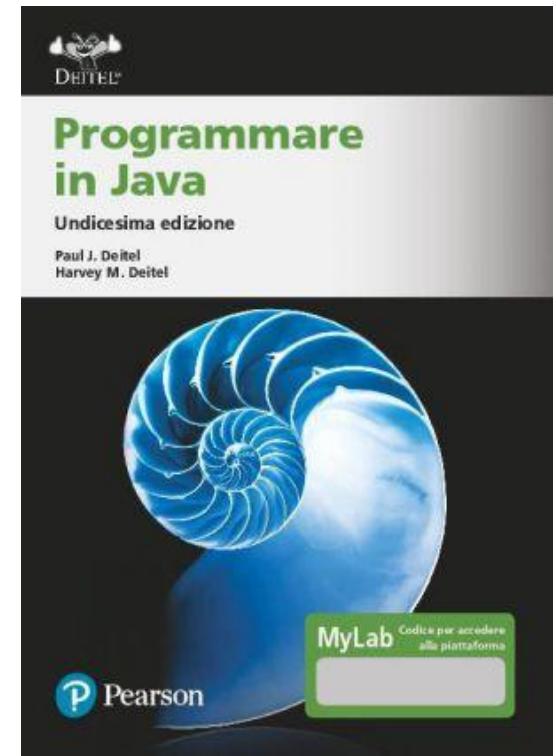


LINGUAGGIO JAVA



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



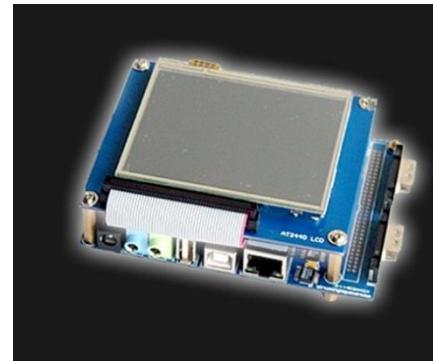
- Mobile



- 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 20
 - <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>



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).



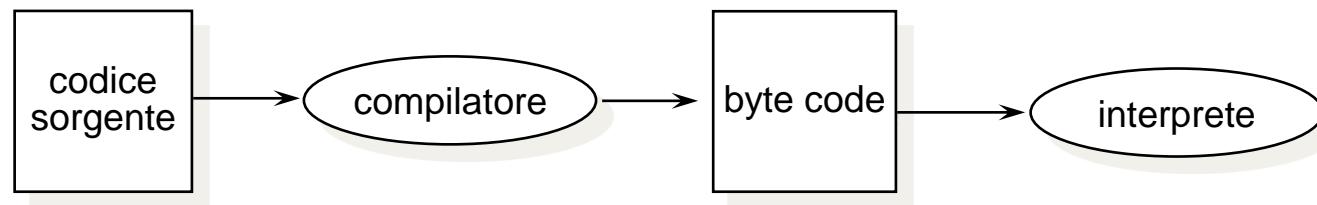
JAVA: SEMPLICE E 0.0.

- 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



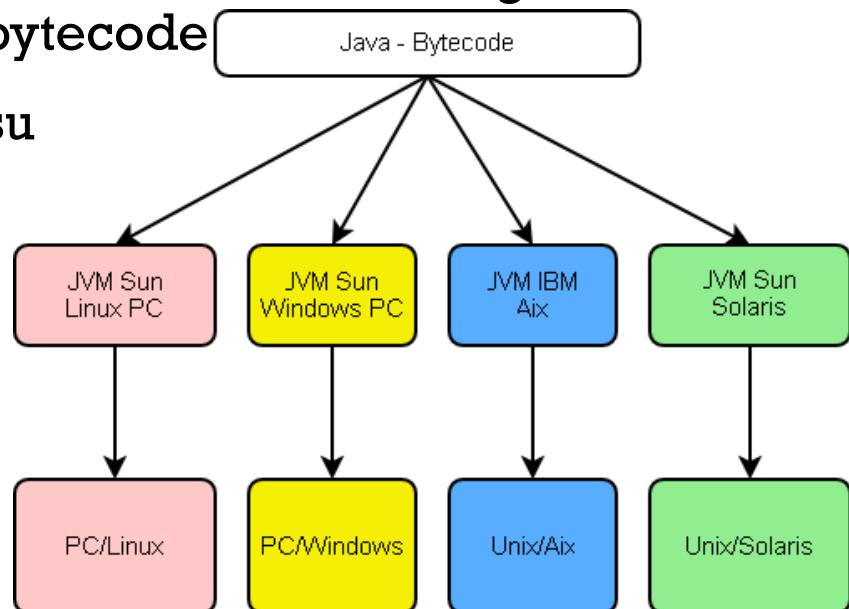
JAVA È COMPILATO E INTERPRETATO

- 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**



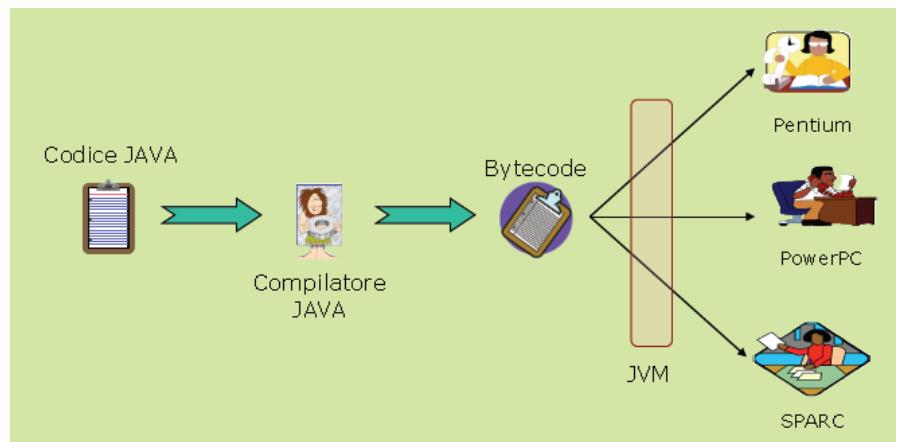
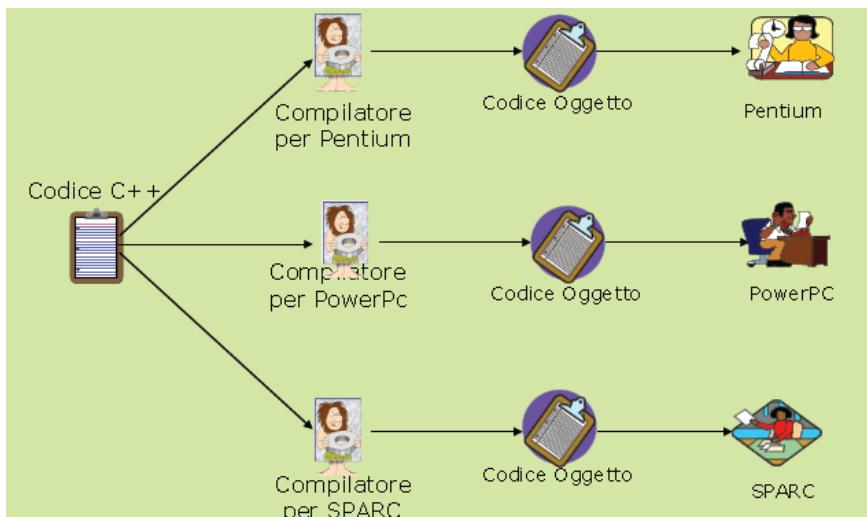
LA JAVA VIRTUAL MACHINE (JVM)

- 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 20, scaricabile a quest'indirizzo:
 - <https://www.oracle.com/java/technologies/downloads/>
- Al termine dell'installazione, per verificare il completamento apriamo *cmd* e scriviamo `java -version`

```
C:\Users\utente>java -version
java version "20.0.2" 2023-07-18
Java(TM) SE Runtime Environment (build 20.0.2+9-78)
Java HotSpot(TM) 64-Bit Server VM (build 20.0.2+9-78, 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



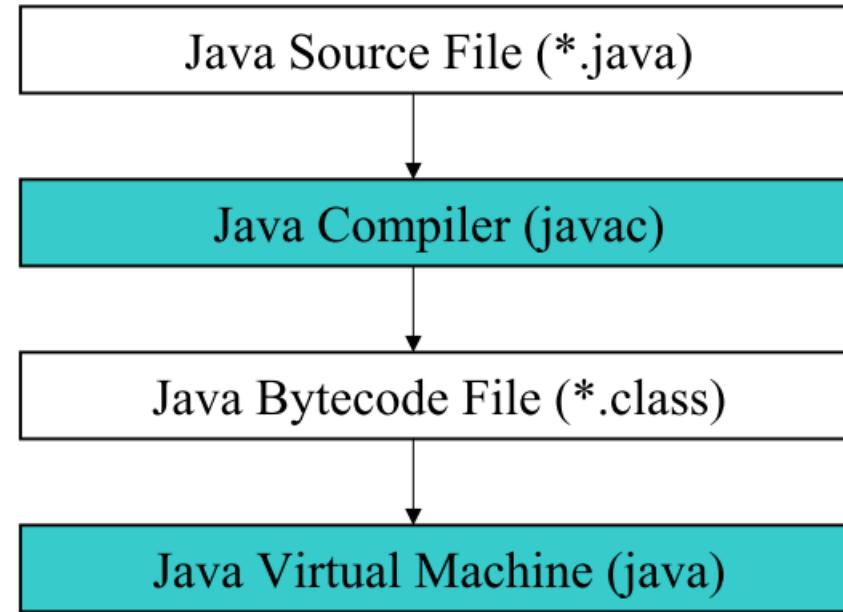
STRUTTURA DI UN SEMPLICE PROGRAMMA JAVA: HELLO, WORLD

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

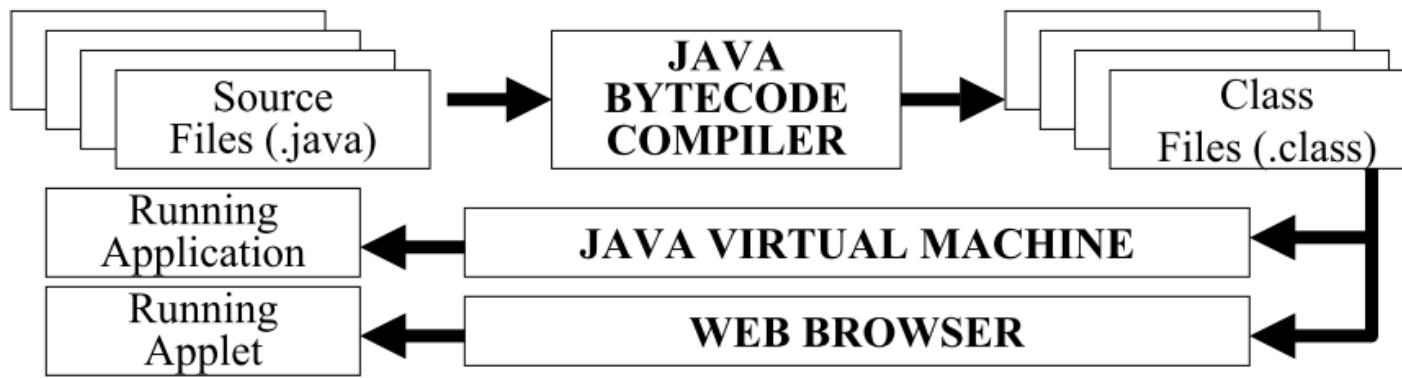
- Tutto deve trovarsi in una classe.
- Il codice sorgente è salvato in un file Hello.java.
- La classe Hello sembra non ereditare da nessun'altra (in realtà eredita come default da Object)



L'AMBIENTE JAVA



Organizzazione dei Programmi Java



COMPILAZIONE ED ESECUZIONE DI HELLO, WORLD

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

Compilazione

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

```
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.
 - **IntelliJ Idea**
 - Potenzialmente, anche Android Studio, nato da IntelliJ Idea, 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.

INTELLIJ IDEA

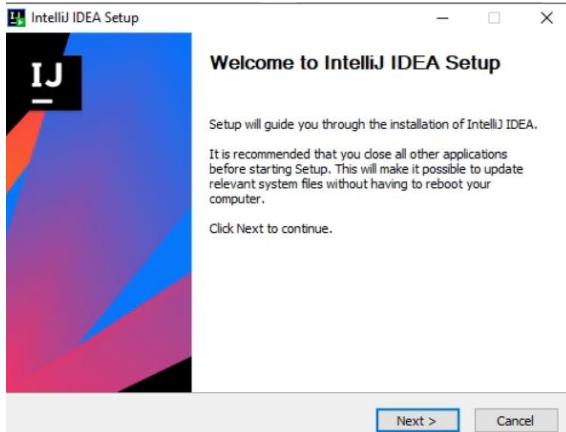


- IntelliJ IDEA è un IDE molto diffuso per lo sviluppo di applicazioni Java, che supporta molti altri linguaggi di programmazione
- IntelliJ IDEA è sviluppato da Jet Brains
 - <https://www.jetbrains.com/idea/>



INSTALLAZIONE DI INTELLIJ IDEA

- Verrà utilizzata la versione denominata Ultimate sfruttando la licenza studente
- Per scaricare e installare lo strumento:
 - <https://www.jetbrains.com/idea/download>
 - (si verrà rediretti automaticamente ad una pagina corrispondente al Sistema operativo installato)
 - Sono necessari circa 3GB di spazio su disco



