LABORATORIO DI SISTEMI OPERATIVI
A.A. 2022-23

Cat&FoxBank

alessandro.desolda@studio.unibo.it

Sofia Campana

Matricola: 0001032420

Alessandro Desolda

Matricola: 0001020790

ALMA MATER STUDIORUM UNIVERSITA' DI BOLOGNA

- INFORMATICA PER IL MANADGMENT -

1. Architettura Generale

Si pone come obiettivo quello di progettare e realizzare un servizio Client-Server per il trasferimento di denaro tra conti bancari, consentendo a più client di connettersi ad un server, attraverso threads dedicati. Garantiamo, inoltre, la muta esclusione e la gestione della concorrenza per un servizio fluido ed efficiente

Gli utenti possono effettuare le seguenti operazioni: aprire nuovi conti, effettuare transazioni di denaro tra conti, avviare sessioni interattive di trasferimento e richiedere la lista dei conti aperti con le relative ultime transazioni.

Ogni conto ha un nome univoco e, durante una transazione, un conto può essere coinvolto una sola volta. La comunicazione avviene tramite socket, e il linguaggio di programmazione utilizzato è Java.

1.1.Visione di Alto Livello

Componenti Principali:

- *Client*: Rappresenta l'interfaccia utente che può connettersi al server per eseguire operazioni bancarie.

- *Sender*: Thread specifico che rileva le richieste dell'utente e le inoltra al server.

- *Receiver*: Thread in ascolto per i messaggi provenienti dal server.

- *Server*: Accetta connessioni multiple dai client e gestisce tutte le richieste provenienti da essi.

- *SocketListener*: Si occupa di accettare nuove connessioni da parte dei client e crea un gestore dedicato a ciascuna connessione.

- *ClientHandler*: Gestisce le richieste inviate da un singolo client interagendo con l'AccountManager per eseguire le operazioni richieste.

- *AccountManager*: è responsabile della gestione degli account e delle operazioni bancarie.

- *Account*: Rappresenta un account bancario con un saldo e un registro delle transazioni.

Interazioni Principali:

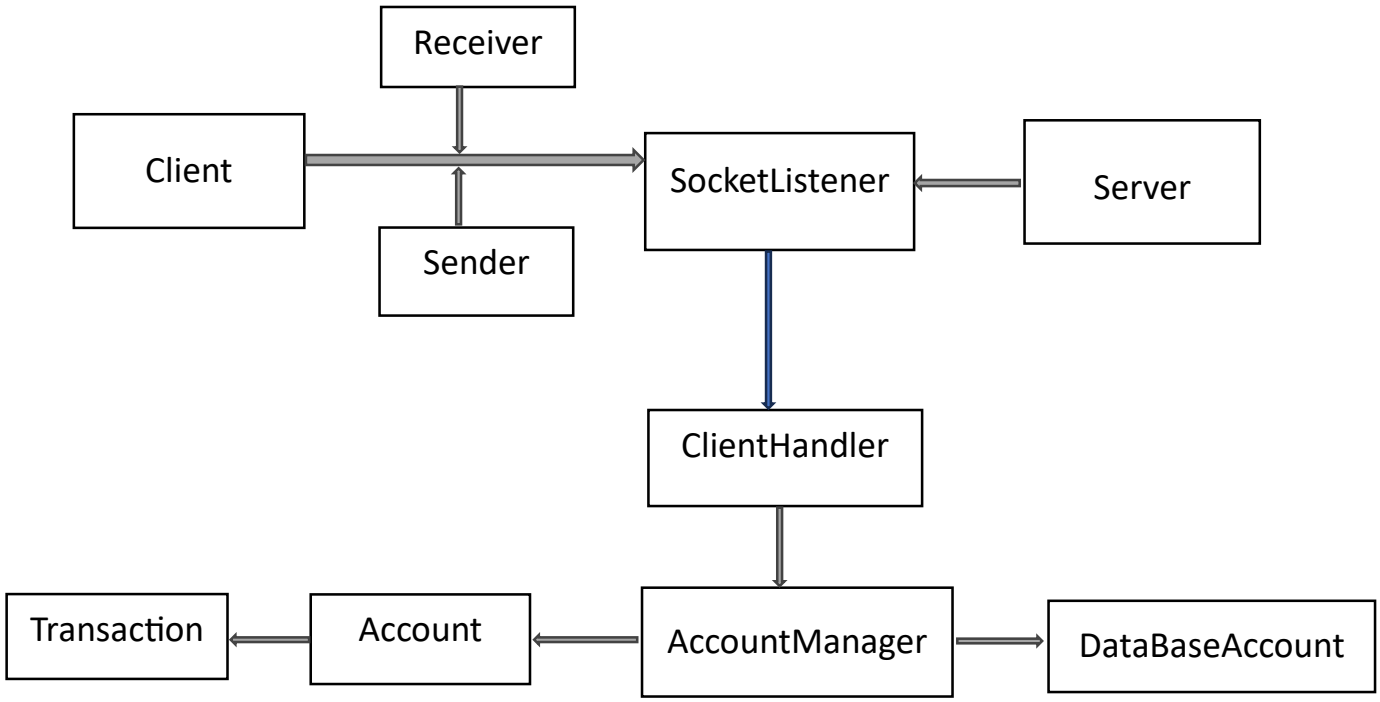
-Connessione Cliente-Server: Il client si connette al server specificando l'host e la porta, il SocketListener del server, che è in attesa, accetta la connessione. Dopo l'avvenuta connessione viene assegnato un thread ClientHandler dedicato per gestire le interazioni con quel client.

