Laboratorio di Sistemi Operativi

A.A. 2022-23

INDICE:

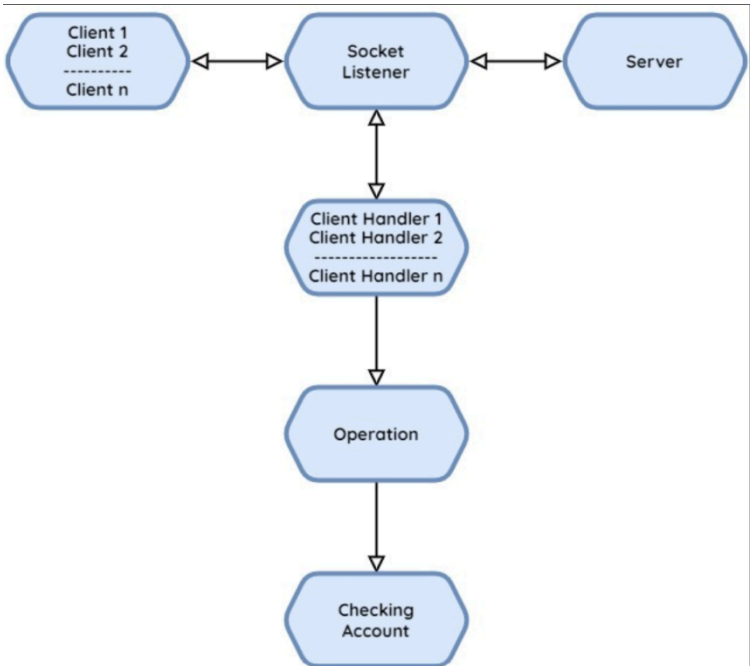
I PARTE - ARCHITETTURA PROGETTO

1 – Funzionamento generale del progetto

Il progetto è strutturato in otto classi:

- Client:
    - Sender
    - Receiver
- Server:
    - SocketListener
    - ClientHandler
    - Operation
    - CheckingAccount

Inizialmente erano presenti classi aggiuntive che rappresentavano i trasferimenti di denaro e il tipo di transazione. Queste sono state eliminate in quanto sono necessarie le informazioni relative solo all'ultima transazione avvenuta e non a tutte quelle realizzate. Per questo motivo abbiamo ritenuto opportuno rappresentare tali informazioni come stringa presente nella classe "CheckingAccount".



2 - Cosa Fanno i singoli codici java?

Client

Il Client è una classe che richiede in input l'indirizzo del server e la porta a cui connettersi. Viene creato un oggetto "Socket" che permette la connessione al server utilizzando l'host e la porta specificati. Vengono creati due thread: uno per l'invio ("sender") e uno per la ricezione ("receiver") dei dati. Questi thread ricevono come argomenti l'oggetto "Socket" per la connessione al server. Quando vengono avviati si inizia l'esecuzione dei metodi "run". Attraverso il comando "join()" Il programma attende che entrambi i thread abbiano terminato la loro esecuzione. Successivamente, la connessione del socket viene chiusa con il comando "s.close()". Se l'attesa dei thread viene interrotta da un'eccezione "InterruptedException", il programma termina. Se si verifica un'eccezione di tipo "IOException" durante il tentativo di connessione al server, viene stampato un messaggio di errore "Connessione con il server non avvenuta".

Server

Classe che permette di aprire il Server prendendo in input da riga di comando il numero della porta. Viene creato un oggetto ServerSocket (che lancia un' IOException) e un thread di SocketListener, classe che utilizza Socket e Thread per ascoltare le richieste del Client. Questo thread gestisce tutte le connessioni dei diversi client. Quando il thread serverThread viene avviato, viene inizializzata una stringa command per leggere i comandi inseriti dall'utente. Il programma entra in un ciclo while che continua fino a quando l'utente inserisce il comando "quit". Durante ogni iterazione del ciclo il programma attende l'input dell'utente tramite userInput.nextLine(). Quando l'utente inserisce "quit", il ciclo termina.
Dopo il ciclo while, il thread serverThread viene interrotto e il programma stampa un messaggio "Main thread terminated." per indicare che il thread principale è terminato. Il metodo ".join" lancia una InterruptedException che viene catturata. Alla fine viene chiuso lo Scanner e viene terminato il programma.