Download IntelliJ IDEA

Windows

macOS

Linux

Ultimate

For web and enterprise development

Download

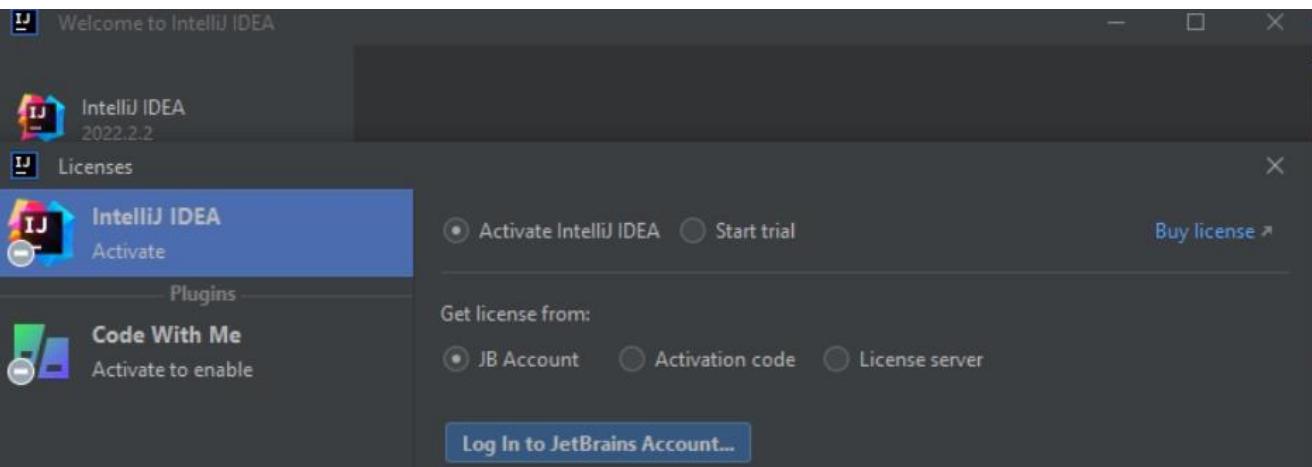
.exe

Free 30-day trial available



INSTALLAZIONE DI INTELLIJ IDEA

- Per richiedere una licenza gratuita:
 - <https://www.jetbrains.com/community/education/#students>
- Per attivare la licenza in JetBrains scegliere come da figura ed eseguire il Login con il JetBrains Account (creato seguendo le istruzioni ricevute via mail nella richiesta della licenza), poi Activate and Continue



JetBrains Products for Learning

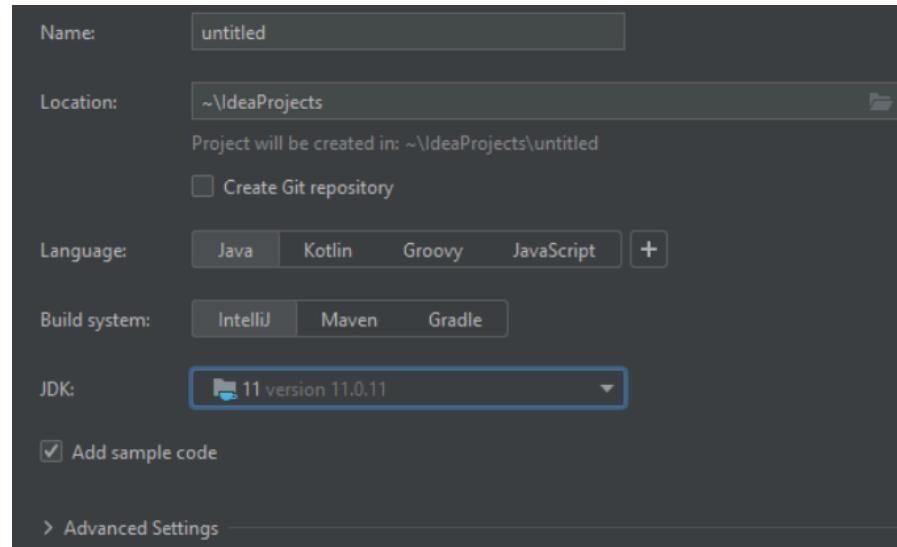
Before you apply, please read the [Educational Subscription Terms and FAQ](#).

This is a screenshot of a web-based application titled "JetBrains Products for Learning". It has tabs at the top: "University email address" (selected), "ISIC/ITIC membership", and "Official do...". Below the tabs, there's a "Status:" section with two radio buttons: "I'm a student" (selected) and "I'm a teacher".

This is a screenshot of the IntelliJ IDEA activation window. At the top, there are two radio buttons: "Activate IntelliJ IDEA" (selected) and "Start trial". Below that, it says "Get license from:" with three options: "JB Account" (selected), "Activation code", and "License server". A message at the bottom states "Licensed to Porfirio Tramontana, For educational use only Subscription is active until 11/09/2023 License ID: T1Z5MXCWDH". At the bottom right, there are "Activate" and "Refresh license list" buttons.

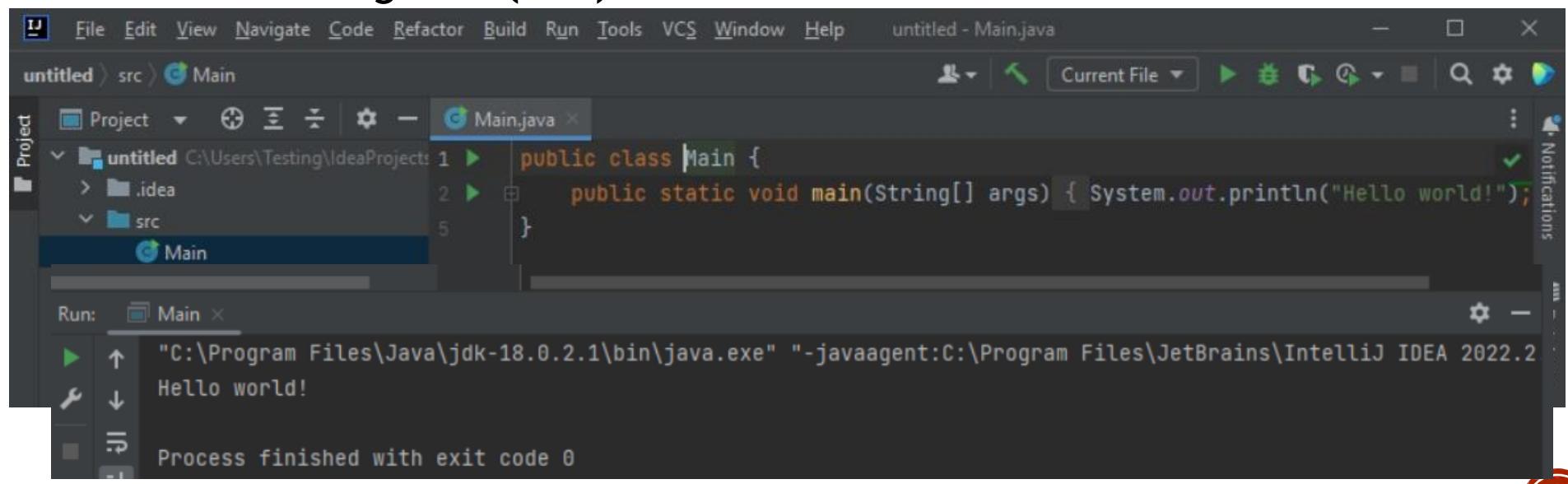
CREAZIONE DEL PRIMO PROGETTO CON INTELLIJ IDEA

- Selezionare New Project
 - IntelliJ IDEA cercherà per le jdk precedentemente installate
- Selezionare, per quest'esempio nome del Progetto e posizione su disco
 - Lasciamo ai valori di default le altre opzioni, ad eccezione della jdk, per la quale proviamo la più recente
- Per utilizzzi future, scegliamo Maven come Build option



CREAZIONE DEL PROGETTO

- Dopo un pò di tempo (più lungo per il primo Progetto) siamo pronti a scrivere codice
- Nella cartella src c'è già un Main.java con Hello, World
- Possiamo eseguirlo (Run) e vedere il risultato



The screenshot shows the IntelliJ IDEA interface. The top menu bar includes File, Edit, View, Navigate, Code, Refactor, Build, Run, Tools, VCS, Window, and Help. The current file is "untitled - Main.java". The Project tool window on the left shows a structure with "untitled" at the root, containing ".idea" and "src". The "src" folder contains a "Main" package, which has a "Main" class. The code editor displays the following Java code:

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

The Run tool window at the bottom shows the command used to run the application: "C:\Program Files\Java\jdk-18.0.2.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2022.2\lib\idea_rt.jar". The output pane shows the result: "Hello world!". Below that, it says "Process finished with exit code 0".



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--};  
}
```



CLASSI E OGGETTI

- Una **classe** è analoga ad una definizione di un tipo struct
- Gli **attributi** di una classe sono analoghi ai campi di una struct
- Le funzioni dichiarate all'interno di una classe (dette **metodi**) sono invece un concetto che non ha analogie con il caso delle struct
- Un **oggetto** è analogo ad una variabile del tipo definito dalla classe
 - Gli attributi assumono quindi valori indipendenti se riferiti ad oggetti indipendenti della stessa classe
 - La dichiarazione di una variabile di tipo classe (oggetto) è quindi analoga alla dichiarazione di una variabile di tipo struct
 - Ma vedremo molte differenze sulla sua definizione e allocazione in memoria



ATTRIBUTI: ANALOGIE CON C

```
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--;};  
}
```

- Gli attributi hanno un tipo
- Esistono tipi scalari (int, float, double, ...)
- Gli attributi possono assumere valori costanti (**final**)
- Il valore ad un attributo (scalare) può essere fornito tramite una assegnazione



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.



METODI: ANALOGIE CON C

```
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 somigliano alle funzioni riguardo alcuni aspetti fondamentali:
 - Essi hanno dei parametri indicati tra parentesi unitamente al loro tipo
 - Essi restituiscono un parametro tramite la parola chiave **return**
 - Se non ci sono parametri di ritorno si utilizza il tipo **void**
 - Essi hanno una dichiarazione (prototipo o firma) e una definizione (tra parentesi graffe, in generale)



METODI

```
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:
 - sono applicabili ad un oggetto, che rappresenta il primo parametro della funzione
 - Esiste l'eccezione dei metodi **static** che verrà trattata più avanti
 - 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



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
 - Verrà trattato meglio più avanti



ALLOCAZIONE

- 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.
- Uno dei motivi fondamentali per i quali non esiste il concetto di puntatore è che il programmatore non deve mai avere la possibilità di indirizzare direttamente la memoria, ma deve sempre chiedere alla Java Virtual Machine di fare da tramite
- Le variabili scalari possono essere allocate nello stack
- Gli oggetti sono allocati nello heap e sono raggiungibili non tramite un puntatore ma tramite un **riferimento**
 - Un riferimento funziona come un puntatore con l'eccezione che il suo valore non rappresenta un indirizzo di memoria ma solo un *riferimento* tramite il quale è possibile raggiungere il valore in memoria



ALLOCAZIONE

```
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--};  
}
```

- Uno scalare può essere allocato tramite dichiarazione e definito tramite definizione:
 - int value;
 - value =0;
 - int max=10;
- Un oggetto invece può essere dichiarato solo tramite la dichiarazione di un riferimento ad esso :
 - Contatore c2;
- La definizione di un valore (per il riferimento) si ottiene invece con l'operatore **new**, che restituisce il riferimento all'istanza di un oggetto, allocato dinamicamente:
 - Contatore c= **new** Contatore();



ALLOCAZIONE

- Un assegnazione come `c2=c` riguarda una assegnazione tra **riferimenti**
- **Essa** corrisponde 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
- Con `c2=c` avremo quindi due riferimenti allo stesso oggetto: se modifichiamo i valori puntati da `c` avremo modificato anche quelli leggibili da `c2`
- Un riferimento può essere inizializzato a `null` per non farlo puntare a nessun oggetto: `c2=null;`
 - I riferimenti dichiarati dovrebbero essere sempre inizializzati a `null`, in mancanza di altre inizializzazioni



DEALLOCAZIONE

- Le variabili scalari vengono distrutte (deallocate) automaticamente al termine del loro scope, eliminandole dallo stack
 - Valgono tutti i principi già visti per il linguaggio C: tenere lo scope delle variabili il più breve possibile:
 - Riduce l'occupazione della memoria RAM
 - Riduce lo sforzo di debugging in caso di errori da correggere
- I riferimenti si comportano come le variabili scalari
 - Così come i puntatori in C
- Gli oggetti invece sono nell'heap
 - In linguaggi come C++ andavano distrutti da programma con l'istruzione *delete*
 - In Java invece sono distrutti automaticamente da un componente della JVM che si chiama Garbage Collector



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.
 - E soprattutto problemi di eccessiva occupazione della memoria



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 .



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;};
```



COSTRUTTORE

- Quando si definisce un oggetto (tramite **new**) viene sempre chiamato un costruttore
- Il costruttore di solito ha lo scopo minimo di dare un valore iniziale agli attribute, ma sono accettabili anche costruttori che non hanno alcun codice { }
- Il costruttore non ha un tipo di ritorno esplicito, poichè esso è predefinito: il costruttore ritorna un riferimento all'oggetto che ha appena allocato in memoria della classe alla quale il costruttore si riferisce



COSTRUTTORE: ESEMPIO

```
Contatore c = new Contatore (2);
```

- Crea un nuovo oggetto della classe contatore
- assegna a c il valore del riferimento all'oggetto contatore appena creato
- La chiamata del costruttore con parametro intero 2 comporta l'esecuzione del costruttore :

```
public Contatore(int c) {value=c;};
```
- È la conseguente assegnazione del valore 2 all'attributo value



ACCESSO AD UN OGGETTO E AI SUOI ELEMENTI

- Sia per accedere ad un attributo che ad un metodo si utilizza l'operatore punto
 - che già era stato introdotto in C per accedere ai campi delle struct
- `c.value=10;`
- `c.incrementa();`
- Da notare che l'operatore . si applica al riferimento
 - In C invece l'operatore . si applicava alle variabili di tipo struct mentre l'operatore -> si applicava ai puntatori



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`.





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
boolean	8	true or false	
char	16	' \u0000' to ' \uFFFF' (0 to 65535)	(ISO Unicode character set)
byte	8	-128 to +127 (-2⁷ to 2⁷ - 1)	
short	16	-32,768 to +32,767 (-2¹⁵ to 2¹⁵ - 1)	
int	32	-2,147,483,648 to +2,147,483,647 (-2³¹ to 2³¹ - 1)	
long	64	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 (-2⁶³ to 2⁶³ - 1)	
float	32	Negative range: -3.4028234663852886E+38 to -1.40129846432481707e-45 Positive range: 1.40129846432481707e-45 to 3.4028234663852886E+38	(IEEE 754 floating point)
double	64	Negative range: -1.7976931348623157E+308 to -4.94065645841246544e-324 Positive range: 4.94065645841246544e-324 to 1.7976931348623157E+308	(IEEE 754 floating point)



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 scalar (
 - la sola assegnazione anche ai riferimenti
 - 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>
Arithmetic assignment operators.			



