Figlie di Iterable, Classi Generics, SQL

Domande della lezione prima

Da List = ArrayList si può fare?

Il casting si può fare, perchè List è un’ interfaccia (ossia una classe più generica di ArrayList).

Si sta facendo un upcasting che è **sempre sicuro.**

```
// Questo si può fare / è un upcasting da ArrayList a List più generica
List<String> list = new ArrayList<String>();
```

File utilissimi alla lezione di oggi:

Cosa sono Upcasting e DownCasting?

https://www.geeksforgeeks.org/upcasting-vs-downcasting-in-java/

Differenza con esempio pratico

https://stackoverflow.com/questions/23414090/what-is-the-difference-between-up-casting-and-down-casting-with-respect-to-class

Cosa è una linkedList?

https://www.geeksforgeeks.org/what-is-linked-list/

→ qui ci sono anche queue, ArrayList etc…

Classi generics

https://www.geeksforgeeks.org/generics-in-java/

https://www.html.it/pag/18028/il-tipo-generics-in-java/

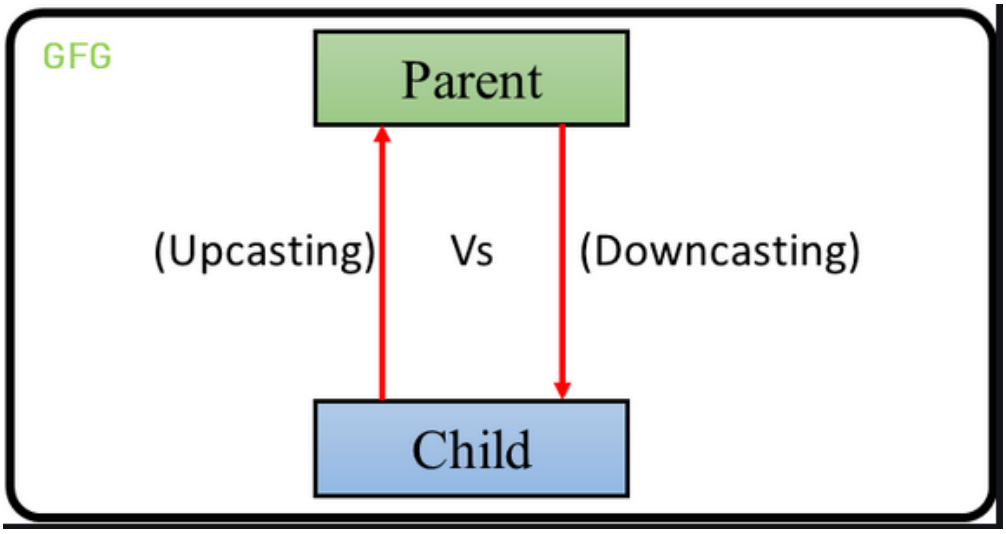
SQL e Database

https://www.w3schools.com/sql/default.asp

Down casting ed Up casting

Cosa sono?

Sono cast di tipo **dal tipo di una classe genitore** al **tipo di una classe figlio (o viceversa).**



Differenza

Upcasting → Sto “trasformando” la classe Figlio nella classe Genitore

Downcasting → Sto “trasformando” la classe Genitore nella classe figlio

Nel caso di **Upcasting** **ho sempre una trasformazione sicura** → quello che succede è che:

- Il figlio ha sempre **dei dati in più** del genitore (perchè li ha ereditati).
- Se trasformo il “figlio” in “padre”, sto solo **prendendo quelle informazioni in più e le sto “nascondendo”**, facendo finta che non esistano.

```
// Esempio:
// Questo è un upcasting: Studente sta diventando persona
Persona p = new Studente("Daniel", "Mateus", 24, "844615");

// Studente -> si sta "togliendo informazioni" per comportarsi come
// una classe di tipo Persona.


// Di conseguenza perdo l'informazione sulla matricola (perchè fa parte di
// studente e non di persona).
```

Studente

Perchè lo faccio?

Perchè così ho una **compatibilità di tipo**.

