16 giugno 2020

Un supermercato permette l'accesso di un numero massimo di compratori pari a N. Quando si raggiunge tale numero massimo, non viene permesso nessun nuovo ingresso finche' il numero di compratori all'interno del supermercato non diventa N − K, dove $0 \leq K < N$.
Il supermercato ha due code di ingresso, quella normale e quella per gli ingressi prioritari riservata a anziani e donne in stato di gravidanza.
Realizzare la classe Supermercato con almeno i seguenti metodi/cosruttori:
• Supermercato(int N, int K): costruttore; lancia IllegalArgumentException se i parametri non sono validi;
• void entrata(TipoVisitatore tipo): un thread chiede di entrare al Supermercato specificando se si tratta di un ingresso prioritario o meno; l'operazione puo' essere bloccante;
• void uscita(): un thread notifica la propria uscita dal museo.

TipoVisitatore deve essere un enumerato con i valori ORDINARIO, ANZIANO, GRAVIDANZA.

Si consideri un sistema composto da N thread, ognuno identificato da un nome univoco, la cui esecuzione è schedulata come segue.
L'esecuzione dei thread è organizzata in run. In ogni run, un thread va in esecuzione solo una volta. Una run termina quando tutti i thread sono andati in esecuzione. La prossima run può iniziare solo quando quella corrente è terminata. All'interno di una run, le esecuzioni dei thread sono sequenzializzate sempre nello stesso ordine. Tale ordine di esecuzione viene fissato nella prima run secondo la politica FCFS.
Realizzare una classe Scheduler che implementa tale politica di scheduling dotata dei seguenti metodi e costruttori:
• Scheduler(int num): costruisce uno scheduler per num thread.
• public void firstRun(String threadName) che permette al thread di nome threadName di richiedere l'esecuzione nella prima run. Lancia l'eccezione InvalidRunException se invocato dopo il primo run.
• public void nextRun(String threadName) che permette al thread di nome threadName di richiedere l'esecuzione in tutte le run successive alla prima.
• public void runCompleted() che permette ad un thread di notificare il completamento della sua esecuzione nella run corrente.

Un varco è composto da N tornelli. Ogni tornello permette di passare una persona alla volta. Una persona può passare da qualunque tornello. Se tutti i tornelli sono occupati, una persona deve attendere che se ne liberi almeno uno. Le persone passano dal varco fintanto che questo è aperto. Quando il varco viene chiuso nessuno può più passare e viene restituito il numero di persone che sono transitate.
Realizzare la classe Varco con i seguenti metodi:
• Varco(int numTornelli) che costruisce contestualmente un varco composto da un numero di tornelli specificati dal parametro numTornelli. I tornelli sono identificati da un numero che va da 0 a numTornelli-1.
• Varco() che costruisce un varco costituito da un solo tornello;
• int occupaTornello() throws VarcoChiusoException, InterruptedException che permette ad una persona di occupare un tornello prima di transitarci attraverso. Se nessun tornello è libero, la persona attende il primo che si libera. Restituisce il numero del tornello occupato.
• void liberaTornello(int i) che permette ad una persona di liberare il tornello i, non appena vi ha transitato attraverso.
• int chiudiVarco() che permette di chiudere il varco.

In un ufficio ci sono N operatori, ognuno identificato da un numero intero compreso tra 1 e N. Per regolare il flusso degli utenti, l'ufficio utilizza un salva-code costituito da un distributore di biglietti. Il distributore eroga ticket agli utenti. Ciascun ticket riporta un numero progressivo. Quando un operatore si libera notifica questo evento al salva-code che visualizza su un display il prossimo biglietto da servire e il numero dell'operatore. Realizzare il salva-code come una classe Java che fornisce le seguenti operazioni:

- SalvaCode(): costruttore;
- int getTicket() che restituisce all'utente un ticket;
- int waitForTurn(int tkt) che prende in ingresso il ticket tkt e restituisce il numero di un qualunque operatore libero che servirà l'utente che detiene quel ticket. L'operazione è bloccante se tutti gli operatori sono occupati.
- int nextTicket(int opr) che prende il numero opr dell'operatore e restituisce il ticket che tale operatore servirà.

Un insieme di thread compete per una risorsa condivisa. Ciascun thread richiede l'uso della risorsa per mezzo dell'operazione request e la rilascia al gestore per mezzo dell'operazione release. L'operazione request può essere bloccante nel caso che la risorsa sia già allocata ad un altro thread. Nel caso che due o più thread siano sospesi in attesa della risorsa, la risorsa verrà allocata al thread che è disposto a pagare il fee maggiore. Il fee è un intero che ciascun thread passa come parametro dell'operazione request ed compreso tra zero e FEE. L'operazione request lancia un'eccezione FeeOutOfBoundException se un thread specifica un fee minore di zero o maggiore di FEE.
Realizzare le classi FeeOutOfBoundExceptione e Gestore. La classe Gestore contiene almeno i seguenti metodi:
- Gestore(String nome): costruisce un gestore per la risorsa di nome nome;
- void request(int fee): richiede l'uso della risorsa, dichiarando di essere disposto a pagare fee in caso di contesa:
- void release(): rilascia la risorsa;
- String toString(): restituisce il nome della risorsa gestita;

Consideriamo un ascensore per hotel. L'ascensore memorizza un numero qualunque di destinazioni prenotate e
accetta nuove prenotazioni in qualunque momento. L'ascensore decide la prossima destinazione in base alle seguenti regole: fino a quando ci sono destinazioni prenotate ancora da raggiungere nella direzione in cui si sta muovendo, non cambia direzione; si ferma solo se non ci sono destinazioni prenotate.

