
ANNO ACCADEMICO 2023/2024

Programmazione III

Teoria

Altair's Notes



DIPARTIMENTO DI INFORMATICA

CAPITOLO 1	INTRODUZIONE	PAGINA 5
1.1	La progettazione a oggetti Oggetti e realtà — 6 • Programmazione procedurale vs. Programmazione object-oriented — 6	5
1.2	Sviluppare a oggetti Come si fa? — 7	7
CAPITOLO 2	RIPASSO DI PROGRAMMAZIONE II	PAGINA 10
2.1	L'ereditarietà	10
2.2	Tipi e metodi	10
2.3	Programmare con l'ereditarietà Reflection — 12	12
2.4	Trattamento delle eccezioni	13
2.5	Gestione della memoria	14
CAPITOLO 3	TIPI GENERICI E COLLEZIONI	PAGINA 17
3.1	Tipi generici	17
3.2	Collezioni	18
CAPITOLO 4	CLASSI INNESTATE E LAMBDA EXPRESSION	PAGINA 21
4.1	Classi innestate	21
4.2	Lambda expression	22
4.3	Input/Output Oggetti — 23 • File — 24	22
4.4	Verso i pattern architetturali	24
CAPITOLO 5	INTERFACCE GRAFICHE	PAGINA 27
5.1	SWING	28
5.2	Pattern Observe - Observable	29
5.3	Pattern MVC	30
5.4	JavaFX	31
5.5	JavaFXML Scene Builder — 32 • Properties — 32	32

CAPITOLO 6	THREAD	PAGINA 35
6.1	Introduzione	35
6.2	Thread in Java	36
6.3	Sincronizzazione lato server e lato client	37
	Librerie — 38	
6.4	Thread Pool	38

Premessa

Licenza

Questi appunti sono rilasciati sotto licenza Creative Commons Attribuzione 4.0 Internazionale (per maggiori informazioni consultare il link: <https://creativecommons.org/version4/>).



Formato utilizzato

Box di "Concetto sbagliato":

Concetto sbagliato 0.1: Testo del concetto sbagliato

Testo contenente il concetto giusto.

Box di "Corollario":

Corollario 0.0.1 Nome del corollario

Testo del corollario. Per corollario si intende una definizione minore, legata a un'altra definizione.

Box di "Definizione":

Definizione 0.0.1: Nome delle definizioni

Testo della definizione.

Box di "Domanda":

Domanda 0.1

Testo della domanda. Le domande sono spesso utilizzate per far riflettere sulle definizioni o sui concetti.

Box di "Esempio":

Esempio 0.0.1 (Nome dell'esempio)

Testo dell'esempio. Gli esempi sono tratti dalle slides del corso.

Box di "Note":

Note:-

Testo della nota. Le note sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive.

Box di "Osservazioni":

Osservazioni 0.0.1

Testo delle osservazioni. Le osservazioni sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive. A differenza delle note le osservazioni sono più specifiche.

1

Introduzione

Il corso mirerà allo sviluppo di applicazioni di grande portata. Questo implica l'insegnamento della programmazione a eventi, alle interfacce grafiche (SWING e JAVA FX/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.

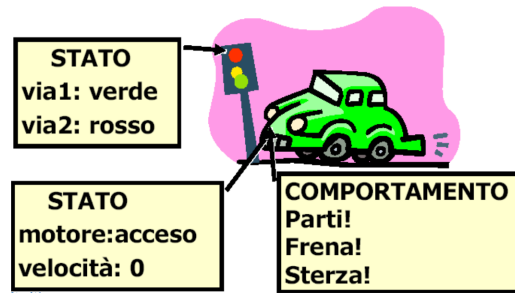
Note:-

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)

- ✓ Riutilizzo 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.

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 sovraclassa (variabili e metodi).

Note:-

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

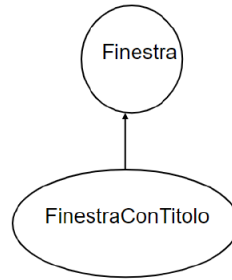
^aE ogni classe discende dalla classe Object

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.

