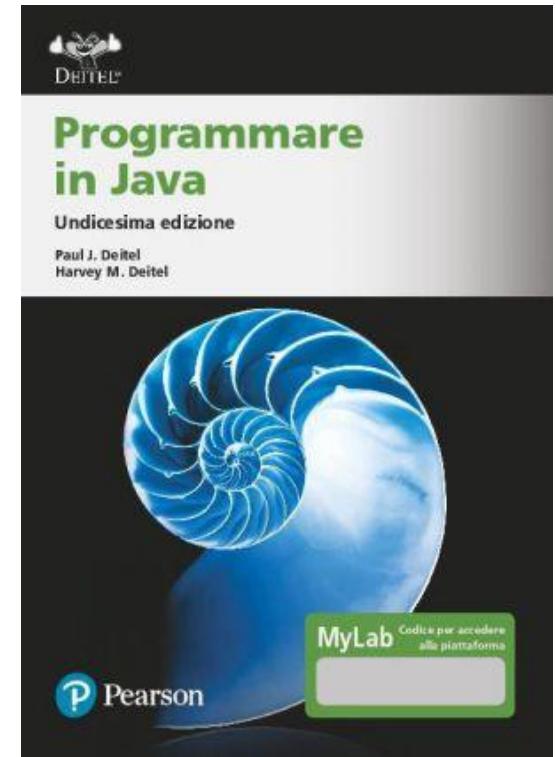


# LINGUAGGIO JAVA



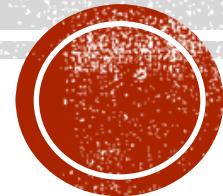
# TESTO DI RIFERIMENTO

- 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



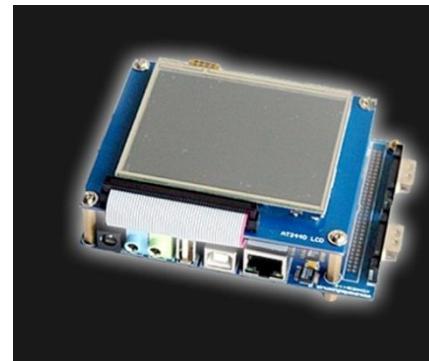
- Mobile



- 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).



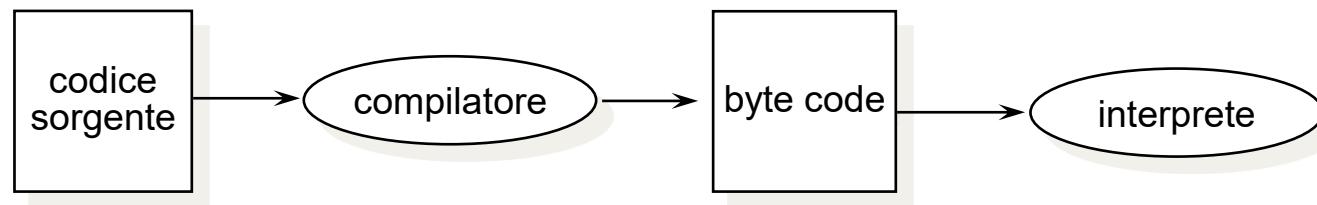
# JAVA: SEMPLICE E O.O.

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



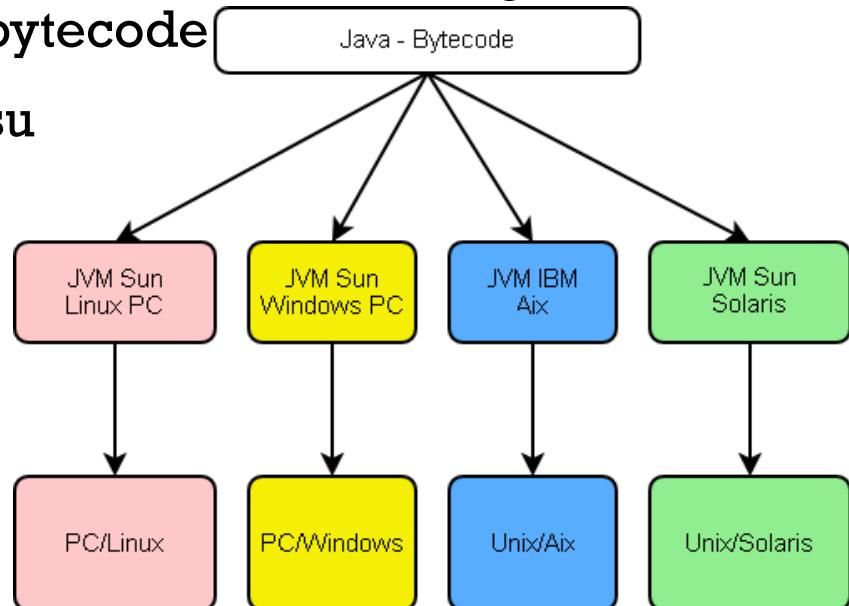
# JAVA È COMPILATI E INTERPRETATO

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



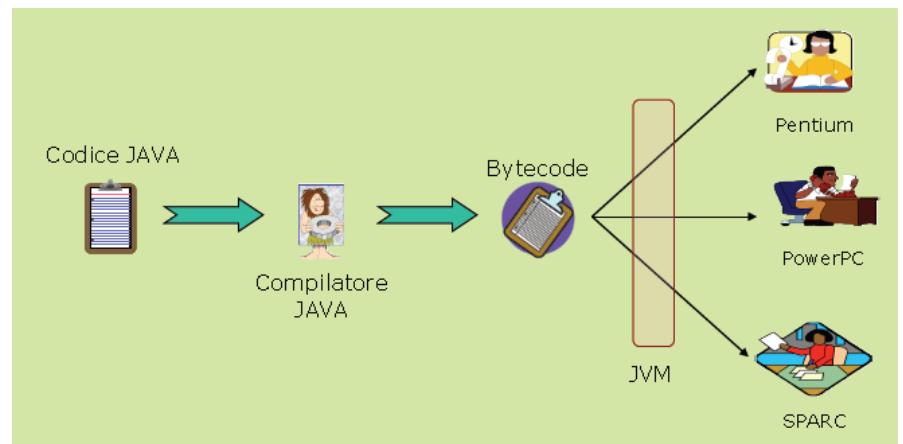
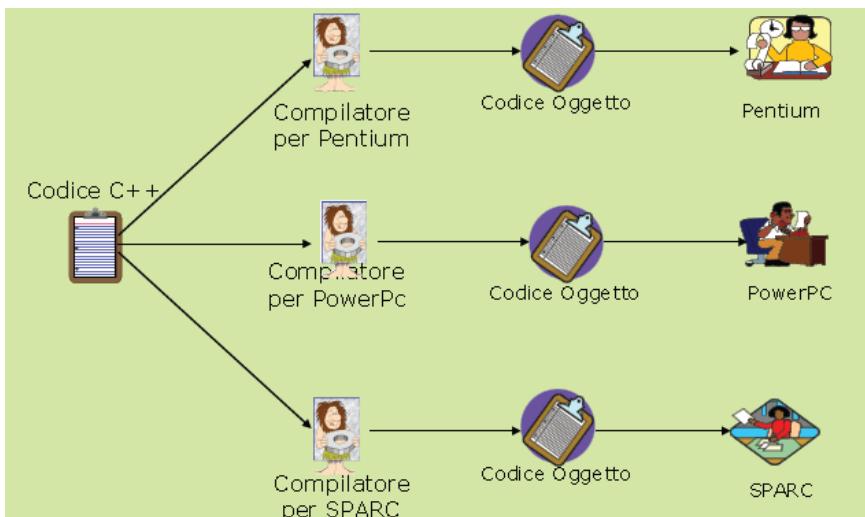
# LA JAVA VIRTUAL MACHINE (JVM)

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



# PORTABILITÀ: C++ VS JAVA

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



# CARATTERISTICHE DI JAVA ...

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



# ... CARATTERISTICHE DI JAVA

## ■ Sicuro

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

## ■ Prestazioni

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

## ■ Esistenza di potenti librerie standard

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

## ■ Multithreaded

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



# INSTALLAZIONE DI JAVA

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



- Per cercare di uniformarci, e tenendo conto che abbiamo bisogno di un intero ambiente di sviluppo (jdk), utilizzeremo come riferimento Java 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



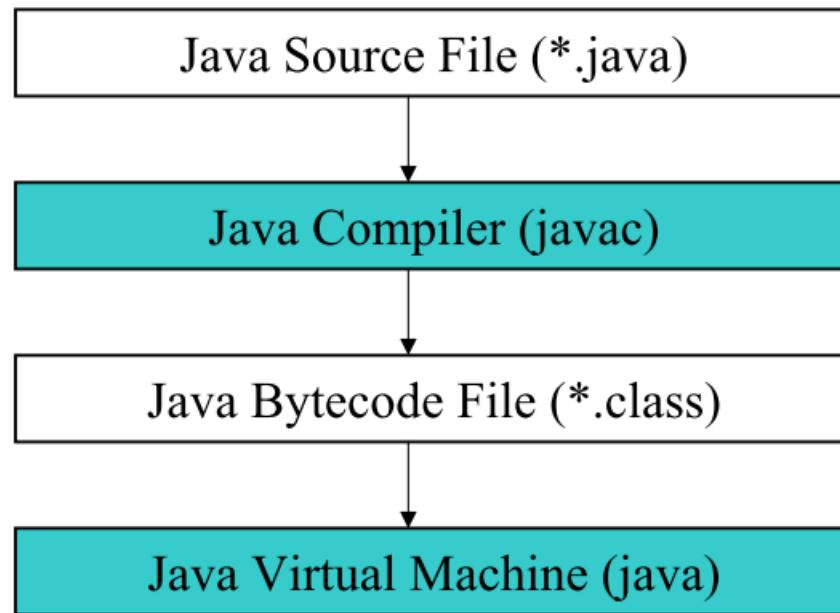
# **STRUTTURA DI UN SEMPLICE PROGRAMMA JAVA: HELLO, WORLD**

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

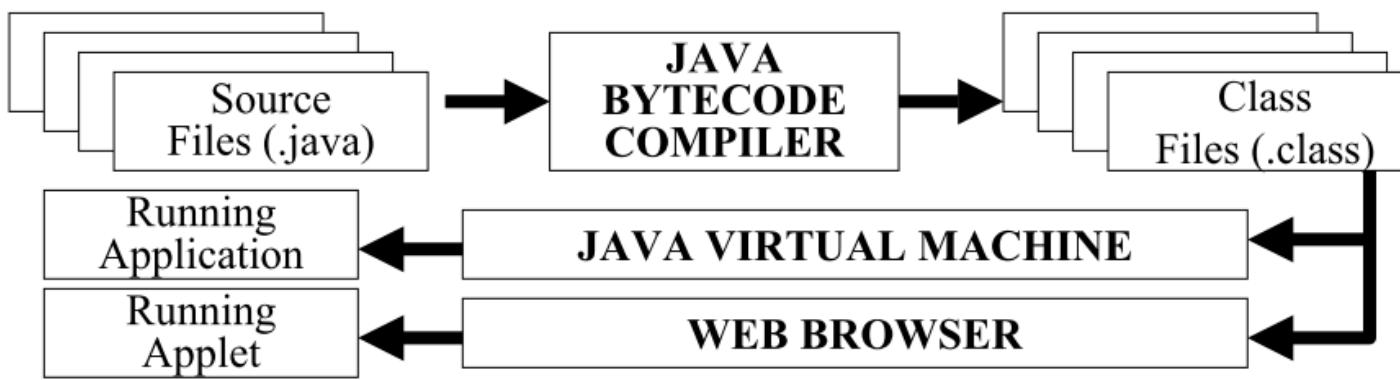
- Tutto deve trovarsi in una classe.
- Il codice sorgente è salvato in un file Hello.java.
- La classe Hello sembra non ereditare da nessun'altra (in realtà eredita come default da Object)



# 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

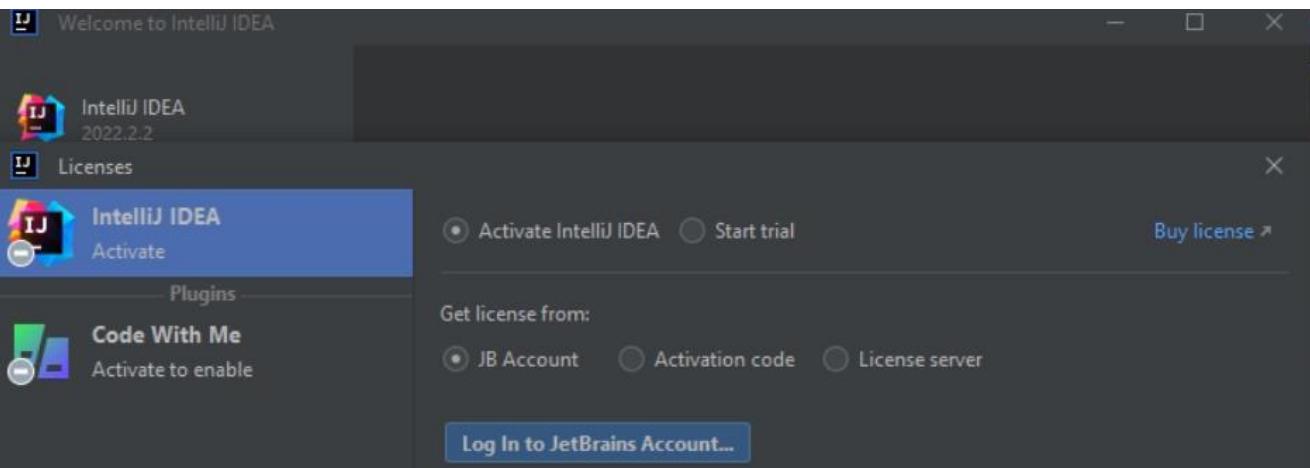
.exe

Free 30-day trial available



# INSTALLAZIONE DI INTELLIJ IDEA

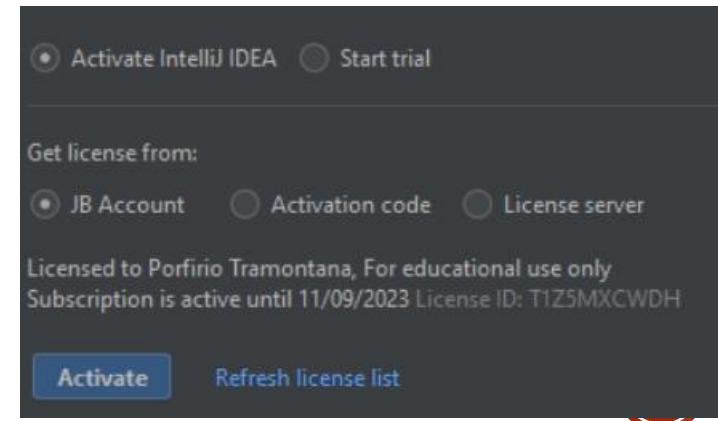
- Per richiedere una licenza gratuita:
  - <https://www.jetbrains.com/community/education/#students>
- Per attivare la licenza in JetBrains scegliere come da figura ed eseguire il Login con il JetBrains Account (creato seguendo le istruzioni ricevute via mail nella richiesta della licenza), poi Activate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



## JetBrains Products for Learning

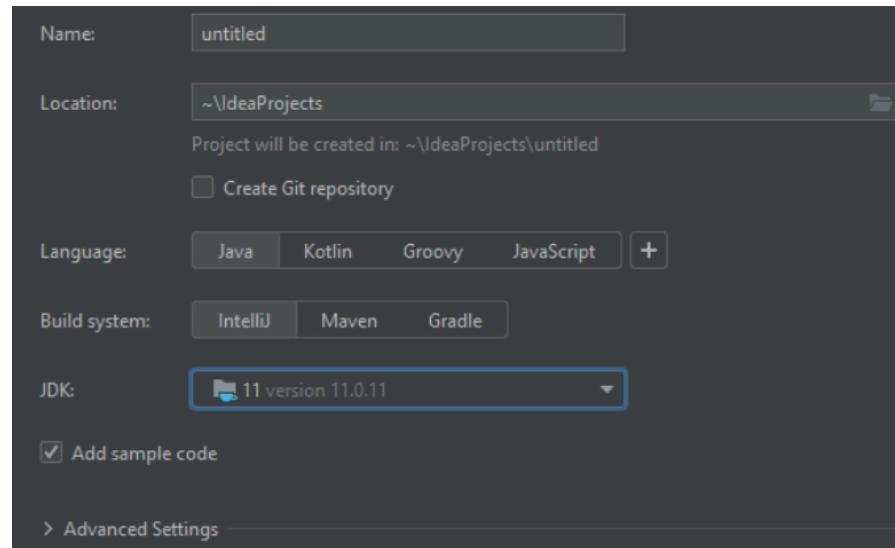
Before you apply, please read the [Educational Subscription Terms and FAQ](#).

This is a screenshot of a web-based application titled "JetBrains Products for Learning". It has a header with tabs: "University email address" (which is selected and underlined in blue), "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".



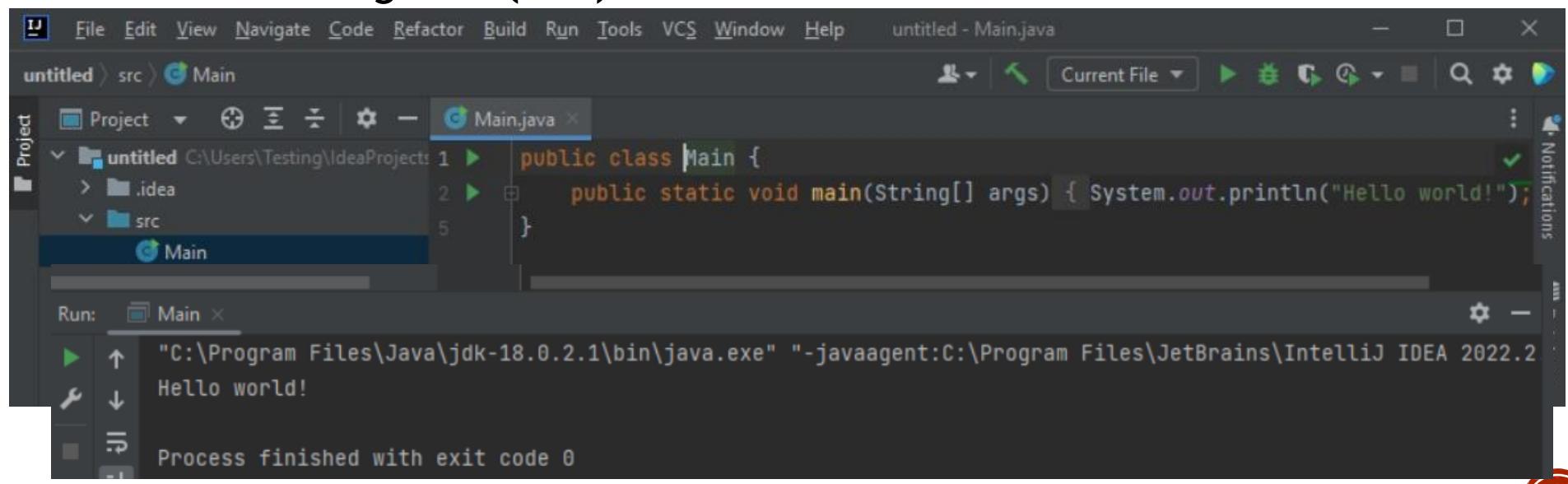
# CREAZIONE DEL PRIMO PROGETTO CON INTELLIJ IDEA

- 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, Help, and untitled - Main.java. The title bar shows the current file path: untitled > src > Main. The Project tool window on the left shows a structure with a .idea folder, a src folder containing a Main.java file, and a Main class under it. The main editor window 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". The output pane shows the result: "Hello world!" followed by "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				
<b>abstract</b>	<b>boolean</b>	<b>break</b>	<b>byte</b>	<b>case</b>
<b>catch</b>	<b>char</b>	<b>class</b>	<b>continue</b>	<b>default</b>
<b>do</b>	<b>double</b>	<b>else</b>	<b>extends</b>	<b>false</b>
<b>final</b>	<b>finally</b>	<b>float</b>	<b>for</b>	<b>if</b>
<b>implements</b>	<b>import</b>	<b>instanceof</b>	<b>int</b>	<b>interface</b>
<b>long</b>	<b>native</b>	<b>new</b>	<b>null</b>	<b>package</b>
<b>private</b>	<b>protected</b>	<b>public</b>	<b>return</b>	<b>short</b>
<b>static</b>	<b>super</b>	<b>switch</b>	<b>synchronized</b>	<b>this</b>
<b>throw</b>	<b>throws</b>	<b>transient</b>	<b>true</b>	<b>try</b>
<b>void</b>	<b>volatile</b>	<b>while</b>		
<i>Keywords that are reserved, but not used, by Java</i>				
<b>const</b>	<b>goto</b>			

# 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

- Vogliamo realizzare come primo esempio una classe che possa funzionare indipendentemente da altre
- Consideriamo una classe Contatore che abbia il compito di contare da 0 fino ad un valore massimo

```
public class Contatore{  
}
```



# ATTRIBUTI

- Gli **attributi** di una classe sono analoghi ai campi di una struct
- 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



# ESEMPI DI ATTRIBUTI

```
public class Contatore {  
    public int value;  
    public final int max=10;  
}
```

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



# 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

- I metodi somigliano alle funzioni di C 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)

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



# METODI

- 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



# METODI: DIFFERENZE CON C

- In Java la *dichiarazione* di un metodo e la sua *definizione* devono avvenire contestualmente, nella stessa istruzione
  - Non è possibili dichiarare un metodo a inizio codice (oppure in un file .h) e poi definirlo più avanti
- Siccome i metodi vengono definiti all'interno delle classi, tra i loro parametri in generale NON c'è l'oggetto a cui si riferiscono
  - Come vedremo, i metodi possono essere chiamati con la notazione oggetto.metodo(), per cui l'oggetto su cui il metodo si applica non è passato come gli altri parametri indicati tra le parentesi ma è prefisso
  - I metodi possono utilizzare liberamente tutti gli attributi della classe senza necessità di passarli come parametri



# ANALOGIE CON C

- 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



# CLASSI, OGGETTI E RIFERIMENTI

- Per poter utilizzare una classe è necessario **istanziare** un suo **oggetto**
- Di ogni classe possiamo istanziare tutti gli oggetti che vogliamo, anche contemporaneamente
- Per utilizzare un oggetto dobbiamo prima di tutto dichiarare un suo **riferimento**
- Un riferimento si dichiara come se fosse una qualsiasi variabile

Contatore c;

- **Attenzione:** c non è un oggetto (e non esiste ancora alcun oggetto) ma è solo un *riferimento* che in futuro potrà puntare ad un oggetto



# COSTRUTTORE

- Per poter creare (**costruire**) un oggetto è necessario utilizzare un metodo specifico della classe, chiamato **costruttore**
  - Ogni classe può avere uno o più costruttori.
    - Se ci sono diversi costruttori, allora devono avere diversi insiemi di parametri, per distinguerli
- È un “metodo”, che ha come nome (obbligato) il nome della classe stessa e può essere invocato solo tramite la parola chiave **new** ritornando come risultato un riferimento all’oggetto istanziato.