Sender
Implementa l'interfaccia "runnable". E' progettata per essere eseguita come un thread separato e si occupa di inviare input da tastiera a un socket. La classe Sender ha un campo dati Socket s, che rappresenta il socket su cui verranno inviati i dati, poi implementa il metodo run() dell'interfaccia Runnable.
Viene creato un oggetto PrintWriter, chiamato to, per scrivere i dati nel socket associato all'oggetto Socket s. All'interno di un while, il programma legge l'input dell'utente da userInput.nextLine() e lo memorizza nella stringa request. Questo ciclo è progettato per essere eseguito infinitamente finché non avviene un'interruzione, che si verifica se il thread corrente è stato interrotto tramite Thread.interrupted(). Se il thread non è stato interrotto, il Sender invia la richiesta inserita dall'utente al server utilizzando il socket (to.println(request)). Se la richiesta inserita dall'utente è "quit", il ciclo while termina, e il thread Sender si chiude.

Receiver
Implementa l'interfaccia "Runnable" e si occupa di ricevere dati da un socket e di stamparli a schermo. Il costruttore riceve un oggetto "Socket" e un thread "Sender" e li memorizza nelle variabili di instanza. Quando il thread viene avviato viene eseguito il metodo "run()".
All'interno del blocco "try/catch" viene creato un oggetto Scanner chiamato "from" per leggere i dati dal flusso di input del socket "s.getInputStream()".
Per indicare che la ricezione è aperta viene impostata la variabile "open" su "true".
Il ciclo "while" legge continuamente linee di testo dal socket tramite lo Scanner.
Se il server invia la stringa "quit", il ciclo termina. In questo caso, la variabile "open" viene impostata su "false" per chiudere la ricezione e il ciclo viene interrotto.
Altrimenti, la stringa ricevuta viene stampata a schermo.
Nel caso in cui si verifichi un'eccezione di tipo "IOException" viene stampato un messaggio di errore.
Se si verifica un'eccezione di tipo "NoSuchElementException" viene stampato un messaggio che indica che il server non è raggiungibile.
Infine, nella clausola "finally" viene stampato il messaggio "Receiver closed" e il thread "sender" viene interrotto.

SocketListener
Implementa l'interfaccia "Runnable" e si occupa di ascoltare le connessioni dei vari Client in un Socket Server. Il costruttore accetta un oggetto ServerSocket e lo memorizza nella variabile server.
Successivamente viene fatto override del metodo "run()" dell'interfaccia Runnable, in cui è presente un loop while che resta in ascolto per eventuali client connessi, con un timeout di 5 secondi, il che vuol dire che il server accetterà nuove connessioni per un massimo di 5 secondi, dopodiché passerà oltre. Poi, viene chiamato this.server.accept(), che è un'operazione bloccante che attende la connessione di un client. Se la connessione avviene, viene creato un nuovo thread che gestirà la comunicazione con il client utilizzando ClientHandler Operation per gestire le varie operazioni.
Il nuovo thread viene avviato (handlerThread.start()), quindi viene aggiunto alla lista children, ovvero una lista di thread, ciascuno dei quali rappresenta un gestore di client connessi.

Se si verifica un SocketTimeoutException, il codice continua l'esecuzione senza interrompere il loop. Se si verifica un'eccezione IOException durante l'accettazione di una connessione o la chiusura di un socket, il loop termina.
Infine, dopo che il loop è terminato, vengono chiusi tutti i thread associati ai client connessi e quindi il socket del server viene chiuso tramite this.server.close().

ClientHandler

Implementa l'interfaccia “Runnable”; è responsabile della gestione delle comunicazioni tra il client e il server e fornisce una serie di comandi per eseguire operazioni su conti bancari. Il costruttore riceve un oggetto “Socket” per la connessione e un oggetto “Operation” (classe che gestisce le operazioni bancarie). Il metodo “run()” gestisce la comunicazione con il client attraverso il socket. Viene creato un oggetto Scanner “from” per leggere dati dal flusso di input del socket e un oggetto PrintWriter “to” per inviare dati al client tramite il flusso di output.