OPERATORI DI INCREMENTO E DECREMENTO

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



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

equals è un metodo che può essere riprogrammato e serve appunto a dire se due oggetti possono essere considerati uguali

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



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
 - In altri termini, l'unico passaggio consentito è quello *per valore* e possono essere passati *scalari* oppure *riferimenti*
 - Nota: un utilizzo più avanzato di Java consente di passare ad un metodo *funzioni* o addirittura *classi*
 - Nel senso che ciò che viene passato è sempre e comunque un riferimento





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 ≥ 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



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
Collection	The root interface in the collections hierarchy from which interfaces Set , Queue and List are derived.
Set	A collection that does <i>not</i> contain duplicates.
List	An ordered collection that <i>can</i> contain duplicate elements.
Map	A collection that associates keys to values and <i>cannot</i> contain duplicate keys. Map does not derive from Collection.
Queue	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.

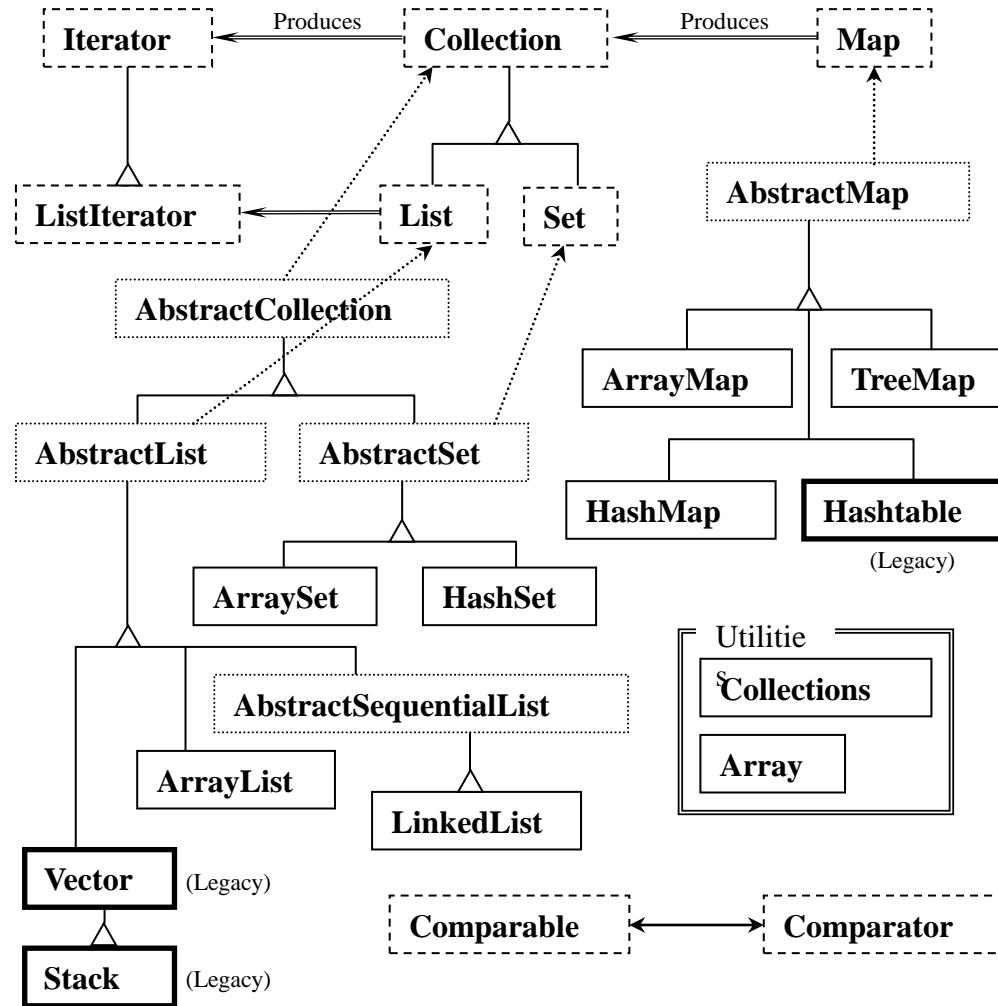


CENNO AGLI ALTRI CONTAINER

- **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.



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
 - 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 attributi/metodi **public** della classe padre sono visibili alla classe figlia
 - E a tutti gli eventuali altri discendenti
- Gli attributi/metodi **private** della classe padre *non* sono visibili alla classe figlia
- Gli attributi/metodi **protected** della classe padre sono visibili *solo* alla classe figlia
 - E a tutti gli eventuali altri discendenti



VISIBILITÀ'

- 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 attributi/metodi aggiuntivi



ESEMPI DI EREDITARIETÀ

- Consideriamo una classe **Utente** e le classi **Studente** e **Docente**
- **Utente** rappresenta una generalizzazione (padre) di **Studente** e **Docente** (*figli*)
- L'operazione di *signIn* è definita sia per docenti che per studenti
 - Verifica la correttezza di *login* e *password*
- Per **Docente** è inoltre definita l'operazione *crea corso* (il corso ha un nome)
- Per **Studente** è definita l'operazione di *iscrizione al corso* (dato il nome del corso)
- Lo **Studente** ha anche un numero di matricola
- **Docente** e **Studente** non hanno un legame diretto (hanno la stessa classe padre: sono in qualche modo fratelli – *sibling*)



ESEMPI DI EREDITARIETÀ

- Che visibilità devono avere *login* e *password* per **Utente**?
- Quale classe deve contenere il codice per scrivere i valori di *login* e *password*?
- Quale classe deve contenere il codice di *signIn*?
- Che relazione c'è tra il costruttore di **Docente** e il costruttore di **Utente**?
- Che relazione c'è tra il costruttore di **Studente** e il costruttore di **Utente**?



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 Studente(String l, String p, int m) {  
    super(l, p);  
    matricola=m;  
}
```

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



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 parametro
 - Si tratta quindi di ridefinizioni/sovraposizioni/override di metodi
 - L'annotazione (opzionale) **@Override** nel codice evidenzia questa occorrenza e aiuta a prevenire errori



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



ESEMPIO DI OVERRIDING

- Ridefiniamo l'eguaglianza (*equals*) in riferimento agli **Studenti**
 - C'è eguaglianza tra due oggetti della classe **Studente** quando hanno la stessa matricola

```
public boolean equals(Studente s) {  
    return matricola.equals(s.matricola);  
}
```



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 : Studente s = new Studente();
 - d sia un oggetto della classe Docente: Docente d = new 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
 - 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(); → 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();



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 tipi:

```
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, l'IDE 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();  
}
```



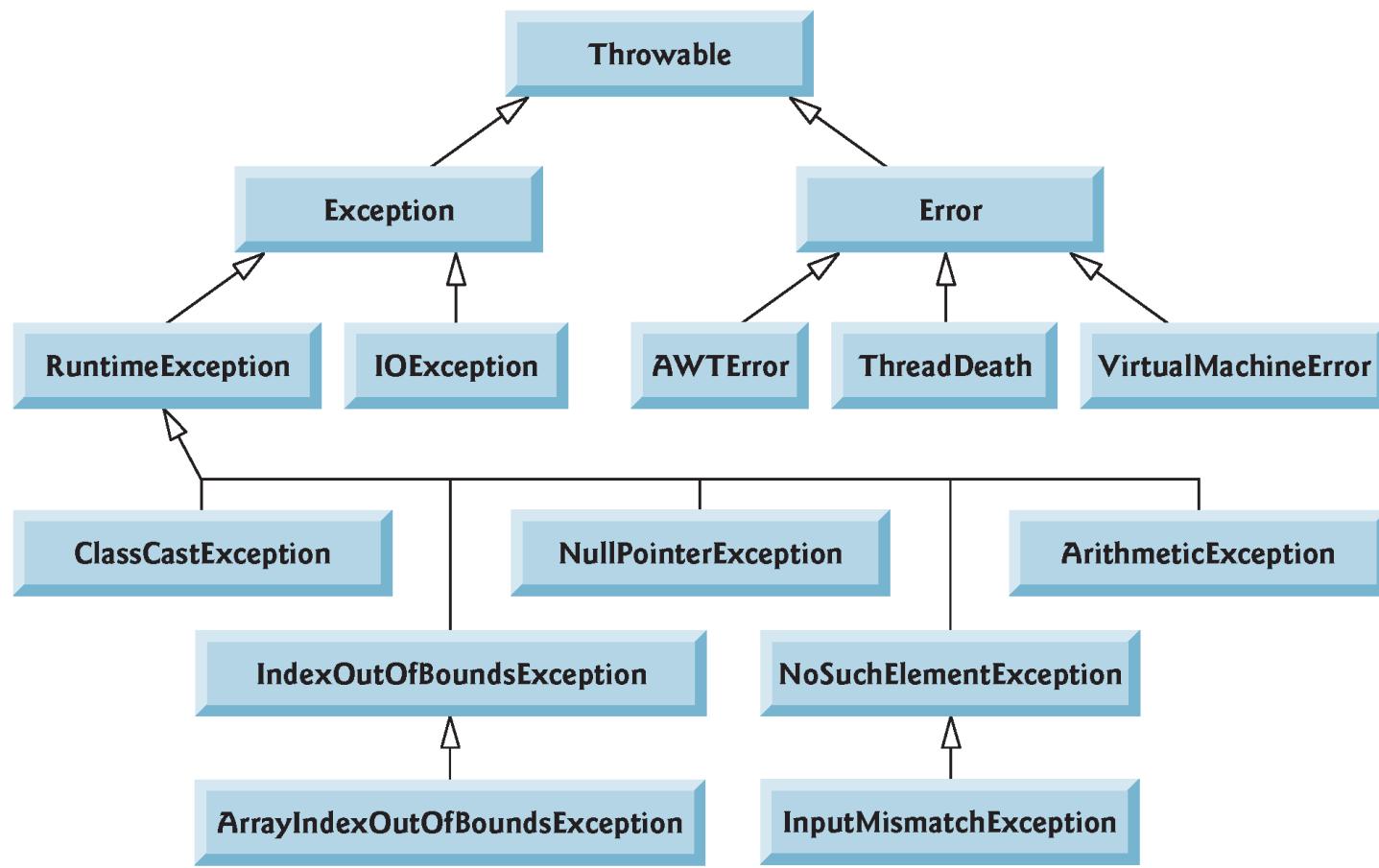


Fig. 11.4 | Portion of class `Throwable`'s inheritance hierarchy.



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 1.0.0 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 1.0.0 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

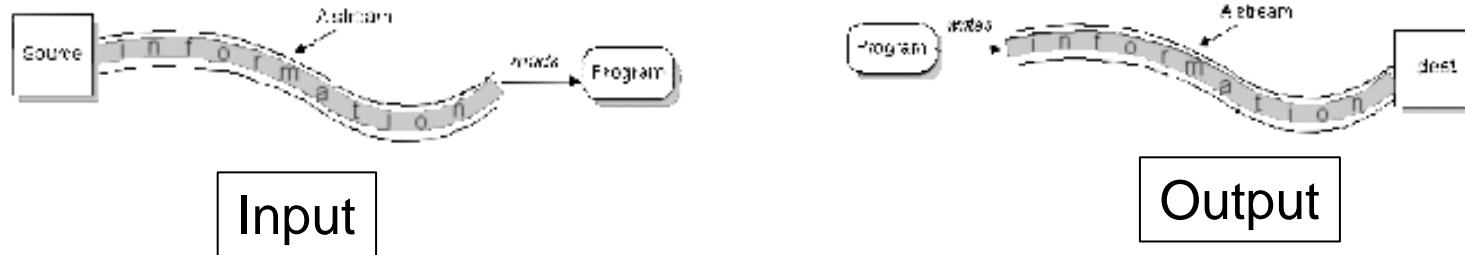


INTERFACCIE UTENTE

- 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



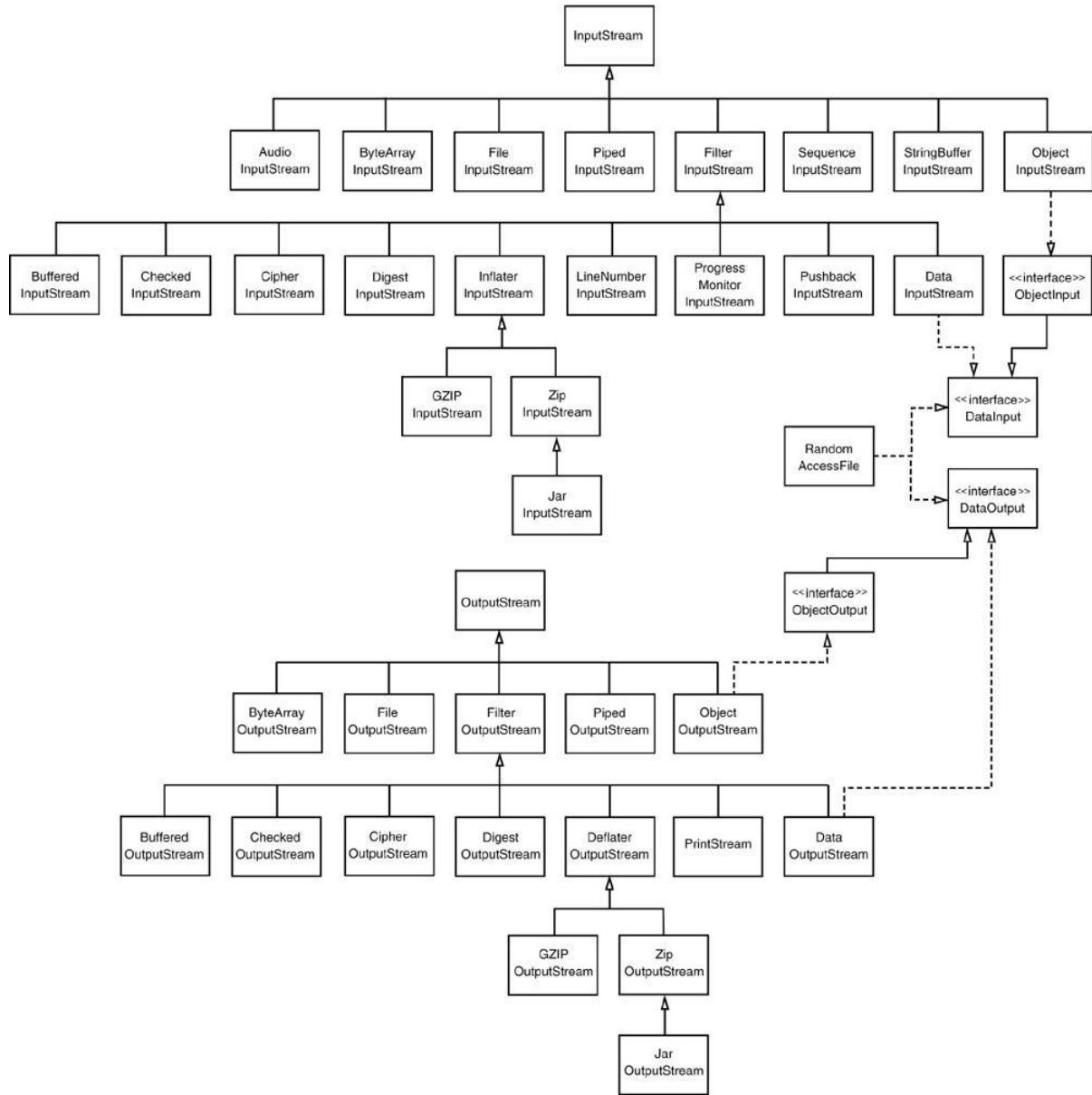
ALTRÉ CLASSI PER L'INPUT E L'OUTPUT

- 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



PATTERN DECORATOR (CENNO)

- Spesso nell'input/output abbiamo visto utilizzare una soluzione che viene chiamata pattern **Decorator**
- Consiste nel creare una classe che va a migliorare (“decorare”) una classe esistente:
 - Estendendo (extends) la classe originale
 - Avendo un costruttore che prende come parametro un oggetto della classe originale
 - Avendo alcuni metodi che arricchiscono quelli già ereditati dalla classe originale
- Esempio:
 - BufferedReader ha bisogno di un FileReader come parametro
 - Fornisce metodi come readLine per una migliore lettura
- E' possibile innestare più decoratori, in modo che l'ultimo vada a migliorare la classe già migliorata da un altro decoratore



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 Concreta extends Base{
    int m(){return 1};
}

Base b=new Concreta();
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

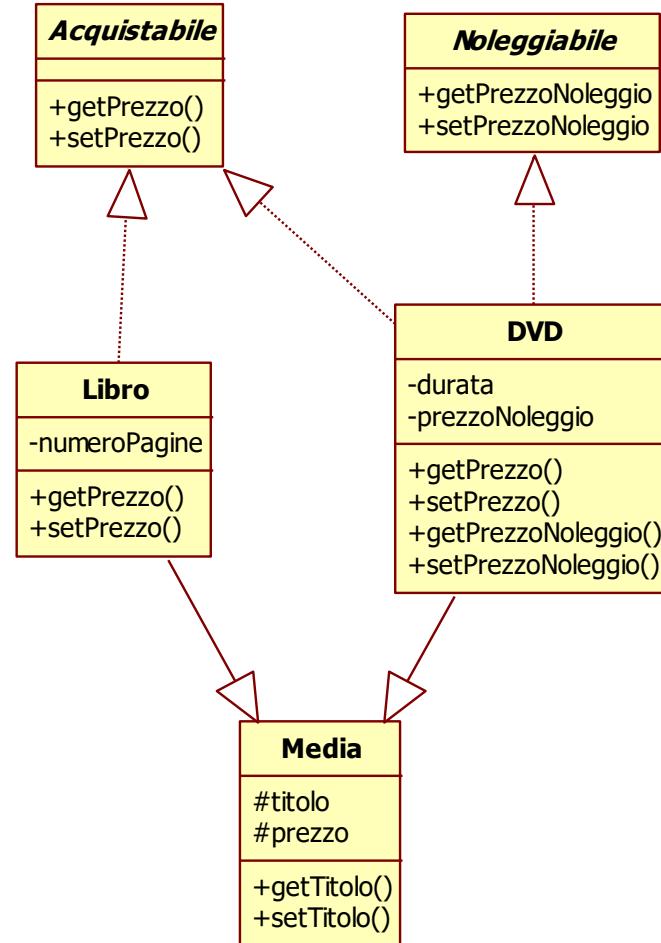


INTERFACCE E IMPLEMENTAZIONE

- 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



SWING UI DESIGNER

- In IntelliJ Idea è integrato uno strumento per la realizzazione di GUI con Swing che può essere considerato lo stato dell'arte per questa piattaforma
- Nella filosofia di Swing UI Designer di IntelliJ Idea l'interfaccia utente grafica va disegnata con lo strumento visuale a disposizione, che la salverà sotto forma di codice XML (dichiarativo) sotto forma di file con estensione *.form*
 - I file *.form* contengono solo le informazioni sulla struttura delle interface grafiche
 - Il *comportamento* delle GUI va realizzato direttamente in Java
- Altri strumenti, disponibili con altre piattaforme, consentono la generazione diretta di codice GUI tutto in Java



SEPARAZIONE TRA LAYOUT E COMPORTAMENTO DELLA GUI

- Una pratica di programmazione moderna: separare il disegno (grafico) dell'interfaccia utente dalla programmazione del suo comportamento
- Vantaggi:
 - Persone diverse con competenze diverse possono occuparsi in momenti diversi di aspetti diversi dello sviluppo
 - Spesso la struttura grafica è realizzata da un grafico anzichè da un informatico
 - Le scelte stilistiche possono essere riutilizzate da un software all'altro, indipendentemente dalle funzionalità da realizzare
- Svantaggi:
 - Per la descrizione dell'interfaccia utente viene utilizzato un diverso linguaggio di programmazione (XML, di natura dichiarativa)
 - L'utilizzo dello strumento visuale ci permette di sviluppare l'interfaccia anche senza conoscere il linguaggio
 - Bisogna capire bene come interagiscono le parti sviluppate nei due linguaggi



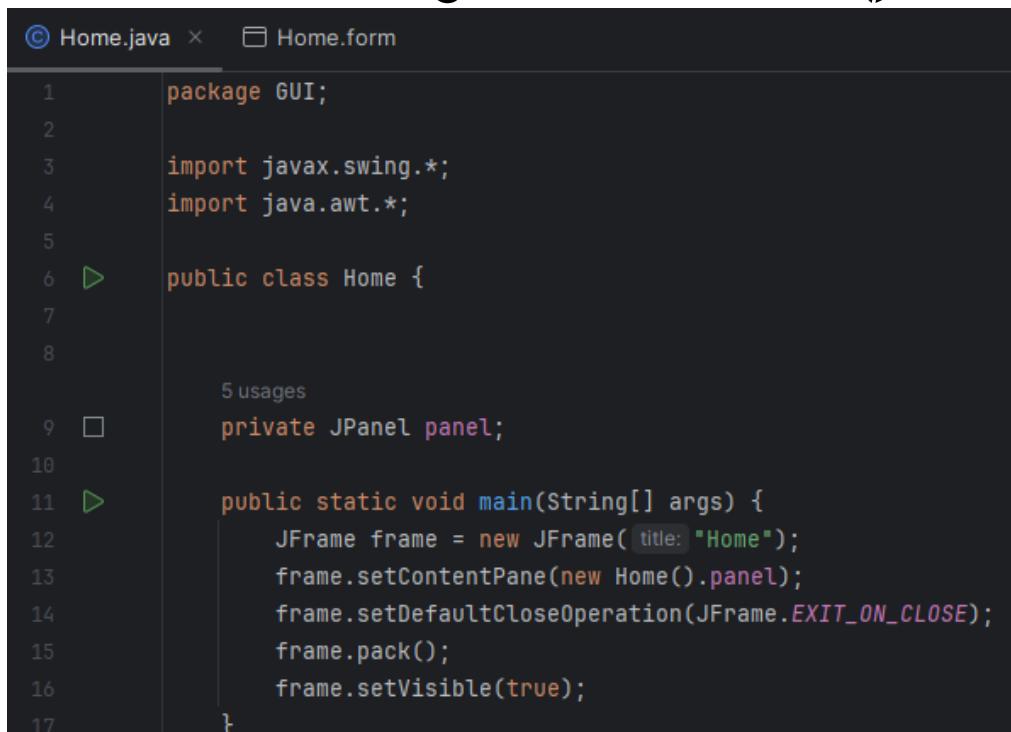
CREAZIONE DI UNA FORM GRAFICA

- L'elemento basilare di una GUI è detto *form*
 - Una form generalmente coincide con una finestra dell'applicazione
- Per creare una form è sufficiente scegliere New/Swing UI Designer/New Form
 - Al momento della creazione, possiamo stabilire sia il nome del file .form che conterrà la struttura, che il nome del corrispondente file java nel quale programmeremo il suo comportamento
 - Il collegamento tra il form e il file java è scritto nel form e verrà letto in fase di esecuzione

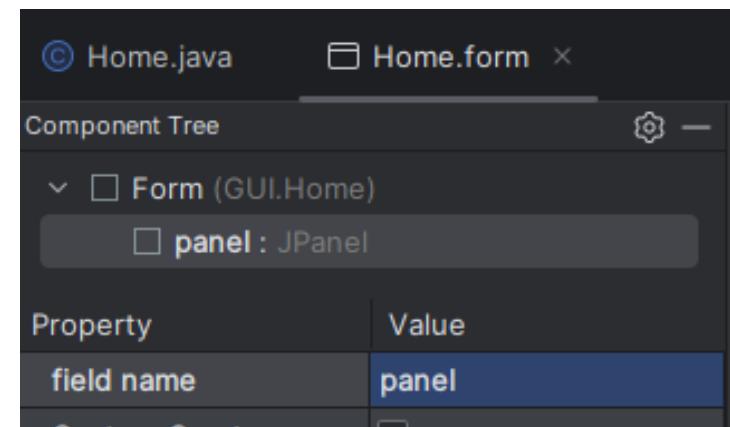


CREAZIONE DELLA PRIMA FORM

1. Nella vista grafica diamo un nome al panel (in field name)
2. Andiamo nel file java, premiamo **Alt+Insert** e scegliamo *form main()*

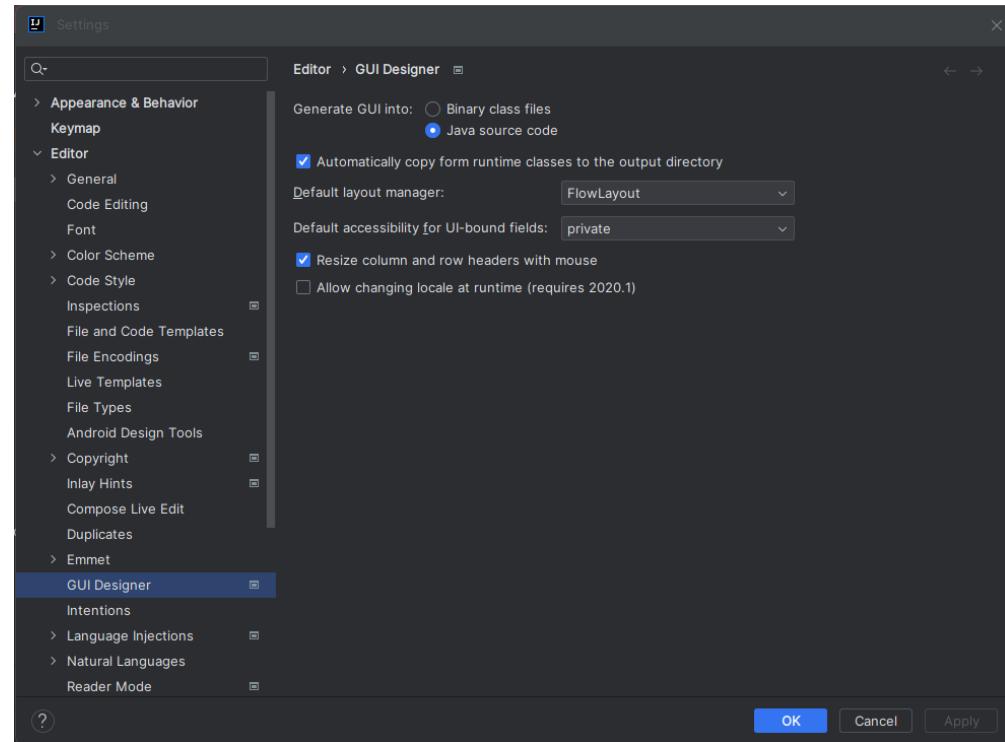