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

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



# COSTRUTTORE: ESEMPIO

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

- **new** crea un nuovo oggetto della classe contatore, lo alloca in memoria e restituisce un **riferimento** a questo nuovo oggetto
  - Attenzione: l'oggetto di per sè NON ha un suo nome
- **=** assegna a **c** il valore del riferimento all'oggetto contatore appena creato e restituito da **new**
- Da ora in poi potremo operare sull'oggetto creato (ad esempio incrementando o decrementando il valore) tramite il suo riferimento **c**
- La chiamata del costruttore con parametro intero 2 comporta l'esecuzione del seguente costruttore :  

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



# 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 attributi, 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
- Ogni classe dovrebbe avere sempre almeno un costruttore
  - In teoria potrebbe non essere scritto esplicitamente ma è buona norma che ci sia sempre almeno un costruttore



# 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

```
Contatore c;  
c = new Contatore();  
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--};  
    public boolean isMaximum(){return value==max;};  
}
```

- 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);  
        c2.decrementa();  
        System.out.println(c2.value);  
    };  
}
```



# ALLOCAZIONE : HEAP E STACK

- Le variabili scalari sono sempre 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
- 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



# DICHIARAZIONI E DEFINIZIONI

- Uno scalare può essere allocato tramite dichiarazione e definito tramite definizione:
  - int value; //dichiarazione
  - value =0; //definizione
  - int max=10; //dichiarazione con definizione contestuale
- Un oggetto invece può essere dichiarato solo tramite la dichiarazione di un riferimento ad esso :
  - Contatore c2; //dichiarazione di un riferimento
- 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(); //dichiarazione con definizione contestuale
  - c= **new** Contatore(); //definizione di un riferimento



# ALIAS

- 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 l'oggetto riferito da `c` avremo modificato anche l'oggetto riferito da `c2` (è lo stesso oggetto!)



# NULL

- 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 DEGLI SCALARI

- 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
- Lo scope di una variabile scalare è il blocco {} all'interno del quale esso è stato dichiarato
- I riferimenti si comportano esattamente come le variabili scalari
  - Così come i puntatori in C
  - I riferimenti sono tecnicamente un tipo di variabile scalare a differenza degli oggetti cui essi si riferiscono



# DEALLOCAZIONE DEGLI OGGETTI

- 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)

- Gli oggetti sono nella memoria **heap**
  - In alcuni linguaggi andavano distrutti da programma con una istruzione specifica (*delete*)
  - In Java invece sono distrutti automaticamente da un componente della JVM che si chiama Garbage Collector
- Il **garbage collector** che ripulisce la memoria eliminando tutti gli oggetti per i quali non esistano più riferimenti attivi.
  - Un riferimento smette di puntare ad un oggetto quando viene cambiato il suo valore, ad esempio quando diventa **null**
  - Vengono eliminati tutti gli oggetti che non sono riferiti da alcun riferimento
- Il **garbage collector** si attiva automaticamente in maniera periodica, indipendentemente dal programma
- Il **garbage collector** lavora in maniera ricorsiva: l'eliminazione di oggetti che hanno come attributi dei riferimenti ad altri oggetti può scatenare l'eliminazione di altri oggetti



# PUNTATORI VS RIFERIMENTI

- I riferimenti sono 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 .



# RIASSUMENDO . . .

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



# RIASSUMENDO . . .

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





## Common Programming Error 2.2

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





## Common Programming Error 2.4

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





## Error-Prevention Tip 2.2

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





## Good Programming Practice 2.3

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





## Good Programming Practice 2.8

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





## Good Programming Practice 2.9

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

2.9





## Good Programming Practice 2.4

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





## Good Programming Practice 2.5

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





## Good Programming Practice 2.7

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





## Common Programming Error 2.3

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





## Error-Prevention Tip 2.1

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





## Good Programming Practice 2.6

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



# TIPI SCALARI (PRIMITIVI)

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



# TIPI PRIMITIVI

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



# JAVA È STRONGLY TYPED

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



# OPERATORI ARITMETICI

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

**Fig. 2.11** | Arithmetic operators.



# ALCUNI OPERATORI DI ASSEGNAZIONE

- Si applicano **solo** ai tipi scalari (
  - 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>
<b>Arithmetic assignment operators.</b>			



# 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`
  - Le costanti dovrebbero sempre avere un nome che inizia con una lettera maiuscola



# 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 SCALARI

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

int c=20;  
(a==c) vale false



# OPERATORI DI CONFRONTO TRA RIFERIMENTI E TRA OGGETTI

```
String a=new String("pippo"); String b=new String("pippo");
```

## ■ Confronto tra riferimenti

`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

I riferimenti si comportano, rispetto al confronto, come degli scalari

## ■ Confronto tra oggetti

`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 nel contenuto



# OPERATORI DI CONFRONTO

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

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



# OPERATORI (E PRECEDENZE)

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

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

# OPERATORI LOGICI E PRECEDENZE

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



## Good Programming Practice 2.11

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





## Error-Prevention Tip 2.4

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





## Common Programming Error 2.7

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





## Error-Prevention Tip 2.5

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



# LOOPING CONSTRUCTS

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

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

---

----

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

---

----

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



# PASSAGGIO DEI PARAMETRI: TIPI PRIMITIVI

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



# PASSAGGIO DEI PARAMETRI: OGGETTI

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



# OSSERVAZIONE

- In realtà in Java esiste **una sola** modalità di passaggio, quella per **valore**
  - Nel caso dei tipi primitivi viene passato il valore dello scalare
  - Nel caso degli oggetti viene passato il valore del riferimento



# **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 (omogenei)
- Le variabili contenute nell'array possono essere:
  - Tipi primitivi (scalari)
  - Riferimenti ad oggetti
- L'array può essere acceduto per indice, con l'indice che assume valori  $\geq 0$



# DICHIARAZIONE DI UN ARRAY

- `int[] c=new int[12];`
- Oppure
  - `int[] c; //dichiarazione di un riferimento ad array di interi`
  - `c = new int[12] //allocazione di un oggetto array di 12 interi e assegnazione a c del riferimento`
- Il numero degli elementi non viene mai specificato 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 primitivo (`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`)





## 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.





## 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*



# APPROFONDIMENTO: METODI STATIC

- Esistono delle classi con metodi *static*
- I metodi static si comportano tecnicamente come delle funzioni C
  - Pur essendo all'interno di una classe possono essere chiamati senza bisogno di istanziare oggetti
  - Vengono chiamati scrivendo  
*NomeDellaClasse.nomeDelMetodo(parametri)*
- Ad esempio:
  - *Arrays.sort(array); // Arrays è una classe con il metodo static sort*
  - *Arrays.fill(array,value); // fill è un metodo static che riempie array di value*





## 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;
  - 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
    - E' 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



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

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

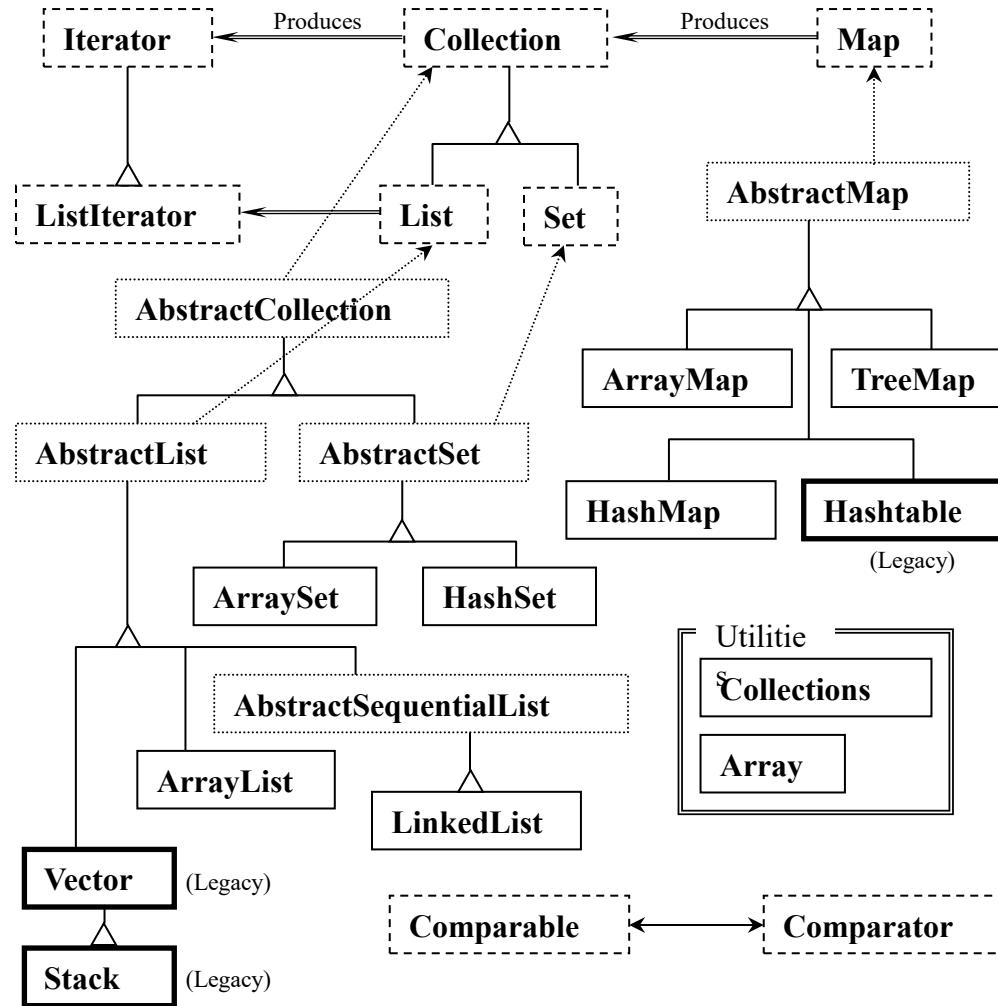


# CENNO AGLI ALTRI CONTAINER

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



# CLASSI CONTENITORE: LA GERARCHIA





## Good Programming Practice 16.1

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

16.1



# ARRAYLIST

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



# GENERICHE E TEMPLATE

- Una classe *generica* o *template* è una classe che ha tra i suoi parametri anche un indicatore di un'altra classe
- Ad esempio:
  - `ArrayList<String>`
- È una classe `ArrayList` sottoposta al vincolo che tutti i suoi elementi devono essere riferimenti ad oggetti della classe `String`
  - Tecnicamente un `ArrayList` senza altre indicazione deve essere visto come un `ArrayList` che può contenere riferimenti ad **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 dalla classe indicata 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()  
        );  
  
        int ultimo=lista.size()-1;  
  
        System.out.println(lista.get(ultimo).toStr  
        ing());  
    }  
}
```

```
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));  
        int 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.



# 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]; }
```



# **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



# **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.



# **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();  
}
```

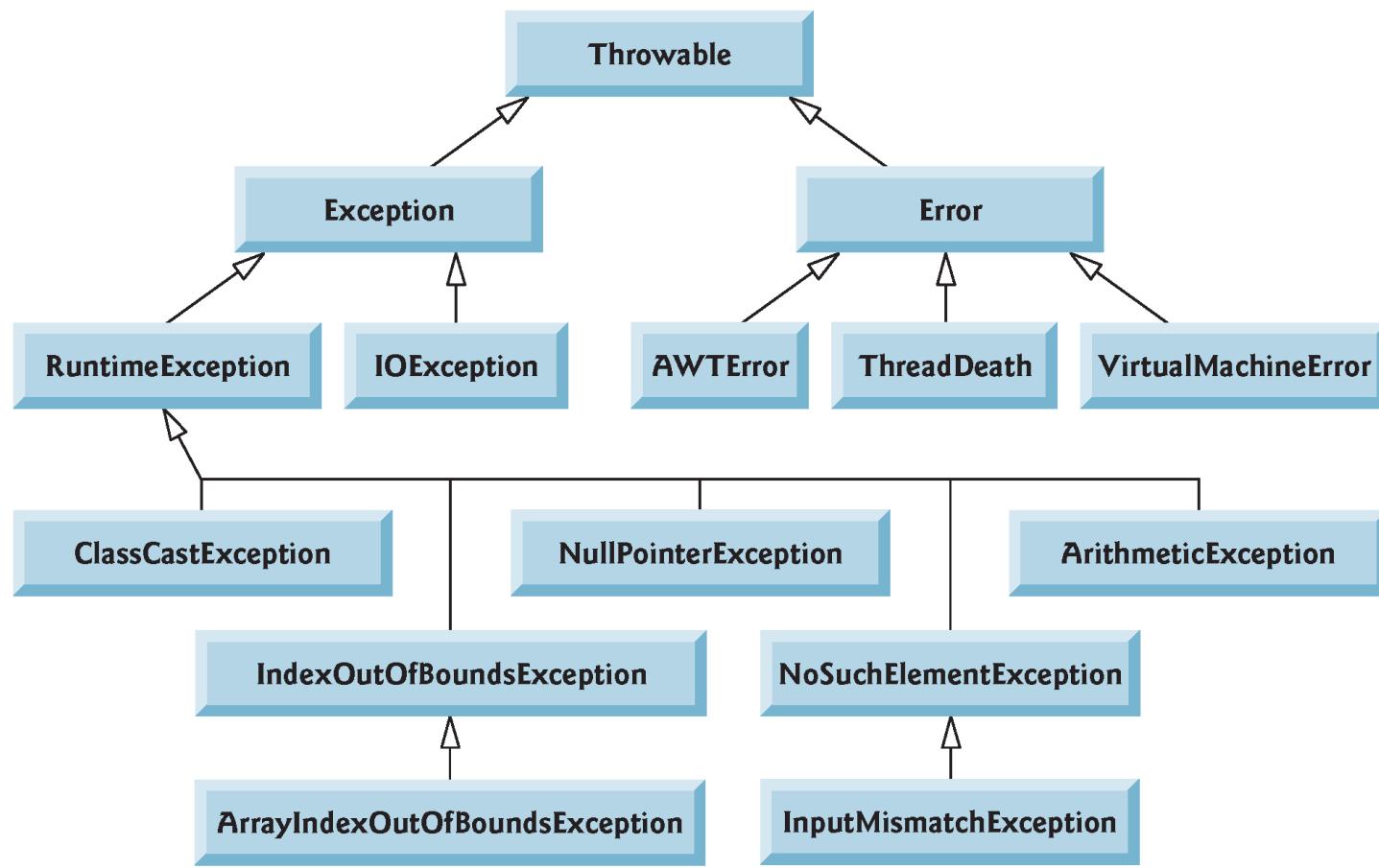
In Esaminati dovremmo gestire un'eccezione ma ci limitiamo a rilanciarla:

```
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*



# **INPUT / OUTPUT**

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

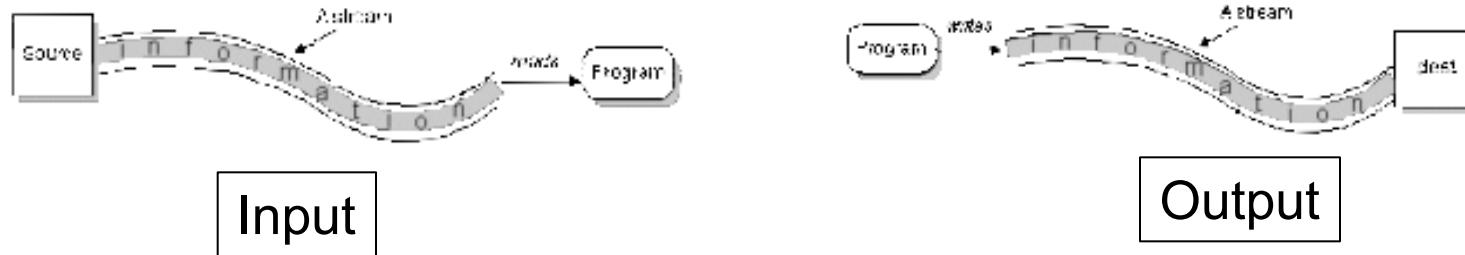


# INTERFACCIE UTENTE

- 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



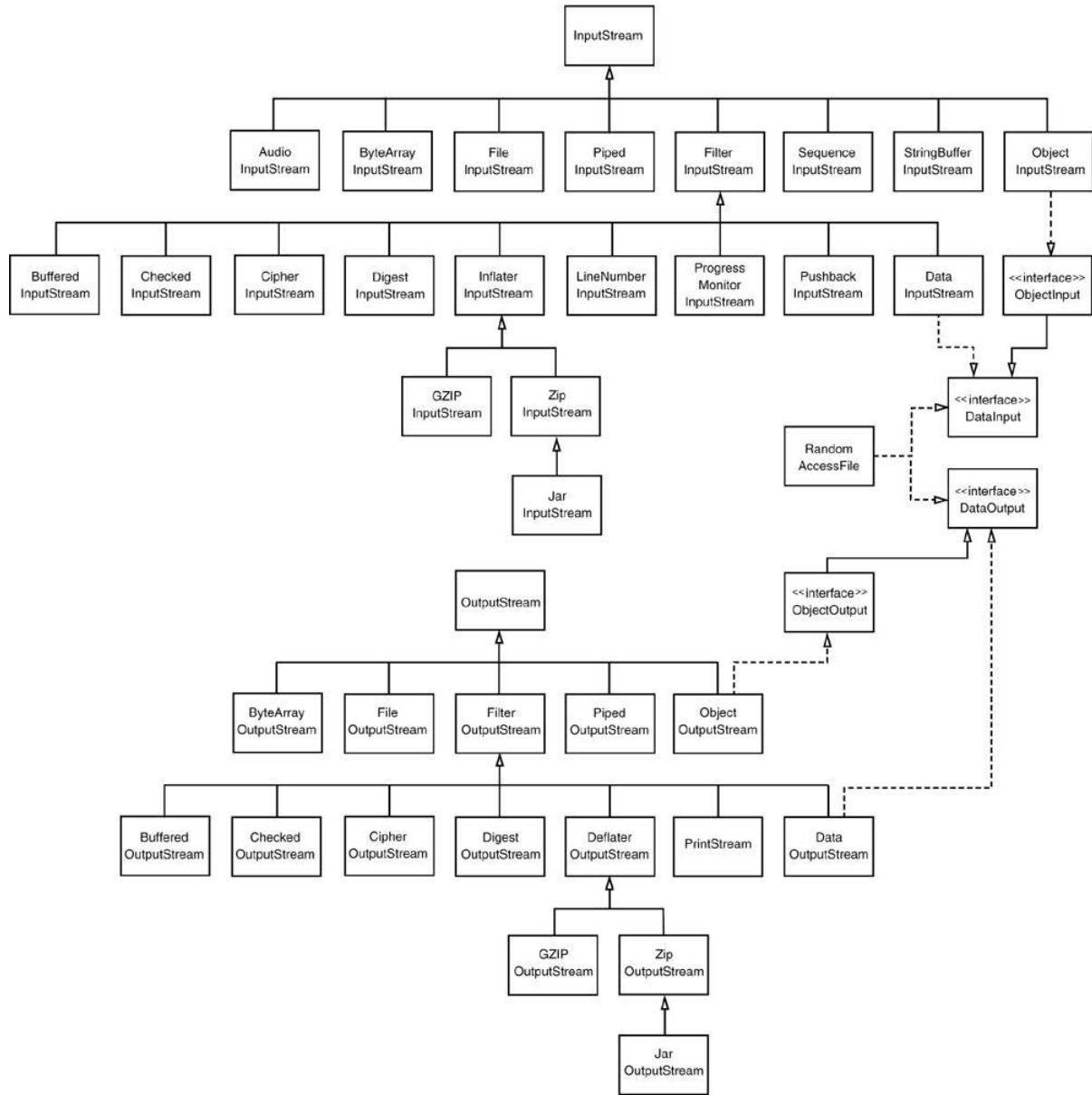
# ALTRÉ CLASSI PER L'INPUT E L'OUTPUT

- 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

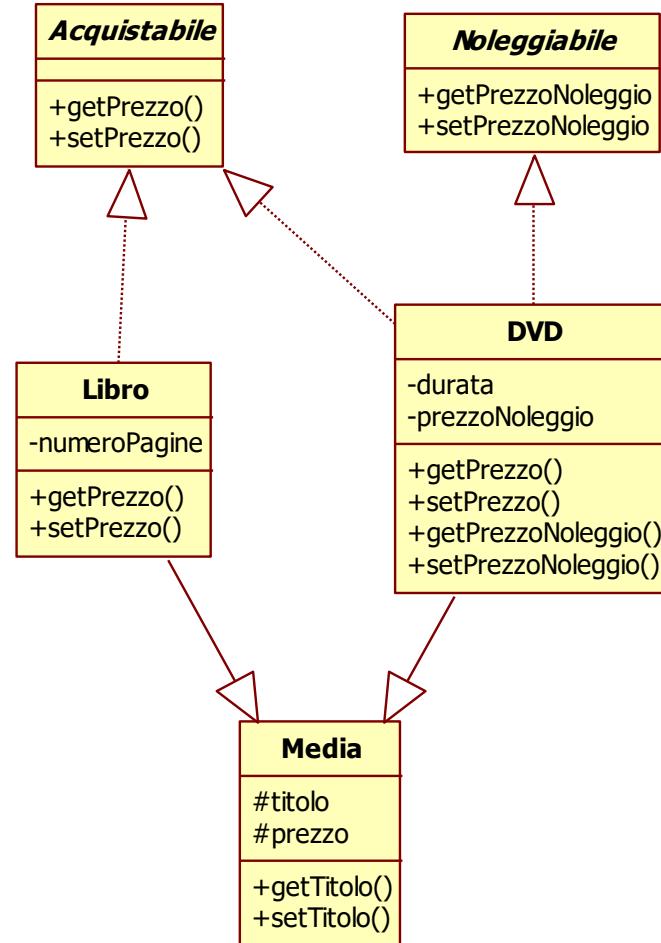


# INTERFACCE E IMPLEMENTAZIONE

- 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



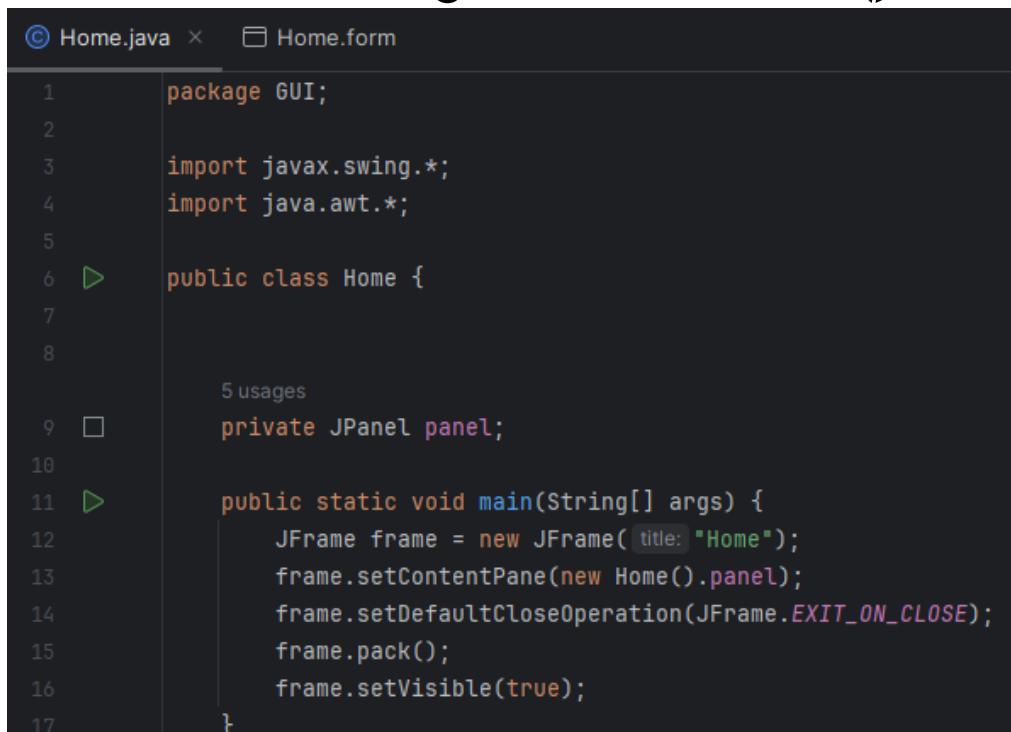
# CREAZIONE DI UNA FORM GRAFICA

- 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

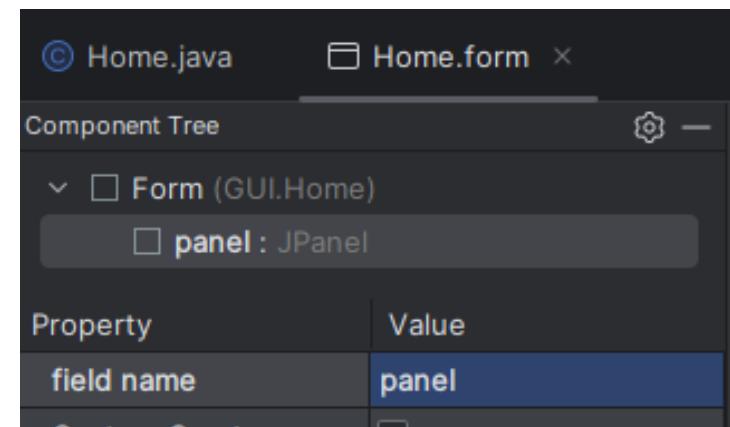


# CREAZIONE DELLA PRIMA FORM

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

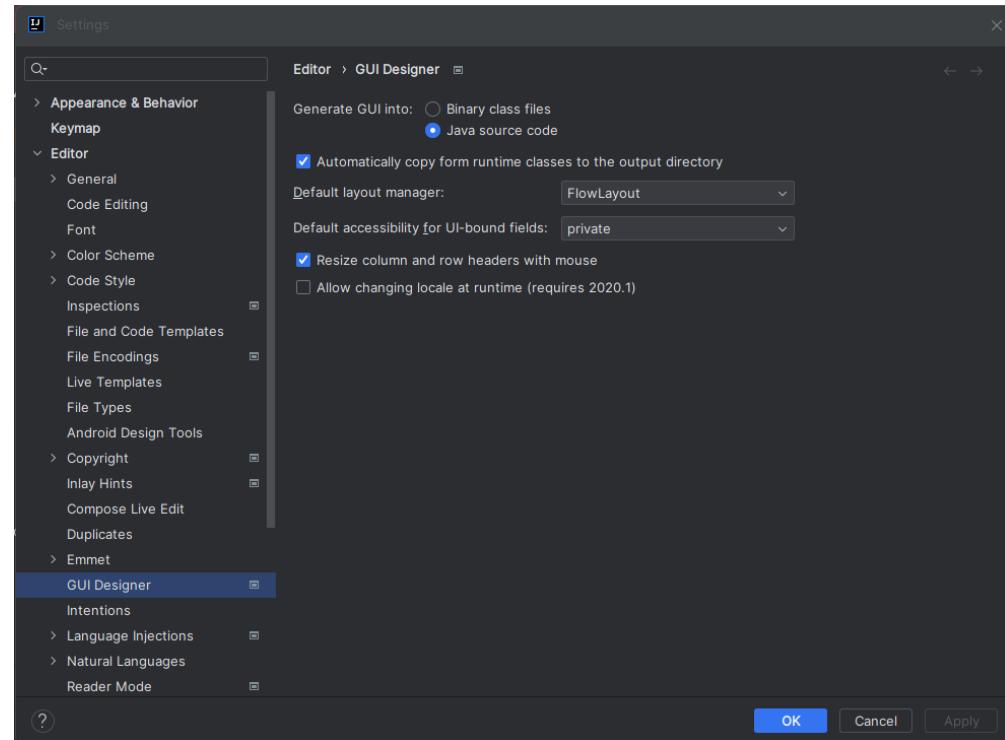


```
© Home.java ×  □ Home.form  
1 package GUI;  
2  
3 import javax.swing.*;  
4 import java.awt.*;  
5  
6 ▷ public class Home {  
7  
8  
9     □ 5 usages  
10    □ private JPanel panel;  
11  
12 ▷     public static void main(String[] args) {  
13         JFrame frame = new JFrame("Home");  
14         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)*

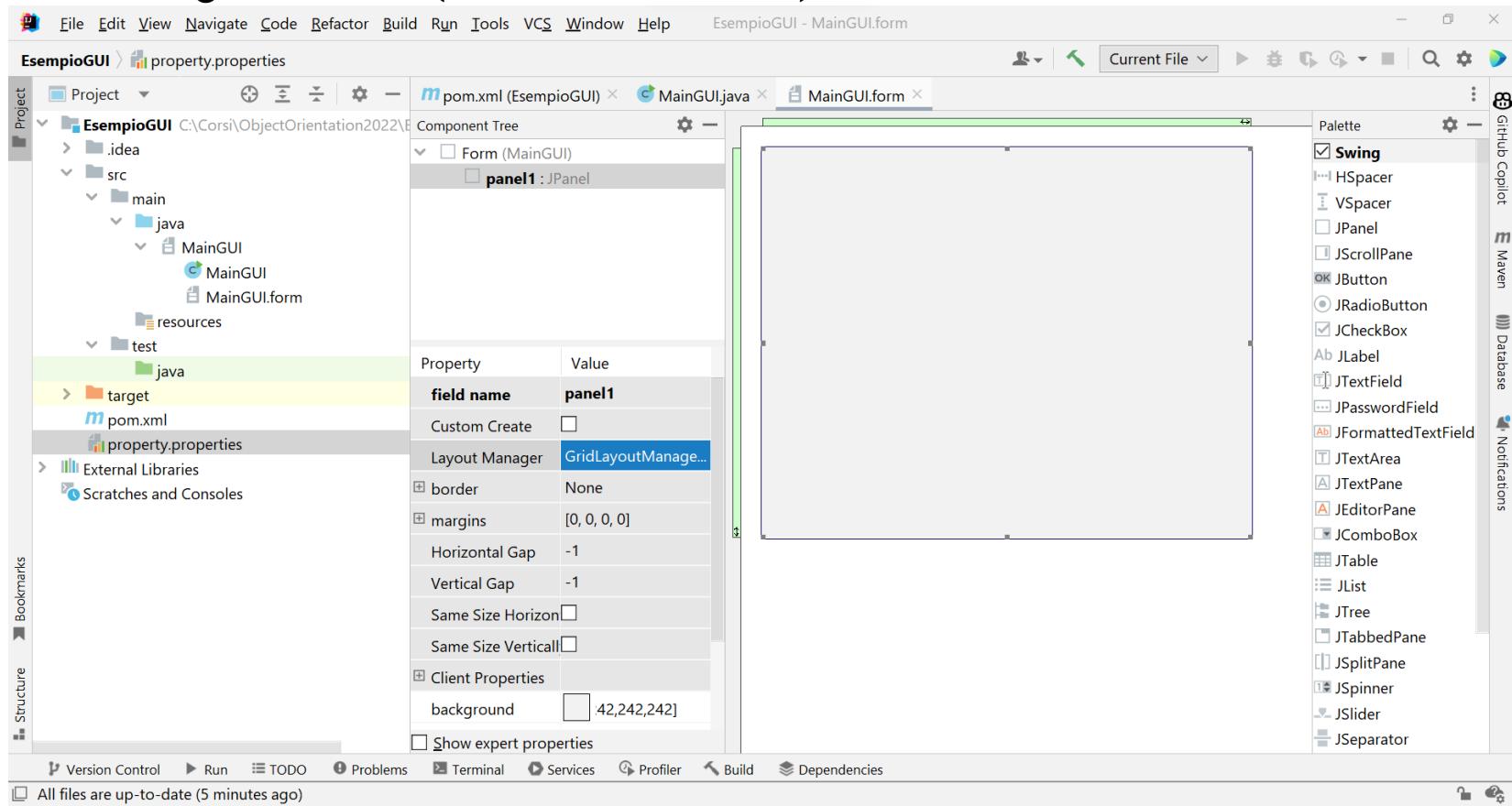
```
public class MainGUI {  
    private JPanel panel1;  
  
    public static void main(String[] args) {  
        JFrame frame = new JFrame("MainGUI");  
        frame.setContentPane(new MainGUI().panel1);  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        frame.pack();  
  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

Se vogliamo aggiungere comportamento al frame, scriviamo nel suo costruttore



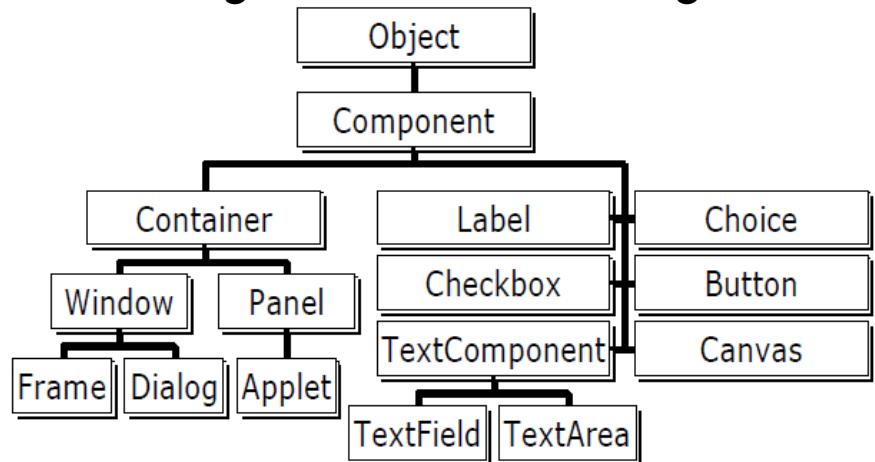
# FINESTRA DI DESIGN

- 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 graficamente e logicamente elementi grafici
    - 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 perché 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:

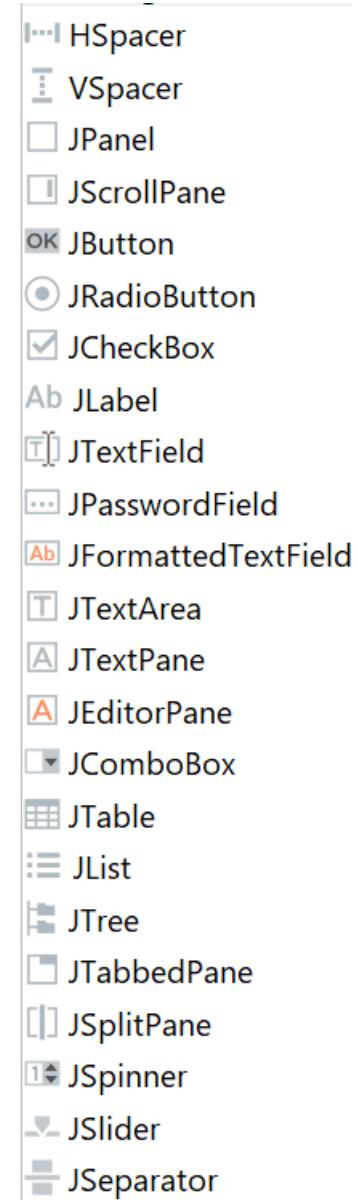
- Component Tree:** A tree view of the application's structure. It starts with "Form (MainGUI)", which contains "panel1 : JPanel". "panel1" contains a "JPanel" which holds "button1 : JButton". Below "button1", another "JPanel" is shown, which contains "button2 : JButton".
- Properties Table:** A table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

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



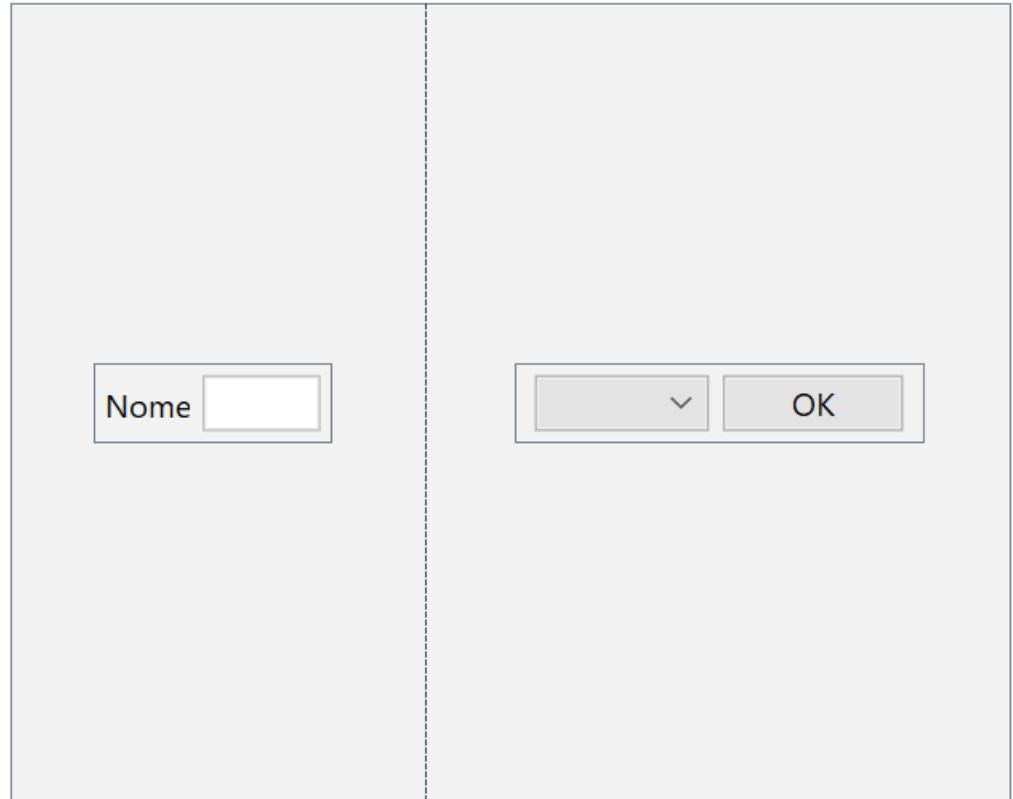
# ESEMPI DI WIDGET

- **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>



# 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 differente
  - 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 add ActionListener
- 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

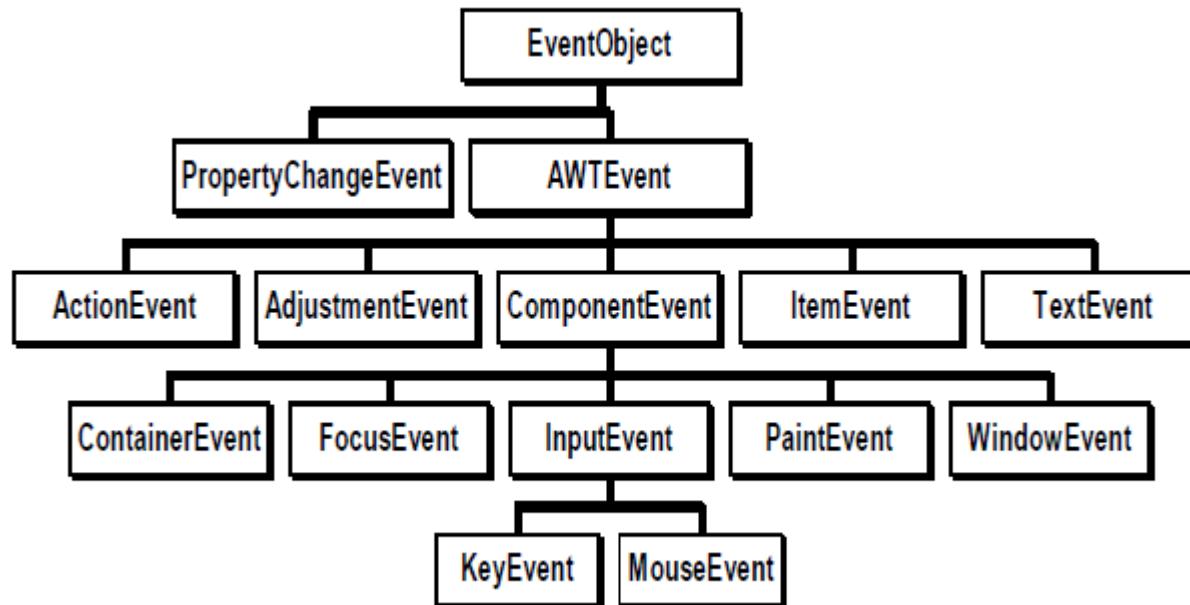
```
btnOK.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        JOptionPane.showMessageDialog(frame,"Hello, World!");  
    }  
});
```