Viene inviato un messaggio di benvenuto al client con le informazioni sulla banca e le modalità di trasferimento disponibili. Vengono inizializzate alcune variabili di controllo come “closed” (per controllare se la connessione è chiusa) e “sessionActive” (per verificare se è in corso una sessione interattiva). Il ciclo principale riceve i comandi inviati dal client, li elabora e risponde di conseguenza. Se “sessionActive” è “false”, vengono gestiti i comandi standard:

- “info” (per ottenere la lista dei comandi disponibili);

```
Siamo qui per rendere il trasferimento dei tuoi fondi facile e comodo, indipendentemente dalla modalità che preferisci utilizza
re. La nostra banca si impegna a fornirti soluzioni finanziarie personalizzate e un'esperienza bancaria all'avanguardia. Grazie
 per aver scelto di fare affari con noi, siamo pronti ad assisterti in ogni passo del percorso.
Per poter vedere la lista dei comandi e come devono essere utilizzati digitare "info"
info
Lista dei comandi disponibili:
- list --> Per poter visualizzare la lista dei conti aperti
- open --> Per poter aprire un conto (utilizzo: open <NomeConto> <Bilancio Iniziale>)
- transfer --> Per trasferire dei soldi da un conto all'altro (utilizzo: transfer <importoDaTrasferire> <Mittente> <Destinatari
o>)
- transfer_i --> Per avviare una sessione interattiva tra due conti (utilizzo: transfer_i <Mittente> <Destinatario>)
- quit --> Per chiudere la connessione con il server
Comandi disponibili nella sessione interattiva:
:move --> Per poter spostare denaro tra il primo conto digitato e il secondo
:end --> Per concludere la sessione interattiva
```

```
Request da parte del cliente: info
```

- “quit” (per chiudere la connessione);

```
quit
Sender closed
Connection closed
Receiver closed
Socket closed
```

```
quit
Timeout, continuing...
Interrupting children...
Interrupting Thread[#23,Thread-1,5,]...
Interrupting Thread[#30,Thread-8,5,main]...
Main thread terminated.
```

- “list” (per ottenere la lista dei conti aperti);

```
list
Elenco dei conti presenti:
Mario: 657.0
DATA: 19-09-2024,AMMONTARE: -343.0, ALTRO CONTO: Chiara
Luigi: 1002131.0
Nessuna transazione effettuata
Chiara: 23755.0
DATA: 19-09-2024,AMMONTARE: +343.0, ALTRO CONTO: Mario
```

- “open” (per aprire un nuovo conto);

```
open Mario 1000
Nuovo conto Mario creato con saldo iniziale di 1000.0
```

- ”transfer” (per effettuare un trasferimento tra conti);

```
transfer 343 Mario Chiara
Transazione effettuata con successo
```

```
transfer t Mario Chiara
Il comando digitato è sbagliato utilizzare il comando info per poter vedere il formato corretto del comando desiderato. Per la sessione interattiva
puoi utilizzare i comandi: :move <importo> oppure :end
```

- “transfer_i” (per avviare una sessione interattiva).

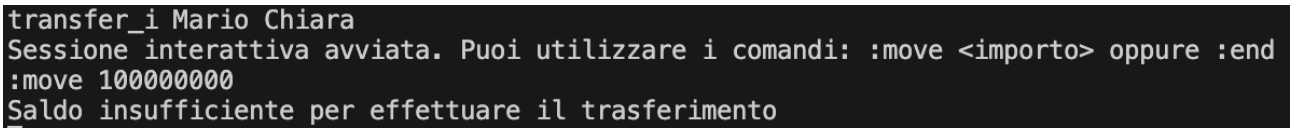
  Se “sessionActive” è “true” il client è in una sessione interattiva e i comandi specifici per questa modalità vengono gestiti. Questi sono:

  - “:move” che sposta denaro tra conti;
  - “:end” che termina la sessione interattiva;
  - “quit” che chiude la connessione.

