

# **CORSO DI LAUREA TRIENNALE IN INFORMATICA (GRUPPO 2)**

## **CORSO DI OBJECT ORIENTATION**

**Docente: Prof. Porfirio Tramontana**

*Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione (DIETI)*

*Via Claudio 21, stanza 4.06*

*Università degli Studi di Napoli Federico II*

**e-mail:** [porfirio.tramontana@unina.it](mailto:porfirio.tramontana@unina.it)

**Website:** <https://sites.google.com/view/porfiriotramontana/home-page>

**tel. 081 768 3901**



# **CONTENUTI DEL CORSO**

- La programmazione orientata agli oggetti
- Il Linguaggio Java
- Realizzazione di interfacce grafiche ad eventi.
- Metodologie di Sviluppo



# I PARADIGMI DI PROGRAMMAZIONE

- Forniscono la filosofia e la metodologia con cui si scrivono i programmi
- I linguaggi devono *consentire* ma soprattutto *spingere* all'adozione di un particolare paradigma
  - Procedurale
  - Funzionale
  - Dichiarativo
  - Orientato agli oggetti



# **PARADIGMA IMPERATIVO/PROCEDURALE**

- Enfasi sulla soluzione algoritmica dei problemi mediante modifica progressiva dei dati in memoria
  - Esecuzione sequenziale di istruzioni
  - Cambiamento dello stato di memoria (le variabili) tramite assegnamento
- Aderenti al modello della macchina di von Neumann
- Molto efficienti
- Ha mostrato limiti nello sviluppo e manutenzione di sw complessi
- I linguaggi imperativi: Pascal, C



# **PARADIGMA DICHIARATIVO**

- Specificare le caratteristiche della soluzione con dichiarazioni che le descrivono piuttosto che il procedimento per ottenerla
  - Il programma viene eseguito da un risolutore di problemi che determina la strategia migliore
  - Paradigma utilizzabile in campi specifici
- Esempi:SQL
  - Recupera dal database tutti gli studenti con matricola che inizia per «N86»



# PARADIGMA FUNZIONALE

- La computazione avviene tramite funzioni che, applicate ai dati, riportano nuovi valori
  - Ogni funzione è un modulo a sé dipendente unicamente dal valore dei suoi argomenti
  - L'effetto globale è ottenuto concatenando opportunamente funzioni anche richiamando sé stesse (ricorsione)
  - Modello che si rifà alla teoria delle funzioni ricorsive
  - Scarso supporto ai costrutti di ripetizione tramite iterazione
  - Di recente utilizzato anche nel contesto di sistemi ad eventi (ad esempio interfacce grafiche)
- Lisp, ML, Scala



# PARADIGMA A OGGETTI

- Evoluzione dell'imperativo/procedurale, per facilitare la programmazione.
- Pone al centro delle attività la *modellazione* del dominio del problema e successivamente la sua soluzione, definendo gli oggetti coinvolti e raggruppandoli in *classi*
- Ogni classe avrà delle proprie *responsabilità* alle quali corrispondono funzioni che andranno risolte con *metodi* che implementaranno *algoritmi*
- Gli *algoritmi* non sono più il centro del processo di sviluppo ma solo una sua parte
- Obiettivo: migliorare l'efficienza del processo di produzione e mantenimento del software, rendendolo adatto allo sviluppo di applicazioni di medie e grandi dimensioni



# **CONTENUTI DEL CORSO**

- La programmazione orientata agli oggetti
- Il Linguaggio Java
- Realizzazione di interfacce grafiche ad eventi.
- Metodologie di Sviluppo



# CONTENUTI DEL CORSO

- La programmazione orientata agli oggetti
  - Concetti di astrazione dei dati, di definizione di tipi personalizzati, e di encapsulamento
  - Oggetti, variabili, classi e metodi
  - Costruttori
  - Ereditarietà e polimorfismo
  - Overloading e Overriding
  - Le interface e loro benefici.
  - This e Super
- Il Linguaggio Java
- Realizzazione di interfacce grafiche ad eventi.
- Metodologie di Sviluppo



# CONTENUTI DEL CORSO

- La programmazione orientata agli oggetti
- Il Linguaggio Java
  - JVM.
  - JDK
  - Tipi Primitivi e Riferimenti in Java.
  - Garbage Collector
  - Le collezioni in Java.
  - Le classi Wrapper in Java. Autoboxing e Unboxing
  - Le Eccezioni in Java.
- Realizzazione di interfacce grafiche ad eventi.
- Metodologie di Sviluppo



# CONTENUTI DEL CORSO

- La programmazione orientata agli oggetti
- Il Linguaggio Java
- Realizzazione di interfacce grafiche ad eventi con AWT e Swing.
  - Programmazione event-driven
  - Differenti tipi di Finestre, pannelli e Component.
  - Architetture per applicazioni con GUI: pattern Boundary – Control – Entity (BCE)
  - Il concetto di classe Controller.
- Metodologie di Sviluppo



# CONTENUTI DEL CORSO

- La programmazione orientata agli oggetti
- Il Linguaggio Java
- Realizzazione di interfacce grafiche ad eventi.
- Metodologie di Sviluppo
  - UML: Class Diagrams e Sequence Diagrams.
    - Corrispondenze con codice sorgente.
  - Gli ambienti di Sviluppo Integrati (IDE): Eclipse
  - I Sistemi di Version Control.
    - GIT
  - Le CRC Cards
  - Pattern DAO



# OBIETTIVI DEL CORSO DI OBJECT ORIENTATION

- Conoscenze che si intendono trasmettere (sapere):
  - Acquisizione delle competenze di base per la progettazione object-oriented attraverso la comprensione dei concetti di astrazione sui dati, di encapsulamento dell'informazione, di coesione e accoppiamento, e di riutilizzo del codice
  - Comprensione delle differenze tra paradigma object-oriented e il paradigma procedurale
  - Conoscenza del linguaggio Java per una buona progettazione object-oriented e per la promozione del riutilizzo del software



# CAPACITÀ APPRESE A VALLE DEL CORSO

- Capacità che si intendono sviluppare (*saper fare*):
  - Definizione di una strategia risolutiva con un approccio orientato agli oggetti, con la sua implementazione nel linguaggio Java, garantendo il giusto equilibrio tra qualità ed efficienza del software
  - Sviluppo di progetti con attività individuali e di gruppo

# ORGANIZZAZIONE DEL CORSO

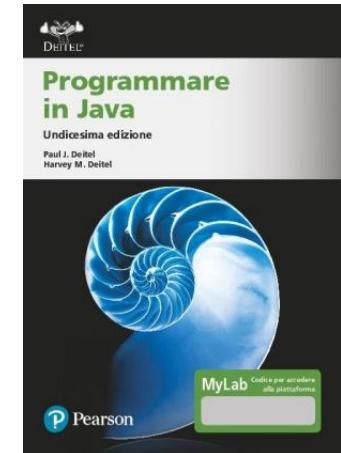
- ~24 lezioni:
  - Generalmente in presenza, trasmesse e videoregistrate anche su Teams
    - Le modalità di accesso/prenotazione delle aule sono quelli stabiliti per tutto l'Ateneo
  - Esercitazioni su carta o con i portatili, eventualmente a distanza
- Orario
  - Mar 08:30 – 10:30 - (posticipato verso 08:45 – 10:45)
  - Gio 16:30 – 18:30 - (anticipato a 16:05 – 18:00)
- Ricevimento studenti:
  - Durante il I semestre, Lunedì 09.30-11.30 nello studio del docente o in caso di necessità su Teams
  - Dopo la fine del semestre, vedere le indicazioni sul sito del docente
    - Nei mesi invernali 2022, martedì mattina, ore 9:30



# MATERIALE DIDATTICO

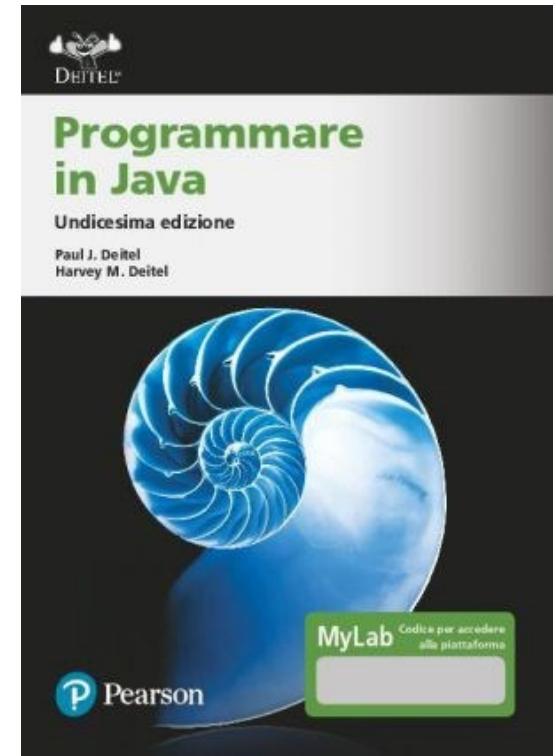
Libri di testo :

- Paul J. Deitel - Harvey M. Deitel: Programmare in Java, 11/Ed., Pearson
- Martin Fowler: UML Distilled, 4/Ed., Pearson



# TESTO DI RIFERIMENTO PER JAVA

- Paul J. Deitel - Harvey M. Deitel: **Programmare in Java**, 11/Ed., Pearson
  - Capitolo 1, sezioni 1.8, 1.9, 1.10
  - Capitolo 2, sezioni da 2.1 a 2.9
  - Capitolo 3 (tranne 3.6)
  - Capitolo 4 (tranne 4.15),
  - Capitolo 5 (tranne 5.11)
  - Capitolo 7, tranne 7.13, 7.17
  - Capitolo 8, sezione 8.4, 8.6, 8.7, 8.8, 8.10, 8.11, 8.13, 8.14
  - Capitolo 9
  - Capitolo 10 da 10.1 a 10.9 e 10.13
  - Capitolo 11, da 11.1 a 11.7 e 11.9
  - Capitolo 14, sezioni 14.1, 14.2, 14.3
  - Capitolo 15, sezioni 15.1, 15.2, 15.4
  - Capitolo 16 da 16.1 a 16.6
- Per chi vuole approfondire la GUI con JavaFX, capitoli 12 e 13



# MATERIALE DIDATTICO

Ulteriori libri consigliati:

- **Parti generali**
  - I. Sommerville. Software Engineering, VIII ed., Addison Wesley, 2007.
- **Progettazione ad oggetti**
  - C. Larman, *Applicare UML e i Pattern - Analisi e Progettazione orientata agli Oggetti*, III ed. Prentice-Hall, 2005.
  - B. Bruegge, A. Dutoit. *Object-Oriented Software Engineering*, Pearson, 2008. (Alternativo a C. Larman).
- **UML**
  - Stevens Rod Pooley, *Usare UML*, Addison Wesley, 2008.
  - J. Arlow, Ila Neustadt, *UML2 e Unified Process*, McGraw-Hill, 2006.
- **Altri testi su aspetti specifici**
  - P. Amman, J. Offutt. *Introduction to software testing*, Cambridge University Press, 2008.
  - E. Gamma, R. Helm, R. Johnson, J. Vissides. *Design patterns*, Addison Wesley



# MATERIALE DIDATTICO

- Strumenti per le esercitazioni:
  - StarUML (o altri equivalenti) per la modellazione UML
    - <http://staruml.sourceforge.net/en/download.php>
- Ambiente Eclipse
  - A supporto sia dello sviluppo che della modellazione
    - <http://www.eclipse.org/downloads/>



# COMUNICAZIONI DOCENTE → STUDENTI

- Sito Istituzionale del docente:  
[www.docenti.unina.it/porfirio.tramontana](http://www.docenti.unina.it/porfirio.tramontana)
- E' richiesta la registrazione al corso su questo sito
  - Consente di ricevere direttamente mail relative agli avvisi più urgenti
- Lo spazio Teams verrà utilizzato per:
  - Lezioni registrate
  - Materiale didattico (generalmente messo in copia anche su docenti)
  - Appunti e annotazioni sul corso



# COMUNICAZIONI STUDENTI → DOCENTE

- Mail :
  - Solo per quesiti brevi!
    - [porfirio.tramontana@unina.it](mailto:porfirio.tramontana@unina.it)
  - Subject: [OO] e poi l'oggetto
  - Utilizzare sempre la mail istituzionale [@studenti.unina.it](mailto:@studenti.unina.it)
  - Firmare SEMPRE le mail
  - Non mandare mail per quesiti su aspetti già descritti nel sito istituzionale e/o nel materiale didattico.
    - Consultare sempre prima il materiale messo a disposizione
- Chat di Teams
  - Solo per comunicazioni urgenti con risposte immediate
- Per quesiti articolati esiste il ricevimento
  - Non è ovviamente ammessa la richiesta di una revisione dell'elaborato d'esame, essendo oggetto della valutazione
- La comunicazione cliente/committente è uno degli aspetti chiave di un professionista dell'informatica!



# MODALITÀ DI ESAME

- **Progetto obbligatorio di gruppo**
  - Progettazione e Implementazione di un piccolo sistema software
  - Presentazione di gruppo al docente del progetto
    - Documentazione + Demo max 10 min.
    - Discussione individuale dell'elaborato
      - (domande poste dal docente ai singoli studenti)
- **Scritto: esercizi e domande aperte sull'intero programma.**
  - Obbligatorio.
- **Voto: Media di Progetto e Scritto**

# PROGETTO

- 4 – 5 tracce, in comune con il corso di Basi di Dati.
- Assegnazione random a gruppo
- Consistenza numerica: 2 o 3 persone (consigliate 2)
  - Allo scopo di iniziare a maturare esperienza di lavoro in team.
- Formazioni dei gruppi
  - Autonome comunicate nel gruppo del corso
    - Fornirò un modulo apposito
  - Operate dal docente in base alle disponibilità per studenti che non riescano a stabilire formazioni autonome.
- Variazione dei gruppi
  - Ogni variazione di un gruppo ufficializzato deve essere concordata con il docente
- Consistenza con basi di dati
  - Nel caso il gruppo si separi dopo l'esame di basi di dati, gli studenti procedono autonomamente

# PROGETTO E PROVA SCRITTA

- I legami di Gruppo terminano con la presentazione dell'elaborato
  - La prova scritta è individuale e non deve essere necessariamente sostenuta contemporaneamente da tutti i membri del Gruppo
- La demo dell'elaborato con la discussione e le domande del docente deve essere sostenuta da tutto il Gruppo contemporaneamente
  - Nel caso in cui un componente non fosse pronto, deve comunicarmi esplicitamente (via mail o chat) la sua rinuncia all'appartenenza al Gruppo
    - Per evitare che un Gruppo si sciolga all'insaputa del componente "ritardatario"
    - Il componente rimasto isolato può riaggregarsi con altri componenti isolati



# VALUTAZIONE DEL PROGETTO

- Qualità degli artefatti e della demo.
  - Ogni martedì mattina per tutto il periodo fino all'inizio dei corsi del secondo semestre (inverno 2022)
    - Previa prenotazione via mail con invio del progetto
- **Valutazione dell'intera interazione committente-contraente.**
  - Interazione e uso degli strumenti di comunicazione.
  - Qualità grafica dei documenti prodotti.
  - Qualità della presentazione finale.
- Il progetto può essere consegnato fino a Ottobre 2022.

# CHEATING POLICY

- Viene usato uno strumento automatico di Cheating Detection per il confronto di tutto ciò che viene consegnato.
- In caso di due o più progetti siano ritenuti troppo simili, ad insindacabile giudizio dei docenti, il progetto più recente (e entrambi i progetti se consegnati nello stesso appello) sarà/saranno annullato/i e sarà/saranno data una nuova traccia, più estesa e complessa di quella originaria al/ai gruppi che hanno prodotto questo/i elaborato/i



# **PARADIGMA OBJECT ORIENTED**



# PARADIGMI DI PROGRAMMAZIONE

- procedurale (Pascal, C)
- a oggetti (C++, Java)
- funzionale (LISP, Scala)
- logico (Prolog)



# PARADIGMA PROCEDURALE

- E' stato trattato in maniera esclusiva nel primo corso di Informatica
- Sia la definizione del Problema che della Soluzione sono incentrate su di una singola elaborazione da risolvere
  - Il risultato è una *procedura* o *funzione* che implementa un *algoritmo*
- Non vengono invece curati molto gli aspetti di
  - Modellazione dei dati
  - Raggruppamento delle funzioni
  - ...



# ESEMPIO PROCEDURALE

```
int voto[20];

void sort(int [] v, int size) { // sort };

int search(int [] v, int size, int c) { // search };

// ...

int i;

void main(){
    for (i=0; i<20; i++) { // voto[i ]=0;};
    sort(voto, 20);
    search(voto, 20, 33);

}
```

- L'analisi è stata centrata sul *problema*, che è stato risolto scomponendolo in due *funzioni*
- Per ogni funzione sono stati individuati gli *input* e gli *output*
- Sono state progettate le chiamate (*call*) alle funzioni e il loro ordine
- (per l'inizializzazione del vettore non è stata utilizzata una funzione)



# POTENZIALI PROBLEMI

- Concettualmente abbiamo scritto un algoritmo che opera su un vettore di voti ma in pratica non abbiamo definito un *tipo* vettore o un *tipo voto* riutilizzabile in altri problemi
  - Non c'è un legame tra le funzioni che operano sul vettore e la variabile *voto*
  - Non viene valutato il raggiungimento del limite della dimensione del vettore
  - L'inizializzazione del vettore è fatta dal main



# POTENZIALI PROBLEMI

- Se andassimo ad espandere il nostro programma, potremmo inserire funzioni che vanno a modificare senza controllo il vettore
- Quante più funzioni vanno a toccare (leggere / scrivere) le stesse variabili, tanto più faticoso può essere il debugging del codice in caso di rilevazione di un fallimento



# TIPO DI DATO ASTRATTO

- Il concetto di Tipo di Dato Astratto (**ADT**) viene introdotto già nella programmazione procedurale
- Caratteristiche fondamentali:
  - Astrazione
  - Modularità
  - Incapsulamento
  - Information Hiding



# ASTRAZIONE

- Si vogliono raggruppare alcune entità che:
  - Corrispondano a concetti astratti oppure ad elementi del mondo reale
  - Abbiano delle proprietà loro intrinseche
  - Sui quali ha senso definire delle operazioni ben precise
- Il processo di *raggruppare* tutti questi elementi sotto un unico concetto si chiama **astrazione**



# ESEMPI DI ASTRAZIONE

- Rispetto al codice precedente, può essere astratto il concetto di **elenco di voti**
- L'**elenco di voti** ha:
  - Un concetto astratto di riferimento
    - Corrispondente ad un elenco di voti
  - Delle proprietà specifiche
    - La dimensione (20 elementi), il tipo degli elementi, la memorizzazione sotto forma di array, i valori degli elementi
  - Un insieme di operazioni applicabili su di esso
    - L'inizializzazione
    - La ricerca di un valore
    - L'ordinamento



# ESEMPI DI ASTRAZIONE

- Per i sistemi e servizi dell'università una astrazione utile può essere quella di **studente Universitario**
- Il concetto di riferimento è chiaramente la persona che si iscrive per studiare all'università
- Tra le proprietà specifiche ci possono essere:
  - I dati anagrafici, il corso di laurea, il piano di studi, gli esami sostenuti, le tasse da pagare, le borse di studio, i sondaggi effettuati, ...
- Tra le operazioni effettuabili
  - L'iscrizione, la rinuncia, la registrazione di un esame, il pagamento delle tasse, la partecipazione a un concorso, ...



# MODULARITA'

- Negli algoritmi, il principio *divide-et-impera* consente di ricondurre la soluzione di un problema a quella di sottoproblemi più piccoli ad esso collegato
- Nell'Object Oriented è opportuno modellare un concetto complesso tramite concetti più semplici
  - Ma rispettando il principio di *astrazione*



# ESEMPIO DI MODULARITA'

- Nel caso degli **studenti universitari** conviene distinguere ad esempio tra:
  - Gli studenti
  - I corsi universitari
  - ...
- I dati anagrafici sono informazioni peculiari di uno studente
- Il nome di un corso Universitario, il suo programma, il nome del docente sono invece proprietà del corso, indipendentemente dall'esistenza degli studenti



# PARADIGMA OBJECT ORIENTED

- Il paradigma Object Oriented si basa sulla possibilità di risolvere un problema cominciando con la *modellazione* degli elementi del problema sotto forma di **classi** e **oggetti**



# OGGETTO (COME ASTRAZIONE)

- Un **oggetto** nel paradigma object oriented rappresenta un elemento ben distinto e distinguibile nell'ambito della descrizione di un problema o della sua soluzione
- Un oggetto ha un insieme di informazioni associate (detti **attributi** o **proprietà**)
- Un oggetti ha un suo comportamenti (*behaviour*) che definisce come agisce o reagisce a seguito di sollecitazioni



# CLASSE (COME ASTRAZIONE)

- **Una Classe :**

- Rappresenta l'astrazione di un concetto del mondo reale oppure di un concetto che fa parte del problema da risolvere
- Può essere *istanziata*, cioè può essere *costruito* un oggetto (*istanza*) della classe

- Definendo la *classe* si definiscono anche:

- Gli *attributi*, cioè le informazioni che dovranno essere valorizzate per ognuno degli oggetti
  - Le *operazioni* (o *metodi*), cioè le operazioni che possono essere eseguite sugli oggetti della classe in modo che l'oggetto segue il suo *comportamento* (*behaviour*)
  - Diversi oggetti possono comunicare tra loro leggendo/scrivendo valori agli attributi oppure chiamando i metodi di altri oggetti
- 
- Una classe nasce per poter essere **riusabile** in altri problemi nei quali gli stessi oggetti e comportamenti sono richiesti



# ESEMPI DI CLASSI

- **Studente**
  - Attributi: nome, cognome, numero di matricola, ...
  - Operazioni : iscrizione, rinuncia, laurea, ...
- **Esame**
  - Attributi: Nome della materia, nome del docente, lista degli appelli ...
  - Operazioni: registra, aggiungi appello, ..
- **Corso di Laurea**
  - Attributi: Nome, universita', ...
  - Operazioni: aggiungi nuovo corso, conta numero di iscritti, ...
- **Esame sostenuto**
  - Attributi: voto, *Esame*, *Docente*
  - Operazioni: aggiungi esame, annulla esame, fissa voto
- ...



# ESEMPI DI OGGETTI

- **Porfirio : Studente**
  - Nome = ‘Porfirio’, cognome = ‘Tramontana’
- **Object Orientation : Esame**
  - Nome della materia = ‘Object Orientation’, nome del docente = ‘Di Martino’, canale = ‘A-G’
- **Informatica : Corso di Laurea**
  - Nome = ‘Informatica’, universita’ = ‘Federico II’
- **Esame#1 : Esame sostenuto**
  - Voto = 30, *Esame* = *Object Orientation*, *Studente* = Porfirio
  - ...



# INCAPSULAMENTO E INFORMATION HIDING

- Un principio fondamentale è quello di limitare l'accesso / visibilità alle informazioni al minimo indispensabile
- Perchè rendere le informazioni invisibili di default?
  - All'aumentare del numero di informazioni visibili
    - Aumenta la complessità della progettazione
    - Aumenta il rischio di commettere errori di programmazione
    - E' più faticoso scoprire le cause dei fallimenti
    - E' più difficile correggere i difetti
    - E' più difficile manutenere i sistemi (ad esempio evolverli, fonderli, integrarli)
- L'insieme di informazioni (dati, operazioni) che un componente mostra all'esterno è detto **interfaccia**



# ESEMPI DI INTERFACCIA

- Il Sistema segrepass mostra agli studenti i propri dati anagrafici ma
  - non mostra il proprio identificativo nel database
  - non mostra ad uno studente l'elenco degli altri studenti
  - fornisce ad uno studente la possibilità di leggere gli esami sostenuti ma sicuramente non quella di aggiungere un esame o modificare il voto di un esame sostenuto!
  - non mostra ad un docente la password di uno studente!
- Il Sistema Segrepass ha *molteplici* interfacce, per diverse tipologie di utenti
  - In ognuna di queste viene specificato l'elenco di dati e operazioni accessibili e le modalità di accesso consentite (lettura / scrittura / ...)



# EREDITARIETÀ

- Quando tra una classe (superclasse o classe padre) e un'altra classe (subclasse o classe figlio) c'è una relazione di ereditarietà si intende che:
  - Dal punto di vista concettuale la classe padre rappresenta una generalizzazione della classe figlio
  - Dal punto di vista tecnico, che tutti gli attributi e i metodi della classe padre sono applicabili (ereditati) anche dalla classe figlio
    - Se la classe figlio può avere una propria versione specifica di un metodo rinunciando ad ereditarlo dal padre: in questo caso si parla di *overriding* del metodo
- Una fondamentale relazione tra le classi che consente il riuso parziale di classi complesse



# ESEMPI DI EREDITARIETÀ

- Consideriamo un sistema analogo al Sistema Teams
  - Semplificato a fini didattici
- Esso deve sicuramente tenere conto dell'esistenza di *Utenti*, tra i quali in particolare *Docenti* e *Studenti*
- Che relazioni ci sono tra queste classi?
  - Analizziamo alcune caratteristiche e operazioni
    - Tutti (utenti, student, docent) devono accedere tramite l'operazione di *sign in* ed hanno propri login e password
    - I docenti hanno il diritto di creare un *canale*
    - Gli studenti hanno il diritto di iscriversi ad un canale

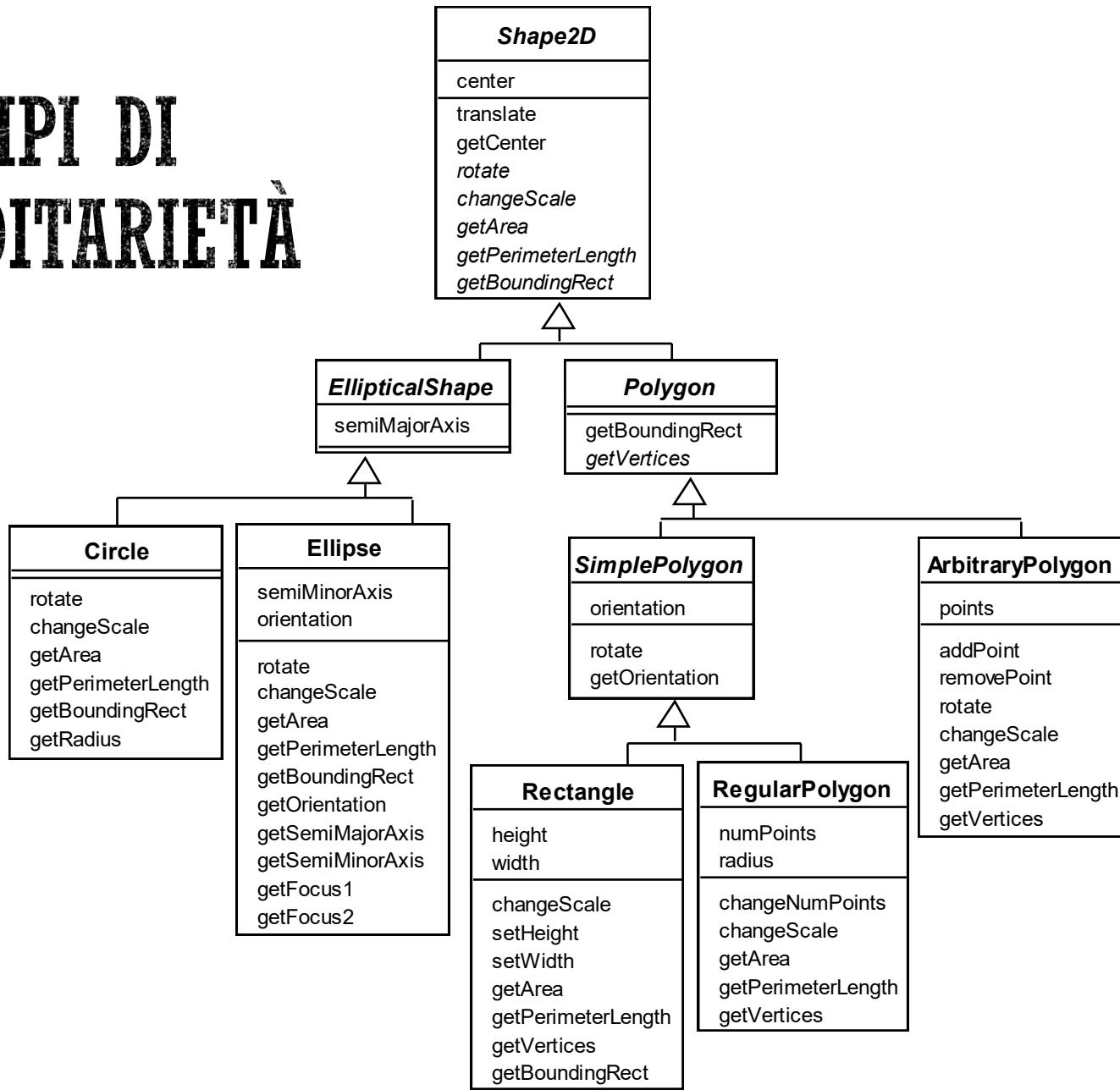


# ESEMPI DI EREDITARIETÀ

- *Utente* rappresenta una generalizzazione (padre) di *Studente* e *Docente* (*figli*)
  - gli attributi *login* e *password* e l'operazione di *sign in* viene definita per la classe utente ed ereditata (è applicabile anche su) *Studente* e *Docente*
- Per *Docente* è inoltre definita l'operazione *crea canale*
- Per *Studente* è definita l'operazione di *iscrizione al canale*
- *Docente* e *Studente* non hanno un legame diretto (hanno la stessa classe padre: sono in qualche modo fratelli – *sibling*)
- Grazie all'ereditarietà possiamo limitare la visibilità/accesso alle informazioni



# ESEMPI DI EREDITARIETÀ



# CONTENIMENTO VS EREDITARIETÀ

- Un errore che si commette spesso consiste nel confondere le gerarchie di contenimento o aggregazione (*tutto-parti*) con quelle di ereditarietà (*padre-figlio* o *is-a*) e con la relazione di istanziazione (*classe-oggetto*)
- **Tutto-parti:** una città è *parte* di una nazione
  - Conseguenza: La popolazione totale di una nazione è la somma delle popolazioni di tutte le sue città
- **Padre-figlio :** una repubblica è un (*is a*) tipo di nazione
  - Conseguenza: Il concetto di popolazione viene stabilito per la classe nazione ed è valido anche per una repubblica (o per una monarchia)
- **Classe-oggetto :** l'Italia è un oggetto della classe repubblica, Napoli è un oggetto della classe città
  - Conseguenza: Siccome repubblica eredita da nazione, l'attributo popolazione assume un valore anche per Italia. L'oggetto Napoli è una parte dell'oggetto Italia



# RETROSPETTIVA: PROCEDURALE

- Quali di questi principi sono parzialmente seguiti già dalla programmazione procedurale?
- Astrazione
  - Le struct raggruppano dati, anche di tipo eterogeneo, in un'unica variabile
    - Ma non è possibile abbinare le operazioni alle struct
  - Le funzioni possono rappresentare *operazioni elementari* da eseguire (*astrazione sul controllo*)
- Modularità
  - Le funzioni rappresentano i *moduli* di un algoritmo complesso
  - I file possono contenere funzioni che sono concettualmente collegate ( *librerie*)



# RETROSPETTIVA

- Quali di questi principi sono parzialmente seguiti già dalla programmazione procedurale?
- Incapsulamento ed information hiding
  - In C++ si raccomanda di dare alle variabili scope più piccoli possibili
    - Una funzione non ha visibilità delle variabili dichiarate in un'altra funzione
    - Un blocco non ha visibilità delle variabili dichiarate in un altro blocco (a meno che non sia un blocco che lo contiene)



# RETROSPETTIVA: TIPI DI DATO ASTRATTO IN C++ (ADT)

**NomeADT.h**

```
// Interfaccia del modulo ADT  
  
//Eventuali definizioni di tipo  
//Prototipi delle operazioni  
//previste sul tipo
```

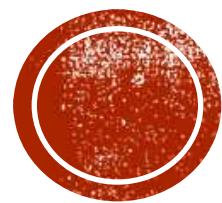
**NomeADT.cpp**

```
// Implementazione del modulo ADT  
  
#include "NomeADT.h"  
//Implementazione delle operazioni  
// previste sul tipo
```

**Utilizzatore.cpp**

```
// Modulo utilizzatore del modulo  
// ADT  
#include "NomeADT.h"
```

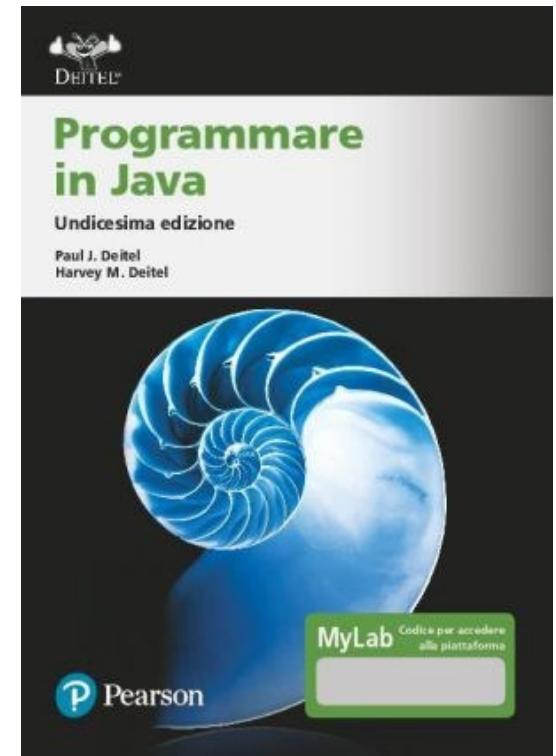




# LINGUAGGIO JAVA

# TESTO DI RIFERIMENTO

- Paul J. Deitel - Harvey M. Deitel: **Programmare in Java**, 11/Ed., Pearson
  - Capitolo 1, sezioni 1.8, 1.9, 1.10
  - Capitolo 2, sezioni da 2.1 a 2.9
  - Capitolo 3 (tranne 3.6)
  - Capitolo 4 (tranne 4.15),
  - Capitolo 5 (tranne 5.11)
  - Capitolo 7, tranne 7.13, 7.17
  - Capitolo 8, sezione 8.4, 8.6, 8.7, 8.8, 8.10, 8.11, 8.13, 8.14
  - Capitolo 9
  - Capitolo 10 da 10.1 a 10.9 e 10.13
  - Capitolo 11, da 11.1 a 11.7 e 11.9
  - Capitolo 14, sezioni 14.1, 14.2, 14.3
  - Capitolo 15, sezioni 15.1, 15.2, 15.4
  - Capitolo 16 da 16.1 a 16.6
- Per chi vuole approfondire la GUI con JavaFX, capitoli 12 e 13



# **INTRODUZIONE AL LINGUAGGIO JAVA**

**Programmare in Java, Capitolo 1, sezioni 1.8, 1.9, 1.10**



# MOTIVAZIONI

- Il linguaggio Java, più di C++, è il linguaggio OO ad alta diffusione sul quale più è facile vedere rispecchiati i principi di modellazione e progettazione di UML
  - In particolare, la maggior parte dei Design Patterns risolvono problemi formulati per il linguaggio Java
- Il linguaggio Java è da parecchi anni uno dei linguaggi più utilizzati nel mondo
- L'esecutore Java è di libero utilizzo
- Un notevole quantitativo di strumenti a supporto dello sviluppo di software in Java è di libero utilizzo (spesso anche open source)
- Ambienti come Android si prestano (ad oggi) ad eseguire applicazioni scritte in Java



# ALCUNI CONTESTI DI UTILIZZO DI JAVA

- Desktop applications



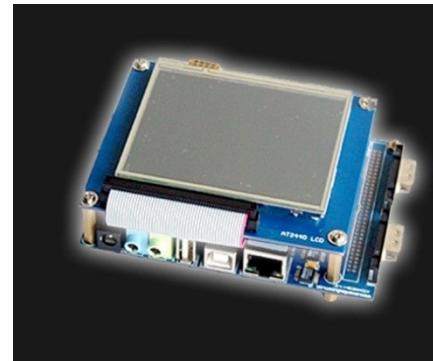
- Mobile



- Sistemi Distribuiti



- Sistemi Embedded



# STORIA DI JAVA

- Java è stato creato a partire da ricerche effettuate alla Stanford University agli inizi degli anni Novanta.
- Nel 1992 nasce il linguaggio Oak (in italiano "quercia"), prodotto da Sun Microsystems e realizzato da un gruppo di esperti sviluppatori capitanati da James Gosling. Successivamente il nome divenne Java e l'icona quella del caffè. Java fu annunciato ufficialmente il 23 maggio 1995
  - Fu creato con una sintassi simile a quella di C++, ma ridotto di alcuni costrutti e caratteristiche ritenuti più proni ad errori di programmazione.
  - Conosce una prima importante diffusione a seguito dell'inclusione della Java Virtual Machine in Netscape, che rese possibile lo sviluppo di Applet
  - Il 13 novembre 2006 la Sun Microsystems ha rilasciato la sua implementazione del compilatore Java e della macchina virtuale (virtual machine) sotto licenza [GPL](#).
- L'8 maggio 2007 Sun ha pubblicato anche le librerie (tranne alcune componenti non di sua proprietà) sotto licenza [GPL](#), rendendo Java un linguaggio di programmazione la cui implementazione di riferimento è libera.

[Fonte: Wikipedia]

- 20 aprile 2009: Oracle corporation, azienda leader nei sistemi per la gestione di basi di dati, ha acquistato la Sun microsystems, azienda californiana produttrice di software e semiconduttori, nota, tra le altre cose, per avere prodotto il linguaggio di programmazione Java. L'acquisto è avvenuto per un controvalore di 7,4 miliardi di dollari (circa 5,7 miliardi di euro)

[Fonte: Corriere della Sera]



# STORIA DI JAVA

- Successivamente, Java si è evoluto costantemente e frequentemente, fino alla versione 17 (formalmente 1.9)
  - Cercheremo, però di mantenere retrocompatibilità fino alla versione 1.8
    - <https://www.java.com/it/download/>
- Java non è più diffuso all'interno delle pagine Web (le applet sono state dichiarate non sufficientemente sicure dalla maggior parte dei produttori di browser) ma, al contrario, è oggi diffuso anche in ambienti nei quali non viene utilizzata la java virtual machine originale
  - Ad esempio, in ambiente Android
- Nonostante la continua proposta di numerosi linguaggi in grado di sostituirlo, la diffusione di Java ne fa ancora uno dei linguaggi più utilizzati in assoluto
  - <https://insights.stackoverflow.com/survey/2021#most-popular-technologies-language>



# JAVA E C++

- Java è un linguaggio OO **completamente a oggetti**
  - Tutto in Java è un oggetto;
  - Tutte le classi appartengono ad un'unica gerarchia
- Rispetto a C++
  - Non è possibile accedere esplicitamente ai puntatori.
  - Non è possibile allocare manualmente la memoria.
  - Non c'è l'ereditarietà multipla.
  - Non è necessario distruggere gli oggetti (ci pensa il *garbage collector*).



# CHE COS'È JAVA

- È un linguaggio (e relativo ambiente di programmazione) definito dalla Sun Microsystems ...
- ... per permettere lo sviluppo di applicazioni *sicure, performanti e robuste su piattaforme multiple, in reti eterogenee e distribuite* (Internet).



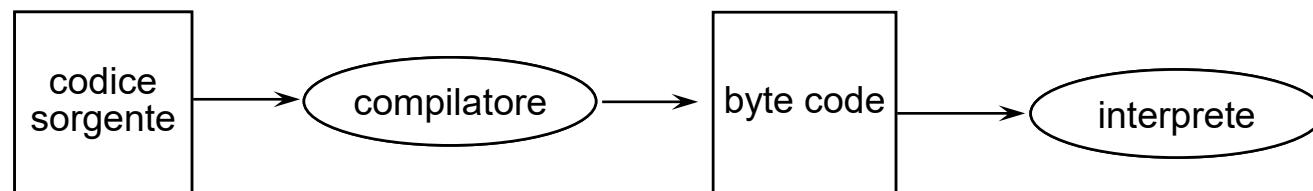
# JAVA: SEMPLICE E O.O.

- Sintassi simile a C e C++ (facile da imparare)
- Elimina i costrutti più "pericolosi" di C e C++
  - aritmetica dei puntatori
  - (de)allocazione esplicita della memoria
  - strutture (struct)
  - definizione di tipi (typedef)
  - preprocessore (#define)
- Aggiunge garbage collection automatica
- Conserva la tecnologia OO di base di C++
- Rivisita C++ in alcuni aspetti



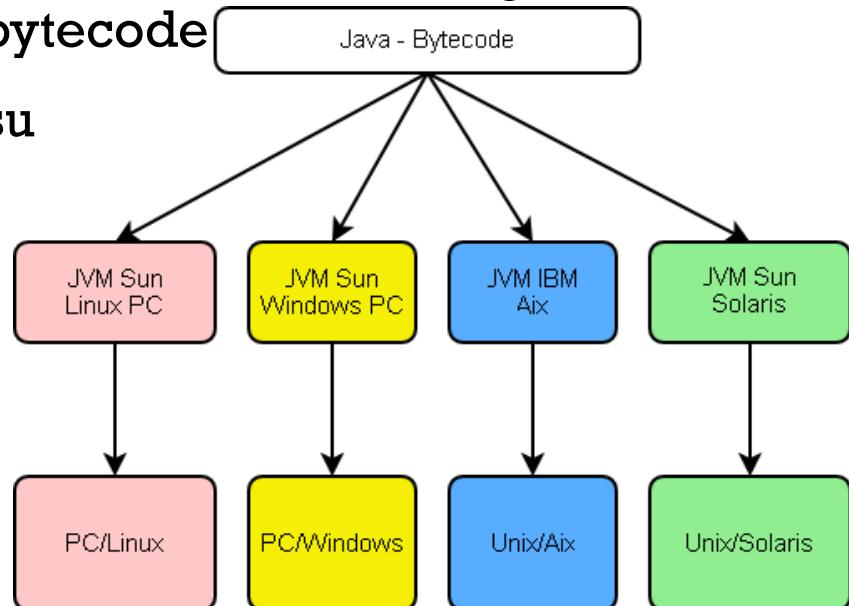
# JAVA È COMPILATI E INTERPRETATO

- Java è un linguaggio per il quale è necessaria sia una *compilazione* che una *interpretazione*.
- Il codice sorgente Java viene compilato producendo un codice di tipo intermedio per una “Java Virtual Machine”, detto *bytecode*.
- Il bytecode viene interpretato dalla **Java Virtual Machine**



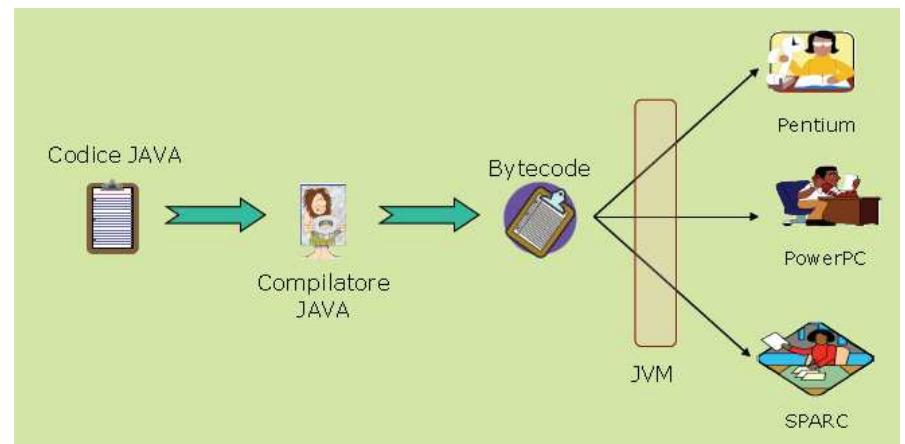
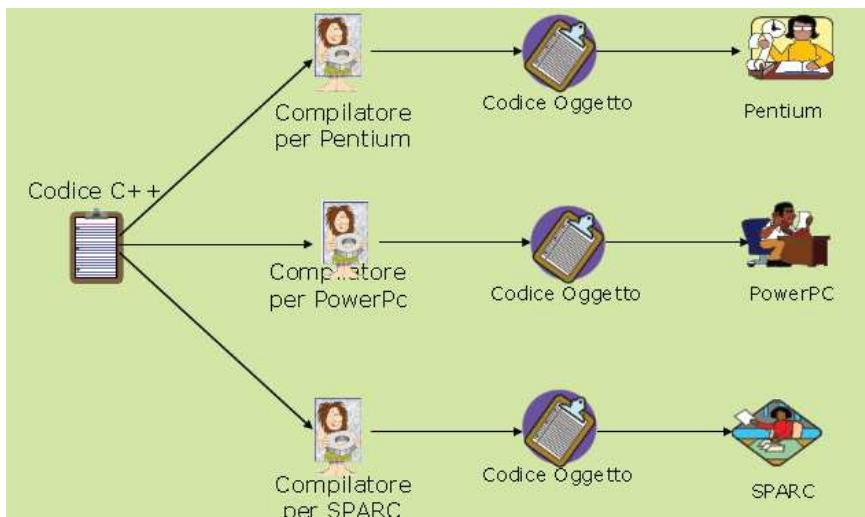
# LA JAVA VIRTUAL MACHINE (JVM)

- Il byte-code è indipendente dall'architettura hardware  
(ANDF: Architecture Neutral Distribution Format)
- Implementazioni della Java Virtual Machine per tantissime macchine e sistemi operativi sono realizzate e distribuite liberamente da Sun-Oracle
- Tramite la Java Virtual Machine si realizza un'astrazione di una macchina virtuale all'interno della macchina reale, in grado di interpretare ed eseguire codice bytecode
- Il bytecode può essere eseguito su qualsiasi sistema provvisto della propria JVM.