- 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>



# 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:
  - Prerequisito: abbiamo bisogno di rendere statico il nostro frame di partenza
    - `private static JFrame frameChiamante; //la home deve poter essere sempre raggiungibile`
  - 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
    - `GUIChiamata guiChiamata = new GUIChiamata(frameChiamante);`
      - Fondamentale passare il riferimento al frameChiamante, altrimenti non sarà possibile tornare indietro
    - `frameChiamante.setVisible(false);`
    - `guiChiamata.frame.setVisible(true); //andava bene anche un metodo getFrame()`
  - 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)
    - `public JFrame frameChiamato; //in alternativa metodo public getFrame()`
    - `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 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); //non necessario in presenza di dispose()  
        frame.dispose(); //se si chiama una terzaGUI si evita il 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

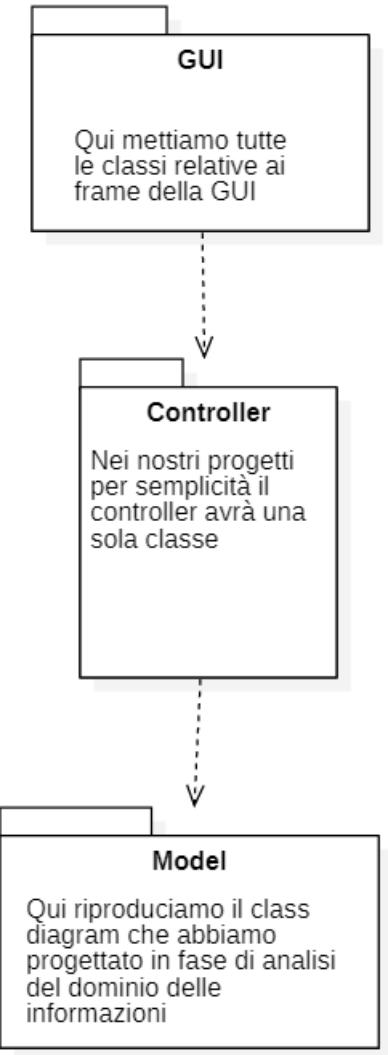
- Lo stile architetturale che adotteremo in questo corso prevede che le comunicazioni tra le finestre riguardino solo il passaggio del *controllo* tra una finestra e l'altra
  - Un JFrame che ne aprirà un altro, quindi, passerà il riferimento a sè stesso in modo da poter essere successivamente restituito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

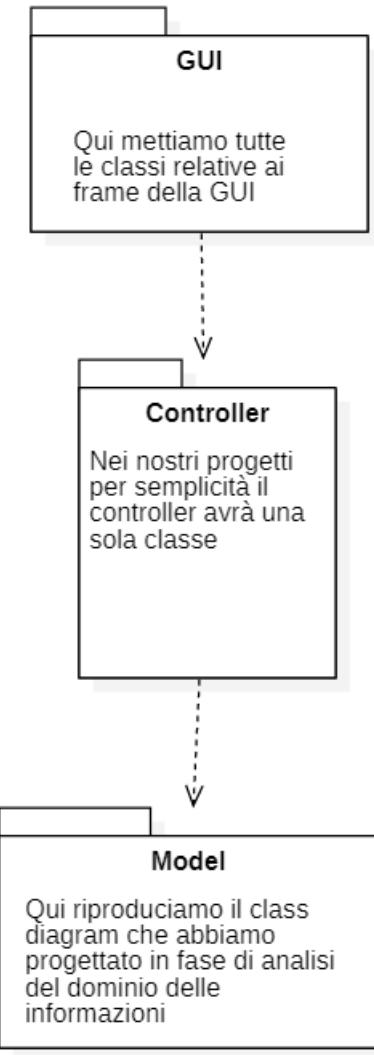
- E' responsabile dell'interfaccia grafica, dell'inserimento degli input, della loro validazione, della visualizzazione degli output
- **NON** realizza elaborazione dei dati
- **NON** accede direttamente alle classi del Model

## ■ 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) {
```



# **RUOLO DEL CONTROLLER**

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



# FILE JAR

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



# RUNNABLE JAR

- 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



# **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 ←  
 * @param name the location of the image, relative to the url argument ←  
 * @return the image at the specified URL ←  
 * @see Image ←  
 */  
  
public Image getImage(URL url, String name) {  
    try {  
        return getImage(new URL(url, name)); ←  
    } 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

Riferimento

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](#)



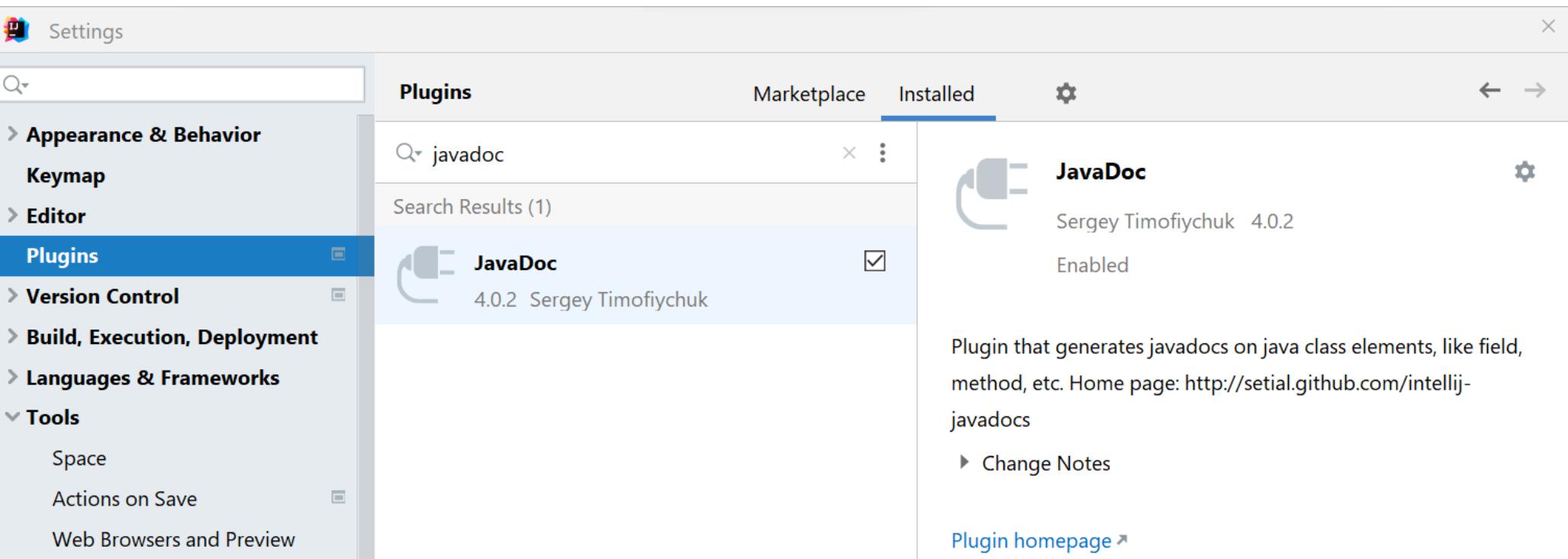
# PRINCIPALI TAG JAVADOC

- **@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>



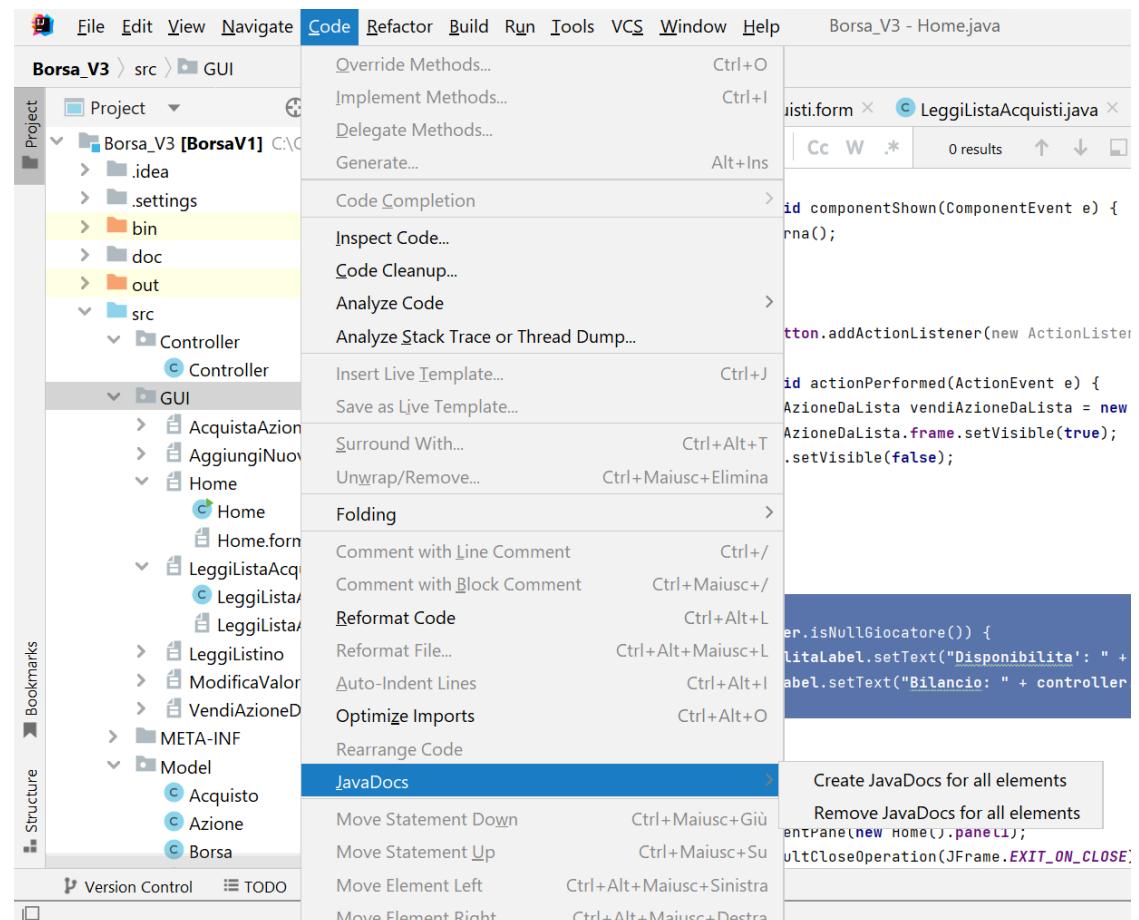
# Generazione automatica di documentazione

- 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 composto di pagine HTML statiche (nella cartella doc) che può essere navigato in locale o pubblicato sul web
  - Anche su github



**Package Controller**

## Class Controller

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

---

```
public class Controller  
extends Object
```

The Class Controller.

### ***Constructor Summary***

**Constructors**

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

### ***Method Summary***

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

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



# DEBUGGING



# DEBUGGING

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



# RICERCA E LOCALIZZAZIONE DEI DIFETTI

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



# ESEMPIO DI SONDA

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