```
© Home.java ×  Home.form
1 package GUI;
2
3 import javax.swing.*;
4 import java.awt.*;
5
6 public class Home {
7
8
9     □ 5 usages
10    □ private JPanel panel;
11
12    □ public static void main(String[] args) {
13        JFrame frame = new JFrame("Home");
14        frame.setContentPane(new Home().panel);
15        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16        frame.pack();
17        frame.setVisible(true);
18    }
19}
```



ABILITARE GENERAZIONE CODICE GUI

- Il disegno visuale di elementi con Swing GUI Design comporta la generazione di un po' di codice Java, che normalmente è nascosto (generato direttamente sotto forma di bytecode).
- Se vogliamo vedere questo codice possiamo abilitarne la visualizzazione dal menu Settings



INIZIALIZZAZIONE

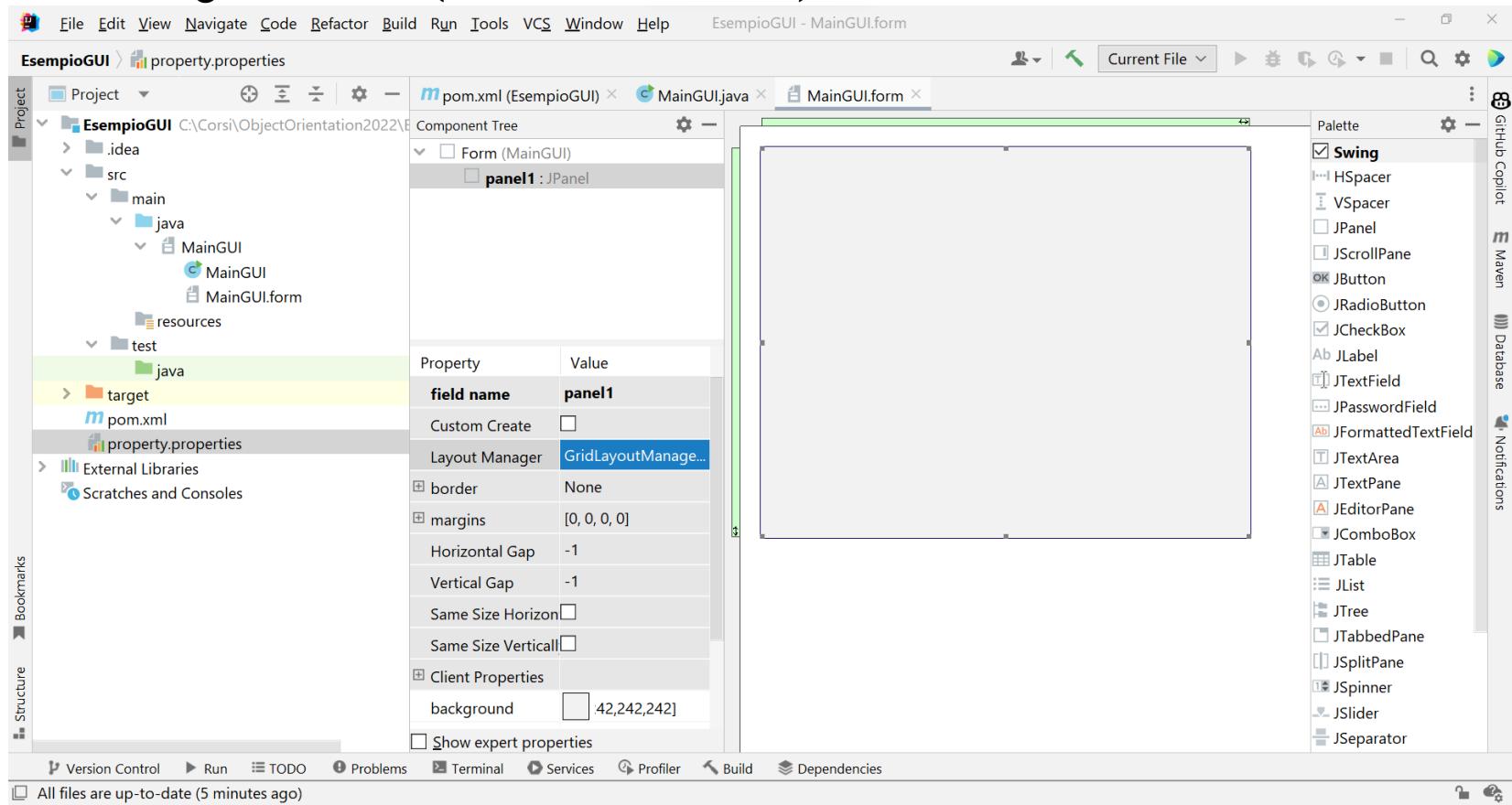
- Il *JFrame* (cornice) è la finestra nella quale ci saranno gli elementi da visualizzare
- Il *JPanel* è un oggetto incluso nel *JFrame* attraverso il quale è possibile specificare il contenuto della GUI
- Sul *JFrame* sono chiamati alcuni metodi notevoli, in particolare *setVisible(true)* che lo rende visible sullo schermo
- Se vogliamo interagire successivamente con il frame è opportuno anticiparne la dichiarazione tra gli attributi
 - *private static JFrame frame;*
(static perché deve essere sempre in memoria, così come il main)

```
public class MainGUI {  
    private JPanel panel1;  
  
    public static void main(String[] args) {  
        JFrame frame = new JFrame("MainGUI");  
        frame.setContentPane(new MainGUI().panel1);  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        frame.pack();  
        frame.setVisible(true);  
    }  
  
    public MainGUI() {  
    }  
}
```



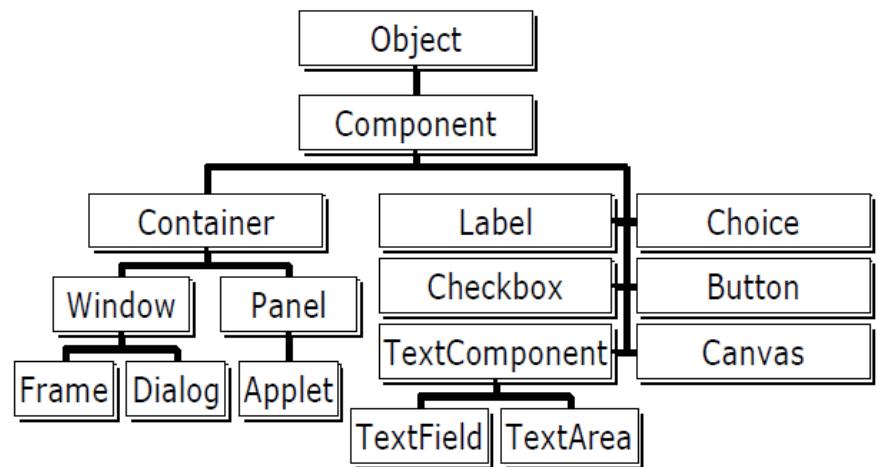
FINESTRA DI DESIGN

- Con Swing UI Designer è possibile disegnare la GUI sfruttando la palette di strumenti (a destra), la struttura gerarchica (al centro in alto) e modificando i valori degli attributi (al centro in basso)



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
- I componenti che hanno un nome che inizia per J sono stati introdotti dalla libreria Swing, mentre gli altri provengono dalla vecchia libreria AWT, ancora utilizzabile
- <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>



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:
 - **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 dalle proprietà del widget)
 - **Grid Layout Manager**
 - Probabilmente la più completa, consigliata da IntelliJ Idea
 - Altri layout più complessi sono disponibili, che consentono di disporre gli elementi ad esempio in griglie, schede o form
 - Il posizionamento esatto, pixel per pixel è sconsigliato nella maggior parte dei casi perchè fallirebbe appena si provasse ad utilizzare il programma in una finestra con dimensioni o proporzioni diverse da quelle progettate



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

The screenshot shows the NetBeans IDE interface with the following details:

- Component Tree:** A tree view of the application's structure. It starts with "Form (MainGUI)", which contains "panel1 : JPanel". "panel1" contains a "JPanel" which holds "button1 : JButton". Below "button1", another "JPanel" is shown, which contains "button2 : JButton".
- Properties Table:** A table showing properties and their values for the selected component.

Property	Value
field name	button1
Custom Create	<input type="checkbox"/>
Layout Manager	FlowLayout
border	None



ESEMPI DI WIDGET

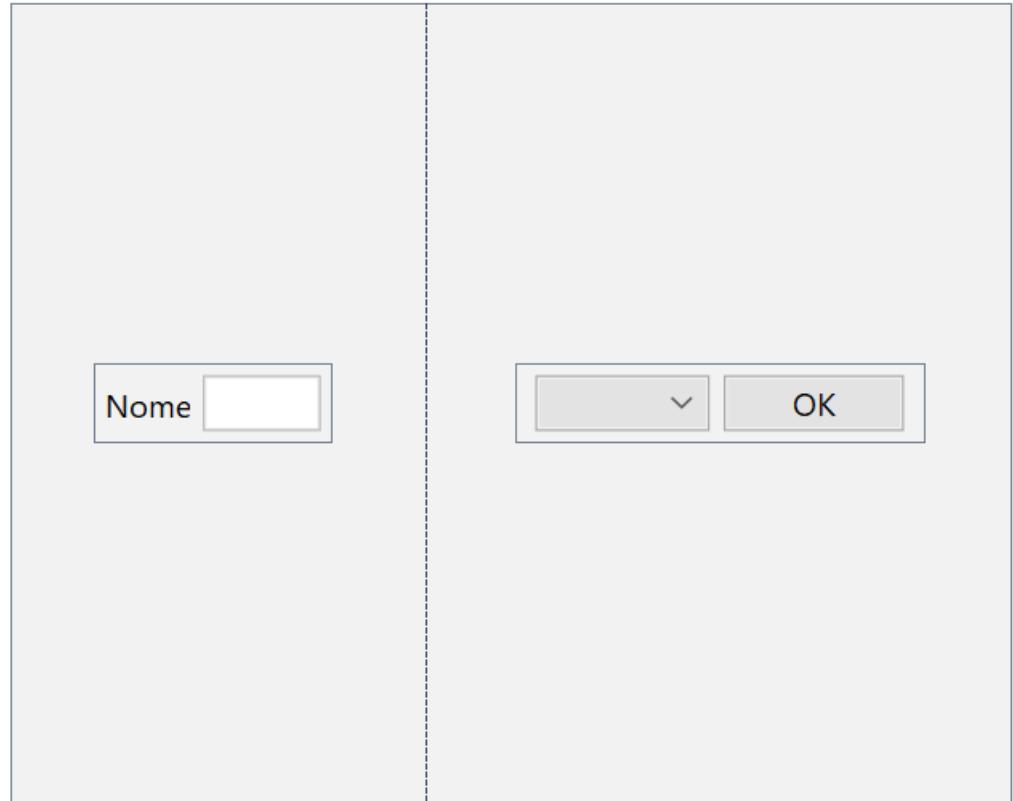
- **JLabel**: un'etichetta di testo
 - **JTextField**: una casella tramite cui inserire un testo
 - **JTextArea**: una casella tramite cui inserire un testo lungo (più righe)
 - **JComboBox**: una casella tramite cui inserire un valore appartenente ad un elenco di valori possibili
 - **JCheckbox**: una 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>

HSpacer
VSpacer
JPanel
JScrollPane
JButton
JRadioButton
JCheckBox
JLabel
JTextField
JPasswordField
JFormattedTextField
JTextArea
JTextPane
JEditorPane
JComboBox
JTable
JList
JTree
JTabbedPane
JSplitPane
JSpinner
JSlider
JSeparator



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



ALCUNI PRINCIPI DI MODELLAZIONE GRAFICA

- Scrivere sempre il campo *field name*, con un nome significativo
 - Sarà il nome dell'oggetto corrispondente nel codice sorgente
- Per aumentare la flessibilità nel posizionamento dei widget, utilizzare una gerarchia di JPanel perché ad ogni JPanel può essere abbinato un layout different
 - I JPanel sono invisibili, come default
 - La scheda Component Tree mostra la gerarchia dei panel
- Quando possibile, modificare sempre i widget tramite l'editor visuale anziché dal codice
 - Swing UI Designer potrebbe non riuscire a mappare sulla sua interfaccia le modifiche che imponiamo noi dal codice



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. Aggiungiamo al costruttore della GUI:

```
cmbGender.addItem("M");
```

```
cmbGender.addItem("F");
```

Ovviamente questa modifica non sarà visibile in Swing GUI Designer

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



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



ESEMPIO DI LISTENER