Java → vede adesso due oggetti dello stesso tipo.

🤔 **Quale versione dei metodi viene eseguita?**

Abbiamo uno Studente che si **comporta da Persona** (perchè è una persona.

Se abbiamo un metodo saluta(), verrà eseguita quale versione del metodo? Quella di persona o quella di Studente?

Verrà eseguita la versione di saluta() della classe figlio, ossia Studente.

Uno studente è anche una persona. Una volta uscito da scuola **può comportarsi come se fosse una persona.**

Il downcasting invece non è **sempre sicuro**, perchè:

- Una classe padre ha delle **informazioni in meno rispetto alla classe figlio.**
- Non conoscendo quali informazioni “dovrebbe avere in più” per diventare del tipo del figlio, **non si può produrre un oggetto della classe figlio “completa” a partire dalla classe padre.**

Una persona **non può far finta di essere uno studente.** Non può comportarsi come uno studente se **non è iscritto a scuola.**

```
// Esempio di downcasting SICURO

Persona p = new Studente("Daniel", "Mateus", 24, "844615");

// Upcasting -> posso farlo! perchè ho un override di stampa in Studente
p.stampa();

// non posso farlo!! Questo lo avevo perso
// p.getNome();

// DownCasting
// posso farlo perchè p PRIMA DELL' UPCASTING ERA UNO STUDENTE
// Quindi le informazioni di p da "studente" ci sono , ma sono NASCOSTE

//Ora quello che faccio è "mostrarle" -> esistevano già, solo non le vedevo
Studente s = (Studente) p ;
System.out.println( p.getNome() );

// Esempio di downcasting NON SICURO
Persona p = new Persona("Daniel", "Mateus", 24, "844615");
Studente s = (Studente) p ;

// In questo caso NON posso farlo, perchè:

// p PRIMA NON ERA UNO STUDENTE -> a lui non avevano mai dato una matricola
// In questo caso non ci sono "informazioni" nascoste. Le informazioni proprio
// non ci sono.

// Di conseguenza chiedere ad una "Persona" di fare lo "Studente" non è
// possibile
```

🤔 **Occhio!**

Java non dà errore quando io faccio downcasting!

Nell’ esempio prima io posso **forzare il downcast anche se è sbagliato.**

Mi accorgo di avere degli errori quando chiedo ad una classe di tipo Persona di fare cose che possono fare solo le classi di tipo Studente.

Ad esempio: p.getMatricola() → come fa una persona che non ha la matricola a darmi la matrica ??? non può, non ce l’ha

🤔 **Trucchetto!**

Per capire se posso o meno fare un downcasting senza avere problemi o meno:

Mi chiedo “la classe X, se volesse, può comportarsi come se fosse della classe Y?”

Uno studente è anche una persona ? Sì, questo ha senso (no problem)

Una persona è anche uno studente ? Non sempre (ecco, questo downcasting non è sicuro).

🤔 **Late binding**

E’ il processo Risoluzione di un metodo (da parte del compilatore) a Run-time.

Questo serve per permettere a Java di fare casting di tipo tra oggetti Genitore e Figlio.

In pratica, quando chiamo un metodo da un oggetto Genitore o Figlio, a “run time” il compilatore “sceglie quale dei due eseguire.

La famiglia Collection

E’ un insieme di classi tutte **derivate da iterable** ossia → di strutture dati **iterabili.**

Ci sono diverse Classi derivate da iterable, **che gestiscono tutte una lista iterabile,** ma differiscono per modalità di **accesso,consulta e gestione in generale.**

🤔 Funziona come la “lista della spesa”, ogni “cosa da comprare” è un elemento della mia struttura dati, ed io posso “andare avanti e indietro” per vedere cosa ho già comprato.

Posso anche “aggiungere qualcosa che avevo dimenticato di segnare, ma che devo comprare”, oppure “cancellare qualcosa perchè ho visto che costa troppo”.

Insomma, è qualcosa che posso consultare andando avanti e indietro come se avessi una lista in mano.