-Richiesta Operazioni del Cliente: Il ClientHandler riceve le richieste del cliente, tramite il Sender, (come l'apertura di un account, trasferimento di denaro, visualizzazione dell'elenco degli account, ...) a questo punto invoca l'AccountManager, per l'elaborazione. Quet'ultimo gestisce logicamente le richieste, interagisce con gli Account e ritorna i risultati al ClientHandler.

-Gestione degli Account*:* L'AccountManager gestisce la creazione, l'aggiornamento e l'accesso concorrente agli account, per evitare conflitti durante le operazioni di trasferimento. L'uso della sincronizzazione garantisce che una sola operazione alla volta possa essere eseguita su un singolo account. Inoltre, legge e scrive i dati degli account su un file di database per garantire la persistenza delle informazioni anche in caso di chiusura del server.

- Interazione Cliente-Server in Sessione: Durante una sessione, il ClientHandler riceve le richieste del cliente e interagisce con l'AccountManager per soddisfarle. La comunicazione avviene tramite il socket aperto tra il client e il server.

-Transazioni concorrenti: Prima di eseguire una transazione, l'AccountManager verifica la disponibilità degli account coinvolti. In caso di account già partecipi in una transazione, la richiesta viene mantenuta in attesa fino a quando gli account si liberano. Durante le transazioni, il ClientHandler e l'AccountManager collaborano per eseguire le operazioni di trasferimento di denaro.



1.2.Descrizione delle componenti principali

Client:

Client
-host: String -port: int
+main()

Il componente Client costituisce l'interfaccia attraverso la quale gli utenti interagiscono con il sistema bancario. Comunica ed inoltra le richieste al server attraverso una connessione socket ed è responsabile della trasmissione delle operazioni richieste dagli utenti e della traduzione dei comandi utente in richieste comprensibili al il server.
Invia comandi e messaggi al server tramite il thread Sender e riceve e legge le risposte tramite il thread Receiver.
I comandi includono l'apertura di nuovi account <open>, l'esecuzione di trasferimenti di denaro <transfer> e <transfer_i>, la visualizzazione della lista dei conti <list> e il comando per uscire dall'applicazione <quit>.

Funzionalità Client:

-È responsabile di stabilire una connessione con il server utilizzando un socket, specificando l'indirizzo host e la porta. Una volta stabilita la connessione, il client avvia due thread separati (Sender e Receiver) per gestire l'invio di comandi e la ricezione delle risposte dal server.

