Programmazione III

Luca Barra

Anno accademico 2023/2024

Indice

Capitolo 1

Introduzione

Il corso mirà allo sviluppo di applicazioni di grande portata. Questo implica l'insegnamento della programmazione a eventi, alle interfacce grafiche (SWING e JAVAFX/FXML), la programmazione parallela (processi e thread) e la programmazione in rete (socket).

1.1 La progettazione a oggetti

Definizione 1.1.1: Oggetti

Bisogna capire i tipi di **entità** da rappresentare, le azioni e il modo in cui interagiscono e comunicano. Il mondo è costituito da oggetti.

Esempio 1.1.1 (Simulatore di guida)

In un simulatore di guida bisogna:

- modellare i semafori (per regolare il traffico);
- modellare le macchine (accese, spente, in movimento);
- modellare la strada (passiva);
- non si modellano i cani che attraversano la strada.

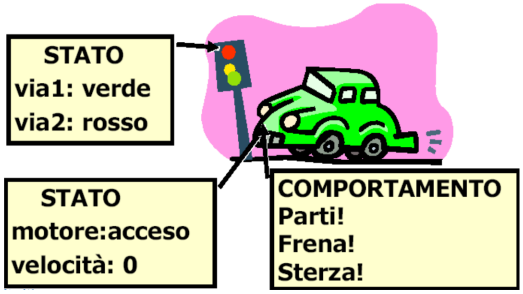
Note:-

Bisogna modellare solo le entità ritenute interessanti.

Definizione 1.1.2: Stato e comportamento

Ogni oggetto ha uno **stato** (che è costituito da i suoi attributi) e un **comportamento** (che è modellato come un insieme di metodi). Il comportamento va a modificare lo stato degli oggetti.

Figure 1.1: Esempio di progettazione a oggetti



1.1.1 Oggetti e realtà

Ogni individuo ha una visione limitata della realtà con una propria identità, uno stato e un comportamento diverso.

Definizione 1.1.3: Incapsulamento

Gli stati si basano sul principio dell'**incapsulamento**: uno stato "appartiene" a un oggetto, per cui un utente esterno non può manipolarlo.

Definizione 1.1.4: Delega

I comportamenti si basano sul principio della **delega**: chi fa la richiesta non vuole conoscere in dettaglio come sia eseguita.

Definizione 1.1.5: Un programma

Un programma viene visto come un insieme di oggetti che comunicano l'un l'altro invocando metodi. Un oggetto può contenere riferimenti ad altri oggetti. Ogni oggetto ha un tipo (**classe**).
Una struttura dati è vista come un insieme di operazioni. Per esempio, una **lista** (astratta) è una sequenza ordinata di dati che possono essere letti sequenzialmente e in cui si può inserire/rimuovere un dato in una posizione i.

Corollario 1.1.1 Object-oriented design.

La progettazione orientata agli oggetti. Si mettono insieme sistemi software visti come collezioni di oggetti.

1.1.2 Programmazione procedurale vs. Programmazione object-oriented

In questa sezione è presente un breve confronto.

Definizione 1.1.6: Programmazione procedurale

La **programmazione procedurale** si concentra sull'organizzare le procedure che operano sui dati. Il suo paradigma è: eseguire una sequenza di passi per raggiungere il risultato.

Note:-

Il programma viene visto come: ALGORITMI + STRUTTURE DATI

Definizione 1.1.7: Programmazione object-oriented (O-O)

La **programmazione object-oriented** si concentra sulle entità incapsulando dati e operazioni. Il suo paradigma è legato a responsabilità e deleghe.

Note:-

Il programma viene visto come: OGGETTI (DATI + ALGORITMI) + COLLABORAZIONE (INTERFACCE)

1.2 Sviluppare a oggetti

Si vedono gli oggetti come fornitori di servizi. Ogni oggetto svolge un piccolo servizio, ma tutti insieme forniscono un grande servizio.

1.2.1 Come si fa?

Si vanno a vedere i sostantivi/nomi che vengono utilizzati perchè diventeranno classi. I verbi andranno a indicare le azioni e i metodi.

Definizione 1.2.1: Classi e istanze

Una **classe** è un'idea astratta che rappresenta caratteristiche comuni a tutte le istanze di un oggetto.
Un'**istanza** è un singolo oggetto "concreto".
Un'istanza ha:

- un'identità;
- uno stato;
- un comportamento.

Note:-

Dobbiamo chiederci:

- quali sono le entità fondamentali da modellare e quali sono i dati di cui abbiamo bisogno?;
- di quante istanze, per ogni concetto, abbiamo bisogno?.

Definizione 1.2.2: Interfacce e implementazioni

L'**interfaccia** è la *firma* dei metodi, ossia la "vista esterna". L'**implementazione** è la "vista interna", come è fatto un metodo.

Note:-

Oltre ancora bisogna capire, di volta in volta, quale tipo di servizio va offerto e mostrato al mondo. Alcuni dati vanno bene pubblici, altri devono essere privati.

Definizione 1.2.3: Modularità

Un'altra componente è la **modularità** cioè la suddivisione in una serie di componenti indipendenti.

Definizione 1.2.4: Gerarchie

Infine si hanno le gerarchie:

- part-of hierarchy: gerarchia di parti;
- kind-of hierarchy: gerarchia di classi e sotto-classi.

Esempio 1.2.1 (Vantaggi di un approccio O-O)

- ✓ Riuso e maggiore leggibilità;
- ✓ Dimensioni ridotte;
- ✓ Compatibilità e portabilità;
- ✓ Estensione e modifica più semplici;
- ✓ Manutenzione del software semplificata;
- ✓ Migliore gestione del team di lavoro.