🤔 **Curiosità**

Differenza tra ArrayList e Vector (che semrano uguali) è che Vector è **sincronizzato** ed ArrayList no.

Se un ArrayList se lo stanno condividendo due Thread o due processi, ed uno aggiorna un elemento l'altro **non vede la modifica perchè non c'è sincronizzazione tra le due parti.**

Classi figlie di Iterable (Stack, ArrayList etc…)

Come abbiamo già visto, tutte le classi che estendono la classe iterator fanno la stessa cosa (gestire una lista di elementi), ma lo fanno in modo diverso.

Ad esempio:

Con lo **Stack** per accedere ad un metodo si usa un metodo .pop().

Con l' **ArrayList** si usa la .get().

Fanno la stessa cosa, ma ci sono **processi che avvengono "nei retroscena della memoria" che rendono la gestione differente.**

Inoltre anche la **"logica" che c'è dietro è diversa:**

Nello stack si può "prendere o togliere solo l'ultimo elemento (a livello di memoria)", nell' ArrayList posso "accedere a qualsiasi cosa in qualsiasi modo".

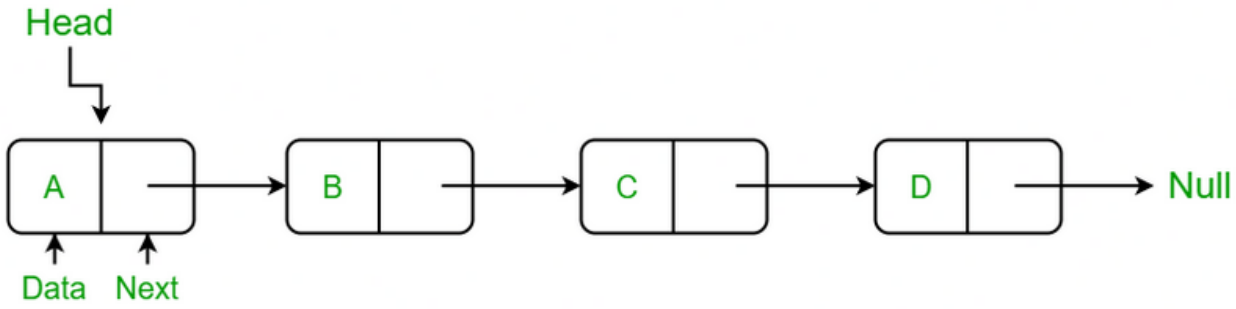
Fanno la stessa roba con una modalità diversa, e con restrizioni diverse.

Ma in memoria, quello che stanno gestendo **è sempre una lista di informazioni.**

Altro esempio (LinkedList):

La LinkedList gestiscono la lista di dati **un elemento alla volta.**

Ogni elemento è collegato al successivo, se io voglio accedere all' elemento i-esimo → devo prima scorrere tutti quelli che vengono prima di lui.



Gestisce la stessa roba degli altri (sempre una lista di elementi in memoria), ma può scorrerli **solo in questa modalità.**

Classi parametriche o generiche

Tutte le classi hanno un costruttore che prende **un insieme di dati precisi.**

Ad esempio Studente(String Nome, String Cognome, int età, String matricola) **prende precisamente 3 String ed un intero.**

Ci sono classi che però possono essere costruite con **insiemi di dati che possono essere qualsiasi cosa:**

```
// Quello che stiamo dicendo è che questa i dati interni di questa classe
// Saranno di tipo T, ma T che tipo è???

// T è QUALSIASI TIPO, cioè : quando creeremo un'istanza della classe Coppia
// ( un oggetto ) -> potrò decidere di che tipo sarà T -> e quindi la class
// sarà del tipo che deciderò io

public class Coppia<T> {
  T a;
  T b;

  // Costruttore
  public Coppia(T a, T b) {
    super();
    this.a = a;
    this.b = b;
  }

.... resto della classe
```

