ESAME DI FONDAMENTI DI INFORMATICA T-2 del 6/07/2021
Proff. E. Denti – R. Calegari – A. Molesini
Tempo a disposizione: 3 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)
NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)
NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)
NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse* e *il JAR eseguibile*

Si ricorda che compiti *non compilabili* o *palesemente lontani da 18/30* NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio "RESPINTO"

Le ***Rupestri Ferrovie di Dentinia*** (RFD), che gestiscono una antica rete ferroviaria a vapore fra alcune città, hanno richiesto un'applicazione per la ricerca dei possibili percorsi fra stazioni della loro rete: i percorsi devono essere o diretti (senza cambi) o con al più un cambio.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

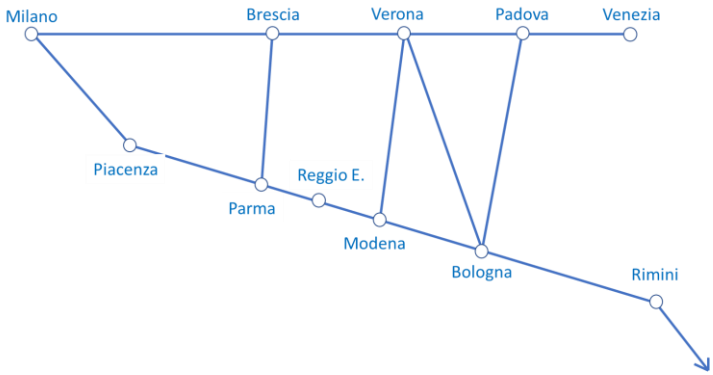
Ogni *linea ferroviaria* è descritta da una sequenza di *punti di interesse,* caratterizzati ciascuno dal nome del luogo (una stringa che può contenere spazi) e dalla *progressiva chilometrica* (ossia, la distanza dal capolinea iniziale): per antica consuetudine, quest'ultima – riportata anche sui caselli e sulle stazioni – è espressa nella forma *chilometri+metri,* dove i *metri* sono sempre espressi su tre cifre, mentre i *chilometri* sono espressi da un numero di almeno una cifra.

ESEMPIO (foto): la stazione di Lodi si trova alla progressiva chilometrica 183+802 della linea Bologna-Milano, il cui capolinea iniziale è Bologna Centrale (0+000).

I percorsi fra città appartenenti alla stessa linea (percorsi *diretti*) hanno una lunghezza facilmente desumibile dalla differenza fra le corrispondenti progressive chilometriche.

ESEMPIO: la lunghezza del percorso fra Lodi (progressiva 183+802) e Modena (progressiva 36+932) è di 146,87 km.

Alcune stazioni, in cui si intersecano due o più linee, costituiscono *punti di interscambio*: ciò consente di offrire ai viaggiatori soluzioni di viaggio *con un cambio intermedio,* unendo due *segmenti* di linee diverse aventi in comune il nodo di interscambio: in tal caso la lunghezza del percorso è pari alla somma delle lunghezze dei segmenti componenti.



ESEMPIO: il percorso fra Lodi (progressiva 183+802 della linea Bologna-Milano) e Ancona (progressiva 203+996 della linea Bologna-Lecce) è di 387,798 km e comprende due segmenti (LO-BO e BO-AN).

Se la rete comprende più linee, possono essere possibili *più percorsi* fra le medesime città, con o senza cambi.