Note:-

Nella programmazione O-O occorre conoscere l'interfaccia di una classe, ma non necessariamente la sua implementazione.

Capitolo 2

Ripasso di programmazione II

In questa sezione si andranno a ripassare e approfondire alcuni argomenti del corso di programmazione II come:

- L'ereditarietà;
- Estensioni di classi;
- Polimorfismo;
- Downcasting e upcasting;
- Overriding;
- Classi astratte;
- Interfacce.

2.1 L'ereditarietà

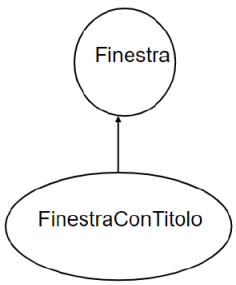
Definizione 2.1.1: Ereditarietà

L'**ereditarietà** è un meccanismo della programmazione a oggetti che consente di espandere alcune classi aggiungendo attributi e/o metodi.

Corollario 2.1.1 Sottoclassi

Le **sottoclassi** ereditano tutti i componenti della propria sovraclasse (variabili e metodi).

Figure 2.1: Esempio di ereditarietà



Note:-

In Java l'ereditarietà è singola, per cui ogni classe ha un solo genitore[a].

[a] E ogni classe discende dalla classe Object

2.2 Tipi e metodi

Definizione 2.2.1: Controllo dei tipi

Java effettua un controllo statico per i tipi (prima dell'esecuzione). Il **checking** controlla che per una variabile si chiami un metodo definito per la classe di quella variabile.

Definizione 2.2.2: Polimorfismo

Un oggetto può avere più di un tipo. Per esempio un oggetto di tipo E che è figlio di un oggetto di tipo C ha entrambi i tipi (E e C). Dato il tipo di una varabile x (A) e un espressione di tipo (B), x = expr è legale se e solo se A = B oppure se B è una sottoclasse di A.

Corollario 2.2.1 Upcasting e downcasting

L'**upcasting** è un movimento da un tipo specifico a uno più generico. Questo assegnamento è sempre legale, per esempio, tutti i cani (specifico) sono animali (generico) oppure tutti i rettangoli (specifico) sono poligoni (generico). Se si effettua un upcasting non si possono più utilizzare i metodi della sottoclasse.
Il **downcasting** è l'operazione opposta.

Corollario 2.2.2 Overriding

L'**overriding** permette a una sottoclasse di sovrascrivere un metodo di una sovraclasse. Per fare ciò si scrive nella sottoclasse un metodo con una firma uguale a un metodo della sovraclasse e si cambia il corpo. Un classico esempio è la funzione toString.

Note:-

Di default toString restituisce il nome della classe + @ + codici alfanumerici

Corollario 2.2.3 Super

Si può usare il codice della classe genitore nella classe figlio mediante la classe **super**. Normalmente se si vuole utilizzare super lo si deve fare come prima cosa. Se non esiste una classe super nel genitore si può causare un loop infinito.

Definizione 2.2.3: Visibilità

- **private**: si vede solo all'interno della classe;
- **protected**: visibile da classi e sottoclassi nello stesso package;
- **public**: visibile da tutti.

Definizione 2.2.4: Binding dinamico

Nel **binding dinamico** si crea un legame durante l'esecuzione. Questo avviene in quasi tutti i linguaggi a oggetti (eccezione C++). In C++ si deve ricorrere all'upcasting. In java non è presente il binding dinamico con le variabili.

2.3 Programmare con l'ereditarietà

Per ricapitolare, i linguaggi a oggetti:

- hanno una struttura modulare;
- implementano tipi di dati astratti;
- offrono gestione automatica della memoria (garbage collector);
- hanno classi;
- ereditarietà singola o multipla;
- polimorfismo e binding dinamico.

Definizione 2.3.1: Riuso del software

Il programmare a oggetti rende possibile riusare il software:

- con il **contenimento** si definiscono nuove classi i cui oggetti sono già compresi in altre classi. Per esempio l'**automobile** ha un **motore**, ha delle **ruote**, etc.;
- con l'**ereditarietà** si estendono delle classi già esistenti. Per esempio un **poligono** può essere un **triangolo**, un **parallelogramma**, etc.

Definizione 2.3.2: Classi astratte

Alcuni classi possono essere **astratte** per cui non è necessario implementare il codice di un metodo in cui si specifica solo la firma. Questi metodi estratti servono da interfaccie di metodi usati dalle sottoclassi. Le classi astratte hanno un **costruttore**, ma non possono essere istanziate.

Definizione 2.3.3: Interfacce

Le **interfacce** sono strutture simili a delle classi, ma possono contenere solo metodi astratti.

Note:-

Un programma può implementare più di un'interfaccia.

2.3.1 Reflection

Definizione 2.3.4: Reflection

La reflection consiste nell'interrogare un oggetto per accertarne alcune caratteristiche.

Corollario 2.3.1 instanceof

Per essere sicuri che la classe di un oggetto, a runtime, sia corretta si usa la instanceof. instanceof restituisce true se l'oggetto è istanza di un certa classe, false altrimenti.

Note:-

instanceof è un particolare tipo di reflection.

Definizione 2.3.5: La classe Class

In java la classe Class contiene tutte le classi C usate in un programma. Rappresenta il *tipo* di un oggetto.

Corollario 2.3.2 isInstance

isInstance è un metodo di Class che funziona come una versione dinamica di instanceof.

Corollario 2.3.3 getClass

getClass è un metodo che restutuisce la classe dell'oggetto su cui è invocato.

Corollario 2.3.4 getName

getName è un metodo che restutuisce, come stringa, il nome dell'oggetto su cui è invocato.