Realizzare la classe AscensoreHotel contenente almeno i seguenti metodi:
- AscensoreHotel(int min, in max): costruisce un nuovo ascensore che gestisce i piani da min a max (compresi).
- void prenota(int piano): esegue una nuova prenotazione con destinazione il piano passato come argomento, quindi attende che l'ascensore vi arrivi.
- void muovi(): attende che ci sia almeno una prenotazione, quindi muove di un piano l'ascensore verso la prossima destinazione e stampa la stringa "Ascensore al piano piano", dove piano è il nuovo piano a cui si trova l'ascensore.

Si assuma che per ogni ascensore ci sia un unico thread che chiama ciclicamente il metodo muovi() e più thread che chiamano prenota().

18 giugno 2020

Un museo consente una visita guidata alla volta, e ogni visita è composta esattamente da *N* visitatori e una guida (supponiamo che ci sia sempre un numero sufficiente di visitatori). Le visite sono numerate. Ogni visitatore deve prima prenotare una visita, ottenendo un numero di visita, quindi attendere che la visita inizi. Una guida deve prima attendere che siano entrati tutti i visitatori per la prossima visita. I visitatori possono uscire in qualunque momento durante la visita. La guida attende che siano usciti tutti i visitatori prima di uscire a sua volta, terminando così la visita.

Realizzare una classe *Museo* che implementi quanto appena descritto con almeno i seguenti metodi:

- Un costruttore *Museo(int N)*, dove *N* è il numero di visitatori richiesti per ogni visita al museo.
- Un metodo *int prenotaVisita()*, usato dai visitatori per ottenere il numero della visita a cui partecipare (la prima visita non ancora al completo).
- Un metodo *void entraVisitatore(int nVisita)*, usato dai visitatori per iniziare la propria visita. Il metodo deve prima attendere che la visita eventualmente già in corso termini e che la prossima abbia numero *nVisita*, quindi stampare la stringa "Entra visitatore" e attendere ulteriormente finché la visita non inizia.
- Un metodo *void esceVisitatore()*, usato dai visitatori per uscire dal museo. Il metodo deve stampare la stringa "Esce visitatore".
- Un metodo *void entraGuida()*, usato dalle guide per iniziare una visita. Il metodo deve prima attendere che l'eventuale visita già in corso termini, quindi attendere che tutti i visitatori per la prossima visita siano entrati, e solo allora dare inizio alla visita stampando "Inizia visita *n*", dove *n* è il numero della visita.
- Un metodo *void esceGuida()*, usato dalle guide per terminare una visita. La guida deve prima attendere che tutti i visitatori siano usciti, quindi stampare "Termina visita *n*".

Definire una classe Contenitore contenente un campo pubblico dati, definito come un array di interi la cui dimensione è passata come unico argomento al costruttore della classe. I thread che vogliono accedere a questo campo devono prima acquisire il diritto di lettura o di scrittura e rilasciarlo quando hanno terminato. Per permettere ciò, la classe Contenitore fornisce i metodi (tutti void) acquisisci_lettura(), acquisisci_scrittura(), rilascia_lettura() e rilascia_scrittura(). Tali metodi garantiscono che:
1) il diritto di scrittura venga concesso solo quando nessuno possiede il diritto di lettura o di scrittura;
2) il diritto di lettura venga concesso solo quando nessuno possiede il diritto di scrittura (non importa se qualcuno possiede il diritto di lettura).
Definire inoltre una classe Lettore che estenda Thread, con costruttore Lettore(Contenitore e, int n). Ogni thread Lettore deve leggere (avendone acquisito il diritto) tutti gli elementi del campo dati di e, sospendendosi per 10 msec tra una lettura e la successiva e sommando i valori letti. Se la somma complessiva è multipla di n, deve stampare "Lettore: OK", altrimenti deve stampare "Lettore: ERRORE". Definire infine una classe Scrittore che estenda Thread, con costruttore Scrittore(Contenitore e). Ogni thread Scrittore deve scrivere (avendone acquisito il diritto) in tutti gli elementi del campo dati di e, incrementando di 1 ogni valore letto e sospendendosi per 10 msec tra una scrittura e l'altra.

Una classe Mailbox ha i metodi pubblici void invia(int tipo, String msg) e String ricevi(int tipo), dove tipo è un numero compreso tra 0 e 3 (estremi inclusi). La classe mantiene una coda di messaggi per ciascun tipo. Il metodo invia accoda un nuovo messaggio msg di tipo tipo. Il metodo ricevi attende che sia disponibile almeno un messaggio di tipo tipo, e di questi restituisce quello accodato da più tempo, rimuovendolo dalla Mailbox.
Due classi Lettore e Scrittore, che estendono Thread e i cui costruttori ricevono un int tipo e un oggetto Mailbox. I thread di classe Scrittore inviano sul Mailbox tre messaggi di tipo tipo e di forma “Messaggio i”, (i=0,1,2), seguiti da un messaggio “Stop” (sempre di tipo tipo), quindi terminano. I thread di classe Lettore leggono tutti i messaggi di tipo tipo dal Mailbox e li stampano sul video, ciascuno preceduto dalla stringa “Tipo tipo: ”. Se il messaggio è “Stop”, terminano.