-Comunica con l'utente permettendo loro di gestire i propri account in modo semplice ed efficiente.

-Gestisce le sessioni interattive fornendo un menù di utilizzo che descrive le possibili operazioni. L'utente può eseguire più azioni di trasferimento tra i due Account coinvolti nella sessione, senza dover acquisire il diritto di utilizzo ma tenendoli occupati per un tempo prolungato.

-Gestisce la chiusura della connessione al server quando l'utente decide di terminare l'applicazione. Ciò assicura una corretta terminazione della sessione tra il cliente e il server.

Server:

Server
-port: int
+main()

Il server è il cuore del sistema, è responsabile di accettare e gestire le connessioni in arrivo dai client. Coordina le operazioni bancarie e le attività dei vari client mantenendo la coerenza e la persistenza delle informazioni nel sistema. Utilizza SocketListener per accettare nuove connessioni e crea per

ogni nuova connessione accettata un nuovo thread, ClientHandler. Quest'ultimo smista le richieste del client e demanda l'esecuzione delle operazioni ad AccountManager che le implementa in modo da garantire un buon livello di concorrenza tramite i suoi metodi synchronized. Questo approccio multithreading consente al server di gestire più client contemporaneamente.

Funzionalità del Server

-Utilizza un oggetto ServerSocket per ascoltare le connessioni sulla porta specificata. Le connessioni sono gestite in un thread separato, SocketListener, che consente la gestione concorrente di più client.

- Ogni connessione è gestita da un ClientHandler, che è responsabile di smistare le richieste specifiche del client associato. Comunica e le inoltra all'AccountManager per l'elaborazione e restituisce le risposte al client.

- Comunica con il client attraverso gli stream di input e output del socket. Il programma rimane in attesa di comandi utente e termina quando viene inserito il comando <quit>; il server interrompe il thread di gestione delle connessioni attendendone la completa terminazione prima di chiudere il programma principale.

- Si occupa della memorizzazione persistente degli account su un file di database. Questo assicura che le informazioni siano conservate anche in caso di riavvio del server.

-Coordina gestisce le transazioni verificando la disponibilità degli account coinvolti, mantenendo le richieste in attesa, se necessario, e collaborando con il ClientHandler per eseguire le operazioni di trasferimento.

- Gestisce possibili eccezioni e situazioni impreviste fornendo un feedback al client o registrando eventuali problemi.

1.3.Sotto-componenti del Client e del Sever

-Le classi **Sender** e **Reciver** sono parti essenziali dell'implementazione del sistema di comunicazione client-server. Collaborano per consentire una comunicazione bidirezionale tra il client e il server, garantendo un flusso di richieste e risposte asincrono e coordinato.

Sender:

Sender
-s: Socket
+Sender(s: Socket) +run(): void

È responsabile della gestione dell'invio di richieste da parte del client al server attraverso una connessione socket. Mantiene un loop che legge continuamente l'input del client da console e gestisce le richieste di <quit>, interrompendo il thread Sender e chiudendo lo scanner per liberare le risorse.

Receiver:

Receiver
-s: Socket -sender: Thread
+Receiver(s: Socket, sender: Thread) +run(): void

Si occupa della ricezione di messaggi inviati dal server al client attraverso una connessione socket. Mantiene un loop che continua a leggere le risposte del server e le visualizza sul terminale del client. Si occupa anche della gestione delle eccezioni, arrestando il thread di invio associato (il thread Sender) quando si chiude.

-La classe **SocketListener** è responsabile di ascoltare e gestire simultaneamente molteplici connessioni in arrivo dai client attraverso un oggetto ServerSocket. Quando accetta una connessione, crea un nuovo thread (ClientHandler) per gestire la comunicazione con il client associato. È inoltre dotato di un timeout che determina per quanto tempo rimarrà in attesa di una nuova connessione prima di controllare se il thread principale del server è stato interrotto.

SocketListener
-server: ServerSocket -worker: ArrayList<Thread>
+ SocketListener (server: ServerSocket) +run(): void

-La classe **ClientHandler** è associata a ciascun singolo client connesso al server ed è responsabile della comunicazione tra essi tramite un socket. Coopera con l'AccountManager garantendo che le richieste siano elaborate in modo corretto e sincronizzato.