Corollario 2.3.5 forName

forName è un metodo che carica una classe.

Corollario 2.3.6 getSuperclass

getSuperclass è un metodo che restutuisce la sopraclasse dell'oggetto su cui è invocato.

Corollario 2.3.7 newInstance

newInstance è un metodo che crea un nuovo oggetto con la stessa classe dell'oggetto su cui è invocato.

Note:-

newInstance non viene mai usato, perchè si preferisce usare "new"

.

Definizione 2.3.6: java.lang.reflect

Il package java.lang.reflect contiene le classi Field, Methods e Constructor.

Class contiene:

- getFields: restituisce un array con i campi della classe su cui è invocato;
- getMethods: restituisce un array con i metodi della classe su cui è invocato;
- getConstructor: restituisce un array con i costruttori della classe su cui è invocato.

Methods contiene:

- getParameterTypes;
- invoke.

2.4 Trattamento delle eccezioni

Durante l'esecuzione di un programma possono verificarsi degli errori.

- errori di programmazione;
- dati errati in ingresso.

Note:-

Ci vuole una separazione tra la gestione degli errori e i risultati dei metodi.

Definizione 2.4.1: Le eccezioni

Il meccanismo delle eccezioni serve per gestire gli errori veri e propri e anche i casi straordinari.

Corollario 2.4.1 Soluzione banale

La prima soluzione che si impara è quella di restituire un valore riservato che indica il successo o il fallimento.

Note:-

Tuttavia non sempre questo è possibile.

Definizione 2.4.2: Throw, try e catch

Il costrutto throw serve per lanciare le eccezioni. Il costrutto try serve per eseguire istruzioni che potrebbero lanciare eccezioni e catturarle con il costrutto catch (exception handler).

Note:-

Le eccezioni hanno un determinato tipo (sono oggetti throwable[a]). Inoltre gli errori hanno un campo message che specifica il perchè l'errore è avvenuto.

[a] Errori irreparabili o eccezioni

Definizione 2.4.3: Finally

Il costrutto finally è sempre eseguito (anche se non sono sollevate eccezioni).

Esempio 2.4.1 (Chiusura di un file)

Le modifiche a un file non sono permanenti finchè non si chiude. In questo caso è utile utilizzare il costrutto finally per chiudere il file sia nel caso in cui non si siano verificate eccezioni sia nel caso ne siano state sollevate.

Definizione 2.4.4: Definizione di eccezioni

Si possono definire eccezioni personalizzate che andranno a estendere Exception o RuntimeException.

Note:-

Alcuni suggerimenti:

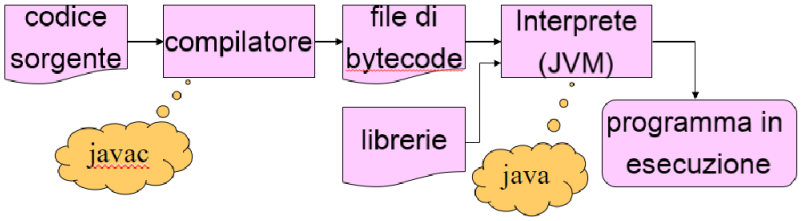
- le eccezioni non devono essere gestite in modo troppo frammentario;
- mettere i catch più specifici per primi e i più generici per ultimi;
- non si devono silenziare le eccezioni;
- se si cattura un errore è preferibile essere severi;
- a volte conviene passare un'eccezione invece di gestirla subito.

2.5 Gestione della memoria

Definizione 2.5.1: Compilazione

La compilazione dei programmi scritti in Java prende in input il codice sorgente e restituisce in output il byte code (eseguibile su differenti S.O.).

Figure 2.2: Come viene compilato un programma Java



Note:-

Alcuni IDE, come IntelliJ, automatizzano questo processo.

Definizione 2.5.2: Memoria della JVM

La memoria della JVM è organizzata in:

⇒ memoria statica: mantiene tutte le parti statiche del programma (alcune variabili, costanti, il codice delle clssi, etc.);

⇒ stack: è gestito come una pila LIFO (Last In First Out), mantiene i record di attivazioni;

⇒ heap: presenta il garbage collector e mantiene i dati creati dinamicamente.

Note:-

I metodi che non hanno bisogno di accedere allo stato di un oggetto vanno dichiarati static

Definizione 2.5.3: Record di attivazione (frame)

I record di attivazione contengono i dati necessari a gestire l'esecuzione di un metodo. Contengono:

- parametri formali;
- varabili locali;
- risultato di ritorno (per metodi non-void);
- l'indirizzo di ritorno.

Definizione 2.5.4: Variabili statiche e di istanza

Variabili statiche: c'è una sola copia di queste variabili ed è condivisa fra tutti gli oggetti di una determinata classe.
Variabili di istanza (o dinamiche): memorizzano lo stato degli oggetti. Ogni oggetto ne ha una copia nel heap.

Capitolo 3

Tipi generici e collezioni

Si vuole poter lavorare in modo "safe" con i tipi di dati senza dover costantemente controllare i tipi di dato.

3.1 Tipi generici

Definizione 3.1.1: Tipi generici

I tipi generici si usano per scrivere codice generico applicabile a più tipi di dati (riusabilità del codice). Il tipo E fa un match con qualunque tipo di dato non primitivo al momento della compilazione.
I generici sono stati introdotti per fare inferenza in fase di type checking statico.

Note:-

Solitamente per i tipi generici si usa la lettera E, ma è solo una convenzione. Qualunque lettera va bene.