Esempio di oggetto:

```
Coppia<Tipo_che_scelgo_io> nome = new Coppia<Tipo_che_scelgo_io>(parametri_costruttore);

// Ad esempio:

// Tutto ciè che è "T" nella classe Coppia, verrà rimpiazzato con String
// Quindi i dati di Coppia<String> saranno Stringhe
Coppia<String> nome = new Coppia<String>("Franco", "Peppino");

// Stesso discorso, ma con gli interi
Coppia<Integer> nome = new Coppia<Integer>(10,20);
```

🤔 **Passare da una classe normale ad una generica**

Su Eclipse:

lanternina gialla del menù > menù search > Riempio “Containing text” con quello che voglio diventi generico > Click su Search > Seleziono tra i risultati i tipi che voglio che diventino T > click destro > replace with > scrivo “T” > Ok fine.

(complicatino, forse è meglio già partire con l’idea di fare una classe generica e scrivo subito T).

> 🤔 **Occhio!**
>
> Non è possibile creare Array di Oggetti Generici (o meglio, istanze concrete di una classe generica).
>
> Posso però creare ArrayList di Oggetti generici.

Codice di esempio della prima parte (fino a qui)

---

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/1e962455-b77a-4559-8326-6151e6ae342b/Teoria_prima_parte.zip

---

Basi di debugging

Abbiamo visto come aggiungere un breakpoint nel codice ed utilizzare il debugging + stepover con **Eclipse**.

Tutto questo contenuto è riassunto nella documentazione Eclipse:

https://www.eclipse.org/community/eclipse_newsletter/2017/june/article1.php

---

SQL

Software da installare prima di iniziare:

Xampp (MySQL, Apache e TomCat)

Successivamente, dopo aver startato MySQL database, accediamo a phpMyAdmin in locale (http://localhost/phpmyadmin) da browser.

Creazione nuovo DB + prima table con phpMyAdmin

Per creare un nuovo database, nel menù a sx basta selezionare “Nuovo” e compilare il form per creare il database.

Successivamente la creazione del database ci porta al setting di una nuova table (ancora compilazione di form, facile).

Ripasso della insert

Inseriamo una query nella tabella Veicolo

```
/*INSERT SQL*/
INSERT INTO Veicolo
VALUES('AB100AB', 'Utilitaria', 'Fiat Panda');
```

Ripasso della select

Le istruzioni sono 3:

1. **SELECT** : Cosa voglio prendere (le colonne della tabella)
2. **FROM** : Da dove voglio prenderlo (la tabella)
3. **WHERE** : Quali requisiti devono avere le righe che scelgo (condizioni sulle righe delle colonne che ho selezionato con select)

```
SELECT
    *
FROM
    `Veicolo`
WHERE
```

```
    Produttore = 'Suzuki';

// Equivale a dire "prendi dalla tabella Veicolo, tutte le colonne
// E delle righe, dammi solo quelle che hanno come produttore 'Suzuki'
```

Ripasso della delete

La sintassi è uguale alla SELECT, ma con DELETE come keyword iniziale

```
DELETE FROM
    `Veicolo`
WHERE
    Produttore = 'Suzuki';

// Equivale a dire "ELIMINA dalla tabella Veicolo
// le query che hanno come produttore 'Suzuki'
```

🤔 Cancellare tutto il DB: DELETE FROM ‘Veicolo’ **WHERE 1**

Ripasso della Update

Anche qui ho 3 keyword:

1. **UPDATE** → chi voglio aggiornare
2. **SET** → contenuto nuovo (quello che voglio immettere nella tabella al posto di un altro)
3. **WHERE** → chi devo aggiornare

```
UPDATE
  Veicolo
SET
  produttore = 'Opel'
WHERE
  produttore = 'Tesla'
```

Collegamento con Java

Colleghiamo il database con Java.