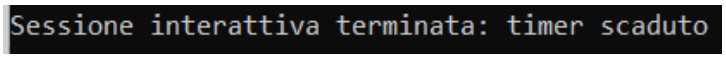
```
transfer_i Luigi Chiara
Sessione interattiva avviata. Puoi utilizzare i comandi: :move <importo> oppure :end
:move 100
Transazione effettuata con successo
:move 32145
Transazione effettuata con successo
:end
Sessione interattiva terminata. Ora puoi utilizzare i comandi standard
```

Vengono gestite varie eccezioni, ad esempio NoSuchElementException per errori nell'input del client e illegalArgumentException per errori nei comandi inseriti.

```
Sessione interattiva avviata. Puoi utilizzare i comandi: :move <importo> oppure :end
:move t
Il comando digitato è sbagliato utilizzare il comando info per poter vedere il formato corretto del comando desiderato. Per la sessione interattiva
puoi utilizzare i comandi: :move <importo> oppure :end
```



Il metodo “clock()” crea un timer che verrà utilizzato per impostare un timeout su sessioni interattive. Il timer parte all’inizio della sessione interattiva e viene riavviato ogni volta che viene richiamato il comando “:move”. Se il timer scade, la sessione interattiva viene terminata. Questo viene implementato nella classe interna “timer”.



Operation

Classe che gestisce le operazioni bancarie.
Sono presenti due liste: una per memorizzare i conti bancari e un’altra per tenere traccia dei conti impegnati nelle operazioni.
Le liste vengono inizializzate nel costruttore.
Metodi presenti:

- “addAccount(String accountName, double initialBalance): aggiunge un nuovo conto bancario alla lista dei conti;
- “getAccountslist()”: restituisce una stringa contenente un elenco di tutti i conti bancari presenti nella lista “checkingAccounts”;
- “containsAccount(String accountName)”: verifica se un conto con il nome specificato è già presente nella lista;
- “findAccount(String accountName)”: trova e restituisce un oggetto “checkingAccount” corrispondente al nome del conto specificato. Se il conto non viene trovato, viene lanciata un'eccezione;
- “transferBalance(String s, String r, double amount, PrintWriter to)”: gestisce il trasferimento di denaro tra due conti bancari. Verifica se gli account sono impegnati in altre operazioni e attende se necessario. Successivamente, esegue il trasferimento di denaro tra i due account e stampa il risultato nel flusso di output specificato (“to”);
- “startSection(String s, String r, PrintWriter to)”:avvia una sessione interattiva tra due conti bancari. Verifica se gli account sono occupati da altre operazioni e attende se necessario. Inoltre, notifica tutti i thread in attesa se l'operazione può essere eseguita;
- “interactiveSessionMOVE(String s, String r, double amount, PrintWriter to)”: gestisce l'operazione di spostamento di denaro durante una sessione interattiva tra due conti bancari e stampa il risultato nel flusso di output specificato (“to”);
- “interactiveSessionEND (String s, String r): conclude una sessione interattiva tra due conti bancari e rimuove gli account dalla lista “busies”. Notifica tutti i thread in attesa che la sessione sia terminata.

CheckingAccount

Rappresenta un conto bancario. Il costruttore prende in input il nome dell'account e il bilancio iniziale. Sono presenti diversi metodi get e set per ottenere e modificare i dati. Il metodo "moveMoney (CheckingAccount receiver, double amount) gestisce il trasferimento di denaro da questo conto a un altro conto ("receiver").
Verifica se il saldo del conto è sufficiente per effettuare il trasferimento.
Se sì, riduce il saldo di questo conto e aumenta il saldo del conto ricevente), registra l'operazione come l'ultima transazione per entrambi i conti e restituisce un messaggio di conferma o di errore. Il metodo privato "formatDate(Date date)" riceve una data come parametro e la formatta nel formato "dd-mm-yyyy" (utilizzato per registrare la data delle transazioni).