# PORTABILITÀ: C++ VS JAVA

- In Java il compilatore e il bytecode sono unici, e non dipendono dalla macchina (né dal sistema operativo)



# CARATTERISTICHE DI JAVA ...

- Semplice
  - Da imparare
  - Ricco di potenti librerie
  - Strumenti di sviluppo adeguati
- Indipendente dall'architettura e portabile
  - Nasce per poter essere eseguito in ambiente Web
  - Avendo a disposizione una JVM, può essere eseguito su qualsiasi sistema
- Dinamico
  - Grazie all'interpretazione, tutte le classi sono caricate in memoria solo quando servono
- Orientato allo sviluppo di software distribuito
  - Si presta nativamente ad una possibile esecuzione in ambiente distribuito, nel quale le classi si vengano a trovare su diverse macchine fisiche
  - Il primo importante utilizzo di Java fu costituito dalle Applet



# ... CARATTERISTICHE DI JAVA

## ■ Sicuro

- Vari meccanismi di sicurezza per la protezione del codice e dei dati da intrusioni di altri processi in esecuzione
- il bytecode viene verificato prima dell'interpretazione ("theorem proving"), in modo da essere certi di alcune sue caratteristiche
- gli indirizzamenti alla memoria nel bytecode sono risolti sotto il controllo dell'interprete

## ■ Prestazioni

- La verifica del bytecode permette di saltare molti controlli a run-time: l'interprete è pertanto efficiente
- Per maggiore efficienza, possibilità di compilazione on-the-fly del bytecode in codice macchina

## ■ Esistenza di potenti librerie standard

- La sua natura free ha reso possibile lo sviluppo, la diffusione e la standardizzazione della maggior parte delle sue librerie principali
  - Ad esempio, le librerie AWT e Swing per l'interfaccia utente

## ■ Multithreaded

- E' nativamente possibile scrivere programmi concorrenti (multithread) per la JVM



# INSTALLAZIONE DI JAVA

- Tutte le istruzioni necessarie si trovano su  
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>
  - JDK (Java Development Kit) contiene la macchina virtuale, tutte le librerie, la documentazione e tutto ciò che può essere utile per lo sviluppo di applicazioni Java
  - JRE (Java Runtime Environment) contiene il minimo di strumenti necessari per poter eseguire un programma Java
- Per quanto esistano strumenti autoinstallanti, è sufficiente copiare tutto il materiale scaricato in una cartella ed eventualmente configurare opportunamente le variabili di ambiente.



- Per cercare di uniformarci, e tenendo conto che abbiamo bisogno di un intero ambiente di sviluppo (jdk), utilizzeremo come riferimento Java 16, scaricabile a quest'indirizzo (Windows):
- <https://www.oracle.com/java/technologies/downloads/#java16>
- Al termine dell'installazione, per verificare il completamento apriamo cmd e scriviamo java -version

```
C:\ Prompt dei comandi  
Microsoft Windows [Versione 10.0.19043.1237]  
(c) Microsoft Corporation. Tutti i diritti sono riservati.  
  
C:\Users\Porfirio>java -version  
java version "16.0.2" 2021-07-20  
Java(TM) SE Runtime Environment (build 16.0.2+7-67)  
Java HotSpot(TM) 64-Bit Server VM (build 16.0.2+7-67, mixed mode, sharing)
```

In ambito windows possiamo verificare che sia stato inserita una cartella Oracle\Java\javapath nel PATH predefinito



# **HELLO, WORLD**

Programmare in Java, capitolo 2, sezioni 2.1, 2.2, 2.3, 2.4



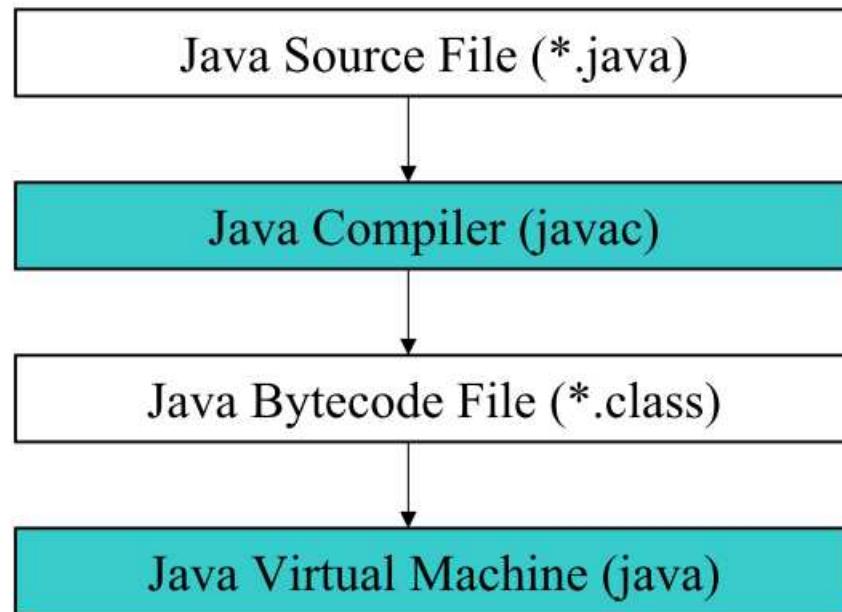
# **STRUTTURA DI UN SEMPLICE PROGRAMMA JAVA: HELLO, WORLD**

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

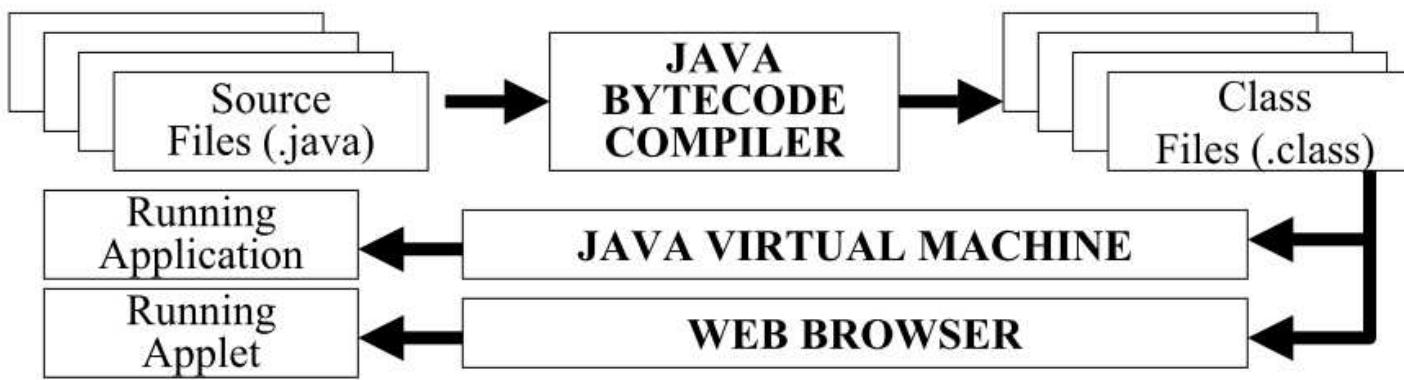
- Tutto deve trovarsi in una classe.
- Il codice sorgente è salvato in un file Hello.java.
- La classe Hello sembra non ereditare da nessun'altra (in realtà eredita come default da Object)



# L'AMBIENTE JAVA

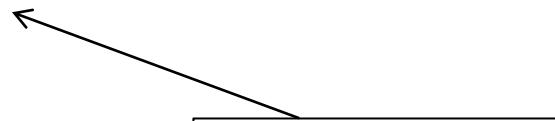


## Organizzazione dei Programmi Java



# COMPILAZIONE ED ESECUZIONE DI HELLO, WORLD

```
C:\Users\Master\eserciziIS1-2012\java\hello>javac Hello.java
```



```
C:\Users\Master\eserciziIS1-2012\java\hello>dir  
Il volume nell'unità C non ha etichetta.  
Numero di serie del volume: CABC-C76B
```

Compilazione

```
Directory di C:\Users\Master\eserciziIS1-2012\java\hello
```

```
11/03/2012 10:28 <DIR> .  
11/03/2012 10:28 <DIR> ..  
11/03/2012 10:28 417 Hello.class  
11/03/2012 10:20 108 Hello.java  
2 File 525 byte  
2 Directory 103.298.981.888 byte disponibili
```

Esecuzione

```
C:\Users\Master\eserciziIS1-2012\java\hello>java Hello  
Hello, World!
```



# **STRUMENTI FONDAMENTALI OFFERTI DALLA JDK**

- **Javac.exe è il compilatore:** trasforma file sorgenti .java in bytecode .class
- **Java.exe è l'esecutore (macchina virtuale)** esegue una classe specificata a partire dal metodo specificato
  - Il metodo non era stato specificato poiché ce n'era uno solo possibile (static)
  - Hello si riferisce alla classe Hello contenuta in Hello.class, NON a Hello.java
- **Il percorso di javac.exe e java.exe può essere indicato nella variabile d'ambiente PATH**
  - Set path=%path%;JAVA\_HOME\bin



# AMBIENTI DI SVILUPPO

- Non è necessario alcuno strumento particolare non contenuto nella JDK per poter eseguire un programma Java
  - I nostri primi esempi mostreranno come sia possibile compilare ed eseguire un qualsiasi programma Java facendo solo uso di istruzioni a linea di comando, in ogni sistema operativo
- Numerosi ambienti di sviluppo (IDE) sono disponibili per i programmatore Java. In particolare:
  - NetBeans, gratuito, direttamente sviluppato da Sun-Oracle.
  - JDeveloper, gratuito, integra strumenti di sviluppo Java con strumenti Oracle per l'interazione con i database
  - Eclipse, gratuito, acquisito da qualche anno da IBM.
  - IntelliJIdea
    - Potenzialmente, anche Android Studio, nato da IntelliJIdea, è in grado di supportare lo sviluppo di programmi Java anche se è fortemente consigliato utilizzarlo solo per sviluppare app Android
  - VSCode
- Tutti gli ambienti sono ulteriormente arricchiti da moltissime estensioni (plug-in) sviluppate da terze parti e, generalmente, gratuite
  - Utilizzeremo diverse estensioni per risolvere problemi di modellazione, analisi, testing.

# INSTALLAZIONE DI ECLIPSE



- Eclipse è uno dei più diffusi ambienti per la realizzazione di programmi in una miriade di diversi linguaggi
  - Nativamente, è stato sviluppato per supportare lo sviluppo di programmi Java
  - È anch'esso, in larga parte, sviluppato in Java
- Utilizzeremo Eclipse per sviluppare alcuni dei primi programmi di esempio in Java
- Nuove versioni di Eclipse sono pubblicate con cadenza annuale, tipicamente associate ad un nome mnemonico che a che fare con l'astronomia
  - La versione più recente attualmente disponibile è Eclipse 2018, liberamente scaricabile da:
    - <https://www.eclipse.org/downloads/>



The Eclipse Installer 2021-09 R now includes a JRE for macOS, Windows and Linux.



## Get Eclipse IDE 2021-09

Install your favorite desktop IDE packages.

[Download x86\\_64](#)

# eclipseinstaller

 by Oomph

type filter text



### Eclipse IDE for Java Developers

The essential tools for any Java developer, including a Java IDE, a Git client, XML Editor, Maven and Gradle integration



### Eclipse IDE for Enterprise Java and Web Developers

Tools for developers working with Java and Web applications, including a Java IDE, tools for JavaScript, TypeScript, JavaServer Pages and Faces, Yaml, Markdown, Web...



### Eclipse IDE for C/C++ Developers

An IDE for C/C++ developers.

Conviene creare una cartella *workspace* destinata a contenere tutti gli esempi e progetti del corso

Eclipse contiene una jre (ambiente di esecuzione di Java) incluso perché Eclipse stesso è realizzato prevalentemente in Java



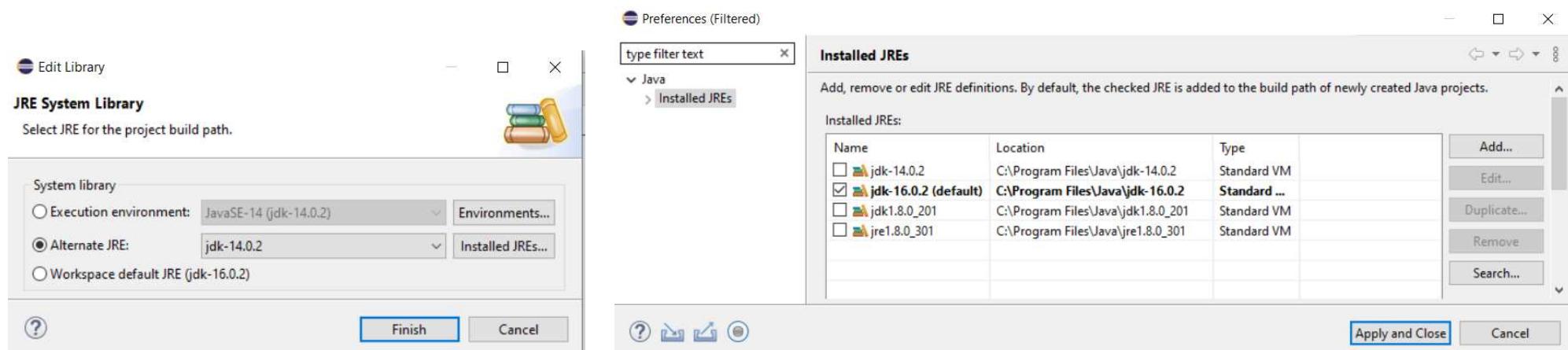
# COESISTENZA DI PIU' VERSIONI DI JAVA ED ECLIPSE

- La coesistenza di un numero arbitrario di versioni di Java ed Eclipse sulla stessa macchina è sicuramente possibile
  - Gli unici problemi possono risiedere nel settaggio di quale debba essere la versione predefinita
- Da linea di comando per essere sicuri della versione di java utilizzata per la compilazione o esecuzione:
  - Scrivere `java -version` per conoscere la versione utilizzata
  - Scrivere il percorso della cartella prima dell'eseguibile `java`
    - Ad esempio con  
`C:\Program Files\Java\jdk1.8.0_201\bin\java Hello`  
Sono sicuro di utilizzare l'esecutore della versione 1.8.0\_201



# COESISTENZA DI PIU' VERSIONI DI JAVA ED ECLIPSE

- Si può scegliere una jre / jdk alternativa o cercare sul disco per tutte quelle presenti



# HELLO, WORLD IN ECLIPSE

- Creare un nuovo Progetto Java
  - Specifica il nome ed eventualmente la versione del compilatore
- Creare una nuova classe
- Completare l'implementazione
  - In questo caso con il codice del metodo main
- Eseguire
  - Nell'esecuzione sono compresi di default il salvataggio dei file non salvati, la compilazione, il linking e l'esecuzione finale utilizzando la scheda Console come terminale



File Edit Source Refactor Navigate Search Project Run Window Help



Package Explorer X

HelloWorld

- > JRE System Library [jdk-16.0.2]
- < src
  - < (default package)
    - > Hello.java

Hello.java X

```
1  
2 public class Hello {  
3  
4     public static void main(String[] args) {  
5         System.out.println("Hello, World");  
6     }  
7  
8 }  
9
```

Outline X

Hello

main(String[]) : void

Problems Javadoc Declaration Console X

<terminated> Hello [Java Application] C:\Program Files\Java\jdk-16.0.2\bin\javaw.exe (20 set 2021, 14:40:38 – 14:40:40)  
Hello, World

Writable

Smart Insert

5:44:106

# STANDARD DI ORGANIZZAZIONE DEI PROGRAMMI IN JAVA

- Ogni classe è implementata nel proprio file sorgente (source file).
- Includere una classe per file:
  - Il nome del file Java è uguale al nome della classe.
- Le applicazioni Java devono includere una classe (quindi un file) con un metodo eseguibile (ad esempio **main**) :

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





## Good Programming Practice 2.2

Use white space to enhance program readability.



# **ELEMENTI FONDAMENTALI DEL LINGUAGGIO JAVA**

Programmare in Java, capitolo 2 (da 2.5 a 2.9), capitolo 3 (tranne 3.6), capitolo 4 (tranne 4.15), capitolo 5 (tranne 5.11)



# PAROLE CHIAVE DA JAVA.LANG

Java Keywords				
<code>abstract</code>	<code>boolean</code>	<code>break</code>	<code>byte</code>	<code>case</code>
<code>catch</code>	<code>char</code>	<code>class</code>	<code>continue</code>	<code>default</code>
<code>do</code>	<code>double</code>	<code>else</code>	<code>extends</code>	<code>false</code>
<code>final</code>	<code>finally</code>	<code>float</code>	<code>for</code>	<code>if</code>
<code>implements</code>	<code>import</code>	<code>instanceof</code>	<code>int</code>	<code>interface</code>
<code>long</code>	<code>native</code>	<code>new</code>	<code>null</code>	<code>package</code>
<code>private</code>	<code>protected</code>	<code>public</code>	<code>return</code>	<code>short</code>
<code>static</code>	<code>super</code>	<code>switch</code>	<code>synchronized</code>	<code>this</code>
<code>throw</code>	<code>throws</code>	<code>transient</code>	<code>true</code>	<code>try</code>
<code>void</code>	<code>volatile</code>	<code>while</code>		
<i>Keywords that are reserved, but not used, by Java</i>				
<code>const</code>	<code>goto</code>			

# LE CLASSI IN JAVA

- In Java (quasi) ogni cosa è una classe:
  - Le classi che scriviamo;
  - Le classi fornite da Java stesso;
  - Le classi fornite da terzi;
  - Estensioni Java.
- Tutte le classi in Java derivano dalla medesima classe di **root** detta **Object**.
  - Lo capiremo meglio quando tratteremo l'ereditarietà.
- Gli oggetti vengono creati tramite la keyword **new**.
- La keyword **new** restituisce un riferimento ad un oggetto che rappresenta un'istanza della classe.
- Tutte le istanze delle classi sono allocate nell'heap.



## ESEMPIO: CLASSE CONTATORE

```
public class Contatore {  
    private int value;  
    private final int max=10;  
  
public Contatore() {value=0;};  
public Contatore(int c) {value=c;};  
public void incrementa(){value++};  
public void decrementa(){value--};  
}
```



# ATTRIBUTI

```
public class Contatore {  
    public int value;  
    private final int max=10;  
  
    public Contatore() {value=0;};  
    public Contatore(int c) {value=c;};  
    public void incrementa(){value++};  
    public void decrementa(){value--};  
}
```

- *value* e *max* sono due attributi della classe
- *int* indica che sono del tipo (scalare) intero
  - *E' un tipo incluso nella libreria standard java lang, che non ha bisogno di essere riferita*
- *value* è pubblica, quindi possiamo leggere o scrivere il suo valore liberamente
- *max* invece è privata, quindi il suo valore può essere letto o modificato solo da altri metodi interni alla classe
- *max* è anche costante (*final*)



# ATTRIBUTI

- Le variabili di una classe sono istanziate e definite per ogni istanza di una classe. Per questo motivo ogni oggetto ha la propria copia della variabile.
- Le variabili di una classe possono essere inizializzate. Altrimenti sono settate a 0 o a null.
- Le variabili locali (interne ai metodi) devono essere inizializzate prima di essere utilizzate.



# MODIFICATORI DI VISIBILITÀ

- Si applicano a classi, metodi, attributi
- Public
  - Visibile a tutto il programma
- Private
  - Visibile solo all'interno della classe
- Protected
  - Visibile all'interno della classe e delle altre classi che la estendono
- Nessun modificatore
  - La visibilità di default si riferisce al package contenente la classe
    - Il concetto di package verrà approfondito più avanti.



# ESEMPIO: CLASSE CONTATORE

```
public class Contatore {  
    private int value;  
    private final int max=10;  
  
    public Contatore() {value=0;};  
    public Contatore(int c) {value=c;};  
    public void incrementa(){value++};  
    public void decrementa(){value--};  
}
```

- I metodi devono essere implementati nello stesso punto in cui sono dichiarati (in alternativa possono essere lasciati astratti e qualche altra classe li dovrà implementare)
- Il metodo con il nome della classe è detto **costruttore** ed è chiamato ogni volta che viene instanziato un oggetto
- Overloading: Sono possibili diverse implementazioni di ogni metodo (ad esempio il costruttore), a patto che si differenzino per il tipo o la quantità di parametri passati



# METODI

- I metodi:
  - sono equivalenti alle funzioni;
    - La differenza fondamentale è che sono applicabili ad un oggetto, che rappresenta il primo parametro della funzione
  - sono definiti all'interno della definizione di una classe;
  - sono “visibili” a tutti gli altri metodi definiti all'interno della classe;
    - La loro visibilità rispetto alle altre classi è definita dagli operatori di visibilità
- Tipo di ritorno
  - Specifica il tipo di dato che il metodo deve restituire.
  - Utilizzo della keyword `return`.
  - Utilizzo di `void` se il metodo non restituisce nessun tipo di dato



# COSTRUTTORE

- Ogni classe può avere uno o più costruttori.
  - Se ci sono diversi costruttori, allora devono avere diversi insiemi di parametri
- È un “metodo” ma può essere invocato solo tramite la parola chiave `new` e ritorna un riferimento all’oggetto istanziato.
- È definito con lo stesso nome della classe a cui appartiene.

```
public Contatore() {value=0;};  
public Contatore(int c) {value=c;};
```



# ESEMPIO DI UTILIZZO

```
public class Contatore {  
    public int value;  
    private final int max=10;  
  
    public Contatore() {value=0;};  
    public Contatore(int c) {value=c;};  
    public void incrementa(){value++;};  
    public void decrementa(){value--};  
}
```

- Codice di utilizzo del contatore (posizionato in altra classe)

```
public class Main {  
    public static void main(String[] args){  
        Contatore c = new Contatore();  
        c.incrementa();  
        System.out.println(c.value);  
        int v=5;  
        Contatore c2;  
        c2 = new Contatore (v);  
        c.decrementa();  
        System.out.println(c.value);  
    };  
}
```



# RIASSUMENDO . . .

- Elementi visti finora:
  - Come si dichiara una classe
  - Come si dichiara un attributo
  - Come si dichiara / definisce un metodo
    - Come si dichiara / definisce un costruttore
  - Come si dichiara una variabile di un tipo primitivo o scalare
  - Come si assegna un valore ad una variabile di tipo primitivo
  - Come si dichiara un riferimento a un oggetto
  - Come si istanzia un oggetto allocato dinamicamente
  - Come si passa un parametro primitivo o scalare per valore



# RIASSUMENDO . . .

- Elementi visti finora:
  - Come si dichiara una classe → **class Contatore ...**
    - Come si dichiara un attributo → **int value; String nome;**
    - Come si dichiara / definisce un metodo → **void incrementa () {value++;};**
      - Come si dichiara / definisce un costruttore → **public Contatore(){...}**
  - Come si dichiara una variabile di un tipo primitivo o scalare → **int v;**
  - Come si assegna un valore ad una variabile di tipo primitivo → **v=5;**
  - Come si dichiara un riferimento a un oggetto → **Contatore c;**
  - Come si istanzia un oggetto allocato dinamicamente
    - → **Contatore c = new Contatore();**
  - Come si passa un parametro primitivo o scalare per valore
    - → **c2 = new Contatore (v);**





## Common Programming Error 2.2

A compilation error occurs if a `public` class's file-name is not exactly the same name as the class (in terms of both spelling and capitalization) followed by the `.java` extension.





## Common Programming Error 2.4

The compiler error message “`class Welcome1 is public, should be declared in a file named Welcome1.java`” indicates that the filename does not match the name of the `public` class in the file or that you typed the class name incorrectly when compiling the class.





## Error-Prevention Tip 2.2

When the compiler reports a syntax error, it may not be on the line that the error message indicates. First, check the line for which the error was reported. If you don't find an error on that line, check several preceding lines.





## Good Programming Practice 2.3

By convention, every word in a class-name identifier begins with an uppercase letter. For example, the class-name identifier `DollarAmount` starts its first word, `Dollar`, with an uppercase D and its second word, `Amount`, with an uppercase A. This naming convention is known as **camel case**, because the uppercase letters stand out like a camel's humps.





## Good Programming Practice 2.8

Choosing meaningful variable names helps a program to be self-documenting (i.e., one can understand the program simply by reading it rather than by reading associated documentation or creating and viewing an excessive number of comments).





## Good Programming Practice 2.9

By convention, variable-name identifiers use the camel-case naming convention with a lowercase first letter—for example, `firstNumber`.

2.9





## Good Programming Practice 2.4

Indent the entire body of each class declaration one “level” between the braces that delimit the class’s. This format emphasizes the class declaration’s structure and makes it easier to read. We use three spaces to form a level of indent—many programmers prefer two or four spaces. Whatever you choose, use it consistently.





## Good Programming Practice 2.5

IDEs typically indent code for you. The Tab key may also be used to indent code. You can configure each IDE to specify the number of spaces inserted when you press Tab.





## Good Programming Practice 2.7

Place a space after each comma ( , ) in an argument list to make programs more readable.





## Common Programming Error 2.3

It's a syntax error if braces do not occur in matching pairs.





## Error-Prevention Tip 2.1

When you type an opening left brace, {, immediately type the closing right brace, }, then reposition the cursor between the braces and indent to begin typing the body. This practice helps prevent errors due to missing braces. Many IDEs do this for you.





## Good Programming Practice 2.6

Indent the entire body of each method declaration one “level” between the braces that define the method’s body. This emphasizes the method’s structure and makes it easier to read.



# TIPI SCALARI (PRIMITIVI)

- Solo alcuni tipi scalari possono essere trattati direttamente come valori:
  - byte, short, int, long
  - float, double
  - boolean, char
- Un'assegnazione tra variabili scalari provoca una copia
  - int a=10; int b;
  - b=a;
- Sui tipi scalari sono definiti tutti gli operatori classici così come in C e C++ (aritmetici, logici, ...)
- I tipi scalari NON sono classi, infatti si scrivono con iniziale *minuscola*



# TIPI PRIMITIVI

Type	Size in bits	Values	Standard
<b>boolean</b>	8	<b>true or false</b>	
<b>char</b>	16	' \u0000' to ' \uFFFF' <b>(0 to 65535)</b>	<b>(ISO Unicode character set)</b>
<b>byte</b>	8	<b>-128 to +127</b> <b>(-2<sup>7</sup> to 2<sup>7</sup> - 1)</b>	
<b>short</b>	16	<b>-32,768 to +32,767</b> <b>(-2<sup>15</sup> to 2<sup>15</sup> - 1)</b>	
<b>int</b>	32	<b>-2,147,483,648 to +2,147,483,647</b> <b>(-2<sup>31</sup> to 2<sup>31</sup> - 1)</b>	
<b>long</b>	64	<b>-9,223,372,036,854,775,808 to</b> <b>+9,223,372,036,854,775,807</b> <b>(-2<sup>63</sup> to 2<sup>63</sup> - 1)</b>	
<b>float</b>	32	<i>Negative range:</i> <b>-3.4028234663852886E+38 to</b> <b>-1.40129846432481707e-45</b> <i>Positive range:</i> <b>1.40129846432481707e-45 to</b> <b>3.4028234663852886E+38</b>	<b>(IEEE 754 floating point)</b>
<b>double</b>	64	<i>Negative range:</i> <b>-1.7976931348623157E+308 to</b> <b>-4.94065645841246544e-324</b> <i>Positive range:</i> <b>4.94065645841246544e-324 to</b> <b>1.7976931348623157E+308</b>	<b>(IEEE 754 floating point)</b>



# JAVA È STRONGLY TYPED

- Java è un linguaggio strongly typed.
- I tipi scalari NON sono compatibili tra loro.
- Lo strong typing riduce molti errori comuni di programmazione.
- Le assegnazioni devono essere fatte tra tipi di dati compatibili, senza perdita di informazione.
- Il casting è permesso se esplicito
  - float risultato= (float ) x / (float) y; //divisione reale
  - int risultato= (int ) x / (int) y; //divisione intera
- Contrariamente, in C:
  - Non c'è differenza in termini di **allocazione e intercambiabilità tra short int, char e boolean**
  - Ci sono cast impliciti tra molti tipi di dati.



# OPERATORI ARITMETICI

Java operation	Operator	Algebraic expression	Java expression
Addition	+	$f + 7$	<code>f + 7</code>
Subtraction	-	$p - c$	<code>p - c</code>
Multiplication	*	$bm$	<code>b * m</code>
Division	/	$x / y$ or $\frac{x}{y}$ or $x \div y$	<code>x / y</code>
Remainder	%	$r \bmod s$	<code>r % s</code>

**Fig. 2.11** | Arithmetic operators.



# ALCUNI OPERATORI DI ASSEGNAZIONE

- Si applicano **solo** ai tipi scalari
  - Per gli oggetti è necessario definire metodi ad hoc

Assignment operator	Sample expression	Explanation	Assigns
<i>Assume:</i> <code>int c = 3, d = 5, e = 4, f = 6, g = 12;</code>			
<code>+=</code>	<code>c += 7</code>	<code>c = c + 7</code>	<code>10 to c</code>
<code>-=</code>	<code>d -= 4</code>	<code>d = d - 4</code>	<code>1 to d</code>
<code>*=</code>	<code>e *= 5</code>	<code>e = e * 5</code>	<code>20 to e</code>
<code>/=</code>	<code>f /= 3</code>	<code>f = f / 3</code>	<code>2 to f</code>
<code>%=</code>	<code>g %= 9</code>	<code>g = g % 9</code>	<code>3 to g</code>
<b>Arithmetic assignment operators.</b>			



# OPERATORI DI INCREMENTO E DECREMENTO

Operator	Called	Sample expression	Explanation
<code>++</code>	preincrement	<code>++a</code>	Increment <b>a</b> by 1, then use the new value of <b>a</b> in the expression in which <b>a</b> resides.
<code>++</code>	postincrement	<code>a++</code>	Use the current value of <b>a</b> in the expression in which <b>a</b> resides, then increment <b>a</b> by 1.
<code>--</code>	predecrement	<code>--b</code>	Decrement <b>b</b> by 1, then use the new value of <b>b</b> in the expression in which <b>b</b> resides.
<code>--</code>	postdecrement	<code>b--</code>	Use the current value of <b>b</b> in the expression in which <b>b</b> resides, then decrement <b>b</b> by 1.

**Fig. 4.13 The increment and decrement operators.**





## Good Programming Practice 2.10

Place spaces on either side of a binary operator for readability.



# COSTANTI

- In C++ le costanti sono definite utilizzando `const` o `#define`.
- In Java è un poco più complicato:
  - `public final <static> <datatype> <name> = <value>;`
- Esempi:
  - `public final static double PI = 3.14;`
  - `public final static int NumStudenti = 60;`
- Le costanti non possono essere modificate.
  - La costanza del valore è espressa da `final`
  - La costanza della sua allocazione in memoria da `static`



# PUNTATORI E RIFERIMENTI

- In Java NON esiste il concetto di puntatore, esiste invece il concetto di Riferimento.
  - non c'è bisogno di un simbolo (come &) per specificare un riferimento.
- L'operatore `new` restituisce il riferimento all'istanza di un oggetto, allocato dinamicamente:
  - Contatore `c = new Contatore();`
- La semplice dichiarazione di un oggetto causa soltanto l'istanziazione (statica) del riferimento, non dell'oggetto
  - Contatore `c2;`
- Un assegnazione come `c2=c` corrisponde solo alla realizzazione di un *alias*: `c2` è un altro riferimento allo stesso oggetto istanziato con l'istruzione `new`
  - Se volessimo avere una copia di un oggetto scriveremmo `c2.clone(c);`
    - Poi discuteremo sulla semantica del metodo `clone` e sulla possibilità di modificarlo
- Un riferimento può essere inizializzato a `null` per non farlo puntare a nessun oggetto: `c2=null;`



# PUNTATORI VS RIFERIMENTI

- I riferimenti sono quindi concettualmente equivalenti ai puntatori ma con alcune limitazioni:
  - Non possiamo conoscere l'indirizzo esatto in memoria di un riferimento
    - Se cerchiamo di stampare a video un riferimento leggiamo un codice univoco che però non è l'indirizzo di memoria, che viene liberamente scelto a tempo di esecuzione dalla Java Virtual Machine
  - Non possiamo modificare da programma l'indirizzo in cui è memorizzato un oggetto
    - Non esiste nulla di equivalente all'aritmetica dei puntatori
  - Non esiste l'operatore \* che trasforma un puntatore nella variabile puntata
    - Di conseguenza non esiste nemmeno -> che era equivalente a (\*p).
    - Al contrario possiamo accedere direttamente all'oggetto tramite il suo riferimento e ai suoi campi (attributi) tramite l'operatore .



# DISTRUZIONE DEGLI OGGETTI (GARBAGE COLLECTOR)

- In Java non è necessario implementare distruttori.
- Esiste un meccanismo interno della JVM, chiamato *garbage collector* che ripulisce la memoria eliminando tutti gli oggetti per i quali non esistano più riferimenti attivi.
  - Quindi vengono eliminati tutti gli oggetti che non sono riferiti da alcun riferimento (ad esempio perché quel riferimento ora punta a null)
- Il *garbage collector* libera il programmatore dall'eseguire manualmente l'allocazione/deallocazione di memoria.
- Sono state ridotte o eliminate in tal modo alcune categorie di bug.
- Per le variabili scalari vale la distruzione automatica al termine del loro scope.



# LOGICA CONDIZIONALE

- La logica condizionale in Java viene eseguita con lo statement `if`
- A differenza di C++ un'espressione logica non valuta lo 0 (FALSE) o un non-0 (TRUE), ma valuta `o false o true`.
- `true e false` sono gli unici due valori assunti da variabili di tipo boolean.



# OPERATORI DI CONFRONTO

- Tra variabili scalari:

```
int a=10; int b=10;  
(a==b) vale true
```

- Tra oggetti

```
String a="pippo"; String b="pippo";  
a.equals(b) vale true  
a==b vale false
```

a==b è il confronto tra i riferimenti (puntatori) agli oggetti: essendo a e b due oggetti diversi, a==b vale false



# OPERATORI DI CONFRONTO

Algebraic operator	Java equality or relational operator	Sample Java condition	Meaning of Java condition
<i>Equality operators</i>			
=	==	$x == y$	$x$ is equal to $y$
≠	!=	$x != y$	$x$ is not equal to $y$
<i>Relational operators</i>			
>	>	$x > y$	$x$ is greater than $y$
<	<	$x < y$	$x$ is less than $y$
≥	≥	$x ≥ y$	$x$ is greater than or equal to $y$
≤	≤	$x ≤ y$	$x$ is less than or equal to $y$

**Fig. 2.14** | Equality and relational operators.



# OPERATORI (E PRECEDENZE)

- Anche questi operatori sono applicabili solo agli scalari e non agli oggetti

Operators	Associativity	Type
( )	left to right	parentheses
++ --	right to left	unary postfix
++ -- + - (type)	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
? :	right to left	conditional
= += -= *= /= %=	right to left	assignment

# OPERATORI LOGICI E PRECEDENZE

Operators	Associativity	Type
<code>()</code>	<b>left to right</b>	<b>parentheses</b>
<code>++ --</code>	<b>right to left</b>	<b>unary postfix</b>
<code>++ -- + - ! (type)</code>	<b>right to left</b>	<b>unary</b>
<code>* / %</code>	<b>left to right</b>	<b>multiplicative</b>
<code>+ -</code>	<b>left to right</b>	<b>additive</b>
<code>&lt; &lt;= &gt; &gt;=</code>	<b>left to right</b>	<b>relational</b>
<code>== !=</code>	<b>left to right</b>	<b>equality</b>
<code>&amp;</code>	<b>left to right</b>	<b>boolean logical AND</b>
<code>^</code>	<b>left to right</b>	<b>boolean logical exclusive OR</b>
<code> </code>	<b>left to right</b>	<b>boolean logical inclusive OR</b>
<code>&amp;&amp;</code>	<b>left to right</b>	<b>logical AND</b>
<code>  </code>	<b>left to right</b>	<b>logical OR</b>
<code>? :</code>	<b>right to left</b>	<b>conditional</b>
<code>= += -= *= /= %=</code>	<b>right to left</b>	<b>assignment</b>



## Good Programming Practice 2.11

Indent the statement(s) in the body of an `if` statement to enhance readability. IDEs typically do this for you, allowing you to specify the indent size.





## Error-Prevention Tip 2.4

You don't need to use braces, { }, around single-statement bodies, but you must include the braces around multiple-statement bodies. You'll see later that forgetting to enclose multiple-statement bodies in braces leads to errors. To avoid errors, as a rule, always enclose an `if` statement's body statement(s) in braces.





## Common Programming Error 2.7

Placing a semicolon immediately after the right parenthesis after the condition in an `if` statement is often a logic error (although not a syntax error). The semicolon causes the body of the `if` statement to be empty, so the `if` statement performs no action, regardless of whether or not its condition is true. Worse yet, the original body statement of the `if` statement always executes, often causing the program to produce incorrect results.





## Error-Prevention Tip 2.5

A lengthy statement can be spread over several lines. If a single statement must be split across lines, choose natural breaking points, such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines until the end of the statement.



# LOOPING CONSTRUCTS

- Java supporta tre looping constructs
  - while, do ... while, for.
- Esempi:

```
for (int i = 0; i < 10; i++) {  
}
```

---

----

```
int i = 0;  
while(i < 10) {  
}
```

---

----

```
int i = 0;  
do {  
} while(i < 10)
```



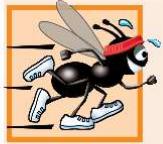
# PASSAGGIO DEI PARAMETRI: TIPI PRIMITIVI

- In Java esiste un solo passaggio: per valore
  - Le variabili scalari sono passate per valore, quindi la funzione riceve una copia del valore dello scalare, che può modificare liberamente e senza effetti collaterali
    - La modifica riguarda la copia e non l'originale
  - Se il valore di ritorno è scalare, allora viene ritornato il valore
    - Se volessimo passare una variabile scalare in un altro modo (per riferimento), dovremo preventivamente trasformarla in un oggetto corrispondente (ad esempio un int in un Integer)
      - L'argomento verrà approfondito più avanti



# PASSAGGIO DEI PARAMETRI: OGGETTI

- In Java esiste un solo passaggio: per riferimento
  - Tutti gli oggetti sono passati per riferimento
    - Nel senso che viene passata una copia del riferimento (alias)
      - Se si modificano gli attributi dell'oggetto saranno modificati gli originali
      - Se si modifica il valore del riferimento è solo una copia che viene modificata
    - Se il valore di ritorno è un oggetto, allora viene ritornato il riferimento
  - Nota: un utilizzo più avanzato di Java consente di passare ad un metodo *funzioni* o addirittura *classi*
    - Tutta questa casistica si riconduce al passaggio per riferimento di un oggetto *di tipo method* o *di tipo class*



## Performance Tip 7.1

Passing references to arrays, instead of the array objects themselves, makes sense for performance reasons. Because Java arguments are passed by value, if array objects were passed, a copy of each element would be passed. For large arrays, this would waste time and consume considerable storage for the copies of the elements.



# **ARRAY, MATRICI, STRINGHE**

Programmare in Java, Capitolo 7, tranne sezioni 7.5, tranne  
7.13, 7.16, 7.17, Capitolo 14, sezioni 14.1, 14.2, 14.3



# ARRAY

- Gli array in Java modellano gruppi di variabili dello stesso tipo
- Le variabili contenute nell'array possono essere:
  - Tipi primitivi (scalari)
  - Riferimenti ad oggetti
- L'array può essere acceduto per indice, con l'indice che assume valori  $\geq 0$



# DICHIARAZIONE DI UN ARRAY

- `int[] c=new int[12];`
- Oppure
  - `int[] c;`
  - `c = new int[12]`
- Il numero degli elementi non viene mai dichiarato in fase di dichiarazione (a sinistra) ma solo in fase di definizione (a destra)
  - In altri termini, quello a sinistra è sempre e comunque un riferimento, mentre la allocazione è ad opera della definizione (dinamica con `new` oppure statica);
- E' possibile conoscere la lunghezza di un array a tempo di esecuzione (in C++ non si poteva), scrivendo
  - `c.length`



# DICHIARAZIONE DI UN ARRAY

- `int[] array = {42,71,23};`
  - Dichiara un array di tre interi con i valori iniziali di 42, 71, 23
  - `array.length` restituisce 3
- Nota: anche se dalla dichiarazione non appare evidente, l'array viene trattato come un oggetto
  - `array` è un riferimento all'array
  - `array.length` è un attributo di array
  - Al contrario `array[1]` è un valore di un tipo primitive (`int`)
  - Se ho due array `a` e `b`, allora l'assegnazione `a=b` indica che il riferimento all'oggetto `b` è assegnato al riferimento all'oggetto `a`, cioè `a` e `b` sono due alias che puntano allo stesso array (quello che abbiamo chiamato `b`)





## Common Programming Error 7.2

In an array declaration, specifying the number of elements in the square brackets of the declaration (e.g., `int[12] c;`) is a syntax error.





## Good Programming Practice 7.1

For readability, declare only one variable per declaration. Keep each declaration on a separate line, and include a comment describing the variable being declared.





## Error-Prevention Tip 7.1

When writing code to access an array element, ensure that the array index remains greater than or equal to 0 and less than the length of the array. This will prevent `ArrayIndexOutOfBoundsExceptions` if your program is correct.



# PASSAGGIO DEI PARAMETRI: ESEMPI E CONTROESEMPI CON GLI ARRAY

```
int[] array = {42,71,23};  
  
void modify(int x){x++;}  
  
void reset(int[] x){x=null;}
```

- La chiamata `modify(array[2])` ;
- Passa al metodo `modify` una *copia* del valore di `array[2]`
  - Il valore di `array[2]` dopo l'esecuzione di `modify` rimane 23
  - La variabile `x` della funzione `modify` vale invece 24
- La chiamata `reset(array)`
- Pone a null la variabile `x` della funzione `reset` ma non la variabile `array`
  - Perchè alla funzione è stata passata una *copia* del riferimento `array`



## 7.15 CLASS ARRAYS

- **Arrays class**
  - Provides static methods for common array manipulations.
- **Methods include**
  - `sort` for sorting an array (ascending order by default)
    - `Arrays.sort(array);`
  - `binarySearch` for searching a sorted array
    - `int location = Arrays.binarySearch(array,value);`
  - `equals` for comparing arrays
    - `Boolean b = Arrays.equals (array1, array2);`
  - `fill` for placing values into an array.
    - `Arrays.fill(array,value);`
- Methods are overloaded for primitive-type arrays and for arrays of objects.
- **System class static `arraycopy` method**
  - Copies contents of one array into another.
  - `System.arraycopy ( array, start, array2, start2, end2);`

Non essendo gli array una vera e propria classe, alcune funzioni di utilità sono contenute in una classe *Arrays*





## Error-Prevention Tip 7.3

When comparing array contents, always use `Arrays.equals(array1, array2)`, which compares the two arrays' contents, rather than `array1.equals(array2)`, which compares whether `array1` and `array2` refer to the same array object.





## Common Programming Error 7.6

Passing an unsorted array to `binarySearch` is a logic error—the value returned is undefined.



# MATRICI

- Le matrici (a due o più dimensioni) sono perfettamente analoghe agli array
  - `int [] [] m = new int [10] [10];`
  - `m[3][5] = 42;`
    - Una matrice in Java è vista come un array di array: ogni riga ha il suo riferimento
    - È possibile passare la matrice per riferimento senza specificare né il numero di righe né il numero di colonne nella dichiarazione di funzione
- Esempio:
  - `int [] [] m = { {1,2}, {3,4,5} }`
    - Crea una matrice di due righe : la prima riga di lunghezza 2, la seconda di lunghezza 3
      - `m[0].length` restituisce 2
      - `m[1].length` restituisce 3



# FOR EACH (ENHANCED FOR)

- Disponibile solo da Java 5 in poi

For-each loop

```
for (type var : arr) { ... }
```

```
for (type var : coll) { ... }
```

Equivalent for loop

```
for (int i = 0; i < arr.length; i++) {  
type var = arr[i]; ... }
```

```
for (Iterator<type> iter =  
coll.iterator(); iter.hasNext(); ) {  
type var = iter.next(); ... }
```

- A parole potremmo leggerlo come «per ogni var di tipo type appartenente ad arr)
  - Ovviamente arr (o coll) deve essere una collezione di elementi per i quali esiste un modo di scorrerli in un determinato ordine



# FOR EACH: ESEMPIO E CONTROESEMPIO

- Con For Each

```
double[] ar = {1.2, 3.0, 0.8};  
int sum = 0;  
for (double d : ar) { sum += d; }
```

- Con il For classico

```
double[] ar = {1.2, 3.0, 0.8};  
int sum = 0;  
for (int i = 0; i < ar.length; i++) {sum += ar[i]; }
```



# STRINGHE

- Le Stringhe NON sono modellate come array di char ma come oggetti della classe String
- Esempi d'uso:
  - `String s=new String("Hello");`
  - `String s="Hello";`
  - `String s2=s1;`
    - Copia solo il riferimento, quindi s2 diventa un alias di s1
  - `String s2=new String(s1)`
    - Copia tutto l'oggetto; s2 e s1 sono due variabili indipendenti
  - `s2==s1`
    - Vero se si tratta dello STESSO oggetto (cioè se s2 ed s1 sono due alias)
  - `s2.equals(s1)`
    - Vero se si tratta di due stringhe con lo stesso valore (equals è definita in object e ridefinita in String)
- <http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/String.html>



---

```
1 // Fig. 14.1: StringConstructors.java
2 // String class constructors.
3
4 public class StringConstructors {
5     public static void main(String[] args) {
6         char[] charArray = {'b', 'i', 'r', 't', 'h', ' ', 'd', 'a', 'y'};
7         String s = new String("hello");
8
9         // use String constructors
10        String s1 = new String();
11        String s2 = new String(s);
12        String s3 = new String(charArray);
13        String s4 = new String(charArray, 6, 3);
14
15        System.out.printf(
16            "s1 = %s%n"
17            "s2 = %s%n"
18            "s3 = %s%n"
19            "s4 = %s%n", s1, s2, s3, s4);
20    }
21 }
```

```
s1 =
s2 = hello
s3 = birth day
s4 = day
```

**Fig. 14.1** | String class constructors.



## 14.3.2 STRING METHODS LENGTH, CHARAT AND GETCHARS

- **String** method **length** determines the number of characters in a string.
- **String** method **charAt** returns the character at a specific position in the **String**.
- **String** method **getChars** copies the characters of a **String** into a character array.
  - The first argument is the starting index in the **String** from which characters are to be copied.
  - The second argument is the index that is one past the last character to be copied from the **String**.
  - The third argument is the character array into which the characters are to be copied.
  - The last argument is the starting index where the copied characters are placed in the target character array.