```
btnOK.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(frame,"Hello, World!");
    }
});
```

- new ActionListener() causa l'istanziazione di un oggetto della classe ActionListener
- Ma la classe ActionListener è astratta, per cui può essere prima completata fornendo implementazioni per i suoi metodi
- JOptionPane.showMessageDialog(frame,"Hello, World!"); è l'implementazione del metodo astratto actionPerformed di ActionListener
- La classe «ActionListener con implementazione di actionPerformed» 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 btnOK con il metodo addActionListener
- Da questo momento in poi, se la JVM intercetta un click button btnOK, esegue immediatamente il metodo actionPerformed definito



EVENTI

- Nell'esempio precedente l'oggetto ActionEvent è un parametro di input di actionPerformed e ci fornisce alcune informazioni
- 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>



DISCUSSIONE SUI LISTENER

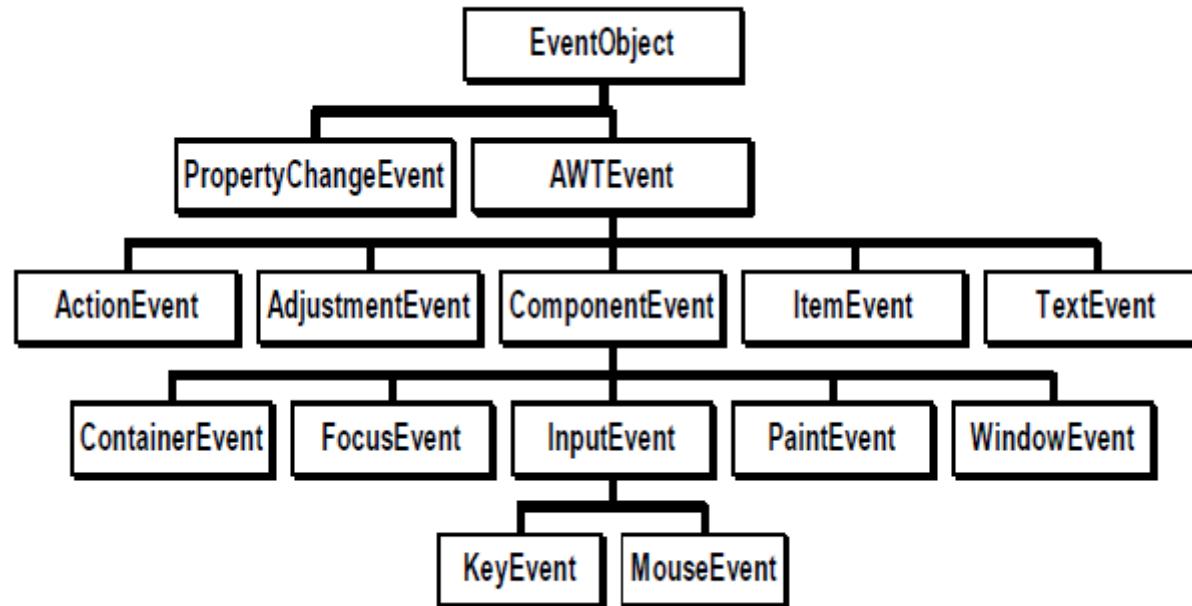
```
btnOK.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(frame,"Hello, World!");
    }
});
```

- Si tratta di un nuovo modo di programmare («programmazione reattiva») che è alla base di paradigmi di programmazione moderni molto usati, ad esempio, nel Web
- Il codice qui sopra è un codice solo dichiarativo: va eseguito prima possibile (nel main o nel costruttore del frame) e rimane dormiente fino a quando non avviene l'evento dichiarato
- Il programma non è più solo un processo che esegue un flusso di istruzione fino a che non termina, ma è una sorta di macchina a stati che esegue le sue inizializzazioni e rimane ferma fino a che non si verifica un evento (e ritorna ferma dopo aver eseguito il codice actionPerformed)
- Ovviamente i due paradigmi di programmazione possono convivere



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>



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>



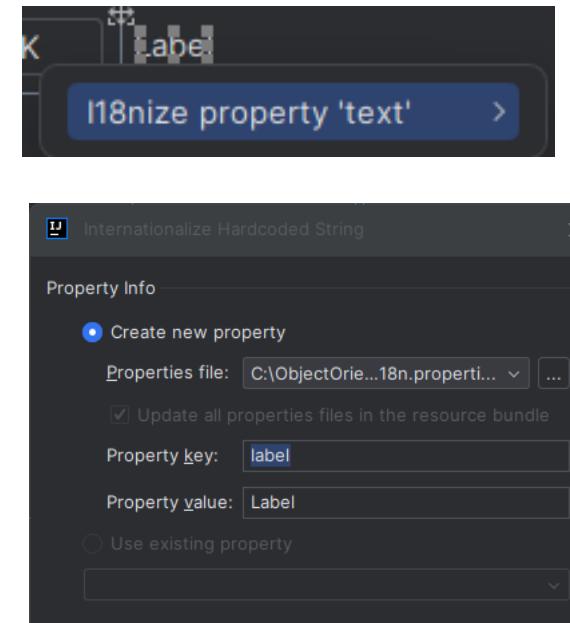
APPROFONDIMENTO : INTERNAZIONALIZZAZIONE

- E' buona norma rendere le applicazioni flessibili rispetto al cambiamento di localizzazione (ad es. lingua dell'utente)
- Tutte le stringhe non dovrebbero essere *cablate* in una espressione tra virgolette (stringa costante) ma essere prese da un file esterno (in IntelliJ viene consigliato un file con estensione *.properties*)
- In questo modo è possibile tradurre l'interfaccia utente da una lingua all'altra semplicemente andando a creare un nuovo file properties con tutte le stringhe tradotte, senza toccare il codice sorgente del programma
- E' anche possibile realizzare un sistema automatico che individui la lingua preferita dell'utente interrogando o settando un oggetto della classe Locale
- <https://docs.oracle.com/javase/8/docs/api/java/util/Locale.html>



INTERNAZIONALIZZAZIONE (ESEMPIO)

- Creare un file `it_IT.properties` vuoto
- Dall'interfaccia accettare il suggerimento `i18nize`
- Scegliere il file di properties creato
- Scrivere una key (nome logico della stringa nel codice) e un value (stringa nella lingua scelta)
- Possiamo osservato un codice generato come:
 - `this.$$$loadLabelText$$$(lblNome,
this.$$$getMessageFromBundle$$$("it_IT", "nome"));`
- Possiamo utilizzare un codice come questo anche nel nostro costruttore, ad esempio per il messaggio restituito dal pulsante OK
- Per poter cambiare dinamicamente lingua, possiamo utilizzare una variabile al posto di `"it_IT"`



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 sia 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();`



ESEMPIO PRATICO: LA PRIMA GUI (HOME)

```
public class Home {  
    private JPanel panel1;  
    private static JFrame frame;  
  
    public Home() {  
        public static void main(String[] args) {  
            frame = new JFrame("Home");  
            frame.setContentPane(new Home().panel1);  
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
            frame.pack();  
            frame.setVisible(true);  
        }  
    }  
}
```



ESEMPIO PRATICO: LA PRIMA GUI CHIAMA LA SECONDA

- Nel codice del costruttore di Home:

```
secondaGUIButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        SecondaGUI secondaGUI = new SecondaGUI(frame);
        secondaGUI.frame.setVisible(true);
        frame.setVisible(false);
    }
});
```

Nel codice di SecondaGUI

```
public JFrame frame;

public SecondaGUI(JFrame frameChiamante) {
    frame= new JFrame("SecondaGUI");
    frame.setContentPane(panel1);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.pack();
    frame.setVisible(true);
```

Home istanzia un nuovo oggetto secondaGUI e lo rende visibile, diventando nel contempo invisibile (ma senza deallocarsi)

SecondaGUI si istanzia e diventa visibile ma non perde il riferimento a frameChiamante



ESEMPIO PRATICO: RITORNO DALLA SECONDA GUI ALLA PRIMA

- Codice del button di SecondaGUI che ritorna verso MainGUI

```
btnClose.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        frameChiamante.setVisible(true);
        frame.setVisible(false);
        frame.dispose();
    }
});
```

- Il riferimento a frameChiamante ora è fondamentale per poter rendere visibile il frame chiamante dopo la scomparsa di questo frame
- dispose() rende deallocabile SecondaGUI (sarà deallocate dal garbage collector)



COMUNICAZIONE TRA JFRAME: PASSAGGIO DEI DATI

- Nell'esempio precedente veniva trasferito il controllo tra due classi GUI 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

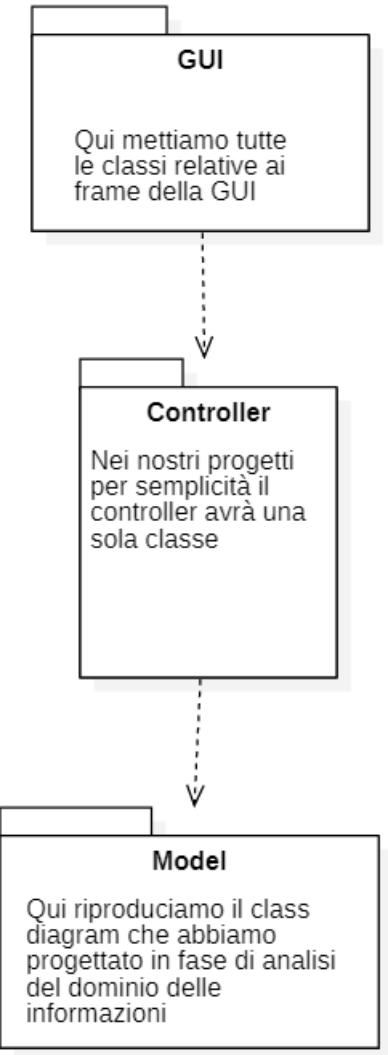
- Lo stile architetturale che adotteremo in questo corso prevede che le comunicazioni tra le finestre riguardino solo il passaggio del *controllo* tra una finestra e l'altra
 - Un JFrame che ne aprirà un altro, quindi, passerà il riferimento a sè stesso in modo da poter essere successivamente restituito 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)



ARCHITETTURA DI MASSIMA DI UN PROGETTO: MODELLO BCE

■ Modello BCE

- B : Boundary – GUI - l'interfaccia del software con l'esterno (in questo caso l'utente)
- C : Controller – l'insieme di classi che realizzano le operazioni algoritmiche, su richiesta diretta dalla GUI e gestendo i dati rappresentati nel Model
- E : Entity – Model – l'insieme di classi che riproduce il diagramma del dominio delle informazioni da rappresentare
- Nota : in questo modello non è prevista la memorizzazione permanente dei dati, che affronteremo dopo



MODELLO BCE : RESPONSABILITÀ'

■ GUI

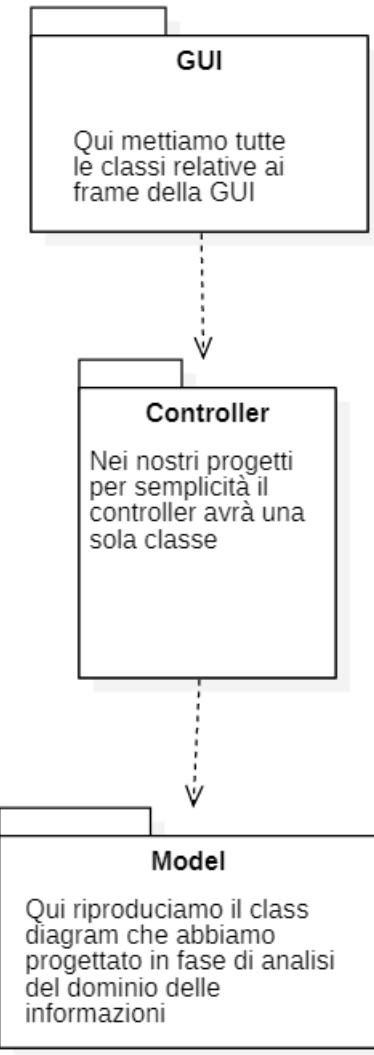
- E' responsabile dell'interfaccia grafica, dell'inserimento degli input, della loro validazione, della visualizzazione degli output
- **NON** realizza elaborazione dei dati
- **NON** accede direttamente alle classi del Model

■ Controller

- E' responsabile della realizzazione delle funzionalità del Sistema
- Riceve richieste **SOLO** dalla GUI
- Chiede letture e scritture dei dati conservati nel Model
- E' unico e non viene distrutto prima della chiusura del programma

■ Model

- E' responsabile della memorizzazione dei dati tramite oggetti
- Riceve richieste **SOLO** dal Controller
- Esegue solo piccole elaborazioni (leggi, scrivi, cerca, ...)



INSERIMENTO DEL CONTROLLER NELLA GUI

- Tutte le classi della GUI del Progetto devono poter accedere allo stesso oggetto controller
- Possibile soluzione:
 - La prima GUI (Home) istanzia un oggetto controller
 - Successivamente lo passerà a tutte le GUI che verranno aperte

```
public class Home {  
    private Controller controller;
```

```
public Home() {  
    controller= new Controller();
```

...

```
SecondaGUI secondaGUI = new SecondaGUI(frame, controller);
```

- La SecondaGUI riceverà quindi il riferimento all'oggetto controller

```
public SecondaGUI(JFrame frameChiamante, Controller controller) {
```



FUNZIONALITA' DA REALIZZARE PER BORSA

- Set nome giocatore
- Set citta borsa
- Aggiungi nuova societa (nome, valore azione)
- Leggi listino societa
- ...
- Modifica valore azione (nome societa)
- Acquista (listino, nome societa, valore azione, quantita)
- Leggi lista acquisti (giocatore)
- Calcola bilancio



SET NOME GIOCATORE : GUI

- Direttamente nella GUI

- `nomeDelGiocatoreButton.addActionListener(new ActionListener() {`

`@Override`

`public void actionPerformed(ActionEvent e) {`

`String s = JOptionPane.showInputDialog("Inserisci il tuo nome");`

`giocatoreText.setText("Benvenuto, "+s);`

`//inserire codice per la validazione del dato in input`

`controller.setNomeGiocatore(s);`

}

`});`

- Tramite JOptionPane prendiamo in input il nome del giocatore con un Dialog di librerie

- Il Widget giocatoreText ci permette di mostrare il nome del giocatore anche in output sullo schermo

- Comunichiamo al controller il nome del giocatore, che lo comunicherà al Model



SET NOME GIOCATORE : CONTROLLER

- Il Controller dovrà prima di tutto occuparsi di mantenere lo stato del gioco, tramite l'istanziazione di oggetti delle classi Model

```
public class Controller {  
  
    private Borsa borsa;  
    private Giocatore giocatore;
```

```
public Controller() {  
    borsa=new Borsa();  
}
```

- Per gestire il giocatore sarà sufficiente un metodo:

```
public void setNomeGiocatore(String text) {  
    giocatore=new Giocatore(text);  
}
```



SET NOME GIOCATORE : MODEL

- In questo è sufficiente implementare il costruttore dell'oggetto Giocatore, che prenda in input una stringa (lo avevamo già realizzato):

```
public class Giocatore {  
    private String nome;  
    private float liquidita;  
    private float disponibilitaIniziale=100000;  
  
    private ArrayList<Acquisto> acquisti;  
  
    public Giocatore(String s) {  
        nome=s;  
        liquidita=disponibilitaIniziale;  
        acquisti = new ArrayList<Acquisto>();  
    }  
}
```



SET CITTA BORSA : GUI

- La soluzione è molto simile al caso precedente

```
cittàDellaBorsaButton.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
  
        String s = JOptionPane.showInputDialog("Inserisci il nome della città della  
        borsa");  
        cittàBorsaText.setText("Si gioca su "+s);  
        //salvare il nome della città della borsa  
        controller.setCittaBorsa(s);  
    }  
});
```



SET CITTA BORSA : CONTROLLER

- Tenuto conto che borsa è già un attributo del controller istanziato dal costruttore, dobbiamo solo settare la citta

```
public void setCittaBorsa(String text) {  
    borsa.setCitta(text);  
}
```



SET CITTA BORSA : MODEL

- Il set della città è banalmente

```
public void setCitta(String text) {  
    citta=text;  
}
```



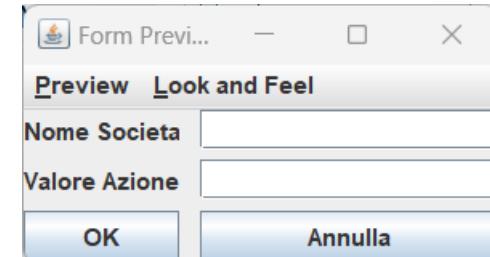
AGGIUNGI NUOVA SOCIETA (NOME, VALORE AZIONE)

- Siccome per aggiungere una nuova società c'è bisogno di un doppio input, per questa volta scegliamo di creare una nuova form e la chiamiamo dalla Home