Definire:
1) una classe Msg con campi String msg e int prio;
2) una classe Log con metodi pubblici void log(int prio, String msg) e Msg[] observe(int min, int max). La classe colleziona messaggi di priorità compresa tra 0 (minima) e 5 (massima), passati tramite il metodo log(), e permette di estrarli tramite il metodo observe(). Il metodo observe() attende che vi sia almeno un messaggio con priorità compresa tra min (incluso) e max (escluso), quindi restituisce tutti i messagi in questo intervallo, estraendoli dal Log.
3) LogWriter, che estende Thread, con costruttore LogWriter(int id, int prio, Log l);
4) LogReader, che estende Thread, con costruttore LogReader(int min, int max, Log l).
I thread di classe LogWriter devono inviare tre messaggi sul Log l, con contenuto "Thread id mess i" (i=0,1,2) e priorita' rispettivamete prio, prio+1 e prio+2. I thread di classe LogReader devono attendere che il Log l contenga almeno un messaggio di priorità compresa tra min (incluso) e max (escluso), quindi stampare una riga della forma "Prio prio: msg" per ogni messaggio estratto dal Log

In un aeroporto ci sono P piazzole di parcheggio ed R piste di decollo ed atterraggio. Un aereo per poter iniziare l'atterraggio deve riservare una pista ed una piazzola. Ad atterraggio ultimato, l'aereo rilascia la pista ed occupa la piazzola. Al decollo, un aereo riserva una pista e rilascia la piazzola che sta occupando. Al termine del decollo, l'aereo rilascia la pista.

Un aereo può essere modellato come un thread che esegue ciclo senza fine costituito dai seguenti passi:

Inizio decollo
Decollo
Fine decollo
Volo
Inizio atterraggio
Atterraggio
Fine atterraggio

realizzare il gestore Aeroporto che fornisce le seguenti operazioni:

- Aeroporto(int P, int R) crea un aeroporto con P piste e R piazzole
- int inizioDecollo(int piazzola) che richiede l'allocazione di una pista e rilascia la piazzola occupata dall'aereo e trasmessa come parametro. La pista viene restituita come valore di ritorno. L'esecuzione dell'operazione si sospende se non c'é almeno una pista disponibile.
- void fineDecollo(int pista) che rilascia la pista riservata all'aereo e trasmessa come parametro.
- Risorse inizioAtterraggio() che inizia l'atterraggio richiedendo l'allocazione di una pista e di una piazzola che vengono restituite incapsulate nel valore di ritorno di classe Risorse. L'esecuzione dell'operazione si sospende se una pista ed una piazzola non sono entrambe disponibili.
- void fineAtterraggio(int pista) che rilascia la pista riservata all'aereo e trasmessa come parametro.

La classe Risorse è definita così:

```
class Risorse {
    private int pista;
    private int piazzola;

    public Risorse(int pista, int piazzola) {
        this.pista = pista;
        this.piazzola = piazzola;
    }

    public int getPista() {
        return pista;
    }

    public int getPiazzola() {
        return piazzola;
    }
}
```

07 luglio 2020

Un museo permette la visita contemporanea di un numero massimo di visitatori pari a N. Quando si raggiunge tale numero massimo, non viene permesso nessun nuovo ingresso finché il numero di visitatori allinterno del museo non diventa $N - K$, dove $0 \leq K < N$. Inoltre il museo ha due code di ingresso, quella normale e quella per gli ingressi prioritari riservata a minori, anziani e prenotazioni.

Progettare il monitor *Museo* con almeno i seguenti metodi:

- *Museo(int N, int K)*: costruttore; solleva *IllegalArgumentException* se i parametri non sono validi;
- *void entrata(TipoVisitatore tipo)*: un thread chiede di entrare al museo specificando se si tratta di un ingresso prioritario o meno; l'operazione può essere bloccante;
- *void uscita()*: un thread notifica la propria uscita dal museo.

TipoVisitatore deve essere un enumerato che ammetta almeno i valori *ORDINARIO*, *ANZIANO*, *MINORE* e *PRENOTATO*.

Un varco è composto da N tornelli. Ogni tornello permette di passare una persona alla volta. Una persona può passare da qualunque tornello. Se tutti i tornelli sono occupati, una persona deve attendere che se ne liberi almeno uno. Le persone passano dal varco fintanto che questo è aperto. Quando il varco viene chiuso nessuno può più passare e viene restituito il numero di persone che sono transitate.
Realizzare la classe Varco con i seguenti metodi:
• Varco(int numTornelli) che costruisce contestualmente un varco composto da un numero di tornelli specificati dal parametro numTornelli. I tornelli sono identificati da un numero che va da 0 a numTornelli-1.
• Varco() che costruisce un varco costituito da un solo tornello;
• int occupaTornello() throws VarcoChiusoException, InterruptedException che permette ad una persona di occupare un tornello prima di transitarci attraverso. Se nessun tornello è libero, la persona attende il primo che si libera. Restituisce il numero del tornello occupato.
• void liberaTornello(int i) che permette ad una persona di liberare il tornello i, non appena vi ha transitato attraverso.
• int chiudiVarco() che permette di chiudere il varco.

