener listener = event -> { Toolkit.getDefaultToolkit().beep();
 i++;
 display.setText(Integer.toString(i)); };
 button.addActionListener(listener);
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 pack(); setVisible(true);
 }

 public static void main(String[] args) {
 BeeperLambda beep = new BeeperLambda();
 }
}
```



# Ringraziamenti

Grazie al Prof. Emerito Alberto Martelli del Dipartimento di Informatica dell'Università di Torino per aver redatto la prima versione di queste slides.



# Programmazione III

Prof.ssa Liliana Ardissono  
Dipartimento di Informatica  
Università di Torino

**Interfacce Utente Grafiche (GUI) – parte 2  
(basi con Java SWING)**

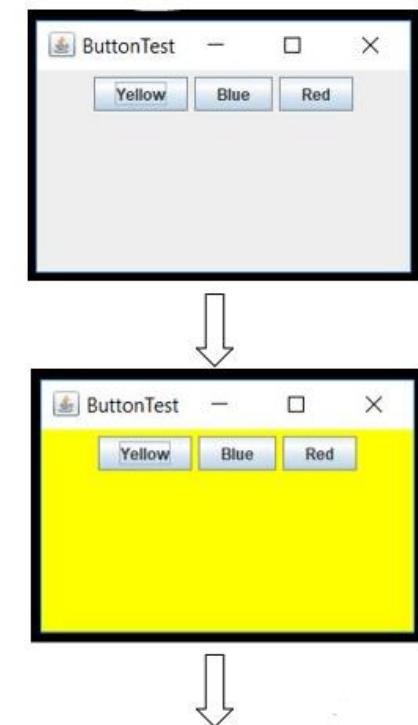
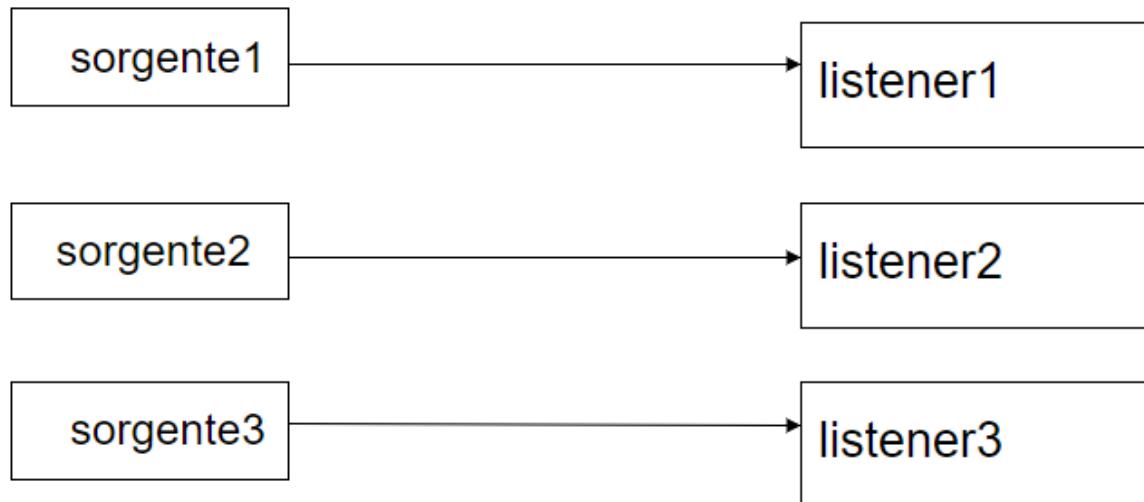


Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND



# Schemi di uso dei Listener - I

In molti casi, ogni componente grafico (widget) ha il suo listener dedicato. Per esempio, in ButtonTest (dal libro «Core Java»), posso associare un listener specifico a ogni bottone per cambiare il colore dello sfondo del pannello:



# Esempio (schema 1) – ButtonTest

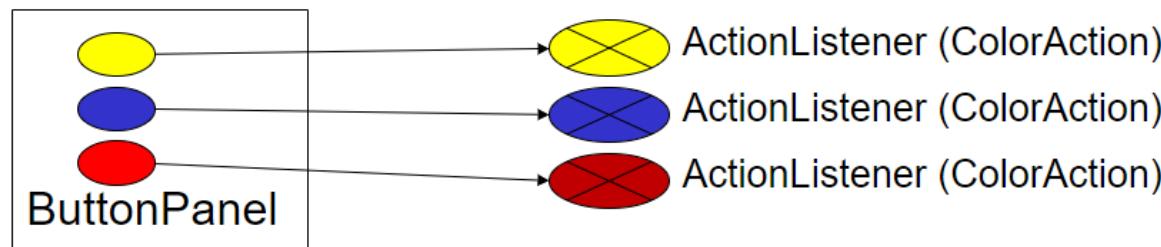


```
class ButtonPanel extends JPanel {
 public ButtonPanel() {
 JButton yellowButton = new JButton("Yellow");
 JButton blueButton = new JButton("Blue");
 JButton redButton = new JButton("Red");

 add(yellowButton); add(blueButton); add(redButton);

 }
}
```

A parte il colore, tutti e tre gli ActionListener devono fare lo stesso tipo di operazione (cambiare il colore dello sfondo) → in una prima implementazione noi possiamo definire la classe ActionListener (ColorAction) e crearne tre istanze, una per bottone, a ciascuna delle quali passiamo il colore da usare.





# ButtonTest – I

```
class ColorAction implements ActionListener {
 private Color backgroundColor;
 private ButtonPanel bp;

 public ColorAction(Color c, ButtonPanel p) {
 backgroundColor = c;
 bp = p;
 }

 public void actionPerformed(ActionEvent event) {
 bp.setBackground(backgroundColor);
 // il metodo setBackground() di JPanel cambia
 // il colore dello sfondo
 }
}
```

# ButtonTest – II



```
public ButtonPanel() {
 JButton yellowButton = new JButton("Yellow");
 JButton blueButton = new JButton("Blue");
 JButton redButton = new JButton("Red");
 add(yellowButton); add(blueButton); add(redButton);

 // il ButtonPanel deve passare se stesso come riferimento al
 // listener ColorAction per permettere la modifica del proprio colore.
 ColorAction yellowAction = new ColorAction(Color.YELLOW, this);
 ColorAction blueAction = new ColorAction(Color.BLUE, this);
 ColorAction redAction = new ColorAction(Color.RED, this);

 yellowButton.addActionListener(yellowAction);
 blueButton.addActionListener(blueAction);
 redButton.addActionListener(redAction);
}
```

# Miglioriamo il codice: Listener come classi interne degli oggetti grafici



Se **ColorAction** è una **classe interna** a **ButtonPanel**, gli oggetti **ColorAction** possono accedere ai campi e metodi, anche privati, di **ButtonPanel**:

```
class ButtonPanel extends JPanel {
```

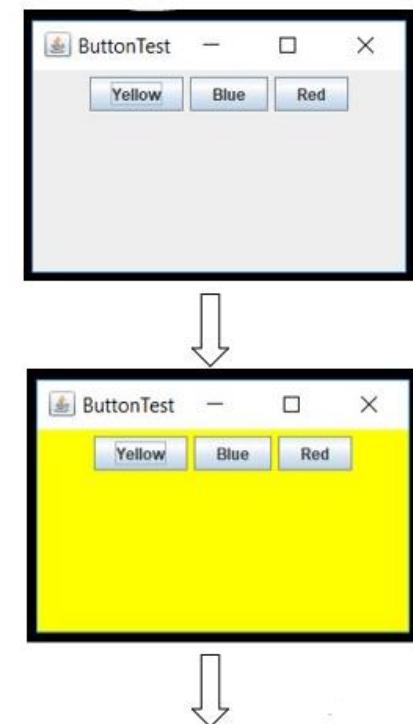
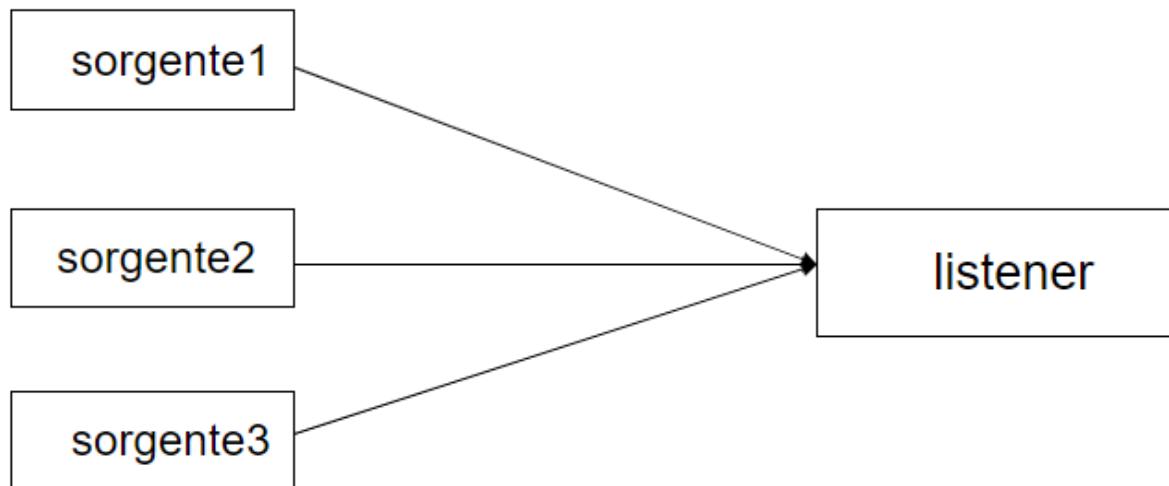
```
....
```

```
private class ColorAction implements ActionListener {
 private Color backgroundColor;
 public ColorAction(Color c) { // non devo più passare il pannello
 backgroundColor = c;
 }
 public void actionPerformed(ActionEvent event) {
 setBackground(backgroundColor);
 }
}
```

# Schemi di uso dei Listener - II



Può essere comodo definire un solo listener che gestisca gli eventi di più sorgenti di eventi **omogenee**. Per esempio, il pannello potrebbe implementare il listener degli eventi generati dai 3 bottoni, distinguendo la sorgente degli eventi attraverso il suo nome:





# Esempio (schema 2) – ButtonPanel

```
class ButtonPanel extends JPanel implements ActionListener {
 public ButtonPanel() {
 JButton yellowButton = new JButton("Yellow");
 JButton blueButton = new JButton("Blue");
 JButton redButton = new JButton("Red");
 yellowButton.addActionListener(this);
 blueButton.addActionListener(this);
 redButton.addActionListener(this);
 }

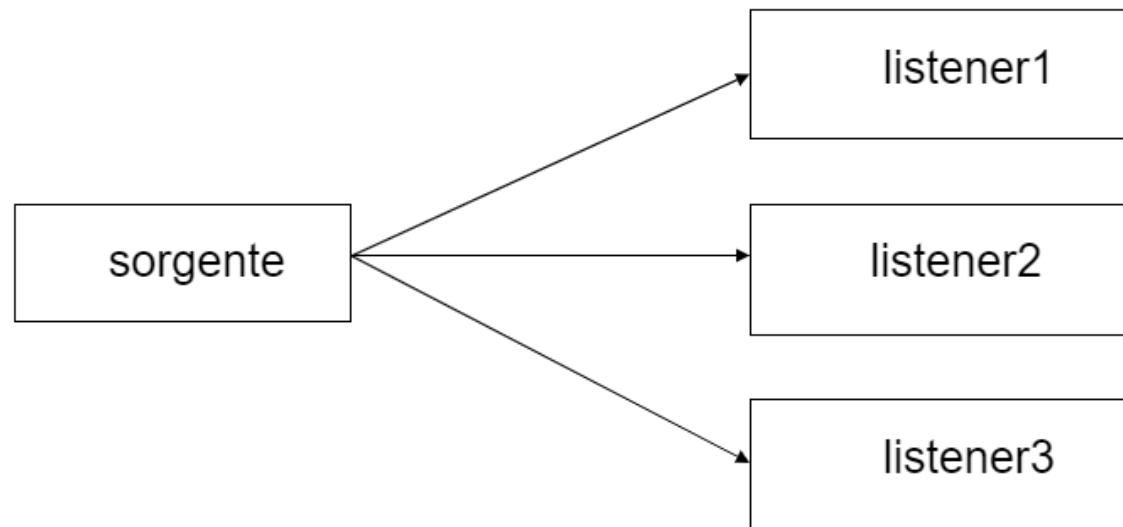
 public void actionPerformed(ActionEvent event) {
 String command = event.getActionCommand();
 if (command.equals("Yellow")) setBackground(Color.YELLOW);
 else if (command.equals("Blue")) setBackground(Color.BLUE);
 else if (command.equals("Red")) setBackground(Color.RED);
 }
}
```

# Schemi di uso dei Listener - III



Talvolta è necessario associare più di un listener alla stessa sorgente di eventi:

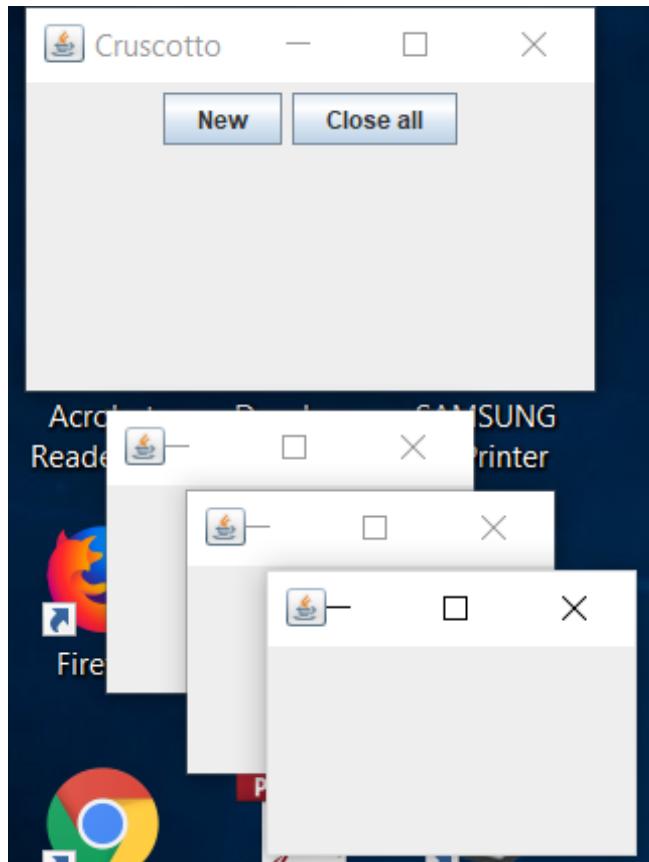
- per gestire diverse tipologie di evento generate dalla sorgente.
- per gestire in più modi gli stessi eventi (fare più cose), in parallelo.



# Esempio (schema 3) – MultiCastPanel



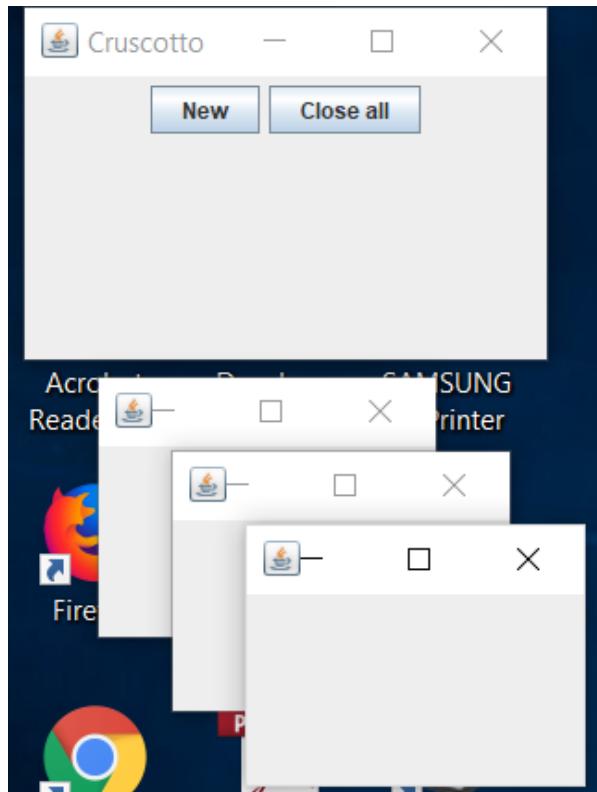
- il pulsante **New** genera una nuova finestra ogni volta che viene premuto.
- il pulsante **Close all** chiude tutte le finestre, con un solo click.



# MultiCastPanel - I



- **Un solo ActionListener ascolta gli eventi di «New» e crea una finestra per ciascun click del bottone.**
- **Ogni finestra è un diverso ActionListener di «Close all» e quando riceve l'evento di click del bottone si chiude.**





# MultiCastPanel - II

```
class MulticastPanel extends JPanel {

 public MulticastPanel() {
 JButton newButton = new JButton("New");
 add(newButton);
 final JButton closeAllButton = new JButton("Close all");
 add(closeAllButton);

 newButton.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent event) {
 BlankFrame fr = new BlankFrame(closeAllButton);
 fr.setVisible(true); }
 });
 }
}
```

# MultiCastPanel - III



```
class BlankFrame extends JFrame {
 public BlankFrame(final JButton closeButton) {
 ActionListener closeListener = new ActionListener() {
 public void actionPerformed(ActionEvent ev) {
 // rimuove il listener prima di distruggere la finestra
 closeButton.removeActionListener(closeListener);
 dispose();
 }
 };
 closeButton.addActionListener(closeListener);
 }
}
```



# Ringraziamenti

Grazie al Prof. Emerito Alberto Martelli del  
Dipartimento di Informatica dell'Università  
di Torino per aver redatto la prima  
versione di queste slides.



# Programmazione III

Prof.ssa Liliana Ardissono  
Dipartimento di Informatica  
Università di Torino

**Pattern architetturale Observer Observable**



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND



# Pattern Observer Observable

Il pattern Observer-Observable è un pattern utilizzato nelle applicazioni software per renderle modulari.

Observer Observable è molto utilizzato nelle librerie per lo sviluppo di interfacce grafiche e dobbiamo comprenderlo a fondo per capire come funzionano tali librerie.

**Observer Observable è però indipendente dalle GUI e viene utilizzato anche in altri contesti di sviluppo di applicazioni. Per esempio, ogni volta che un componente di una applicazione deve reagire autonomamente ai cambiamenti di stato di un altro componente.**

Noi vediamo Observer Observable in relazione alle GUI.



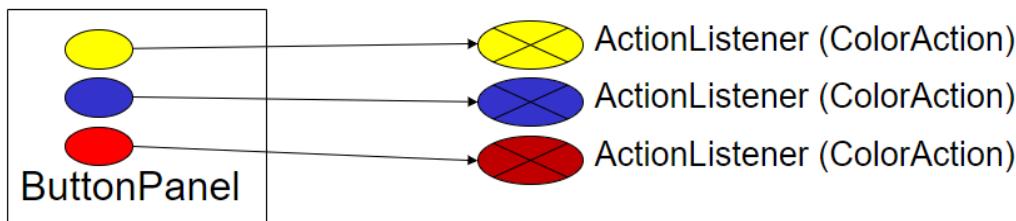
Il pattern Observer-Observable viene utilizzato nelle GUI per permettere di disaccoppiare la gestione delle azioni utente dalla gestione delle componenti delle interfacce grafiche. In pratica, si assume che:

- L'applicazione rappresenti il proprio stato in modo esplicito, attraverso opportune variabili;
- Le azioni dell'utente sull'interfaccia grafica, catturate dai Listener, cambino lo stato (cioè, il valore delle variabili) dell'applicazione;
- L'interfaccia utente osservi lo stato dell'applicazione e, se ci sono cambiamenti, reagisca modificando i dati visualizzati secondo le proprie regole interne.

Es. nel ButtonPanel (*l'esempio non è preciso – serve solo per dare un'idea intuitiva*)



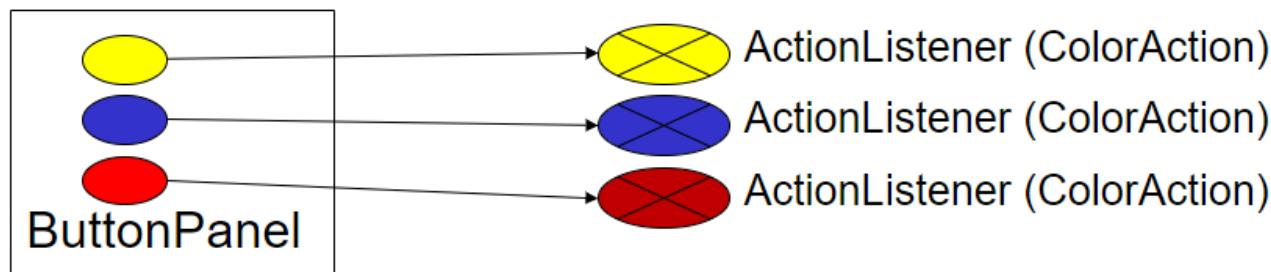
- rendiamo esplicito il colore del background, dichiarando una variabile «colore» dell'applicazione;
- quando l'utente schiaccia un bottone, il Listener del bottone modifica il valore della variabile «colore» con il valore opportuno (giallo/blu/rosso)
- Il ButtonPanel osserva i cambiamenti di valore della variabile «colore» e, a seconda del valore che essa assume, colora il proprio background opportunamente





Questo permette di:

- Semplificare i listener dei componenti grafici: i listener hanno solo il compito di interpretare le azioni utente e modificare lo stato dell'applicazione di conseguenza.
- Definire le regole di visualizzazione dell'interfaccia grafica nell'interfaccia stessa. Le regole potrebbero essere complesse, e devono accedere ai componenti dell'interfaccia utente → è bene che siano interne ad essa.





# Observer Observable in Java

Vediamo il pattern Observer Observable direttamente con esempi Java.

In Java è possibile creare degli oggetti **Observer** o **Observable** (**deprecated da JDK 10**).

Un oggetto **Observer** osserva uno o più oggetti **Observable**, registrandosi presso questi oggetti.

Quando un oggetto **Observable** modifica il proprio stato, notifica il cambiamento a tutti gli **Observer** registrati presso di lui.

La notifica consiste nella esecuzione del metodo **update** degli oggetti **Observer**.



Più precisamente:

**Observer** è un'*interfaccia* che contiene il metodo  
**update(Observable ob, Object extra\_arg)**  
che viene chiamato ogni volta che l'oggetto osservato  
**(Observable)** è modificato.

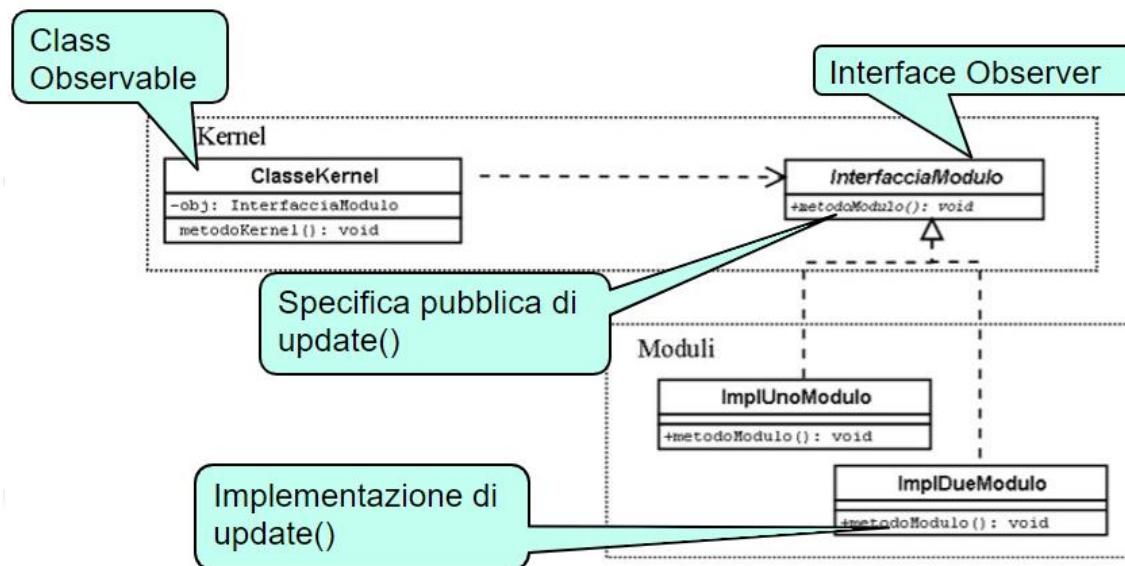
Il parametro **ob** è l'oggetto **Observable** che notifica il cambiamento

Il parametro **extra\_arg** può essere usato per passare informazione addizionale.



# Verso i pattern architetturali

Situazione simile all'esempio di Arrays e soprattutto di JButton!



**Observable** è una classe (*deprecata da Java 10*) che fornisce i metodi



### **addObserver(Observer o)**

aggiunge l'**Observer o** all'insieme degli osservatori di questo oggetto

### **setChanged()**

marca questo oggetto, indicando che il suo stato è cambiato

### **notifyObservers(Object arg)**

se questo oggetto ha cambiato stato, notifica tutti i suoi osservatori chiamando il loro metodo **update**. I due argomenti di **update** sono questo oggetto e l'oggetto **arg**.

### **notifyObservers()**

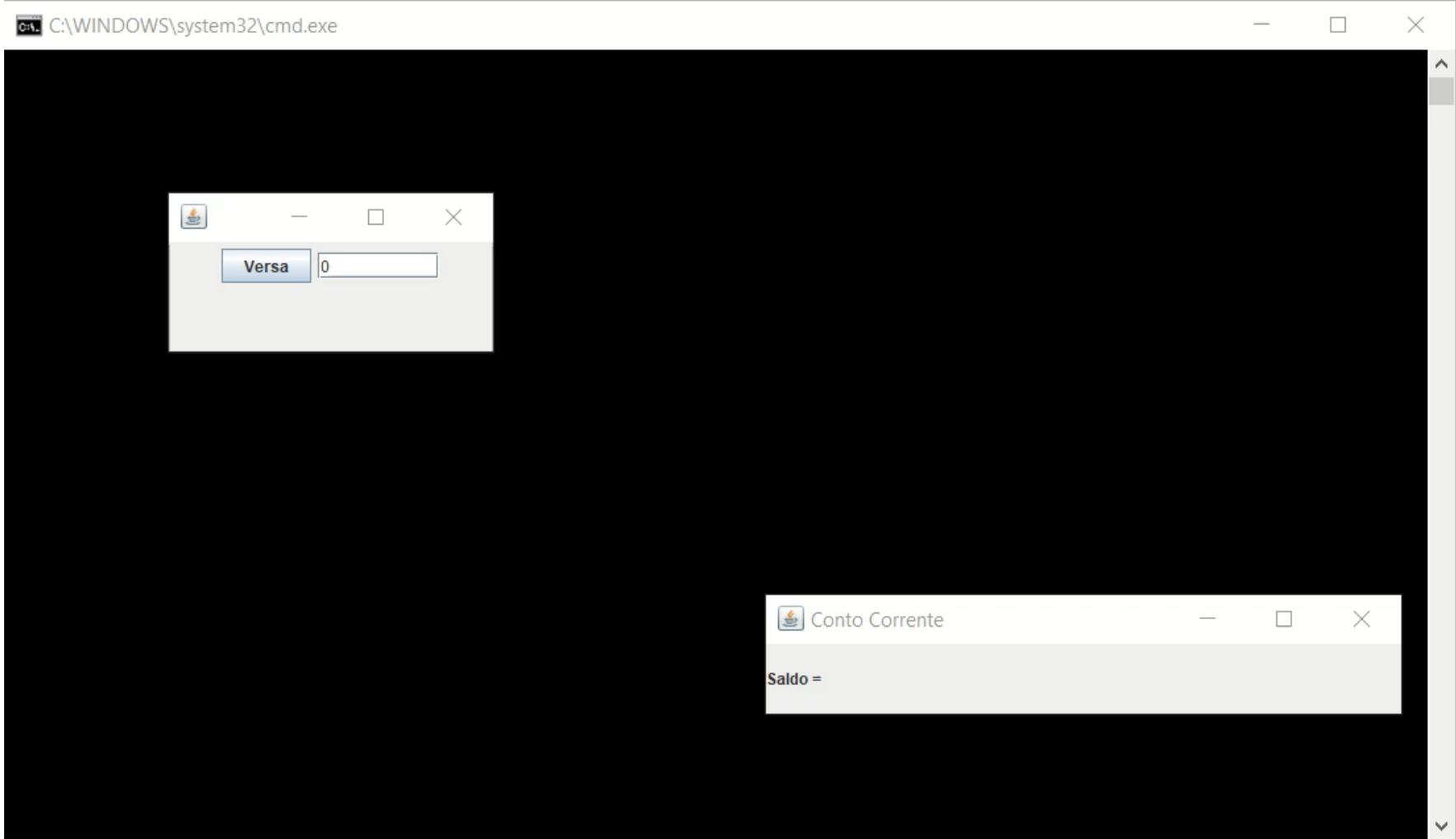
come sopra, ma il secondo argomento di **update** è **null**.



Si supponga ad esempio di avere una classe **ContoBancario** e di voler realizzare una finestra associata ad un oggetto **ContoBancario** che mostri sempre il valore aggiornato del saldo del conto.

Questo può essere realizzato definendo il conto come **Observable** e registrando la finestra come **Observer** del conto.

Ogni volta che il conto viene modificato con un versamento o un prelievo, il conto notificherà il cambiamento alla finestra eseguendone il metodo **update()**.





```
class ContoBancario extends Observable {
 private int saldo = 0;
 public void prelievo(int val)
 { saldo -= val;
 setChanged(); // marca il cambiamento di stato
 notifyObservers(); // notifica il cambiamento agli osservatori
 }
 public void versamento(int val)
 { saldo += val;
 setChanged(); // marca il cambiamento di stato
 notifyObservers(); // notifica il cambiamento agli osservatori
 }
 public int getSaldo()
 { return saldo;
 }
}
```

la finestra non conosce l'oggetto che osserva: essa chiede solo che il parametro ob di update sia un ContoBancario



```
public class Finestra extends JFrame implements Observer
{ private JLabel display;

 public Finestra() {
 display = new JLabel();
 add(display);
 display.setText("Saldo = " + 0);
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 pack();
 setVisible(true);
 }

 public void update(Observable ob, Object extra_arg) {
 if (ob!=null && ob instanceof ContoBancario) {
 display.setText("Saldo = " +
 ((ContoBancario)ob).getSaldo());
 }
 }
}
```





```
class GestisciOperazioni extends JFrame implements ActionListener
```

```
{
```

```
private JButton button;
private JTextField inputVal;
private JPanel panel;
private ContoBancario cb;
```

```
public GestisciOperazioni(ContoBancario conto) {
```

```
 super ("GestisciOperazioni");
 cb = conto;
 panel = new JPanel();
 add(panel);
 button = new JButton("Versa");
 panel.add(button);
 button.addActionListener(new ActionListener() {
 public void actionPerformed (ActionEvent e){
 int val = Integer.parseInt(inputVal.getText())
 cb.versamento(val);
 }
 });
```

```
inputVal = new JTextField("0", 8);
 panel.add(inputVal);

setLocation(100,100);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setSize(400,100);
setVisible(true);

}
```





Il file **ObserverConto** contiene un metodo *main()* che crea un **ContoBancario**, una **Finestra** collegata a questo conto e registra la Finestra presso il conto.

Viene fornita anche una interfaccia grafica **GestisciOperazioni** attraverso cui è possibile eseguire dei versamenti sul conto.

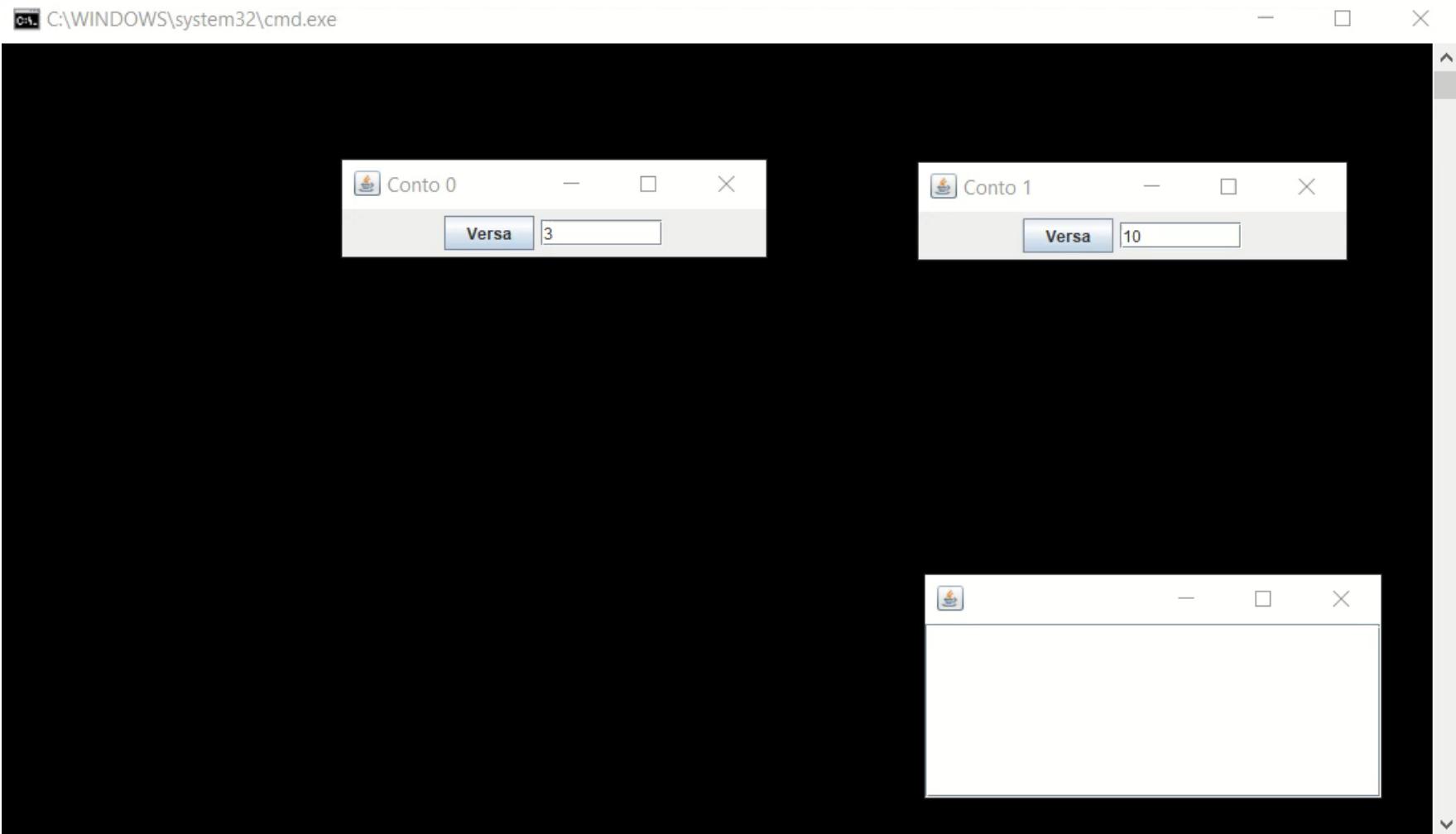
```
class ObserverConto {
 public static void main(String[] args) {
 ContoBancario cb = new ContoBancario();
 Finestra f = new Finestra();
 cb.addObserver(f); // aggancia l'osservatore all'osservato
 GestisciOperazioni v = new GestisciOperazioni(cb);
 }
}
```



La **Finestra** è completamente disaccoppiata dal conto osservato: la finestra lo conosce solo attraverso il parametro del metodo **update**.

In questo modo, per esempio, è possibile registrare la finestra presso conti diversi: la finestra può determinare l'identità del conto dinamicamente analizzando il primo parametro di **update**.

Ad esempio il file **ObserverContoApp1** realizza una finestra che osserva più conti. La finestra contiene una **JTextArea** inserita in un **JScrollPane** in cui vengono scritti numero e saldo di un conto, ogni volta che questo viene modificato.



**NB: le classi/interface Observer e Observable sono state deprecate a partire da Java10 in quanto non si usano più in modo esplicito nello sviluppo delle interfacce utente.**



Le moderne librerie per lo sviluppo di interfacce grafiche (per esempio, JavaFX) hanno classi che implementano il meccanismo internamente → il programmatore può utilizzare queste classi per

- Definire oggetti osservabili (per esempio, liste osservabili - ObsevableList)
- Definire le componenti grafiche che osservano tali oggetti (per esempio, ListView)

Tuttavia, il programmatore deve conoscere il meccanismo sottostante perché deve comunque:

- Definire i metodi di gestione delle componenti grafiche
- «Agganciare» gli osservatori agli osservati esplicitamente (solo il programmatore sa chi deve osservare chi)



# Ringraziamenti

Grazie al Prof. Emerito Alberto Martelli del Dipartimento di Informatica dell'Università di Torino per aver redatto la prima versione di queste slides.



# Programmazione III

Prof.ssa Liliana Ardissono  
Dipartimento di Informatica  
Università di Torino

**Pattern architetturale Model View Controller (MVC)**



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND



# Pattern MVC per le GUI

Il pattern MVC (Model View Controller) organizza l'architettura delle applicazioni che hanno un'interfaccia grafica (GUI) in componenti per aumentare la loro modularità e per separare le attività da svolgere nella gestione dell'interazione con l'utente.

MVC divide l'applicazione in 3 componenti principali:

- Modello (Model)
- Vista (View)
- Controllore (Controller)



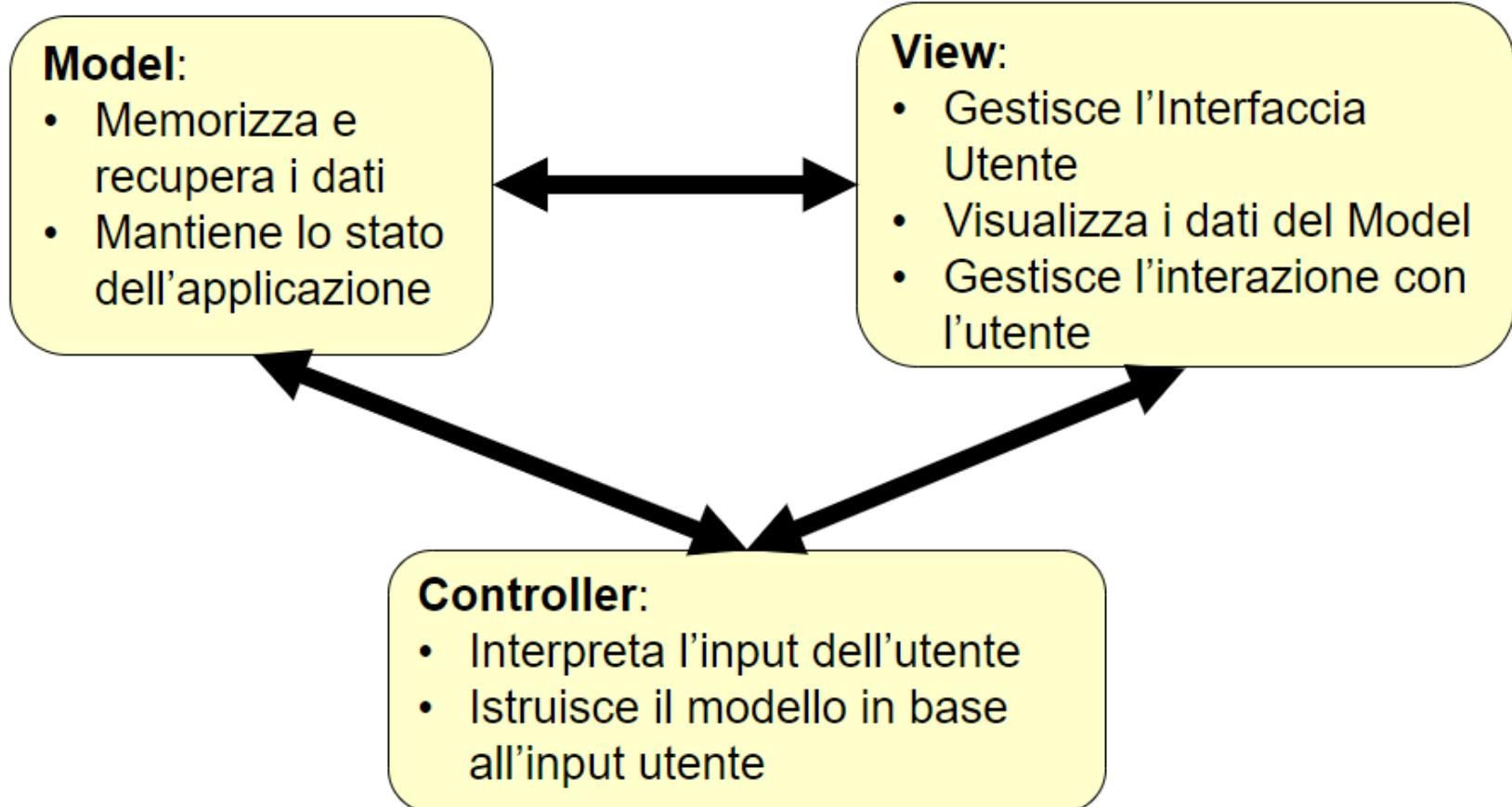
# MVC

Un programma si compone di

- **Modello (Model)**: modella e calcola il problema che desideriamo risolvere.
- **Vista (View)**: rappresenta una “fotografia” dello stato interno del modello spesso per facilitarne la sua lettura/interpretazione all’utente umano.
- **Controllore (Controller)**: controlla il flusso di dati nel programma, dalla vista al modello e quindi nuovamente alla vista.



# Pattern MVC





## MVC – flusso dei dati

- L'utente agisce sulla vista di un programma agendo su una delle sue componenti di controllo (per esempio un bottone).
- Il controllore è avvertito di tale evento ed esamina la vista per rilevarne le informazioni aggiuntive.
- Il controllore invia tali informazioni al modello che effettua la computazione richiesta e aggiorna il proprio stato interno.
- Il controllore (o il modello) richiede alla vista di visualizzare il risultato della computazione.
- La vista interroga il modello sul suo nuovo stato interno e visualizza l'informazione all'utente.



# MVC

- La computazione viene guidata dalla serie di eventi generati dall'utente tramite la GUI.
- Il programma processa gli eventi come input, aggiorna il proprio modello interno e lo visualizza tramite la vista.
- Il controllore ha il compito di gestire il flusso di eventi e dati dalla vista al modello e quindi nuovamente verso la vista.
- Più controllori, viste e modelli possono coesistere a formare un programma, se necessario.

# MVC – applicazione di gestione di un conto corrente bancario – modello dei dati

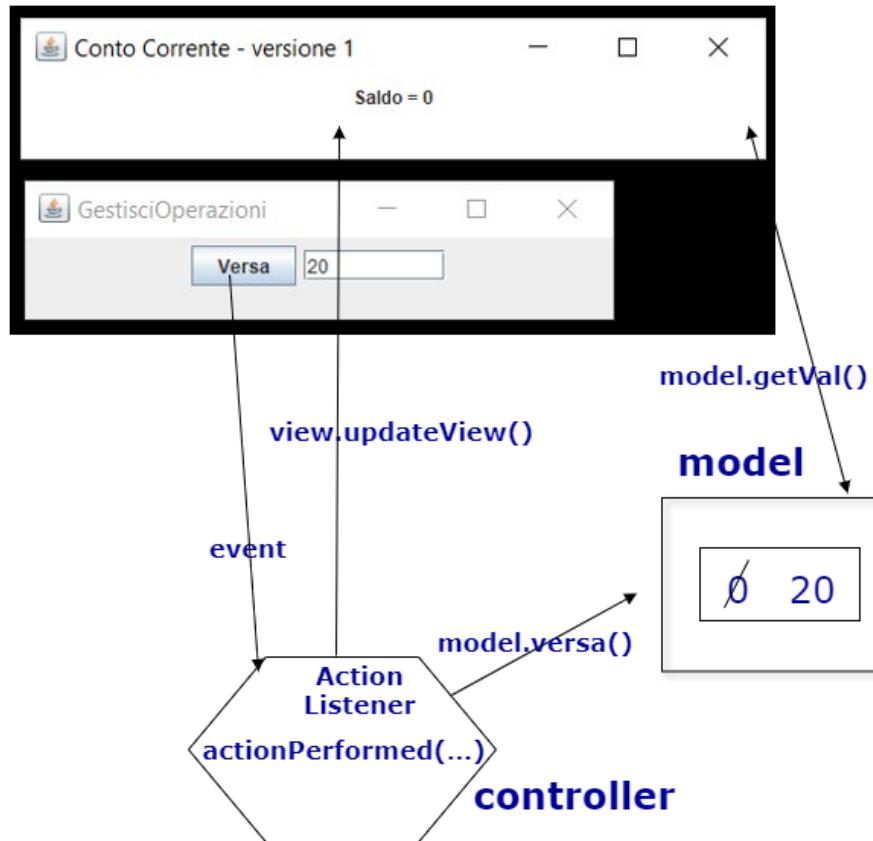
```
class ContoBancario { // Model dell'applicazione
 private int saldo;
public ContoBancario() {
 saldo = 0;
}
public void versamento(int val) {
 saldo += val;
}
public int getSaldo() {
 return saldo;
}
}
```





# MVC – esempio – senza Observer Observable - I

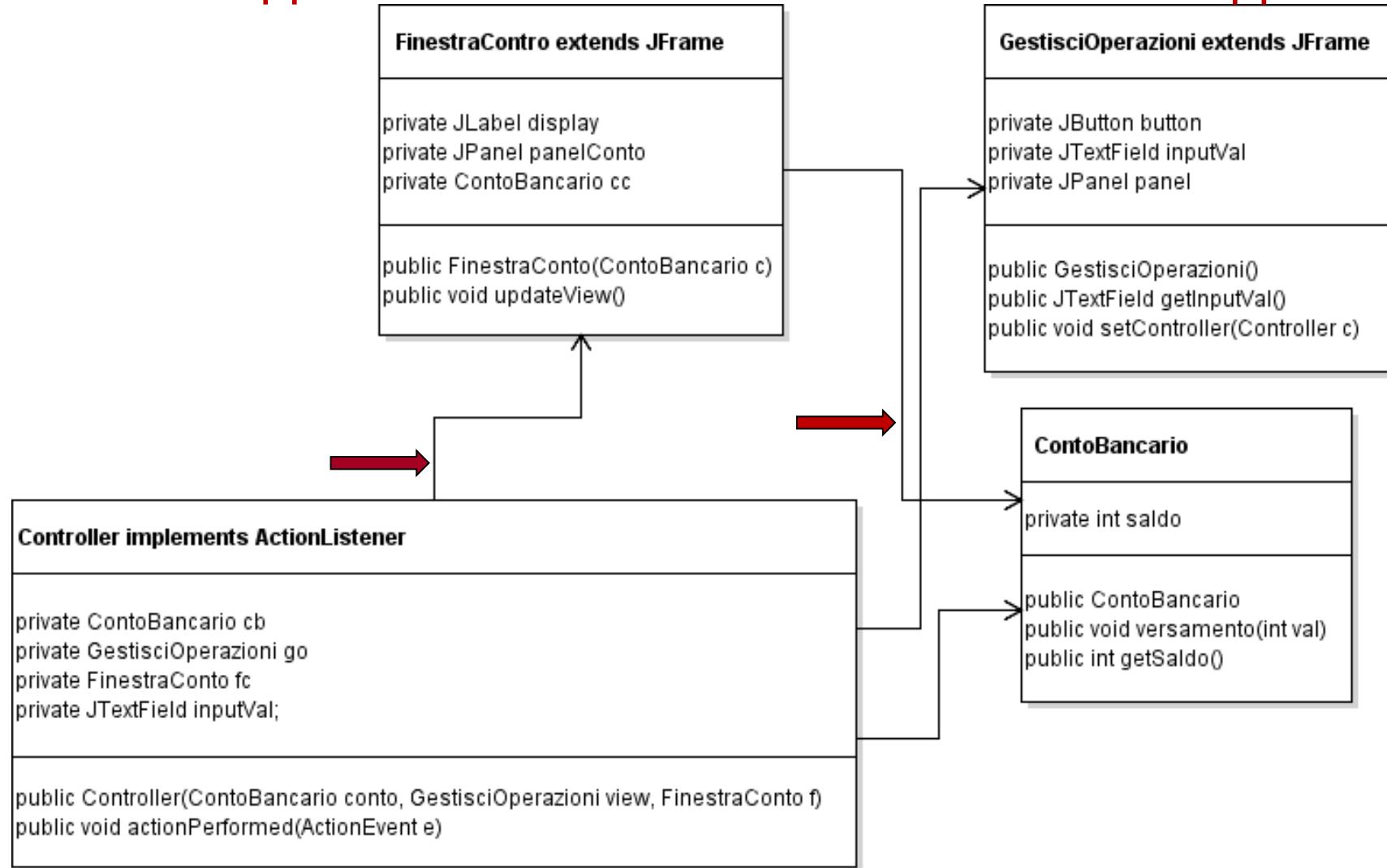
1. Utente inserisce il numero di soldi e preme “Versa”
2. l’evento è ascoltato dal controller
3. il controller invia il messaggio di versa() al modello
4. il controller invia il messaggio di updateView() alla vista
5. la vista richiede i dati al modello per aggiornarsi (getVal())



# Diagramma delle classi



Ci sono troppe relazioni tra le classi → non è disaccoppiato!!

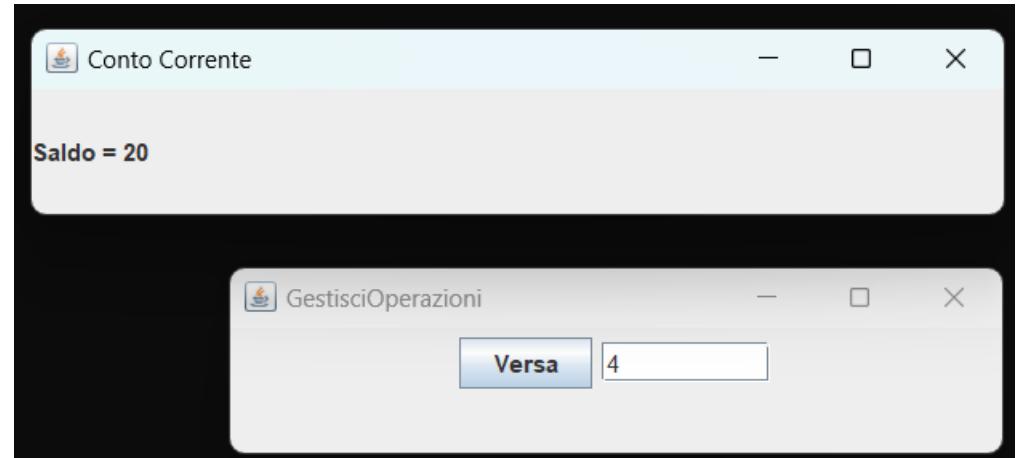


# MVC – esempio con Observer Obs. - I



**Combinando MVC con il pattern Observer Observable si disaccoppiano parzialmente le componenti:**

- la comunicazione tra il modello (osservato) e la vista del conto bancario (osservatore) viene gestita attraverso la registrazione di observers e la notifica dei cambiamenti di stato agli observers



MA: ricordate che `Observer.java` e `Observable.java` sono deprecate → voi dovrete usare le librerie grafiche nuove (e le property), descritte con Java FXML!

# MVC – Model osservabile



```
class ContoBancario extends Observable {
 private int saldo;
public ContoBancario() { saldo = 0; }
public void settaSaldoIniziale(int val) {
 saldo = val; setChanged(); notifyObservers(); }
public void prelievo(int val) {
 saldo -= val; setChanged(); notifyObservers(); }
public void versamento(int val) {
 saldo += val; setChanged(); notifyObservers(); }
public int getSaldo() { // serve per interrogare il model
 return saldo; }
}
```

# MVC – Vista 1: Osservatore

```
class FinestraConto extends JFrame implements Observer {
 private JLabel display;
public FinestraConto() {
 super("Conto Corrente");
 display = new JLabel(); display.setText("Saldo = ");
 add(display);
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 setSize(500,100); setVisible(true); }
public void update(Observable ob, Object extra_arg) {
 if (ob!=null && ob instanceof ContoBancario) {
 display.setText("Saldo = " +
 ((ContoBancario)ob).getSaldo()); }
}
}
```

# MVC – Vista 2 con Controller

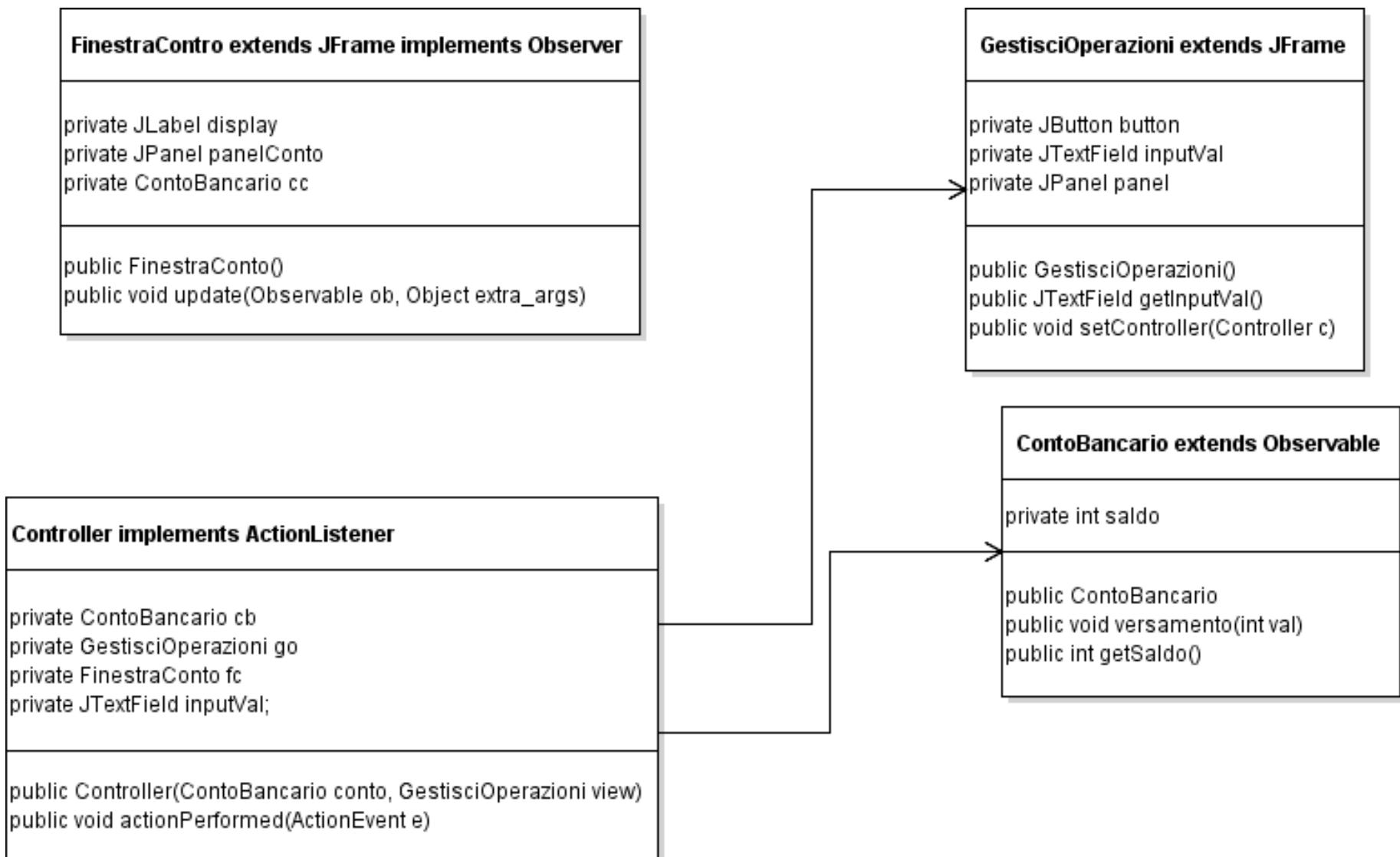
```
class GestisciOperazioni extends JFrame {
 private JButton button; private JTextField inputVal;
 private JPanel panel; private ContoBancario cb;
 public GestisciOperazioni(ContoBancario conto) {
 super("GestisciOperazioni"); cb = conto;
 panel = new JPanel(); button = new JButton("Versa"); ...
 button.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent e) {
 int val = Integer.parseInt(inputVal.getText());
 cb.versamento(val); } });
 inputVal = new JTextField("0", 8);
 panel.add(inputVal); ...
 }
}
```

# MVC – Main