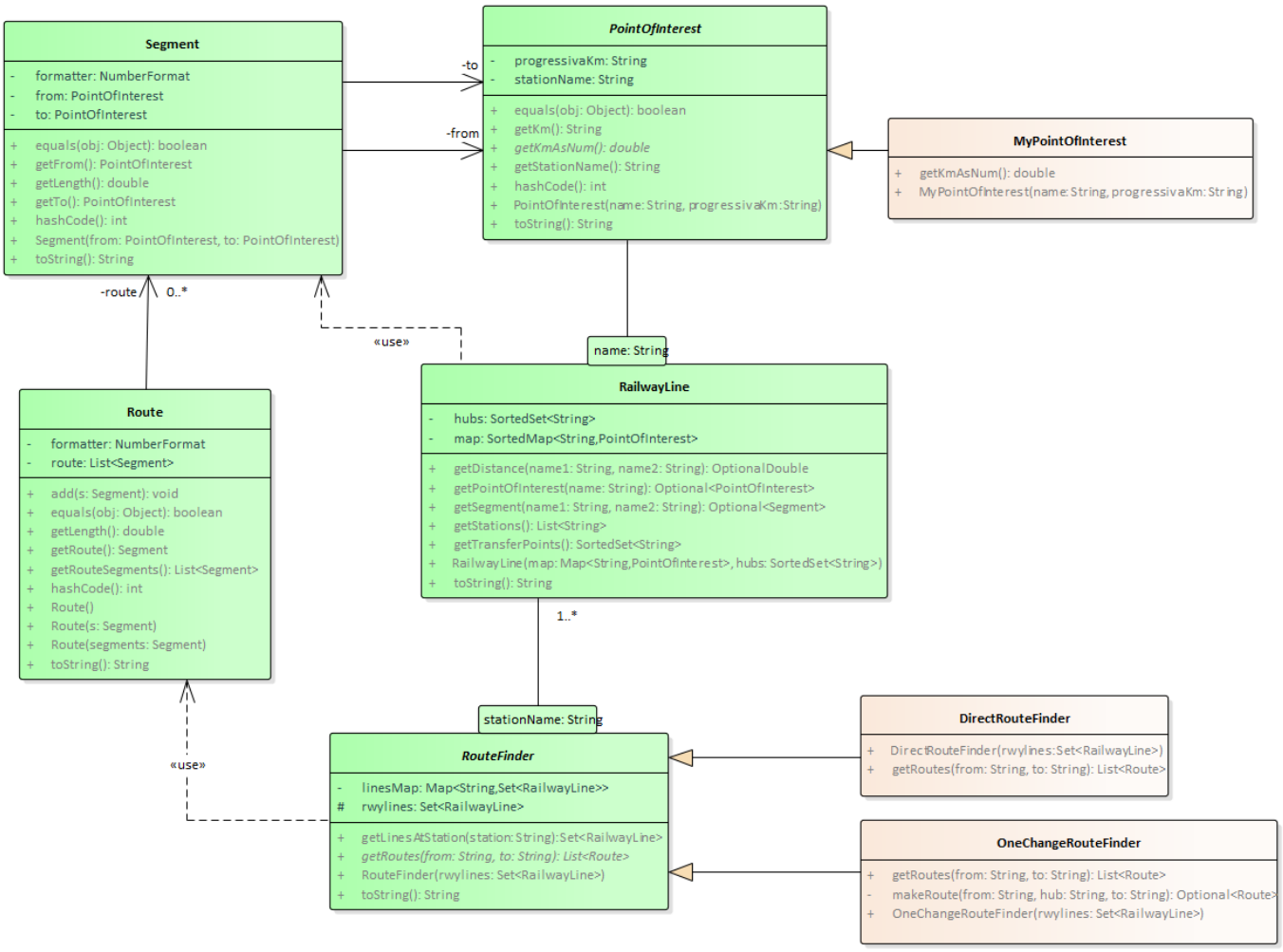
ESEMPIO: nella rete a lato, fra Reggio Emilia e Verona sono possibili due percorsi, uno via Modena, l'altro via Bologna.

La rete delle ***Rupestri Ferrovie di Dentinia***, costituita da *sette linee*, è illustrata sopra: ogni linea è descritta in un singolo file di testo, il cui formato è riportato più oltre.

TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO: 1h45 – 2h20

Parte 1 (punti: 23)

Modello dei dati (package rfd.model) [TEMPO STIMATO: 1h10-1h30] *(punti: 18)*



SEMANTICA:

a) la classe astratta ***PointOfInterest*** (fornita) rappresenta un punto di interesse, caratterizzato da nome e progressiva chilometrica, *senza fare ipotesi sul formato di quest'ultima*: il metodo *getKmAsNum*, che riporta sotto forma di numero il valore della progressiva chilometrica, rimane perciò astratto

b) la classe ***MyPointOfInterest*** (**da realizzare**) concretizza la precedente assumendo per la progressiva chilometrica **il formato *chilometri+metri***, dove i chilometri sono *un intero di almeno una cifra*, mentre i metri sono *sempre espressi su esattamente tre cifre*: se questo formato è violato, il costruttore deve lanciare ***IllegalArgumentException*** con apposito messaggio d'errore. Ovviamente, il metodo *getKmAsNum* segue anch'esso questa convenzione. *(punti: 4)* **[TEMPO STIMATO: 10-15 minuti]**

c) la classe ***Segment*** (fornita) rappresenta un segmento di linea compreso fra due ***PointOfInterest*** distinti (anche non adiacenti, ossia anche con ulteriori ***PointOfInterest*** intermedi): i metodi consentono di recuperare gli elementi caratterizzanti, inclusa la lunghezza del tratto come numero reale, nonché di produrre un'opportuna stringa descrittiva. Sono presenti anche *equals* e *hashcode*.