### 14.3.3 COMPARING STRINGS

- Strings are compared using the numeric codes of the characters in the strings.
- Figure 14.3 demonstrates `String` methods `equals`,  
`equalsIgnoreCase`, `compareTo` and `regionMatches` and using the  
equality operator `==` to compare `String` objects.



# APPROFONDIMENTI SULLE STRINGHE

- Il capitolo 14 del libro contiene una esauriente trattazione di tutte le modalità più interessanti di interazione con le stringhe



# **EREDITARIETA' E POLIMORFISMO**

Programmare in Java, Capitolo 9, Capitolo 10 da 10.1 a 10.9  
e 10.13



# EREDITARIETÀ

- Quando tra una classe (superclasse o classe padre) e un'altra classe (subclasse o classe figlio) c'è una relazione di ereditarietà si intende che:
  - Dal punto di vista concettuale la classe padre rappresenta una generalizzazione della classe figlio
  - Dal punto di vista tecnico, che tutti gli attributi e i metodi della classe padre sono applicabili (ereditati) anche dalla classe figlio
    - Se la classe figlio può avere una propria versione specifica di un metodo rinunciando ad ereditarlo dal padre: in questo caso si parla di *overriding* del metodo
- Una fondamentale relazione tra le classi che consente il riuso parziale di classi complesse



# SINTASSI

- Per esprimere che la classe Figlio eredita dalla classe Padre:

```
class Figlio extends Padre { ...}
```

- Il resto della dichiarazione della classe è identica
- Una classe in Java ne può estendere (può ereditare direttamente da) al più una sola classe
  - Per la precisione, vedremo che ne può estendere una e una sola, ma che talvolta la classe che va a estendere è lasciata sottointesa
    - La classe genitore *di default* è la classe Object



# VISIBILITÀ'

- Gli attribute/metodi **public** sono visibili alla classe figlia
  - E a tutti gli eventuali altri discendenti
- Gli attribute/metodi **private** non sono visibili alla classe figlia
- Gli attribute/metodi **protected** sono visibili solo alla classe figlia
  - E a tutti gli eventuali altri discendenti
- Su di un oggetto f della classe Figlia possono quindi essere richiamati:
  - Attributi/metodi della classe Figlia
  - Attributi/metodi **public** o **protected** della classe Padre
- La classe Figlia è in generale un arricchimento (estensione) della classe Padre poichè ne eredita attributi/metodi (ad eccezione di quelli **private**) e definisce ulteriori attribute/metodi aggiuntivi



# ESEMPI DI EREDITARIETÀ

- *Utente* rappresenta una generalizzazione (padre) di *Studente* e *Docente* (*figli*)
  - gli attributi *login* e *password* e l'operazione di *sign in* vengono definite per la classe utente ed ereditati (sono applicabile anche su) *Studente* e *Docente*
- Per *Docente* è inoltre definita l'operazione *crea canale*
- Per *Studente* è definita l'operazione di *iscrizione al canale*
- *Docente* e *Studente* non hanno un legame diretto (hanno la stessa classe padre: sono in qualche modo fratelli – *sibling*)



# EREDITARIETÀ IN JAVA: ESEMPIO

```
public class Utente {  
    public String login;  
    public String password;  
    public boolean signIn(String l, String p) {  
        if (l.contentEquals(login) && p.contentEquals(password))  
            return true;  
        else  
            return false;  
    }  
}  
  
public class Docente extends Utente {  
    public void creaCanale(String nomeCanale) {  
        System.out.println("IL docente "+login);  
        System.out.println("si iscrive al canale "+ nomeCanale);  
    }  
}  
  
public class Studente extends Utente {  
    public void iscrizioneCanale(String nomeCanale) {  
        System.out.println("Lo studente "+login);  
        System.out.println("si iscrive al canale "+ nomeCanale);  
    }  
}
```

Sia Docente che Studente “estendono” (ereditano, specializzano, sono un tipo di) Utente

signIn, login e password sono applicabili anche su Studente e Docente



# EREDITARIETA IN JAVA: MAIN DI PROVA

```
public class Main {  
  
    public static void main(String[] args) {  
  
        Docente d=new Docente();  
  
        d.login="Paperino";  
  
        d.password="Paperone";  
  
        if (d.signIn("Paperino", "Paperone"))  
  
            d.creaCanale("Gruppo2");  
  
        Studente s=new Studente();  
  
        s.login="Pippo";  
  
        s.password="Pluto";  
  
        if (s.signIn("Pippo", "Pluto"))  
  
            s.iscrizioneCanale("Gruppo 2");  
  
    }  
}
```

Output:

Il docente Paperino  
si iscrive al canale Gruppo2  
Lo studente Pippo  
si iscrive al canale Gruppo 2



# RICAPITOLANDO

- Ogni classe figlio può avere una sola classe padre (indicata da `extend`)
  - Se non c'è nessuna ereditarietà esplicita, allora la classe eredita dalla classe `Object` (vedremo in seguito)
- Gli attributi e i metodi del padre sono visibili e utilizzabili dal figlio a meno che non siano dichiarati `private`
  - se sono dichiarati `protected` saranno visibili solo dai figli e dai loro eredi
- Quando si dichiara un oggetto della classe figlio vengono quindi istanziati attributi e collegamenti ai metodi della classe padre, oltre quelli del figlio



# OVERRIDING

- Una classe figlio non dovrebbe mai ridefinire un attributo ereditato dalla classe padre
  - Lo ha già a sua disposizione!
- La classe figlio può ridefinire un metodo già dichiarato nella classe padre dichiarando un Overriding (sovraposizione)
- In questo caso la classe Figlio, essendo una *specializzazione* della classe Padre, sta definendo la sua soluzione *specificata* ad un problema risolto in altro modo dalla classe padre



# ESEMPIO

- Supponiamo che Utente abbia un metodo stampa che mostra a video login e password
- Studente ridefinisce questo metodo aggiungendo anche che si tratta di uno studente
- Docente ridefinisce anch'esso il metodo aggiungendo che si tratta di un docente
  
- Tutti e tre i metodi stampa non hanno alcun parametron
  - Si tratta quindi di ridefinizioni/sovraposizioni/override di metodi
  - L'annotazione (opzionale) **@Override** nel codice evidenzia questa occorrenza e aiuta a prevenire errori



# SUPER

- Nella definizione della classe figlio si può fare diretto riferimento ai metodi della classe padre
- La classe padre è identificata durante la scrittura del codice della classe figlia con la parola chiave **super**
  - Proviamo a modificare il metodo stampa (override) di Studente sfruttando il metodo stampa di Utente
- Spesso il costruttore della classe figlio nella sua realizzazione utilizza il costruttore della classe padre chiamandolo **super()**
  - L'override **non** è consentito con i costruttori poichè ogni classe **dove** avere il proprio costruttore



# SUPER

- Nella definizione della classe figlio si può fare diretto riferimento ai metodi della classe padre

```
public class Studente extends Utente {  
    ...  
    @Override  
    public boolean signIn(String l, String p) {  
        System.out.println("Eseguo il signIn come studente");  
        l=l+"@studente";  
        return super.signIn(l, p);  
    }  
}
```

- Spesso il costruttore della classe figlio nella sua realizzazione utilizza il costruttore della classe padre chiamandolo super()



# POLIMORFISMO

- Il Polimorfismo consente di applicare soluzioni *generali* a diversi tipi di oggetto, lasciando alla Virtual Machine il compito di riconoscere l'oggetto specifico



# POLIMORFISMO: ESEMPIO

- Continuando con l'esempio precedente supponiamo che:
  - s sia un oggetto della classe Studente
  - d sia un oggetto della classe Docente
- Sono consentite le assegnazioni:
  - Utente u1 = s;
  - Utente u2 = d;
  - Utente u = new Utente();
- Cosa avverrà quando scriveremo:
  - u1.stampa();
  - u2.stampa();
  - u.stampa();



# POLIMORFISMO: ESEMPIO

- Continuando con l'esempio precedente supponiamo che:
  - s sia un oggetto della classe Studente
- Sono consentite le assegnazioni:
  - Utente u1 = s;
  - Utente u2 = new Docente();
  - Utente u = new Utente();
- Cosa avverrà quando scriveremo:
  - u1.stampa(); → viene chiamato il metodo stampa() di studente
  - u2.stampa(); → viene chiamato il metodo stampa() di docente
  - u.stampa(); → viene chiamato il metodo stampa() di utente



# POLIMORFISMO: UPCASTING

- La regola generale è che in un assegnazione la classe del riferimento (a sinistra):
  - Deve coincidere con la classe del riferimento a destra oppure
  - deve essere una classe che viene estesa, direttamente o indirettamente dalla classe sul lato destro
    - Quindi va bene sia che si tratti di una classe figlia, che di una classe discendente qualsiasi
- Esempio:
  - Utente u = new Studente();
  - Utente u = new Docente();



# OVERLOADING

- L'Overloading si verifica quando una classe contiene più di una definizione dello stesso metodo
  - Queste definizioni devono però avere un diverso prototipo, cioè diversi tipi di parametri
  - La JVM cerca quindi il metodo giusto basandosi sui tipi dei parametri
- Esempio:  
**public** Contatore() {value=0;};  
**public** Contatore(int c) {value=c;};



# CLASSE OBJECT

- Tutte le classi per le quali non sia dichiarata esplicitamente l'ereditarietà da una classe genitore, estendono di default la classe Object
  - I metodi definiti dalla classe Object possono essere richiamati su qualsiasi oggetto di qualsiasi altra classe (a meno che non siano stati ridefiniti). Particolarmente utilizzati:
    - `toString()`: cerca di porre in formato stringa il nome o un identificativo dell'oggetto
    - `getClass()`: restituisce un oggetto `Class`, che descrive la classe cui un oggetto appartiene
    - `equals(Object o)`: vero se i due oggetti sono uguali
    - `clone()` : restituisce un oggetto copia dell'oggetto su cui è applicato



# CONTAINER

Programmare in Java, Capitolo 7 sezione 7.16, Capitolo 16  
da 16.1 a 16.7



# CLASSI CONTENITORE

- Un array consente di memorizzare insiemi di dimensione predefinita di dati tra loro omogenei (a meno del polimorfismo)
- Per utilizzi più generali è possibile utilizzare una delle cosiddette classi **Contenitore** (*Container*):
  - Collection
  - List
  - Set
  - Map
  - ...
- Tutti i container sono legati tra loro, in una gerarchia
- Alcuni container coincidono con le strutture definite abitualmente negli altri linguaggi



Interface	Description
<b>Collection</b>	The root interface in the collections hierarchy from which interfaces <b>Set</b> , <b>Queue</b> and <b>List</b> are derived.
<b>Set</b>	A collection that does <i>not</i> contain duplicates.
<b>List</b>	An ordered collection that <i>can</i> contain duplicate elements.
<b>Map</b>	A collection that associates keys to values and <i>cannot</i> contain duplicate keys. Map does not derive from Collection.
<b>Queue</b>	Typically a <i>first-in, first-out</i> collection that models a <i>waiting line</i> ; other orders can be specified.

**Fig. 16.1** | Some collections-framework interfaces.

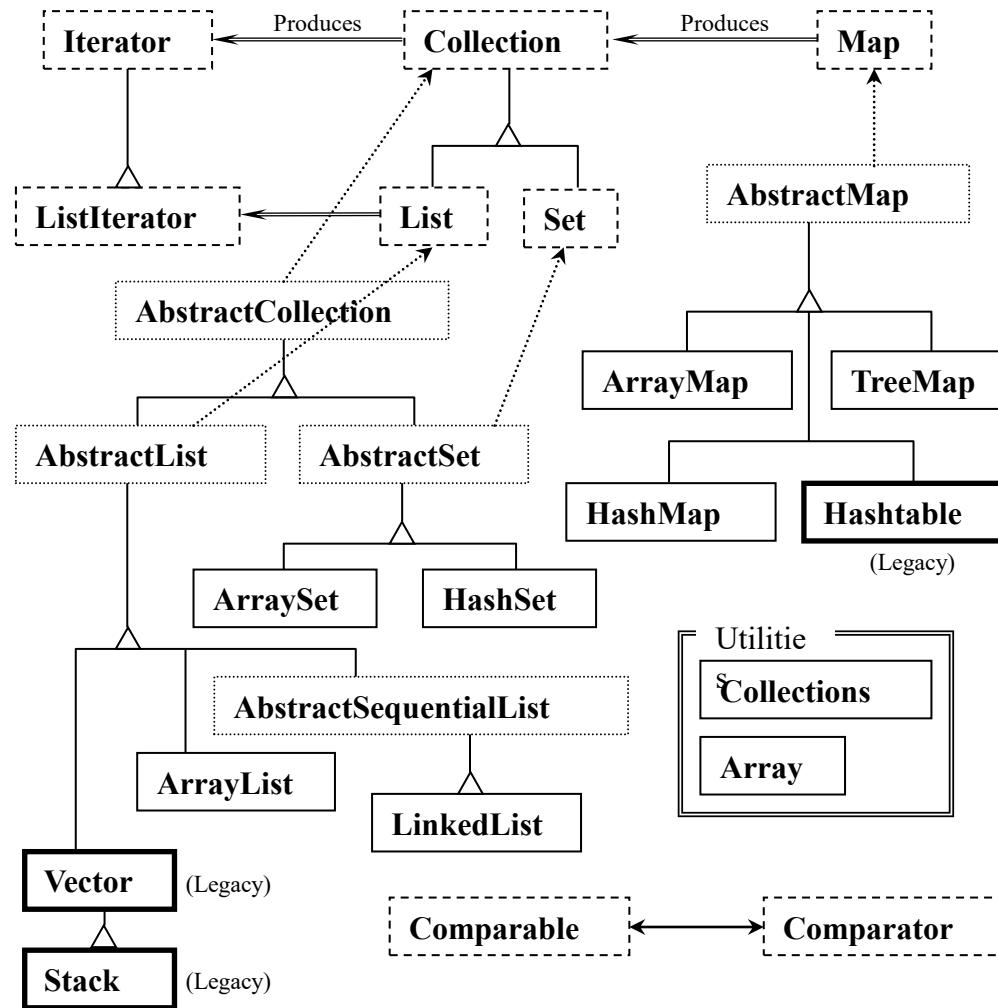


# CENNO AGLI ALTRI CONTAINER

- **Iterator**
  - Iterator è una classe molto generale, da cui molti ereditano. Essa ha i metodi fondamentali per scorrere una lista
- **List**
  - List è la parte comune di un insieme di classi (tra cui ArrayList) che hanno i metodi fondamentali per essere scorse (ereditano da Iterator) e per essere accedute direttamente (per indice)
- **Set**
  - Set è un insieme di classi (ArraySet, HashSet) che realizzano insiemi, quindi hanno anche metodi (add aggiunge un elemento solo se non esiste già, contains controlla se un elemento è nell'insieme, etc.)
- **Map**
  - E' una classe astratta con diverse implementazioni (HashMap, ArrayMap, TreeMap) che realizza strutture dati chiave-valore (cioè array che hanno un'ulteriore possibilità di indicizzazione oltre quella della loro posizione numerica)



# CLASSI CONTENITORE: LA GERARCHIA





## Good Programming Practice 16.1

Avoid reinventing the wheel—rather than building your own data structures, use the interfaces and collections from the Java collections framework, which have been carefully tested and tuned to meet most application requirements.

16.1



# ARRAYLIST

- ArrayList è una delle più semplici ed utili classi container
- Implementa una lista (sequenziale) di elementi, come in un array
  - La dimensione dell'ArrayList varia dinamicamente (come in una lista linkata)
  - Si può accedere sia alla testa (come in una pila), che alla coda (come in una coda), che all'indice di un qualunque elemento (come in un array)
    - String a="pippo"
    - ArrayList lista = new ArrayList()
    - lista.add(a);
    - lista.get(0);
    - lista.size()



# GENERICHE E TEMPLATE

- Una classe *generica* o *template* è una classe che ha tra i suoi parametri anche un indicatore di un'altra classe
- Ad esempio:
  - `ArrayList<String>`
- È una classe `ArrayList` sottoposta al vincolo che tutti i suoi elementi devono essere riferimenti ad oggetti della classe `String`
  - Tecnicamente un `ArrayList` senza altre indicazione deve essere visto come un `ArrayList` che può contenere riferimenti ad oggetti ad `Object` oppure ad una qualsiasi altra classe che erediti direttamente o indirettamente da `Object` (cioè qualsiasi classe)
  - Un `ArrayList<String>` invece può contenere solo riferimenti ad oggetti della classe `String` oppure di un'altra classe che eredita direttamente o indirettamente da `String`



# CARATTERISTICHE

- Tecnicamente l'indicazione del template <String> potrebbe sembrare superflua, ma serve:
  - Per esprimere un concetto di modellazione
    - Es. "vogliamo una lista di nomi di utenti"
  - Per consentire al compilatore controlli di tipo più precisi
- Tecnicamente, una lista (in generale un container) è una struttura dati eterogenea che può contenere oggetti di classi diverse
  - Con l'unico eventuale vincolo di appartenere o ereditare alla classe indicato nel parametro di template
- In questo corso non approfondiremo il meccanismo di creazione di nuove classi template
  - E' introdotto nel capitolo 16 del libro



# ESEMPIO ARRAYLIST

```
import java.util.ArrayList;  
  
public class EsempioArrayList {  
  
    public static void main(String[] args) {  
        String a="pippo";  
  
        int b=3;  
  
        ArrayList lista = new ArrayList();  
  
        lista.add(a);  
        lista.add(b);  
  
        System.out.println(lista.get(0).toString()  
        );  
  
        Integer ultimo=lista.size()-1;  
  
        System.out.println(lista.get(ultimo));  
    }  
}
```

```
import java.util.ArrayList;  
  
public class Esempio2ArrayList {  
  
    public static void main(String[] args) {  
        String a="pippo";  
        String b="pluto";  
  
        ArrayList<String> lista = new ArrayList<String>();  
  
        lista.add(a);  
        lista.add(b);  
  
        System.out.println(lista.get(0));  
        Integer ultimo=lista.size()-1;  
        System.out.println(lista.get(ultimo));  
    }  
}
```



Method	Description
<code>add</code>	Overloaded to add an element to the <i>end</i> of the <code>ArrayList</code> or at a specific index in the <code>ArrayList</code> .
<code>clear</code>	Removes all the elements from the <code>ArrayList</code> .
<code>contains</code>	Returns <code>true</code> if the <code>ArrayList</code> contains the specified element; otherwise, returns <code>false</code> .
<code>get</code>	Returns the element at the specified index.
<code>indexOf</code>	Returns the index of the first occurrence of the specified element in the <code>ArrayList</code> .
<code>remove</code>	Overloaded. Removes the first occurrence of the specified value or the element at the specified index.
<code>size</code>	Returns the number of elements stored in the <code>ArrayList</code> .
<code>trimToSize</code>	Trims the capacity of the <code>ArrayList</code> to the current number of elements.

**Fig. 7.23** | Some methods of class `ArrayList<E>`.



Method	Description
<code>sort</code>	Sorts the elements of a <code>List</code> .
<code>binarySearch</code>	Locates an object in a <code>List</code> , using the efficient binary search algorithm which we introduced in Section 7.15 and discuss in detail in Section 19.4.
<code>reverse</code>	Reverses the elements of a <code>List</code> .
<code>shuffle</code>	Randomly orders a <code>List</code> 's elements.
<code>fill</code>	Sets every <code>List</code> element to refer to a specified object.
<code>copy</code>	Copies references from one <code>List</code> into another.

**Fig. 16.5** | Some Collections methods.



Method	Description
min	Returns the smallest element in a Collection.
max	Returns the largest element in a Collection.
addAll	Appends all elements in an array to a Collection.
frequency	Calculates how many collection elements are equal to the specified element.
disjoint	Determines whether two collections have no elements in common.

**Fig. 16.5** | Some Collections methods.



# POLIMORFISMO E CONTAINER

- Proviamo a creare una `ArrayList` di oggetti `Utente`
- Inseriamo in essa studenti, docenti e utenti generici
- Proviamo a stampare tutta la lista



# CLASSI WRAPPER

- Da quanto visto rispetto ai container, essi sono visti come contenitori di *oggetti*, ma sembrano non essere utilizzabili come contenitori di tipi primitivi
- Le classi **Wrapper** risolvono questo problema per ogni tipo primitivo
  - Boolean, Byte, Character, Double, Float, Integer, Long, Short
    - Le classi numeriche ereditano inoltre tutte da una classe Number
    - Le classi wrapper non possono essere ulteriormente estese
- Possiamo facilmente trasformare uno scalare in un oggetto della classe Wrapper chiamando il costruttore

```
int x=42;  
Integer y = new Integer (x);
```
- Le classi Wrapper hanno in sè parecchi metodi utili, ad esempio per la conversione tra tipo:

```
String s="42";  
int x = Integer.parseInt(s);
```



# AUTOBOXING E UNBOXING

- Java fornisce meccanismi automatici per le conversioni tra ogni tipo primitivo e il suo corrispondente tipo Wrapper e viceversa tramite I meccanismi di **Autoboxing** e **Unboxing**
- `Integer [] array = new Integer [5];`
  - Dichiara un array di 5 oggetti Integer
- `array [0] = 42;`
  - **Autoboxing**: il numero (int) 42 viene automaticamente inscatolato e trasformato in Integer per essere aggiunto come oggetto in array
- `int value = array [0];`
  - **Unboxing**: l'Integer in array[0] viene automaticamente estratto e trasformato in int per essere assegnato a value



# **ECCEZIONI**

Programmare in Java, Capitolo 7, sezione 7.5, Capitolo 11,  
da 11.1 a 11.7 e 11.9



# GESTIONE DELLE ECCEZIONI

- Nei linguaggi classici, come nel caso del C, in caso di errore a tempo di esecuzione il processo si blocca
- In Java, siccome l'esecuzione dei programmi avviene all'interno di un processo Java Virtual Machine, è possibile una più ampia e dettagliata gestione delle eccezioni
- Un'eccezione in Java è un oggetto che viene istanziato (*thrown*) quando se ne verifica una causa scatenante e può essere catturata (*caught*, participio passato di *catch*) e gestita opportunamente dal programma stesso



# ESEMPIO DI ECCEZIONE (NON GESTITA)

```
public class Esaminati {  
    ArrayList<Studente> esaminato= new ArrayList();  
  
    Studente leggi(int i) {  
        Studente s=esaminato.get(i);  
        return s;  
    }  
}
```

Nel main:

```
Esaminati e=new Esaminati();  
e.leggi(0).stampa();
```

Causa l'interruzione del programma con un messaggio di errore



# STACKTRACE DI UNA ECCEZIONE

- Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 1 out of bounds for length 0
- at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:64)
- at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:70)
- at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:266)
- at java.base/java.util.Objects.checkIndex(Objects.java:359)
- at java.base/java.util.ArrayList.get(ArrayList.java:427)
- at esempio.Esaminati.leggi(Esaminati.java:9)
- at esempio.Main.main(Main.java:75)



# GESTIONE DELLE ECCEZIONI: TRY/CATCH

- Vogliamo evitare che il programma si chiuda con un errore a run-time e, viceversa, provare a gestirlo

```
try {  
    //Blocco di codice  
  
}  
  
catch (ExceptionClass1 e1) {  
    //codice di gestione di eccezioni ExceptionClass1}  
  
catch (ExceptionClass2 e2) {  
    //codice di gestione di eccezioni ExceptionClass2}  
  
...  
  
finally {  
    //codice da eseguire al termine di una qualsiasi  
    //eccezione}
```



# TRY/CATCH/FINALLY

- Nel blocco try devono entrare le istruzioni che si considerano “a rischio”
  - Ad esempio se all’interno del try ci fossero le istruzioni per l’apertura di un file, potremmo monitorare eventuali eccezioni a run time dovute alla inesistenza del file o alla sua protezione
- Possono esserci uno o più blocchi catch
- L’oggetto parametro del catch deve essere di una classe che *extends* la classe **Exception**
  - Quest’oggetto contiene tutte le informazioni salvate dal sistema relativamente alla natura dell’eccezione. Alcuni metodi per accedervi:
    - e.getMessage() restituisce il messaggio d’errore
    - e.printStackTrace() restituisce lo stack di tutte le chiamate di metodo innestate che erano presenti al momento dell’eccezione



# ESEMPIO: TRY-CATCH NELLA CHIAMATA

- Nel main:

```
Studente ss=null;  
  
try {  
    ss= e.leggi(0);  
  
}catch (IndexOutOfBoundsException e1) {  
    System.out.println("Errore: L'elemento di indice "+0+" non e' presente  
nell'array");  
}  
  
try{  
    ss.stampa();  
}catch (NullPointerException e2) {  
    System.out.println("Non e' stato selezionato uno studente");  
}
```



# ESEMPIO: TRY-CATCH NELLA FUNZIONE

- Nella funzione leggi (qui rinominata leggiEccezione):

```
public Utente leggiEccezione(int i) {  
  
    Studente s = null;  
  
    try {  
  
        s=esaminato.get(i);  
  
    }catch (IndexOutOfBoundsException e) {  
  
        System.out.println("Errore: L'elemento di indice "+i+" non e'  
        presente nell'array");  
  
    } catch (NullPointerException e) {  
  
        System.out.println("Non e' stato selezionato uno studente");  
  
    }  
  
    return s;  
}
```



# GESTIONE DELLE ECCEZIONI

- All'interno di un catch in generale si può:
  - Stampare o scrivere su un file di log un messaggio dettagliato d'errore
    - Per stampare i messaggi standard possiamo scrivere e.getMessage() e e.printStackTrace()
  - Provare a correggere il problema
    - Ad esempio potremmo decidere che se l'indice è troppo grande restituiamo l'ultimo studente inserito oppure chiamiamo un metodo che chiede all'utente di scrivere un valore corretto
  - Chiudere il programma
    - System.exit(error\_code);
    - Questo è il comportamento che avremmo anche senza try-catch, ma con la differenza che scriviamo un valore di error\_code che potrà essere utile ad un altro programma
  - Ritentare
    - Si fa quando il problema potrebbe essere temporaneo, ad esempio un errore di mancata connessione a Internet



# CREAZIONE DELLE ECCEZIONI

- Una eccezione può essere anche sollevata direttamente dal programma con l'istruzione **throw**
  - `throw new IOException ("File not found");`
  - Meccanismo paragonabile a quello delle trap in assembler
- Il metodo che solleva l'eccezione deve preventivamente dichiarare la propria capacità di sollevarle con la parola **throws**

```
public void divisione (int x, int y) throws ArithmeticException {  
    if (y!=0)  
        r=x/y;  
    else  
        throw new ArithmeticException("Divisione per zero");  
}
```

- Se ci «annoiamo» di scrivere codice nella classe per gestire un'eccezione possiamo scrivere **throws** nel metodo che può creare l'eccezione ed essa sarà gestita dal metodo chiamante. Se anche il metodo chiamante fa **throws** l'eccezione può risalire fino al main o fino al sistema operativo



# ESEMPIO DI LANCIO (THROW) DI UNA ECCEZIONE

- In Esaminati.java:

```
public void falsificaEsame() throws Exception {  
    throw new Exception("Allarme: tentativo di frode!");  
}
```

In Main.java:

```
try {  
    e.falsificaEsame();  
} catch (Exception e1) {  
    System.out.println(e1.getMessage());  
    System.out.println("Non si fa!");  
}
```



# NECESSITA' DELLA THROWS

- Grazie alla throws, Eclipse può suggerirci, al momento in cui scriviamo la chiamata a `e.falsificaEsame()` l'obbligo di:
  - Aggiungere try-catch
  - Oppure *redirigere* a qualcun altro la gestione dell'eccezione
- Ricapitolando, un metodo che dichiara di lanciare (throws) una eccezione:
  - Può essere il creatore dell'eccezione (con l'istruzione *throw new Exception*)
    - In questo caso la crea e la rilancia al chiamante
  - Può eseguire un metodo che a sua volta dichiarava di lanciare una eccezione
    - In questo caso la riceve da questo metodo e lo rilancia al chiamante



# REDIREZIONE DELL'ECCEZIONE CON THROWS

- In Utente abbiamo un metodo che crea e lancia un'eccezione:

```
public void cambiaLogin(String s) throws Exception {  
    if (s.equals(""))  
        throw new Exception();  
}
```

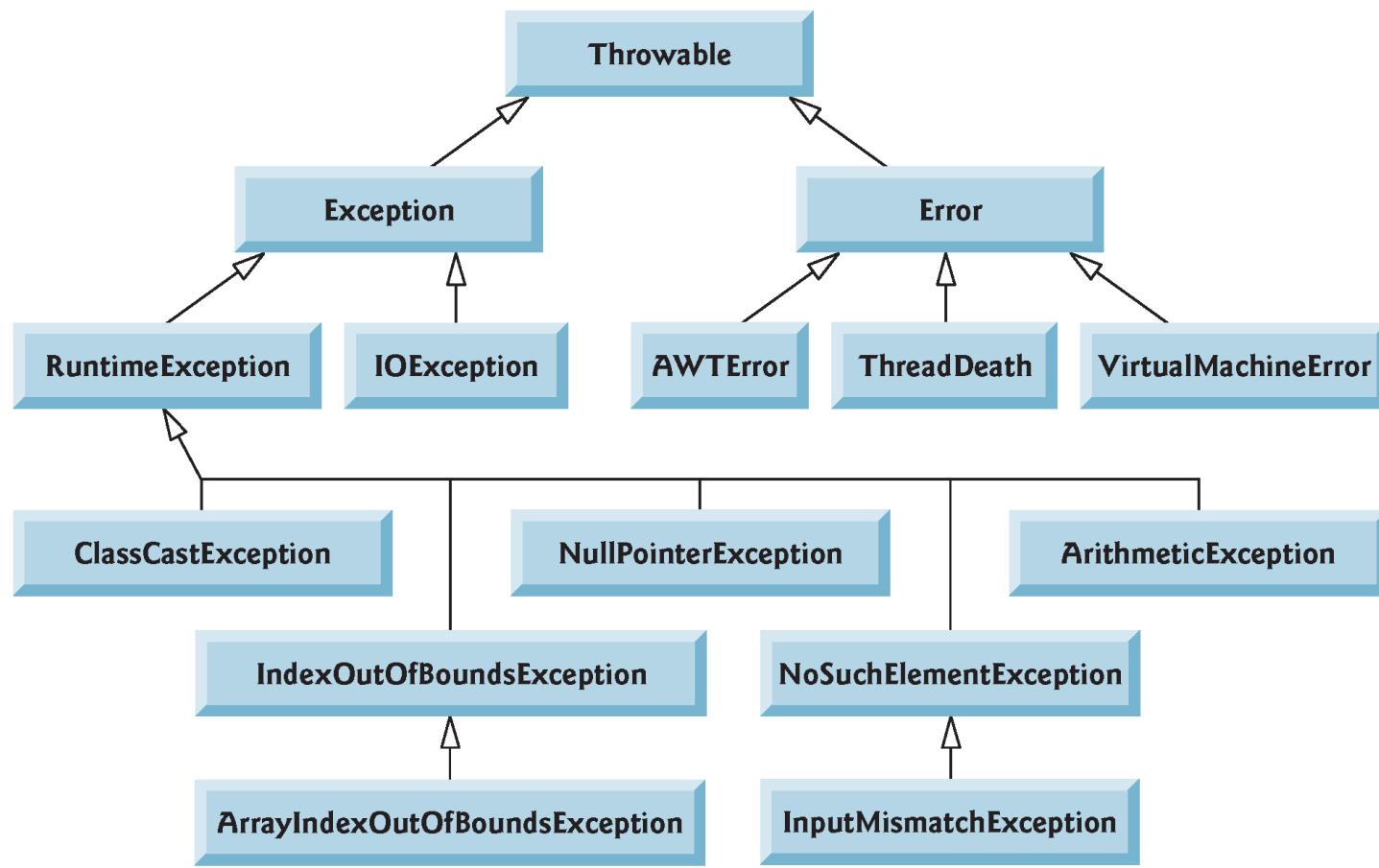
In Esaminati dovremmo gestire un'eccezione ma ci limitiamo a rilanciarla:

```
public void cambiaLogin(Studente ss, String s) throws Exception {  
    ss.cambiaLogin(s);  
}
```

In Main siamo costretti a gestire l'eccezione nata in Utente e rilanciata da Esaminati:

```
try {  
    s.cambiaLogin("");  
} catch (Exception e1) {  
    e1.printStackTrace();  
}
```





**Fig. 11.4** | Portion of class `Throwable`'s inheritance hierarchy.



# GERARCHIA DELLE ECCEZIONI

- Exception è l'avo comune di tutte le eccezioni
  - Se dichiariamo un catch per Exception siamo sicuri che qualsiasi eccezione sia sempre gestita da almeno un catch
  - Se dichiariamo un catch per un Exception e uno per una eccezione più specifica (ad esempio NullPointerException) e si verifica una NullPointerException verrà eseguita solo quest'ultima catch
    - In generale viene eseguita la catch relativa alla classe più specifica rispetto all'eccezione verificata (idealmente la classe corretta)
- Siamo liberi nel nostro Progetto di creare nuove classi di eccezione che ereditano da Exception o da un'altra classe di eccezione
  - Ovviamente se creiamo una nuova classe di eccezione *MiaException* l'unico modo con il quale possa verificarsi è quello nel quale una parte del nostro stesso programma esegue un *throw new MiaException*



# **PACKAGE E LIBRERIE**

Programmare in Java, Capitolo 2



# PACKAGE

- I file sorgenti possono essere organizzati in package
  - La definizione di package in Java è perfettamente coerente con la definizione UML
  - I package sono rappresentati come cartelle nel file system
- 
- In cima ad ogni file si definisce il package di appartenza con la sintassi:
    - `package nomepackage;`
  - Il nome di un package dovrebbe essere univoco
  - I package possono essere innestati gli uni negli altri
    - `com.sun.java`
      - Il package java è nel package sun che è nel package com
    - I nomi dei package sono scelti in modo da garantirne l'unicità
      - Spesso si utilizza il nome del dominio Web dell'autore (univoco perché rilasciato da un ente internazionale) rovesciato, da destra a sinistra



# IMPORTAZIONE

- In Java è possibile **importare** package, ovvero dichiararne il collegamento
- All'inizio di ogni file vengono dichiarati i package che esso importa, ovvero dei quali verranno istanziati oggetti, richiamati metodi, etc.
  - `import nomePackage;`
  - Per importare tutte le classi di un package
    - `import nomePackage.*;`
  - Per importare una specifica classe
    - `import nomePackage.nomeClasse;`
- L'importazione in Java è risolta dal compilatore, non dal precompilatore
- Per accedere al nome di una classe di un package non importato, è necessario scrivere tutto il nome del package quando si usa la classe
- Per accedere ad una classe importata è sufficiente specificare il nome della classe (salvo omonimie che il compilatore segnalerebbe)
- Ai package corrispondono cartelle del file system che ne contengono i file
  - Così come le cartelle, anche i package possono essere innestati in una gerarchia ad albero
- Il package `java.lang` è importato di default



# ESEMPIO PACKAGE

```
com.inovaos.WatsOn
it.inova.database
Database.java 281 16/01/12 0.30 mari
it.inova.rubrica
ApplicationObserver.java 274 13/01/12 0.30 mari
RubricaManager.java 281 16/01/12 0.30 mari
it.inova.utility
MemoryManager.java 236 04/01/12 1.0 mari
Option.java 209 29/12/11 17.16 mari
it.inova.wcs.webservices.model
InovaAddress.java 140 09/12/11 18.4! mari
InovaFax.java 179 21/12/11 12.45 mari
InovaMail.java 191 26/12/11 20.35 mari
InovaSms.java 109 05/12/11 19.06 mari
it.inova.wfacile.webservice
AppServicePortType.java 274 13/01/12 0.30 mari
AppServicePortTypeProxy.java 276 1.0 mari
WebService.java 259 10/01/12 13.48 mari
it.inova.wfacile.webservice.model
it.inovaos.Marshaller
ObjectToSoapEquivalentUtil.java 268 mari
```

```
package it.inova.rubrica;

import it.inova.database.Database;
import java.util.ArrayList;
import java.util.Calendar;
...
Calendar cal=Calendar.getInstance();
int y=cal.get(Calendar.YEAR);

// (senza import sarebbe stato:
java.util.Calendar
    cal=java.util.Calendar.getInstance();
int y=cal.get(java.util.Calendar.YEAR);
```



# JAVA FOUNDATION PACKAGES

- Java fornisce un gran numero di classi raggruppate in packages diversi in base alle loro funzionalità offerte.
- I sei packages di base (foundation packages) offerti da Java sono:
  - `java.lang`
    - Contiene le classi per i tipi primitivi, le stringhe, le funzioni matematiche, i threads e le eccezioni.
  - `java.util`
    - Contiene classi come vettori, hash tables, date, etc.
  - `java.io`
    - Contiene classi per la gestione degli stream di I/O.
  - `java.awt`
    - Contiene classi per implementare le GUI
  - `java.net`
    - Contiene le classi per il networking
  - `java.applet`
    - Contiene le classi per creare e implementare le applets
- Il nucleo centrale del linguaggio Java è definito nel package `java.lang` che è importato automaticamente: la frase `import java.lang.*` è sottintesa.
  - ad esempio `System` è definita in `java.lang`, motivo per cui in `HelloWorld` non era presente alcun `import` esplicito
- La jdk fornisce più di 50 packages.



# **INPUT / OUTPUT**

**Programmare in Java, Capitolo 2, Capitolo 15, sezioni 15.1,  
15.2, 15.4**

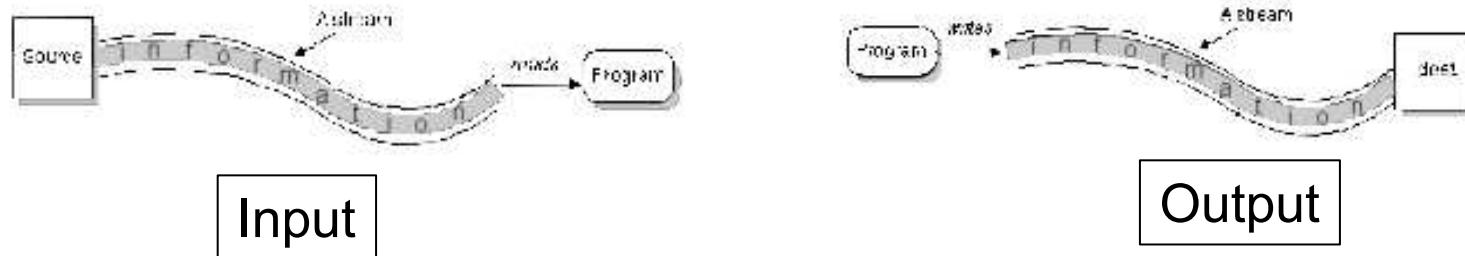


# INTERFACCIE UTENTE

- Le interfacce utente possono essere classificate in tre tipologie fondamentali:
  - Interfacce utente a carattere (CUI): sono le classiche interfacce utilizzate dalle shell dei sistemi operativi. In esse gli input arrivano tramite uno stream di input
  - Interfacce utente form-based: sono utilizzate in alcuni classici calcolatori IBM e in molti programmi vecchi (ad esempio il BIOS); sono ad esse analoghe anche le interfacce delle pagine web, almeno se escludiamo le interazioni con mouse o altri dispositivi di puntamento. Nelle interfacce form-based, gli input arrivano tramite uno stream nel quale ci sono sia caratteri di input che caratteri speciali (tabulazioni, backspace, tasti funzione, etc.)