```
public class ObserverContoApp2 {
 public static void main(String[] args) {
 // crea il Model (osservabile)
 ContoBancario cb = new ContoBancario();
 // crea la prima view
 FinestraConto f = new FinestraConto();
 // aggiunge l'osservatore del Model
 cb.addObserver(f);
 // crea la seconda view che conosce
 // il Model ma contiene il Controller
 GestisciOperazioni v = new GestisciOperazioni(cb);
 }
}
```



# Diagramma delle classi



# MVC – ulteriore miglioramento - I



**Si può migliorare il codice introducendo una interface per rappresentare le GUI (pattern Façade) → risulta facile cambiare la GUI senza modificare sensibilmente il programma.**

**→ INDIPENDENZA DEL CONTROLLER DALLA IMPLEMENTAZIONE DELLA VISTA**

```
interface IGestisci {
 public JTextField getInputVal();
 public void setController(Controller c);
}
class GestisciOperazioni1 extends JFrame implements IGestisci { ... }
class GestisciOperazioni2 extends JFrame implements IGestisci { ... }
```

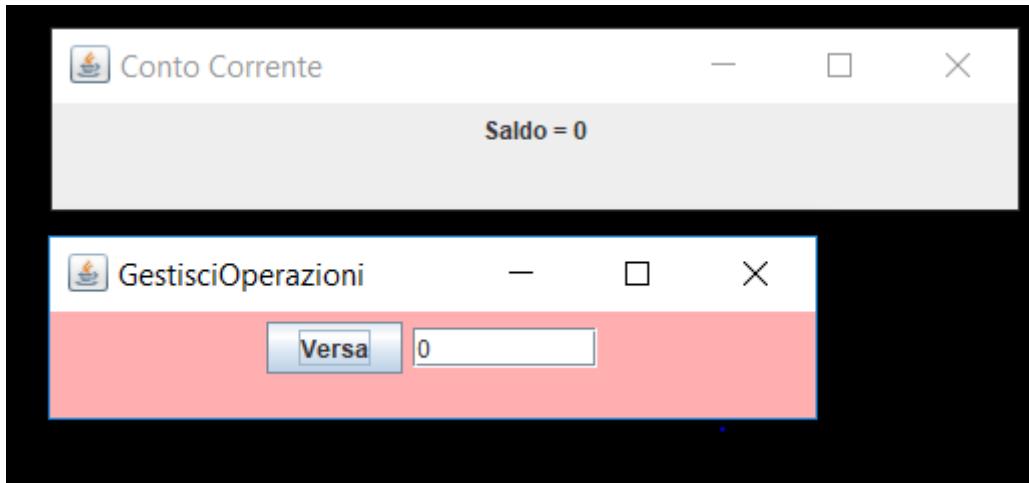
# MVC – ulteriore miglioramento - II



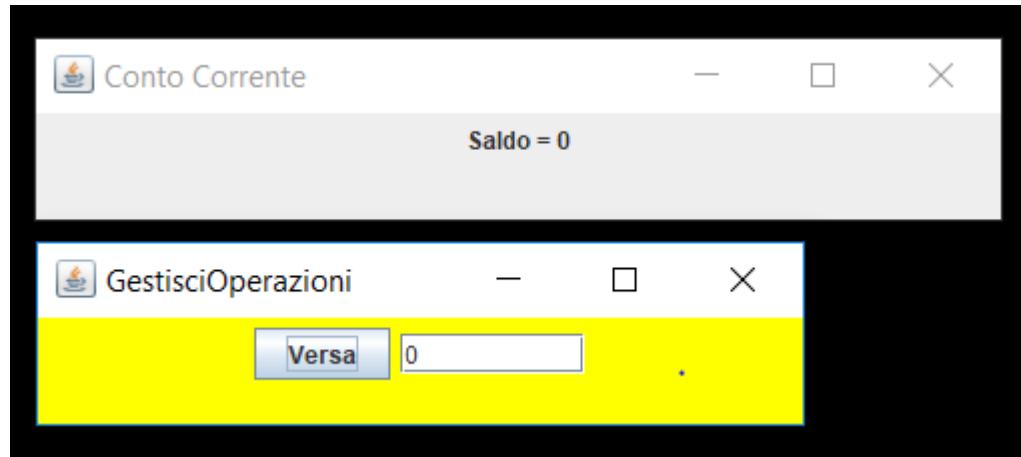
```
public class ObserverContoApp2MVC {
 public static void main(String[] args) {
 ContoBancario cb = new ContoBancario(); // modello
 FinestraConto f = new FinestraConto(); // prima vista
 cb.addObserver(f); // aggiungo la prima vista come
 osservatrice del model conto corrente bancario
 cb.settaSaldoIniziale(0);

 //IGestisci v = new GestisciOperazioni1(); // prima vista
 IGestisci v = new GestisciOperazioni2(); // seconda vista
 Controller c = new Controller(cb, v); // controller
 v.setController(c); // aggancio il controller alla vista
 }
}
```

# MVC – ulteriore miglioramento - III

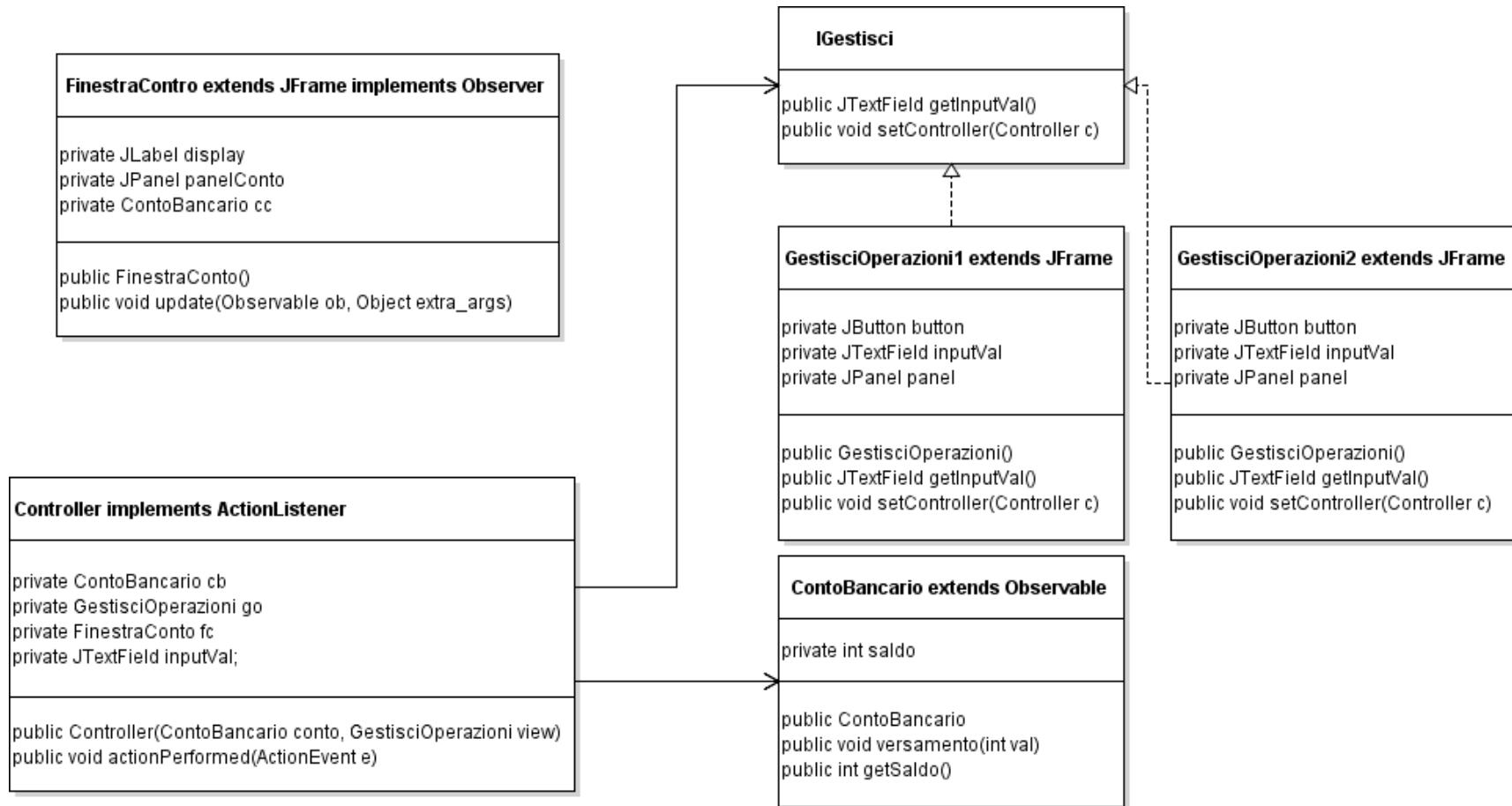


Siccome il Controller richiede un **IGestisci**, si può passare come parametro la GUI desiderata cambiando solo una riga di codice nel main() dell'applicazione. Il Controller non deve cambiare codice e controlla GUI diverse basandosi sui loro API.





# Diagramma delle classi dell'esempio



Vedere l'applicazione `ObserverContoApp2MVC`



# MVC - vantaggi

- Le classi che formano l'applicativo possono essere più facilmente riutilizzate
- L'applicativo è organizzato in parti semplici e comprensibili (ogni parte ha le sue specifiche finalità)
- La modifica di una parte non coinvolge e non interferisce con le altre parti (maggiore flessibilità nella manutenzione del software)



# Ringraziamenti

Grazie al Prof. Emerito Alberto Martelli del  
Dipartimento di Informatica dell'Università  
di Torino per aver redatto la prima  
versione di queste slides.



# Programmazione III

Prof.ssa Liliana Ardissono  
Dipartimento di Informatica  
Università di Torino

**Interfacce Utente Grafiche (GUI)**  
**Overview di JavaFX – parte 2**  
**JavaFXML**



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND

# E' possibile semplificare il codice delle GUI



La semplificazione si basa su due concetti:

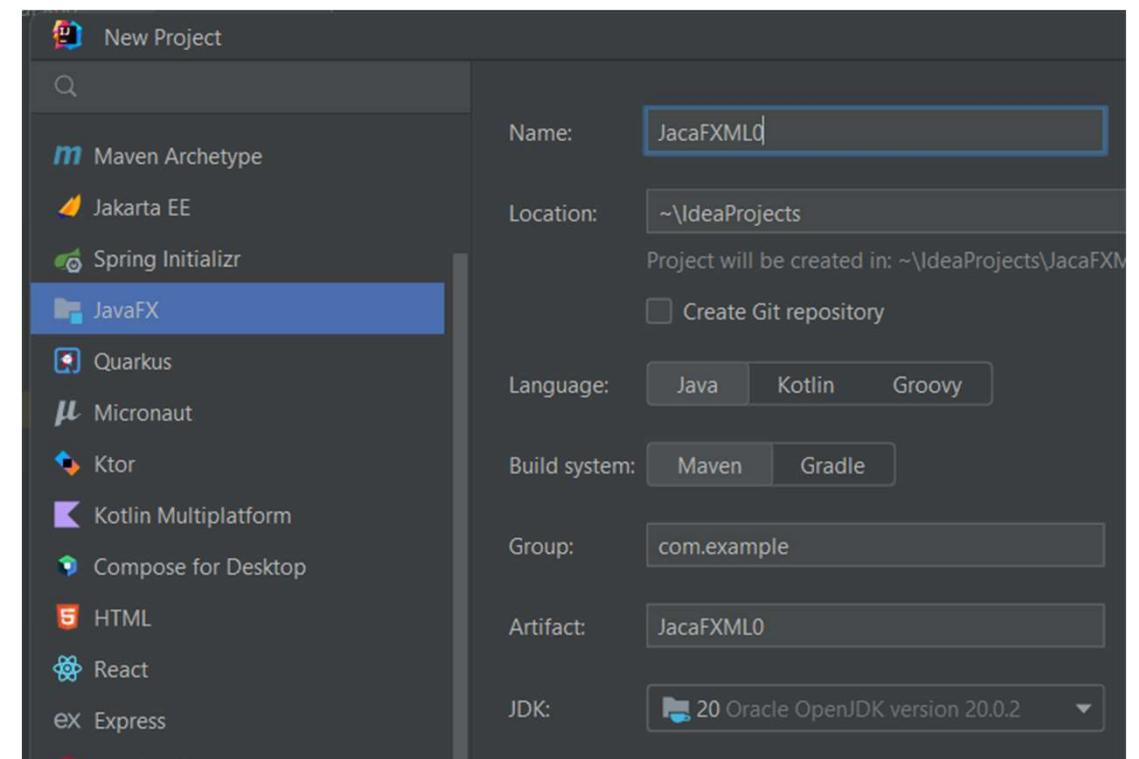
- **Definizione dichiarativa dell'interfaccia grafica, attraverso il linguaggio XML (JavaFXML)** – Evita di sviluppare le componenti grafiche pezzo per pezzo!
- **Uso di java beans e properties** per
  1. Caratterizzare i dati del model in modo standard
  2. Legare i dati del model tramite binding alle componenti dell'interfaccia grafica (o a componenti di osservatori, in generale) che devono reagire ai cambiamenti di valore in automatico



# PASSO 1 - XML

Impariamo a definire le interfacce grafiche in modo dichiarativo, utilizzando il linguaggio XML per specificare i componenti delle GUI:

JavaFXML



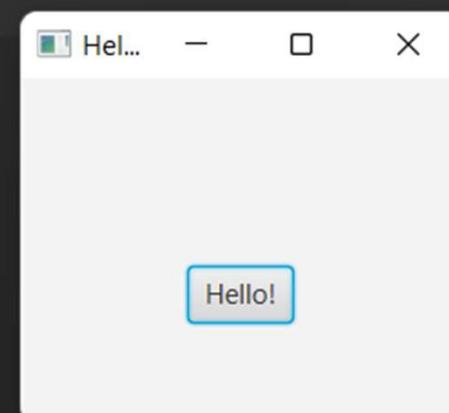
# Esempio – applicazione JavaFXML0



La creiamo come new Project JavaFX, nasce già organizzata con architettura MVC (ma il Model non c'è)

```
public class HelloApplication extends Application {
 @Override
 public void start(Stage stage) throws IOException {
 FXMLLoader fxmlLoader = new FXMLLoader(HelloApplication.class.getResource("Hello.fxml"));
 Scene scene = new Scene(fxmlLoader.load(), 320, 240);
 stage.setTitle("Hello!");
 stage.setScene(scene);
 stage.show();
 }

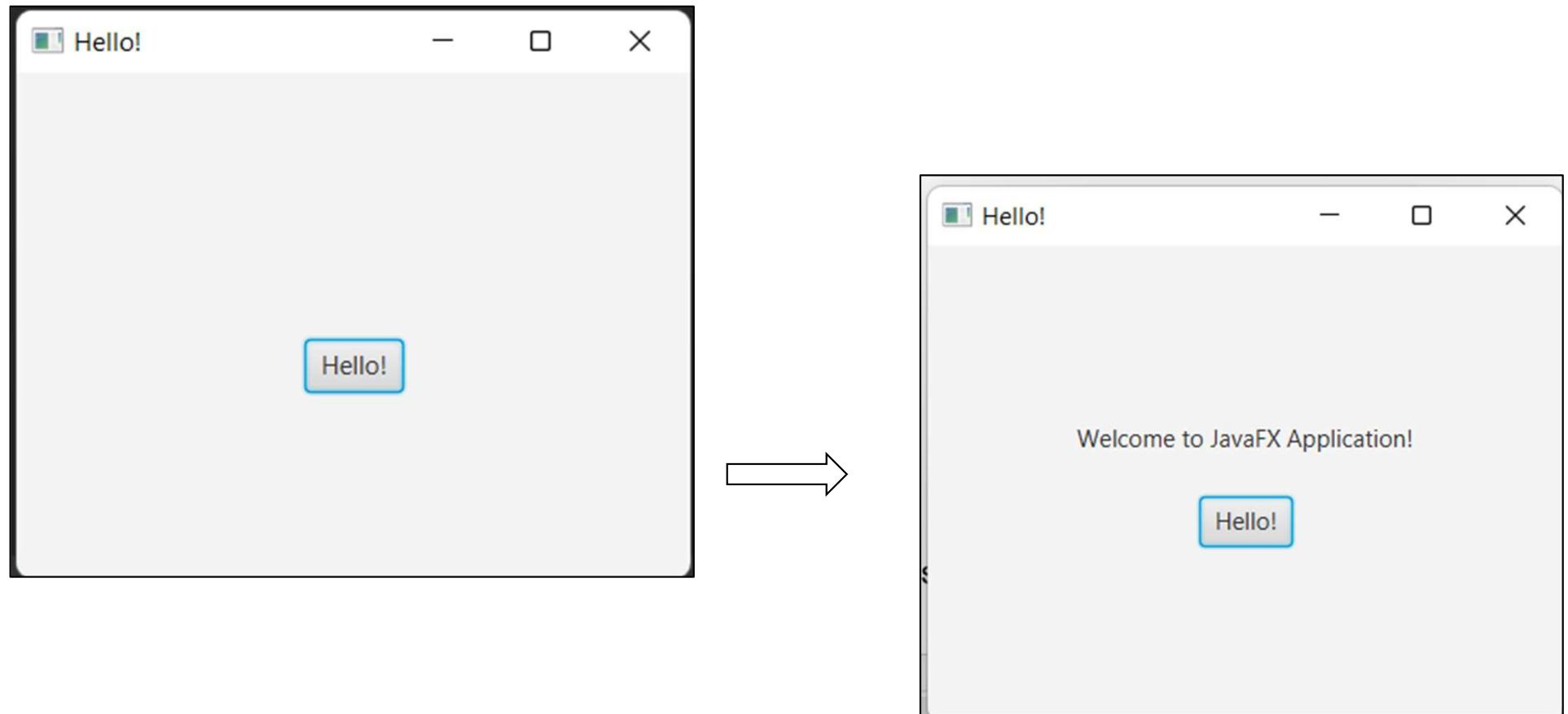
 public static void main(String[] args) { launch(); }
}
```



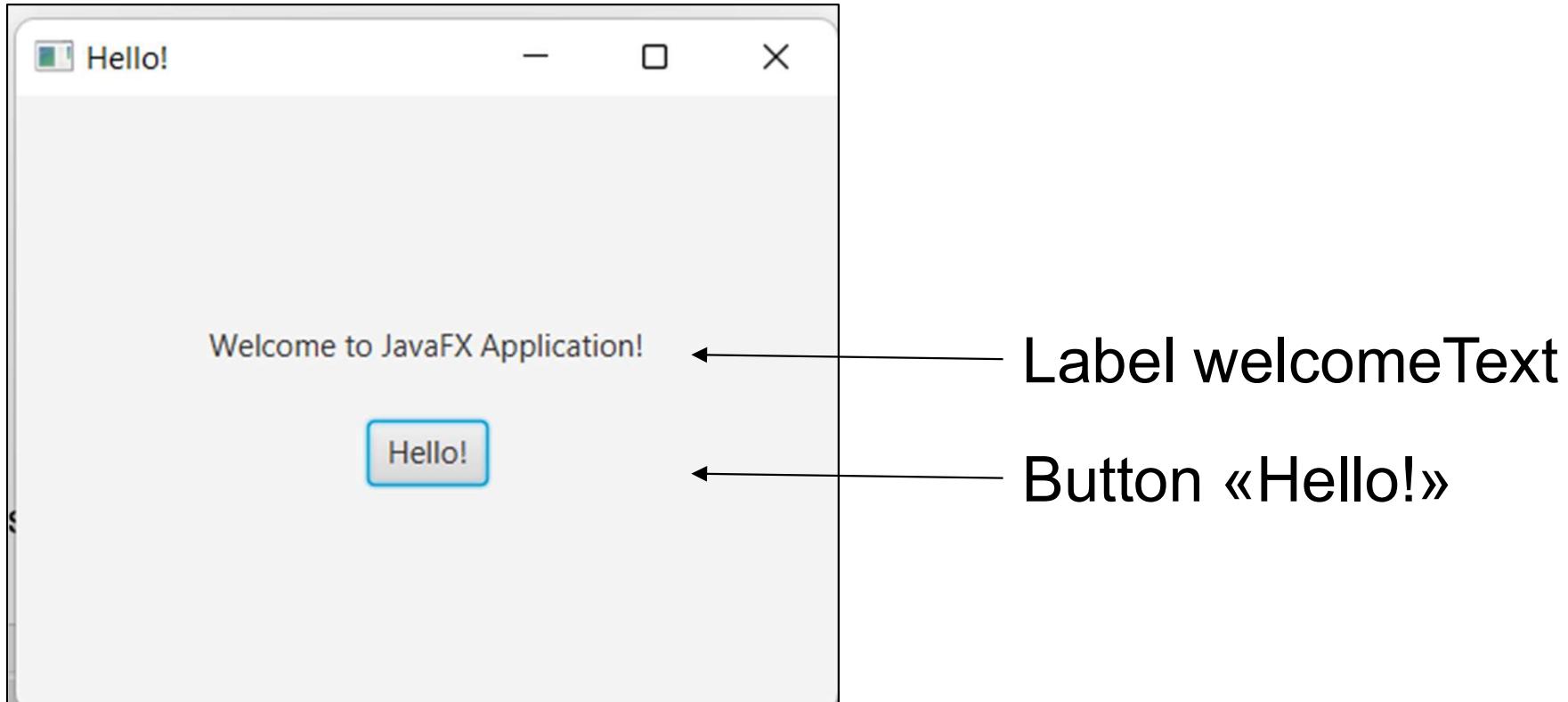
# JavaFXML0 - funzionamento



Cliccando il bottone appare la scritta “Welcome...”



# JavaFXML0 - ID dei componenti della GUI





# JavaFXML0 – hello-view.fxml

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.geometry.Insets?>
... altre import per far funzionare i componenti grafici della view
<VBox alignment="CENTER" spacing="20.0"
xmlns:fx="http://javafx.com/fxml"
 fx:controller="com.example.javafxml0.HelloController">
 <padding>
 <Insets bottom="20.0" left="20.0" right="20.0" top="20.0"/>
 </padding>
 <Label fx:id="welcomeText"/>
 <Button text="Hello!" onAction="#onHelloButtonClick"/>
</VBox>
```

# JavaFXML0 – Il controller



```
// il Controller usa le Java annotations per inizializzare le variabili in automatico
// il Controller si basa sugli ID dati alle componenti grafiche (welcomeText)
```

```
public class HelloController {
 @FXML
 private Label welcomeText;

 @FXML
 protected void onHelloButtonClick() {
 welcomeText.setText("Welcome to JavaFX Application!");
 }
}
```

# JavaFXML0 – Applicazione (main)



```
public class HelloApplication extends Application {
 @Override
 public void start(Stage stage) throws IOException {
 FXMLLoader fxmlLoader =
 new FXMLLoader(HelloApplication.class.getResource("hello-view.fxml"));
 Scene scene = new Scene(fxmlLoader.load(), 320, 240);
 stage.setTitle("Hello!");
 stage.setScene(scene);
 stage.show();
 }
 public static void main(String[] args) {
 launch();
 }
}
```



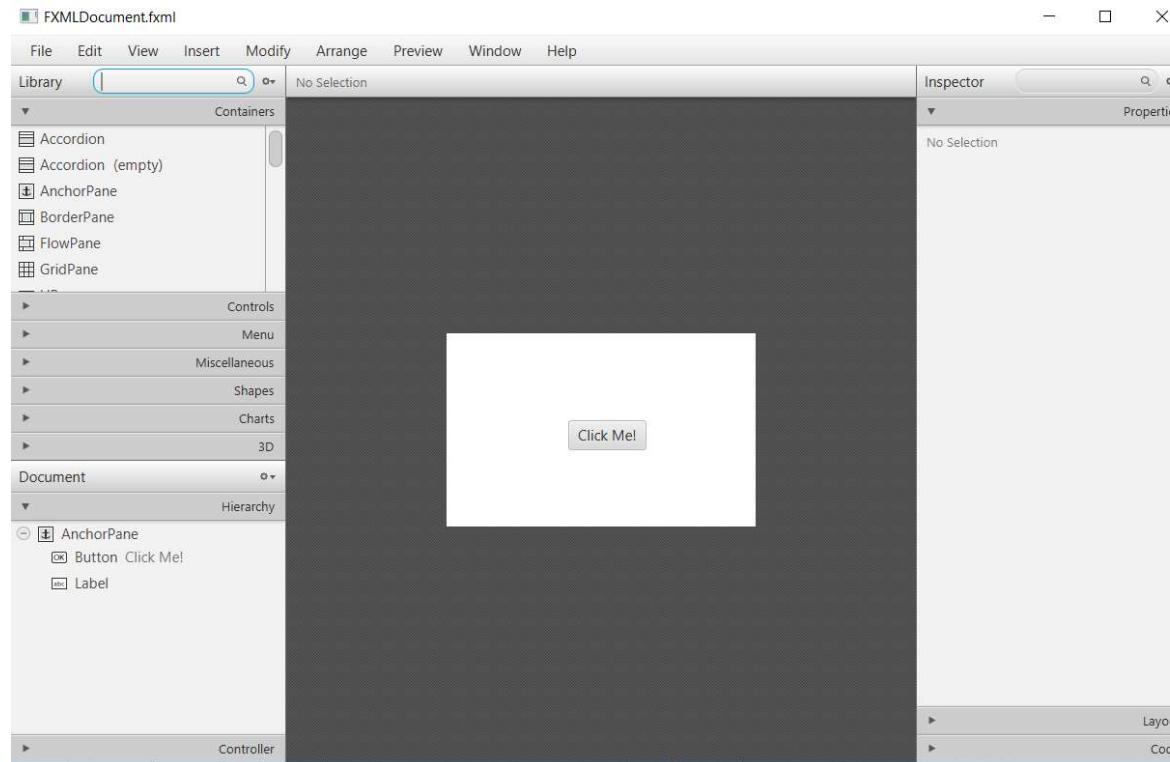
Per evitare di scrivere i file XML a mano:

- Utilizzare Scene Builder come strumento grafico per il design dell'interfaccia utente → Scene Builder genera automaticamente i file XML che specificano le componenti dell'interfaccia e la loro posizione nella finestra dell'applicazione
- Scene Builder: scaricare l'installazione e installare il software sul proprio computer.
- Aprire i documenti FXML con SceneBuilder e modificarli nelle loro parti grafiche

# Scene Builder – modifica della GUI



- Aprire il file XML che definisce la GUI su cui si deve lavorare



- Poi, drag and drop (e ridimensionamento) delle componenti grafiche  
→ SceneBuilder modifica il documento XML che descrive la GUI (FXMLDocument.fxml) per includere le nuove componenti secondo il layout specificato graficamente (salvare!)



# Come aggiornare il codice dell'applicazione alla nuova GUI XML? – I

Il codice generato da SceneBuilder descrive i componenti grafici ma **manca la logica di gestione dell'interfaccia** (per es., l'aggancio al controller)

→ bisogna completare il codice con gli ID delle componenti grafiche e i gestori degli eventi. Per esempio:

```
<Label fx:id="welcomeText"/>
```

```
<Button text="Hello!" onAction="#onHelloButtonClick"/>
```

dove onHelloButtonClick è il metodo definito nel Controller.

Per apportare modifiche al codice XML bisogna modificare direttamente il codice XML da IDE (o da SceneBuilder attraverso il pannello di destra).

**NB: gli id permettono anche di agganciare le regole di un foglio stile ai singoli componenti dell'interfaccia grafica**

# Come aggiornare il codice dell'applicazione alla nuova GUI XML? - II



Nel Controller, inserite le dichiarazioni delle variabili globali da utilizzare, con le java annotation. I nomi delle variabili devono corrispondere agli ID specificati nella GUI (inoltre il controller può definire variabili locali). **Il controller inizializza automaticamente le variabili globali dichiarate** → esse possono essere usate nel metodo di gestione degli eventi o in altri metodi della classe. Es:

**@FXML**

```
private Label welcomeText;
```

**@FXML**

```
protected void onHelloButtonClick() {
```

```
 welcomeText.setText("Welcome to JavaFX Application!");
```

```
}
```



# Programmazione III

Prof.ssa Liliana Ardissono  
Dipartimento di Informatica  
Università di Torino

**Interfacce Utente Grafiche (GUI) – parte 2  
(basi con Java SWING)**  
**Implementazione della gestione di eventi**

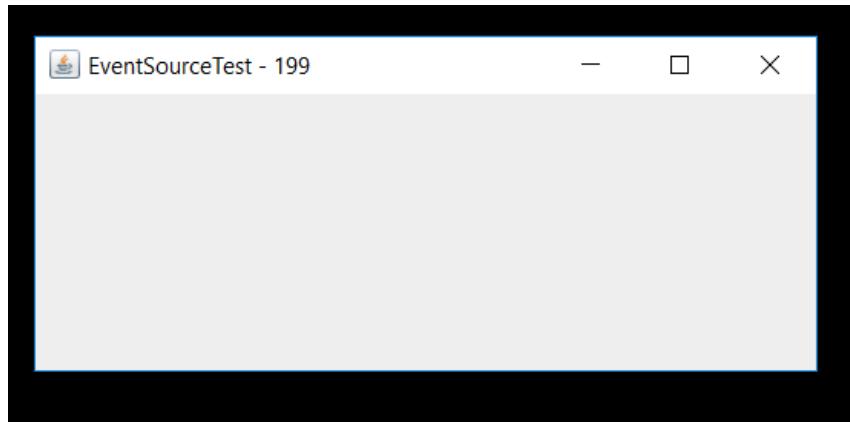


Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND

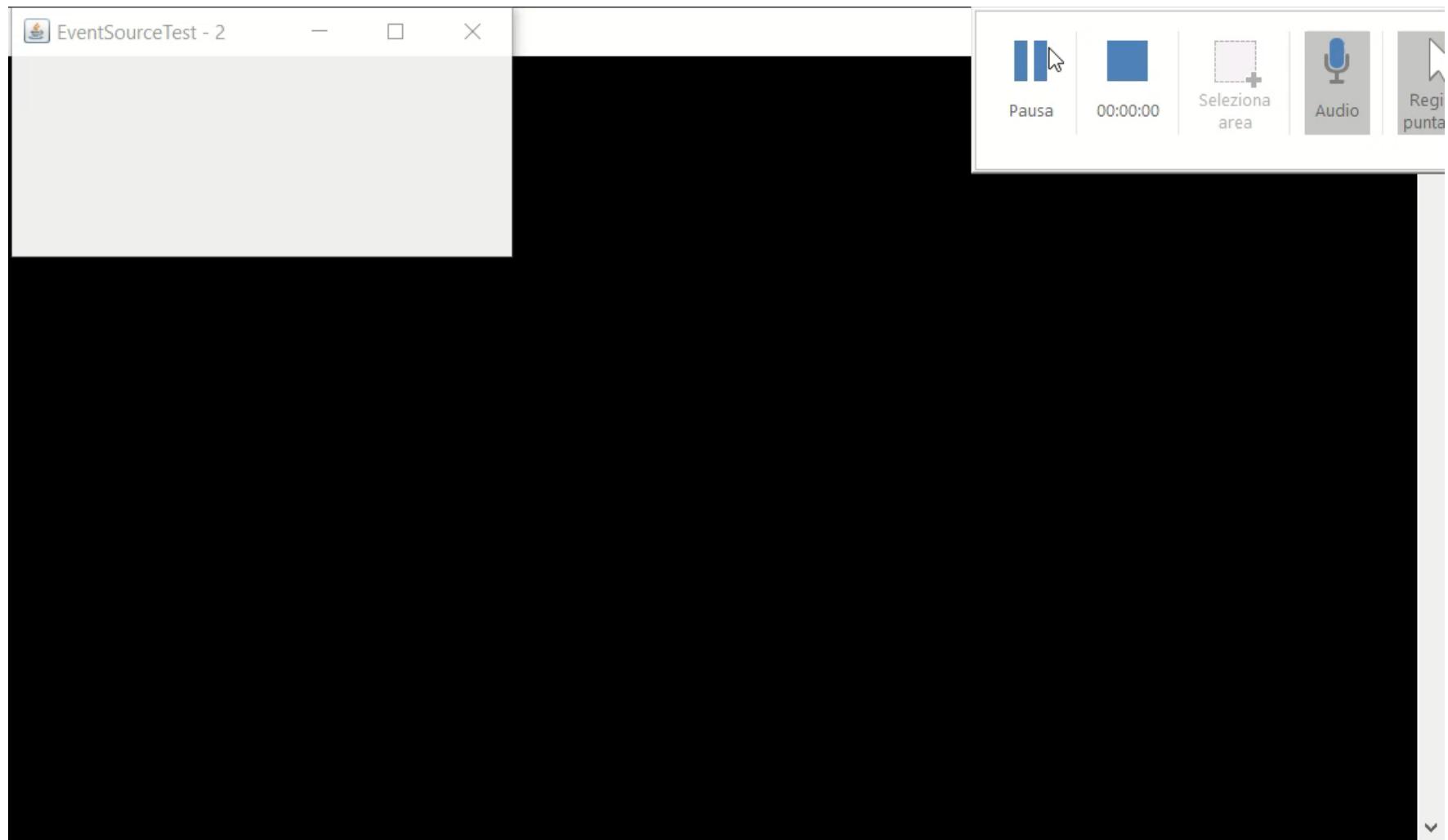


Per capire meglio il meccanismo della gestione degli eventi vediamo un esempio **EventSourceTest** da Core Java che usa eventi diversi da quelli standard offerti dalle componenti grafiche → devono essere gestiti esplicitamente nel codice delle componenti grafiche.

Data una finestra, vogliamo contare quante volte il metodo **paintComponent()** è chiamato. Per farlo, ad ogni invocazione facciamo in modo che la finestra generi un evento e lo invii a un *listener*, che provvede ad aggiornare il titolo della finestra.



In questo esempio la finestra è stata ridimensionata e questo ha fatto eseguire `paintComponent()` 199 volte





Il metodo **paintComponent()** è chiamato automaticamente ogni volta che una finestra deve essere ridisegnata: ad es. quando si cambia la dimensione, quando la finestra è coperta da un'altra.

## JPanel



Esegue `paintComponent()` a ogni ridimensionamento del pannello, o quando si copre la finestra con un'altra finestra



Possiamo quindi utilizzare il metodo `paintComponent()` per far lanciare un evento a ogni sua invocazione

## **PaintCountPanel extends JPanel**

Esegue `paintComponent()`

Lancia evento da gestire

Listener: fa visualizzare i dati



# EventSourceTest - III

Definiamo la classe **PaintCountPanel** che è il pannello che genera gli eventi «non standard».

**PaintCountPanel extends JPanel**

Ci servono tre ingredienti:

- il **tipo degli eventi** generati dal **PaintCountPanel**
- l'**interfaccia del *listener*** per questo tipo di eventi
- i **metodi per aggiungere (registrare) o togliere i *listener*** al PaintCountPanel (che è la sorgente degli eventi)

E poi le modifiche del metodo di `paintComponent()` per far generare gli eventi.

# EventSourceTest - IV



Come tipo di evento da lanciare scegliamo  
**PropertyChangeEvent, appartenente alla libreria degli eventi Java** (classe di eventi predefinita)

**class PropertyChangeEvent** - descrive gli eventi lanciati tutte le volte che un bean (qui il JPanel) cambia il valore di una sua property.

**Costruttore:**

**PropertyChangeEvent(Object source,**

**String propertyName, Object oldValue, Object newValue)**

- Source è la sorgente dell'evento
- oldValue il valore precedente della property
- newValue il valore attuale della property (dopo il cambiamento)



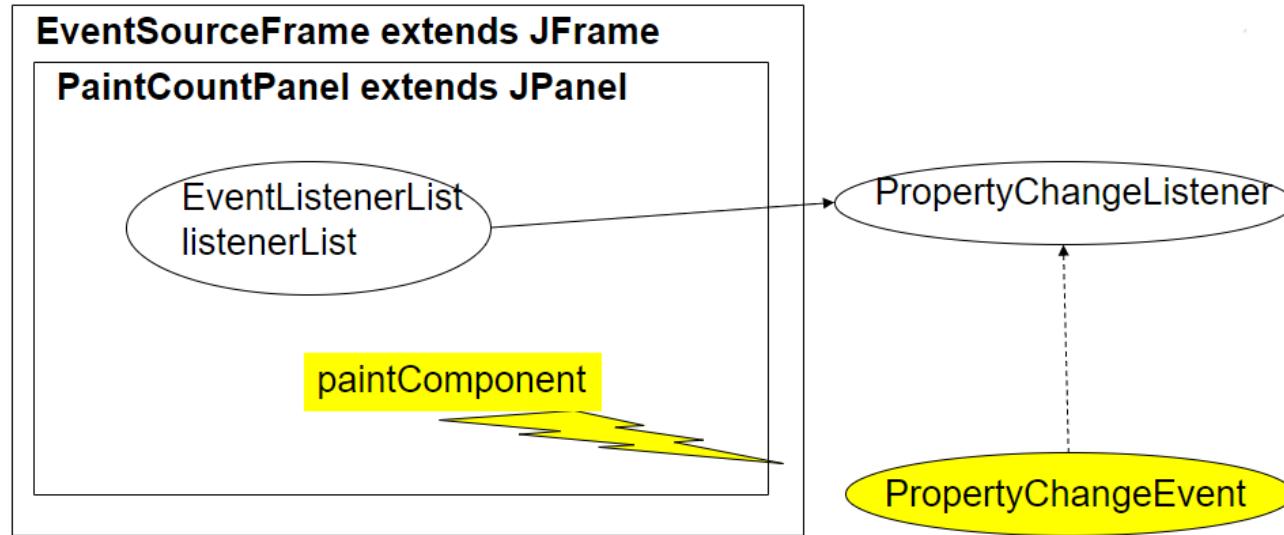
Il listener dei PropertyChangeEvent è il  
PropertyChangeListener (Interface)

```
Interface PropertyChangeListener {
 propertyChange(PropertyChangeEvent event)
}
```

**Metodi per aggiungere e togliere listener al  
PaintCountPanel** - occorre definire in PaintCountPanel i  
metodi:

addPropertyChangeListener(PropertyChangeListener l)

removePropertyChangeListener(PropertyChangeListener l)



Per gestire la lista dei *listener*, la classe **JPanel** ha un campo **listenerList** di tipo **EventListenerList**, ereditato da **JComponent**. La classe **EventListenerList**, fornita dalle librerie di Java, serve per tenere tutti i *listener* (di qualunque tipo) associati a una sorgente di eventi.



## EventListenerList: metodi (da API Java)

**public <T extends EventListener> T[] getListeners(Class<T> t):**

restituisce un array di tutti i *listener* di tipo t

**public <T extends EventListener> void add(Class<T> t, T l):**

aggiunge il listener l di tipo t alla lista dei listeners

**public <T extends EventListener> void remove(Class<T> t, T l):**

toglie il listener l di tipo t dalla lista dei listeners

# PaintCountPanel - I



Definiamo i metodi per aggiungere o togliere i  
*listener* ad un **PaintCountPanel**:

```
public void addPropertyChangeListener(PropertyChangeListener l) {
 listenerList.add(PropertyChangeListener.class, l);
}
```

```
public void removePropertyChangeListener(PropertyChangeListener l) {
 listenerList.remove(PropertyChangeListener.class, l);
}
```

Il listener è definito come classe anonima che viene registrata nel panel:

```
PropertyChangeEvent(Object source,
 String propertyName, Object oldValue, Object newValue)
```

```
class EventSourceFrame extends JFrame {
```



```
public EventSourceFrame() {
 setTitle("EventSourceTest");
 setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

 final PaintCountPanel panel = new PaintCountPanel();
 add(panel);

 panel.addPropertyChangeListener(new
 PropertyChangeListener() {
 public void propertyChange(PropertyChangeEvent ev){
 setTitle("EventSourceTest - " + ev.getNewValue());
 }
 });
}
```

# PaintCountPanel – II



PaintCountPanel deve generare un evento ogni volta che si esegue `paintComponent()`, e inviare questo evento ai propri listener → dobbiamo ridefinire il metodo `paintComponent(Graphics g)`

```
class PaintCountPanel extends JPanel {
 private int paintCount; // conta il numero di «refresh» del panel
 @override
 public void paintComponent(Graphics g) {
 int oldPaintCount = paintCount;
 paintCount++;
 firePropertyChangeEvent(new PropertyChangeEvent(this,
 "paintCount", oldPaintCount, paintCount));
 super.paintComponent(g);
 }
 public void addPropertyChangeListener(PropertyChangeListener l) {
 listenerList.add(PropertyChangeListener.class, l);
 }
 public void removePropertyChangeListener(PropertyChangeListener l) {
 listenerList.remove(PropertyChangeListener.class, l);
 }
 // continua...
```



# PaintCountPanel – III

// continua...

```
public void firePropertyChangeEvent(PropertyChangeEvent event) {
 EventListener[] listeners =
 listenerList.getListeners(PropertyChangeListener.class);
 for (EventListener l : listeners) {
 ((PropertyChangeListener) l).propertyChange(event);
 }
} // end firePropertyChangeEvent
} // end PaintCountPanel
```



# Ringraziamenti

Grazie al Prof. Emerito Alberto Martelli del Dipartimento di Informatica dell'Università di Torino per aver redatto la prima versione di queste slides.



# Programmazione III

Prof.ssa Liliana Ardissono  
Dipartimento di Informatica  
Università di Torino

## Interfacce Utente Grafiche (GUI) Overview di JavaFX



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND



# JavaFX - I

JavaFX è una libreria grafica di Java per lo sviluppo di GUI in applicazioni stand-alone.

- **Per usare JavaFX serve aver compreso SWING:** SWING fornisce i concetti fondamentali di componente, listener, etc. in modo «didattico».
- Nella programmazione delle GUI, JavaFX separa il contenuto dalla sua visualizzazione tramite **fogli stile CSS** (simile a HTML).
- JavaFX permette il **binding di property dei Model** con elementi dell'interfaccia utente per aggiornare automaticamente le viste.
- JavaFX offre le **classi/interface che implementano Observer Observable** e non sono deprecated.
- JavaFX(ML) permette anche di **scrivere le GUI con XML**.



## JavaFX - II

- Tutorials:  
[https://docs.oracle.com/javafx/2/get\\_started/jfxpub-get\\_started.htm](https://docs.oracle.com/javafx/2/get_started/jfxpub-get_started.htm)
- Documentazione: <http://docs.oracle.com/javafx/2/>
- API: <https://openjfx.io/javadoc/13/>

# JavaFX – struttura delle applicazioni

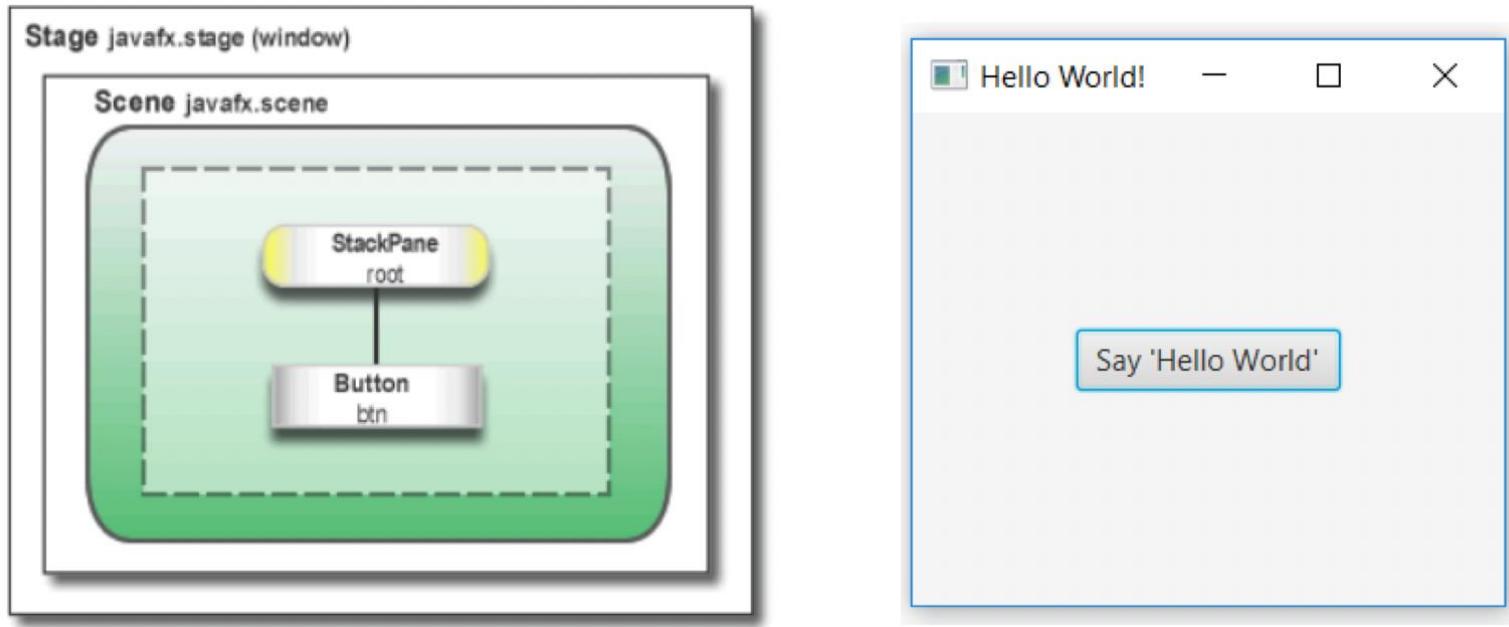


- Ogni applicazione JavaFX deve estendere **javafx.application.Application**
- **L'entrypoint di un'applicazione JavaFX** (che viene eseguito da JVM quando la si lancia) è il metodo **start()**. Questo metodo prende uno **Stage (finestra dell'applicazione)** come parametro e viene invocato dal launcher dell'applicazione

```
public class JavaFXApplication1 extends Application {
 @Override
 public void start(Stage primaryStage) {
 ...
 }
 public static void main(String[] args) {
 launch(args); //esegue il launcher dell'applicazione
 }
}
```



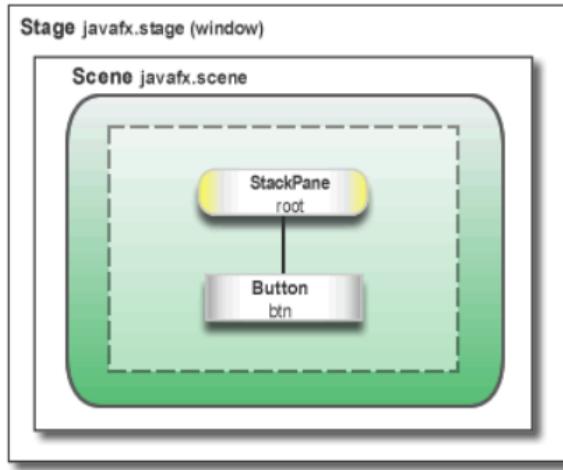
# JavaFX – Componenti principali - I



## Componenti principali di JavaFX:

- Lo **Stage** rappresenta la finestra (simile a JFrame di SWING)
- La **Scene** è il contenitore principale (e unico) da aggiungere a una finestra. Una Scene va associata a un pannello per definire il layout (per es. **StackPane**, posiziona i componenti "figli" inseriti al suo interno in una pila, uno sopra l'altro)

# JavaFX – Componenti principali - II



- **La struttura della GUI è gerarchica:** nel **pannello** della Scene noi possiamo inserire i componenti figli. I componenti figli possono a loro volta avere componenti figli.  
Non si possono avere 2 Scene in un solo Stage: bisogna dare alla Scene un pannello con il layout desiderato, es., StackPane, o GridPane, e aggiungere al pannello i sotto-componenti grafici come suoi figli.



## Attenzione: usare JavaFXML

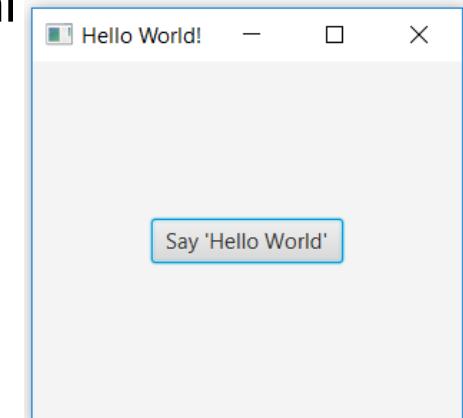
- Nel seguito vedremo una bozza di applicazione JavaFX per capire quali tipi di oggetti formano le GUI.
- Questo NON sarà il modo di sviluppare le applicazioni. A tale scopo useremo JavaFXML, che permette di progettare la GUI graficamente.