Figure 2.1: Esempio di ereditarietà

**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 variabile x (A) e un'espressione di tipo (B), $x = \text{expr}$ 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 sovraclasses. Per fare ciò si scrive nella sottoclasse un metodo con una firma uguale a un metodo della sovraclasses 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: Riutilizzo del software

Il programmare a oggetti rende possibile riutilizzare 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

Alcune 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 interfacce 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 una 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 restituisce la classe dell'oggetto su cui è invocato.

Corollario 2.3.4 `getName`

`getName` è un metodo che restituisce, 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 restituisce la superclasse 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.

^aErrori 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:

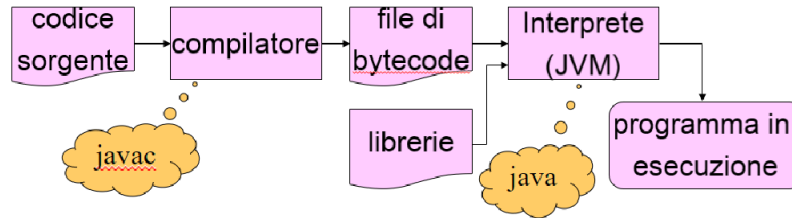
- le eccezioni non devono essere gestite in modo troppo frammentario;
- mettere i catch più specifici per primi e i più generici per ultimi;
- non si devono silenziare le eccezioni;
- se si cattura un errore è preferibile essere severi;
- a volte conviene passare un'eccezione invece di gestirla subito.

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 classi, 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;
- variabili 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.

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:-

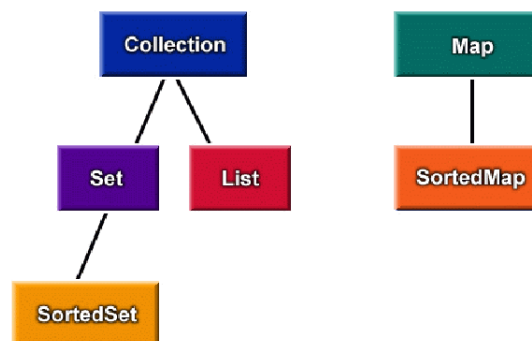
Non si possono avere metodi statici con tipi generici all'interno delle classi che usano quei tipi.

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 si usa la notazione `collection<?>`. In una collezione di `?` non si può aggiungere nulla, ma si può rimuovere.

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.

4.4 Verso i pattern architetturali

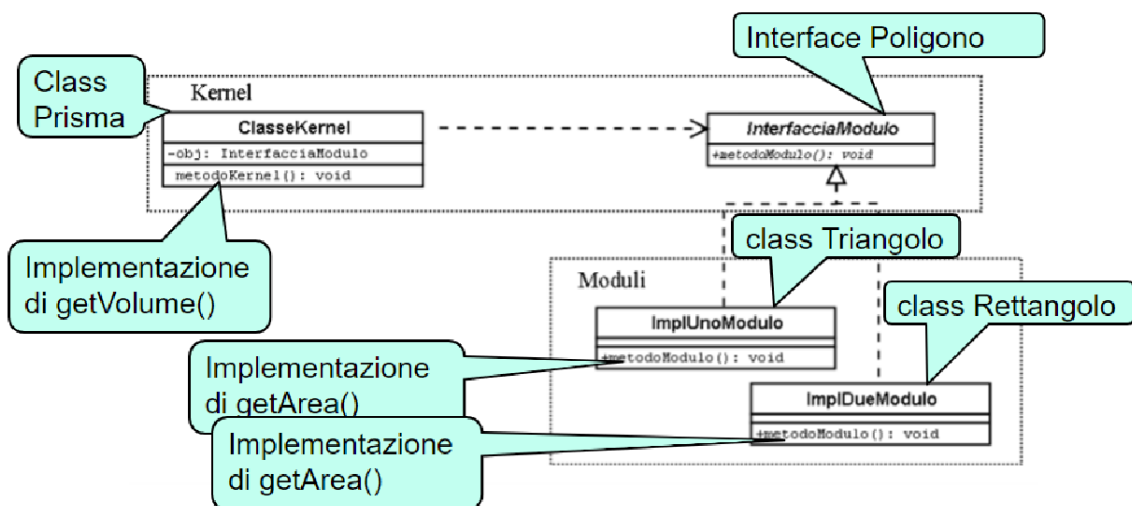
Definizione 4.4.1: Ereditarietà, parte 2

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

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.