# I/O CON STREAM

- Java supporta un ampio set di librerie di I/O.
  - Network, File, Screen (Terminal, Windows, Xterm), Screen Layout, Printer.
- In Java esiste il concetto di Stream
  - Di caratteri;
  - Di bytes;
- Uno stream è un canale di comunicazione tra un programma (Java) e una sorgente (destinazione) da cui importare (verso cui esportare) dati.
- L'informazione viene letta (scritta) serialmente, con modalità FIFO



# SCANNER

- La più semplice classe per l'input da tastiera (o da altri stream)
  - Scanner scanner=new Scanner(System.in);
    - Istanzia un oggetto scanner che si va a collegare allo stream di testo da tastiera (System.in)
  - **int n=scanner.nextInt();**
    - Legge un intero da tastiera
      - Se non inserisco un intero si crea una exception: se non viene gestita il programma va in crash
      - Riconosce come terminazione del numero intero un carattere di separazione come spazio, invio, tab ma non li elimina
  - **String s=new String(scanner.nextLine());**
    - Legge un'intera linea (cioè fino ad un invio) e mette il risultato in s
    - Attenzione: un nextLine subito dopo un nextInt leggerebbe semplicemente l'invio della linea nella quale c'era l'int



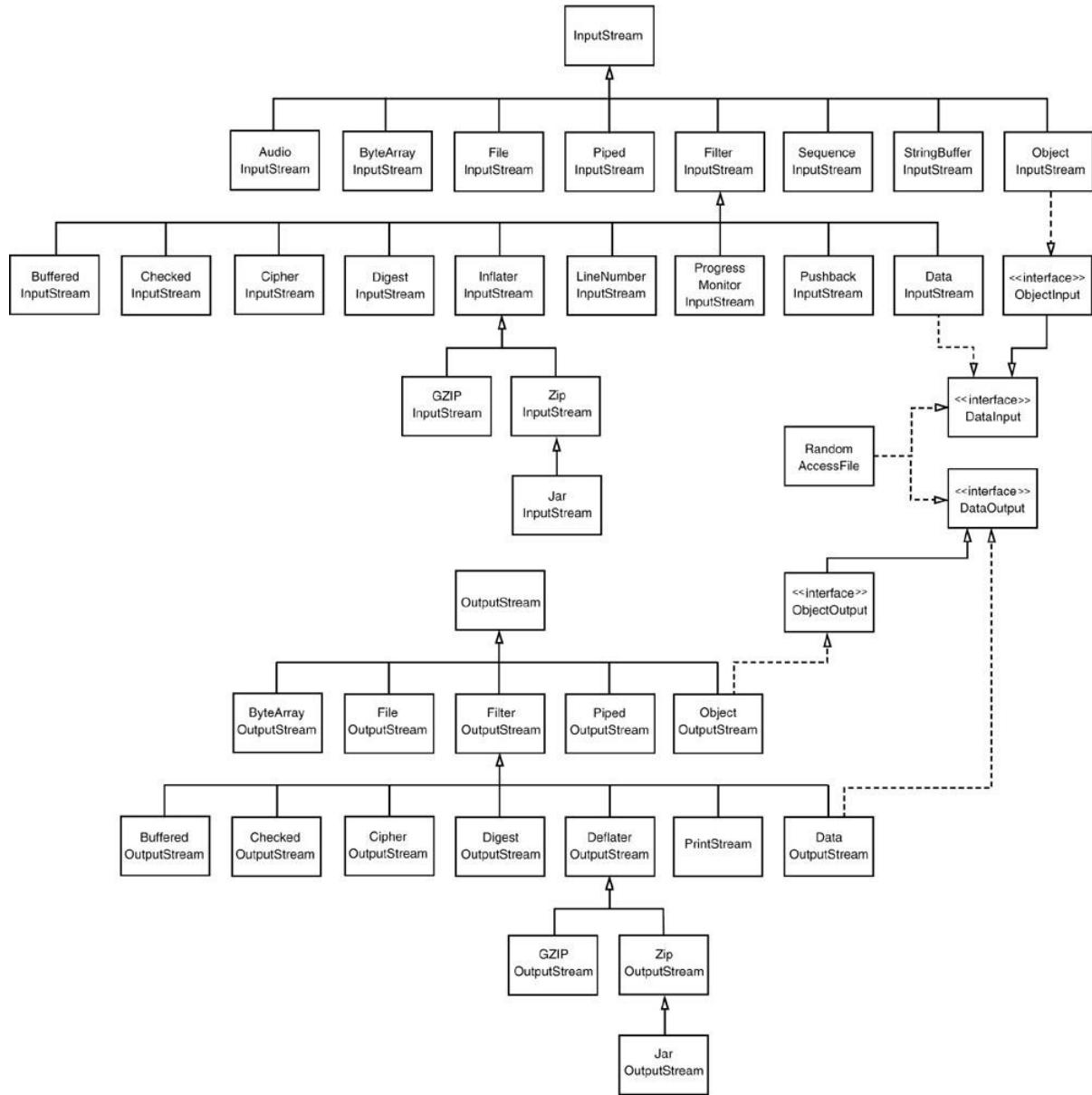
# ALTRÉ CLASSI PER L'INPUT E L'OUTPUT

- A partire da Scanner è stata realizzata un'intera gerarchia di classi per l'input/output da stream
- Ognuna di queste classi arricchisce in qualche modo le funzionalità di base a disposizione
- Siccome anche un file può essere visto come uno stream di input, le classi per la lettura da tastiera possono essere adattate ed utilizzate anche per la lettura da file
- Analogamente, le classi per la scrittura a video possono essere adattate e arricchite per utilizzarle nella scrittura su file



# GERARCHIA DELLE CLASSI DI I/O

- Le classi per la lettura e per la scrittura di un qualsiasi stream (console, file, ...) sono organizzate in un'unica gerarchia



# CLASSI INPUTSTREAM

- Passare attraverso più classi permette di «arricchire» l'offerta di metodi con i quali operare sullo stream
- Ad esempio
  - System.in è un oggetto della classe InputStream (<http://docs.oracle.com/javase/6/docs/api/java/io/InputStream.html>), per il quale è definita l'operazione `read()` che permette di leggere un byte
  - InputStreamReader (<http://docs.oracle.com/javase/6/docs/api/java/io/InputStreamReader.html>) ha un'operazione `read()` che permette di leggere un char
  - BufferedReader (<http://docs.oracle.com/javase/6/docs/api/java/io/BufferedReader.html>) ha anche un'operazione `readline()` che consente di leggere una riga intera (fino ad un ritorno da capo)



# ALTRÉ CLASSI PER L'INPUT E L'OUTPUT

```
BufferedReader in = new BufferedReader(new  
    InputStreamReader(System.in));  
  
String frase = in.readLine();  
int n=Integer.parseInt(in.readLine());
```

- Legge una frase e un intero (uno per linea) da tastiera



# I/O DA FILE (CENNO)

- Lettura di un file (e visualizzazione a video)

```
String s;  
  
BufferedReader reader = new BufferedReader( new FileReader("nomefile.txt") ) ;  
while( (s = reader.readLine()) != null )  
    System.out.println(s);  
reader.close();
```

- Scrittura di un file

```
PrintWriter writer = new PrintWriter( new BufferedWriter( new  
    FileWriter("nomefile.txt", true)) );  
  
writer.print("Testo ");  
writer.close()
```

- Da notare che la classe BufferedReader è la stessa utilizzata in precedenza per l'input da tastiera



# **CLASSI ASTRATTE, INTERFACCE E IMPLEMENTAZIONI**

Programmare in Java, Capitolo 10 da 10.1 a 10.6, e 10.9 e  
10.13



# CLASSI E METODI ASTRATTI (CENNO)

- Java non permette di separare l'interfaccia dall'implementazione come in C e C++, ma consente di definire metodi astratti (senza implementazione)
- Una classe che contiene metodi astratti è definita classe astratta
- Una classe astratta non può essere direttamente istanziata
- Una classe astratta può essere estesa da una classe che ne implementa i metodi astratti
- Una classe astratta può contenere anche attributi e metodi non astratti



# ESEMPIO

```
abstract class Base{
    abstract int m();
}

class Derivata extends Base{
    int m(){return 1};
}

Base b=new Derivata();
System.out.println(b.m());
```



# ESEMPIO

```
public abstract class ClasseAstratta {  
    abstract void stampa();  
    void chiSono() {  
        System.out.println("Sono la classe astratta");  
    }  
}  
  
public class ClasseCheEstendeLaAstratta extends ClasseAstratta {  
    void stampa() {  
        System.out.println("Stai stampando in una classe che estende la astratta");  
    }  
    @Override  
    void chiSono() {  
        System.out.println("Sono un oggetto della classe che estende la astratta");  
    }  
}
```



# ESEMPIO

ClasseAstratta astr= new ClasseAstratta(); → ERRORE: non si può istanziare ClasseAstratta

```
ClasseAstratta astr= new ClasseCheEstendeLaAstratta();
astr.stampa();
astr.chiSono();
ClasseCheEstendeLaAstratta astrEst=new ClasseCheEstendeLaAstratta();
astrEst.stampa();
astrEst.chiSono();
```

- Output:
  - Stai stampando in una classe che estende la astratta
  - Sono un oggetto della classe che estende la astratta
  - Stai stampando in una classe che estende la astratta
  - Sono un oggetto della classe che estende la astratta
- Il metodo chiSono di ClasseAstratta sarebbe stato eseguito solo se non fosse stato implementato anche da ClasseCheEstendeLaAstratta

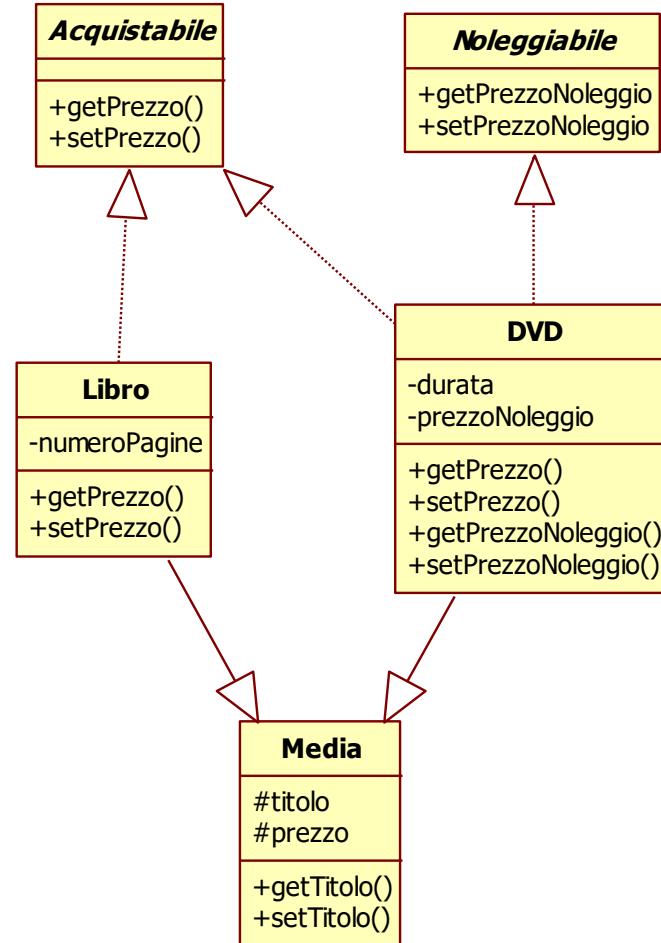


# INTERFACCE E IMPLEMENTAZIONE

- Una interfaccia è una classe **completamente** astratta
- Per definire un'interfaccia non si utilizza la parola *class* ma la parola **interface**
- Comprende metodi astratti e costanti
  - Una variabile dichiarata in una classe astratta deve essere vista come una costante (il modificatore *final* rimane sottinteso)
- L'ereditarietà che si viene ad instaurare tra una classe concreta ed una interfaccia è detta **implementazione** (parola chiave **implements**)
  - Un'interfaccia può anche essere *vuota*, ovvero senza metodi. In questo caso serve solo a ricordare una certa proprietà delle classi che la implementano
- In Java non è consentita l'ereditarietà multipla
  - Una classe non può estenderne più di un'altra
- Ma è consentita l'implementazione multipla
  - Una classe (che eventualmente già ne estenda un'altra) può implementare uno o più interfacce



# ESEMPIO



# ESEMPIO

```
public interface Acquistabile {  
    public void setPrezzo(double prezzo);  
    public double getPrezzo();  
}  
  
public interface Noleggiabile {  
    public void setPrezzoNoleggio(double  
        prezzo);  
    public double getPrezzoNoleggio();  
}  
  
public class Media {  
    protected String titolo;  
    protected double prezzo;  
    public void setTitolo(String titolo) {  
        this.titolo = titolo;  
    }  
    public double getTitolo() {  
        return titolo;  
    }  
}
```

```
public class Libro extends Media implements Acquistabile {  
    private int numeroPagine;  
    public void setPrezzo(double prezzo) {  
        this.prezzo = prezzo;  
    }  
    public double getPrezzo() {  
        return prezzo;  
    }  
}  
public class DVD extends Media implements Acquistabile,  
Noleggiabile {  
    private double durata;  
    private double prezzoNoleggio;  
    public void setPrezzo(double prezzo) {  
        this.prezzo = prezzo;  
    }  
    public double getPrezzo() {  
        return prezzo;  
    }  
    public void setPrezzoNoleggio(double prezzo) {  
        this.prezzoNoleggio = prezzo;  
    }  
    public double getPrezzoNoleggio() {  
        return prezzoNoleggio;  
    }  
}
```



# ESEMPIO

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

public interface Clonable{
    //serve a ricordare che è lecito utilizzare il metodo //Object.clone()
}

public class Rettangolo extends Forma implements Comparable, Clonable {
    public int compareTo(Object o) {
        //codice per il confronto
    }

    public Rettangolo clone(Rettangolo r) {
        //codice per la copia
    }

    //resto del codice della classe Rettangolo
}
```



# UTILIZZO DI INTERFACE

- Interface può essere utilizzato per separare la *dichiarazione* di una classe e dei suoi metodi dalla sua *implementazione*
- In questo modo si possono ottenere risultati simili a quelli ottenuti in C/C++:
  - Consentire di diffondere la conoscenza della dichiarazione dei metodi tenendo nascosta la loro implementazione
  - Separare il momento in cui si progettano I prototipi dei metodi da quello in cui si vanno a realizzare gli algoritmi risolutivi
- In aggiunta è possibile:
  - Avere diverse implementazioni possibili di una stessa interfaccia
    - Tecnica particolarmente utile nei test, per poter utilizzare realizzazioni temporanee semplificate allo scopo di poter provare parti del programma prima che sia del tutto concluso
    - Tecnica utilizzabile nel caso di accesso ad un database se vogliamo separare la parte dipendente da uno specifica tecnologia di database da quella generale



# **INTERFACCIA GRAFICA (GUI)**



# INTERFACCE GUI

- **Interfacce utente grafiche (GUI): sono le interfacce più utilizzate nei PC. In esse gli input arrivano tramite uno stream di eventi, comprendenti eventi da tastiera (tasti premuti), eventi da altri dispositivi di puntamento (mouse, touch pad, touch screen, etc.). Tutti questi eventi confluiscono in uno stream, nel quale vengono interpretati e vengono riconosciuti eventi di più alto livello.**
  - Ad esempio un doppio click si ottiene da uno stream di eventi nel quale si notano due coppie di operazioni di prenota e rilascio del pulsante sinistro del mouse, avvenute a distanza ravvicinata di tempo e su pixel ravvicinati sullo schermo

# SISTEMI BASATI SUGLI EVENTI

- **Le interfacce utente grafiche rappresentano un caso di sistema ad eventi**
  - Il sistema è in uno stato stabile fino all'intervenire di un evento utente, che fa partire un codice di event handling (ascoltatore o listener)
- **Anche un calcolatore, dotato di sistema delle interruzioni, può essere considerato come un sistema ad eventi**
  - Un segnale è in grado di avviare un'interruzione, che viene successivamente servita
- **I moderni sistemi distribuiti a sensori devono essere considerati sistemi ad eventi**
  - Ogni dispositivo è in grado sia di ricevere input, che di elaborarli

# GUI DI APPLICAZIONI JAVA

- In Java convivono diverse librerie di base per realizzare GUI:
  - AWT (Abstract Windowing Toolkit – `java.awt`)
    - Semplice ma primitivo
  - **Swing** (`javax.swing`)
    - Più complesso e ricco di funzionalità
    - Estende AWT
  - JavaFX
    - Proposto più recentemente
- Verrà presentata Swing, con l'utilizzo di alcune operazioni di base da Awt



# PROGETTO GRAFICO DELLA GUI

- Uno strumento visuale molto utile nella progettazione e implementazione delle GUI in Java è Window Builder Pro, realizzato da Google
  - <https://developers.google.com/java-dev-tools/wbpro/>
  - Il plug-in per Eclipse si può scaricare tramite eclipse da:
    - <https://download.eclipse.org/windowbuilder/latest/> è un'estensione di Eclipse che aggiunge tutta una serie di opzioni, pulsanti e finestre che aiutano durante la realizzazione di una GUI
    - fornisce un IDE per la progettazione grafica degli elementi della GUI (gestendone anche il layout)
    - Genera automaticamente codice sorgente Java corrispondente all'interfaccia progettata visualmente



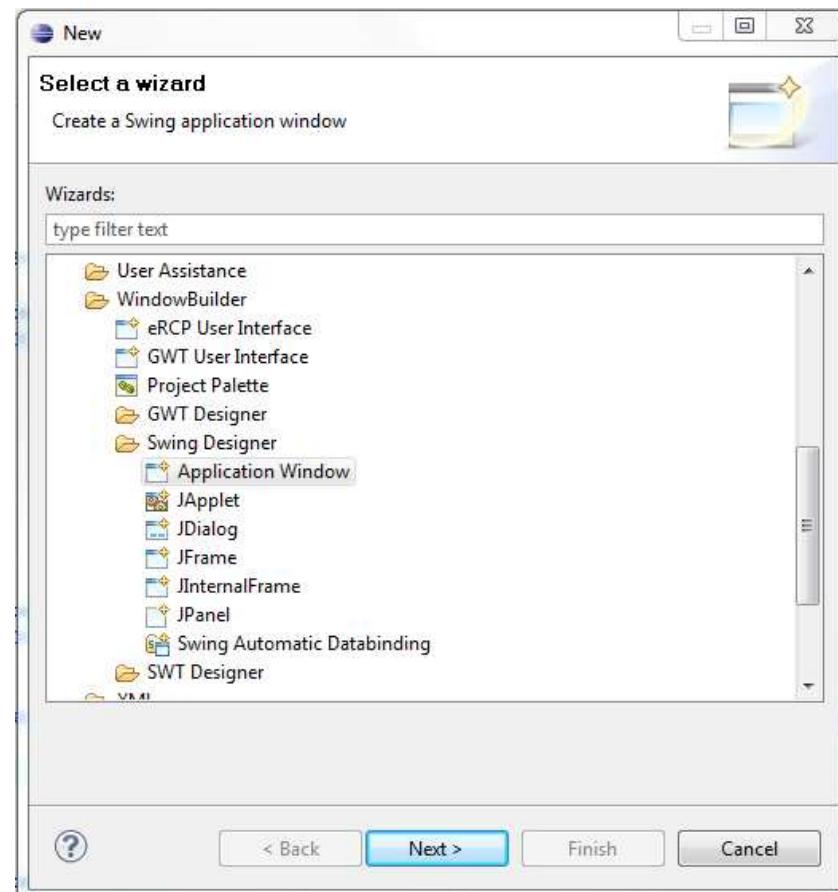
# INSTALLAZIONE DI WINDOW BUILDER

- In Eclipse
  - Menu Help
  - Install New Software ...
  - Add ...
    - Scegliere un nome a piacere (ad es. Window Builder)
    - Location : <https://download.eclipse.org/windowbuilder/latest/>
  - Add
  - Selezionare WindowBuilder (mettendo la spunta)
  - Next >
  - Finish
  - Accettare la richiesta di riavvio di Eclipse



# USO DI WINDOW BUILDER

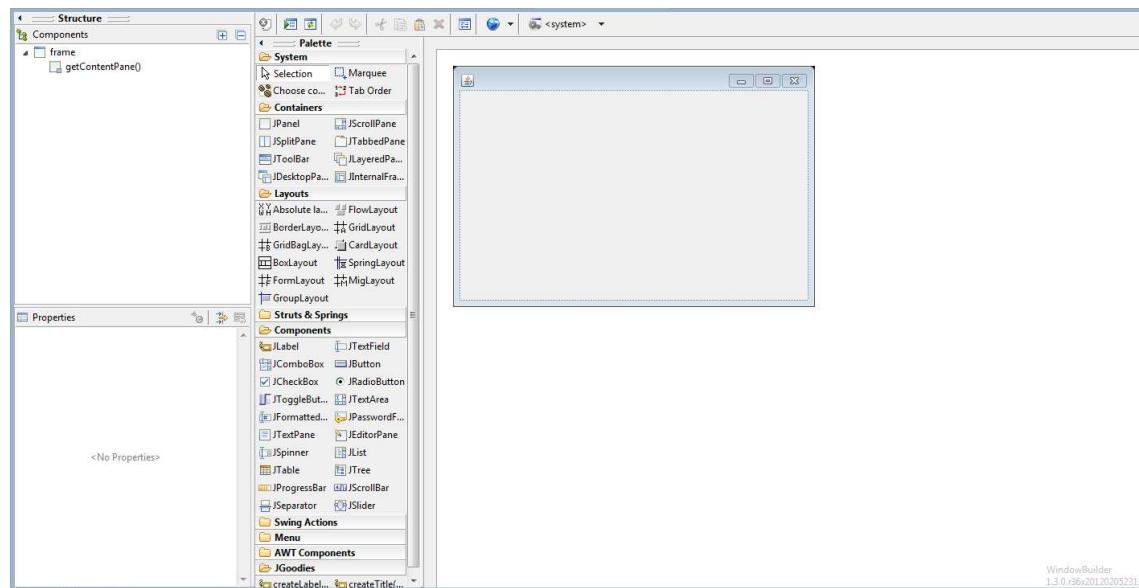
- Per creare una classe corrispondente ad un oggetto della GUI basta scegliere New→Other
- Vengono proposte come alternative (swing) i possibili Container di widget grafici
- Il punto di forza di Window Builder è che consente di disegnare visualmente delle GUI e contemporaneamente traduce il disegno in codice Java che possiamo modificare
  - E' in grado, limitatamente, anche di ricavare il disegno a partire da codice Java



# FINESTRA DI DESIGN

- Window Builder consente una progettazione duale delle classi GUI, in modalità codice sorgente e in modalità design

```
1 package Boundary;
2
3 import java.awt.EventQueue;
4
5 public class MainWindow {
6
7     private JFrame frame;
8
9     /**
10      * Launch the application.
11      */
12
13     public static void main(String[] args) {
14         EventQueue.invokeLater(new Runnable() {
15             public void run() {
16                 try {
17                     MainWindow window = new MainWindow();
18                     window.frame.setVisible(true);
19                 } catch (Exception e) {
20                     e.printStackTrace();
21                 }
22             }
23         });
24     }
25
26
27     /**
28      * Create the application.
29      */
30     public MainWindow() {
31         initialize();
32     }
33
34     /**
35      * Initialize the contents of the frame.
36      */
37     private void initialize() {
38         frame = new JFrame();
39         frame.setBounds(100, 100, 450, 300);
40         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
41     }
42
43 }
```



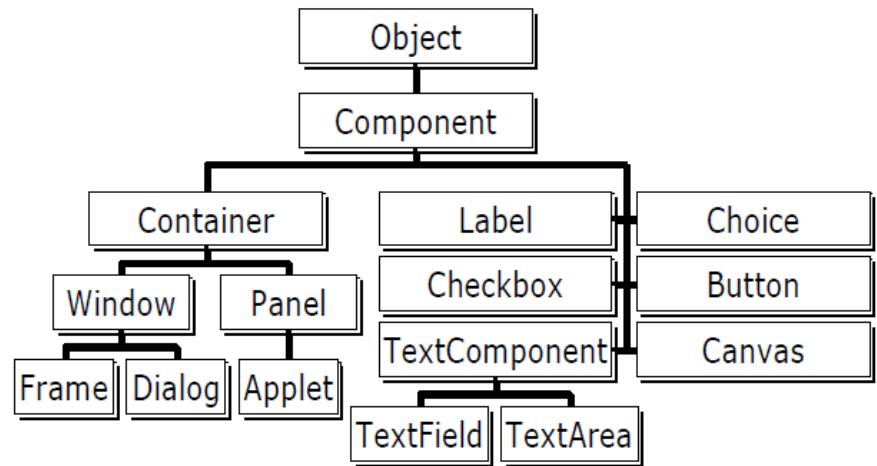
# COMPATIBILITÀ

- L'attuale versione dell'editor associato a Window Builder è compatibile con Java 14
- Nel caso si abbia una versione di Java successiva (ad es. 16) bisogna cambiare le impostazioni del singolo Progetto:
  - Click destro sul Progetto
  - Properties
  - Java Compiler
  - Enable project specific setting
  - Compiler Compliance Level: 14
  - Apply and Close



# COMPONENTI GRAFICI

- Si distingue tra:
  - **Container**, che raggruppano grafica
  - **Window, Frame, Panel, Dialog**
  - **Widget** elementari
    - **Label, Button, ...**
- In aggiunta, esistono numerosi classi di layout per specificare le posizioni assolute e relative dei componenti e dei contenitori



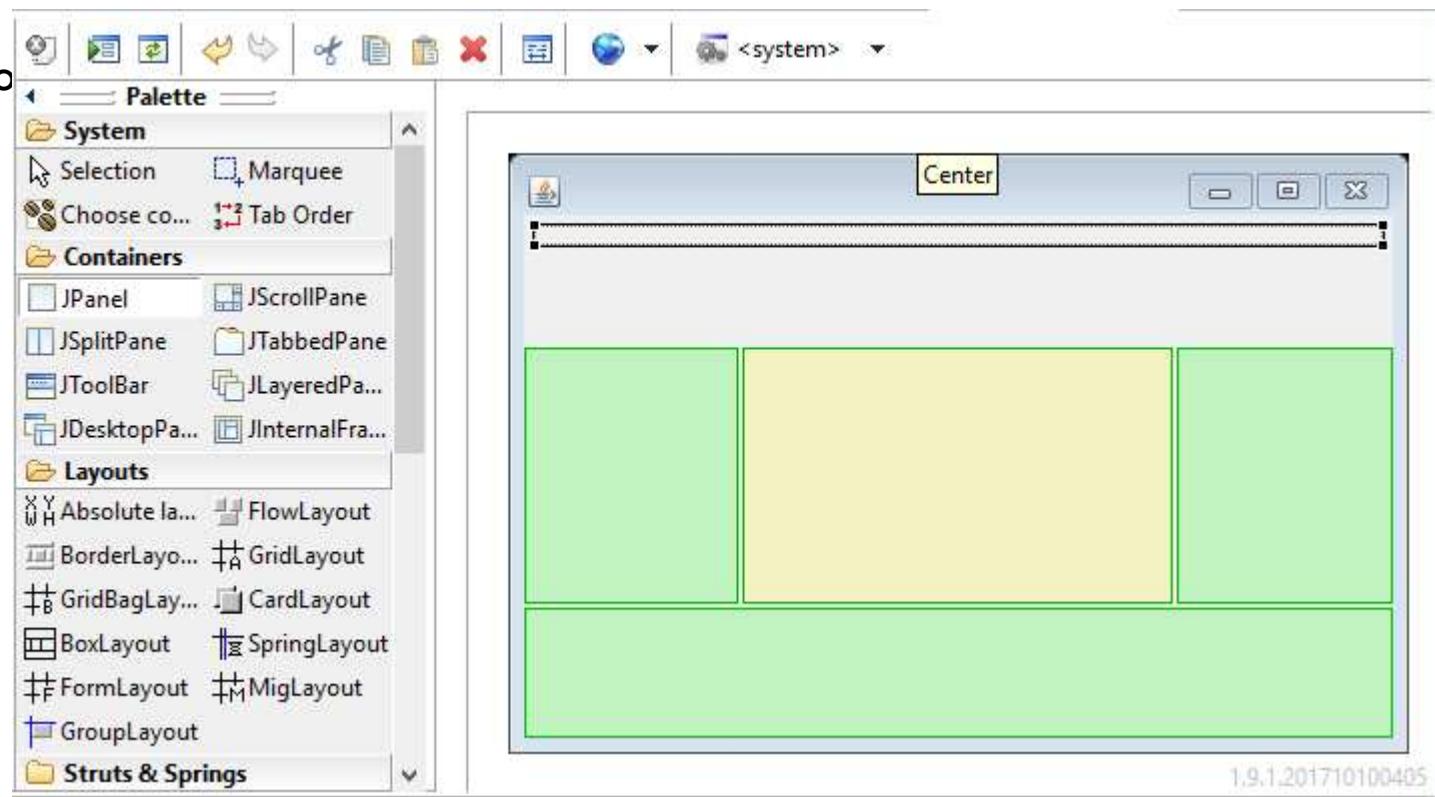
# ESEMPI DI CONTENITORI

- **Window**, che rappresenta una finestra di un'applicazione, così come vista dal sistema operativo
  - **JFrame**, contenuta in una Window, che può essere decorata con menu, bordi, pulsanti di ingrandimento, riduzione a icona, chiusura, etc.
  - **JPanel**, contenuto in un JFrame. In ogni JFrame ci possono essere uno o più Panel di forma e dimensioni tra loro indipendenti
- 
- <https://docs.oracle.com/javase/tutorial/uiswing/components/toplevel.html>
  - <http://www.i-programmer.info/ebooks/modern-java/5041-building-a-java-gui-containers.html>



# ESEMPIO: DISEGNO DI UN'INTERFACCIA BASILARE

- Disegniamo



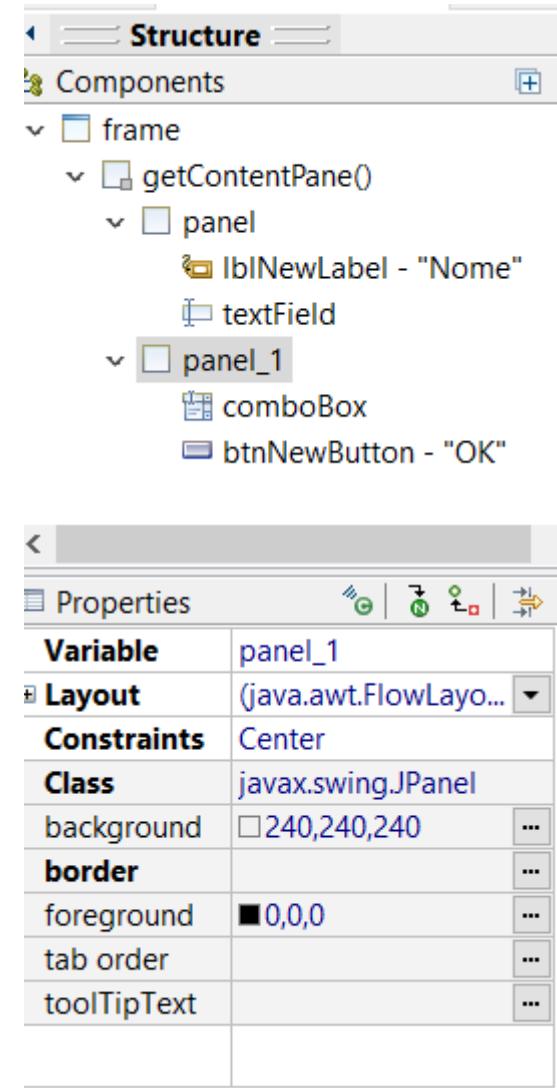
# LAYOUT

- All'interno di un contenitore (ad esempio un JPanel) possono esserci molti widget. Come posizionarli?
  - Bisogna indicare una *strategia di posizionamento* (**layout**)
- Le strategie più semplici sono:
  - **Absolute Layout**
    - Ogni widget ha la sua posizione in pixel: generalmente sconsigliato perché diventa inutilizzabile appena si prova a cambiare la dimensione della finestra (oppure se si utilizza uno schermo di diverse dimensioni)
  - **Flow Layout**
    - I widget sono posizionati logicamente uno dopo l'altro, facendo occupare ad ognuno di essi lo spazio che gli necessita (che può essere impostato dale proprietà del widget)
    - Altri layout più complessi sono disponibili, che consentono di disporre gli elementi ad esempio in griglie, schede o form



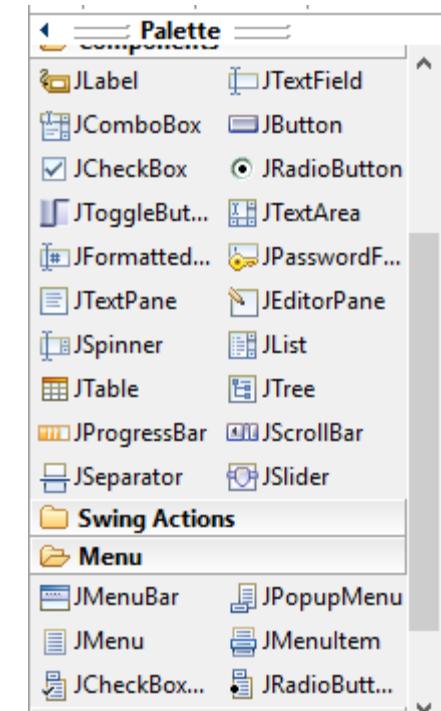
# LAYOUT

- E' sempre consigliato inserire un layout in ogni contenitore
- Se vogliamo avere diverse zone della JFrame con diversi layout, allora utilizziamo diversi JPanel, eventualmente anche uno dentro l'altro
- Dal riquadro delle Properties possiamo leggere o modificare le caratteristiche di ogni widget o container
  - In questo caso vediamo il layout abbinato a un JPanel
- Dal riquadro dei components possiamo vedere graficamente come sono innestati frame, panel e widget



# ESEMPI DI WIDGET

- **JLabel**: un'etichetta di testo
- **JTextField**: una casella tramite cui inserire un testo
- **JTextArea**: una casella tramite cui inserire un testo lungo (più righe)
- **JComboBox**: una casella tramite cui inserire un valore appartenente ad un elenco di valori possibili
- **JCheckbox**: una casella che può essere segnata oppure no
- **JRadioButton**: un insieme di caselle solo una delle quali può essere segnata

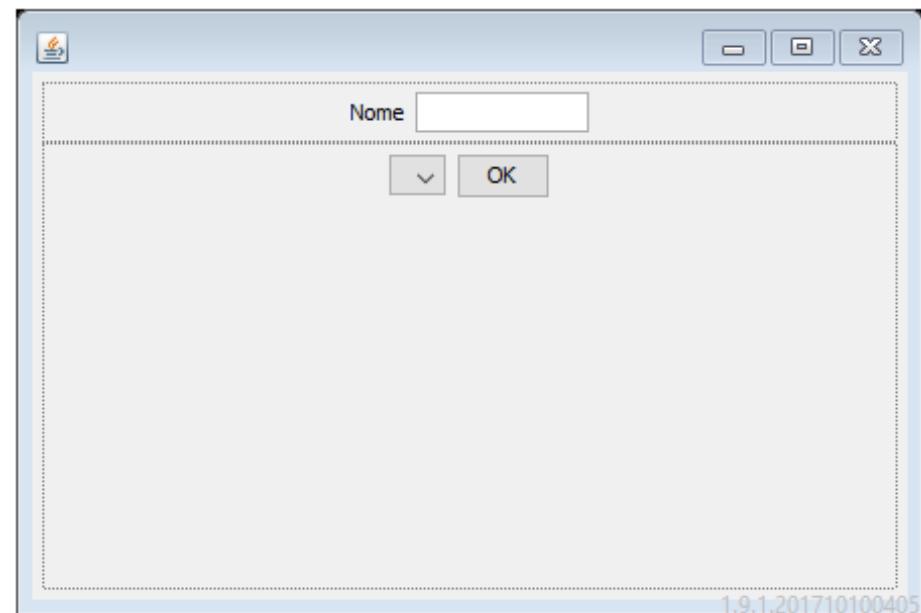


- <https://docs.oracle.com/javase/tutorial/uiswing/components/button.html>



# ESEMPIO

Disegniamo un testo in un panel, una casella con elenco di scelte (ComboBox) e un pulsante (Button) in un altro panel dello stesso Frame



# ESEMPIO

Osserviamo il codice generato:

- Widget e panel sono dichiarati private oppure come variabili locali del costruttore
- Il primo JFrame dichiarato rappresenta l'intera window e ha indirizzi assoluti
- I metodi add ci fanno vedere come i widget sono stati inseriti nei panel

```
public class ProvaGUI extends JFrame {  
    public ProvaGUI() {  
  
        frame = new JFrame();  
        frame.setBounds(100, 100, 860, 424);  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        JPanel panel = new JPanel();  
        frame.getContentPane().add(panel, BorderLayout.NORTH);  
        panel.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));  
  
        JLabel lblNewLabel = new JLabel("Nome");  
        panel.add(lblNewLabel);  
  
        JTextField textField = new JTextField();  
        panel.add(textField);  
        textField.setColumns(10);  
  
        JPanel panel_1 = new JPanel();  
        frame.getContentPane().add(panel_1, BorderLayout.CENTER);  
        panel_1.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));  
  
        JComboBox comboBox = new JComboBox();  
        panel_1.add(comboBox);  
  
        JButton btnNewButton = new JButton("OK");  
        panel_1.add(btnNewButton);  
    }  
}
```



# ESEMPIO

- In aggiunta, viene creato un metodo che consente di vedere la window creata come entry point (main) del programma :

```
public static void main(String[] args) {  
    EventQueue.invokeLater(new Runnable() {  
        public void run() {  
            try {  
                ProvaGUI window = new ProvaGUI();  
                window.frame.setVisible(true);  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
        }  
    });  
}
```



# INIZIALIZZAZIONE

- WindowBuilder genera un metodo main per ogni container, che ne rende possibile l'istanziazione
  - Tale main comprende un metodo run(), che ne consente l'esecuzione, eventualmente concorrente
- Un Widget grafico può avviare un altro semplicemente istanziandolo come oggetto
  - Il metodo setVisible() consente di mostrare sullo schermo il widget istanziato
- Window Builder genera anche un metodo initialize() che contiene il codice per l'istanziazione, il posizionamento e l'inizializzazione di ogni widget grafico contenuto nel container

```
InputGUI frame = new InputGUI();  
frame.setVisible(true);
```



# MODIFICARE UN WIDGET

- Per modificare un widget è necessario vedere quali sono i suoi campi e quali metodi sono messi a disposizione per modificarli
- Esempio: vogliamo aggiungere due valori Maschio e Femmina nella combobox, tra i quali poter scegliere:

```
JComboBox comboBox = new JComboBox();  
comboBox.addItem("M");  
comboBox.addItem("F");  
panel_1.add(comboBox);
```



# ANTEPRIMA ED ESECUZIONE

- Window Builder fornisce una funzionalità per l'anteprima della GUI, tramite la quale possiamo valutare come appare e se funzionano cambi di scheda, risposta multipla, etc.
- Per valutare il comportamento invece possiamo far partire l'intero programma



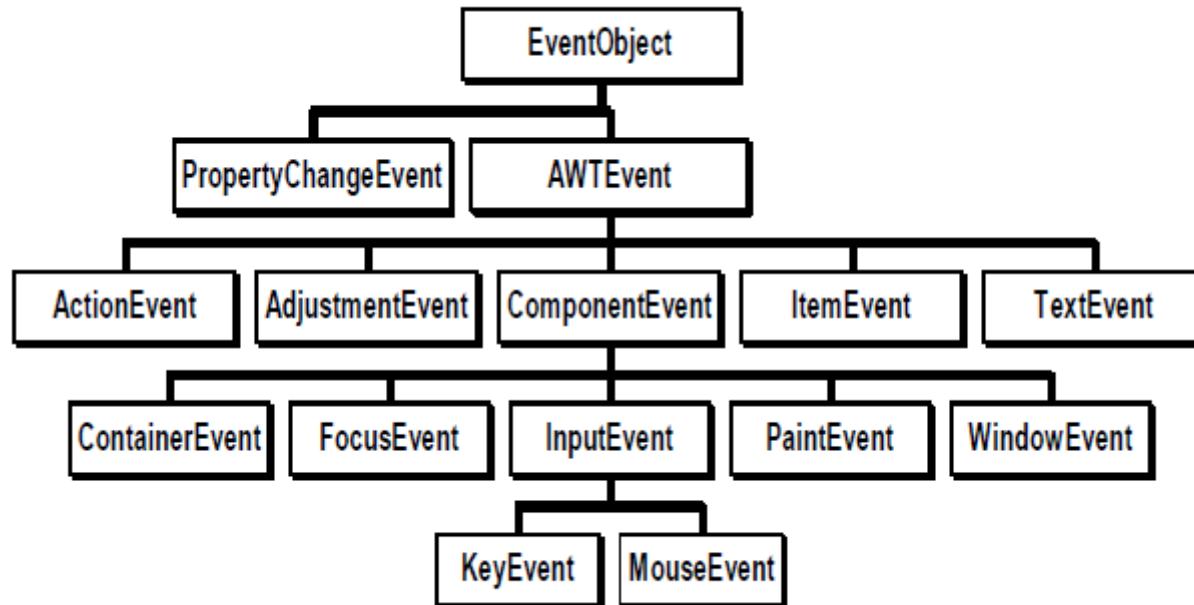
# EVENTI E ASCOLTATORI

- La GUI deve essere programmata per *reagire* ad azioni dell'utente (*eventi*)
- Il programmatore in generale non può prevedere nè imporre quale sia il prossimo evento eseguito dall'utente, cosicchè non possiamo avere un codice composto di un unico algoritmo sequenziale
- Bisogna realizzare un algoritmo in risposta ad ogni evento dell'utente
  - Quest'algoritmo viene associato ad un *ascoltatore* (*listener*) che viene attivato automaticamente dalla JVM quando si verifica l'evento
- Il Sistema di ascolto degli eventi somiglia molto al Sistema di gestione delle interruzioni
  - Il processore è sempre pronto a servire le interruzioni che si verificano in maniera imprevedibile



# EVENTI

- Un evento è un oggetto di una classe che eredita da Event, che viene riconosciuto automaticamente dalla JVM (analogamente ad una Exception)



# ASCOLTATORI (LISTENER)

- Un enorme numero di interface EventListener sono disponibili in Java per essere implementate
  - <https://docs.oracle.com/javase/7/docs/api/java/util/EventListener.html>
- Ad esempio MouseListener dichiara i seguenti metodi:
  - `void mouseClicked(MouseEvent e)`
    - Invoked when the mouse button has been clicked (pressed and released) on a component.
  - `void mouseEntered(MouseEvent e)`
    - Invoked when the mouse enters a component.
  - `void mouseExited(MouseEvent e)`
    - Invoked when the mouse exits a component.
  - `void mousePressed(MouseEvent e)`
    - Invoked when a mouse button has been pressed on a component.
  - `void mouseReleased(MouseEvent e)`
    - Invoked when a mouse button has been released on a component.
- <https://docs.oracle.com/javase/7/docs/api/java/awt/event/MouseListener.html>



# ESEMPIO CON MOUSECLICKED

```
JButton cancelButton = new JButton("Cancel");
cancelButton.addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent arg0) {
        setVisible(false);
    }
});
buttonPane.add(cancelButton);
```

- `new MouseAdapter()` causa l'istanziazione di un oggetto della classe `MouseAdapter`
- Ma la classe `MouseAdapter` è astratta, per cui può essere prima completata fornendo implementazioni per i suoi metodi
- `setVisible(false)` è l'implementazione del metodo astratto `mouseClicked` di `MouseAdapter`
- La classe «`MouseAdapter con implementazione di mouseClicked`» rimane una classe anonima, dichiarata al solo scopo di istanziarne un unico oggetto (pure lui anonimo) con `new`
- L'oggetto di questa classe anonima viene passato a `cancelButton` con il metodo `addMouseListener`
- Da questo momento in poi, se la JVM intercetta un click del mouse sul button `cancelButton`, esegue immediatamente il metodo `mouseClicked` definito



# EVENTI

- Nell'esempio precedente l'oggetto MouseEvent arg0 è un parametro di input di mouseClicked e ci fornisce alcune informazioni
  - Ad esempio arg0.getButton() ci dice se è stato cliccato il pulsante sinistro, destro o centrale
    - [https://docs.oracle.com/javase/7/docs/api/java.awt.event/MouseEvent.html](https://docs.oracle.com/javase/7/docs/api/java.awt/event/MouseEvent.html)
- La soluzione più generale è quella di implementare un oggetto dell'interfaccia ActionListener e in esso il metodo

```
void actionPerformed(ActionEvent e)
```
- Analizzando l'oggetto e possiamo risalire a quale evento (mouse, tastiera o qualsiasi altra cosa) sia realmente avvenuto
  - <https://docs.oracle.com/javase/tutorial/uiswing/events/actionlistener.html>



# MENU

- Rappresentano una soluzione molto comoda per aggiungere comandi legati ad una window e non necessariamente all'interno dei panel
- Per inserire una voce di menu è necessario creare:
  - Un **JMenuBar**, rappresenta la barra contenente i menu
  - Un **JMenu** all'interno della JMenuBar, che rappresenta i menu contenuti nella barra
  - Un **JMenuItem** all'interno del JMenu, che rappresenta una voce di menu
  - Un ascoltatore (ad esempio **ActionListener**) abbinato al JMenuItem



# MENU: ESEMPIO

```
JMenu mnFile = new JMenu("File");  
menuBar.add(mnFile);  
  