# JavaFX – Componenti principali - III

Estratto di codice per inserire in uno Stage una Scene (con pannello StackPane) e un bottone:

```
public void start(Stage primaryStage) {
 Button btn = new Button(); // crea il bottone
 btn.setText("Say 'Hello World'");
 StackPane root = new StackPane(); // componente ROOT
 root.getChildren().add(btn); //aggiunge btn a pannello
 Scene scene = new Scene(root, 300, 250); //dimensioni
 primaryStage.setTitle("Hello World!");
 primaryStage.setScene(scene);
 primaryStage.show(); // visualizza la finestra
}
```



# JavaFX – Event Handlers



Alle componenti grafiche possono essere associati i Listener, per reagire alle azioni dell'utente. Es. per i bottoni:

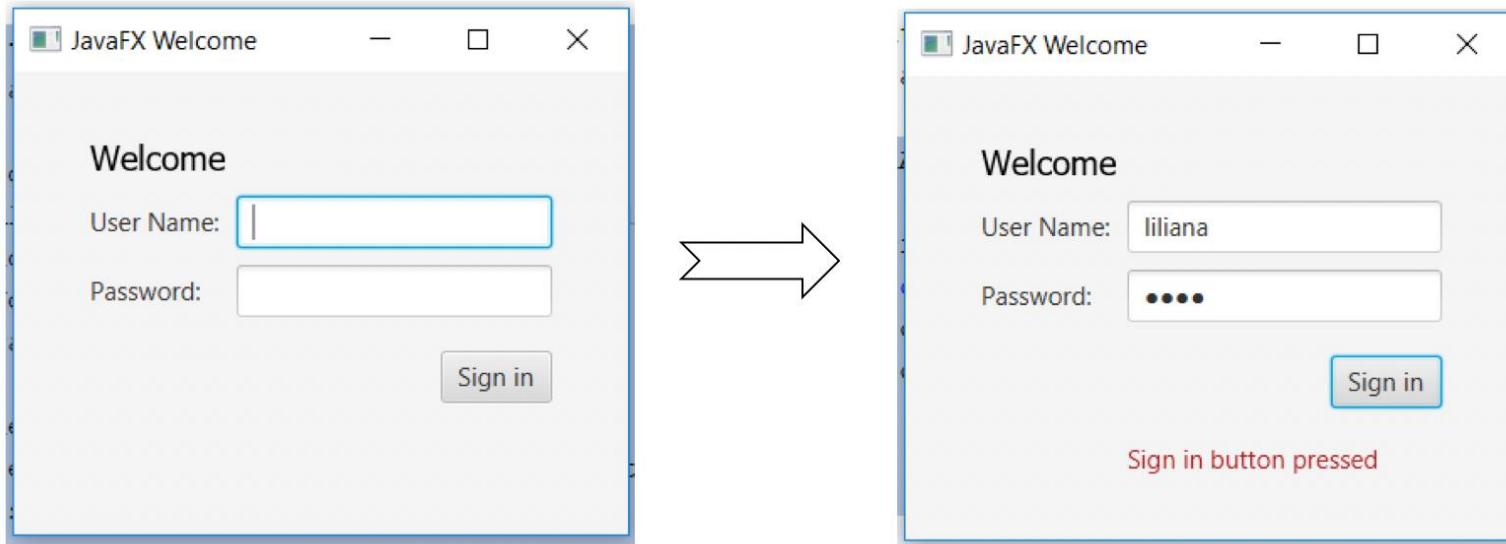
```
Button btn = new Button(); // creo un bottone
btn.setText("Say 'Hello World'");
btn.setOnAction(new EventHandler<ActionEvent>() {
 public void handle(ActionEvent event) {
 System.out.println("Hello World!");
 }
});
// il listener è stato definito come classe
// anonima. Il metodo handle() viene ridefinito
// per scrivere su output standard un
// messaggio ad ogni click sul bottone
```

# JavaFX – Codice completo dell'applicazione



```
public class JavaFXApplication1 extends Application {
 @Override
 public void start(Stage primaryStage) {
 Button btn = new Button();
 btn.setText("Say 'Hello World'");
 btn.setOnAction(new EventHandler<ActionEvent>() {
 @Override
 public void handle(ActionEvent event) {
 System.out.println("Hello World!");}});
 StackPane root = new StackPane();
 root.getChildren().add(btn);
 Scene scene = new Scene(root, 300, 250);
 primaryStage.setTitle("Hello World!");
 primaryStage.setScene(scene);
 primaryStage.show(); }
 public static void main(String[] args) {
 launch(args);
 }
}
```

# JavaFX – Forms (moduli) - I



Si basano su:

- Pannello **GridPane** per inserire i componenti grafici in una griglia
- **Label** per scrivere i titoli dei campi delle form
- **TextField** per definire i campi di input delle form
- **Bottoni** con listener per sottomettere le form
- **Text** per scrivere messaggi di output sulla finestra



# JavaFX – Forms (moduli) - II

```
public void start(Stage primaryStage) {
 primaryStage.setTitle("JavaFX Welcome");
 GridPane grid = new GridPane(); // pannello a griglia

 // ... omissis ...
```

```
Text scenetitle = new Text("Welcome");
scenetitle.setFont(Font.font("Tahoma", FontWeight.NORMAL, 20));
 // columnindex, rowindex, column span, row span
grid.add(scenetitle, 0, 0, 2, 1);
```

```
Label userName = new Label("User Name:");
grid.add(userName, 0, 1);
```

```
TextField userTextField = new TextField();
grid.add(userTextField, 1, 1);
```

```
// continua
```

| sceneTitle | sceneTitle    |
|------------|---------------|
|            |               |
| userName   | userTextField |
|            |               |

# JavaFX – Forms (moduli) - III



// continua ....

```
Label pw = new Label("Password:");
grid.add(pw, 0, 2);
```

```
PasswordField pwBox = new PasswordField();
grid.add(pwBox, 1, 2);
```

```
// creo il bottone di sottomissione dei dati
Button btn = new Button("Sign in");
// aggiungo il bottone alla griglia... omesso
```

// continua ....

| sceneTitle | sceneTitle    |
|------------|---------------|
| userName   | userTextField |
| pw         | pwBox         |
|            |               |
|            | hbBtn         |



# JavaFX – Forms (moduli) - IV

// continua ....

```
final Text actiontarget = new Text();
grid.add(actiontarget, 1, 6);
```

```
btn.setOnAction(new EventHandler<ActionEvent>() {
 @Override
 public void handle(ActionEvent e) {
 actiontarget.setFill(Color.FIREBRICK);
 actiontarget.setText("Sign in button pressed");
 }
});
```

// qui creo la Scene e la inserisco nella finestra

```
Scene scene = new Scene(grid, 300, 275);
primaryStage.setScene(scene);
primaryStage.show();
```

```
}
```

| sceneTitle | sceneTitle    |
|------------|---------------|
| userName   | userTextField |
| pw         | pwBox         |
|            |               |
|            | hbBtn         |
|            |               |
|            | actionTarget  |



# JavaFX – uso di CSS - I

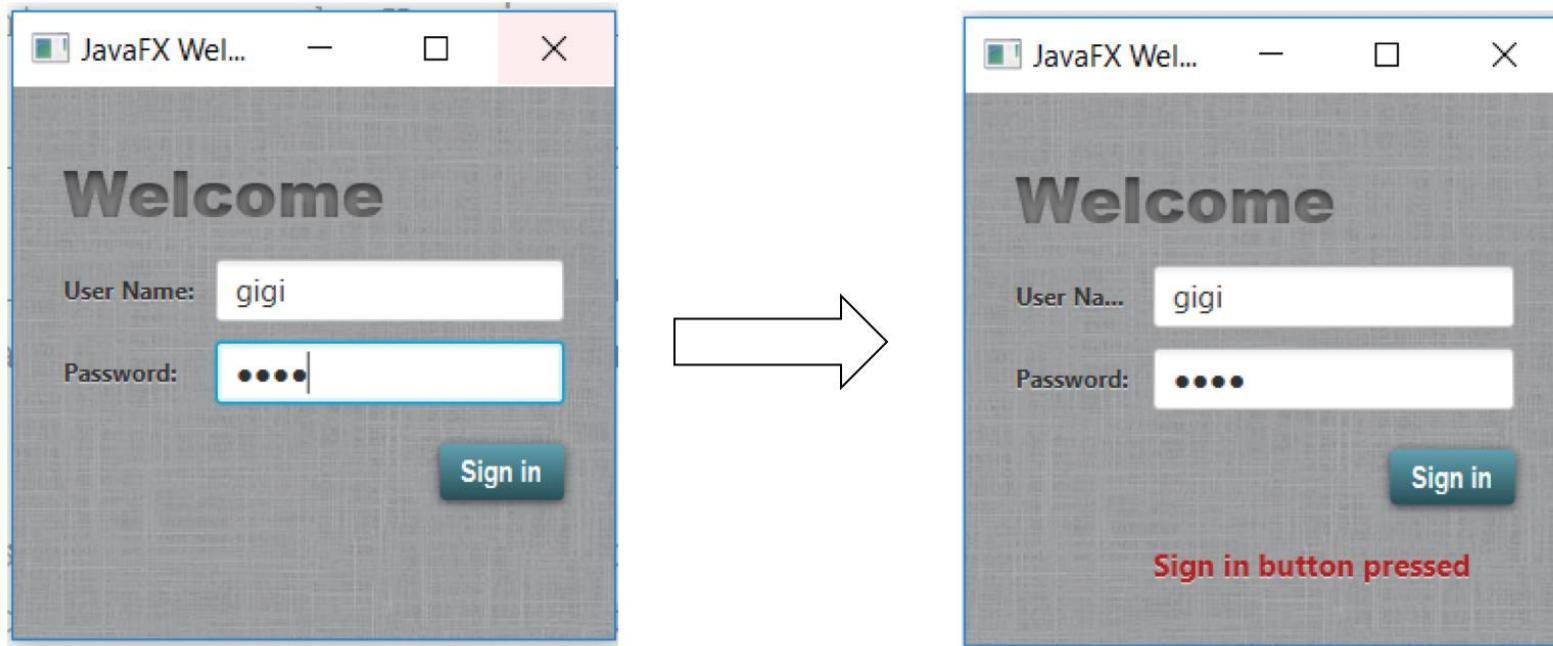
JavaFX permette di separare il contenuto di una GUI dal suo layout. A tale scopo, JavaFX offre i seguenti elementi:

- **Un foglio stile CSS** (es. **stile.css**), che posizioniamo nella cartella del codice dell'applicazione.
- **Un metodo di Scene per caricare il foglio stile:**  
**scene.getStylesheets().add(**  
    **MyApp.class.getResource("stile.css").toExternalForm());**
- **Un modo di dare gli ID ai componenti grafici**, ove si voglia applicare una regola CSS ad uno specifico elemento.  
Es: **scenetitle.setId("welcome-text");**



# Esempio: layout desiderato

Data l'applicazione che genera la form precedente, aggiungiamo il layout come foglio stile





# Foglio stile CSS – stile.css - I

```
.root { /* selettore di classe → per componenti tipati */
 -fx-background-image: url("background.jpg"); }

.label {
 -fx-font-size: 12px;
 -fx-font-weight: bold;
 -fx-text-fill: #333333; /* grigio */
 -fx-effect: dropshadow(gaussian , rgba(255,255,255,0.5) , 0,0,0,1);
/* testo shaded */
}

#welcome-text { /* selettore per ID */
 -fx-font-size: 32px;
 -fx-font-family: "Arial Black";
 -fx-fill: #818181;
 -fx-effect: innershadow(three-pass-box , rgba(0,0,0,0.7) , 6, 0.0 , 0 , 2);
}
```



# Foglio stile CSS – stile.css - I

```
#actiontarget {
 -fx-fill: FIREBRICK;
 -fx-font-weight: bold;
 -fx-effect: dropshadow(gaussian , rgba(255,255,255,0.5) , 0,0,0,1);}

.button {
 -fx-text-fill: white;
 -fx-font-family: "Arial Narrow";
 -fx-font-weight: bold;
 -fx-background-color: linear-gradient(#61a2b1,#2A5058);
 -fx-effect: dropshadow(three-pass-box , rgba(0,0,0,0.6) , 5, 0.0 , 0 , 1); }

.button:hover {
 -fx-background-color: linear-gradient(#2A5058, #61a2b1);
}
```



# Pannello con ID etc.

```
public MyPanel() {
 super(); // pannello a griglia
 this.setAlignment(Pos.CENTER);
 ...
 Text scenetitle = new Text("Welcome");
//scenetitle.setFont(Font.font("Tahoma", FontWeight.NORMAL, 20));
 // NON SERVE PIÙ
 this.add(scenetitle, 0, 0, 2, 1);
scenetitle.setId("welcome-text"); // per applicare CSS
 Label userName = new Label("User Name:");
 this.add(userName, 0, 1);
 ...
 final Text actiontarget = new Text();
actiontarget.setId("actiontarget"); // per applicare CSS
 this.add(actiontarget, 1, 6);
 ...
}
```



# JavaFX – tipi di pannelli (Pane)

JavaFX offre i seguenti tipi di pannello per organizzare il layout grafico dell'interfaccia utente:

- **GridPane** fa inserire i componenti figli in una griglia
- **StackPane** fa sovrapporre i figli in uno stack
- **BorderPane** fa inserire i figli a nord, est, ovest, sud, centro come per il BorderLayout di SWING.

*Ci interessa sapere che questi tipi di pannello esistono, anche se non li usiamo direttamente qui, perché ci serviranno per la grafica di JavaFXML. Svilupperemo tale grafica in combinazione con un tool grafico (SceneBuilder) per il design dell'interfaccia utente che genera il codice XML della GUI.*



# JavaFX e MVC

- Le classi di JavaFX possono essere integrate con Observer Observable per sviluppare applicazioni MVC
- Es: ristrutturiamo la precedente applicazione in modo che salvi in un data model i dati inseriti dall'utente e utilizzi il data model per visualizzarli sulla GUI:





# JavaFX e MVC – versione base (usa le classi/interface Observer e Observable - deprecate)

Per iniziare, sviluppiamo il data model etc. utilizzando le librerie di base del pattern Observer Observable in java

NB: In JavaFX esistono librerie che offrono le funzionalità degli Observer e degli Observable, le useremo:

- **ObservableList** implementa una lista di oggetti osservabili. ObservableList invia in automatico le notifiche agli osservatori quando cambia lo stato degli oggetti.
- **ListView** è un visualizzatore di liste. Una ListView può essere agganciata come osservatore ad una ObservableList per far sì che si aggiorni la visualizzazione della lista ad ogni cambiamento dell'osservato.
- ...
- **<https://docs.oracle.com/javafx/2/collections/jfxpub-collections.htm>**



# Data Model

```
public class Utente extends Observable { // DEPRECATO!!!!
 private String userName;
 private String password;
 public Utente() { userName = ""; password = ""; }
 public String getUserName() { return userName; }
 public String getPassword() { return password; }
 public void setData(String u, String p) {
 userName = u; password = p;
 setChanged(); notifyObservers();
 }
 @Override
 public String toString() {
 return "userName=" + userName + ", password=" + password;
 }
}
```



# Controller

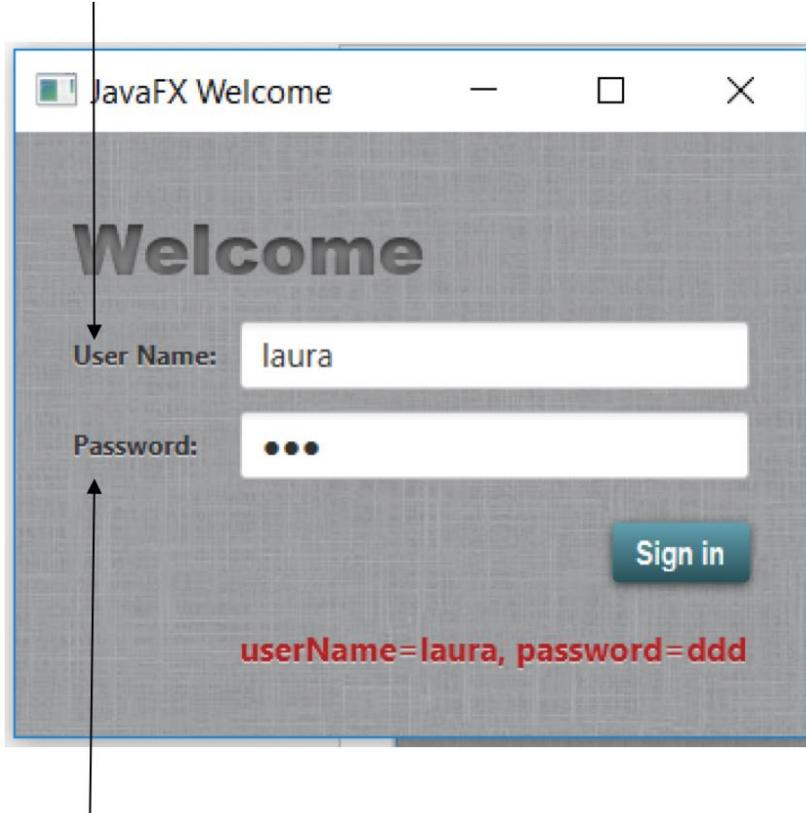
```
public class Controller {

 public Controller(MyPanel grid, Utente person) {
 Button btn = grid.getButton(); // prendo il
 // riferimento al bottone per agganciare il listener (EventHandler)
 btn.setOnAction(new EventHandler<ActionEvent>() {
 @Override
 public void handle(ActionEvent e) {
 person.setData(grid.getUserName(),
 grid.getPassword());
 }
 });
 }
}
```



# View – componenti

userName



pw

scenetitle

userTextField

pwBox (ignoriamo)

btn

actiontarget



# View- MyPanel – I

```
public class MyPanel extends GridPane
 implements Observer { //DEPRECATO
 private final Text actiontarget;
 private final Button btn;
 private final TextField userTextField;
 ...
}

public MyPanel() {
 super();
 ...
 Text scenetitle = new Text("Welcome");
 this.add(scenetitle, 0, 0, 2, 1);
 scenetitle.setId("welcome-text");
 // continua...
```



## View – MyPanel - II

```
Label userName = new Label("User Name:");
// 0,1: posizionamento della label in griglia
this.add(userName, 0, 1);
userTextField = new TextField();
this.add(userTextField, 1, 1);

//... field text per la password – ignoriamo

// continua ...
```



# View – MyPanel - III

```
btn = new Button("Sign in");
...
hbBtn.getChildren().add(btn);
this.add(hbBtn, 1, 4);

actiontarget = new Text();
actiontarget.setId("actiontarget");
this.add(actiontarget, 1, 6);

}

Button getButton() {
 return btn;
}
```



## View – MyPanel - IV

```
String getUserName() {
 return userTextField.getText();
}
}
```

```
String getPassword() {
 return pwBox.getText();
}
}
```

```
public void update(Observable obs, Object extra_arg) {
 System.out.println("Refresh GUI");
 actiontarget.setFill(Color.FIREBRICK);
 if (obs instanceof Utente) {
 actiontarget.setText(((Utente) obs).toString()); }
 }
} // end MyPanel
```



# Applicazione principale - I

```
public class JavaFXApplication5MVCsimple extends Application {
 @Override
 public void start(Stage primaryStage) {
 primaryStage.setTitle("JavaFX Welcome");
 Utente person = new Utente(); // creo il model
 MyPanel grid = new MyPanel(); // creo la view
 // aggancio la view come observer del model
 person.addObserver(grid);
 Controller control = new Controller(grid, person);
 Scene scene = new Scene(grid, 300, 275);
 primaryStage.setScene(scene);
 }
}
```



# Applicazione principale – versione base - II

```
 scene.getStylesheets().add(// foglio stile
 JavaFXApplication5MVCsimple.class.getResource("stile.cs
s").toExternalForm());
 primaryStage.show();
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
 launch(args);
}
}
```



# PASSO 2 – properties

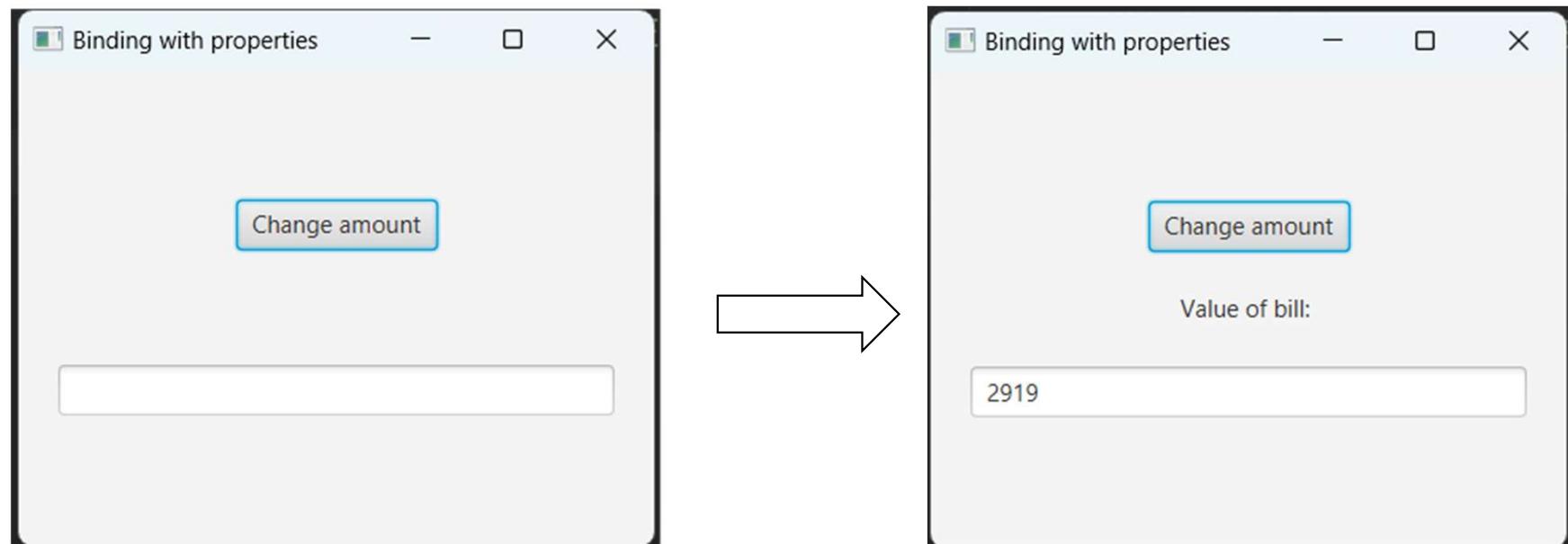
**Uso di java beans e properties per:**

- Rappresentare i dati del model in modo standard, come java beans.
- Legare direttamente i dati del model alla view in modo che la view conosca i valori dei dati e li visualizzi in modo responsive.
- Associare listeners ai dati (properties dei java beans) per effettuare operazioni quando cambiano valore.

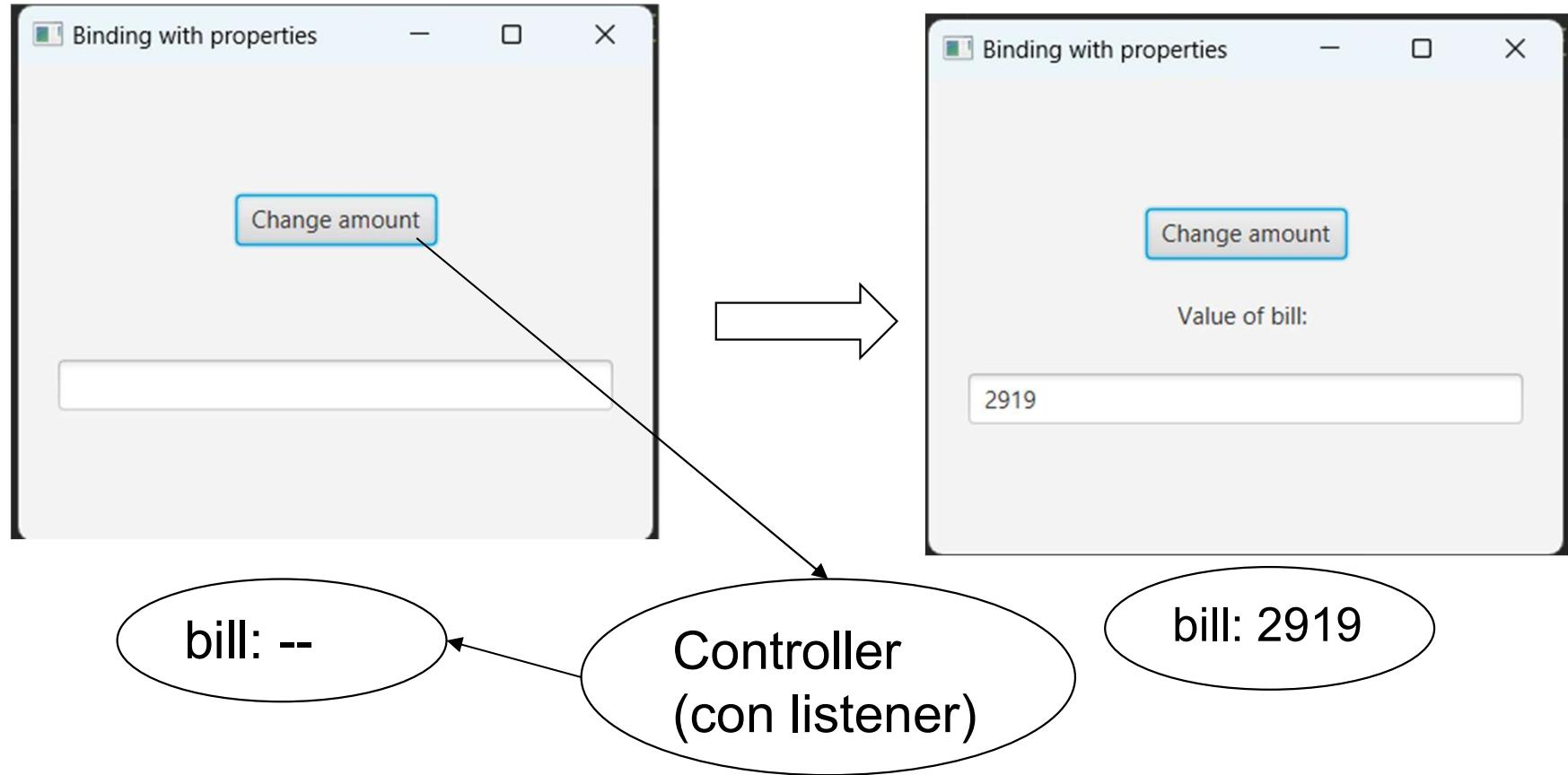
## Esempio – properties - I

In un'app MVC, noi vogliamo che la view sia responsive rispetto ai cambiamenti di stato del model.

Es: cliccando Change amount, noi generiamo delle fatture (bill) e visualizziamo il nuovo valore della spesa.



# Esempio – properties - II

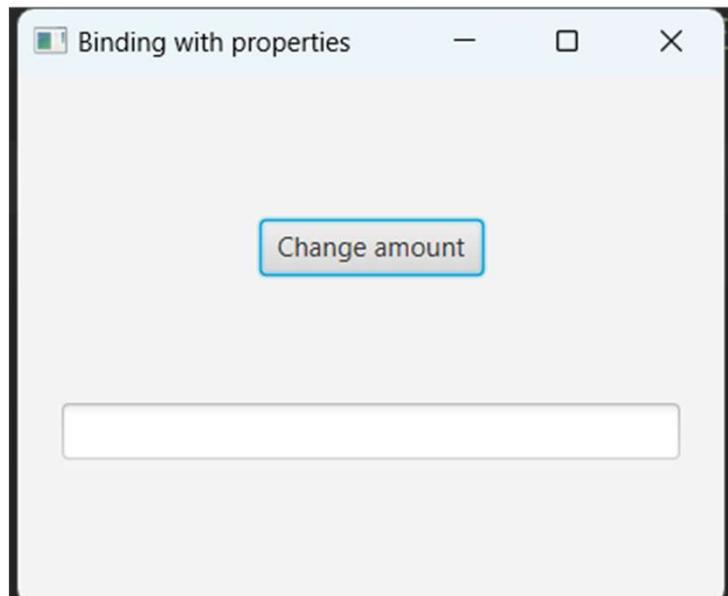


**Il bill è il model dell'applicazione.**

Quando il valore di bill cambia, noi vogliamo che la GUI mostri subito il nuovo valore → **facciamo in modo che l'area di testo della GUI osservi bill!**

## Esempio – properties - III

Obiettivo: evitare di scrivere codice dettagliato per visualizzare nella GUI i valori del model mentre cambiano, e/o per inizializzare i valori del model a partire dai dati acquisiti in input (form) nella GUI.



Le properties dei java beans permettono di fare questo.

# JavaBeans - I



I Java Beans sono classi java che rispettano uno standard di definizione dei metodi e delle loro variabili di istanza (o di stato)

- Per ogni variabile xxx di istanza che vogliamo esporre come **property** del javaBean, noi dobbiamo definire il metodo **public Type getXxx()**, per leggere valore della property.
- Se vogliamo permettere di modificare il valore della property, dobbiamo definire anche il metodo **public void setXxx()**, per assegnare un valore alla property.



## JavaBeans - II

**Il concetto di property è slegato dall'esistenza di variabili di istanza nel java bean:**

- Il java bean può avere **variabili di istanza private che non sono property** (in quanto prive di metodo `getXxx()`) e servono internamente per le computazioni.
- Il **nome di una property è determinato dai metodi `getXxx()` e `setXxx()`, NON dal nome della variabile di istanza ad essa associata.**



## JavaBeans - III

- Una **property** potrebbe derivare dall'esecuzione di un metodo `getXxx()` del java bean
  - Noi possiamo definire metodi `getYyy()` (e quindi properties) utili per eseguire codice applicativo che utilizza lo stato del bean e fa riferimento a più variabili del bean per restituire risultati.
  - Es: se un java bean memorizza due valori numerici in variabili di istanza private, noi possiamo definire una property "prodotto", basata sulla definizione del metodo `getProdotto()`, che restituisce il prodotto dei due valori.



# JavaBeans – esempio

NB: prop1, prop2 e prodotto sono properties; var1 e var2 **NON** sono property

```
public class ExampleBean {
 private int var1; private int var2;
 public ExampleBean() { }
 public void setProp1(int n) {
 this.var1 = n; }
 public int getProp1() {
 return var1; }
 public void setProp2(int n) {
 this.var2 = n; }
 public int getProp2() {
 return var2; }
 public int getProdotto() {
 return var1*var2;}
}
```

# Java Beans e Properties



Le properties dei java beans servono per facilitare la definizione di **bindings** tra variabili. I bindings permettono di **cambiare il valore di variabili in dipendenza da altre**: quando si definisce un binding (dipendenza) tra due variabili, se una cambia valore, l'altra viene modificata di conseguenza. I bindings sono utili in varie situazioni. Es:

- Definire una variabile che ha come valore la somma dei valori di altre n variabili, con la somma che si aggiorna automaticamente quando una di queste cambia valore
- Nelle interfacce grafiche, legare la componente grafica a un modello, in modo che la componente grafica si aggiorni automaticamente ogni volta che il modello cambia valore

**Strettamente legato a Observer Observable e a MVC**



# public interface ObservableValue<T>

**public interface ObservableValue<T> extends**

**Observable** definisce entità che racchiudono un valore e permettono di osservare i cambiamenti di tale valore. Metodi:

- **T getValue():** restituisce il valore dell'oggetto ObservableValue.
- **void addListener(ChangeListener<? super T> listener):** aggiunge all'ObservableValue un listener di cambiamento.
- **void removeListener(ChangeListener<? super T> listener):** toglie il listener dall'ObservableValue.

Dove **<? super T>** indica che il tipo del ChangeListener deve essere una **superclasse** di T (il contrario di **<? extends T>**)

(qui Observable è javafx.beans.Observable, non l'Observable deprecata)

# **public interface ChangeListener<T>**



**public interface ChangeListener<T> definisce i listener di cambiamento di valore delle properties di tipo T**

Un ChangeListener viene notificato ogni volta che l'oggetto ObservableValue a cui è associato il listener cambia valore.

Le classi che implementano ChangeListener devono implementare il metodo

**void changed(ObservableValue<? extends T> obs,**

**T oldValue, T newValue)**

per specificare cosa deve fare il listener quando rileva un evento di cambiamento dell'ObservableValue obs.

# Properties con JavaFX



JavaFX offre classi di libreria che rappresentano e gestiscono javabeans e properties. Es.: la **abstract class DoubleProperty** rappresenta **una property di tipo Double** di un bean. La property **implementa ObservableValue<Number>**.

javafx.beans.property

## Class DoubleProperty

java.lang.Object

    javafx.beans.binding.NumberExpressionBase

        javafx.beans.binding.DoubleExpression

            javafx.beans.property.ReadOnlyDoubleProperty

                javafx.beans.property.DoubleProperty

### All Implemented Interfaces:

NumberExpression, Observable, Property<Number>, ReadOnlyProperty<Number>,  
ObservableDoubleValue, ObservableNumberValue, ObservableValue<Number>,  
WritableDoubleValue, WritableNumberValue, WritableValue<Number>

# SimpleDoubleProperty



Una implementazione di DoubleProperty è **SimpleDoubleProperty**, che racchiude un valore Double.

Consideriamo una applicazione JavaFX che gestisce un oggetto con tale property. Vediamo come fare a rilevare automaticamente il cambiamento di valore della property, e a gestire la property facendo delle operazioni.

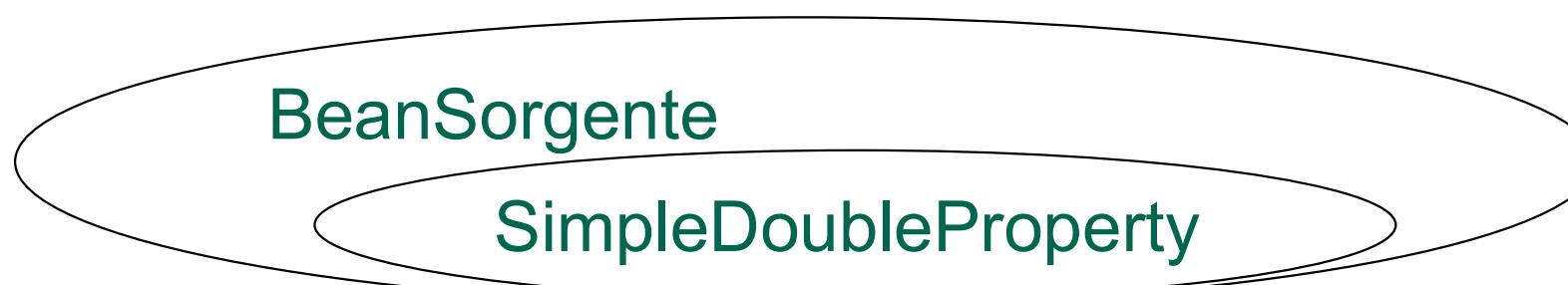
Per rilevare i cambiamenti di valore della property dobbiamo associare un **listener** (ChangeListener) alla property stessa.

**NB: per ora noi non usiamo JavaFXML e trascuriamo MVC.**

# Listeners di properties – bindingsNoGUI2 - I

```
public class BeanSorgente {
 // Define a variable to store the property
 private SimpleDoubleProperty amount = new SimpleDoubleProperty();

 public final double getAmount() { // getter for the property's value
 return amount.get();
 }
 public final void setAmount(double value) { // setter for property's value
 amount.set(value);
 }
 // getter for the property itself
 public SimpleDoubleProperty amountProperty() {
 return amount;
 }
}
```



## Listeners di properties – bindingsNoGUI2 - II

```
public class BeanDipendente { // Define a variable to store the property
 private SimpleDoubleProperty amount = new SimpleDoubleProperty();

 public final double getAmount() { // getter for the property's value
 return amount.get();
 }

 public final void setAmount(double value) { // setter for property's value
 amount.set(value);
 }

 public SimpleDoubleProperty amountProperty() { // getter for the property itself
 return amount;
 }
}
```



# Listeners di properties – bindingsNoGUI2 - III

```
public static void main(String[] args) {
 BeanSorgente b1 = new BeanSorgente();
 BeanDipendente b2 = new BeanDipendente();
 SimpleDoubleProperty property = b1.amountProperty();
 property.addListener(new ChangeListener<Number>() {
 @Override
 public void changed(ObservableValue<? extends Number> obs,
 Number oldValue, Number newVal) {
 //b2.setAmount((Double)newVal);
 System.out.println("b1: " + newVal);
 }
 });
 for (int i=0; i<5; i++) {
 b1.setAmount(i);
 System.out.println("Bean sorgente: " + b1.getAmount() +
 "; Bean dipendente: "+b2.getAmount());
 }
}
```

# Listeners di properties – bindingsNoGUI2 - IV

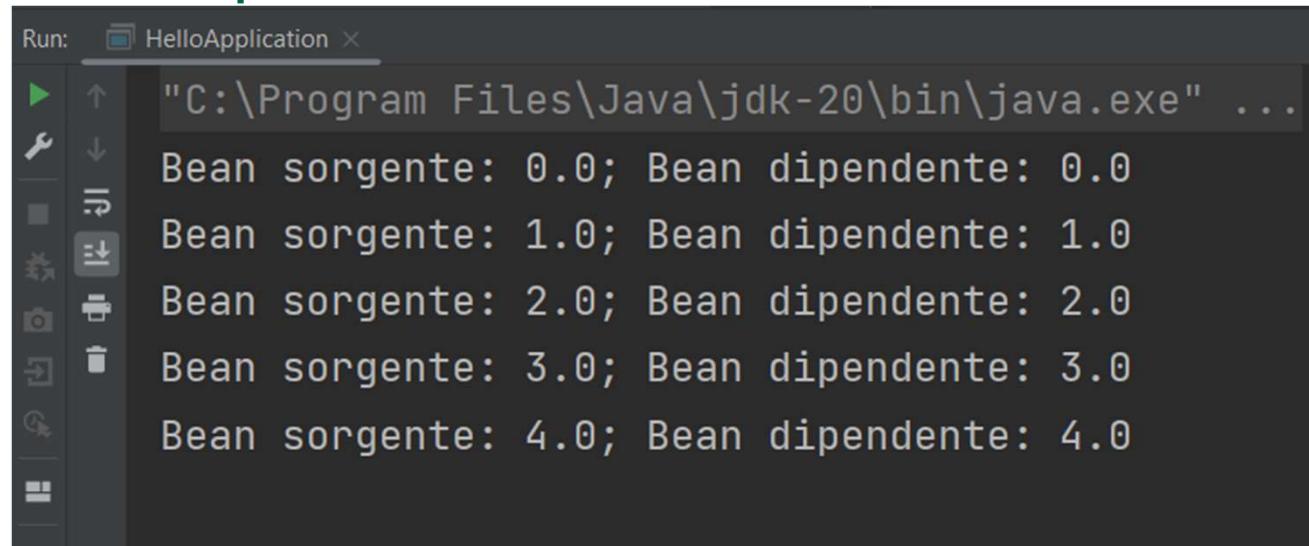
```
Bean sorgente: 0.0; Bean dipendente: 0.0
b1: 1.0
Bean sorgente: 1.0; Bean dipendente: 0.0
b1: 2.0
Bean sorgente: 2.0; Bean dipendente: 0.0
b1: 3.0
Bean sorgente: 3.0; Bean dipendente: 0.0
b1: 4.0
Bean sorgente: 4.0; Bean dipendente: 0.0
b1: 1.0
Bean sorgente: 1.0; Bean dipendente: 1.0
b1: 2.0
Bean sorgente: 2.0; Bean dipendente: 2.0
b1: 3.0
Bean sorgente: 3.0; Bean dipendente: 3.0
b1: 4.0
Bean sorgente: 4.0; Bean dipendente: 4.0
Process finished with exit code 0
```

# Esempio – binding di properties - I

Non sempre serve definire un listener perché non ci sono operazioni complesse da fare. Talvolta basta un **binding**!

Sperimentiamo un legame tra properties nell'applicazione **bindingsNoGUI**. Vogliamo che una property rifletta sempre il valore di un'altra property. Lo possiamo ottenere tramite un binding unidirezionale:

Bean borgente → Bean dipendente



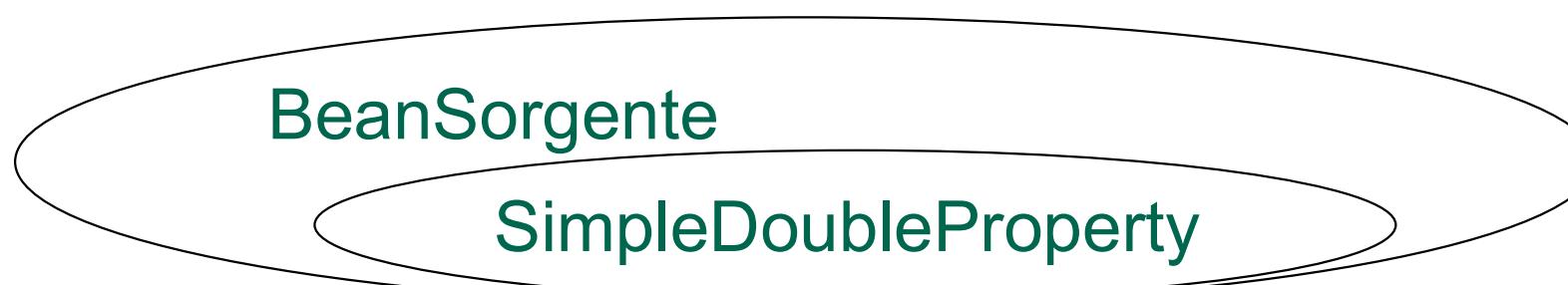
The screenshot shows a Java application window titled "HelloApplication". The run configuration dropdown is set to "C:\Program Files\Java\jdk-20\bin\java.exe" ... . The main pane displays the following text output:

```
Run: HelloApplication ×
" C:\Program Files\Java\jdk-20\bin\java.exe" ...
Bean sorgente: 0.0; Bean dipendente: 0.0
Bean sorgente: 1.0; Bean dipendente: 1.0
Bean sorgente: 2.0; Bean dipendente: 2.0
Bean sorgente: 3.0; Bean dipendente: 3.0
Bean sorgente: 4.0; Bean dipendente: 4.0
```

## Esempio – binding di properties - II

```
public class BeanSorgente {
 // Define a variable to store the property
 private SimpleDoubleProperty amount = new SimpleDoubleProperty();

 public final double getAmount() { // getter for the property's value
 return amount.get();
 }
 public final void setAmount(double value) { // setter for property's value
 amount.set(value);
 }
 // getter for the property itself
 public SimpleDoubleProperty amountProperty() {
 return amount;
 }
}
```



## Esempio – binding di properties - III

```
public class BeanDipendente { // Define a variable to store the property
 private SimpleDoubleProperty amount = new SimpleDoubleProperty();

 public final double getAmount() { // getter for the property's value
 return amount.get();
 }

 public final void setAmount(double value) { // setter for property's value
 amount.set(value);
 }

 public SimpleDoubleProperty amountProperty() { // getter for the property itself
 return amount;
 }
}
```

The diagram consists of two nested ovals. The inner oval is labeled "SimpleDoubleProperty". The outer oval is labeled "BeanDipendente". The "SimpleDoubleProperty" oval is positioned entirely within the "BeanDipendente" oval.

BeanDipendente

SimpleDoubleProperty

## Esempio – binding di properties - IV

```
public class HelloApplication {
 public static void main(String[] args) {
 BeanSorgente b1 = new BeanSorgente();
 BeanDipendente b2 = new BeanDipendente();
 bindProperties(b1, b2);
 for (int i=0; i<5; i++) {
 b1.setAmount(i);
 System.out.println("Bean sorgente: " + b1.getAmount() +
 "; Bean dipendente: "+b2.getAmount());
 }
 }

 public static void bindProperties(BeanSorgente b1, BeanDipendente b2){
 b2.amountProperty().bind(b1.amountProperty());
 }
}
```

In seguito al binding il valore di b2 dipende dal valore di b1.

# Binding di properties con GUI - I



## abstract class StringProperty

class SimpleStringProperty racchiude un valore String

Sviluppiamo una semplice applicazione JavaFXML in cui leghiamo due campi di testo della GUI in modo che il secondo prenda sempre il valore che viene inserito nel primo:



← inserimento  
← visualizza

# Come legare i due campi di testo della GUI - I



Sia il textfield «inserimento» che il textfield «visualizza» hanno una SimpleString property predefinita che mantiene il valore che viene visualizzato a video → basta legare le due properties in un metodo del controller della vista (per ora, il binding è attivato cliccando il bottone):

## Applicazione JavaFXML2

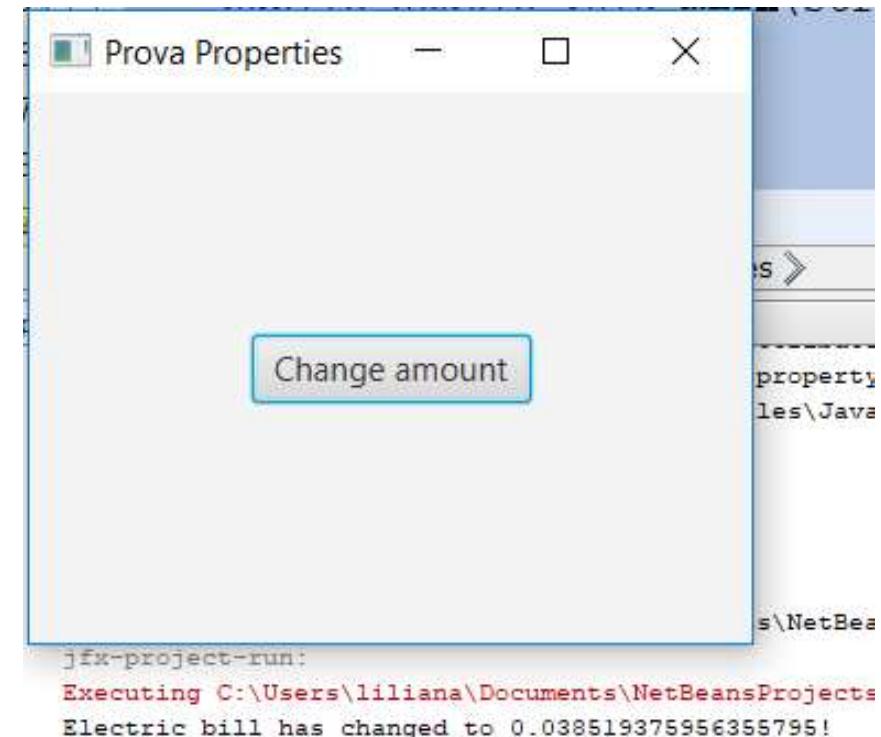
```
public class HelloController {
 @FXML
 private TextField inserimento;
 @FXML
 private TextField visualizza;
 @FXML
 private void onHelloButtonClick() { bindProperties(); }
 @Override
 public void bindProperties() {
 visualizza.textProperty().bind(inserimento.textProperty());
 }
}
```

# Properties dei Java Beans



Da: <https://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm>

Definiamo una applicazione JavaFX che crea un oggetto Bill e offre un pulsante per assegnare un quantitativo di soldi random alla property dell'oggetto.



Cliccando sul pulsante «Change amount» (btn nel codice) cambia il valore della property di Bill: vd. la stampa su output standard: «Electric bill has changed to ....»

# Properties dei Java Beans – Esempio



Da: <https://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm>

Definiamo un JavaBean Bill che rappresenta una fattura.

Bill ha la DoubleProperty amountDue che rappresenta il quantitativo di soldi da pagare:

```
class Bill {

 private DoubleProperty amountDue = new SimpleDoubleProperty();

 public final double getAmountDue(){return amountDue.get();}

 public final void setAmountDue(double value){amountDue.set(value);}
 // Getter della property come oggetto

 public DoubleProperty amountDueProperty() {return amountDue;}
}
```

# Properties – esempio – I



```
public class Main extends Application {
 @Override
 public void start(Stage primaryStage) {
 Bill electricBill = new Bill();
 final Random r = new Random();
 // assegno il ChangeListener alla property amountDue per rilevare i suoi cambiamenti
 electricBill.amountDueProperty().addListener(new ChangeListener() {
 @Override
 public void changed(ObservableValue o, Object oldV, Object newV) {
 double n = ((Double) newV).doubleValue();
 System.out.println("Electric bill has changed to " + n + "!");
 }
 });
 ... Continua ...
 }
}
```

# Properties – esempio – II



```
Button btn = new Button();
btn.setText("Change amount");
// definisco il bottone e l'Event Listener del bottone, per modificare il
// valore della property «amountDueProperty»
btn.setOnAction(new EventHandler<ActionEvent>() {
 @Override
 public void handle(ActionEvent event) {
 double d = r.nextDouble();
 electricBill.setAmountDue(d);
 }
});
// Continua ...
```

# Properties – esempio – III



```
StackPane root = new StackPane();
root.getChildren().add(btn);
Scene scene = new Scene(root, 300, 250);
primaryStage.setTitle("Prova Properties");
primaryStage.setScene(scene);
primaryStage.show();
}
/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
 launch(args);
}
}
```

# PASSO 3



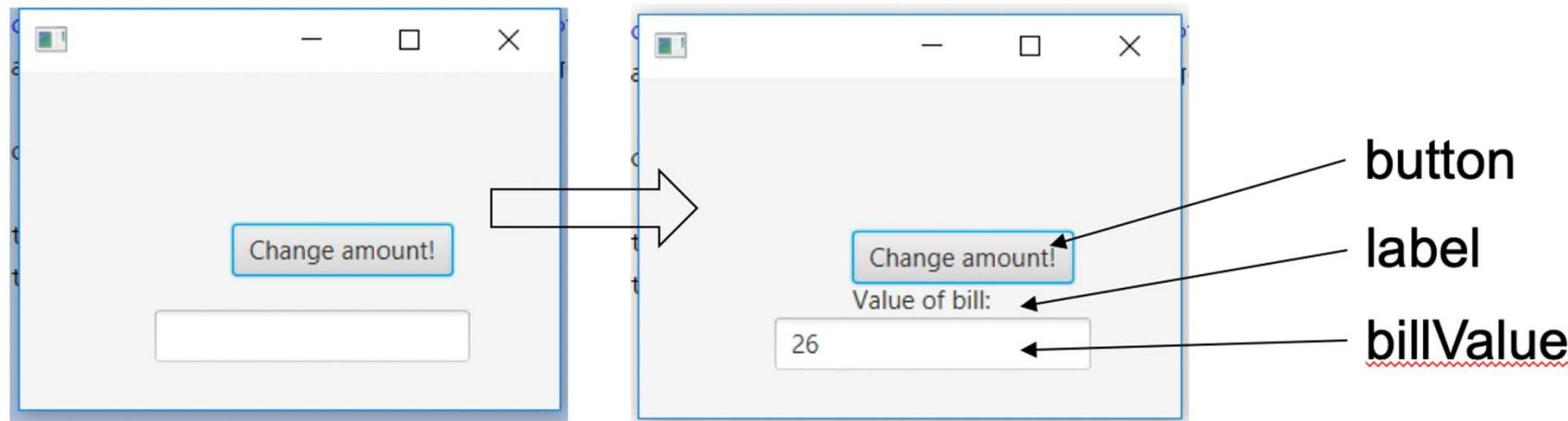
- **Uso di java beans e BINDING delle properties ai componenti della GUI per semplificare il codice della GUI – in combinazione con la specifica XML della GUI**

Mostriamo l'uso dei binding con l'applicazione JavaFXML4-properties

# JavaFXML4-properties

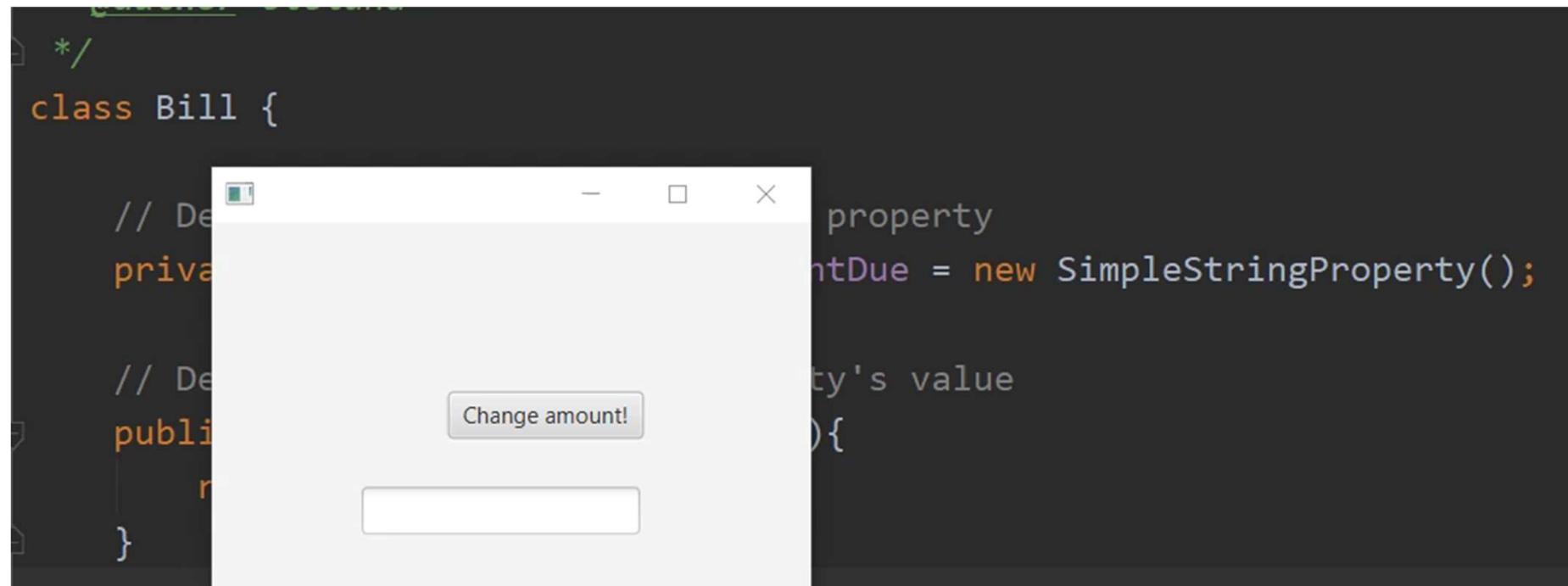


La seguente applicazione visualizza il valore del "bill" ad ogni suo cambiamento. Il bill cambia valore quando l'utente clicca sul bottone. Come prima, il click assegna al bill un numero intero casuale. Bill rappresenta una fattura.



NB: i TextField hanno una **StringProperty** che mantiene il loro valore → **billValue.textProperty() restituisce tale property.**

# JavaFXML4-properties - esecuzione



The image shows a Java code editor and a running JavaFX application window. The code editor displays a class named Bill with a private property amountDue and a public setter method setAmountDue. A tooltip 'Change amount!' is visible over the setter method. The running application window shows a simple UI with a text input field and a button labeled 'Change amount!'. The code is as follows:

```
/*
 *
 */
class Bill {

 // Define a property
 private SimpleStringProperty amountDue = new SimpleStringProperty();

 // Define a setter for the property's value
 public final void setAmountDue(String value){}

 // Define a setter for the property's value
 public final void setAmountDue(String value){
}
```

# JavaFXML4-properties – Model



```
class Bill {
 // Definiamo la property «amountDue» come StringProperty affinché sia
 // compatibile con la text property del campo della GUI
 private SimpleStringProperty amountDue = new SimpleStringProperty();

 // Define a getter for the property's value
 public final String getAmountDue(){return amountDue.get();}

 // Define a setter for the property's value
 public final void setAmountDue(String value){amountDue.set(value);}

 // Define a getter for the property itself
 public SimpleStringProperty amountDueProperty() {return amountDue;}
}
```

# JavaFXML4-properties – View (hello-view.fxml)

```
<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.*?>...<?import javafx.scene.layout.*?>

<\VBox
 <Button text="Change amount" onAction="#onHelloButtonClick"/>
 <Label fx:id="ris" layoutX="126" layoutY="120"
 minHeight="16" minWidth="69" />
 <TextField fx:id="billValue" layoutX="80.0" layoutY="141.0" />
</VBox>
```

# JavaFXML4-properties – Controller – I



```
public class Controller implements Initializable {
 private Bill electricBill = new Bill(); //variabile del controller
 private final Random r = new Random(); //costante del controller
 @FXML
 private Label ris;
 @FXML
 private TextField billValue;
 @FXML
 protected void onHelloButtonClick() {
 ris.setText("Value of bill:"); // brutto...
 int i = r.nextInt(10000);
 electricBill.setAmountDue(new StringBuilder().append(i).toString());
 }
 // continua...
```

# JavaFXML4-properties – Controller - II



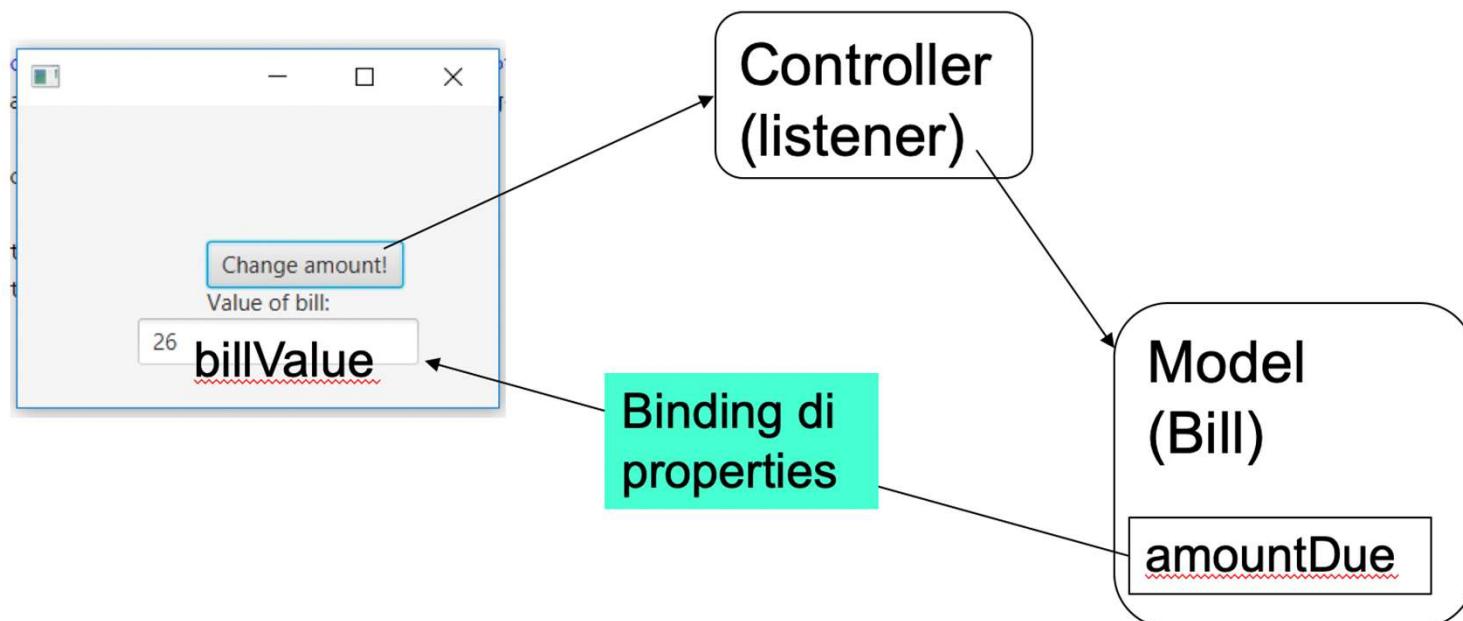
```
public void bindProperties() {
 // Questa istruzione associa la SimpleStringProperty
 // del textField billValue (billValue.textProperty())
 // alla SimpleStringProperty amountDueProperty di electricBill
 // → ogni volta che amountDueProperty viene modificata,
 // si aggiorna la visualizzazione di billValue nella GUI
billValue.textProperty().bind(electricBill.amountDueProperty());
 // continua ...
 // il binding tra queste due property evita di definire un listener di
 // cambiamento esplicito (ChangeListener) → semplifica il codice del controller
```

# JavaFXML- binding



`billValue.textProperty().bind(electricBill.amountDueProperty());`

lega la textProperty della GUI alla SimpleStringProperty del model



# JavaFXML-properties – Controller - III

/\* continua:



se vogliamo visualizzare il valore della property su standard output dobbiamo definire anche il listener di cambiamento della property (vedi sotto). Se non ci serve questa visualizzazione, il codice sotto può essere rimosso. In ogni caso, non bisogna assegnare esplicitamente il nuovo valore a billValue perché l'assegnamento viene fatto in automatico grazie al binding delle due property.