d) la classe ***RailwayLine*** (fornita) rappresenta una linea ferroviaria intesa come insieme di ***PointOfInterest***, che il costruttore si aspetta di ricevere sotto forma di *mappa* indicizzata per *nome del punto di interesse*. Il secondo argomento del costruttore è l'insieme ordinato dei *nomi* delle stazioni della linea che possono fungere da punti di interscambio. La classe offre svariati metodi per:

- recuperare la lista dei nomi delle stazioni (metodo *getStations*) o dei punti di interscambio (metodo *getTransferPoints*)
- recuperare un ***PointOfInterest***, se esiste, dato il suo nome (metodo *getPointOfInterest*)

- calcolare la distanza fra due ***PointOfInterest*** distinti, se esistono sulla linea (metodo *getDistance*)
- recuperare il ***Segment***, se esiste, corrispondente alla tratta compresa fra i due ***PointOfInterest*** distinti dati (metodo *getSegment*)
- emettere una stringa descrittiva (metodo *toString*)

e) la classe ***Route*** (fornita) rappresenta un *percorso* per un viaggiatore, inteso come *sequenza di* ***Segment*** su linee diverse. I tre costruttori consentono di costruire la ***Route*** in tre situazioni tipiche (inizialmente vuota, inizialmente con un solo segmento, o a partire da un numero variabile di segmenti): successivamente è comunque possibile aggiungere via via altri segmenti *in coda* alla sequenza, tramite il metodo *add*. Opportuni accessor consentono di recuperare i vari elementi: anche in questo caso sono presenti *equals*, *hashcode* e *toString*.

f) La classe astratta ***RouteFinder*** (fornita) rappresenta l'entità in grado di cercare percorsi fra due città date: il costruttore riceve a tal fine l'insieme di ***RailwayLine*** che rappresenta la rete societaria. Il metodo (astratto) *getRoutes* cerca l'insieme dei percorsi fra due stazioni date, che devono essere non-null: altrimenti, per ipotesi deve lanciare ***IllegalArgumentException*** con apposito messaggio. A differenza del precedente, il metodo (concreto) *getLinesAtStation* restituisce l'insieme – eventualmente vuoto - delle ***RailwayLine*** che servono una data stazione, senza mai lanciare eccezioni.

g) la classe ***DirectRouteFinder*** (**da realizzare**) concretizza ***RouteFinder*** implementando il metodo *getRoutes* in modo che cerchi i percorsi diretti fra le due città date. ***(punti: 5)*** **[TEMPO STIMATO: 15-20 minuti]**

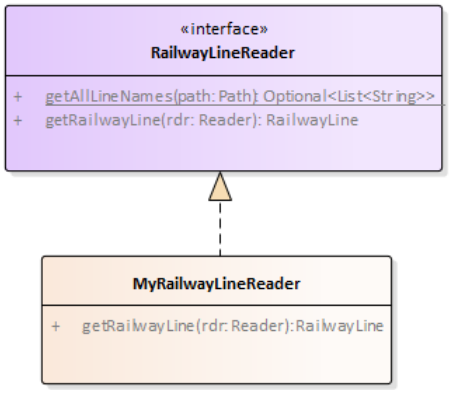
h) la classe ***OneChangeRouteFinder*** (**da realizzare**) concretizza ***RouteFinder*** implementando *getRoutes* in modo che cerchi i percorsi indiretti con un solo cambio (T) fra le due città date A e B: ***(punti: 9)*** **[TEMPO STIMATO: 45-55 minuti]**