Figure 4.1: La classe prisma



5

Interfacce grafiche

Definizione 5.0.1: 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.

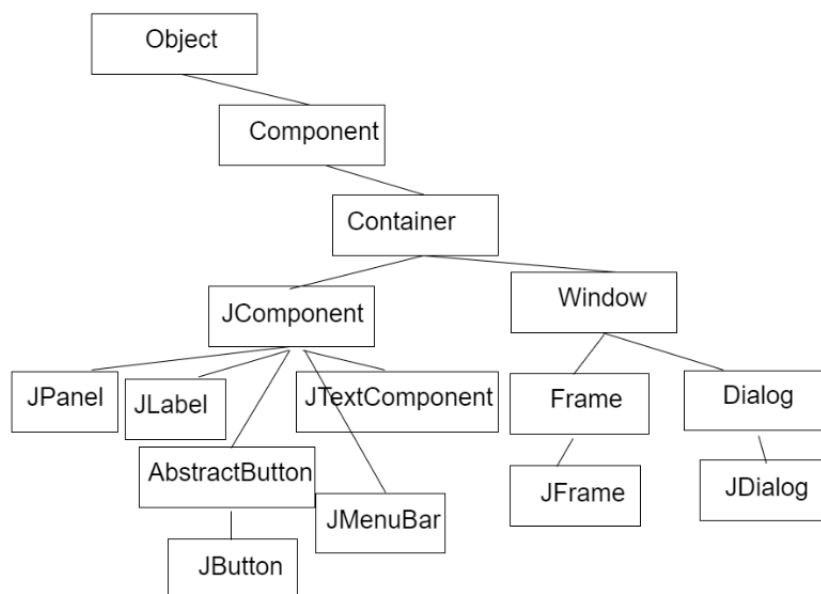


Figure 5.1: Gerarchia delle classi di interfaccia grafica

5.1 SWING

Definizione 5.1.1: 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 5.1.2: 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.

^aQuesto significa che ogni event-handler è legato a un tipo di evento.

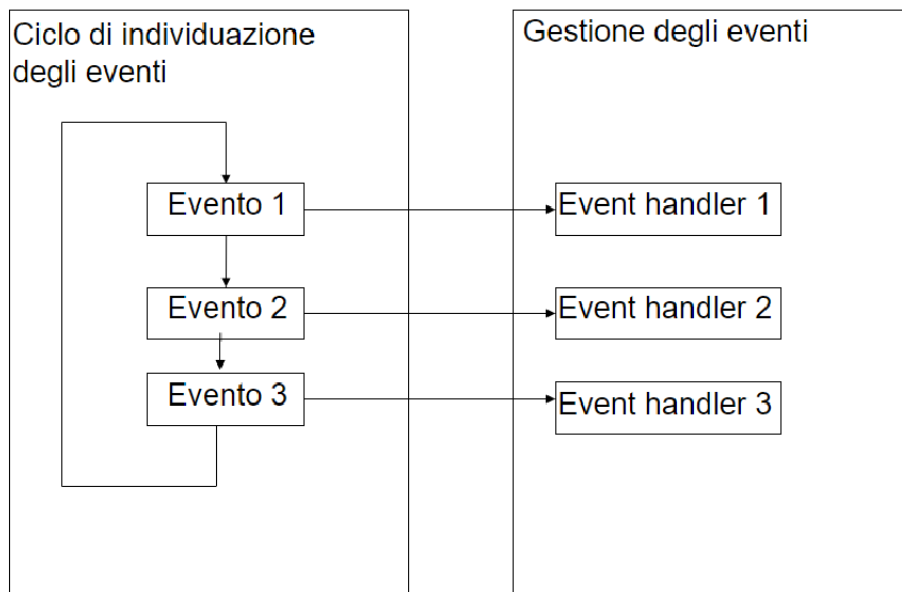


Figure 5.2: Eventi

Note:-

Il ciclo degli eventi è un concetto astratto.

Definizione 5.1.3: 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 5.1.1 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 5.1.4: 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.

5.2 Pattern Observe - Observable

Definizione 5.2.1: 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 5.2.2: Observer

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

Definizione 5.2.3: 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 5.2.1 Metodi di Observer

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

Corollario 5.2.2 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.