\*/

```
electricBill.amountDueProperty().addListener(new
ChangeListener<String>(){

 public void changed(ObservableValue<? extends String> o,
 String oldVal, String newVal) {

 System.out.println("Electric bill has changed to " + newVal + "!");
 //billValue.setText((new Double(newVal)).toString()); // no – binding!
 }
});
})
```

# JavaFXML-properties – Main



```
public class Main extends Application {
 @Override
 public void start(Stage stage) throws IOException {
 FXMLLoader fxmlLoader = new
 FXMLLoader(HelloApplication.class.getResource("hello-view.fxml"));
 Scene scene = new Scene(fxmlLoader.load(), 320, 240);
 stage.setTitle("Binding with properties");
 stage.setScene(scene); // invoco il metodo bindProperties nel main
 HelloController contr = fxmlLoader.getController();
 contr.bindProperties();
 stage.show();
 }
 public static void main(String[] args) {
 launch(args);
 }}
```

# JavaFX – note tecniche



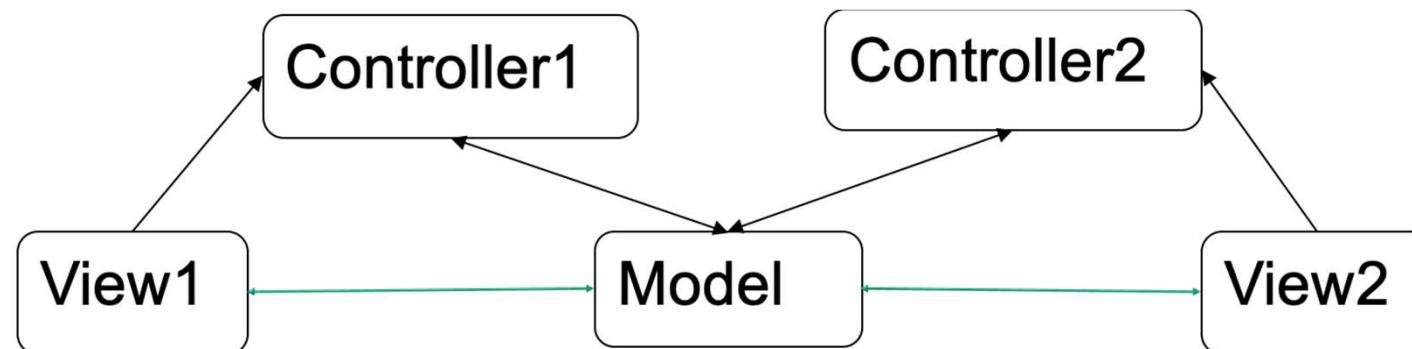
- Quando dovete risolvere i nomi di classi che non appartengono al core Java (es. GridPane, Button, etc.), risolveteli sempre con la versione del pacchetto javafx. Non scegliete le altre versioni delle classi, se presenti, altrimenti non funzioneranno le applicazioni JavaFX.
- Ricordate che se l'applicazione utilizza file (es., immagini, o fogli stile) e a runtime si blocca per via di una NullPointerException, il tutto potrebbe essere dovuto alla mancanza dei file nella cartella del codice eseguibile (sotto build/classes). Controllate, e se necessario caricate i file manualmente (può accadere, anche se raramente).

# JavaFX – applicazioni con view e controller multipli



Se un'applicazione JavaFXML ha viste multiple necessita di più di un controller (uno per ogni view). In tal caso normalmente i controller devono condividere il model dell'applicazione per sincronizzarsi attraverso il model. Quindi, voi dovrete passare il riferimento delle variabili a tutti i controller. Per fare questo, nel metodo start(Stage) dell'applicazione JavaFXML eseguite le seguenti operazioni:

- Create il Model  $m$  (e inizializzatelo).
- Create i controller con FXMLLoader e prendete i loro riferimenti.
- ...



 Pausa 00:00:00 Selezione area Audio Registrare puntatore

File

Jacob Smith  
Isabella Johnson  
Ethan Williams  
Emma Jones  
Michael Brown

First Name:

Last Name:

Email:

# JavaFXML-TUTORIAL-MVC – I



```
public class Main extends Application { // codice non aggiornato a ultima versione di JavaFXML
 @Override
 public void start(Stage primaryStage) throws Exception {
 ...
 DataModel model = new DataModel(); // creo e inizializzo il model
 // poi creo i controller e prendo i riferimenti a ciascuno di essi
 FXMLLoader listLoader = new FXMLLoader(getClass().getResource("list.fxml"));
 root.setCenter(listLoader.load());
 ListController listController = listLoader.getController();
 FXMLLoader editorLoader = new FXMLLoader(getClass().getResource("editor.fxml"));
 root.setRight(editorLoader.load());
 EditorController editorController = editorLoader.getController();
 listController.initModel(model); editorController.initModel(model); // passo il model ai controller
 Scene scene = new Scene(root, 800, 600); primaryStage.setScene(scene);
 primaryStage.show();
 }
```

# JavaFXML-TUTORIAL-MVC – II



```
FXMLLoader listLoader =
 new FXMLLoader(getClass().getResource("list.fxml"));

root.setCenter(listLoader.load());

ListController listController = listLoader.getController();
```

Per ciascun controller, prendete il riferimento al controller  $c_i$ , interrogando il loader (metodo getController() del loader) e invocate  $c_j.initModel(m)$  per dargli il model. A quel punto tutti i controller hanno il riferimento allo stesso model.

# JavaFXML-TUTORIAL-MVC - III



```
public class ListController {

 @FXML
 private ListView<Person> listView;

 private DataModel model; // variabile globale del ListController, definita nella classe

 public void initModel(DataModel model) {
 if (this.model != null) {// ensure model is only set once:
 throw new IllegalStateException("Model can only be initialized once");
 }
 this.model = model ;
 ... continua il metodo di inizializzazione con le altre operazioni da fare
 } ...
```

# JavaFX – collezioni osservabili



Tutorial per usare **ObservableList** etc.:

<https://docs.oracle.com/javafx/2/collections/jfxpub-collections.htm>

Per esempio:

API delle ObservableList (gli osservabili):

<https://docs.oracle.com/javase/8/javafx/api/javafx/collections/ObservableList.html>

API delle ListView (gli osservatori):

<https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/ListView.html>



# Programmazione III

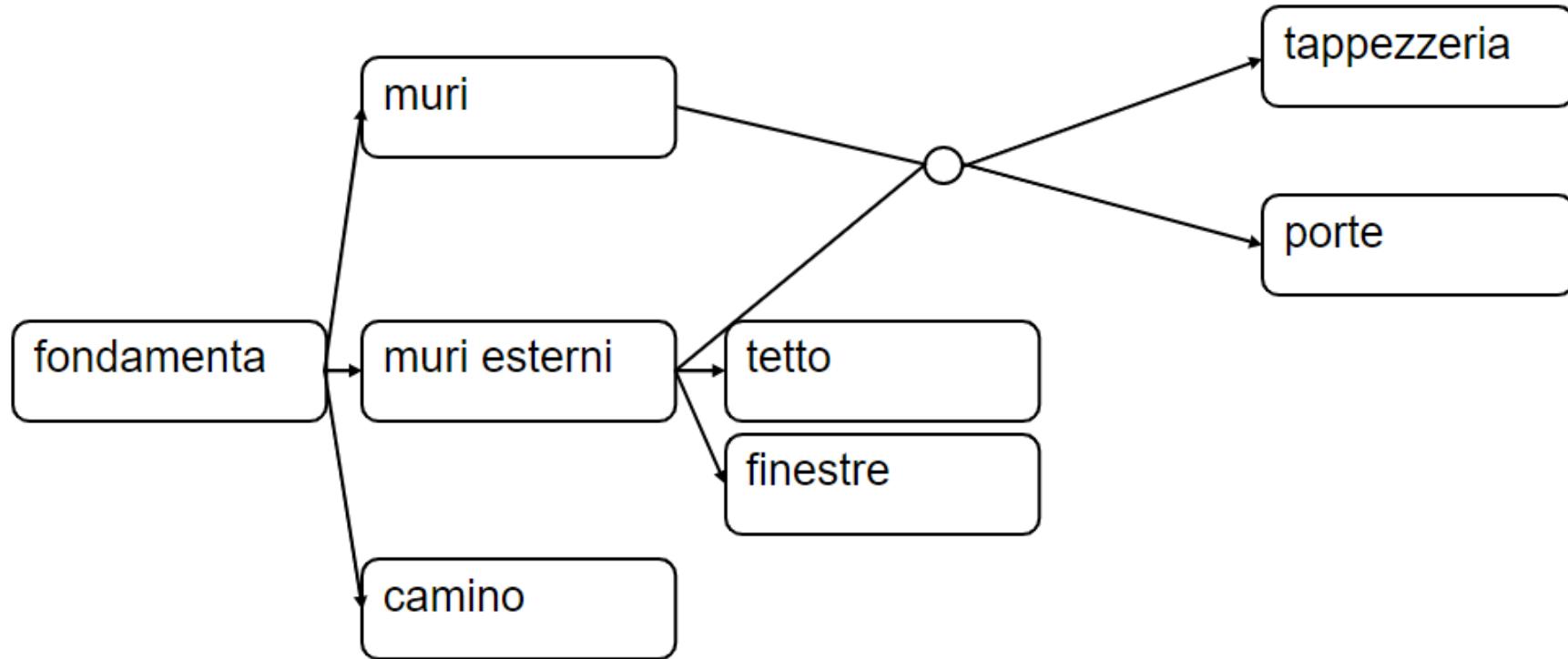
Prof.ssa Liliana Ardissono  
Dipartimento di Informatica  
Università di Torino

**Programmazione parallela con i Java Thread –  
parte 1 – esempio di uso di join(): scheduling di  
task**



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND

# Esempio di join: Scheduling di task per gestire un workflow - I



TextPad - C:\Users\liliana\Dropbox\ DIDATTICA\ PROGR III-2021\ LUCIDI\ 0000-ESEMPI\ 8-THREAD\ 09-scheduling\Scheduling.java

File Edit Search View Tools Macros Configure Window Help

Scheduling.java x

```
class Scheduling {
 public static void main (String[] args) {
 Step fondamenta = new Step("fondamenta", new ArrayList<Step>());
 ArrayList<Step> prec1 = new ArrayList<>();
 prec1.add(fondamenta);

 Step muri = new Step("muri", prec1);
 Step me = new Step("muri esterni", prec1);
 Step camino = new Step("camino", prec1);

 ArrayList<Step> prec2 = new ArrayList<>();
 prec2.add(me);
 Step tetto = new Step("tetto", prec2);
 Step fi = new Step("finestre", prec2);

 ArrayList<Step> prec3 = new ArrayList<>();
 prec3.add(muri); I
 prec3.add(me);
 Step tappezzi = new Step("tappezzeria", prec3);
 Step porte = new Step("porte", prec3);

 fondamenta.start();

 muri.start();
 me.start();
 camino.start();
```

tetto.start();
 fi.start();

 porte.start();
 tappezzi.start();
 }
}

Tool Output

# Esempio di join: Scheduling di task per gestire un workflow - II



```
class Step extends Thread {
 List<Step> precondizioni;
 public Step(String str, List<Step> precondizioni) {
 super(str); this.precondizioni = precondizioni;
 }
 private void myJoin() {
 for (Thread prec : precondizioni) {
 try {prec.join();} catch (InterruptedException e)
 {System.out.println(e.getMessage());}
 }
 }
 public void run() {
 myJoin();
 try {sleep((long)(Math.random() * 1000));}
 } catch (InterruptedException e)
 {System.out.println(e.getMessage());}
 System.out.println(getName() + " terminato");
 }
}
```

# Esempio di join: Scheduling di task per gestire un workflow - III



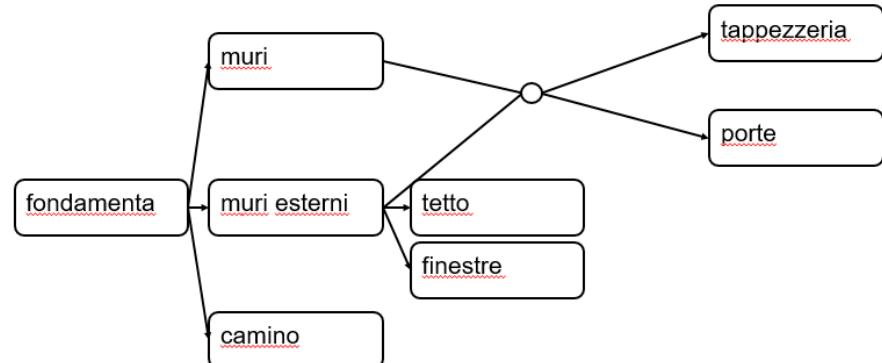
```
public static void main (String[] args) {
 Step fondamenta = new Step("fondamenta", new ArrayList<Step>());

 ArrayList<Step> prec1 = new ArrayList<>();
 prec1.add(fondamenta);

 Step muri = new Step("muri", prec1);
 Step me = new Step("muri esterni", prec1);
 Step camino = new Step("camino", prec1);

 ArrayList<Step> prec2 = new ArrayList<>();
 prec2.add(me);
 Step tetto = new Step("tetto", prec2);
 Step fi = new Step("finestre", prec2);

 // ... continua
```



# Esempio di join: Scheduling di task per gestire un workflow - IV



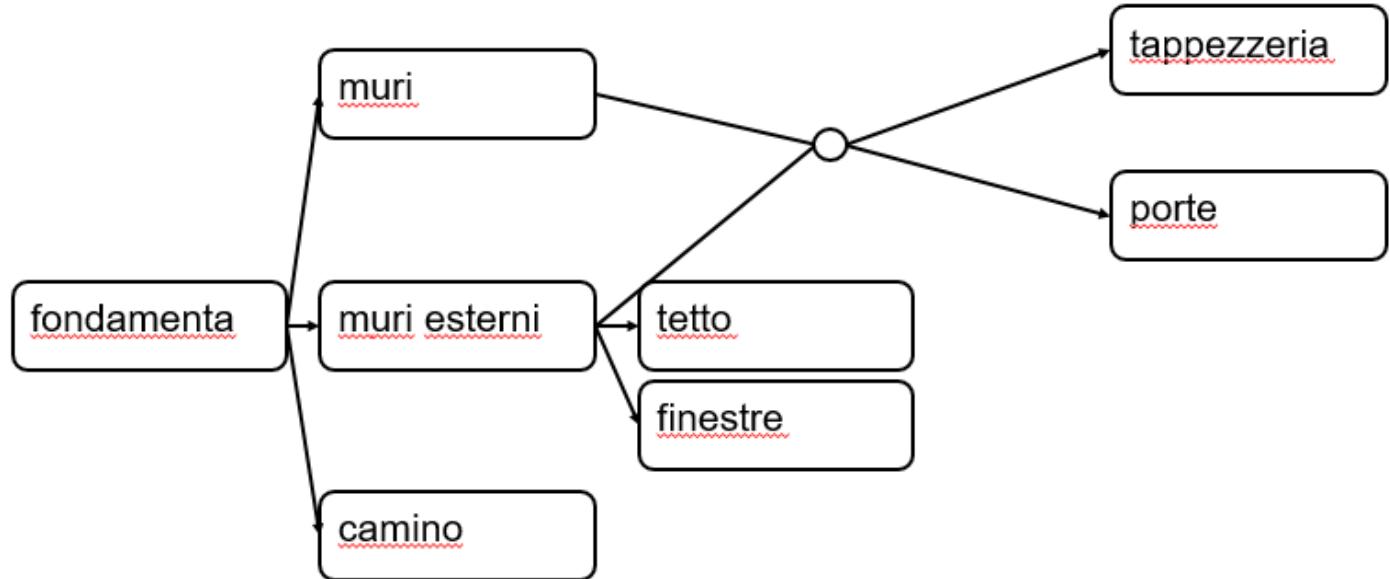
```
ArrayList<Step> prec3 = new ArrayList<>();
prec3.add(muri);
prec3.add(me);
Step tappezzi = new Step("tappezzeria", prec3);
Step porte = new Step("porte", prec3);
```

```
fondamenta.start();
```

```
muri.start();
me.start();
```

```
camino.start();
tetto.start();
fi.start();
porte.start();
tappezzi.start();
```

```
}
```





# Programmazione III

Prof.ssa Liliana Ardissono  
Dipartimento di Informatica  
Università di Torino

## Programmazione parallela con i Java Thread – parte 1



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND



# Thread

**Thread:** flusso sequenziale di controllo (esecuzione di istruzioni) in un programma.

Nello stesso programma si possono far partire più *Thread* che sono eseguiti in modo concorrente.

Tutti i *Thread* condividono le stesse variabili del programma.

A differenza dai *processi* che hanno ciascuno il proprio contesto, PID, etc..

Un *Thread* viene visto come un *lightweight process*.

Nei computer a singola CPU la concorrenza viene simulata con una politica di *scheduling* che alterna l'esecuzione dei singoli *Thread*.



Una applicazione Java che usa i Thread può eseguire più attività contemporaneamente. Esempio: aggiornare l'informazione grafica sullo schermo e accedere alla rete.

In alcuni casi i Thread possono procedere in modo indipendente uno dall'altro (comportamento asincrono), in altri devono essere sincronizzati fra loro (es. produttore - consumatore).

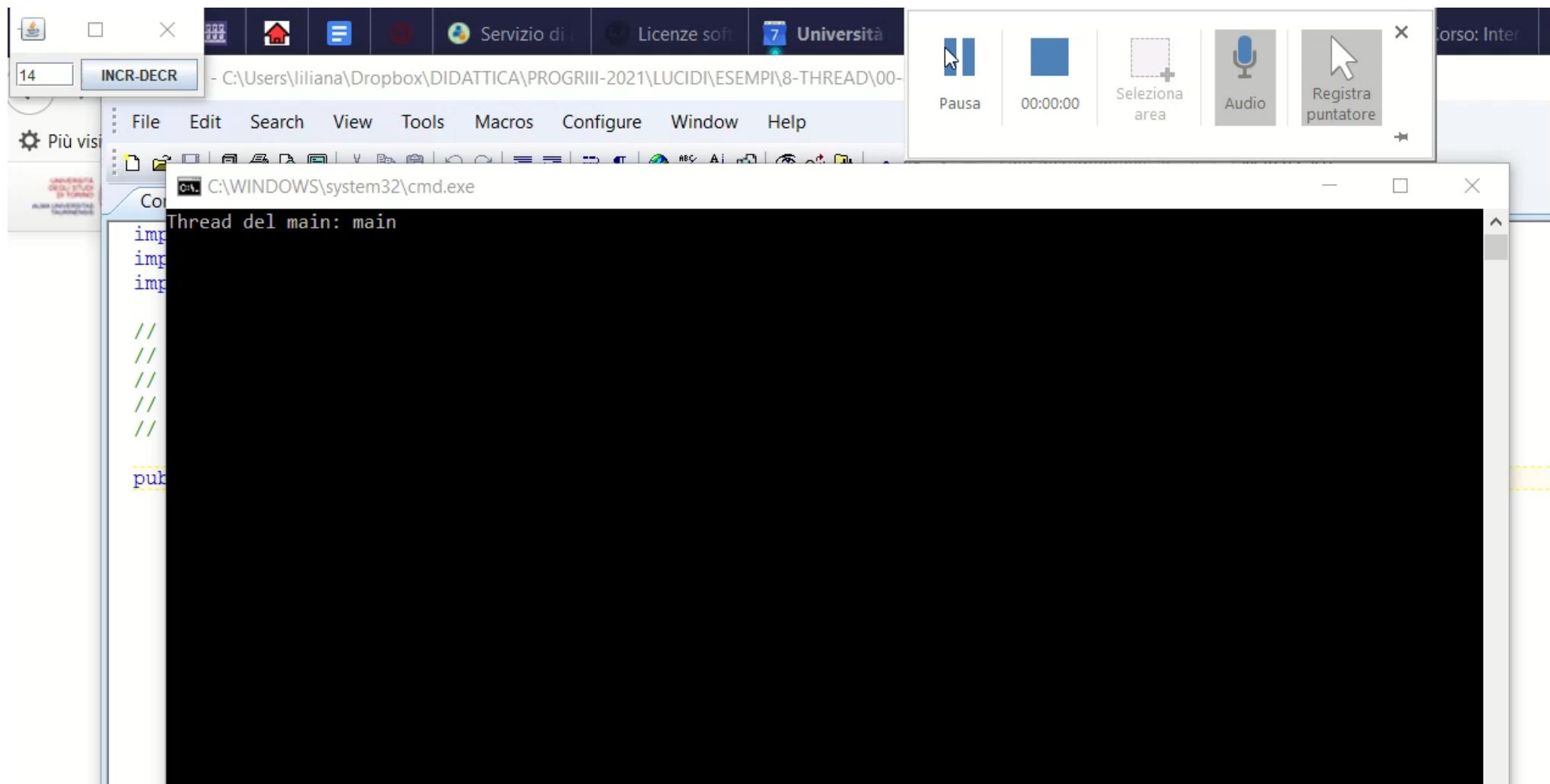
Ad esempio, quando un main crea una finestra (**JFrame**), viene attivato un **Thread** di interfaccia utente, diverso da quello del main().



Il costruttore del **JFrame** è eseguito dal Thread del main(), mentre il codice dei *listener* viene eseguito dal nuovo Thread (che gestisce tutti i listener).

I Thread sono indispensabili per realizzare interfacce grafiche che rispondano prontamente ai comandi dell'utente, anche se il programma sta eseguendo altre computazioni. In generale, li si usa per ottenere un uso ottimale della CPU (facendo in modo che un flusso di esecuzione si sospenda quando è in attesa di condizioni, I/O, etc., e lasciando la CPU a flussi che possono essere portati avanti).

# Esecuzione di Contatore2

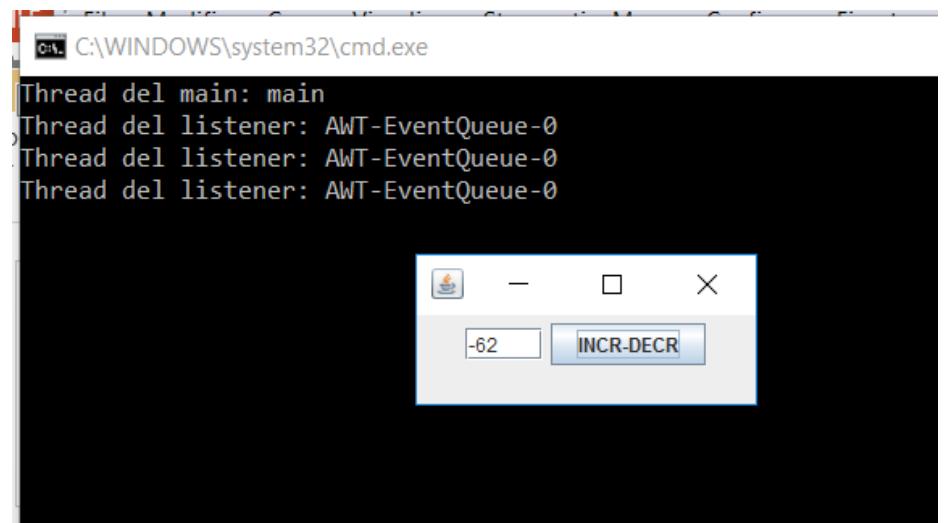




## Esempio: Contatore 2

La label visualizza il valore della variabile intera *count*, mentre il bottone può cambiare il valore della variabile booleana *runFlag*.

Dopo aver creato la finestra, il main() entra in un ciclo infinito in cui la variabile *count* viene incrementata o decrementata a seconda del valore di *runFlag*. Nonostante il main() esegua una computazione che non termina, è possibile eseguire il metodo *actionPerformed()* del listener del bottone ogni volta che questo viene premuto.



```
public class Contatore2 {
 private static boolean runFlag = true;

 public static void main(String[] argv) {
 JButton onOff = new JButton("ON-OFF");
 JTextField t = new JTextField(4);
 JFrame f = new JFrame();
 int count = 0;
 setLayout(new FlowLayout());
 onOff.addActionListener(new ActionListener()
 {
 public void actionPerformed(ActionEvent e)
 {
 runFlag = !runFlag;
 }
 });
 f.add(onOff);
 f.add(t);

 while (true) {
 try {Thread.sleep(500);}
 catch(InterruptedException e) {}
 if (runFlag) t.setText(Integer.toString(count++));
 else t.setText(Integer.toString(count--));
 }
 }
}
```



# classe Thread



**sleep(long millis)** è un metodo di Thread che blocca l'esecuzione del Thread per il numero specificato di millisecondi.

Può generare una eccezione **InterruptedException**.

```
try {
 Thread.sleep(500);
} catch(InterruptedException e) {...}
```

**sleep()** è un metodo statico e quindi può essere usato con **Thread.sleep(...)** anche in una classe che non deriva da Thread, per far bloccare l'esecuzione del Thread che lo esegue.



Per sapere in quale Thread ci si trova si può usare il metodo statico `currentThread()`:

**Thread.currentThread().getName()**

Nell'esempio si può verificare che, quando si usa una interfaccia grafica, ci sono (almeno) due Thread. Uno per il `main()` e l'altro per l'interfaccia grafica.

Anche se si creano più finestre, c'è un solo Thread per l'esecuzione di tutte le finestre.



## Esempio: Contatore 1 – non funziona bene

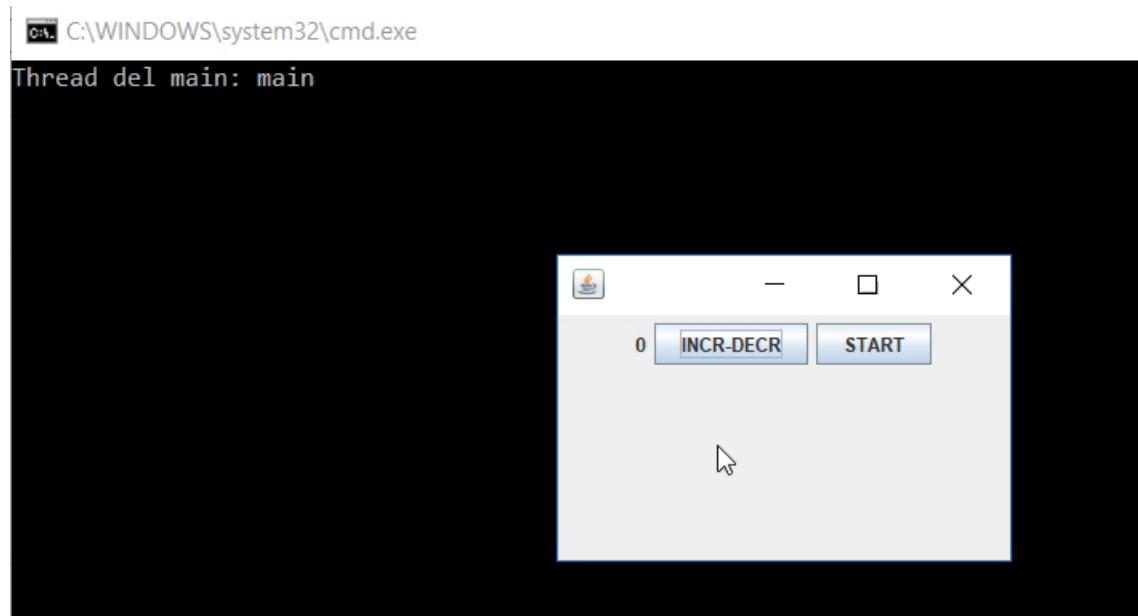
Vediamo adesso un altro esempio analogo al precedente ([Contatore1](#)).

In questo caso la computazione infinita viene eseguita dal listener di un bottone START, che impedisce qualunque altra computazione dell'interfaccia grafica, in particolare la gestione di altri eventi.

Dopo aver premuto START, la finestra non viene più rinfrescata.

Il bottone ON-OFF non ha effetto e non si riesce a chiudere la finestra e terminare l'esecuzione.

# Esecuzione di Contatore1





```
public class Contatore1 extends JFrame
{
 private int count = 0;
 private JButton onOff = new JButton("ON-OFF");
 private JButton start = new JButton("START");
 private JTextField t = new JTextField(4);
 private boolean runFlag = true;
 public Contatore1()
 { }

 class OnOffL implements ActionListener
 {
 public void actionPerformed(ActionEvent e)
 {
 runFlag = !runFlag;
 }
 }

 class StartL implements ActionListener {
 public void actionPerformed(ActionEvent e) {
 while (true) {
 try {Thread.sleep(500);}
 catch(InterruptedException exc) {}
 if (runFlag) t.setText(Integer.toString(count++));
 else t.setText(Integer.toString(count--));
 System.out.println(count);
 }
 }
 }
}
```

# Come si crea un Thread



Si definisce una classe che eredita da **Thread** e che ridefinisce il metodo **run()** per specificare le operazioni che il Thread deve fare.  
Il metodo **run()** di **Thread** non fa niente.

```
class MiaClasse extends Thread {
 public void run() {
 System.out.println("Sono il thread " + getName());
 }
}
```

```
....
MiaClasse t1 = new MiaClasse();
MiaClasse t2 = new MiaClasse();
t1.start();
t2.start();
```

```
C:\WINDOWS\system32
Sono il thread main
Sono il thread Thread-0
Sono il thread Thread-1
Press any key to continue . . .
```

Per avviare un Thread si deve eseguire il metodo **start()**, che manda in esecuzione un nuovo Thread e poi invoca **run()**; start() lancia una eccezione se viene invocato più di una volta. NB: se si invoca direttamente il metodo **run()**, questo viene eseguito ma non viene creato un nuovo Thread.



Altra formulazione: **start()** viene chiamato direttamente dal costruttore del **Thread**.

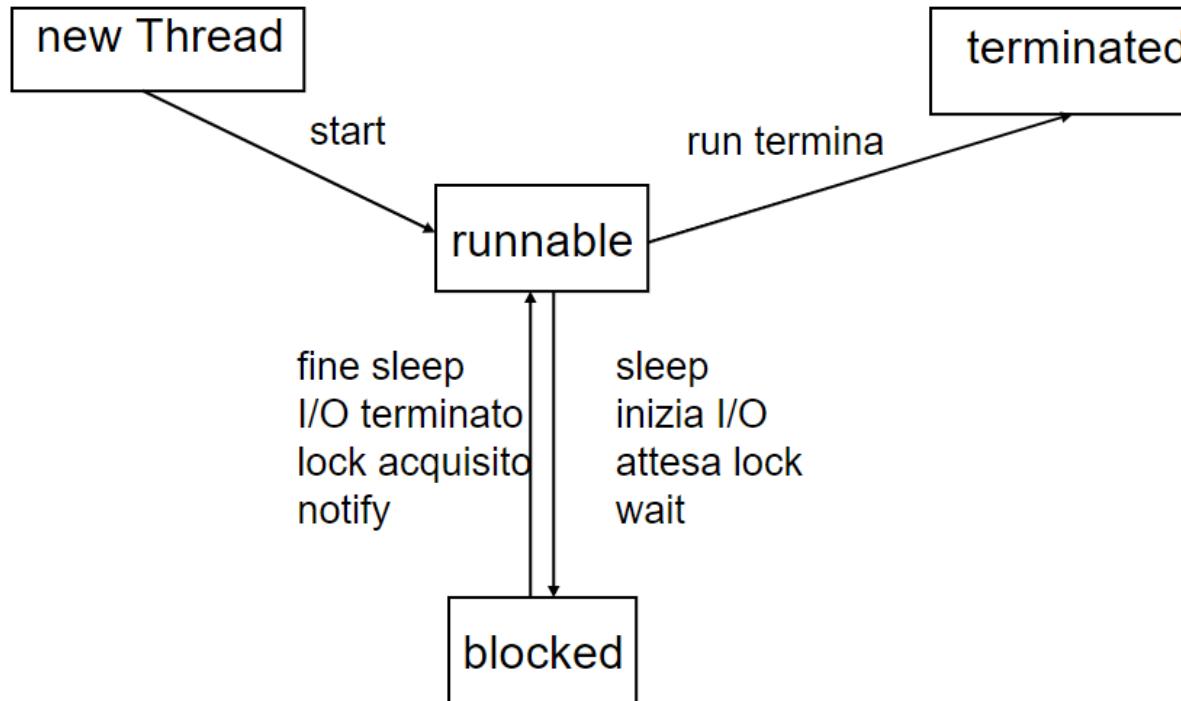
```
class MiaClasse extends Thread
{ public MiaClasse(String s)
{ super(s);
 start();
}
public void run()
{System.out.println("Sono il thread " + getName());}
}
....
new MiaClasse("primo");
new MiaClasse("secondo");
```

**NOTA** Anche se l'oggetto **MiaClasse** non viene assegnato ad una variabile non ci sono problemi col Garbage Collector: l'oggetto rimane finché il Thread non termina.

```
C:\WINDOWS\system...
Sono il thread main
Sono il thread secondo
Sono il thread primo
Press any key to continue . . .
```



# Ciclo di vita di un Thread





NB: runnable NON significa running!

Un Thread **runnable** può essere o meno in esecuzione.

Se c'è una sola CPU ci sarà al massimo un Thread in esecuzione ad ogni istante. L'effettiva esecuzione dipende dalla politica dello scheduler.

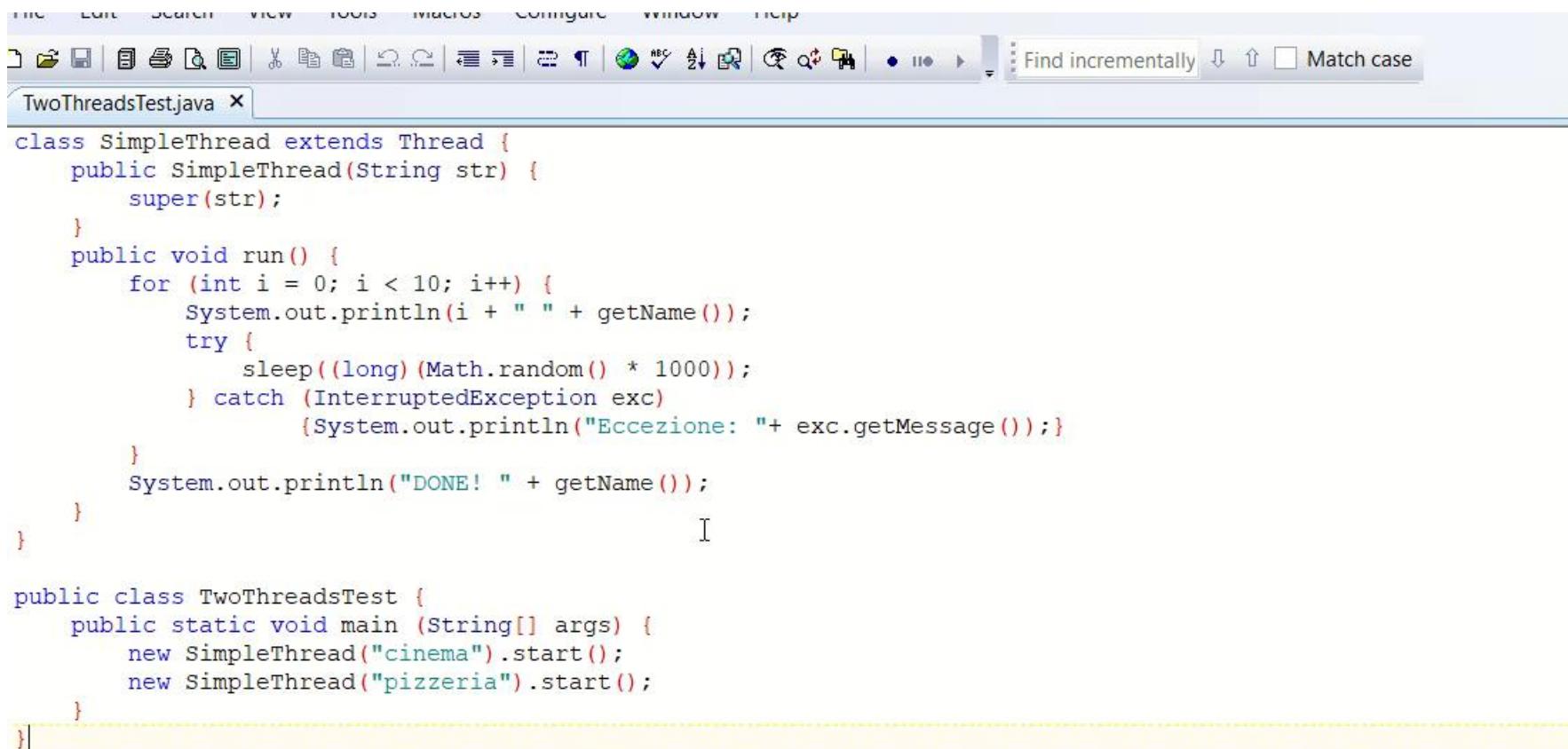
La scelta di quale Thread eseguire e per quanto tempo è arbitraria (es. preemptive scheduling).

# Esempio: TwoThreadsTest



```
class SimpleThread extends Thread {
 public SimpleThread(String str) {super(str);}
 public void run() {
 for (int i = 0; i < 10; i++) {
 System.out.println(i + " " + getName());
 try {sleep((long)(Math.random() * 1000));}
 } catch (InterruptedException exc)
 {System.out.println("Eccezione: "+ exc.getMessage());}
 }
 System.out.println("DONE! " + getName());
 }
}
public class TwoThreadsTest {
 public static void main (String[] args) {
 new SimpleThread("cinema").start();
 new SimpleThread("pizzeria").start();
 }
}
```

# TwoThreadsTest - esecuzione



```
class SimpleThread extends Thread {
 public SimpleThread(String str) {
 super(str);
 }
 public void run() {
 for (int i = 0; i < 10; i++) {
 System.out.println(i + " " + getName());
 try {
 sleep((long)(Math.random() * 1000));
 } catch (InterruptedException exc) {
 System.out.println("Eccezione: " + exc.getMessage());
 }
 }
 System.out.println("DONE! " + getName());
 }
}

public class TwoThreadsTest {
 public static void main (String[] args) {
 new SimpleThread("cinema").start();
 new SimpleThread("pizzeria").start();
 }
}
```



Ritorniamo all'esempio iniziale (Contatore1) in cui il listener di un bottone esegue una computazione infinita, bloccando il funzionamento del resto dell'interfaccia.

Si può risolvere il problema creando un nuovo Thread e eseguendo il ciclo infinito del listener in questo Thread.

In questo modo il Thread dell'interfaccia grafica può servire gli altri eventi.

# ContConThread



```
ThreadCont tc = new ThreadCont();
```

```
.....
```

```
class ThreadCont extends Thread {
 public void run() {
 while (true) {
 try {
 Thread.sleep(500);
 } catch(InterruptedException exc) {}
 if (runFlag) t.setText(Integer.toString(count++));
 else t.setText(Integer.toString(count--));
 }
 }
}
```

```
class StartL implements ActionListener {
 public void actionPerformed(ActionEvent e) {
 tc.start();
 }
}
```





# Gestione eccezioni nei Thread

Il metodo run() di un Thread non può fare throw di eccezioni controllate → vanno gestite tutte con opportuni handler (try e catch).

Se si verifica un'eccezione non controllata si blocca il Thread e si visualizza a video lo stack di esecuzione.

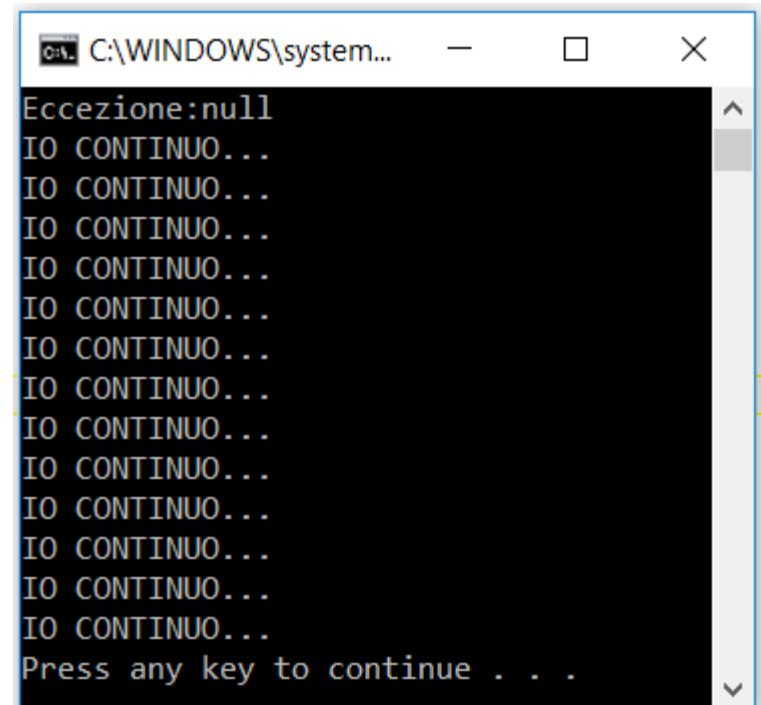
Volendo gestire le eventuali eccezioni non controllate: aggiungere al Thread un UncaughtExceptionHandler, implementando il suo metodo uncaughtException() per gestire le eccezioni. **Vd. EccezioneApp1**



# EccezioneApp1 - I

```
public MyThread() {
 super();
 setUncaughtExceptionHandler(new UncaughtExceptionHandler() {
 public void uncaughtException(Thread th, Throwable exc) {
 System.out.println("Eccezione:"+ exc.getMessage());
 }
 });
}
public void run() {
 // genero eccezione non controllata
 // (RunTimeException);
 String s=null;
 s.toString();
}
}

(IO CONTINUO... viene visualizzato dal main())
```



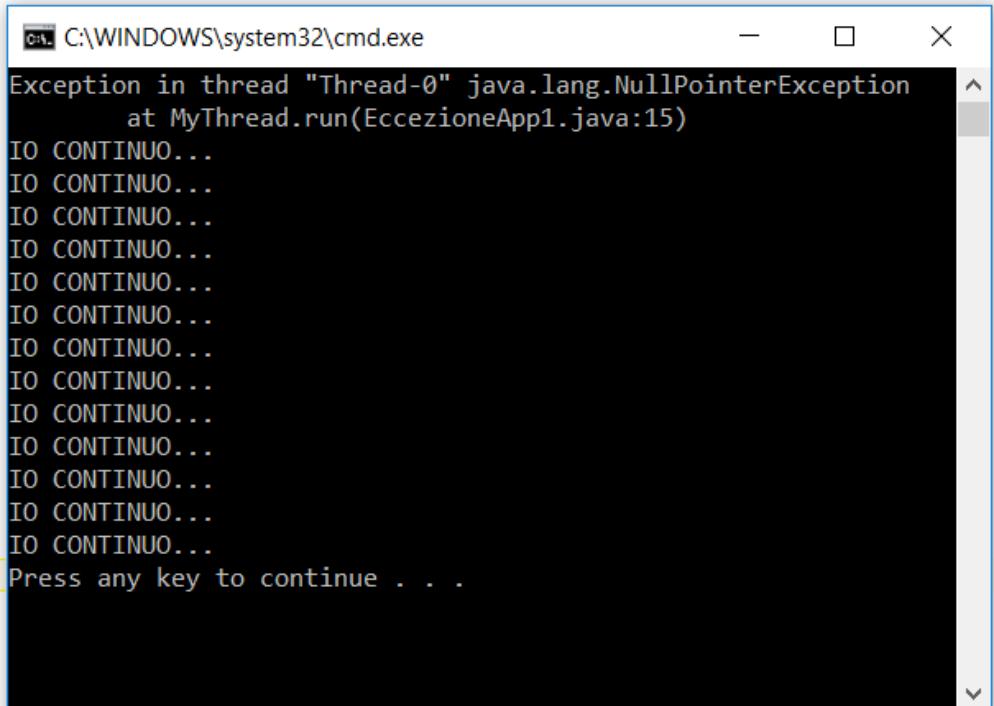
```
C:\WINDOWS\system...
Eccezione:null
IO CONTINUO...
Press any key to continue . . .
```



# EccezioneApp1 - II

```
public MyThread() {
 super();
 /* setUncaughtExceptionHandler(new UncaughtExceptionHandler() {
 public void uncaughtException(Thread th, Throwable exc) {
 System.out.println("Eccezione:"+ exc.getMessage());
 }
 }); */
}

public void run() {
 // genero eccezione non controllata
 // (RunTimeException);
 String s=null;
 s.toString();
}
```



```
C:\WINDOWS\system32\cmd.exe
Exception in thread "Thread-0" java.lang.NullPointerException
 at MyThread.run(EccezioneApp1.java:15)
IO CONTINUO...
Press any key to continue . . .
```



# Interface Runnable

Si definisce una classe che implementa l'interfaccia **Runnable** che possiede il metodo **run()**.

```
class Esempio implements Runnable {
 public void run() {...}}
```

Per attivare un Thread è necessario creare un **Thread** passando come parametro al costruttore un oggetto **Runnable**. Quando si fa partire il Thread con **start()**, inizia l'esecuzione del metodo **run()** nel nuovo Thread.

```
Esempio es = new Esempio();
Thread t = new Thread(es),
t.start();
```

# ProvaRunnable



```
class MiaClasse implements Runnable {
 public void run() {
 System.out.println("Sono il thread " +
 Thread.currentThread().getName());
 }

 MiaClasse mt = new MiaClasse();
 new Thread(mt).start();
 new Thread(mt).start();
```

Questo secondo modo di creare i Thread deve essere usato quando la classe contenente il metodo **run()** è già sottoclasse di un'altra classe (ereditarietà singola). **In generale, usare i Runnable è utile per separare le specifiche delle istruzioni da eseguire in un flusso parallelo rispetto all'esecutore, che è l'oggetto Thread.**

# ProvaRunnable2 - I



```
class MiaClasse implements Runnable {
 private int i = 0;
 public void run() {
 i++;
 System.out.println(i);
 }

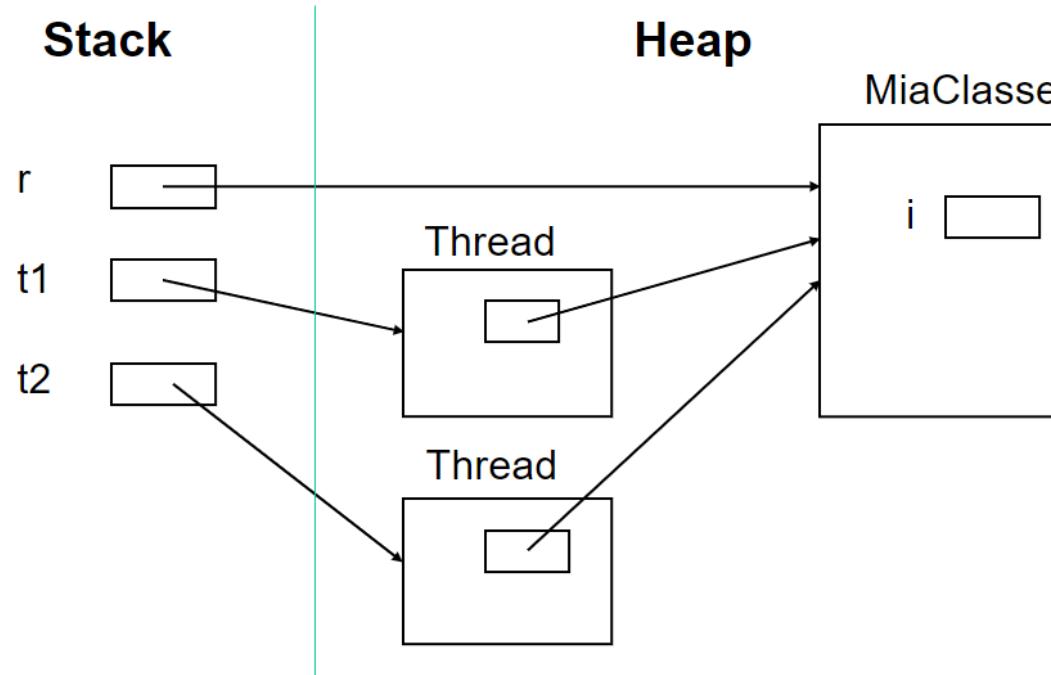
 MiaClasse r = new MiaClasse();
 Thread t1 = new Thread(r);
 Thread t2 = new Thread(r);
 t1.start();
 t2.start();
```

```
C:\WINDOWS\system32\cmd....
thread: main
i = 2
i = 2
nella run- thread Thread-1: i = 2
nella run- thread Thread-0: i = 2
Press any key to continue . . .
```

La variabile **i** viene incrementata due volte. Infatti esiste un unico oggetto di **MiaClasse**, legato alla variabile **r**, che ha una variabile locale **i**. I Thread **t1** e **t2** eseguono entrambi il metodo **run()** di questo oggetto e incrementano la stessa variabile **i**.



# ProvaRunnable2 - II





Per capire meglio il funzionamento dei Thread introduciamo un modello semplificato di esecuzione, che è una estensione di quello per l'esecuzione di programmi sequenziali (vedi slide su gestione della memoria).

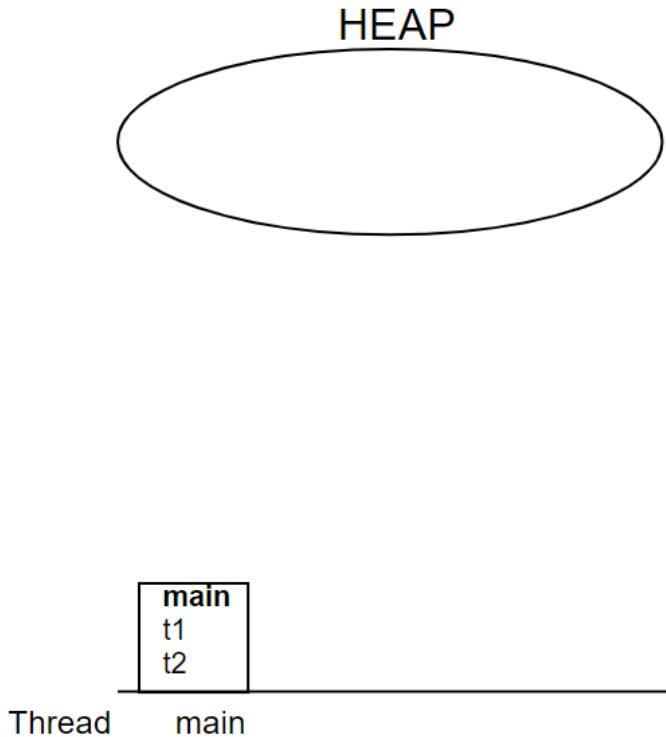
La differenza principale, se il programma contiene dei Thread, è dovuta al fatto che **ogni Thread ha un suo stack di esecuzione**.  
Invece lo heap, viene condiviso.

Inoltre, ad ogni passo, la macchina virtuale si dovrà ricordare quale è il Thread *running*.



```
class EsempioT extends Thread {
 private int counter = 0;
 public void run() {
 incr();
 return;
 }
 public void incr() {
 counter++;
 return;
 }
public static void main(String[] args){
 EsempioT t1 = new EsempioT();
 EsempioT t2 = new EsempioT();
 t1.start();
 t2.start();
}
}
```

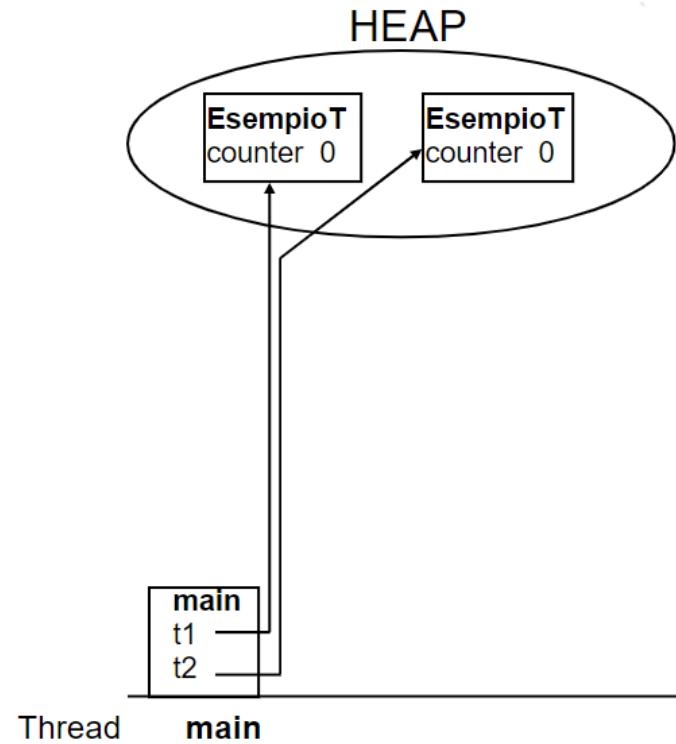
**Running:** Thread main  
In **grassetto** l'istruzione  
che viene eseguita.