Un insieme di thread compete per una risorsa condivisa. Ciascun thread richiede l'uso della risorsa per mezzo dell'operazione request e la rilascia al gestore per mezzo dell'operazione release. L'operazione request può essere bloccante nel caso che la risorsa sia già allocata ad un altro thread. Nel caso che due o più thread siano sospesi in attesa della risorsa, la risorsa verrà allocata al thread che è disposto a pagare il fee maggiore. Il fee è un intero che ciascun thread passa come parametro dell'operazione request ed compreso tra zero e FEE.
L'operazione request lancia un'eccezione FeeOutOfBoundException se un thread specifica un fee minore di zero o maggiore di FEE.
Realizzare le classi FeeOutOfBoundExceptione e Gestore. La classe Gestore contiene almeno i seguenti metodi:
• Gestore(String nome): costruisce un gestore per la risorsa di nome nome;
• void request(int fee): richiede l'uso della risorsa, dichiarando di essere disposto a pagare fee in caso di contesa:
• void release(): rilascia la risorsa;
• String toString(): restituisce il nome della risorsa gestita;

28 luglio 2020

Una classe Mailbox ha i metodi pubblici void invia(int tipo, String msg) e String ricevi(int tipo), dove tipo è un numero compreso tra 0 e 3 (estremi inclusi). La classe mantiene una coda di messaggi per ciascun tipo. Il metodo invia accoda un nuovo messaggio msg di tipo tipo. Il metodo ricevi attende che sia disponibile almeno un messaggio di tipo tipo, e di questi restituisce quello accodato da più tempo, rimuovendolo dalla Mailbox.
Due classi Lettore e Scrittore, che estendono Thread e i cui costruttori ricevono un int tipo e un oggetto Mailbox. I thread di classe Scrittore inviano sul Mailbox tre messaggi di tipo tipo e di forma "Messaggio i", (i=0,1,2), seguiti da un messaggio "Stop" (sempre di tipo tipo), quindi terminano. I thread di classe Lettore leggono tutti i messaggi di tipo tipo dal Mailbox e li stampano sul video, ciascuno preceduto dalla stringa "Tipo tipo: ". Se il messaggio è "Stop", terminano.

Un insieme di thread compete per una risorsa condivisa. Ciascun thread richiede l'uso della risorsa per mezzo dell'operazione request e la rilascia al gestore per mezzo dell'operazione release. L'operazione request può essere bloccante nel caso che la risorsa sia già allocata ad un altro thread. Nel caso che due o più thread siano sospesi in attesa della risorsa, la risorsa verrà allocata al thread che è disposto a pagare il fee maggiore. Il fee è un intero che ciascun thread passa come parametro dell'operazione request ed compreso tra zero e FEE. L'operazione request lancia un'eccezione FeeOutOfBoundException se un thread specifica un fee minore di zero o maggiore di FEE.
Realizzare le classi FeeOutOfBoundExceptione e Gestore. La classe Gestore contiene almeno i seguenti metodi:
- Gestore(String nome): costruisce un gestore per la risorsa di nome nome;
- void request(int fee): richiede l'uso della risorsa, dichiarando di essere disposto a pagare fee in caso di contesa:
- void release(): rilascia la risorsa;
- String toString(): restituisce il nome della risorsa gestita;

In un ufficio ci sono N operatori, ognuno identificato da un numero intero compreso tra 1 e N. Per regolare il flusso degli utenti, l'ufficio utilizza un salva-code costituito da un distributore di biglietti. Il distributore eroga ticket agli utenti. Ciascun ticket riporta un numero progressivo. Quando un operatore si libera notifica questo evento al salva-code che visualizza su un display il prossimo biglietto da servire e il numero dell'operatore. Realizzare il salva-code come una classe Java che fornisce le seguenti operazioni:

- SalvaCode(): costruttore;
- int getTicket() che restituisce all'utente un ticket;
- int waitForTurn(int tkt) che prende in ingresso il ticket tkt e restituisce il numero di un qualunque operatore libero che servirà l'utente che detiene quel ticket. L'operazione è bloccante se tutti gli operatori sono occupati.
- int nextTicket(int opr) che prende il numero opr dell'operatore e restituisce il ticket che tale operatore servirà.

Definire una classe Contenitore contenente un campo pubblico dati, definito come un array di interi la cui dimensione è passata come unico argomento al costruttore della classe. I thread che vogliono accedere a questo campo devono prima acquisire il diritto di lettura o di scrittura e rilasciarlo quando hanno terminato. Per permettere ciò, la classe Contenitore fornisce i metodi (tutti void) acquisisci_lettura(), acquisisci_scrittura(), rilascia_lettura() e rilascia_scrittura(). Tali metodi garantiscono che:
1) il diritto di scrittura venga concesso solo quando nessuno possiede il diritto di lettura o di scrittura;
2) il diritto di lettura venga concesso solo quando nessuno possiede il diritto di scrittura (non importa se qualcuno possiede il diritto di lettura).
Definire inoltre una classe Lettore che estenda Thread, con costruttore Lettore(Contenitore e, int n). Ogni thread Lettore deve leggere (avendone acquisito il diritto) tutti gli elementi del campo dati di e, sospendendosi per 10 msec tra una lettura e la successiva e sommando i valori letti. Se la somma complessiva è multipla di n, deve stampare "Lettore: OK", altrimenti deve stampare "Lettore: ERRORE". Definire infine una classe Scrittore che estenda Thread, con costruttore Scrittore(Contenitore e). Ogni thread Scrittore deve scrivere (avendone acquisito il diritto) in tutti gli elementi del campo dati di e, incrementando di 1 ogni valore letto e sospendendosi per 10 msec tra una scrittura e l'altra.