- per ogni linea che serve la stazione di partenza A (suggerimento: usare l'apposito metodo) *che non serve anche la stazione di arrivo B* (altrimenti, sarebbe un percorso diretto!)
- recupera l'insieme dei suoi punti di interscambio (suggerimento: usare l'apposito metodo) e, per ciascuno di essi, cerca se esiste un percorso *diretto* fra tale punto (T) e la destinazione finale (B) SUGGERIMENTO: utilizzare un ***DirectRouteFinder*** per cercare tale (unico) percorso diretto
- se tale percorso diretto T-B esiste, recupera l'altro percorso *diretto*, che esiste certamente, fra la stazione di partenza (A) e il punto di interscambio considerato (T)
- confeziona quindi una nuova ***Route*** costituita dai segmenti A-T e T-B, che costituiscono per definizione i primi e unici segmenti delle due sotto-***Route*** appena trovate
- aggiunge la nuova ***Route*** così sintetizzata all'insieme delle ***Route*** da restituire.

Persistenza (rfd.persistence)

[TEMPO STIMATO: 20-30 minuti] ***(punti 5)***

Sono presenti tanti file di testo con estensione "**.txt**" quante le linee ferroviarie, tutti formattati secondo lo stesso schema. Ogni riga contiene nell'ordine *la progressiva chilometrica*, nella forma *chilometri+metri,* poi *una o più tabulazioni,* indi il *nome della stazione* (che può contenere spazi o qualunque altro carattere). Le stazioni che fungono da interscambio con altre linee sono identificabili dal simbolo "+" in coda al nome della stazione. Come anticipato nel "Dominio del Problema", la progressiva chilometrica prevede per i *chilometri* un valore intero espresso *fra una e tre cifre*, per i *metri* un valore intero espresso *sempre* su *esattamente tre* cifre. Ad esempio:




```
216+176          Milano Centrale+
212+397          Milano Lambrate+
208+751          Milano Rogoredo+
...
36+932           Modena+
25+008           Castelfranco Emilia
```

17+130	Samoggia
12+735	Anzola dell'Emilia
0+000	Bologna Centrale+

SEMANTICA:

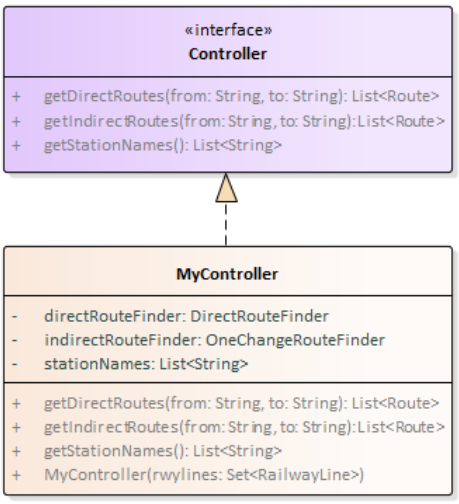
a) L'interfaccia ***RailwayLineReader*** (fornita) dichiara il metodo *getRailwayLine,* che legge da un Reader (ricevuto come argomento) i dati di una singola linea ferroviaria, restituendo la corrispondente ***RailwayLine***. L'interfaccia contiene altresì il metodo statico *getAllLineNames*, che restituisce la lista dei nomi di file di tipo ".txt" che descrivono le linee ferroviarie contenuti nella cartella passata come argomento. Tale metodo è invocato automaticamente dal main dell'applicazione (vedere Parte 2).

b) La classe ***MyRailwayLineReader*** (**da realizzare**) implementa ***RailwayLineReader***: non prevede costruttori, si limita a implementare il metodo *getRailwayLine* come sopra specificato. In caso di problemi di I/O deve essere propagata l'opportuna ***IOException***, mentre in caso di ***Reader*** nullo o altri problemi di formato dei file deve essere lanciata una opportuna ***IllegalArgumentException***, il cui messaggio dettagli l'accaduto.
In particolare, il reader deve verificare: 1) che ogni riga sia composta esattamente di 2 elementi;
2) che il nome della stazione non inizi con cifra numerica; 3) che, nel caso di stazione di interscambio, il "+" finale segua il nome della stazione senza altri caratteri intermedi e sia l'ultimo carattere della riga.

Parte 2) *[TEMPO STIMATO: 15-20 minuti]* (punti: 7)

Controller (rfd.controller) *(punti 0)*

Il Controller (fornito) è organizzato secondo il diagramma UML in figura.