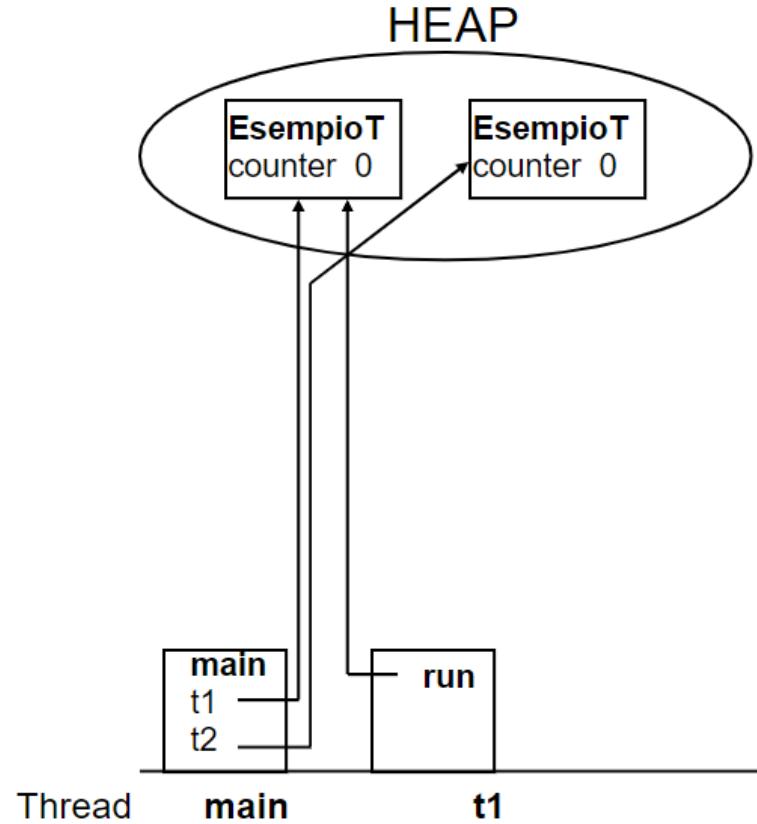
```
class EsempioT extends Thread {
 private int counter = 0;
 public void run() {
 incr();
 return;
 }
 public void incr() {
 counter++;
 return;
 }
 public static void main(String[] args){
 EsempioT t1 = new EsempioT();
 EsempioT t2 = new EsempioT();
 t1.start();
 t2.start();
 }
}
```

Running: Thread main





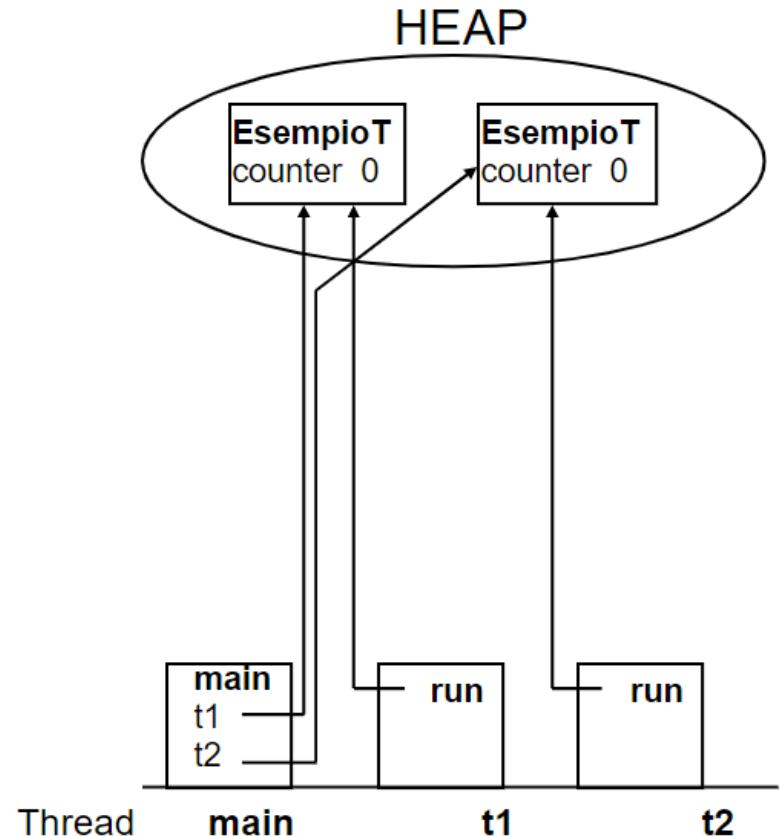
```
class EsempioT extends Thread {
 private int counter = 0;
 public void run() {
 incr();
 return;
 }
 public void incr() {
 counter++;
 return;
 }
 public static void main(String[] args){
 EsempioT t1 = new EsempioT();
 EsempioT t2 = new EsempioT();
 t1.start();
 t2.start();
 }
}
```



**Running:** Thread main



```
class EsempioT extends Thread {
 private int counter = 0;
 public void run() {
 incr();
 return;
 }
 public void incr() {
 counter++;
 return;
 }
 public static void main(String[] args){
 EsempioT t1 = new EsempioT();
 EsempioT t2 = new EsempioT();
 t1.start();
 t2.start();
 }
}
```

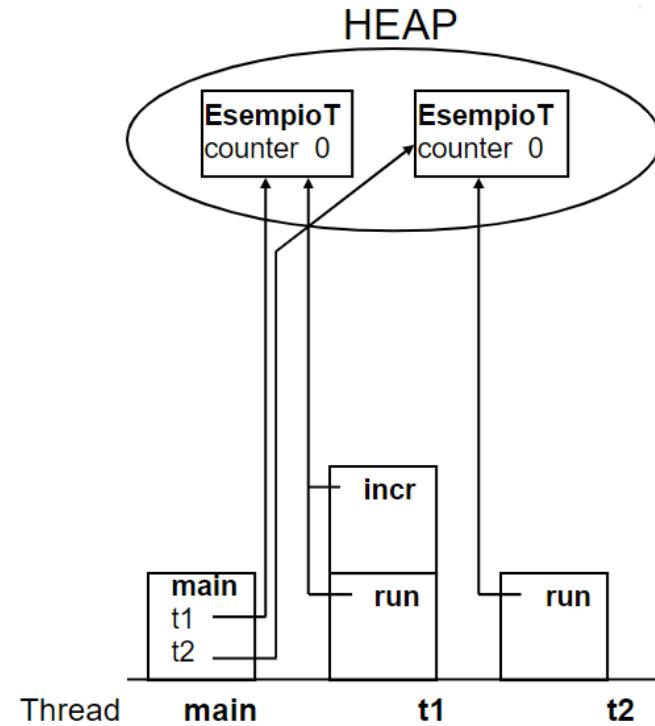


**Running:** Thread main



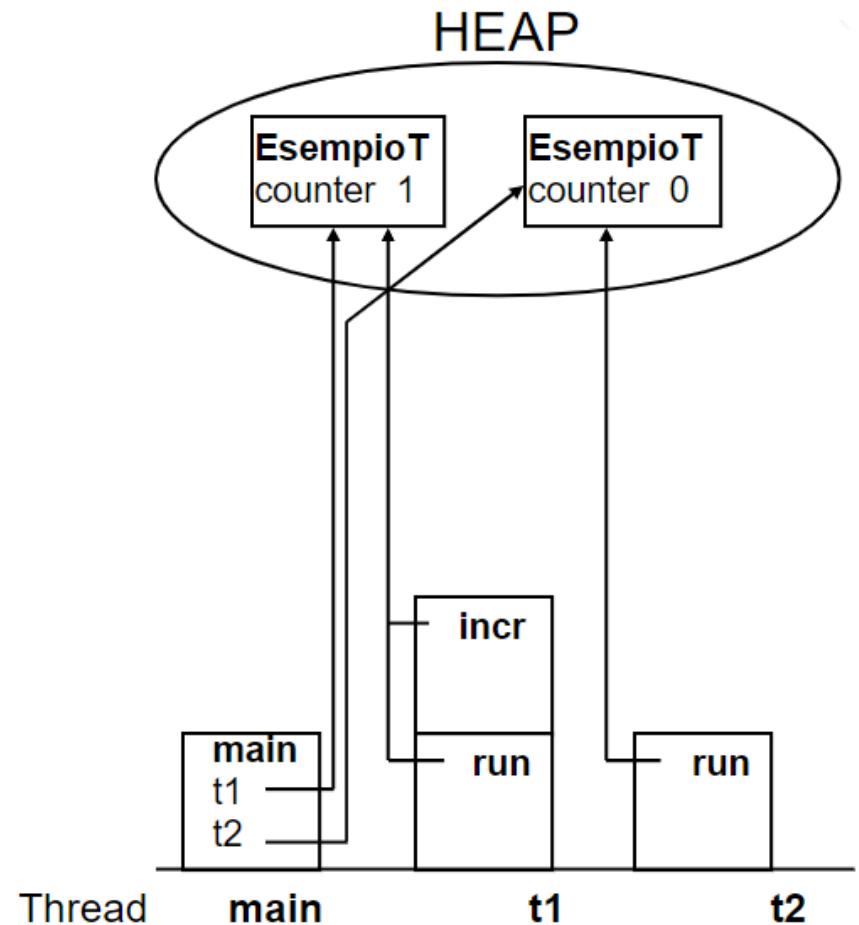
```
class EsempioT extends Thread {
 private int counter = 0;
 public void run() {
 incr();
 return;
 }
 public void incr() {
 counter++;
 return;
 }
 public static void main(String[] args){
 EsempioT t1 = new EsempioT();
 EsempioT t2 = new EsempioT();
 t1.start();
 t2.start();
 }
}
```

**Running: Thread t1**





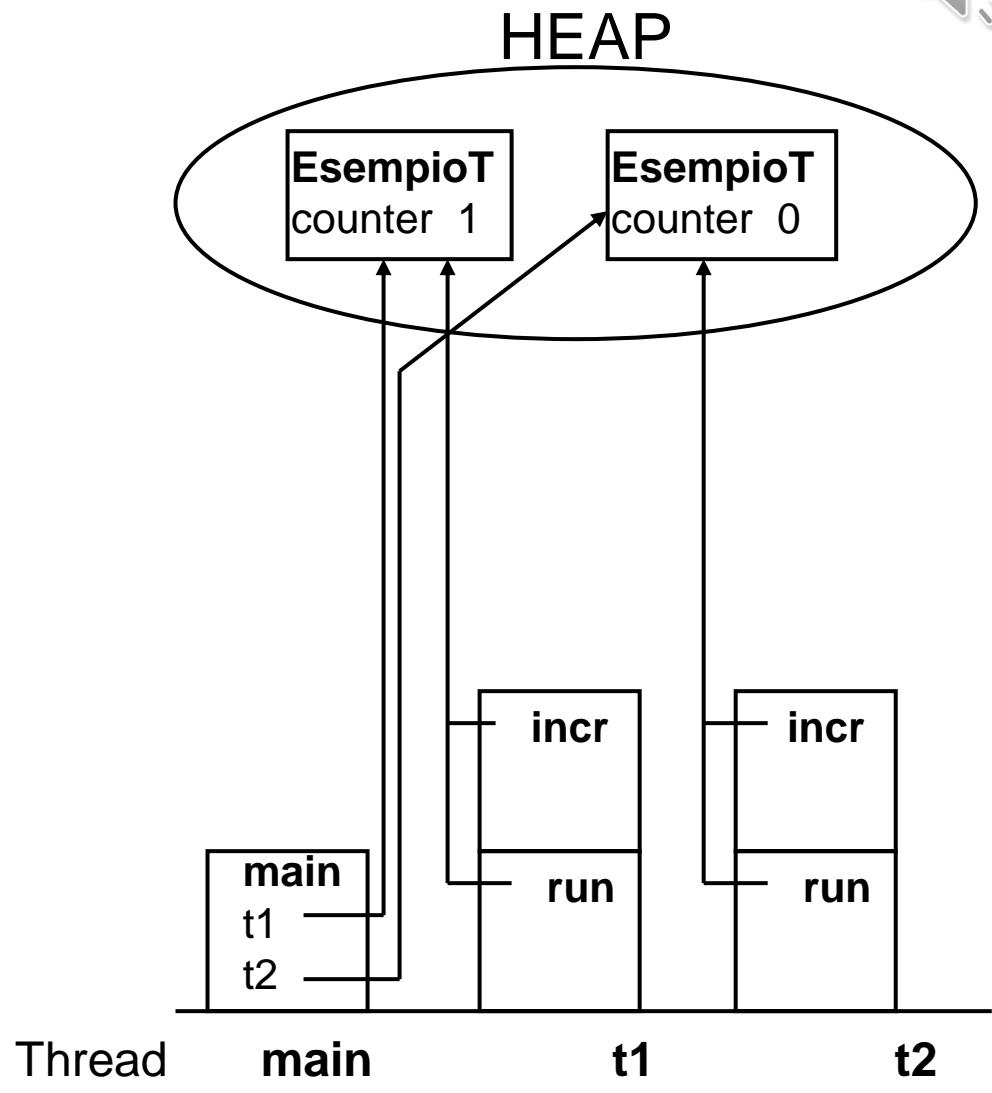
```
class EsempioT extends Thread {
 private int counter = 0;
 public void run() {
 incr();
 return;
 }
 public void incr() {
 counter++;
 return;
 }
 public static void main(String[] args){
 EsempioT t1 = new EsempioT();
 EsempioT t2 = new EsempioT();
 t1.start();
 t2.start();
 }
}
```



**Running:** Thread t1



```
class EsempioT extends Thread {
 private int counter = 0;
 public void run() {
 incr();
 return;
 }
 public void incr() {
 counter++;
 return;
 }
 public static void main(String[] args){
 EsempioT t1 = new EsempioT();
 EsempioT t2 = new EsempioT();
 t1.start();
 t2.start();
 }
}
```



**Running:** Thread t2



Naturalmente la computazione dei vari Thread può procedere con ordini diversi. Ad esempio, il Thread del **main()** può terminare prima dei Thread **t1** e **t2**, oppure viceversa.

Per avere la traccia di tutti gli stack in un qualunque punto della computazione si può usare il metodo statico **getAllStackTraces()** della classe **Thread**, che restituisce un **Map<Thread, StackTraceElement[]>**.

Ad ogni Thread attivo al momento della chiamata, viene associato un array di **StackTraceElement**.

# EsempioT – visualizzazione degli stack dei Thread della Java Virtual Machine



# Terminazione di Thread (metodi join())

Un Thread può chiamare il metodo **join()** su un altro Thread t per aspettare che t sia terminato prima di continuare la propria esecuzione.

Se un Thread chiama **t.join()** su un altro Thread t, il Thread chiamante viene sospeso finché t termina l'esecuzione del proprio metodo **run()**.

NB: il metodo **join()** va invocato DOPO aver fatto partire il Thread da attendere, altrimenti non è bloccante per chi lo invoca. Es:

```
Thread t1 = new MyThread();
t1.start();
try {t1.join();}
catch (InterruptedException e)
 {System.out.println(e.getMessage());}
//... Istruzioni da eseguire dopo la terminazione di t1
```

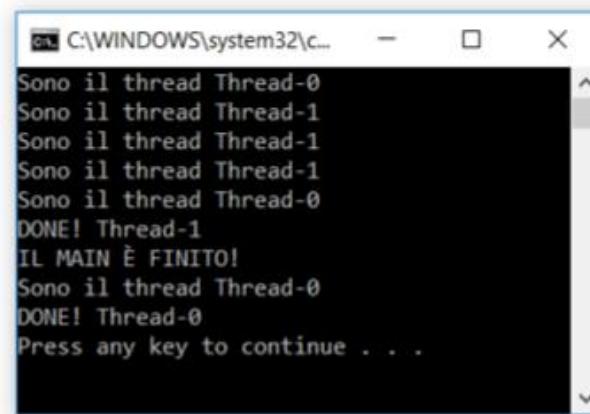
```

public class JoinApp {

 public static void main (String[] args) {
 Thread t1 = new MyThread(800);
 /* Se metto qui la join il main termina prima di t1 */
 /* try {
 t1.join();
 } catch (InterruptedException e) {System.out.println(e.getMessage());} */

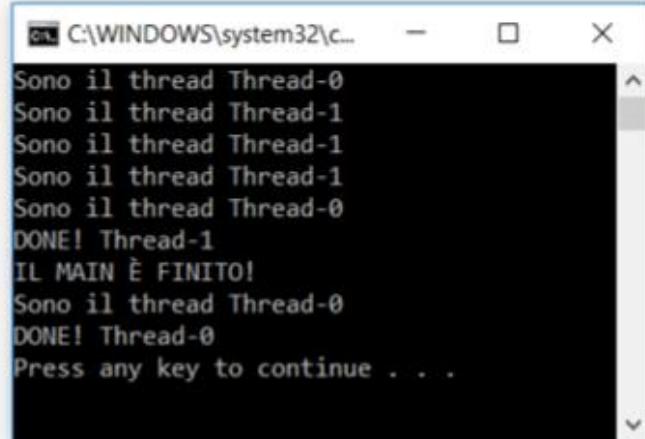
 t1.start();
 Thread t2 = new MyThread(300);
 t2.start();
 try {
 t2.join();
 } catch (InterruptedException e) {System.out.println(e.getMessage());}
 System.out.println("IL MAIN È FINITO!");
 }
}

```



```
class MyThread extends Thread {
 private int num;

 public MyThread(int num) {
 super();
 this.num = num;
 }
 public void run() {
 for (int i = 1; i < 4; i++) {
 System.out.println("Sono il thread " + getName());
 try {
 sleep(num);
 } catch (InterruptedException e) {return;}
 }
 System.out.println("DONE! " + getName());
 }
}
```

```
C:\WINDOWS\system32\cmd.exe
Sono il thread Thread-0
DONE! Thread-0
IL MAIN È FINITO!
Sono il thread Thread-0
DONE! Thread-0
Press any key to continue . . .
```



# Thread Demoni

L'esecuzione di un programma termina quando sono terminati *tutti i Thread* attivati. Un Thread può essere dichiarato come "**daemon**". Un *daemon* è un Thread normale, che però non influenza la terminazione del programma.

Un programma termina quando sono terminati *tutti i Thread non-daemon*. Se ci sono dei *daemon* attivi, la loro esecuzione viene terminata.

Per rendere un Thread Demone invocare il metodo `setDaemon(true)` prima del metodo `start()`. Es:

```
Class MyDaemon extends Thread {
 public MyDaemon() {setDaemon(true);}
 public void run() {...}
}
```

# Ringraziamenti



Grazie al Prof. Emerito Alberto Martelli del Dipartimento di Informatica dell'Università di Torino per aver redatto la prima versione di queste slides.



# Programmazione III

Prof.ssa Liliana Ardissono  
Dipartimento di Informatica  
Università di Torino

**Programmazione parallela con i Java Thread –  
parte 2**  
**Sincronizzazione di Thread**



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND



Finora abbiamo visto esempi di thread indipendenti.

In molti casi i thread si devono sincronizzare,

sia per coordinarsi per la risoluzione di un problema,

sia perché competono per una risorsa.

Consideriamo ad esempio il caso di più thread che usano la stessa stampante. Se un thread sta stampando un documento sulla stampante, gli altri thread che cercano di usare la stampante si devono fermare fino a quando il primo thread ha terminato.



Supponiamo di avere una classe **Stampante** con un metodo **stampa()** che stampa un array di stringhe (**Stampa**)

```
class Stampante
{ public void stampa(String[] a)
 { for(int i = 0; i < a.length; i++)
 { try {Thread.sleep((long)(Math.random() * 100));}
 catch(InterruptedException e) {}
 System.out.println(a[i]);
 }
 }
}
```



e una classe **ThreadStampa** che realizza un thread che stampa un array a sulla stampante st.

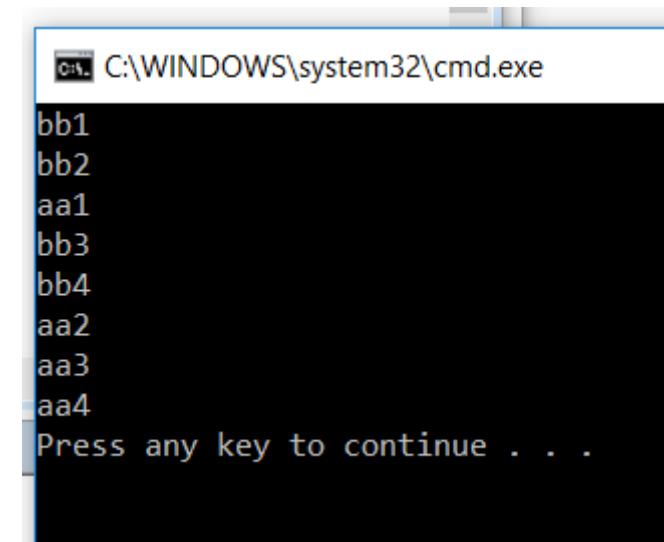
```
class ThreadStampa extends Thread
{ String[] a;
 Stampante st;
 public ThreadStampa(String[] arr, Stampante s)
 { a = arr; st = s;}
 public void run()
 { st.stampa(a);}
}
```



se si esegue questo main()

```
public static void main(String[] args)
{
 Stampante st = new Stampante();
 String[] a = {"aa1", "aa2", "aa3", "aa4"};
 String[] b = {"bb1", "bb2", "bb3", "bb4"};
 ThreadStampa t1 = new ThreadStampa(a,st);
 ThreadStampa t2 = new ThreadStampa(b,st);
 t1.start();
 t2.start();
}
```

le righe di stampa dei due array  
**a** e **b** vengono mescolate,  
perché i due thread eseguono  
contemporaneamente il metodo  
**stampa()** dell'oggetto  
**Stampante**.



```
bb1
bb2
aa1
bb3
bb4
aa2
aa3
aa4
Press any key to continue . . .
```



```
public class AccessoWrong {
 public static void main(String[] args) {
 C c = new C(); T t1 = new T(1, c); T t2 = new T(2, c);
 t1.start(); t2.start();
 try { t1.join();}
 catch(InterruptedException e1) {System.out.println(e1.getMessage());}
 try { t2.join(); }
 catch(InterruptedException e1) {System.out.println(e1.getMessage());}
 System.out.println("Main: c.i= " + c.i); }
 }
}
```

```
class C {
 public int i=0;
 public void m() {
 for(int k = 0; k < 100000; k++) i++;
 for(int k = 0; k < 100000; k++) i--;
 }
}
```

```
class T extends Thread {
 private int num; private C c;
 public T(int x, C y) { num = x; c = y; }
 public void run() {
 for (int i=0; i<10; i++) {
 c.m();
 System.out.println("Thread " + num + ": c.i= " + c.i);
 }
 }
}
```

| Output strumenti                   |
|------------------------------------|
| Thread 2: c.i= -26978              |
| Thread 1: c.i= 62714               |
| Thread 2: c.i= 72795               |
| Thread 1: c.i= -21658              |
| Thread 2: c.i= 100669              |
| Thread 2: c.i= 18008               |
| Thread 1: c.i= 18008               |
| Thread 2: c.i= 18008               |
| Thread 2: c.i= 18008               |
| Thread 2: c.i= 18008               |
| Thread 1: c.i= 18008               |
| Thread 2: c.i= 18008               |
| Thread 1: c.i= 18008               |
| Thread 2: c.i= 18008               |
| Thread 1: c.i= 18008               |
| Thread 2: c.i= 18008               |
| Thread 1: c.i= 18008               |
| Thread 2: c.i= 18008               |
| Thread 1: c.i= 18008               |
| Thread 2: c.i= 18008               |
| Thread 1: c.i= 18008               |
| Thread 2: c.i= 18008               |
| Main: c.i= 18008                   |
| Procedura completata correttamente |



## Vediamo un altro caso di un oggetto condiviso.

```
class MiaClasse implements Runnable
{
 int counter = 0;
 public void run()
 {
 incr();
 }
 public void incr()
 {
 counter++;
 }
}
```

Se ci sono due thread che eseguono contemporaneamente il metodo run() dell'oggetto condiviso, questi potrebbero interferire l'uno con l'altro producendo effetti non desiderati.

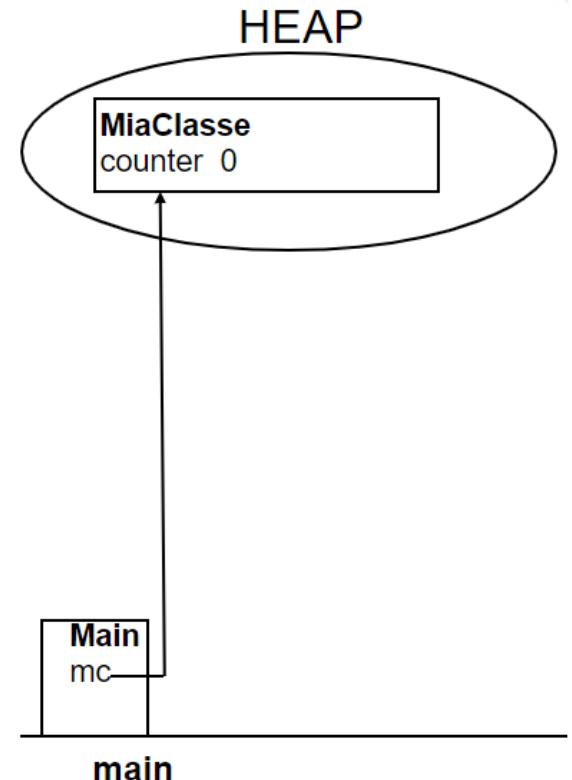


```
class MiaClasse implements Runnable
{
 int counter = 0;
 public void run()
 {
 incr();
 }
 public void incr()
 {
 counter++;
 }
 public static void main(String[] args)
 {
MiaClasse mc = new MiaClasse();
 Thread t1 = new Thread(mc);
 Thread t2 = new Thread(mc);
 t1.start();
 t2.start();
 }
}
```

**Running:** main

In rosso l'istruzione appena eseguita

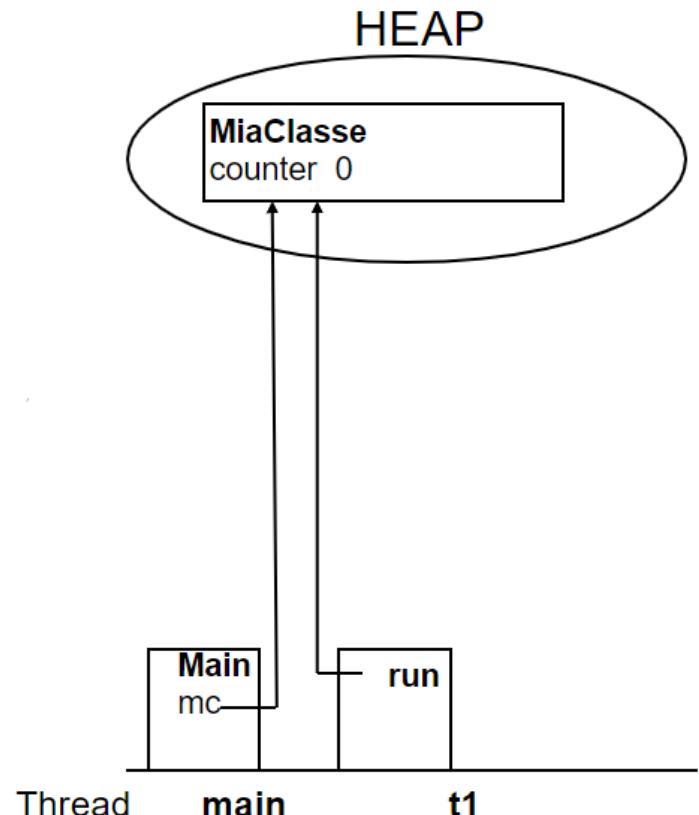
Thread





```
class MiaClasse implements Runnable {
 int counter = 0;
 public void run() {
 incr();
 }
 public void incr() {
 counter++;
 }
 public static void main(String[] args) {
 MiaClasse mc = new MiaClasse();
 Thread t1 = new Thread(mc);
 Thread t2 = new Thread(mc);
 t1.start();
 t2.start();
 }
}
```

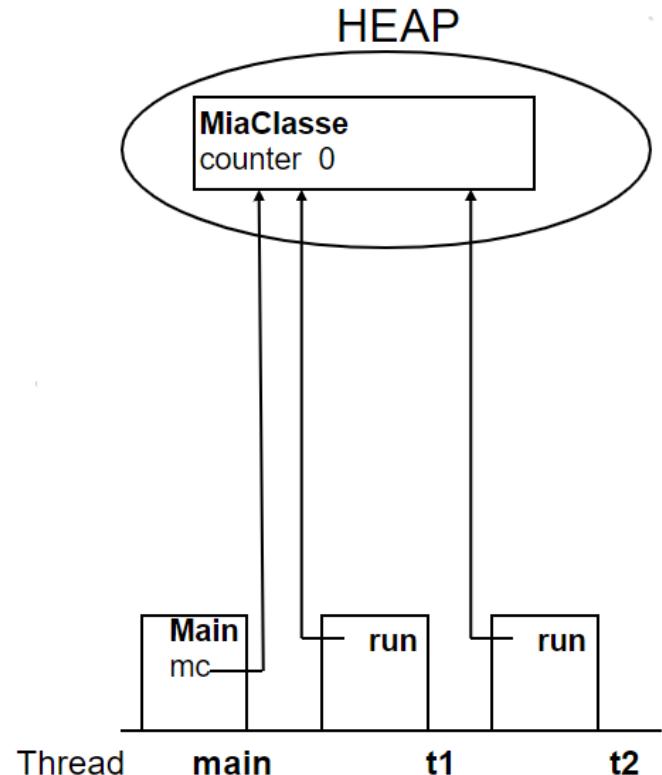
Running: main





```
class MiaClasse implements Runnable {
 int counter = 0;
 public void run() {
 incr();
 }
 public void incr() {
 counter++;
 }
 public static void main(String[] args)
 {
 MiaClasse mc = new MiaClasse();
 Thread t1 = new Thread(mc);
 Thread t2 = new Thread(mc);
 t1.start();
 t2.start();
 }
}
```

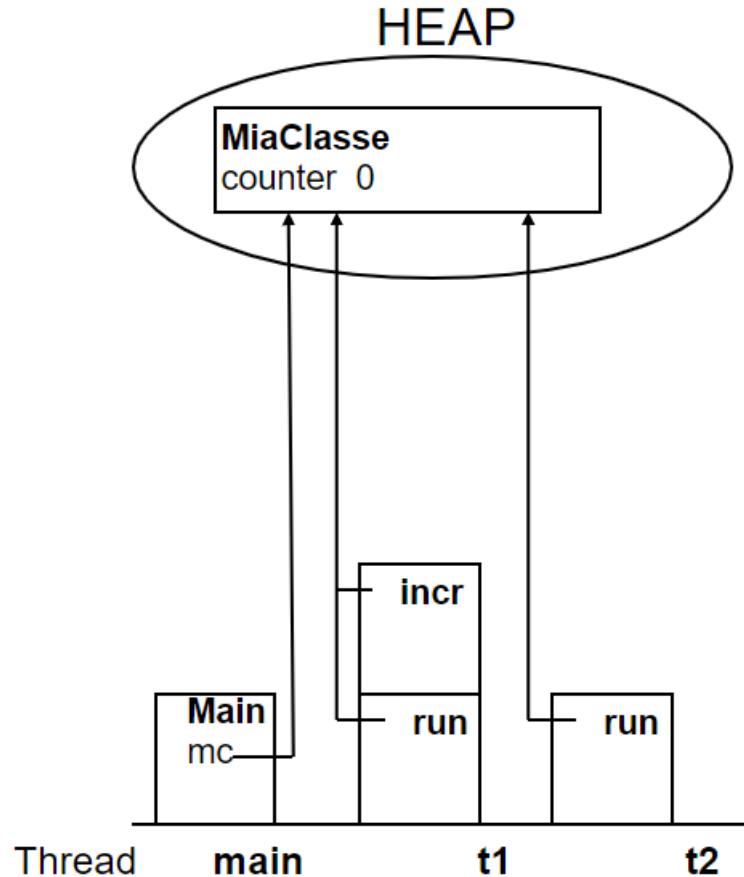
Running: main





```
class MiaClasse implements Runnable {
 int counter = 0;
 public void run() {
 incr();
 }
 public void incr() {
 counter++;
 }
 public static void main(String[] args) {
 MiaClasse mc = new MiaClasse();
 Thread t1 = new Thread(mc);
 Thread t2 = new Thread(mc);
 t1.start();
 t2.start();
 }
}
```

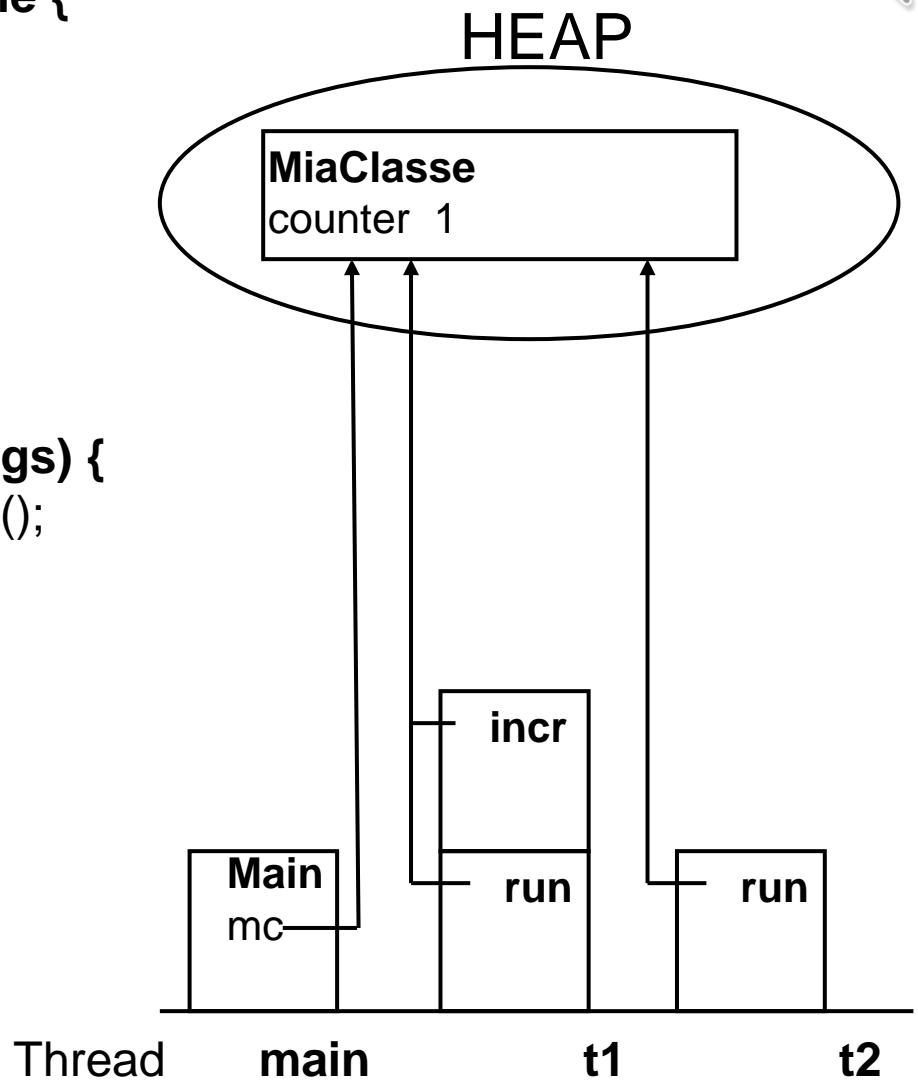
Running: t1





```
class MiaClasse implements Runnable {
 int counter = 0;
 public void run() {
 incr();
 }
 public void incr() {
 counter++;
 }
 public static void main(String[] args) {
 MiaClasse mc = new MiaClasse();
 Thread t1 = new Thread(mc);
 Thread t2 = new Thread(mc);
 t1.start();
 t2.start();
 }
}
```

Running: t1

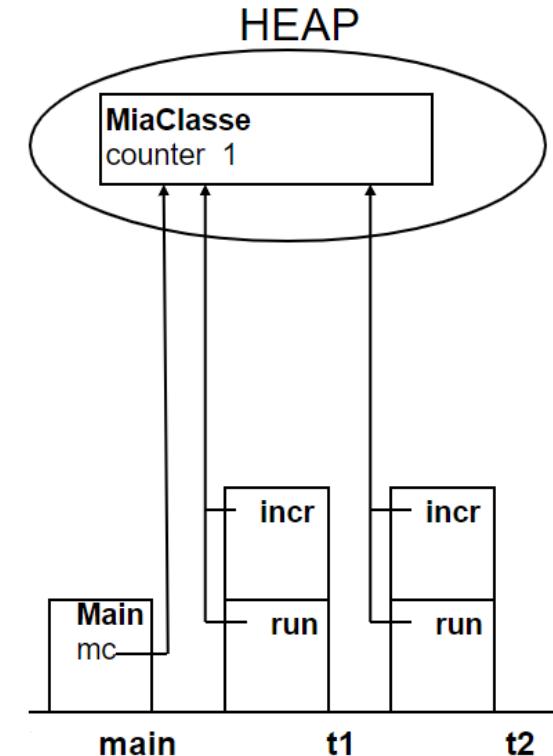




```
class MiaClasse implements Runnable {
 int counter = 0;
 public void run() {
 incr();
 }
 public void incr() {
 counter++;
 }
 public static void main(String[] args) {
 MiaClasse mc = new MiaClasse();
 Thread t1 = new Thread(mc);
 Thread t2 = new Thread(mc);
 t1.start();
 t2.start();
 }
}
```

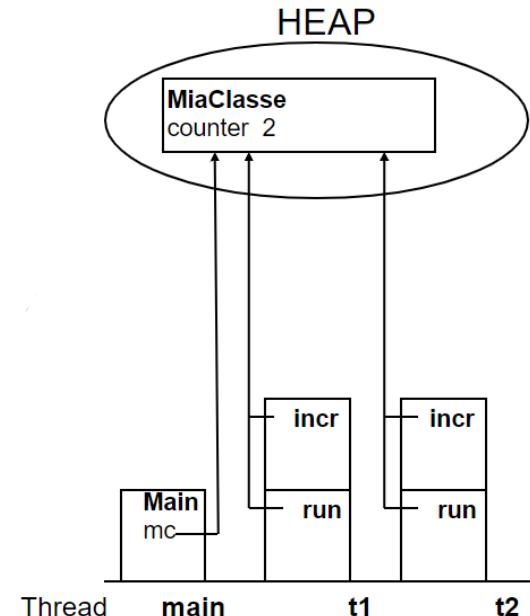
Running: t2

Thread





```
class MiaClasse implements Runnable {
 int counter = 0;
 public void run() {
 incr();
 }
 public void incr() {
 counter++;
 }
 public static void main(String[] args)
 {
 MiaClasse mc = new MiaClasse();
 Thread t1 = new Thread(mc);
 Thread t2 = new Thread(mc);
 t1.start();
 t2.start();
 }
}
```



**Running:** t2



Si potrebbe concludere che il programma termini sempre con **counter** = 2.

In realtà la conclusione si basa sull'assunzione che tutte le istruzioni, in particolare **counter++**, siano **atomiche**, ossia non interrompibili.

L'assunzione non è corretta. La macchina virtuale non esegue codice Java ma bytecode prodotto dal compilatore.

Ad esempio, l'istruzione **counter++** potrebbe essere tradotta in:

- carica **counter** in un registro
- incrementa il registro di 1
- sposta il risultato in **counter**

→ Se un thread si interrompe mentre esegue queste istruzioni, alla fine il **counter** potrebbe anche avere valore 1.

**Esempio da JAVA2 (secondo volume)** Abbiamo una classe **Bank** con un metodo **transfer()** per trasferire denaro da un conto ad un altro.



```
class Bank
{
 public Bank(int n, int initialBalance)
 {
 accounts = new int[n];
 int i;
 for (i = 0; i < accounts.length; i++)
 accounts[i] = initialBalance;
 }

 public void transfer(int from, int to, int amount)
 {
 if (accounts[from] < amount) return;
 accounts[from] -= amount;
 accounts[to] += amount;
 }

 private int[] accounts;
}
```



```
class BankTestWrong {
 public static void main(String[] args) {
 Bank b = new Bank(10, 1000);
 for (int i=0; i < 10; i++)
 new TransferThread(b, 100).start();
 }
}

class BankAccount {
 private int balance;

 public void deposit(int amount) {
 int temp = balance;
 temp = temp + amount;
 balance = temp;
 }

 public boolean withdraw(int amount) {
 if (amount > balance) return false;
 int temp = balance;
 try {
 Thread.sleep(1);
 } catch(InterruptedException e) {}
 temp = temp - amount;
 balance = temp;
 return true;
 }

 public int getBalance() {
 return balance; }
}
```

Output strumenti

```
trasferimento da 1 a 0 di 43
trasferimento da 4 a 3 di 86
trasferimento da 6 a 4 di 27
trasferimento da 7 a 5 di 54
trasferimento da 7 a 1 di 88
trasferimento da 2 a 8 di 56
trasferimento da 7 a 5 di 30
trasferimento da 8 a 5 di 43
trasferimento da 2 a 1 di 79
trasferimento da 0 a 2 di 31
Somma totale di denaro in banca: 9877
Somma totale di denaro in banca: 9978
Somma totale di denaro in banca: 9908
Somma totale di denaro in banca: 9978
Procedura completata correttamente
```



Supponiamo di creare un oggetto **Bank** e di creare più thread che eseguono il metodo **transfer()** su questo oggetto, per eseguire dei trasferimenti fra i conti della banca.

I thread possono interferire causando una variazione sul totale delle somme presenti in tutti i conti (che dovrebbe rimanere invariato).

Ad esempio, supponiamo che il conto **i** abbia una somma di 5000, e che vengano eseguiti due thread che trasferiscono in questo conto le somme di 1000 e 2000, rispettivamente.

I due thread dovranno eseguire ciascuno l'istruzione  
**accounts[i] += amount;**



Nella Java Virtual Machine l'istruzione **accounts[i] += amount;** viene realizzata con una sequenza di istruzioni elementari, che trasferiscono il valore di **accounts[i]** in un registro, lo incrementano e lo riportano in **accounts[i]**.

Se un thread viene interrotto dall'altro thread mentre esegue questa sequenza di istruzioni, il risultato può essere sbagliato.

thread1                   registro1 = 5000

thread1                   registro1 = 6000       il thread 1 viene interrotto

thread2                   registro2 = 5000

thread2                   registro2 = 7000

thread2                   accounts[i] = 7000    il thread 1 riprende

thread1                   accounts[i] = 6000



Il corpo del metodo **stampa()**, **incr()** o **transfer()** costituisce una **sezione critica**, che deve essere eseguita da un solo thread per volta senza interruzioni (mutua esclusione).

Come si può realizzare una sezione critica?

Con un **semaforo**: classe **Semaphore**.

Il costruttore **Semaphore(int n)** costruisce un semaforo con **n** permessi. Se  $n = 1$  si ha un semaforo binario.

I metodi **acquire()** e **release()** acquisiscono o rilasciano un permesso. **Quando non ci sono più permessi un thread rimane in attesa nella coda del semaforo.**

Attn: mettere la **release() in blocco finally** per garantire il rilascio del semaforo e non bloccare gli altri thread.



Ad esempio la classe Stampante potrebbe essere modificata così:

```
class Stampante
{
 private Semaphore sem = new Semaphore(1);
 public void stampa(String[] a)
 {
 try
 {
 sem.acquire();
 for(int i = 0; i < a.length; i++)
 {
 Thread.sleep((long)(Math.random() * 100));
 System.out.println(a[i]);
 }
 //sem.release(); // non qui...
 } catch(InterruptedException e) {...}
 finally{sem.release();} // qui per garantire
 // rilascio del semaforo anche in caso di eccezioni
 }
}
```



Dato un oggetto *st* di tipo **Stampante**, il primo thread che esegue il metodo **stampa()** di *st* acquisisce il semaforo e blocca l'accesso a qualunque altro thread che cerchi di eseguire **stampa()** dello stesso oggetto. Gli altri thread che cercano di eseguire **stampa()** vengono messi in attesa.

Quando il primo thread termina di eseguire la sua sezione critica (il metodo **stampa()**) libera il semaforo, che potrà dare accesso ad un altro thread in attesa.

**NOTARE: ogni istanza della classe Stampante ha il proprio semaforo (è una variabile di istanza) →**  
programmi che usano stampanti diverse possono stampare in parallelo.



In Java non è necessario usare esplicitamente un semaforo per realizzare una **sezione critica** (la classe **Semaphore** non esisteva prima della versione 1.5).

Ogni istanza della classe **Object** (e quindi ogni oggetto) possiede un semaforo binario chiamato **lock**.

Se un metodo di qualunque oggetto viene dichiarato **synchronized**, automaticamente viene inserita una **acquire** del lock **dell'oggetto** all'inizio del metodo ed una **release** alla fine.

Non ci sono metodi per eseguire esplicitamente una acquire o una release del lock.



**Lista di Lock:** thread1, thread3, thread2, thread4



La classe **Stampante** può essere definita anche come segue:

```
class Stampante {
 synchronized public void stampa(String[] a) {
 try {
 for(int i = 0; i < a.length; i++) {
 Thread.sleep((long)(Math.random() * 100));
 System.out.println(a[i]);
 }
 } catch(InterruptedException e) {}
 }
}
```

Questa formulazione è equivalente alla precedente che usava il semaforo, perchè il **lock** di un oggetto **Stampante** è usato esattamente come il semaforo binario *sem*.



# Sincronizzazione

Per ogni classe in Java è possibile definire dei metodi **synchronized** (la classe può anche contenere metodi non sincronizzati)

Java associa **un lock ad ogni oggetto** della classe (+ un lock alla classe per sincronizzare i metodi statici).

Quando un thread chiama un metodo **synchronized**, acquisisce il **lock** dell'oggetto. Altri thread che tentino di accedere allo stesso oggetto chiamando metodi sincronizzati rimangono in coda sul lock fino a quando il thread precedente non lo rilascia, terminando l'esecuzione del metodo. A questo punto uno dei thread in coda può passare, bloccando di nuovo il lock.



## Ogni oggetto ha il proprio lock

- **Due oggetti distinti hanno ciascuno il proprio lock.** I thread che eseguono metodi sincronizzati di un oggetto non interferiscono con i thread che eseguono metodi sincronizzati dell'altro.
- **Ogni oggetto ha un solo lock** e la sincronizzazione dei metodi synchronized avviene attraverso questo unico lock. Se un thread sta eseguendo un metodo synchronized, nessun altro thread può eseguire alcun altro metodo synchronized dello stesso oggetto, fino a quando il primo thread non rilascia il lock.



Il lock di un oggetto consente di realizzare *sezioni critiche*.

Se la sezione critica è costituita dal corpo di un metodo, è sufficiente dichiarare il metodo **synchronized**.

Tuttavia la sezione critica potrebbe essere solo una parte del corpo di un metodo. In questo caso si può sincronizzare un blocco:

**synchronized(obj) {... sezione critica ...}**

blocca il *lock* dell'oggetto **obj** per tutta l'esecuzione del blocco di istruzioni.

**Es.:      synchronized(this) {counter++;}**



Si noti che le due formulazioni

```
synchronized public void stampa(String[] a)
{...blocco...}
```

e

```
public void stampa(String[] a) {
 synchronized(this) {...blocco...}}
```

sono equivalenti, perché nella seconda formulazione il blocco viene sincronizzato sul lock di this, ossia sul lock dell'oggetto stesso. Ma la seconda può avere granularità più fine rispetto all'intero corpo del metodo.

**NB: una sezione critica demarca una sequenza di istruzioni da eseguire in modo atomico indipendentemente dal fatto che accedano o meno a variabili condivise.**



# AtomicInteger: contatori con incremento in mutua esclusione

Abbiamo visto che non è possibile considerare una singola istruzione come **counter++** come azione atomica.

Tuttavia Java fornisce dei meccanismi per rendere atomiche le operazioni su particolari variabili.  
Ad esempio, la classe **AtomicInteger** implementa un valore **int** che può essere aggiornato atomicamente.

Il metodo

**int incrementAndGet( )**

incrementa di 1 atomicamente il valore corrente e lo restituisce.



# Sincronizzazione di thread lato client

- L'oggetto non viene protetto da accessi paralleli
- Tutti i client di un oggetto condiviso accedono all'oggetto attraverso blocchi sincronizzati sul lock dell'oggetto stesso
- Limitazioni: se un client non implementa correttamente gli accessi all'oggetto condiviso si ottiene un malfunzionamento
- Ma talvolta è necessario implementare la mutua esclusione in questo modo

# Sincronizzazione di thread lato client – esempio



```
class Stampa extends Thread {
 MiaClasse obj;
 public Stampa(MiaClasse o) {obj = o;}
 public void run() {
 synchronized(obj) {
 for (int i=0; i<7; i++) {
 try {Thread.sleep(200);} catch (InterruptedException e)
 {System.out.println(e.getMessage());}
 System.out.println(Thread.currentThread().getName() + ":" + obj);
 }
 }
 }
}

class MiaClasse {
 public String toString() {
 return "stato dell'oggetto";
 }
}
```

```
C:\WINDOWS\system32\cmd.exe
Thread-0: stato dell'oggetto
Thread-1: stato dell'oggetto
Premere un tasto per continuare . . .
```



# Sincronizzazione di thread lato server

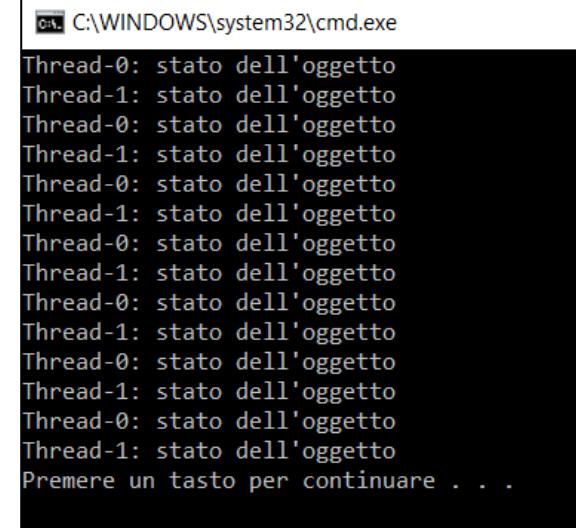
- L'oggetto protegge le variabili condivise e offre metodi synchronized per operare su di esse per cui si auto- protegge dagli accessi esterni.
- I client, invocando metodi synchronized sull'oggetto condiviso, automaticamente si sincronizzano nell'accesso all'oggetto stesso.
- Limitazioni: definire synchronized i metodi dell'oggetto potrebbe non essere sufficiente per sincronizzare le attività dei thread.



# Sincronizzazione di thread lato server – esempio

```
class Stampa extends Thread {
 MiaClasse obj;
 public Stampa(MiaClasse o) {obj = o;}
 public void run() {
 for (int i=0; i<7; i++) {
 try {Thread.sleep(200);} catch (InterruptedException e)
 {System.out.println(e.getMessage());}
 System.out.println(Thread.currentThread().getName() + ": " + obj);
 }
 }
}
class MiaClasse {
 public synchronized String toString() {
 return "stato dell'oggetto";
 }
}
```

Non basta sincronizzare lato client se si vuole  
che le 7 stampe avvengano in sezione critica



```
C:\WINDOWS\system32\cmd.exe
Thread-0: stato dell'oggetto
Thread-1: stato dell'oggetto
Premere un tasto per continuare . . .
```



# Cooperazione fra thread

Spesso un thread non può eseguire un metodo sincronizzato, anche se ha ottenuto il possesso del lock, perché deve aspettare che si verifichi una condizione che non dipende da lui. Per esempio, deve aspettare che la risorsa condivisa entri in un certo stato prima di utilizzarla.

→ per non occupare inutilmente la CPU, il thread deve rilasciare il lock. Per fare questo si può mettere in attesa della condizione, eseguendo il metodo **wait()**, in modo che un altro thread possa entrare e realizzare la condizione.

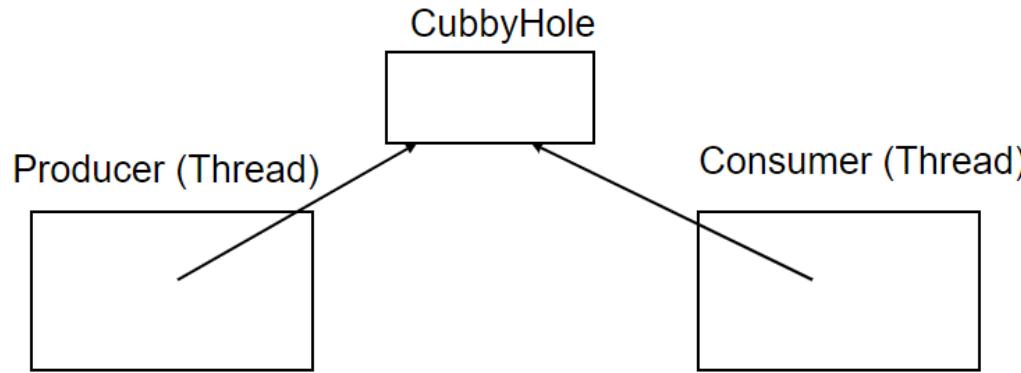
Quando un thread realizza la condizione, avvisa, eseguendo **notify()** o **notifyAll()**, i thread in wait → questi possono riprendere l'esecuzione.



## Es. produttore-consumatore.

Come realizzare un buffer di un elemento (*cubbyhole*)

**ProducerConsumerTest**



Il produttore può inserire un elemento nel buffer quando questo è vuoto. Viceversa il consumatore può togliere un elemento dal buffer quando questo è pieno.



```
public class Producer extends Thread {
 private CubbyHole cubbyhole;
 private int number;

 public Producer(CubbyHole c, int number) {
 cubbyhole = c;
 this.number = number;
 }

 public void run() {
 for (int i = 0; i < 10; i++) {
 cubbyhole.put(i);
 System.out.println("Producer #" + this.number + " put: " + i);
 try {
 sleep((int)(Math.random() * 100));
 } catch (InterruptedException e) {}
 }
 }
}
```



```
public class Consumer extends Thread {
 private CubbyHole cubbyhole;
 private int number;

 public Consumer(CubbyHole c, int number) {
 cubbyhole = c;
 this.number = number;
 }

 public void run() {
 int value = 0;
 for (int i = 0; i < 10; i++) {
 value = cubbyhole.get();
 System.out.println("Consumer #" + this.number + " got: " + value);
 }
 }
}
```



```
public class ProducerConsumerTest {

 public static void main(String[] args) {
 CubbyHole c = new CubbyHole();
 Producer p1 = new Producer(c, 1);
 Consumer c1 = new Consumer(c, 1);

 p1.start();
 c1.start();
 }
}
```



# Implementazione errata di CubbyHole

```
public class CubbyHole {
 private int contents;

 public int get() {
 return contents;
 }

 public void put (int value) {
 contents = value;
 }
}
```

- L'accesso alla risorsa condivisa non avviene in mutua esclusione
- Se il produttore è più lento del consumatore, il consumatore legge più volte lo stesso valore. Se il produttore è più veloce, il consumatore perde dei valori.



I thread **Producer** e **Consumer** si devono sincronizzare.

Dichiarare i metodi **get()** e **put()** **synchronized** è necessario per modificare il buffer in modo atomico. MA non è sufficiente, perché questi metodi possono essere eseguiti solo sotto certe condizioni (buffer pieno o vuoto rispettivamente). Questa condizione può essere descritta dal valore di una variabile booleana **available** (`available==true` significa che il buffer è pieno).

Un thread che vuole eseguire una `get()` la potrà eseguire solo se **available** è true, altrimenti dovrà mettersi in attesa che qualche altro thread esegua una `put()`.

Dopo che la `get()` è stata eseguita, **available** diventerà false. Viceversa per la `put()`.



## Implementazione corretta:

```
public class CubbyHole {
 private int contents;
 private boolean available = false;

 public synchronized int get() {
 while (!available) {
 try {
 wait();
 } catch (InterruptedException e) { ... }
 }
 available = false;
 notifyAll();
 return contents;
 }
 // continua...
```



```
public synchronized void put(int value) {
 while (available) {
 try {
 wait();
 } catch (InterruptedException e) { ... }
 }
 contents = value;
 available = true;
 notifyAll();
}
}

// fine di CubbyHole
```

Perché la condizione di wait viene verificata all'interno di un **while**? Si potrebbe sostituire il **while** con un **if**?



```
public synchronized int get() {
 while (!available) {
 try {
 wait();
 } catch (InterruptedException e) { ... }
 }

}
```

Un thread in wait che viene risvegliato dalla notify non ha garanzia di riprendere subito l'esecuzione. Quando il thread acquisisce il lock, la condizione di wait potrebbe essere nuovamente falsa, e il thread si deve rimettere in wait. È quindi indispensabile verificare la condizione di nuovo prima di continuare.



## **wait(), notify(), notifyAll()**

**wait(), notify() e notifyAll()** sono metodi di **Object** → vengono ereditati da qualunque classe.

Se si tenta di invocarli da un metodo non sincronizzato, si ha un errore a runtime.

Per ogni oggetto ci sono due liste di thread:

- **Lista del lock**: contiene i thread in attesa di acquisire il lock dell'oggetto,
- **Lista di wait**: contiene i thread che sono in wait su quell'oggetto.

Ricordiamo che un thread è blocked se sta eseguendo una sleep() o una operazione di I/O, oppure si trova nella lista del lock o nella lista di wait di qualche oggetto.

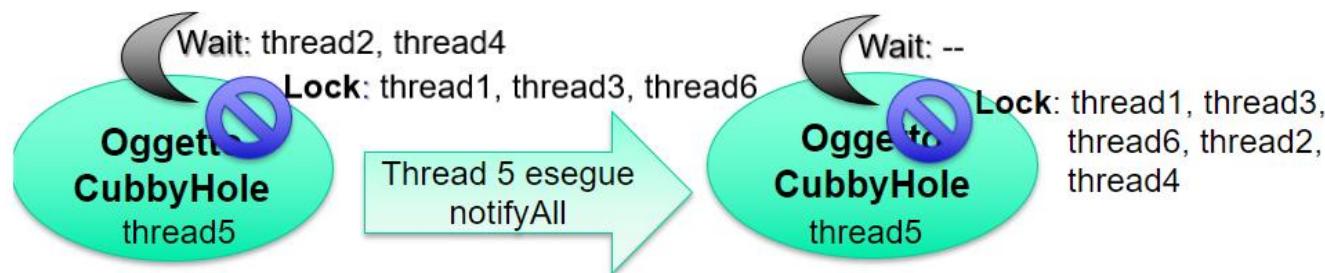


Da un metodo sincronizzato di un oggetto oggi si possono invocare i metodi:

**wait()** - rilascia il *lock* di oggi e mette il thread che lo ha eseguito nella *lista di wait* di oggi;

**notify()** - prende un thread a caso dalla *lista di wait* di oggi e lo sposta nella *lista del lock* di oggi;

**notifyAll()** - prende tutti i thread nella *lista di wait* di oggi e li sposta nella *lista del lock* di oggi.





Il thread che esegue **notify()** o **notifyAll()** **non** rilascia il lock.