Un varco è composto da N tornelli. Ogni tornello permette di passare una persona alla volta. Una persona può passare da qualunque tornello. Se tutti i tornelli sono occupati, una persona deve attendere che se ne liberi almeno uno. Le persone passano dal varco fintanto che questo è aperto. Quando il varco viene chiuso nessuno può più passare e viene restituito il numero di persone che sono transitate.
Realizzare la classe Varco con i seguenti metodi:
• Varco(int numTornelli) che costruisce contestualmente un varco composto da un numero di tornelli specificati dal parametro numTornelli. I tornelli sono identificati da un numero che va da 0 a numTornelli-1.
• Varco() che costruisce un varco costituito da un solo tornello;
• int occupaTornello() throws VarcoChiusoException, InterruptedException che permette ad una persona di occupare un tornello prima di transitarci attraverso. Se nessun tornello è libero, la persona attende il primo che si libera. Restituisce il numero del tornello occupato.
• void liberaTornello(int i) che permette ad una persona di liberare il tornello i, non appena vi ha transitato attraverso.
• int chiudiVarco() che permette di chiudere il varco.

22 settembre 2020

Un treno è dotato di *n* posti. Un cliente può prenotare con un'unica richiesta un numero di posti compreso tra 1 e *n* (estremi inclusi). Quando un cliente esegue una prenotazione con successo gli viene restituito un intero univoco che identifica la prenotazione stessa. Un cliente può disdire una prenotazione precedente, rendendo i posti di nuovo liberi.

Realizzare una classe *Treno* dotata almeno dei seguenti metodi e costruttori:

- *Treno()*: crea un treno con 100 posti.
- *Treno(int n)*: crea un treno con *n* posti.
- *int prenota(int q, long t)*: prenota *q* posti; se il numero di posti disponibili è minore di *q* il cliente viene messo in attesa per un tempo non superiore a *t* millisecondi; in particolare, se prima di *t* millisecondi vengono liberati abbastanza posti per consentire la prenotazione del cliente in questione, allora il metodo esce dalla situazione di blocco il più presto possibile e completa la prenotazione; se invece non vengono liberati abbastanza posti allo scadere del tempo *t* il metodo lancia un'eccezione di tipo *PrenotazioneFallitaException* (da definire); il metodo restituisce l'identificatore univoco della prenotazione eseguita.
- *void disdici(int p)*: disdice la prenotazione identificata da *p* (rendendo di nuovo liberi i posti associati a tale prenotazione).

Realizzare inoltre una classe *Cliente* che

- prenota un certo numero di posti;
- nel 50% dei casi li disdice;
- termina la propria esecuzione.

Un *Contenitore* è in grado di conservare valori di tipo intero. Su un contenitore sono possibili due operazioni fondamentali: l'aggiunta di un nuovo valore e la verifica della presenza di un certo valore nel contentitore. Le operazioni di verifica possono essere eseguite in contemporanea ad altre operazioni di verifica (la struttura dati contenitore non viene modificata). L'aggiunta di nuovi valori avviene in modo mutuamente esclusivo rispetto a altre operazioni di aggiunta e di verifica (in quanto l'aggiunta di un valore richiede di modificare la struttura dati contenitore). Realizzare una classe *Contenitore* dotata almeno dei seguenti metodi e costruttori:

- *Contenitore(int s, int q, int m)*: crea un contenitore in grado di ospitare al più *s* interi. Il contenitore che viene creato contiene *q* interi casuali compresi tra 1 e *m* (estremi inclusi). I valori sono memorizzati in un array in ordine crescente.
- *void aggiungi(int v) throws ContenitorePienoException*: aggiunge il valore *v* mantenendo l'ordinamento. Se non c'è spazio per il nuovo valore il metodo lancia *ContenitorePienoException* (da definire). Il metodo può essere bloccante, nel caso in cui siano già in corso delle operazioni di aggiunta o di verifica.
- *boolean verifica(int v)*: restituisce *true* se il valore *v* è presente nel contenitore, *false* altrimenti. Il metodo può essere bloccante, nel caso in cui siano già in corso operazioni di aggiunta. E' possibile avere esecuzioni concorrenti del metodo *verifica()* sullo stesso oggetto contenitore.
- *String toString()*: restituisce una rappresentazione in formato stringa del contenitore secondo il formato illustrato dal seguente esempio:

< 23 49 201 367 543 666 898 981 >

Un analizzatore è in grado di cercare vocaboli all'interno di un testo. Realizzare una classe *Analizzatore* dotata almeno dei seguenti metodi e costruttori:

- *Analizzatore(String s)*: crea un analizzatore; il testo su cui possono essere eseguite le ricerche di vocaboli è quello indicato dall'argomento *s*.
- *Analizzatore(int lung)*: crea un analizzatore; il testo su cui possono essere eseguite le ricerche di vocaboli è una stringa casuale di lunghezza *lung*, formata dai soli caratteri maiuscoli dell'alfabeto inglese.
- *void cerca(String[] vocaboli, int numThreads)*: fa partire la ricerca dei vocaboli, specificati dal primo argomento, nel testo da analizzare. Per ogni vocabolo deve essere determinato il numero delle sue occorrenze nel testo. La ricerca viene svolta in parallelo usando un numero di thread pari a quanto specificato dal secondo argomento.
- *void stampaRisultato()*: stampa a video la lista dei vocaboli affiancata dal numero di occorrenze. Per esempio:

ABC: 32

PIPPO: 8

PLUTO: 11

XYZ: 29

è il risultato che si ottiene se nel testo la stringa "ABC" è presente 32 volte, la stringa "PIPPO" è presente 8 volte e così via. Il metodo è bloccante nel caso in cui venga invocato prima che sia terminata la ricerca di tutti vocaboli specificati in *cerca()*.