Note:-

Si potrebbe usare il tipo Object, ma ciò ha delle limitazioni: per esempio, in un array, possono essere inseriti elementi di tipi diversi. Ovviamente si può usare la reflection, ma ciò è scomodo e inefficiente.

Esempio 3.1.1 (ArrayList)

La classe ArrayList è generica, per cui può contenere oggetti di qualunque tipo. Tuttavia se non si specifica il tipo (ArrayList a = new ArrayList();) verrà considerato Object causando i problemi visti sopra.

Definizione 3.1.2: Tipi parametrici

Un tipo parametrico è una classe in cui è specificato il tipo generico da inferire.

Esempio 3.1.2 (ArrayList parametrico)

ArrayList<Double> a = new ArrayList<Double>;

Note:-

Si possono creare classi generiche mettendo il parametro E nel nome della classe (<E>).

Definizione 3.1.3: Tipo grezzo

Il compilatore non ragiona in termini di tipi generici. Quindi il compilatore li trasforma in tipi grezzi (raw types), ossia unicamente il tipo della classe senza i parametri.

Esempio 3.1.3 (ArrayList)

Quindi:
ArrayList<String> a = new ArrayList<String>
ArrayList<Double> a = new ArrayList<Double>
hanno lo stesso tipo ArrayList.

Note:-

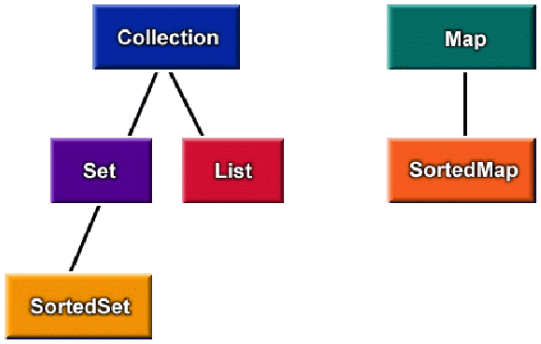
Non si possono avere metodi statici con tipi generici all'interno delle classi che usano quei tipi.

3.2 Collezioni

Definizione 3.2.1: Collezioni

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

Figure 3.1: Le collezioni



Note:-

Queste sono tutte interfacce:

- **Collection**: un arbitrario gruppo di oggetti;
- **List**: un gruppo ordinato di oggetti;
- **Set**: un gruppo di oggetti senza duplicati;
- **Map**: una collezione di coppie chiave-valore.

Definizione 3.2.2: Iteratori

Un iteratore è un oggetto che permette di scorrere una collezione, ottenendo gli elementi uno alla volta. Il metodo iterator<> della classe Collection restituisce un iteratore per la collezione. Si può "ciclare" su una collezione usando un iteratore con next() o con il for each.

Note:-

Gli array mantengono sempre il loro tipo (a runtime), mentre le collezioni no.

Definizione 3.2.3: Il tipo jolly (wildcard)

Per definire una collection di qualunque generico su usa la notazione collection<?>. In una collezione di ? non si può aggiungere nulla, ma si può rimuovere.

Capitolo 4

Classi innestate e lambda expression

Note:-

Le lambda expression e, in generale, il λ-calcolo sono spiegati in dettaglio nei corsi "Linguaggi e paradigmi di programmazione" e "Metodi formali dell'informatica".

4.1 Classi innestate

Le classi possono essere dichiarate:

- all'interno di altre classi in qualità di membri;
- all'interno di blocchi di codice.

Definizione 4.1.1: Classi innestate

Le classi innestate sono utili per:

- definire tipi strutturati visibili all'interno di gruppi correlati;
- connettere in modo semplice oggetti correlati;
- information hiding di tipi di dati.

Il nome di una classe innestata è: NomeContenitore.NomeInnestata.

Note:-

La visibilità di una classe innestata è *almeno* la stessa della classe che la contiene.

Note:-

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

4.2 Lambda expression

Definizione 4.2.1: Lambda expression

Le lambda expression sono utili per implementare interfacce funzionali (cioè interfacce con un solo metodo astratto) in modo compatto:

- possono avere o non avere parametri;
- possono avere o non avere un tipo di ritorno.

Note:-

Se il body ha una sola istruzione e restituisce un valore (non void) si possono omettere le parentesi graffe e la keyword `return`.

4.3 Input/Output

Definizione 4.3.1: I/O

Le operazioni di I/O avvengono attraverso *stream*: successioni di byte che rappresentano i dati in input o in output. Gli stream possono essere combinati. Ci sono due tipi di stream:

- *byte stream*: stream di byte;
- *character stream*: stream di caratteri.

Esistono oltre 60 classi di I/O divise in due gerarchie:

- `InputStream` e `OutputStream` per i byte;
- `Reader` e `Writer` per i caratteri.

Note:-

Sorgente e destinazione di un flusso di byte o di caratteri possono essere:

- `File`: file sul disco;
- `Array`: array di byte o di caratteri;
- `String`: stringa di caratteri;
- `Pipe`: stream di byte o di caratteri in memoria.

Definizione 4.3.2: Input a caratteri

La classe read è la classe astratta che rappresenta un generico stream di caratteri. I metodi più importanti sono:

- `int read()`: legge un carattere e lo restituisce come intero;
- `int read(char[] c)`: legge un array di caratteri e restituisce il numero di caratteri letti;
- `int read(char[] c, int off, int len)`: legge un array di caratteri a partire da un offset e restituisce il numero di caratteri letti.

La classe InputStreamReader è una classe che converte un InputStream in un Reader.