I thread risvegliati dalla **notify()** o **notifyAll()** vengono inseriti nella coda del lock dell'oggetto e entrano in competizione per accedere all'oggetto, quando il lock verrà rilasciato.

Quando uno di questi thread acquisisce il lock, riparte dal punto in cui era andato in wait.

Non è garantito che un thread risvegliato dalla **notify()** passi prima di un thread che era già nella coda del lock.

# Wait e notify – schemi di uso



Wait() e notify()/notifyAll() sono metodi di Object → ereditati da tutti gli oggetti. Per sincronizzare correttamente i thread su un oggetto obj condiviso, seguire questi due schemi (suggerimento):

```
synchronized(obj) {
 while (!condition)
 obj.wait();
 ... istruz da eseguire
 quando condition è
 vera
}
```

```
synchronized(obj) {
 ... istruz che
 modificano
 condition
 obj.notifyAll();
}
```



Esempio: come realizzare un semaforo binario.

```
class Semaforo {
 private boolean locked = false;

 public synchronized void p() {
 while (locked)
 try {wait();}
 catch(InterruptedException e){...}
 locked = true;
 }

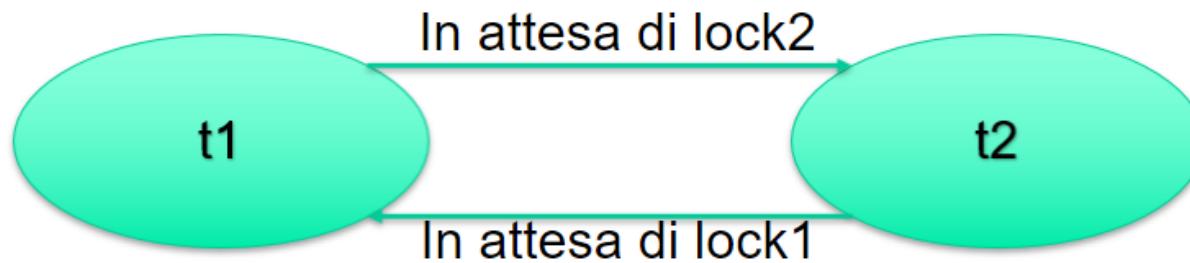
 public synchronized void v() {
 if(locked) notify(); // notifyAll()
 locked = false;
 }
}
```



# Problemi che si possono verificare nella programmazione parallela

**Deadlock:** blocco fatale.

Per es., dati 2 soli thread, t1 e t2, si verifica quando t1 richiede una risorsa (qui, lock) detenuta da t2 che, a sua volta, richiede una risorsa detenuta da t1. Nessuno dei due può procedere con l'esecuzione e restano entrambi in attesa della risorsa.

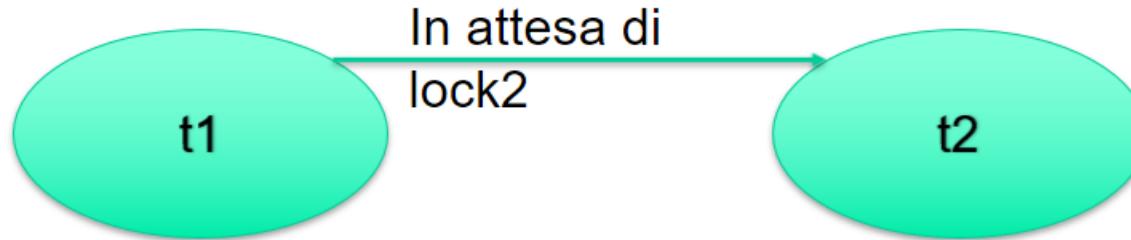




# Problemi che si possono verificare nella programmazione parallela

**Starvation:** blocco di un thread che non riesce a continuare l'esecuzione → non termina.

Si verifica quando un thread richiede una risorsa che non gli viene mai data → non può procedere con l'esecuzione.



Nella programmazione parallela bisogna prevenire questi fenomeni con un'attenta progettazione della sincronizzazione tra thread



# Passi di esecuzione di un programma concorrente (produttori-consumatori)

Un esempio di computazione che si può bloccare in una situazione di **deadlock**.



# Consideriamo il problema dei produttori-consumatori e la sua implementazione vista in precedenza.

Supponiamo che il main crei tre thread **Producer**, p1,p2 e p3, e tre thread **Consumer**, c1,c2 e c3, che fanno tutti riferimento ad un unico oggetto **ch CubbyHole**. Assumiamo anche che ogni thread esegua una sola operazione, **put()** o **get()**, su ch.

Descriviamo i passi di una computazione di questo programma (fra le tante possibili).

Lo stato è costituito da tre elementi:

- lista dei thread in competizione per il lock di ch
- lista dei thread in wait
- stato dell'oggetto ch (valore della variabile booleana *available*)

Ad esempio lo stato iniziale sarà: [ $\langle p1, p2, p3, c1, c2, c3 \rangle$ ,  $\langle \rangle$ , false]



Questo programma può avere un comportamento diverso ad ogni esecuzione, perché non possiamo fare nessuna ipotesi su quale thread, fra quelli in attesa, acquisirà il lock quando questo viene rilasciato, né su quale thread verrà tolto dalla lista di wait se si esegue una notify().

Nel nostro esempio ipotizziamo che le liste siano gestite come code FIFO e che quindi venga sempre selezionato il primo elemento. In questo modo descriviamo una computazione fra tutte quelle possibili.

Lo stato iniziale è:

[<p1,p2,p3,c1,c2,c3>, <>, false]

Il thread p1 acquisisce il lock, esegue il metodo put() e termina rilasciando il lock.

Il nuovo stato sarà:

[<p2,p3,c1,c2,c3>, <>, true]



[<p2,p3,c1,c2,c3>, <>, true]

Il thread p2 acquisisce il *lock*, inizia l'esecuzione della **put()**, verifica che la condizione non è soddisfatta, va in *wait* e rilascia il *lock*:

[<p3,c1,c2,c3>, <p2>, true]

Stessa cosa per p3:

[<c1,c2,c3>, <p2,p3>, true]

Il thread c1 acquisisce il *lock*, esegue il metodo **get()** → modifica la variabile *available*, invoca **notifyAll()** e rilascia il *lock*:

[<c2,c3, p2,p3 >, <>, false]

Il thread c2 acquisisce il *lock* e va subito in *wait*, e analogamente c3:

[<p2,p3 >, <c2,c3>, false]

Il thread p2 esegue la **put** e risveglia c2 e c3



[<p3,c2,c3>, <>, true]

Si procede analogamente ai passi precedenti:

[<c2,c3>, <p3>, true]

[<c3,p3>, <>, false]

[<p3>, <c3>, false]

[<c3>, <>, true]

[<>, <>, false]

Tutti i thread sono terminati e l'esecuzione termina correttamente.



Cosa succede se la **put()** e la **get()** fanno una **notify()** invece della **notifyAll()**? Si procede come prima fino a:

[<c1,c2,c3>, <p2,p3>, true]

c1 esegue la **get()** e alla fine esegue la **notify()** estraendo un solo thread dalla coda di wait:

[<c2,c3,p2>, <p3>, false]

Poi c2 e c3 acquisiscono il lock ma vanno subito in wait:

[<p2>, <p3,c2,c3>, false]

p2 esegue la **put()** e estrae un thread dalla lista di wait:

[<p3>, <c2,c3>, true]

p3 va in wait:

[<>, <c2,c3,p3>, true] e l'esecuzione si blocca (**DEADLOCK**).



## Riprendiamo l'Esempio da JAVA2 (secondo volume):

Abbiamo una classe **Bank** con un metodo **transfer** per trasferire denaro da un conto ad un altro.

```
class Bank {
 private int[] accounts;

 public Bank(int n, int initialBalance) {
 accounts = new int[n];
 int i;
 for (i = 0; i < accounts.length; i++)
 accounts[i] = initialBalance;
 }

 public void transfer(int from, int to, int amount) {
 if (accounts[from] < amount) return;
 accounts[from] -= amount;
 accounts[to] += amount;
 }
}
```



Creiamo un oggetto **Bank** con 10 conti inizializzati con 1000 euro ciascuno:

**Bank bank = new Bank(10, 1000);**

Creiamo più thread che eseguono il metodo **transfer()** su **bank**, per eseguire dei trasferimenti fra i conti della banca.

I thread possono interferire causando una variazione sul totale delle somme presenti in tutti i conti (il totale dovrebbe rimanere invariato).

```
C:\WINDOWS\system32\cmd.exe
trasferimento da 7 a 8 di 3
trasferimento da 8 a 3 di 73
trasferimento da 8 a 1 di 84
trasferimento da 4 a 0 di 61
trasferimento da 2 a 4 di 55
trasferimento da 1 a 8 di 61
trasferimento da 2 a 8 di 34
trasferimento da 5 a 7 di 66
trasferimento da 0 a 8 di 13
trasferimento da 8 a 0 di 96
Somma totale di denaro in banca: 10073
Somma totale di denaro in banca: 10052
Somma totale di denaro in banca: 10107
Somma totale di denaro in banca: 10107
Somma totale di denaro in banca: 10107
Somma totale di denaro in banca: 9957
Somma totale di denaro in banca: 9800
Press any key to continue . . .
```



# Sincronizzazione

Per risolvere il problema è sufficiente dichiarare il metodo **transfer()** **synchronized**. In questo modo solo un thread per volta lo potrà eseguire.

Quando un thread riesce ad acquisire il *lock* della banca e ad eseguire il metodo **transfer()**, potrà procedere fino al termine del metodo senza che nessun altro thread possa interferire.



# Cooperazione fra thread

Il metodo **transfer()** non esegue il trasferimento se la somma presente sul conto di partenza è più bassa di quella che si vuole trasferire.

Modifichiamo transfer() in modo che, nel caso di somma insufficiente, il thread che lo esegue rimanga in attesa che la somma sul conto sia sufficiente per eseguire il trasferimento.

Il metodo può essere modificato come segue:



```
public synchronized void transfer(int from, int to, int amount) {
 try {
 while (accounts[from] < amount)
 wait();
 accounts[from] -= amount;
 accounts[to] += amount;
 ntransacts++;
 notifyAll();
 }
 catch(InterruptedException e) {...}
}
```



## ATTENZIONE ALLA GRANULARITA' DELLA SINCRONIZZAZIONE

Se la sincronizzazione viene fatta sul lock della banca, si può eseguire solo un trasferimento alla volta.

Vediamo un altro esempio in cui vengono introdotte anche le classi **BankAccount**.

Nella prima versione (**BankTest**) non c'è sincronizzazione.

La classe **BankAccount** contiene i metodi **deposit()** e **withdraw()**. Il metodo **withdraw()** è booleano e restituisce *false* quando il saldo del conto non è sufficiente per fare il prelievo.

Per potere osservare gli effetti di una esecuzione concorrente di questi metodi, è stata introdotta una variabile *temp* ed è stata inserita una **sleep()** nel metodo **withdraw()**.



```
class BankAccount {
 private int balance = 0;

 public synchronized void deposit(int amount) {
 int temp = balance;
 temp = temp + amount;
 balance = temp;
 notifyAll();
 }

 // se la soglia di prelevamento è troppo alta withdraw si blocca
 public synchronized void withdraw(int amount) {
 while (amount > balance)
 try { wait(); }
 catch (InterruptedException e){}
 int temp = balance;
 try { Thread.sleep(1); }
 catch(InterruptedException e) {}
 temp = temp - amount;
 balance = temp;
 }

 public synchronized int getBalance() {
 return balance;
 }
}
```



**BankTestSynchr**: dichiaro i due metodi **deposit()** e **withdraw()** di **BankAccount** come **synchronized**. Inoltre il metodo **withdraw()** si mette in **wait** se il suo saldo non è sufficiente per il prelievo.

In questo caso la sincronizzazione è fatta sul lock di un **BankAccount**, quindi sul singolo conto corrente e non sulla banca intera

- Due thread si devono sincronizzare solo se eseguono ambedue una operazione sullo stesso conto.
- Non c'è bisogno di sincronizzare il metodo **transfer()** di **Bank** perché sia il metodo di prelevamento sul singolo conto che quello di versamento sono sincronizzati.



Eseguendo questo programma si nota che spesso il programma non termina. Questo succede quando un thread rimane bloccato in wait nella **withdraw()** di un conto, in attesa che ci sia una cifra sufficiente, e nessun altro thread esegue un versamento sullo stesso conto.

Questo comportamento è più probabile più è alta la soglia sulle cifre da trasferire.

TextPad - C:\Users\liliana\Dropbox\ DIDATTICA\ PROGR III-2021\ LUCIDI\ 0000-ESEMPI\ 8-THREAD\ 17-bancaWait\ BankTestSynchr.java

File Edit Search View Tools Macros Configure Window Help

Find incrementally Match case

BankTestSynchr.java

```
class BankTestSynchr {
 public static void main(String[] args) {
 Bank b = new Bank(10, 1000);
 b.printTotal();
 System.out.println("%%%%%%%%%%%%%\n\n");
 int num = 10;
 TransferThread[] tt = new TransferThread[num];
 for (int i=0; i < num; i++) {
 // se metto una soglia di prelevamento altissima rischio la non terminazione
 // del programma perché le operazioni di prelevamento si bloccano
 //tt[i] = new TransferThread(b,80000);
 tt[i] = new TransferThread(b, 80);
 tt[i].start();
 }
 // attendo fine di tutti i trasferimenti per stampare totale in banca
 for (int i=0; i < num; i++)
 try {
 tt[i].join();
 }
 catch (InterruptedException e) {System.out.println(e.getMessage());}
 System.out.println("\n\n%%%%%%%%%%%%%");
 System.out.println("%%%%%%%%%%%%%");
 b.printTotal();
 }
}

class BankAccount {
 private int balance = 0;

 public synchronized void deposit(int amount) {
 int temp = balance;
 temp = temp + amount;
 balance = temp;
 }
}
```

# Pipe - Comunicazione fra thread attraverso stream di I/O - I



Per far comunicare thread all'interno di una singola Java Virtual Machine si possono usare pipe – tubi. Le pipe sono canali di comunicazione su cui si possono scrivere e leggere dati.

È comodo usare le pipe quando si deve gestire la sincronizzazione tra thread produttore e thread consumatore su uno stream di byte:

- Il thread produttore produce dati nello stream, e se produce troppo velocemente rispetto al consumatore, si blocca;
- Il thread consumatore legge dallo stream man mano che riceve dati, bloccandosi quando non ci sono dati.

Con le pipe, non è necessario sincronizzare esplicitamente produttore e consumatore perché la sincronizzazione avviene in automatico man mano che scrivono e leggono sul/dal canale di comunicazione.

# Comunicazione fra thread attraverso stream di I/O - II



Esempio: con uso di PipedWriter e PipedReader in quanto il produttore genera caratteri UNICODE.

Per scrivere in uno stream di tipo pipe, usare un PipeWriter. Per leggere da pipe, usare un PipeReader impostato sullo stream del writer:

```
class PipeTest {
 public static void main(String[] args) {
 PipedWriter out = new PipedWriter();
 PipedReader in = null;
 try { in = new PipedReader(out); } catch(IOException e)
 {System.out.println(e.getMessage());}
 new Sender(out).start();
 new Receiver(in).start();
 }
}
```

Il **Sender** e **Receiver** corrispondono ad un produttore e ad un consumatore rispettivamente. La pipe realizza un buffer.



Piu visi

File Edit Search View Tools Macros Configure Window Help



00:00:00

PipeTest.java

```
 sleep(1000);
 } catch (IOException e) {}
 catch(InterruptedException e) {System.out.println(e.getMessage());}
}

class Receiver extends Thread {
 private Reader in;

 public Receiver (Reader r)
 {in = r; }

 public void run()
 {
 try {
 while (true) {
 sleep(1000);
 System.out.println("leggi " + (char)in.read());
 }
 } catch(IOException e) {System.out.println("fine input");
 //System.out.println(e.getMessage());
 }
 catch(InterruptedException e) {System.out.println(e.getMessage());}
 }
}

class PipeTest {
 public static void main(String[] args) {
 PipedWriter out = new PipedWriter();
 PipedReader in = null;
 try {
 in = new PipedReader(out);
 } catch(IOException e) {System.out.println(e.getMessage());}
 new Sender(out).start();
 new Receiver(in).start();
 }
}
```



# Comunicazione fra thread attraverso stream di I/O - III

Vedere l'esempio **PipeTestBynaryData** per comunicazione tra 3 thread in pipeline, scambiando dati binari. Qui il produttore:

```
class Producer extends Thread {
 private DataOutputStream out;
 private Random rand = new Random();
 public Producer(OutputStream os) {out = new DataOutputStream(os);}
 // os viene Instanziato con un PipedOutputStream
 public void run() {
 while (true) {
 try {
 double num = rand.nextDouble();
 out.writeDouble(num);
 out.flush();
 sleep(Math.abs(rand.nextInt() % 1000));
 } catch(Exception e) {System.out.println("Error: " + e);}
 }} // end Producer
```



# Ringraziamenti

Grazie al Prof. Emerito Alberto Martelli del  
Dipartimento di Informatica dell'Università  
di Torino per aver redatto la prima  
versione di queste slides.



# Programmazione III

Prof.ssa Liliana Ardissono  
Dipartimento di Informatica  
Università di Torino

**Programmazione parallela con i Java Thread –  
classi per la gestione della concorrenza nel  
package `java.util.concurrent`**



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND



# Classi per la programmazione concorrente, disponibili nel package `java.util.concurrent`.

Abbiamo già visto la classe **Semaphore**.

Il costruttore **Semaphore(int n)** costruisce un semaforo con **n** permessi. Se  $n = 1$  si ha un semaforo binario.

I metodi **acquire()** e **release()** acquisiscono o rilasciano un permesso.

Quando non ci sono più permessi un thread rimane in attesa al semaforo.

# Interface BlockingQueue



L'interfaccia **BlockingQueue** definisce una coda sincronizzata:

- Quando la coda è piena, il thread che cerca di scrivere nella coda con il metodo **put()** viene messo in attesa.
- Quando la coda è vuota, il thread che cerca di leggere dalla coda con il metodo **take()** viene messo in attesa.

Con questa classe è molto semplice realizzare uno schema produttore-consumatore.

java.util.concurrent

## Interface BlockingQueue<E>

### Type Parameters:

E - the type of elements held in this collection

### All Superinterfaces:

[Collection<E>](#), [Iterable<E>](#), [Queue<E>](#)

### All Known Subinterfaces:

[BlockingDeque<E>](#), [TransferQueue<E>](#)

### All Known Implementing Classes:

[ArrayBlockingQueue](#), [DelayQueue](#), [LinkedBlockingDeque](#), [LinkedBlockingQueue](#), [LinkedTransferQueue](#), [PriorityBlockingQueue](#), [SynchronousQueue](#)



**BlockingQueue** è un'interfaccia: ci sono diverse classi che la implementano. Ad esempio:

```
BlockingQueue<Integer> c =
 new ArrayBlockingQueue<Integer>(5);
```

Integer: tipo degli elementi della coda  
5: capacità della coda



```
class Producer extends Thread {
 private BlockingQueue<Integer> bq;
 private int num;

 public Producer(BlockingQueue<Integer> c, int num) {
 bq = c;
 this.num = num;
 }

 public void run() {
 for (int i = 0; i < 10; i++) {
 try {
 bq.put(num*10+i);
 System.out.println("Producer #" + this.num
 + " put: " + (num*10+i));
 sleep((int)(Math.random() * 1000));
 } catch (InterruptedException e) {}
 }
 }
}
```

# Interface Lock e Condition



In Java ogni oggetto ha un lock implicito. È anche possibile usare dei lock esplicativi mediante l'interfaccia **Lock**. **Lock** fornisce una funzionalità analoga con il metodo **newCondition()** che restituisce un oggetto di tipo **Condition**.

L'interfaccia **Condition** fornisce i metodi:

|                    |                                    |
|--------------------|------------------------------------|
| <b>await()</b>     | analogo alla wait() di Object      |
| <b>signal()</b>    | analogo alla notify() di Object    |
| <b>signalAll()</b> | analogo alla notifyAll() di Object |

`java.util.concurrent.locks`

## Interface Lock

### All Known Implementing Classes:

`ReentrantLock`, `ReentrantReadWriteLock$ReadLock`, `ReentrantReadWriteLock$WriteLock`



```
Lock lock = new ReentrantLock();
Condition notFull = lock.newCondition();
Condition notEmpty = lock.newCondition();
.....
lock.lock(); // ottiene il lock
.....
notFull.await(); // il thread viene messo in attesa
 // sulla condizione notFull
.....
notEmpty.signal(); // viene risvegliato un thread
 // in attesa sulla condizione notEmpty
.....
lock.unlock(); // rilascia il lock
```

## ReentrantLock: implementazione dell'interface Lock



**Lock** può essere usato per realizzare un **bounded buffer**, con due metodi **put()** e **take()**.

I due metodi si sincronizzano su un lock esplicito, con due condizioni **notFull** e **notEmpty**.

Quando il buffer è pieno, un thread che cerca di scrivere nel buffer viene messo in attesa sulla condizione **notFull**.

Quando il buffer è vuoto, un thread che cerca di leggere dal buffer viene messo in attesa sulla condizione **notEmpty**.

Quando un thread finisce di scrivere fa una **signal()** su **notEmpty**.

Quando un thread finisce di leggere fa una **signal()** su **notFull**.

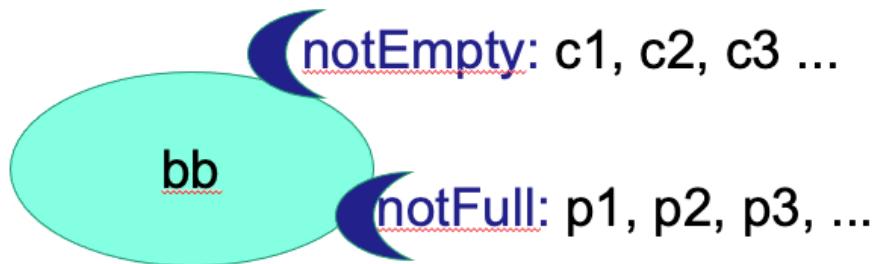
Es. produttore-consumatore:



```
class BoundedBuffer {
 final Lock lock = new ReentrantLock();
 final Condition notFull = lock.newCondition();
 final Condition notEmpty = lock.newCondition();

 final int[] items = new int[10];
 int putptr, takeptr, count;

 public void put(int x) throws InterruptedException {
 lock.lock();
 try {
 while (count == items.length)
 notFull.await();
 items[putptr] = x;
 if (++putptr == items.length) putptr = 0;
 ++count;
 notEmpty.signal();
 } finally {
 lock.unlock();
 }
 }
}
```





```
public int take() throws InterruptedException {
 lock.lock();
 try {
 while (count == 0)
 notEmpty.await();
 int x = items[takeptr];
 if (++takeptr == items.length) takeptr = 0;
 --count;
 notFull.signal();
 return x;
 } finally {
 lock.unlock();
 }
}
```



Nell'esempio precedente non è necessario usare la **signalAll()** anche se ci sono molti produttori e molti consumatori.

Infatti i thread in wait su **notFull** sono tutti produttori e non c'è differenza fra svegliarne uno o un altro.

Analogamente i thread in wait su **notEmpty** sono tutti consumatori.

Viceversa nella implementazione con la classe **CubbyHole** c'è una sola coda di wait in cui vengono inseriti sia produttori che consumatori. Facendo la **notify()** invece della **notifyAll()** si potrebbe svegliare un thread che non è in grado di proseguire, causando una situazione di deadlock.



# Lettori e scrittori

Interface **ReadWriteLock** specifica due metodi:

**Lock readLock()**

**Lock writeLock()**

Il primo metodo restituisce un lock che può essere tenuto simultaneamente da più lettori, purché non ci siano scrittori.

Il secondo metodo restituisce un lock esclusivo.

Con questi due lock è facile definire una struttura dati con un metodo `read()` che usa il primo lock e un metodo `write()` che usa il secondo.

# Esempio: ReadersWritersLock - I



```
class Database {
 private ReadWriteLock rwl = new ReentrantReadWriteLock();
 private Lock rl = rwl.readLock();
 private Lock wl = rwl.writeLock();

 public void read(int i) { // i rappresenta l'ID del thread reader
 rl.lock();

 rl.unlock();
 }
 public void write(int i) { // i rappresenta l'ID del thread writer
 wl.lock();

 wl.unlock();
 }
}
```

Questo garantisce la mutua esclusione ma un lettore potrebbe leggere prima che qualche scrittore scriva → vd esempio

File Edit Search View Tools Macros Configure Window Help

Find incrementally Match case

ReadersWritersLock.java

```
import java.util.concurrent.locks.*;

// Implementa i lettori-scrittori usando il
// ReadWriteLock

class ReadersWritersLock {
 public static void main(String[] args) {
 int numReaders = 3;
 int numWriters = 2;

 Database db = new Database();

 for (int i = 0; i < numReaders; i++)
 new Reader(i, db);
 try {
 Thread.sleep(300);
 } catch(InterruptedException e) {}
 for (int i = 0; i < numWriters; i++)
 new Writer(numReaders + i, db);
 }
}

/* NB: SE SI USANO I READER/WRITER LOCKS POSSONO LEGGERE
IN PARALLELO PIU' READERS, SE UN WRITER SCRIVE E' IN MUTUA
ESCLUSIONE TOTALE, MA UN LETTORE PUO' INIZIARE A LEGGERE
PRIMA CHE ALCUNO SCRITTORE SCRIVA. DIRE CHE I LETTORI ACCEDONO
IN MUTUA ESCLUSIONE RISPETTO AGLI SCRITTORI, E CHE OGNI SCRITTORE
OPERA IN MUTUA ESCLUSIONE TOTALE NON BASTA...
*/
```

```
class Database {
```

Tool Output

Explorer Document Selector Search Results Tool Output

# Esempio: ReadersWritersLock - II



Come si resolve il problema della lettura di dati inesistenti?

Usando le Condition per mettere in attesa i reader che tentano di leggere quando non ci sono dati disponibili, come fatto negli esempi precedenti (notFull, notEmpty).



# Ringraziamenti

Grazie al Prof. Emerito Alberto Martelli del  
Dipartimento di Informatica dell'Università  
di Torino per aver redatto la prima  
versione di queste slides.



# Programmazione III

Prof.ssa Liliana Ardissono  
Dipartimento di Informatica  
Università di Torino

**Programmazione parallela con i Java Thread –  
Thread e Task – Pool di Thread, Future e  
Callable**



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND



# Thread e Task - I

**Task (Runnable):** unità logica di lavoro. Si tratta della sequenza di istruzioni che va eseguita, possibilmente in parallelo rispetto a altri task.

**Thread:** è il meccanismo di esecuzione di un task.

```
Class MyTask implements Runnable {
```

```
 ...
 public void run() {...}
}
```

```
// Creo un nuovo Thread e gli passo il task da eseguire
Thread t = new Thread(new MyTask());
```



# Thread e Task - II

Sebbene i thread siano processi "light", occupano una discreta quantità di RAM e la loro creazione e distruzione genera lavoro (in termini di interazione con il sistema operativo) che, in presenza di molti thread, aumenta l'overhead nella JVM →

- La creazione di thread nuovi può andare bene per piccoli programmi, ma non è scalabile
- Limitare la creazione di nuovi thread e usare i Runnable per definire i task, da far eseguire ai thread in modo controllato



# Thread e Task - III

Per organizzare un'applicazione in modo modulare e facilitare il parallelismo conviene

- suddividere la sua logica applicativa in Task - i compiti da fare (in modo indipendente) e
- specificare nella gestione dei Thread le politiche di esecuzione dei task (permettendo parallelismo, o imponendo sequenzialità, a seconda delle relazioni di dipendenza esistenti tra i task).



# Thread pool - I

Un programma che crea numerosi thread, ciascuno dei quali fa attività brevi, è inefficiente

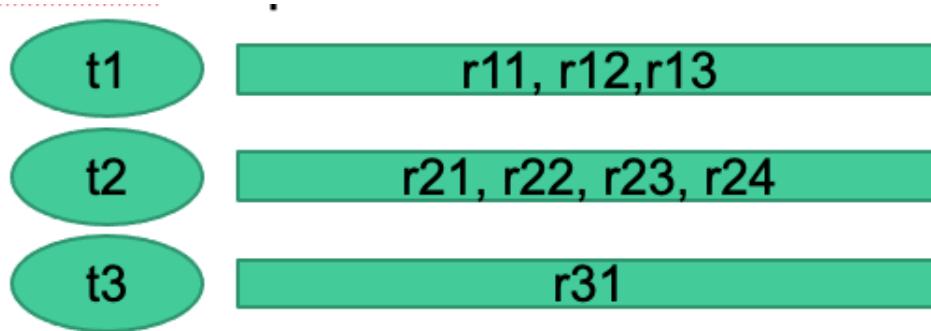
- **Thread pool:** il programma può creare un certo numero di thread in un colpo solo e poi distribuire su questi i task da eseguire
- La creazione di pool di thread e la loro esecuzione viene gestita dagli **Executors**, a cui si chiede di creare i pool e di mandarli in esecuzione



# Thread pool - II

Insieme di thread omogenei e pronti all'esecuzione dei task che vengono loro assegnati.

Ogni thread del pool ha una coda di task da eseguire:



Quando un thread ha finito un task prende un task dalla sua coda. Se la coda è vuota, il thread ruba il task ad un altro thread. Se non ci sono task da fare il thread si sospende in attesa di nuovi task o di terminazione.



# Thread pool: Executors - I

Package `java.util.concurrent`:

**Executor interface**: specifica la policy di esecuzione dei task di un'applicazione

- **void execute(Runnable command)**: esegue command ad un certo punto, nel futuro, secondo la policy specificata (ci sono tipi diversi di executor, vd. prossimo lucido)



# Thread pool: Executors - II

**Executors:** factory degli oggetti di tipo Executor:

- **ExecutorService newFixedThreadPool(int n): crea un pool di thread con un numero fisso di elementi**
- **ScheduledExecutorService**  
**newScheduledThreadPool(int n):** crea un pool thread con scheduling delle attività (serve per creare pool di thread che eseguiranno i task con un ritardo specificato in input, oppure ciclicamente)
- **ExecutorService newSingleThreadExecutor()** crea un Executor che usa un solo thread e usa la ThreadFactory per crearne di nuovi
- ...

# Thread pool fisso – Esempio - I



```
class PingPong implements Runnable {
 private int delay; private String name;

 public PingPong(String str, int d) {
 delay = d; name = str; }

 public void run() {
 for (int i = 1; i < 3; i++) {
 System.out.println(name + " " +
 Thread.currentThread().getName());}
 try {
 Thread.sleep(delay);
 } catch (InterruptedException e) {return;}
 }
 System.out.println("DONE! "+name+ "
 "+Thread.currentThread().getName());
}
```

# Thread pool fisso – Esempio - II

```
public class PingPongThreadPool {
 private static final int NUM_THREAD = 5;
```



```
public static void main (String[] args) {
 ExecutorService exec = Executors.newFixedThreadPool(NUM_THREAD);
 for (int i=0; i<10; i++) { // creo 10 task da eseguire
 Runnable task = new PingPong("PING"+i, 200);
 exec.execute(task); }
 exec.shutdown(); // initiates an orderly shutdown in which previously
 // submitted tasks are executed, but no new tasks will be accepted
 try {
 exec.awaitTermination(5, TimeUnit.SECONDS);
 //Blocks until all tasks have completed execution after a shutdown
 // request,
 // or the timeout occurs, or the current thread is interrupted,
 // whichever happens first (lo uso se voglio terminare entro una
scadenza)
 } catch (InterruptedException e) {
 System.out.println(e.getMessage());
 }
}
```



# Thread pool fisso – Esecuzione

i 5 thread del pool eseguono  
2 task per ciascuno

```
C:\WINDOWS\system32\cmd... - □ ×
PING1 pool-1-thread-2
PING2 pool-1-thread-3
PING4 pool-1-thread-5
PING3 pool-1-thread-4
PING0 pool-1-thread-1
PING2 pool-1-thread-3
PING4 pool-1-thread-5
PING3 pool-1-thread-4
PING0 pool-1-thread-1
PING1 pool-1-thread-2
DONE! PING3 pool-1-thread-4
DONE! PING1 pool-1-thread-2
DONE! PING0 pool-1-thread-1
DONE! PING2 pool-1-thread-3
DONE! PING4 pool-1-thread-5
PING8 pool-1-thread-3
PING7 pool-1-thread-1
PING6 pool-1-thread-2
PING5 pool-1-thread-4
PING9 pool-1-thread-5
PING6 pool-1-thread-2
PING7 pool-1-thread-1
PING5 pool-1-thread-4
PING8 pool-1-thread-3
PING9 pool-1-thread-5
DONE! PING6 pool-1-thread-2
DONE! PING7 pool-1-thread-1
DONE! PING8 pool-1-thread-3
DONE! PING9 pool-1-thread-5
DONE! PING5 pool-1-thread-4
Premere un tasto per continuare . . .
```

# Thread pool fissi e "graceful degrading" delle applicazioni



I thread pool fissi sono essenziali per permettere alle applicazioni di degradare le proprie prestazioni, quando sotto carico eccessivo, senza smettere di funzionare.

Es. web server: ogni richiesta HTTP viene servita da un thread. Se il web server creasse un thread per ogni richiesta, in presenza di molte richieste ne genererebbe troppi → smetterebbe di funzionare. Invece con un pool fisso di thread servirà le richieste in coda man mano che termina l'esecuzione delle precedenti → ritardo, ma senza crash dell'applicazione



# Interface Callable

- I task che devono restituire un risultato al chiamante possono essere implementati come Callable
- **Callable<T>:**
  - Offre metodo **T call()** (analogo a run()) per l'esecuzione del task
  - La sua esecuzione restituisce un valore di tipo T



# Interface Future<T>

Rappresenta il risultato di una computazione asincrona. Offre i metodi per verificare se la computazione è completata e per estrarre il risultato. Il tipo T è quello del risultato della computazione

- **FutureTask(Callable<T> task)**: crea un FutureTask che, quando parte, esegue il task passato come parametro.
- **boolean isDone()**: per verificare se il task è completato.
- **T get()**: se necessario attende che la computazione asincrona termini; poi restituisce il suo risultato.
  - NB: se il task non termina, chi invoca la get() resta bloccato in attesa – ma esistono metodi con time-out, in caso si possa ignorare qualche risultato



# Future-Callable – Esempio - I

```
class Computazione implements Callable<Integer> {
 private int num;
 // Computazione è un task che restituisce un risultato
 public Computazione(int n) {num = n;}

 public Integer call() {
 System.out.println("INIZIO computazione " + num);
 try {
 Thread.sleep((int) Math.round(Math.random() * 4000));
 } catch (InterruptedException e)
 {System.out.println("Interruzione thread");}
 System.out.println("FINE computazione " + num +
 ": risultato = " + 2*num);
 return new Integer(2*num);
 }
}
```



# Future-Callable – Esempio - II

```
public static void main (String[] args) {
 // estratto di codice da esempio FutureCallable
 // preparo il pool di esecutori dei task callable
ExecutorService exec =
 Executors.newFixedThreadPool(NUM_THREADS);
... // creo il task callable
FutureTask<Integer> ft = new FutureTask<>(new Computazione(i));
exec.execute(ft);

...
try {
 int result = (ft.get()).intValue();
} catch (Exception e) {System.out.println(e.getMessage());}
}
```

# Future-Callable – Esempio - esecuzione



```
inizio il main
INIZIO computazione 0
INIZIO computazione 1
FINE computazione 1: risultato = 2
INIZIO computazione 2
FINE computazione 0: risultato = 0
INIZIO computazione 3
FINE computazione 3: risultato = 6
INIZIO computazione 4
FINE computazione 4: risultato = 8
INIZIO computazione 5
FINE computazione 2: risultato = 4
INIZIO computazione 6
FINE computazione 5: risultato = 10
INIZIO computazione 7
FINE computazione 7: risultato = 14
INIZIO computazione 8
FINE computazione 6: risultato = 12
INIZIO computazione 9
FINE computazione 9: risultato = 18
FINE computazione 8: risultato = 16

Il totale è: 90
```





# Thread pool: Executors - III

**Executors:** factory degli oggetti di tipo Executor:

- **ExecutorService newFixedThreadPool(int n):** crea un pool di thread con un numero fisso di elementi
- **ScheduledExecutorService**  
**newScheduledThreadPool(int n): crea un pool thread con scheduling delle attività** (serve per creare pool di thread che eseguiranno i task con un ritardo specificato in input, oppure ciclicamente)
- **ExecutorService newSingleThreadExecutor()** crea un Executor che usa un solo thread e usa la ThreadFactory per crearne di nuovi
- ...



# ScheduledExecutorService - I

**schedule (Runnable task, long delay, TimeUnit timeunit)**

```
public static void main (String[] args) {
 ScheduledExecutorService exec =
 Executors.newScheduledThreadPool(3); // pool di 3 thread

 ScheduledFuture<Integer> task =
 exec.schedule(new Computazione1(2),
 5, // tempo di attesa
 TimeUnit.SECONDS);

 try {
 System.out.println("result = " + task.get());
 } catch (Exception e){System.out.println(e.getMessage());}
 exec.shutdown();
}
// TimeUnit temporizza esecuzione di task con ritardo
```



# ScheduledExecutorService - II

Esecuzione ciclica di task con ritardo iniziale (initDelay)

**scheduleAtFixedRate (Runnable, long initDelay, long period, TimeUnit timeunit)**

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

```
 ScheduledExecutorService exec =
```

```
 Executors.newScheduledThreadPool(3);
```

```
 exec.scheduleAtFixedRate (new MyTask("TASK"),
```

```
 2,
```

```
 3,
```

```
 TimeUnit.SECONDS);
```

```
// esecuzione ciclica infinita. Se voglio fermarla devo inserire lo
// shutdown ma dopo un certo tempo se no il thread termina subito
// e non esegue mai il task. Vd. ScheduledExecutorTest
```

```
}
```

# ScheduledExecutorTest – Esecuzione



```
C:\WINDOWS\system32\cmd.exe
INIZIO computazione
FINE computazione
Risultato = 4

Ora lancio un task ciclico ma lo lascio andare avanti solo per pochi secondi
TASK pool-1-thread-1
DONE! TASK pool-1-thread-1
Premere un tasto per continuare . . .
```

The screenshot shows a Java IDE interface with the following details:

- Title Bar:** TextPad - C:\Users\liliana\Dropbox\ DIDATTICA\PROGR III-2021\ LUCIDI\0000-ESEMPI\8-THREAD\25
- Toolbar:** Includes icons for file operations, search, and various tools.
- Menubar:** File, Edit, Search, View, Tools, Macros, Configure, Window, Help.
- Toolbars:** Includes a "Video Tools" toolbar with icons for Pausa (Pause), 00:00:00, Selezione area (Selection area), Audio, and Registrare puntatore (Record pointer).
- Code Editor:** The main window displays the following Java code:

```
public class CallableTest {
 private static int N = 10;
 private static int NUM_THREADS = 2;

 public static void main(String[] args) {

 System.out.println("inizio il main");
 int total = 0;
 ExecutorService exec = Executors.newFixedThreadPool(NUM_THREADS);
 //ExecutorService exec = Executors.newSingleThreadExecutor();
 // creo i task callable
 Vector<FutureTask<Integer>> tasks = new Vector<>();

 for (int i = 0; i < N; i++) {
 FutureTask<Integer> ft = new FutureTask<>(new Computazione(i));
 tasks.add(ft);
 exec.execute(ft);
 }
 try {
 for (int i = 0; i < tasks.size(); i++) {
 FutureTask<Integer> f = (FutureTask<Integer>) tasks.get(i);
 total = total + (f.get()).intValue();
 }
 } catch (Exception e) {
 System.out.println(e.getMessage());
 }
 System.out.println("\nIl totale è: " + total);
 exec.shutdown();
 }
}
```

- Tool Output:** A panel at the bottom showing the output of the execution.
- Bottom Bar:** Includes tabs for Explorer, Document Selector, Search Results, Tool Output, and navigation buttons (9, 17, Read, Ovr, Block, Sync, Rec, Caps).



# Conclusioni

**I pool di thread non aggiungono nulla ai concetti che ho descritto (parallelismo etc.), ma servono per sviluppare applicazioni scalabili.**

Anche nel caso di pool di thread potrebbe essere necessario gestire accessi in mutua esclusione a risorse condivise → si usano i lock, Semaphore, synchronized, etc., per gestire le sezioni critiche, wait e notify per attendere condizioni, etc.



# Networking: Socket

---

Programmazione III

Matteo Baldoni e Alberto Martelli  
con integrazioni di Liliana Ardissono



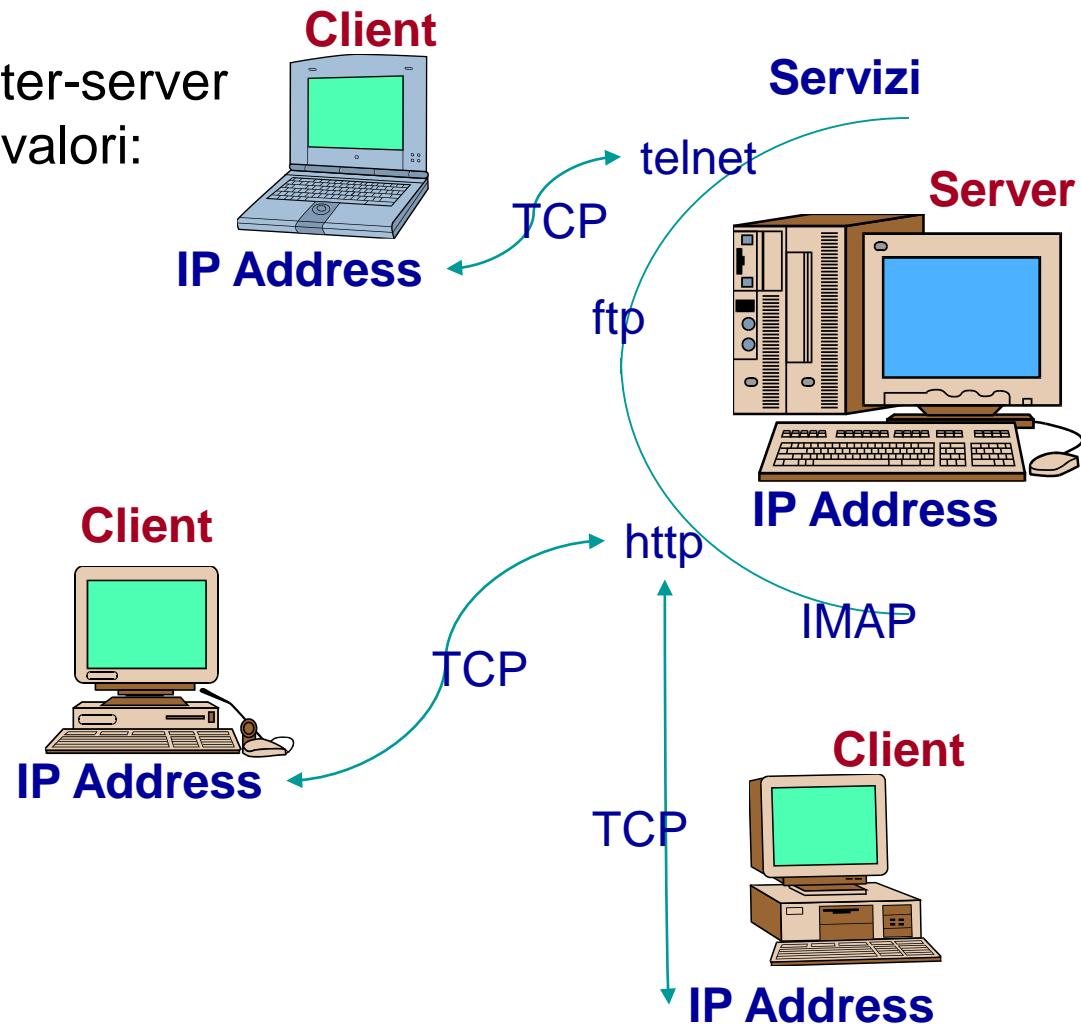
# Architettura Client/Server

Un servizio presso un computer-server è identificato dai seguenti valori:

- **IP** (32 bit o 128 bit)
- **Port** (16 bit)

Alcuni tipici servizi:

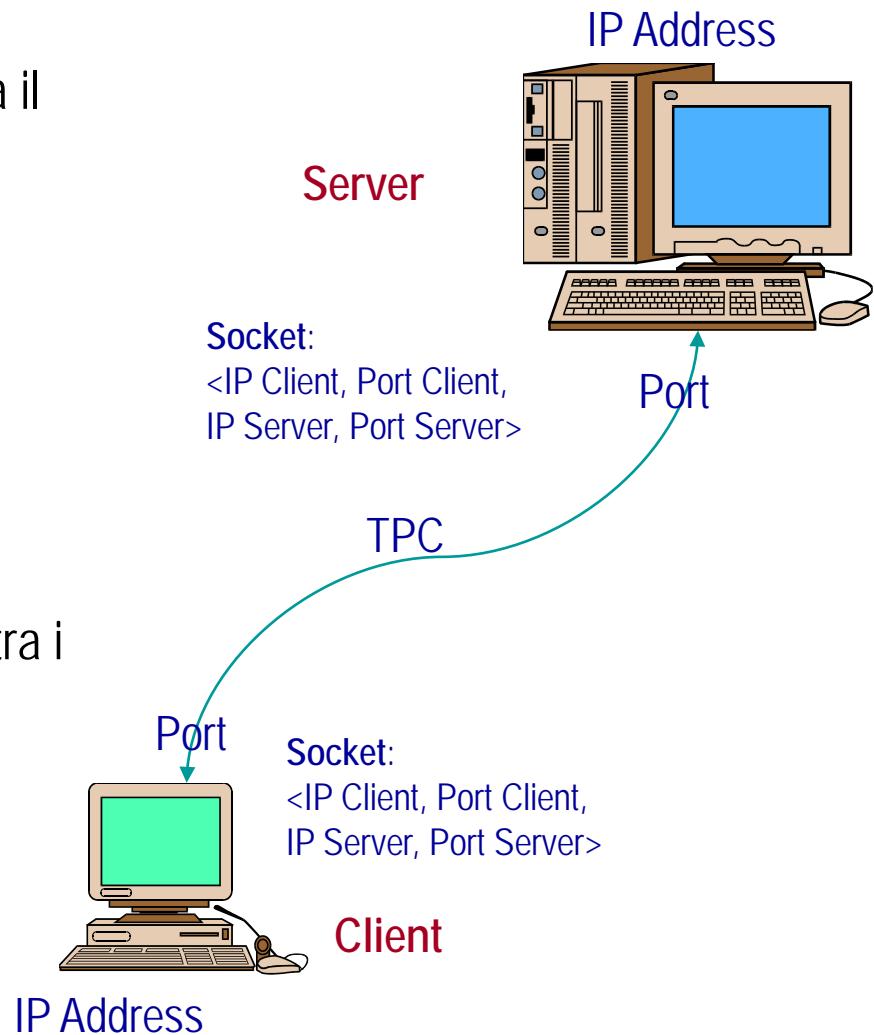
- telnet
- http
- ftp
- SMTP, IMAP4
- NFS, DNS, NIS, ...





# Socket

- È una *astrazione software* che rappresenta il terminale di una connessione tra due computer
- Per ogni connessione esiste un *socket* in ognuno dei computer coinvolti
- Il client effettua la richiesta di una connessione ad un server per un servizio collegato ad una determinata porta
- Se la richiesta è accettata la connessione tra i due applicativi dei due computer è stabilita





# Socket e Stream in Java

- Da ogni socket è possibile ottenere uno *stream* di *input* ed uno di *output*
- L'uso dei socket è trasparente: **readLine** e **println**

## BufferedReader

```
in = new BufferedReader(
 new InputStreamReader(
 socket.getInputStream()));
```

## PrintWriter

```
out = new PrintWriter(
 new BufferedWriter(
 new OutputStreamWriter(
 socket.getOutputStream()),
 true));
```

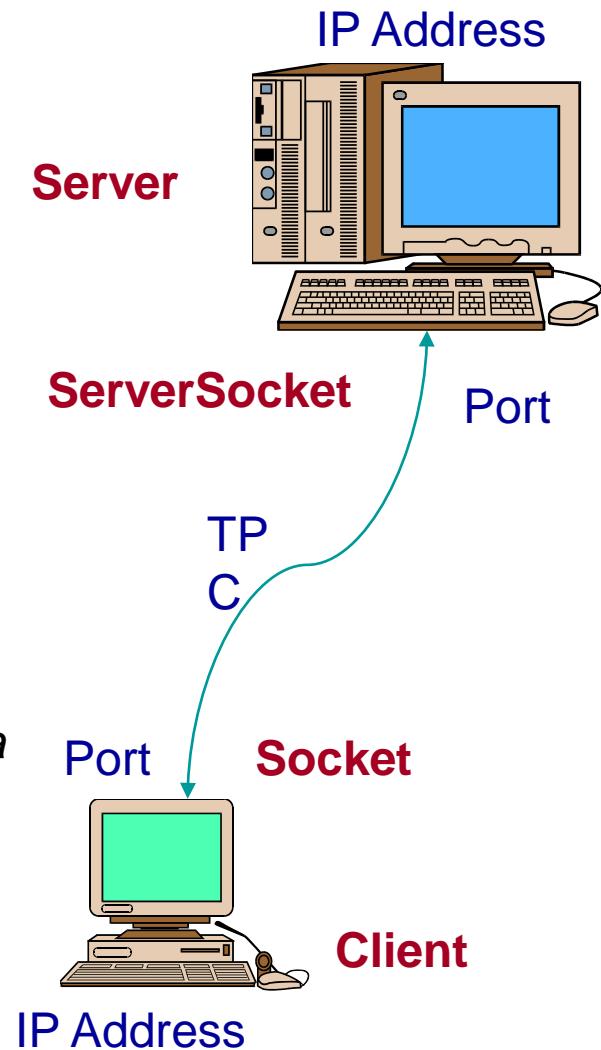


Abilita il *flush* immediato



# Socket in Java

- Package da importare `java.net`
- La classe `java.net.Socket` implementa un “*client socket*”
  - ◆ richiede l’indirizzo IP e il numero della porta del servizio a cui vogliamo connetterci
- La classe `java.net.ServerSocket` implementa un “*server socket*”
  - ◆ richiede il numero della porta a cui vogliamo associare il servizio
  - ◆ crea un “*server*” che attende le richieste di connessione
  - ◆ restituisce un socket nel momento in cui *accetta* un collegamento completamente istanziato nei parametri di collegamento (IP e Porta locale e remota).





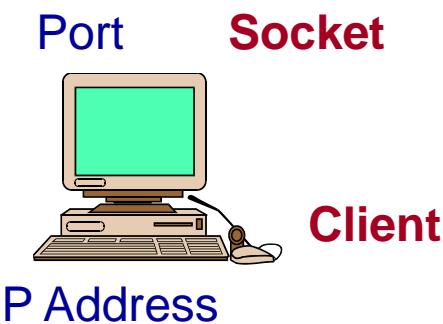
# Socket (Client)

```
Socket socket = new Socket(IP, PORT);
[...]
socket.close();
```

- Crea una connessione con un'applicazione server (in attesa in una `accept()`) e restituisce il relativo socket
- Le eccezioni che possono essere sollevate sono del tipo `IOException` (da catturare e gestire!)

E`un valore intero, in genere non inferiore a 1024 (riservati al sistema) e rappresenta il numero della porta a cui Vogliamo connetterci

E`un oggetto del tipo `java.net.InetAddress` contenente l'indirizzo IP della macchina a cui vogliamo connetterci ad esempio `speedy/130.192.241.1`





# Inet Address

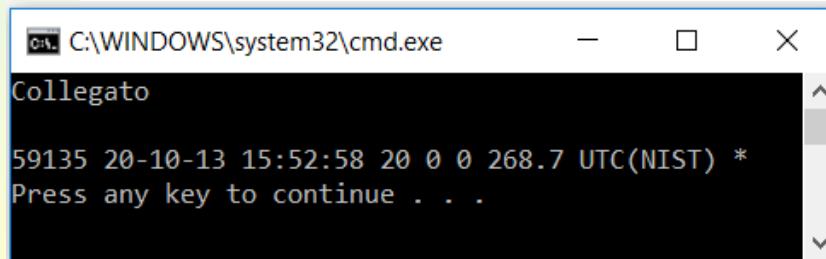
```
public class SocketTest1 {
 public static void main(String[] args) {
 String host = "www.unito.it";
 try {
 InetAddress local = InetAddress.getLocalHost();
 InetAddress addr = InetAddress.getByName(host);
 System.out.println("Locale: indirizzo IP: " + local);
 System.out.println("Remoto: indirizzo IP: " + addr);
 }
 catch(UnknownHostException e)
 {System.out.println(host +"sconosciuto");}
 }
}
```

The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window contains the following text:  
Locale: indirizzo IP: lilianasantech/192.168.1.5  
Remoto: indirizzo IP: www.unito.it/130.186.27.242  
Press any key to continue . . .



# Un esempio: connessione a un server

```
public class SocketTest {
 public static void main(String[] args){
 try {
 Socket s =
 new Socket("time-c.nist.gov", 13);
 System.out.println("Collegato");
 try {
 InputStream inStream = s.getInputStream();
 Scanner in = new Scanner(inStream);
 while (in.hasNextLine()) {
 String line = in.nextLine();
 System.out.println(line);
 }
 }
 finally
 { s.close();}
 }
 catch (IOException e)
 { e.printStackTrace();}
 }
}
```



- Si collega a un servizio che fornisce l'ora esatta

- La stringa **time-c.nist.gov** viene convertita nell'indirizzo IP del server

Il servizio di ora è disponibile sulla porta 13



# Astrazione: connessione URL

```
public class URLReader {
 public static void main(String[] args) throws Exception {
 URL yahoo = new URL("http://www.yahoo.com/");
 BufferedReader in = new BufferedReader(
 new InputStreamReader(yahoo.openStream()));
 PrintWriter out = new PrintWriter(
 new FileWriter("esempio.html"));
 String inputLine;
 while ((inputLine = in.readLine()) != null)
 out.println(inputLine);
 in.close();
 out.close();
 }
}
```



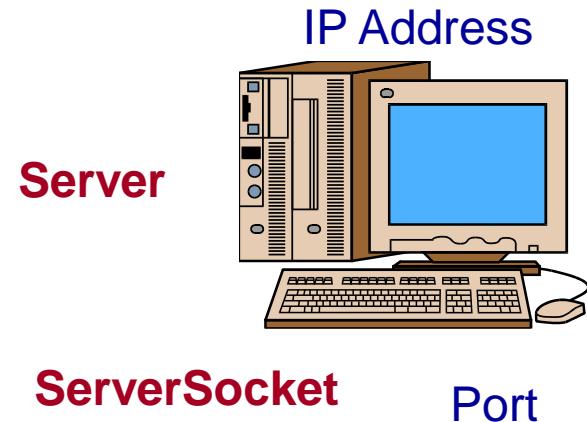
Scarica un file HTML dal sito di yahoo

- Per effettuare delle connessioni sia ad un server di posta che ad un server web è possibile utilizzare delle classi che realizzano delle astrazioni sul tipo di protocollo utilizzato
- L'uso dei socket è nascosto come quello del protocollo HTTP



# ServerSocket

- Il metodo `accept()` si mette in attesa di una richiesta di connessione da un socket client
- Le eccezioni che possono essere sollevate sono del tipo `IOException` (da catturare e gestire!)



```
ServerSocket s = new ServerSocket(PORT);
[...]
Socket socket = s.accept();
System.out.println("Accettato socket " + socket);
[...]
socket.close();
[...]
s.close();
```

Crea un nuovo socket sul server per accettazione di una richiesta di connessione

Chiude il socket di una connessione precedentemente accettata

Chiude il server



# ServerSocket e Socket: gestione errori

- E` importante che i socket siano propriamente chiusi al termine di un'applicazione in quanto questi sono una risorsa di rete condivisa del sistema
- La chiusura dei socket deve essere eseguita indipendentemente dal flusso di esecuzione per questo e` bene utilizzare il costrutto *try-finally* per garantirne la chiusura
- La stessa considerazione vale anche per il ServerSocket

```
[...]
try {
 [...]
 String str = in.readLine();
 [...]
 out.println("OFFERTA");
 [...]
}
catch(IOException e) {
 System.err.println("IO Exception" + e);
}
finally {
 try {
 socket.close();
 } catch(IOException e) {
 System.err.println("Socket not closed");
 }
}
[...]

try {
 Socket socket = s.accept();
 [...]
} finally {
 s.close();
}
```





# Un esempio: un server Eco (ripetizione)

Si vuole realizzare un server che invia al client l'eco di quanto riceve dal client.

