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.

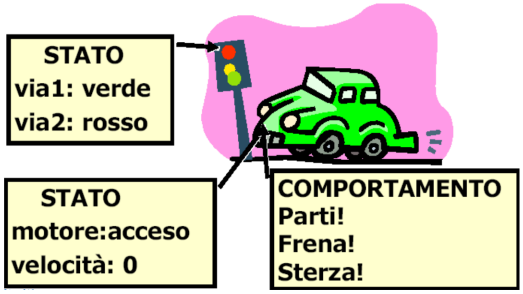
Note:-

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

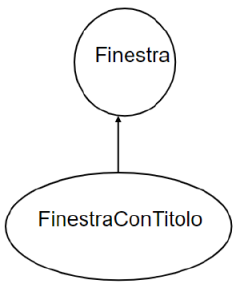
Definizione 2.1.1: Ereditarietà

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:

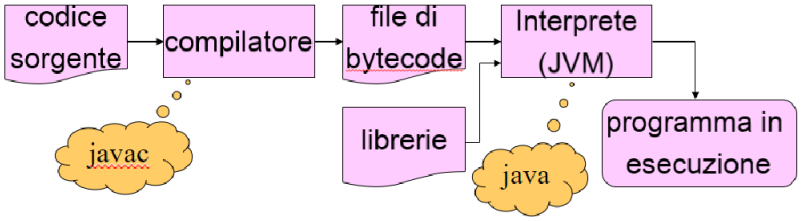
- le eccezioni non devono essere gestite in modo troppo frammentario;
- mettere i catch più specifici per primi e i più generici per ultimi;
- non si devono silenziare le eccezioni;
- se si cattura un errore è preferibile essere severi;
- a volte conviene passare un'eccezione invece di gestirla subito.

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.

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".

Capitolo 5

Interfacce grafiche

5.1 Pattern Observe - Observable

5.2 Pattern MVC

5.3 SWING

5.4 XML

5.5 Java FX

5.6 Java FXML

5.6.1 Properties

Capitolo 6

Laboratorio

6.1 Lezione 1

6.1.1 La piattaforma IntelliJ

Definizione 6.1.1: 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 6.1.2: 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 6.1.1

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.

6.1.2 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.

6.1.3 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 6.1.1 (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**.

6.2 Lezione 2

6.3 Lezione 3

6.4 Lezione 4

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