```
public float calcolaCapitale() {  
    float capitale=liquidita;  
    System.out.println(capitale);  
    for (Acquisto a : acquisti) {  
        capitale+=a.getAzione().getSocieta().getValoreAzione()*a.getQuantita();  
    }  
    System.out.println(capitale);  
    return capitale;  
}
```



# ESEMPIO DI ASSEGNAZIONE

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

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

- Problema: l'interruzione improvvisa può lasciare il database in uno stato inconsistente
- Anzichè RuntimeException potremmo utilizzare una eccezione personalizzata che fornisca dati relativi all'asserzione fallita



# AUTOMATIZZAZIONE DEL DEBUGGING

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



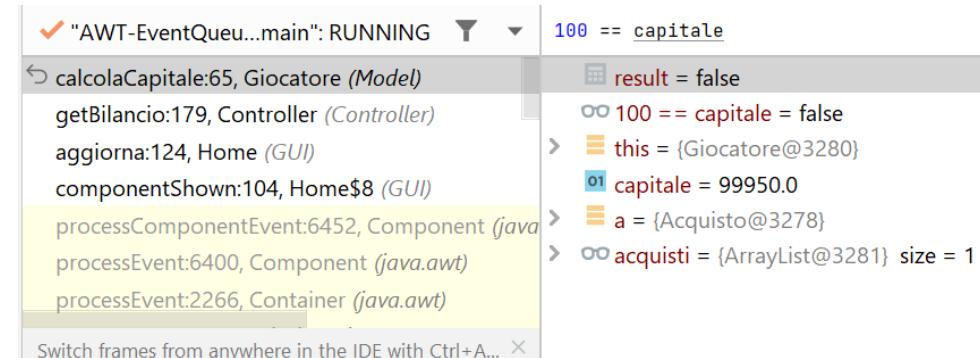
# FUNZIONALITÀ DI DEBUGGING

- **Inserimento break point**
  - Tramite IntelliJ è possibile attivare un breakpoint semplicemente con un click di fianco al numero della riga (si attiva un cerchio rosso che indica la presenza del breakpoint)
  - In aggiunta nel menu di contesto del tasto **destroy** è possibile accedere ad un completo menu con proprietà avanzate dei breakpoint
- **Esecuzione passo passo del codice**
  - Entrando o meno all'interno dei metodi chiamati
  - Uscendo da un metodo verso il chiamante



# ESECUZIONE STEP BY STEP

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

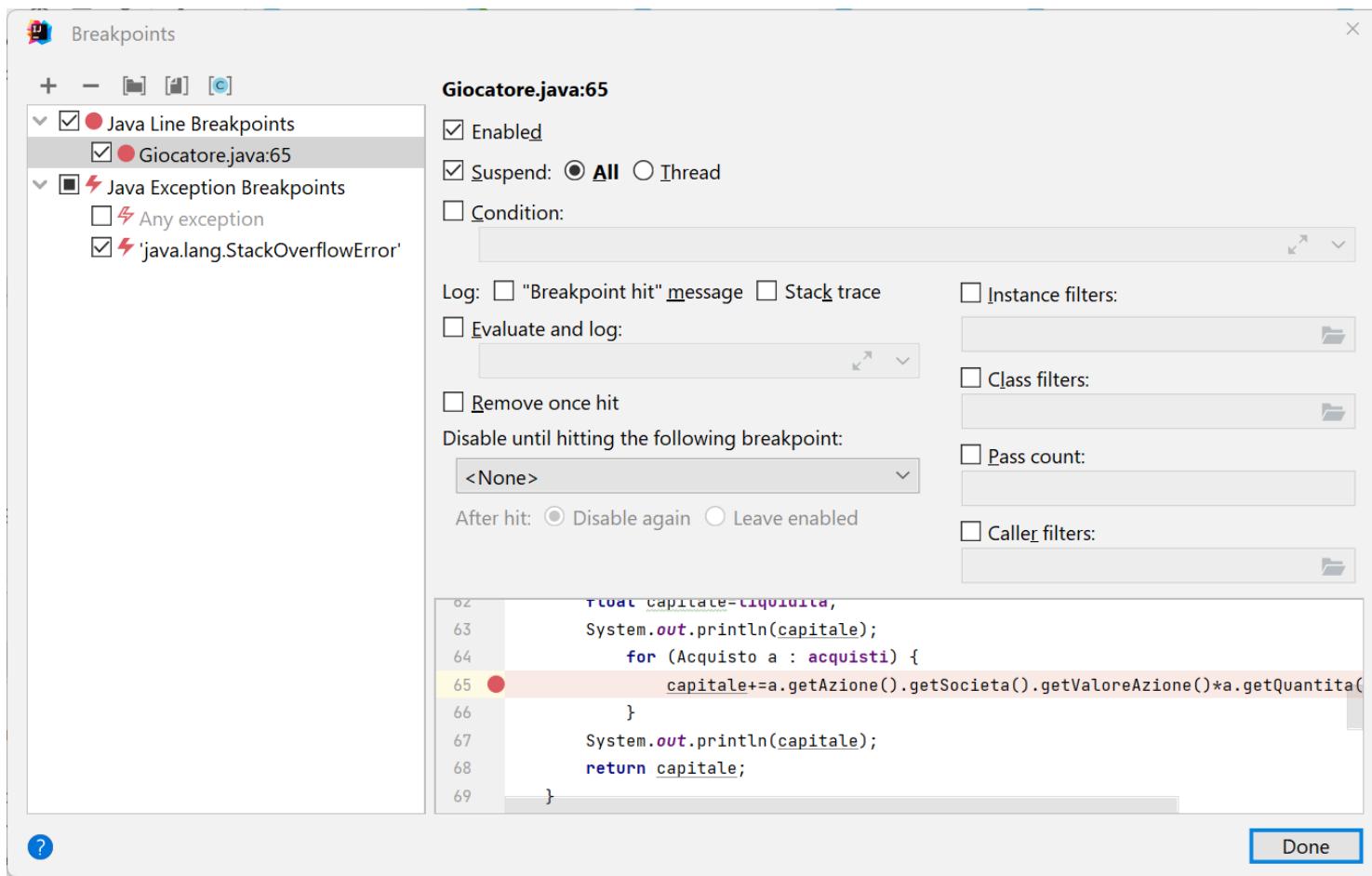


The screenshot shows a Java application running in an IDE. The stack trace indicates the current frame is `calcolaCapitale:65, Giocatore (Model)`. The variable pane on the right shows:

- `100 == capitale` (highlighted in yellow)
- `result = false`
- `100 == capitale = false`
- `this = {Giocatore@3280}`
- `capitale = 99950.0`
- `a = {Acquisto@3278}`
- `acquisti = {ArrayList@3281} size = 1`

At the bottom, a message says: "Switch frames from anywhere in the IDE with Ctrl+A..."

# PROPRIETÀ AVANZATE DEI BREAKPOINT



# PROPRIETÀ AVANZATE DEI BREAKPOINT

- Condition

- Il breakpoint ferma l'esecuzione solo se una certa espressione booleana è soddisfatta
    - Ad esempio se capitale è inferiore a 0

- Log

- Genera un log del passaggio per il breakpoint
    - Ci permette di avere una sonda senza modificare il codice sorgente

- Instance filters / Class filters / Caller filters

- Ulteriori filtri per limitare I casi nei quali il breakpoint si attiva

- Pass count

- Il breakpoint si attiva solo quando viene eseguito un certo numero di volte
    - Ad esempio, se vogliamo fermare un ciclo alla decima esecuzione scriviamo 10

- Exception Breakpoint

- Il breakpoint viene eseguito in seguito ad una eccezione



# WATCHPOINT

- Un watchpoint è un breakpoint collegato non ad una riga ma ad un attributo di una classe
- Si inserisce come un breakpoint, puntando sulla riga di dichiarazione dell'attributo
- Fa sospendere l'esecuzione ogni volta che l'attributo è letto (**Field Access**) o scritto (**Field Modification**)
- Tramite le Properties è possibile personalizzare quando effettuare la sospensione, con le stesse opzioni che ci sono per I breakpoint



# WATCHPOINT

Breakpoints

+ - [ ] [ ] [ ]

Java Field Watchpoints  
   Model.Giocatore.acquisti

Java Exception Breakpoints  
   Any exception  
   'java.lang.StackOverflowError'

**Model.Giocatore.acquisti**

Enabled

Suspend:  All  Thread

Condition:  
  

Log:  "Breakpoint hit" message  Stack trace

Evaluate and log:  
  

Remove once hit

Disable until hitting the following breakpoint:

After hit:  Disable again  Leave enabled

Instance filters:  
    
 Class filters:  
    
 Pass count:  
    
 Caller filters:  
  

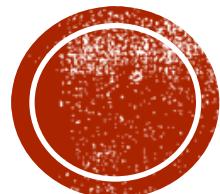
Watch

Field access  
 Field modification

12  
13  private ArrayList<Acquisto> acquisti;  
14

Done

gging



# **ANALISI STATICA E QUALITA' DEL CODICE**



# CODE SMELLS

- **Code smells** sono il risultato di cattive pratiche di programmazione
- Possono causare il deterioramento del codice sotto molti punti di vista, come prestazioni, manutenibilità (facilità di modificarlo), testabilità (capacità di provarne facilmente il funzionamento), flessibilità, estendibilità, ...
- Essi possono essere anche la causa diretta di bug, talvolta
- Code smells possono essere rivelati da **analisi statica**



# CODE SMELLS

- Ci sono molti libri e siti web che descrivono possibili code smells e cattive pratiche di progettazione
  - Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice-Hall
  - Robert C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Prentice-Hall
  - <https://blog.cleancoder.com/>



# MIGLIORARE LA QUALITA' DEL CODICE

- Negli ultimi anni, ci si sta occupando, con ricerche ed esperimenti:
  - Di valutare la tendenza naturale degli studenti a seguire cattive pratiche di programmazione nei loro primi progetti
  - Di valutare se l'utilizzo di strumenti di analisi statica può aiutare gli studenti a migliorare la qualità del loro codice



# BUONE TECNICHE DI PROGRAMMAZIONE

- Il libro di testo, come visto precedentemente, contiene molti consigli, raccomandazioni e buone pratiche di programmazione
- Nel seguito vengono riportate quelle più rilevanti, già discusse





## Good Programming Practice 2.2

Use white space to enhance program readability.





## 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 2.7

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





## 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.





## 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.





## 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.





## 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.





## Good Programming Practice 7.2

Constant variables also are called **named constants**. They often make programs more readable—a named constant such as `ARRAY_LENGTH` clearly indicates its purpose, whereas a literal value such as `10` could have different meanings based on its context.





## Good Programming Practice 7.3

Constants use all uppercase letters by convention and multiword named constants should have each word separated from the next with an underscore (\_) as in `ARRAY_LENGTH`.





## Common Programming Error 7.4

Assigning a value to a previously initialized `final` variable is a compilation error. Similarly, attempting to access the value of a `final` variable before it's initialized results in a compilation error like, “`variableName` might not have been initialized.”





## Good Programming Practice 8.1

By convention, predicate method names begin with `is` rather than `get`.





## Good Programming Practice 10.1

When declaring a method in an interface, choose a method name that describes the method's purpose in a general manner, because the method may be implemented by many unrelated classes.

10.1





## Good Programming Practice 10.2

Use `public` and `abstract` explicitly when declaring interface methods to make your intentions clear. As you'll see in Sections 10.10–10.11, Java SE 8 and Java SE 9 allow other kinds of methods in interfaces.





## Good Programming Practice 11.1

Exception handling removes error-processing code from the main line of a program's code to improve program clarity. Do not place `try...catch... finally` around every statement that may throw an exception. This decreases readability. Rather, place one `try` block around a significant portion of your code, follow the `try` with `catch` blocks that handle each possible exception and follow the `catch` blocks with a single `finally` block (if one is required).





## Good Programming Practice 11.2

Associating each type of serious execution-time malfunction with an appropriately named **Exception** class improves program clarity.





## Good Programming Practice 11.3

By convention, all exception-class names should end with the word `Exception`.





## Good Programming Practice 15.1

When building **Strings** that represent path information, use **File.separator** to obtain the local computer's proper separator character rather than explicitly using / or \. This constant is a **String** consisting of one character—the proper separator for the system.



# STRUMENTI PER L'ANALISI STATICÀ

- Molti strumenti sono disponibili per:
  - Rilevare l'esistenza di cattive pratiche di programmazione
  - Suggerire possibili miglioramenti
- Questi strumenti eseguono **l'analisi statica** del codice
  - Cioè leggono il codice sorgente e possono rilevare **potenziali** problemi
  - Questi strumenti possono essere integrati come plug-in nelle IDE di sviluppo
- Tra gli strumenti disponibili:
  - PMD
  - Checkstyle
  - SonarLint / SonarQube

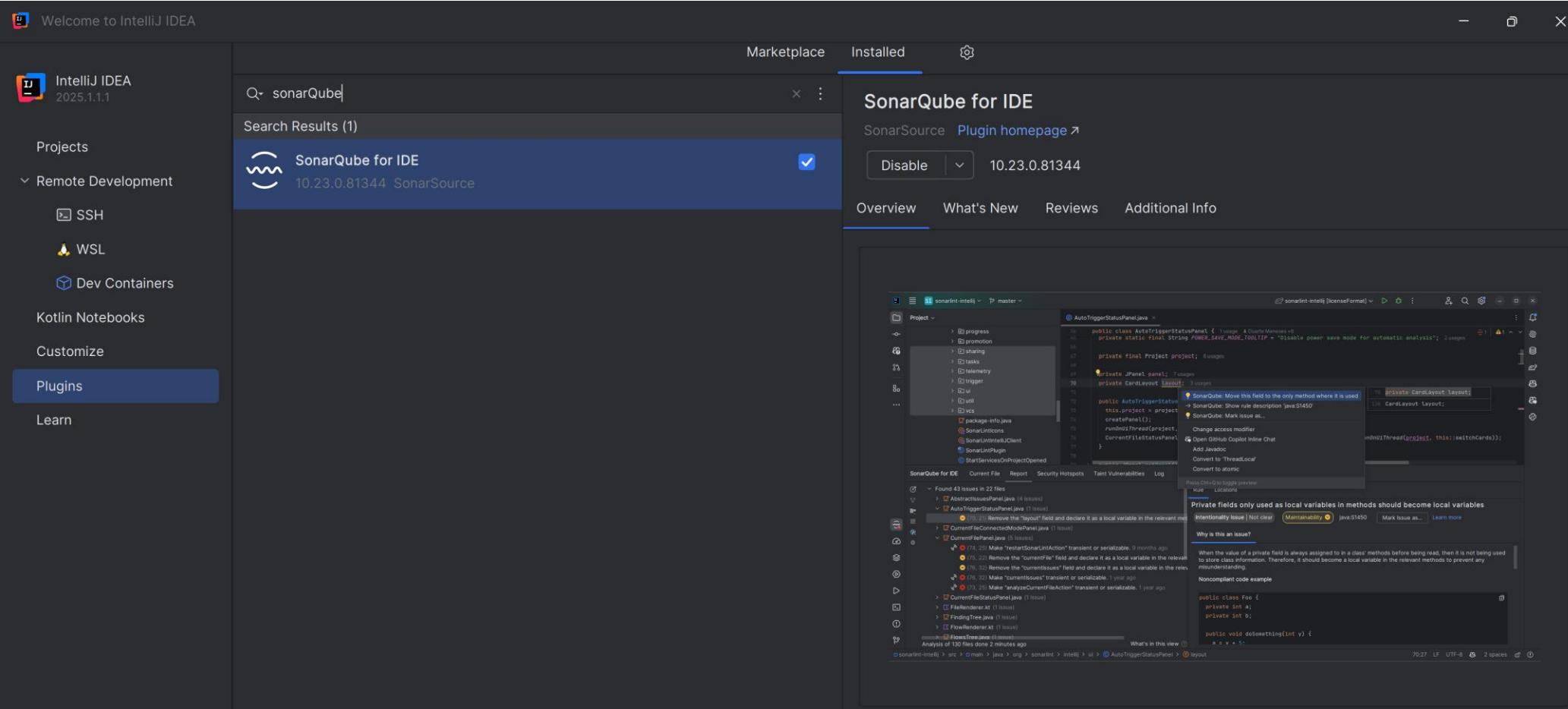


# SONARLINT

- SonarLint è un analizzatore statico contenuto nella piattaforma SonarQube
  - Negli ultimi anni, infatti è stato rinominato come **SonarQube for IDE**
- La piattaforma SonarQube è un più ampio strumento, che può valutare aspetti di qualità del lavoro di un intero team di sviluppo
  - La piattaforma completa SonarQube potrebbe essere studiata in esami successivi

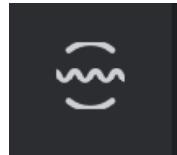


# INSTALLARE SONARQUBE FOR IDE



# UTILIZZARE SONARQUBE FOR IDE

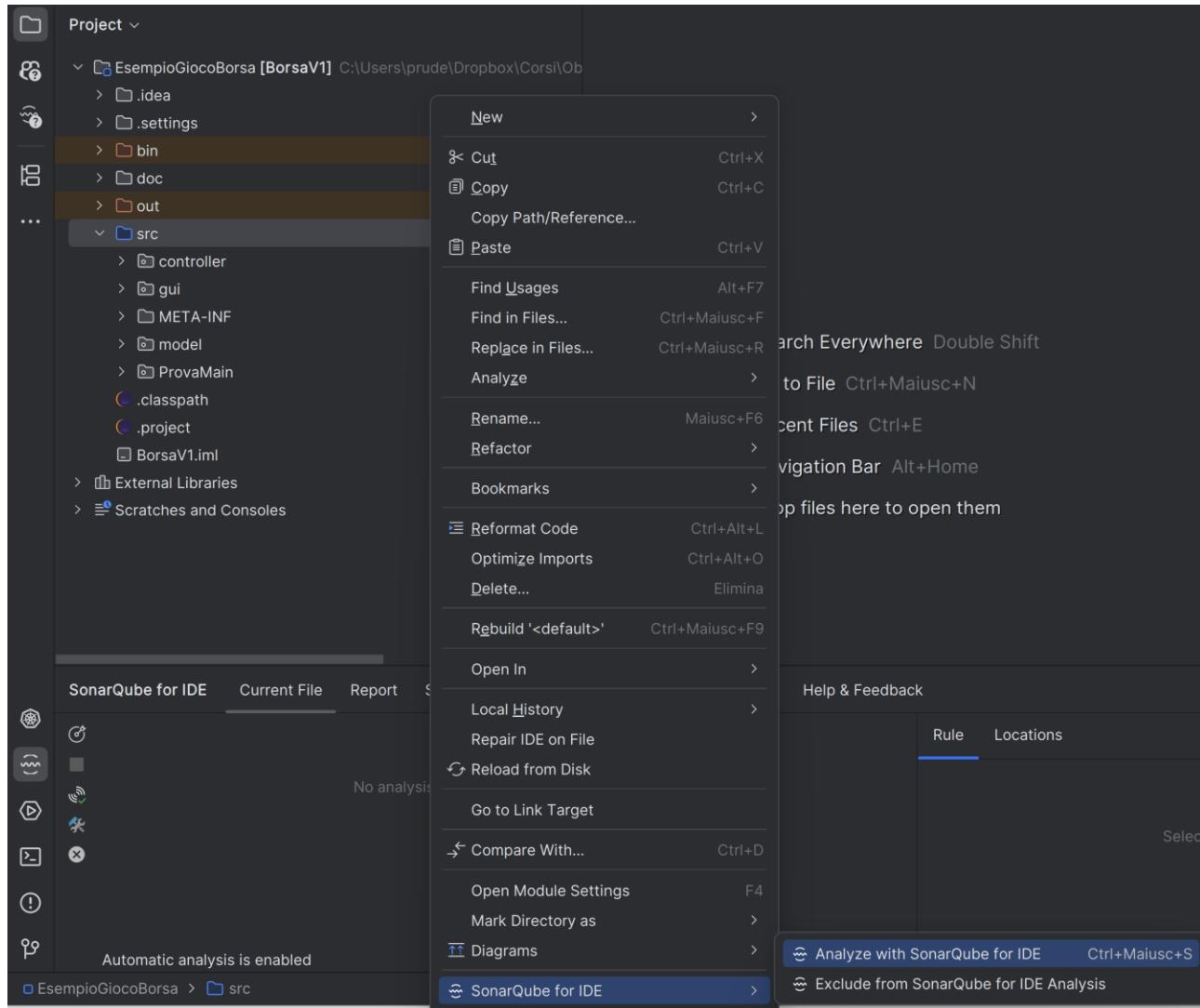
- L'icona di SonarQube for IDE



A screenshot of the SonarQube for IDE interface. The top navigation bar includes tabs for "SonarQube for IDE", "Current File" (which is selected), "Report", "Security Hotspots", "Taint Vulnerabilities", "Log", and "Help & Feedback". On the left, there's a vertical toolbar with various icons: a gear, a file, a signal, a play button, a minus sign, an exclamation mark, and a person. Below the toolbar, a message says "No analysis done on the current opened file" with a blue "Analyze Current File" button. In the center, under the "Rule" tab, it says "Select a finding to display the rule description". At the bottom, it shows "Automatic analysis is enabled" and a "What's in this view" link. A small red circular badge is visible in the bottom right corner.

# ANALIZZARE CON SONARQUBE FOR IDE

- Se non è stata già attivata l'analisi automatica, è possibile chiedere un'analisi dal menu di contesto (tasto destro)



# ANALISI DI SONARQUBE : ESEMPI

- Esempio:
  - Un attributo non viene mai utilizzato
    - Problema di maintainability: il programma è più complesso del necessario
    - Problema poco grave (arancione)

The screenshot shows the SonarQube for IDE interface. The top navigation bar includes links for SonarQube for IDE, Current File, Report, Security Hotspots, Taint Vulnerabilities, Log, and Help & Feedback. The 'Report' tab is selected.

The left sidebar displays analysis results for 9 files, with 45 issues found in total. The 'AggiungiNuovaSocieta.java' file is currently selected, showing 5 issues:

- (20, 19) Remove this unused "nomeSocietaLabel" private field. 3 hours ago
- (33, 46) Use static access with "javax.swing.WindowConstants" for "EXIT\_ON\_CLOSE". 3 hours ago
- (21, 20) Rename this field "OKButton" to match the regular expression "[a-zA-Z][a-zA-Z0-9]\*\$". 3 hours ago
- (36, 44) Make this anonymous inner class a lambda 3 hours ago
- (44, 39) Make this anonymous inner class a lambda 3 hours ago

The main panel on the right details the selected issue for 'AggiungiNuovaSocieta.java':

**Unused "private" fields should be removed**

Intentionality issue | Not clear      Maintainability (warning icon)      java:S1068      Learn more

Why is this an issue?

Parameters

ignoreAnnotations (none)  
Parameter values can be set in Rule Settings. In connected mode, server side configuration overrides local settings.

Analysis of 15 files done 1 hour ago      What's in this view ?



# ANALISI DI SONARQUBE : ESEMPI

- Esempio:

- L'attributo corrispondente al pulsante ok si chiama OKButton
  - Problema di maintainability: per convenzione gli attributi dovrebbero avere sempre un nome che inizia per minuscola. Chi legge il codice potrebbe credere che OKButton sia una classe
  - Problema lieve (giallo)

The screenshot shows the SonarQube for IDE interface. The top navigation bar includes 'SonarQube for IDE', 'Current File', 'Report', 'Security Hotspots', 'Taint Vulnerabilities', 'Log', and 'Help & Feedback'. The left sidebar displays a tree view of issues across 9 files, with 'AggiungiNuovaSocieta.java' expanded to show five specific issues. The main panel on the right details a single issue: 'Field names should comply with a naming convention' (Consistency issue | Not identifiable, Maintainability, java:S116). It provides a 'Why is this an issue?' link and a 'More Info' link. Below this, parameters are listed: 'format ^[a-z][a-zA-Z0-9]\*\$'. A note states: 'Parameter values can be set in Rule Settings. In connected mode, server side configuration overrides local settings.' At the bottom of the main panel, there's a 'What's in this view' link.



# ANALISI DI SONARQUBE : ESEMPI

## ■ Esempio:

- C'è un metodo con un Cognitive Complexity troppo alta
  - Problema di maintainability: la Cognitive Complexity è una metrica che indica appunto la difficoltà che avrà il lettore nel comprendere un metodo troppo complesso (troppe righe, troppe condizioni, troppi cicli)
  - Bisogna spezzare questo metodo in più metodi più piccoli e significativi
  - Problema considerato grave (rosso)

SonarQube for IDE    Current File    Report    Security Hotspots    Taint Vulnerabilities    Log    Help & Feedback

The screenshot shows the SonarQube interface with a list of issues on the left and a detailed view of a specific rule on the right.

**Issues List:**

- (243, 4) Replace this use of System.out by a logger. 2 minutes ago
- (83, 34) Use static access with "javax.swing.WindowConstants" for "EXIT\_ON\_CLOSE". 2 minutes ago
- (80, 8) Remove this unused "GrigioApple" local variable. 2 minutes ago
- (78, 0) Complete the task associated to this TODO comment. 2 minutes ago
- (74, 0) Complete the task associated to this TODO comment. 2 minutes ago
- (59, 8) Refactor this method to reduce its Cognitive Complexity from 71 to the 15 allowed. [+25 locations] 2 minutes ago
- (46, 19) Move the array designators [] to the type. 2 minutes ago
- (45, 13) Make non-static "provaTest" transient or serializable. 2 minutes ago
- (43, 20) Make non-static "controller" transient or serializable. 2 minutes ago
- (46, 19) Rename this field "MultiplaRisposta" to match the regular expression "[a-zA-Z][a-zA-Z0-9]\*\$". 2 minutes ago

Analysis of 36 files done 2 minutes ago

**Rule Details:**

**Cognitive Complexity of methods should not be too high**

Adaptability issue | Not focused    Maintainability (red)    java:S3776    Learn more

This rule raises an issue when the code cognitive complexity of a function is above a certain threshold.

Why is this an issue?    How can I fix it?    More Info

Parameters

Threshold 15  
Parameter values can be set in Rule Settings. In connected mode, server side configuration overrides local settings.



# ANALISI DI SONARQUBE : ESEMPI

## ■ Esempio:

- Nel codice è rimasto un commento con scritto TODO
  - Problema di maintainability: spesso i programmatore lasciano dei commenti con indicazione TODO come promemoria di codice da scrivere. Non è un vero e proprio problema ma vale come indicazione di incompletezza del codice.
  - Bisogna completare il codice e rimuovere il commento TODO
  - Problema considerato di natura solo informativa (azzurro)

SonarQube for IDE   Current File   Report   Security Hotspots   Taint Vulnerabilities   Log   Help & Feedback

The screenshot shows the SonarQube interface with a list of issues on the left and a detailed view of a specific rule on the right.

**Issues List:**

- (243, 4) Replace this use of System.out by a logger. 4 minutes ago
- (83, 34) Use static access with "javax.swing.WindowConstants" for "EXIT\_ON\_CLOSE". 4 minutes ago
- (80, 8) Remove this unused "GrigioApple" local variable. 4 minutes ago
- (78, 0) Complete the task associated to this TODO comment. 4 minutes ago** (This item is highlighted)
- (74, 0) Complete the task associated to this TODO comment. 4 minutes ago
- (59, 8) Refactor this method to reduce its Cognitive Complexity from 71 to the 15 allowed. [+25 locations] 4 minutes ago
- (46, 19) Move the array designators [] to the type. 4 minutes ago
- (45, 13) Make non-static "provaTest" transient or serializable. 4 minutes ago
- (43, 20) Make non-static "controller" transient or serializable. 4 minutes ago
- (46, 19) Rename this field "MultiplaRisposta" to match the regular expression '^a-zA-Z0-9\*'. 4 minutes ago

**Rule Details:**

Rule: Track uses of "TODO" tags

Intentionality issue | Not complete   Maintainability   java:S1135   Learn more

Why is this an issue?   More Info

Developers often use TODO tags to mark areas in the code where additional work or improvements are needed but are not implemented immediately. However, these TODO tags sometimes get overlooked, leading to incomplete or unfinished code. This rule aims to identify and address unattended TODO tags to ensure a clean and maintainable codebase. This description explores why this is a problem and how it can be fixed to improve the overall code quality.



# PROBLEMI PER CLASSE

- Le icone in alto a destra ci aprono direttamente la finestra dei problemi della classe aperta nell'IDE

The screenshot shows an IDE interface with the SonarLint plugin active. The code editor displays a Java file named `Borsa.java`. The SonarLint panel on the right shows 10 issues found in the file:

- (48, 16) Add a nested comment explaining why this method is empty, throw a
- (219, 12) Replace this use of System.out by a logger.
- (220, 12) Replace this use of System.out by a logger.
- (32, 52) Replace the type specification in this constructor call with the diamo
- (128, 11) The return type of this method should be an interface such as "List" i
- (137, 32) The type of "listaSocieta" should be an interface such as "List" rath
- (247, 11) The return type of this method should be an interface such as "List"
- (248, 51) Replace the type specification in this constructor call with the diamo
- (260, 11) The return type of this method should be an interface such as "List"
- (261, 51) Replace the type specification in this constructor call with the diamo

The SonarLint panel also includes sections for "Rule", "Locations", "Methods should not be empty", and "Why is this an issue? How can I fix it?".



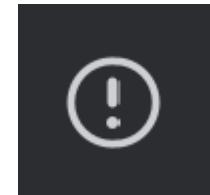
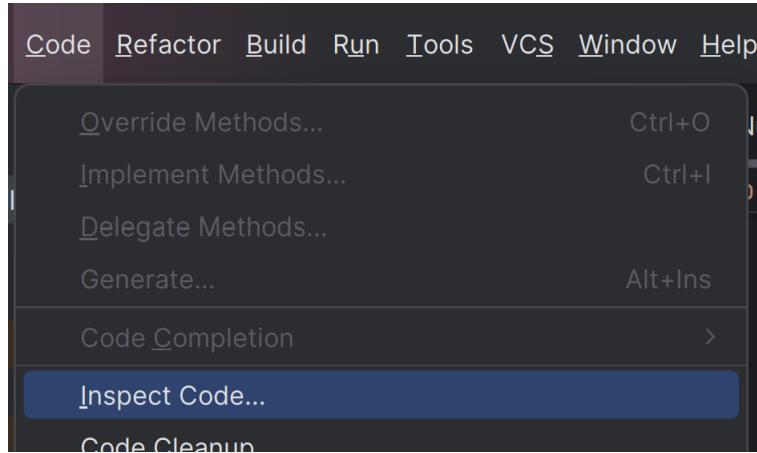
# CONSIDERAZIONI

- Bisognerebbe sempre utilizzare strumenti come SonarQube mentre si programma:
  - Leggendo e comprendendo le indicazioni di SonarQube
  - Correggendo il codice (anche se funziona)
- Chiaramente molte delle indicazioni di SonarQube potrebbero risultare al momento poco comprensibili poiché trattano aspetti di Java non trattati in questo corso
  - Ad esempio le lambda functions
- Oppure potrebbero riguardare codice generato automaticamente
  - Ad esempio da Swing UI Designer
- In tutti questi casi, i problemi possono essere *ignorati*, nell'ambito di questo corso



# ALTRI STRUMENTI DI ANALISI STATICHE

- SonarQube è uno strumento standalone, che può essere utilizzato anche indipendentemente dall'IDE, oppure come plug-in in quasi tutti gli IDE
- Gli IDE hanno però anche altri strumenti di analisi statica loro specifici
- Ad esempio in IntelliJ è disponibile la funzione **Inspect Code** dal menu **Code**



# UTILIZZARE L'ANALISI STATICÀ DI INTELLIJ

- IntelliJ Idea classifica ulteriormente gli elementi analizzati per tipologia
- Alcuni dei problemi si riferiscono a parti di codice generati automaticamente. Quelli più rilevanti sono di solito quelli di Java, Security, UI Form
- Molto numerosi sono i problemi di Proofreading, specialmente se il sistema non ha riconosciuto la lingua (italiano nel nostro caso)