```
aggiungiNuovaSocietaButton.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        AggiungiNuovaSocieta aggiungiNuovaSocieta = new  
        AggiungiNuovaSocieta(controller,frame);  
        aggiungiNuovaSocieta.frame.setVisible(true);  
        frame.setVisible(false);  
    }  
});
```



AGGIUNGI NUOVA SOCIETA (NOME, VALORE AZIONE) : BUTTON OK

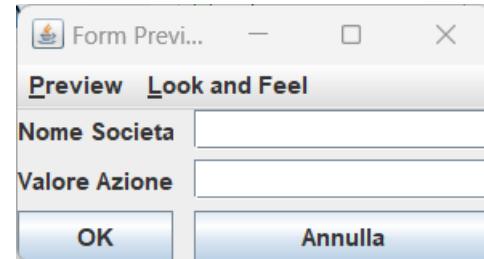


- Alla pressione del button OK bisogna:
 - Verificare che I dati in input siano validi (lo faremo in futuro)
 - Inserire I dati nel model tramite il controller
 - Tornare alla GUI precedente (annulla farà solo quest'ultima operazione)

```
OKButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        //aggiungere la società al listino
        controller.aggiungiSocietaBorsa(nomeSocietaText.getText(),Double.parseDouble(valo
reAzioneText.getText()));
        frame.setVisible(false);
        frameChiamante.setVisible(true);
        frame.dispose();
    }
});
```



AGGIUNGI NUOVA SOCIETA (NOME, VALORE AZIONE) : CONTROLLER



- Il codice nel controller è molto semplice:

```
public void aggiungiSocietaBorsa(String text, double parseDouble) {  
    borsa.aggiungiSocieta(text, parseDouble);  
}
```



AGGIUNGI NUOVA SOCIETA (NOME, VALORE AZIONE) : AGGIUNGISOCIETA NEL MODEL

- L'oggetto borsa del model ha un attributo ArrayList di Societa, che inizialmente è vuoto
- Il metodo aggiungiSocieta istanzia un oggetto società e lo aggiunge all'ArrayList

```
public class Borsa {  
    private ArrayList<Societa> societa;  
  
    public Borsa() {  
        societa = new ArrayList<Societa>();  
    }  
    public void aggiungiSocieta(String text, double parseDouble) {  
        Societa s = new Societa(text, parseDouble);  
        societa.add(s);  
    }  
}
```



AGGIUNGI NUOVA SOCIETA (NOME, VALORE AZIONE) : COSTRUTTORE SOCIETA

- Il costruttore per la Societa è semplicemente:

```
public class Societa {  
    private Double valoreAzione;  
    private String nome;  
  
    public Societa(String n, Double valore) {  
        nome=n;  
        valoreAzione = valore;  
    }  
}
```



LEGGI BORSA SOCIETA : GUI HOME

- Per visualizzare a video tutte le società e il valore delle loro azioni scegliamo di creare un nuovo frame che dovremo chiamare dalla Home
- Non dimentichiamo di passare il riferimento al frameChiamante (per potervi ritornare) e al Controller

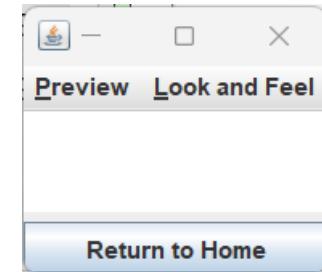
```
leggiSocietaButton.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        LeggiSocieta leggiSocieta = new LeggiSocieta(controller,frame);  
        leggiSocieta.frame.setVisible(true);  
        frame.setVisible(false);  
    }  
});
```



LEGGI SOCIETA : GUI LEGGI SOCIETA

- La GUI LeggiListino deve sicuramente contenere una tabella e un pulsante per tornare alla Home
- La tabella (vuota) non è visibile dall'anteprima
- Codice fondamentale (senza tabella)

```
public class LeggiSocieta {  
    private JPanel panel1;  
    private JTable table1;  
    private JButton returnToHomeButton;  
    JFrame frame;  
  
    public LeggiSocieta(Controller controller, JFrame frameChiamante) {  
        frame = new JFrame("Leggi Listino societa");  
        frame.setContentPane(panel1);  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        frame.pack();  
        frame.setLocationRelativeTo(frameChiamante);  
        returnToHomeButton.addActionListener(new ActionListener() {  
            @Override  
            public void actionPerformed(ActionEvent e) {  
                frame.setVisible(false);  
                frameChiamante.setVisible(true);  
                frame.dispose();  
            }  
        });  
    }  
}
```



LEGGI SOCIETA : TABLE IN GUI LEGGI SOCIETA

- All'interno della GUI LeggiSocieta dobbiamo realizzare una Table
- Sarebbe opportuno consultare la documentazione di JTable
- Utilizzo possibile :

```
// Fissiamo uno schema (model) per la tabella, come matrice quadrata con due  
// colonne e i valori Societa e Valore Azione sull'intestazione
```

```
table1.setModel(new DefaultTableModel( new Object[][] { },  
        new String[] { "Societa", "Valore Azione" } ) );  
DefaultTableModel model = (DefaultTableModel) table1.getModel();  
  
// leggiamo tramite il controller le liste di nomi societa e valori azioni  
ArrayList listaNomiSocieta = controller.getNomiSocieta ();  
ArrayList listaValoriAzione = controller.getValoriAzione ();  
if (listaNomiSocieta!=null)  
    for (int i=0;i<listaNomiSocieta.size();i++)  
  
        //riempiamo il model che mostrerà i valori sullo schermo  
        model.addRow(new Object[]{listaNomiSocieta.get(i),listaValoriAzione.get(i)});  
    }
```



LEGGI SOCIETA : CONTROLLER

- Il codice nel controller è molto semplice:

```
public ArrayList getNomiSocieta () {  
    return borsa.getNomiSocieta();  
}
```

```
public ArrayList getValoriAzione () {  
    return borsa.getValoriAzione();  
}
```



LEGGI SOCIETA : LEGGI SOCIETA DA CONTROLLER

- Nell'oggetto Listino del Model abbiamo bisogno di metodi per avere la lista dei nomi delle società e la lista dei valori delle loro azioni:

```
public ArrayList<String> getNomiSocieta() {  
    ArrayList<String> nomiSocieta = new ArrayList<String>();  
    for (Societa s : societa) {  
        nomiSocieta.add(s.getNome());  
    }  
    return nomiSocieta;  
}
```

```
public ArrayList<String> getValoriAzione() {  
    ArrayList<String> valoriSocieta = new ArrayList<String>();  
    for (Societa s : societa) {  
        valoriSocieta.add(s.getValoreAzione().toString());  
    }  
    return valoriSocieta;  
}
```



CALCOLA BILANCIO

- In questo caso c'è da fare un ragionamento nuovo
- Il bilancio può cambiare a seguito di diverse operazioni (ad esempio modifica valore azione è la prima di queste)
- Analogamente può cambiare anche la liquidità
- La Home deve mostrare bilancio e liquidità, che però sono modificati da altri
- In assenza di altri meccanismi, una soluzione per avere un bilancio sempre aggiornato mostrato sulla Home è quello di aggiornare i valori visualizzati ogni volta che la Home viene aperta
 - Così il valore verrà aggiornato a prescindere dall'effettiva necessità di calcolarlo
- Come si scrive un evento che parte ogni volta che la Home diventa visible?



CALCOLA BILANCIO E LIQUIDITA

- È possibile scrivere un evento il cui ascoltatore è direttamente il frame, e viene chiamato quando esso diventa visibile
- C'è anche un altro evento che viene eseguito quando diventa invisibile

```
frame.addComponentListener(new ComponentAdapter() {  
    public void componentHidden(ComponentEvent e) {  
        /* code run when component hidden */  
    }  
    public void componentShown(ComponentEvent e) {  
        aggiorna();  
    }  
});  
  
void aggiorna() {  
    if (!controller.isNullGiocatore()) {  
        disponibilitaLabel.setText("Disponibilita': " + controller.getDisponibilita());  
        bilancioLabel.setText("Bilancio: " + controller.getBilancio());  
    }  
}
```



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

- 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



RUNNABLE JAR IN INTELLIJ IDEA

- Menu File → Project Structure ... → Artifacts
- Aggiungere un nuovo profilo di esecuzione con +
- Scegliere Jar e From modules with dependencies
 - così sfruttiamo le impostazioni di base create da Maven quando fu creato da zero il Progetto
- Scegliere la classe eseguibile (con static void main) tra tutte quelle eventualmente presenti
- Specificare eventualmente il percorso del .jar (predefinito: out\artifacts\)
- Dal menu Build scegliere Build artifacts e il profilo di esecuzione appena creato
- Il jar così ottenuto può essere eseguito anche con doppio click



INTEGRAZIONE CON IL DATABASE



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 Model e si occupa di interagire direttamente con il Database
- Pregi di questa soluzione:
 - Soluzione a livelli perfettamente separati
 - 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 ADOTTATA: BCE + PATTERN DAO

- In questo corso adotteremo una soluzione basata sul pattern **DAO**
 - DAO sta per Data Access Object
- Introduciamo un package **DAO** che contiene solo **interface**, relativamente a tutti i metodi che vogliamo realizzare tramite **query** sul database per gestire la **persistenza dei dati**
 - Ad esempio: inserimenti, aggiornamenti, eliminazioni, ricerche
- Per ogni database supportato introdurremo un package **ImplementazioneDAO** che implementi con classi concrete le interface del package DAO, realizzando le query che andremo a fare sul vero database
- Per comodità, l'apertura della connessione al database, commune a tutte le classi di un package **ImplementazioneDAO** la realizzeremo in una classe di utilità apposita



CLASSE CONNESSIONEDATABASE

- Per semplicità inseriamo le funzionalità relativa all'apertura del database ad una classe **ConnessioneDatabase** che offre:

- 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



CLASSE 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



ESEMPIO DI UTILIZZO DI CONNESSIONEDATABASE

- In una classe che deve eseguire interrogazioni (package ImplementazioneDAO)

```
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) { ... }

}
```

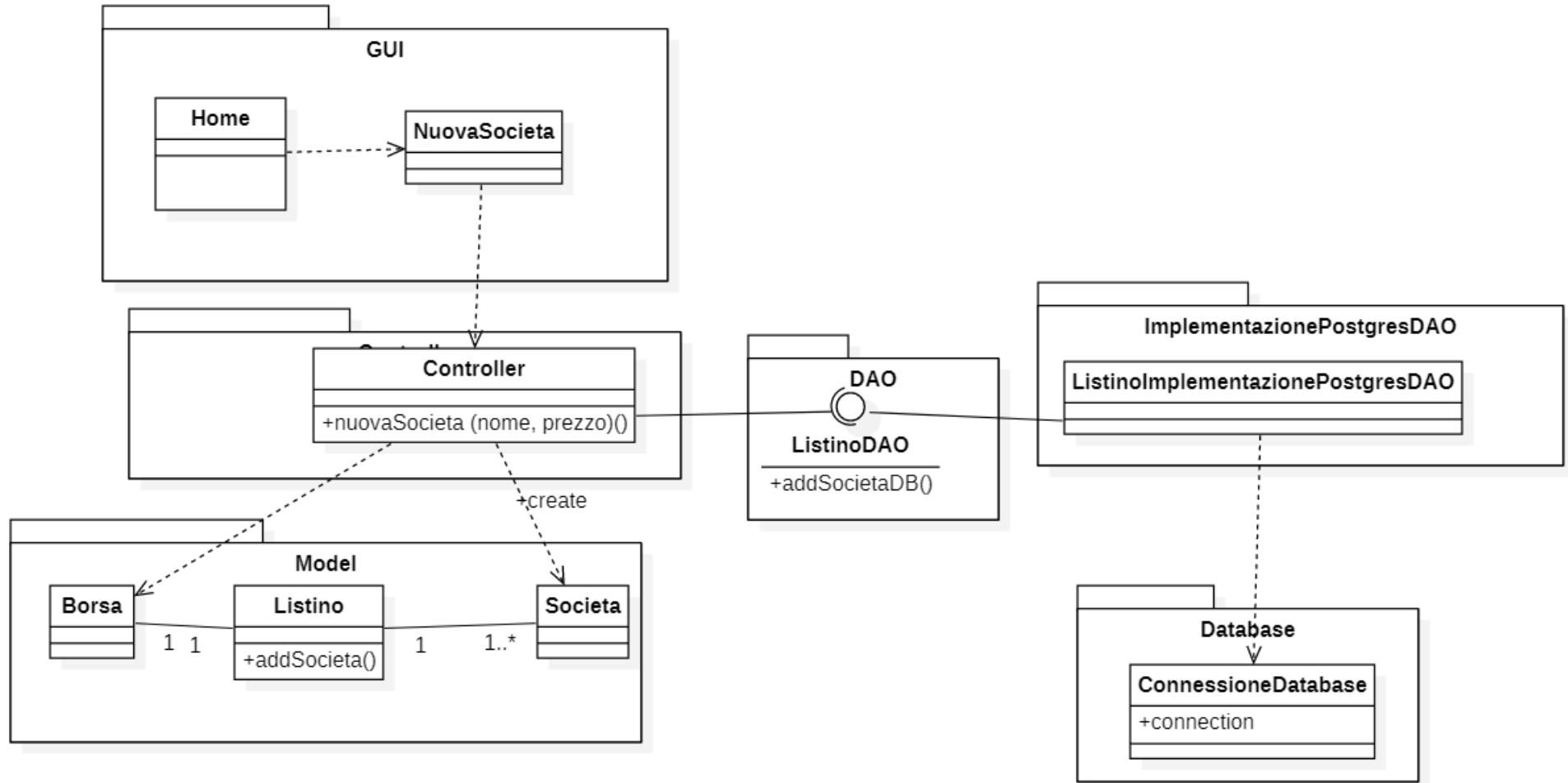


ARCHITETTURA COMPLESSIVA DEL PROGETTO

- 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 interface relative ai metodi da eseguire sul DB
- Esiste un package ImplementazioneDAO per ogni database da supportare (ad es. ImplementazionePostgresDAO). All'interno del package ci sono le classi con tutte le implementazioni delle interfacce DAO rispetto a quel database
- La connessione al database viene di solito gestita da una classe di utilità ConnessioneDatabase



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.AggiungiNuovaSocieta