JMenuItem mntmQuit = new JMenuItem("Quit");  
mntmQuit_1.addMouseListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        System.exit(0);  
    }  
});  
mnFile.add(mntmQuit);
```

- Crea un menu (File) con una voce (Quit)
- Associa il codice System.exit(0) all'evento pressione del pulsante del mouse sulla voce del menu
- <https://docs.oracle.com/javase/tutorial/uiswing/components/menu.html>



# FINESTRE MODALI: DIALOG

- Rappresentano una soluzione molto comoda, sia dal punto di vista della programmazione che del testing, per ottenere singoli input
- `JOptionPane.showMessageDialog(frame, testo);`
  - Mostra una finestra di dialogo con il testo in input nel contesto del frame (quello aperto) e un pulsante OK: l'utente dovrà per forza premere su OK prima di fare qualsiasi altra cosa
- `String inputValue = JOptionPane.showInputDialog("Input a value");`
  - Mostra una finestra di dialogo con un testo e una casella di testo nella quale inserire una stringa che verrà restituita in inputValue
- Altri esempi:  
<https://docs.oracle.com/javase/7/docs/api/javax/swing/JOptionPane.html>



# COMUNICAZIONE TRA LE FINESTRE

- Si potrebbe implementare qualsiasi interfaccia utente con un unico JFrame e una serie di JPanel che compaiono e si nascondono durante l'esecuzione
- Questa implementazione è sconsigliata poichè diventerebbe insostenibile all'aumentare della complessità della GUI, sia per ragioni di memoria che di complessità del codice
- Si preferisce, invece, avere dei frame dal disegno statico, che una volta istanziati ed utilizzati per il loro scopo vengono distrutti
- Per comunicare informazioni tra una GUI e l'altra, quindi, è necessario passare esplicitamente i dati necessari da un JFrame all'altro, in maniera unidirezionale



# COMUNICAZIONE TRA JFRAME: ESEMPIO

- Una tecnica semplice:
  - C'è un frame *chiamante* che passa il controllo ad un frame *chiamato*, che in quel momento diventa visible, mentre il frame chiamante diventa invisibile
    - `JFrame frameChiamato=new frameChiamato(frameChiamante);`
      - Fondamentale passare il riferimento al frameChiamante, altrimenti non sarà possibile tornare indietro
    - `frameChiamante.setVisible(false);`
    - `frameChiamato.setVisible(true);`
  - Al termine delle elaborazioni sul frame chiamato vogliamo che esso passi il controllo al frame chiamante e venga distrutto (in modo da poter essere successivamente deallocated)
    - `frameChiamante.setVisible(true);`
    - `frameChiamato.setVisible(false); //non necessario`
    - `frameChiamato.dispose();`



# COMUNICAZIONE TRA JFRAME: PASSAGGIO DEI DATI

- Nell'esempio precedente veniva trasferito il controllo tra due JFrame ma non venivano passati dati
- I dati potrebbero essere passati:
  - Sotto forma di oggetti nella chiamata al costruttore di Jframe
    - `JFrame frameChiamato=new frameChiamato(frameChiamante, String dato, String risultato);`
- In alternativa, preferiremo il passaggio di un unico riferimento ad un oggetto che abbia la capacità di poter leggere e modificare tutti i dati del programma
  - Questo oggetto verrà denominato per semplicità **Controller**
    - `JFrame frameChiamato=new frameChiamato(controller, frameChiamante);`



# PRINCIPI DI PROGETTAZIONE DELLA GUI

- Esistono diversi tipi di architetture software, con diversi vincoli ed obiettivi, che prevedono la realizzazione di più o meno funzionalità nella GUI
- Nella realizzazione dei sistemi informativi di questo corso seguiremo I seguenti principi:
  - **1) Leggerezza della GUI**
  - **2) Separazione tra i package**
  - **3) Comunicazione tra le finestre**



# LEGGEREZZA DELLA GUI

- Le classi della GUI conterranno il minimo di codice ed elaborazioni possibile: in particolare le elaborazioni saranno quasi tutte demandate alle classi di controllo che faranno da raccordo
  - Nell'ambito delle classi della GUI verranno letti i dati in input provenienti dai widget della GUI e verranno chiamate le funzioni che avviano le elaborazioni ( contenute in un package di controllo)
  - Eccezionalmente potrebbero essere programmate nelle classi della GUI le più semplici validazioni della correttezza dei dati
- In alcuni ambienti questa scelta viene fatta anche per limiti tecnici dell'ambiente nel quale l'interfaccia viene eseguita
- In altri casi invece si deroga questa regola allo scopo di eseguire alcune elaborazioni e validazione anticipatamente



# SEPARAZIONE TRA I PACKAGE

- Le classi della GUI verranno inserite tutte in uno (o più) package nei quali non ci saranno altre classi che non siano dedicate alla gestione dell'interfaccia utente
- I motivi per cui questa suddivisione viene effettuata derivano dalla possibilità di suddividere il lavoro di sviluppo sia nel tempo che tra i programmatore, per poter:
  - anticipare (o posticipare) lo sviluppo della GUI rispetto allo sviluppo di altre parti del software
  - Affidare lo sviluppo della GUI a specialisti
    - In alcuni ambienti viene anche separato il disegno della GUI, affidato a grafici, dalla sua programmazione
  - Consentire la possibilità di avere più implementazioni alternative della GUI (ad esempio per diversi schermi o combinazioni di colori)



# COMUNICAZIONE TRA LE FINESTRE

- Lo stile architetturale che adotteremo in questo corso prevede che le comunicazioni tra le finestre riguardino solo il passaggio del *controllo* tra una finestra e l'altra
  - Un JFrame che ne aprirà un altro, quindi, passerà il riferimento a sè stesso in modo da poter essere successivamente restituito il controllo
- Non c'è invece passaggio diretto di dati tra finestre
  - Fanno eccezione solo i dialoghi JOptionPane.showInputDialog
- La logica di funzionamento è tutta delegata a classi del package controller
  - In questo corso, per semplicità, supporremo che il package controller abbia una sola classe Controller il cui unico oggetto sia istanziato all'avvio della GUI e il cui riferimento sia passato da una finestra all'altra
- Per ottimizzare l'occupazione di memoria, I JFrame che rilasciano il controllo dovrebbero sempre essere distrutti (con il metodo dispose)



# **RUOLO DEL CONTROLLER**

- La classe Controller nel package controller genera un unico oggetto controller al momento dell'avvio dell'applicazione, che :
  - Viene chiamato da tutte le classi della GUI quando vogliono essere fornite il valore di un dato
  - Chiama tutte le classi del package Model per inoltrare richieste di lettura o modifica di un qualsiasi dato
  - Non viene mai distrutto: scompare alla chiusura del programma



# FILE JAR

- Jar sta per Java ARchive
- Un file jar è il risultato della compressione dei bytecode di tutte le classi di un package, eventualmente con l'aggiunta anche di codice sorgente, documentazione ed altro
- La compressione in un archivio jar è, quindi, il modo migliore per esportare un package
- E' possibile eseguire direttamente un programma da un jar
  - `java -jar nome_package.jar`
  - Bisogna, però, dichiarare quale classe sia quella di partenza (static void main)
- Per specificare in quali cartelle un progetto debba cercare le classi, da sistema operativo si setta una variabile CLASSPATH
  - Set CLASSPATH = .;c:\Hello.jar; ...



# RUNNABLE JAR

- E' possibile generare un jar da Eclipse con File → Export → Java → JAR o Runnable JAR
- Un JAR è semplicemente una libreria contenente un insieme di classi compilate che possono essere riutilizzate da un qualsiasi altro programma
- Un Runnable JAR invece è un programma eseguibile, dotato di uno o più punti di partenza (entry point)
  - Nella creazione di un Runnable JAR dobbiamo specificare quale sia il metodo static di partenza (uno degli static void main, ad esempio)
  - Per eseguire il JAR così creato dal Sistema operativo sarà sufficiente un doppio click
    - Equivalente a `java -jar programma.jar`
  - Aprendo il jar con un programma di compressione (ad es. Winzip) sarà possibile vedere tutti i bytecode (file .class) e un file Manifest.mf nel quale è scritto quale sia l'entry point



# **DOCUMENTAZIONE INTERNA DEL CODICE**



# RIFERIMENTI

- Javadoc, <http://java.sun.com/j2se/javadoc/>
  - <http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/javadoc.html#tags>
  - <http://java.sun.com/j2se/javadoc/doccheck/>
  - <https://www.oracle.com/it/technical-resources/articles/java/javadoc-tool.html>



# STANDARD DI DOCUMENTAZIONE

- La qualità del software è direttamente o indirettamente influenzata in maniera positiva dalla presenza di documentazione
  - In particolare, la *tracciabilità* della documentazione nel codice o nell'architettura riduce notevolmente lo sforzo legato al processo di comprensione del software
- L'utilizzo di standard di documentazione noti presenta diversi vantaggi:
  - Semplicità di scrittura della documentazione;
  - Possibilità di valutare e verificare velocemente la completezza della documentazione;
  - Possibilità di generare automaticamente manuali utente e diagrammi statici di dettaglio



# QUALITÀ DELLA DOCUMENTAZIONE DEL CODICE

- La documentazione interna al codice dovrebbe essere:
  - Coerente
  - Consistente
    - Non devono esserci ambiguità o contraddizioni
  - Conforme ad uno standard
  - Tracciabile
    - Deve essere possibile poter collegare, nella maniera più rapida possibile, i concetti presenti nel codice con la loro documentazione e con i concetti ad esso associati



# STANDARD DI DOCUMENTAZIONE DEL CODICE

- Tra gli innumerevoli standard di documentazione esistenti, verrà presentato Javadoc
  - Ideato per Java
  - Affiancato da un tool (javadoc) in grado di generare manualistica a partire dall'analisi della documentazione presente nel codice sorgente
  - Generalizzabile a qualsiasi altro linguaggio
- Javadoc è un linguaggio nel linguaggio
  - Le righe di codice Javadoc sono in realtà delle righe di commento
    - Sono quindi ignorate dal compilatore Java
    - Sono considerate, invece, dal compilatore Javadoc
  - Le righe di codice Javadoc sono interpretate in funzione della loro posizione
    - Ad esempio subito prima della definizione di una classe, di un metodo, etc.



# ESEMPIO JAVADOC

Inizio commento per Javadoc

```
/**  
 * Returns an Image object that can then be painted on the screen.  
 * The url argument must specify an absolute {@link URL}. The name  
 * argument is a specifier that is relative to the url argument.  
 * <p> ← Nuova linea nell'HTML risultante  
 * This method always returns immediately, whether or not the  
 * image exists. When this applet attempts to draw the image on  
 * the screen, the data will be loaded. The graphics primitives  
 * that draw the image will incrementally paint on the screen.  
 *  
 * @param url an absolute URL giving the base location of the image ← Parametri  
 * @param name the location of the image, relative to the url argument  
 * @return the image at the specified URL ← Valore di ritorno  
 * @see Image ← Riferimento  
 */  
  
public Image getImage(URL url, String name) {  
    try {  
        return getImage(new URL(url, name)); ← Codice del metodo  
    } catch (MalformedURLException e) {  
        return null;  
    }  
}
```

URL diventerà un link nella documentazione HTML

Breve riassunto dello scopo e dei parametri del metodo  
(Description Block)

Parametri

Valore di ritorno

Codice del metodo

# HTML RISULTANTE

## **getImage**

```
public Image getImage(URL url,  
                      String name)
```

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute [URL](#). The `name` argument is a specifier that is relative to the `url` argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

**Parameters:**

`url` - an absolute URL giving the base location of the image  
`name` - the location of the image, relative to the `url` argument

**Returns:**

the image at the specified URL

**See Also:**

[Image](#)



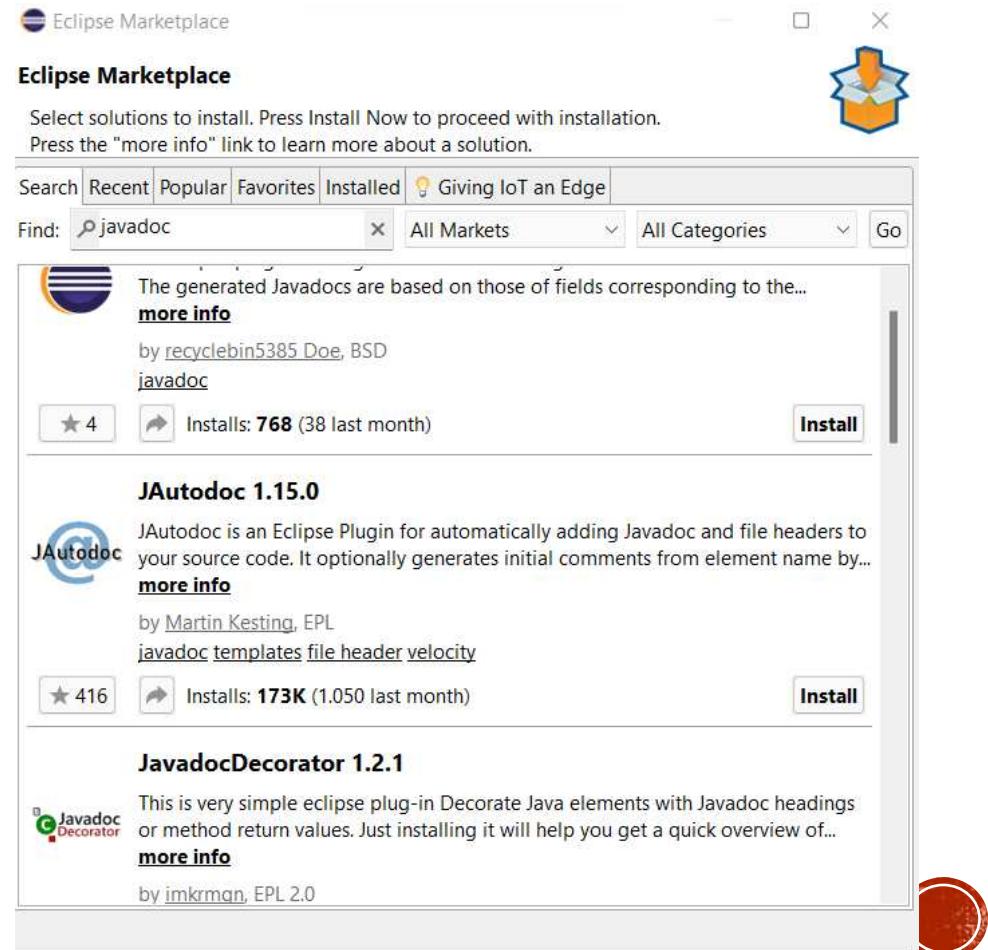
# PRINCIPALI TAG JAVADOC

- **@author** *name-text*
- **@deprecated** *deprecated-text*
- **{@code** *text***}**
- **@exception** *class-name description*
- **{@link** *package.class#member label***}**
- **@param** *parameter-name description*
- **@return** *description*
- **@see** *reference*
- **@throws** *class-name description*
- **@version** *version-text*
  
- <https://www.oracle.com/it/technical-resources/articles/java/javadoc-tool.html>



# GENERAZIONE AUTOMATICA DI DOCUMENTAZIONE CON JAUTODOC

- JAutoDoc è un'estensione free di Eclipse che può essere scaricata e installata da Eclipse Marketplace

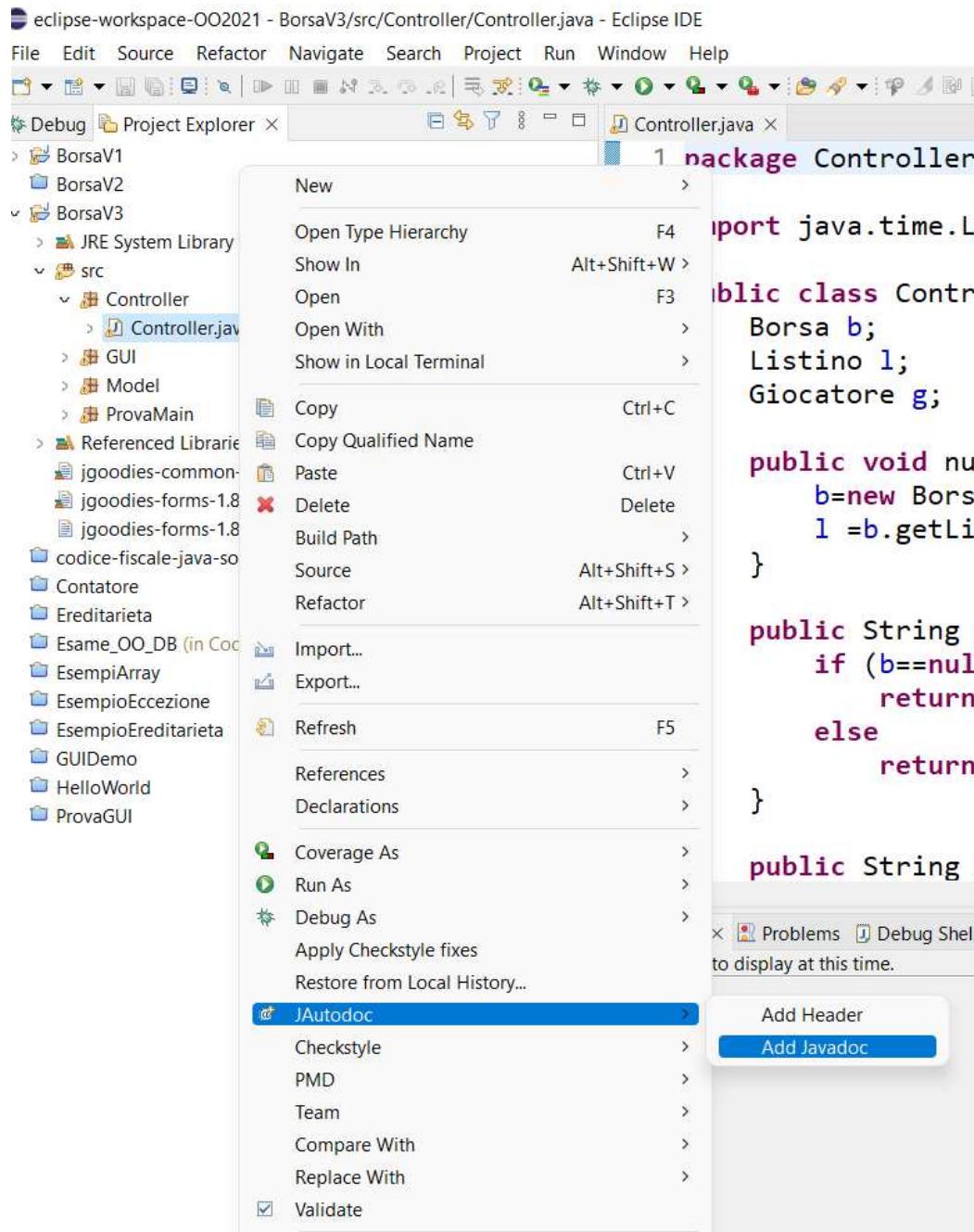


The screenshot shows the Eclipse Marketplace interface. At the top, there's a search bar with the text "Search Recent Popular Favorites Installed Giving IoT an Edge" and a "Find:" field containing "javadoc". Below the search bar, there are two main sections of results:

- JAutodoc 1.15.0**  
by recyclebin5385 Doe, BSD  
javadoc  
4 stars (416 installs)  
**Install**
- JavadocDecorator 1.2.1**  
by Martin Kesting, EPL  
javadoc templates file header velocity  
4 stars (173K installs)  
**Install**
- JavadocDecorator**  
This is very simple eclipse plug-in Decorate Java elements with Javadoc headings or method return values. Just installing it will help you get a quick overview of...  
**more info**  
by imkrmn, EPL 2.0

# JAUTODOC

- Jautodoc può essere utilizzato per generare o completare la documentazione del codice di classi o package



# DOCUMENTAZIONE GENERATA

- La documentazione generata da Javadoc va completata con descrizioni appropriate
- Dalle properties è possibile imporre a Javadoc di generare ulteriori tag Javadoc

```
20
21     /** The l. */
22     Listino l;
23
24     /** The g. */
25     Giocatore g;
26
27     /**
28      * Nuova borsa.
29      *
30      * @param citta the citta
31      */
32     public void nuovaBorsa(String citta) {
33         b=new Borsa(citta);
34         l =b.getListino();
35     }
36
37     /**
38      * Gets the citta borsa.
39      *
40      * @return the citta borsa
41      */
```



# GENERAZIONE JAVADOC

- Per generare la documentazione in HTML è sufficiente la sequenza di comandi
  - File → Export → Java → Javadoc
- Viene generato un completo sito web compost di pagine HTML statiche (nella cartella doc) che può essere navigato in locale o pubblicato sul web
  - Anche su github



**Package Controller**

## Class Controller

`java.lang.Object`<sup>✉</sup>  
Controller.Controller

---

```
public class Controller  
extends Object
```

The Class Controller.

### **Constructor Summary**

**Constructors**

Constructor	Description
<code>Controller()</code>	

### **Method Summary**

**All Methods**   **Instance Methods**   **Concrete Methods**

Modifier and Type	Method	Description
boolean	<code>acquista(String<sup>✉</sup> nomeSocieta, int quantita)</code>	Acquista.
boolean	<code>cercaSocieta(String<sup>✉</sup> nomeSocieta)</code>	Cerca societa.
<code>String<sup>✉</sup></code>	<code>getBilancio()</code>	Gets the bilancio.
<code>String<sup>✉</sup></code>	<code>getCittaBorsa()</code>	Gets the citta borsa.
<code>ArrayList<sup>✉</sup></code>	<code>getListeAcquisti()</code>	Gets the lista acquisti.



# **INTEGRAZIONE CON IL DATABASE**



# ARCHIETTURA DELLA SOLUZIONE PRIMA DELL'INTRODUZIONE DEL DB: PATTERN BCE

- Il Pattern Architetturale utilizzato è stato un pattern BCE: Boundary-Control-Entity, nel quale:
  - Boundary (confine) è il livello di interazione con l'utente o con altri programmi
    - Nel nostro caso il Boundary è il package GUI
  - Control è il livello di coordinamento, che fornisce servizi al Boundary e gestisce tutto il resto, implementando gli algoritmi relative alla logica di funzionamento del programma
    - Nel nostro caso il Control è il package Controller, che per semplicità contiene una sola classe
  - Entity è il livello del modello delle informazioni, che mappa tutti gli elementi del Dominio del Problema risultanti dall'analisi
    - Nel nostro caso Entity è il package Model, i cui oggetti sono istanziati e chiamati dalla classe Controller



# SOLUZIONI POSSIBILI PER L'INTRODUZIONE DEL DATABASE

- Il Database consente una gestione persistente dei dati
  - Anche un file è concettualmente un database
- Una soluzione architetturale possibile è un'estensione del pattern BCE che diventa così BCED (D per Database)
  - Database viene chiamato dalle classi entity e si occupa di interagire direttamente con il Database
- Pregi di questa soluzione:
  - Soluzione a livelli perfettamente separate
  - Si può progettare il Controller senza sapere quale database verrà utilizzato e nemmeno se verrà utilizzato o no un database per mantenere I dati persistenti
- Difetti:
  - Il Model, che era stato progettato per primo ora andrà modificato e mantiene dipendenze dallo specifico database



# SOLUZIONE BASATA SUL PATTERN DAO

- In questo corso adotteremo una soluzione basata sul pattern DAO
  - DAO sta per Data Access Object
- Per ogni oggetto del Model che ne abbia bisogno implementeremo una interfaccia nel package DAO che specificherà quali sono tutti i metodi di questa classe che si occupano di gestione della persistenza dei dati (ad es. Lettura/scrittura su di un db)
- Per ogni database supportato e per ogni interfaccia DAO scriveremo una classe concreta ImplementazioneDAO che realizzi le query che andremo a fare sul vero database
- Per comodità, l'apertura della connessione al database conviene che sia realizzata in una classe di utilità apposita



# CONNESIONE AL DATABASE

- Per modularità inseriamo le funzionalità relativa all'apertura del database ad una classe dedicate ConnessioneDatabase che offer:
  - Un costruttore che carica il driver del database richiesto e avvia la connessione
    - `Class.forName(driver);`
    - `connection = DriverManager.getConnection(url, nome, password);`
  - Un metodo getInstance che opera in questo modo:
    - Se la connessione è aperta restituisce l'oggetto ConnessioneDatabase;
    - Se la connessione è chiusa o è stata distrutta ne avvia una nuova istanziando un nuovo oggetto ConnessioneDatabase
      - Solo per questa volta, utilizzeremo l'attributo static che ci consente di eseguire il metodo anche in assenza di un oggetto ConnessioneDatabase
  - Un metodo getConnection che restituisce l'oggetto connection sul quale eseguire le query



# ESEMPIO DI UTILIZZO DI CONNESSIONEDATABASE

- In una classe che deve eseguire interrogazioni

```
private Connection connection;

public Costruttore() {
    try {
        connection = ConnessioneDatabase.getInstance().getConnection();
    } catch (SQLException e) { ... }

}

public void eseguiQueryDB(...) {
    try {
        PreparedStatement query = connection.prepareStatement(querySQL);
        query.executeQuery();
        ... // elabora risultati
        connection.close(); // fa risparmiare memoria ma perdere tempo alla prossima query
    } catch (SQLException e) { ... }

}
```



# ESEMPIO DI UTILIZZO DI CONNESSIONEDATABASE

```
public static ConnessioneDatabase getInstance() throws SQLException {  
    // se la connessione non esiste o è chiusa ne creo una nuova  
    if (instance == null) {  
        instance = new ConnessioneDatabase();  
    } else if (instance.getConnection().isClosed()) {  
        instance = new ConnessioneDatabase();  
    }  
    // altrimenti restituisco il riferimento a quella esistente  
    return instance;  
}
```

Con questa tecnica avrò sempre al più una connessione attiva



# PATTERN

- L'ascoltatore della GUI chiama un metodo del controller dedicato a quell'operazione
- Il metodo del controller istanzia oggetti del Model in memoria e/o chiama metodi per leggere/scrivere valori nel DB
  - I metodi per leggere/scrivere nel DB si trovano nelle classi DAO (Data Access Object)
- Il package DAO contiene una interfaccia per ogni classe del Model che necessita di persistenza
- Esiste un package ImplementazioneDAO per ogni database das supportare (ad es. ImplementazionePostgresDAO). All'interno del package ci sono le classi con tutte le implementazioni delle interfacce DAO rispetto a quell database
- La connessione al database viene di solito gestita da una classe di utilità ConnessioneDatabase

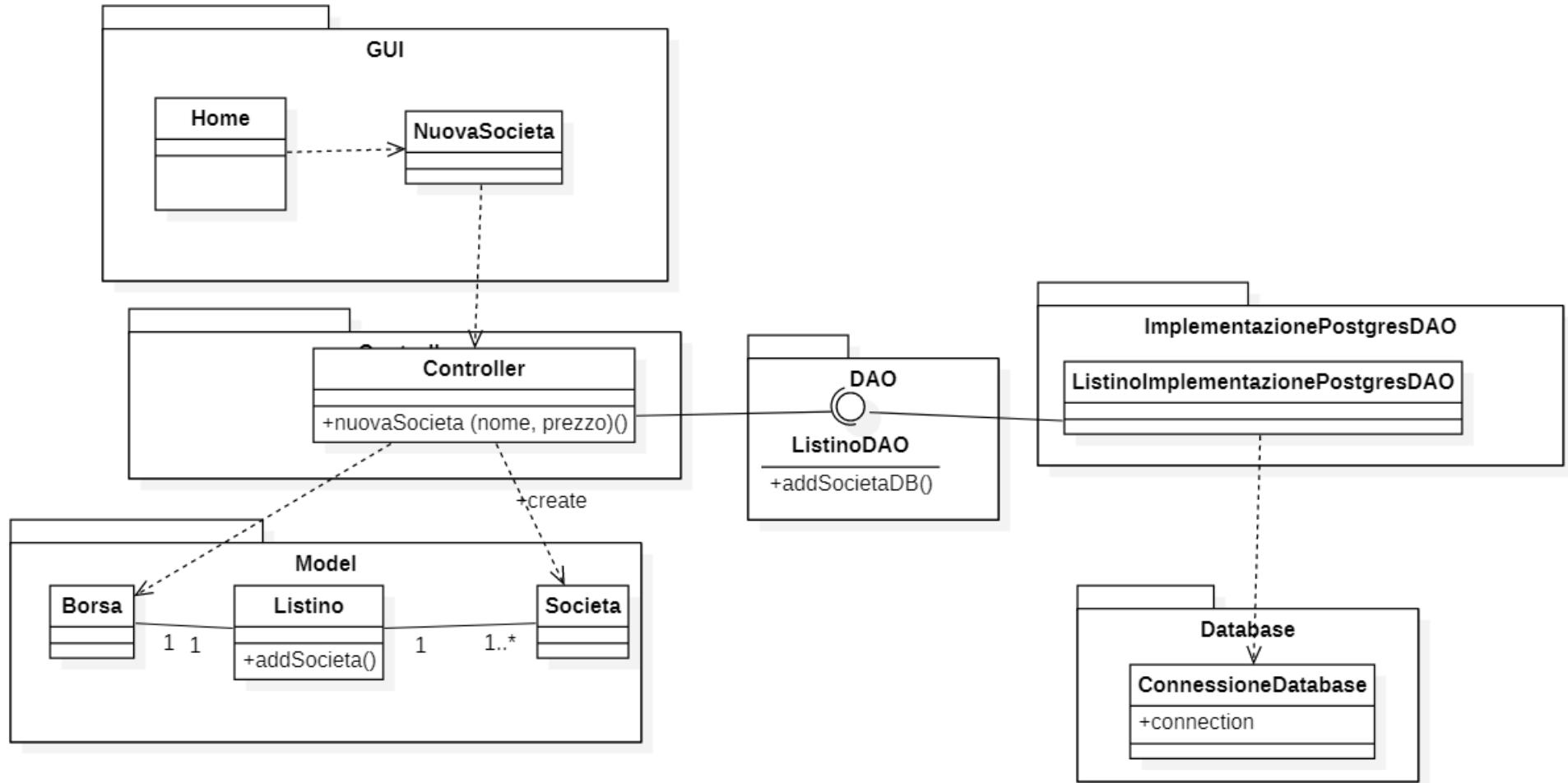


# OSSERVAZIONI

- Per ragioni di efficienza si può scegliere se aprire e chiudere connessioni o mantenerle aperte a lungo
- La separazione tra interfaccia DAO e implementazione consente di minimizzare il lavoro in caso si voglia cambiare DB
  - E' sufficiente aggiungere nuove classi ImplementazioneDAO e specificare nel Controller l'utilizzo di questo nuovo DB
    - Passando al controller il database scelto (dependency injection) si potrebbe anche rendere il controller indipendente da questa scelta
- La separazione tra rappresentazione in memoria e rappresentazione nel DB ci consente di gestire in maniera distinta i due aspetti
  - Operazioni come calcolaCapitale le svolgiamo direttamente in memoria per essere più rapidi
  - Potremmo decidere anche di salvare tutte le modifiche sul database solo alla chiusura della sessione



# ESEMPIO: AGGIUNTA DI UNA NUOVA SOCIETA AL LISTINO – FRAMMENTO DI CLASS DIAGRAM



# ESEMPIO: AGGIUNTA DI UNA NUOVA SOCIETA AL LISTINO – FRAMMENTI DI CODICE

- In GUI.NuovaBorsa
  - controller.nuovaSocieta(textNomeSocieta.getText(),textPrezzoAzione.getText());
- In Controller.nuovaSocieta
  - Societa s= new Societa(nomeSocieta,p); //crea in memoria
  - b.getListino().addSocieta(s);
  - ListinoDAO l=new ListinoImplementazionePostgresDAO();
  - l.addSocietaDB(s,b); //scrive sul DB
- Nel costruttore di ListinoImplementazionePostgresDAO
  - connection = ConnessioneDatabase.getInstance().getConnection();
- In ListinoImplementazionePostgresDAO.addSocieta
  - PreparedStatement leggiListinoPS = connection.prepareStatement("INSERT INTO \"Borsa\".\"Societa\" " + "("+"Nome\\"", \"PrezzoAzione\\"", \"CittaBorsa\\"\")" + "VALUES ('"+s.getNome()+"','"+s.getPrezzoAzione()+"','"+b.getCitta()+"');");
  - leggiListinoPS.executeUpdate();
  - connection.close(); //opzionale



# ESEMPIO: LETTURA DELLE SOCIETA DEL LISTINO DI UNA BORSA – FRAMMENTI DI CODICE

- In GUI.NuovaBorsa

- controller.nuovaBorsa(textCitta.getText());

- In Controller.nuovaBorsa

- b=new Borsa(citta); //crea Borsa in memoria
  - ListinoDAO l=new ListinoImplementazionePostgresDAO();
  - //legge listino da DB e lo scrive in memoria
  - b.setListino(l.leggiListinoDB(b));

- In ListinoImplementazionePostgresDAO.leggiListinoDB

- Listino l=null;
  - leggiListinoPS = connection.prepareStatement("SELECT \* FROM \"Borsa\".\"Societa\" WHERE \"CittaBorsa\"='"+b.getCitta()+"'");
  - ResultSet rs = leggiListinoPS.executeQuery();
  - while (rs.next()) {
    - l.addSocieta(new Societa(rs.getString("Nome"),rs.getFloat("PrezzoAzione")));
  - }
  - rs.close();
  - connection.close(); //opzionale
  - return l;



# ESEMPIO: ACQUISTO DI AZIONI – FRAMMENTI DI CODICE

- In GUI.AcquistoDaListino
  - controller.acquista(nomeSocieta, quantita);
- In Controller.acquista
  - g.acquista(quantita, LocalDate.now(), s.getPrezzoAzione(), s); //in memoria
  - g.calcolaCapitale(); //con dati in memoria
  - GiocatoreDAO gDAO=new GiocatoreImplementazionePostgres();
  - gDAO.acquistaDB(g, quantita, LocalDate.now(), s.getPrezzoAzione(), s); //su DB
- Nel costruttore di GiocatoreImplementazionePostgresDAO
  - connection = ConnessioneDatabase.getInstance().getConnection();
- In GiocatoreImplementazionePostgresDAO.acquistaDB
  - PreparedStatement nuovoAcquistoPS = connection.prepareStatement("INSERT INTO \"Borsa\".\"Acquisto\" " + "(\"Societa\", \"Giocatore\", \"Quantita\", \"Prezzo\")" + "VALUES ('"+societa.getNome()+"','"+g.getNome()+"',"+quantita+","+prezzo+");");
  - nuovoAcquistoPS.executeUpdate();



# ANNOTAZIONI TECNICHE PER L'ESEMPIO

- Per poter eseguire l'esempio è necessario ovviamente creare preventivamente un database Postgres
  - Per semplicità login, password e indirizzo del database sono stati aggiunti nel codice: ovviamente non è una soluzione generale e sicura!
  - Volendo si poteva anche creare il database via SQL dal programma stesso
- Al Progetto è stata aggiunta la libreria postgresql-42.2.24.jar
  - Basta copiarla nella cartella di Progetto dove sono già le jgoodies e poi aggiungerle al Build Path con click destroy → Build Path ... → Add to Build Path



# RIASSUMENDO

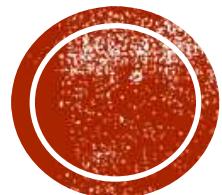
- GUI chiama Controller
- Controller chiama Model (per interagire con modello in memoria)
- Controller chiama ImplementazionePostgresDAO che implementa DAO (per interagire con la persistenza dei dati sul DB)
- ImplementazionePostgresDAO interagisce col database tramite Connessione Database
- ConnessioneDatabase effettua la connessione al database



# ESERCIZIO : AGGIORNA IL PREZZO DELLE AZIONI DI UNA SOCIETA

- Da Basi di Dati la query potrebbe essere:
  - `UPDATE "Borsa"."Societa" SET "PrezzoAzione" = 21 WHERE "Nome" = 'google';`





# MODELЛАZIONE CON UML



# SOFTWARE PER LA MODELLAZIONE CON UML

- La modellazione con UML nasce con la possibilità di essere realizzata banalmente con carta e penna
- Ciò nonostante, può essere comodo utilizzare dei software di supporto al disegno di diagrammi UML
- Nel contesto di questo corso, è consigliato l'utilizzo di uno strumento basilare e di libero utilizzo, StarUML che è più che sufficiente per i problemi di modellazione e progetto che dovremo affrontare
  - <https://staruml.io/>



- In alternativa, strumenti più maturi come ad esempio Visual Paradigm sono pure consigliati
  - <https://www.visual-paradigm.com/>



# INTRODUZIONE A UML

UML Distilled, Capitolo 1



# MODELLAZIONE DEL DOMINIO DEL PROBLEMA

- Diversi tipi di modelli possono descrivere il dominio del problema. Tali modelli sono detti modelli di *dominio* oppure modelli *concettuali* o, talvolta, modelli di *business*
  - Attenzione: non bisogna confondere i *modelli di dominio*, che descrivono *il problema*, dai *modelli di progetto*, che descrivono *la soluzione*
- Si distingue tra:
  - Modelli statici, che descrivono gli elementi del dominio del problema e le relazioni tra loro
  - Modelli dinamici, che descrivono i comportamenti riconoscibili nel dominio del problema



# MODELLAZIONE DI SISTEMA

- Per descrivere tali modelli si utilizzano linguaggi di modellazione
- In passato, diversi linguaggi di modellazione erano utilizzati a supporto delle metodologie che si applicavano nelle varie fasi del processo di sviluppo del software
  - Ad esempio:
    - Diagrammi di flusso
    - Data Flow Diagrams
    - ...
- Negli ultimi anni il linguaggio UML si sta affermando come linguaggio unificato che possa essere utilizzato in tutte le attività di modellazione, nonchè in molte altre attività del ciclo di vita del software



# CHE COSA È UML?

- UML (Unified Modelling Language) nasce come linguaggio grafico standard per modellare software object oriented
  - Tra la fine degli anni '80 e gli inizi degli anni '90 fecero la comparsa i primi processi di sviluppo object-oriented
  - La proliferazione di metodi e notazioni diverse creavano confusione
  - Due importanti metodologi, Rumbaugh e Booch, decisero di fondere i loro approcci nel 1994.
    - Cominciarono a lavorare insieme alla Rational Software Corporation
  - Nel 1995, un altro metodologo, Jacobson, si unì al gruppo
    - Il suo lavoro si concentrava sugli use cases
  - Nell'1997 l'Object Management Group (OMG) cominciò il processo di standardizzazione di UML
  - Nel Luglio 2005 è stata rilasciata la prima release di UML 2
- Su <http://www.omg.org/spec/UML/> è possibile leggere la storia di tutte le specifiche UML
  - Una specifica UML in pratica definisce un **metamodello** che indica secondo quali regole sia possibile costruire modelli UML



# USO DI UML

- UML può essere usato:
  - Come *abbozzo*, cioè per tracciare un modello di massima di un sistema da realizzare
  - Come *progetto*, cioè per realizzare un modello completo della soluzione architetturale del sistema
  - Come *linguaggio di programmazione* in grado di modellare in maniera completa e precisa il sistema software
    - L'approccio MDA (Model Driven Architectures) esplora la possibilità di usare UML come linguaggio di programmazione
      - Questa ultima possibilità è al momento soprattutto un obiettivo cui mirare in futuro. Si vorrebbe, in pratica, stabilire una sintassi e una semantica precisi per UML, che portino alla generazione automatica di codice sorgente rappresentativo del modello tracciato



# REGOLE DI UML

- UML è dotato sia di regole *prescrittive* che di regole *descrittive*
  - Le regole prescrittive sono regole stabilite da organismi standardizzanti che caratterizzano precisamente lessico, sintassi e semantica
  - Le regole descrittive, invece, hanno come scopo solo la comunicazione del significato dei diagrammi, con estensioni, libere ma intuitive, delle regole base
- UML ha come scopo principale la descrizione efficace di situazioni reali, cosicchè le regole descrittive sono al momento predominanti
- Regola delle informazioni sopprese:
  - L'assenza di qualche informazione in un diagramma UML non significa, in generale, che tale informazione non esista o sia nulla, ma semplicemente che si tratti di un aspetto del problema non ancora trattato nella fase in cui è stato tracciato il diagramma (o che non si voglia palesare in tale diagramma per poi inserirlo in diagrammi ulteriori)



# TIPI DI MODELLI IN UML 2

- UML 2 possiede 13 differenti tipi di diagrammi, appartenenti a tre categorie:
  - **Diagrammi Comportamentali**, quali use-case diagrams, activity diagrams, state machine diagrams
  - **Diagrammi di Interazione**, per modellare interazioni fra entità del sistema, quali sequence diagrams e communication diagrams.
  - **Diagrammi Strutturali**, che modellano l'organizzazione del sistema, quali class diagrams, package diagrams, e deployment diagrams.
- In questo primo corso che tratta UML, ci concentreremo sui due diagrammi più utilizzati:
  - **Class diagrams**
  - **Sequence Diagrams**



# **CLASS DIAGRAM**

UML Distilled, Capitolo 3, Capitolo 5



# CLASS DIAGRAM

- Il più diffuso diagramma compreso in UML è il diagramma delle classi
- Si tratta di un diagramma statico che può essere utilizzato:
  - Per la modellazione concettuale del dominio di un problema
  - Per la modellazione delle specifiche richieste ad un sistema
  - Per modellare l'implementazione (object-oriented) di un sistema software
- I concetti fondamentali di un class diagram sono estensioni dei concetti fondamentali dei paradigmi object-oriented
- Nel seguito verrà presentato il class diagram nella sua accezione più completa, relativa alla modellazione dell'implementazione di sistemi object-oriented



# ASPETTI PRINCIPALI

■ *I principali elementi dei class diagram sono:*

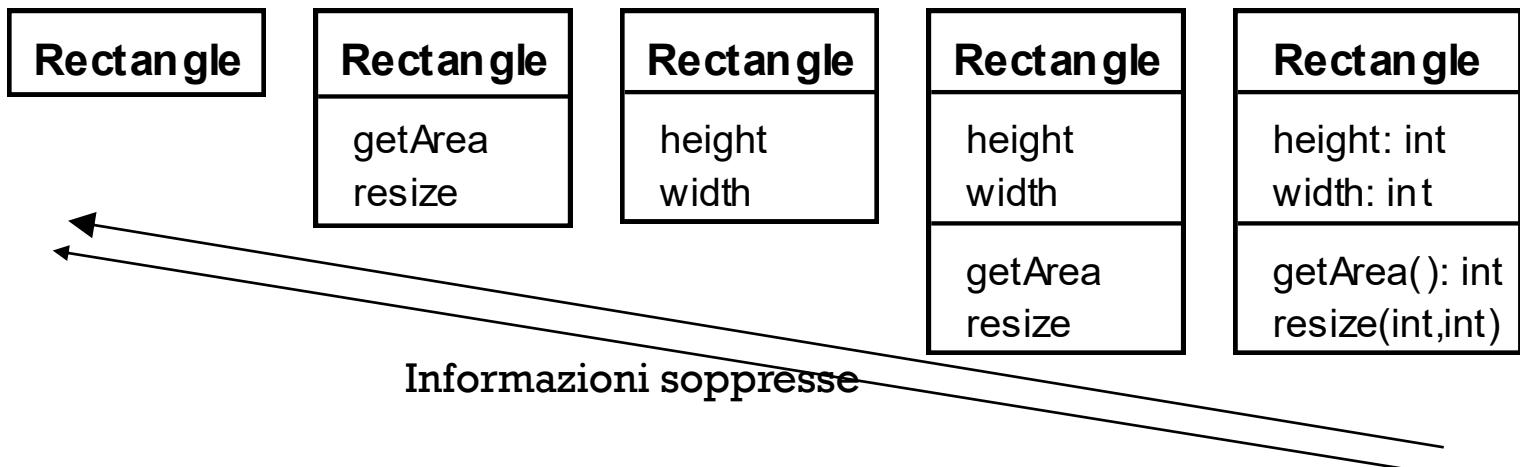
- *Classi*
  - Rappresentanti i tipi di dati presenti in un sistema
- *Associazioni*
  - Rappresentano i collegamenti fra istanze di classi
- *Attributi*
  - Sono i dati semplici presenti nelle classi e nelle loro istanze
- *Operazioni*
  - Rappresentano le funzioni svolte dalle classi e dalle loro istanze
- *Generalizzazioni*
  - Raggruppano le classi in gerarchie di ereditarietà



# CLASSI

- Una classe è semplicemente rappresentata da un rettangolo con il nome della classe all'interno
  - Il concetto di classe è lo stesso dell'OO
  - La signature completa di un'operazione è:

operationName(parameterName: parameterType ...): returnType



# ATTRIBUTI

**visibilità nome molteplicità: tipo = default {proprietà}**

Sono consentiti tre livelli di visibilità:

- + **Pubblico:** L'utilizzo viene esteso a tutte le classi
- # **Protetto:** L'utilizzo è consentito soltanto alle classi che derivano dalla classe originale
- **Privato:** Soltanto la classe originale può utilizzare gli attributi e le operazioni definite come tali.

Il **nome** dell'attributo é l'unico parametro necessario

Il **tipo** dell'attributo può essere un tipo primitivo (int, double, char, etc...) oppure il nome di una classe definita nello stesso diagramma (in tal caso forse l'attributo andrebbe indicato con un'associazione ...)

**Default** rappresenta il valore di default dell'attributo



# ATTRIBUTI

**visibilità nome molteplicità: tipo = default {proprietà}**

La **molteplicità** indica il quantitativo degli attributi (ad esempio la dimensioni per un array). Tramite la molteplicità è possibile indicare come attributi degli array o matrici. Il valore di default è 1.

Alcuni valori possibili sono:

- **1** (uno e uno solo). E' il valore di default
- **0..1** (al più uno)
- **\*** (un numero impreciso, eventualmente anche nessuno; equivalente a 0..\*)
- **1..\*** (almeno uno)

Gli elementi di una molteplicità sono considerati come un insieme.

Se essi sono dotati anche di ordine si aggiunge l'indicazione {ordered}.

Se sono possibili valori duplicati si aggiunge l'indicazione {nonunique}

**{proprietà}** rappresenta caratteristiche aggiuntive dell'attribute (ad esempio la sola lettura)

Esempio: name: String [1] = "Untitled" {readOnly}



# METODI

**visibilità nome (lista parametri) : tipo-ritornato {proprietà}**

La **visibilità** e il **nome** seguono regole analoghe a quelle degli attributi.

**Lista parametri** contiene nome e tipo dei parametri della funzione, secondo la forma:

**direzione nome parametro: tipo = valore-di-default**

**direzione**: input (*in*), output (*out*) o entrambi (*inout*). Il valore di default è *in*

**nome, tipo e valore di default** sono analoghi a quelli degli attributi

**Tipo-ritornato** è il tipo del valore di ritorno: dovrebbe essere un tipo appartenente ad una classe standard

**Esempio:**

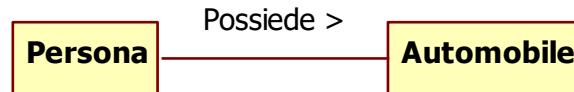
+ balanceOn (date: Date) : Money



# ASSOCIAZIONI

- Un'associazione rappresenta una relazione (fisica o concettuale) tra classi

Esempio: **Persona possiede Automobile**



- Il verso dell'associazione indicato in figura indica in che direzione deve essere *letta* l'associazione
  - In questo caso indica che è la Persona a posseder l'Automobile e non l'Automobile a possedere la Persona!
- Possibile implementazione:

```
class Persona{  
    Automobile automobilePosseduta;  
}  
class Automobile{  
}
```



# ASSOCIAZIONI

- In alternativa, si può indicare il *ruolo* di uno dei due estremi dell'associazione



- Possibile implementazione:

```
class Persona{  
}  
  
class Automobile{  
    Persona proprietario;  
}
```



# VERSO DI NAVIGAZIONE

- **Verso di navigazione di un'associazione**
    - Il verso di navigazione è un'informazione utile soprattutto in fase di progetto di dettaglio
    - Indica in quale direzione è possibile reperire le informazioni
- ```
graph LR; Persona[Persona] -- "Possiede >" --> Automobile[Automobile]
```

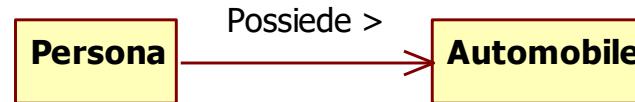
The diagram shows two yellow rectangular boxes representing classes: 'Persona' on the left and 'Automobile' on the right. A red arrow points from 'Persona' to 'Automobile', with the label 'Possiede >' written above the arrow.
- Nell'esempio, nota una persona è possibile sapere quali sono le automobili che possiede (se ne possiede)
  - Viceversa, non è possibile conoscere il possessore di una data automobile
  - Non ci sono, però, indicazioni sul quantitativo di automobili possedute, né sul numero di proprietari di un automobile (da questo diagramma non possiamo sapere se si tratti di informazioni non note o di informazioni sopprese)
  - Di solito, il verso di navigazione rappresenta una scelta di progetto, per cui non è presente nei diagrammi concettuali



# VERSO DI NAVIGAZIONE

- Implementazione possibile:

```
class Persona{  
    Automobile automobilePosseduta;  
}  
  
class Automobile{  
}
```



- Implementazione errata:

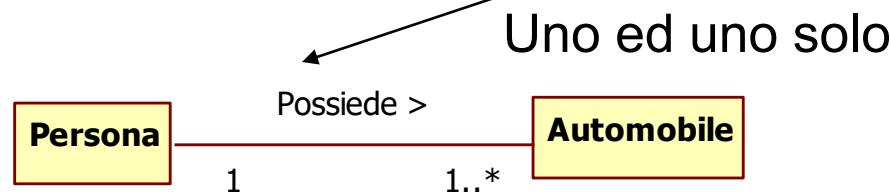
```
class Persona{  
}  
  
class Automobile{  
    Persona possessore;  
}
```

- Data la persona non è possibile risalire alla sua automobile!



# MOLTEPLICITÀ DELLE ASSOCIAZIONI

- La molteplicità delle associazioni indica il numero di istanze di oggetti di ogni classe che possono appartenere ad una istanza della associazione



- Nell'esempio,
  - una Persona possiede almeno una Automobile
    - evidentemente le persone che non possiedono Automobile non fanno parte del problema in oggetto
  - Un'Automobile può essere posseduta da una e una sola Persona
    - Evidentemente, non è nel problema in oggetto il mantenimento di informazioni riguardo i proprietari di automobili di seconda, terza mano, etc.
  - Quest'esempio si configura come associazione uno a molti (una Persona, molte Automobili)

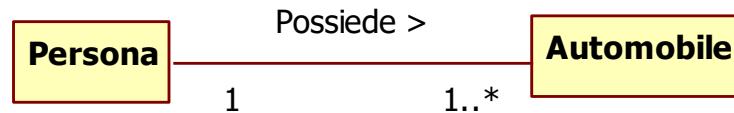


# MOLTEPLICITÀ DELLE ASSOCIAZIONI

- **Possibile implementazione:**