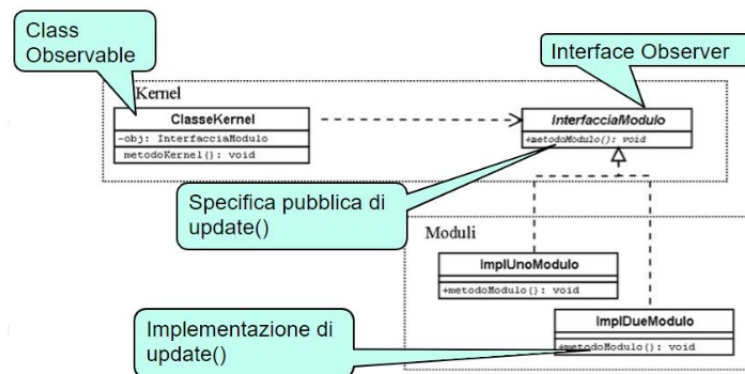


Figure 5.3: Pattern Observer - Observable

Note:-

Le moderne librerie grafiche JavaFX utilizzano questo pattern internamente.

5.3 Pattern MVC

Definizione 5.3.1: 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.

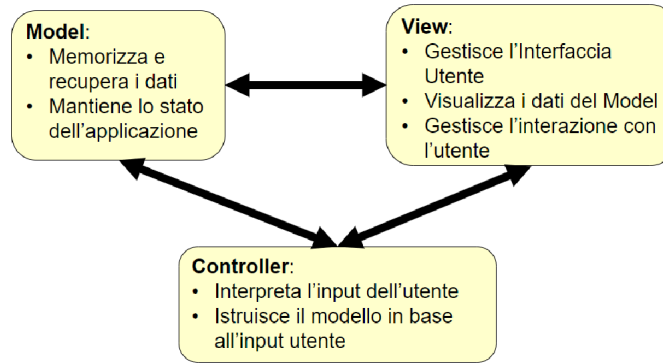


Figure 5.4: Pattern MVC

5.4 JavaFX

Definizione 5.4.1: 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 5.4.1 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 5.4.2: 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 5.4.3: 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.

5.5 JavaFXML

Definizione 5.5.1: 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.

5.5.1 Scene Builder

Definizione 5.5.2: Scene Builder

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

5.5.2 Properties

Definizione 5.5.3: Java beans

Un Java bean è una classe che:

- Ha un costruttore senza argomenti;
- Ha metodi getter e setter per ogni attributo;
- Implementa `Serializable`.

Definizione 5.5.4: Properties

Una property è un attributo di un Java bean. Le properties di libreria servono per:

- Binding: permette di collegare due properties;
- Listener: permette di associare un listener ad una property.

Esempio 5.5.1 (`ObservableValue<T>`)

L'interfaccia `ObservableValue<T>` è un'interfaccia generica che rappresenta una property. Ha i seguenti metodi:

- `getValue()`: ritorna il valore della property;
- `addListener(ChangeListener<? super T> listener)`: aggiunge un listener;
- `removeListener(ChangeListener<? super T> listener)`: rimuove un listener;
- `addListener(InvalidationListener listener)`: aggiunge un listener;
- `removeListener(InvalidationListener listener)`: rimuove un listener.

Esempio 5.5.2 (`ChangeListener`)

L'interfaccia `ChangeListener<T>` è un'interfaccia generica che rappresenta un listener per una property. Ha il seguente metodo:

- `changed(ObservableValue<? extends T> observable, T oldValue, T newValue)`: viene invocato quando il valore della property cambia.

Definizione 5.5.5: Properties in JavaFX

In JavaFX esistono le seguenti properties:

- `SimpleStringProperty`: rappresenta una property di tipo `String`;
- `SimpleIntegerProperty`: rappresenta una property di tipo `Integer`;
- `SimpleDoubleProperty`: rappresenta una property di tipo `Double`;
- `SimpleBooleanProperty`: rappresenta una property di tipo `Boolean`.