3 - Suddivisione del lavoro

A seguito di alcuni incontri iniziali avvenuti online, tramite Discord, abbiamo ritenuto opportuno dividere il lavoro volta per volta, incontrandoci settimanalmente per la discussione e risoluzione collettiva dei problemi riscontrati. Questi incontri ci hanno permesso di modificare le implementazioni dei codici così da coordinare le varie parti stabilendo una base da cui partire.
Dopo aver discusso su una base da cui iniziare il progetto la suddivisione del lavoro è stata gestita nel modo seguente:

- - e - si sono occupati principalmente della gestione del Client e del Server, con le relative classi annesse, della gestione dei thread e della base dello switch del ClientHandler;
- - e - si sono occupati della gestione della concorrenza, ovvero della classe Operation, del completamento della classe ClientHandler e della realizzazione di CheckingAccount.

Dopo ogni implementazione i codici venivano rivisti da tutto il gruppo e veniva eseguito un debug costante in base alle prove che ognuno di noi faceva sul codice.
Il risultato finale è quindi frutto della collaborazione tra tutti i membri del gruppo e delle modifiche che ognuno di noi faceva di volta in volta.

PARTE 2 - DESCRIZIONE DEL PROCESSO DI IMPLEMENTAZIONE

1 - Descrizione dei problemi e degli ostacoli con le relative soluzioni

Per quanto riguarda l'implementazione della struttura iniziale del programma non abbiamo incontrato molti ostacoli. Abbiamo creato una sorta di mappa basandoci sulle indicazioni date nella traccia, e condiviso le varie idee su come iniziare a realizzare i codici. Successivamente, dopo qualche giorno di lavoro, era già stata fatta una prima bozza del progetto funzionante, a cui mancava la gestione della concorrenza e la gestione delle eccezioni.

In seguito ci siamo concentrati sulla gestione di quest'ultima, in particolare abbiamo deciso in quali classi inserire dei blocchi try-catch, dove usare i comandi throws e fatto alcune prove per vedere come queste influivano sulla sincronizzazione del programma.
Questo perché, contestualmente all'implementazione della parte di concorrenza, ci siamo accorti che era necessario cambiare qualche parte di gestione delle eccezioni, visto che in alcuni casi lanciare un'eccezione causava il blocco dell'esecuzione della sessione interattiva. Questo era dovuto al fatto che, nella bozza iniziale del programma, non c'era la classe "Operation" ma l'obiettivo era quello di gestire la concorrenza nel ClientHandler. Questo non permetteva la creazione di lock in comune, ne conseguiva la creazione di lock diversi per ogni ClientHandler. La non presenza di un metodo che termina la sessione interattiva ha portato all'impossibilità di uscire dalla sessione una volta riscontrata un'eccezione.
Dopo aver fatto diversi tentativi per capire quale metodo di gestione della concorrenza fosse più adatto alle nostre esigenze abbiamo optato per l'uso delle strutture già implementate e disponibili in Java, ovvero synchronized e notifyAll.
In seguito abbiamo creato la classe Operation per gestire la concorrenza.

2 - Descrizione degli strumenti utilizzati per l'organizzazione

Per quanto riguarda l'organizzazione interna al gruppo, abbiamo deciso insieme che piattaforme usare prima ancora di iniziare ad implementare il codice.
Innanzitutto, abbiamo deciso di utilizzare Visual Studio Code come editor su cui scrivere i codici, compilando dal prompt di Windows, sistema operativo condiviso da tutti e quattro i membri del gruppo.
Successivamente abbiamo creato il gruppo e la cartella condivisa su GitHub per inserire i codici a portata di tutti e tenere traccia delle modifiche fatte nel corso dell'implementazione.

Per quanto riguarda la comunicazione tra noi membri del gruppo, dopo aver creato un gruppo WhatsApp per scambiarci le informazioni di massima, abbiamo aperto un server su Discord. Quest'ultimo è stato indispensabile, visto che per tutto il tempo in cui abbiamo lavorato al progetto siamo sempre stati in luoghi diversi, e quindi trovare un'occasione per vedersi tutti insieme di persona non è stato possibile.
In media abbiamo fatto circa due chiamate di gruppo a settimana per tenerci aggiornati sui problemi e decidere come agire, inoltre è stata utile anche la chat del server, metodo immediato per condividere i codici in aggiunta a GitHub.