Definizione 4.3.3: Output a caratteri

La classe `Writer` è la classe astratta che rappresenta un generico stream di caratteri. I metodi più importanti sono:

- `void write(int c)`: scrive un carattere;
- `void write(char[] c)`: scrive un array di caratteri;
- `void write(char[] c, int off, int len)`: scrive un array di caratteri a partire da un offset;
- `void flush()`: svuota il buffer di output;
- `void close()`: chiude lo stream.

La classe OutputStreamWriter è una classe che converte un OutputStream in un Writer.

Definizione 4.3.4: Buffer

Un buffer è un'area di memoria temporanea dove vengono memorizzati i dati prima di essere letti o scritti. I vantaggi dell'utilizzo di un buffer sono:

- riduzione del numero di accessi al disco;
- riduzione del tempo di esecuzione.

Note:-

I flussi standard di input e output sono:

- `System.in`: flusso di input standard;
- `System.out`: flusso di output standard;
- `System.err`: flusso di errori standard.

4.3.1 Oggetti

Definizione 4.3.5: I/O di oggetti

Con ObjectInputStream e ObjectOutputStream è possibile leggere e scrivere oggetti. Per poter scrivere un oggetto su un file è necessario che la classe dell'oggetto sia serializzabile, cioè che implementi l'interfaccia `Serializable`. L'interfaccia Serializable è una *marker interface*, cioè un'interfaccia senza metodi.

Note:-

Per leggere e scrivere oggetti su un file si usano i metodi:

- `void writeObject(Object o)`: scrive un oggetto;
- `Object readObject()`: legge un oggetto.

4.3.2 File

Definizione 4.3.6: File

Un file è una sequenza di byte memorizzata su un dispositivo di memorizzazione permanente. Un file è caratterizzato da:

- nome;
- dimensione;
- tipo;
- posizione.

I file possono essere:

- *testuali*: contengono caratteri;
- *binari*: contengono byte.

Note:-

Per leggere e scrivere file si usano le classi:

- `File`: rappresenta un file o una directory;
- `FileReader` e `FileWriter`: leggono e scrivono file di caratteri;
- `FileInputStream` e `FileOutputStream`: leggono e scrivono file di byte.

Definizione 4.3.7: Scanner

La classe `Scanner` permette di leggere file di caratteri o di byte. I metodi più importanti sono:

- `String nextLine()`: legge una riga;
- `String next()`: legge una parola;
- `int nextInt()`: legge un intero;
- `double nextDouble()`: legge un double;
- `boolean hasNext()`: verifica se ci sono altri dati da leggere.

Capitolo 5

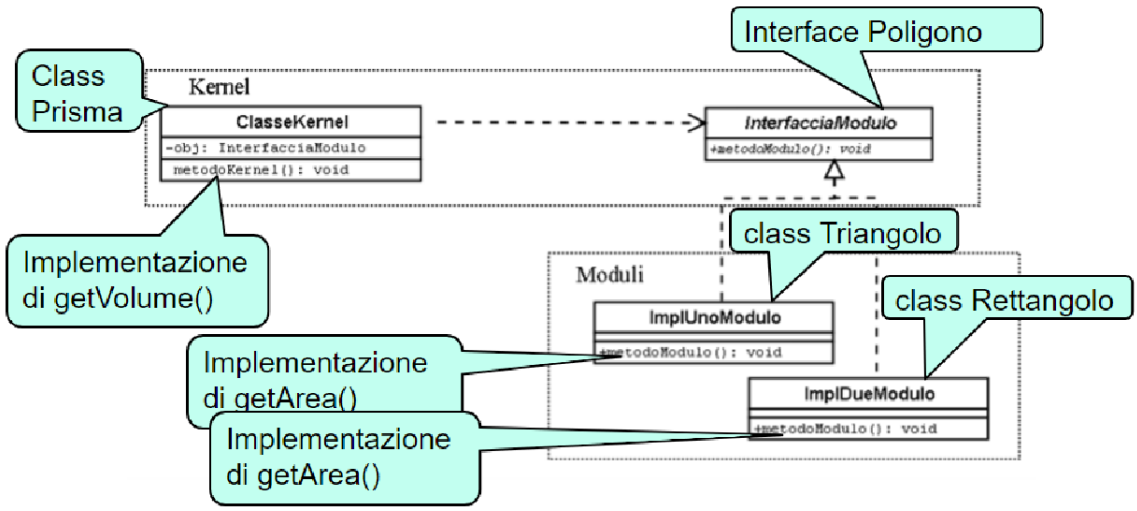
Kernel-modulo

5.1 Verso i pattern architetturali

Definizione 5.1.1: Ereditarietà, parte 2

L'ereditarietà può essere usata per modellare oggetti complessi. Combinando l'uso di interfaccie/classi che implementano le interfacce con relazioni di combinamento si ottengono modelli di programmazione avanzati.

Figure 5.1: La classe prisma



Note:-

Questa separazione tra interfaccia e implementazione permette la compilazione separata. Inoltre, se si cambia l'implementazione di una classe, non si cambia l'interfaccia, quindi non si cambia il codice che usa l'interfaccia.

Capitolo 6

Interfacce grafiche

Definizione 6.0.1: GUI

Una GUI (Graphical User Interface) è un'interfaccia utente che permette l'interazione uomo-macchina in modo visuale utilizzando rappresentazioni grafiche piuttosto che utilizzando una interfaccia a riga di comando.

Note:-

Nelle prime versioni di Java (1.0, 1.1) era presente la libreria AWT (Abstract Window Toolkit) che permetteva di creare interfacce grafiche. Questa libreria era basata sulle API native del sistema operativo.