Realizzare anche una classe *Prova* che può essere usata da riga di comando per lanciare l'analisi su un testo casuale, come illustrato dal seguente esempio:

java Prova 1234 3 ABC PIPPO PLUTO XYZ

Il primo argomento (1234) indica la lunghezza del testo casuale da generare, il secondo (3) il numero di thread da utilizzare, i rimanenti (ABC PIPPO ...) i vocaboli da cercare.

Un'asta è caratterizzata da una stringa, che descrive il bene messo in vendita, e da un intero, che indica la durata dell'asta espressa in secondi. Il tempo associato a un'asta comincia a trascorrere nel momento in cui l'asta viene attivata. I partecipanti possono fare offerte per potersi aggiudicare il bene messo all'asta. Allo scadere dell'asta il partecipante che ha fatto l'offerta più alta si aggiudica il bene. Un'asta può essere terminata in anticipo rispetto alla sua normale scadenza; in tal caso, il partecipante che ha fatto l'offerta più alta fino a tale istante si aggiudica il bene. Scrivere una classe *Asta* dotata dei seguenti costruttori e metodi:

- *Asta(String descrizione, int durata)*: crea un oggetto asta dalle caratteristiche specificate.
- *boolean attiva()*: fa partire l'asta.
- *boolean offerta(float quanto, Partecipante p)*: il partecipante *p* offre la somma indicata da *quanto*; il metodo restituisce *true* se l'offerta fatta è la più alta fino a questo istante, *false* altrimenti.
- *boolean hoVinto(Partecipante p)*: viene chiamato da un partecipante per capire se, al termine dell'asta, si è aggiudicato il bene o meno; il metodo può essere bloccante nel caso in cui l'asta non sia ancora terminata; restituisce *true* al partecipante che si è aggiudicato il bene, *false* agli altri.
- *void aggiudicaAdesso()*: termina prematuramente l'asta; il partecipante che ha fatto l'offerta più alta fino a questo istante si aggiudica il bene.

I metodi *offerta()*, *hoVinto()* e *aggiudicaAdesso()* lanciano un'eccezione di tipo *AstaNonAttivaException* nel caso in cui vengano invocati su un'asta non ancora attivata. Realizzare tale eccezione. Realizzare anche una classe *Partecipante* che si comporta come segue:

- Riceve in fase di costruzione un oggetto *Asta*.
- Fa un'offerta relativa a tale asta.
- Si mette in attesa che l'asta termini con il metodo *hoVinto()*.
- Stampa un messaggio che indica se ha vinto o meno.

Un supermercato permette l'accesso di un numero massimo di compratori pari a N. Quando si raggiunge tale numero massimo, non viene permesso nessun nuovo ingresso finche' il numero di compratori all'interno del supermercato non diventa $N - K$, dove $0 \leq K < N$.

Il supermercato ha due code di ingresso, quella normale e quella per gli ingressi prioritari riservata a anziani e donne in stato di gravidanza.

Realizzare la classe Supermercato con almeno i seguenti metodi/cosruttori:

• Supermercato(int N, int K): costruttore; lancia IllegalArgumentException se i parametri non sono validi;

• void entrata(TipoVisitatore tipo): un thread chiede di entrare al Supermercato specificando se si tratta di un ingresso prioritario o meno; l'operazione puo' essere bloccante;

• void uscita(): un thread notifica la propria uscita dal museo.

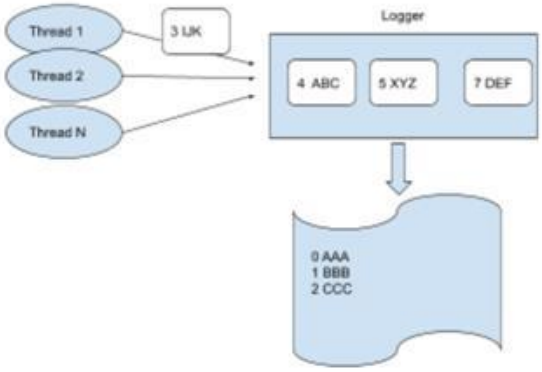
TipoVisitatore deve essere un enumerato con i valori ORDINARIO, ANZIANO, GRAVIDANZA.