Il server si mette in attesa della connessione di un client sulla porta 8189.

**ServerSocket s = new ServerSocket(8189);**

Dopo aver stabilito una connessione, il metodo **accept** del **ServerSocket** restituisce un **Socket** che rappresenta la connessione.

**Socket incoming = s.accept();**

Da questo oggetto si possono ricavare gli stream di input e output che collegano client e server

**InputStream inStream = incoming.getInputStream();**  
**OutputStream outStream = incoming.getOutputStream();**

# Un esempio: un server Eco (ripetizione)



```
public class EchoServer {
 public static void main(String[] args) {
 try{
 ServerSocket s = new ServerSocket(8189);
 Socket incoming = s.accept();
 try {
 InputStream inStream = incoming.getInputStream();
 OutputStream outStream = incoming.getOutputStream();
 Scanner in = new Scanner(inStream);
 PrintWriter out = new PrintWriter(outStream, true);
 out.println("Hello! Enter BYE to exit.");

 boolean done = false;
 while (!done && in.hasNextLine()) {
 String line = in.nextLine();
 out.println("Echo: " + line);
 if (line.trim().equals("BYE"))
 done = true;
 }
 } finally {incoming.close();}
 } catch (IOException e){e.printStackTrace();}
 }
}
```



# Come provare il server

Per usare il server si può scrivere un client che si connette alla porta 8189 all'indirizzo IP a cui si trova il server.

Se il server si trova sullo stesso computer del client, si può usare l'indirizzo 127.0.0.1, che indica il computer locale.

Si veda l'esempio **EchoClient**.





# Come servire più client

Il server descritto prima può servire solo un client. Se un altro client cerca di connettersi, rimane bloccato perché il server sta già gestendo un altro client.

Per servire più client si posso usare i thread. Ogni volta che il server ha stabilito una nuova connessione con un client, viene avviato un thread che si prende cura della connessione fra il server e quel client. Il programma principale del server può mettersi in attesa della connessione successiva.

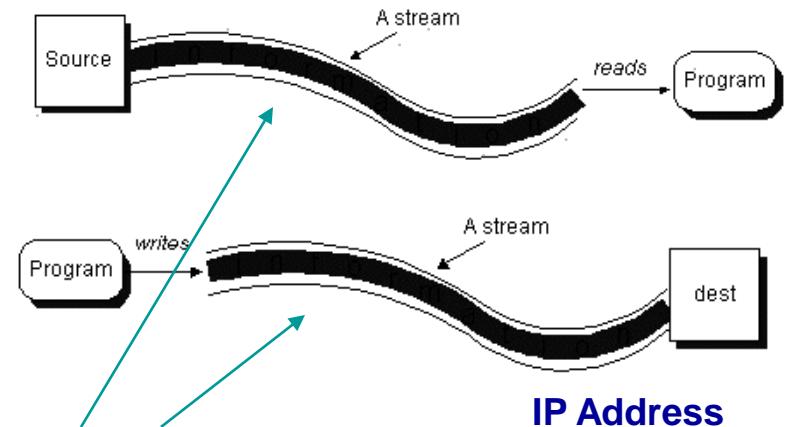
```
while (true)
{
 Socket incoming = s.accept();
 Runnable r = new ThreadedEchoHandler(incoming);
 Thread t = new Thread(r);
 t.start();
}
```



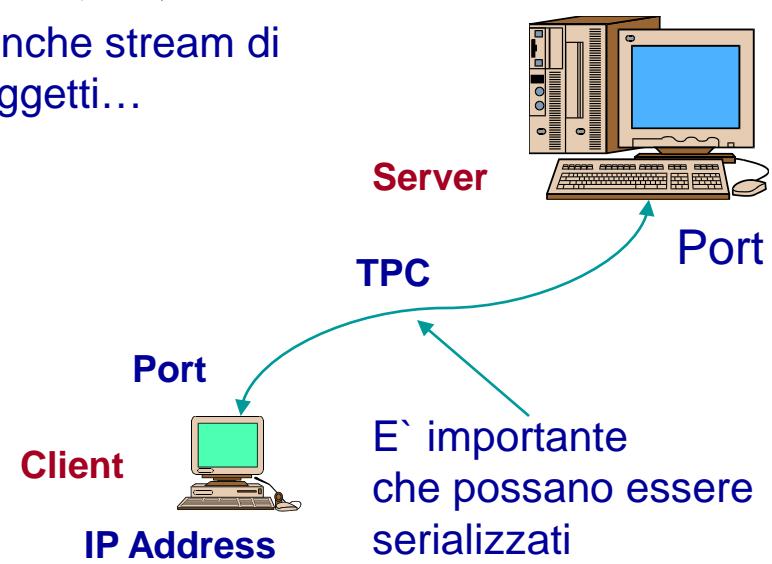


# Stream di oggetti e Socket

- Nell'esempio precedente i dati venivano trasmessi come stringhe da un client ad un server e viceversa
- In Java si puo` pero` utilizzare anche stream di oggetti (ObjectInputStream e ObjectOutputStream)
- In Java e` possibile utilizzare stream di oggetti tramite socket
- Gli oggetti devono implementare l'interfaccia java.io.Serializable



Anche stream di oggetti...





# Stream di oggetti e Socket

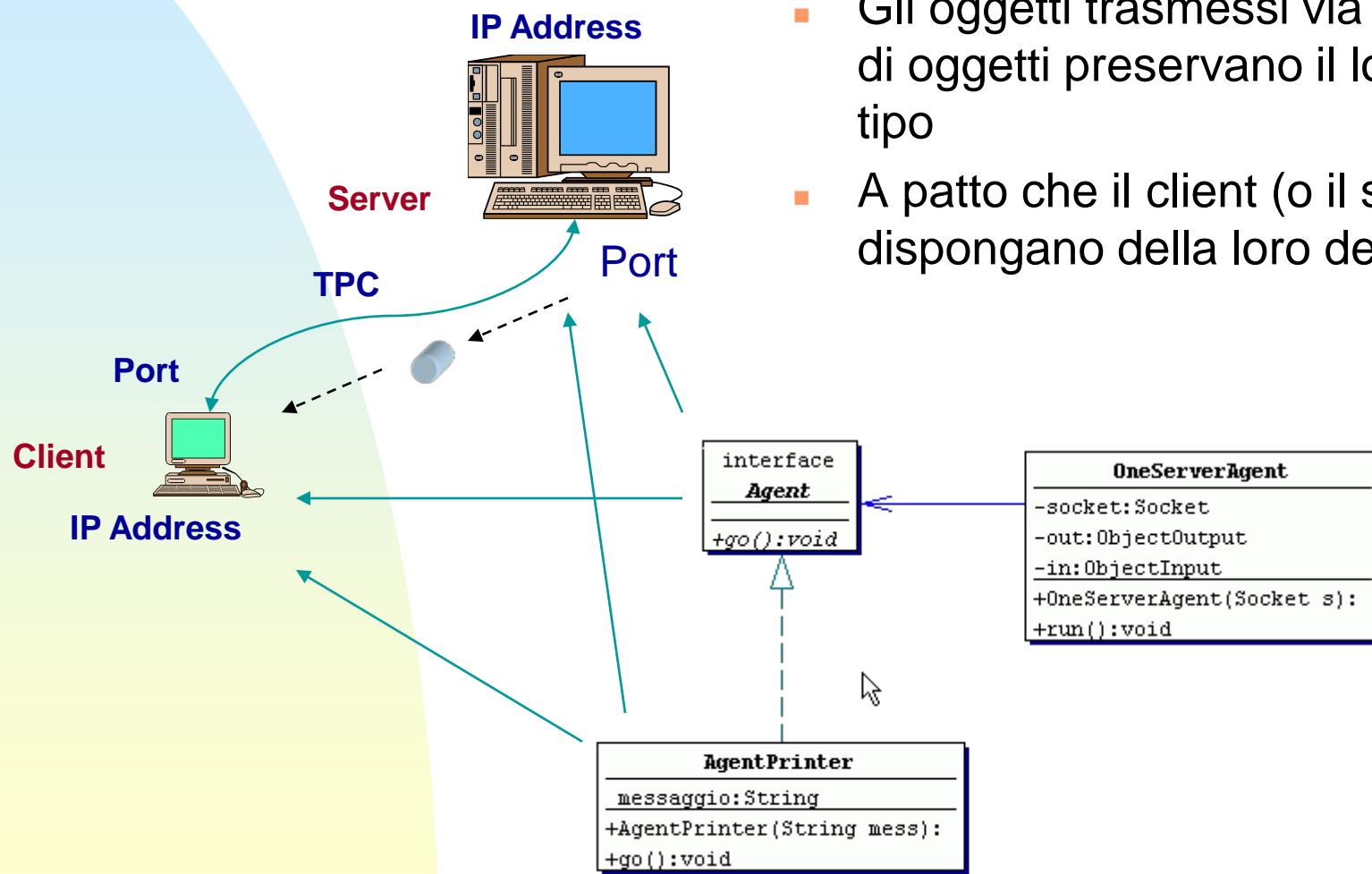
```
[...]
ServerSocket s = new ServerSocket(PORT);
[...]
Socket socket = s.accept();
[...]
ObjectInputStream in =
 new ObjectInputStream(socket.getInputStream());
ObjectOutputStream out =
 new ObjectOutputStream(socket.getOutputStream());
[...]
Offerta nuova = (Offerta)in.readObject();
[...]
out.writeObject("accettata");
[...]
```

← Server

- **Importante:** la dichiarazione nel client è analoga MA è l'oggetto di tipo ObjectOutputStream va creato prima dell'oggetto di tipo ObjectInputStream pena il deadlock (non è un bug!)



# Polimorfismo via socket?



- Gli oggetti trasmessi via stream di oggetti preservano il loro vero tipo
- A patto che il client (o il server) dispongano della loro definizione

Gianni Forlastro

# Introduzione a GIT



# Gianni Forlastro

- CTO presso Finwave  
(gruppo Lutech)
- Docente
- Organizzatore di eventi/conferenze  
(droidcon italy, Swift Heroes)
- Community Manager del  
GDG Cloud Torino  
Flutter Torino



<https://www.linkedin.com/in/giovanni-forlastro/>



<https://medium.com/@gianni.forlastro>



@mosquitoman81

# CONTROLLO DI VERSIONE

Il controllo versione (version control), in informatica, è la gestione di versioni multiple di un insieme di informazioni: gli strumenti software per il controllo versione sono ritenuti molto spesso necessari per la maggior parte dei progetti di sviluppo software o documentali gestiti da un team collaborativo di sviluppo o redazione.

Viene dunque usato prevalentemente nello sviluppo di progetti informatici per gestire la continua evoluzione dei documenti digitali come i disegni tecnici, il codice sorgente del software, la documentazione testuale e altre informazioni importanti su cui può lavorare una squadra di persone.

## Come funziona?

Il controllo versione si è sviluppato dai processi formali basati sui disegni cartacei. Le modifiche a questi documenti sono identificate incrementando un **numero o un codice** associato ad essi, denominato "**numero di versione**", "etichetta di versione", o semplicemente "versione", e sono etichettate con il nome della persona che ha apportato la modifica.

Una semplice forma di controllo, per esempio, assegna il numero 1 alla prima versione di un progetto. Quando viene apportata la prima modifica, il numero identificativo di versione passa a 2 e così via.

# LA STORIA

Lo sviluppo di Git è iniziato dopo che molti sviluppatori del kernel di Linux sono stati costretti ad abbandonare l'accesso al codice sorgente tramite il sistema proprietario BitKeeper.

**BitKeeper** è un software di controllo di versione distribuito per il codice sorgente dei programmi, prodotto da BitMover Inc., e in voga da inizio degli anni 2000.

La possibilità di utilizzare BitKeeper gratuitamente era stata ritirata dal detentore dei diritti d'autore Larry McVoy dopo avere sostenuto che Andrew Tridgell aveva effettuato il reverse engineering dei protocolli.

Linus Torvalds, conosciuto soprattutto per essere stato l'autore della prima versione del kernel Linux e coordinatore del progetto di sviluppo, voleva un sistema distribuito che potesse usare come BitKeeper, ma nessuno dei sistemi disponibili gratuitamente soddisfaceva i suoi bisogni, particolarmente il suo bisogno di velocità.

Lo sviluppo di Git è cominciato il 3 aprile 2005 e Torvalds raggiunse i suoi obiettivi in termini di prestazioni il 29 aprile 2005: Git riusciva ad applicare 6 o 7 patch a Linux in un secondo. Il 16 giugno 2005 è stata pubblicata la versione 2.6.12 del kernel di Linux, la prima gestita con Git.

# DOWNLOAD E INSTALLAZIONE

<https://git-scm.com/downloads>

## Downloads

 macOS       Windows

 Linux/Unix

Older releases are available and the Git source repository is on GitHub.

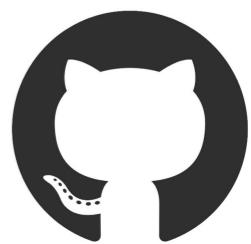
Latest source Release  
**2.38.0**  
[Release Notes \(2022-10-02\)](#)

[Download for Mac](#)



Oppure seguire le informazioni a questo link:

<https://www.git-scm.com/book/en/v2/Getting-Started-Installing-Git>



# GITHUB

**GitHub** è un servizio di hosting per progetti software. Il nome deriva dal fatto che "GitHub" è una implementazione dello strumento di controllo versione distribuito Git.

<https://github.com/>



## Write better code

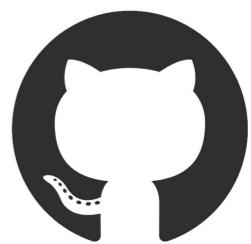
Collaboration makes perfect. The conversations and code reviews that happen in pull requests help your team share the weight of your work and improve the software you build. [Learn about code review.](#)

## Manage your chaos

Take a deep breath. On GitHub, project management happens in issues and project boards, right alongside your code. All you have to do is mention a teammate to get them involved. [Learn about project management.](#)

## Find the right tools

Browse and buy apps from GitHub Marketplace with your GitHub account. Find the tools you like or discover new favorites—then start using them in minutes. [Learn about integrations.](#)



# GITHUB

GitHub Inc. venne fondata nel 2008 con il nome di Logical Awesome.

Il sito era realizzato utilizzando Ruby on Rails ed Erlang dagli sviluppatori Chris Wanstrath, PJ Hyett e Tom Preston-Werner.

Il 24 febbraio 2009 fu annunciata l'attivazione di 46.000 repository pubblici di cui 17.000 solo nei 30 giorni precedenti e il 5 luglio raggiunsero i 100.000 utenti.

Nel luglio 2012 la compagnia ha ricevuto 100 milioni di dollari da Andreessen Horowitz per avviare la società.

Il 16 gennaio 2013 GitHub ha annunciato di avere 3 milioni di utenti e più di 5 milioni di repository.

Il 4 giugno 2018 Microsoft ha annunciato di aver acquisito la società per 7,5 miliardi di dollari in azioni (dopo questa acquisizione è aumentato il numero di progetti importati su GitLab, un sito analogo a GitHub).



# GITLAB

<https://gitlab.com/>

GitLab è una piattaforma web **open source** che permette la gestione di repository Git e di funzioni trouble ticket e appartiene a GitLab Inc.

A dicembre 2016, l'azienda conta 150 membri interni, più oltre 1400 contributori open source.

Viene utilizzato da organizzazioni come IBM, Sony, NASA, Alibaba, O'Reilly Media, CERN, GNOME Foundation.

Originariamente il codice venne scritto in Ruby, ma successivamente alcune parti furono riscritte in Go.

Nel luglio 2013, il prodotto è stato diviso in: *GitLab CE (Community Edition)* e *GitLab EE (Enterprise Edition)*.

# CREATE ACCOUNT GITHUB

[https://github.com/join?  
ref\\_cta=Sign+up&ref\\_loc=header+logged+out&ref\\_page=%2F&source=header-home](https://github.com/join?ref_cta=Sign+up&ref_loc=header+logged+out&ref_page=%2F&source=header-home)

## Create your account

Username \*

Email address \*

Password \*

Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter. [Learn more.](#)

Email preferences  
 Send me occasional product updates, announcements, and offers.

### Verify your account

Risolvere questo puzzle, così sappiamo che è una persona reale

L'account è gratuito e per registrarsi sono necessari:

- Username
- Email
- Password valida

# CREATE ACCOUNT GITLAB

[https://gitlab.com/users/sign\\_up](https://gitlab.com/users/sign_up)

## Register for GitLab

First name  Last name

Username

Email

Password

Minimum length is 8 characters

I'd like to receive updates via email about GitLab.

Non sono un robot  reCAPTCHA  
Privacy - Termini

**Register**

L'account anche qui è gratuito e per registrarsi sono necessari:

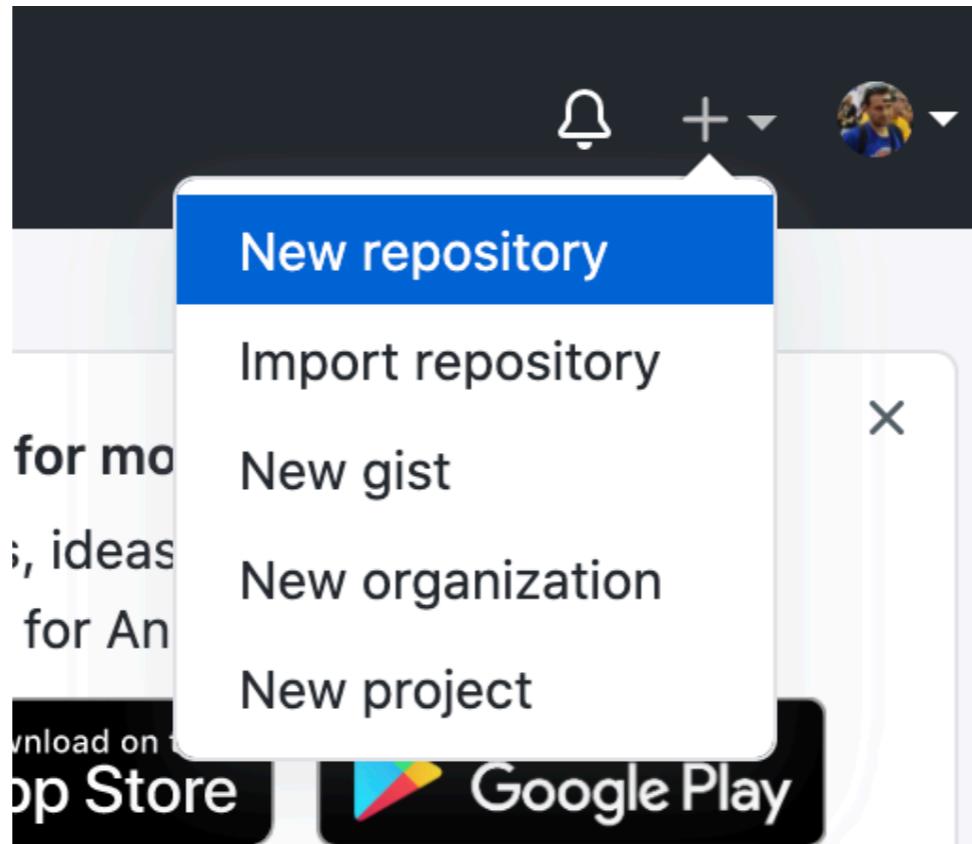
- Cognome e nome
- Username
- Email
- Password valida

# CREATE REPOSITORY GITHUB

Per creare un nuovo repository, è necessario andare in alto a destra premere sul

+ > New repository

come mostrato nell'immagine sottostante



# CREATE REPOSITORY GITHUB

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Owner \*



supergianni ▾

Repository name \*

testprogrammazione



Great repository names are short and memorable. Need inspiration? How about [probable-broccoli](#)?

Description (optional)

Progetto Java



Public

Anyone on the internet can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.

Add a README file

This is where you can write a long description for your project. [Learn more](#).

Add .gitignore

Choose which files not to track from a list of templates. [Learn more](#).

Choose a license

A license tells others what they can and can't do with your code. [Learn more](#).

[Create repository](#)

# CREATE REPOSITORY GITHUB

## Add .gitignore

Choose which files not to track from a list of templates. [L](#)

.gitignore template: Java ▾

### .gitignore template

Filter ignores...

Android

AppEngine

AppceleratorTitanium

ArchLinuxPackages

Autotools

C

C++

CFWheels

CMake

CUDA

CakePHP

Ogni volta che aggiungeremo un nuovo file o ne modificheremo uno già esistente, **Git** si accorgerà delle modifiche apportate e ci segnalerà quali file saranno all'interno del prossimo commit.

Ci sono però alcune situazioni in cui è necessario **ignorare dei file e fare in modo che Git non tenga traccia di eventuali modifiche**. Alcuni possibili casi sono:

- cartelle create da un package manager come NPM o bower
- directory contenenti i file da caricare su un server per un'applicazione web
- File di supporto generati dal sistema operativo o da altri strumenti
- file eseguibili generati a partire dal codice sorgente
- archivi compressi come file.gz o.zip

Per selezionare dei file da ignorare in un determinato progetto, possiamo creare un file **.gitignore** nella cartella base.

All'interno del file **.gitignore** inseriremo su ogni riga le regole che Git deve seguire per capire quale file ignorare o meno. È possibile limitarsi a elencare una lista di file o usare delle espressioni regolari. Per inserire dei commenti nel file **.gitignore** basta iniziare una riga con il carattere '#'.

# CREATE REPOSITORY GITHUB

The screenshot shows a GitHub repository page for 'supergianni / testprogrammazione'. The repository is private. The 'Code' tab is selected. At the top, there are links for Issues, Pull requests, Actions, Projects, Security, Insights, and Settings. Below these, it shows 1 branch and 0 tags. A commit by 'supergianni' titled 'Initial commit' is listed, along with a '.gitignore' file. A note says 'Add a README with an overview of your project.' On the right, a dropdown menu is open under the 'Code' button, showing options for Clone (HTTPS, SSH, GitHub CLI), Open with GitHub Desktop, and Download ZIP.

🔒 supergianni / testprogrammazione Private

<> Code Issues Pull requests Actions Projects Security Insights Settings

main ▾ 1 branch 0 tags

supergianni Initial commit

.gitignore Initial commit

Add a README with an overview of your project.

Go to file Add file ▾ Code ▾

Clone

HTTPS SSH GitHub CLI New

git@github.com:supergianni/testprog

Use a password-protected SSH key.

Open with GitHub Desktop

Download ZIP

Il repository è stato creato. Ora è possibile utilizzarlo andando a salvare il vostro progetto all'interno.

# Il git config

Con git viene fornito con uno strumento chiamato **git config** che consente di ottenere e impostare variabili di configurazione che controllano tutti gli aspetti di come git opera.

Una delle prime cose che si può fare è impostare il tuo nome ed il tuo indirizzo e-mail:

```
$ git config --global user.name "Cognome e nome"
$ git config --global user.email email@example.com
```

Git utilizza una serie di files di configurazione per determinare comportamenti non standard che si potrebbero volere.

In primo luogo Git cercherà questi valori nel file /etc/gitconfig, il quale contiene valori per ogni utente e repository di sua proprietà presenti sul sistema.

Se si passa l'opzione --system a git config, il programma leggerà e scriverà in modo specifico su questo file.

# Il git config

Successivamente Git controlla il file

`~/.gitconfig`

che è specifico per ogni utente.

Si può fare in modo che Git legga e scriva su questo file utilizzando l'opzione `--global`

Infine, git controlla i valori di configurazione nel file di configurazione presente nella directory git (`.git/config`) di qualsiasi repository che stai utilizzando. Questi valori sono specifici del singolo repository.

Ogni livello sovrascrive i valori del livello precedente: ad esempio i valori contenuti in `.git/config` sovrascrivono quelli in `/etc/gitconfig`

Si può inoltre impostare questi valori modificando manualmente il file ed inserendo la corretta sintassi, tuttavia solitamente è più semplice eseguire il comando `git config`

# II git config

Quindi:

## Your Identity

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

## Your Editor

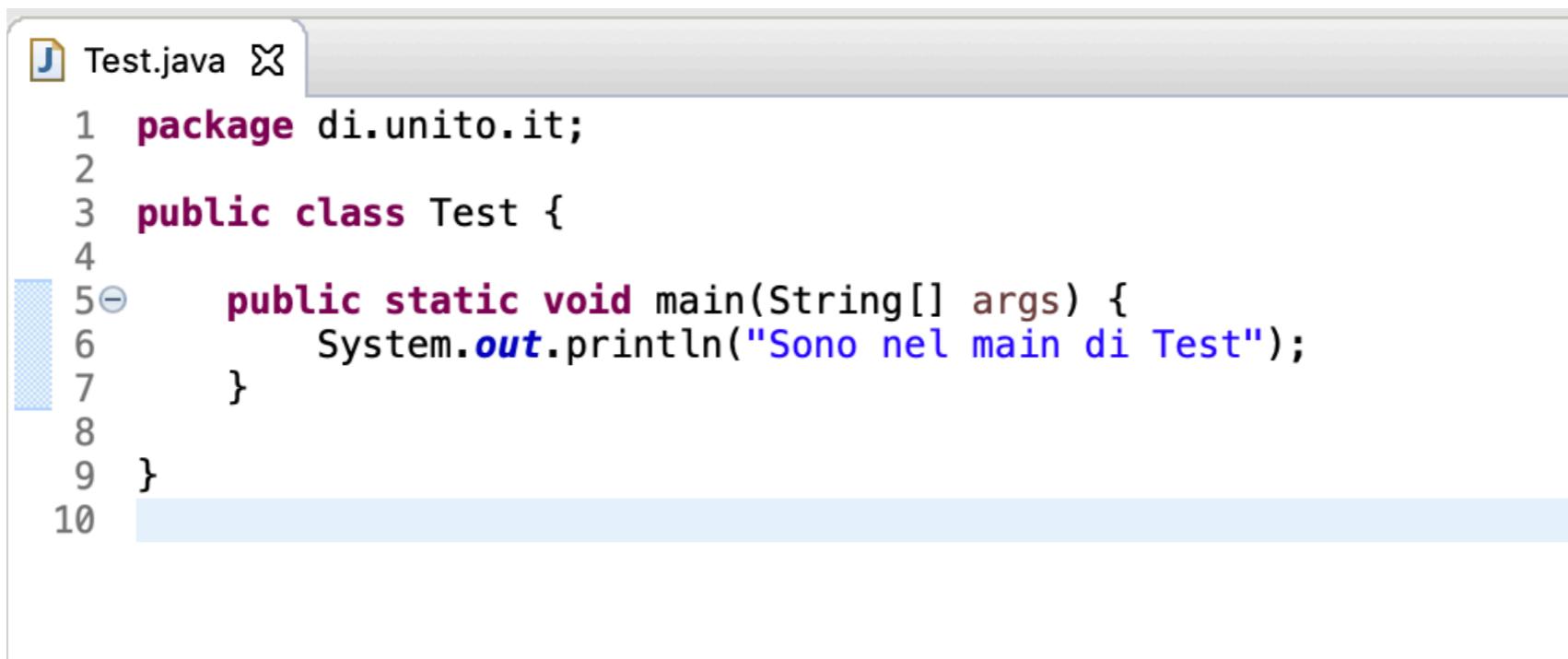
```
$ git config --global core.editor emacs
```

## Your Editor -- On windows -- 🤦

```
$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' \
-multiInst -notabbar -nosession -noPlugin"
```

# SALVARE UN PROGETTO SU GITHUB

Immaginiamo di creare ora un nuovo progetto Java, molto semplice, così come da immagine seguente.



```
Test.java ✘
1 package di.unito.it;
2
3 public class Test {
4
5 public static void main(String[] args) {
6 System.out.println("Sono nel main di Test");
7 }
8
9 }
10
```

Una volta creato il progetto possiamo andare a salvarlo all'interno del repository creato.

# SALVARE UN PROGETTO SU GITHUB

cd YOUR\_FOLDER

```
git init
git remote add origin LINK REPOSITORY REMOTO
```

Solo la prima volta per inizializzare il progetto

```
git add .
git commit -m "Initial commit"
git push -u origin master
```

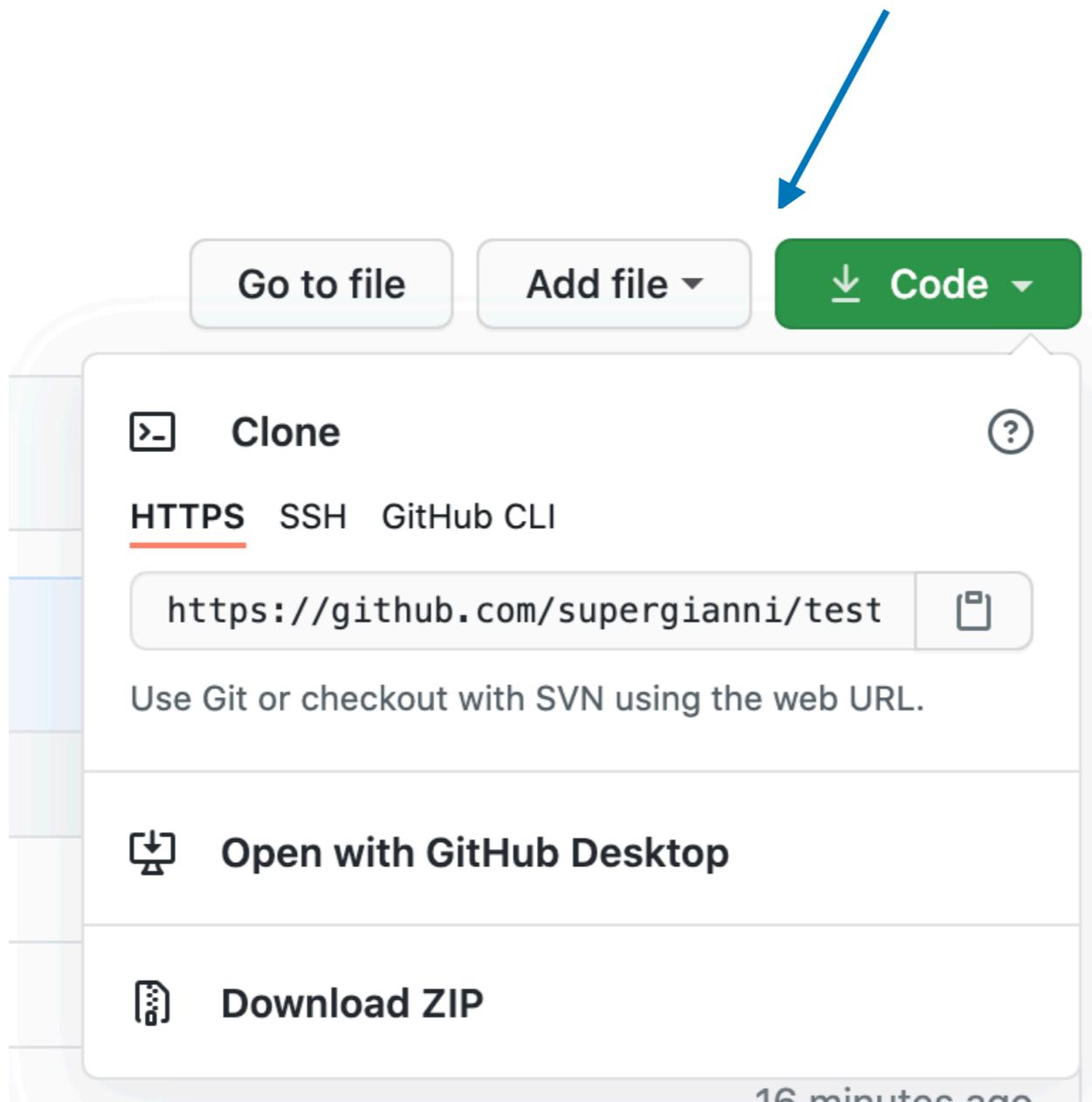
Sempre, per salvare le modifiche

The screenshot shows a GitHub repository interface. At the top, there's a yellow header bar with the message "master had recent pushes 1 minute ago" and a green "Compare & pull request" button. Below this, there are navigation buttons for "master", "2 branches", "0 tags", and links to "Go to file", "Add file", and "Code". A message in the center says "This branch is 1 commit ahead, 1 commit behind main." with "Pull request" and "Compare" buttons. The main content area displays a list of commits from "supergianni" with the message "Initial commit". Each commit includes a timestamp ("13 minutes ago") and a "1 commits" badge. Below the commits, there's a list of files: ".settings", "bin/di/unito/it", "src/di/unito/it", ".classpath", and ".project", each with an "Initial commit" message and a "13 minutes ago" timestamp. At the bottom, there's a blue footer bar with the text "Add a README with an overview of your project." and a green "Add a README" button.

| File            | Commit Message | Time Ago       |
|-----------------|----------------|----------------|
| .settings       | Initial commit | 13 minutes ago |
| bin/di/unito/it | Initial commit | 13 minutes ago |
| src/di/unito/it | Initial commit | 13 minutes ago |
| .classpath      | Initial commit | 13 minutes ago |
| .project        | Initial commit | 13 minutes ago |

# SALVARE UN PROGETTO SU GITHUB

*git remote add origin <https://github.com/supergianni/testprogrammazione.git>*

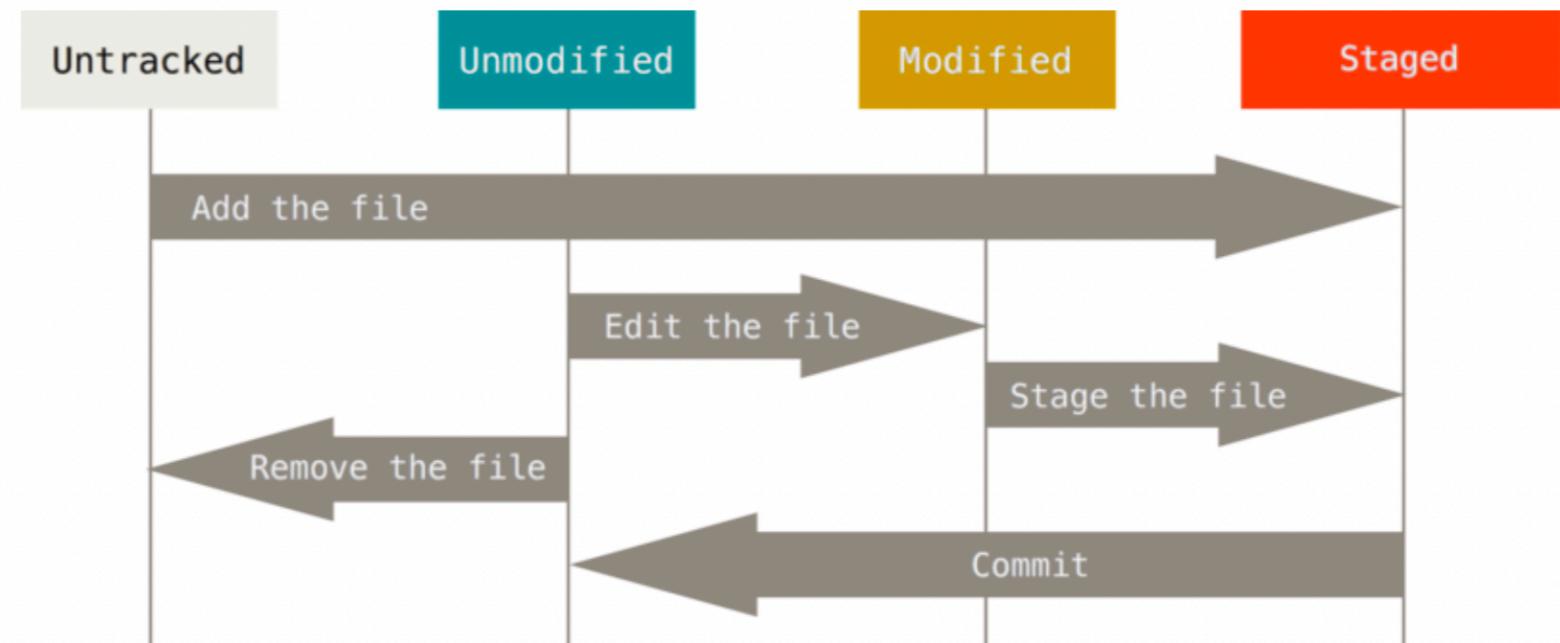


# CICLO DI VITA DI UN FILE

## Files lifecycle

working copy.

- files can be:
  - untracked
  - tracked
    - unmodified
    - modified
    - staged



Verificare la “situazione” attuale dei file del progetto:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

# CONDIVIDERE UN PROGETTO SU GITHUB

E' necessario dare accesso alle implementazioni ad altre utenze (colleghi di lavoro, colleghi universitari...).

Da quel momento anche loro potranno effettuare modifiche sul progetto.  
La gestione è per progetto, ed è possibile farlo sotto Settings > Manage access

The screenshot shows the GitHub repository settings page for 'supergianni / testprogrammazione'. The 'Settings' tab is selected. On the left, a sidebar menu includes 'Options', 'Manage access' (which is currently selected), 'Security & analysis', 'Branches', 'Webhooks', 'Notifications', 'Integrations', 'Deploy keys', 'Secrets', and 'Actions'. The main content area is titled 'Who has access'. It shows two sections: 'PRIVATE REPOSITORY' (with a lock icon) and 'DIRECT ACCESS' (with a person icon). Under 'PRIVATE REPOSITORY', it says 'Only those with access to this repository can view it.' and has a 'Manage' button. Under 'DIRECT ACCESS', it says '0 collaborators have access to this repository. Only you can contribute to this repository.' A large box below is titled 'Manage access' and contains a network icon and the text 'You haven't invited any collaborators yet'. A green 'Invite a collaborator' button is at the bottom.

supergianni / testprogrammazione Private

Unwatch 1 Star

Code Issues Pull requests Actions Projects Security Insights Settings

Who has access

PRIVATE REPOSITORY

Only those with access to this repository can view it.

Manage

DIRECT ACCESS

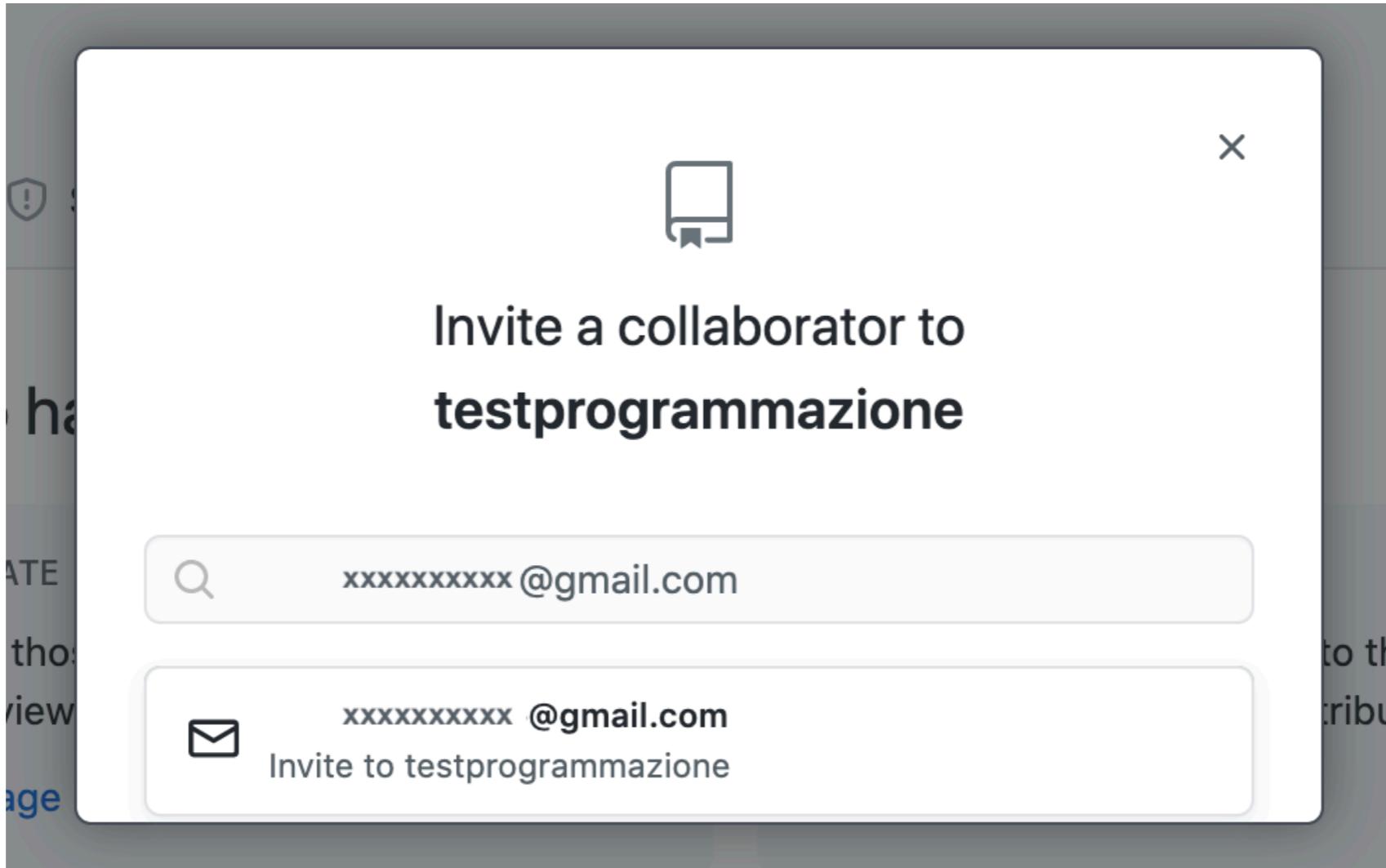
0 collaborators have access to this repository. Only you can contribute to this repository.

Manage access

You haven't invited any collaborators yet

Invite a collaborator

# CONDIVIDERE UN PROGETTO SU GITHUB



Invitiamo tramite mail (o userid) il collega/compagno scelto, che riceverà una mail per accettare l'invito ad unirsi al team di sviluppo.

# CONDIVIDERE UN PROGETTO SU GITHUB

A questo punto il secondo utente dovrà reperire tutto il codice sorgente creato fino a quel momento, clonando il progetto con l'istruzione

```
git clone URL.git
```

Il secondo utente, quindi, modifica il codice della seconda utenza ed effettua anche lui le istruzioni per inviare il codice a GitHub (effettua quindi commit/push).

This branch is 2 commits ahead, 1 commit behind main.

[Pull request](#) [Compare](#)

Con la mia utenza quindi, sempre da terminale, uso l'istruzione :

```
git pull
```

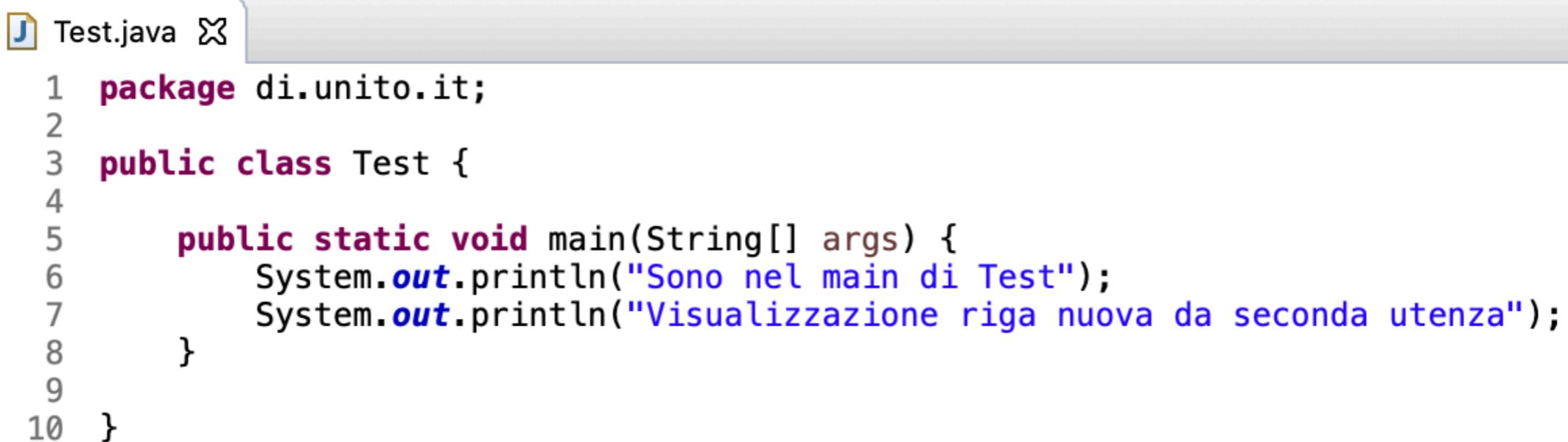
che estrarrà dal repository remoto le modifiche :

```
remote: Enumerating objects: 16, done.
remote: Counting objects: 100% (16/16), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 10 (delta 2), reused 0 (delta 0), pack-reused 0
Decompressione degli oggetti in corso: 100% (10/10), 1.66 KiB | 212.00 KiB/s, fatto.
Da https://github.com/supergianni/testprogrammazione
 9981358..596e1c3 master -> origin/master
 * [nuovo branch] main -> origin/main
Aggiornamento di 9981358..596e1c3
Fast-forward
 src/di/unito/it/Test.java | 1 +
 1 file changed, 1 insertion(+)
```

# CONDIVIDERE UN PROGETTO SU GITHUB

Il comando “pull” aggiorna il repository locale (sul proprio PC) alla “push” più recente. In questo modo si avrà il codice sempre aggiornato.

Tornando al codice sorgente, troverò la modifica che il secondo utente ha effettuato:



The screenshot shows a code editor window with a tab labeled "Test.java". The code is a Java program with the following content:

```
1 package di.unito.it;
2
3 public class Test {
4
5 public static void main(String[] args) {
6 System.out.println("Sono nel main di Test");
7 System.out.println("Visualizzazione riga nuova da seconda utenza");
8 }
9
10 }
```

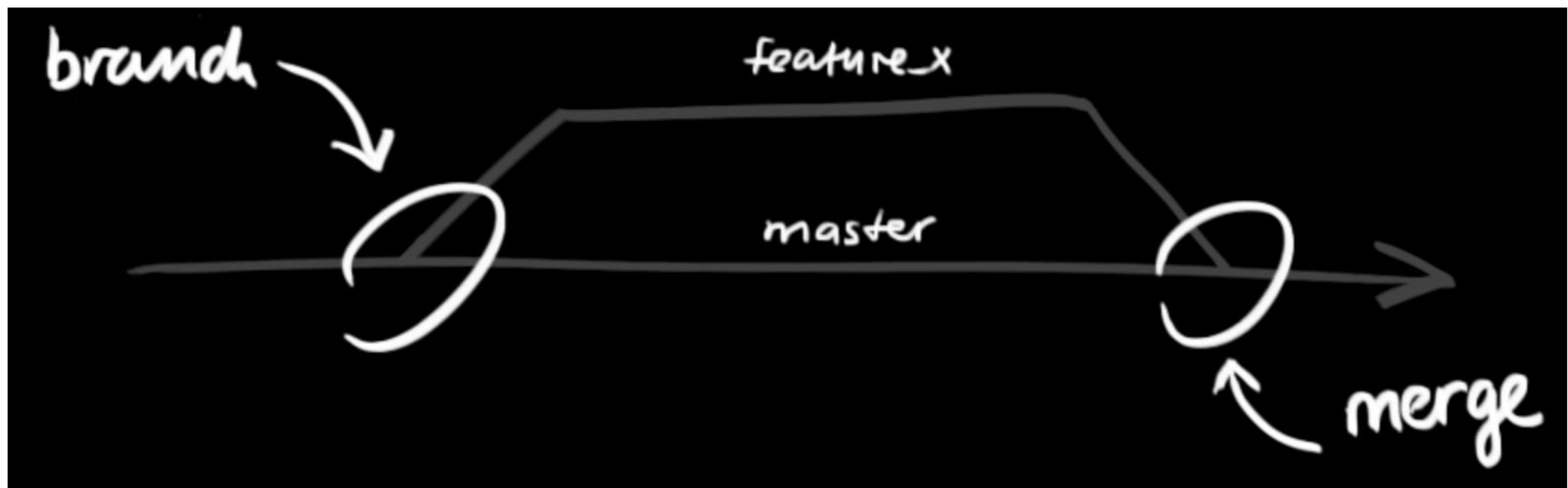
The code editor highlights certain parts of the code in blue, specifically the string literals "Sono nel main di Test" and "Visualizzazione riga nuova da seconda utenza". These highlighted lines are the changes made by the second user, as mentioned in the accompanying text.

# I BRANCH

I branch ('ramificazioni') sono utilizzati per sviluppare features che sono isolate l'una dall'altra.

Il branch master è quello di default quando crei un repository.

Puoi usare altri branch per lo sviluppo ed infine incorporarli ('merge') nel master branch una volta completati.



crea un nuovo branch chiamato "feature\_x" e passa al nuovo branch usando  
`git checkout -b feature_x`  
`git push origin feature_x`

# I BRANCH

## A cosa servono i branch?

Si ipotizzi di dover lavorare ad un progetto specifico come per esempio la documentazione di un software.

Una volta completata la bozza iniziale quest'ultima dovrà passare il vaglio della revisione per poter essere approvata e pubblicata.

Se lo sviluppo dell'applicazione rimane attivo nel tempo si presenterà certamente l'esigenza di effettuare degli aggiornamenti anche a carico della documentazione.

Non sempre però le necessità di aggiornamento si presentano dopo che la revisione della prima bozza viene completata.

Motivo per cui si potrebbe avere l'esigenza di disporre di una versione modificata con i nuovi dati, pur conservando quanto prodotto fino a quando sono stati apportati i cambiamenti richiesti dall'evoluzione del progetto in corso.

In sostanza si avranno due versioni distinte della medesima bozza.

# I BRANCH

## A cosa servono i branch?

Bisognerà, a questo punto, effettuare l'aggiornamento del repository principale (master) stando attenti a non andare in conflitto.

Il comando “**git merge**” permette quindi di unire due branch differenti incorporando le modifiche apportate nei commit di un branch, che passiamo come argomento, nel branch corrente.

### Potranno accadere due casi :

1. Caso *più semplice*. Ci troviamo in un caso particolare in cui il percorso che unisce i due branch è lineare. Git si limiterà quindi a 'spostare' il branch **master** in avanti in modo da puntare al nuovo commit. Si parla in questo caso di **Fast-forward merge** che è un caso particolare in cui non viene creato un nuovo commit, ma viene solo riposizionato il branch in cui vengono incorporate le modifiche.
2. Caso *con conflitti*. Un conflitto o conflict può presentarsi quando due o più cambiamenti concorrenti sono stati applicati sulla stessa linea di codice, o quando una persona sta lavorando su un file che è stato eliminato. I conflitti tra due cambiamenti sullo stesso codice, avvengono quando si esegue un merge di due branch. In questo caso bisogna scegliere manualmente quale dei due cambiamenti, vogliamo tenere nel codice.

# I BRANCH

## Risolvere i conflitti

Due sviluppatori stanno modificando lo stesso file e linea su due branch diversi, Git mostrerà un errore di conflitto quando si proverà a unire (merge) i due branch.

*CONFLICT (content): Merge conflict in lib/hello.html*

*Automatic merge failed; fix conflicts and then commit the result.*

Per risolvere i problemi di conflitto tra due branch in Git, bisogna fare un nuovo commit dove manualmente specifichiamo come risolvere i conflitti.

Vediamo come:

Prima di tutto aprite il Terminale e navigate fino alla directory della vostra repository Git che causa il conflict.

Utilizzando il comando git status su una repository con dei conflict presenti vi mostrerà quale file sta causando il conflitto.

```
On branch branch
You have unmerged paths.
(fix conflicts and run "git commit")
#
Unmerged paths:
(use "git add ..." to mark resolution)
#
both modified: fileConflitto.js
#
no changes added to commit (use "git add" and/or "git commit -a")
```

# I BRANCH

## Risolvere i conflitti

Ora che sappiamo quale sia il file che sta causando il conflict del merge, vediamo come risolvere questo problema.

Utilizzando il vostro text editor, aprite il file contenente il conflitto (nel nostro caso, ad esempio, fileConflitto.js)

Git automaticamente inserisce nel file dei caratteri per mostrarvi dove il conflitto sia presente.

L'inizio del conflitto è dato da:

**<<<<< HEAD**

I due cambiamenti concorrenti sono separati da:

**=====**

seguito da :

**>>>>> nome-branch**

per marcare la fine del conflict.

# I BRANCH

Vediamo un esempio:

fileConflitto.js

```
... var x = 10;
<<<<< HEAD
x = x + 1;
=====
x = x - 1;
>>>>> branch-sottrazione
...
```

Nell'esempio superiore vediamo che in un branch la variabile x è stata incrementata, mentre nell'altro branch è stata diminuita.

Questo causa un conflitto quando si esegue il merge.

Per eliminare il conflitto in Git bisogna manualmente eliminare il comando non desiderato, oltre ad eliminare i delimitatori del conflict <<<<<, =====, >>>>>

# I BRANCH

Se nel nostro caso volessimo incrementare la variabile x dopo aver eseguito il merge, possiamo riscrivere il nostro file in questo modo:

```
...
var x = 10;
x = x + 1;
```

```
...
```

Ora non ci rimane altro che aggiungere i cambiamenti eseguiti alla repository Git con i soliti comandi visti in precedenza (add/commit/push).

Ovviamente, prima di proseguire dovrete esser certi di non avere ulteriori conflitti nell'intero progetto (e in caso fare questo stesso procedimento per risolverli tutti).