```
class Persona{  
    Persona(Automobile a) {  
        automobiliPossedute.add(a);  
        a.proprietario = this;  
    }  
    ArrayList<Automobile> automobiliPossedute;  
}  
  
class Automobile{  
    Automobile (Persona p) {  
        proprietario=p;  
    }  
    public Persona proprietario;  
}
```

Grazie al costruttore, la persona ha sicuramente almeno un'automobile della quale è l'unico proprietario  
dovremo fare attenzione quando realizzeremo il metodo remove, che dovrà eliminare la persona che non ha più automobili



# ASSOCIAZIONI MOLTI A MOLTI

## ▪ Associazioni multi-a-molti

- Uno studente può conseguire un numero potenzialmente non limitato di esami
- Un esame può essere conseguito da un numero potenzialmente non limitato di studenti
- Possono esserci studenti che non hanno conseguito esami
- Possono esserci esami non conseguiti (ancora) da nessuno studente



Possibile implementazione:

```
class Studente{  
    ArrayList<Esame> esamiConseguiti; }  
  
class Esame{  
    ArrayList<Studente> studentiEsaminati; }
```



# ASSOCIAZIONI MOLTI A MOLTI

## ▪ Associazioni multi-a-molti

- Se avessimo voluto modellare il caso in cui uno studente era considerato solo dal momento del conseguimento del primo esame, allora sarebbe stato



Possibile implementazione:

```
class Studente{  
    Studente (Esame e) {  
        esamiConseguiti.add(e);  
        e.studentiEsaminati.add (this);  
    }  
    ArrayList<Esame> esamiConseguiti; }  
class Esame{  
    public ArrayList<Studente> studentiEsaminati; }
```

Soluzione valida se  
studentiEsaminati è public,  
altrimenti è necessario un metodo  
pubblico di Esame che permetta di  
aggiungere uno studente  
esaminato



# ASSOCIAZIONI UNO A UNO

## ▪ Uno-a-uno

- Ogni studente ha uno e un sol badge
  - Non è possibile modellare, in caso di smarrimento e rilascio di un nuovo badge, l'elenco di tutti i badge avuti da uno studente nel tempo
- Un badge identifica uno e un solo studente

## ▪ Possibile implementaz:

```
class Studente{  
    Studente() { badge = new Badge(this) ; }  
    Badge badge;  
}  
  
class Badge{  
    Badge (Studente s){ studente=s; }  
    Studente studente;  
}
```



La soluzione simmetrica è ugualmente valida



# ASSOCIAZIONI UNO A UNO

## ▪ Uno-a-uno

- Se avessimo voluto considerare anche studenti che, magari temporaneamente, non abbiano un badge, allora diventa:



## ▪ Possibile implementazione:

```
class Studente{  
    Badge badge;  
}  
  
class Badge{  
    Badge (Studente s) { studente=s; }  
    Studente studente;  
}
```

La soluzione simmetrica qui non è ugualmente valida

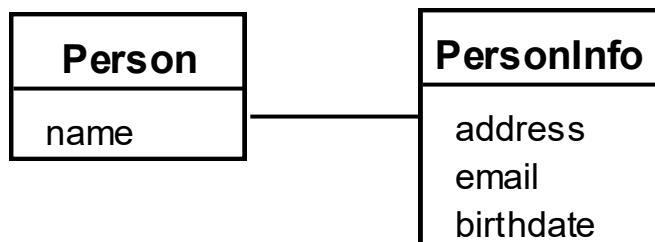
Da notare che lo student può esistere anche senza Badge, quindi il costruttore di Badge non deve necessariamente istanziare un nuovo studente



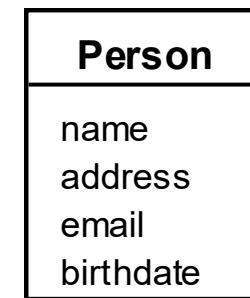
# CONTROESEMPI

- A volte le associazioni uno a uno sono inutili

Evitare



migliore soluzione!



# UN ESEMPIO PIÙ COMPLESSO

- Una prenotazione si riferisce sempre ad un solo passeggero
  - Non esistono prenotazioni con zero passeggeri
    - Ciò implica che prima di creare una prenotazione deve esistere un passeggero
  - Una prenotazione non può mai riferirsi a più di un passeggero.
- Un Passeggero può avere più prenotazioni
  - Un passeggero potrebbe avere zero prenotazioni
  - Un passeggero potrebbe avere più di una prenotazione
- Una prenotazione si riferisce ad un volo
- Un volo può avere più passeggeri prenotati



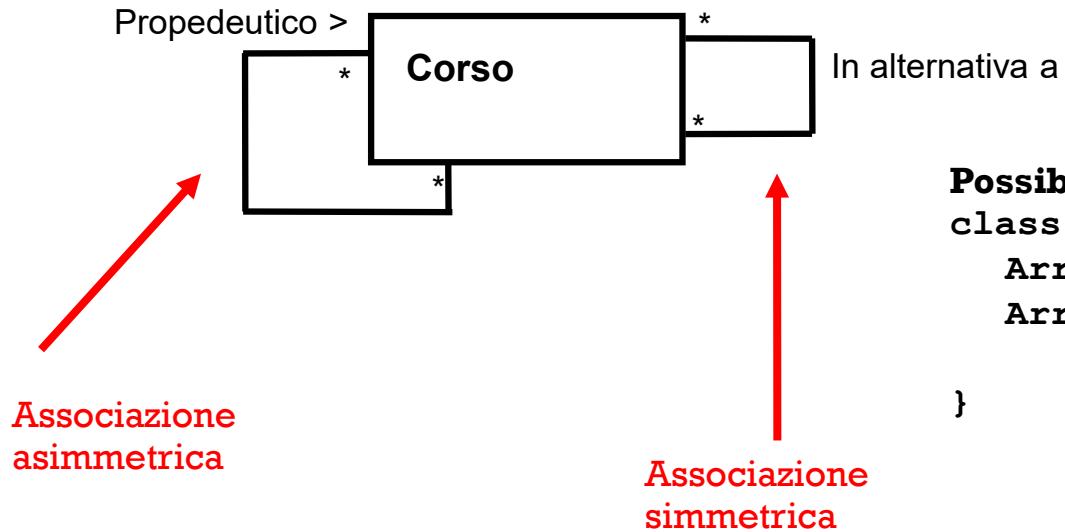
# UN ESEMPIO PIÙ COMPLESSO

- Una prenotazione si riferisce sempre ad un solo passeggero
  - Non esistono prenotazioni con zero passeggeri
    - Ciò implica che prima di creare una prenotazione deve esistere un passeggero
  - Una prenotazione non può mai riferirsi a più di un passeggero.
- Un Passeggero può avere più prenotazioni
  - Un passeggero potrebbe avere zero prenotazioni
  - Un passeggero potrebbe avere più di una prenotazione
- Una prenotazione si riferisce ad un volo
- Un volo può avere più passeggeri prenotati



# ASSOCIAZIONI RIFLESSIVE

- Associazioni che collegano una classe con se' stessa



In alternativa a

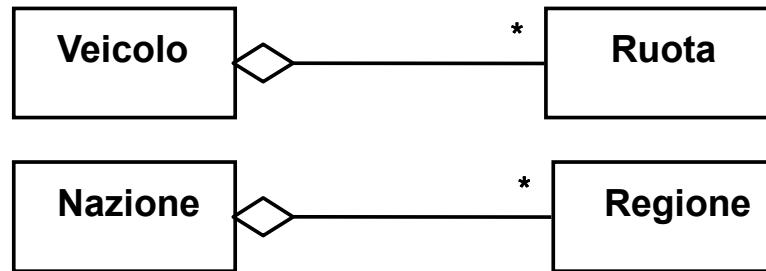
**Possibile implementazione:**

```
class Corso{  
    ArrayList<Corso> corsiPropedeutici;  
    ArrayList<Corso> corsiAlternativi;  
}
```



# AGGREGAZIONE

- Le Aggregazioni sono speciali associazioni che rappresentano una relazione ‘tutto-parti’.
  - Il lato del ‘tutto’ è spesso chiamato *l’aggregato*
  - La molteplicità dal lato del tutto, quando è sottintesa, vale 0..1



# QUANDO USARE UNA AGGREGAZIONE

- Una associazione diventa una aggregazione se:
  - È possibile affermare che:
    - Le parti sono ‘parte di’ un insieme
    - L’aggregato è ‘composto da’ parti
  - Quando qualcosa possiede o controlla l’aggregato, allora esso possiede o controlla anche le sue parti
  - Per quanto le aggregazioni siano importante dal punto di vista della espressività del modello, spesso sono implementate in maniera identica rispetto ad associazioni di pari cardinalità



# POSSIBILE IMPLEMENTAZIONE

```
class Veicolo{  
    ArrayList<Ruota> ruote;  
}  
  
class Ruota{  
    Veicolo v=null;  
}
```

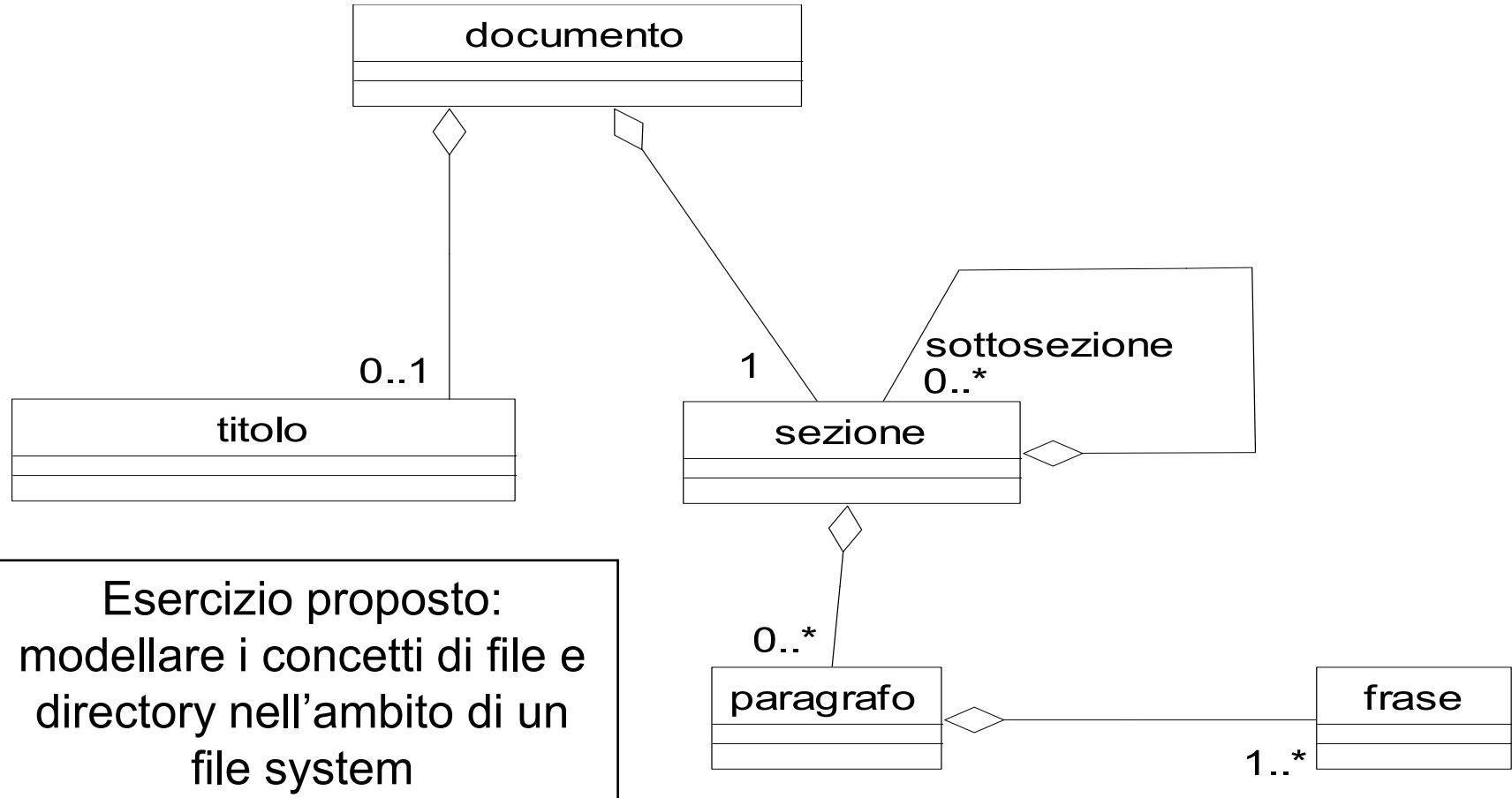
L'attributo `v` in `Ruota` è consigliato perché una ruota può essere in un veicolo e così possiamo navigare direttamente questa relazione.

Potrebbe trattarsi anche di una ruota di scorta che non è in un veicolo.

L'attributo `ruote` in `Veicolo` è invece obbligatorio perché esprime il fatto che un veicolo aggrega ruote

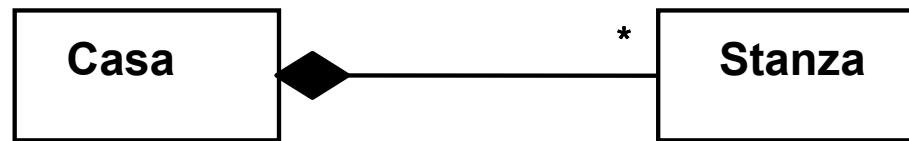


# UNA GERARCHIA DI AGGREGAZIONE



# COMPOSIZIONE

- Una *composizione* è una forma forte di aggregazione
  - Se l'aggregato viene distrutto, anche le sue parti saranno distrutte (le parti non esistono senza il tutto)
    - Evidentemente, in questo dominio, non ha senso parlare di stanze fintantochè esse non siano state legate alla casa in cui si trovano
    - La cardinalità dell'aggregazione, se sottintesa, vale 1



# POSSIBILE IMPLEMENTAZIONE