02 Novembre 2020

Un treno è dotato di *n* posti. Un cliente può prenotare con un'unica richiesta un numero di posti compreso tra 1 e *n* (estremi inclusi). Quando un cliente esegue una prenotazione con successo gli viene restituito un intero univoco che identifica la prenotazione stessa. Un cliente può disdire una prenotazione precedente, rendendo i posti di nuovo liberi.
Realizzare una classe *Treno* dotata almeno dei seguenti metodi e costruttori:

- *Treno()*: crea un treno con 100 posti.
- *Treno(int n)*: crea un treno con *n* posti.
- *int prenota(int q, long t)*: prenota *q* posti; se il numero di posti disponibili è minore di *q* il cliente viene messo in attesa per un tempo non superiore a *t* millisecondi; in particolare, se prima di *t* millisecondi vengono liberati abbastanza posti per consentire la prenotazione del cliente in questione, allora il metodo esce dalla situazione di blocco il più presto possibile e completa la prenotazione; se invece non vengono liberati abbastanza posti allo scadere del tempo *t* il metodo lancia un'eccezione di tipo *PrenotazioneFallitaException* (da definire); il metodo restituisce l'identificatore univoco della prenotazione eseguita.
- *void disdici(int p)*: disdice la prenotazione identificata da *p* (rendendo di nuovo liberi i posti associati a tale prenotazione).

Un insieme di thread produce dei messaggi di log. Ogni messaggio è caratterizzato da un numero d'ordine intero diverso da tutti gli altri. Un *Logger* viene usato dai thread per salvare i messaggi in un file secondo l'ordine specificato (a partire da 0). Il Logger è in grado di mantenere al suo interno un certo numero di messaggi che non possono essere trasferiti sul file in quanto fuori ordine (il messaggio con numero d'ordine X può essere trasferito su file solo quando quest'ultimo contiene, come ultimo elemento, il messaggio con numero d'ordine X-1).



La figura mostra la situazione in cui i messaggi con numero d'ordine 0, 1 e 2 sono già stati trasferiti su file e nel Logger sono contenuti i messaggi con numeri d'ordine 4, 5 e 7. Tali messaggi non possono essere trasferiti su file in quanto manca il messaggio con numero d'ordine 3. Quando il messaggio con tale numero d'ordine viene loggato dal Thread 1 i messaggi con numeri d'ordine 3, 4 e 5 vengono trasferiti su file, mentre quello con numero d'ordine 7 rimane nel Logger perché manca quello con numero d'ordine 6.
Realizzare la classe Logger secondo le seguenti specifiche:

- *Logger(String fn, int n)*: crea un Logger che salva i messaggi nel file di nome *fn*; il Logger è in grado di bufferizzare al più *n* messaggi.
- *void log(String r, int o)*: viene chiamato da un thread per salvare il messaggio *r*. L'argomento *o* indica il numero d'ordine del messaggio in questione. Il metodo può essere bloccante nel caso in cui non sia possibile trasferire immediatamente su file il messaggio *r* (perché fuori ordine) e allo stesso tempo non sia possibile mantenerlo nel Logger (perché già pieno). Viceversa il metodo non è bloccante quando è possibile trasferire immediatamente il messaggio nel file o se è possibile bufferizzarlo nel Logger.

Il file che contiene i messaggi ha il formato illustrato dal seguente esempio:

```
0     Operazione fallita
1     Provo a connettermi con il server
2     L'utente ha selezionato il menu File
3     Connessione con il server attiva
...   …
```

Un museo consente una visita guidata alla volta, e ogni visita è composta esattamente da *N* visitatori e una guida (supponiamo che ci sia sempre un numero sufficiente di visitatori). Le visite sono numerate. Ogni visitatore deve prima prenotare una visita, ottenendo un numero di visita, quindi attendere che la visita inizi. Una guida deve prima attendere che siano entrati tutti i visitatori per la prossima visita. I visitatori possono uscire in qualunque momento durante la visita. La guida attende che siano usciti tutti i visitatori prima di uscire a sua volta, terminando così la visita.

Realizzare una classe *Museo* che implementi quanto appena descritto con almeno i seguenti metodi:

- Un costruttore *Museo(int N)*, dove *N* è il numero di visitatori richiesti per ogni visita al museo.
- Un metodo *int prenotaVisita()*, usato dai visitatori per ottenere il numero della visita a cui partecipare (la prima visita non ancora al completo).
- Un metodo *void entraVisitatore(int nVisita)*, usato dai visitatori per iniziare la propria visita. Il metodo deve prima attendere che la visita eventualmente già in corso termini e che la prossima abbia numero *nVisita*, quindi stampare la stringa "Entra visitatore" e attendere ulteriormente finché la visita non inizia.
- Un metodo *void esceVisitatore()*, usato dai visitatori per uscire dal museo. Il metodo deve stampare la stringa "Esce visitatore".
- Un metodo *void entraGuida()*, usato dalle guide per iniziare una visita. Il metodo deve prima attendere che l'eventuale visita già in corso termini, quindi attendere che tutti i visitatori per la prossima visita siano entrati, e solo allora dare inizio alla visita stampando "Inizia visita *n*", dove *n* è il numero della visita.
- Un metodo *void esceGuida()*, usato dalle guide per terminare una visita. La guida deve prima attendere che tutti i visitatori siano usciti, quindi stampare "Termina visita *n*".

Un supermercato permette l'accesso di un numero massimo di compratori pari a N. Quando si raggiunge tale numero massimo, non viene permesso nessun nuovo ingresso finche' il numero di compratori all'interno del supermercato non diventa N − K, dove $0 \leq K < N$.
Il supermercato ha due code di ingresso, quella normale e quella per gli ingressi prioritari riservata a anziani e donne in stato di gravidanza.
Realizzare la classe Supermercato con almeno i seguenti metodi/cosruttori:
• Supermercato(int N, int K): costruttore; lancia IllegalArgumentException se i parametri non sono validi;
• void entrata(TipoVisitatore tipo): un thread chiede di entrare al Supermercato specificando se si tratta di un ingresso prioritario o meno; l'operazione puo' essere bloccante;
• void uscita(): un thread notifica la propria uscita dal museo.