6

Thread

6.1 Introduzione

Definizione 6.1.1: Thread

Il Thread è un flusso sequenziale di controllo (esecuzione di istruzioni) in un programma. Un programma può avere più thread che eseguono contemporaneamente (in modo concorrente). Tutti i thread di un programma condividono lo stesso spazio di indirizzamento^a.

^aA differenza dei processi

Note:-

Un thread può essere visto come un light-weight process.

Definizione 6.1.2: Classe Thread

La classe Thread è una classe Java che permette di creare e gestire thread. La classe Thread ha un costruttore che prende un oggetto Runnable come parametro. Runnable è un'interfaccia che ha un solo metodo run() che deve essere implementato. Il metodo run() contiene il codice che deve essere eseguito dal thread.

Corollario 6.1.1 API della classe Thread

- ⇒ start() - avvia il thread;
- ⇒ run() - contiene il codice che deve essere eseguito dal thread;
- ⇒ sleep(int ms) - sospende il thread per ms millisecondi;
- ⇒ join() - attende la terminazione del thread;
- ⇒ isAlive() - ritorna true se il thread è in esecuzione;
- ⇒ getName() - ritorna il nome del thread;
- ⇒ setName(String name) - imposta il nome del thread.

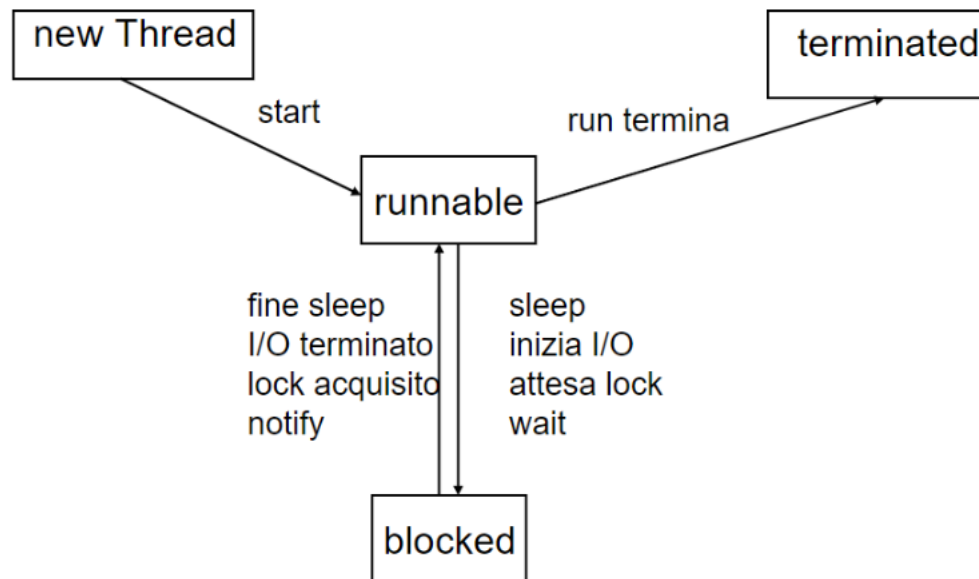


Figure 6.1: Ciclo di vita di un thread

Note:-

Lo stato Runnable indica che il thread è pronto per essere eseguito (non necessariamente in esecuzione).

Definizione 6.1.3: Interface Runnable

L'interfaccia Runnable è un'interfaccia Java che ha un solo metodo `run()` che deve essere implementato. Il metodo `run()` contiene il codice che deve essere eseguito dal thread.

Definizione 6.1.4: Join

Il metodo `join()` permette di attendere la terminazione del thread su cui è invocato. Il thread che invoca il metodo `join()` viene sospeso fino a quando il thread su cui è invocato non termina.

Note:-

Il metodo `join()` va invocato dopo il metodo `start()` e deve essere invocato con il meccanismo `try/catch`.

6.2 Thread in Java

Definizione 6.2.1: 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 6.2.2: 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...}`

6.3 Sincronizzazione lato server e lato client

Definizione 6.3.1: 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 6.3.2: 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.

Corollario 6.3.1 Cooperazione tra thread

- `wait()` - sospende il thread corrente e rilascia il lock dell'oggetto su cui è invocato;
- `notify()` - risveglia un thread sospeso su un oggetto;
- `notifyAll()` - risveglia tutti i thread sospesi su un oggetto.