ClientHandler
-s: Socket -a: AccountManager
+ClientHandler (s: Socket, a: AccountManager) +run(): void +request: String

-La classe **AccountManager** è responsabile della gestione degli account, inclusa la creazione, la lettura e la scrittura su file di database; nonché l'esecuzione di operazioni come i trasferimenti di denaro. Presenta due mappe, serchAccount e busyAccount, per memorizzare gli account disponibili e quelli attualmente coinvolti in transazioni. Garantisce la persistenza dei dati e l'accesso concorrenziale agli account per prevenire conflitti.

AccountManager
-serchAccount: HashMap<String, Account> -busyAccount: HashMap<String, Account>
+AccountManager() +addAccount(key: String, account: Account): void +extract(key: String): Account +extractAll(): String +readDataBase(r_a: AccountManager): void +writeDataBase(): void +transfer(M: double, a_S: String, a_R: String, to: PrintWriter): String +sectionMove(S: Account, R: Account, M: double): String +interactive(aSender_i: String, aReceiver_i: String, to_i: PrintWriter, from_i: Scanner): void +checkBusy(cSender: Account, cReceiver: Account, to_i: PrintWriter): void +busyEnd(eSender: Account, eReceiver: Account): void

Metodi di AccountManager:

-AccountManager() è il costruttore che inizializza le due mappe (serchAccount e busyAccount), rispettivamente per la gestione degli account e per tracciare gli account coinvolti in transazioni.

-addAccount() aggiunge un nuovo account alla mappa serchAccount. Verifica che non esista già un account con la stessa chiave e in caso contrario lancia un'eccezione (IllegalArgumentException)

-extract() estrae un account dalla mappa serchAccount sulla base della chiave specificata. Se l'account non esiste, lancia un'eccezione.

-extractAll() estrae tutti gli account dalla mappa serchAccount e li restituisce sotto forma di stringa formattata.

-readDataBase() legge gli account da un file di database (DataBaseAccount.csv).

-writeDataBase() scrive gli account presenti in serchAccount su un file di database (DataBaseAccount.csv).

-transfer() gestisce una richiesta di trasferimento di denaro tra due account. Verifica la disponibilità degli account, chiama sectionMove per eseguire la transazione e restituisce un messaggio risultante.

-sectionMove() esegue la sezione critica per il trasferimento di denaro. Verifica se l'account mittente ha fondi sufficienti, quindi decrementa il suo saldo e incrementa il saldo dell'account destinatario.

-interactive() gestisce una sessione interattiva tra due account permettendo il trasferimento di denaro.

-checkBusy() controlla se gli account sono occupati in altre transizioni e, se necessario, attende fino a quando non tornano disponibili.

-busyEnd() libera gli account dall'occupazione, notificando eventuali threads in attesa.

-La classe **Account** è progettata per rappresentare un account bancario all'interno del sistema. Il suo scopo principale è gestire e manipolare gli attributi come nome dell'account (name), saldo (money), informazioni sulla transazione (lastT).

Account
-name: String -money: double -lastT: transaction
+Account(name: String, money: double) +Account(name: String, money: double, t: String, m: double) +Account(name: String, money: double, t: String) +getName(): String +getMoney(): double +getTransaction(): Transaction +getTransfer(): String +setTransation(m: double, n: String): void +InFlow(cash: double): void +OutFlow(cash: double): void +toString(): String

-La classe **Transaction** fornisce un modo ordinato per registrare e conservare le informazioni relative alle transazioni associate agli account bancari. La rappresentazione testuale delle transazioni è utile per il monitoraggio delle attività finanziarie degli account nel sistema di gestione di trasferimenti di denaro. Rappresenta ogni singola transazione, mantenendo informazioni sulla data e l'ora (d), l'importo trasferito (cashMoved) e la chiave dell'account coinvolto nella transazione (cashAccountKey).