TipoVisitatore deve essere un enumerato con i valori ORDINARIO, ANZIANO, GRAVIDANZA.

In un aeroporto ci sono P piazzole di parcheggio ed R piste di decollo ed atterraggio. Un aereo per poter iniziare l'atterraggio deve riservare una pista ed una piazzola. Ad atterraggio ultimato, l'aereo rilascia la pista ed occupa la piazzola. Al decollo, un aereo riserva una pista e rilascia la piazzola che sta occupando. Al termine del decollo, l'aereo rilascia la pista.

Un aereo può essere modellato come un thread che esegue ciclo senza fine costituito dai seguenti passi:

Inizio decollo
Decollo
Fine decollo
Volo
Inizio atterraggio
Atterraggio
Fine atterraggio

realizzare il gestore Aeroporto che fornisce le seguenti operazioni:

- Aeroporto(int P, int R) crea un aeroporto con P piste e R piazzole
- int inizioDecollo(int piazzola) che richiede l'allocazione di una pista e rilascia la piazzola occupata dall'aereo e trasmessa come parametro. La pista viene restituita come valore di ritorno. L'esecuzione dell'operazione si sospende se non c'é almeno una pista disponibile.
- void fineDecollo(int pista) che rilascia la pista riservata all'aereo e trasmessa come parametro.
- Risorse inizioAtterraggio() che inizia l'atterraggio richiedendo l'allocazione di una pista e di una piazzola che vengono restituite incapsulate nel valore di ritorno di classe Risorse. L'esecuzione dell'operazione si sospende se una pista ed una piazzola non sono entrambe disponibili.
- void fineAtterraggio(int pista) che rilascia la pista riservata all'aereo e trasmessa come parametro.

La classe Risorse è definita così:

```
class Risorse {
    private int pista;
    private int piazzola;

    public Risorse(int pista, int piazzola) {
        this.pista = pista;
        this.piazzola = piazzola;
    }

    public int getPista() {
        return pista;
    }

    public int getPiazzola() {
        return piazzola;
    }
}
```

Una classe Mailbox ha i metodi pubblici void invia(int tipo, String msg) e String ricevi(int tipo), dove tipo è un numero compreso tra 0 e 3 (estremi inclusi). La classe mantiene una coda di messaggi per ciascun tipo. Il metodo invia accoda un nuovo messaggio msg di tipo tipo. Il metodo ricevi attende che sia disponibile almeno un messaggio di tipo tipo, e di questi restituisce quello accodato da più tempo, rimuovendolo dalla Mailbox.
Due classi Lettore e Scrittore, che estendono Thread e i cui costruttori ricevono un int tipo e un oggetto Mailbox. I thread di classe Scrittore inviano sul Mailbox tre messaggi di tipo tipo e di forma "Messaggio i", (i=0,1,2), seguiti da un messaggio "Stop" (sempre di tipo tipo), quindi terminano. I thread di classe Lettore leggono tutti i messaggi di tipo tipo dal Mailbox e li stampano sul video, ciascuno preceduto dalla stringa "Tipo tipo: ". Se il messaggio è "Stop", terminano.

Un insieme di thread compete per una risorsa condivisa. Ciascun thread richiede l'uso della risorsa per mezzo dell'operazione request e la rilascia al gestore per mezzo dell'operazione release. L'operazione request può essere bloccante nel caso che la risorsa sia già allocata ad un altro thread. Nel caso che due o più thread siano sospesi in attesa della risorsa, la risorsa verrà allocata al thread che è disposto a pagare il fee maggiore. Il fee è un intero che ciascun thread passa come parametro dell'operazione request ed compreso tra zero e FEE. L'operazione request lancia un'eccezione FeeOutOfBoundException se un thread specifica un fee minore di zero o maggiore di FEE.
Realizzare le classi FeeOutOfBoundExceptione e Gestore. La classe Gestore contiene almeno i seguenti metodi:
• Gestore(String nome): costruisce un gestore per la risorsa di nome nome;
• void request(int fee): richiede l'uso della risorsa, dichiarando di essere disposto a pagare fee in caso di contesa:
• void release(): rilascia la risorsa;
• String toString(): restituisce il nome della risorsa gestita;

In un ufficio ci sono N operatori, ognuno identificato da un numero intero compreso tra 1 e N. Per regolare il flusso degli utenti, l'ufficio utilizza un salva-code costituito da un distributore di biglietti. Il distributore eroga ticket agli utenti. Ciascun ticket riporta un numero progressivo. Quando un operatore si libera notifica questo evento al salva-code che visualizza su un display il prossimo biglietto da servire e il numero dell'operatore. Realizzare il salva-code come una classe Java che fornisce le seguenti operazioni:

• SalvaCode(): costruttore;
• int getTicket() che restituisce all'utente un ticket;
• int waitForTurn(int tkt) che prende in ingresso il ticket tkt e restituisce il numero di un qualunque operatore libero che servirà l'utente che detiene quel ticket. L'operazione è bloccante se tutti gli operatori sono occupati.
• int nextTicket(int opr) che prende il numero opr dell'operatore e restituisce il ticket che tale operatore servirà.