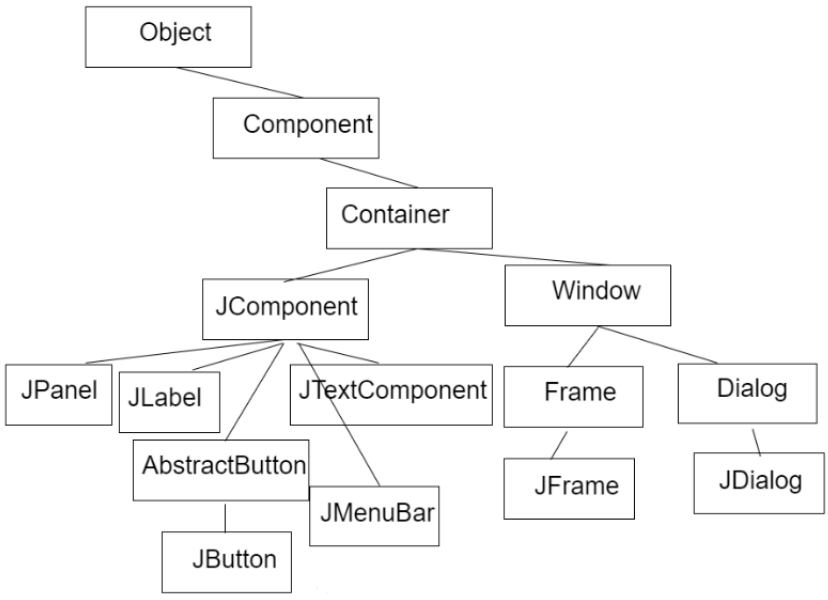


Figure 6.1: Gerarchia delle classi di interfaccia grafica

6.1 SWING

Definizione 6.1.1: JFrame

Un JFrame è un contenitore che permette di creare una finestra. Si possono utilizzare diversi layout. La sintassi è:

- `JFrame frame = new JFrame("Titolo");`
- `frame.setSize(300, 300);`
- `frame.setVisible(true);`
- `JLabel label = new JLabel("Testo");`
- `frame.add(label);`

Definizione 6.1.2: Event-driven programming

L'event-driven programming è un paradigma di programmazione in cui il flusso di esecuzione del programma è determinato dagli eventi che avvengono. Un evento è un segnale che indica che qualcosa è accaduto. Un evento può essere generato da un utente (click del mouse, pressione di un tasto, ...) o da un altro programma.

1. L'applicazione crea gli event-handlers;
2. L'applicazione registra gli event-handlers[a].

[a] Questo significa che ogni event-handler è legato a un tipo di evento.

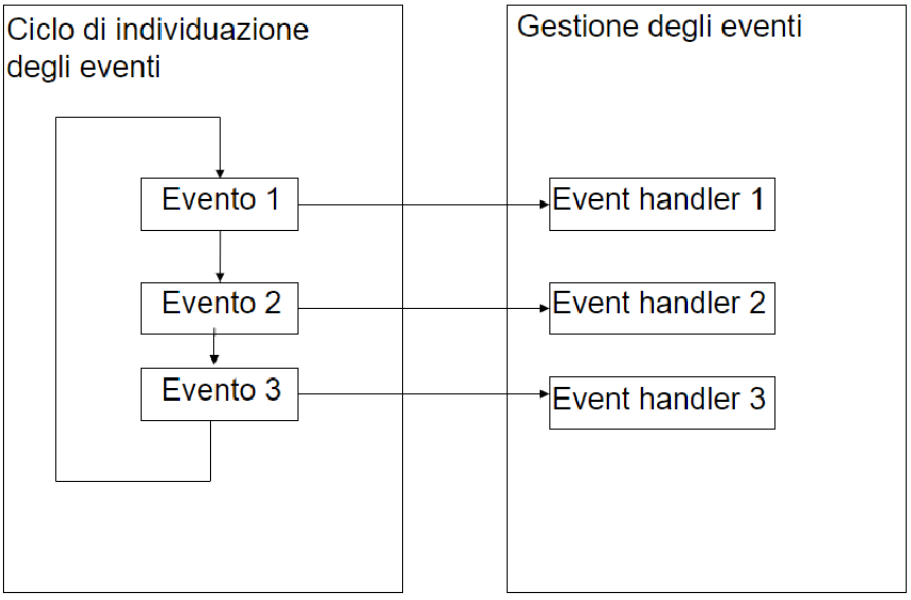


Figure 6.2: Eventi

Note:-

Il ciclo degli eventi è un concetto astratto.

Definizione 6.1.3: Gestione degli eventi

Gli eventi sono passati da un oggetto a un listener che lo gestisce. Il listener deve essere registrato presso la sorgente dell'evento. Il passaggio dell'evento causa l'invocazione di un metodo del listener.

Corollario 6.1.1 Listener

Esistono differenti `interface`:

- ActionListener: gestisce gli eventi generati da un componente che genera azioni (es. JButton);
- MouseListener: gestisce gli eventi generati da un componente che genera azioni del mouse;
- MouseMotionListener: gestisce gli eventi generati da un componente che genera azioni del mouse;
- WindowListener: gestisce gli eventi generati da un componente che genera azioni della finestra (JFrame).

Definizione 6.1.4: Adapters

Gli adapters sono classi che implementano un'interfaccia e forniscono un'implementazione vuota di tutti i metodi. Questo permette di implementare solo i metodi che interessano.

Note:-

Ci possono essere casi con ogni bottone associato al proprio listener. Oppure più bottoni con lo stesso listener.

6.2 Pattern Observe - Observable

Definizione 6.2.1: Pattern Observe - Observable

Il pattern Observe - Observable è un pattern che serve per rendere più modulare il codice e per permettere la comunicazione tra oggetti. Viene usato anche in altri ambiti, ma in questo corso ci occuperemo del suo uso in relazione alla GUI.