SEMANTICA:

a) L'interfaccia ***Controller*** (fornita) dichiara i metodi *getStationNames, getDirectRoutes* e *getIndirectRoutes*.

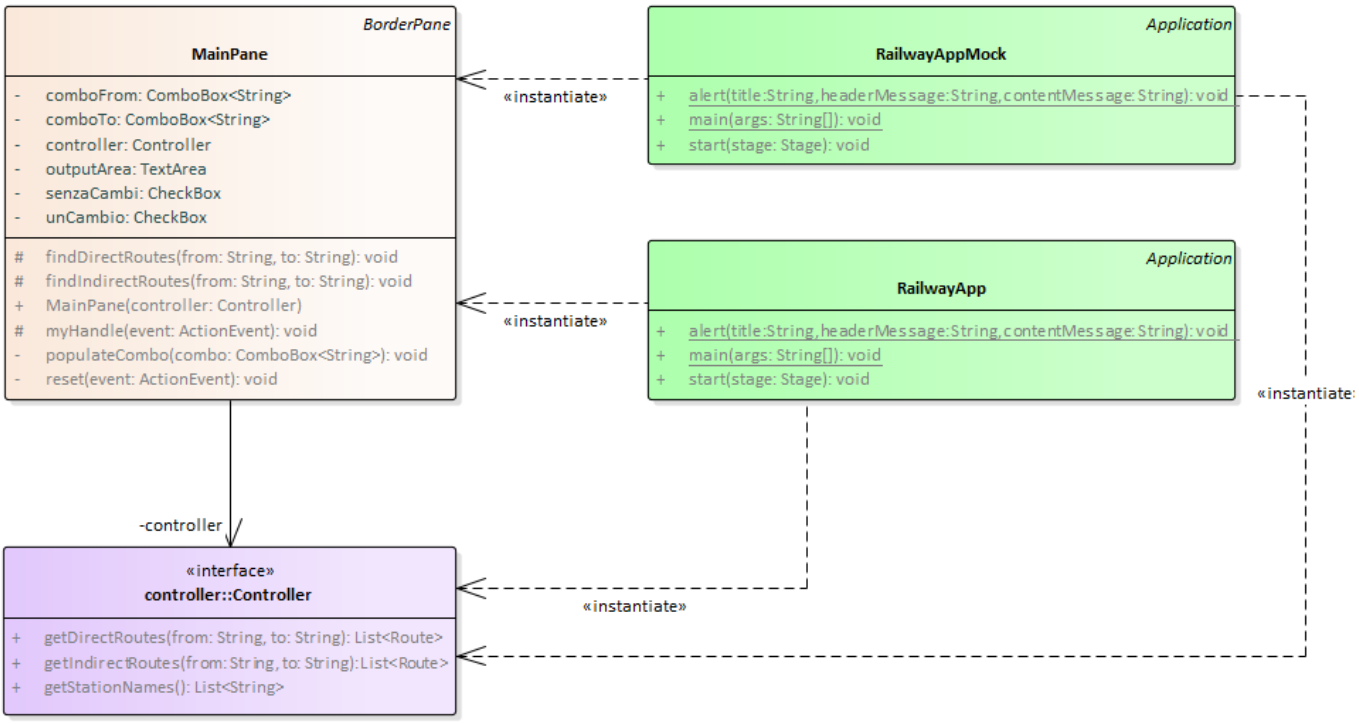
b) La classe ***MyController*** (fornita) implementa tale interfaccia, fornendo:

- il costruttore che, dall'elenco delle linee societarie, estrae l'elenco dei nomi delle stazioni e crea internamente i finder necessari;
- implementa i tre metodi delegando, nel caso delle ricerche, il lavoro ai rispettivi *finder*.

NB: la lista dei nomi di stazione restituita da *getStationNames* è già ordinata alfabeticamente.

Interfaccia utente (rfd.ui) **[TEMPO STIMATO: 15-20 minuti]** *(punti 7)*

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



La classe ***RailwayApp*** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il ***MainPane***. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe ***RailwayAppMock***.

Entrambe le classi contengono anche il **metodo statico ausiliario *alert***, utile per mostrare avvisi all'utente.