```
✓ Inspection Results 'Project Default' profile (1,883 items)
  > CSS 2 errors 62 warnings 5 weak warnings
  > HTML 60 warnings
  > Internationalization 4 warnings
  > Java 70 warnings
  > JavaScript and TypeScript 20 warnings 222 weak warnings
  > Kotlin 1 weak warning
  > Proofreading 47 grammar errors 1.367 typos 1 information
  > Security 1 weak warning
  > UI form 21 warnings
```



# ESEMPIO DI PROBLEMA

- L'analisi rileva una throws corrispondente ad una eccezione che non può essere (apparentemente) mai causata dal codice
- Per approvare, si può eseguire il suggerimento (lampadina) di rimuoverlo

The screenshot shows an IDE interface with a code editor and a navigation pane.

**Navigation pane:**

- > **Societa** 2 warnings
- ↳ Declaration redundancy 29 warnings
  - > Declaration can have 'final' modifier 19 warnings
  - ↳ Redundant 'throws' clause 1 warning
  - ↳ **Giocatore** 1 warning
    - The declared exception 'Exception' is never thrown
  - > Unused declaration 9 warnings
  - > Java language level migration aids 17 warnings
  - > JavaScript and TypeScript 20 warnings 222 weak warnings
  - > Kotlin 1 weak warning
  - > Proofreading 47 grammar errors 1.367 typos 1 information

**Code Editor:**

```
* @return valore del capitale totale del giocatore
* @throws Exception eccezione generica messa li tanto per provare
*/
public float calcolaCapitale() throws Exception {
    float capitale=liquidita;
    for (Acquisto a : acquisti) {
        capitale+=a.getAzione().getPrezzo()*a.getQuantita();
    }
    return capitale;
```

A tooltip is displayed above the code editor:  
Remove unnecessary 'throws' declarations      Suppress ▾



# ALCUNI PROBLEMI COMUNI

## ▼ JVM languages 1 warning

### ▼ Non-safe string is used as SQL 1 warning

-   ListinoImplementazionePostgresDAO 1 warning

String, which is used in SQL, can be unsafe

## ▼ Code maturity 2 warnings

### ▼ Call to 'printStackTrace()' 2 warnings

#### ➤ ConnesioneDatabase 1 warning

Call to 'printStackTrace()' should probably be replaced with more robust logging

## ▼ Performance 3 warnings

### ▼ 'String.equals()' can be replaced with 'String.isEmpty()' 3 warnings

#### ➤ FinestraAcquistaAzione 1 warning

#### ➤ finestraAggiungiSocieta 2 warnings

## ▼ No label for component 4 warnings

### ➤ FinestraAcquistaAzione.form 2 warnings

### ➤ finestraAggiungiSocieta.form 2 warnings

### ➤ Scrollable component not in JScrollPane 1 warning

## ▼ Declaration redundancy 30 warnings

### ▼ Declaration can have 'final' modifier 1 warning

#### ➤ Home 1 warning

Declaration can have final modifier

### ➤ Empty method 1 warning

### ➤ Unused declaration 28 warnings

## ▼ UI form 27 warnings

### ▼ Hardcoded string literal in a UI form 13 warnings

#### ➤ FinestraAcquistaAzione.form 3 warnings

#### ➤ finestraAggiungiSocieta.form 3 warnings

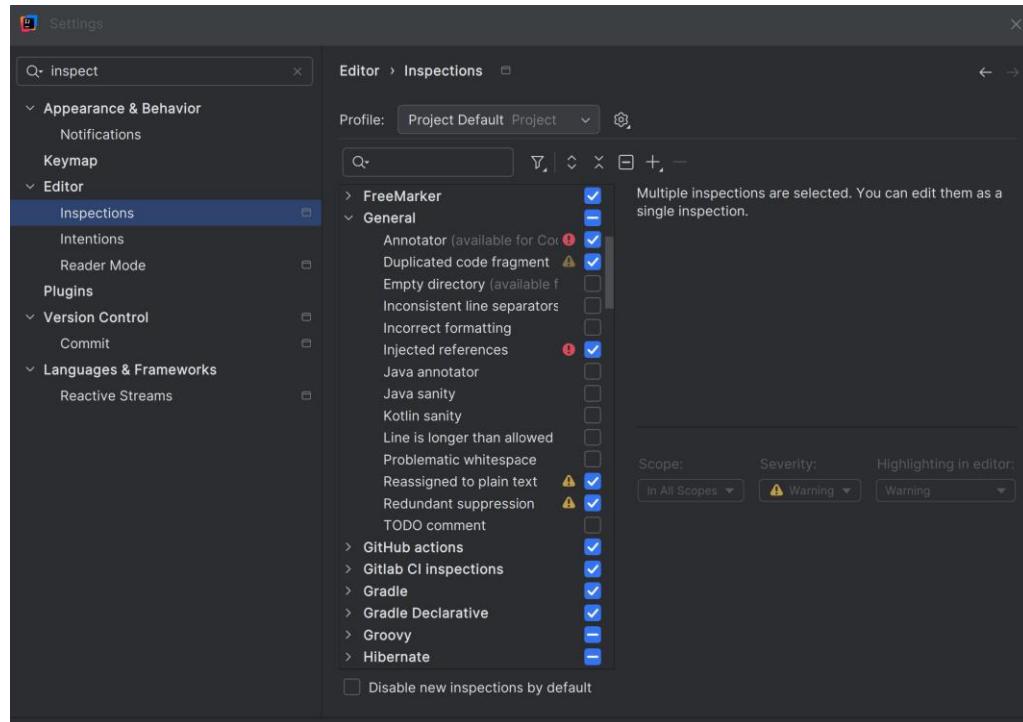
#### ➤ Home.form 6 warnings

#### ➤ ListaSocieta.form 1 warning



# IMPOSTAZIONI DELL'ANALISI DI INTELLIJ

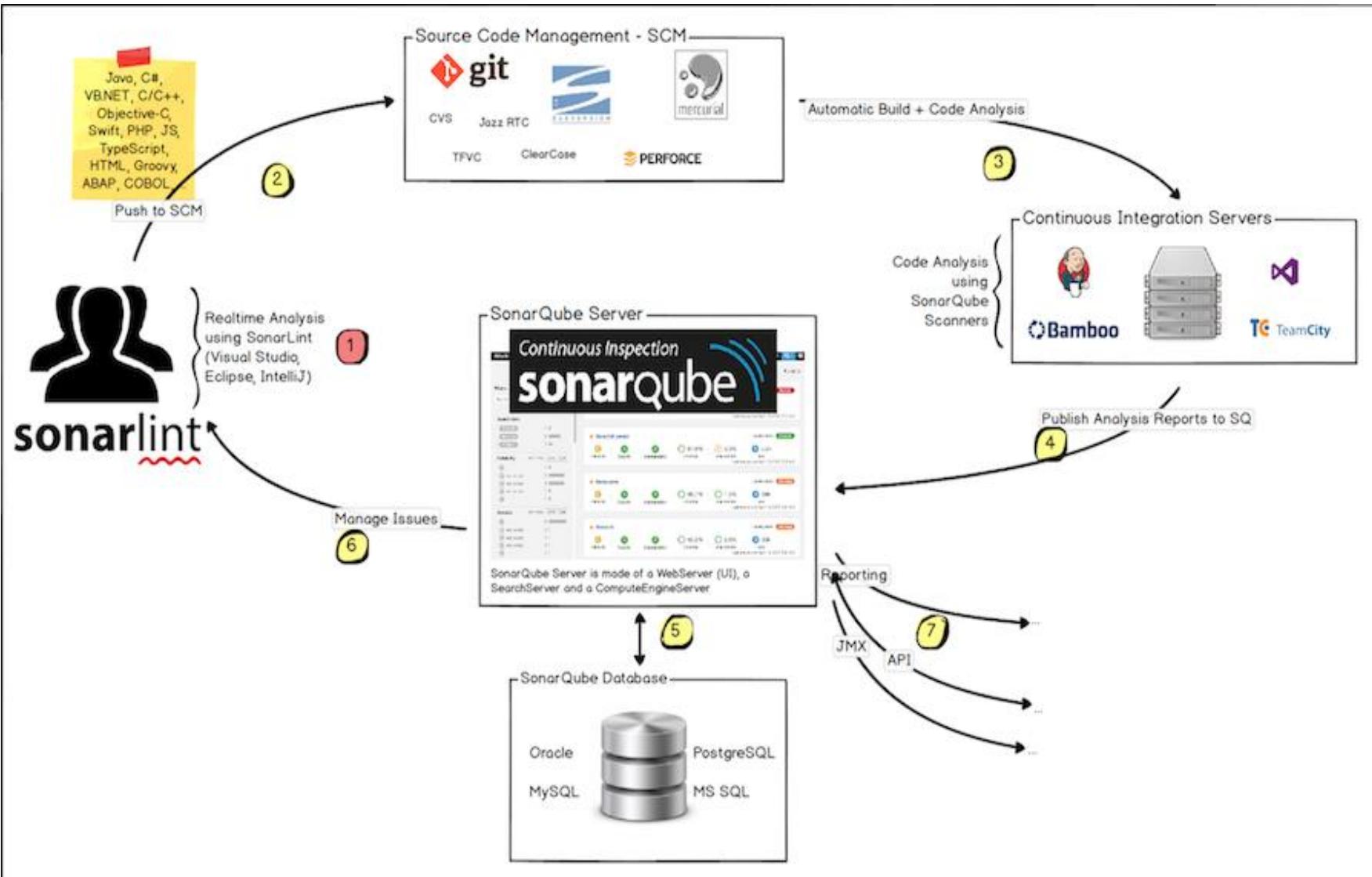
- Dai Settings di IntelliJ è possibile aumentare o ridurre l'insieme di controlli da effettuare

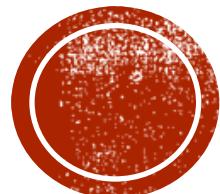


# SONARQUBE PROCESS

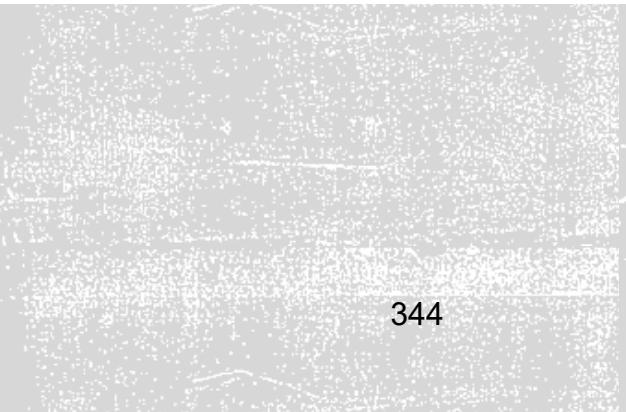
1. Developers code in their IDEs and use SonarLint to perform local analysis during development
2. Developers submit their code in CVS (e.g. push to git)
3. The Continuous Integration Server (CIS) triggers an automatic build of the project and the execution of the SonarQube Scanner required to perform the analysis.
4. The analysis report is sent to the SonarQube server for processing.
5. SonarQube Server processes and stores the results of the analysis report in the SonarQube database and displays the results in the user interface.
6. Developers review, comment and address their issues to manage and reduce their technical debt through the SonarQube user interface.
7. Managers receive reports from the analysis.







# CONCURRENT VERSIONING SYSTEMS



# Gestione delle versioni di un software

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



# Requisiti di un sistema per la gestione delle versioni

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



# Gestione delle release

- Una release di un sistema è una sua versione che viene distribuita ai clienti. Essa comprende, tra l'altro:
  - Codice eseguibile
  - File di configurazione
  - Programma di installazione
  - Documentazione
  - ...
- Il processo di creazione e rilascio di una release deve quindi riuscire a gestire la generazione di tutti questi deliverables
  - In particolare, bisogna decidere se distribuire l'intero sistema oppure se distribuire unicamente delle patch di aggiornamento



# Identificazione delle versioni

- Numerazione
  - #versione.#modifica.#variazione
- Identificazione basata su attributi
  - Le modifiche vengono etichettate con attributi (eventualmente ordinati), in modo da poter caratterizzare anche l'impatto della modifica oltre che l'ordine delle versioni
- Identificazione orientata alle modifiche
  - Le modifiche al sistema vengono etichettate in base alle modifiche sui singoli componenti



# Protocolli e strumenti per la gestione delle versioni

- CVS
- SVN
- Git
- ...
- Dropbox
- Google Drive
- Google Docs
- OneDrive
- ...



# CVS: Concurrent Versioning System

- CVS è stato il primo sistema di gestione delle versioni ad essere diventato di uso comune
- E' stato adottato da molte industrie software e ha supportato i primi progetti open source su larga scala tramite una piattaforma chiamata **sourceforge**.
- Con il nome CVS intendiamo diverse cose:
  - Un Sistema di gestione di repository contenenti la storia delle versioni di un software
  - Un protocollo di comunicazione utilizzato da questo sistema
  - Un eseguibile in grado di interpretare i comandi di questo protocollo e che consente di interagire con il Sistema CVS
- L'architettura CVS è centralizzata
  - Su di un server viene mantenuto lo stato attuale del Progetto e tutta la sua storia precedente (**repository**)
  - Ogni partecipante al Progetto può scaricare (**checkout**) una copia locale (**sandbox**) dello stato attuale del Progetto, programmare delle sue modifiche personali al progetto (**edit**) e caricarle sul repository centralizzato (**commit**)
- C'è un grande problema: se diversi programmatore lavorano in parallelo, è possibile che essi generino delle versioni in conflitto tra loro
  - CVS implementa strategie per risolvere oppure evitare questi conflitti



# Modello Lock/Modify/Unlock

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

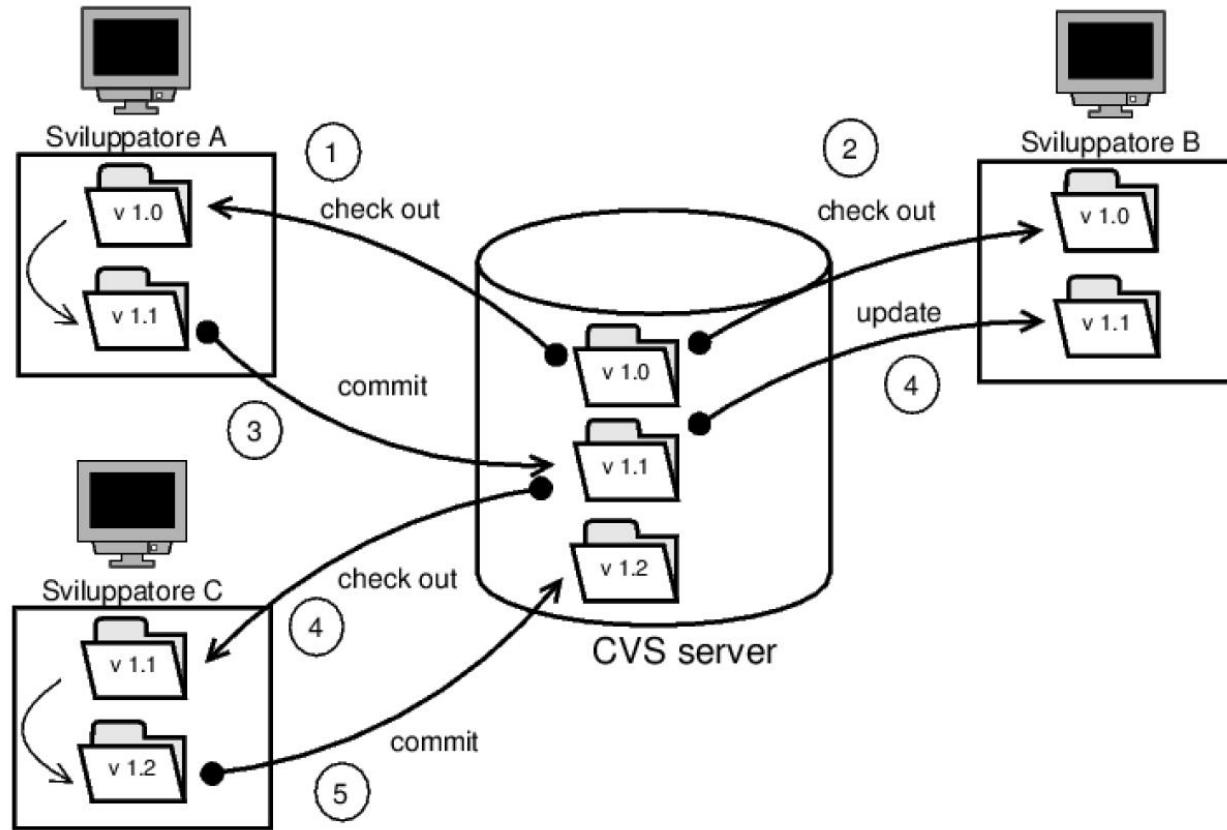


# Modello Copy/Modify/Merge

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



# Modello Copy/Modify/Merge



# Conflitti

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



# Risoluzione dei conflitti

- Ci sono tre possibili soluzioni al problema dei conflitti tra versioni diverse :
  1. **Accettare** una delle due versioni e **gettare** l'altra
    - C'è stato solo uno spreco di tempo, da parte di un programmatore
  2. **Fondere** (**Merge**) le due versioni, provando a mantenere le modifiche di entrambe
    - Se le modifiche riguardano file differenti, potrebbe essere semplice fonderle in maniera quasi automatica, altrimenti bisogna procedere manualmente
  3. **Duplicare** (**Fork**) i repository, mantenendo vive due diverse versioni (**branch**), ognuno dei quali con una versione delle modifiche
    - In un momento successivo potrebbe essere possibile fondere queste due versioni in una sola



# DISCUSSIONE

- Il modello lock/modify/unlock previene la possibilità di conflitti
  - Ma la produttività di un team diminuisce con le dimensioni del team
- La modularità del software e una suddivisione ottimale dei compiti possono migliorare la produttività
- Il copy/modify/merge consente una maggiore produttività poiché consente ai diversi programmatori di lavorare in parallelo
  - Ma introduce la necessità di risolvere i conflitti
  - Per i team di grandi dimensioni, la risoluzione dei conflitti può rappresentare un collo di bottiglia, perché deve essere svolta in generale sempre dall'amministratore del progetto
- In ogni caso, ogni sviluppatore dovrebbe sempre aggiornare preventivamente la propria versione del progetto (**update**) prima di iniziare a modificarlo



# GIT & GITHUB



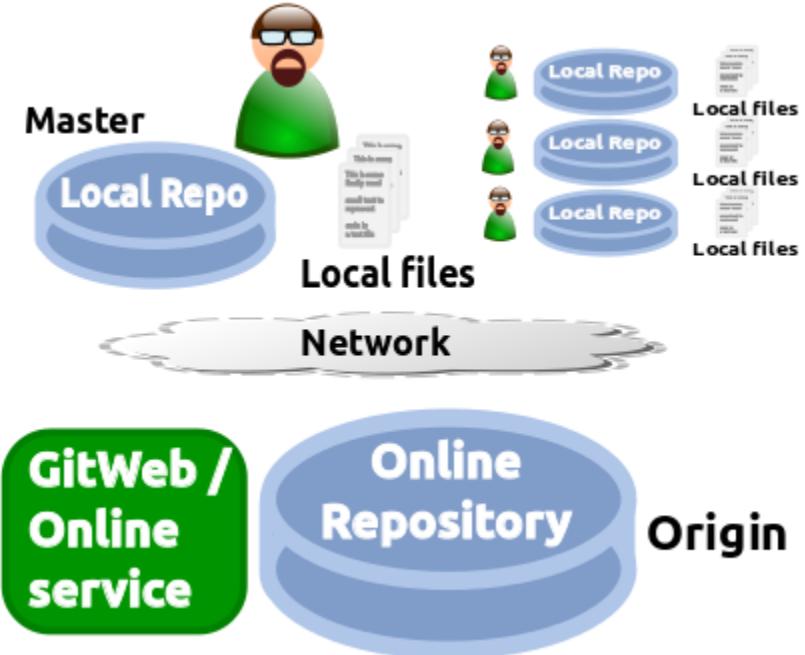
# Git

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

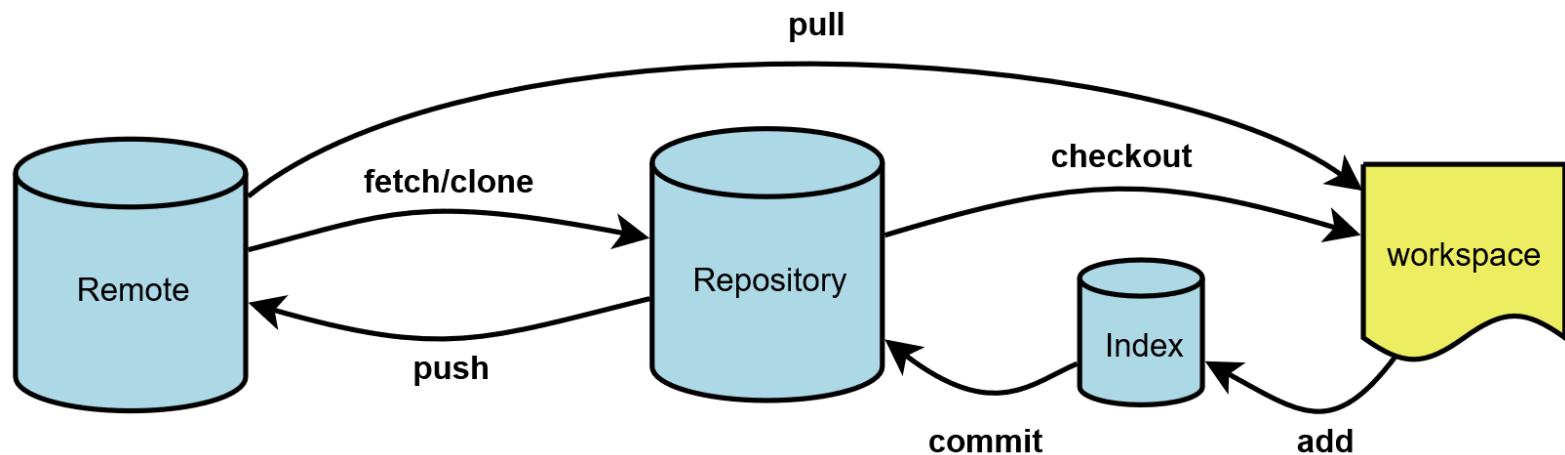


# Git Architecture

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



# RELAZIONE TRA OPERAZIONI LOCALI E REMOTE



# COMMIT & PUSH

- **Commit** è un'operazione che aggiorna *localmente* il nostro Progetto supportato da git (in generale va a modificare file .git)
- **Push** è un'operazione che invia al repository remote (ad esempio ospitato da Github) una richiesta di aggiornamento di tutti i file modificati in locale
  - Solo un proprietario (owner) del progetto può fare richiesta di push: tutti gli altri al massimo possono fare una *pull request*
  - La richiesta di Push viene analizzata in cerca di possibili conflitti tra la versione precedente e quella successiva alla modifica





# PULL REQUEST

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



# Differenze con CVS

## Il sistema Git:

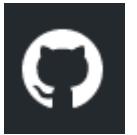
- È più sicuro: non esiste un'unica copia centralizzata del progetto. I contributori possono mantenere proprie copie e scegliere se provare a condividerle con quella principale. Il proprietario può scegliere di non accettare modifiche di cui non si fida
- È più scalabile: il numero di partecipanti al progetto può aumentare liberamente
  - Invece in CVS e SVN, ad esempio, aumentando il numero di partecipanti aumenta il tempo in cui le risorse sono bloccate e/o il numero di conflitti sui commit
    - Il sistema Linux è stato sviluppato storicamente sotto Git, con il master sotto il diretto controllo di Linus Torvalds
  - Il lavoro di merge delle versioni in conflitto non è svolto dal proprietario ma da chi propone la modifica, che deve proporre sue modifiche che non siano in conflitto con l'originale



# GIT HOSTING

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





# NAVIGAZIONE NEL CODICE

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

The screenshot shows a GitHub repository page for 'reverse-unina / AndroidRipper'. The top navigation bar includes links for Pull requests, Issues, Marketplace, and Explore. The repository name 'reverse-unina / AndroidRipper' is displayed, along with a 'Watch' button (4), a 'Star' button (22), and a 'Code' button. Below the header, there are tabs for Code, Issues (5), Pull requests, Actions, Projects, Wiki, Security, and Insights. The 'Code' tab is selected. A dropdown menu shows 'master' (selected), 1 branch, and 1 tag. The main content area displays a list of commits for the 'master' branch, showing files like 'AndroidRipper', 'AndroidRipperDriver', 'AndroidRipperService', '.gitignore', 'LICENSE', and 'README.md' with their respective update details. To the right, sections include 'About' (describing it as a toolset for mobile GUI testing), 'Releases' (listing 'Android Ripper 2017.10' as the latest release from Nov 7, 2017), 'Packages' (no packages published), and 'Contributors' (2 contributors listed). The bottom of the page contains a summary of the repository's purpose and development information.

**About**

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

**Releases** 1

Android Ripper 2017.10 (Latest)  
on 7 Nov 2017

**Packages**

No packages published

**Contributors** 2

reverse-unina REsEarth gRoup of So...

**AndroidRipper**

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

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





# CREAZIONE DI UN PROGETTO

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

 GitHub Free  
The basics for all developers

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

## Create a new repository

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

Owner \* Repository name \*

 PorfirioTramontana /

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

Description (optional)

 Public  
Anyone on the internet can see this repository. You choose who can commit.

 Private  
You choose who can see and commit to this repository.

Initialize this repository with:  
Skip this step if you're importing an existing repository.

Add a README file  
This is where you can write a long description for your project. [Learn more.](#)

Add .gitignore  
Choose which files not to track from a list of templates. [Learn more.](#)

Choose a license  
A license tells others what they can and can't do with your code. [Learn more.](#)

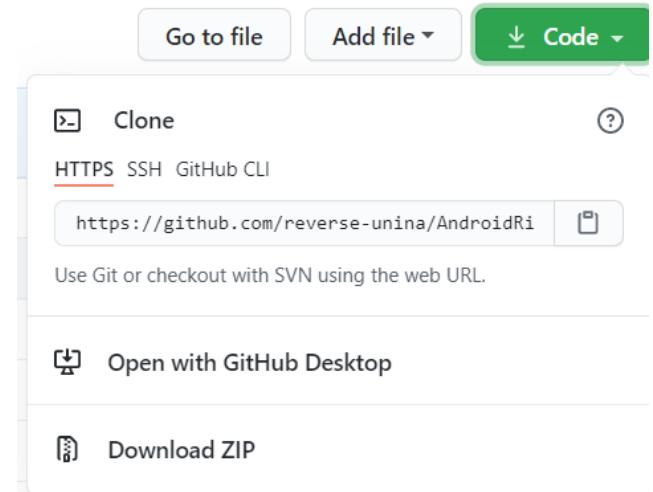
[Create repository](#)



# IMPORTAZIONE DI UN PROGETTO ESISTENTE



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





# NAVIGAZIONE NEL CODICE

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





# FORK DI UN PROGETTO ESISTENTE

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

Fork 17

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





# ALTRÉ INTERAZIONI CON I PROGETTI

- **Watch**

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

- **Star**

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





# ISSUE

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

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

- Enable stalebot status:team discussion type:task #2602 opened 2 days ago by marphilipp 0 of 1 5.8 M2
- Beginning with 1.7.1 standalone console does not find tests on Java 8 component:Platform type:bug #2600 opened 3 days ago by dmitriy 5.7.2
- Support AutoCloseable for arguments provided to a @ParameterizedTest component:Jupiter theme:parameterized tests type:enhancement #2597 opened 10 days ago by sbrannen 0 of 1 5.8 M2
- Discover tests in suites packaged into jars, e.g. TCKs #2594 opened 14 days ago by t1 0 of 2
- Add concrete steps to "Migrating from JUnit 4" docs section. #2593 opened 14 days ago by t1 2



# BUILD AUTOMATION



# Build Automation

- Build Automation è l'insieme di attività legate all'automatizzazione di alcuni task del ciclo di vita del software
  - Compilazione;
  - Linking;
  - Esecuzione di test;
  - Analisi statica
  - Deployment;
  - Creazione di documentazione;
  - ...
- Per poter automatizzare queste attività è spesso necessario scrivere del codice in un linguaggio di scripting (oppure utilizzare programmi visuali)
  - Il codice di build automation è da considerare a pieno titolo nell'insieme del codice del programma!



# Build Automation

- Tramite un efficace codice di Build Automation è possibile, ad esempio:
  - Generare e deployare automaticamente diverse versioni del programma
    - Versioni in lingue diverse
    - Versioni con insiemi di feature diverse
    - Versioni adatte a diversi sistemi operativi
    - Versioni di testing/debugging con alcune parti di codice sostituite da moduli fintizi
  - Reperire automaticamente le risorse necessarie a completare l'esecuzione del programma



# Strumenti di Build Automation

- Make
- Apache Ant
- **Apache Maven**
- Gradle
- YAML
- Visual Build
- Jenkins
- [http://en.wikipedia.org/wiki/List\\_of\\_build\\_automation\\_software](http://en.wikipedia.org/wiki/List_of_build_automation_software)



# Shell Script

- La soluzione più semplice ai problemi di build automation passa per la scrittura di script di shell
  - La maggior parte degli strumenti necessari (compilatore, linker) sono eseguibili da linea di comando
  - Tutte le shell (es. Bash, DOS shell, etc.) forniscono comandi a sufficienza per la gestione del file system
- I linguaggi di shell scripting sono interpretati: se vengono utilizzati per realizzare elaborazioni complesse, queste ultime sono difficili da testare
- I linguaggi di shell offrono poco supporto alla programmazione strutturata e nessun supporto alla programmazione a oggetti
- I linguaggi di shell non offrono supporto al testing e al debugging
- I linguaggi di shell non hanno tipi (salvo alcune eccezioni parziali)



# Comandi Shell DOS

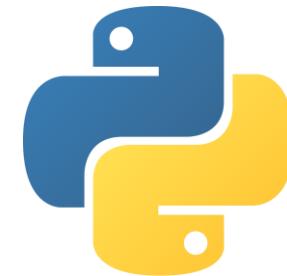
- Call, per chiamare un altro batch file (.bat)
- Start, per avviare un eseguibile, eventualmente in un'altra istanza di shell
- Choice, per implementare una sorta di switch
- For, per implementare un ciclo for (in un insieme predefinito di valori)
  - Forfiles, per implementare un ciclo su di un insieme di files
- Goto, salto incondizionato
- If, che supporta come condizioni l'identità tra stringhe (==), l'esistenza di un file (EXIST) e la restituzione di un certo valore di ritorno di errore (ERRORLEVEL)
- Set, per le assegnazioni di variabile
- Inoltre è possibile utilizzare un costrutto di *blocco* (rappresentato da parentesi tonde), che però può andare in conflitto con i goto
- Tipici comandi di gestione file (copy, xcopy, dir, delete, rename, cd, md, rd, ...)
- Le shell di Unix (bash o altre shell) forniscono tipicamente più feature ma seguono la stessa filosofia



# Esempi shell (Windows)