Infine, abbiamo condiviso un documento Google tra di noi dove di volta in volta scrivevamo tutte le modifiche e le aggiunte al progetto, inserendo la data in cui venivano fatte, per aiutarci ulteriormente a gestire il lavoro e tenere traccia dei vari problemi riscontrati.

III PARTE - ISTRUZIONI PER COMPILARE E USARE LE APPLICAZIONI CONSEGNATE

1 - Guida all'Utilizzo dell'Applicazione Server Multi-Client

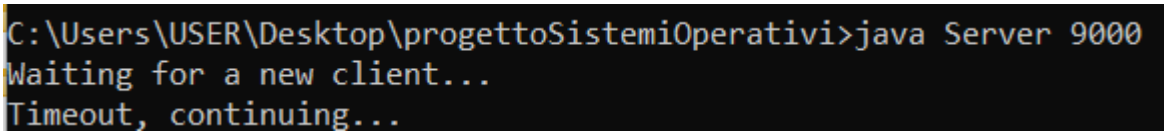
Passo 1: Aprire un prompt dei comandi per il server e diversi prompt dei comandi per i client. Per aprire un prompt dei comandi su Windows, premere il tasto Win + R, digitare "cmd" e premere Invio. Assicurarsi di aprire tanti prompt dei comandi a seconda di quanti client si vogliono usare (è consigliato l'utilizzo di almeno due client per vedere come funziona la concorrenza).

Passo 2: Selezionare la cartella in cui si trovano i file dell'applicazione con il comando cd. Ad esempio, se i file dell'applicazione sono nella cartella "C:\MiaApp", digitare cd C:\MiaApp nel prompt dei comandi per posizionarsi nella cartella corretta.

Passo 3: Utilizzare il comando javac *.java per compilare tutti i file Java presenti all'interno della cartella. Questo comando compila tutti i file Java presenti nella cartella corrente per garantire una corretta esecuzione.

Passo 4: In un terminale separato, eseguire il Server con il seguente comando:
java Server n_porta.
È importante che il nome "Server" coincida con il nome del file presente nella cartella. Sostituire "n_porta" con il numero di porta desiderato per l'apertura del server.

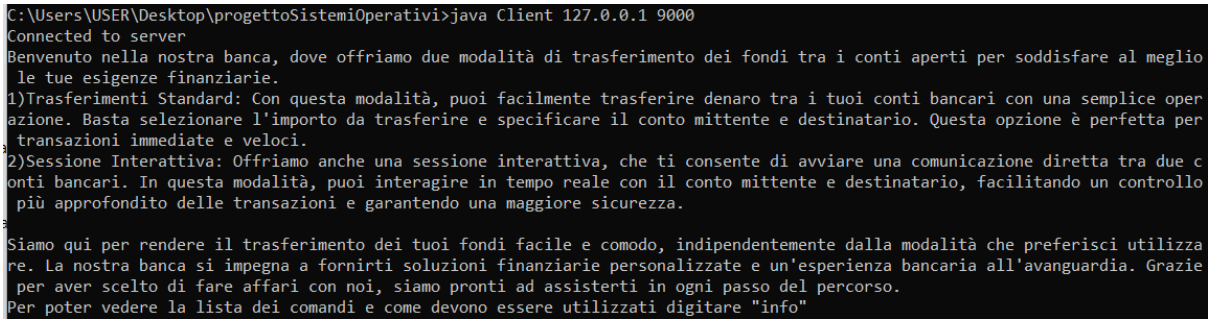
Passo 5: Se il server si apre correttamente, sullo schermo si vedrà il messaggio "Waiting for a new client...". Questo indica che il server è in attesa di connessioni dai client.



Passo 6: Eseguire sugli altri prompt dei comandi la classe Client con il seguente comando:

java Client localhost n-porta

"localhost" può essere sostituito con "127.0.0.1". Assicurarsi che il valore di "n-porta" sia lo stesso utilizzato per l'apertura del server. Se la connessione ha successo, verrà visualizzato il messaggio "Connected to server" insieme al messaggio di benvenuto nel sistema.



Passo 7: Per visualizzare i comandi disponibili e la loro funzionalità, si deve digitare il comando "info" all'interno del client e premere Invio.