Note:-

In Java esistono gli oggetti `Observer` e `Observable` che implementano questo pattern, tuttavia sono deprecate.

Definizione 6.2.2: Observer

L'Observer osserva uno o più oggetti Observable, registrandosi presso di essi.

Definizione 6.2.3: Observable

L'Observable è un oggetto che può essere osservato da uno o più Observer. Quando l'Observable cambia stato, notifica gli Observer.

Note:-

La notifica viene effettuata tramite il metodo `update()`.

Corollario 6.2.1 Metodi di Observer

- `update(Observable o, Object arg)`: viene invocato quando l'Observable cambia stato.

Corollario 6.2.2 Metodi di Observable

- `addObserver(Observer o)`: aggiunge un Observer;
- `deleteObserver(Observer o)`: rimuove un Observer;
- `notifyObservers()`: notifica tutti gli Observer;
- `notifyObservers(Object arg)`: notifica tutti gli Observer con un argomento;
- `deleteObservers()`: rimuove tutti gli Observer.

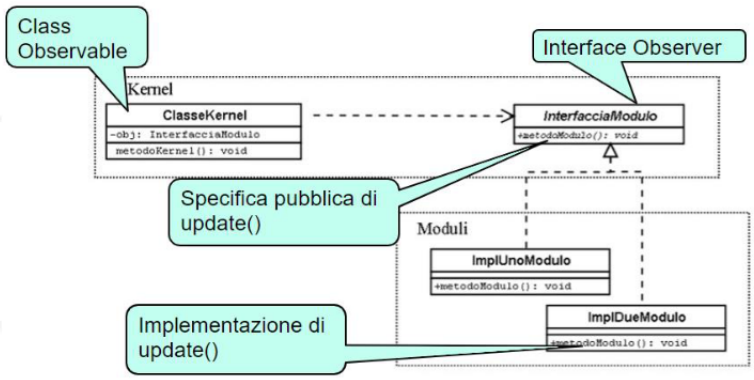


Figure 6.3: Pattern Observer - Observable

Note:-

Le moderne librerie grafiche JavaFX utilizzano questo pattern internamente.

6.3 Pattern MVC

Definizione 6.3.1: MVC

Un programma si compone di:

- Model: rappresenta i dati e le operazioni che possono essere effettuate su di essi;
- View: visualizza i dati e permette l'interazione con l'utente;
- Controller: gestisce gli eventi generati dall'utente e aggiorna il model e la view.

Note:-

Il model è collegato alla view tramite il controller.

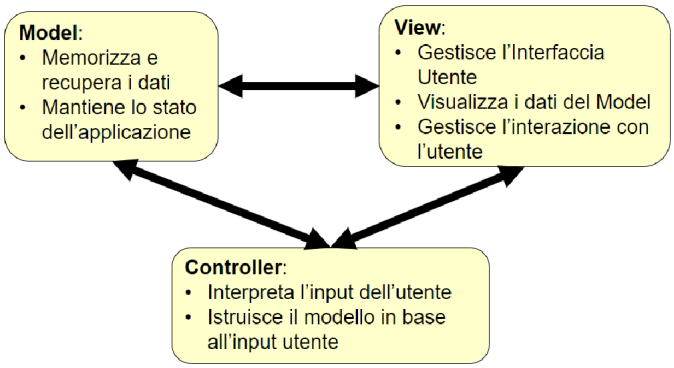


Figure 6.4: Pattern MVC

6.4 XML

6.5 JavaFX

Definizione 6.5.1: JavaFX

JavaFX è una libreria grafica per lo sviluppo di GUI:

⇒ per usarla bisogna aver compreso SWING;

⇒ separa il contenuto dalla sua visualizzazione tramite fogli di stile CSS;

⇒ permette il binding di properties dei Model;

⇒ offre classi/interfacce che implementano Observer/Observable;

⇒ permette di creare interfacce grafiche tramite XML.

Corollario 6.5.1 Componenti di JavaFX

- `Stage`: rappresenta una finestra (simile a JFrame);
- `Scene`: rappresenta il contenuto di una finestra.

La struttura della GUI è gerarchica: nel pannello della scene si inseriscono i componenti figli. In uno stage ci può essere una sola scene.

Definizione 6.5.2: Form (moduli)

In JavaFX si possono anche creare dei form basati su:

- `GridPane`: per inserire i componenti grafici in una griglia;
- `Label`: titoli dei campi;
- `TextField`: campi di input;
- `Button`: pulsanti;
- `Text`: testo non modificabile (output).

Definizione 6.5.3: Uso di CSS

Per utilizzare CSS in JavaFX bisogna:

- Creare un file CSS;
- Creare un oggetto `Scene` e associargli il file CSS;
- Associare la scena allo stage.

6.6 JavaFXML

Definizione 6.6.1: JavaFXML

JavaFXML è un linguaggio di markup basato su XML che permette di creare interfacce grafiche. Permette di separare la struttura della GUI dal codice Java. Permette di creare interfacce grafiche in modo dichiarativo. Permette di utilizzare CSS. Permette di utilizzare il pattern MVC.

6.6.1 Scene Builder

Definizione 6.6.2: Scene Builder

Scene Builder è un tool che permette di creare interfacce grafiche in modo visuale. Permette di creare interfacce grafiche in modo dichiarativo.

6.6.2 Properties

Capitolo 7

Thread

7.1 Introduzione

7.2 Thread in Java

Definizione 7.2.1: Semaphore

La classe Semaphore è stata introdotta per aiutare chi aveva programmato in C. Semaphore(int n) è un semaforo con n permessi. Possiede i metodi:

- acquire() - prende un permesso;
- release() - rilascia un permesso.