```
REM Lettura dei parametri dalla linea di comando
set name=%1
REM setta una variabile in funzione di data e ora
for /f "tokens=1-6 delims=.:/" %%a in ("%date% %time%") do set mydatetime=%%c-%%b-
%%a_%%d-%%e-%%f
set seedvalue=!RANDOM!
REM Esempio di ciclo do-while
set /a cycle=0
:START
set /a cycle+=1
if !cycle!==1 goto END
goto START
:END
REM Esempio di ciclo for
for /L %%N in (1,1,%seedvalue%) do (
(
echo Number
echo %%N
) >> list.txt
)
```





# Python (cenno)

- Python può rappresentare una soluzione recente per realizzare:
  - Build automation basata su script imperativi
  - Indipendente (parzialmente) dal sistema operativo
  - Sfruttando costrutti sicuramente più potenti di quelli della shell
- Esempio:

```
import os
import sys
import shutil
import time

def startEmulator():
    os.popen("emulator -avd leakAVD -wipe-data")
    time.sleep(50)
startEmulator()
les=["doc","bf","stai"]
nevent=[1,2,10]
for event in les:
    for number in nevent:
        os.system("python AndroLeak.py emulator-5554 "+event+" "+str(number))
os.popen("adb -s emulator-5554 emu kill")
time.sleep(50)
print("Testing Accomplished!")
```



# Make

- Sotto il nome ‘make’ è possibile raggruppare parecchie utility diffuse sui vari sistemi Linux o Windows
  - La più nota è GNU Make
- È distribuita in forma di eseguibile a linea di comando
  - Le sue elaborazioni dipendono da un file di scripting nominato *makefile*
  - L’unico parametro, opzionale, è un TARGET che rappresenta la label del punto del makefile da cui comincerà l’esecuzione
- Il makefile è uno script in un linguaggio dichiarativo, non imperativo, che permette di specificare come debbano avvenire le operazioni di deployment e installazione di un software



# Makefile

- Uno script makefile è organizzato in regole (rules)
- Ogni regola è individuata da:
  - un'etichetta (Target) seguita da:
  - una lista (opzionale) di componenti da cui la regola dipende
  - Un elenco di comandi (commands) di shell collegati a quella regola

```
targets : prerequisites ; command
```

- Uno script makefile è organizzato in regole (rules)

```
copia: originale.txt; copy originale.txt copia.txt
```



# Esempio Makefile

```
#Variabili
CC    = gcc
CFLAGS = -g

all: helloworld

helloworld: helloworld.o
$(CC) $(LDFLAGS) -o $@ $^

helloworld.o: helloworld.c
$(CC) $(CFLAGS) -c -o $@ $<

clean:
rm -f helloworld helloworld.o
```

- CC e CFLAGS sono variabili
- All (target predefinito) per essere eseguito prevede come prerequisito l'esistenza del file helloworld: se non esiste, allora viene eseguito preventivamente il target omonimo
- helloworld ha bisogno che esista helloworld.o, altrimenti esegue preventivamente il target chiamato helloworld.o
- \$@ è una variabile che rappresenta il nome del target mentre \$< rappresenta i parametri in ingresso alla chiamata make



# Limiti dei makefile

- Esistono alcune (poche) altre possibilità di implementare strutture di controllo nell'ambito di un makefile
- Make dipende dal sistema operativo: con sistemi operativi diversi non è possibile riutilizzare identicamente gli stessi script (perchè dipendono dal linguaggio di shell) e sono anche necessari diversi porting di make



# Maven



- Maven non è soltanto un tool per la build automation (come Ant) ma:
  - *“attempt to apply patterns to a project's build infrastructure in order to promote comprehension and productivity by providing a clear path in the use of best practices”*
  - In particolare fu pensato per unificare il modo di sviluppare e organizzare diversi progetti della famiglia Apache
- Non si occupa solo della costruzione, installazione e deployment dei progetti, ma anche della generazione della documentazione, di metriche, reports e casi di test
- Allo stesso modo di make e ant, può essere eseguito da linea di comando
- legge uno o più file xml di configurazione



# Maven

Maven promuove prima di tutto la formazione di uno schema di riferimento del codice e delle risorse (**archetipo**)

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

- Pom sta per Project Object Model
- Il tag packaging sta ad indicare che il risultato eseguibile del building verrà messo in forma di file compresso jar
- version è la versione del progetto che verrà generata da questo build. Dovrebbe ricalcare il numero di release mantenuto dal cvs
- url è il sito del progetto (bisogna modificare il valore di default che rappresenta il sito di maven stesso)
- L'unica dipendenza dichiarata nell'archetipo è quella da Junit necessario per il testing

# Albero dei file generati

```
my-app
|--- pom.xml
`--- src
    |-- main
    |   '-- java
    |       '-- com
    |           '-- mycompany
    |               '-- app
    |                   '-- App.java
    '-- test
        '-- java
            '-- com
                '-- mycompany
                    '-- app
                        '-- AppTest.java
```

- Lo script di esempio può essere eseguito con
  - mvn compile
- Eseguendo maven nella cartella che contiene pom.xml, viene generato quest'albero dei file
- I file Java sono creati sotto forma di template da completare



# Altre modalità di esecuzione di mvn

- Per eseguire i test è sufficiente scrivere:
  - mvn test
    - Test è il valore di un tag scope, che è equivalente al concetto di target visto in make ed ant
- Per creare l'eseguibile (jar):
  - mvn package
- Altre modalità di esecuzione standard:
  - mvn site → genera il sito web di documentazione
  - mvn clean → cancella tutti i file generati
- Tutte queste opzioni possono essere osservate da IntelliJ dal menu



# Dependencies

- E' possibile dichiarare dependencies da altri progetti non disponibili in locale

```
<dependencies>
  ...
  <dependency>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1.0-SNAPSHOT</version>
    <scope>compile</scope>
  </dependency>
</dependencies>
```

- Come si può notare, NON viene nominata esattamente l'URL del progetto da cui dipende ciò che dobbiamo costruire, ma il groupId, l'artifactId e la version saranno sufficienti a maven per reperirlo nel caso esso sia stato correttamente pubblicato nell'ambito del dominio com.mycompany.app
- Junit faceva eccezione: in assenza della dichiarazione della URL, maven cerca in locale e sul sito ufficiale del progetto Maven



# Maven e il Web

- Maven osserva le dipendenze:
  - Se le librerie sono disponibili in locale le utilizza
  - Altrimenti le scarica sul momento
    - Tramite le properties ci sono dei siti predefiniti che possono contenere le librerie, ma è possibile anche indicarne degli altri
    - La prima esecuzione di Maven può quindi essere molto più lunga delle altre
    - E' quindi fondamentale indicare le librerie esattamente per nome e per versione
    - Per non sbagliare, è possibile cercare le librerie sul web e trovare la stringa esatta da aggiungere nelle dependencies
    - Ad esempio, per postgres 42.7.5
      - <https://mvnrepository.com/artifact/org.postgresql/postgresql/42.7.5>
      - <!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->
      - <dependency>
      - <groupId>org.postgresql</groupId>
      - <artifactId>postgresql</artifactId>
      - <version>42.7.5</version>
      - </dependency>



# **INTEGRAZIONE CON IL DATABASE**



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

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



# PROBLEMA : GESTIONE DEI DATI PERSISTENTI

- Nell'architettura BCE adottata finora, i dati sono mantenuti unicamente dalle classi del Model, in modo temporaneo
  - Alla apertura dell'applicazione non ci sono dati
  - Alla chiusura dell'applicazione tutti i dati letti vengono dimenticati
- Abbiamo bisogno di una soluzione per mantenere in forma persistente alcuni dati
- Soluzioni tecnologiche possibili:
  - File esterni (questa soluzione è stata accennata nei corsi precedenti)
  - **Database** (si assume la conoscenza basilare dei database dal corso di basi di dati)
- Soluzioni architetturali possibili:
  - BCED
  - **BCE + DAO**



# INTRODUZIONE DEL DATABASE NEL PROGETTO

- Vogliamo inserire un database **Postgres** in un Progetto software di esempio
- Al progetto è stata aggiunta la libreria **postgresql-42.7.5.jar**  
E' stata aggiunta già nel progetto di partenza nel **pom.xml**