Note:-

Generalmente conviene usare `notifyAll()` per evitare che un thread venga risvegliato e poi rimesso in attesa (causando un deadlock).

Definizione 6.3.3: BlockingQueue

`BlockingQueue` è un'interfaccia Java che estende l'interfaccia `Queue`. `BlockingQueue` ha i metodi `put()` e `take()` che permettono di inserire e rimuovere elementi dalla coda. `BlockingQueue` è una coda sincronizzata: se la coda è vuota, il metodo `take()` sospende il thread finché non viene inserito un elemento nella coda. Se la coda è piena, il metodo `put()` sospende il thread finché non viene rimosso un elemento dalla coda.

Note:-

Un'implementazione di `BlockingQueue` è la classe `ArrayBlockingQueue`.

6.3.1 Librerie

Definizione 6.3.4: ReentrantLock

La classe `ReentrantLock` è una classe Java che ha i metodi `lock()` e `unlock()` che permettono di acquisire e rilasciare il lock. La classe `ReentrantLock` ha anche i metodi `tryLock()` e `tryLock(long timeout, TimeUnit unit)` che permettono di acquisire il lock se è disponibile e di rilasciarlo se non è disponibile entro il timeout specificato.

Corollario 6.3.2 Lock e Condition

La classe `Lock` è un'interfaccia Java che ha i metodi `lock()` e `unlock()` che permettono di acquisire e rilasciare il lock. La classe `Condition` è un'interfaccia Java che ha i metodi `await()` e `signal()` che permettono di sospendere e risvegliare un thread.

Definizione 6.3.5: ReadWriteLock

La classe `ReadWriteLock` è un'interfaccia Java che ha i metodi `readLock()` e `writeLock()` che permettono di acquisire il lock in lettura e scrittura. La classe `ReadWriteLock` ha anche i metodi `readUnlock()` e `writeUnlock()` che permettono di rilasciare il lock in lettura e scrittura.

6.4 Thread Pool

Definizione 6.4.1: Thread e Task

Task: unità logicamente indipendente di lavoro che può essere eseguita da un thread, in parallelo con altri task.

Thread: flusso di controllo che esegue un task.

Definizione 6.4.2: Thread pool

Un thread pool è un insieme di thread che possono essere riutilizzati per eseguire task. Un thread pool ha un numero massimo di thread che possono essere creati. Se tutti i thread sono occupati, i task vengono messi in una coda e vengono eseguiti quando un thread è disponibile.

Corollario 6.4.1 Executor

Un executor è un'interfaccia Java che ha il metodo `execute(Runnable command)` che permette di eseguire un task.

Ci sono 3 tipi di executor:

- ⇒ `ThreadPoolExecutor`: implementazione di un thread pool;
- ⇒ `ScheduledThreadPoolExecutor`: implementazione di un thread pool che permette di eseguire task in un momento specificato;
- ⇒ `ForkJoinPool`: implementazione di un thread pool che permette di eseguire task in parallelo.

Note:-

La graceful degrading è una tecnica che permette di ridurre il carico di lavoro quando il sistema è sovraccarico.

Definizione 6.4.3: Callable

`Callable` è un'interfaccia Java che ha il metodo `call()` che permette di eseguire un task e ritornare un risultato di tipo generico `T`.

Definizione 6.4.4: Future

Future è un'interfaccia Java che ha il metodo `get()` che permette di ottenere il risultato di un task. Il metodo `get()` è bloccante: se il risultato non è ancora disponibile, il thread viene sospeso finché il risultato non è disponibile. Future ha anche il metodo `FutureTask` che crea un `FutureTask` che, quando parte, esegue il task passato come parametro. Future offre anche il metodo `isDone()` che ritorna true se il task è completato.

Note:-

I pool thread sono utili per evitare di creare e distruggere thread, aumentando la scalabilità.

Definizione 6.4.5: Pipe

Una pipe è un canale di comunicazione tra due thread. Una pipe ha un buffer che permette di scambiare dati tra i due thread.

Corollario 6.4.2 Sender e Receiver

Sender è un'interfaccia Java che ha il metodo `send(Object message)` che permette di inviare un messaggio. Receiver è un'interfaccia Java che ha il metodo `receive()` che permette di ricevere un messaggio.