Transaction
-d: LocalDateTime -cashMoved: double -cashAccountKey: String -f: DateTimeFormatter
+Transaction() +Transaction(cashMoved: double, cashAccountKey: String, d: String) +Transaction(cashMoved: double, cashAccountKey: String, d: LocalDateTime) +moveTransaction(cashMoved: double, cashAccountKey: String): void +getKey(): String +toString(): String +getValueTransf(): String

1.4.Suddivisione del lavoro

Inizialmente abbiamo analizzato insieme, punto per punto, le varie specifiche e richieste del progetto, per poi suddividerci i compiti nel seguente modo:

-Insieme abbiamo implementato la struttura generale del progetto, cioè le classi Client e Server con le relative classi annesse.

-Campana si è occupata principalmente della programmazione delle classi ClientHandler, Account e della documentazione.

-Desolda ha implementato le classi AccountManager, con la gestione delle Hashmap e della concorrenza, Transaction e della gestione di DataBaseAccount.

Insieme abbiamo riguardato tutte le classi e la documentazione per modificare e migliorare il progetto, in caso di problemi riscontarti o perplessità ci siamo scambiati i codici per un confronto e una risoluzione efficiente.

2. Descrizione e discussione del processo di implementazione

2.1.Descrizione dei problemi

Durante la progettazione del nostro programma abbiamo riscontrato alcune perplessità e alcuni ostacoli, risolti senza troppe difficoltà durante gli incontri.

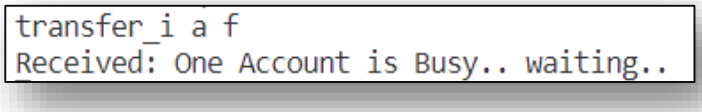
- *Gestione Concorrenza:*

Nell'ambito della concorrenza il problema affrontato è stato inerente alla scelta del metodo da utilizzare. Inizialmente avevamo scelto di gestirla tramite l'uso di Lock ma dopo alcuni test ci siamo resi conto che il loro utilizzo comportava alcuni problemi, una maggiore complessità e un rischio superiore di mal funzionamento del programma in caso di un'errata gestione. Abbiamo cosi preferito optare per l'utilizzo del costrutto java, synchronized, esso fornisce un livello più alto di astrazione, gestendo automaticamente l'acquisizione e il rilascio del lock in modo implicito e viene dichiarato direttamente nei metodi. Questo ci ha portato ha ragionare in modo differente inquanto i lock su risorse condivise vengono esercitati all'ingresso del metodo synchronized, creando quindi dei metodi appositi per la realizzazione di operazioni in modo da garantire la mutua esclusione.
La struttura del nostro codice è gestita in modo tale da assegnare tutti i metodi per la gestione della concorrenza alla classe AccountManager.
Inoltre per garantire la possibilità a due Client di eseguire transazioni sullo stesso account in modo corretto, abbiamo implementato un metodo checkBusy che controlla che un account non sia già occupato in una transazione, scegliendo di utilizzare un HashMap busyAccount per memorizzare gli account in uso. Questo metodo gestisce in contemporanea sia l'account Sender sia l'account Receiver cosi da evitare un possibile DeadLock.
Sarebbe stato possibile creare un metodo generale in cui si inserisce in busyAccount un account per volta dopo il controllo del suo stato di disponibilità, ma questo avrebbe portato a un probabile DeadLock. Difatti se Client_1 avesse eseguito "transfer_i a b" e un Client_2 "transfer_i b a" il sistema avrebbe eseguito dei context switch subito dopo aver bloccato il primo Account, Client_1. Avrebbe successivamente atteso all'infinito il rilascio della risorsa da parte del Client_2 che, similmente, avrebbe aspetto la risorsa da Client_1; questi rilasci avverrebbero solo dopo le relative acquisizioni. Ecco verificato il DeadLock (facciamo notare che il metodo interactive di transfer_i non è synchronized). Una soluzione alternativa poteva essere la gestione dell'acquisizione prioritaria, esempio ordine alfabetico, tra i due account.