```
<dependencies>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.7.5</version>
    </dependency>
</dependencies>
```

- In alternativa, è sempre possibile 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**



# OPERAZIONI PRELIMINARI

- Come imparato nel corso di dati, per fare una prova è necessario:
  - Installare postgres, scegliendo nome utente e password
  - Aprire postgres (pgAdmin 4)
  - Connettere il server (localhost)
  - Creare un database (Database/Create Database ...) dal nome appropriato (Borsa, nel programma di esempio)
  - Creare lo schema del database (tasto destro su Borsa, Query Tool ed eseguire la query di creazione riportata nella slide successiva)
  - Successivamente, per ispezionare le tabelle del database sarà sufficiente andare in Borsa/Schemas/public/Tables, selezionare una tabella e dal tasto destro scegliere View/Edit Data



# 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
);
```



# CONNESSIONE AL DATABASE DAL CODICE

- Risolviamo prima di tutto il problema di accedere al database dal codice
- Per semplicità lo risolviamo con una classe apposita di utilità che chiameremo ConnessioneDatabase
  - Volendo si poteva anche creare il database via SQL dal programma stesso
    - Soluzione poco generale : il database sarebbe stato senza dati ad ogni avvio dell'applicazione
- Un esempio della classe ConnessioneDatabase è stato caricato nel Progetto di esempio



# CLASSE CONNESSIONE DATABASE

```
public class ConnessioneDatabase {  
    private static ConnessioneDatabase instance;  
    public Connection connection = null;  
    private String nome = "postgres";  
    private String password = "password";  
    private String url = "jdbc:postgresql://localhost:5433/Borsa";  
    private String driver = "org.postgresql.Driver";
```

- Per semplicità login, password e indirizzo del database sono stati aggiunti nel codice: ovviamente **non** è una soluzione generale e sicura!
- Il porto (5433 o 5432) è un'impostazione data all'avvio di postgres
- Instance è static perchè vogliamo essere sicuri che esiste sempre al massimo una sola connessione al database per prevenire problemi di concorrenza



# **COSTRUTTORE DELLA CLASSE CONNESSIONEDATABASE**

```
private ConnessioneDatabase() throws SQLException {  
    try {  
        Class.forName(driver);  
        connection = DriverManager.getConnection(url, nome, password);  
    } catch (ClassNotFoundException ex) {  
        System.out.println("Database Connection Creation Failed : " + ex.getMessage());  
        ex.printStackTrace();  
    }  
}
```

- La connessione avviene tramite il protocollo JDBC, che è un protocollo standard per la connessione tra un programma Java e un database
- Il caricamento del driver del database dinamicamente dal codice ci faciliterà se vorremo cambiare tipo di database in futuro



# CONNESIONE DATABASE : METODO GETINSTANCE

```
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.connection.isClosed()) {  
  
        // altrimenti restituisco il riferimento a quella esistente  
  
        instance = new ConnessioneDatabase();  
  
    } return instance;  
  
}
```

- Il metodo **getInstance** 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



# OTTENERE CONNESSIONE DATABASE

```
private Connection connection;  
  
public GiocatoreImplementazionePostgresDAO() {  
    try {  
        connection = ConnessioneDatabase.getInstance().connection;  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```

- Da una delle classi che realizzeremo per effettuare query potremo ottenere un riferimento all'oggetto connection per connetterci al database chiamando  
**ConnessioneDatabase.getInstance().connection**



# ESEMPIO DI UTILIZZO DI CONNESSIONEDATABASE

- In una classe che deve eseguire interrogazioni

```
private Connection connection;

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

}

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



# METODI FONDAMENTALI DELL'OGGETTO CONNECTION

- <https://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html>
- **PreparedStatement `prepareStatement(String sql)`**  
**throws SQLException**
  - Istanzia un oggetto della classe PreparedStatement tramite il quale sarà possibile inviare al database per l'esecuzione la query sql scritta nella stringa in input
- **void `close()` throws SQLException**
  - Chiude la connessione al database. In questo modo non sarà più possibile effettuare query tramite questo oggetto connection
  - Il database potrà però accettare altre connessioni dal programma stesso oppure da altri programmi



# QUERY DI INSERIMENTO CON JDBC

- Dato un oggetto connection, con prepareStatement istanziamo un oggetto Prepared Statement sulla base di un query di insert scritta in SQL
- Con executeIpdate() inviamo la query al db dove verrà eseguita

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

In quest'esempio i valori da scrivere sono stati aggiunti direttamente nella stringa SQL (soluzione semplice ma poco robusta in caso di errori)



# QUERY DI AGGIORNAMENTO CON JDBC

- Come nel caso precedente, stavolta si tratta di una query di UPDATE

```
PreparedStatement updatePrezzoPS = connection.prepareStatement(  
    "UPDATE \"Societa\" " + "SET \"PrezzoAzione\" = "+nuovoPrezzo+  
    "WHERE \"Nome\" = '"+nomeSocieta+" ;");  
  
updatePrezzoPS.executeUpdate();
```

- Anche questa volta i valori sono stati scritti direttamente nella stringa: soluzione valida ma poco robusta agli errori



# QUERY DI SELEZIONE CON JDBC

- Le query di selezione (SELECT) forniscono come output un risultato strutturato sotto forma di un oggetto **ResultSet**

```
PreparedStatement leggiListinoPS;
```

```
try {
```

```
    leggiListinoPS = connection.prepareStatement("SELECT *\nFROM \"Societa\" WHERE \"CittaBorsa\"=\""+cittaBorsa+"\"");
```

```
ResultSet rs = leggiListinoPS.executeQuery();
```

- Come possiamo estrarre le informazioni dall'oggetto ResultSet?



# OGGETTO RESULTSET

- La classe ResultSet offre l'accesso ai risultati letti, una riga per volta. Appena eseguita la query abbiamo un puntatore (rs nell'esempio) alla prima riga dei risultati.
- **String s = rs.getString("Nome")** ci permette di leggere nella riga puntata il dato relativo alla colonna chiamata "Nome ", di interpretarlo come una stringa e salvarlo in s
- **Double d = rs.getDouble("PrezzoAzione")** ci permette di leggere nella riga puntata il dato relativo alla colonna chiamata "PrezzoAzione", di interpretarlo come un double e salvarlo in d
- **rs.next()** ci restituisce false se non ci sono più righe, altrimenti sposta il puntatore alla riga successiva



# UTILIZZO DELL'OGGETTO RESULTSET

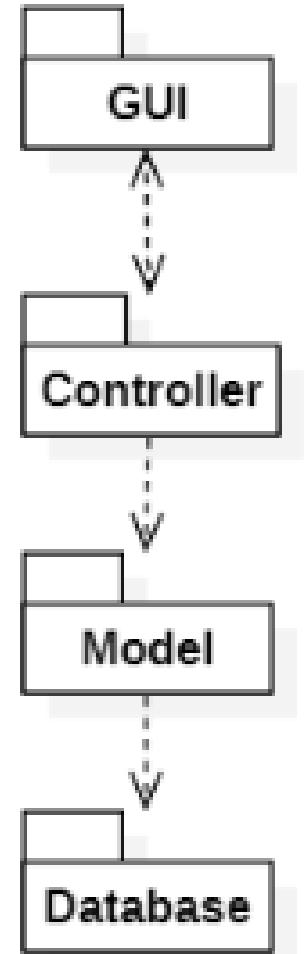
- Per leggere tutti i valori di un ResultSet corrispondente ad una query che restituisce un valore di tipo Stringa (Nome) e uno di tipo Double (PrezzoAzione) e salvarli in due ArrayList chiamati nomiSocieta e prezziSocieta scriviamo

```
while (rs.next()) {  
    nomiSocieta.add(rs.getString("Nome"));  
    prezziSocieta.add(rs.getDouble("PrezzoAzione"));  
}  
  
rs.close();
```



# ARCHITETTURA DEL PROGETTO CON IL DATABASE : SOLUZIONE BCED

- 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)
  - In questa soluzione:
    - Boundary chiama solo Controller, per chiedere servizi
    - Controller chiama Entity, per avere i dati (transienti) dal Model
    - Model chiama Database per avere i dati persistenti
  - Questa soluzione è sicuramente generale perchè consente, rispetto alla precedente BCE di gestire anche i dati persistenti



# PREGI E DIFETTI DELLA SOLUZIONE BCED

- 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
  - Inefficienza: per qualsiasi operazione (anche la lettura di un dato appena letto) dovremo passare per una query al database
    - Le query al database vanno sempre centellinate per due ragioni:
      - Il tempo di lettura dal database è parecchi ordini di grandezza più lento rispetto ad una lettura dalla memoria
      - Il database può essere utilizzato in concorrenza da tantissimi utente. Si rischia di creare delle code di attesa che riducono ulteriormente il tempo di attesa e lo rendono molto variabile
  - I difetti presentati ci consigliano di **EVITARE** la soluzione BCED in favore di una soluzione ibrida BCE + DAO



# PATTERN DAO

- DAO sta per Data Access Object
- Si tratta di un pattern nel quale separiamo :
  - La descrizione dei metodi di accesso ai dati persistenti
    - Li indicheremo in una serie di Interface che inseriremo in un package chiamato dao
  - L'implementazione dei metodi di accesso dichiarati in DAO
    - L'implementazione dipende dalla tipologia di database e dalla sua tecnologia
    - Noi andremo a creare un package chiamato **implementazioniPostgresDAO**



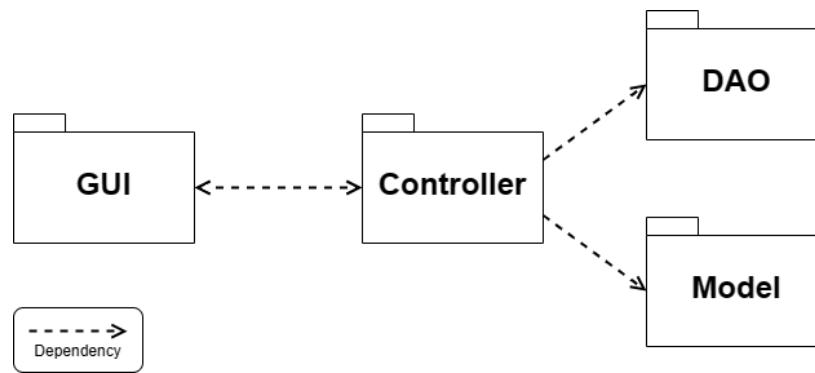
## PREGI DEL PATTERN DAO

- Il pregio maggiore del pattern DAO è quello di separare i requisiti (responsabilità richieste) al database dalla sua implementazione specifica
  - Nulla vieta che un package implementazione possa realizzare letture e scritture su file anzichè su database sql
- Essendo dao un semplice package di interface, senza codice da eseguire, non c'è alcun rallentamento in fase di esecuzione
- Per comodità, l'apertura della connessione al database, comune a tutte le classi di un package implementazioneDAO la manterremo in una classe di utilità apposite

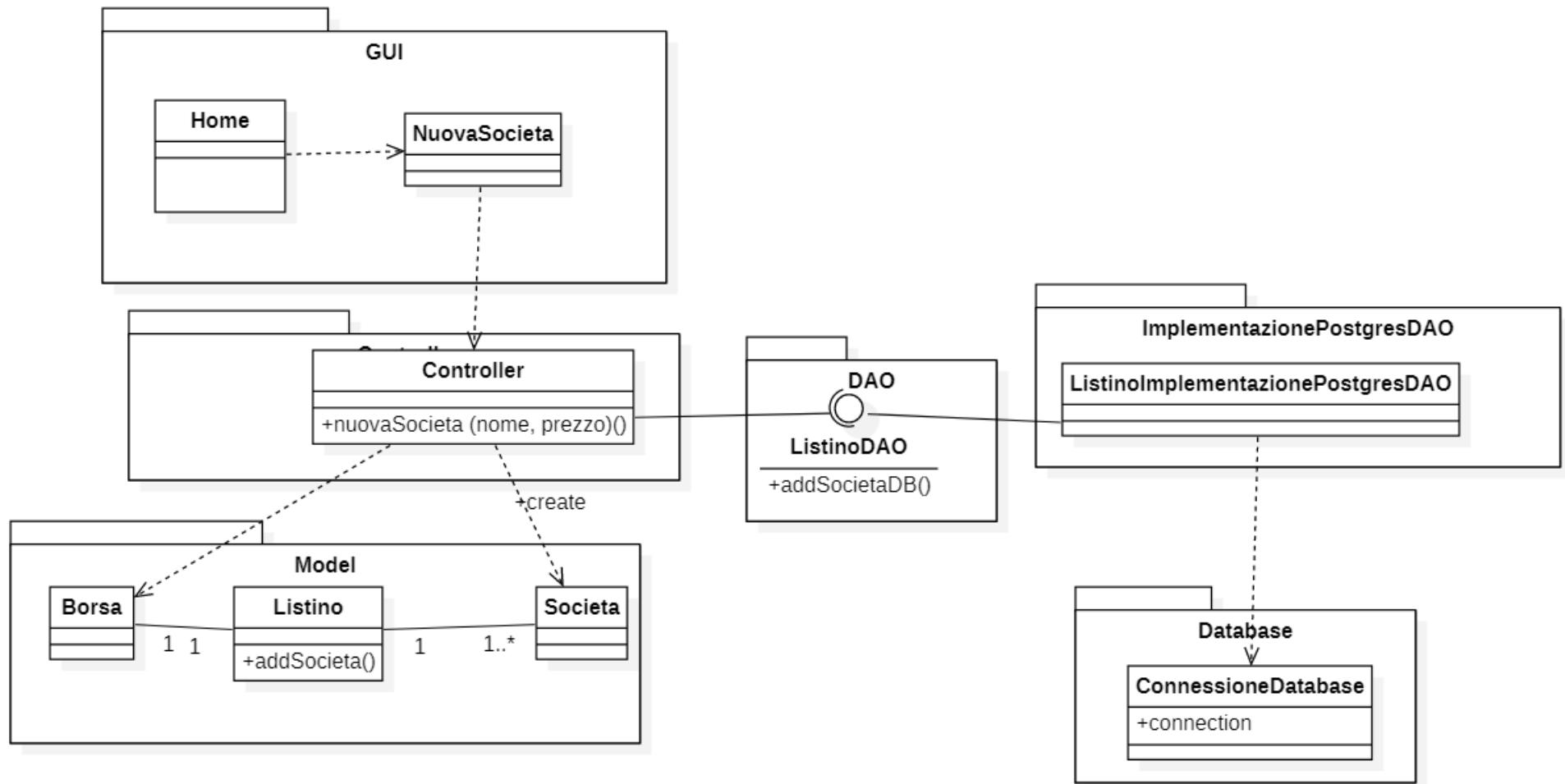


# PATTERN BCE + DAO

- Il pattern adottato verrà chiamato BCE + DAO
- In esso :
  - La GUI (boundary) richiede l'esecuzione di funzionalità al controller (come nel BCE)
  - Il controller interroga il model quando ha bisogno di leggere o scrivere dati in memoria (transienti), come nel BCE
  - Il controller scrive o interroga il database tramite i metodi descritti nelle interfacce presenti nel dao
    - I dati ottenuti dal controller tramite dao potrebbero poi essere salvati in memoria istanziando oggetti del model



# ESEMPIO DI ARCHITETTURA BCE + DAO



# ESEMPIO DI ARCHITETTURA BCE + DAO

- 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



## PREGI E DIFETTI DI BCE + DAO

- Con questo pattern il controller può decidere se accedere a dati transienti, persistenti o entrambi
- L'accesso ai dati transienti (model) è molto più rapido
- L'accesso ai dati persistenti (dao) è l'unico che ci consente di non perdere le informazioni al termine dell'esecuzione
  - In alcuni casi il controller dovrà scrivere sia su model che tramite dao
- I principi da seguire per avere un programma sicuro e robusto sono quelli di :
  - Usare sempre model
  - Usare dao il meno possibile
  - Avere funzioni esplicite che utilizzano dao (ad esempio Salva ...)
  - Assicurarsi che non sia possibile uscire dall'applicazione senza aver salvato i dati in forma persistente



# ESEMPIO

- Se stiamo sviluppando un blocco note per salvare testo:
  - Le funzioni di scrittura del testo dovrebbero operare su stringhe memorizzate nel model (più veloci)
  - Deve esserci una funzione Salva che esplicitamente salva il testo tramite dao su database (o file)
  - Dovrebbe esserci una funzione Esci che, prima di chiudere il programma, controlla se esso sia stato scritto su database e, in caso contrario, salva le modifiche (oppure chiede di salvarle tramite un dialog) tramite dao
- Da osservare che alcuni software, come Google Docs, invece, salvano su cloud ogni modifica, senza attendere che venga cliccato il pulsante salva



# OSSERVAZIONI SU BCE + DAO

- 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: AGGIUNTA DI UNA NUOVA SOCIETÀ AL LISTINO

- Tramite un form indichiamo nome e valore delle azioni di una società da aggiungere al listino.
  - Vogliamo che questa nuova società sia aggiunta alla lista delle società del listino nel model
  - Ma vogliamo anche che essa venga aggiunta nel database
- 
- Prima di tutto creiamo nel package dao una interface ListinoDAO con il metodo addSocietaDB

```
public interface ListinoDAO {  
    void addSocietaDB(String text, double parseDouble, String citta);  
}
```

- Poi creiamo nel package implementazioniPostgresDAO una classe ListinoImplementazionePostgresDAO implements ListinoDAO che realizzerà il metodo addSocietaDB



# 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

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

- 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(); //esegue la query sul db  
connection.close(); //opzionale
```

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



# ALTRÒ ESEMPIO : LETTURA DEL LISTINO

- Nella tabella Societa del database ci sono tutte le società del listino
- Per capire se sono nel listino della borsa che ci interessa basta filtrare attorno all'attributo cittaBorsa
- Appena apriamo il programma e scriviamo il nome della città della Borsa, vogliamo leggere tutte le società del listino di quella borsa per poterle salvare nel model

- Nell'interface ListinoDB aggiungiamo il metodo

```
public void leggiListinoDB(String cittaBorsa, ArrayList<String>  
nomiSocieta, ArrayList<Double> prezziSocieta);
```

- Lo implemetneremo nella classe  
ListinoImplementazionePostgresDAO implements ListinoDAO del  
package implementazioniPostgresDAO



# 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: gli viene restituito sotto forma di due ArrayList
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

```
public void leggiListinoDB(String cittaBorsa, ArrayList<String> nomiSocieta,  
ArrayList<Double> prezziSocieta) {  
  
...  
    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;
```



# OSSERVAZIONI

- Abbiamo dovuto introdurre le due ArrayList in ListinoImplementazionePostgresDAO perché non volevamo che ImplementazionePostgresDAO dipendesse da Model
  - si sarebbero potuti utilizzare anche altri container più complessi
- Quest'indipendenza rispetta l'architettura che ci siamo imposti
- Ci consente di poter cambiare eventualmente il Model senza dover cambiare l'implementazione del DAO
- Non potevamo restituire un riferimento al recordset rs direttamente al controller perchè il recordset deve essere chiuso da ListinoImplementazionePostgresDAO che lo ha aperto
  - Altrimenti, il controller poteva non chiuderlo e il database rischiava di rimanere bloccato ad altre operazioni ed utenti per la presenza del recordset pendente



# 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



# ALTRO ESEMPIO : ACQUISTO DI AZIONI

- Nella tabella Societa del database ci sono tutte le società del listino
- Per capire se sono nel listino della borsa che ci interessa basta filtrare attorno all'attributo cittaBorsa
- Appena apriamo il programma e scriviamo il nome della città della Borsa, vogliamo leggere tutte le società del listino di quella borsa per poterle salvare nel model
- Aggiungiamo una interface GiocatoreDB con il metodo

```
void acquistaDB(String nome, Double valoreAzione, int quantita,  
String societa);
```
- Lo implementeremo nella classe  
`GiocatoreImplementazionePostgresDAO implements GiocatoreDAO`  
del package `implementazioniPostgresDAO`



# ESEMPIO: ACQUISTO DI AZIONI – FRAMMENTI DI CODICE

- In GUI.AcquistaAzione
  - controller.acquistaAzione(nomeSocieta, valoreAzione, quantita);
- In Controller.acquista
  - GiocatoreDAO gDAO=new GiocatoreImplementazionePostgresDAO();
  - gDAO.acquistaDB(giocatore.getNome(), valoreAzione, quantita, societa); *//su DB*
  - giocatore.acquistaAzione(societa, valoreAzione, listino, quantita);  
*//in memoria*
  - g.calcolaCapitale(); *//con dati in memoria*
- 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



# COME GESTIRE LE ECCEZIONI DEL DATABASE?

- Nell'esempio per semplicità le eccezioni del database sono state gestite nelle classi del package `implementazionePostgresDAO`
- In realtà una eccezione del database corrisponde ad una mancata lettura o scrittura, che ha conseguenze sull'elaborazione effettuata dal Controller
- In generale è più corretto se le classi del database redirigano (`throws`) le eccezioni verso il Controller, che può gestirle al meglio
- Ad esempio se l'inserimento nel database di un dato fallisce (ad esempio perché il record già esiste), il Controller dovrà evitare di aggiungerlo nel Model e comunicare alla GUI che l'inserimento non è avvenuto per questa ragione



# 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