```
class Casa{  
    ArrayList<Stanza> stanze;  
}  
  
class Stanza{  
    Stanza(Casa casa) {  
        c=casa;  
    }  
    Casa c;  
}
```

Rispetto all'esempio precedente ora l'attributo c in Stanza è obbligatorio e deve essere inizializzato dal costruttore, cosicchè non possa esistere una Stanza che non sia in associazione con una casa

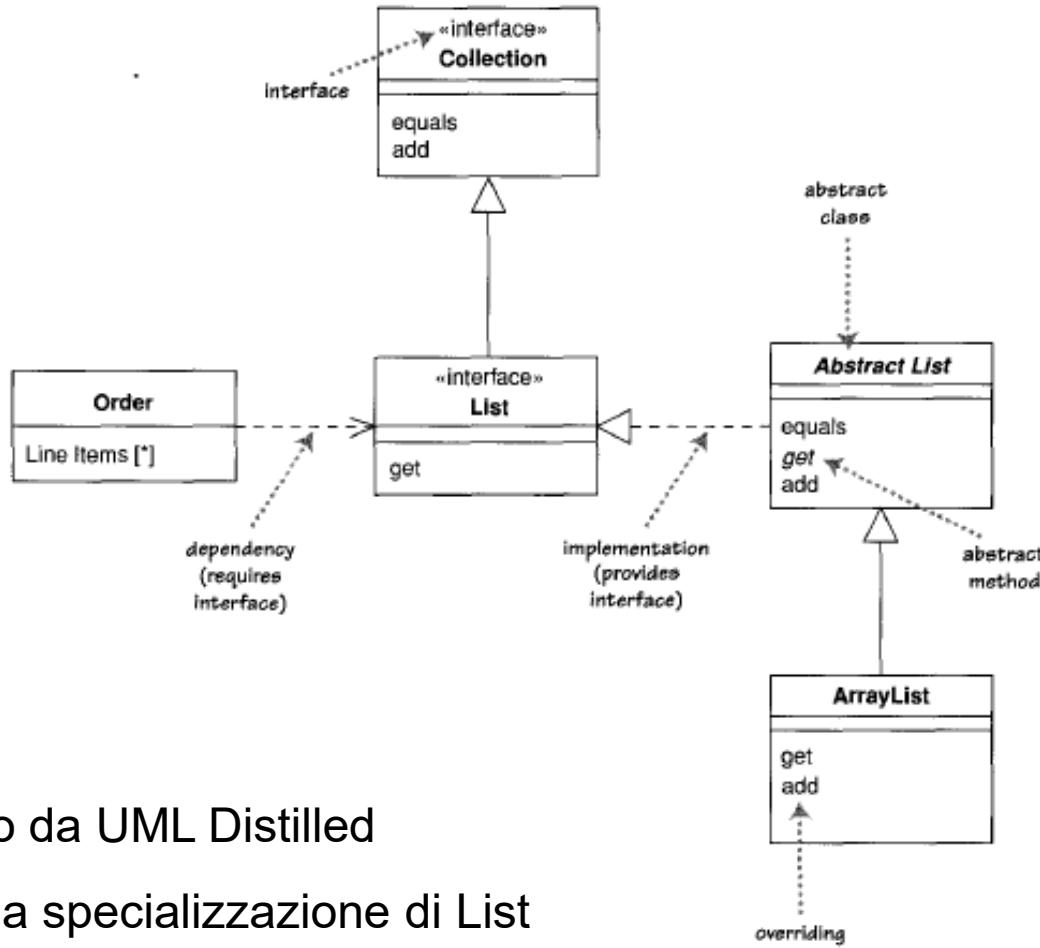


# GENERALIZZAZIONE

- I concetti di generalizzazione e specializzazione UML sono del tutto analoghi a quelli object oriented
- A livello concettuale, una gerarchia di generalizzazione esprime una relazione **is-a** tra un concetto generale e le sue specializzazioni
- A livello di progetto di dettaglio, invece, può essere interpretato:
  - Come una relazione di ereditarietà tra due classi concrete
    - Le classi derivate (figlie) ereditano attributi e metodi **public** e **protected** dalla classe padre



# ESEMPIO



Esempio tratto da UML Distilled

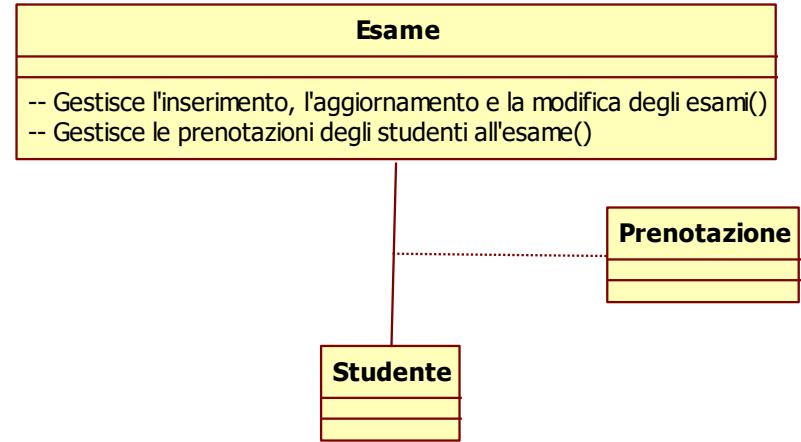
ArrayList è una specializzazione di List

List è una specializzazione di Collection



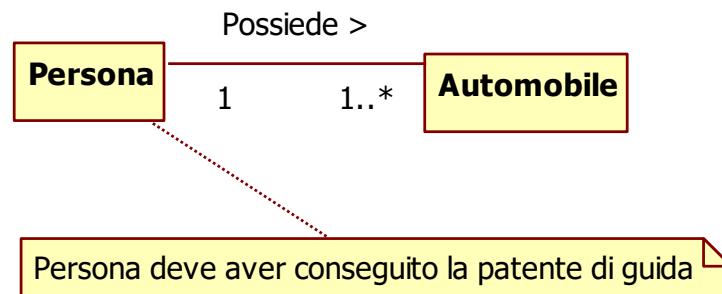
# RESPONSABILITÀ

- Nei diagrammi che descrivono il dominio del problema non si inseriscono di solito metodi perché essi rappresenterebbero un elemento del dominio della soluzione
- In alternativa si possono utilizzare le Responsabilità
  - Una responsabilità è un insieme di servizi e compiti che la classe dovrebbe garantire
  - Sono utili per controllare la completezza del modello di dominio



# NOTE E TESTO DESCRITTIVO

- Si possono aggiungere ulteriori informazioni direttamente in linguaggio naturale ad un qualsiasi diagramma UML utilizzando le annotazioni
- Note:
  - Una nota è un pezzo di testo incluso *in* un diagramma UML
  - È come un commento in un linguaggio di programmazione



# UN PROCESSO DI ASTRAZIONE

1. Identifica un primo insieme di **classi candidate**
2. Aggiungi **associazioni** ed **attributi** a queste classi
3. Trova le **generalizzazioni**
4. Trova le principali **responsabilità** di ogni classe
5. **Itera** il processo finchè il modello ottenuto è soddisfacente



# 1. IDENTIFICARE LE CLASSI

- Quando si sviluppa un domain model si tende a *scoprire* classi che fanno parte del dominio
- Nel lavorare all'interfaccia utente o all'architettura, si tende ad *inventare* classi
  - Necessarie a risolvere un problema di design
  - Si possono inventare classi anche per il domain model!
- Ad una classe dovrebbe corrispondere un'entità del dominio del problema
  - In pratica, bisogna applicare i concetti generali di modellazione Object Oriented



# **COME È FATTA UNA BUONA CLASSE DI ANALISI?**

- Il suo nome ne rispecchia l'intento.
- è una astrazione ben definita che modella un elemento del dominio del problema
- ha un insieme ridotto e ben definito di responsabilità
- ha una massima coesione interna
  - Una classe è tanto più coesa quanto più riesce ad assolvere da sola alle proprie responsabilità



# ANALISI DEI NOMI

- Semplice tecnica per scoprire le classi del dominio
  - Analizzare la documentazione di partenza, come la descrizione dei requisiti
  - Estrarre *nomi* e *predicati nominali* (aggettivi di nomi)
  - Eliminare nomi che:
    - Sono ridondanti (rappresentano la stessa classe)
    - Rappresentano istanze e non classi
    - Sono vaghi, troppo generici
    - Corrispondono a classi che non sono necessarie al livello considerato
  - Fare attenzioni a classi nel domain model che rappresentano *tipi di utente* o altri attori (servono davvero?)



# CRC CARDS

- CRC sta per Class Responsibility Collaboration
- Per ogni classe identificata, porre il nome della classe su una scheda (Card)
- Man mano che vengono individuati attributi e responsabilità, elencarli sulle Card
- Sistemare le card su una lavagna per creare il Class diagram
- Disegnare le linee corrispondenti ad associazioni e generalizzazioni.
  - L'utilizzo delle card serve per imporre, quanto meno psicologicamente, all'analista di non realizzare classi con un numero troppo elevato di attributi e metodi → Se la card è piena allora probabilmente bisogna dividere la classe in due o più classi più semplici



|                                                                                                                                                    |                                               |
|----------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------|
| <p><u>Nome classe</u></p> <ul style="list-style-type: none"><li>- Responsabilità 1</li><li>- Responsabilità 2</li><li>- Responsabilità 3</li></ul> | <p>Collaboratore 1</p> <p>Collaboratore 2</p> |
|----------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------|

**Figura 17.2** Template per una scheda CRC.



### Group Figure

Holds more Figures.  
(not in Drawing)  
Forwards transformation  
Cache image, valid  
on update of master.

### Figures

### Drawing

Holds Figures.  
Accumulates updates,  
refreshes on demand.

Figure  
Drawing View  
Drawing Controller

### Selection Tool

Selects Figures (adds  
Handles to Drawing View)  
Invokes Handles

Drawing Controller  
Drawing View  
Figures  
Handles

### Scroll Tool

Adjusts The View's  
Window

Drawing View

**Figura 17.3** Quattro esempi di schede CRC. Questo esempio ha semplicemente lo scopo di mostrare il livello tipico di dettaglio, e non il testo specifico.



## 2. IDENTIFICARE ASSOCIAZIONI ED ATTRIBUTI

- Parti con le classi ritenute **centrali** ed importanti
- Stabilisci i dati ovvi e chiari che esse contengono e le loro relazioni con altre classi.
- Procedi con le classi meno importanti.
- Evita di aggiungere troppi attributi ed associazioni ad una classe.
  - Un sistema è più semplice se manipola meno informazioni
  - Le classi devono avere poche dipendenze da altre classi



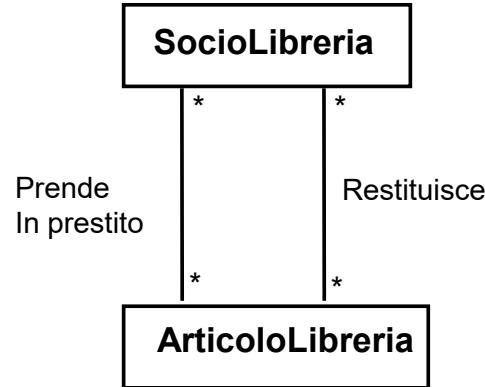
# SUGGERIMENTI PER TROVARE E SPECIFICARE ASSOCIAZIONI VALIDE

- Una associazione dovrebbe esistere se una classe:
  - *possiede*
  - *controlla*
  - *è collegata a*
  - *si riferisce a*
  - *è parte di*
  - *ha come parti*
  - *è membro di oppure*
  - *ha come membri*
- qualche altra classe del modello
- Specificare le molteplicità da entrambi i lati.
- Assegnare un nome chiaro all'associazione.



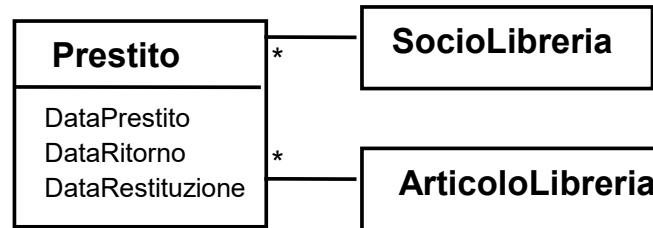
# AZIONI VS ASSOCIAZIONI

- Un errore comune consiste nel considerare *azioni* come se fossero *associazioni*.



Errore!

Nell'implementazione della classe **Prestito** compariranno due riferimenti a un oggetto **SocioLibreria** e un oggetto **ArticoloLibreria**



L'operazione **presta** crea un **Prestito** e  
L'operazione **restituisci** setta la data di restituzione

Entrambe le operazioni vanno assegnate  
alla classe **Prestito**



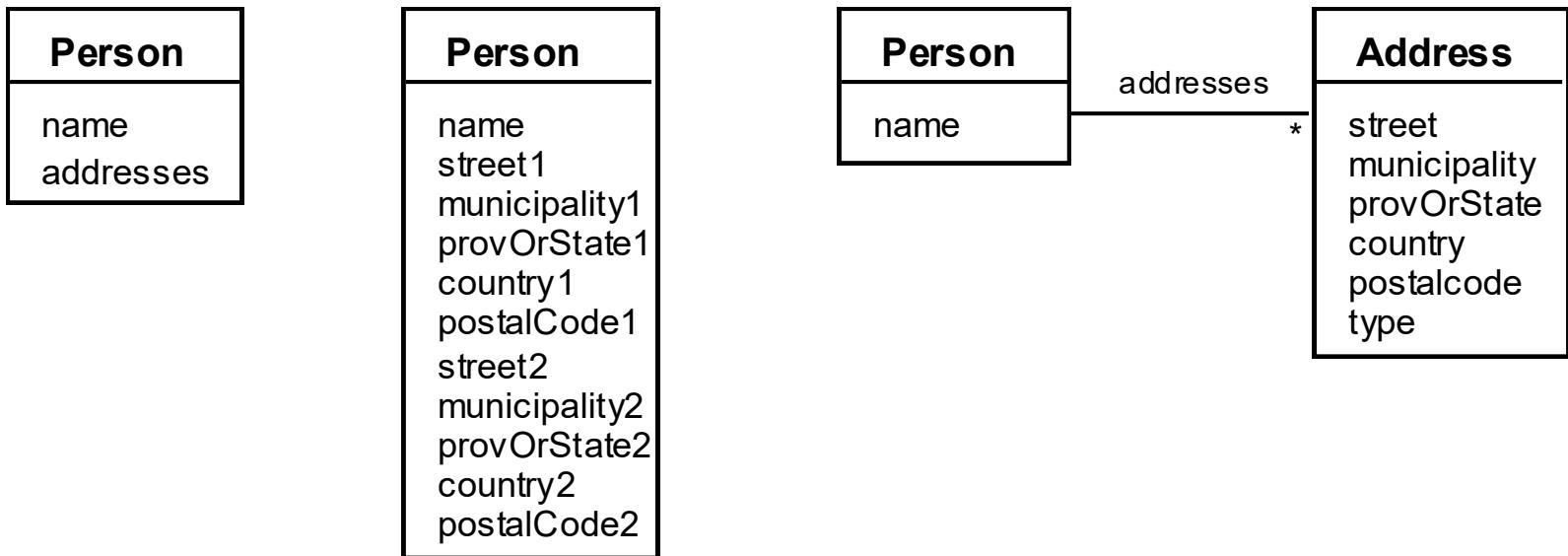
## 2.A. IDENTIFICARE GLI ATTRIBUTI

- Cercare le informazioni che devono essere conservate per ciascuna classe
- È possibile che nomi che sono stati scartati come classi, possano ora essere considerati attributi
- Un attributo dovrebbe in genere contenere un solo valore
  - Es. stringa, numerico



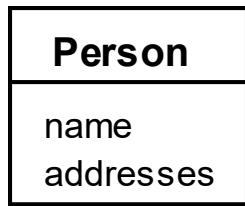
# SUGGERIMENTI PER IDENTIFICARE E SPECIFICARE ATTRIBUTI VALIDI

- È bene non avere molti attributi duplicati
- Se un sottoinsieme degli attributi di una classe forma un gruppo coerente, crea una classe distinta per questi attributi

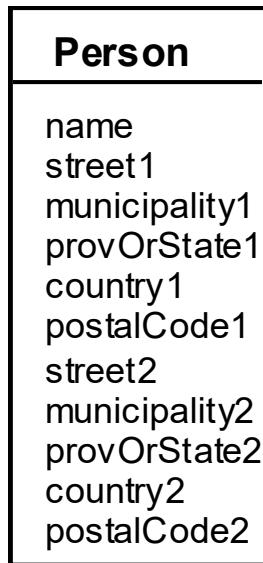


# SUGGERIMENTI PER IDENTIFICARE E SPECIFICARE ATTRIBUTI VALIDI

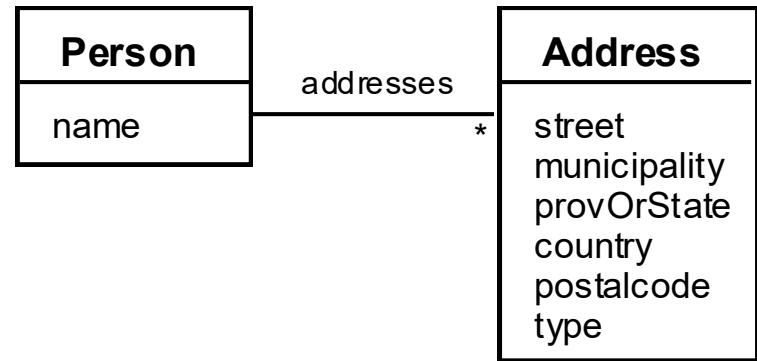
- È bene non avere molti attributi duplicati
- Se un sottoinsieme degli attributi di una classe forma un gruppo coerente, crea una classe distinta per questi attributi



**Errore: uso  
di attributi al  
plurale:  
sostituire  
con address  
[1..\*], ad  
esempio**



**Errore: troppi attributi  
duplicati, e incapacità  
di gestire più indirizzi**



**Bene: l'attributo  
tipo indica il tipo  
di indirizzo**

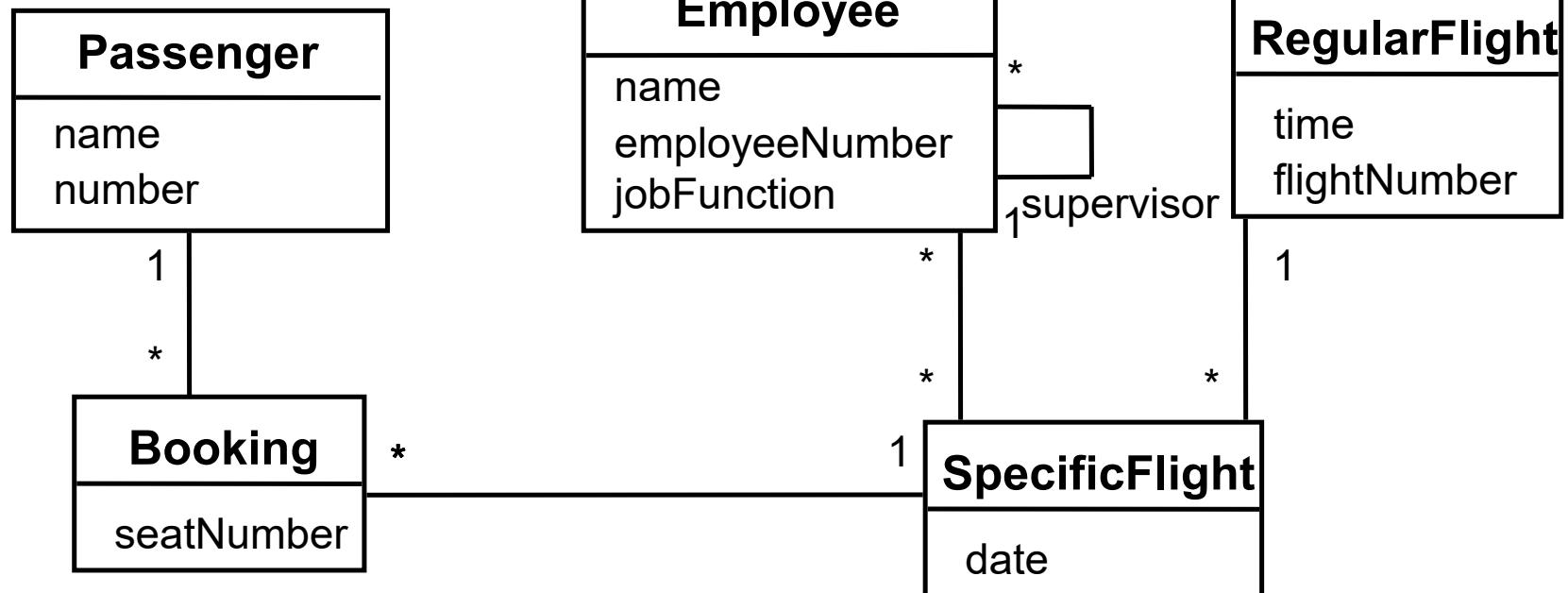


## ESEMPIO: VOLI

- Un passeggero può prenotare dei voli;
- Una prenotazione riguarda un passeggero e uno specifico volo;
- Un volo prevede molte prenotazioni per i suoi posti;
- Su un volo sono imbarcati molti lavoratori;
- Un lavoratore può lavorare su più voli;
- Un lavoratore ha un supervisore;
- Un volo viene ripetuto regolarmente in un certo orario e ha un numero identificativo prefissato.



# ESEMPIO: VOLI



- Un passeggero può prenotare dei voli;
- Una prenotazione riguarda un passeggero e uno specifico volo;
- Un volo prevede molte prenotazioni per i suoi posti;
- Su un volo sono imbarcati molti lavoratori;
- Un lavoratore può lavorare su più voli;
- Un lavoratore ha un supervisore;
- Un volo viene ripetuto regolarmente in un certo orario e ha un numero identificativo prefissato.

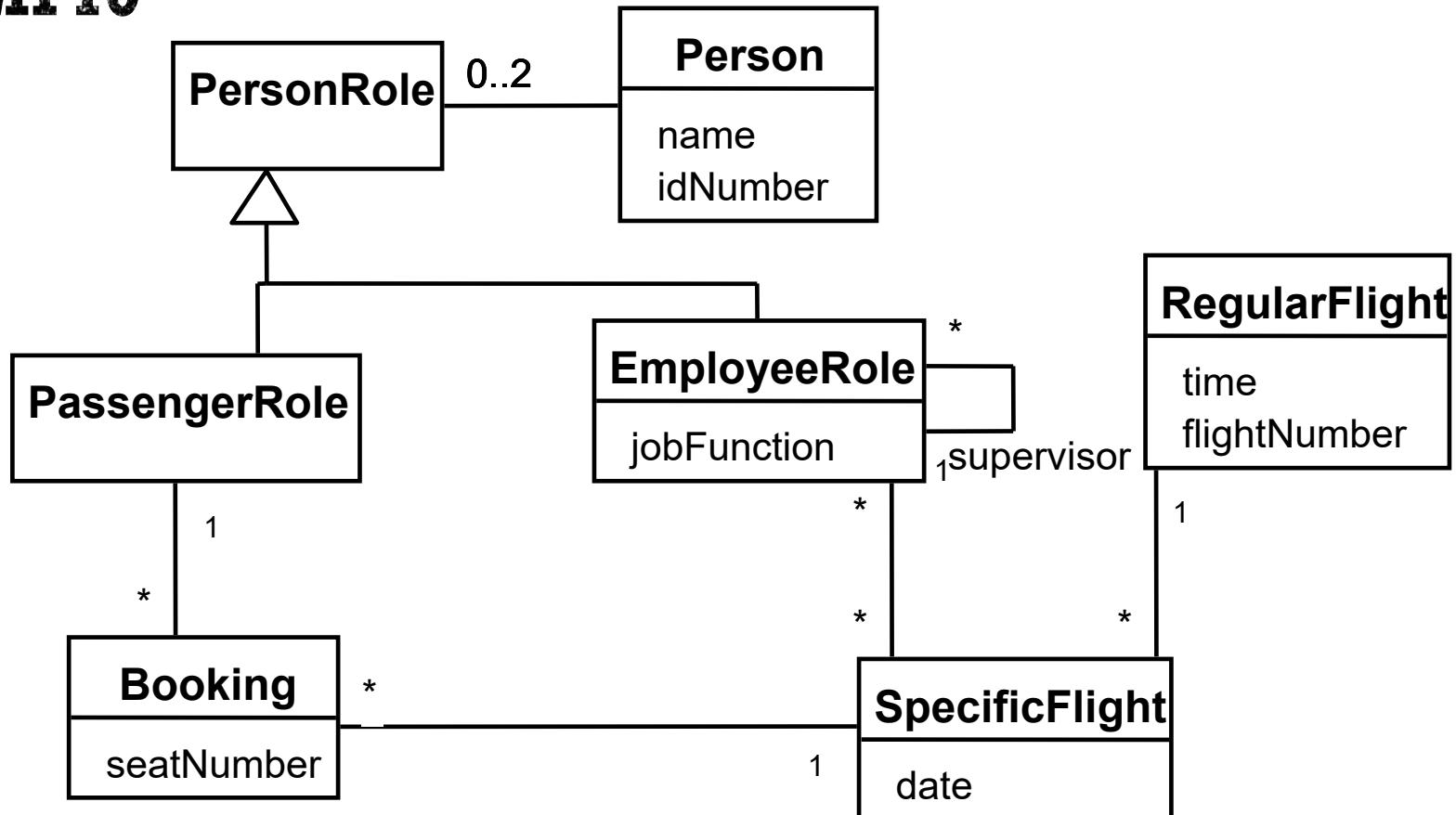


### 3. IDENTIFICARE GENERALIZZAZIONI E INTERFACCE

- Due modi per trovare le generalizzazioni:
  - bottom-up
    - Raggruppo classi simili creando una nuova superclasse
  - top-down
    - Cerco prima le classi più generali, e poi specializzo



# ESEMPIO



- Passagero e Impiegato sono entrambi persone con un nome e un idNumber, quindi ha senso inserire una generalizzazione
- Una persona però può agire sia da Passeggero che da Impiegato, quindi si introduce il concetto di Ruolo: un individuo che agisce da Passeggero e uno che agisce da Impiegato sono entrambi Persone che hanno un ruolo; una persona può avere nessuno, uno o due ruoli nel sistema



## 4. ASSEGNAME LE RESPONSABILITÀ ALLE CLASSI

- Una *responsabilità* è un qualcosa che è richiesto al sistema.
- La responsabilità di **ogni requisito funzionale** deve essere attribuita ad una delle classi, anche se tale requisito potrà essere svolto mediante una collaborazione fra più classi.
  - Tutte le responsabilità di una classe dovrebbero essere *chiaramente correlate*.
  - Se una classe ha troppe responsabilità, valutare l'ipotesi di *dividerla* in più classi
  - Se una classe non ha responsabilità, potrebbe essere *inutile*
  - Quando una responsabilità non può essere attribuita a nessuna delle classi esistenti, dovrebbe essere creata una *nuova classe*
- Per stabilire le responsabilità:
  - Cercare verbi e nomi che descrivono *azioni* nella descrizione del sistema



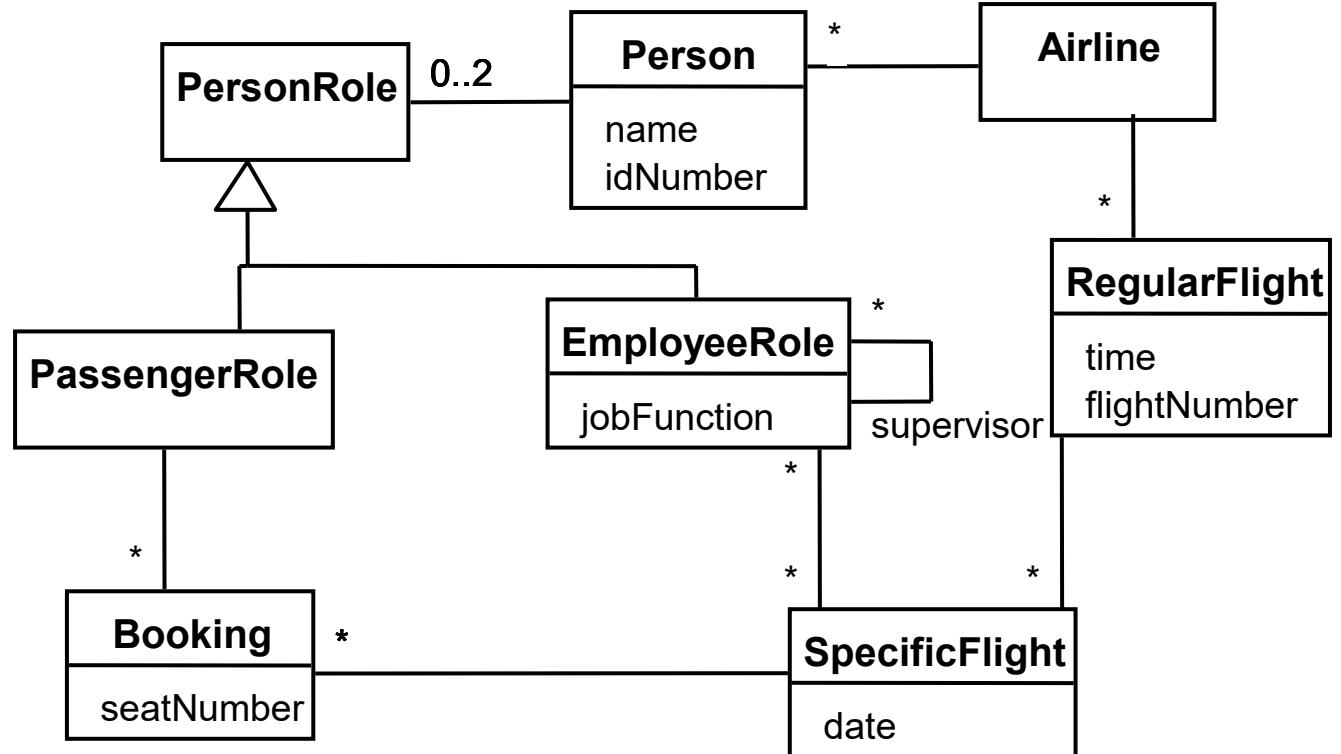
# CATEGORIE DI RESPONSABILITÀ

- Set e get dei valori degli attributi
- Creare ed inizializzare nuove istanze
- Prelevare da o memorizzare dati in una memoria persistente
- Distruggere istanze
- Aggiungere e cancellare istanze di associazioni
- Copiare, convertire, trasformare, trasmettere o fornire in output dati.
- Calcolare risultati numerici
- Navigare e cercare dati di particolari istanze
- Altro lavoro specifico
  
- In un diagramma del dominio (quindi senza elementi del progetto di dettaglio e dell'implementazione) la maggior parte di queste responsabilità non sono indicate
  - Viceversa, le responsabilità riconosciute in fase di analisi dovrebbero essere sempre considerate



# L'ESEMPIO DEI VOLI (CON RESPONSABILITÀ)

- Creare un nuovo VoloRegolare
- Cercare un VoloRegolare
- Modificare gli attributi di un volo
- Creare un VoloSpecifico
- Prenotare un passeggero
- Cancellare una prenotazione



| Responsabilità                                        | Classe                                                                                                                         |
|-------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|
| Creare un nuovo RegularFlight                         | RegularFlight <b>oppure meglio</b> Airline                                                                                     |
| Cercare un RegularFlight                              | Airline <b>che mantiene la collezione delle istanze dei voli</b>                                                               |
| Modificare gli attributi di un volo(RegularFlight)    | RegularFlight ( <b>ogni classe è responsabile dei propri attributi</b> )                                                       |
| Creare un VoloSpecifico                               | RegularFlight, <b>siccome un VoloSpecifico è un'occorrenza di VoloRegolare</b>                                                 |
| Prenotare un passeggero su un VoloSpecifico           | <b>O</b> SpecificFlight <b>o</b> PassengerRole: <b>meglio</b> PassengerRole ( <b>è il passeggero che decide di prenotare</b> ) |
| Cancellare una prenotazione (da parte del passeggero) | PassengerRole ( <b>è il passeggero ad annullare la prenotazione</b> )                                                          |



# ESEMPIO: VIDEOGIOCO

- Si vuole progettare un videogioco nel quale il giocatore impersona un agente di borsa, rivivendone le interazioni e basandosi sulle azioni delle società effettivamente quotate alla borsa di New York.
- Ogni **giocatore** ha una dotazione monetaria iniziale, pari a 100 milioni di dollari, che può essere investita acquistando delle **azioni**. Possono essere acquistate azioni di ognuna delle **società quotate** nel listino della **borsa** di New York.
- Le **azioni** di una **società** vengono acquistate al prezzo cui sono quotate nell'istante dell'**acquisto**. In fase di **acquisto**, il **giocatore** acquirente deve specificare anche il quantitativo di **azioni** che vuole acquistare. Il sistema provvederà a valutare l'effettiva disponibilità di liquidi da parte dell'acquirente e, in caso di acquisto possibile, scalerà dalla sua liquidità il denaro necessario all'**acquisto** delle **azioni**.
- Il **giocatore** potrà anche **vendere** le **azioni** che possiede, al prezzo cui sono quotate in quel momento (il sistema provvederà ad aggiornare la liquidità).
- **Acquisti** e **vendite** di **azioni** potranno essere effettuati soltanto nel periodo di tempo tra l'orario di apertura e l'orario di chiusura della **borsa** di New York.
- I **giocatori** possono anche scegliere di monitorare le **azioni** di alcune **società**. In tal caso il sistema manterrà un valore del prezzo delle **azioni** di ognuna delle **società monitorate** per ogni giorno nel quale il giocatore vorrà monitorarla.
- I **giocatori** sono organizzati in **squadre** di al più 4 elementi. Ogni **squadra** ha un nome e il sistema assegnerà un **premio mensile** alla squadra che ha il bilancio migliore. Una **bachecca** manterrà tutti i **premi** mensili ricevuti.



# RICERCA CLASSI

- Si vuole progettare un videogioco nel quale il giocatore impersona un agente di borsa, rivivendone le interazioni e basandosi sulle azioni delle società effettivamente quotate alla borsa di New York.
- Ogni **giocatore** ha una **dotazione monetaria iniziale**, pari a 100 milioni di dollari, che può essere investita acquistando delle **azioni**. Possono essere **acquistate** azioni di ognuna delle **società quotate** nel **listino** della **borsa** di New York.
- Le **azioni** di una **società** vengono **acquistate** al **prezzo** cui sono quotate nell'**istante dell'acquisto**. In fase di acquisto, il **giocatore acquirente** deve specificare anche il **quantitativo** di **azioni** che vuole acquistare. Il sistema provvederà a valutare l'**effettiva disponibilità di liquidi** da parte dell'**acquirente** e, in caso di acquisto possibile, scalerà dalla sua **liquidità** il denaro necessario all'**acquisto** delle **azioni**.
- Il **giocatore** potrà anche **vendere** le **azioni** che possiede, al **prezzo** cui sono quotate in quel **momento** (il sistema provvederà ad aggiornare la **liquidità**).
- Acquisti e vendite di azioni potranno essere effettuati soltanto nel periodo di tempo tra l'**orario di apertura** e l'**orario di chiusura** della **borsa** di New York.
- I **giocatori** possono anche scegliere di **monitorare** le **azioni** di alcune società. In tal caso il sistema manterrà un valore del **prezzo** delle **azioni** di ognuna delle **società monitorate** per ogni **giorno** nel quale il **giocatore** vorrà monitorarla.
- I **giocatori** sono **organizzati** in **squadre** di al più 4 elementi. Ogni **squadra** ha un **nome** e il sistema **assegnerà** un **premio mensile** alla squadra che ha il **bilancio** migliore. Una **bacheca** manterrà tutti i **premi mensili** ricevuti.



# RICERCA ASSOCIAZIONI ED ATTRIBUTI

- Si vuole progettare un videogioco nel quale il giocatore impersona un agente di borsa, rivivendone le interazioni e basandosi sulle azioni delle società effettivamente quotate alla borsa di New York.
- Ogni **giocatore** ha una **dotazione monetaria iniziale**, pari a 100 milioni di dollari, che può essere investita acquistando delle **azioni**. Possono essere acquistate azioni di ognuna delle **società quotate** nel **listino della borsa** di New York.
- Le **azioni** di una **società vengono acquistate** al **prezzo** cui sono quotate nell'**istante dell'acquisto**. In fase di acquisto, il **giocatore** acquirente deve specificare anche il **quantitativo** di **azioni** che vuole acquistare. Il sistema provvederà a valutare l'**effettiva disponibilità di liquidi** da parte dell'acquirente e, in caso di acquisto possibile, scalerà dalla sua **liquidità** il **denaro necessario all'acquisto** delle **azioni**.
- Il **giocatore** potrà anche vendere le **azioni** che **possiede**, al **prezzo** cui sono quotate in quel momento (il sistema provvederà ad aggiornare la **liquidità**).
- Acquisti e vendite di azioni potranno essere effettuati soltanto nel periodo di tempo tra **l'orario di apertura** e **l'orario di chiusura** della **borsa** di New York.
- I **giocatori** possono anche scegliere di monitorare le **azioni** di alcune società. In tal caso il sistema manterrà un valore del **prezzo** delle **azioni** di ognuna delle **società monitorate** per ogni giorno nel quale il **giocatore** vorrà monitorarla.
- I **giocatori** sono organizzati in **squadre** di al più 4 elementi. Ogni **squadra** ha un **nome** e il sistema assegnerà un **premio mensile** alla squadra che ha il **bilancio** migliore. Una **bachecca** manterrà tutti i premi mensili ricevuti.



# RICERCA GENERALIZZAZIONI

- Si vuole progettare un videogioco nel quale il giocatore impersona un agente di borsa, rivivendone le interazioni e basandosi sulle azioni delle società effettivamente quotate alla borsa di New York.
- Ogni **giocatore** ha una **dotazione monetaria iniziale**, pari a 100 milioni di dollari, che può essere investita acquistando delle **azioni**. Possono essere acquistate azioni di ognuna delle **società quotate** nel **listino della borsa** di New York.
- Le **azioni** di una **società** vengono acquistate al **prezzo** cui sono quotate nell'**istante dell'acquisto**. In fase di acquisto, il **giocatore** acquirente deve specificare anche il **quantitativo di azioni** che vuole acquistare. Il sistema provvederà a valutare l'effettiva **disponibilità di liquidi** da parte dell'acquirente e, in caso di acquisto possibile, scalerà dalla sua **liquidità** il **denaro necessario** all'**acquisto** delle **azioni**.
- Il **giocatore** potrà anche vendere le **azioni** che **possiede**, al **prezzo** cui sono quotate in quel momento (il sistema provvederà ad aggiornare la **liquidità**).
- Acquisti e vendite di azioni potranno essere effettuati soltanto nel periodo di tempo tra **l'orario di apertura** e **l'orario di chiusura** della **borsa** di New York.
- I **giocatori** possono anche scegliere di monitorare le **azioni** di alcune società. In tal caso il sistema manterrà un valore del **prezzo** delle **azioni** di ognuna delle **società monitorate** per ogni giorno nel quale il **giocatore** vorrà monitorarla.
- I **giocatori** sono organizzati in **squadre** di al più 4 elementi. Ogni **squadra** ha un **nome** e il sistema assegnerà un **premio mensile** alla squadra che ha il **bilancio** migliore. Una **bachecca** manterrà tutti i **premi mensili** ricevuti.

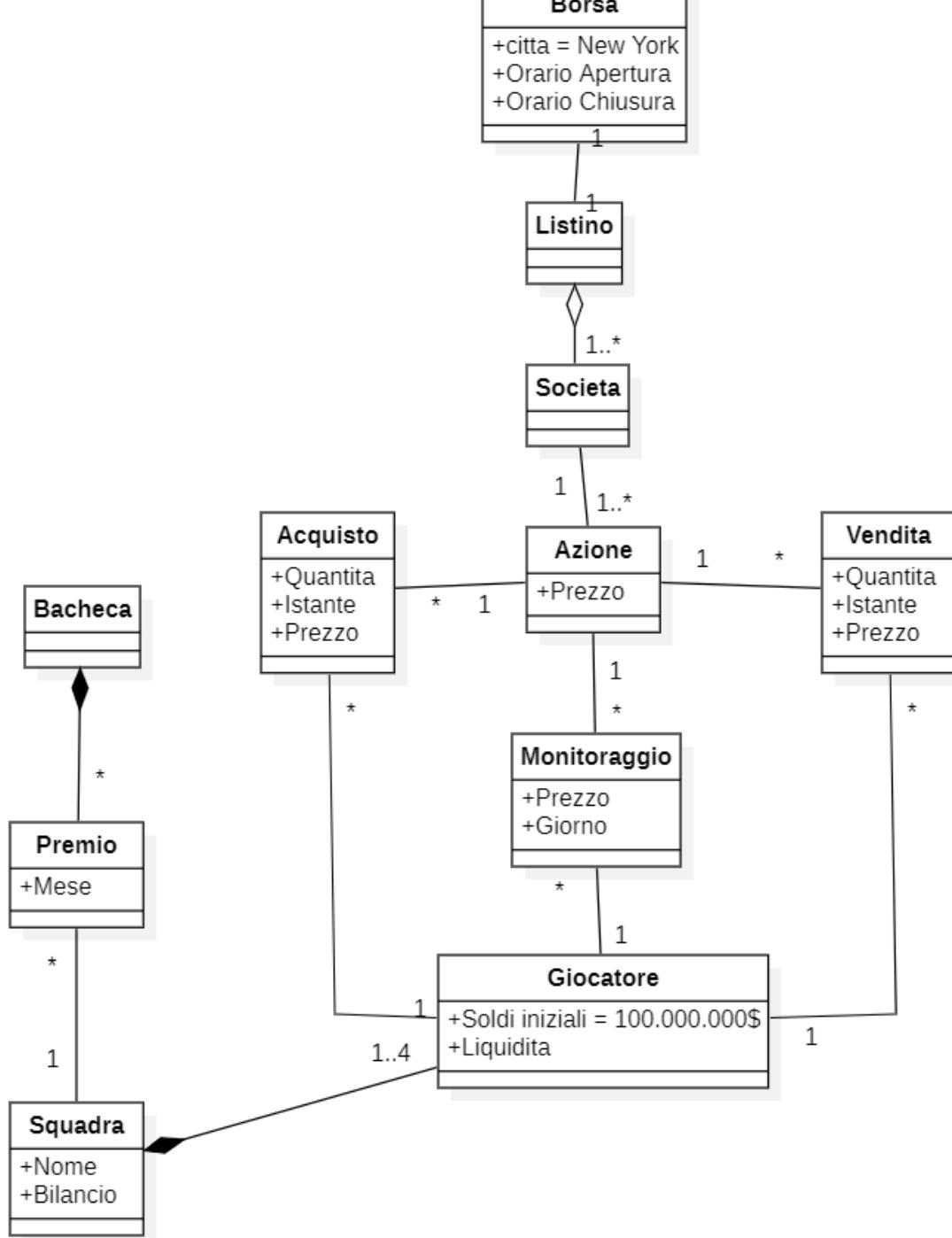


# RICERCA E ASSEGNAZIONE DI RESPONSABILITÀ

- Si vuole progettare un videogioco nel quale il giocatore impersona un agente di borsa, rivivendone le interazioni e basandosi sulle azioni delle società effettivamente quotate alla borsa di New York.
- Ogni **giocatore** ha una **dotazione monetaria iniziale**, pari a 100 milioni di dollari, che può **essere investita acquistando delle azioni**. Possono essere acquistate azioni di ognuna delle **società quotate** nel **listino della borsa** di New York.
- Le **azioni** di una **società** vengono acquistate al **prezzo** cui sono quotate nell'istante dell'**acquisto**. In fase di **acquisto**, il **giocatore** acquirente deve specificare anche il **quantitativo** di **azioni** che vuole acquistare. Il sistema provvederà a valutare l'effettiva **disponibilità di liquidi** da parte dell'acquirente e, in caso di acquisto possibile, **scalerà** dalla sua **liquidità** il **denaro necessario** all'**acquisto** delle **azioni**.
- Il **giocatore** potrà anche **vendere** le **azioni** che **possiede**, al **prezzo** cui sono quotate in quel momento (il sistema provvederà ad **aggiornare** la **liquidità**).
- Acquisti e vendite di azioni potranno essere effettuati soltanto nel periodo di tempo tra **l'orario di apertura** e **l'orario di chiusura** della **borsa** di New York.
- I **giocatori** possono anche **scegliere di monitorare** le **azioni** di alcune società. In tal caso il sistema manterrà un valore del **prezzo** delle **azioni** di ognuna delle **società monitorate** per ogni giorno nel quale il **giocatore** vorrà monitorarla.
- I **giocatori** sono organizzati in **squadre** di al più 4 elementi. Ogni **squadra** ha un **nome** e il sistema **assegnerà** un **premio mensile** alla squadra che ha il **bilancio** migliore. Una **bacheca** manterrà tutti i premi mensili ricevuti.



# POSSIBILE SOLUZIONE



# ESEMPIO: AZIENDA ALIMENTARE

- Una azienda produttrice di prodotti alimentari, vuole organizzare un sistema informativo aziendale. Tutti gli utenti dell'applicazione devono essere in grado di visualizzare informazioni relative al catalogo dei prodotti. Inoltre i dipendenti devono essere in grado di accedere ad informazioni relative alle loro mansioni.
- I prodotti sono organizzati in linee di prodotto che accomunano prodotti dello stesso tipo: ad esempio, due linee possono essere pasta e sughi. I prodotti possono essere confezionati in diversi stabilimenti. I dipendenti si suddividono in diverse categorie: i manager, che sono responsabili di una o più linee di produzione (ogni linea è però gestita esattamente da tre manager, per assicurare una gestione equa), i supervisori della produzione che sono responsabili di tutti i prodotti di una specifica linea in uno specifico stabilimento, e gli operai che lavorano su uno specifico prodotto in un determinato stabilimento.
- L'applicazione deve consentire ai manager di accedere alle informazioni relative ai dipendenti di cui sono responsabili, e a tutti di accedere alle informazioni sui prodotti.



# RICERCA CLASSI

- Una azienda produttrice di prodotti alimentari, vuole organizzare un sistema informativo aziendale. Tutti gli utenti dell'applicazione devono essere in grado di visualizzare informazioni relative al **catalogo** dei prodotti. Inoltre i **dipendenti** devono essere in grado di accedere ad informazioni relative alle loro mansioni.
- I **prodotti** sono organizzati in **linee di prodotto** che accomunano **prodotti** dello stesso tipo: ad esempio, due linee possono essere pasta e sughi. I **prodotti** possono essere confezionati in diversi **stabilimenti**. I **dipendenti** si suddividono in diverse categorie: i **manager**, che sono responsabili di una o più **linee di produzione** (ogni **linea** è però gestita esattamente da tre **manager**, per assicurare una gestione equa), i **supervisori della produzione** che sono responsabili di tutti i **prodotti** di una specifica **linea** in uno specifico **stabilimento**, e gli **operai** che lavorano su uno specifico **prodotto** in un determinato **stabilimento**.
- L'applicazione deve consentire ai **manager** di accedere alle informazioni relative ai **dipendenti** di cui sono responsabili, e a tutti di accedere alle informazioni sui **prodotti**.



# RICERCA ASSOCIAZIONI ED ATTRIBUTI

- Una azienda produttrice di prodotti alimentari, vuole organizzare un sistema informativo aziendale. Tutti gli utenti dell'applicazione devono essere in grado di visualizzare informazioni relative al **catalogo dei prodotti**. Inoltre i **dipendenti** devono essere in grado di accedere ad informazioni relative alle loro **mansioni**.
- I **prodotti sono organizzati in linee di prodotto** che accomunano **prodotti** dello stesso **tipo**: ad esempio, due linee possono essere pasta e sughi. I **prodotti possono essere confezionati in diversi stabilimenti**. I **dipendenti** si suddividono in diverse categorie: i **manager**, che sono responsabili di una o più linee di produzione (ogni linea è però gestita esattamente da tre manager, per assicurare una gestione equa), i **supervisori della produzione** che sono responsabili di tutti i prodotti di una specifica linea in uno specifico stabilimento, e gli **operai** che lavorano su uno specifico prodotto in un determinato stabilimento.
- L'applicazione deve consentire ai **manager** di accedere alle informazioni relative ai **dipendenti** di cui **sono responsabili**, e a tutti di accedere alle informazioni sui **prodotti**.



# RICERCA GENERALIZZAZIONI

- Una azienda produttrice di prodotti alimentari, vuole organizzare un sistema informativo aziendale. Tutti gli utenti dell'applicazione devono essere in grado di visualizzare informazioni relative al **catalogo dei prodotti**. Inoltre i **dipendenti** devono essere in grado di accedere ad informazioni relative alle loro **mansioni**.
- I **prodotti sono organizzati in linee di prodotto** che accomunano **prodotti** dello stesso **tipo**: ad esempio, due linee possono essere pasta e sughi. I **prodotti** possono essere **confezionati in diversi stabilimenti**. I **dipendenti** si **suddividono in diverse categorie**: i **manager**, che sono responsabili di una o più linee di produzione (ogni linea è però gestita esattamente da tre manager, per assicurare una gestione equa), i **supervisori della produzione** che sono responsabili di tutti i prodotti di una specifica linea in uno specifico stabilimento, e gli **operai** che lavorano su uno specifico prodotto in un determinato stabilimento.
- L'applicazione deve consentire ai **manager** di accedere alle informazioni relative ai **dipendenti** di cui **sono responsabili**, e a tutti di accedere alle informazioni sui **prodotti**.



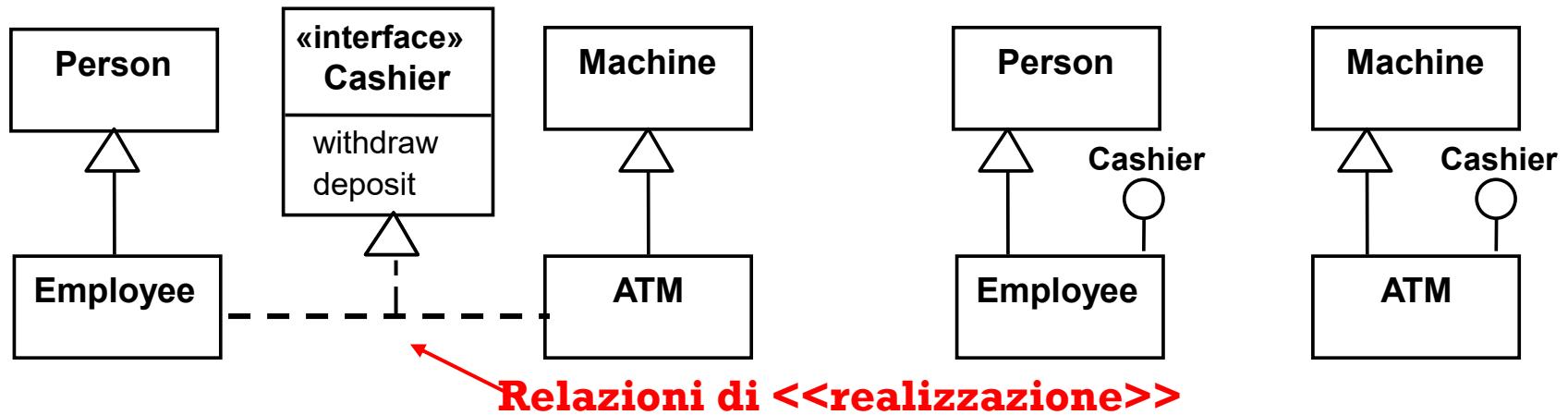
# RICERCA E ASSEGNAZIONE DI RESPONSABILITÀ

- Una azienda produttrice di prodotti alimentari, vuole organizzare un sistema informativo aziendale. Tutti gli utenti dell'applicazione devono essere in grado di **visualizzare informazioni** relative al **catalogo dei prodotti**. Inoltre i **dipendenti** devono essere in grado di **accedere ad informazioni** relative alle loro **mansioni**.
- I **prodotti sono organizzati in linee di prodotto** che accomunano **prodotti** dello stesso **tipo**: ad esempio, due linee possono essere pasta e sughi. I **prodotti** possono essere **confezionati in diversi stabilimenti**. I **dipendenti** si **suddividono in diverse categorie**: i **manager**, che sono responsabili di **una o più linee di produzione** (ogni **linea** è però gestita esattamente da **tre manager**, per assicurare una gestione equa), i **supervisori della produzione** che sono responsabili di **tutti i prodotti** di **una specifica linea** in **uno specifico stabilimento**, e gli **operai** che lavorano su **uno specifico prodotto** in **un determinato stabilimento**.
- L'applicazione deve consentire ai **manager** di **accedere alle informazioni** relative ai **dipendenti** di cui **sono responsabili**, e a tutti di accedere alle **informazioni sui prodotti**.



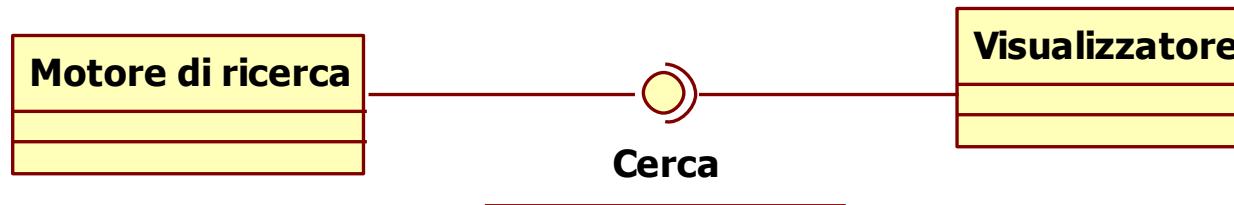
# INTERFACCIA

- Un'interfaccia descrive una porzione del comportamento visibile di un insieme di oggetti.
  - Un' *interfaccia* è simile ad una classe, tranne che essa non possiede variabili d'istanza nè metodi implementati.
  - Una o più classi possono fornire l'implementazione dell'interfaccia.



# NOTAZIONE “Lollipop”

- Una notazione molto utilizzata per le interfacce è quella a “Lollipop”:
  - La pallina (lollipop) rappresenta l’interfaccia esposta da una classe (quindi è una realization)
  - Il semicerchio (socket) rappresenta il servizio richiesto (quindi è una dependency)
- Esempio:
  - un motore di ricerca fornisce la possibilità di accedere al proprio servizio Cerca tramite un’interfaccia
  - La classe visualizzatore richiama la ricerca



# DEPENDENCY

- Una Dependency rappresenta una relazione tra le istanze di due classi, che viene a realizzarsi a tempo di esecuzione.
- Esempi: c'è dependency dalla classe A verso la classe B se:
  - Metodi della classe A vanno a leggere/scrivere/modificare il valore di attributi di oggetti della classe B;
  - Metodi della classe A invocano metodi della classe B;
    - caso particolare: la classe A istanzia oggetti della classe B (ovvero ne invoca il costruttore)
- La Dependency si rappresenta con una **linea tratteggiata** che termina con una freccia dall'oggetto dipendente verso quello da cui dipende.



# DEPENDENCY

- Le relazioni di dependency sono individuate principalmente durante la fase di progetto di dettaglio: esse raramente contribuiscono al modello concettuale
- Esprimono una dipendenza (**accoppiamento**) tra le classi, nel senso che la classe origine della dipendenza non potrà essere riusata senza la classe destinazione della dependency. Una modifica nella classe da cui c'è dipendenza implicherà modifiche anche nella classe dipendente.
- Analogamente, la correttezza della classe da cui parte una relazione di dipendenza è condizionata alla verifica della correttezza della classe dipendente



# TIPI DI DIPENDENZE IN UML 2.0

- Per distinguere diverse tipologie di dipendenze, UML mette a disposizione gli *stereotipi*, che possono essere indicati con keywords scritti tra parentesi angolari doppie
- Alcuni esempi di stereotipi:

<<call>>                            Chiamata di metodo

<<create>>                        Creazione di un'istanza di oggetto

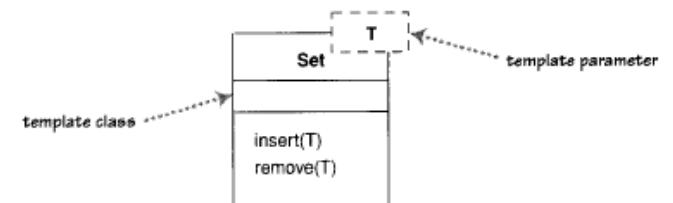
<<use>>                            Utilizza un attributo



# ULTERIORI NOTAZIONI (CENNI)

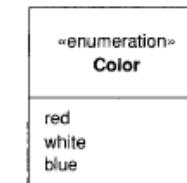
## ■ Classi parametriche (template)

- Utili più che altro nella modellazione di diagrammi di progetto di dettaglio di applicazioni da sviluppare in C++ o simili

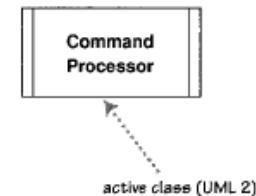


## ■ Tipi enumerativi

- Utilizzano la stessa notazione delle classi ma esprimono un elenco di valori



- Per qualsiasi altro elemento bisogna consultare la guida di riferimento ... oppure inventare stereotipi comprensibili!



# **SEQUENCE DIAGRAM**

UML Distilled, Capitolo 4



# DIAGRAMMI DI INTERAZIONE

- I diagrammi di Interazione sono usati per modellare gli aspetti dinamici di un sistema software, evidenziando in particolare le interazioni tra gli elementi che li compongono (e che sono stati descritti nei diagrammi strutturali)
  - Ci sono quattro tipi di diagrammi di interazione :
    - *Sequence diagrams*
    - *Communication diagrams*
      - *in UML 1 erano chiamati collaboration diagrams*
    - *Interaction Overview Diagrams*
    - *Timing Diagrams*
  - I diagrammi di interazione appartengono alla famiglia dei *Behaviour Diagram*

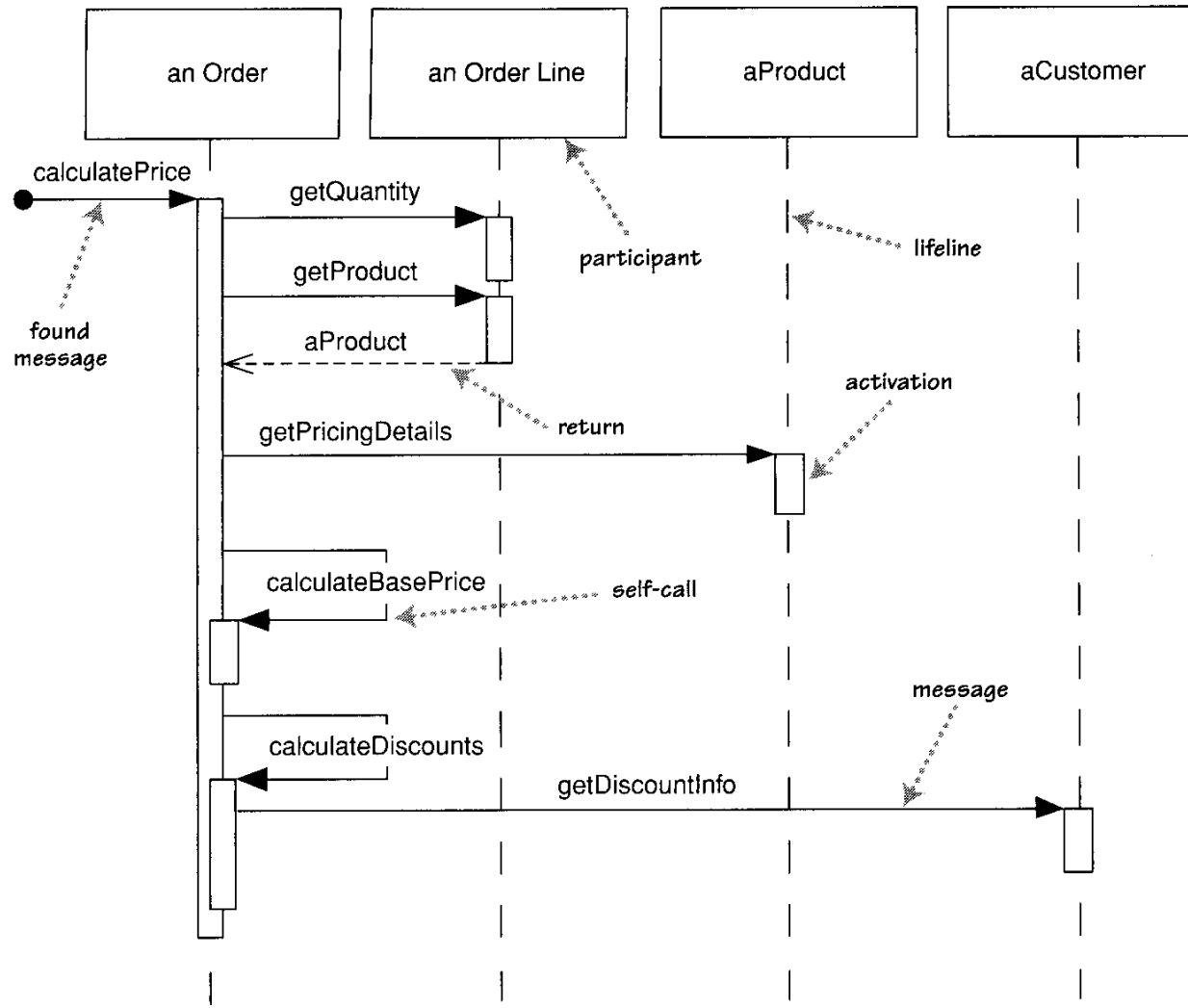


# SEQUENCE DIAGRAMS

- Rappresentano il tipo di diagramma di interazione largamente più utilizzato
- In generale, un sequence diagram modella le interazioni tra uno o più attori e il sistema software, nell'ambito dell'esecuzione di uno scenario di esecuzione
  - Le interazioni tra attori e sistema e tra le varie parti del sistema sono modellate in forma di *messaggi*, così come prevede il paradigma object-oriented



# SEQUENCE DIAGRAMS: ESEMPIO

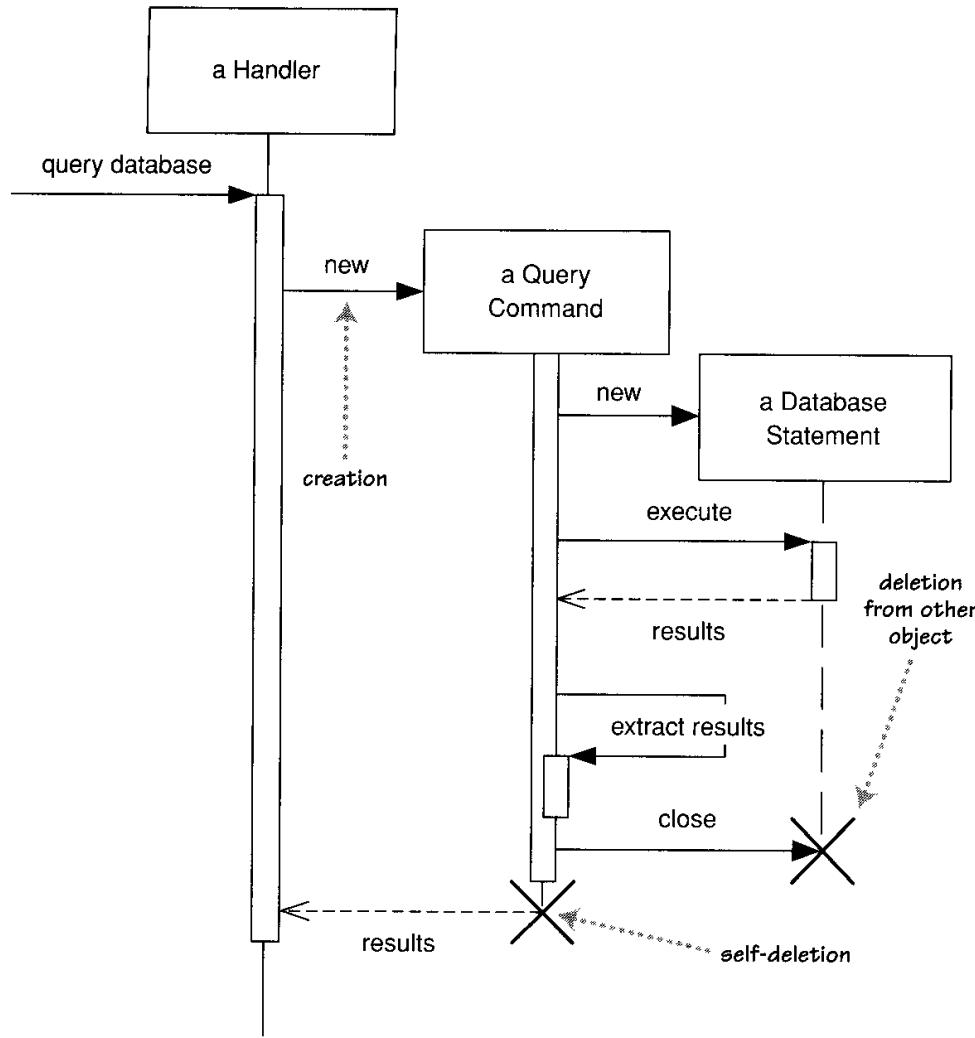


# SEQUENCE DIAGRAM: ELEMENTI

- Istanze di classi (oggetti)
  - Rappresentate da rettangoli col nome della classe e l'identificatore dell'oggetto sottolineati (notazione UML 1) oppure semplicemente con un nome dal quale si evinca che si sta considerando un'istanza della classe (ad esempio *anOrder* oppure *aProduct*)
- Attori o EndPoint
  - Sono riportati sulla sinistra, con frecce di interazione verso oggetti del sistema
  - Possono anche non essere riportati, nel caso in cui lo scenario venga avviato a sua volta da un altro scenario
  - Rappresentano la persona o il metodo che dà l'avvio all'elaborazione
- Messaggi
  - Rappresentati come frecce da un attore ad un oggetto, o fra due oggetti.
  - I messaggi di ritorno (se riportati) hanno una linea tratteggiata
  - I messaggi sincroni terminano con una freccia triangolare piena
  - I messaggi asincroni terminano con una freccia semplice →
  - Un messaggio può anche insistere all'interno di uno stesso oggetto: in tal caso è indicato da un autoanello
  - L'ordine dei messaggi (dall'alto verso il basso) ricalca l'ordine sequenziale con il quale vengono scambiati



# SEQUENCE DIAGRAMS



# LIFELINES E BARRE DI ATTIVAZIONE

- Lifelines (linee di vita)

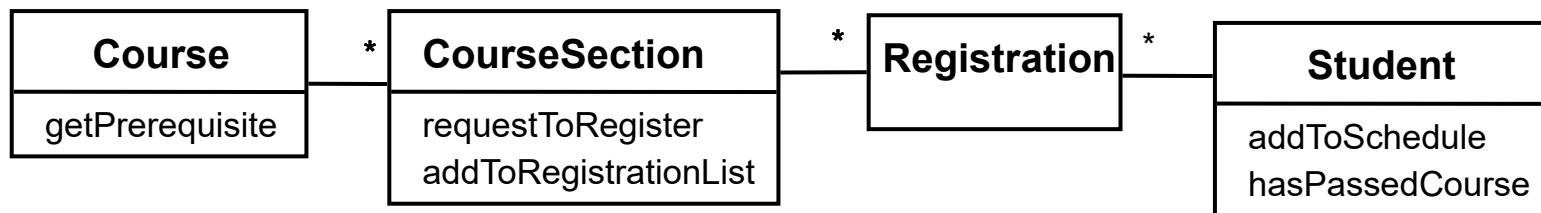
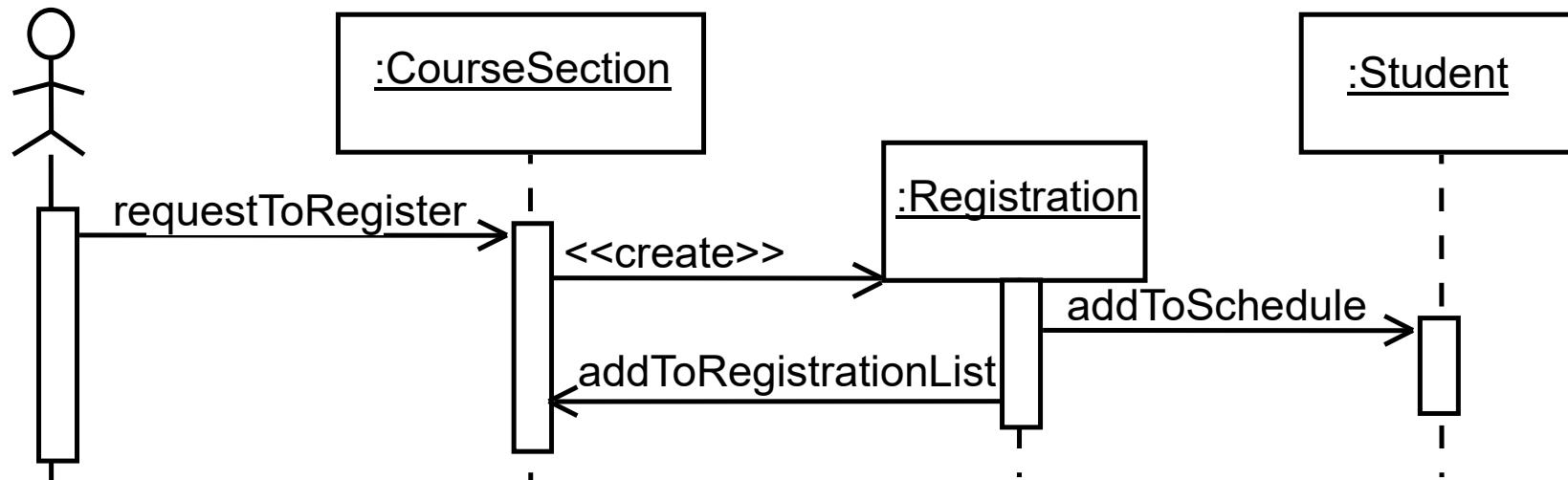
- Si tratta di linee tratteggiate verticali, che partono dal rettangolo rappresentativo dell'oggetto e giungono fino al fondo del diagramma. Indicano il periodo temporale di vita dell'oggetto, dalla sua costruzione alla sua distruzione
  - Le linee di vita di oggetti staticamente definiti (*static*) partono dalla cima del diagramma
  - Una richiesta al costruttore può creare un oggetto: in questo caso l'oggetto viene disegnato al termine della freccia corrispondente al messaggio di creazione
  - La distruzione di oggetti è indicata da una croce (ics) che interrompe la lifeline dell'oggetto

- Activation box (Barra di attivazione )

- E' rappresentata da un rettangolo che copre una parte della lifeline di un oggetto cui giunge un messaggio
- Rappresenta idealmente il periodo di tempo necessario per elaborare la richiesta giunta all'oggetto



# SEQUENCE DIAGRAM: UN ESEMPIO

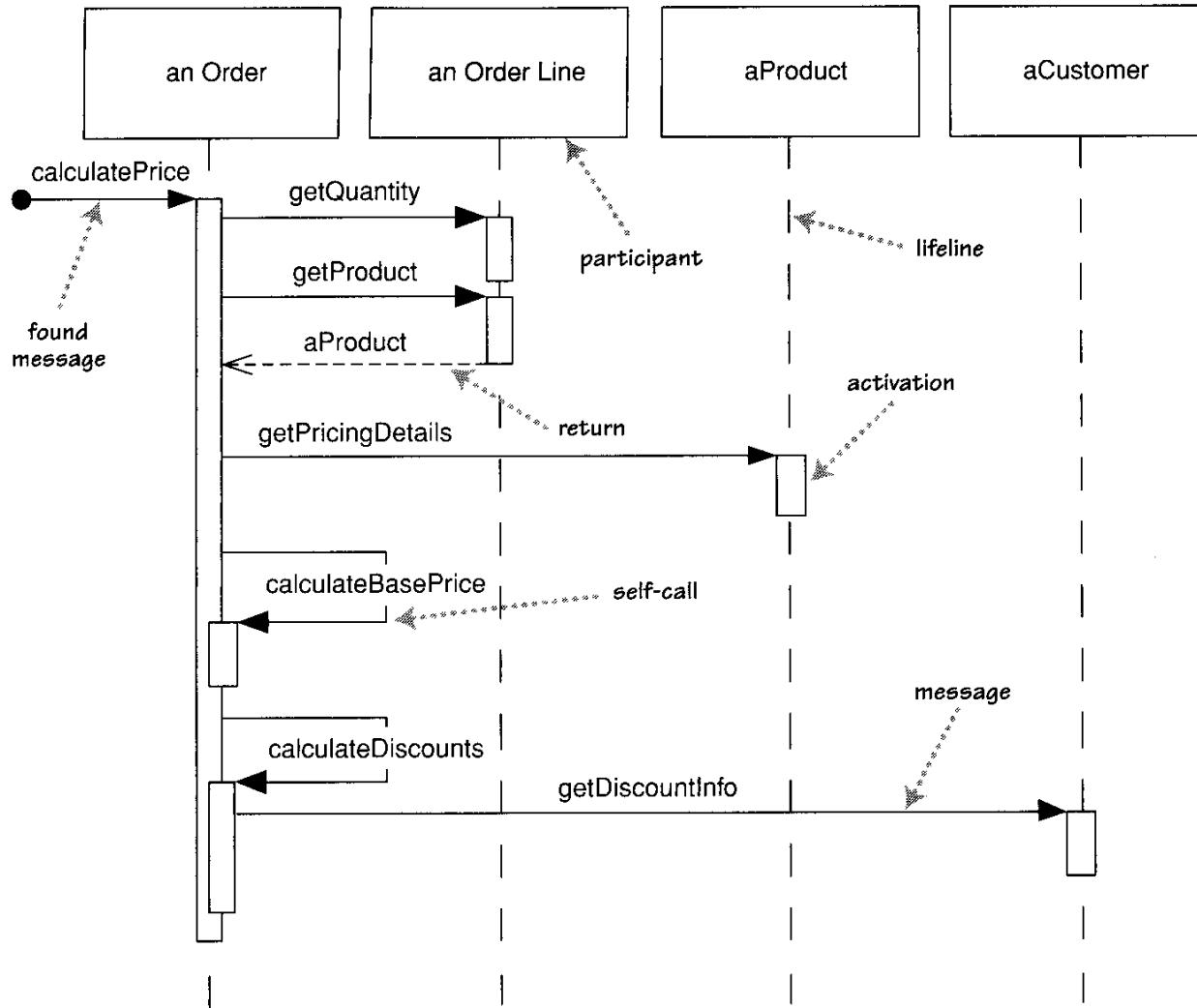


# SEQUENCE DIAGRAM: CAMPO DI APPLICAZIONE

- In fase di progettazione di dettaglio i sequence diagram descrivono realizzazioni di metodi
  - Come oggetti compaiono istanze delle classi di progetto di dettaglio
  - Come messaggi compaiono le chiamate di metodo sugli oggetti
    - Al posto degli elementi architetturali dovrebbero comparire oggetti delle classi che ne consentono l'accesso (ad esempio, anzichè database potrebbe comparire l'oggetto JDBC sul quale si effettuano le query)
    - Per la descrizione di algoritmi, i sequence diagram non sono lo strumento più adatto: ad essi verranno spesso preferiti activity e statechart diagrams



# SEQUENCE DIAGRAMS: ESEMPIO



# SEQUENCE DIAGRAM: CICLI E CONDIZIONI

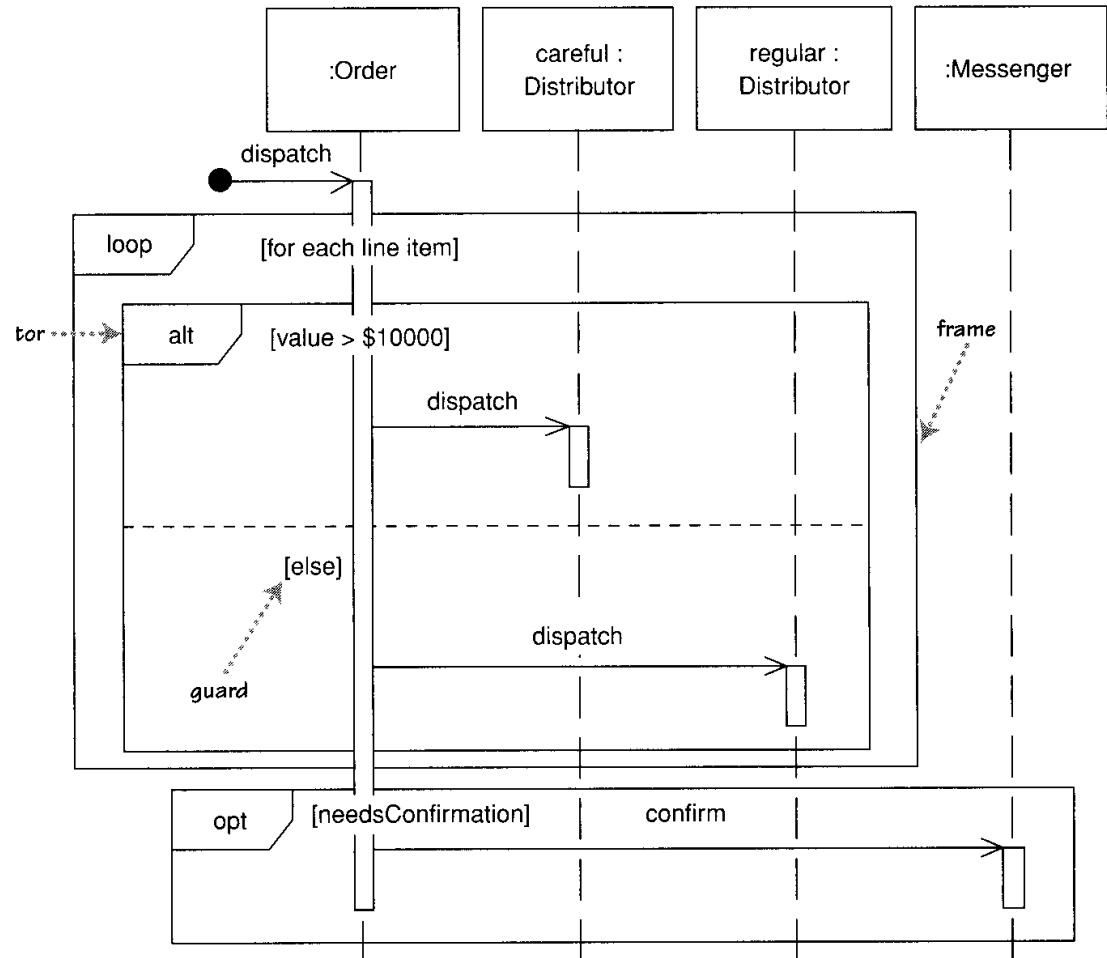
- Cicli e condizioni si indicano con un riquadro (*frame*) che racchiude una sottosequenza di messaggi
  - Nell'angolo in alto è indicato il costrutto. Tra i costrutti possibili
    - Loop (ciclo while-do o do-while): la condizione è indicata tra parentesi quadra all'inizio o alla fine
    - Alt (if-then-else): la condizione si indica in cima; se ci sono anche dei rami else allora si usa una linea tratteggiata per separare la zona *then* dalla zona *else* indicando eventualmente un'altra condizione accanto alla parola *else*
    - Opt (if-then): racchiude una sottosequenza che viene eseguita solo se la condizione indicata in cima è verificata
      - Sono possibili anche altri costrutti per indicare parallelismo, regioni critiche, etc.
- In realtà, è buona norma utilizzare altri tipi di diagramma quando l'algoritmo da modellare si fa complesso



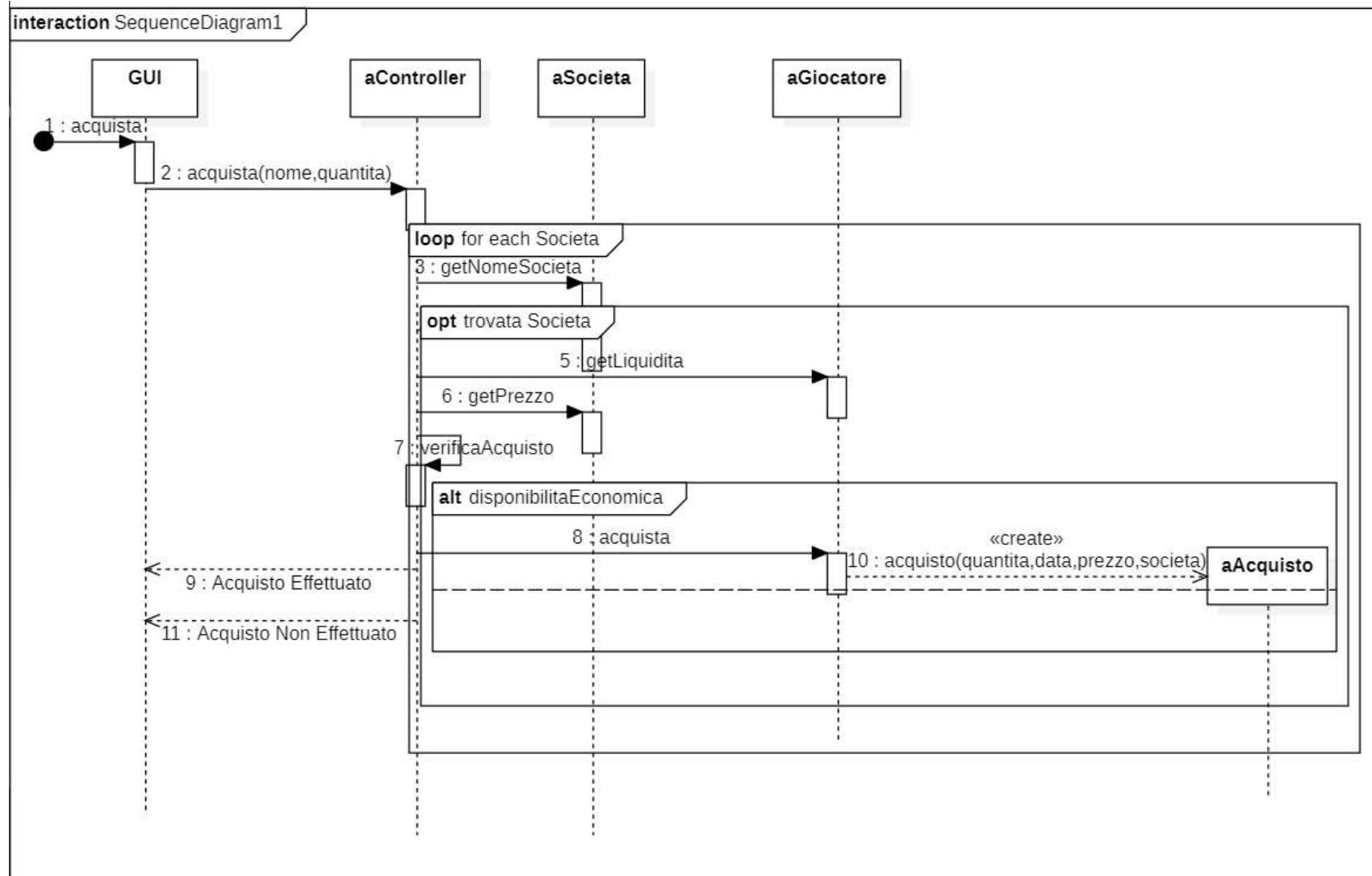
# SEQUENCE DIAGRAMS: ESEMPIO