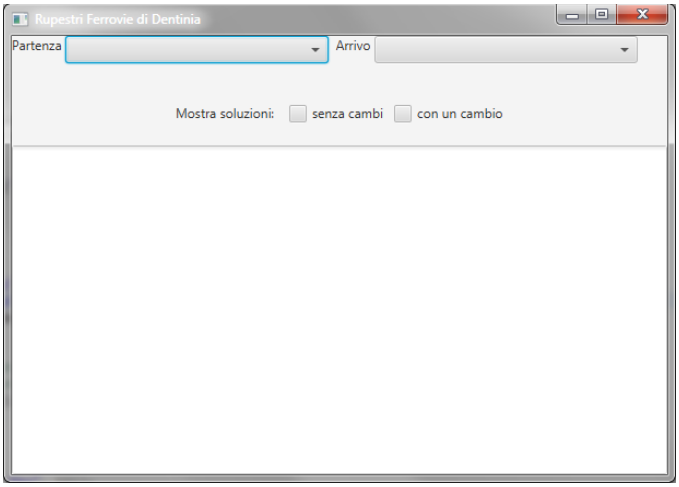
Il MainPane è fornito *parzialmente realizzato*: è presente la parte strutturale, mentre manca la parte di popolamento combo e gestione degli eventi.

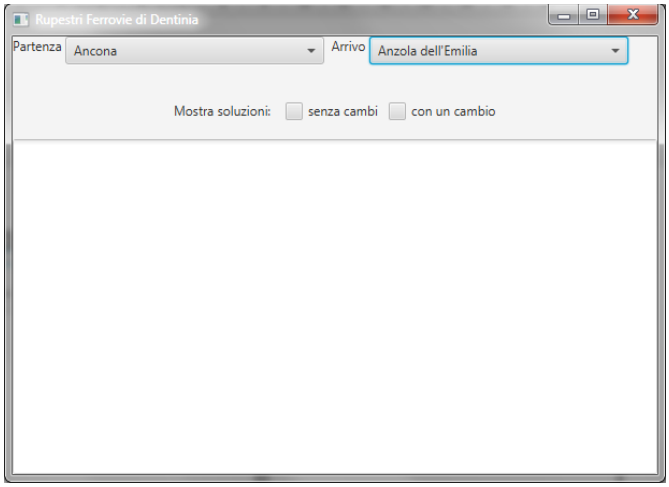
La classe ***MainPane*** (**da completare**) estende ***BorderPane*** e prevede:

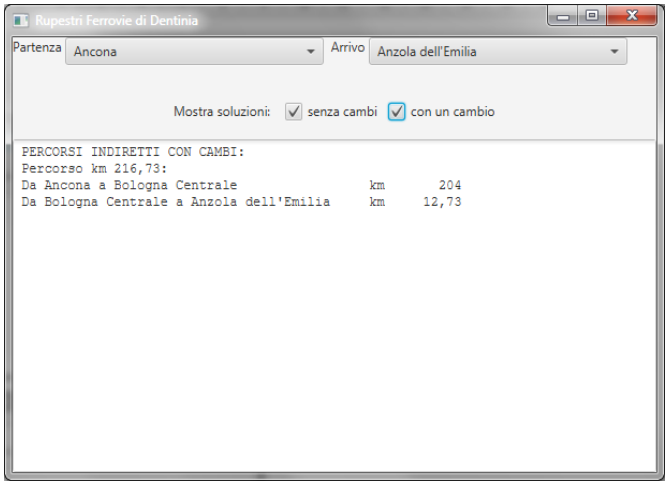
1) in alto, due ***ComboBox*** da popolare con *l'elenco alfabetico ordinato di tutte le stazioni di tutte le linee*
2) sotto, due ***Checkbox*** per scegliere la ricerca di percorsi diretti e/o con un cambio
3) in basso, una ***TextArea*** che mostra i percorsi (***Route***) trovati, utilizzando un font non proporzionale tipo Courier New (o analogo) 12 punti