- *Gestione transazioni (transfer e transfer_i) codice comune:*

Un'altra situazione che ci ha fatto riflettere è inerente all'implementazione dei metodi transfer e interactive, sempre all'interno di AccountManager, che si riferiscono ai comandi transfer e transfer_i. Queste due operazioni, infatti, apparentemente distinte eseguono le stesse funzioni di decremento e di incremento del livello di denaro all'interno di un account. Per evitare inutili ripetizioni di codice e per gestire la concorrenza nel modo più intelligente, abbiamo creato un metodo apposito per eseguire lo spostamento di denaro, sectionMove, che esegue le operazioni base di trasferimento. Il metodo è accessibile solo dopo controllo della disponibilità degli account; se un account è occupato da una transazione in corso, al Client in questione sarà chiesto di aspettare fino a che non si liberi. A quel

punto il programma continua ed esegue automaticamente le richieste in attesa.



- *Gestione delle Eccezioni:*

Le eccezioni sono state gestite, nella maggior parte dei casi, all'interno dei blocchi try-catch, dove sono state inseriti più situazioni di possibile cattura, in modo da gestire le casistiche più specifiche. Per fare alcuni esempi: FileNotFoundException in lettura e scrittura da Database o nei vari casi in cui può verificarsi InterruptedException. Le eccezioni lasciano messaggi di avvertimento o sul terminale del Server o sul terminale del Client, nel modo più opportuno, ne vedremo alcuni esempi con l'esecuzione delle istruzioni passo per passo al punto [3.]. Vista la complessità nel percepire tutte le eccezioni specifiche, come si può vedere nel codice, è presente un ultimo catch più generico, che riesce a rivelare nuovi errori non specificatamente gestiti, così da facilitare un futuro miglioramento del codice.
Un secondo modo per gestire le eccezioni lo possiamo osservare all'interno della classe AccountManager, tramite i vari metodi che, nel caso vengano interrotti, lanciano un InterruptedException; oppure alla riga 37 dove viene espressamente lanciata l'eccezione tramite il comando 'throw new IllegalArgumentException'. Queste eccezioni lanciate vengono gestite dalla classe che richiama quei metodi, in questo caso ClientHandler, in un suo blocco try-catch.

- *Gestione ultima transazione (lastT):*

Per poter controllare e memorizzare quando è stata eseguita l'ultima transazione relativa a ogni Account avevamo creato inizialmente un HashMap <String, Transaction> per la gestione delle transizioni effettuate sugli account bancari. Questo comportava l'implemento di metodi aggiuntivi per l'inserimento di una transazione quando veniva effettuata, con controllo dell'esistenza della relativa chiave. In seguito abbiamo semplificato questo passaggio implementando all'interno della classe Account un oggetto Transaction, che si crea alla creazione dell'account stesso, chiamando il suo costruttore all'interno del costruttore di Account. Possiamo, così, richiamare metodi come moveTransaction(), quando necessario, cercando sempre di rispettare la gerarchia che ci impone di passare dalla classe Account.

- *Gestione del Database:*

Abbiamo deciso di inserire un database per evitare il continuo reinserimento degli stessi dati in fare di testing. L'aggiunta del file .csv che ci permette di mantenere i dati persistenti ci ha portato difficoltà dal momento che non essendo stata pensata in precedenza abbiamo dovuto riadattare il codice scritto fino a quel momento. La scrittura su file non ha dato molti problemi usando il metodo writeDatabase(), alla chiusura del Client eseguiamo una semplice trascrizione di ciò che è presente nell'HashMap, usato per memorizza gli account.

Più difficile è stato implementare il metodo readDatabase(), in quanto è necessaria la creazione e la memorizzazione degli account all'avvio del Server. Una volta creato l'oggetto Account, è bastato memorizzarlo all'interno dell'HashMap tramite il metodo addAccount() che già utilizzavamo per il

comando add. Nel Creare l'oggetto Account è stato necessario introdurre nella classe due costruttori appositi che fossero in grado di rispondere alle necessità delle due letture possibili.