Scarichiamo il **Connettore Java per SQL.** Che sta qui:

https://dev.mysql.com/downloads/connector/j/

Questo è un file Jar **da inserire nella cartella del progetto.**

Per iniziare la connessione bisogna **importare java.sql** ed utilizzare l' **oggetto Connection** (che come da nome, rappresenta una connesione):

```
Connection c =  null;
Statement s = null;

try {

      // Oggetto per iniziare la connessione
      DriverManager.getConnection("jdbc:mysql//localhost:3306/Cluster_0", "root", "");

  } catch (SQLException e) {

      // TODO Auto-generated catch block
      e.printStackTrace();

  }
```

🤔 **Prima di connettere**

Bisogna importare il connector J tra le **librerie del progetto.**

Per Linux il jar si trova in : **/usr/share/java/mysql/**

Fare una richiesta al database da Java

```
//DOPO AVERE ESEGUITO LA CONNESSIONE, sempre dentro il try_Catch:
```

```
// Oggetto "statement" -> questo oggetto è un gestore di richieste SQL
      s = c.createStatement();


      // Per fare una richiesta vera e propria uso executeQuery(...query...);
      // E salvo il tuotto il un oggetto ResultSet

      r = s.executeQuery(" SELECT * FROM Veicolo WHERE 1");

      // Visualizzo tutte le righe
      while(r.next()) {

        System.out.print(r.getString(1) + " ");
        System.out.print(r.getString(2) + " ");
        System.out.print(r.getString(3));

        System.out.println("");

      }

}catch(Exception e){

  // resto del codice

}
```

Qui abbiamo stampato le colonne di una tabella di cui **sappiamo il numero (perchè l'abbiamo creata noi).**

Se volessimo stampare tutte le colonne di una tabella **di cui non conosciamo il numero di colonne**, possiamo usare un oggetto ResultSetMetaData che ci fornisce varie informazioni su una tabella di un DB (tra cui guarda un po'! Il numero di colonne). In questo caso il codice risulta:

```
import java.util.ArrayList;


// Connessione con il DB
import java.sql.*;

public class Main {

  public static void main(String[] args) {

    Connection connection = null;
    Statement DbRequest = null;
    ResultSet QueryResult = null;

    try {
```

```
      // Oggetto per iniziare la connessione -> mettiamo in ascolto il DB per l'invio di una query
      // Occhio bisogna importare il connector con Maven!
      connection =  DriverManager.getConnection("jdbc:mysql://localhost:3306/Cluster_0", "root", "");

      // Oggetto "statement" -> questo oggetto è un gestore di richieste SQL
      DbRequest = connection.createStatement();


      // Per fare una richiesta vera e propria uso executeQuery(...query...);
      // E salvo il tuotto il un oggetto ResultSet

      QueryResult = DbRequest.executeQuery(" SELECT * FROM Veicolo WHERE 1");

      // Numero di colonne della tabella
      ResultSetMetaData queryMetaData = QueryResult.getMetaData();
      int numeroColonne = queryMetaData.getColumnCount();

      // Visualizzo tutte le righe
      while(QueryResult.next()) {

        for(int i = 1; i <= numeroColonne ; i++ ) {
          System.out.print( queryMetaData.getColumnName(i)+" : " + QueryResult.getString(i) + " " );
        }

        System.out.println("");
      }



    } catch (SQLException e) {

      // TODO Auto-generated catch block
      e.printStackTrace();

    }

  }

}
```

🤔 Se volessi anche il **Nome delle colonne** (cioè voglio proprio stamparmi la tabella come la vedo su phpMyAdmin, posso usare getcolumnName()

🤔 Gli oggetti fondamentali sono:

1. **Oggetto statement per fare richieste**
2. **Oggetto ResultSet per raccogliere il risultato di una richiesta**
3. **Oggetto ResultSetMetaData per raccogliere i metadati della richiesta salvata in ResultState**

    Tutti i metodi di questi oggetti che servono per fare la richiesta e stamparla sono nell’ esempio sopra.

Esercizio

Dobbiamo fare: Garage con DB Mysql e funzione di Visualizzazione Auto,

Soluzione:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/7cf2f004-7892-421f-a080-98ac8bd9000c/Garage_db_visual.zip

🤔 **Occhio!**

Meglio tenere la connessione pulita nel costruttore, perchè tutto ciò che viene **con statement e dopo la query muore all’ uscita di scope.**