```
procedure dispatch
foreach (lineitem)
  if (product.value > $10K)
    careful.dispatch
  else
    regular.dispatch
  end if
end for
if (needsConfirmation) messenger.confirm
end procedure
```

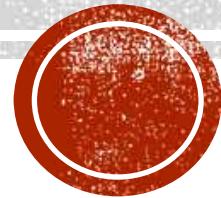
Questa notazione è stata  
introdotta con UML  
versione 2



# SEQUENCE DIAGRAM ACQUISTO



# CONCURRENT VERSIONING SYSTEMS



# Gestione delle versioni di un software

- I sistemi software
  - non sono realizzati in una sola sessione di programmazione
  - spesso non sono realizzati da un solo programmatore
  - Non sono rilasciati una sola volta
    - La correzione dei difetti può portare a nuovi rilasci
    - L'introduzione di nuove funzionalità porta a nuovi rilasci
  - Spesso un software necessita di essere rilasciato in diverse configurazioni
    - Versioni complete e versioni ridotte
    - Versioni in diverse lingue
    - Versioni diverse per diverse configurazioni hardware
    - ...
- Per tutte queste necessità, è opportuno utilizzare un sistema di gestione del ciclo di vita di software, o quantomeno, della storia delle sue versioni



# Requisiti di un sistema per la gestione delle versioni

- Supporto per la gestione delle versioni
  - Identificazione delle versioni e delle release
  - Gestione della memorizzazione
  - Registrazione dello storico delle modifiche
  - Sviluppo indipendente
    - Gestione dei branch e delle versioni concorrenti
- Supporto alla build automation
- Supporto ad attività DevOps
  - Continuous Integration
  - Continuous Delivery / Deployment



# Protocolli e strumenti per la gestione delle versioni

- CVS
- SVN
- Git
- ...



# CVS: Concurrent Versioning System

- Sistema di controllo delle versioni di un progetto legato alla produzione e alla modifica di file. In pratica, permette a un gruppo di persone di lavorare simultaneamente sullo stesso gruppo di file (generalmente si tratta di sorgenti di un programma), mantenendo il controllo dell'evoluzione delle modifiche che vengono apportate.
- Per attuare questo obiettivo, il sistema CVS mantiene un deposito centrale (*repository*) dal quale i collaboratori di un progetto possono ottenere una copia di lavoro. I collaboratori modificano i file della loro copia di lavoro e sottopongono le loro modifiche al sistema CVS che le integra nel deposito.
- Il compito di un sistema CVS non si limita a questo; per esempio è sempre possibile ricostruire la storia delle modifiche apportate a un gruppo di file, oltre a essere anche possibile ottenere una copia che faccia riferimento a una versione passata di quel lavoro.
- Storicamente, il primo strumento open source di gestione della configurazione con ampia diffusione



# Modello Lock/Modify/Unlock

- In principio, l'unico modello secondo il quale più programmati accedevano in concorrenza ai diversi file di un progetto era il modello “*lock/modify/unlock*”
  - Secondo questo modello un utente che vuole modificare un file del progetto, prima di tutto lo blocca (*lock*), impedendo a chiunque altro di modificarlo, dopodichè, quando ha terminato le modifiche lo sblocca (*unlock*)
  - Questa strategia, per quanto garantisca la massima sicurezza da problemi di manomissione contemporanea involontaria, non ottimizza nel modo migliore le operazioni
    - Adoperando questo modello, si tende a spezzettare il più possibile un progetto, in modo da ridurre gli impedimenti al lavoro causati dai lock

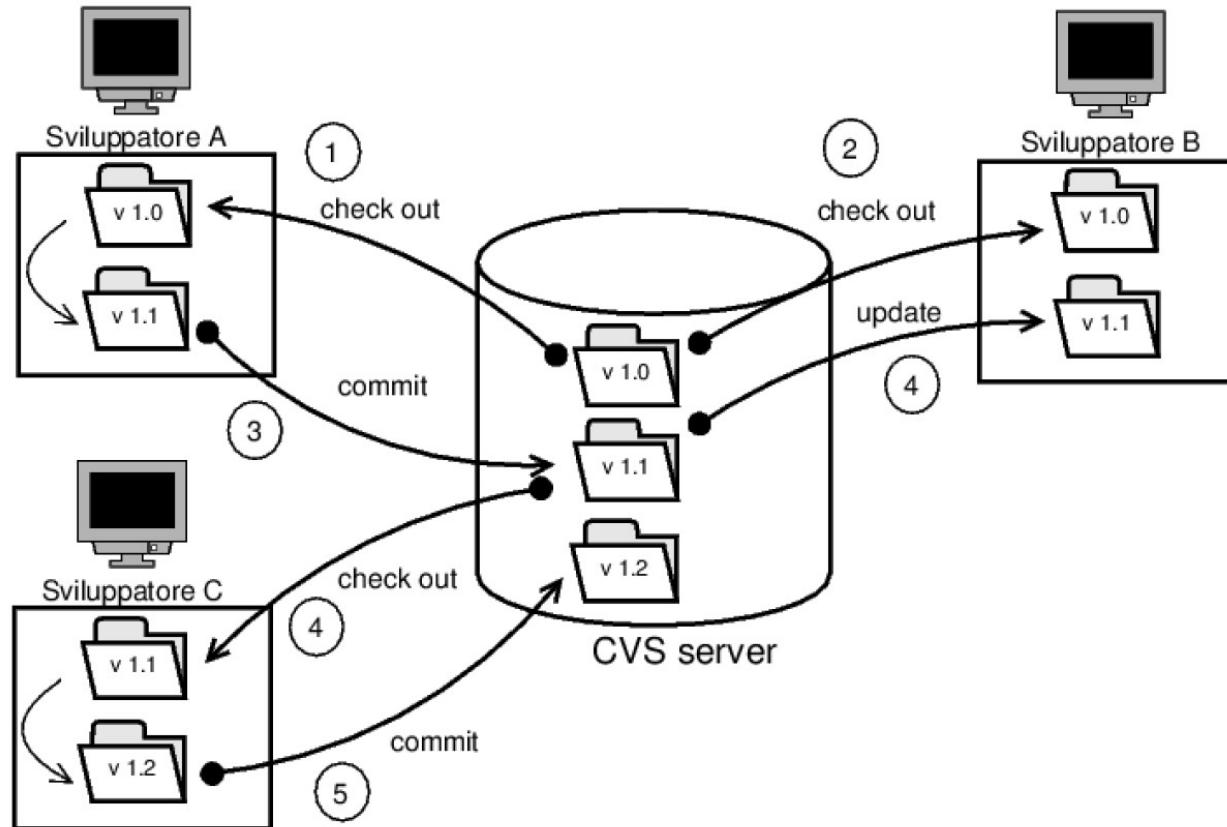


# Modello Copy/Modify/Merge

- In alternativa, il modello *Copy/Modify/Merge* prevede che:
  1. Lo sviluppatore A scarica una copia del progetto (*working copy* o *sandbox*) dal server CVS (*repository*)
  2. Applica liberamente tutte le modifiche. Nel frattempo altri programmatori (B) potrebbero fare lo stesso
  3. Al termine del suo lavoro il programmatore A aggiorna il progetto sul server CVS (*commit*)
  4. Altri programmatori potrebbero richiedere aggiornamenti della loro working copy (*update*) al repository o generare delle ulteriori versioni (*commit*)



# Modello Copy/Modify/Merge



# Conflitti

- Nel caso in cui due programmatori modificano lo stesso file, il sistema CVS può fondere (*merge*) le due versioni, sovrapponendo le modifiche, allorchè si riferiscano a linee di codice diverse
- Se invece ci sono modifiche alle stesse righe di codice si verifica un *conflitto*
  - La soluzione del conflitto è in questo caso demandata ai singoli programmatori: la versione unificata che viene generata diventa la nuova versione di riferimento
  - In alternativa si potrebbe scegliere di mantenere entrambe le versioni come alternative, generando un *branch*



# CVS

- Il sistema CVS è un software, presente per diversi sistemi operativi, che consente di gestire a linea di comando le principali operazioni previste dai modelli lock/modify/unlock e *copy/modify/merge*
- Il lato server gestisce il *repository*, contenente sia tutti i file da gestire che tutte le informazioni sulle versioni
  - In alternativa il deposito potrebbe anche trovarsi sulla macchina client
- Il lato client consente di effettuare tutte le operazioni riguardanti la copia locale (*sandbox*) del progetto



# Operazioni CVS

- Ogni persona coinvolta nel progetto, ha una copia locale dei file (*sandbox*)
- Chi avvia il progetto crea per la prima volta il repository (*Make new module*), indicando anche quali directory dovranno essere gestite
- Successivamente un qualsiasi collaboratore può aggiungere nuovi file/directory al CVS (*add*)
- Un collaboratore che voglia inserirsi nel CVS dovrà per prima cosa effettuare il *Checkout* per prelevare dal repository le versioni più recenti di ogni file



# Operazioni CVS

- Sui file presenti nella propria *sandbox* si possono effettuare le seguenti operazioni:
  - *Checkout* (o *update*): preleva una copia aggiornata dal repository;
    - Se copia locale e copia del repository non coincidono viene segnalato un *conflict*
    - Dopo il checkout, la copia locale è in stato di *lock* e non può essere modificata
    - Di solito con checkout si intende il primo prelievo, con update i successivi
  - *Edit*: richiede il permesso di scrivere sul file locale
    - Se il file è già in stato di edit da parte di qualche altro utente, viene segnalato il rischio di modifiche concorrenti (nel caso di file binari o di politica di lock/modify/unlock viene impedito l'accesso)
  - *Commit*: rende pubbliche a tutti le proprie modifiche al file
    - Le modifiche vengono propagate al repository. Il repository incamera il file ricevuto come nuova versione; le versioni precedenti rimangono reperibili

# Operazioni CVS

- **Gestione conflitti**
  - Se due utenti vanno a modificare in concorrenza lo stesso file, e il primo di essi effettua il commit, verrà impedito al secondo di fare lo stesso
    - In questo caso si consiglia al secondo di fare un update: il sistema nota la differenza tra la versione sul repository e quella locale e popone alcune soluzioni semiautomatiche (*merge*) per la soluzione dei conflitti. Al termine, il secondo utente avrà una versione locale che tiene conto sia delle proprie modifiche che di quelle degli altri utenti. Di questa versione potrà essere fatto il commit, ottenendo quindi una versione successiva
- **Generazione branch**
  - Genera un ramo “alternativo” nella storia del file (se ne terrà conto nella diversa numerazione: ad esempio dopo 1.2 ci sarà 1.2.1 anzichè 1.3)
    - Sono disponibili funzionalità per vedere graficamente tutta la “storia” delle versioni dei files
- **Fusione tra versioni diverse**
- **Eliminazione copia locale**
- **Eliminazione originale (da operare direttamente sul repository)**

# GIT & GITHUB



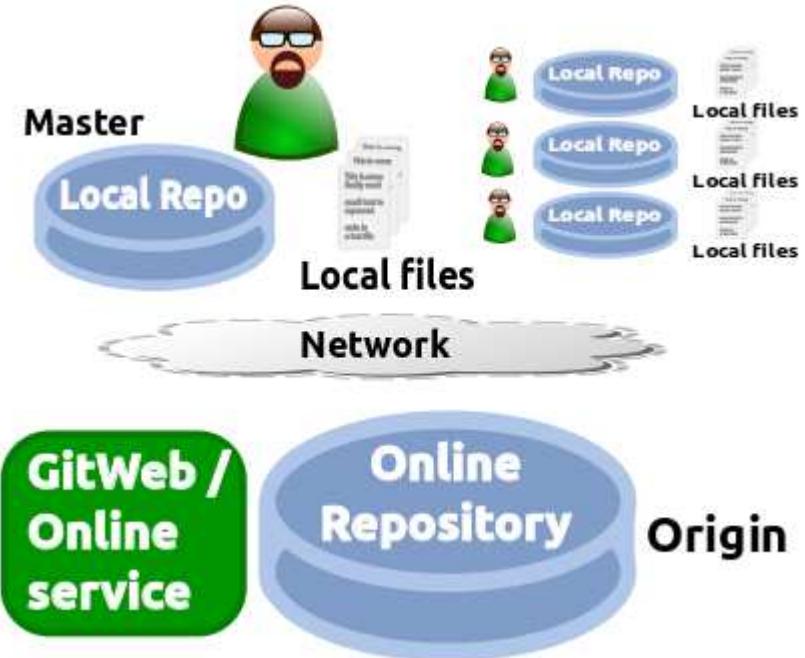
# Git

- Proposto da Linux Torvalds
  - <http://git-scm.com/>
- Si riferisce ad un paradigma più aperto, nel quale chiunque può partecipare ad un progetto:
  - Biforcando (Fork) un progetto esistente
  - Proponendo le sue aggiunte al progetto (patch)



# Git Architecture

- L'architettura di Git è distribuita
- A differenza di CVS ed altri, non esiste un'unica copia centralizzata del progetto
- Esiste, però, una copia di riferimento del progetto (**Master**) gestita solitamente dal fondatore del progetto
- Ogni utente può creare una copia locale dell'intero progetto
- L'utente locale può chiedere al fondatore di propagare nel master una propria modifica al progetto proponendo una **Patch**



# GIT HOSTING

- Il protocollo git è particolarmente utilizzato in progetti open source distribuiti sul Web
- E' particolarmente semplice sfruttare le potenzialità di git basandosi su piattaforme che mettono a disposizione funzionalità di hosting, in gran parte gratuite
  - **GitHub** è il più famoso fornitore di hosting, in continua espansione e acquisito nel 2018 da Microsoft
    - <https://github.com/>
  - GitLab, anch'esso in rapida espansione, che consente anche la possibilità di ospitare progetti su proprie risorse di hosting (la piattaforma GitLab è anch'essa open source)
    - <https://about.gitlab.com/>
  - Bitbucket, anch'esso utile anche per ospitare su proprie risorse oltre che su quelle a disposizione sul sito. E' stato acquisito da Atlassian
    - <https://bitbucket.org/>





# NAVIGAZIONE NEL CODICE

- I **Repository** nascono concettualmente come pozzi di storage di tutta la storia del codice e degli altri artefatti di un Progetto, durante tutto il suo ciclo di vita

The screenshot shows a GitHub repository page for 'reverse-unina / AndroidRipper'. The top navigation bar includes links for Pull requests, Issues, Marketplace, and Explore. The repository name 'reverse-unina / AndroidRipper' is displayed, along with a 'Watch' button (4), a 'Star' button (22), and a 'Code' button. Below the header, there are tabs for Code, Issues (5), Pull requests, Actions, Projects, Wiki, Security, and Insights. The 'Code' tab is selected. On the left, there's a sidebar with 'Code' and 'Issues' sections. The main content area shows a list of files in the 'master' branch:

| File                 | Action         | Last Commit |
|----------------------|----------------|-------------|
| AndroidRipper        | Update         | 3 years ago |
| AndroidRipperDriver  | Update         | 3 years ago |
| AndroidRipperService | Update         | 3 years ago |
| .gitignore           | ignore         | 4 years ago |
| LICENSE              | Initial commit | 4 years ago |
| README.md            | updated readme | 4 years ago |

Below the file list, there's a section for 'README.md' containing the following text:

**AndroidRipper**

AndroidRipper is a toolset for the automatic GUI testing of mobile Android Applications.

It is developed and maintained by the REVERSE (REsearch laboRatory of Software Engineering) Group of the University of Naples "Federico II".

On the right side of the page, there are sections for 'About', 'Releases', 'Packages', and 'Contributors'.

**About**: A toolset for the automatic GUI testing of mobile Android Applications.

**Releases**: 1 - [Android Ripper 2017.10](#) (Latest)

**Packages**: No packages published

**Contributors**: 2 - [reverse.unina REsEArch gRoup of So...](#)





# CREAZIONE DI UN PROGETTO

- I progetti possono essere pubblici o privati
  - Dato lo spirito open source della piattaforma, i progetti Pubblici sono completamente gratuiti e hanno meno limitazioni di quelli private
  - Alcune funzionalità più avanzate sono disponibili in quantità limitata o possono essere sbloccate solo a pagamento

 GitHub Free  
The basics for all developers

- ∞ Unlimited public/private repos
- ∞ Unlimited collaborators
- ✓ 2,000 Actions minutes/month
- ✓ 500MB of Packages storage
- ✓ Community support

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere?  
[Import a repository.](#)

Owner \* Repository name \*

 PorfirioTramontana /

Great repository names are short and memorable. Need inspiration? How about [symmetrical-octo-potato?](#)

Description (optional)

 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.](#)

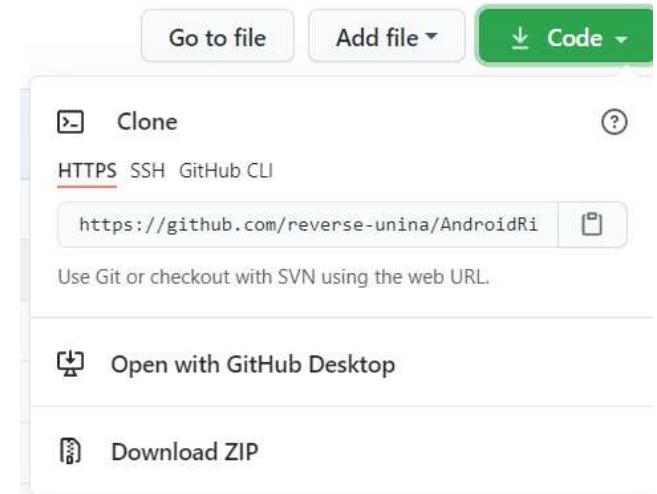
**Create repository**



# IMPORTAZIONE DI UN PROGETTO ESISTENTE



- Lo stato attuale di un progetto può essere scaricato istantaneamente con il pulsante **Clone (Code)**
- Con **Download Zip** il progetto viene scaricato in locale e può poi essere importato in un IDE
- Dall'indirizzo fornito, invece, può essere gestito in un client git (ad es. **Github Desktop**) oppure direttamente da un IDE (ad es. **Eclipse**)
  - Login e Password sono necessari nel caso si voglia successivamente continuare ad evolvere il progetto





# NAVIGAZIONE NEL CODICE

- Si possono individuare tutte le versioni del software, identificate come:
  - **Commit**
    - Ad ogni operazione di commit su github è associato un titolo: cliccando su questo titolo si possono vedere le modifiche introdotte da quell commit
  - **Tag**
    - Ad uno specifico commit può essere associato un Tag (etichetta): si individua così una release. Cliccando sulla release si può avere lo stato del Progetto intero al tempo di quella release
  - **Branch**
    - Il Progetto può avere delle ramificazioni (branch) identificate da un nome: ogni ramificazione procede da quell momento in maniera indipendente e selezionandola si può vedere un ramo di Progetto diverso





# FORK DI UN PROGETTO ESISTENTE

- Una fondamentale opportunità fornita da github è quella di creare una **fork** di un progetto
  - Basta premere l'apposite pulsante
- Github crea una copia esatta del progetto scelto e la aggiunge ai progetti di chi ha fatto la fork
  - Si può così fare proprio lo stato del progetto e iniziare a modificarlo dal proprio profilo come si preferisce
  - Viene mantenuto un conteggio delle fork di ogni progetto



|  |                   |               |
|--|-------------------|---------------|
|  | reverse-unina /   | AndroidRipper |
|  | ahmetsen93 /      | AndroidRipper |
|  | caizhenxing /     | AndroidRipper |
|  | cetinturan /      | AndroidRipper |
|  | cxz13250 /        | AndroidRipper |
|  | ebarsallo /       | AndroidRipper |
|  | friendlyJLee /    | AndroidRipper |
|  | gamzesari /       | AndroidRipper |
|  | imdea-software /  | AndroidRipper |
|  | Ivouch /          | AndroidRipper |
|  | kilincceker /     | AndroidRipper |
|  | LZH99 /           | AndroidRipper |
|  | nicola-amatucci / | AndroidRipper |
|  | pawanlathwal /    | AndroidRipper |
|  | sangamk /         | AndroidRipper |
|  | vicctor /         | AndroidRipper |
|  | behzadnaz /       | AndroidRipper |
|  | zmqgeek /         | AndroidRipper |





# ALTRÉ INTERAZIONI CON I PROGETTI

- **Watch**

- Selezionando questo pulsante notifiche sulle attività di questo progetto saranno incluse nella nostra Dashboard

- **Star**

- Selezionando questo pulsante aggiungiamo questo progetto ai nostri preferiti, intendendo quindi dare un feedback positivo agli sviluppatori





# PULL REQUEST

- La Pull Request è un elemento fondamentale della visione open source di Github
- Un qualsiasi sviluppatore, anche esterno al progetto, può realizzare un potenziale miglioramento dell'applicazione e proporre al proprietario di inglobarla nel progetto tramite una Pull Request
  - In questo caso la Pull Request parte da un fork del progetto sul quale sono stati fatti dei commit successivi
  - Una Pull Request può anche essere fatta a partire da un branch del progetto stesso





# ISSUE

- Si può contribuire anche da esterni ad un progetto altrui facendo notare difetti, problemi o semplicemente proponendo evoluzioni, tramite le **Issue**

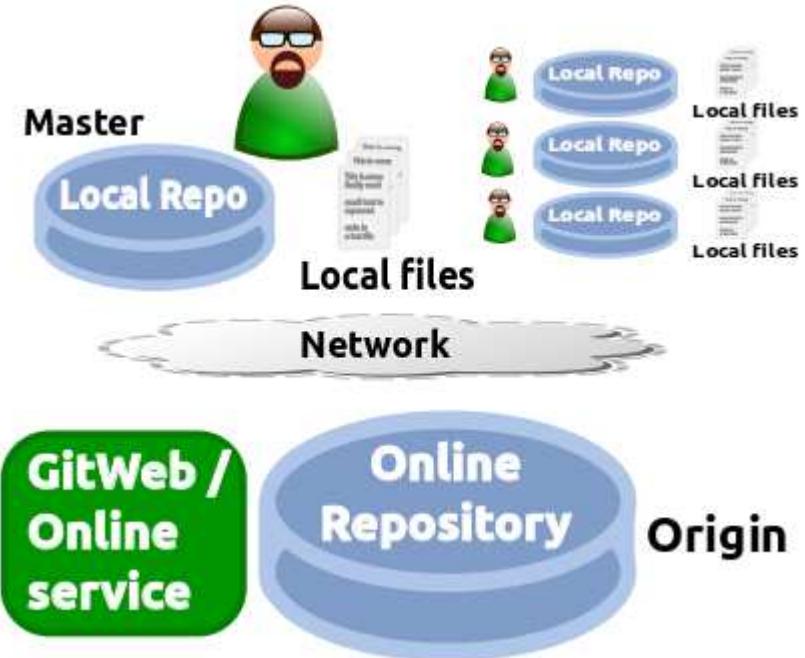
The screenshot shows the GitHub Issues page for the `junit-team/junit5` repository. The URL in the address bar is `github.com/junit-team/junit5/issues`. The page has a dark theme. At the top, there are navigation links for `Code`, `Issues 211`, `Pull requests 22`, `Discussions`, `Actions`, `Wiki`, `Security`, and `Insights`. Below the navigation, there is a search bar with the query `is:issue is:open`, a `New issue` button, and filters for `Labels 45` and `Milestones 6`. A modal window titled `Want to contribute to junit-team/junit5?` is open, with the message: `If you have a bug or an idea, read the contributing guidelines before opening an issue.` The main content area displays a list of 211 open issues. The first few issues are:

- `Enable stalebot status:team discussion type:task` - #2602 opened 2 days ago by `marcphilipp` - 5.8 M2
- `Beginning with 1.7.1 standalone console does not find tests on Java 8 component:Platform type:bug` - #2600 opened 3 days ago by `ndmitry` - 5.7.2
- `Support AutoCloseable for arguments provided to a @ParameterizedTest component:Jupiter theme:parameterized tests type:enhancement` - #2597 opened 10 days ago by `sbrannen` - 5.8 M2
- `Discover tests in suites packaged into jars, e.g. TCKs` - #2594 opened 14 days ago by `t1` - 0 of 2
- `Add concrete steps to "Migrating from JUnit 4" docs section.`

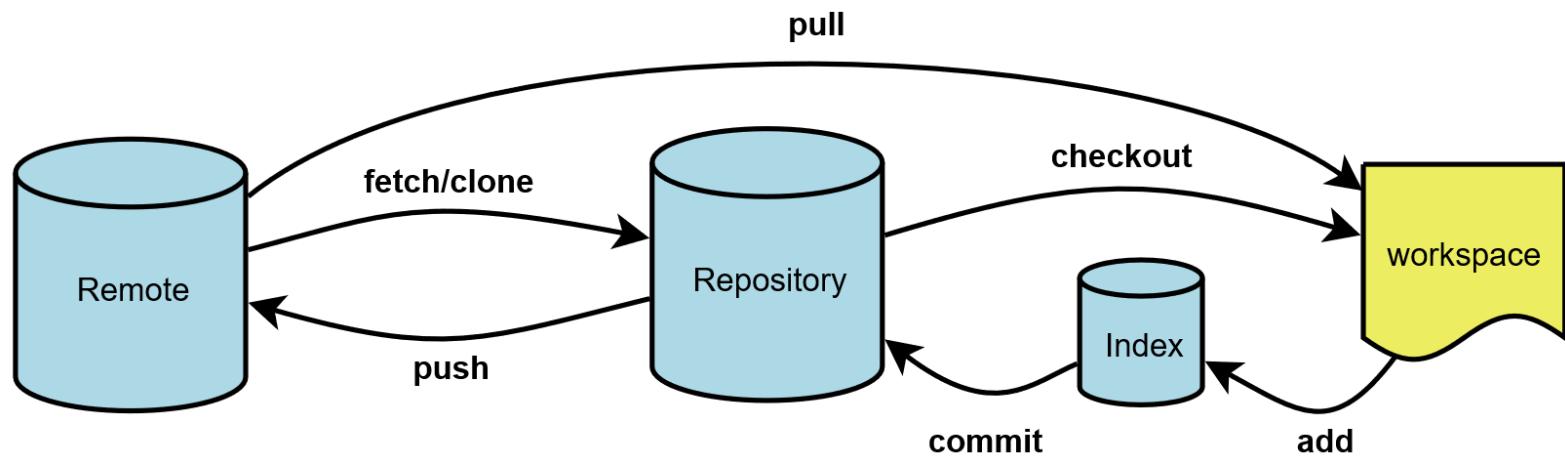


# Git Architecture

- L'architettura di Git è distribuita
- A differenza di CVS ed altri, non esiste un'unica copia centralizzata del progetto
- Esiste, però, una copia di riferimento del progetto (**Master**) gestita solitamente dal fondatore del progetto
- Ogni utente può creare una copia locale dell'intero progetto
- L'utente locale può chiedere al fondatore di propagare nel master una propria modifica al progetto proponendo una **Patch**



# RELAZIONE TRA OPERAZIONI LOCALI E REMOTE



# Git

- All'operazione di **checkout** (copia dell'immagine attuale del progetto in locale) corrisponde l'operazione di **Clone** (copia di tutto il progetto in locale)
- Si può pubblicare la propria copia locale, che diventa un nuovo progetto Git di proprietà dell'utente stesso. Tale nuovo progetto è da considerarsi concettualmente come un **Branch** del progetto originale
- L'operazione di **Commit** diventa un'operazione locale, quindi quasi immediata e senza alcuna necessità di review
- L'operazione di **Patching**, invece, corrisponde ad una richiesta di Commit indirizzata verso il progetto Master
  - Se il proprietario del master approva, viene creata una *patch*, cioè un insieme di operazioni per trasformare la copia master in una copia che integri anche le modifiche dell'utente



# COMMIT & PUSH

- Commit è un'operazione che aggiorna *localmente* il nostro Progetto supportato da git (in generale va a modificare file .git)
- Push è un'operazione che invia al repository remote (ad esempio ospitato da Github) una richiesta di aggiornamento di tutti I file modificati in locale
  - Solo un proprietario (owner) del progetto può fare richiesta di push: tutti gli altri al Massimo possono fare una *pull request*
  - La richiesta di Push viene analizzata in cerca di possibili conflitti tra la versione precedente e quella successiva alla modifica
    - I conflitti sono risolti in generale come in cvs, con la possibilità di scartare modifiche, fare merge manuali o automatici, creando branch



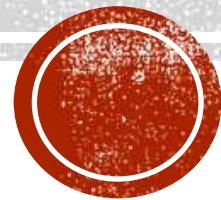
# Differenze con CVS

## Il sistema Git:

- È più sicuro: non esiste un'unica copia centralizzata del progetto
- E' più scalabile: il numero di partecipanti al progetto può aumentare liberamente
  - Invece in CVS e SVN, ad esempio, aumentando il numero di partecipanti aumenta il tempo in cui le risorse sono bloccate e/o il numero di conflitti sui commit
    - Il sistema Linux è stato sviluppato storicamente sotto Git, con il master sotto il diretto controllo di Linus Torvalds
- Necessita di opportune licenze di utilizzo
  - Spesso è legato a software open source distribuito con licenze Creative Commons
  - Non è generalmente utilizzato all'interno di aziende che realizzano software closed source



# DEBUGGING



# DEBUGGING

- Attività di ricerca e correzione dei difetti che sono causa di malfunzionamenti.
- È l'attività consequenziale alla scoperta di un malfunzionamento. Comprende due fasi:
  - **Ricerca del difetto**
  - **Correzione del difetto**
- Il debugging è ben lungi dall'essere stato formalizzato
  - Metodologie e tecniche di debugging rappresentano soprattutto un elemento dell'esperienza del programmatore/tester



# RICERCA E LOCALIZZAZIONE DEI DIFETTI

- Ridurre la distanza tra difetto e malfunzionamento
  - Mantenendo un'immagine dello stato del processo in esecuzione in corrispondenza dell'esecuzione di specifiche istruzioni
    - Watch point e variabili di watch (Sonde)
      - Un watch, in generale, è una semplice istruzione che inoltra il valore di una variabile verso un canale di output
        - L'inserimento di un watch (sonda) è un'operazione invasiva nel codice: anche nel watch potrebbe annidarsi un difetto
        - In particolare l'inserimento di sonde potrebbe modificare sensibilmente il comportamento di un software concorrente
      - Asserzioni, espressioni booleane dipendenti da uno o più valori di variabili legate allo stato dell'esecuzione
      - Possiamo realizzare una asserzione con una interruzione programmata



# ESEMPIO DI SONDA

- Con le due System.out.println verifichiamo il capitale prima e dopo l'acquisto
- Avremmo potuto scrivere su di un file anzichè sullo schermo per poter avere alla fine un *log* di informazioni relative all'esecuzione

```
public boolean acquista(String nomeSocieta, int quantita) {  
    boolean ok=true;  
    for (Societa s:l.getSocieta())  
        if (s.getNome().contentEquals(nomeSocieta)) {  
            if (ok) {  
                System.out.println("Capitale attuale : "+ g.getCapitale());  
                g.acquista(quantita, LocalDate.now(), s.getPrezzoAzione(), s);  
                System.out.println("Capitale dopo l'acquisto : "+ g.getCapitale());  
                return true;  
            }  
        }  
    return false;  
}
```



# ESEMPIO DI ASSEGNAZIONE

- Con questo controllo vigiliamo che la liquidità non diventi negative e in tal caso blocchiamo il programma in modo da sapere da che punto in poi le cose vanno male

```
public boolean acquista(String nomeSocieta, int quantita) {  
    boolean ok=true;  
    for (Societa s:l.getSocieta())  
        if (s.getNome().contentEquals(nomeSocieta)) {  
            if (ok) {  
                g.acquista(quantita, LocalDate.now(), s.getPrezzoAzione(), s);  
                if(g.getLiquidita()<0)  
                    throw new RuntimeException("Errore: non avevi soldi per questo acquisto");  
            }  
            return true;  
        }  
    return false;  
}
```



# AUTOMATIZZAZIONE DEL DEBUGGING

- Il debugging è un'attività estremamente intuitiva, che però deve essere operata nell'ambito dell'ambiente di sviluppo e di esecuzione del codice
- Strumenti a supporto del debugging sono quindi convenientemente integrati nelle piattaforme di sviluppo (IDE), in modo da poter accedere ai dati del programma, anche durante la sua esecuzione, senza essere invasivi rispetto al codice
  - In assenza di ambienti di sviluppo, l'inserimento di codice di debugging invasivo rimane l'unica alternativa



# FUNZIONALITÀ DI DEBUGGING

- Inserimento break point
  - Tramite Eclipse è possibile accedere a delle “Breakpoint properties”
    - Breakpoint condizionali:
      - Breakpoint che si attivano solo quando si passa in una certa riga e si verifica una certa condizione
    - Breakpoint dipendenti dallo Hit Count:
      - Breakpoint che si attivano solo dopo un certo numero di passaggi su quella riga di codice
- Esecuzione passo passo del codice
  - Entrando o meno all'interno dei metodi chiamati
  - Uscendo da un metodo verso il chiamante



# ESECUZIONE STEP BY STEP

- Quando il programma si ferma su di un breakpoint è possibile:
  - Farlo continuare (pulsante play)
  - Proseguire all'interno della funzione chiamata nella riga attuale
  - Proseguire con la riga successiva
  - Proseguire fino alla fine dell'esecuzione della funzione corrente
- Durante l'esecuzione step by step possiamo vedere facilmente i valori di tutte le variabili e chiedere di calcolare eventuali espressioni

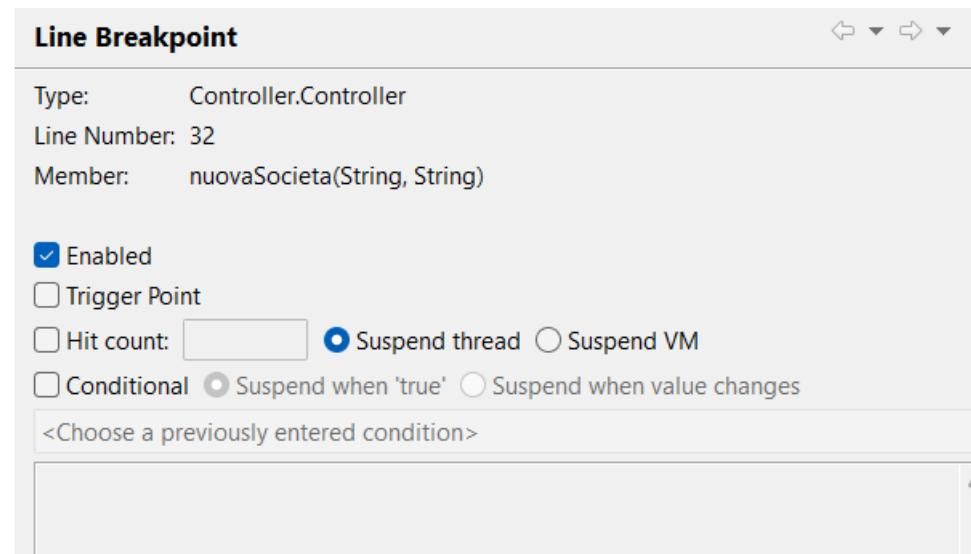


| (x)= Variables           |      | Breakpoints | Expressions |                    |
|--------------------------|------|-------------|-------------|--------------------|
| Name                     | Type |             |             |                    |
| ↳ no method return value |      |             |             |                    |
| > ⚪ this                 |      |             |             | Controller (id=44) |
| > ⚪ nomeSocieta          |      |             |             | "uno" (id=73)      |
| > ⚪ prezzoAzione         |      |             |             | "10" (id=79)       |



# BREAKPOINT CONDIZIONALI

- Per abbreviare il debugging dovremmo poter sistemare il breakpoint il più vicino possibile alla causa del difetto
- Nel dubbio, c'è il rischio di dover eseguire troppe righe prima di arrivare a quella incriminata
  - Ad esempio se c'è un ciclo ripetuto 1000 volte, come possiamo fare a fermarci dopo 789 esecuzioni senza eseguire altrettante volte il ciclo?
- Breakpoint condizionali
  - Si attivano con il tasto destro su di un breakpoint e scegliendo Breakpoint properties

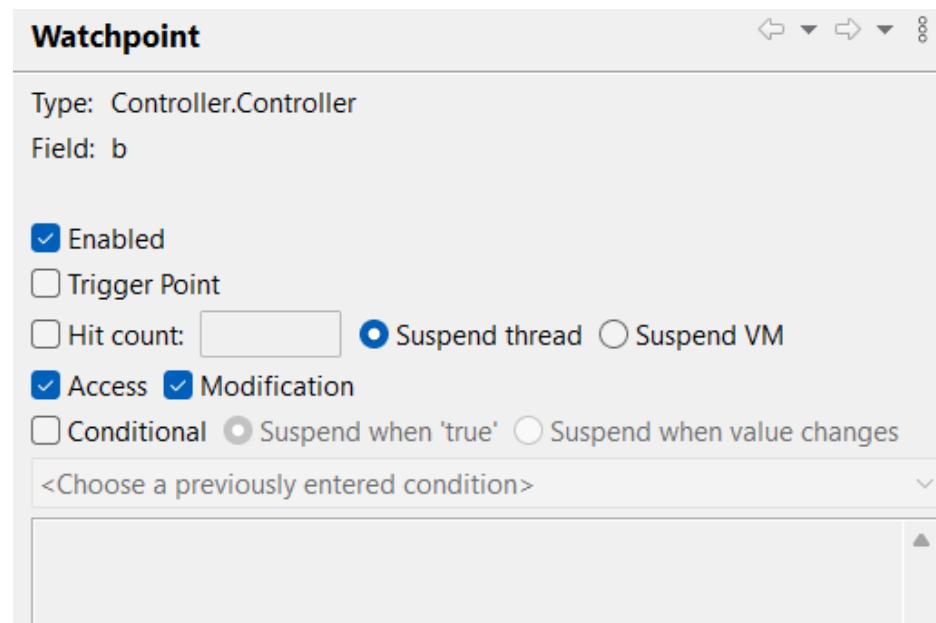


- Hit Count: il breakpoint ferma il programma solo quando si passa per la i-esima volta su di esso
- Conditional: ogni volta che passiamo sul breakpoint si valuta la condition: solo se è vero il programma si sospende



# WATCHPOINT

- Un watchpoint è un breakpoint collegato non ad una riga ma ad un attributo di una classe
- Si inserisce come un breakpoint, puntando sulla riga di dichiarazione dell'attributo
- Fa sospendere l'esecuzione ogni volta che l'attributo è letto o scritto
- Tramite le Properties è possibile personalizzare quando effettuare la sospensione



# ULTERIORI TIPI DI BREAKPOINT

- Similmente a breakpoint e watchpoint:
  - Breakpoint per le eccezioni;
  - Breakpoint per il caricamento di classi;
  - Breakpoint per un metodo;
- Ulteriori approfondimenti ed esempi:
  - <http://www.vogella.com/articles/EclipseDebugging/article.html>