2.2. Descrizione degli strumenti utilizzati per l'organizzazione.

Per uno svolgimento efficacie e fruttuoso del progetto abbiamo scelto Visual Studio Code, WhatsApp e GitHub come principali piattaforme per lo svolgimento del programma e per la comunicazione tra i membri del gruppo. Il linguaggio di programmazione utilizzato è Java. Per mantenere una buona organizzazione e obbiettivi chiari abbiamo creato un file Google Docs, condiviso tra i membri, con la traccia del progetto, le specifiche da effettuare e quelle già implementate, che venivano aggiornante dal membro ogni volta che si apportavano modifiche al codice, in modo tale da essere tutti costantemente aggiornati sui progressi in un modo subito chiaro ed efficace. Abbiamo sfruttato GitHub per la condivisione del codice. Abbiamo, inoltre, effettuato spesso incontri per discutere sull'avanzamento del programma e dei problemi riscontarti. Quando la distanza non ci ha consentito di incontrarci di persona, l'utilizzo di WhatsApp per chiamate o per condividere le parti di codice in cui riscontravamo delle difficoltà è stato essenziale.

3. Requisiti e istruzioni per compilare e usare l'applicazione

Per eseguire e utilizzare il programma è necessario seguire i seguenti passaggi, portiamo qui alla luce anche il comportamento del codice in alcuni casi particolari, gestiti grazie all'utilizzo delle eccezioni

1) Aprire un terminale dei comandi per compilare tutti i file con il comando *javac *.java.*
2) Eseguire il Server, in un terminale separato, con il comando *java Server <port>*. È importante scrivere il nome "Server" nello stesso modo in cui è salvato il file nella cartella e sostituire a "<port>" il numero della porta desiderato per la connessione.

```
PS C:\Users\sotti\OneDrive\Desktop\ProgettoSO\LABSO_Cat-FoxBank-master> java Server 9000
Apro la connessione
Funziona
DataBase loaded.
```

Se non viene inserito inserito il numero di porta:

```
PS C:\Users\sotti\OneDrive\Desktop\ProgettoSO\Correzionifinali\LABSO_Cat-FoxBank> java Server
Usage: java Server <port>
```

3) Aprire altri terminali, tanti quanti Client si vogliano utilizzare, ed eseguire in essi il comando *java Client <host> <port>* per aprire la connessione con la classe Client. Bisogna sempre assicurarsi che la scrittura di "Client" coincida con il nome del file presente in cartella e inserire al posto di "<port>" il numero di porta inserito precedentemente nel Server. Inoltre è necessario scrivere al posto di "<host>" il numero *127.0.0.1* o la parola *localhost* .

   Se la connessione ha successo verrà visualizzata la lista dei comandi che il cliente può effettuare:

```
OUTPUT   CONSOLE DI DEBUG   TERMINALE   PROBLEMI   PORTE

PS C:\Users\sotti\OneDrive\Desktop\ProgettoSO\LABSO_Cat-FoxBank-master> java Client 127.0.0.1 9000
Connected to server
Usage:   1.Open <name> <money>
         2.list
         3.transfer <cash> <SenderAccount> <ReciverAccount>
         4.transfer_i <SenderAccount> <ReciverAccount>
         5.quit
```

I comandi possono essere scritti sia in maiuscolo che in minuscolo, grazie a una specifica gestione

Sul terminale dedicato al Server, in caso dell'effettivo funzionamento del programma, la schermata apparirà cosi:

```
Client connected
Thread Thread[Thread-1,5,main] listening...
```

Nel caso in cui sia inserito un numero di porta errato o il server non sia attivo, sarà lanciata un'eccezione e la connessione verrà rifiutata.

```
PS C:\Users\sotti\OneDrive\Desktop\ProgettoSO\Correzionifinali\LABSO_Cat-FoxBank> java Client 127.0.0.1 900
Connection refused
```

Mentre nel caso non si inserisca la porta viene visualizzato il seguente messaggio:

```
PS C:\Users\sotti\OneDrive\Desktop\ProgettoSO\Correzionifinali\LABSO_Cat-FoxBank> java Client 127.0.0.1
Usage: java Client <host> <port>
```