```
controller.aggiungiSocietaListino(nomeSocietaText.getText(),Double.parseDouble(v  
aloreAzioneText.getText()));
```

- In Controller.aggiungiSocietaListino

```
listino.aggiungiSocieta(text, parseDouble); //in memoria  
ListinoDAO l=new ListinoImplementazionePostgresDAO();  
l.addSocietaDB(text, parseDouble,borsa.getCitta()); //scrive sul DB
```

- Nel costruttore di ListinoImplementazionePostgresDAO

```
connection = ConnessioneDatabase.getInstance().connection;
```

- In ListinoImplementazionePostgresDAO.addSocietaDB

```
PreparedStatement leggiListinoPS = connection.prepareStatement("INSERT INTO  
\\"Societa\\" " +"(\"Nome\", \"PrezzoAzione\", \"CittaBorsa\")" +"VALUES  
( '"+text+"', "+parseDouble+", '"+citta+"' );");  
addSocietaPS.executeUpdate();  
connection.close(); //opzionale
```

- Nota bene: a addSocietaDB sono stati passati dati e non oggetti del Model, con cui non deve interagire



OSSERVAZIONI

- Per ragioni di efficienza si può scegliere se aprire e chiudere connessioni o mantenerle aperte a lungo
 - Tenere aperta una connessione a lungo ci fa risparmiare il tempo per aprirla e chiuderla ma ci tiene occupata memoria e soprattutto potrebbe bloccare la possibilità di altre applicazioni di operare sulle stesse tabelle del database
- La separazione tra interfaccia DAO e implementazione consente di minimizzare il lavoro in caso si voglia cambiare DB
 - E' sufficiente aggiungere un nuovo package 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



ESEMPIO: LETTURA DELLE SOCIETA DEL LISTINO DI UNA BORSA – FRAMMENTI DI CODICE 1/2

- In GUI.Home

```
String s= JOptionPane.showInputDialog("Inserisci la città della borsa");
controller.setCittaBorsa(s);
```

- In Controller.setCittaBorsa

```
borsa=new Borsa(); //crea Borsa in memoria
borsa.setCitta(cittaBorsa);
ListinoDAO l=new ListinoImplementazionePostgresDAO();
ArrayList<String> nomiSocieta=new ArrayList<String>();
ArrayList<Double> prezziSocieta=new ArrayList<Double>();
l.leggiListinoDB(borsa.getCitta(),nomiSocieta,prezziSocieta); //legge listino
dal DB
for(int i=0;i<nomiSocieta.size();i++) {
    Societa s=new Societa(nomiSocieta.get(i),prezziSocieta.get(i));
    borsa.addSocieta(s);
} // costruisce gli oggetti Model a partire dai risultati del db
```



ESEMPIO: LETTURA DELLE SOCIETA DEL LISTINO DI UNA BORSA – FRAMMENTI DI CODICE 2/2

- In ListinoImplementazionePostgresDAO.leggiListinoDB

```
leggiListinoPS = connection.prepareStatement("SELECT * FROM \"Societa\" WHERE  
\"CittaBorsa\"='"+cittaBorsa+"'");  
ResultSet rs = leggiListinoPS.executeQuery();  
while (rs.next()) {  
    nomiSocieta.add(rs.getString("Nome"));  
    prezziSocieta.add(rs.getFloat("PrezzoAzione"));  
}  
rs.close();  
connection.close(); //opzionale  
return 1;
```

- Abbiamo dovuto introdurre le due ArrayList per non far interagire ListinoImplementazionePostgresDAO con Model (si sarebbero potuti utilizzare anche altri container più complessi)
- Non si poteva restituire rs al controller perchè deve essere chiuso da ListinoImplementazionePostgresDAO che lo ha aperto



OSSERVAZIONI

- 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
- Operare sul db oltre che in memoria ci mette al sicuro dalla perdita di dati in caso di eccezioni non gestite
- Ma impegna risorse (tempo e memoria)
 - Potremmo decidere anche di salvare tutte le modifiche sul database solo alla chiusura della sessione



ESEMPIO: ACQUISTO DI AZIONI – FRAMMENTI DI CODICE

- In **GUI.AcquistaAzione**
 - `controller.acquistaAzione(nomeSocieta, valoreAzione, quantita);`
- In **Controller.acquista**
 - `giocatore.acquistaAzione(societa, valoreAzione, listino, quantita); //in memoria`
 - `g.calcolaCapitale(); //con dati in memoria`
 - `GiocatoreDAO gDAO=new GiocatoreImplementazionePostgresDAO();`
 - `gDAO.acquistaDB(giocatore.getNome(), valoreAzione, quantita, societa); //su DB`
- Nel costruttore di **GiocatoreImplementazionePostgresDAO**
 - `connection = ConnessioneDatabase.getInstance().getConnection();`
- In **GiocatoreImplementazionePostgresDAO.acquistaDB**
 - `PreparedStatement nuovoAcquistoPS = connection.prepareStatement("INSERT INTO \"Acquisto\" " +"(\\" NomeSocieta\\", \\" NomeGiocatore\\", \\"Quantita\\", \\"PrezzoAzione\\")" +"VALUES (""+societa+"', '"+nomeGiocatore+"', "+quantita+", "+valoreAzione+");");`
 - `nuovoAcquistoPS.executeUpdate();`
 - `Connection.close(); //opzionale`



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 principale del progetto e poi indicare che si tratta di una libreria premendo il tasto destro e l'opzione **Add as Library**



DATABASE SCHEMA DUMP

```
CREATE TABLE public."Acquisto" (
    "NomeSocieta" character varying,
    "NomeGiocatore" character varying,
    "Quantita" integer,
    "PrezzoAzione" real
);
```

```
CREATE TABLE public."Societa" (
    "Nome" character varying,
    "PrezzoAzione" real,
    "CittaBorsa" character varying
);
```



RIASSUMENDO

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



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

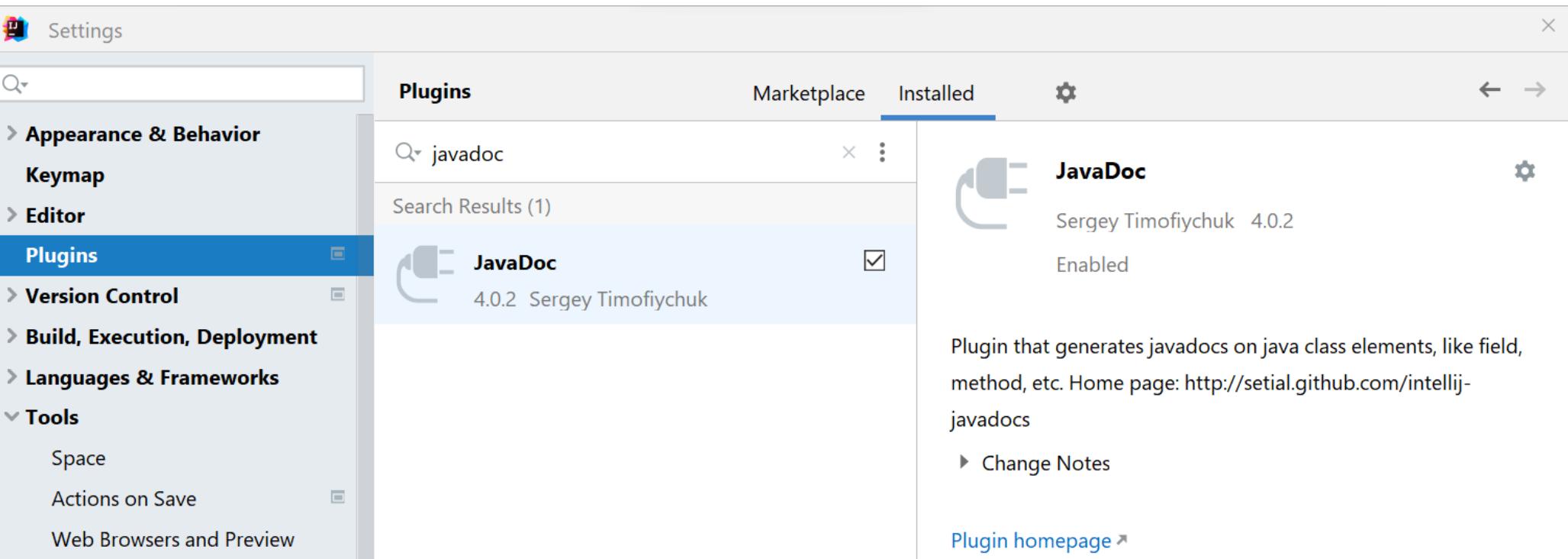
- **@author** *name-text*
- **@deprecated** *deprecated-text*
- **{@code** *text***}**
- **@exception** *class-name* *description*
- **{@link** *package.class#member label***}**
- **@param** *parameter-name* *description*
- **@return** *description*
- **@see** *reference*
- **@throws** *class-name* *description*
- **@version** *version-text*

- <https://www.oracle.com/it/technical-resources/articles/java/javadoc-tool.html>



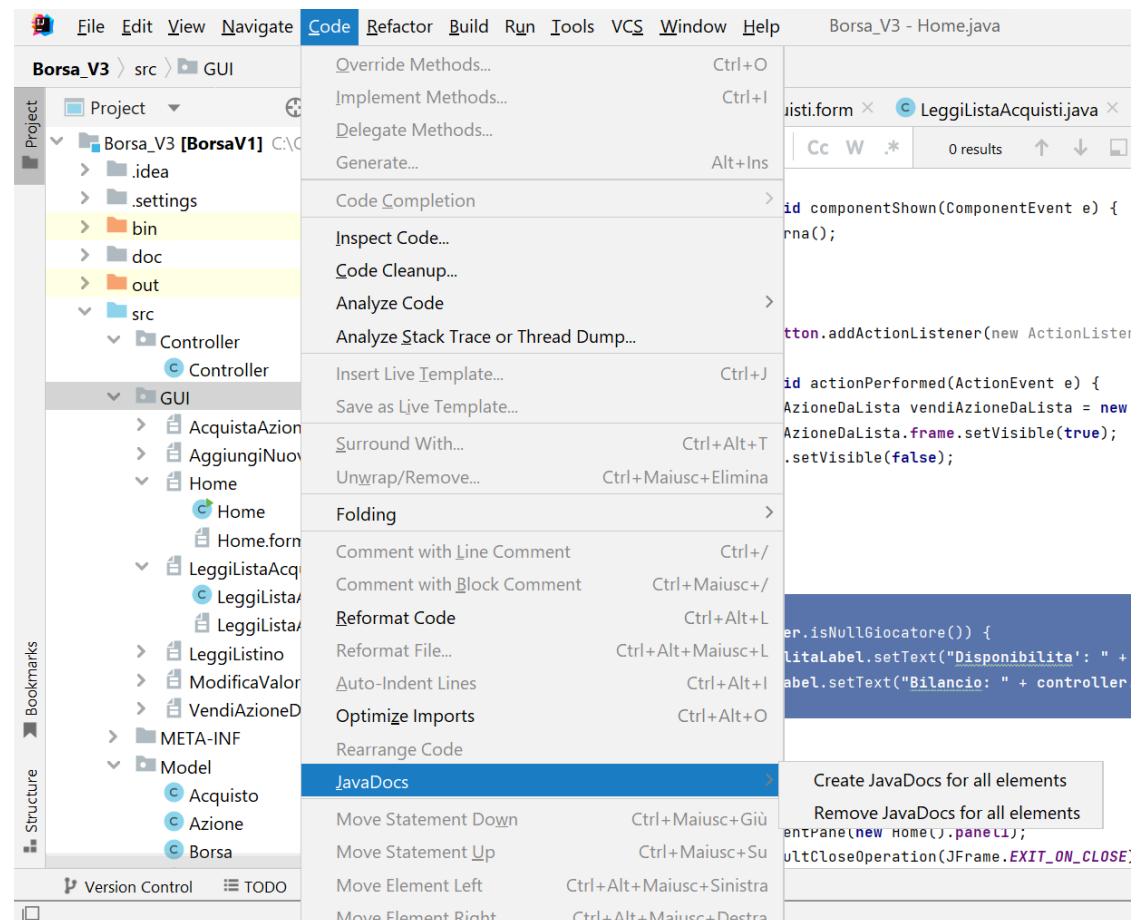
Generazione automatica di documentazione

- JavaDoc è un plug in che è in grado di generare automaticamente documentazione
- Si può installare da Settings→Plugin, cercando per Javadoc



GENERATORE JAVADOC

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



DOCUMENTAZIONE GENERATA

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

```
1 package Model;  
2  
3 import java.util.ArrayList;  
4  
5 /**  
6  * The type Giocatore.  
7  */  
8 public class Giocatore {  
9     1 usage  
10    private String nome;  
11    5 usages  
12    private float liquidita;  
13    1 usage  
14    private float disponibilitaIniziale=100000;  
15  
16    5 usages  
17    private ArrayList<Acquisto> acquisti;  
18  
19    /**  
20     * Instantiates a new Giocatore.  
21     *  
22     * @param s the s  
23     */  
24    public Giocatore(String s) {  
25        nome=s;  
26        liquidita=disponibilitaIniziale;  
27        acquisti = new ArrayList<Acquisto>();  
28    }  
29}
```



GENERAZIONE JAVADOC

- Per generare la documentazione in HTML è sufficiente la sequenza di comandi
 - Tools → Generate 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`[✉]
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[✉] nomeSocieta, int quantita)</code>	Acquista.
boolean	<code>cercaSocieta(String[✉] nomeSocieta)</code>	Cerca societa.
<code>String[✉]</code>	<code>getBilancio()</code>	Gets the bilancio.
<code>String[✉]</code>	<code>getCittaBorsa()</code>	Gets the citta borsa.
<code>ArrayList[✉]</code>	<code>getListeAcquisti()</code>	Gets the lista acquisti.