# IN CASO DI ERRORI

Nel caso si sia fatto qualcosa di sbagliato nelle implementazioni del proprio repository locale (quello del proprio PC), si possono sostituire i cambiamenti fatti in locale andando a prendere quelli del repository remoto con il comando

*git checkout -- <nomedelfile>*

E' possibile anche eliminare tutti i cambiamenti e commits fatti in locale, recuperando l'ultima versione dal server, facendo puntare il proprio master branch a quella versione in questo modo

*git fetch origin  
git reset --hard origin/master*

# ALTRI COMANDI UTILI

Per vedere le modifiche del progetto possiamo utilizzare il comando :

*git status*

Per vedere i cambiamenti dei singoli files possiamo utilizzare :

*git diff*

Mentre se ci interessano tutti i commit eseguiti :

*git log*

# DOCUMENTAZIONE UTILE

**Reference Manual:** <https://book.git-scm.com/doc>

**Approfondimento:** <https://www.html.it/guide/git-la-guida/>

**Comandi utili:** <https://gist.github.com/tesseslol/da62aabec74c4fed889ea39c95efc6cc>

**Interessato alla conferenza GitHub?** <https://githubuniverse.com/>

*Online*  
8-10 Dicembre 2020



# Ambiente di sviluppo SW IntelliJ IDEA

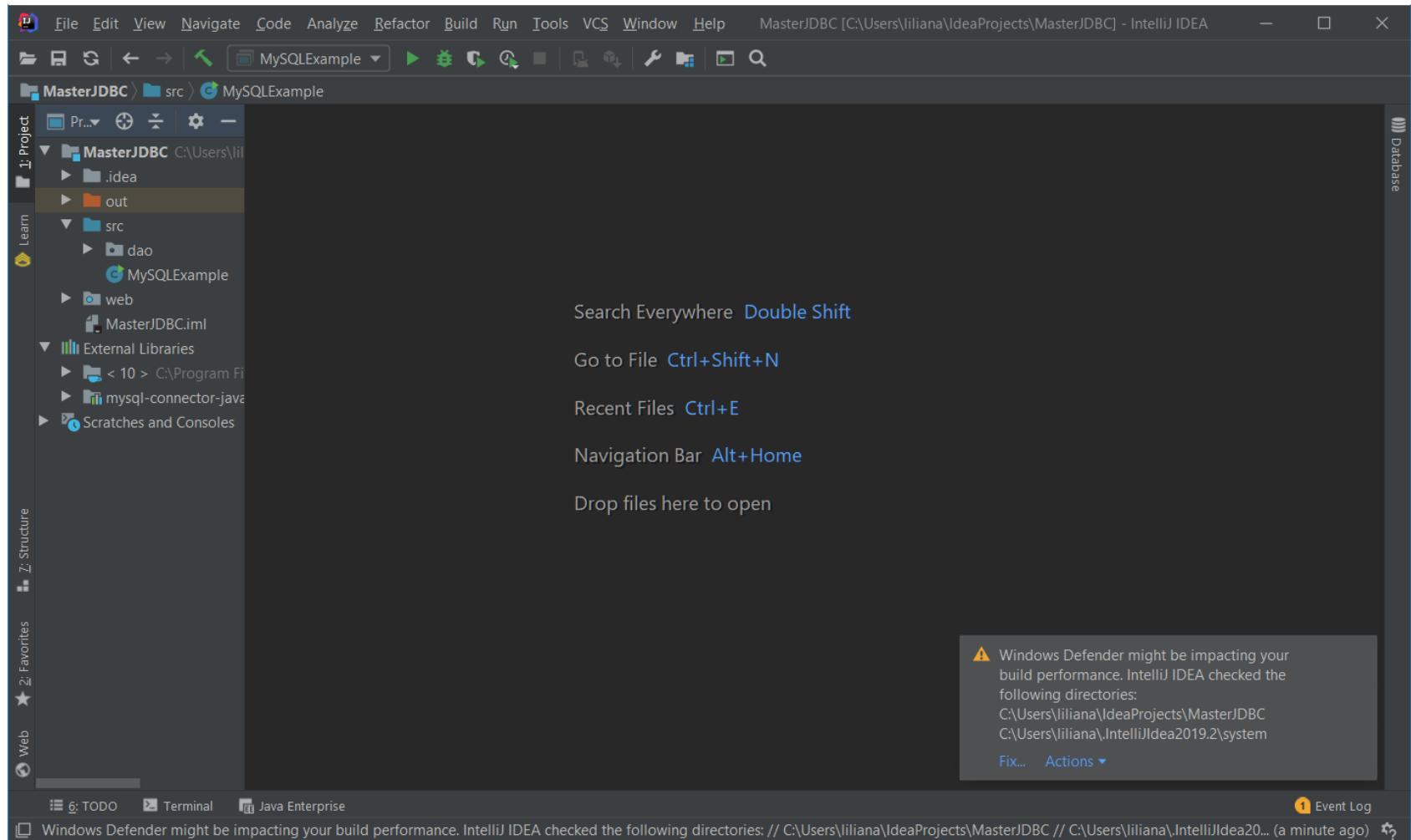
Nello sviluppo di applicazioni SW di medie/grandi dimensioni è indispensabile usare un IDE (Integrated Development Environment). L'IDE ci aiuta a compilare e eseguire i programmi, caricarli su web server (se web), etc.

- IntelliJ IDEA è un editor per sviluppo di applicazioni web e stand-alone con il supporto al debugging.
- IntelliJ IDEA offre l'ambiente per eseguire le applicazioni.

# IntelliJ IDEA

- **La versione Community** permette di sviluppare applicazioni java stand-alone (non web)
- **La versione ULTIMATE** è la versione completa dell'IDE. Richiede licenza, voi potete scaricarla con licenza education annual (usando il vostro account di email di myUnito)
- **Download:**  
**<https://www.jetbrains.com/idea/download/>**

# Start Page di IntelliJ IDEA



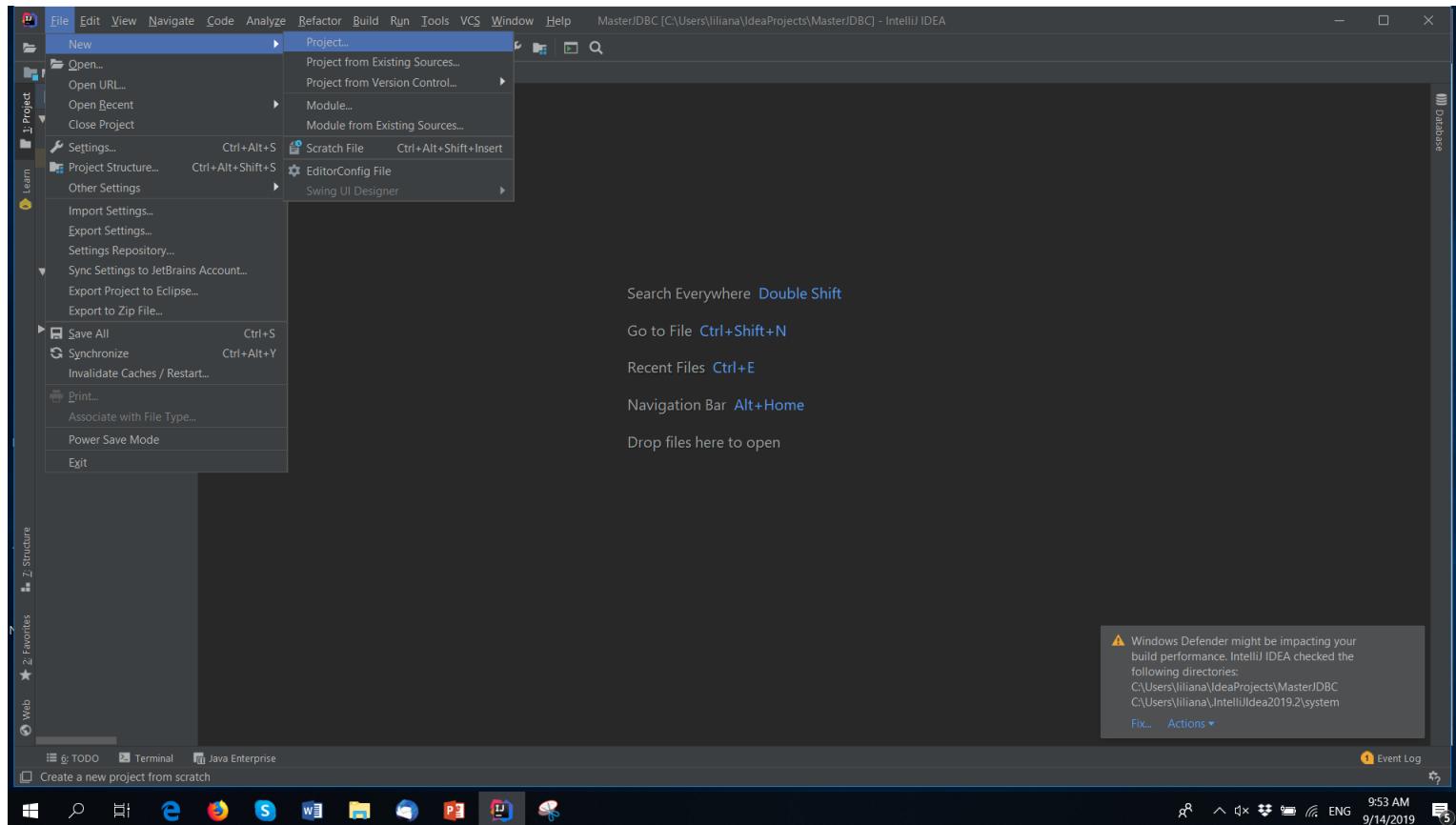
# Progetti IntelliJ

All'interno dell'IDE, ogni applicazione è organizzata come progetto (Project). Un progetto include varie cartelle e file

- **Source Packages (src)** contiene il codice sorgente (classi java)
- **External Libraries** contiene le librerie utilizzate (JDK etc.)
- Altre cartelle ...

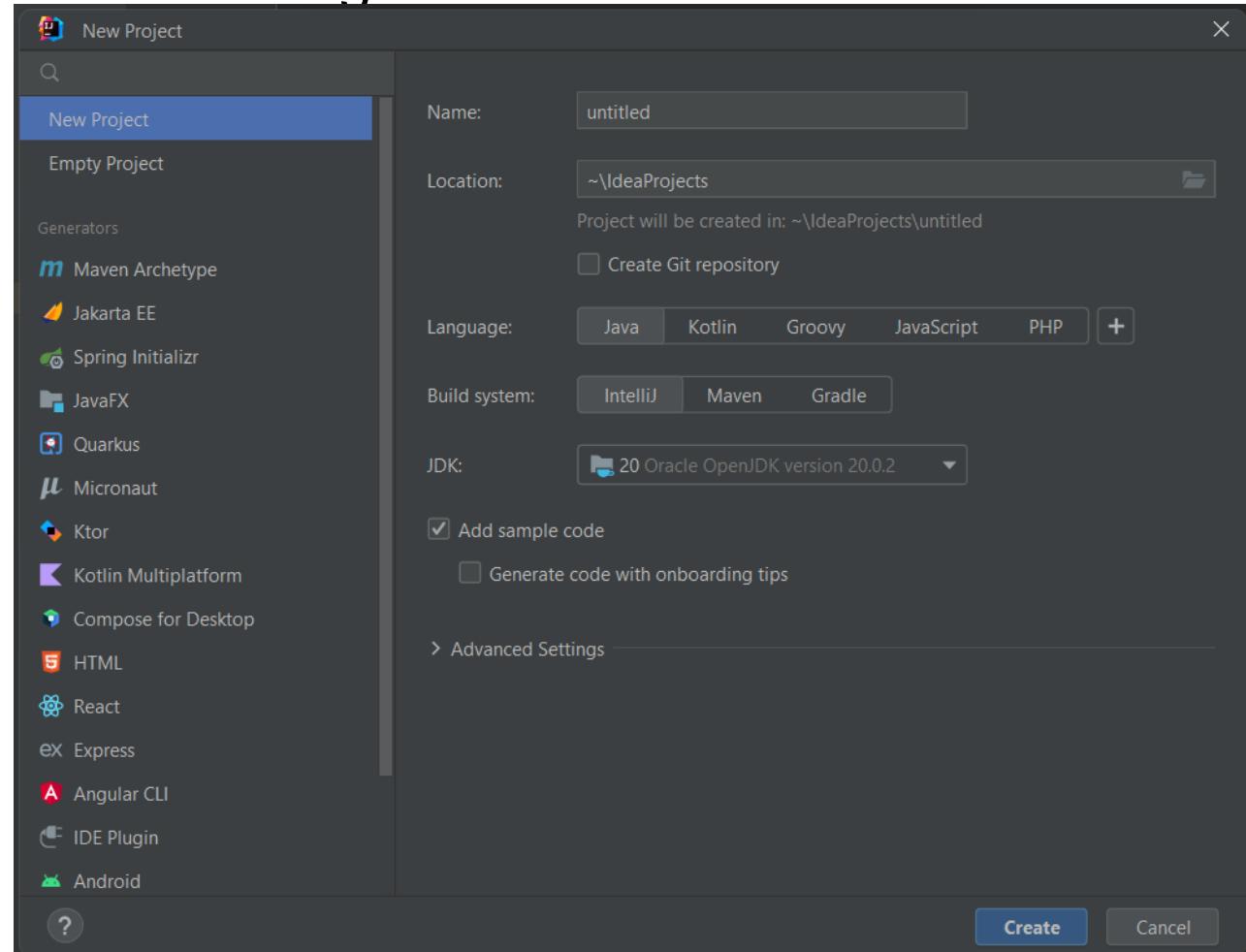
# Come creare un progetto java - I

Aprite il menu File → New → Project...



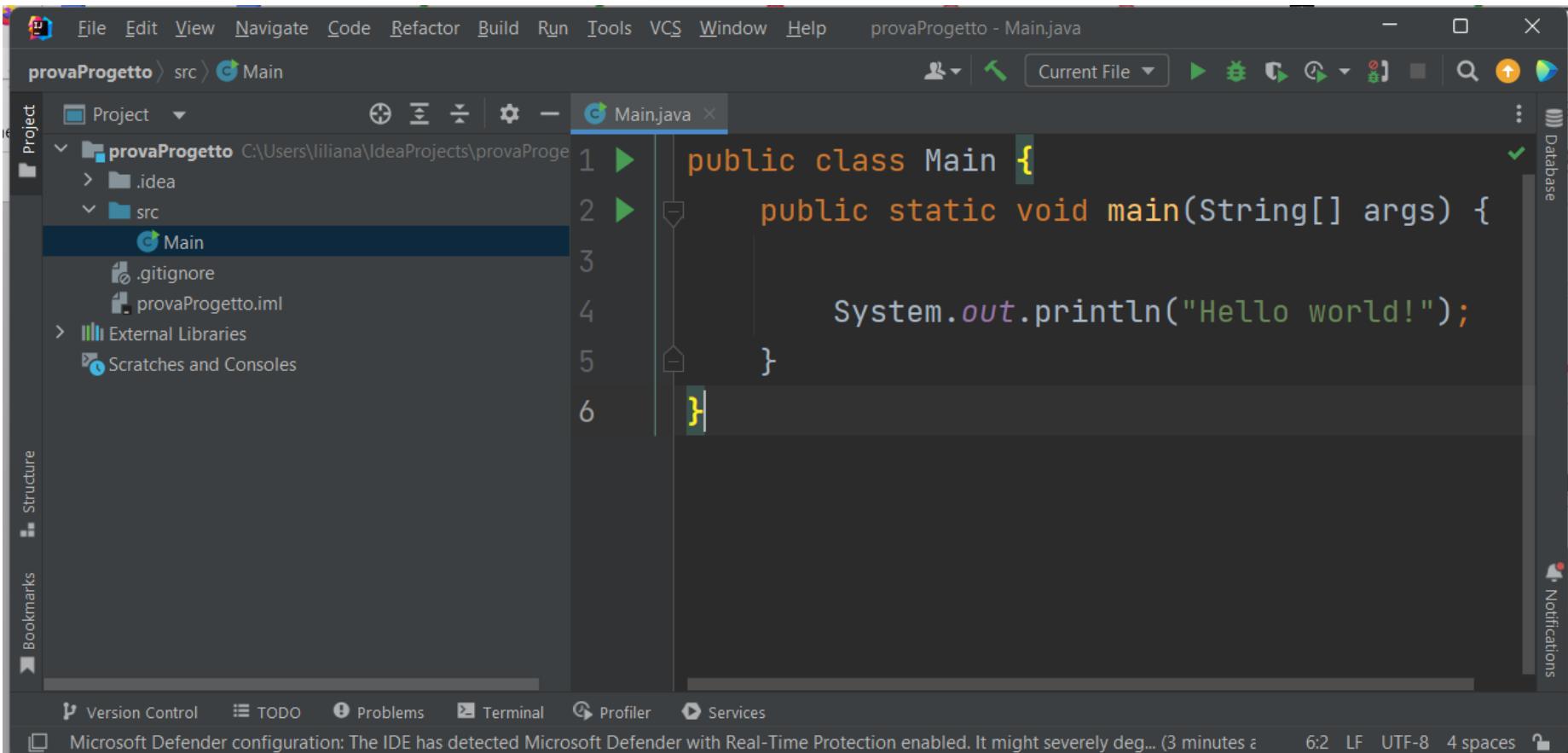
# Come creare un progetto java - II

- Nel pop-up, selezionare New Project, con Language: Java
- Definire il nome del Progetto
- → Create



# Nasce il progetto base - I

In src si sviluppa il codice java (partendo da Main.java)



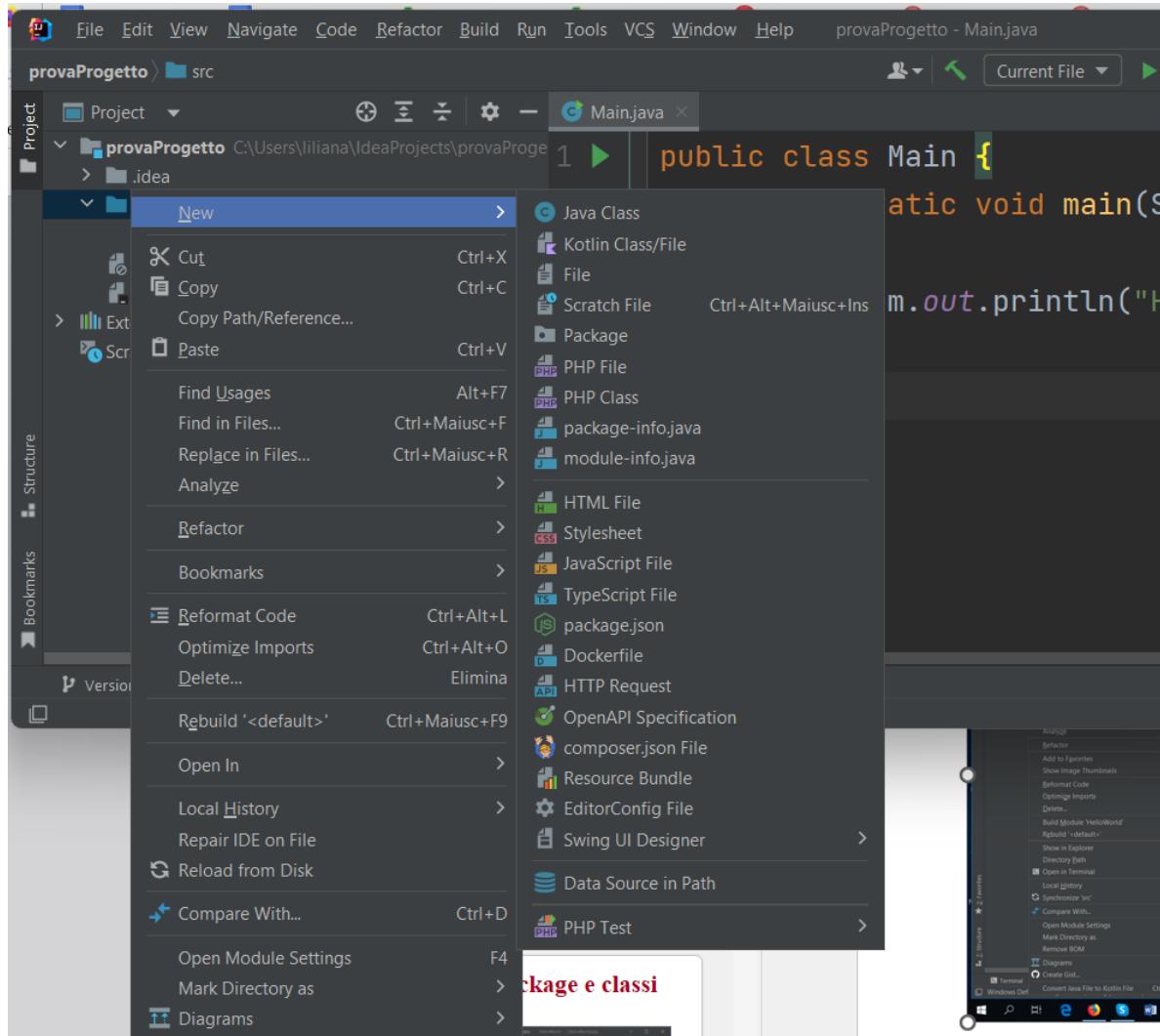
The screenshot shows the IntelliJ IDEA interface with the following details:

- File Menu:** File, Edit, View, Navigate, Code, Refactor, Build, Run, Tools, VCS, Window, Help.
- Title Bar:** provaProgetto - Main.java
- Toolbar:** Includes icons for file operations like New, Open, Save, and Run.
- Project Tool Window:** Shows the project structure with a tree view of files and folders. The 'src' folder contains a 'Main.java' file, which is currently selected and highlighted in blue.
- Code Editor:** Displays the Java code for 'Main.java':

```
public class Main {
 public static void main(String[] args) {
 System.out.println("Hello world!");
 }
}
```
- Toolbars and Panels:** Includes 'Database', 'Structure', 'Bookmarks', and 'Notifications' panels.
- Bottom Status Bar:** Shows 'Version Control', 'TODO', 'Problems', 'Terminal', 'Profiler', 'Services', and a note about Microsoft Defender configuration.
- Bottom Right:** Shows file statistics: 6:2 LF, UTF-8, 4 spaces, and a refresh icon.

# Nasce il progetto vuoto - II

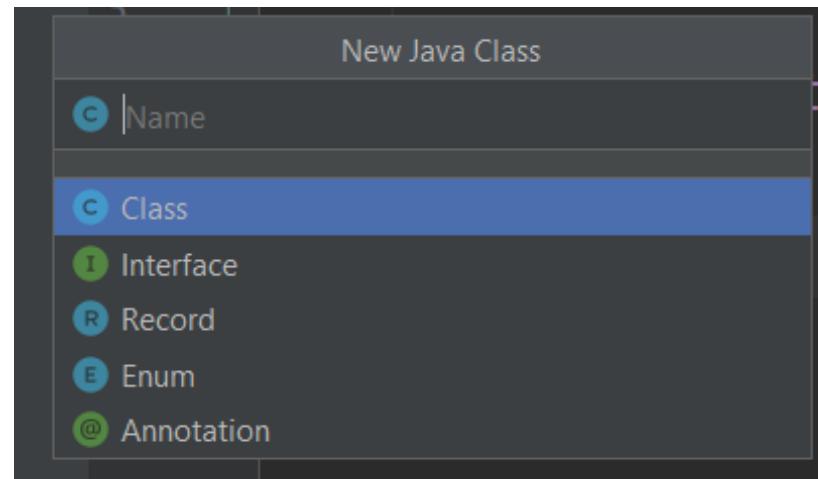
Creazione di package e classi: schiacciate il tasto destro del



mouse puntato  
sul package  
all'interno del quale  
volette creare il  
nuovo package o  
la nuova classe.

# Creazione di classi

- Selezionando New → Java Class si apre il pop-up per scegliere se creare una classe, interface, o altro. Scegliere Class e impostare il nome della classe (per es., Prodotto).



# Nasce la classe Prodotto

- Inserite il codice della classe (variabili, metodi, etc.)

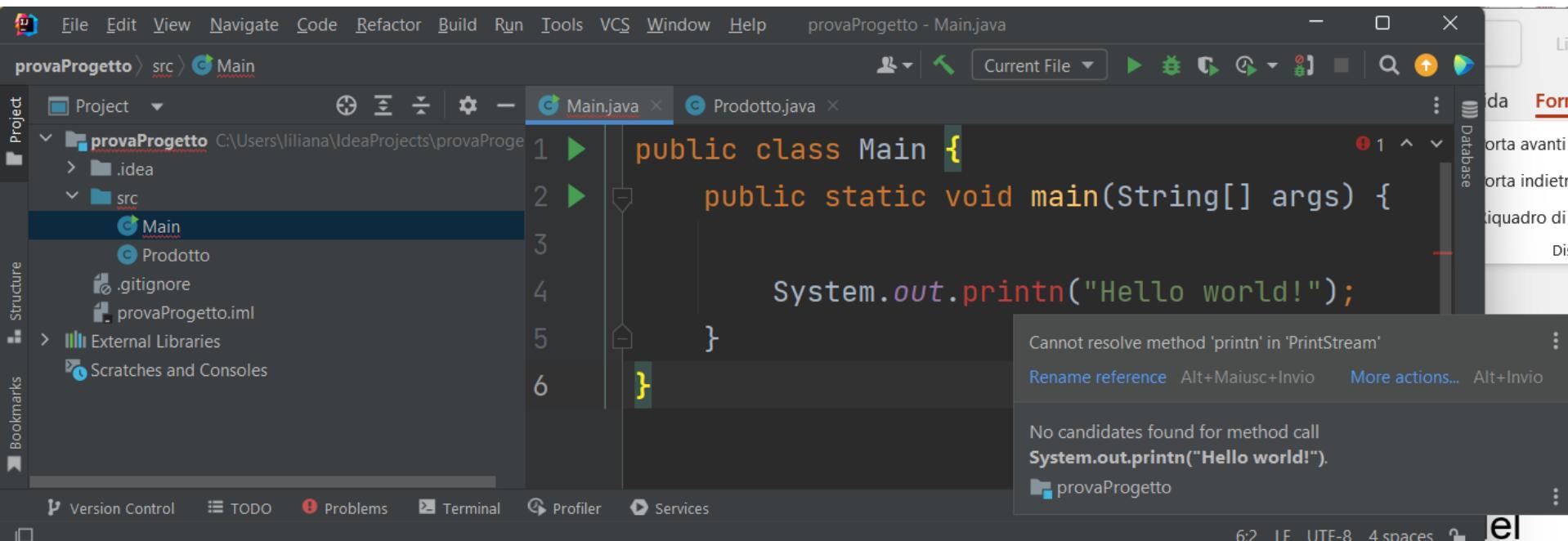
The screenshot shows the IntelliJ IDEA interface with the following details:

- File Bar:** File, Edit, View, Navigate, Code, Refactor, Build, Run, Tools, VCS, Window, Help, provaProgetto.
- Title Bar:** Current File: Prodotto.java
- Project Tool Window:** Shows the project structure under "provaProgetto". The "src" folder contains "Main.java" and "Prodotto.java", which is currently selected and highlighted in blue.
- Code Editor:** Displays the code for "Prodotto.java":

```
public class Prodotto { }
```
- Toolbars and Panels:** Includes Database, Notifications, Version Control, TODO, Problems, Terminal, Profiler, Services, and status bar showing 3:1 CRLF, UTF-8, 4 spaces.

# Compilazione automatica del codice delle classi

Con mouse-over si visualizza il messaggio del compilatore



The screenshot shows the IntelliJ IDEA interface with a Java file named Main.java open. The code contains a simple main method:`public class Main {  
 public static void main(String[] args) {  
 System.out.println("Hello world!");  
 }  
}`

A tooltip is displayed over the `println` method call, providing compiler feedback:

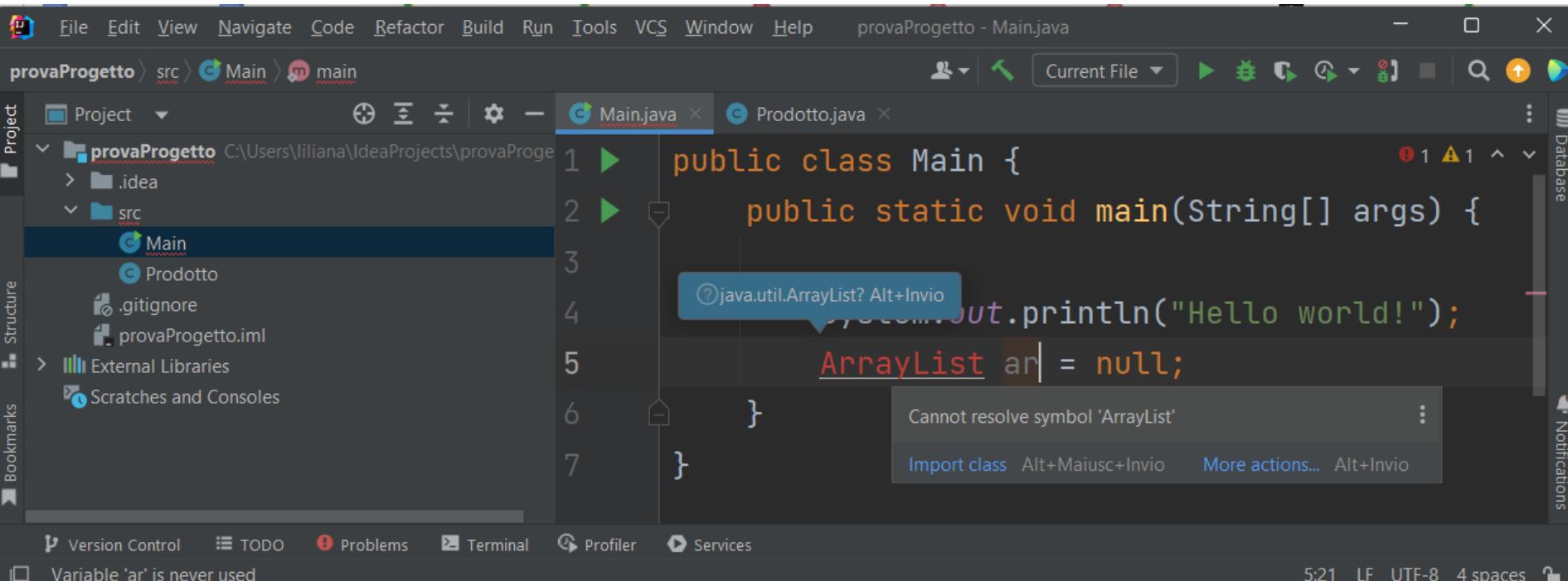
- Cannot resolve method 'println' in 'PrintStream'
- Rename reference Alt+Maiusc+Invio
- More actions... Alt+Invio

The tooltip also indicates that no candidates were found for the method call.

# Autocompletamento e supporto

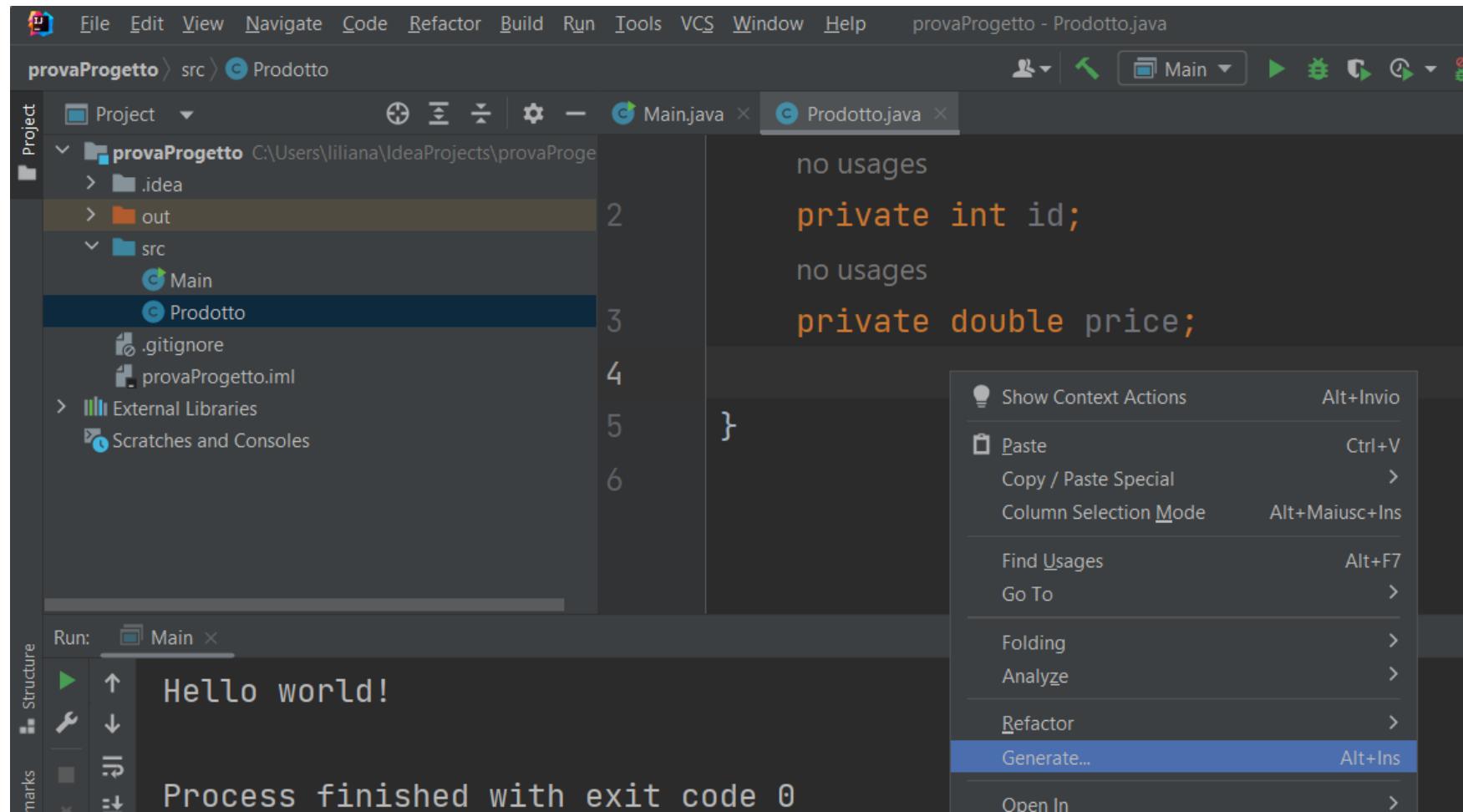
L'IDE offre funzioni di aiuto come autocompletamento, import di classi (vd. Figura), indentazione, etc.

Con mouse over `ArrayList` appare il suggerimento Alt+Invio per importare la classe (`java.util.ArrayList`)



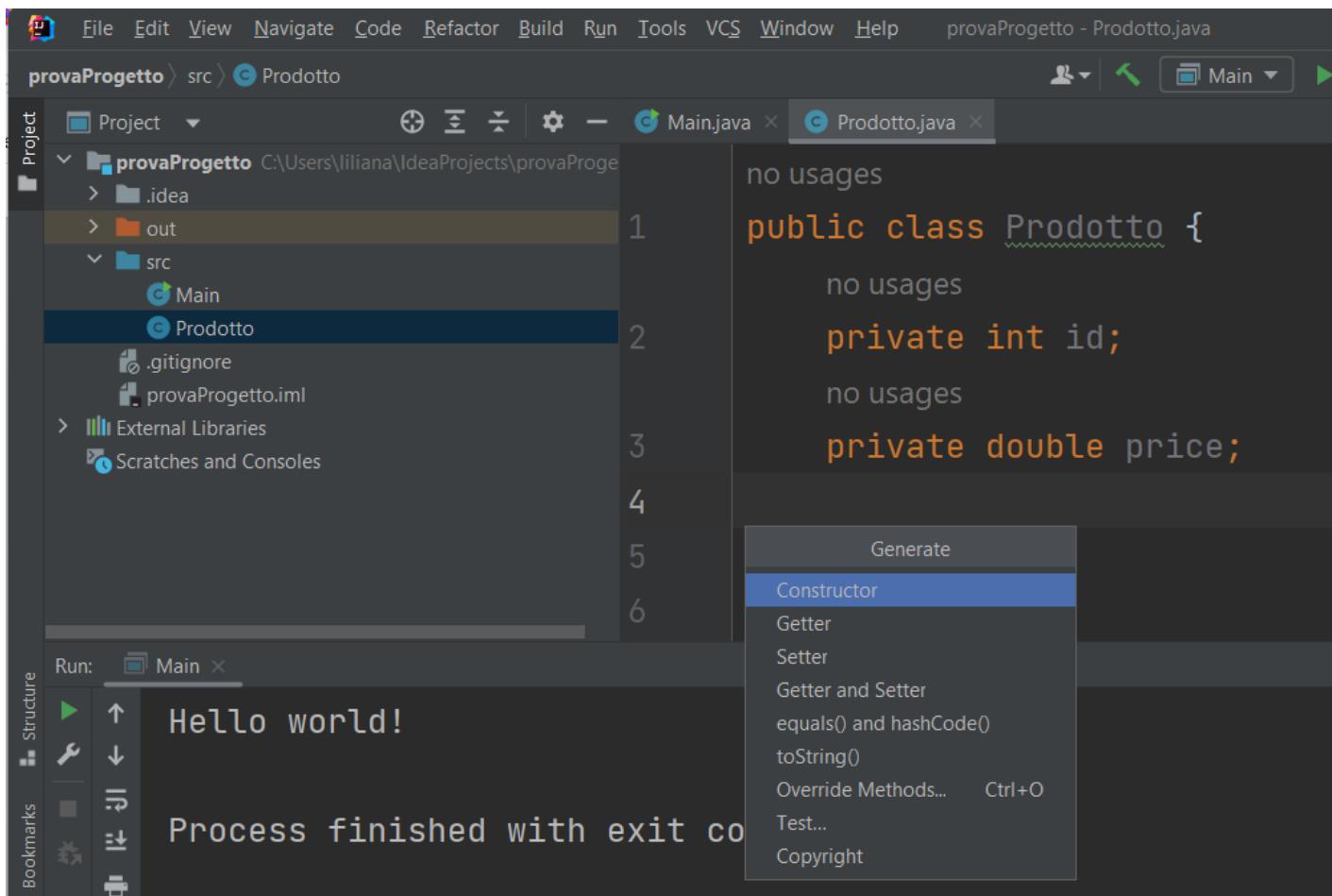
# Autocompletamento: costruttore, getter, setter, etc.

L'autocompletamento permette di creare i costruttori delle classi, i metodi di get() e set(), etc.



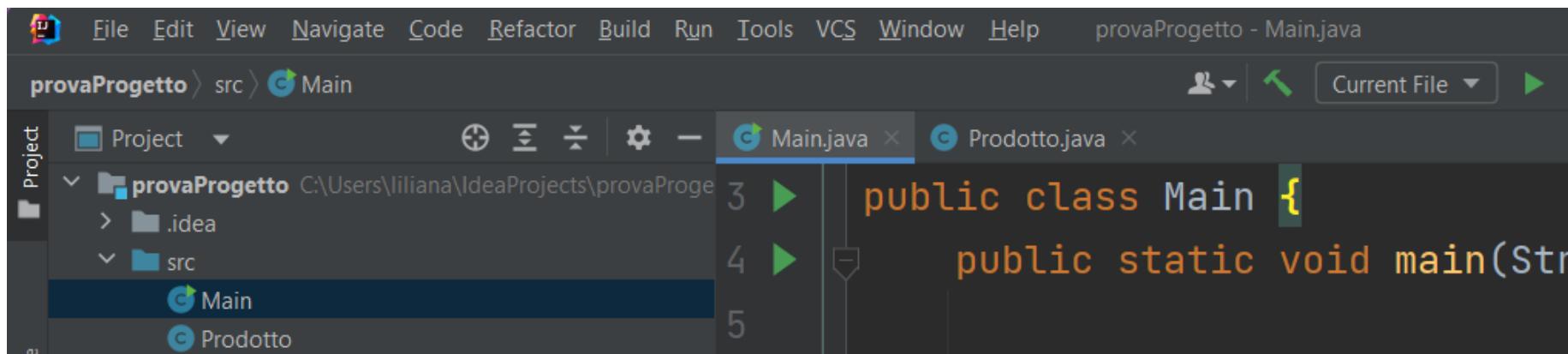
# Autocompletamento: costruttore, getter, setter, etc.

Cliccando su Generate voi potete creare i metodi:



# Compilazione dell'applicazione

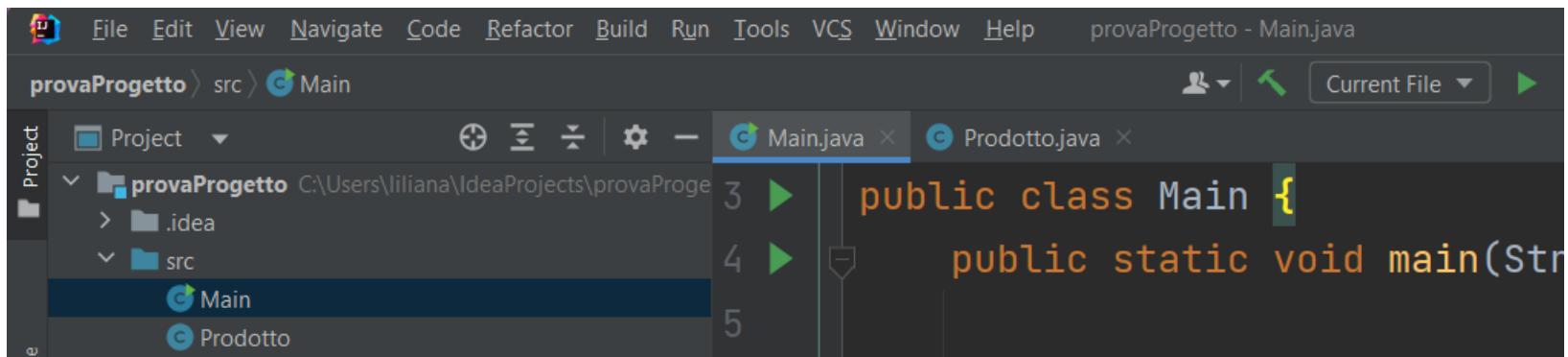
La compilazione è automatica ma se la si vuole far eseguire forzatamente → eseguire la *build* schiacciando il martello verde



# Esecuzione dell'applicazione - I

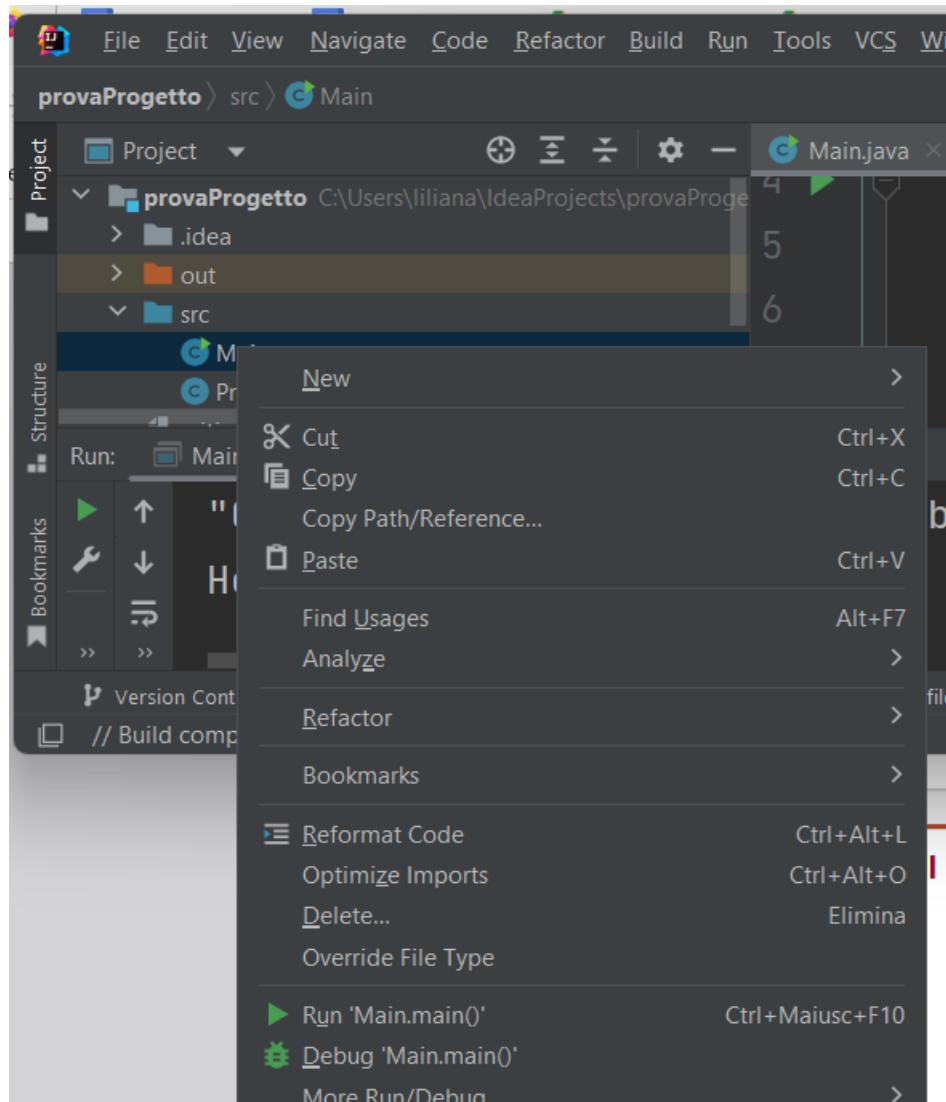
Per eseguire il progetto (Run): schiacciare il triangolo verde → si apre un'area nella parte bassa in cui si vede l'output standard dell'applicazione (vedere i prossimi lucidi).

Se il triangolo non è verde, potrebbero esserci problemi di compilazione oppure state cercando di eseguire una classe priva del metodo main().

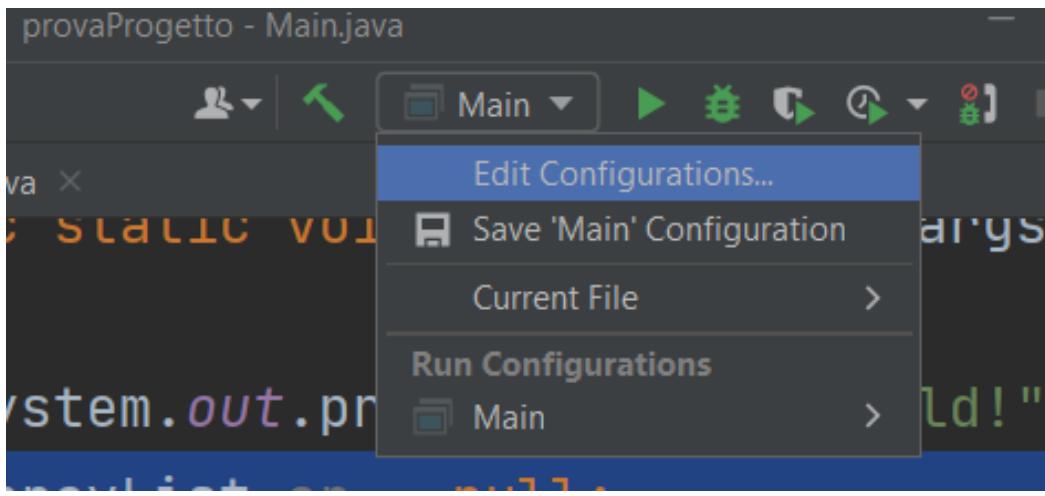


# Esecuzione dell'applicazione - II

Per eseguire una applicazione voi potete anche schiacciare il tasto destro del mouse puntando sulla classe da eseguire. Si apre un menu; scegliete la voce Run (o Debug se volete eseguirla con il debugger).



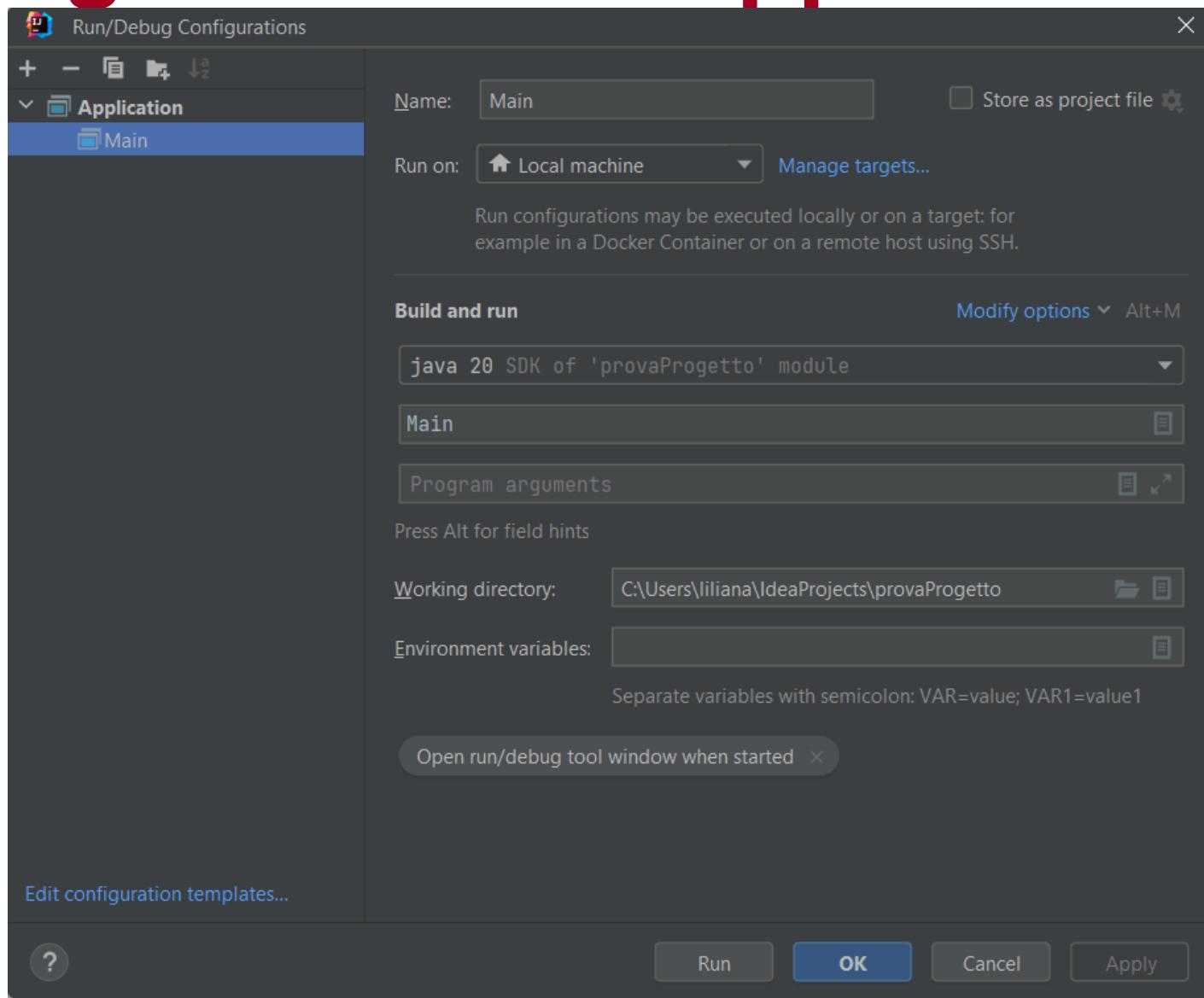
# Configurazione dell'applicazione - I



Ogni applicazione ha un file di configurazione che specifica le librerie usate, etc.

Voi potete aprire questo file, e modificarlo se necessario, con Edit configurations

# Configurazione di applicazione-II



# Gestione dell'applicazione

The screenshot shows the IntelliJ IDEA interface with a Java application running. The main window displays the code of the `Main.java` file, which contains a simple "Hello world!" program. The output window shows the application's output and exit status.

**Code View:**

```
2
3 ► public class Main {
4 ► public static void main(String[] args) {
5
6 System.out.println("Hello world!");
7
8 }
9 }
```

**Output Window:**

```
Hello world!
Process finished with exit code 0
```

**Bottom Status Bar:**

Version Control Run TODO Problems Terminal Profiler Services Build  
All files are up-to-date (6 minutes ago) 4:26 LF UTF-8 4 spaces

# Estendiamo il codice dell'applicazione

- Completate il codice dell'applicazione creando la classe Persona.java etc. etc.
- L'editor segnala parecchi tipi di errore sintattico durante l'editing. Io consiglio di controllare sempre tali segnalazioni e di correggere il codice via via. Non accumulate troppi errori senza correggere!
- Familiarizzate con l'IDE, per scoprire il supporto allo sviluppo di software che offre.

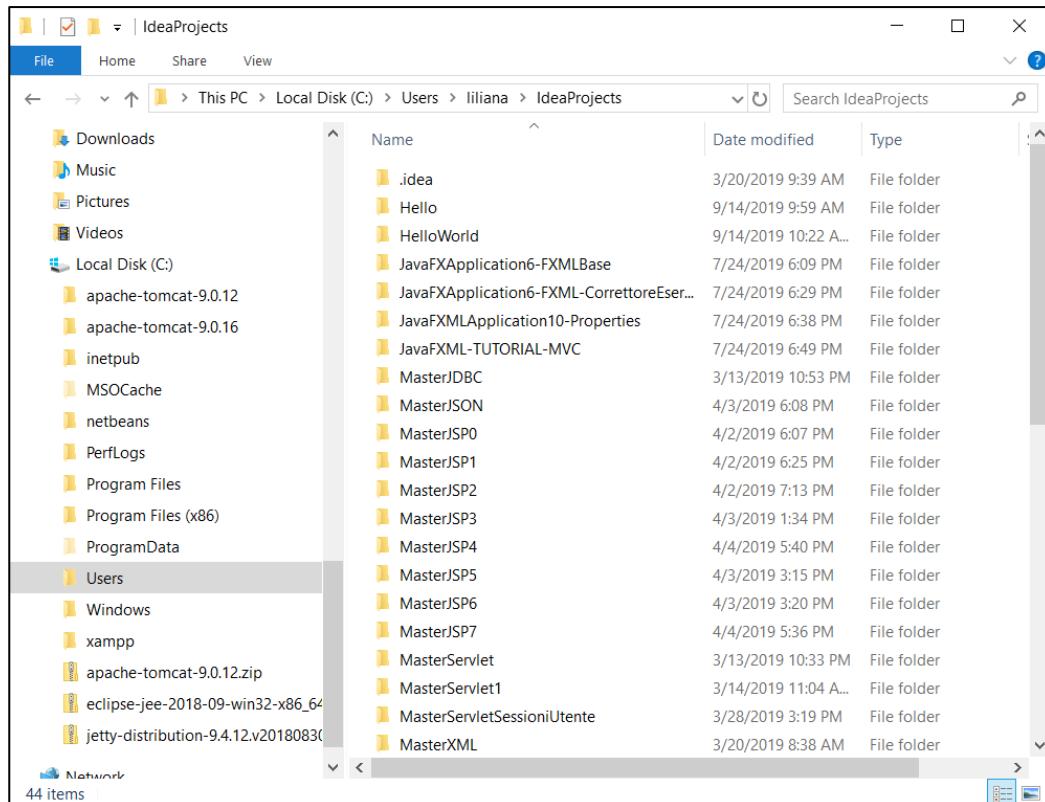
# Progetti multipli

- Usando l'IDE potete creare diverse applicazioni, ognuna inserita in un Progetto separato.
- L'IDE vi permette di visualizzare i nuovi progetti in finestre nuove o di usarne solo una. A voi la scelta.

Se volete racchiudere in una sola applicazione esercizi diversi (per esempio, gli esempi di programmazione III), usate i package per dividerli. In questo modo non avrete problemi di duplicazione dei nomi delle classi.

# Importazione/esportazione di progetti - I

IntelliJ IDEA salva i progetti in una cartella IdeaProjects. Ogni progetto è una cartella separata.



# Importazione/esportazione di progetti - II

- Per esportare un progetto, salvare su pen drive o altro la cartella del progetto stesso.
- Per importare un progetto: Menu File → Open... → spostarsi nella cartella dei progetti e selezionare quello da aprire.