4) Esempi operazioni eseguibili dal Client:
   a. Open

```
open mario 50
Received: made account called: mario
```

Viene sollevata un'eccezione nel momento che si cerca di creare un account con lo stesso nome di uno gia esistente:

```
open mario 60
Received: java.lang.IllegalArgumentException: Account: mario already exist.
```

   b. List

```
list
Received: 1. name: a  balance: 995.0    #:# last transaction: -5.0      at: lun-27/11/2023 14:25:43
Received: 2. name: b  balance: 1000.0 #:# no transaction --> creation date: lun-27/11/2023 14:25:43
Received: 3. name: c  balance: 500.0 #:# no transaction --> creation date: lun-27/11/2023 14:25:37
Received: 4. name: d  balance: 505.0    #:# last transaction: 5.0       at: lun-27/11/2023 14:25:37
Received: 5. name: e  balance: 1500.0    #:# last transaction: 0.0      at: mer-01/01/2020 12:00:00
Received: 6. name: f  balance: 1500.0    #:# last transaction: 0.0      at: mer-01/01/2020 12:00:00
Received: 7. name: mario  balance: 50.0 #:# no transaction --> creation date: lun-27/11/2023 21:26:47
```

   c. Transfer

```
transfer 30 a b
Received: successful transaction
```

Se si vuole trasferire denaro da o verso un account inesistente, l'applicazione ci fa visualizzare un errore:

```
transfer 10 a lucia
Received: java.lang.IllegalArgumentException: Account lucia not found
```

Se il denaro che si vuole trasferire supera il denaro posseduto dall'account:

```
transfer 1000 mario a
Received:  insufficient balance
```

d. Transfer_i

```
transfer_i a b
Received: Start interattive transaction:
Received: Commands:
Received:  1.:move <money>  2.:end
```

Come è possibile osservare dentro la sessione interattiva si utilizzano i seguenti comandi:

i. :move
ii. :end

Se i comandi non vengono riconosciti compare il seguente messaggio:

```
moved
Received: unknown cmd.. try again
```

e. quit

```
quit
Sender closed.
Receiver closed.
Socket closed.
```

In tutti i casi, se non viene inserito uno dei comandi predefiniti, compare una scritta di errore:

```
comando
Received: unknown cmd.. try again
```

5) È possibile digitare il comando ‘quit’ anche sul Server, questo farà interrompere tutti i threads e chiudere la connessione:

```
quit
Interrupting workers...
Interrupting Thread[Thread-1,5,main]...
Main thread terminated.
```

La chiusura del Server causerà, alla successiva operazione su Client, un’eccezione e verrà dichiarato che il server non è più disponibile:

```
open
Server not available
Receiver closed.
```

6) Nei file consegnati sarà presente anche un file csv (DataBaseAccount.csv) nel quale sono già stati caricati alcuni esempi di account per i test da effettuare.

È possibile effettuare modifiche sul file per inserire manualmente nuovi account o per eliminare account già esistenti. Importante è mantenere la sintassi del file:

ogni dato è seguito necessariamente dal carattere: ‘**;**’
ordine sequenza dati (due casi possibili):

*NomeAccount***;***Denaro***;***settimana-giorno/mese/anno ora/minuti/secondi***;***ultimoTrasferimento*

*NomeAccount***;***Denaro***;***settimana-giorno/mese/anno ora/minuti/secondi*

N.B. nel secondo caso si inserisce un account con data e ora relativi alla sua creazione.

DataBaseAccount.csv M

DataBaseAccount.csv

```
1  a;995.0;lun-27/11/2023 14:25:43;-5.0
2  b;1000.0;lun-27/11/2023 14:25:43
3  c;500.0;lun-27/11/2023 14:25:37
4  d;505.0;lun-27/11/2023 14:25:37;5.0
5  e;1500.0;mer-01/01/2020 12:00:00;0.0
6  f;1500.0;mer-01/01/2020 12:00:00;0.0
```