Note:-

Ogni oggetto ha un proprio lock: un semaforo binario con una lista che, se utilizzato, blocca gli accessi concorrenti garantendo la mutua esclusione. Lo scheduler del lock ha una propria politica di gestione.

Definizione 7.2.2: Sincronizzazione

Per ogni classe Java è possibile definire dei metodi synchronized. Quando un thread invoca un metodo synchronized, il thread acquisisce il lock dell'oggetto su cui è invocato il metodo. La sintassi è: public synchronized void metodo() {...sezione critica...}

7.3 Sincronizzazione lato server e lato client

Definizione 7.3.1: Sincronizzazione lato server

⇒ L'oggetto protegge le variabili condivise e offre metodi synchronized per operare su di esse per cui si auto- protegge dagli accessi esterni;

⇒ I client, invocando metodi synchronized sull'oggetto condiviso, automaticamente si sincronizzano nell'accesso all'oggetto stesso;

⇒ imitazioni: definire synchronized i metodi dell'oggetto potrebbe non essere sufficiente per sincronizzare le attività dei thread.

Definizione 7.3.2: Sincronizzazione lato client

⇒ L'oggetto non viene protetto da accessi paralleli;

⇒ Tutti i client di un oggetto condiviso accedono all'oggetto attraverso blocchi sincronizzati sul lock dell'oggetto stesso;

⇒ Limitazioni: se un client non implementa correttamente gli accessi all'oggetto condiviso si ottiene un malfunzionamento;

⇒ Ma talvolta è necessario implementare la mutua esclusione in questo modo.

Capitolo 8

Laboratorio

8.1 Lezione 1

8.1.1 La piattaforma IntelliJ

Definizione 8.1.1: IDE

Quando si sviluppa un software (SW) di grandi dimensioni è necessario utilizzare un **IDE**[a].

[a]Integrated development environment

Note:-

L'IDE offre supporto alla compilazione e all'esecuzione dei programmi, a caricarli sul web, etc.

Definizione 8.1.2: IntelliJ

IntelliJ offre un editor per lo sviluppo di applicazioni web e standalone.
Le versioni principali sono due:

- Community: non permette lo sviluppo web;
- ULTIMATE: è la versione completa ed è gratuita per studenti universitari.

Corollario 8.1.1

IntelliJ organizza tutte le applicazioni in progetti (**Project**), ognuno dei quali include:

- *Source Package (src)*: il codice sorgente, ossia le classi java;
- *External library*: le librerie utilizzate;
- altre cartelle.

8.1.2 Installare IntelliJ

1. Come prerequisito bisogna aver installato almeno la versione 13 di JDK (meglio se 20 o successiva);
2. Installare **JetBrains** Toolbox da questo link: https://www.jetbrains.com/toolbox-app/;
3. Avviare JetBrains Toolbox;
4. Cercare e installare **IntelliJ IDEA ultimate**;
5. Avviare IntelliJ IDEA ultimate;
6. Cliccare sui tre puntini in basso a sinistra e selezionare Manage Licenses;

7. Acquisire la licenza di IntelliJ IDEA[1].

Note:-
Per verificare la propria JDK basta eseguire il comando "java –version" da terminale.

8.1.3 Estensioni utili

Breve elenco di plugins che possono migliorare la **quality of life** (QOL).

- Atom Material Icons: un set di icone che rende più "vivace" l'ambiente di sviluppo favorendo visivamente il riconoscimento di file e cartelle;
- CodeGlance Pro: mostra una "**mappa**" del proprio codice a destra dello schermo, permettendo una rapida visione d'insieme e la possibilità di spostarsi precisamente usando l'interfaccia grafica;
- Conventional Commit: fornisce un completamento per commit "standard" su git;
- Key Promoter X: serve per imparare le combinazioni di tasti (**shortcuts**). Ogni volta che si utilizza il menu testuale viene mostrata l'alternativa con la tastiera insieme a un contatore che segna quanti "miss" di quella shortcut sono stati fatti;
- PDF Viewer: permette di visualizzare i file PDF all'interno dell'IDE;
- Rainbow CSV: migliora la lettura dei file CSV colorando i vari campi;
- Rainbow Brackets: migliora la leggibilità del codice colorando le parentesi.

Esempio 8.1.1 (Per iniziare)
Per creare un progetto bisogna aprire il menu File → New → Project. Nel menu che compare si seleziona New Project, con language Java, si definisce il nome del progetto e si clicca su create.
Il progetto nasce con la cartella **src**. Si possono creare classi, package, etc. facendo clic con il tasto destro all'interno della cartella che si vuole usare.
IntelliJ possiede anche utili funzioni che segnalano gli errori e aiutano con la compilazione. Riguardo all'autocompilazione: può essere usata per creare costruttori, getter, setter, toString, etc.
Quando si compila il programma viene creata la cartella **out** che contiene i file *.class* del progetto. Essa viene distrutta e ricreata ogni volta che si compila il progetto in modo da eliminare eventuali problemi di dipendenze.

Note:-
IntelliJ crea e gestisce i progetti in una cartella di default "IdeaProjects".

Note:-
Per convenzione, in un progetto Java, le classi che rappresentano oggetti delle applicazioni vanno inseriti in una cartella **model**, le classi che gestiscono le operazioni di input/output vanno inseriti in una cartella **io**.

8.2 Lezione 2

8.3 Lezione 3

8.4 Lezione 4

[1]Nota: la licenza è fornita gratuitamente agli studenti universitari, ma deve essere rinnovata ogni anno