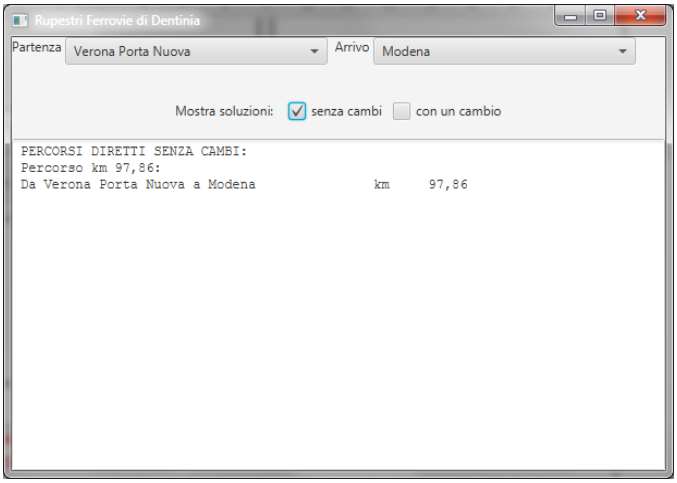
La **parte da completare** comprende l'uso e/o l'implementazione dei seguenti metodi:

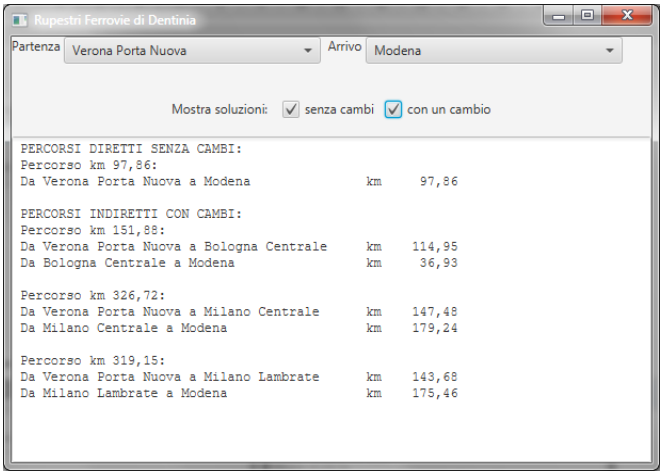
1) *populateCombo,* che popola la combo con *l'elenco alfabetico ordinato di tutte le stazioni*
2) *myHandle,* da chiamare in risposta all'evento di pressione di entrambe le ***Checkbox***: si appoggia operativamente ai due metodi ausiliari *findDirectRoutes* e *findIndirectRoutes*.
    - NB: se ci sono sia percorsi diretti che indiretti, devono essere mostrati prima quelli *diretti*
3) *findDirectRoutes* e *findIndirectRoutes,* che gestiscono concretamente il caso di percorsi rispettivamente senza cambi / con un cambio, emettendo frasi costruite secondo le seguenti regole (v. figure):
    - prima di tutti i percorsi diretti (risp. indiretti), apposita frase maiuscola di presentazione seguita da un unico *newline*, *purché tali percorsi esistano*: altrimenti non deve essere visualizzato nulla
    - due *newline* dopo ogni percorso, così che fra i vari percorsi ci sia una riga vuota

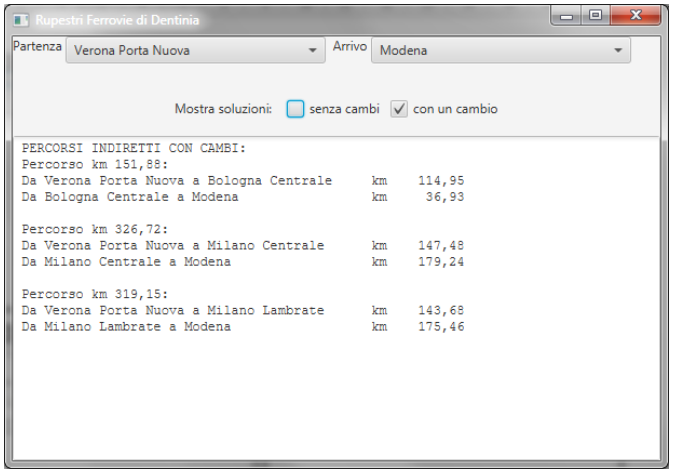












Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere "subdolamente ostile"..
- se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili** e **ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato** IL PROGETTO, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- Hai fatto un **unico file ZIP (NON .7z, rar o altri formati)** contenente **l'intero progetto?** In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- Hai consegnato **due** file distinti, ossia lo ZIP col progetto e il JAR eseguibile?
- **Su EOL, hai premuto il tasto "CONFERMA"** per inviare il tuo elaborato?